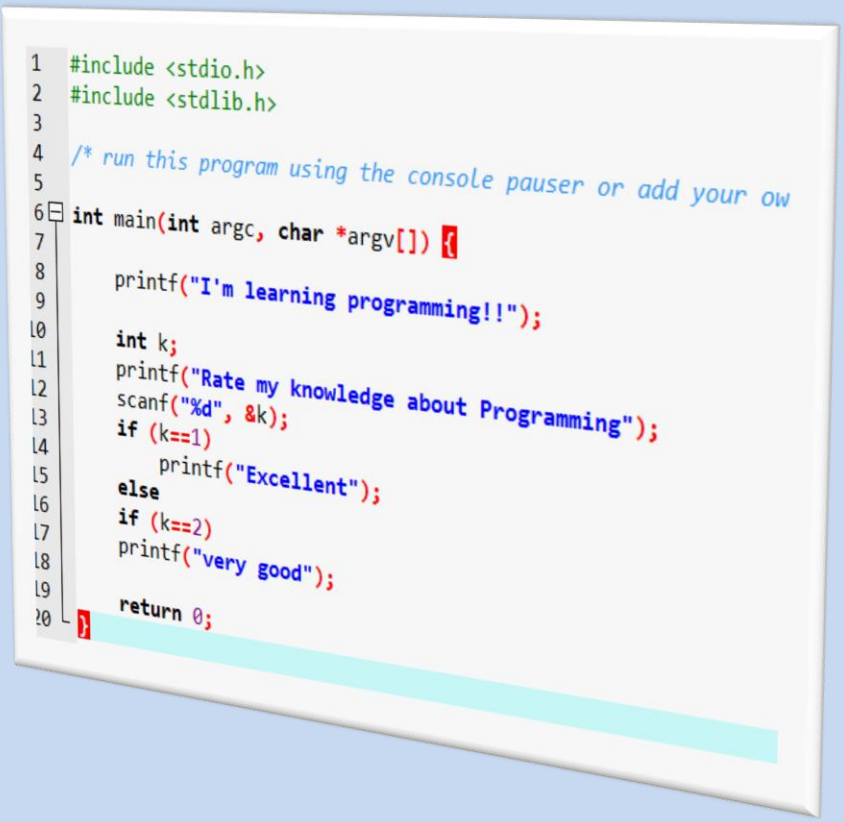
განათლების ხარისხის განვითარების ეროვნული ცენტრი
NATIONAL CENTER FOR EDUCATIONAL QUALITY ENHANCEMENT

ნონა ოთხოზორია

ლელა გაჩეჩილაძე



პროგრამული უზრუნველყოფის დეველოპერი

2015

მთელ მსოფლიოში პროგრამისტის სპეციალობა ყველაზე მოთხოვნადი და მაღალანაზღაურებადი პროფესიაა. ეს ის სპეციალობაა, რომელიც ვითარდება და იძენს მეტ პოპულარობას ახალი ტექნოლოგიების განვითარების პარალელურად. მაღალკვალიფიცირებულ პროგრამისტებზე მოთხოვნა საკმაოდ დიდია, თუმცა დღეს ასეთი კადრების დეფიციტია.

პროგრამისტობა საკმაოდ მაღალინტელექტუალური ხელობაა, რომლის დაუფლებაც დიდ შრომას და გარჯას მოითხოვს. ამ საინტერესო ხელობის დაუფლება არის ჩვენი სახელმძღვანელოს მთავარი მიზანი. ვფიქრობთ, რომ სახელმძღვანელო თითოეულ სტუდენტს საფუძველს შეუქმნის, რომ პროფესიონალ პროგრამისტ-დეველოპერად ჩამოყალიბდეს. პროგრამისტ-დეველოპერი კი ის ადამიანია, რომელიც ავითარებს მოცემულ პროგრამულ გარემოს IDE (Integrated Development Environment – განვითარების ინტეგრირებული გარემო) სისტემის საფუძველზე, ქმნის სამომხმარებლო, სამეცნიერო ან გასართობი ტიპის აპლიკაციებს, არის მოთხოვნადი შრომის ბაზარზე და აქვს მაღალანაზღაურებადი სამუშაო.

წინამდებარე სახელმძღვანელო შეიქმნა პროფესიული განათლების სტუდენტებისათვის, პროფესიულ კვალიფიკაციათა განვითარების ხელშეწყობის პროგრამის ფარგლებში.

წიგნის რეცენზენტები:

გია სურგულაძე - პროფესორი, ტექნიკის მეცნიერებათა დოქტორი, საქართველოს ტექნიკური უნივერსიტეტი

ზაალ აზმაიფარაშვილი - პროფესორი, ტექნიკის მეცნიერებათა კანდიდატი, საქართველოს ტექნიკური უნივერსიტეტი

ავტორები:

ნონა ოთხოზორია

პროფესორი, ტექნიკის მეცნიერებათა კანდიდატი, საქართველოს ტექნიკური უნივერსიტეტი

ლელა გაჩეჩილაძე

ასოცირებული პროფესორი, ტექნიკის მეცნიერებათა კანდიდატი, საქართველოს ტექნიკური უნივერსიტეტი. Oracle Academy-ის საერთაშორისო სერტიფიკატის მფლობელი Java-დაპროგრამებაში

დაპროგრამების ენების ისტორია

XIX საუკუნის 20-იან წლებში ჩარლზ ბებიჯის მიერ შემოთავაზებულ იქნა რევოლუციური იდეა, რომელმაც დასაბამი დაუდო ავტომატური ციფრული გამომთვლელი მანქანების შექმნას. ამ დროიდან იწყება დაპროგრამების ენების ისტორიაც.

დაპროგრამების ენების ისტორიაში რევოლუციური მომენტი პენსილვანიის უნივერსიტეტის თნამშრომლის ჯონ მოუჩლის მიერ შემოთავაზებული კოდირების სიტემა გახლდათ სპეციალური სიმბოლოების გამოყენებით. მოუჩლის იდეამ გაიტაცა ამავე კომპანიის ერთ-ერთი თანამშრომელი გრეის მიურეი ჰოპერი, რომლმაც მთელი თავისი ცხოვრება კომპიუტერს და დაპროგრამებას მიუძღვნა.

დაპროგრამების ენების განვითარების საწყის ეტაპზე მანქანური კოდი იყო კომპიუტერთან ადამიანის ურთიერთობის ერთადერთი საშუალება. 40-იანი წლების დასაწყისში ჯონ მოუჩლიმ შექმნა სისტემა „short code", რომელიც წარმოადგენდა მაღალი დონის დაპროგრამების ენას. ამ ენაზე პროგრამისტი წერდა ამოცანას მათემატიკური ფორმულების სახით, ხოლო შემდეგ სპეციალური ცხრილის გამოყენებით ამ ფორმულებს ორლიტერიან კოდებში წარმოადგენდა. ეს უკანასკნელი კი კომპიუტერის სპეციალურ პროგრამას ორობით მანქანურ კოდში გადაჰყავდა. ჯონ მოუჩლის მიერ შემუშავებული სისტემა პირველი პრიმიტიული ინტერპრეტატორი გახლავთ.

1951 წელს ჰოპერმა შექმნა მსოფლიოში პირველი კომპილატორი - ჰოპერი.

30-იანი წლებიდან უკვე მნიშვნელოვანი წინსვლები შეინიშნება დაპროგრამების ენების ისტორიაში. დაიწყო ახალი ტიპის დაპროგარმების ენების წარმოქმნა. პირველი და ყველაზე გავრცელებული ენა ფორტრანი (Fortran) გახლდათ, რომელიც IBM-ის ფირმის პროგრამისტთა ჯგუფის მიერ 1954 წელს იქნა შემუშავებული.

60-იან წლებში დარმუტკის კოლეჯის მათემატიკური ფაკულტეტის თანამშრომლებმა ტომას კურტმა და ჯონ კემენმა შექმნეს დაპროგრამების სპეციალიზირებული ენა, რომელიც მარტივი ინგლისური სიტყვებისგან შედგებოდა. ამ ენას მათ Basic უწოდეს. ამ დროიდან დაიწყო სხვადასხვა დაპროგრამების ენების შექმნა. მათი უმრავლესობა ფართოდ ვერ გავრცელდა და პრაქტიკული გამოყენება ვერ ჰპოვა, რადგან ეს ენები ორინეტირებული იყო გადასაწყვეტ ამოცანაზე და დამოკიდებული გახლდათ კომპიუტერის არქტიტექტურაზე. 60-იან წლებში პირველი ნაბიჯები გადაიდგა დაპროგრამების უნივერსალური ენის შესაქმნელად. ამ მიმართულებით პირველი PL/1 (Program Language One) იყო, რომელიც IBM-მა 1967 წელს შეიმუშავა. ამ ენას შესაძლებლობა ჰქონდა ნებისმიერი ამოცანა გადაეწყვიტა: შეესრულებინა

გამოთვლები, დაემუშავებინა ტექსტები, უზრუნველყო ინფორმაციის მოგროვება და ძიება. თუმცა ენა საკმაოდ რთული აღმოჩნდა, ხოლო ტრანსლიატორი - არასაკმარისად ოპტიმალური. ამ პერიოდში ძველმა დაპროგრამების ენებმაც განიცადეს მოდერნიზაცია და მივიღეთ Algol-68, Fortran-77. თუმცა ვერც ერთმა ცდამ ვერ მოიტანა წარმატება.

დაპროგრამების ენების ისტორიაში მნიშვნელოვან მოვლენად 1971 წელს დაპროგრამების ენის - პასკალის შექმნა იქცა. მისი ავტორი შვეიცარიელი მეცნიერი ნიკლაუს ვირტი გახლდათ. ვირტმა ენას სახელი XVII საუკუნის უდიდესი ფრანგი მათემატიკოსისა და ფილოსოფოსის ბლეზ პასკალის პატივსაცემად დაარქვა. როგორც ცნობილია, პასკალმა პირველი ამჯამავი მოწყობილობა შექმნა. „პასკალი“ შემუშვებულ იქნა, როგორც სტრუქტურული დაპროგრამების სასწავლო ენა და უკანასკნელ წლებამდე სკოლებსა თუ უმაღლეს სასწავლებლებში ეს ენა წარმოადგენდა დაპროგრამების დაუფლების ერთ-ერთ ძირითად ენას.

1975 წელს დაპროგრამების ისტორიაში ორი მნიშვნელოვანი მოვლენა დაფიქსირდა. ბილ გეითსმა და პოლ ალენმა გამოაცხადეს, რომ შეიმუშავეს Basic-ის ახალი ვერსია, ხოლო ვირტმა და იენსენმა გამოსცეს ენის კლასიკური აღწერა „Pascal User Manual and Report”. არანაკლებ შთამბეჭდავ ფინანსურ წარმატებას მიაღწია ფრანგმა ფილიპ კანმა, რომელმაც 1983 წელს ტურბო-პასკალი შექმნა. კანის იდეის არსი იყო პროგრამის დამუშავების თანმიმდევრული ეტაპების: კომპილაციის, რედაქტირების, გამართვის და შეცდომების დიაგნოსტიკის ერთ ინტერფეისში გაერთიანება. ტურბო-პასკალი არა მხოლოდ ენაა ან ტრანსლიატორი, არამედ ამასთან ერთად, ის ოპერაციული გარსიც არის, რომელიც საშუალებას აძლევს მომხმარებელს კომფორტულად იმუშაოს. ეს ენა გასცდა სასწავლო დანიშნულების ჩარჩოებს და პროფესიონალური დაპროგრამების ენად იქცა. პასკალი საფუძვლად დაედო მრავალ თანამედროვე ენას. Borland/Inprise ფირმამ დაასრულა Turbo-Pascal-ის სრულყოფის ხაზი და გადავიდა Windows-ის ოპერაციული სისტემისათვის ვიზუალური სისტემის შემუშავებაზე (Delphi).

თანამედროვე დაპროგრამებაში უზარმაზარი კვალი დაპროგრამების C ენამ დატოვა (პირველი ვერსია 1972 წელს შეიქმნა), რომელიც დღესაც ძალზედ პოპულარულია პროგრამული უზრუნველყოფის დეველოპერებს შორის. C ენა 1972 წელს ნიუ-ჯერსის შტატის ქალაქ მიურეი-ჰილის bell laboratories კომპანიის სისტემურმა პროგრამისტმა დენის რიჩმა შექმნა. ეს ენა შემუშავდა, როგორც ოპერაციული სისტემების, ტრანსლიატორების, მონაცემთა ბაზებისა და გამოყენებითი პროგრამების განვითარების ინსტრუმენტი. C აერთიანებს როგორც მაღალი დონის, ასევე მანქანაზე ორიენტირებული ენების თვისებებს. პროგრამისტს აძლევს ნებისმიერი მანქანური რესურსის წვდომის საშუალებას, რომელსაც ვერცერთი სტრუქტურული ენა ვერ იძლევა, ვერც Basic და ვერც Pascal. C ენაზე დაიწერა UNIX-ის ბირთვი. მისი პოპულარობა

გრძელდებოდა მანამ, სანამ არ დადგა კიდევ უფრო რთულ პროგრამებთან მუშაობის პრობლემა. გამოსავალი ობიექტზე ორიენტირებულ დაპროგრამებაზე გადასვლა იყო.

bell laboratories კომპანიაში ბერნ სტრაუსტრუპის მიერ შემუშავებულ იქნა C ენის ობიექტზე ორიენტირებული ვერსია, რომელსაც 1983 წლიდან C++ ეწოდა. 90-იანი წლებიდან C++-ის მასობრივი გამოყენება იწყება.

1993 წელს თავს იჩენს დაპროგრამების ენა Java. იგი Sun Microsystem-ის კომპანიაში შეიმუშავეს ჯეიმს გოსლინგმა, პატრიკ ნოტონმა, კრის ვორტმა, ედ ფრანკმა და მარკ შერიდენმა. Java-მ დაპროგრამების C++ ენიდან აიღო სინტაქსი და სტრატეგია. დაიწყო რა ინტერნეტის გავრცელება, წინა პლანზე ერთი პლატფორმიდან მეორეზე პროგრამების ადვილად გადატანის პრობლემამ წამოიწია. ეს გახდა Java-ს პოპულარობის მთავარი მიზეზი. Java დაპროგრამება შესაძლებელია ყველა ოპერაციულ სისტემაზე, სადაც Java ვირტუალური მანქანაა დაყენებული (Java Virtual Machine). Java-ს უპირატესობა იმაში მდგომარეობს, რომ მას შეუძლია პლატფორმა-თაშორისი გადატანითი კოდის შექმნა. თავდაპირველად აქ ადგილი აქვს საწყისი კოდის შუალედურ კოდში ტრანსლირებას, რომელსაც ბაიტ-კოდი ეწოდება, ხოლო შემდგომ მის გადაყვანას მანქანურ კოდში. რაც შეეხება Java-ს ნაკლოვანებებს, პირველი და ძირითადი ისაა, რომ ის ვერ უზრუნველყოფს მრავალენობრივ დაპროგრამებას და მეორე, Windows-ის პლატფორმის პირდაპირი უზრუნველყოფა მას არ გააჩნია. ამ პრობლემების გადასაწყვეტად 1990 წელს Microsoft-მა დაპროგრამების ენა C# შექმნა. მისი ავტორი ანდერს ჰელსბერგია. C#-ს C ენის სინტაქსი გააჩნია, ხოლო Java-დან მემკვიდრეობით მიღებული აქვს შუალედური ენის გამოყენების შესაძლებლობა. თავად კი C++-ის ობიექტურ მოდელზეა აგებული. მის უპირატესობას მრავალენობრივ გარემოში მუშაობის უნარი წარმოადგენს. C# ენის ერთ-ერთი მნიშვნელოვანი სიახლე პროგრამული უზრუნველყოფის კომპონენტების ჩადგმის უზრუნველყოფაა. ფაქტიურად, C# ენა შექმნილია, როგორც კომპონენტებზე ორიენტირებული ენა, რომელიც ისეთ ელემენტებს მოიცავს, როგორიცაა თვისებები, მეთოდები და მოვლენები.

ამჟამად დაპროგრამების ენები ისეთი მრავალფეროვანი ამოცანების გადასაწყვეტად გამოიყენება, როგორიცაა მონაცემთა საინფორმაციო სისტემების მართვა, რთული მათემატიკური და ეკონომიკური ამოცანების გადაწყვეტა, მედიცინა და ა.შ. C# ენა მთლიანად პასუხობს დაპროგრამების თანამედროვე სტანდარტებს და განკუთვნილია .NET Framework ტექნოლოგიის განვითარების უზრუნველყოფისათვის. იგი არის დაპროგრამების მძლავრი ენა, რომელიც Windows გარემოში მომუშავე იმ თანამედროვე კომპიუტერული სისტემებისთვის არის განკუთვნილი, რომლებიც ინტერნეტ-ტექნოლოგიებს იყენებენ.

ასეთია, დაპროგრამების ენების მოკლე ისტორიული ექსკურსი. ცხადია, თითოეული მათგანის თუნდაც მიმოხილვა ერთი სახელმძღვანელოს ფარგლებში შეუძლებელია. ამიტომ

მთავარი აქცენტი ჩვენ დაპროგრამების იმ ენებზე გადავიტანეთ, რომელთაც გარკვეული როლი შეასრულეს თანამედროვე დაპროგრამების ენების განვითარებაში. საბოლოო არჩევანი კი ყოველთვის მკითხველის პრეროგატივაა.

1. ალგორითმიზაციის საფუძვლები

1.1 ალგორითმიზაციის ზოგადი ცნებები

1.1.1 ამოცანების გადაწყვეტის ძირითადი ეტაპები

მათემატიკური და საინჟინრო ამოცანების კომპიუტერის საშუალებით გადაწყვეტა საკმაოდ რთული და შრომატევადი პროცესია, რომელიც მთელი რიგი ეტაპების შესრულებას ითვალისწინებს. ეს ეტაპებია:

- ამოცანის დასმა;
- ამოცანის მათემატიკური ფორმულირება;
- რიცხვითი მეთოდის შერჩევა;
- გამოთვლითი პროცესის ალგორითმიზაცია;
- კომპიუტერული პროგრამის შედგენა;
- კომპიუტერული პროგრამის გამართვა;
- ამოცანის გადაწყვეტა.

ამოცანის დასმა. ნებისმიერი ამოცანის გადაწყვეტის საწყის ეტაპს ამოცანის დასმა წარმოადგენს. იგი განსაზღვრავს ამოცანის გადაწყვეტის მიზანს. ამ ეტაპზე ამოცანა პროფესიონალური ცნებების დონეზე ფორმულირდება და იგი კორექტული და გასაგები უნდა იყოს შემსრულებლისათვის (მომხმარებლისათვის).

ამოცანის მათემატიკური ფორმულირება. მათემატიკური ფორმულირების ეტაპზე ადგილი აქვს ფორმულების საშუალებით ამოცანის ფორმალიზაციას, რომლის დროსაც საწყისი პირობები, გამოთვლის ცდომილება, საწყისი და საბოლოო მონაცემები განისაზღვრება. არსებითად აქ ადგილი აქვს გადასაწყვეტი ამოცანის მათემატიკური მოდელის შემუშავებას.

რიცხვითი მეთოდის შერჩევა. რიგ შემთხვევაში, ერთიდაიგივე ამოცანა სხვადასხვა რიცხვითი მეთოდის საშუალებით შეიძლება გადავწყვიტოთ. მეთოდის შერჩევა დამოკიდებულია მრავალ ფაქტორზე, რომელთაგან ძირითადია ამოცანის გადაწყვეტის სიზუსტე, ამოცანის გადაწყვეტისათვის საჭირო კომპიუტერული დრო და გამოყენებული ოპერატიული მეხსიერების ტევადობა. თითოეულ კონკრეტულ შემთხვევაში რიცხვითი მეთოდის შერჩევა აღნიშნული კრიტერიუმების გათვალისწინებით ხორციელდება.

საჭიროა შევნიშნოთ, რომ უმარტივესი ამოცანების გადაწყვეტის შემთხვევაში აღნიშნული ეტაპი არ გამოიყენება, რადგან რიცხვითი მეთოდი თვით ამოცანის მათემატიკური ფორმულირების საფუძველზეა განსაზღვრული. ასე, მაგალითად, ჰერონის ფორმულით სამკუთხედის ფართობის გამოთვლის, კვადრატული განტოლების ფესვების განსაზღვრის ამოცანები და სხვ.

გამოთვლითი პროცესის ალგორითმიზაცია. მოცემულ ეტაპზე, შერჩეული რიცხვითი მეთოდის მიხედვით, განისაზღვრება ელემენტარული გამოთვლების ლოგიკური მიმდევრობა, რომლის საფუძველზეც ამოცანის ამონახსნი მიიღება.

ალგორითმი არ არის მეთოდი. იგი მხოლოდ შერჩეული რიცხვითი მეთოდის რეალიზაციაა. თუ ერთიდაიგივე რიცხვითი მეთოდის რამდენიმე რეალიზაცია არსებობს, მაშინ საჭიროა ალგორითმის ის ვარიანტი ავირჩიოთ, რომელიც კომპიუტერის ეფექტურ გამოყენებას უზრუნველყოფს. სხვა სიტყვებით რომ ვთქვათ, საჭიროა ისეთი ალგორითმის შერჩევა, რომელსაც ნაკლები რაოდენობის მათემატიკური ოპერაციების შესრულება და მინიმალური ტევადობის მეხსიერება ესაჭიროება.

კომპიუტერული პროგრამის შედგენა. ალგორითმის საფუძველზე კომპიუტერული პროგრამის შედგენის პროცესს **დაპროგრამება** ეწოდება. თუ პროგრამა მანქანურ ენაზეა შესრულებული (ე.ი. პროგრამა დაწერილია გამომთვლელი მანქანის კოდებში), მაშინ მისი კომპიუტერული რეალიზაცია უშუალოდ შეიძლება განხორციელდეს, ხოლო თუ პროგრამა რომელიმე ალგორითმულ ენაზეა ჩაწერილი, მაშინ იგი სპეციალური პროგრამული უზრუნველყოფის - ტრანსლიატორის საშუალებით გადაიყვანება მანქანურ კოდებში და მხოლოდ ამის შემდეგ არის შესაძლებელი მისი კომპიუტერული რეალიზაცია.

კომპიუტერული პროგრამის გამართვა. პროგრამაში შეცდომების მოძებნისა და გასწორების (რედაქტირების) პროცესს **გამართვა** ეწოდება. განასხვავებენ სინტაქსური და ლოგიკური სახის შეცდომებს.

სინტაქსური შეცდომების შემთხვევაში პროგრამის ნორმალური ფუნქციონირება შეუძლებელია მასში დაუშვებელი პროგრამული კონსტრუქციების გამო. აღნიშნული შეცდომების მოძებნა შესაძლებელია ტრანსლიატორში არსებული სინტაქსური კონტროლის პროგრამის საშუალებით.

იმ შემთხვევაში, როცა კომპიუტერული პროგრამის მუშაობის შედეგი მოსალოდნელ შედეგს არ ემთხვევა, მაშინ ადგილი აქვს ლოგიკური სახის შეცდომებს. აღნიშნული შეცდომები განპირობებულია ალგორითმის არასწორი შემუშავების ან არასწორი დაპროგრამების შედეგად. ლოგიკური შეცდომების არსებობა იმის მანიშნებელია, რომ საჭიროა წინა ეტაპების გულდასმით გადამოწმება, ალგორითმში და პროგრამაში შესაბამისი შესწორებების შეტანა. ეს პროცედურა დაკავშირებულია დროის მნიშვნელოვან დანახარჯებთან.

ამოცანის გადაწყვეტა. პროგრამის გამართვის შემდეგ ადგილი აქვს კომპიუტერზე ამოცანის უშუალოდ გადაწყვეტას, რომლის დროსაც კომპიუტერში შეიტანება საწყისი მონაცემები და პროგრამის თვლაზე გაშვების შედეგად საჭირო რეზულტატები მიიღება.

1.1.2 ალგორითმის ცნება

ამოცანების გადაწყვეტის პროცესში ალგორითმების შემუშავება ერთ-ერთი ყველაზე მნიშვნელოვანი და საპასუხისმგებლო ეტაპია. ამ ეტაპზე ადგილი აქვს იმ ელემენტარულ მოქმედებათა თანმიმდევრობის განსაზღვრას, რომელსაც შემდგომში კომპიუტერი უშუალოდ ასრულებს. ცხადია, ალგოროთმის შემუშავების პროცესში შეცდომების დაშვება დაუშვებელია, რადგან იგი, თავის მხრივ, გამოთვლით პროცესს არასწორად წარმართავს და შედეგიც არასწორი მიიღება.

ალგორითმი თანამედროვე მათემატიკის ფუნდამენტური ცნებების რიცხვს მიეკუთვნება და იგი სპეციალური დისციპლინის - ალგორითმების თეორიის კვლევის ობიექტს წარმოადგენს.

ალგორითმის შედგენის პროცესს **ალგორითმიზაციას** უწოდებენ.

დღეს აღიარებული ვერსიის მიხედვით, სიტყვა ალგორითმის წარმოშობა უკავშირდება შუა აზიის ცნობილი მეცნიერის, აბუ ჯაფარ მუჰამედ იბნ მუსა ალ-ხორეზმი ალ-მაჯუსის (783-850) სახელს. ხორეზმი დღევანდელი უზბეკეთის ტერიტორიაზე მდებარეობს, ხოლო ალ-ხორეზმი სიტყვა-სიტყვით ნიშნავს ხორეზმში მცხოვრებს – „ჯაფარის მამა, მუჰამედი, მუსას შვილი, ხორეზმში მცხოვრები, ჯადოქართა ოჯახიდან“. შუა საუკუნეების ევროპელი მეცნიერები თვლის ინდურ, პოზიციურ, ათობით სისტემას და თვლის მეთოდებს ამ სისტემაში ალ-ხორეზმის არაბული ტრაქტატის ლათინური თარგმანების მიხედვით გაეცნენ. ალ-ხორეზმის არითმეტიკული ტრაქტატის ლათინური თარგმანი იწყებოდა სიტყვებით „Dixit algorizmi …” - " და თქვა ალ-ხორეზმმა...".

ამგვარად, თავიდან სიტყვა ალგორითმი გამოიყენებოდა ათობითი პოზიციური სისტემის და არითმეტიკული ოპერაციების ალგორითმის, ანუ წესების აღწერისთვის, ხოლო შემდეგ კი, ნებისმიერი ალგორითმისთვის.

ნახ. 1 ალ-ხორეზმი

დღევანდელ ინგლისურ ენაში სიტყვა algorism ნიშნავს როგორც არაბულ (ათობით) პოზიციურ სისტემას, ასევე ალგორითმს. უფრო გავრცელებულია სინონიმი, კერძოდ, სიტყვა algorithm, რომელიც ითარგმნება როგორც მეთოდი, წესი.

საინტერესოა, რომ ალ-ხორეზმის ალგებრული ტრაქტატის სახელს უკავშირდება სიტყვა ალგებრის წარმოშობაც (“Kitab al jabr w’al muqabala” – “აღდგენისა და გარდაქმნის წესები”).

მოგვიანებით კი ალგორითმს აიგივებდნენ ევკლიდეს შრომებში მოცემული ორი რიცხვის უდიდესი საერთო გამყოფის მოძებნის წესთან (ევკლიდეს ალგორითმი).

ამჟამად, ტერმინი ალგორითმი სხვადასხვა სახის ცალკეული წესების საზოგადო სახელია და იგი შემდეგნაირად არის განმარტებული: **ალგორითმი გარკვეულ მითითებათა სასრული მიმდევრობაა, რომლის შესრულება საშუალებას გვაძლევს მივიღოთ მოცემული ამოცანის ამონახსნი.**

კომპიუტერული მეცნიერებების წარმოშობასა და განვითარებასთან დაკავშირებით 1968 წ. დ. კნუტის მიერ შემოტანილ იქნა ალგორითმის შემდეგი განმარტება:

ალგორითმი არის ამოცანის ამოხსნისთვის საჭირო ოპერაციების თანმიმდევრობის განმსაზღვრელი წესები.

მოყვანილი განმარტებაში, ცხადია, დასაზუსტებელია, რომელ ამოცანებზეა საუბარი, რა ოპერაციები და რა წესები იგულისხმება, რომელი ობიექტისა და რომელი სუბიექტის მიმართ. ასე საკითხის დასმა რთულ ზოგად სიტუაციასთან მიგვიყვანდა. საბედნიეროდ, კომპიუტერულ მეცნიერებებში ჩვენ ვგულისხმობთ კომპიუტერისთვის დასმულ ამოცანებს და ამდენად, ოპერაციებიც იქნება ის ოპერაციები, რომლებსაც კომპიუტერი ასრულებს.

მოვიყვანოთ ალგორითმების რამდენიმე მაგალითი.

ალგორითმი 1. განვიხილოთ **ევკლიდეს ალგორითმი**, რომლის საშუალებითაც შეიძლება ამოხსნილ იქნეს შემდეგი ტიპის ამოცანა: ვიპოვოთ მოცემული ორი ნატურალური და რიცხვის უდიდესი საერთო გამყოფი.

ნახ. 2 ევკლიდე ალექსანდრიელი

როგორც ცნობილია, ორი რიცხვის უდიდესი საერთო გამყოფი (უსგ) ეწოდება იმ უდიდეს რიცხვს, რომელზედაც მოცემული რიცხვები უნაშთოდ იყოფა. მის მოსაძებნად საჭიროა თანმიმდევრობით შევასრულოთ შემდეგი ოპერაციები:

თავდაპირველად უდიდესი რიცხვი გავყოთ უმცირესზე, შემდეგ უმცირესი რიცხვი გავყოთ მიღებულ ნაშთზე, შემდეგ პირველი ნაშთი გავყოთ მეორე ნაშთზე და ა.შ. ოპერაციები გავაგრძელოთ მანამ, სანამ ნაშთი ნულის ტოლი არ გახდება. რიგით ბოლო გამყოფი უდიდეს საერთო გამყოფს წარმოადგენს. ვინაიდან გაყოფის ოპერაცია მრავალჯერ-განმეორებად გამოკლების ოპერაციაზე დაიყვანება, ამიტომ ალგორითმი შემდეგნაირად შეიძლება იქნეს ფორმულირებული:

1. თუ $a > b$, მაშინ გადავიდეთ მე-3 პუნქტზე, წინააღმდეგ შემთხვევაში - მე-2 პუნქტზე;
2. თუ $b > a$, მაშინ გადავიდეთ მე-4 პუნქტზე, წინააღმდეგ შემთხვევაში - მე-5 პუნქტზე;

3. a-ს გამოვაკლოთ b და მიღებული სხვაობა ჩავთვალოთ a-ს მნიშვნელობად. დავბრუნდეთ პირველ პუნქტში;
4. b-ს გამოვაკლოთ a და მიღებული სხვაობა ჩავთვალოთ b-ს მნიშვნელობად. დავბრუნდეთ პირველ პუნქტში;
5. a-ს ან b-ს მიღებული მნიშვნელობა ჩაითვლება უდიდეს საერთო გამყოფად.
6. დასასრული.

აღნიშნული ალგორითმი გამოვიყენოთ კონკრეტული რიცხვების უდიდესი საერთო გამყოფის მოსაძებნად. დავუშვათ, რომ $a=95$ და $b=60$. მაშინ ევკლიდეს ალგორითმის თანახმად შესრულებული პუნქტების მიმდევრობა და ამ დროს მიღებული a და b სიდიდეების ცვლადი მნიშვნელობები პირველ ცხრილშია წარმოდგენილი.

ცხრილი 1. ევკლიდეს ალგორითმის შესაბამისად შესრულებული მიმდევრობა

a	b	ალგორითმის ბიჯები
95	60	1, 3
35	60	1, 2, 4
35	25	1, 3
10	25	1, 2, 4
10	15	1, 2, 4
10	5	1, 3
5	5	1, 2, 5

მართლაც, პირველი პუნქტის თანახმად შესრულებული ოპერაციის შედეგად მე-3 პუნქტზე გადავდივართ, რადგან $a=95>60$. მე-3 პუნქტის თანახმად a-ს მნიშვნელობა გახდა ტოლი $a=95-60=35$, ხოლო b წინანდებურად ტოლია $b=60$. მე-3 პუნქტში მოცემული მითითების შესაბამისად გადავდივართ პირველ პუნქტში, რომლის შესრულების შედეგად გადავდივართ მე-2 პუნქტში, რადგან $a>b$ უტოლობა მცდარია. მე-2 პუნქტში მოცემული მითითების შესაბამისად გადავდივართ მე-4 პუნქტში, სადაც გამოითვლება b-ს ახალი მნიშვნელობა $b=60-35=25$ და კვლავ პირველ პუნქტს ვუბრუნდებით, და ა.შ. ანალოგიური პროცესების განმეორების შედეგად მივიღებთ, რომ მოცემული a და b რიცხვების უდიდესი საერთო გამყოფია ციფრი 5.

ნახ. 3 ერატოსთენე კირენელი

ალგორითმი 2. განვიხილოთ ალგორითმი, რომელიც **„ერატოსთენეს ცხავის“** სახელითაა ცნობილი.

ელემენტარული მათემატიკის კურსიდან ცნობილია, რომ ერთისაგან განსხვავებულ მთელ, დადებით რიცხვებს, რომლებიც უნაშთოდ იყოფა მხოლოდ ერთსა და თავის თავზე, **მარტივი** რიცხვები ეწოდება.

მარტივი რიცხვებია, მაგალითად: 2,3,5,7,11 და სხვ. ძველბერძენმა მეცნიერმა ერატოსთენემ, რომელიც ჩვენს წელთაღრიცხვამდე III-II საუკუნეებში მოღვაწეობდა, შეიმუშავა იმ მარტივი რიცხვების განსაზღვრის წესი, რომელთა მნიშვნელობა n-ს არ აღემატება. აღნიშნული წესი შეიძლება აღწერილ იქნეს შემდეგი ალგორითმით.

1. თანმიმდევრობით ჩამოვწეროთ ყველა ნატურალური რიცხვი 1-დან n-მდე. გადავხაზოთ 1 და გადავიდეთ მე-2 პუნქტზე;
2. დავუშვათ, რომ m ცვლადს მივანიჭეთ რიცხვი 2-ის მნიშვნელობა. გადავიდეთ მე-3 პუნქტზე;
3. თუ $m^2 \le n$, მაშინ გადავიდეთ მე-4 პუნქტზე, წინააღმდეგ შემთხვევაში – მე-6 პუნქტზე;
4. რიცხვების მოცემულ მიმდევრობაში $m+1$ რიცხვიდან დაწყებული გადავხაზოთ თითოეული მე-m რიცხვი (ყურადღებას ნუ მივაქცევთ, თუ იგი ადრე გადახაზული იყო). გადავიდეთ მე-5 პუნქტზე;
5. მოცემულ რიცხვთა მიმდევრობაში m-ის შემდეგ პირველი გადაუხაზავი რიცხვი ჩაითვალოს m-ის ახალ მნიშვნელობად. დავბრუნდეთ მე-3 პუნქტში;
6. რიცხვთა მიმდევრობის ყველა გადაუხაზავი რიცხვი ჩაითვლება მარტივ რიცხვებად.
7. დასასრული.

ალგორითმი 3. განვიხილოთ ე.წ. საყოფაცხოვრებო ალგორითმი. როგორც ცნობილია, ჩაის დასაყენებლად საჭიროა ვიხელმძღვანელოდ შემდეგი მითითებების მიმდევრობით.

1. ჩაის დაყენების წინ ჩაიდანი მდუღარე წყლით გამოვავლოთ;
2. ჩაიდანში ჩავყაროთ ჩაი, მასში ჩავასხათ მთელი მოცულობის ნაწილი მდუღარე წყალი და გავაჩეროთ 5-6 წუთი;
3. ჩაიდანი დაყენებული ჩაით შევავსოთ მდუღარე წყლით;
4. გემოვნების მიხედვით ჩავასხათ ფინჯანში.

ჩაის დაყენების წესი ნებისმიერი ადამიანისათვის ცნობილია. მისთვის რომ ალგორითმის სახე მიგვეცა, ჩვენ მხოლოდ დავნომრეთ აღნიშნული წესის თითოეული მითითება. მსგავსი ალგორითმები მრავლად შეიძლება მოვიპოვოთ, თუ ჩავიხედავთ სამზარეულო წიგნებში, მებაღეობის, მებოსტნეობისა და მეყვავილეობის პოპულარულ სახელმძღვანელოებში, საყოფაცხოვრებო ტექნიკის მოხმარების ინსტრუქციებში. მათ შეიძლება მივაკუთვნოთ აგრეთვე ის ბრძანებები, რომლითაც სხვადასხვა საწარმოში მომუშავე პერსონალის მოქმედება განისაზღვრება. ჩვენ დროის უმეტეს ნაწილს ალგორითმების შესრულებაზე ვხარჯავთ.

1.1.3 ალგორითმების ძირითადი თვისებები

ალგორითმები მთელი რიგი საერთო ნიშან-თვისებებით ხასიათდებიან, რომელთაგან აღსანიშნავია:

1. **უნივერსალობა.** როგორც წესი, ალგორითმი მუშავდება არა ცალკეული ამოცანებისათვის ან ცალკეული საწყისი მონაცემებისათვის, არამედ ამოცანების ფართო კლასისათვის და საწყისი მონაცემების სხვადასხვა მნიშვნელობებისათვის. ასე, მაგალითად, ევკლიდეს ალგორითმში საწყის მონაცემად შეიძლება ავიღოთ ნებისმიერი მთელი დადებითი რიცხვების წყვილი.

2. **დისკრეტულობა.** ერთ-ერთი ძირითადი მოთხოვნა, რომელიც ალგორითმის შემუშავების დროს აუცილებელია დაკმაყოფილდეს, არის გამოთვლითი პროცესის ისეთ ელემენტარულ ეტაპებად დანაწევრება, რომელთა შესრულება ეჭვს არ იწვევს. ასეთი სახით მიღებული ჩანაწერი წარმოადგენს მითითებათა მოწესრიგებულ ერთობლიობას, რომელიც ალგორითმის დისკრეტულ (წყვეტილ) სტრუქტურას წარმოადგენს. ეს თვისება განსაკუთრებით თვალნათლივ ჩანს „საყოფაცხოვრებო“ ალგორითმებში.

3. **დეტერმინირებულობა.** ალგორითმის თითოეული პუნქტი საჭიროა ფორმულირებულ იქნეს ისე, რომ მისი რეალიზაციის დროს შემსრულებლის მოქმედებები ცალსახად განისაზღვროს. ამიტომ ერთიდაიგივე საწყისი მონაცემების დროს ალგორითმის როგორც საბოლოო, ისე შუალედური შედეგები, და, აგრეთვე, თითოეული პუნქტის შესრულების მიმდევრობა სხვადასხვა შემსრულებლის შემთხვევაშიც იდენტური იქნება. ასე, მაგალითად, 95-სა და 60-ის უდიდესი საერთო გამყოფის მოსაძებნად გამოთვლითი პროცესი ყოველთვის იმ მიმდევრობით წარიმართება, როგორც ეს პირველ ცხრილშია წარმოდგენილი და, ამასთან, იგივე შედეგი მიიღება.

4. **შედეგიანობა.** ნებისმიერი ალგორითმი ხასიათდება შედეგიანობით, რაც იმაში მდგომარეობს, რომ ნებისმიერი დასაშვები საწყისი მონაცემებისა და ალგორითმის ყველა პუნქტის ზუსტად შესრულების შემთხვევაში გამოთვლითი პროცესი სასრული რაოდენობის ბიჯის შემდეგ შეწყდება და ამა თუ იმ სიზუსტით საძებნი შედეგი მიიღება. იმ შემთხვევაში, როცა საბოლოო შედეგის მიღება შეუძლებელია, მაშინ გამოთვლითი პროცესი უსასრულოა და იგი არასდროს შეწყდება ან ერთ-ერთ ბიჯზე მისი შესრულება გარკვეულ წინააღმდეგობას წააწყდება. ასე, მაგალითად, ზემოთ განხილული ევკლიდეს ალგორითმი შედეგს მხოლოდ

მთელი დადებითი რიცხვების შემთხვევაში იძლევა. წინააღმდეგ შემთხვევაში, გამოთვლითი პროცესი უსასრულოდ გაგრძელდება. აღნიშნული ალგორითმის გამოყენება 0 და 0 რიცხვების უდიდესი საერთო გამყოფის მოსაძებნად ასე მცდარ შედეგს იძლევა: ალგორითმის თანახმად, იგი ნულის ტოლია, მაშინ როცა სანამდვილეში იგი საერთოდ არ არსებობს. გარდა ამისა, შესაძლებელია შემთხვევები, როცა გარკვეული ალგორითმით ზოგიერთი მოქმედების შესრულება შეუძლებელია და გამოთვლითი პროცესი რომელიმე პუნქტზე შეწყდება. ასე, მაგალითად, $y = x/(1-x^2)$ ფორმულის საშუალებით y -ს გამოთვლის ალგორითმი შეწყდება გაყოფის ეტაპზე, როცა $x = 1$, რადგან, ამ შემთხვევაში, ნულზე გაყოფას ექნება ადგილი.

განხილული მაგალითები იმაზე მეტყველებს, რომ საზოგადოდ, ალგორითმის შემუშავების დროს საჭიროა განისაზღვროს მისი გამოყენების არე, რაც, თავის მხრივ, უკავშირდება იმ საწყისი მონაცემების სიმრავლეს, რომლისთვისაც ალგორითმი შედეგიანია.

კითხვები თვითშეფასებისათვის

1. *განმარტეთ ალგორითმის ცნება.*
2. *განმარტეთ, რას გულისხმობს გამოთვლითი პროცესის ალგორითმიზაცია.*
3. *დაასაბუთეთ, თუ რა განსხვავებაა პროგრამაში დაშვებულ სინტაქსურ და ლოგიკურ შეცდომებს შორის.*
4. *განმარტეთ ალგორითმის უნივერსალობის თვისება.*
5. *განმარტეთ ალგორითმის შედეგიანობის თვისება.*
6. *განმარტეთ ალგორითმის დეტერმინირებულობის თვისება.*
7. *განმარტეთ ალგორითმის დისკრეტულობის თვისება.*

1.2. ალგორითმების გამოსახვის ფორმები

1.2.1. სიტყვიერი ფორმა

შემუშავებული ალგორითმი მომხმარებლისთვის რომ გასაგები გახდეს, საჭიროა გამოსახვის კონკრეტული საშუალებების საფუძველზე მისი ფორმალიზაცია, რომელიც გარკვეული წესების დაცვით უნდა განხორციელდეს. ალგორითმების გამოსახვის მრავალი ფორმა არსებობს, რომელიც ერთმანეთისაგან სიმარტივის, თვალსაჩინოების, კომპაქტურობისა და სხვ. მაჩვენებლების მიხედვით განსხვავდება.

ალგორითმების თეორიაში ძირითადად გავრცელებულია ალგორითმების გამოსახვის შემდეგი ფორმები: სიტყვიერი, გრაფიკული, სხვადასხვა ფორმალური ალგორითმული ენების საფუძველზე შემუშავებული საშუალებები - ოპერატორული ფორმა, ფსევდოკოდები და სხვ.

ალგორითმების გამოსახვის სიტყვიერი ფორმა შედარებით ყველაზე გავრცელებული ფორმაა. ამ ფორმით ალგორითმების წარმოდგენისათვის ჩვეულებრივი სასაუბრო ენა გამოიყენება. სიტყვიერი ფორმით წარმოდგენილ ალგორითმებში სპეციალური შეთანხმებებისა და დაშვებების საფუძველზე შესაძლებელია მათი შედარებით კომპაქტურად ჩაწერა. ქვემოთ მოცემულია ევკლიდეს ალგორითმის კომპაქტური ვარიანტი წარმოდგენილი სიტყვიერი ფორმით.

1. დასაწყისი
2. საწყისი მონაცემების შეტანა (a , b);
3. თუ $a > b$, მაშინ გადავიდეთ მე-5 პუნქტზე, წინააღმდეგ შემთხვევაში - მე-4 პუნქტზე;
4. თუ $b > a$, მაშინ გადავიდეთ მე-6 პუნქტზე, წინააღმდეგ შემთხვევაში - მე-7 პუნქტზე;
5. $a = a - b$; გადავიდეთ მე-3 პუნქტზე;
6. $b = b - a$; გადავიდეთ მე-3 პუნქტზე;
7. უსგ $= a$;
8. შედეგის (უსგ) ამობეჭდვა;
9. დასასრული.

ზემოთ წარმოდგენილი ალგორითმი შეიცავს დაწყებისა და დამთავრების აღმნიშვნელ მითითებებს. გარდა ამისა, ვინაიდან ნებისმიერი ალგორითმის შემუშავების დროს საჭიროა გავითვალისწინოთ მისი კომპიუტერული რეალიზაციაც, ამიტომ აქ დამატებულია საწყისი მონაცემების კომპიუტერში შეტანისა და საბოლოო შედეგის მონიტორის ეკრანზე ან საბეჭდ მოწყობილობაზე გამოტანის მითითებებიც (პუნქტები 2 და 8). იმ პუნქტებში, სადაც მითითებული არ არის გადასვლის მიმართულება (პუნქტები 2 და 7), იგულისხმება რიგით შემდეგ პუნქტზე გადასვლა.

საჭიროა შევნიშნოთ, რომ მათემატიკური თვალსაზრისით ჩანაწერი $a = a - b$ მცდარი გამოსახულებაა. იგი არც იგივეობაა, არც განტოლება და არც ტოლობის პირობა. გაუგებარია, რა უნდა დავითვალოთ ამგვარი ფორმულით? სამაგიეროდ, ალგორითმების თეორიაში მსგავს გამოსახულებას გარკვეული დატვირთვა გააჩნია, ვინაიდან ნიშანი " = ", რომელიც მათემატიკაში გამოიყენება როგორც ტოლობის პირობა, იგივეობის ნიშანი, განტოლების ნიშანი და, ბოლოს, ჩვეულებრივად, ფორმულის ნიშანი, აქ **მინიჭების ოპერაციას** აღნიშნავს. აქედან გამომდინარე, ზემოთ წარმოდგენილი ჩანაწერი შემდეგნაირად იკითხება: a-ს მივანიჭოთ $a - b$ სხვაობის მნიშვნელობა. ანალოგიურად იკითხება ჩანაწერი " $a = b$ ": a-ს მივანიჭოთ b-ს მნიშვნელობა.

ახლა კი შევადგინოთ $y = \frac{x^2 - 3}{\sqrt{3x} + 1}$ გამოსახულების გამოთვლის ალგორითმი სიტყვიერი ფორმით (მითითება: გამოვიყენოთ დამხმარე z ცვლადი შუალედური მნიშვნელობის დასამახსოვრებლად).

ალგორითმს შემდეგი სახე აქვს:

1. დასაწყისი
2. x-ის წაკითხვა (x-ის კონკრეტული მნიშვნელობის შეტანა);
3. $z = 3 \cdot x$
4. $z = \sqrt{z}$
5. $z = z + 1$
6. $y = x \cdot x$
7. $y = y - 3$
8. $y = y / z$
9. y-ის მნიშვნელობის ამობეჭდვა;
10. დასასრული.

1.2.2. გრაფიკული ფორმა

ალგორითმის გრაფიკულ გამოსახულებას **ბლოკ-სქემა** ეწოდება. ბლოკ-სქემის თითოეული შემადგენელი უბანი გარკვეული ფორმის გეომეტრიული ფიგურებით გამოისახება. თითოეული ფიგურა გამოთვლების ერთ გარკვეულ ეტაპს აღნიშნავს და მას **ბლოკს** უწოდებენ.

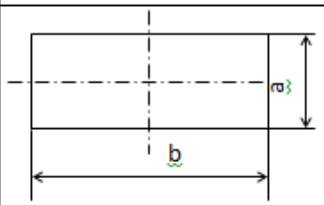
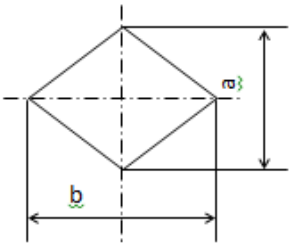
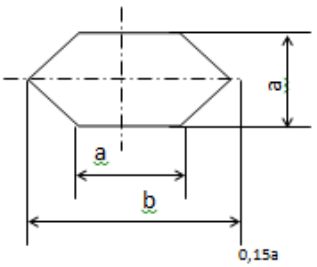
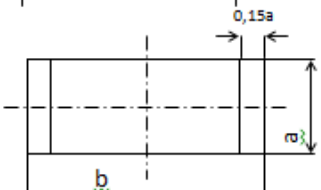
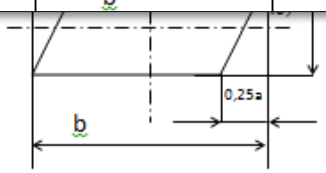
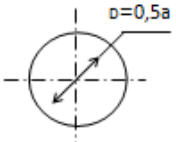
ქვემოთ მოცემულ მე-2 ცხრილში წარმოდგენილია დაპროგრამების პრაქტიკაში გამოყენებული და ყველაზე გავრცელებული ბლოკები, რომლებიც აუცილებელია ნებისმიერი ბლოკ-სქემის შედგენის დროს. ჩვეულებრივ, ბლოკში იწერება ტექსტი, რომელიც შესასრულებელ მოქმედებას აკონკრეტებს. იმ შემთხვევაში, თუ ტექსტი არ ეტევა ბლოკის ჩარჩოებში, მაშინ იგი ცალკე, შენიშვნის (კომენტარის) სახით იწერება.

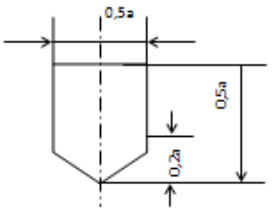
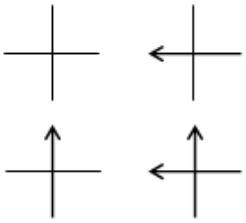
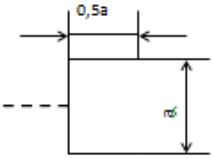
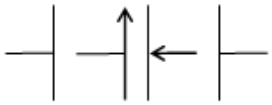
ბლოკ-სქემაში ალგორითმის შესრულების თანმიმდევრობა შესასრულებელი ოპერაციების შესაბამისი ბლოკების თანმიმდევრობით შეერთების საფუძველზე ხორციელდება შემაერთებელი ხაზებით, რომელსაც **ინფორმაციის ნაკადის ხაზებს** უწოდებენ.

ნაკადის ხაზების ძირითად მიმართულებად მიჩნეულია მიმართულება ზემოდან ქვემოთ და მარცხნიდან მარჯვნივ. იმ შემთხვევაში, როცა ერთი ბლოკის მეორე ბლოკთან შეერთების მიმართულება ძირითად მიმართულებას ემთხვევა, მაშინ ნაკადის ხაზს, როგორც წესი, მიმართულების მაჩვენებელი ისარი არ უკეთდება; ყველა დანარჩენ შემთხვევაში კი მისი გათვალისწინება აუცილებელია.

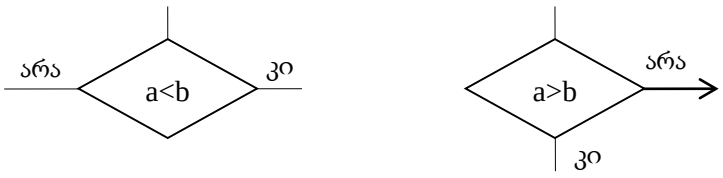
ბლოკთან მიმართებაში ინფორმაციის ნაკადის ხაზი შეიძლება იყოს შემავალი და გამომავალი. ბლოკში შემავალი ნაკადის ხაზების რაოდენობა შეზღუდული არ არის, ბლოკიდან გამომავალი ნაკადის ხაზი კი მხოლოდ ერთია შესაძლებელი. გამონაკლისს ლოგიკური ბლოკი წარმოადგენს.

ცხრილი 2. ბლოკ-სქემის ბლოკების აღნიშვნები და ფუნქციები

#	დასახელება	აღნიშვნა და ზომები	ფუნქცია
1.	2	3	4
1.	გამოთვლების ბლოკი		გამოთვლების (ოპერაციების) შესრულება, რომლის შედეგად მონაცემების მნიშვნელობები იცვლება.
2.	ლოგიკური ბლოკი		გარკვეული პირობების მიხედვით ალგორითმის შესრულების მიმართულების ამორჩევა.
3.	მოდიფიკაციის ბლოკი		ოპერაციების შესრულება ალგორითმების პუნქტების შესაცვლელად.
4.	წინასწარ განსაზღვრული პროცესის ბლოკი		ადრე შედგენილი და ცალკე აღწერილი ალგორითმების გამოყენება (ქვეპროგრამის ან სტანდარტული პროგრამის მიხედვით გამოთვლა).
	ბლოკი		
6.	გაშვება-გაჩერების ბლოკი		მონაცემების დამუშავების პროცესის დასაწყისი და დასასრული.
	წაკადის ხაზი		ბლოკებს შორის კავშირის მითითება
7.	პირველი ტიპის		

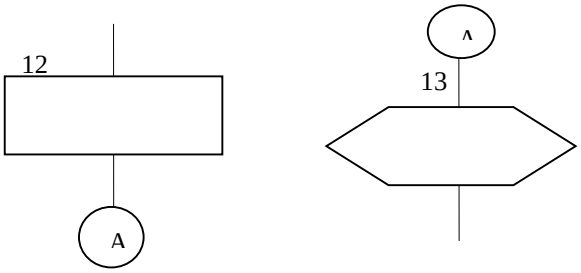
8.	შემაერთებელი		ნაკადის გაწყვეტილ ხაზებს შორის კავშირის მითითება.
9.	მეორე ტიპის შემაერთებელი		სხვადასხვა ფურცლებზე განთავსებული ალგორითმის ნაწილებს შორის კავშირის მითითება.
10.	ნაკადის ხაზების შერწყმა		სქემის ელემენტსა და განმარტებას შორის კავშირის მითითება.
11.	კომენტარი		კომენტარის მითითება.
12.	ნაკადის ხაზების გადაკვეთა		ნაკადების ხაზების ძირითადი (პირველი და ბოლო) და არაძირითადი მიმართულებები.

ლოგიკურ ბლოკს გააჩნია არანაკლებ ორი გამომავალი ნაკადის ხაზი, რომელთაგან თითოეული შეესაბამება ლოგიკური პირობის ერთ-ერთ შესაძლო ვარიანტს. ამ შემთხვევაში საჭიროა ნაკადის ხაზების მარკირება სიტყვებით „კი“ და „არა“ ისე, როგორც ეს ნახ.4-ზეა ნაჩვენები.



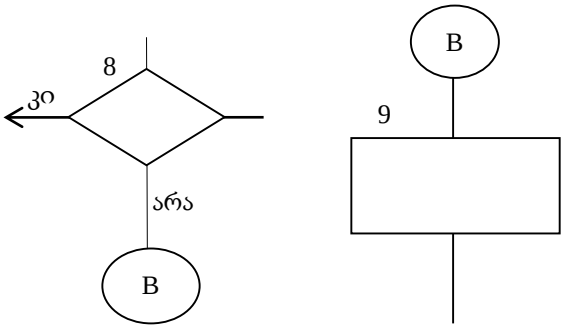
ნახ. 4 ლოგიკური ბლოკი

ბლოკ-სქემის აღწერის გაადვილების მიზნით ყველა ბლოკი, გაშვება-გაჩერების ბლოკის გარდა, თანმიმდევრობით ინომრება. ნომერი იწერება ბლოკის ზემოთ მარცხენა კუთხეში (იხილეთ ნახ. 5).



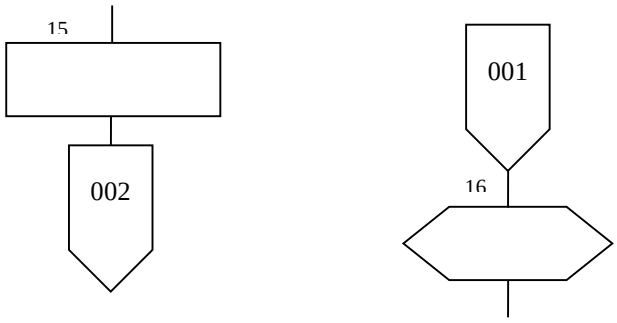
ნახ. 5 ბლოკის გადანომვრის მაგალითები

ერთმანეთისაგან დაშორებულ ბლოკებს შორის ნაკადის ხაზები შეიძლება გავწყვიტოთ, მაგრამ გაწყვეტის წერტილებში ორივე მხრიდან საჭიროა მოვათავსოთ პირველი ტიპის შემაერთებელი, რომელიც იდენტიფიცირებული იქნება რომელიმე ასოთი, ციფრით ან მათი კომბინაციით (იხილეთ ნახ. 6).



ნახ. 6 დაშორებული ბლოკების იდენტიფიკაცია

შედარებით რთული ალგორითმების შემუშავებისა და მისი გამოსახვის დროს შესაძლებელია ნახაზი გადაიტვირთოს ურთიერთგადამკვეთი ნაკადის ხაზებით. ასეთ შემთხვევაში, დასაშვებია ერთმანეთისგან დაშორებულ ბლოკებს შორის ნაკადის ხაზების გაწყვეტა. თუ ნაკადის ხაზებით ერთმანეთთან დაკავშირებული ბლოკები სხვადასხვა ფურცლებზეა მოთავსებული, მაშინ საჭიროა გამოყენებულ იქნეს მეორე ტიპის შემაერთებელი, რომელშიც ფურცლის ნომერი და იმ ბლოკის ნომერი ჩაიწერება, რომლისკენაც მიმართულია ნაკადის ხაზი (იხილეთ ნახ.7).



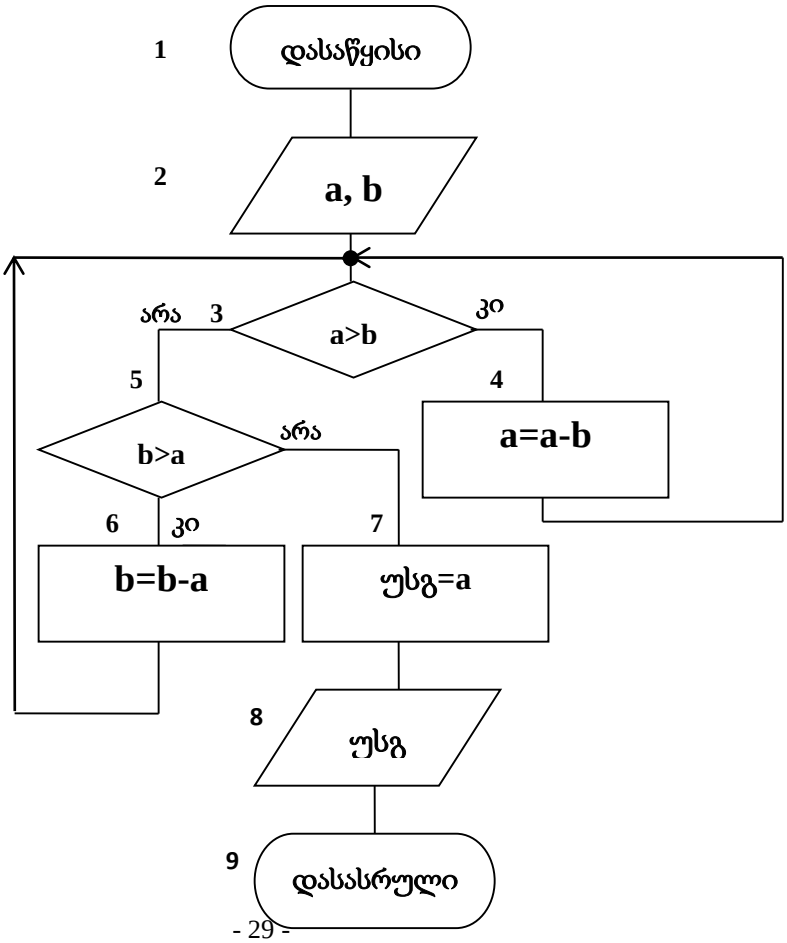
ნახ. 7 დაშორებული ბლოკების იდენტიფიკაცია

ფურცელი 001 ფურცელი 002

იმ შემთხვევაში, როცა ალგორითმის ბლოკ-სქემა ერთ ფურცელზე არ ეტევა და იგი რამდენიმეზეა წარმოდგენილი, მაშინ ნაკადის ხაზის ერთი ფურცლიდან მეორეზე გადასვლის დასაფიქსირებლად საჭიროა მეორე ტიპის შემაერთებლის გამოყენება. ამასთან, პირველ ფურცელზე გამოხაზულ შემაერთებელში საჭიროა ჩაიწეროს მეორე ფურცლის ნომერი და იმ ბლოკის ნომერი, რომელსაც ნაკადის ხაზი უერთდება, ხოლო მეორე ფურცელზე გამოხაზულ შემაერთებელში – პირველი ფურცლის ნომერი და იმ ბლოკის ნომერი, საიდანაც ნაკადის ხაზი მიემართება.

ბლოკ-სქემებით ალგორითმების წარმოდგენის პროცესი მეტად მარტივია და იგი საკმაო თვალსაჩინოებით ხასიათდება. ამასთან, ამ ფორმით ალგორითმის გამოსახვის პროცესში დეტალიზაციის ხარისხი შეუზღუდავია. დაპროგრამების პრაქტიკაში ცნობილი ფაქტია, რომ როცა ბლოკ-სქემა რთულია, იგი თავის საილუსტრაციო თვისებას კარგავს. ამიტომ ასეთ შემთხვევაში თავდაპირველად დასაშვებია მთელი გამოთვლითი პროცესის ზოგადი გამ-სხვილებული ბლოკ-სქემის შედგენა, ხოლო შემდეგ თითოეული მისი ბლოკის ცალკე ბლოკ-სქემის საშუალებით წარმოდგენა.

ახლა კი, ჩვენთვის უკვე კარგად ცნობილი ევკლიდეს ალგორითმი წარმოვადგინოთ ბლოკ-სქემის სახით (იხილეთ ნახ. 8).



ნახ. 7 ბლოკ-სქემის მაგალითი

1.2.3. ოპერატორული ფორმა

1953 წელს რუსმა მათემატიკოსმა ა. ლიაპუნოვმა დაპროგრამების თეორიაში შემოიტანა ოპერატორის ცნება, რომელმაც ალგორითმების ოპერატორული სქემებით გამოსახვის საშუალება მოგვცა.

ოპერატორი წარმოადგენს გამოთვლითი პროცესის ფუნქციურად ერთგვაროვანი ეტაპის აღწერას. ოპერატორული სქემა კი ალგორითმების წარმოდგენის ანალიზური ფორმაა ოპერატორების საშუალებით.

თავისი დანიშნულების მიხედვით ძირითადად განასხვავებენ არითმეტიკულ, ლოგიკურ და ცვლად ოპერატორებს.

არითმეტიკული ოპერატორებით აღიწერება ალგორითმის გამოთვლითი ნაწილი, რომელიც უშუალოდ არითმეტიკულ ოპერაციებთანაა დაკავშირებული; ლოგიკური ოპერატორებით - სხვადასხვა ლოგიკური პირობები, რომელზეც გამოთვლითი პროცესის მიმდინარეობის მიმართულებაა დამოკიდებული; ცვლადი ოპერატორებით კი გათვალისწინებულია დამხმარე სიდიდეების, რომელსაც პარამეტრებს უწოდებენ, შესაძლო ცვლილებების დაფიქსირება.

ალგორითმის სრულყოფილად წარმოდგენის მიზნით, გარდა არითმეტიკული და ლოგიკური ოპერატორებისა, გამოყენებულია სპეციალური ნიშნაკები (ოპერატორები), რომლითაც აღიწერება პროცესის დაწყება, დამთავრება, საწყისი მონაცემების შეტანა-გამოტანა, შედეგების ამობეჭდვა და ა.შ.

სქემაში ოპერატორები, ჩვეულებრივ, პირობითი (სტანდარტული) სიმბოლოებით აღინიშნება. დაპროგრამების პრაქტიკაში გამოყენებულია შემდეგი სიმბოლოები:

И0 – პროცესის დასაწყისი;

В - საწყისი მონაცემების შეტანა;

А - არითმეტიკული ოპერატორი;

Р - ლოგიკური ოპერატორი;

П - ბეჭდვის ოპერატორი;

V - ცვლადი ოპერატორი;

Я – გაჩერების ოპერატორი.

გარდა ამისა, მართვის მარცხნიდან მარჯვნივ გადაცემის აღსანიშნავად ერთი ოპერა-ტორიდან იმ ოპერატორამდე, რომელიც მეზობლად არ მდებარეობს, გამოყენებულია ისარი. თუ ერთი ოპერატორიდან მის მარჯვნივ მდგომ მეზობელ ოპერატორზე მართვის გადაცემას ადგილი არა აქვს, მაშინ გამოყენებულია ნიშანი ";" (წერტილ-მძიმე). თითოეული ოპერატორი აღჭურვილია

ქვედა ინდექსით, რომელიც მის რიგით ნომერს შეესაბამება სტრიქონში. პარამეტრზე დამოკიდებული არითმეტიკული და ლოგიკური ოპერატორები აღინიშნება როგორც $A5^i$, $A10^{i,j}$ $P12^i$ და ა.შ., სადაც i, j პარამეტრებია. ლოგიკურ ოპერატორებს ხშირად გამოსახავენ ფუნქციის სახით, რომლის არგუმენტს შესამოწმებელი პირობა წარმოადგენს. ასე მაგალითად, P(x>0), P(1≤x≤2) და ა.შ.

ცვლადი ოპერატორები, მართალია, პარამეტრებზე არაა დამოკიდებული, მაგრამ მის მნიშვნელობას ცვლის. ამ ფაქტს აღნიშნავენ შემდეგი სიმბოლოებით: V3(i), V7(i,j) და ა.შ., სადაც ფრჩხილებში იმ პარამეტრს მიუთითებენ, რომელიც იცვლება.

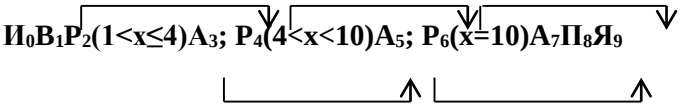
სქემას, რომელშიც გამოყენებულია ზემოთ განხილული სპეციალური ნიშნაკები, პროგრამის **ლოგიკურ სქემას** უწოდებენ.

ოპერატორული ფორმით გამოსახული ალგორითმების ფუნქციონირება იწყება ყველაზე მარცხნივ მდგომი ოპერატორით, რომლის შემდეგ სრულდება მის მარჯვნივ მეზობლად მდგომი ოპერატორი, თუკი რომელიმე სხვა ოპერატორზე გადასვლა მითითებული არ არის.

აღსანიშნავია, რომ ოპერატორების საშუალებით წარმოდგენილი ალგორითმები შედარებით კომპაქტურობით ხასიათდება, თუმცა ნაკლები თვალსაჩინოების გამო მათი გამოყენება რთული ალგორითმების შესამუშავებლად შეზღუდულია.

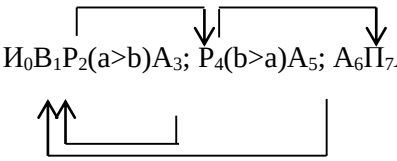
შევადგინოთ და ოპერატორული ფორმით ჩავწეროთ ქვემოთ წარმოდგენილი გამო-სახულების გამოთვლის ალგორითმი:

$$F = \begin{cases} x + a, & 1 < x \le 4 \\ x - c^2 z, & 4 < x < 10 \\ x - A, & x = 10 \end{cases}$$



სადაც $A_3=x+A$; $A_5=x-c^2z$; $A_7=x-A$.

ევკლიდეს ალგორითმს კი ოპერატორული ფორმით წარმოდგენის შემთხვევაში შემდეგი სახე აქვს:



სადაც И0 – დაწყების ოპერატორი.

В1 – საწყისი მონაცემების (a და b) შეტანის ოპერატორი.

Р2 (a > b) – ლოგიკური ოპერატორი (შემოწმება პირობაზე a > b).

А3 – არითმეტიკული ოპერატორი (a=a-b).

Р4 (b>a)- ლოგიკური ოპერატორი (შემოწმება პირობაზე b>a).

А5 – არითმეტიკული ოპერატორი (b=b-a).

А6 – არითმეტიკული ოპერატორი (მინიჭება უსგ=a).

П7 – ბეჭდვის ოპერატორი (უსგ).

Я8 – დასასრული.

ფორმალური ალგორითმული ენების საშუალებით ალგორითმების წარმოდგენისათვის სპეციალური საკვანძო სიტყვები და ბრძანებებია გამოყენებული. ჩვენ მათ არ განვიხილავთ, ვინაიდან იგი საჭიროებს ერთი მანქანური ენის ცოდნას მაინც. აქ მხოლოდ აღვნიშნავთ, რომ აღწერის ეს ფორმა იმ შემთხვევაში გამოიყენება, როცა გათვალისწინებულია ალგორითმის კომპიუტერზე რეალიზაცია.

რაც შეეხება, ალგორითმის ჩაწერის ცხრილურ ფორმას, მისი საშუალებით ევკლიდეს ალგორითმი უკვე წარმოვადგინეთ (იხილეთ პირველი ცხრილი), სადაც მითითებულია a და b ცვლადების მნიშვნელობები ალგორითმის შესრულების ყოველ ბიჯზე და შესასრულებელი ბიჯების (ჩვენ შემთხვევაში, პუნქტების) ნომრები.

კითხვები თვითშეფასებისათვის

1. *განმარტეთ, რას წარმოადგენს ალგორითმის ბლოკ-სქემა და რომელი ძირითადი ნაწილებისგან შედგება ის.*
2. *განმარტეთ ნაკადის ხაზების ძირითადი და არაძირითადი მიმართულებები.*
3. *დაახასიათეთ ალგორითმის ბლოკ-სქემაში გამოყენებული ლოგიკური ბლოკი.*
4. *განმარტეთ, რთული ალგორითმების შემუშავების შემთხვევაში (როდესაც ალგორითმის ბლოკ-სქემა სხვადასხვა ფურცლებზეა მოთავსებული), როგორ ხდება ალგორითმის შემადგენელი ბლოკების ერთმანეთთან დაკავშირება.*
5. *განმარტეთ ოპერატორის ცნება ალგორითმების ოპერატორული სქემებით გამოსახვის შემთხვევაში.*

1.3. ალგორითმების სტრუქტურა

1.3.1. წრფივი სტრუქტურის ალგორითმები

გამოთვლით პროცესში მოქმედებათა შესრულების თანმიმდევრობის მიხედვით ალგორითმები შეიძლება სამ ჯგუფად დაიყოს. პირველ ჯგუფს შეადგენენ წრფივი სტრუქტურის ალგორითმები, მეორე ჯგუფს – განშტოებული სტრუქტურის ალგორითმები, ხოლო მესამე ჯგუფს კი – ციკლური სტრუქტურის ალგორითმები.

ალგორითმებს, რომელშიც ყველა მოქმედება, ელემენტარული ოპერაციები, წრფივი თანმიმდევრობით, ერთიმეორის მიყოლებით სრულდება და გამოთვლების მიმართულება საწყისი მონაცემების კონკრეტულ მნიშვნელობებზე არ არის დამოკიდებული, **წრფივი სტრუქტურის** ალგორითმები ეწოდება.

განვიხილოთ წრფივი სტრუქტურის ალგორითმების შედგენის მაგალითები.

მაგალითი 1.: ამოცანის მათემატიკური ფორმულირება. განვსაზღვროთ სამკუთხედის ფართობი ჰერონის ფორმულით: $S = \sqrt{p(p-a)(p-b)(p-c)}$,

სადაც a, b, და c – სამკუთხედის გვერდების სიგრძეებია, ხოლო $p = (a+b+c)/2$ - სამკუთხედის ნახევარპერიმეტრი.

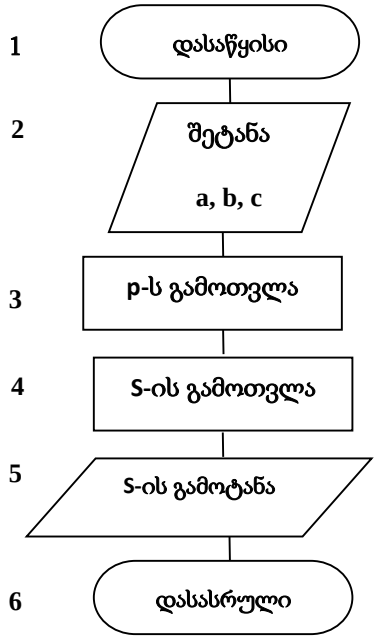
ამოცანის გადაწყვეტის ალგორითმი:

1. საწყისი მონაცემების შეტანა(a, b, c).
2. ნახევარპერიმეტრის გამოთვლა(p).
3. ფართობის გამოთვლა(s).
4. შედეგის გამოტანა (s).

ალგორითმის ბლოკ-სქემა მე-9 ნახაზზეა წარმოდგენილი.

შენიშვნა: მოცემულ სქემაში თავდაპირველად გამოითვლება ნახევარპერიმეტრის მნიშვნელობა p, ხოლო შემდეგ ფართობის მნიშვნელობა s, რომლის გამოსათვლელად გამოიყენება p. ეს საშუალებას გვაძლევს თავიდან ავიცილოთ ერთიდაიგივე სიდიდის რამდენჯერმე გამოთვლა, რაც თავის მხრივ, ამოცანის გადაწყვეტისათვის საჭირო კომპიუტერული დროის შემცირებას უზრუნველყოფს. სქემაში ბლოკები განლაგებულია იმ მიმდევრობით, რა მიმდევრობითაც გამოთვლებია შესრულებული. ბლოკების ნებისმიერი

გადაადგილების შემთხვევაში ამოცანის გადაწყვეტა შეუძლებელი გახდება.



ნახ. 8 მაგალითი 1-ის შესაბამისი ბლოკ-სქემა

მაგალითი 2: ამოცანის მათემატიკური ფორმულირება. გამოვთვალოთ წაკვეთილი კონუსის მოცულობა ფორმულით: $V = \frac{1}{3}\pi H(R^2 + Rr + r^2)$, სადაც H- წაკვეთილი კონუსის სიმაღლეა, ხოლო R და r- შესაბამისად ქვედა და ზედა ფუძეების რადიუსები.

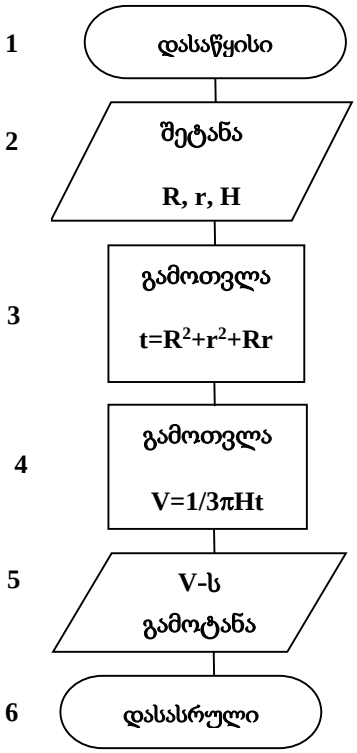
ამოცანის გადაწყვეტის ალგორითმი.

შემოვიტანოთ შუალედური პარამეტრი:

$t=R^2+Rr+r^2$, მაშინ $V=\frac{1}{3}\pi Ht$

1. შეტანა(R, r, H)
2. t-ს გამოთვლა
3. V-ს გამოთვლა
4. შედეგის გამოტანა (V).

ალგორითმის ბლოკ-სქემა მე-10 ნახაზზეა წარმოდგენილი.



ნახ. 9 მაგალითი 2-ის შესაბამისი ბლოკ-სქემა

1.3.1. განშტოებული სტრუქტურის ალგორითმები

პრაქტიკაში ალგორითმების წრფივი სტრუქტურით წარმოდგენა იშვიათად არის შესაძლებელი. ასე მაგალითად: ყველაზე უმარტივესი გამოთვლითი პროცესის აღსაწერად, როგორიცაა

$$y = \begin{cases} 2x^2 + 3, & x \le 0 \\ 2x^2 - 3, & x > 0 \end{cases}$$

წრფივი სტრუქტურა არ არის საკმარისი, რადგან x ცვლადის მოცემული მნიშვნელობის მიხედვით საჭიროა ერთ-ერთი ფორმულის ამორჩევა. ამოცანების ალგორითმიზაციის დროს ძალიან ხშირად აქვს ადგილი მსგავსი პროცედურების გამოყენებას, რომლის დროსაც საწყისი

მონაცემების ან შუალედური შედეგების ანალიზის საფუძველზე გამოთვლითი პროცესის მიმართულება იცვლება.

ალგორითმებს, რომლებიც ასეთ პროცესებს იყენებენ, **განშტოებული სტრუქტურის** ალგორითმებს უწოდებენ, ხოლო მიმართულებებს, რომლის მიხედვითაც გამოთვლითი პროცესი მიმდინარეობს - ალგორითმის შტოებს.

საწყისი ან შუალედური მონაცემების ანალიზი ხორციელდება ლოგიკური ოპერატორის (ლოგიკური ბლოკის) საშუალებით, რაც ალგორითმში განშტოებების წარმოქმნას განაპირობებს. თითოეულ კონკრეტულ შემთხვევაში, გამოთვლითი პროცესი ერთ-ერთი შტოს მიხედვით შესრულდება, მეორეს მიხედვით კი გამოირიცხება.

მაგალითი 3: ამოცანის მათემატიკური ფორმულირება.

გამოვთვალოთ $y = \begin{cases} x+5, & x > 0 \\ x-5, & x \le 0 \end{cases}$

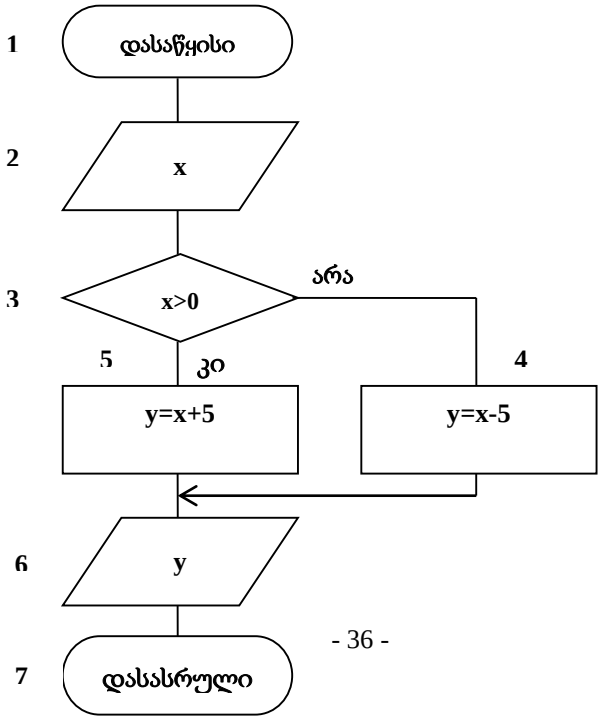
გამოსახულების მნიშვნელობა *x*-ის ნებისმიერი მნიშვნელობის შემთხვევაში.

ამოცანის გადაწყვეტის ალგორითმი.

1. საწყისი მონაცემის შეტანა (*x*).
2. პირობის შემოწმება *x*>0; თუ უტოლობა სამართლიანია, მაშინ გამოვთვალოთ *y*=*x*+5, წინააღმდეგ შემთხვევაში გამოვთვალოთ *y*=*x*-5.
3. შედეგის გამოტანა (*y*).

ალგორითმის ბლოკ-სქემა მე-11 ნახაზზეა წარმოდგენილი.

როგორც ნახაზიდან ჩანს, ბლოკ-სქემას ორი შტო გააჩნია. როცა *x*>0, მაშინ პროცესი წარიმართება 3, 5 ბლოკებში წარმოდგენილი გამოთვლების შესაბამისად; წინააღმდეგ შემთხვევაში - 3, ბლოკების შესაბამისად. ამგვარად, ორი ბლოკიდან მხოლოდ ერთი შესრულდება: 4 ან 5



.

ნახ. 10 მაგალითი 3-ის შესაბამისი ბლოკ-სქემა

მაგალითი 4: ამოცანის მათემატიკური ფორმულირება.

გამოვთვალოთ: $z = \frac{1}{xy}$ ფუნქციის მნიშვნელობა.

ერთი შეხედვით ამოცანის გადაწყვეტა წრფივი სტრუქტურის ალგორითმითაა შესაძლებელი. მაგრამ, ვინაიდან ალგორითმი, საზოგადოდ, უნივერსალობის გათვალისწინებით უნდა შემუშავდეს, ანუ, სხვა სიტყვებით რომ ვთქვათ, მოცემული ამოცანა საწყისი სიდიდეების ნებისმიერი მნიშვნელობის დროს უნდა ამოიხსნას, ამიტომ მიზანშეწონილია განშტოებული სტრუქტურის ალგორითმის შედგენა, რომელშიც გათვალისწინებული იქნება ცვლადების ორი სახის მნიშვნელობები: ნულოვანი და არანულოვანი. ამგვარად, გამოთვლითი პროცესი შეიძლება აღწერილ იქნეს შემდეგი გამოსახულებით:

გამოვთვალოთ: $z = \frac{1}{xy}$, როცა $xy \neq 0$ (I შტო).

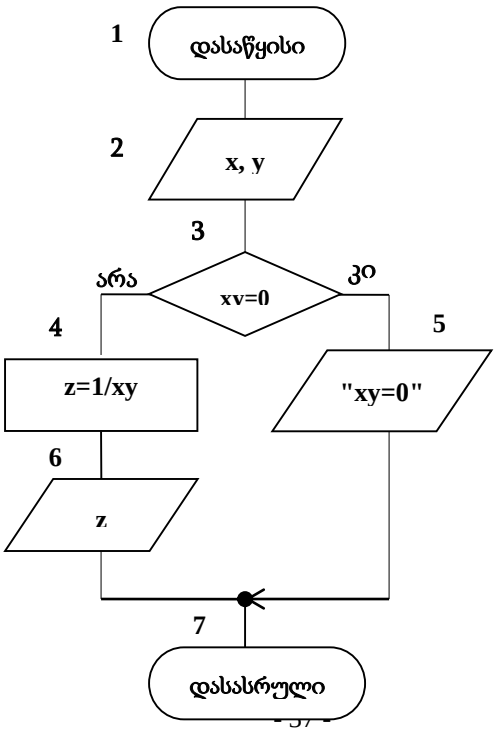
ამოვბეჭდოთ $xy = 0$, როცა $xy = 0$ (II შტო).

ამოცანის გადაწყვეტის ალგორითმი.

1. საწყისი მონაცემების შეტანა (x, y).
2. პირობის შემოწმება $xy = 0$; თუ ტოლობა სამართლიანია, მაშინ ამოვბეჭდოთ $xy = 0$, წინააღმდეგ შემთხვევაში გამოვთვალოთ $z = \frac{1}{xy}$
3. შედეგის გამოტანა (z).

ალგორითმის ბლოკ-სქემა მე-12 ნახაზზეა წარმოდგენილი.

როგორც ნახაზიდან ჩანს, ბლოკ-სქემის პირველ შტოს შეადგენენ 3, 4, 6 ბლოკები, ხოლო მეორე შტოს 3, 5 ბლოკები. ამგვარად, ორი ბლოკიდან მხოლოდ ერთი შესრულდება: 4, 6 ან 5.



ნახ. 11 მაგალითი 4-ის შესაბამისი ბლოკ-სქემა

1.3.2. ციკლური სტ[illegible]

ციკლური სტრუქტურის ალგორითმები დანარჩენი სტრუქტურის ალგორითმებთან შედარებით ყველაზე რთულია.

ალგორითმებს, რომელშიც ადგილი აქვს გამოთვლითი პროცესის ცალკეული ეტაპის მრავალჯერ გამეორებას, **ციკლური სტრუქტურის** ალგორითმები ეწოდება, ხოლო გამოთვლითი პროცესის მრავალჯერგამეორებად უბნებს, სადაც ბოლო მოქმედების შესრულების შემდეგ ისევ პირველი მოქმედება სრულდება (როგორც წესი, განსხვავებული რიცხვითი მონაცემებით) - **ციკლი** ეწოდება.

ციკლი ამოცანის ალგორითმიზაციისა და დაპროგრამების საფუძველს შეადგენს. მისი გამოყენება ალგორითმის მოცულობისა და შესაბამისი კომპიუტერული პროგრამის სიგრძის მნიშვნელოვანი შემცირების საშუალებას იძლევა. ჩვეულებრივ, ციკლის შესრულების დროს რაღაც პარამეტრის მნიშვნელობა თანმიმდევრობით იცვლება. ამ პარამეტრს **ციკლის პარამეტრი** ან **მმართველი ცვლადი** ეწოდება. მისი საშუალებით, ჯერ ერთი, დასამუშავებელი სიდიდეების მნიშვნელობები იცვლება, ხოლო, მეორე მხრივ, რაღაც მომენტში ადგილი აქვს ციკლიდან გამოსვლას. ერთ ციკლში შეიძლება გვქონდეს რამდენიმე მმართველი ცვლადი (ციკლის პარამეტრი).

ციკლის ორგანიზაციისათვის საჭიროა შევასრულოთ შემდეგი მოქმედებები:

1. ციკლის დაწყების წინ დავუშვათ მმართველი ცვლადის (პარამეტრის) საწყისი მნიშვნელობა;
2. ციკლის ყოველი ახალი გამეორების წინ შევცვალოთ მმართველი ცვლადის (პარამეტრის) მნიშვნელობა;
3. შევამოწმოთ ციკლის დამთავრების ან გამეორების პირობა;
4. ვმართოთ ციკლი ანუ შევასრულოთ ციკლის დასაწყისში გადასვლის პროცედურა, თუ ციკლი არაა დამთავრებული, ან განვახორციელოთ ციკლიდან გამოსვლის პროცედურა მისი დამთავრების შემთხვევაში.

გამეორებათა რაოდენობის განსაზღვრის წესის მიხედვით განასხვავებენ არითმეტიკულ და იტერაციულ ციკლებს.

არითმეტიკული ეწოდება ისეთ ციკლს, რომელშიც გამეორებათა რიცხვი წინასწარ ცნობილია. მის დამახასიათებელ თვისებებს შეადგენს ის, რომ მისი ორგანიზაციისათვის საჭიროა დავუშვათ: ციკლის პარამეტრის საწყისი და საბოლოო მნიშვნელობები, რომელიც, თავის მხრივ,

გამეორებათა საჭირო რაოდენობას განსაზღვრავს, და აგრეთვე ციკლის პარამეტრის ცვლილების კანონი.

ციკლის რაოდენობათა რიცხვის მართვა შესაძლებელია როგორც პირდაპირი მთვლელის, ისე უკუმთვლელის საშუალებით.

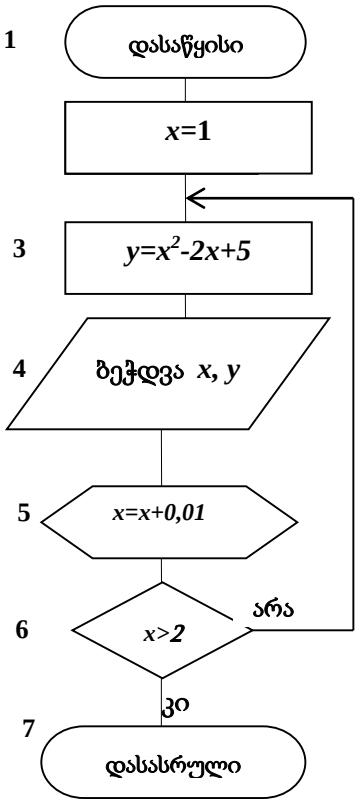
პირდაპირი მთვლელი. ციკლის რაოდენობთა რიცხვის დასათვლელად შეირჩევა ცვლადი, რომელიც მხოლოდ მთელრიცხვა მნიშვნელობებს იღებს. მას მთვლელს უწოდებენ. ციკლის თითოეული გამეორების დროს მისი მნიშვნელობა იზრდება, რითაც მიმდინარე ციკლის შესრულება ფიქსირდება. მთვლელის მნიშვნელობა გამეორებათა მოცემულ რიცხვთან (ეტალონურ მნიშვნელობასთან) შედარდება, რომლის საფუძველზე განხორციელდება ციკლის დასაწყისში გადასვლის ან ციკლიდან გამოსვლის პროცედურა.

უკუმთვლელი. ამ შემთხვევაში მთვლელს მიენიჭება ეტალონური მნიშვნელობა. მორიგი ციკლის შესრულების შემდეგ მთვლელის მნიშვნელობა შემცირდება და მიღებული მნიშვნელობა ნულთან შედარდება. შედარების საფუძველზე განხორციელდება ციკლის დასაწყისში გადასვლის ან ციკლიდან გამოსვლის პროცედურა.

განვიხილოთ ციკლური სტრუქტურის ალგორითმები:

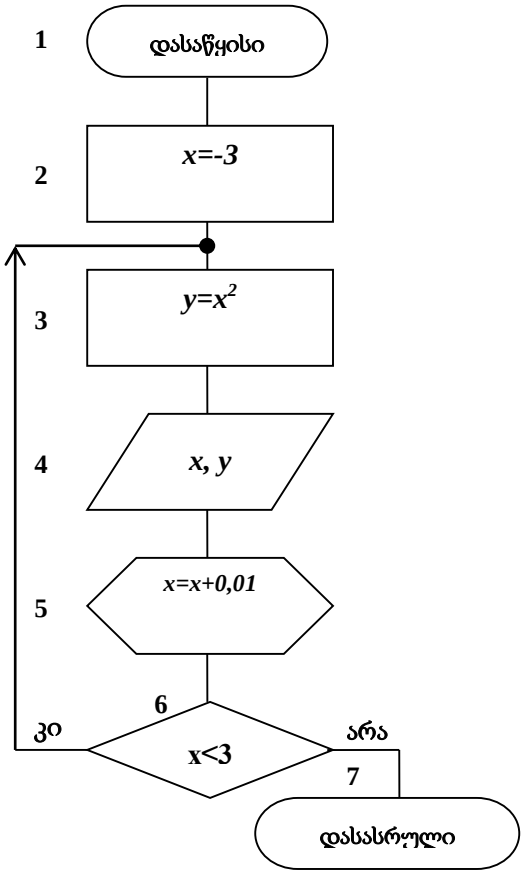
მაგალითი 5: შევადგინოთ $y(x)=x^2 - 2x+5$ ფუნქციის მნიშვნელობათა ცხრილი, თუ x არგუმენტი [1;2] ინტერვალში h=0,01 ბიჯით იცვლება.

ალგორითმის ბლოკ-სქემა მე-13 ნახაზზეა წარმოდგენილი.



ნახ. 12 მაგალითი 5-ის შესაბამისი ალგორითმის ბლოკ-სქემა

მაგალითი 6. შევადგინოთ $y=x^{-}$ ფუნქციის ძიძვხელოგათა ცხრილის გამოთვლის ალგორითმი x არგუმენტის (-3)-დან 3-მდე 0,01 ბიჯით ცვლილების შემთხვევაში. ალგორითმი მე-14 ნახაზზეა წარმდგენილი ბლოკ-სქემის სახით.



ნახ. 13 მაგალითი 6-ის შესაბამისი ალგორითმის ბლოკ-სქემა

1.3.3. იტერაციული ციკლები

იტერაციული ეწოდება ისეთ ციკლს, რომელშიც ელემენტარული პროცედურების გამეორებათა რიცხვის წინასწარ განსაზღვრა შეუძლებელია. ციკლში შემავალი პროცედურები მეორდება მანამ, სანამ ადგილი აქვს რაღაც ლოგიკური პირობის შესრულებას. გამოთვლები იტერაციულ პროცესებში თანდათანობითი მიახლოების მეთოდების საფუძველზეა რეალიზებული.

იტერაციული ციკლის სტრუქტურა არითმეტიკულის ანალოგიურია, მხოლოდ იმ განსხვავებით, რომ ციკლიდან გამოსვლის მთავარ კრიტერიუმს წარმოადგენს იმ პირობის შესრულება, რომელიც დაკავშირებულია მონოტონურად ცვლადი სიდიდის მნიშვნელობის

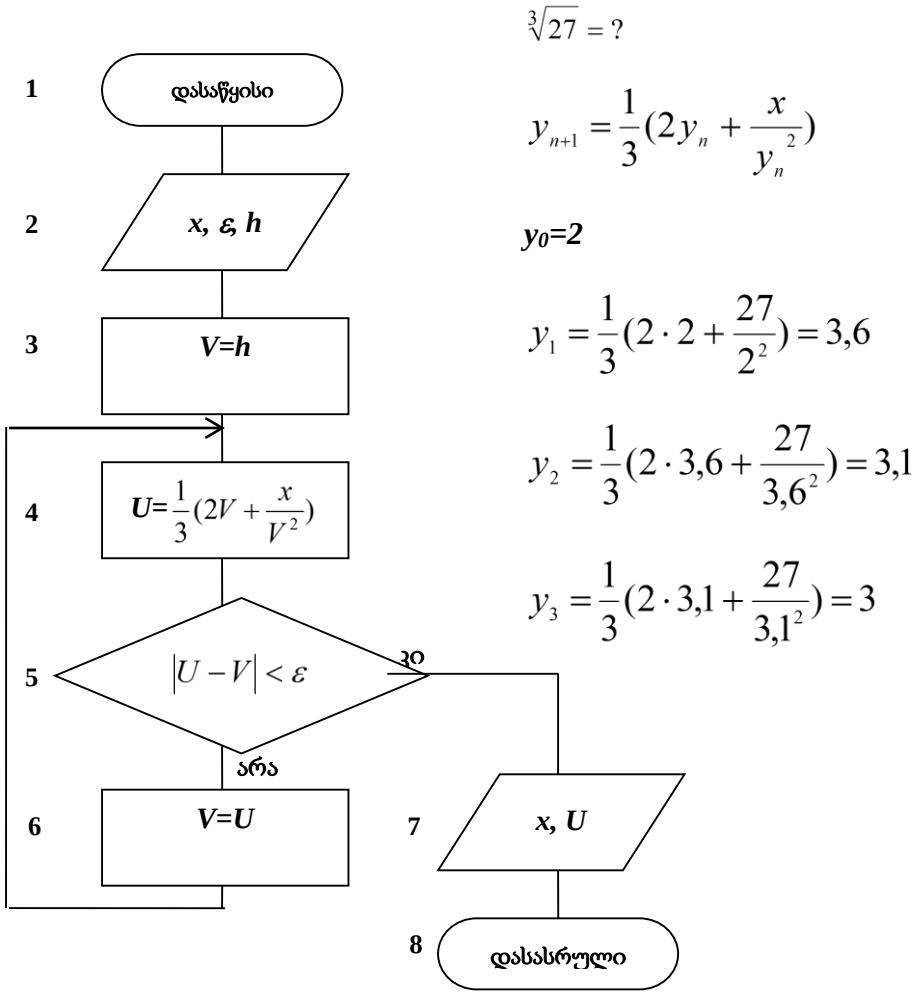
შემოწმებასთან. პრაქტიკულად იგი, როგორც წესი, იტერაციული პროცესის თითოეულ ბიჯზე გამოთვლების სიზუსტეს (ცდომილებას) განსაზღვრავს.

მაგალითი 7: შევადგინოთ $y=\sqrt[3]{x}$ კუბური ფესვის გამოთვლის ალგორითმის ბლოკ-სქემა შემდეგი იტერაციული ფორმულის საფუძველზე $Y_{n+1}=\frac{1}{3}\left(2y_n+\frac{x}{y_n^2}\right)$;

თუ ცნობილია გამოთვლების ცდომილება $|y_{n+1}-y_n|<\varepsilon$ და საწყისი მიახლოება $y_0=h$.

მოცემულ ამოცანაში ციკლის გამეორებათა რიცხვი წინასწარ უცნობია. ციკლის სამართავად გამოყენებულია გამოთვლების ცდომილების მოცემული ε სიდიდე. თუ იტერაციის მორიგ ბიჯზე ცდომილება მეტია ε სიდიდეზე, მაშინ ციკლში შემავალი პროცედურები მეორდება ზემოთ მოცემული ფორმულებით შემდეგი მიახლოების მისაღებად, წინააღმდეგ შემთხვევაში ადგილი აქვს ციკლიდან გამოსვლას.

ამოცანის ამოხსნის ბლოკ-სქემა მე-15 ნახაზზეა წარმოდგენილი.



ნახ. 14 მაგალითი 7-ის შესაბამისი ალგორითმის ბლოკ-სქემა

1.3.4. რთული ციკლები

ციკლის სტრუქტურის მიხედვით განასხვავებენ მარტივ და რთულ ციკლებს. **მარტივი** ეწოდება ციკლს, რომელიც თავის ტანში არ შეიცავს სხვა ციკლს. ასეთ ციკლებს ჩვენ ზემოთ ვიყენებდით.

ნებისმიერ ციკლს, რომელიც თავის ტანში შეიცავს ერთ ან რამდენიმე მარტივ სხვა ციკლს, **რთული** ციკლი ეწოდება.

რთული ციკლის შემთხვევაში ციკლს, რომელიც თავის თავში სხვა ციკლებს შეიცავს, **გარე** ციკლი ეწოდება, ხოლო ციკლს, რომელიც გარე ციკლშია მოთავსებული, **შიდა** ციკლი.

როგორც გარე, ისე შიდა ციკლის ორგანიზაციის წესი ისეთივეა, როგორც მარტივი ციკლისა. საზოგადოდ, რთულ ციკლში შიგა და გარე ციკლის პარამეტრები ერთდროულად არ იცვლება: გარე ციკლის პარამეტრის ერთი მნიშვნელობის დროს ადგილი აქვს შიდა ციკლის პარამეტრის ყველა მნიშვნელობის გადარჩევას.

მაგალითი 8: მოცემულია 5 სტრიქონიანი და 6 სვეტიანი მატრიცა

$$A = \begin{vmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \end{vmatrix}$$

შევადგინოთ A მატრიცის ელემენტების ჯამის გამოთვლის ბლოკ-სქემა. მათემატიკური ფორმულირება: ვიპოვოთ A მატრიცის ელემენტების S ჯამი:

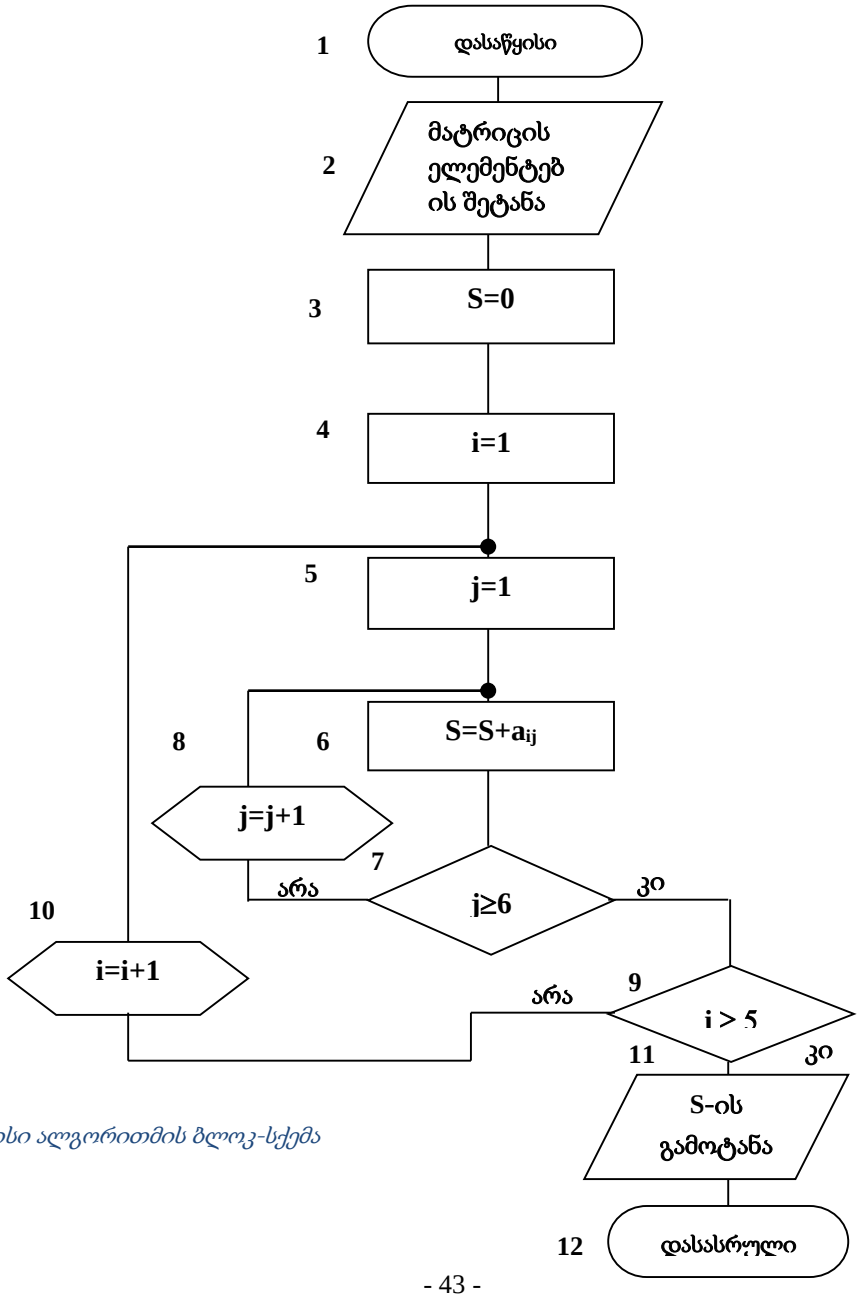
$$S = a_{11} + a_{12} + ... + a_{16} + a_{21} + a_{22} + ... + a_{26} + a_{51} + a_{52} + ... + a_{56} =$$
$$= \sum_{i=1}^{5} \sum_{j=1}^{6} a_{ij}$$

მსგავსი ტიპის ამოცანებში ელემენტების აჯამვა მიზანშეწონილია განვახორციელოთ მატრიცის ან სტრიქონის ან სვეტის მიხედვით. შევირჩიოთ პირველი ვარიანტი. გამოთვლების ცალკეული ეტაპის შესრულება რომ გასაგები გახდეს S ჯამის გამოთვლის ფორმულა შემდეგნაირად წარმოვადგინოთ:

$$S = \begin{cases} a_{11} + a_{12} + a_{13} + a_{14} + a_{15} + a_{16} \\ + a_{21} + a_{22} + a_{23} + a_{24} + a_{25} + a_{26} \\ + a_{31} + a_{32} + a_{33} + a_{34} + a_{35} + a_{36} \\ + a_{41} + a_{42} + a_{43} + a_{44} + a_{45} + a_{46} \\ + a_{51} + a_{52} + a_{53} + a_{54} + a_{55} + a_{56} \end{cases}$$

როგორც ამ უკანასკნელიდან ჩანს, ფიქსირებულ i-ურ სტრიქონში ყველა ელემენტი შეჯამდება j პარამეტრის 1-დან 6-მდე 1-ის ტოლი ბიჯით თანმიმდევრობით ცვლილების შედეგად. ერთი სტრიქონის ელემენტების ჯამის მიღების შემდეგ ადგილი აქვს შემდეგ სტრიქონზე გადასვლას i პარამეტრის მიმდინარე მნიშვნელობის 1 ერთეულით გაზრდასთან ერთად. ამასთან, შემდეგი სტრიქონის ელემენტების ჯამის გამოსათვლელად j პარამეტრის მნიშვნელობები კვლავ თანმიმდევრობით შეიცვლება 1-დან 6-მდე 1-ის ტოლი ბიჯით და ა.შ. (სანამ i≤5).

ალგორითმის ბლოკ-სქემა მე-16 ნახაზზეა წარმოდგენილი.

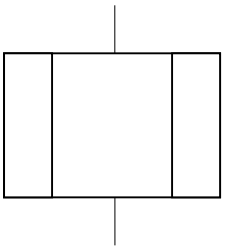


ნახ. 15 მაგალითი 8-ის შესაბამისი ალგორითმის ბლოკ-სქემა

1.3.5. დამხმარე ალგორითმი

(წინასწარ განსაზღვრული პროცესი)

საწყისი ამოცანის ალგორითმის შედგენის პროცესში შეიძლება დაგვჭირდეს სხვა, უკვე ცნობილი ამოცანის ალგორითმის გამოყენება. ალგორითმს, რომელსაც სხვა ალგორითმების შედგენისას ვიყენებთ **დამხმარე ალგორითმი** ეწოდება. საწყისი ანუ ძირითადი ალგორითმის ბლოკ-სქემაში დამხმარე ალგორითმის მთლიანად მოყვანა აუცილებელი არ არის. მის ნაცვლად ბლოკ-სქემაში უნდა ჩავრთოთ სპეციალური ბლოკი „წინასწარ განსაზღვრული პროცესი“, რომელსაც აქვს სახე (ნახ.17):

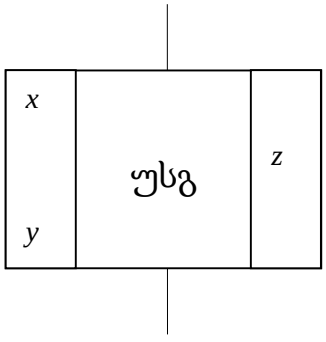


ნახ. 16 ბლოკი „წინასწარ განსაზღვრული პროცესი“

ბლოკის შუა ნაწილში იწერება დამხმარე ალგორითმის სახელი. მარცხნივ - ალგორითმის საწყისი სიდიდეები, მარჯვნივ - ის სიდიდეები, რომლებიც დამხმარე ალგორითმის შესრულების შედეგების მნიშვნელობებს ღებულობენ.

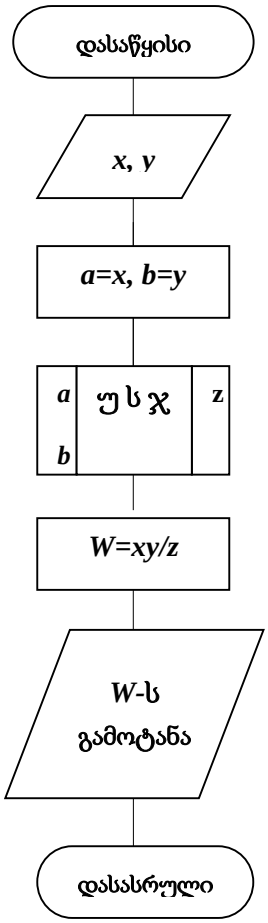
მაგალითისთვის განვიხილოთ *x* და *y* რიცხვების უმცირესი საერთო ჯერადის განსაზღვრის ამოცანა. იგი გამოითვლება ფორმულით: $W = \frac{xy}{z}$

სადაც *W*- უმცირესი საერთო ჯერადია, *Z* – *x* და *y* რიცხვების უდიდესი საერთო გამყოფი. თუ ორი რიცხვის უდიდესი საერთო გამყოფის განსაზღვრის ალგორითმი ცნობილია, მის აღსანიშნავად შეიძლება გამოვიყენოთ ბლოკი (ნახ.18):



ნახ. 17 ბლოკი უდიდესი საერთო გამყოფი

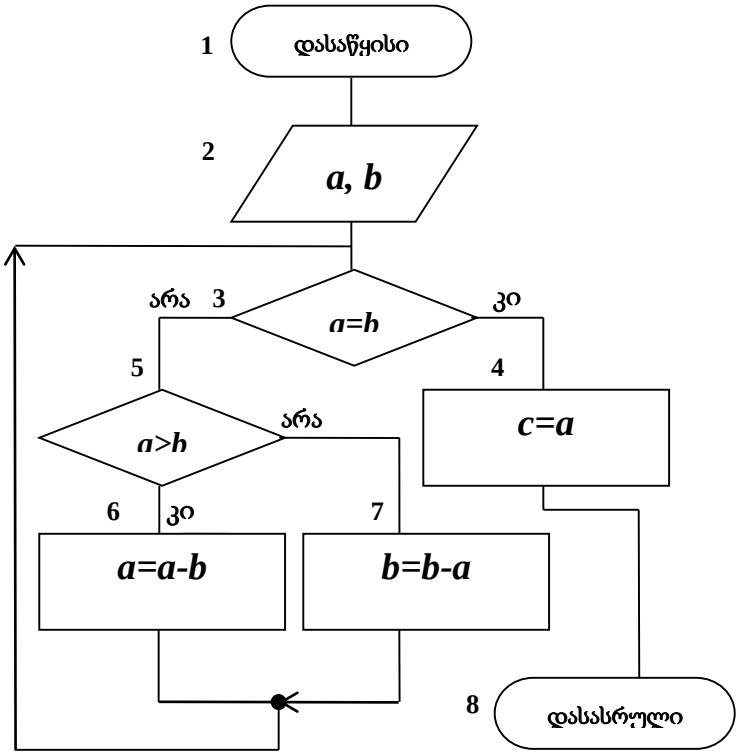
ძირითადი ალგორითმის ბლოკ-სქემას მე-19 ნახაზზე წარმოდგენილი სახე ექნება.



ნახ. 18 ძირითადი ალგორითმის ბლოკ-სქემა

დამხმარე ალგორითმის აღმნიშვნელი ბლოკი მიმართავს ორი რიცხვის უდიდესი საერთო გამყოფის პოვნის ალგორითმს. იგი გადასცემს მას *x* და *y* საწყის მონაცემებს. დამხმარე ალგორითმის შესრულების შედეგი კი მიენიჭება *z* ცვლადს. *x* და *y* სიდიდეებს **ფაქტიურ პარამეტრებს** უწოდებენ.

დამხმარე ალგორითმის ბლოკ-სქემას, რომელიც განსაზღვრავს ორი ნატურალური რიცხვის უდიდეს საერთო გამყოფს, აქვს მე-20 ნახაზზე წარმოდგენილი სახე.



ნახ. 19 დამხმარე ალგორითმის ბლოკ-სქემა

დამხმარე ალგორითმის შესასვლელ და გამოსასვლელ სიდიდეებს წარმოადგენენ *a*, *b* და *c* ცვლადები. მათ **ფორმალური პარამეტრები** ეწოდება. როგორც ვხედავთ, ფორმალური და ფიქტიური პარამეტების აღნიშვნები შეიძლება არ ემთხვეოდეს ერთმანეთს.

დამხმარე ალგორითმის შესრულება ძირითადი ალგორითმიდან მიმართვის შედეგად ხდება. ამ დროს ფაქტიური პარამეტრები *x* და *y* შეცვლიან დამხმარე ალგორითმში *a* და *b* ფორმალურ პარამეტრებს. მათი მნიშვნელობებისათვის გამოითვლება უდიდესი საერთო გამყოფის *c* მნიშვნელობა. იგი გადაიცემა ძირითად ალგორითმში და მიენიჭება *z* ცვლადს. ამის შემდეგ ალგორითმის შესრულება გაგრძელდება.

დამხმარე ალგორითმის გამოყენება ამოკლებს ძირითადი ალგორითმის ჩაწერას და უფრო თვალსაჩინოს ხდის მას. გარდა ამისა, იგი საშუალებას გვაძლევს ალგორითმის ერთნაირი ფრაგმენტები ერთხელ აღვწეროთ, საჭიროების შემთხვევაში კი, მივმართოთ მას.

1.3.6. ალგორითმის ფუნქციონირების შემოწმება

მას შემდეგ, რაც ალგორითმი შედგენილია, ამოცანის ამოხსნისათვის აღარ არის აუცილებელი ამოცანის შინაარსის ცოდნა, საჭიროა მხოლოდ ალგორითმით გათვალისწინებული ბრძანებების მექანიკური შესრულება.

ალგორითმის შესრულება მდგომარეობს ალგორითმით გათვალისწინებული მოქმედებების ზუსტ შესრულებაში. შესრულება, როგორც წესი, ხდება გარკვეული საკონტროლო მონაცემებისათვის, რაც საშუალებას გვაძლევს დავრწმუნდეთ ალგორითმის სისწორეში. ალგორითმის ფუნქციონირების შემოწმება მოსახერხებელია შემდეგი ცხრილის გამოყენებით:

ცხრილი 3 ალგორითმის ფუნქციონირების შემოწმების ცხრილი

ბიჯის №	ბრძანების მნიშვნელობა (შედეგი)	ცვლადები			
				. . .	

პირველ გრაფაში იწერება შესასრულებელი ბიჯის ნომერი. მეორე გრაფაში - მოცემული ბიჯის შესრულების შედეგი. მესამე გრაფაში - ალგორითმში გამოყენებული ცვლადების მნიშვნელობები ყოველი ბიჯის შემდეგ.

მაგალითისათვის განვიხილოთ $ax=b$ წრფივი განტოლების ამოხსნის ალგორითმის შესრულება. ამ ალგორითმს აქვს სახე:

0. დასაწყისი
1. შეტანა a, b;
2. თუ $a=0$, მაშინ გადავიდეთ მე-6 პუნქტზე;
3. $x= b / a$;
4. ამობეჭდვა x;
5. გადავიდეთ მე-10 პუნქტზე;
6. თუ $b=0$, მაშინ გადავიდეთ მე-9 პუნქტზე;
7. ამობეჭდვა "განტოლებას ამონახსნი არ აქვს";
8. გადავიდეთ მე-10 პუნქტზე;
9. ამობეჭდვა "განტოლებას აქვს ამონახსნთა უსასრულო სიმრავლე";
10. დასასრული.

შევასრულოთ ეს ალგორითმი, როცა $a=2$; $b=4$.

ცხრილი 4 ალგორითმის შესრულების შედეგი მნიშვნელობებისათვის a=*2;* b=*4.*

ბიჯის #	ბრძანების მნიშვნელობა (შედეგი)	ცვლადები		
		a	*b*	*x*
1	2, 4	2	4	-
2	„მცდარი“	2	4	-
3	2	2	4	2
4	2	2	4	2
5	10	2	4	2
10	დასასრული	2	4	2

ახლა შევასრულოთ ეს ალგორითმი, როცა a=0; b=4.

ცხრილი 5 ალგორითმის შესრულების შედეგი მნიშვნელობებისათვის a=*0;* b=*4.*

ბიჯის #	ბრძანების მნიშვნელობა (შედეგი)	ცვლადები		
		a	*b*	*x*
1	0, 4	0	4	
2	„ჭეშმარიტი“	0	4	
6	„მცდარი“	0	4	
7	„განტოლებას ამონახსნი არ აქვს“	0	4	
8	10	0	4	
10	დასასრული	0	4	

იგივე ალგორითმი შევასრულოთ, როცა a=0; b=0.

ცხრილი 6 ალგორითმის შესრულების შედეგი მნიშვნელობებისათვის a=*0;* b=*0.*

ბიჯის #	ბრძანების მნიშვნელობა (შედეგი)	ცვლადები		
		a	*b*	*x*
1	0, 0	0	0	-
2	„ჭეშმარიტი“	0	0	
6	„ჭეშმარიტი“	0	0	
9	„ამონახსნთა უსასრულო სიმრავლე“	0	0	
10	დასასრული	0	0	

დავალება:

1. შეადგინეთ $y = \frac{x^2 + c}{\sqrt{x + c}}$ გამოსახულების გამოთვლის ალგორითმი.
2. შეადგინეთ ქვემო მოცემული გამოსახულების გამოთვლის ალგორითმის ბლოკ-სქემა:

$$y = \begin{cases} x^2 , & x < 0 \\ 25 , & x = 0 \\ \ell nx , & x > 0 \end{cases}$$

3. შეადგინეთ ქვემო მოცემული გამოსახულების გამოთვლის ალგორითმის ბლოკ-სქემა:

$$z = \begin{cases} \sin x , & x \le a \\ \cos x , & a < x < b \\ tgx , & x \ge b \end{cases}$$

4. შეადგინეთ ალგორითმი, რომელიც ზრდადობის მიხედვით დაალაგებს a, b, c რიცხვით ცვლადებს ისე, რომ a ცვლადს უმცირესი რიცხვი შეესაბამებოდეს, b ცვლადს - საშუალო და c ცვლადს - უდიდესი.
5. შეადგინეთ $ax^2+bx+c=0$ კვადრატული განტოლების ფესვების გამოთვლის ალგორითმი a, b, c კოეფიციენტების ნებისმიერი მნიშვნელობების დროს.
6. შეადგინეთ $y=x^2+5$ ფუნქციის მნიშვნელობათა ცხრილის გამოთვლის ალგორითმი x არგუმენტის (-5)-დან 5-მდე 0,5 ბიჯით ცვლილების შემთხვევაში.
7. შეადგინეთ $y = \frac{cosx}{x}$ გამოსახულების გამოთვლის ალგორითმი (გაითვალისწინეთ, რომ ნოლზე გაყოფა დაუშვებელი ოპერაციაა).
8. შეადგინეთ ალგორითმი, რომელიც სამ a, b, c რიცხვით ცვლადს შორის განსაზღვრავს უმცირესი ნიშვნელობის ცვლადს.
9. შეადგინეთ ალგორითმი, რომელიც განსაზღვრავს უნაშთოდ იყოფა თუ არა ნებისმიერი სამნიშნა რიცხვი თავის ციფრთა ჯამზე.

1.3.7. ალგორითმის ანალიზი და სირთულე

ალგორითმების ანალიზის აუცილობლობას სამი მიზეზი განაპირობებს:

- ორი ალგორითმის შედარება;
- ალგორითმის ყოფაქცევის წინასწარი შეფასება;
- ალგორითმში შემავალი პარამეტრების მნიშვნელობების დადგენა.

სამივე მიზეზი გულისხმობს, რომ გვაქვს გარკვეული კრიტერიუმები, რომელთა საშუალებითაც მოვახდენთ აღნიშნულ შეფასებებს. ასეთ კრიტერიუმებად განიხილება ალგორითმის სირთულე (complexity). ძირითადად, ალგორითმის ორი ტიპის სირთულე განიხილება:

სირთულე დროის მიხედვით: ამოცანის ამოხსნისას ალგორთმის შემადგენელი ოპერაციები იყოფა ე.წ. ელემენტარულ ბიჯებად (შესასრულებელ მოქმედებებად). ბლოკ-სქემით ჩაწერილი ალგორითმისთვის ეს იქნება შესასრულებელი და პირობითი ბლოკები. შესაძლებელია მათ მივაწეროთ განსხვავებული ბიჯების რაოდენობა, ანუ, როგორც ამბობენ განსხვავებული წონები. აქაც ბუნებრივია, განსხვავებულ ოპერაციებს უნდა მივაწეროთ განსხვავებული ბიჯების რაოდენობა ე.ი. წონები. ანუ რამდენ ბიჯს "იწონის" თითოეული ოპერაცია. ანალიზისთვის მთავარია მხოლოდ, რომ ეს ბიჯები ერთნაირად ითვლებოდეს ყოველ სიტუაციაში. ჩვეულებრივ, შეფასებისას იგულისხმება, რომ კომპიუტერზე არითმეტიკული ოპერაციები ერთმანეთის მიმდევრობით სრულდება.

პირველი მიდგომით, ბუნებრივია, სწორედ ელემენტარული ბიჯების სრულ რაოდენობას ვუწოდოთ ალგორითმის სირთულე დროის მიხედვით. მაგრამ, რადგან ალგორითმი დამოკიდებულია შემავალ მონაცემებზე, ამიტომ ჩვენ ფაქტიურად გვაინტერესებს დამოკიდებულება შემავალ მონაცემებსა და ბიჯების სრულ რაოდენობას შორის. თუ, მაგალითად, გვაქვს *n* მონაცემი, მაშინ დაგვაინტერესებს როგორი იქნება ბიჯების სრული რაოდენობა *T(n)*, ე.ი. **ამოცანის დროითი სირთულე.** ზოგიერთი ავტორი შემავალი მონაცემების ზომას (რაოდენობრივ ზომას) უწოდებს **ამოცანის ზომას**, სიდიდეს. როდესაც შევისწავლით ამოცანის *n* ზომის ზრდის მიხედვით *T(n)* ამოცანის დროითი სირთულის ყოფაქცევას, მაშინ ვიტყვით, რომ ვიკვლევთ **ასიმპტოტურ დროით სირთულეს.**

შესაძლებელია გვაინტერესებდეს არა ალგორითმის ყველა ოპერაციის შესაბამისი ელემენტარული ბიჯების რაოდენობა, არამედ მხოლოდ რომელიღაც ამორჩეული ოპერაციების შესაბამისი ელემენტარული ბიჯების რაოდენობა. ზუსტად თუ ვიტყვით, გვაინტერესებს ამორჩეული ოპერაციების შესაბამისი ელემენტარული ბიჯების რაოდენობის დამოკიდებულება შემავალ მონაცემებზე. ამ დროს ვამბობთ, რომ ვსაუბრობთ **ალგორითმის ანალიზზე** ამ ამორჩეული ოპერაციების მიმართ.

სირთულე მეხსიერების მიხედვით: ამოცანის ამოხსნისას კომპიუტერში, ბუნებრივია, გარკვეული რაოდენობის სხვადასხვა რესურსების გამოყენება ხდება. თუ განვიხილავთ, ერთი რომელიმე ტიპის მეხსიერების დაკავებულ რაოდენობას, მაშინ წინა შემთხვევის ანალოგიურად შეიძლება შემოვიტანოთ მოცულობითი სირთულე ანუ სხვა ტერმინოლოგიით **სირთულე მეხსიერების მიხედვით** - ესაა დამოკიდებულება ამოცანის *n* ზომასა და *T(n)* მეხსიერების დაკავებულ რაოდენობას შორის. ანალოგიურად განიმარტება **ასიმპტოტური სირთულე მეხსიერების მიხედვით.**

ვთქვათ, ალგორითმის შესაფასებლად შევარჩიეთ კრიტერიუმი. გარკვეულობისთვის ვიგულისხმოთ, რომ ესაა დროითი სირთულე. ახლა, შემავალ მონაცემებზე დამოკიდებულებით ერთი და იგივე ალგორითმმა შეიძლება ამოცანის ამოხსნას განსხვავებული ბიჯების რაოდენობა მოახმაროს. ამიტომ, ალგორითმის შეფასებისას გათვალისწინებულ უნდა იქნეს სხვადასხვა შემავალ მონაცემებთან დაკავშირებული ბიჯების რაოდენობები. ასეთი მიდგომით, ალგორითმი ამომწურავად ხასიათდება სამი მახასიათებლით: **დროითი სირთულე საუკეთესო შემთხვევაში, დროითი სირთულე უარეს შემთხვევაში, დროითი სირთულე საშუალო შემთხვევაში.**

საუკეთესო შემთხვევა – ეს ნიშნავს შემავალი მონაცემების მიხედვით დროითი სირთულის მინიმალურ მნიშვნელობას ანუ ელემენტარული ბიჯების ყველაზე ნაკლებ რაოდენობას. ანალოგიურად, **უარესი შემთხვევა** იქნება ელემენტარული ბიჯების მაქსიმალური რაოდენობა შემავალი მონაცემების მიხედვით. ბევრი ავტორი მხოლოდ ამ უკანასკნელ მაჩვენებელს უწოდებს **ალგორითმის სირთულეს.**

ანალიზისთვის ძალიან მნიშვნელოვანია ე.წ. **საშუალო შემთხვევა** (ყველაზე მოსალოდნელი შემთხვევა). ანუ გვაინტერესებს ალგორითმს (ან ალგორითმის გარკვეულ ოპერაციას) რამდენი ელემენტარული ბიჯი სჭირდება საშუალოდ, შემავალი მონაცემების მიხედვით. მარტივ შემთხვევაში ამის გამოთვლა შემდეგნაირად ხდება: საჭიროა განვიხილოთ შემავალი მონაცემების ყველა შესაძლო ვარიანტი (ყველა დასაშვები ვარიანტის სიმრავლე). ჯერ ეს სიმრავლე უნდა დავყოთ ისეთ თანაუკვეთ ქვესიმრავლეებად, რომ თითოეული ქვესიმრავლის ელემენტებზე ალგორითმს ჰქონდეს **მუდმივი სირთულე.** დავუშვათ,სულ გამოგვივიდა *k* ცალი ქვესიმრავლე და *i* – ურ ქვესიმრავლეში, რომელიც შეიცავს m_i ელემენტს ($i = 1, \ldots, k$) სირთულეა x_i. თუ მთელ სიმრავლეში *m* ცალი ელემენტია, მაშინ *i* – ური ქვესიმრავლის წილი არის $\frac{m_i}{m} = p_i$.

საშუალოს ცნების თვალსაჩინოებისთვის გამოვიკვლიოთ შეკითხვა: კურსზე საშუალოდ რა ნიშანს ღებულობს სტუდენტი. *k* –ბალიან შეფასების სისტემაში საჭიროა დავითვალოთ *i* – ნიშნის ($i = 1, \ldots, k$) მქონე სტუდენტთა რაოდენობა m_i და თუ სულ კურსზე სტუდენტთა რაოდენობაა *m*, მაშინ საშუალო ნიშანი იქნება $\frac{1}{m}\sum_{i=1}^{k} i \cdot m_i$. საშუალო ნიშანს ყველაზე მოსალოდნელი ნიშანიც შეგვიძლია ვუწოდოთ.

ალგორითმების ანალიზი ფაქიზი და შრომეტევადი საქმეა. ზოგიერთი ალგორითმი იმდენად კარგადაა შესწავლილი, რომ ცნობილია ზუსტი მათემატიკური ფორმულები, რომლებიც შესრულების დროის გამოსათვლელად გამოიყენება რეალურ სიტუაციებში. სხვა ალგორითმების სამუშაო მახასიათებლები შესაძლოა არ იყოს ცნობილი, თუნდაც იმის გამო, რომ მათ შესწავლას მივყავართ ჯერჯერობით გადაუჭრელ მათემატიკურ ამოცანებთან. ნებისმიერ შემთხვევაში, ალგორითმის ანალიზი შესაძლებლობების ფარგლებში მაქსიმალური სიზუსტით უნდა ჩატარდეს.

დავახასიათოთ რამდენიმე ფაქტორი, რომლებიც არსებითად მოქმედებენ ანალიზის სიზუსტეზე, მაგრამ ამავე დროს, ვერ იმართებიან პროგრამისტის მიერ.

1. დაპროგრამების კონკრეტულ ენაზე დაწერილი პროგრამები ტრანსლატორის მიერ ითარგმნება კონკრეტული ტიპის კომპიუტერის მანქანურ კოდებში. ამიტომ იმის გარკვევა, თითოეული ოპერატორის შესრულებას რა დრო სჭირდება, საკმაოდ რთულია. განსაკუთრებით ისეთ გარემოში, სადაც პროგრამას დროის სხვადასხვა მომენტში სხვადასხვა სამუშაო მახასიათებლები შეიძლება ჰქონდეს.
2. ზოგიერთი პროგრამა ძალიან მგრძნობიარეა შემავალი მონაცემების მიმართ, ამიტომ მათი წარმადობა შესაძლოა ძლიერ მერყეობდეს, შემავალი მონაცემების მიხედვით.
3. ბევრი გავრცელებული პროგრამისთვის ჯერ არაა ცნობილი ჩვენთვის საინტერესო შეფასებები.
4. პროგრამების ზოგიერთი წყვილი შესაძლოა საერთოდაც არ იყოს ერთმანეთთან შესადარებელი იმ აზრით, რომ ერთი მათგანი შესაძლოა განსაკუთრებულად ეფექტური იყოს შემავალი მონაცემების ერთი ტიპის მიმართ, მაშინ როდესაც სხვა ვითარებაში მეორე იყოს უკიდურესად ეფექტური.

ამ ფაქტორების მიუხედავად, ხშირად საკმაოდ ზუსტად შეიძლება ითქვას, თუ რა დროს დაიკავებს მოცემული პროგრამა, ან გარკვეულ სიტუაციებში შესრულდება თუ არა იგი რომელიმე სხვა პროგრამაზე უკეთ. რაც მთავარია, ასეთი ინფორმაციის მიღება შესაძლებელია საკმაოდ შეზღუდული მათემატიკური არსენალის საფუძველზე. განვიხილოთ ზოგიერთი აუცილებელი ფაქტორი.

ალგორითმების ანალიზის დროს მთავარ პარამეტრს (primary parameter) წარმოადგენს ამოცანის ზომა. ამოცანის ზომა შესაძლოა იყოს ფაილის ზომა მონაცემთა დახარისხების ან ძებნის ამოცანებში, ან სიმბოლოების ზომა სტრიქონში. ძალიან ხშირად, ამოცანის ზომა დასამუშავებელი მონაცემების ზომის პირდაპირპროპორციულია. შესაძლოა, რომ ამოცანის ზომის შესაფასებლად ვექტორული სიდიდეების განხილვა უფრო ბუნებრივი აღმოჩნდეს, თუმცა ამ შემთხვევაშიც შესაძლებელი არის ერთ პარამეტრზე გადასვლა.

ალგორითმი შეგვიძლია წარმოვიდგინოთ როგორც ფუნქცია, რომელიც ამოცანას მის პასუხს შეუსაბამებს. ასეთი გააზრებით, მონაცემები წარმოადგენენ არგუმენტს ალგორითმისთვის. ზოგადად, ამოცანის ამოხსნის მიზნით, მონაცემთა დანიშნულებას მათზე გარკვეული მოქმედებების განხორციელება წარმოადგენს. მონაცემები და მათზე დაშვებული მოქმედებები ერთობლიობაში **მონაცემთა სტრუქტურებს** ქმნიან.

რადგან ამოცანის ამოხსნასთან მისვლა სხვადასხვა მოქმედებებით არის შესაძლებელი, ამიტომ ერთიდაიგივე ამოცანაში შესაძლებელია მონაცემთა სხვადასხვა სტრუქტურის გამოყენება. მაგალითად, მასივის (ერთიდაიმავე ტიპის ინდექსიანი ცვლადების ერთობლიობის) ელემენტების ზრდადობით დასახარისხებლად შეგვიძლია გამოვიყენოთ მასივი, გროვა, ბინარული ხე. ძალიან მნიშვნელოვანია, რომ სწორად იქნეს შერჩეული მონაცემთა ის სტრუქტურა, რომელიც ოპტიმალურია ჩვენი ამოცანისთვის. შერჩევის პროცესში ძალიან მნიშვნელოვანია ვიცოდეთ, თუ რა პარამეტრებით განსხვავდება მონაცემთა სტრუქტურები ერთმანეთისგან და რა შემთხვევებშია თითოეული მათგანის გამოყენება რეკომენდირებული.

მიზანშეწონილია თავიდანვე გავამახვილოთ ყურადღება ორ მნიშვნელოვან ასპექტზე.

მონაცემთა სტრუქტურების ერთ-ერთი განმასხვავებელი მათ ელემენტებზე წვდომის მეთოდია. ამ თვალსაზრისით, სულ ორი გზა არსებობს: მონაცემებს მივწვდეთ სტრუქტურაში მათი განლაგების (ანუ პოზიციის) საშუალებით, ან მონაცემებს მივწვდეთ მათი შინაარსიდან (მნიშვნელობიდან) გამომდინარე. იმისდა მიხედვით, თუ როგორაა წვდომა ორგანიზებული, ერთიდაიგივე დანიშნულების ფუნქციები სხვადასხვა ალგორითმებს (მაგალითად მონაცემთა ძებნის ალგორითმები) ეფუძნება.

არანაკლებ მნიშვნელოვანი ფაქტორი არის მონაცემთა სტრუქტურის (მაქსიმალური) ზომა. მარტივ შემთხვევაში შეგვიძლია გამოვიყენოთ მასივის საფუძველზე ორგანიზებული მონაცემთა სტრუქტურები, რომელთა მაქსიმალური ზომა ფიქსირებულია. უფრო რთულ შემთხვევებში აუცილებელი ხდება დინამიკური ზომის მქონე სტრუქტურების გამოყენება, რომლებიც ეფექტურად და სწრაფად ახდენენ ელემენტების დამატებას, წაშლას, ძებნას და ზოგიერთ სხვა ფუნქციას.

კითხვები თვითშეფასებისათვის

1. *განმარტეთ, რას გულისხმობს ალგორითმის დროის მიხედვით სირთულე.*
2. *განმარტეთ ალგორითმის დორითი სირთულის საუკეთესო და უარესი შემთხვევები.*
3. *დაახასიათეთ ფაქტორები, რომლებიც არსებითად მოქმედებენ ალგორითმების ანალიზის სიზუსტეზე.*
4. *განმარტეთ მონაცემთა სტრუქტურის არსი.*
5. *განმარტეთ, რას გულისხმობს ალგორითმის მუდმივი სირთულე.*

1.3.8. რეკურსია

რეკურსია არის მოვლენა, როდესაც ამოსახსნელი ამოცანა შესაძლოა გამარტივდეს და დაიყოს ცალკეულ ნაწილებად, მანამ სანამ იგი **საბაზო** (უმარტივეს) ამოცანამდე არ დაიყვანება. ასეთი ამოცანების გადასაწყვეტად, დაპროგრამებაში, ტრადიციულად რეკურსიული ფუნქციები გამოიყენება, რომელთაც ჩვენ მომდევნო თავებში განვიხილავთ. ზოგადად, შეგვიძლია აღვნიშნოთ, რომ **რეკურსიული** ეწოდება **ფუნქციას,** რომელსაც საკუთარი თავის გამოძახება შეუძლია.

რეკურსიული ალგორითმები სავსებით ბუნებრივია იმ შემთხვევაში, როდესაც რაიმე ამოცანის ამოხსნა შესაძლებელია ამავე ტიპის, ოღონდ უფრო მარტივი ამოცანის ამონახსნზე დაყრდნობით. ასეთი ამოცანები ძალიან ბევრია. მაგალითად: არაუარყოფითი მთელი რიცხვის ფაქტორიალის გამოთვლა, არანულოვანი რიცხვის ახარისხება, ფიბონაჩის რიცხვითი მიმდევრობის წევრთა მნიშვნელობების გამოთვლა და სხვა.

რეკურსიული ალგორითმის მიერ თავისი თავის გამოძახებას **რეკურსიული ბიჯი** ეწოდება. ცხადია, იმისათვის, რომ რეკურსიულმა ალგორითმმა ამოცანა ამოხსნას, რეკურსიული ბიჯების რაოდენობა სასრული უნდა იყოს, ხოლო ბოლო ბიჯი უნდა განიხილავდეს უკვე იმდენად მარტივ ამოცანას, რომლის ამოხსნაც ტრივიალურია და ცნობილი. დაპროგრამებაში, რეკურსია არის ფუნქციის უნარი რომ შესრულების პროცესში გამოიძახოს თავისი თავი. გასათვალისწინებელია, რომ ფუნქციის ცნება (ისევე როგორც ბევრი სხვა) განსხვავებული შინაარსის მატარებელია მათემატიკასა და დაპროგრამების ენებში. დაპროგრამებაში ფუნქცია გაცილებით მეტია ვიდრე ასახვა, თუმცა განსაზღვრისა და მნიშვნელობათა არეების ანალოგიით, ფორმალურად მიეთითება არგუმენტებისა და თვითონ ფუნქციის ტიპები. იმ შემთხვევაშიც კი, როდესაც მათემატიკური ფუნქციის დაპროგრამება ხდება, პროგრამული კოდი უფრო ინფორმატიულია მათემატიკურ აღწერასთან შედარებით და მისი გამოთვლის ალგორითმსაც მოიცავს.

რეკურსიული ფუნქციები უდიდესი მრავალფეროვნებით გამოირჩევა, მაგრამ მათ რეალიზაციას ბევრი საერთო აქვს. უპირველეს ყოვლისა, ერთიდაიგივე რეკურსიული ფუნქცია შეგვიძლია გამოვიყენოთ არაერთი ამოცანის, არამედ ამოცანათა მთელი კლასის ამოსახსნელად. ფაქტიურად, ამოცანათა ამ კლასიდან რეკურსიული ფუნქცია უშუალოდ (თავისი თავის ხელახალი გამოძახების გარეშე) ხსნის მხოლოდ უმარტივეს, ანუ საბაზო შემთხვევებს. მაგალითად, ფაქტორიალისთვის საბაზო შემთხვევებია 1! და 0!, ახარისხებაში $a^1=a$ ან $a^0=1$, ხოლო ფიბონაჩის რიცხვებისთვის $f(1)=0$, $f(2)=1$. უფრო რთულ შემთხვევებში, რეკურსიული ფუნქცია ხლეჩს ამოცანას რამდენიმე ნაწილად, რომლებიც საწყისი ამოცანის ანალოგიურებია, ოღონდ უფრო "ახლო მდგომები" საბაზო შემთხვევასთან, თითოეული მათგანისთვის იძახებს საკუთარ თავს (ესაა ე.წ. რეკურსიული ბიჯი) და მათი შედეგებისგან ახდენს საწყისი ამოცანის ამონახსნის

სინთეზირებას. რეკურსიული ფუნქცია მუშაობას ასრულებს მაშინ, როდესაც მის მიერ გახსნილი რეკურსიული ფუნქციები ასრულებენ მუშაობას. იმისათვის რომ რეკურსია ადრე თუ გვიან დასრულდეს, საჭიროა რომ რეკურსიული ბიჯების მიმდევრობა დადიოდეს საბაზო შემთხვევებამდე.

რეკურსიის არსის უკეთ გასაგებად შემდეგი ამოცანები განვიხილოთ.

მაგალითი 1: წარმოვადგინოთ არაუარყოფითი მთელი რიცხვის ფაქტორიალის გამოთვლის რეკურსიული ალგორითმი და შესაბამისი ფუნქცია.

ამოხსნა: ვთქვათ მოცემულია მთელი რიცხვი *n*. მისი ფაქტორიალი ჯერ ჩავწეროთ მათემატიკური ფუნქციის სახით: $f(n) = \begin{cases} 1, & n \leq 1 \\ n \cdot f(n-1), & n > 1 \end{cases}$

მათემატიკიდან ცნობილია, რომ მთელი, არაუარყოფითი *n* რიცხვის ფაქტორიალი შემდეგი ფორმულით გამოითვლება: $n! = n*(n-1)*(n-2)*...*1$, ამასთან, 1!=1 და 0!=1. მაშასადამე, თუ *n*=5, მაშინ 5!=5*4*3*2*1=120.

რეკურსიული გზით ამოცანის გადასაწყვეტად ვიყენებთ შემდეგ ფორმულას: $n! = n*(n-1)!$ ცხადია, 5!=5*4! შესაბამისად:

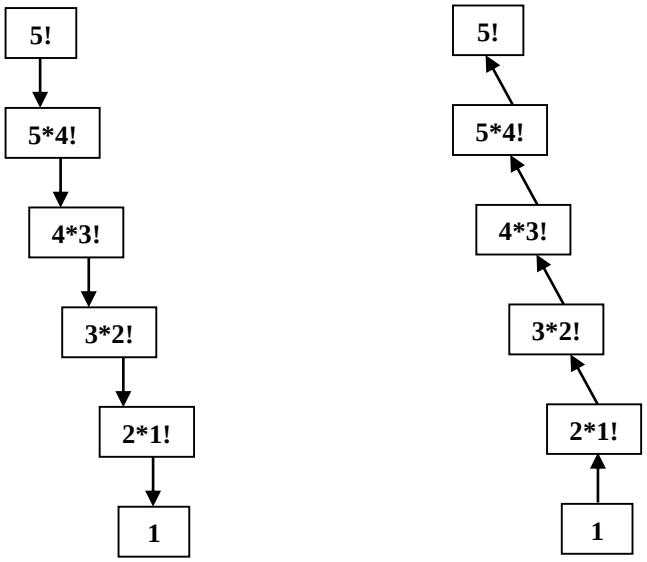
5!=5*4*3*2*1 ანუ 5!=5*(4*3*2*1)=5*4!

მაგალითად, 5-ის ფაქტორიალის მნიშვნელობის გამოთვლის პროცესი 21-ე ა) ნახაზზეა წარმოდგენილი, ხოლო რეკურსიული გამოძახების თანამიმდევრობა - 21-ე ბ) ნახაზზე.

ახლა კი მივუბრუნდეთ დასმულ ამოცანას და გადავწყვიტოთ იგი რეკურსიული factorial სახელის მქონე ფუნქციის გამოყენებით. აღნიშნული ფუნქცია თავდაპირველად ამოწმებს რეკურსიის დასრულების პირობას, ანუ, თუ *n* ცვლადის მნიშვნელობა ერთის ტოლი აღმოჩნდება, ეს ნიშნავს, რომ საწყისი ამოცანა დაყვანილია საბაზომდე და მაშასადამე, factorial ფუნქცია აბრუნებს ერთს; შესაბამისად, პროგრამა სრულდება. ხოლო, თუ *n*>1, მაშინ წარმოებს ფუნქციის რეკურსიული გამოძახება შემდეგი სახით:

***n**factorial(*n*-1);**

ბუნებრივია, factorial(*n*-1)-ის გამოთვლა უფრო მარტივი ამოცანაა, ვიდრე საწყისი factorial(*n*)-ის.



ა) ბ)

ნახ. 20

ა) - ფაქტორიალის მნიშვნელობის გამოთვლის პროცესი

ბ) - რეკურსიული გამოძახებების თანმიმდევრობა

არაუარყოფითი მთელი *n* რიცხვის ფაქტორიალის გამოთვლის ალგორითმის სიტყვიერი ფორმა შემდეგია:

1. დასაწყისი
2. *n* -ის შეტანა
3. თუ *n*<1, factorial (*n*)-ის მნიშვნელობად დააბრუნე 1 და გადადი მე-5 პუნქტზე
4. თუ *n*>1, factorial (*n*)-ის მნიშვნელობად დააბრუნე *n**factorial(*n*-1) და დაბრუნდი მე-3 პუნქტზე
5. შედეგის გამოტანა
6. დასასრული.

მაგალითი 2: მოცემულია დადებითი ნამდვილი რიცხვი *a* და მთელი რიცხვი *n*. მოვძებნოთ სწრაფი ალგორითმი, რომელიც *a* რიცხვს *n* ხარისხში აიყვანს.

შენიშვნა. თავისი სიმარტივის მიუხედავად, ამ ამოცანას მნიშვნელოვანი გამოყენება აქვს. იგივე ალგორითმებით სარგებლობა შეიძლება მატრიცების ასახარისხებლად ნატურალურ ხარისხში.

ამოხსნა. ყველაზე პირდაპირ გზას მოცემული რიცხვის თავის თავზე საჭირო რაოდენობით გადამრავლება წარმოადგენს, განმეორების შეტყობინების გამოყენებით. ცხადია, ამ ალგორითმის სისწრაფე ხარისხის პროპორციული იქნება.

განვიხილოთ მარტივი რეკურსიული ალგორითმი, რომელიც ხარისხის რეკურსიულ განმარტებაზეა დაფუძნებული: $a^n = a \cdot a^{n-1},\ a^1 = a.$ და მოვახდინოთ მისი გაუმჯობესება. გამოვიყენოთ აღნიშვნა: $f(a,n) = a^n$.

დასმული ამოცანის გადაწყვეტის ალგორითმის სიტყვიერ ფორმას შემდეგი სახე ექნება:

1. დასაწყისი
2. *a*-სა და *n* -ის შეტანა
3. თუ *n*>0, $f(a,n)$-ის მნიშვნელობად დააბრუნე $a \cdot f(a,n-1)$
4. თუ *n*=1, $f(a,n)$-ის მნიშვნელობად დააბრუნე *a* და გადადი მე-7 პუნქტზე
5. თუ *n*=0, $f(a,n)$-ის მნიშვნელობად დააბრუნე 1 და გადადი მე-7 პუნქტზე
6. თუ *n*<0, $f(a,n)$-ის მნიშვნელობად დააბრუნე $(f(a,-n))^{-1}$
7. შედეგის გამოტანა
8. დასასრული.

მაგალითი 3: რეკურსიული გზით გამოვთვალოთ ფიბონაჩის რიცხვითი მიმდევრობის ნებისმიერი *n* წევრის მნიშვნელობა.

ამოხსნა: მათემატიკიდან ცნობილია, რომ ფიბონაჩის რიცხვითი მიმდევრობა 0-ით და 1-ით იწყება და ხასიათდება იმ თვისებით, რომ მიმდევრობის ყოველი მომდევნო წევრის მნიშვნელობა წინა ორი წევრის მნიშვნელობათა ჯამის ტოლია. ბუნებაში ფიბონაჩის რიცხვითი მიმდევრობა სპირალის ფორმას აღწერს და მათემატიკურად აქვს შემდეგი სახე:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

რეკურსიული გზით ფიბონაჩის რიცხვითი მიმდევრობა ამავე სახელის მქონე ფუნქციის შემოტანით შემდეგნაირად შეიძლება განისაზღვროს:

Fibonacci(0)=0;

Fibonacci(1)=1;

Fibonacci(*n*)= Fibonacci(*n*-1)+ Fibonacci(*n*-2).

დასმული ამოცანის გადაწყვეტის ალგორითმის სიტყვიერ ფორმას შემდეგი სახე ექნება:

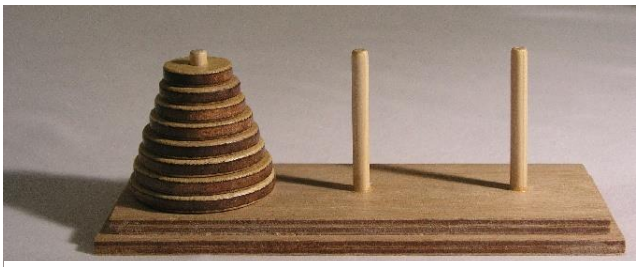
1. დასაწყისი
2. *n* -ის შეტანა
3. თუ *n*=1, Fibonacci(*n*)-ის მნიშვნელობად დააბრუნე 1 და გადადი მე-6 პუნქტზე
4. თუ *n*=0, Fibonacci(*n*)-ის მნიშვნელობად დააბრუნე 0 და გადადი მე-6 პუნქტზე
5. თუ *n*>1, Fibonacci(*n*)-ის მნიშვნელობად დააბრუნე Fibonacci(*n*-1)+ Fibonacci(*n*-2)
6. შედეგის გამოტანა
7. დასასრული.

მაგალითი 4: რეკურსიული გზით გადავწყვიტოთ **ჰანოის კოშკების** ამოცანა, რომლის არსი შემდეგში მდგომარეობს:

არსებობს ლეგენდა, რომ შორეული აღმოსავლეთის ერთ-ერთ მონასტერში ბერები ცდილობდნენ ფირფიტები ერთი ღერძიდან მეორეზე გადაეტანათ. ფირფიტები ღერძზე ჩამოცმული ისე იყო, რომ მათი ზომა ქვემოდან ზემოთ მცირდებოდა. ფირფიტების გადატანა შემდეგი წესების გათვალისწინებით ხდებოდა:

- ერთ გადატანაზე შესაძლებელი იყო მხოლოდ ერთი ფირფიტის გადატანა.
- დიდი ზომის ფირფიტა პატარა ფირფიტის ზემოდან არასდროს არ უნდა აღმოჩენილიყო
- შესაძლებელი იყო მხოლოდ ერთი დამხმარე ღერძის გამოყენება.

ბერები ცდილობდნენ ეს პროცესი შეესრულებინათ 64 ფირფიტისთვის. ლეგენდა ასევე ამბობს, რომ როდესაც ისინი ამ პროცესს დაასრულებდნენ (ამ ყველაფერს ისინი კომპიუტერის გარეშე აკეთებდნენ), ქვეყნიერების დასასრული დადგებაო. ☺



ნახ. 21 ამოცანის ვიზუალური წარმოდგენა

ამოხსნა: განვიხილოთ პირობა, რომ, როდესაც პირველი ღერძიდან მე-3 ღერძზე გადასატანია 3 ფირფიტა, მე-2 ღერძს ვიყენებთ ფირფიტების დროებით განსათავსებლად. ამოცანას თუ დავუკვირდებით შევამჩნევთ, რომ 3 ფირფიტის გადატანა შეიძლება გამარტივდეს 2 ფირფიტის გადატანის საშუალებით ანუ *n* ფირფიტის გამარტივება ხდება *n*-1 ფირფიტის გადანაცვლებით.

შემოვიტანოთ ცვლადები:

disc – ფირფიტების რაოდენობა

first – ღერძი, რომელზეც განთავსებულია ფირფიტები

last – ღერძი, რომელზეც გადაგვაქვს ფირფიტები

temp – დამხმარე ღერძი, ფირფიტების დროებით განსათავსებლად

ამოცანის გადაწყვეტის ალგორითმი თითოეული რეკურსიული ბიჯისთვის ასეთია:

1. გადავიტანოთ **disc – 1** ფირფიტა **first** – დან **temp**– ზე
2. გადავიტანოთ **1** დარჩენილი ფირფიტა **first** – დან **last** – ზე
3. გადავითანოთ **disc – 1** ფირფიტა **temp** – დან **last** – ზე

ეს პროცესი დასრულდება, მაშინ როდესაც გადასატანი იქნება მხოლოდ 1 ფირფიტა, ანუ ფუნქცია მიაღწევს საბაზო ამოცანამდე. ეს ალგორითმი შეიძლება გამოვიყენოთ 64 ფირფიტის ამოცანის გადაწყვეტისთვისაც.

ახლა კი ამ ალგორითმის წარმოსადგენად მივმართოთ HanoiTower ფუნქციას, რომელსაც შემდეგი პარამეტრები ექნება: HanoiTower(disc, first, last, temp)

1. დასაწყისი
2. საწყისი მონაცემების disc, first, last, temp შეტანა
3. თუ disc=1, დაბეჭდე first -> last (იბეჭდება, საიდან უნდა გადავიტანოთ ფირფიტა) და გადადი მე-7 პუნქტზე
4. თუ disc>1, HanoiTower(disc – 1, first, temp, last); (ფუნქცია თავის თავს ალგორითმის წესების შესაბამისად იძახებს, აბრუნებს);
5. HanoiTower(1 ,first, last, temp);
6. HanoiTower(disc – 1, temp, last, first);
7. შედეგების ბეჭდვა
8. დასასრული.

თუ ფუნქციას გამოვიძახებთ პარამეტრებით HanoiTower(3,1,3,2) შემდეგი შედეგი გვექნება:

1–>3
1–>2
3–>2
1–>3
2–>1
2–>3
1–>3

რეკურსიას რამდენიმე სერიოზული ნაკლი აქვს, რის გამოც მისი გამოყენება შეზღუდულია. ცნობილია, რომ ყველაფერი, რისი განხორციელებაც რეკურსიების საშუალებითაა შესაძლებელი, არარეკურსიულადაც სრულდება. რეკურსიული ფუნქცია მრავალჯერ იძახებს თავის თავს, რაც ზრდის პროგრამის შესრულების დროს. ამას გარდა, ყოველი რეკურსიული გამოძახება ფუნქციის ახალ ასლს (ფაქტიურად, მხოლოდ მისი ცვლადების ასლებს) ქმნის, რადგან ადრე გახსნილი ფუნქციები იხურება უფრო გვიან ვიდრე შემდეგ გახსნილები, მეხსიერების ხარჯი იზრდება გამოძახებული (გახსნილი) ფუნქციების რაოდენობის პროპორციულად. ის მონაცემები, რომლე-

ბიც რეკურსიული ფუნქციების გახსნისას იჩენს თავს, ქმნიან ძალიან გავრცელებულ და საინტერესო მონაცემთა სტრუქტურას, რომელსაც **სტეკი** ეწოდება. ორი სიტყვით სტეკი ხასიათდება პრინციპით: "მოვიდა პირველი, წავიდა ბოლო" ანუ FILO. სტეკში ინფორმაციის დაგროვების პრინციპი იგივეა, რაც მჭიდეში ტყვიების ჩაწყობა- ამოღების, ან ჩიხში მანქანების შეგროვების და იქიდან მათი გამოყვანის.

რეკურსიის ძირითადი ღირსება იმაში მდგომარეობს, რომ მისი საშუალებით ადვილად აღქმადი და შედარებით კომპაქტური ხდება პროგრამული კოდის შემუშავება. თუმცა რეკურსიის საშუალებით თითქმის ყველა იმ ამოცანის ამოხსნა შეიძლება, რომელიც არარეკურსიული ალგორითმებით იხსნება, მაგრამ, მარტივ შემთხვევებში ამის გაკეთება არ ღირს.

რეკურსიების გამოყენება საკმაოდ რთული და ფაქიზი საქმეა, საჭიროა გარკვეული პრაქტიკა, რომ რეკურსია ბუნებრივად მოგვეჩვენოს.

კითხვები თვითშეფასებისათვის

1. *განმარტეთ რეკურსიის მოვლენა.*
2. *შეადგინეთ ალგორითმი (სიტყვიერი ფორმით), რომელიც განსაზღვრავს ფიბონაჩის რიცხვითი მიმდევრობის მე-20 წევრის მნიშვნელობას.*
3. *შეადგინეთ ალგორითმი (სიტყვიერი ფორმით), რომელიც გამოთვლის 10!-ის მნიშვნელობას.*
4. *შეადგინეთ ალგორითმი, რომელიც რეკურსიული გზით გამოთვლის ნებისმიერი ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფის მნიშვნელობას.*
5. *დაახასიათეთ რეკურსიული ალგორითმების უპირატესობები და ნაკლი.*

1.4. მონაცემთა დახარისხების და ძებნის ალგორითმები

სანამ მონაცემთა დახარისხებისა და ძებნის ალგორითმებს განვიხილავდეთ, სასურველია გავეცნოთ მონაცემთა ისეთ ერთობლიობას, როგორსაც მასივები წარმოადგენს.

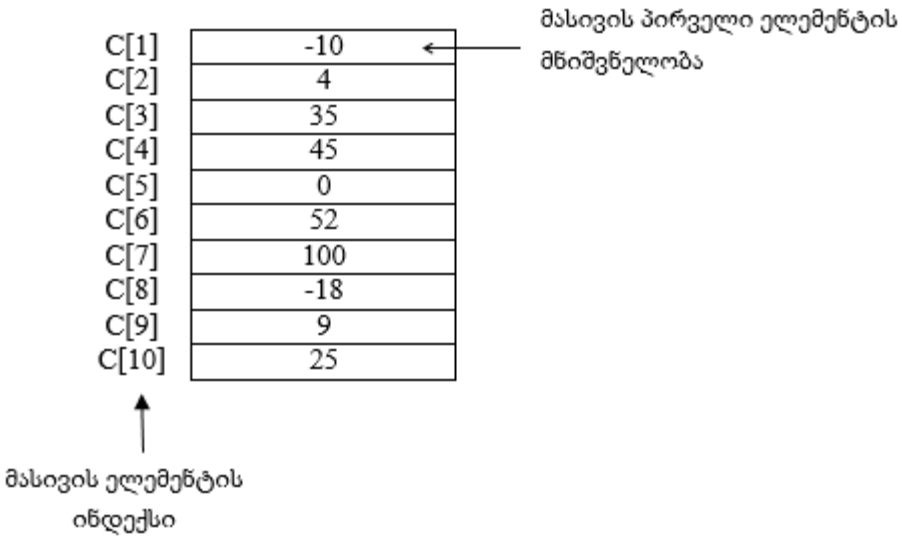
მასივი კომპიუტერის მეხსიერებაში არსებული მიმდევრობითი უჯრედების ჯგუფს წარმოადგენს, რომელთაც საერთო სახელი და ერთიდაიგივე ტიპი გააჩნიათ. ყოველი მასივი ხასიათდება ზომით და განზომილებით.

მასივში შემავალი ელემენტების რაოდენობა განსაზღვრავს მის ზომას. თითოეულ ელემენტს შეიძლება ერთი ან მეტი ინდექსი გააჩნდეს, რაც მასივის განზომილებას განსაზღვრავს.

თუ მასივში შემავალ ყოველ ელემენტს თითო ინდექსი გააჩნია, მას **ერთგანზომილებიანი მასივი** ანუ **ვექტორი** ეწოდება, ხოლო თუ მასივში ყოველი ელემენტი ორინდექსიანია, მას **ორგანზომილებიანი მასივი** ანუ **მატრიცა** ეწოდება.

23-ე ნახაზზე წარმოდგენილია მთელრიცხვა მასივი იდენტიფიკატორით "C", რომელიც ათი ელემენტისაგან შედგება. ნებისმიერ ელემენტზე მიმართვა მასივის სახელისა და ელემენტის ინდექსის მითითებითაა შესაძლებელი. ეს უკანასკნელი, ჩვენ შემთხვევაში, კვადრატულ ფრჩხილებშია მოთავსებული და ელემენტის პოზიციის ნომერს წარმოადგენს მასივში. მასივის ელემენტის ინდექსი მთელი ტიპის რიცხვია, ან გამოსახულება, რომლის შედეგი ასევე მთელი ტიპისაა.

მოვახდინოთ 23-ე ნახაზზე წარმოდგენილი ერთგანზომილებიანი მასივის ანალიზი.



ნახ. 22 ერთგანზომილებიანი მასივის ანალიზი

მასივის სახელი ანუ იდენტიფიკატორია "C". მისი შემადგენელი ათი ელემენტი აღნიშნულია, როგორც C [1], C [2], C [3],…, C [10]; სადაც C [1] ელემენტის მნიშვნელობა უდრის -10-ს, C [2] ელემენტის მნივნელობა უდრის 4-ს და ა.შ. ბოლო C [10] ელემენტის მნიშვნელობა – 25-ს.

ამგვარად, მასივი არის სტრუქტურა, რომელიც მონაცემთა სახით მოიცავს ერთმანეთთან დაკავშირებულ ერთიდაიმავე ტიპის მქონე ელემენტებს.

1.4.1. მარტივი გადანაცვლების ალგორითმი

პრაქტიკაში ძალიან ხშირად აქვს ადგილი მასივების მოწესრიგების ამოცანებს ანუ მასივების დახარისხების ამოცანებს, რომლებიც თავისი არსით კომბინატორული ამოცანების კლასს მიეკუთვნება.

სპეციალისტები თვლიან, რომ მანქანური დროის 25% სისტემატიურად დახარისხების ამოცანებზე იხარჯება. ამიტომ აღნიშნული ალგორითმები განსაკუთრებულ ყურადღებას იმსახურებს. შემუშავებულია დახარისხების უამრავი ალგორითმი, რომელთაგან ჩვენ მხოლოდ ზოგიერთს განვიხილავთ.

ძირითადად განასხვავებენ დახარისხების სამი კატეგორიის ალგორითმებს:

I - დახარისხება გადანაცვლებით.

II - დახარისხება ამორჩევით.

III - დახარისხება ჩასმით.

დახარისხების ყველა სხვა ალგორითმი აღნიშნული კატეგორიებიდან არის ნაწარმოები. ყველა ალგორითმი ერთგანზომილებიანი მასივების დასამუშავებლად არის გათვალისწინებული. აუცილებლობის შემთხვევაში, ყოველთვის შესაძლებელია ორგანზომილებიანი მასივების ერთგანზომილებიან მასივებზე დაყვანა ინდექსების გადათვლის გზით. ზოგადობისათვის დავუშვათ, რომ ადგილი აქვს მასივების დახარისხებას ელემენტების ზრდადი მნიშვნელობების მიხედვით.

მარტივი გადანაცვლების ალგორითმი I კატეგორიის ალგრითმებს მიეკუთვნება.

განვიხილოთ $A=(a_1, a_2,\ldots, a_n)$ ვექტორი, რომლის ელემენტები საჭიროა, ზრდადობის მიხედვით მოვაწესრიგოთ ანუ დავახარისხოთ. მარტივი გადანაცვლების ალგორითმის თანახმად შედარდება ვექტორის მეზობელი ელემენტების წყვილი a_i და a_{i+1}.

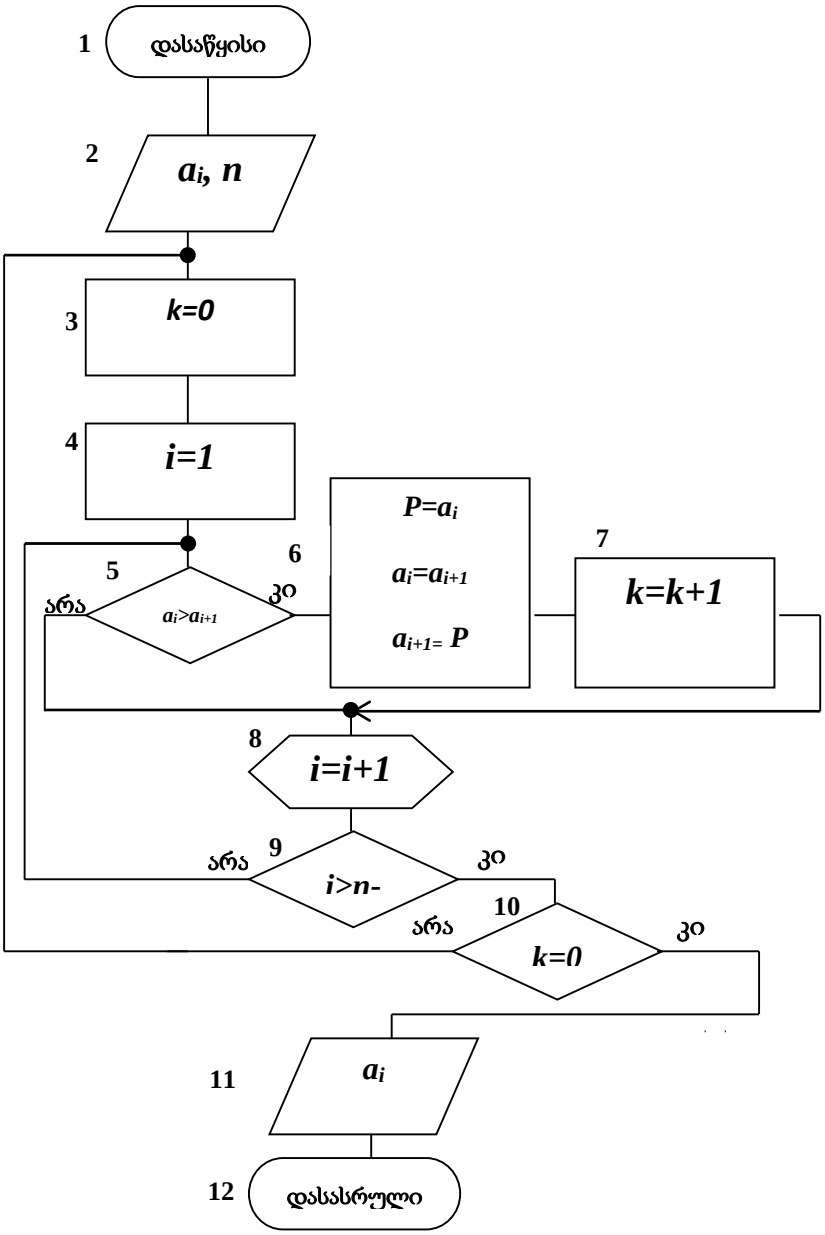
თუ $a_i < a_{i+1}$, გადავდივართ ელემენტების შემდეგი წყვილის შედარებაზე და ა.შ. ვექტორის ბოლო ელემენტამდე. იმ შემთხვევაში, თუ $a_i > a_{i+1}$, ადგილი აქვს ელემენტების გადაადგილებას,

რისთვისაც დამატებითი ცვლადი შემოგვაქვს. ამის შემდეგ გადავდივართ მომდევნო ელემენტების შედარებაზე. ალგორითმი ითვალისწინებს ერთგვარი k მთვლელის გამოყენებას, რომელიც აფიქსირებს ელემენტების გადანაცვლებათა რიცხვს მასივის ელემენტების ერთჯერადი დათვალიერების დროს.

ყოველი დათვალიერების შემდეგ საჭიროა მთვლელში არსებული ინფორმაციის გაანალიზება. თუ $k \neq 0$, ეს ნიშნავს, რომ ვექტორის ელემენტების გადაადგილებას დათვალიერების დროს ჰქონდა ადგილი და ე.ი. მასივი ჯერ მთლიანად დახარისხებული არ არის. მაშასადამე, აუცილებელია გარდაქმნილი მასივის ხელახალი დათვალიერება. ეს პროცესი გრძელდება მანამ, სანამ $k \neq 0$, ხოლო როდესაც k გაუტოლდება ნულს ($k=0$), იმის მიმანიშნებელია, რომ დათვალიერების დროს ელემენტების გადანაცვლებას ადგილი არ ჰქონია და მასივი მთლიანად დახარისხებულია.

ალგორითმის ბლოკ-სქემა 24-ე ნახაზზეა წარმოდგენილი.

გავაანალიზოთ ალგორითმის მუშაობის ლოგიკა. საჭიროა აღვნიშნოთ, რომ მასივის ელემენტების ყოველი ახალი დათვალიერების დროს საჭიროა k მთვლელის განულება (გასუფთავება). ℓ ცვლადი განვიხილოთ, როგორც დათვალიერების ნომერი. აღნიშნული პარამეტრის მნიშვნელობა დამოკიდებულია ვექტორის დახარისხების (მოწესრიგების) ხარისხზე: მოწესრიგების ხარისხი რაც უფრო ნაკლებია, მით უფრო მეტი იქნება დათვალიერების რიცხვი და პირიქით; i – ციკლის პარამეტრია, რომელიც უზრუნველყოფს თითოეული დათვალიერების დროს მეზობელი ელემენტების წყვილ-წყვილად შედარებას. წყვილ-წყვილი შედარების რიცხვი შეიძლება გამოისახოს ფორმულით: $q=(n-1)\,\ell$.



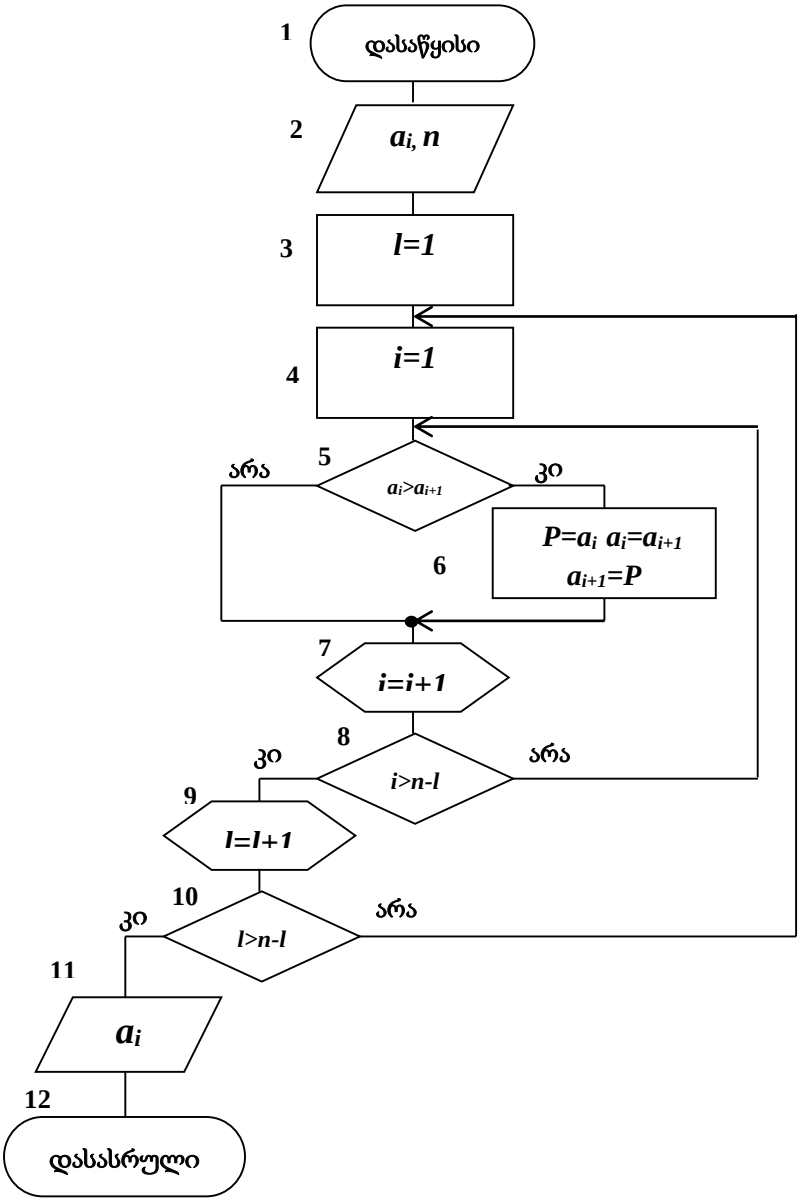
ნახ. 23 მარტივი გადანაცვლების ალგორითმის ბლოკ-სქემა

1.4.2. „ჩაძირვის“ (ბუშტისებრი დახარისხების) ალგორითმი

განვიხილოთ A=(a_1, a_2,…, a_n) ვექტორი და მოვახდინოთ მისი ელემენტების დალაგება ზრდადობის მიხედვით ბუშტისებრი დახარისხების მეთოდის გამოყენებით. დახარისხების აღნიშნულ მეთოდს "ჩაძირვის" მეთოდსაც უწოდებენ, რადგან მასივის უმცირესი მნიშვნელობა, მსგავსად წყალში ჰაერის ბუშტისა, სულ ზევით (მასივის დასაწყისისკენ) მიიწევს, ხოლო უდიდესი მნიშვნელობა სულ უფრო უახლოვდება წყლის ფსკერს (მასივის ბოლოს), ანუ ადგილი აქვს მის ერთგვარ "ჩაძირვას".

აღნიშნული მეთოდი მასივის რამდენიმე დათვალიერებას საჭიროებს. ყოველ ეტაპზე, მსგავსად მარტივი გადანაცვლების მეთოდისა, ადგილი აქვს მასივის წყვილი მეზობელი a_i და a_{i+1} ელემენტების შედარებას და აქაც ისინი გადაადგილდებიან მხოლოდ იმ შემთხვევაში, როდესაც სრულდება პირობა: $a_i > a_{i+1}$; ხოლო თუ $a_i \le a_{i+1}$, მოწმდება მასივის მომდევნო მეზობელი წყვილი და ა.შ. ბოლო ელემენტამდე.

განსხვავებით მარტივი გადანაცვლების მეთოდისა, აქ არ გამოიყენება k მთვლელი და პირველივე ეტაპის დასასრულს მასივის უდიდესი მნიშვნელობის მქონე ელემენტი გარანტირებულად იკავებს მასივის ბოლოს თავის ადგილს. მომდევნო ეტაპზე შემდეგი უდიდესი მნიშვნელობის მქონე ელემენტი თავსდება საჭირო ადგილას მასივში და პროცესი გრძელდება მანამ, სანამ არ მოხდება მასივის ელემენტების სრული დახარისხება ზრდადობის მიხედვით.



ნახ. 24 „ჩაძირვის“ (ბუშტისებრი დახარისხების) ალგორითმის ბლოკ-სქემა

ამგვარად, თუ მასივი n რაოდენობის ელემენტისაგან შედგება, მისი დახარისხებისთვის საჭიროა n-1 რაოდენობის ეტაპის გამოყენება და თვითოეულ ეტაპზე n-1 რაოდენობის შედარების ოპერაციის განხორციელება. აღნიშნული მეთოდი საკმაოდ მარტივია დაპროგრამების თვალსაზრისით, მაგრამ თავად დახარისხების პროცესი მიმდინარეობს ხანგრძლივი დროის მანძილზე, რაც დიდი ზომის მასივების შემთხვევაში აშკარაა და აღნიშნული მეთოდის ნაკლს წარმოადგენს.

ალგორითმის შესაბამისი ბლოკ-სქემა 25-ე ნახაზზეა წარმოდგენილი, სადაც ℓ ცვლადი ეტაპების მთვლელია.

1.4.3. კომბინირებული დახარისხების ალგორითმი

დახარისხების კომბინირებული მეთოდი მოიცავს მარტივი გადანაცვლებისა და ბუშტისებრი დახარისხების ალგორითმების უპირატესობებს.

განვიხილოთ A=(a_1, a_2,…, a_n) ვექტორი და აღნიშნული ალგორითმის გამოყენებით მოვახდინოთ მისი ელემენტების დალაგება ზრდადობის მიხედვით.

მსგავსად „ჩაძირვის“ ალგორითმისა, აქაც ადგილი აქვს მასივის მეზობელი ელემენტების წყვილ-წყვილად შედარებას და საჭიროების შემთხვევაში მათ გადაადგილებას.

აქაც, ისევე, როგორც მარტივი გადანაცვლების მეთოდში, გამოიყენება ერთგვარი k მთვლელი; მაგრამ დახარისხების კომბინირებული მეთოდის უპირატესობა მდგომარეობს შემდეგში:

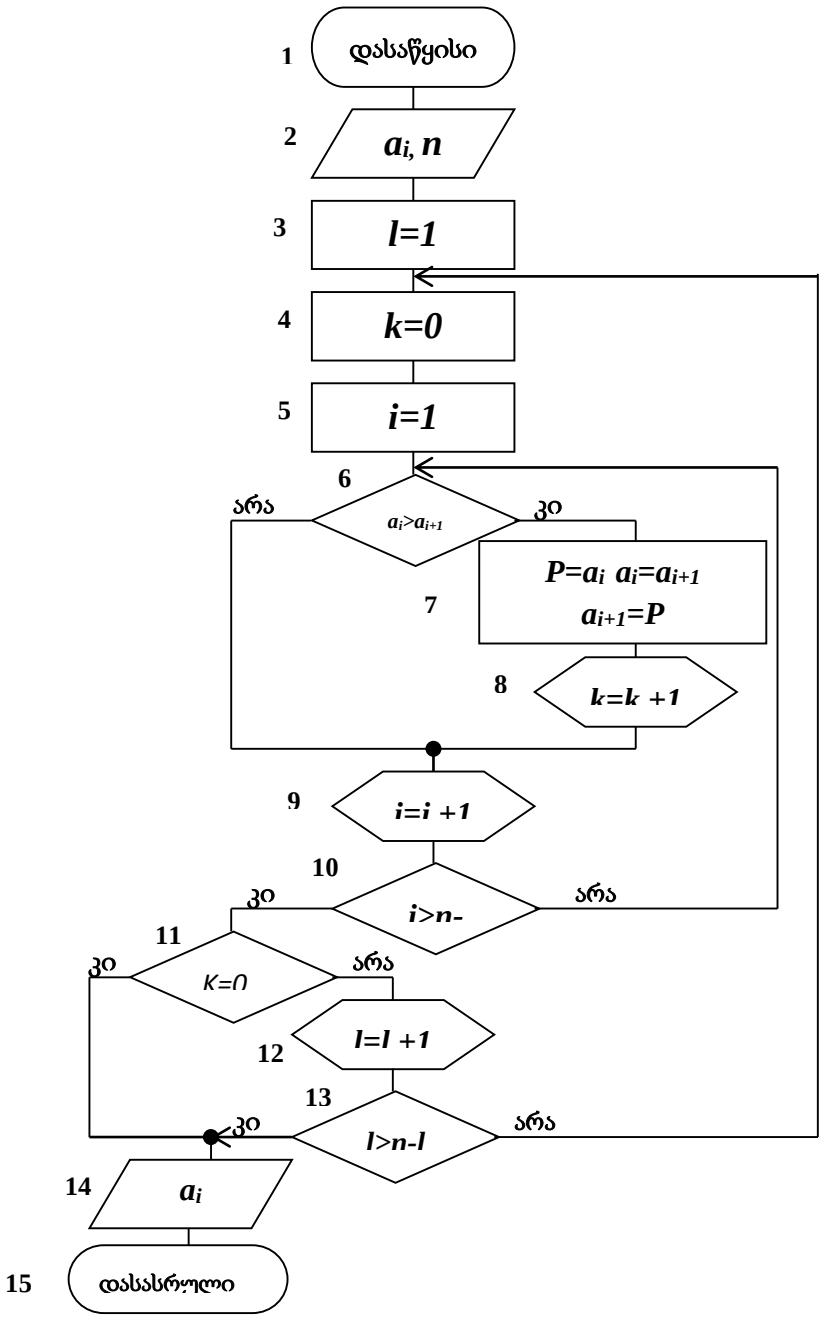
აქ განიხილება დახარისხების პროცესის დასრულების ორი შესაძლო ვარიანტი, რაც მნიშვნელოვნად აჩქარებს ელემენტების დახარისხების პროცესს:

- ვინაიდან ყოველ ეტაპზე მასივის უდიდესი მნიშვნელობის ელემენტები გა-რანტირებულად იკავებენ თავიანთ ადგილებს მასივის ბოლოში, ამიტომ ყოველ მომდევნო ეტაპზე ეს ელემენტები აღარ განიხილება, რაც მნიშვნელოვნად ამცირებს შესადარებელი ელემენტების რაოდენობას და შესაბამისად, აჩქარებს დახარისხების პროცესს.
- დახარისხების დასრულება შესაძლებელია შედარების ყველა ოპერაციის შესრულებამდე. ეს არის ციკლებიდან ალტერნატიული გამოსვლის შესაძლებლობა შემთხვევისათვის, როდესაც k მთვლელის მნიშვნელობა უტოლდება ნულს (k=0).

ალგორითმის შესაბამისი ბლოკ-სქემა 26-ე ნახაზზეა წარმოდგენილი.

აღსანიშნავია, რომ მასივთა დახარისხების ზემოთ განხილულ ყველა ალგორითმში, თუ მეზობელი ელემენტების წყვილების შედარების პირობას საწინააღმდეგოთი შევცვლით (a_i<a_{i+1}) და ამ პირობის ჭეშმარიტების შემთხვევაში იმავე წესით გადავაადგილებთ ელემენტებს, მაშინ

შესაძლებელი გახდება მასივთა ელემენტების მათი კლებადი მნიშვნელობების მიხედვით დახარისხება.



ნახ. 25 კომბინირებული დახარისხების ალგორითმის ბლოკ-სქემა

1.4.4. მატრიცის დახარისხების ალგორითმი

მატრიცის ელემენტების დახარისხებისათვის საჭიროა მატრიცის ელემენტების ერთგანზომილებიანი მასივის სახით წარმოდგენა.

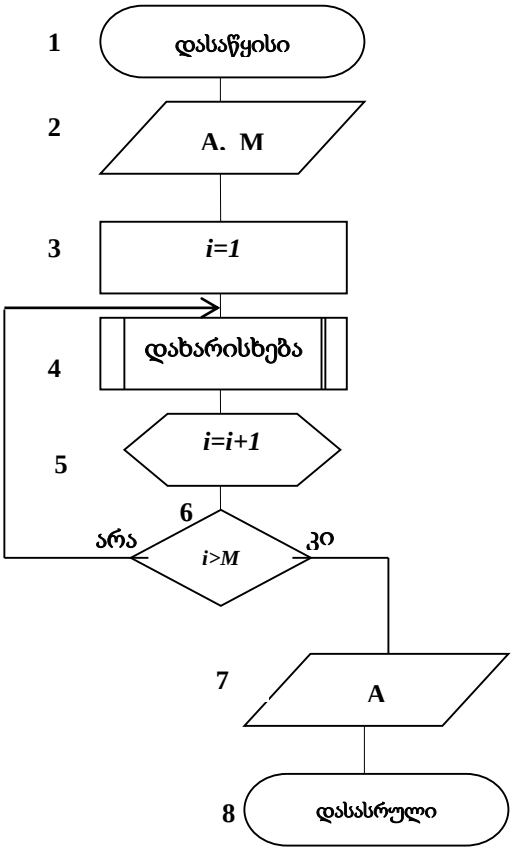
ამგვარად, მატრიცის ყველა ელემენტის დახარისხების ამოცანის გადასაწყვეტად მიზანშეწონილია მოცემული მატრიცის როგორც *mxn* განზომილების ვექტორის სახით წარმოდგენა.

იმ შემთხვევაში, როცა საჭიროა მატრიცის თითოეული სტრიქონის ან სვეტის დახარისხება, მაშინ ამოცანა დაიყვანება ვექტორის *m* ან *n* ელემენტების დახარისხების ამოცანაზე, რადგან მატრიცის თითოეული სტრიქონი ან სვეტი საჭიროა განვიხილოთ როგორც ვექტორი.

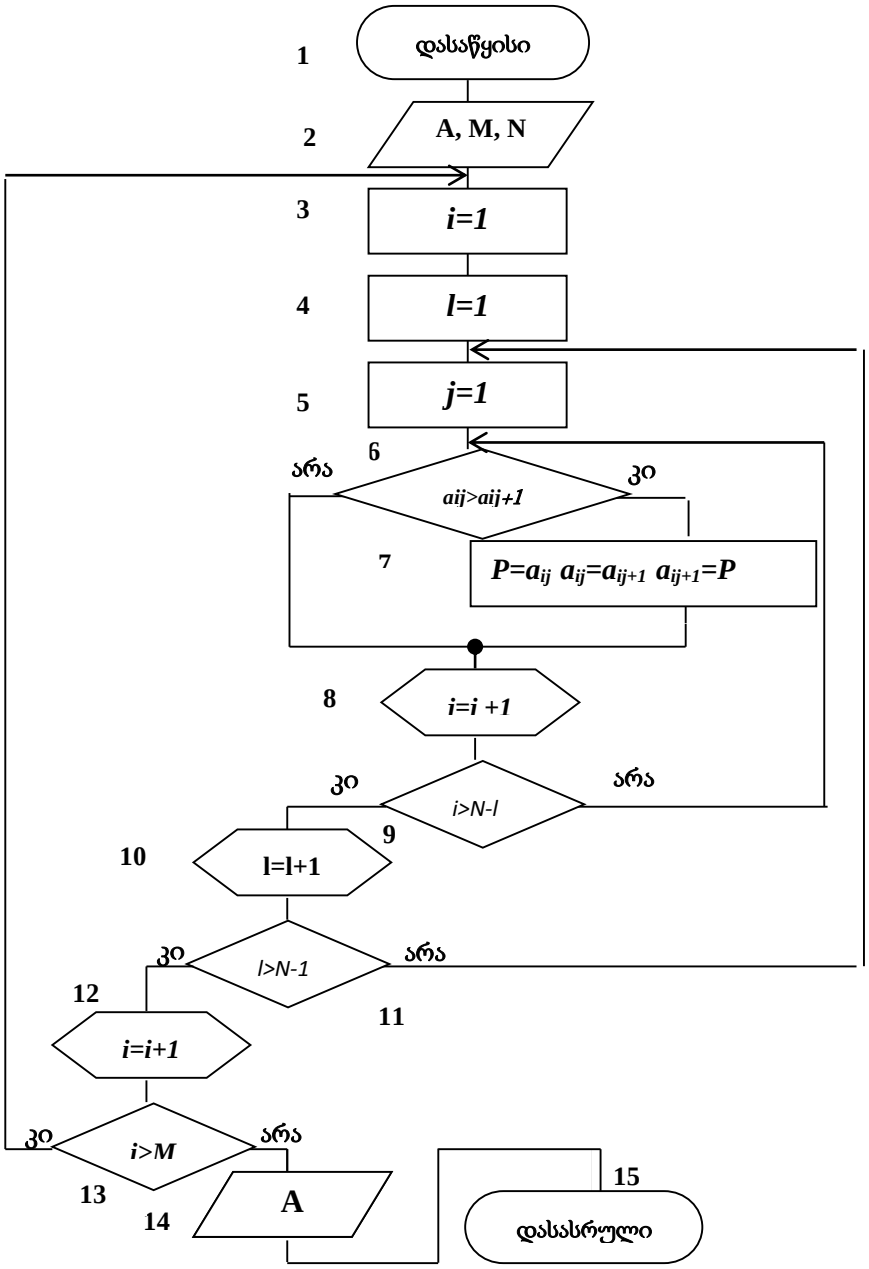
A მატრიცის სტრიქონების ელემენტების დახარისხების ამოცანის გადაწყვეტის ალგორითმის ზოგადი ბლოკ-სქემა 27-ე ნახაზზეა წარმოდგენილი.

ალგორითმის მე-3 ბლოკში შეიძლება გამოყენებული იქნეს ზემოთ განხილული ნებისმიერი ალგორითმი.

მატრიცის ელემენტების დახარისხების ალგორითმის ბლოკ-სქემა, რომელშიც გამოყენებულია ჩაძირვის ალგორითმის პროცედურა 28-ე ნახაზზეა წარმოდგენილი.



ნახ. 26 მატრიცის დახარისხების ალგორითმის ბლოკ-სქემა



ნახ. 27 მატრიცის ელემენტების დახარისხების ალგორითმის ბლოკ-სქემა, რომელშიც გამოყენებულია ჩაძირვის ალგორითმის პროცედურა

1.4.5. წრფივი ძებნის ალგორითმი

ხშირ შემთხვევაში პროგრამისტებს მუშაობა უწევთ მონაცემთა დიდ მოცულობასთან, რომელიც მასივების სახითაა წარმოდგენილი. ზოგჯერ საჭიროა განისაზღვროს, შეიცავს თუ არა მასივი ამათუიმ მნიშვნელობის მქონე ელემენტს. აღნიშნულ პროცესს ეწოდება ძიება (ძებნა) მასივში. არსებობს საჭირო მნიშვნელობის მქონე ელემენტის ძებნის ორი ალგორითმი:

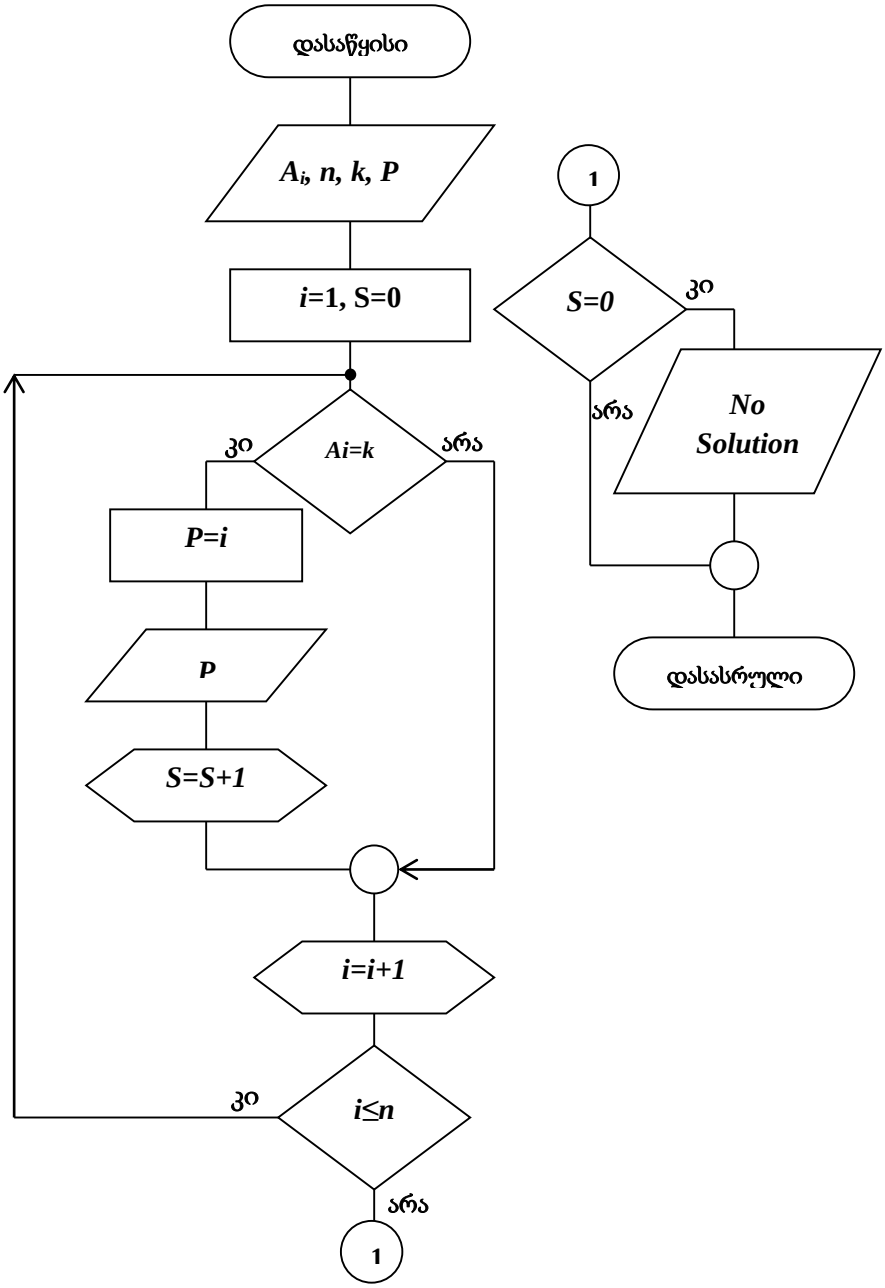
1. მარტივი წრფივი ძებნის ალგორითმი;
2. ორმაგი (ბინარული) ძებნის ანუ დიხოტომიის ალგორითმი;

წრფივი ძებნის ალგორითმის არსი შემდეგში მდგომარეობს: წინასწარ მოიცემა განსაზღვრული მნიშვნელობის მქონე ცვლადი და ის თანმიმდევრობით შედარდება მასივის ყველა ელემენტის მნიშვნელობას. თუ მასივის რომელიმე ელემენტის მნიშვნელობა დაემთხვევა მოცემული ცვლადის მნიშვნელობას, პროგრამა გამოიტანს შესაბამისი ელემენტის ინდექსს; წინააღმდეგ შემთხვევაში პროგრამა დასრულდება შეტყობინებით: "No Solution", რაც გულისხმობს იმ ფაქტს, რომ ამონახსნი არ გვაქვს, ვინაიდან საჭირო მნიშვნელობის ელემენტი მასივში არ არსებობს.

ამგვარად, ძებნის პროცესი აღნიშნული მეთოდის გათვალისწინებით გრძელდება მანამდე, სანამ არ შემოწმდება მასივის ყველა ელემენტი და არ მოიძებნება მათ შორის ისეთი, რომლის მნიშვნელობაც ტოლი იქნება მოცემული ცვლადის მნიშვნელობის (ცხადია, თუ ასეთი ელემენტი მასივში საერთოდ არსებობს).

აღნიშნული ალგორითმი მარტივია დაპროგრამების თვალსაზრისით, მაგრამ დიდი ზომის მასივების შემთხვევაში ხასიათდება არაეფექტურობით.

დავუშვათ, მოცემულია $A[n]$ მასივი და k ცვლადი. საჭიროა წრფივი ძებნის მეთოდით მოვძებნოთ მასივის ის ელემენტი, რომლის მნიშვნელობაც ემთხვევა k ცვლადის მნიშვნელობას. განხილული ალგორითმის შესაბამის ბლოკ-სქემას 29-ე ნახაზზე წარმოდგენილი სახე აქვს. აღსანიშნავია, რომ ალგორითმში გამოყენებულია შემდეგი ცვლადები: S, რომელიც ითვლის საძიებელ ცვლადთან თანხვედრილი ელემენტების რაოდენობას, K- საძიებელი მნიშვნელობის მქონე ცვლადი და P, რომელშიც საძიებელი ცვლადის მნიშვნელობასთან მასივის თანხვერდილი ელემენტების ინდექსები ინახება.



ნახ. 28 წრფივი ძებნის ალგორითმი

1.4.6. ბინარული ძებნის ალგორითმი

(დიხოტომიის მეთოდი)

განვიხილოთ *n* რაოდენობის ელემენტისგან შემდგარი *A* მასივი და დიხოტომიის ანუ ორმაგი ძებნის (იგივე ბინარული ძებნის) ალგორითმით მოვძებნოთ საჭირო მნიშვნელობის მქონე ელემენტი მასში.

აღნიშნული ალგორითმის არსი შემდეგში მდგომარეობს:

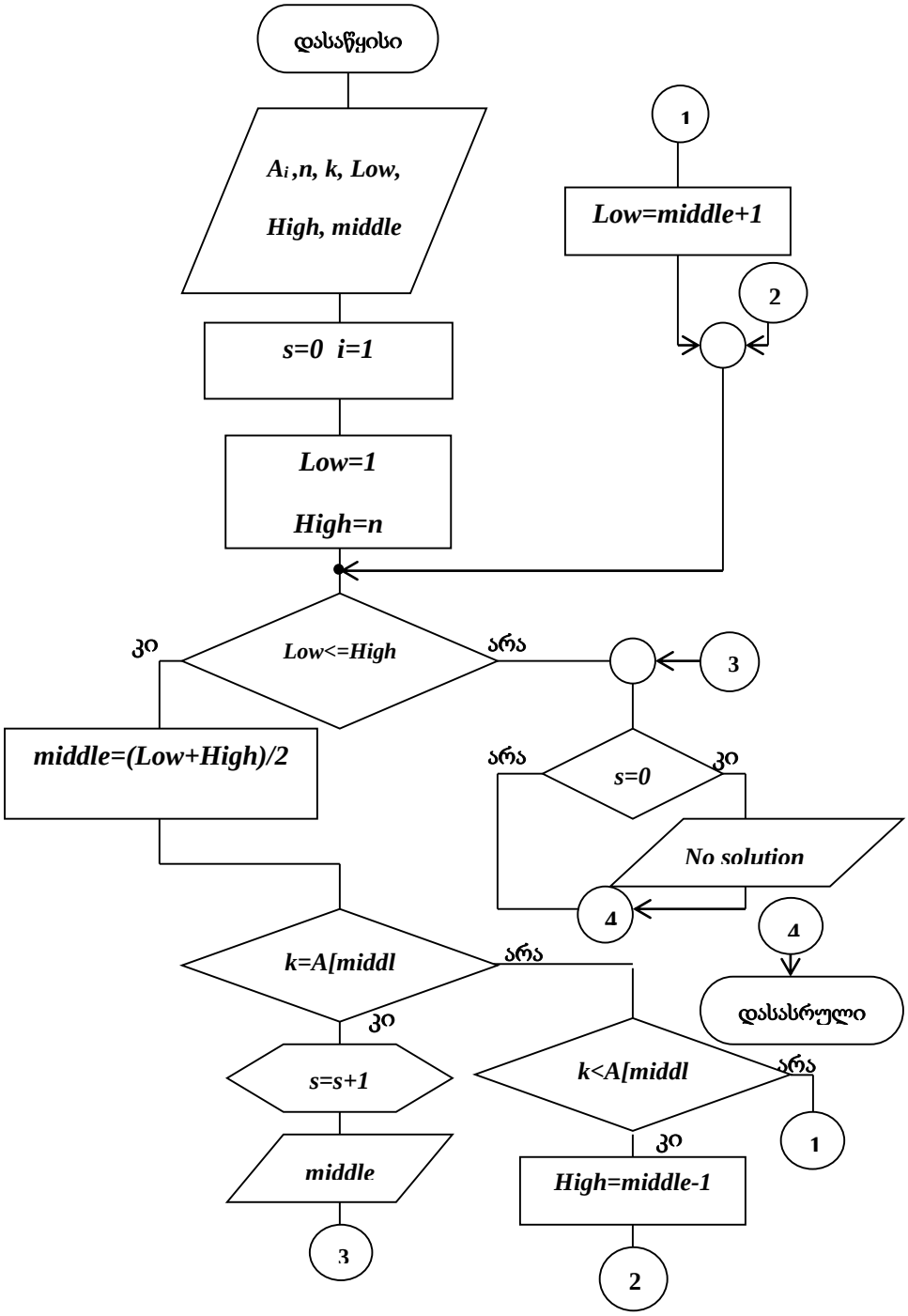
დიხოტომიის მეთოდის მუშაობის აუცილებელი პირობა არის მასივთა დახარისხება, ე.ი. ორმაგი ძებნის პროცესი ხორციელდება მხოლოდ დახარისხებულ მასივებში.

დავუშვათ, *A* მასივის ელემენტები დალაგებულია ზრდადობის მიხედვით. ალგორითმი ითვალისწინებს მასივის შუა ელემენტის მოძებნას, რომლის მნიშვნელობაც უნდა შედარდეს წინასწარ მოცემული ცვლადის მნიშვნელობას; თუ მათი მნიშვნელობები დაემთხვა, ეს ნიშნავს, რომ საჭირო ელემენტი მოიძებნა მასივში და ძიების პროცესი წყდება. წინააღმდეგ შემთხვევაში, თუ ცვლადის მნიშვნელობა აღმოჩნდება ნაკლები მასივის შუა ელემენტის მნიშვნელობაზე, ძიება წარმოებს მასივის პირველ ნახევარში, ხოლო თუ ცვლადის მნიშვნელობა გადააჭარბებს მასივის შუა ელემენტის მნიშვნელობას, ძიება ხორციელდება საწყისი მასივის მეორე ნახევარში.

ამდენად, ძიების პირველივე ეტაპზე ადგილი აქვს მასივის განახევრებას, შემდეგ მიღებული ნახევრებიდან ერთ-ერთის იგნორირების გზით და მასივის მეორე ნახევრის შუაზე გაყოფით ძებნა წარმოებს საწყისი მასივის მეოთხედში და ა.შ. პროცესი გრძელდება მანამდე, სანამ მასივის ერთ-ერთი ნახევრის შუა ელემენტის მნიშვნელობა არ დაემთხვევა მოცემული ცვლადის მნიშვნელობას, ან სანამ საჭირო ელემენტი არ მოიძებნება.

ამგვარად, მართალია, დიხოტომიის ალგორითმი მასივთა დახარისხებას წინასწარ მოითხოვს, მაგრამ სანაცვლოდ ძებნის პროცესს აჩქარებს ყოველ ეტაპზე საწყისი და მიღებული მასივების განახევრების გზით.

დასმული ამოცანის გადაწყვეტის ალგორითმის ბლოკ-სქემა 30-ე ნახაზზეა წარმოდგენილი. მასში გამოყენებულია შემდეგი ცვლადები: ***middle*** – აღნიშნავს მასივის შუა ელემენტის ინდექსს, ***Low*** – მასივის საწყისი ელემენტის ინდექსს, ***High*** - მასივის ბოლო ელემენტის ინდექსს; ***S*** არის ცვლადი, რომლის მიხედვით განისაზღვრება არსებობს თუ არა საძიებელი მნიშვნელობის ელემენტი მასივში, ***k*** – საძიებელი მნიშვნელობის ცვლადი.



ნახ. 300 ბინარული ძებნის ალგორითმი

დავალება

1. *შეადგინეთ 15-ელემენტიანი მთელრიცხვა მასივის „ჩაძირვის“ მეთოდით კლებადობით დახარისხების ალგორითმის ბლოკ-სქემა.*
2. *შეადგინეთ 20-ელემენტიანი მთელრიცხვა მასივის მარტივი გადანაცვლების მეთოდით კლებადობით დახარისხების ალგორითმის ბლოკ-სქემა.*
3. *შეადგინეთ ალგორითმის ბლოკ-სქემა, რომელიც ბინარული ძებნის მეთოდით განსაზღვრავს საჭირო ელემენტის ინდექსს 30-ელემენტიან მთელრიცხვა მასივში (მასივის ელემენტები წინასწარ დახარისხებულია კლებადობით).*
4. *შეადგინეთ [4x4] კვადრატული მატრიცის ელემენტების სტრიქონების მიხედვით ზრდადობით დახარისხების ალგორითმის ბლოკ-სქემა.*
5. *შეადგინეთ [5x5] კვადრატული მატრიცის ელემენტების სვეტების მიხედვით ზრდადობით დახარისხების ალგორითმის ბლოკ-სქემა.*
6. *შეადგინეთ ალგორითმის ბლოკ-სქემა, რომელიც წრფივი ძებნის მეთოდით განსაზღვრავს საჭირო ელემენტის/ელემენტების ინდექსს/ინდექსებს 20-ელემენტიან მთელრიცხვა მასივში და დათვლის მათ რაოდენობას.*
7. *შეადგინეთ 15-ელემენტიანი მთელრიცხვა მასივის კომბინირებული დახარისხების მეთოდით კლებადობით დახარისხების ალგორითმის ბლოკ-სქემა.*

1.5. მატრიცების გამრავლების ალგორითმი

ერთნაირი თვისებების მქონე სკალარული ცვლადების მართკუთხა ცხრილს **მარტიცა** ეწოდება და მას აღნიშნავენ:

$$A=\begin{pmatrix} a_{11}\ a_{12}.....a_{1n} \\ a_{21}\ a_{22}.....a_{2n} \\ ..... \\ a_{m1}\ a_{m2}.....a_{mn} \end{pmatrix}$$

მატრიცის თითოეული ელემენტი შესაბამისი ინდექსებითაა აღნიშნული. ორინდექსიანი აღნიშვნის შემთხვევაში პირველი ინდექსი მიუთითებს ყოველთვის იმ სტრიქონის ნომერს, ხოლო მეორე ინდექსი იმ სვეტის ნომერს, რომელთა გადაკვეთაზე მოცემული ელემენტია მოთავსებული.

თუ $[Ai,j]_{m,n}$ მატრიცაში $n=m$, მაშინ მას **კვადრატულს** უწოდებენ.

კვადრატული მატრიცის მთავარი დიაგონალი წარმოადგენს იმ ელემენტების სიმრავლეს, რომლებიც ერთიდაიგივე ნომრის სტრიქონისა და სვეტის გადაკვეთაზე მდებარეობენ:

$$\{a_{11},\ a_{22},\ a_{33},.....,\ a_{nn}\}.$$

ორი მართკუთხა მატრიცის ნამრავლი

$$A=\begin{pmatrix} a_{11}\ a_{12}.....a_{1n} \\ a_{21}a_{22}.....a_{2n} \\ .... \\ a_{m1}\ a_{m2}.....a_{mn} \end{pmatrix} \quad \text{და} \quad B=\begin{pmatrix} b_{11}\ b_{12}.....b_{1q} \\ b_{21}b_{22}.....b_{2q} \\ .... \\ b_{n1}\ b_{n2}.....b_{qn} \end{pmatrix}$$

ეწოდება ისეთ

$$C=\begin{pmatrix} c_{11}\ c_{12}.....c_{1q} \\ c_{21}\ c_{22}.....c_{2q} \\ .... \\ c_{m1}\ c_{m2}.....c_{mq} \end{pmatrix}$$

მატრიცას, რომლის i-ური სტრიქონისა და j-ური სვეტის გადაკვეთაზე მოთავსებული C_{ij} ელემენტი A მატრიცის i-ური სტრიქონისა და B მატრიცის j-ური სვეტის ელემენტების ნამრავლების ჯამის ტოლია.

$$C_{ij}= \sum_{k=1}^{n} a_{ik}+b_{kj} \qquad (i=1, 2,\ldots,m;\ j=1, 2,\ldots,n).$$

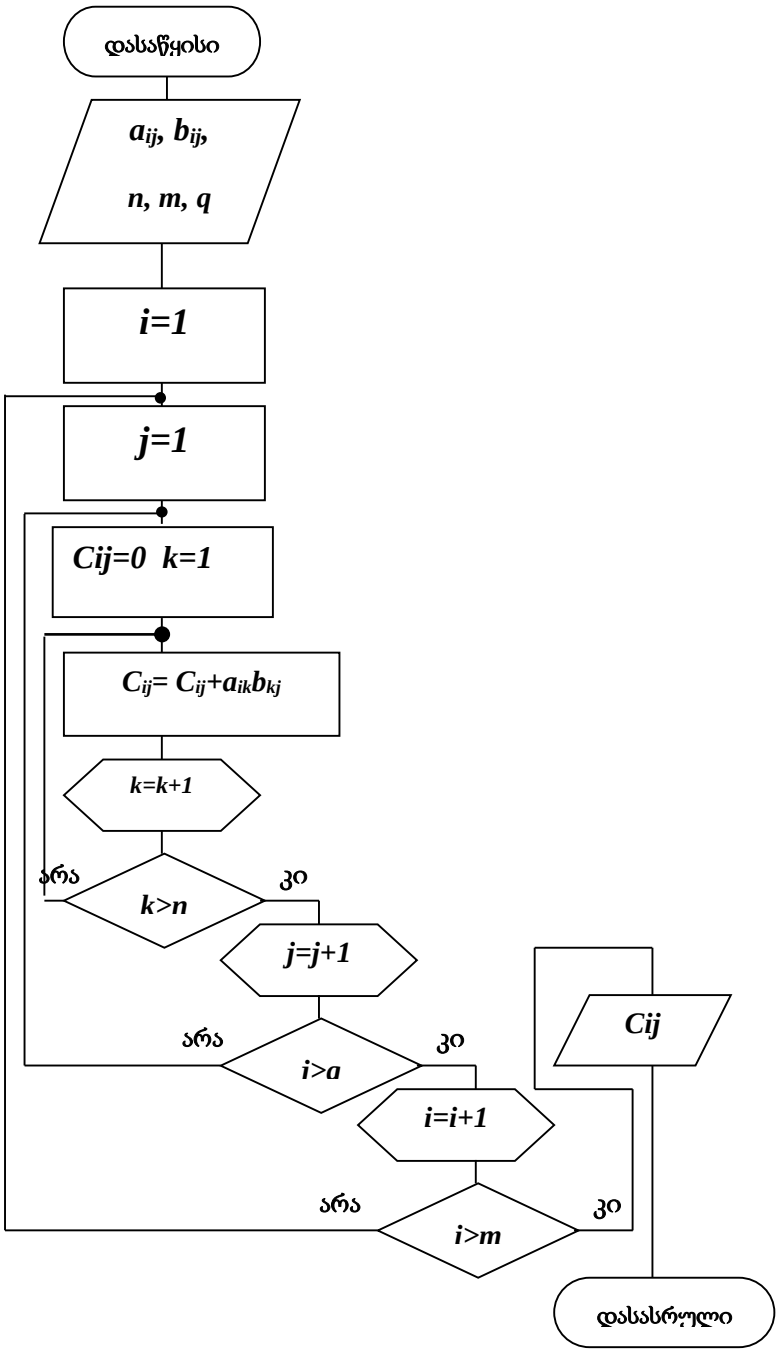
დავუშვათ, მოცემულია ორი *a[2x3]* და *b[3x4]* მატრიცა. მათი ნამრავლი მათემატიკურად შემდეგნაირად შეგვიძლია წარმოვადგინოთ:

$$\begin{pmatrix} a_{11}\ a_{12}\ a_{13} \\ a_{21}\ a_{22}\ a_{23} \end{pmatrix} \begin{pmatrix} b_{11}\ b_{12}\ b_{13}\ b_{14} \\ b_{21}\ b_{22}\ b_{23}\ b_{24} \\ b_{31}\ b_{32}\ b_{33}\ b_{34} \end{pmatrix} =$$

$$\begin{pmatrix} a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31} \ldots\ldots\ldots & a_{11}b_{14}+a_{12}b_{24}+a_{13}b_{34} \\ a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31} \ldots\ldots\ldots & a_{21}b_{14}+a_{22}b_{24}+a_{23}b_{34} \end{pmatrix}$$

ხოლო, მათი გადამრავლების ალგორითმის ბლოკ-სქემა 31-ე ნახაზზეა ნაჩვენები.

საზოგადოდ, საჭიროა აღვნიშნოთ, რომ ორი მართკუთხა მატრიცის გამრავლება შესაძლებელია მხოლოდ იმ შემთხვევაში, თუ პირველი მატრიცის სვეტების რაოდენობა ემთხვევა მეორე მატრიცის სტრიქონების რაოდენობას. შედეგად მიიღება მატრიცა, რომელიც იმდენ სტრიქონს შეიცავს, რამდენი სტრიქონიცაა პირველ მატრიცაში და იმდენ სვეტს, რამდენი სვეტისგანაც შედგება მეორე მატრიცა.



ნახ. 31 მატრიცების გამრავლების ბლოკ-სქემა

კითხვები თვითშეფასებისათვის:

1. *განმარტეთ კვადრატული მატრიცის მთავარი დიაგონალის ელემენტების არსი.*
2. *განმარტეთ, თუ რა შემთხვევაშია შესაძლებელი ორი მართკუთხა მატრიცის გადამრავლება.*
3. *მოცემულია ორი a[3x4] და b[4x3] მატრიცა. შეადგინეთ ამ მატრიცების ნამრავლის ამსახველი ალგორითმის ბლოკ-სქემა.*

2. სტრუქტურული დაპროგრამება (C)

2.1. დაპროგრამების პარადიგმები

დაპროგრამების პარადიგმა – ცნებათა და კონცეფციათა ერთობლიობაა, რომელიც პროგრამების დაწერის სტილს განსაზღვრავს. მას ენაში შეაქვს თვისობრივი ცვლილებები. დღეისათვის ცნობილია დაპროგრამების რამდენიმე სტილი.

დაპროგრამების პარადიგმებია:

- **პროცედურული დაპროგრამება** – პროგრამა წარმოადგენს კონკრეტული ამოცანის შესაბამის მარტივ პროცედურას ან ფუნქციას. შექმნილი პროცედურები, როგორც "სამშენებლო აგურები", სხვა პროგრამებში შეიძლება იქნეს გამოყენებული. შესაბამისად, დროთა განმავლობაში იქმნება სხვადასხვა საგნობრივი სფეროსათვის დამახასიათებელი პროცედურებისა და ფუნქციების უამრავი ბიბლიოთეკა.
- **სტრუქტურული დაპროგრამება** – რეკომენდაციები, რომლებიც ხსნიან თუ როგორ ჩამოყალიბდეს პროგრამის სტრუქტურა მისი პროცედურებად დაყოფის მიზნით, კოდის რა ნაწილი გამოიყოს ცალკეული პროცედურისთვის, როგორ გახდეს ამოცანის გადაწყვეტის ალგორითმი მარტივი და აღქმადი, როგორ მოხდეს პროცედურების ერთმანეთთან დაკავშირება (მაგალითი: დაპროგრამების ენა **Pascal**).
- **მოდულური დაპროგრამება** – პროგრამები იქმნება ცალკეული მოდულისგან, რომლებიც ათობით სხვადასხვა პროცედურისა და ფუნქციისგან შედგება. ასეთი პროგრამების ეფექტურობა მით უფრო მაღალია, რაც უფრო ნაკლებადაა ცალკეული მოდული ერთმანეთზე დამოკიდებული. მოდულების ავტონომიურობა საშუალებას იძლევა შეიქმნას მოდულების ბიბლიოთეკა, რომელიც შემდგომში შეიძლება გამოყენებულ იქნეს პროგრამის ასაგებ ბლოკებად.
- **ობიექტზე ორიენტირებული დაპროგრამება (Object-Oriented Programming – OOP)** - დაპროგრამების თანამედროვე პარადიგმა. პროგრამა იყოფა მოდულებად და გარდაიქმნება ურთიერთდაკავშირებულ ობიექტებად.

ობიექტზე ორიენტირებულ დაპროგრამებას, თავის მხრივ, ორი კონცეფცია გააჩნია. ესენია:

- **პროცესზე ორიენტირებული მოდელი** – კონცეპტუალურად პროგრამა შეიძლება განხილულ იქნეს, როგორც მონაცემების დამამუშავებელი კოდი. ამ მოდელს იყენებენ პროცედურული ენები, მაგალითად: **C, Pascal** და სხვა. პროგრამის ორგანიზება იმით განისაზღვრება,

თუ “რა ხდება”. თუმცა, ასეთი მიდგომა სხვადასხვა პრობლემას წარმოშობს პროგრამის ზომისა და სირთულის ზრდასთან დაკავშირებით.

- **ობიექტზე ორიენტირებული მოდელი** – კონცეპტუალურად პროგრამა ორგანიზებულია თავისი მონაცემების (ანუ, ობიექტების) გარშემო. აქ უმთავრესია მონაცემები, რომლებიც მართავენ კოდთან წვდომას. პროგრამის ორგანიზება განისაზღვრება იმით, თუ “რაზე ხდება მოქმედებები”. ამ მოდელს იყენებს დაპროგრამების ენა C++.

2.2. თვლის სისტემები

სისტემებს, რომლებშიც ნებისმიერი რიცხვი შეიძლება ჩაიწეროს სასრული რაოდენობის ციფრების (სიმბოლოების) მეშვეობით თვლის სისტემები ეწოდება.

ციფრულ ტექნიკაში, კომპიუტერებში გამოიყენება ეგრეთ წოდებული თვლის პოზიციური სისტემები, რომელშიც რიცხვის ყოველი ციფრის მნიშვნელობა (წონა) დამოკიდებულია მის მდებარეობაზე (პოზიციაზე) ამ რიცხვის ჩანაწერში.

ციფრების რაოდენობას, რომელიც გამოიყენება თვლის პოზიციურ სისტემაში, სისტემის ფუძე ეწოდება. იმისდამიხედვით, თუ რამდენ ციფრს ავირჩევთ რიცხვების ჩასაწერად გვექნება „ათობითი“, „ორობითი“, „რვაობითი“, და ა.შ სისტემები. ყველაზე უფრო გავრცელებულია თვლის ათობითი სისტემა, სადაც გამოყენებული ათი ციფრი: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

მაგ. რიცხვი 2578 შედგება ოთხი ციფრისაგან, რომლებიც მდებარეობის (პოზიციის) მიხედვით მიუთითებენ, რომ რიცხვი 2578 შედგება 8 ერთეულის, 7 ათეულის, 5 ასეულის და 2 ათასეულისაგან, ე.ი.

$$2578 = 2 \cdot 10^3 + 5 \cdot 10^2 + 7 \cdot 10^1 + 8 \cdot 10^0$$

ციფრულ ტექნიკაში, მათ შორის კომპიუტერში გამოიყენება თვლის ორობითი სისტემა, რომელშიც ნებისმიერი რიცხვი ჩაიწერება მხოლოდ ორი ციფრით, 0-სა და 1-ის საშუალებით.

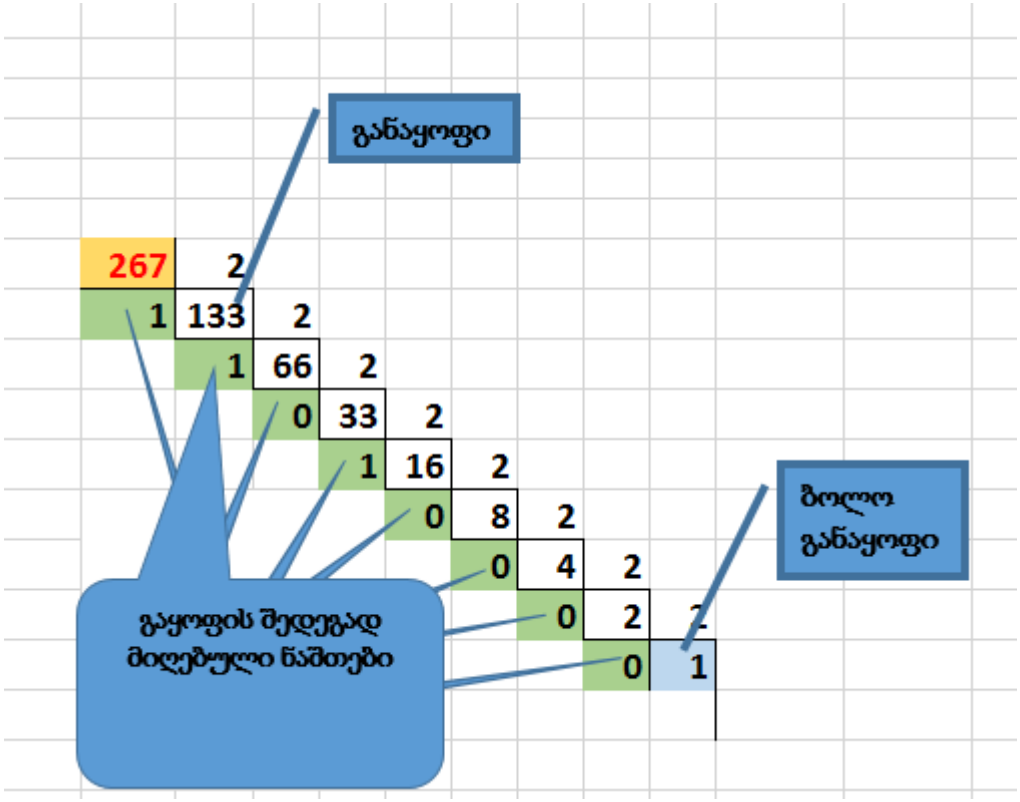
ორობითი რიცხვებია მაგ:

0110, 101, 110101, 00001, 10.

ნებისმიერი ათობითი რიცხვი შეიძლება გადავიყვანოთ თვლის ნებისმიერ სისტემაში და და პირიქით.

თვლის ათობითი სისტემიდან რიცხვის მთელი ნაწილის არაათობითში გადასაყვანად საჭიროა **მოცემული რიცხვი** და გაყოფის პროცესში მიღებული **განაყოფები** თანმიმდევრულად გავყოთ იმ სისტემის ფუძეზე, რომელშიც გადაგვყავს მოცემული რიცხვი (ნახ.32). გაყოფა უნდა ვაწარმოოთ მანამ, სანამ ბოლო განაყოფი არ აღმოჩნდება არაათობითი სისტემის ფუძეზე ნაკლები. შემდეგ უნდა ამოვწეროთ ბოლო განაყოფი და მიღებული ნაშთები შებრუნებული

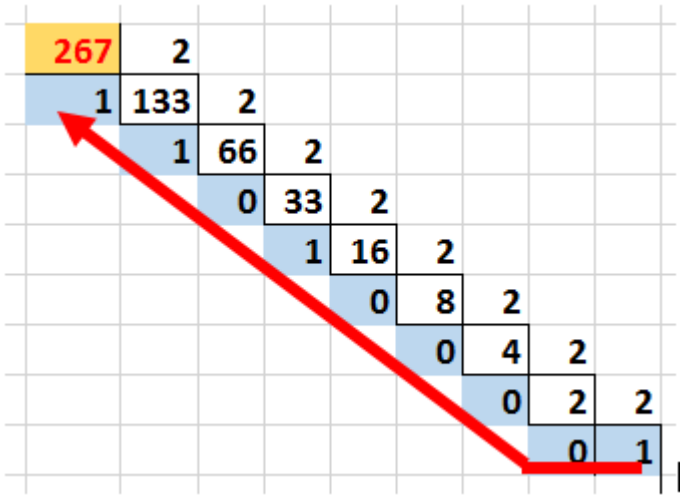
რიგით. მივიღებთ ათობითი რიცხვის არაათობით ეკვივალენტს. ნახ. 33-ზე ასახულია რიცხვის ათობითი სისტემიდან ორობითში გადაყვანის სქემა.



ნახ. 312 ათობითი რიცხვის ორობითში გადაყვანის სქემა

267 - არის ათობითი რიცხვი, რომელიც უნდა გადავიყვანოთ ორობით სისტემაში;

ამოვწერით ბოლო განაყოფი და მიღებული ნაშთები შებრუნებული რიგით, მივიღებთ 100001011 (ნახ.35)

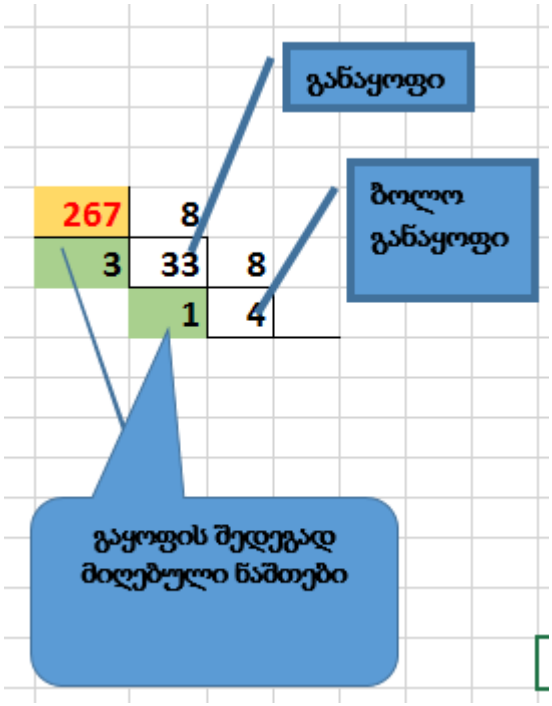


ნახ. 33 ათობითიდან ორობითში გადაყვანისას მიღებული ნაშთები

ე. ი. $267_{10} = 100001011_2$

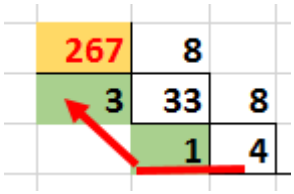
თვლის რვაობით სისტემაში რიცხვის ჩასაწერად გამოიყენება მხოლოდ 8 ციფრი: 0, 1, 2, 3, 4, 5, 6, 7.

ათობითი სისტემიდან რვაობით სისტემაზე გადასვლის წესები ორობით სისტემაზე გადასვლის წესის ანალოგიურია (ნახ.34), ოღონდ ცხადია აქ გაყოფა ხდება 8-ზე და წონა არის 8^n, სადაც n არის ციფრის პოზიცია 8-ობით რიცხვში.



ნახ. 324 ათობითი სისტემიდან რვაობითში გადაყვანა

ამოვწეროთ ბოლო განაყოფი და მიღებული ნაშთები შებრუნებული რიგით, მივიღებთ 413 (ნახ.35).



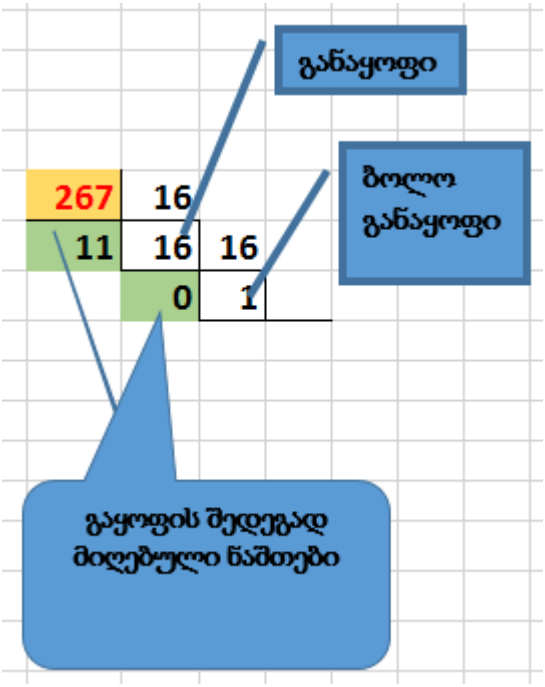
ნახ. 335 ათობითი სისტემიდან რვაობითში გადაყვანის შედეგად მიღებული ნაშთები

ე. ი. $267_{10} = 413_8$

თვლის 16-ობით სისტემაში გვაქვს 16 ციფრი, ესენია:

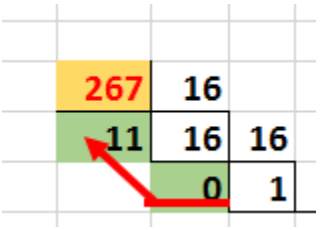
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

ათობითი სისტემიდან თექვსმეტობით სისტემაზე გადასვლის წესები ზემოთ აღწერილი წესის ანალოგიურია, ოღონდ ცხადია აქ გაყოფა ხდება 16-ზე და წონა არის 16^n სადაც n არის ციფრის პოზიცია 16-ობით რიცხვში (ნახ. 36).



ნახ. 346 ათობითი სისტემიდან თექვსმეტობითში გადაყვანა

ამოვწეროთ ბოლო განაყოფი და მიღებული ნაშთები შებრუნებული რიგით (ნახ.37).



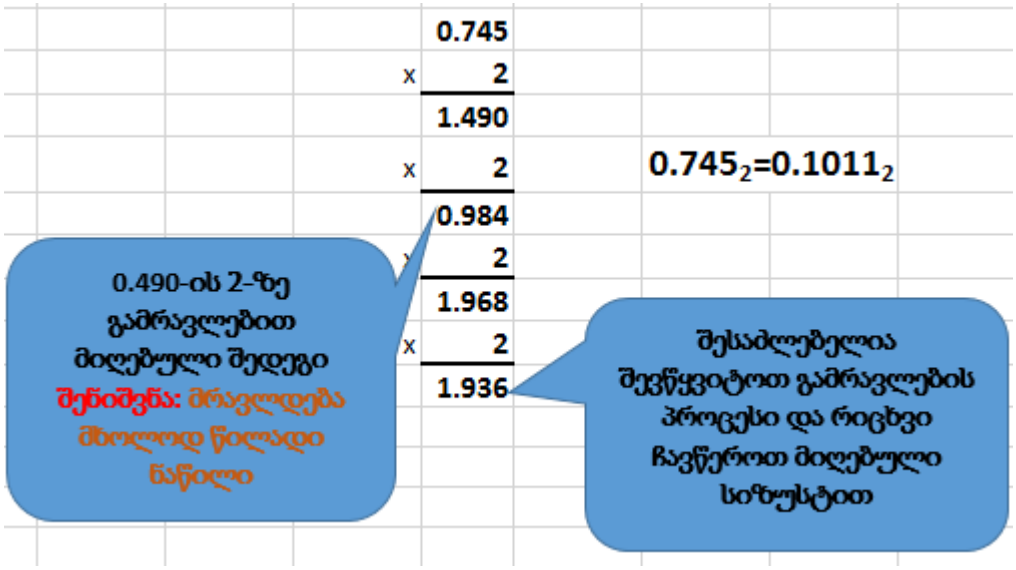
ნახ. 357 ათობითი სისტემიდან თექვსმეტობითში გადაყვანისას მიღებული ნაშთები

16-ობითში 10-ს შეესაბამება A, 11-ს - B, 12-ს - C, 13-ს - D, 14-ს - E, 15- ს - F, 16-ს - E, შესაბამისად ბოლო განაყოფისა და ნაშთების ამოწერით მივიღებთ : 10B-ს.

ე. ი. $267_{10} = 10B_{16}$

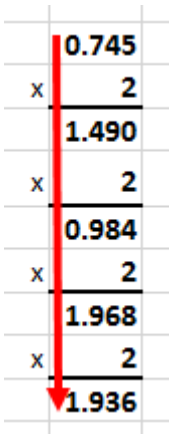
თვლის ათობითი სისტემიდან რიცხვის წილადი ნაწილის არაათობითში გადასაყვანად საჭიროა ამ რიცხვის წილადი ნაწილი და გამრავლების შედეგად მიღებული რიცხვის წილადი ნაწილი თანმიმდევრულად გავამრავლოთ არაათობითი სისტემის ფუძეზე (ნახ. 38). გამრავლებას ვაწარმოებთ მანამ, სანამ წილად ნაწილში არ მივიღებთ ნულს, ან გამრავლების პროცედურას ვწყვეტთ სიზუსტის მოთხოვნილი მნიშვნელობის შესაბამისად. ამოწერილი მთელი ნაწილები გვაძლევს ათობითი წილადის არაათობით ექვივალენტს (ნახ. 39) .

მაგ.



ნახ. 368 ათობითი სისტემიდან რიცხვის წილადი ნაწილის არაათობითში გადაყვანა

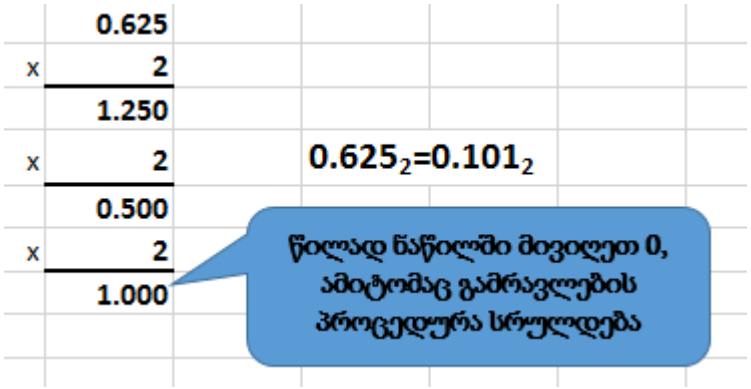
ამოვწეროთ მთელი ნაწილები მიმდევრობით (ნახ.41)



ნახ. 39

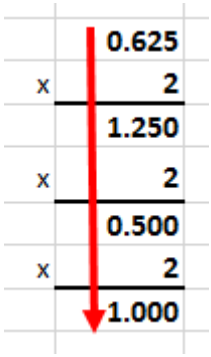
ე. ი. $0{,}745_2 \approx 10{,}1011_2$

მაგ. 2



ნახ. 370

ამოვწეროთ მთელი ნაწილები მიმდევრობით



ნახ. 381

ე. ი. $0{,}625_2 = 0{,}101_2$

რიცხვის არაათობითი სისტემიდან ათობითში გადასაყვანად საჭიროა ამ რიცხვის თითოეული თანრიგი გავამრავლოთ თავის „წონაზე“ და მიღებული შედეგები შევკრიბოთ.

მაგალითი 1.

$$100110_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 32 + 0 + 0 + 4 + 2 + 0 = 38_{10}$$

$$110101{,}011_2 = 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}$$

$$= 32 + 16 + 0 + 4 + 0 + 1 + 0 + \frac{1}{4} + \frac{1}{8} = 53{,}375_{10}$$

$$1340_8 = 1 \cdot 8^3 + 3 \cdot 8^2 + 4 \cdot 8^1 + 0 \cdot 8^0 = 512 + 192 + 32 = 736_{10}$$

$$134B_{16} = 1 \cdot 16^3 + 3 \cdot 16^2 + 4 \cdot 16^1 + 11 \cdot 16^0 = 4096 + 768 + 64 + 11 = 4939_{10}$$

დავალება

- **რიცხვი 1101,01 გადაიყვანეთ ორობითი სისტემიდან ათობითში**
- **რიცხვი $0{,}5625_{10}$ გადაიყვანეთ ათობითი სისტემიდან ორობითში**
- **რიცხვი 25 გადაიყვანეთ ათობითი სისტემიდან ორობითში**
- **რიცხვი 789 გადაიყვანეთ ათობითი სისტემიდან თექვსმეტობითში**

2.3. C დაპროგრამების ენის ძირითადი ცნებები

2.3.1. ანბანი

ენის ანბანს წარმოადგენს იმ სიმბოლოთა ერთობლიობა, რომელიც გამოიყენება ენაში. C ენის ანბანს შეადგენს:

1. ლათინური ანბანის ასოები: a–z A–Z
2. ათობითი ციფრები: 0-9
3. გრაფიკული სიმბოლოები: ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~
4. ცარიელი სიმბოლოები: გამოტოვება (space), ჰორიზონტალური ტაბულაცია, ვერტიკალური ტაბულაცია, ახალი ხაზი

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . /
: ; < = > ? [ \ ] ^ _ { | } ~
გამოტოვება (space), ჰორიზონტალური ტაბულაცია, ვერტიკალური ტაბულაცია,
form feed, ახალი ხაზი
```

2.3.2. იდენტიფიკატორი

იდენტიფიკატორი ეწოდება ციფრების, ასოების და სპეციალური სიმბოლოების თანმიმდევრობას. იდენტიფიკატორის მისაღებად შესაძლებელია ზედა და ქვედა რეგისტრის სიმბოლოების გამოყენება. სპეციალურ სიმბოლოდ შესაძლებელია გამოვიყენოთ (_).

C ენა სხვა დაპროგრამების ენებისგან ერთმანეთისაგან ანსხვავებს მაღალი და დაბალი რეგისტრის სიმბოლოებს.

ორი იდენტიფიკატორი **abc** და **ABC** ითვლება განსხვავებულად, რამდენადაც მასში განსხვავებულია რეგისტრები.

იდენტიფიკატორი იქმნება ცვლადის, ფუნქციის, სტრუქტურის გამოცხადებისას, არ შეიძლება იდენტიფიკატორის სახელი ემთხვეოდეს გასაღები (რეზერვირებული) სიტყვის სახელს;

იდენტიფიკატორის სახელის შერჩევისას სასურველია გათვალისწინებულ იქნეს ობიექტის ან მოქმედების არსი, მაგალითად: **speed_of_body, SpeedOfBody, BodySpeed**

იდენტიფიკატორის სწორი დასახელების მაგალითებია:
Counter, get_line, a, Parami _ab
იდენტიფიკატორის არასწორი დასახელების მაგალითებია:
%ab 12abc -x

დასახელებების შერჩევისას სასურველია გავითვალისწინოთ შემდეგი **რეკომენდაციები:**

სახელები გამოიყენეთ დასმული ამოცანის შესაბამისად;

შეარჩიეთ მოკლე აზრობრივი სახელი, რომელიც ასახავს ცვლადის, ფუნქციის, ობიექტის ან ტიპის დანიშნულებას;

არ დაიწყოთ ხაზგასმის სიმბოლოთი, რამდენადაც ასეთი წესი ფართოდ გამოიყენება დაპროგრამების სისტემის ბიბლიოთეკებში;

გამოიყენეთ დასახელებების ერთიანი სისტემა; აქ შეიძლება გამოყოთ რამდენიმე ვარიანტი:

დაიწყეთ ზედა რეგისტრის ანბანის ასოთი, თუ საჭიროა ხაზგასმულ იქნას იდენტიფიკატორის უნიკალურობა;

გამოიყენეთ ხაზგასმის სიმბოლო ან ზედა რეგისტრის სიმბოლო იდენტიფიკატორის შიგნით, კარგად აღქმადი რთული იდენტიფიკატორის ასაგებად.

შესაძლებელია ცვლადების, ობიექტების, ტიპების სახელები შევარჩიოთ მათი დანიშნულების მიხედვით, მაგრამ იშვიათად ხდება, რომ ეს დაგვეხმაროს პროგრამის შინაარსის გარკვევაში, ამიტომაც აუცილებელი ხდება კომენტარების დართვა.

ერთსტრიქონიანი კომენტარისათვის გამოიყენება // სიმბოლოთა წყვილი, რომელსაც ჩვეულებრივ მოსდევს კომენტარის სტრიქონი;

რამდენიმე სტრიქონიანი კომენტარი იწყება / * სიმბოლოთა წყვლით და მთავრდება * / სიმბოლოებით.

მაგ. / კომენტარის სტრიქონი*

**/*

რეკომენდაციები:

დაიწყეთ პროგრამა მოკლე კომენტარებით, რომელიც აღწერს ალგორითმის ძირითად ეტაპებს, ცვლადებს მონაცემთა შენახვისათვის, შუალედური და გამოსავალი შედეგებისათვის.

არ ჩართოთ კომენტარი პროგრამის სტრიქონის შუაში;

არ დაწეროთ ადვილად გასაგები პროგარმული სტრიქონის კომენტარები.

2.3.3. რეზერვირებული სიტყვები

გასაღები სიტყვები ეს არის დარეზერვირებული იდენტიფიკატორები, რომელთაც გააჩნიათ გარკვეული დანიშნულება. მათი გამოყენება შესაძლებელია მხოლოდ დანიშნულებისამებრ.

გასაღები სიტყვებია:

auto double int struct break else long switch register tupedef char extern return void case float unsigned default for signed union do if sizeof volatile continue enum short while.

ამასთან ერთად განსაზღვრულ ვერსიებში დარეზერვირებულ სიტყვებს წარმოადგენს:

_asm, fortran, near, far, cdecl, huge, pascal, interrupt.

2.4. მარტივი პროგრამების შექმნა დაპროგრამების კონსტრუქციებისა და მმართველი სტრუქტურების გამოყენებით

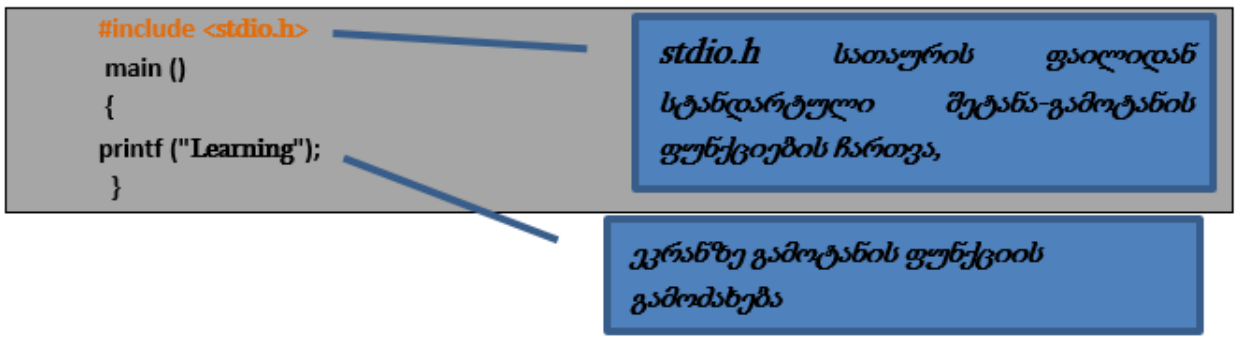
2.4.1. პროგრამის სტრუქტურა

C ენაზე დაწერილი უმარტივესი პროგრამა, შესაძლებელია წარმოვადგინოთ შემდეგი სახით:

```
main ()
{
}
```

პროგრამის ძირითადი ფუნქციის სახელწოდებაა main. ცარიელი ფრჩხილები მიუთითებს, რომ main ფუნქციას არ გააჩნია არგუმენტები. ფიგურული ფრჩხილები აღნიშნავს ძირითადი პროგრამის დასაწყისს და დასასრულს. რამდენადაც ჩვენს მიერ მოტანილ ფრაგმენტში ფიგურული ფრჩხილების შიგნით არაფერი წერია, ჩვენი პროგრამა არაფერს გააკეთებს, უბრალოდ შესაძლებელია მისი კომპილაცია და exe ფაილის მიღება.

შევადგინოთ პროგრამა, რომელიც ეკრანზე გამოიტანს ტექსტურ შეტყობინებას, მაგ. **Learning-ს.**



სტანდარტული ფუნქციების გამოსაყენებლად, აუცილებელია ტრანსლიატორს მივუთითოთ, რომ არსებობს ასეთი დასახელების ფუნქცია და ჩამოვთვალოთ მისი არგუმენტები. მხოლოდ ამ შემთხვევაში შეუძლია ტრანსლიატორს განსაზღვროს სწორად ვიყენებთ თუ არა ამ ფუნქციას. სტანდარტული ფუნქციები აღწერილია ე.წ. სათაურის ფაილებში გაფართოებით ***.h** (საქაღალდე **C:\Dev-Cpp\include**)

სათაურის ფაილის ჩართვისათვის გამოიყენება პრეპროცესორის დირექტივა (ბრძანება) **#include**, რომლის შემდეგაც კუთხურ ფრჩხილებში ვუთითებთ ფაილის სახელს. ფრჩხილების შიგნით არ შეიძლება ცარიელი სიმბოლოს (გამოტოვების) გამოყენება. ყოველი სათაურის ფაილის ჩასართავად გამოიყენება ახალი ბრძანება **#include.**

```
#include <stdio.h>
#include <stdlib.h>
```

მაგ.

ინფორმაციის ეკრანზე გამოსატანად გამოიყენება ფუნქცია **printf.** უმარტივეს შემთხვევაში მას გააჩნია ერთადერთი არგუმენტი - ეკრანზე გამოსატანი სტრიქონი ბრჭყალებში.

C ენის ყოველი ოპერატორი მთავრდება წერტილ-მძიმით.

2.4.2. პროგრამული ინტერფეისი

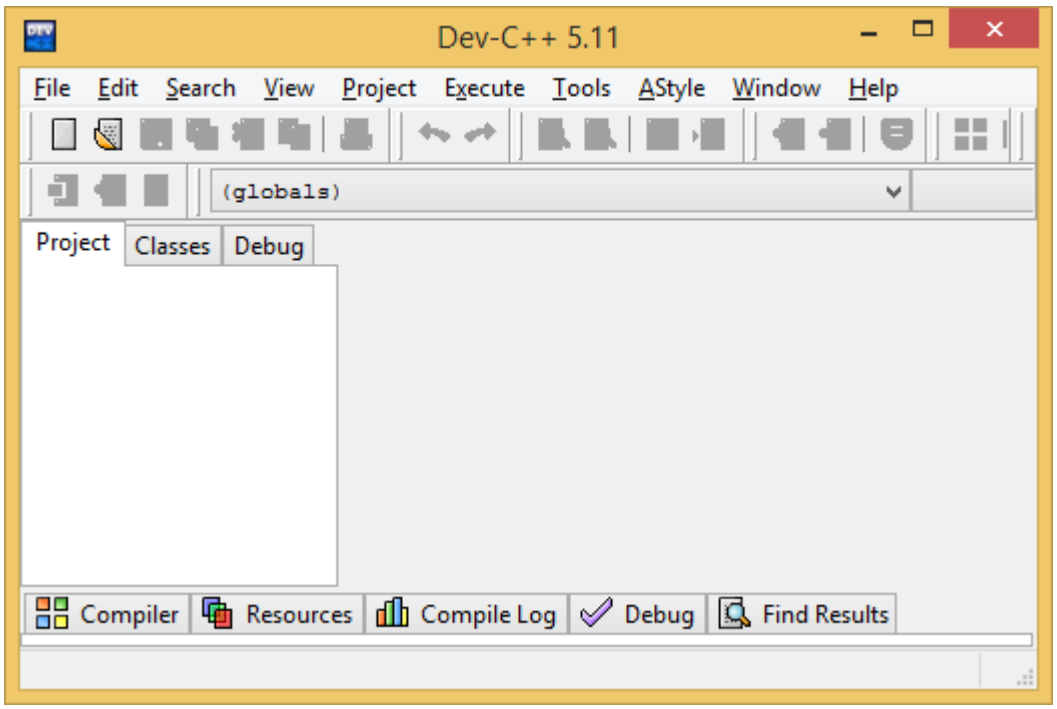
პროგრამის შესრულებაზე გაშვებამდე აუცილებელია პროგრამის ტრანსლაცია, გამართვა და შემოწმება. თანამედროვე ეტაპზე ყველა ეს მოქმედება გაერთიანებულია სპეციალური პროგრამა-გარსის შიგნით, რომელსაც ეწოდება პროგრამის შემუშავების ინტეგრირებული გარემო **(IDE – Integrated Development Environment).**

მასში შედის:

- ტექსტის რედაქტორი;
- ტრანსლიატორი;
- გამმართველი

პროექტის შექმნა

პროგრამის დასაწერად აუცილებელია ორი რამ: საწყისი ფაილის შესაქმნელად რედაქტორი და პროგრამა, რომელიც საწყის ტექსტს გარდაქმნის მანქანისათვის გასაგებ კოდირებაზე. ერთ-ერთი პოპულარული გარემო პროგრამის შემუშავებისათვის ეს არის **Microsoft** -ის პროდუქცია **Visual C++.** არსებობს ასევე სხვა მრავალი გარემოც: **GNU C++, Dev-C++.**



ნახ. 392 Dev-C++ დაპროგრამების გარემო

განვიხილოთ დაპროგრამების გარემო **Dev-C++ (ნახ.42).**

Dev-C ++ ეს არის უფასო **IDE Windows** სისტემისათვის, რომელიც, როგორც ძირითად კომპილიატორს იყენებს ან **MinGW** ან **TDM-GCC**-ს.

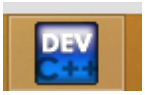
პროგრამის ინსტალაცია

www.bloodshed.net ვებ-გვერდიდან შესაძლებელია საინსტალაციო ფაილის ჩამოტვირთვა. გავუშვათ ჩამოტვირთული ფაილი შესრულებაზე. ინსტალაციის დროს უნდა მივყვეთ შესაბამის ინსტრუქციას.

2.4.3. პროექტის შექმნა

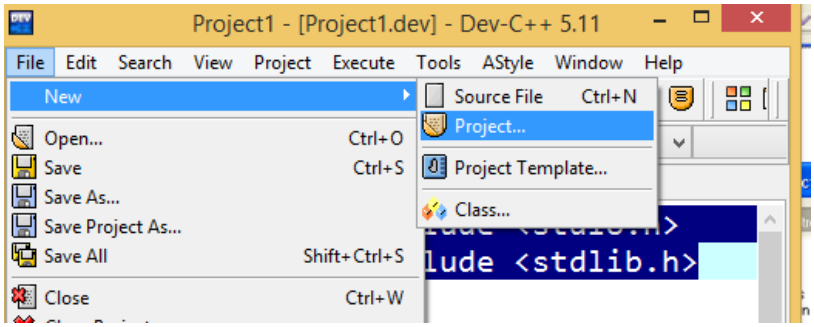
გავუშვათ Dev-C++-ის გამშვები ფაილი შესრულებაზე.

გამშვებ ფაილს აქვს ნახ. 43-ზე მოცემული გრაფიკული აღნიშვნა



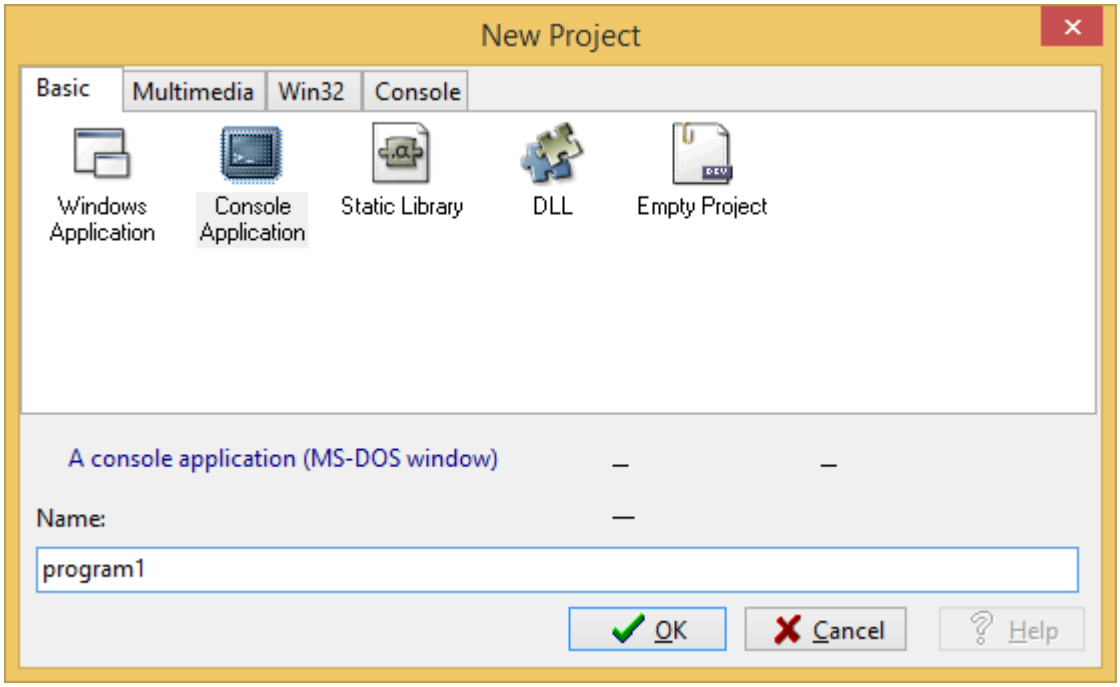
ნახ. 403

საწყის ეტაპზე უნდა შეიქმნას პროექტი. ამისათვის მენიუში **File** შევირჩიოთ ბრძანება **New –> Project** (ნახ.44).

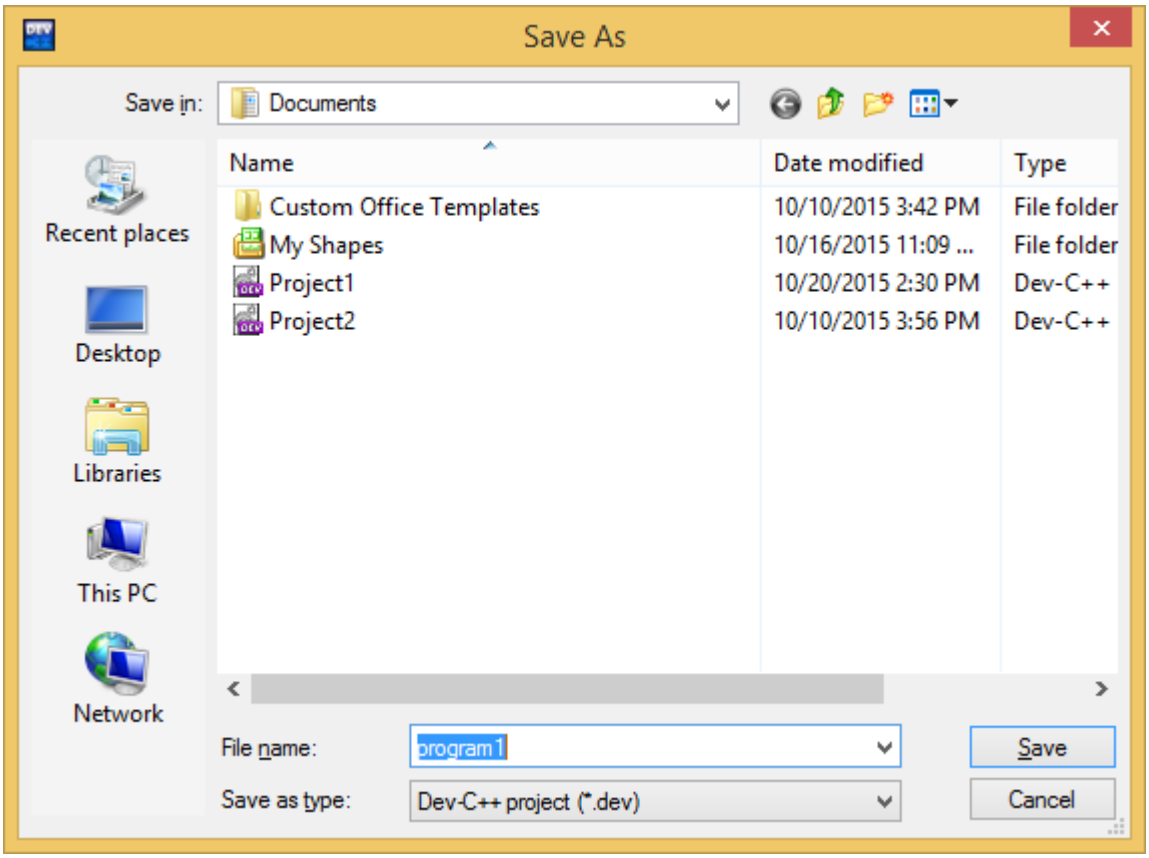


ნახ. 414 ახალი პროექტის შექმნა

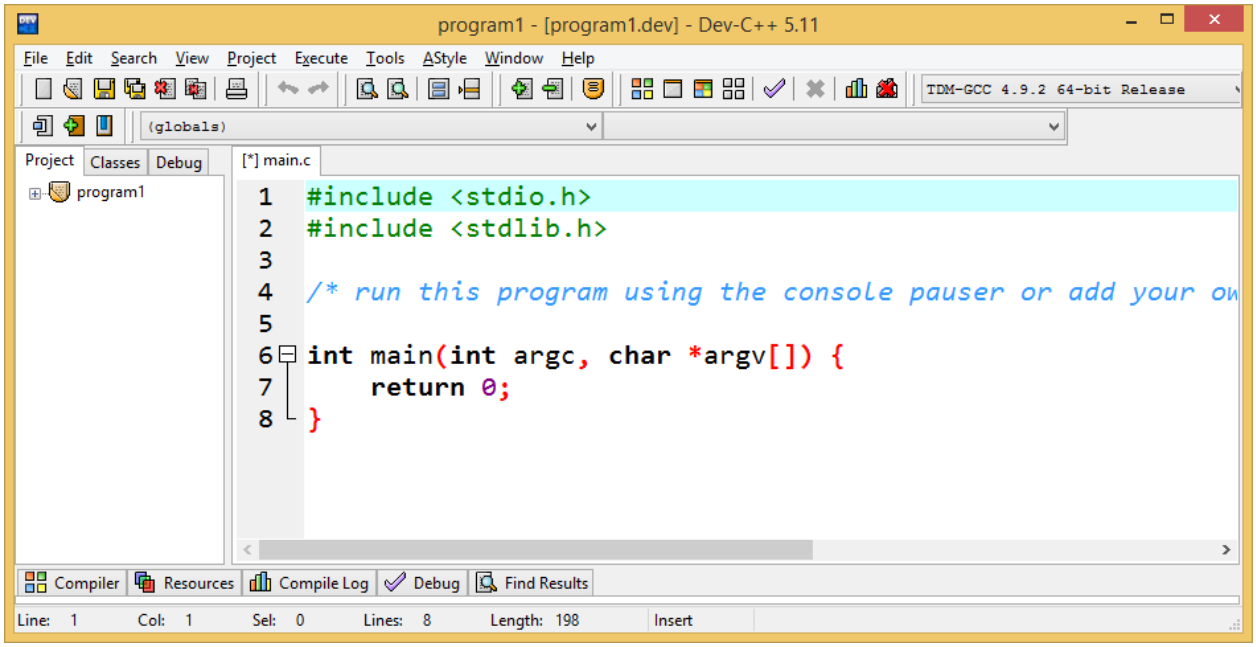
დიალოგურ ფანჯარაში (ნახ.45) შევირჩიოთ **Console Application.** დავარქვათ პროექტს სახელი (ნახ.46) და გამოსულ ფანჯარაში დავიწყოთ პროგრამის ტექსტის შეტანა (ნახ.47).



ნახ. 425 Console Aplication რეჟიმის შეცვლა

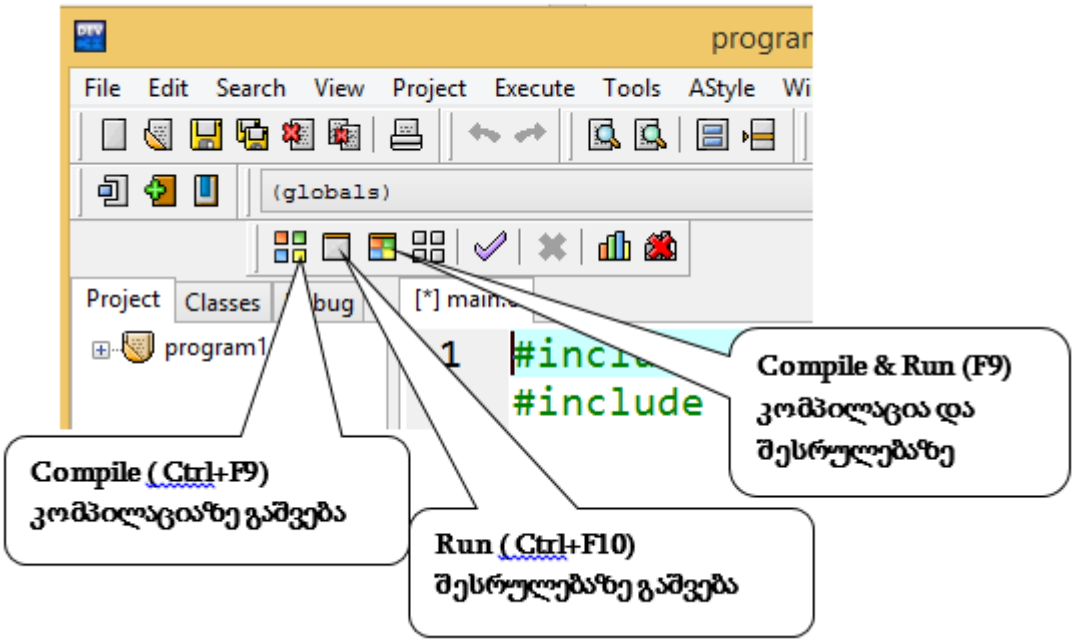


ნახ. 436 პროექტის შენახვა



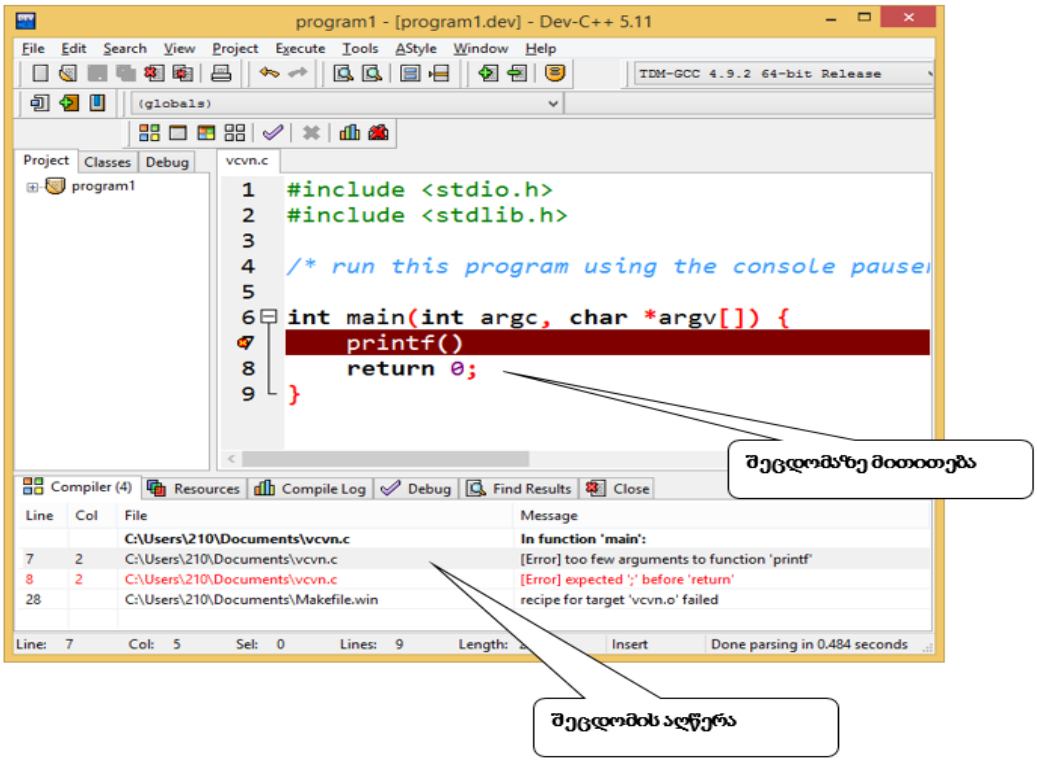
ნახ. 447

პროექტის კომპილაცია და შესრულებაზე გაშვება ხდება სპეციალური ღილაკების ან კლავიატურის კლავიშების კომბინაციით (ნახ.48).



ნახ. 458 პროექტის კომპილაციის და შესრულებაზე გაშვების ღილაკები

კომპილაციის დროს აღმოჩენილ შეცდომაზე კომპილერი რეაგირებს და ახორციელებს მითითებას: გამოყოფს სტრიქონს, რომელშიც შეცდომა იქნა აღმოჩენილი, ხოლო ქვედა ხაზზე მიუთითებს შეცდომის ტიპს (ნახ.49).



ნახ. 49 პროგრამის კომპილაციის შედეგად შეცდომებზე მითითება

ინტეგრირებულ გარემოში საკმარისია ავკრიფოთ პროგრამის ტექსტი და ერთი კლავიშის საშუალებით გავუშვათ შესრულებაზე.

DEV_C++ გარემოში პროგრამის გასაშვებად საკმარისია **F9** ღილაკზე თითის დაჭერა. თუ პროგრამაში არის შეცდომები, ჩვენ ფანჯრის ქვედა კუთხეში დავინახავთ შეტყობინებებს ამ შეცდომების შესახებ.

შეცდომის ძიებისას უნდა გავითვალისწინოთ, რომ ხშირად პროგრამის ტექსტში შეცდომა არის არა მონიშნულ სტრიქონში, არამედ ამ სტრიქონის ზედა სტრიქონზე.

თუ 50-ე ნახაზზე მოცემული სახით გავუშვებთ პროგრამას შესრულებაზე, მაშინ დავინახავთ, რომ პროგრამის შესრულების შედეგი ეკრანზე გამოვა მყისიერად.

```
#include <stdio.h>
main()
{
printf("Learning");  // ეკრანზე ტექსტის “Learning” გამოტანა
}
```

ნახ. 460 პროგრამის ფრაგმენტი

იმისათვის, რომ შესრულების შედეგი ეკრანზე დაყოვნდეს, საჭიროა სპეციალური ბრძანების გამოყენება, რომელიც უზრუნველყოფს შესრულების შედეგის ეკრანზე დარჩენას, მანამ სანამ არ დავაჭერთ კლავიატურის რომელიმე ღილაკს.

როგორც 51-ე ნახაზზე მოცემულ პროგრამაში მითითებული კომენტარებიდან ჩანს, შედეგის ეკრანზე დატოვების უზრუნველყოფა, მანამ სანამ არ დავაჭერთ კლავიატურის ნებისმიერ ღილაკს, ხორციელდება ფუნქციით **getch();**

ეს ფუნქცია აღწერილია სათაურის ფაილში **conio.h** .

```
#include <stdio.h>
#include <conio.h> // სათურის ფაილის ჩართვა
main()
{
printf("Learning");  // ეკრანზე ტექსტის “Learning” გამოტანა
getch();  // კლავიშის დაჭერამდე ლოდინი
        }
```

ნახ. 471 პროგრამის ფრაგმენტი

2.4.4. მონაცემთა ტიპები და ცვლადები

პროგრამა ახორციელებს სხვადასხვა ტიპის მონაცემების დამუშავებას. მონაცემები შესაძლებელია იყოს მარტივი და სტრუქტურული. მარტივი მონაცემებია - მთელი და ნამდვილი რიცხვები, სიმბოლოები და მიმთითებელი (მეხსიერებაში ობიექტის მისამართზე). მთელ რიცხვებს არ გააჩნიათ წილადური ნაწილი, ხოლო წილად რიცხვებს - კი. სტრუქტურული მონაცემებია - მასივები და სტრუქტურები.

ენაში ერთმანეთისაგან არჩევენ ცნებებს „მონაცემთა ტიპები“ და „ტიპის მოდიფიკატორი“. მონაცემთა ტიპი - ეს არის მაგალითად, მთელი ტიპის ცვლადი, ხოლო მოდიფიკატორი მიუთითებს მონაცემი არის ნიშნით თუ ნიშნის გარეშე. მთელი ტიპის ცვლადი ნიშნით ღებულობს, როგორც დადებით, ასევე უარყოფით მნიშვნელობას, ხოლო მთელი ნიშნის გარეშე, ღებულობს მხოლოდ დადებით მნიშვნელობებს. C-ში შეიძლება გამოვყოთ 4 საბაზო ტიპი, რომლებიც შემდეგი გასაღები სიტყვებით მოიცემა:

char - სიმბოლური

int - მთელი

float - ნამდვილი

double - ნამდვილი რიცხვი ორმაგი სიზუსტით

მთელი ტიპის ცვლადი - **int** (ინგლ. **integer** - მთელი), მეხსიერებაში იკავებს 4 ბაიტს;

ნამდვილი ტიპის ცვლადი, მნიშვნელობა, რომელსაც გააჩნია წილადური ნაწილი - **float** (ინგლ. **floating point** - მცოცავი წერტილი), მეხსიერებაში იკავებს 4 ბაიტს

სიმბოლური ტიპის ცვლადი - **char** (ინგლ. **character** -სიმბოლო), მეხსიერებაში იკავებს 1 ბაიტს.

ცხრილი 7 მონაცემთა ტიპები, ზომები და მნიშვნელობათა დიაპაზონები

ტიპი	ზომა	მნიშვნელობათა დიაპაზონი
unsigned char	1 ბაიტი	0 ... 255
signed char	1 ბაიტი	-128 ... 127
int	2 ან 4 ბაიტი	-32,768 ... 32,767 ან -2,147,483,648 ... 2,147,483,647
unsigned int	2 ან 4 ბაიტი	0 ... 65,535 ან 0 ... 4,294,967,295

2.4.4.1. ცვლადები

დაპროგრამების ენებში სხვადასხვა მნიშვნელობების შესანახად გამოიყენება ცვლადები. ცვლადი ეწოდება მეხსიერების ადგილს, რომელსაც ჰქვია სახელი (იდენტიფიკატორი).

C დაპროგრამების ენაში ყველა ცვლადი უნდა იყოს გამოცხადებული მის გამოყენებამდე. როდესაც ცვლადი გამოცხადდება, კომპილატორი მეხსიერებაში ანიჭებს მას ადგილს ტიპის მიხედვით.

გამოცხადებისას აღიწერება ცვლადის თვისებები.

მაგ. **: int fahr;**

სადაც **fahr** არის ცვლადის სახელი, ხოლო **int** არის ცვლადის ტიპი. ცვლადის ტიპი **int** მიუთითებს, რომ **fahr** ცვლადი იღებს მთელ მნიშვნელობებს.

C-ში ცვლადები შეიძლება ინიციალიზირებული იყოს განსაზღვრისას:

```
int a = 25, h = 6;
char g = 'Q', k = 'm';
float r = 1.89;
```

2.4.4.2. კონსტანტა (მუდმივა)

დაპროგრამებაში მონაცემები: რიცხვები, სიმბოლოები, სტრიქონები და ა.შ წარმოადგენენ პროგრამულ ობიექტებს. თუ პროგრამული ობიექტი პროგრამაში ინარჩუნებს ერთი და იგივე მნიშვნელობას, მას კონსტანტა (მუდმივა) ეწოდება.

C-ში განასხვავებენ ოთხი ტიპის კონსტანტებს: მთელი კონსტანტები, კონსტანტები მცოცავი მძიმით, სიმბოლური კონსტანტები და სტრიქონული ლიტერალები.

მთელი კონსტანტა: ეს არის ათობითი, რვაობითი ან თექვსმეტობითი რიცხვი;

ათობითი კონსტანტა შედგება ერთი ან რამდენიმე ათობითი ციფრისაგან, პირველი ციფრი არ უნდა იყოს 0 (ამ შემთხვევაში რიცხვი აღიქმება, როგორც რვაობითი)

რვაობითი კონსტანტა შედგება აუცილებელი 0-გან და ერთი ან რამდენიმე რვაობითი ციფრისაგან;

თექვსმეტობითი კონსტანტა აუცილებლად იწყება მიმდევრობით 0x ან 0X და შეიცავს ერთ ან რამდენიმე თექვსმეტობით ციფრს (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)

მთელი კონსტანტების მაგალითები:

ათობითი	რვაობითი	თექვსმეტობითი
16	020	0x10
127	0117	0x2B
240	0360	0XF0

ნახ. 48.2 მთელი კონსტანტების მაგალითები

კონსტანტა მცოცავი მძიმით - ეს არის ათობითი რიცხვი, რომელიც წარმოდგენილია ნამდვილი რიცხვის სახით, წილადი ნაწილი გამოყოფილია წერტილით ან ექსპონენტით.

მაგ. 115.75, 1.5E-2, -0.025, .075, -0.85E2

სიმბოლური კონსტანტა - წარმოადგენს სიმბოლოებს აპოსტროფებში. სიმბოლური კონსტანტის მნიშვნელობა წარმოადგენს სიმბოლოს რიცხვით კოდს.

მაგ. ' '- გამოტოვება,

'Q'- ასო Q,

'\n' - ახალი სტრიქონის სიმბოლო.

სტრიქონული კონსტანტა (ლიტერი) წარმოადგენს ბრჭყალებში მოქცეულ სიმბოლოთა თანმიმდევრობას (ზედა და ქვედა რეგისტრის სიმბოლოები და ციფრები) . მაგ. “Number N”; “Tbilisi”, “College N34”.

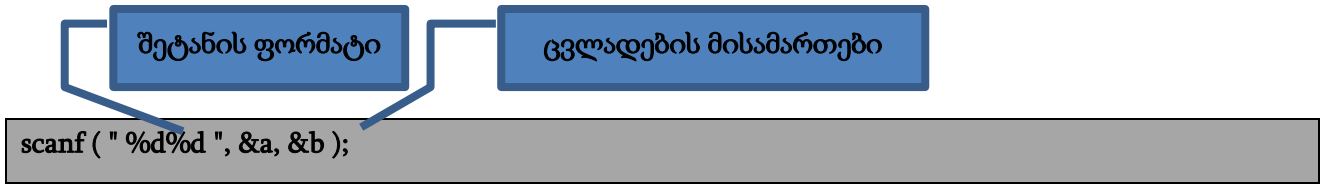
კონსტანტების გამოცხადება, ინიციალიზაცია

პირველ რიგში იწერება გასაღები სიტყვა const, შემდეგ ცვლადის ტიპი და მისი მნიშვნელობა.

მაგ. **`const double PI=3.14;`**

2.4.5. მონაცემთა შეტანა

მონაცემთა შეტანისათვის გამოიყენება ფუნქცია **scanf.**



ნახ. 53 Scanf ფუნქციის მაგალითი

შეტანის ფორმატი - ეს არის სტრიქონი ბრჭყალებში, რომელშიც მოცემულია ერთი ან რამდენიმე მონაცემთა შეტანის ფორმატი;

%d მთელი ტიპის მონაცემის შეტანა (**int** ტიპის ცვლადი);

%f ნამდვილი რიცხვის შეტანა (**float** ტიპის ცვლადი);

%c ერთი სიმბოლოს შეტანა (**char** ტიპის ცვლადი)

შეტანის ფორმატის შემდეგ უნდა დავსვათ მძიმე და ჩამოვთვალოთ მეხსიერების უჯრების მისამართები, რომელშიც უნდა ჩაიწეროს შეტანილი მნიშვნელობები.

განვასხვავოთ ერთმანეთისაგან ჩანაწერები **a** და **&a.**

a – a ცვლადის მნიშვნელობა;

&a – a ცვლადის მისამართი.

ფორმატების რაოდენობა სტრიქონში უნდა ემთხვეოდეს ცვლადების მისამართების რაოდენობას. ამასთან ერთად ცვლადების ტიპი უნდა ემთხვეოდეს მითითებულ ფორმატს.

მაგ. თუ **a** და **b** მთელი ტიპის ცვლადებია, შეტანის ფუნქციის ჩაწერის არასწორი ვარიანტებია:

scanf ("%d%d", &a); - ***შეცდომა:*** არ არის მითითებული სად უნდა ჩაიწეროს მეორე შეტანილი მნიშვნელობა;

scanf ("%d%d", &a, &b, &c); - ***შეცდომა:*** არ არის მოცემული **c** ცვლადის ფორმატი;

scanf ("%f%f", &a, &b); - **შეცდომა:** შეუძლებელია მთელი მნიშვნელობების შეტანა ათწილადი ფორმატით.

2.4.6. მონაცემთა გამოტანა

ცვლადების მნიშვნელობების და რიცხვების გამოტანა ეკრანზე ხორციელდება **printf** ფუნქციით.

printf-ის ფორმატი წააგავს **scanf**-ის ფორმატს

გამოტანის ფორმატში შესაძლებელია გამოვიყენოთ სპეციალური სიმბოლოები:

%d - მთელი რიცხვის გამოტანა;

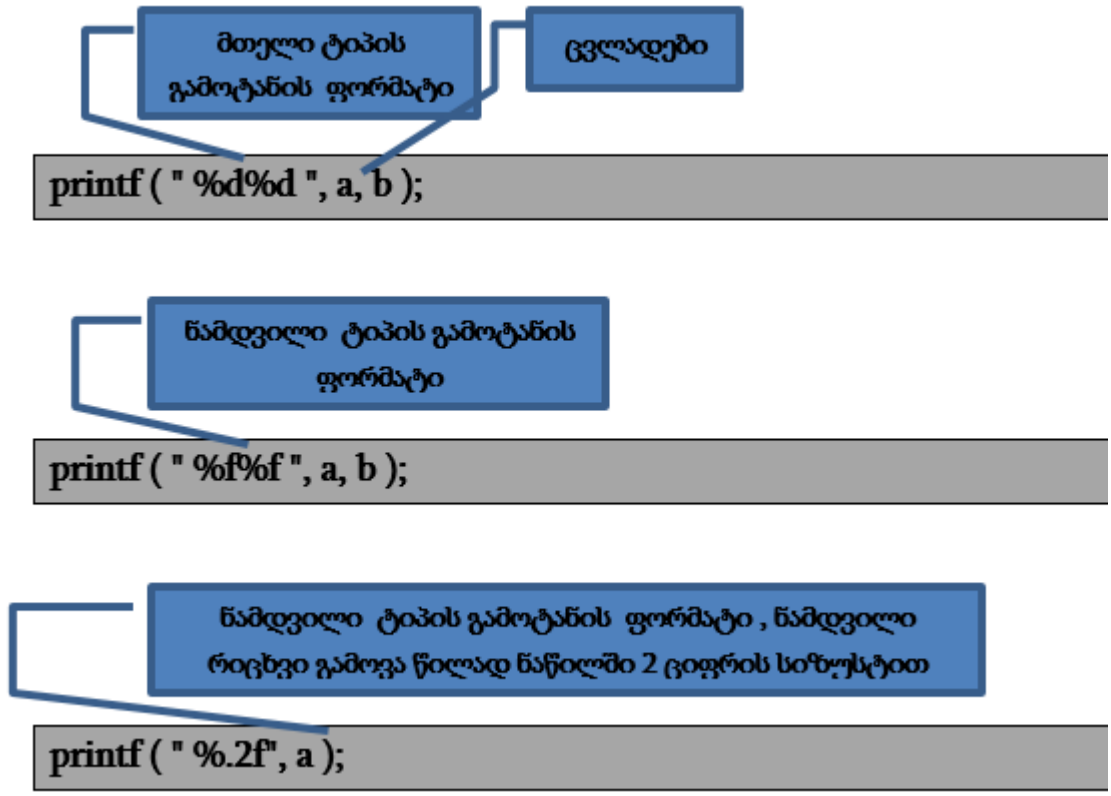
%f - ნამდვილი რიცხვის გამოტანა;

%c - ერთი სიმბოლოს გამოტანა;

%s - სიმბოლური სტრიქონის გამოტანა;

\n - ახალ სტრიქონზე გადასვლა;

\t - ტაბულაცია.



ნახ. 54 printf ფუნქციის ჩაწერის მაგალითები

მონაცემთა გამოტანის ფორმატები

ა) მთელი რიცხვისათვის

მაგალითი	შედეგი	კომენტარი
printf(“ [%d]”, 1234);	[1234]	მინიმალური დასაშვები ველი
printf(“ [%6d]”, 1234);	[1234]	6 პოზიცია, მონაცემი მარჯვენა კიდეში
printf(“ [%-6d]”, 1234);	[1234]	6 პოზიცია, მონაცემი მარცხენა კიდეში
printf(“ [%2d]”, 1234);	[1234]	რიცხვი არ ეტევა გამოყოფილ 2 პოზიციაში, ამიტომ გამოტანის არე ფართოვდება

ბ) ათწილადი რიცხვებისათვის

მაგალითი	შედეგი	კომენტარი
printf(“ [%f]”, 123.45);	[123.450000]	მინიმალური დასაშვები ველი, 6 ნიშანი წილადურ ნაწილში
printf(“ [%9.3d]”, 123.45);	[123.450]	სულ 9 პოზიცია, მათ შორის 3 წილადურ ნაწილში. მონაცემი მარჯვენა კიდეში
printf(“ [%-9.3d]”, 123.45);	[123.450]	სულ 9 პოზიცია, მათ შორის 3 წილადურ ნაწილში., მონაცემი მარცხენა კიდეში
printf(“ [%6.4d]”, 123.45);	[123.4500]	რიცხვი არ ეტევა გამოყოფილ 6 პოზიციაში (4 წილადურ ნაწილში), ამიტომ გამოტანის არე ფართოვდება

ფორმატი **%e** გამოიყენება სამეცნიერო გამოთვლებში ძალიან დიდი ან ძალიან მცირე რიცხვების გამოსატანად, მაგ. ატომის ზომა ან მზემდე მანძილი.

მაგალითი	შედეგი	კომენტარი
printf("[%e]", 123.45);	**[1.234500e+02]**	მინიმალური ველი, მანტისის წილადურ ნაწილში 6 ნიშანი
printf("[%12.3e]", 123.45);	**[1.234e+02]**	სულ 12 პოზიცია, აქიდან 3 მანტისის წილადური ნაწილისათვის, მონაცემი მარჯვენა კიდეში
printf("[%-12.3e]", 123.45);		სულ 12 პოზიცია, აქიდან 3 მანტისის წილადური ნაწილისათვის, მონაცემი მარცხენა კიდეში

პროგრამა: ორი რიცხვის ჯამი

პროგრამა ძირითადად შედგება 4 ნაწილისაგან:

- ცვლადების გამოცხადება;
- საწყისი მონაცემების შეტანა;
- მონაცემთა დამუშავება (გამოთვლები);
- შედეგის გამოტანა

მონაცემთა შეტანამდე სასურველია გამოვიდეს მითითება (წინააღმდეგ შემთხვევაში კომპიუტერი დაელოდება მონაცემების შეტანას, მომხმარებელს კი არ ეცოდინება, რას მოითხოვს მისგან პროგრამა)

```
#include <stdio.h>
 #include  <conio.h>
main()
 { int a, b, c; // ცვლადების გამოცხადება
 printf ( " Enter two integer numbers\n" ); // მითითება შეტანისათვის
scanf ( "%d%d", &a, &b ); // მონაცემების შეტანა
 c = a + b; // გამოთვლა (მინიჭების ოპერატორი)
 printf ( "Rezult: %d + %d = %d \n", a, b, c ); // შედეგის გამოტანა
 getch();
}
```

ნახ. 55 ორი რიცხვის ჯამის ალგორითმის მარეალიზებელი პროგრამული კოდი

გამოთვლებისათვის გამოიყენება მინიჭების ოპერატორი **„=”.**
მაგ. **c = a + b**; // **a** და **b** რიცხვების ჯამი უნდა ჩაიწეროს **c**-ში.

2.4.7. არითმეტიკული გამოსახულება

არითმეტიკული გამოსახულება, რომელიც მინიჭების ოპერატორის მარჯვენა ნაწილში გვხვდება, შეიძლება შეიცავდეს:

- მთელ და წილად რიცხვებს;
- არითმეტიკული ოპერაციების ნიშნებს:
  + - შეკრება,
  - - გამოკლება
  ***** - გამრავლება
  **** - გაყოფა
  % - გაყოფის შედეგად მიღებული ნაშთი

- სტანდარტულ ფუნქციებს:
  abs(i) - მთელი რიცხვის მოდული;
  fabs(x) - წილადი რიცხვის მოდული;
  sqrt(x) - კვადრატული ფესვი;
  pow(x,y) - x რიცხვის y ხარისხში აყვანა.

დავალება:

1.

```
#include <stdio.h>
#include <stdlib.h>
  int main  ( )
{
        int k=10;
 printf("%d*%d=%d\n",k,k,k*k);
        system("PAUSE");
         return 0;
}
```

ა) მოცემული პროგრამა აკრიფეთ თქვენს სისტემაში. გაუშვით კომპილაციაზე და შესრულებაზე.

ბ) კომენტარად აქციეთ სტრიქონი, რომლის საშუალებითაც ახდენთ სათაო ფაილის string-ის ჩართვას
//#include<stdio.h>
გაუშვით კომპილაციაზე. რა მიიღეთ?

გ) მოხსენით კომენტარის სიმბოლოები და კომენტარად აქციეთ ამჯერად სტრიქონი:
int k=10;
რა მიიღეთ?
შეეცადეთ ახსნათ მიღებული მდგომარეობა.
მოხსენით კომენტარის სიმბოლოები

2. დაწერეთ პროგრამული კოდი, რომელიც უზრუნველყოფს ორი წილადი რიცხვის შეტანას და მათი ჯამის პოვნას. შედეგი გამოიტანეთ წილად ნაწილში 3 ციფრის სიზუსტით;

3. დაწერეთ პროგრამული კოდი, რომელიც უზრუნველყოფს ორი, a და b (a>b) მთელი რიცხვის შეტანას. იპოვეთ:

- *a რიცხვიდან კვადრატული ფესვი;*
- *b რიცხვი აიყვანეთ კვადრატში;*
- *a და b რიცხვების განაყოფი;*
- *შედეგები გამოიტანეთ ეკრანზე, ერთ სტრიქონში.*

2.4.8. არითმეტიკული ოპერაციების თავისებურებები

გაყოფის ოპერაციის გამოყენებისას უნდა გვახსოვდეს, რომ ორი მთელი რიცხვის გაყოფის შედეგი არის მთელი რიცხვი. მაგ. 7-ის 4-ზე გაყოფის შედეგი იქნება 1. თუ საჭიროა მივიღოთ წილადი რიცხვი, გასაყოფი ან გამყოფი უნდა გარდავქმნათ წილად რიცხვად.

მაგ.

```
int i, n;
 float x;
i = 7;
 x = i / 4;
/* x-ის მნიშვნელობა იქნება 1, რადგან მთელი რიცხვი იყოფა მთელ რიცხვზე */
x = i / 4.;
/* x-ის მნიშვნელობა იქნება 1.75, მთელი რიცხვი იყოფა წილად რიცხვზე */
x =(float) i / 4;
/* x-ის მნიშვნელობა იქნება 1.75, წილადი  რიცხვი იყოფა მთელ რიცხვზე; */
```

ნახ. 56 სხვადასხვა არითმეტიკული ოპერაციების ამსახველი პროგრამული კოდის ფრაგმენტი

ერთ რიცხვის მეორე რიცხვზე გაყოფისას მიღებული ნაშთის გამოტანისას მინიჭების ოპერაცია ჩაიწერება შემდეგი სახით:

```
nashti = a % b;
```

2.4.9. არითმეტიკული ოპერაციების პრიორიტეტები

არითმეტიკული ოპერაციები სრულდება შემდეგი თანმიმდევრობით:

1. ფრჩხილებს შიგნით მოქცეული მოქმედება
2. ფუნქციის გამოძახება
3. მარცხნიდან მარჯვნივ: გამრავლება, გაყოფა, ნაშთი
4. მარცხნიდან მარჯვნივ: შეკრება და გამოკლება

```
          2   1     5   4       3     8     6   7
x  =  ( a + 5 * b ) * fabs ( c + d ) - ( 3 * b - c );
```

ნახ. 57 არითმეტიკული ოპერაციების შესრულების პრიორიტეტები

გამოსახულება:

$$y = \frac{4x + 5}{(2x - 15z)(3z - 3)} - \frac{5x}{x + z + 3}$$

პროგრამაში ჩაიწერება შემდეგი სახით:

```
y = (4*x + 5) / ((2*x - 15*z)*(3*z - 3)) - 5 * x /
    (x + z + 3);
```

პროგრამირებაში ჩანაწერი: **p=p+3;** ნიშნავს **p** ცვლადის საწყისი მნიშვნელობა გაიზარდოს **3**-ით და ჩაიწეროს **p** ცვლადში.

2.4.10. ინკრემენტი და დეკრემენტი

დაპროგრამების ენაში განსაზღვრულია სპეციალური ოპერატორები ცვლადის ერთით გასაზრდელად (ინკრემენტი) და ერთით შესამცირებლად (დეკრემენტი).

ინკრემენტის ოპერატორი

```
i++;
++i;
```

ეს ოპერატორები ტოლფასია **i=i+1** ჩანაწერის.

დეკრემენტის ოპერატორი

```
i--;
--i;
```

ეს ოპერატორები კი ტოლფასია **i=i-1** ჩანაწერის.

i++ და **++i** ოპერატორებს შორის განსხვავება არის მხოლოდ მაშინ, როცა ისინი გამოყენებულია სხვა ოპერატორებთან ერთად, მაგ. მინიჭების ოპერატორთან

მაგ.

```
int m, k=4, p=4,t;
    m=k++;
     t=++p;
printf("%d",m);
```

პროგრამის ამ ფრაგმენტის შესრულების შედეგად **m** ცვლადის მნიშვნელობა ხდება **4**, ხოლო **t** ცვლადის მნიშვნელობა **5;**

t=++p; - ამ შემთხვევაში **t-** ცვლადს ენიჭება ერთით გაზრდილი **p** ცვლადის მნიშვნელობა;

m=k++; ამ შემთხვევაში **m-** ცვლადს ენიჭება k ცვლადის მნიშვნელობა და შემდეგ ხდება **k** ცვლადის მნიშვნელობის ერთით გაზრდა.

არითმეტიკული გამოსახულებების მოკლე ჩანაწერები

მოკლე ჩანაწერი	სრული ჩანაწერი
x + = a ;	**x = x + a ;**
x - = a ;	**x = x – a;**
x * = a ;	**x = x * a;**
x / = a ;	**x = x / a;**
x % = a ;	**x = x % a;**

2.4.11. ცვლადების ხილვადობის არე და არსებობის დრო

პროგრამაში გამოცხადებულ ყოველ ცვლადს გააჩნია ორი მნიშვნელოვანი მახასიათებელი:

- ✓ არსებობის დრო;
- ✓ ხილვადობის არე.

ეს მახასიათებლები ერთმანეთთან ურთიერთდაკავშირებულია და არსებით გავლენას ახდენენ პროგრამაში ცვლადის გამოყენების შესაძლებლობაზე.

ცვლადის არსებობის დრო შესაძლებელია იყოს ლოკალური და გლობალური.

თუ ცვლადი არსებობს მთელი პროგრამის შესრულების პერიოდში, მაშინ ცვლადს გააჩნია არსებობის გლობალური დრო.

თუ ცვლადი გამოცხადებულია ბლოკში, მაშინ მას გააჩნია არსებობის ლოკალური დრო.

ხილვადობის არე დამოკიდებულია ასევე იმაზე, თუ როგორ არის გამოცხადებული ცვლადი. ცვლადი ხილვადია იმ ბლოკში ან იმ ფაილში, რომელშიც ის არის აღწერილი.

ამ მახასიათებლების მართვა პროგრამისტს ორი სხვადასხვა გზით შეუძლია:

- შეცვალოს ცვლადის გამოცხადების ადგილი პროგრამაში;
- გამოიყენოს მოდიფიკატორები auto, register, static, extern.

ავტომატურ (auto) ცვლადს ან კონსტანტას გააჩნია მოქმედების და ცნობადობის ლოკალური არე იმ ბლოკის შიგნით, რომელშიც ის არის გამოცხადებული. ავტომატური ცვლადისათვის გამოიყოფა დროებითი მეხსიერება ბლოკში შესვლისას და ბლოკიდან გამოსვლისას მეხსიერების ეს ადგილი ითვლება თავისუფლად, ცვლადი ქრება. თუ კლასის სპეციფიკატორი არ არის მითითებული, მაშინ ცვლადი ბლოკი თავისთვად ითვლება ავტომატურად.

რეგისტრული (register) ცვლადი განსხვავდება ავტომატურისაგან მხოლოდ მეხსიერებით, რომელიც გამოიყოფა მის შესანახად. რეგისტრული ცვლადი ინახება პროცესორის რეგისტრში და შესაბამისად დაშვება ამ ცვლადთან გაცილებით სწრაფია, ვიდრე იმ ცვლადთან, რომელიც ინახება ოპერატიულ მეხსიერებაში. თავისუფალი რეგისტრების არარსებობის შემთხვევაში რეგისტრული ცვლადი ხდება ავტომატური.

გარე (extern) ცვლადი წარმოადგენს გლობალურ ცვლადს. ეს სპეციფიკატორი ინფორმაციას აძლევს კომპილატორს, რომ ცვლადი გამოცხადებული იქნება (extern-ის გარეშე) სხვა ფაილში, სადაც მას გამოეყოფა მეხსიერება.

სტატიკური (static) ცვლადისათვის ან კონსტანტისათვის გამოიყოფა მეხსიერება მისი გამოცხადების შემდეგ და ინახება პროგრამის შესრულების ბოლომდე. სტატიკური ცვლადები გამოცხადებისას თავისთავად ინიციალიზირდებიან ნულით (ლოგიკური, მთელი და წილადი) ან ცარიელი მნიშვნელობებით.

2.4.12. განშტოებადი ალგორითმები

მარტივ პროგრამებში ყველა პროგრამა სრულდება ერთმანეთის თანმიმდევრობით (წრფივი ალგორითმი). თუმცა ზოგჯერ საჭიროა შევასრულოთ მოქმედება პირობის შესაბამისად (განშტოებადი ალგორითმი). ამისათვის გამოიყენება პირობითი ოპერატორები.

C-ში არსებობს პირობითი ოპერატორების ორი სახე:

- ოპერატორი **if – else** ორი ვარიანტიდან ერთის ამოსარჩევად;
- მრავლობითი ამორჩევის **switch** ოპერატორი

პირობით ოპერატორს გააჩნია შემდეგი სახე:

```
if    ( პირობა )
{ ... // ბლოკი „თუ“  -
  // ოპერატორები, რომლებიც უნდა შესრულდეს, თუ პირობა ჭეშმარიტია
 }
   else
   { ... // ბლოკი „წინააღმდეგ შემთხვევაში“
     //ოპერატორები, რომლებიც უნდა შესრულდეს, თუ პირობა მცდარია   }
```

თუ ბლოკში **„თუ“** ან **„წინააღმდეგ შემთხვევაში“** არის მხოლოდ ერთი ოპერატორი, მაშინ ფიგურული ფრჩხილები აუცილებელი არ არის.

პირობაში შესაძლებელია გამოვიყენოთ ლოგიკური დამოკიდებულების ნიშნები:

- > < მეტია, ნაკლებია
- >= <= მეტია ან ტოლი, ნაკლებია ან ტოლი
- == ტოლია
- != არ უდრის

C ენაში ნებისმიერი გამოსახულება, რომელიც არ უდრის 0-ს, აღნიშნავს ჭეშმარიტ პირობას, ხოლო ნული - მცდარ პირობას.

თუ ბლოკში „წინააღმდეგ შემთხვევაში“ არ არის საჭირო არანაირი მოქმედების შესრულება, მაშინ მთელი ეს ბლოკი შეგვიძლია საერთოდ გამოვტოვოთ და გამოვიყენოთ პირობითი ოპერა-ტორის შემოკლებული ფორმა:

```
if   ( პირობა )
{ ... // ბლოკი „თუ“  -
  // ოპერატორები, რომლებიც უნდა შესრულდეს, თუ პირობა ჭეშმარიტია
 }
```

ამოცანა: შევიტანოთ კლავიატურიდან ორი განსხვავებული ნამდვილი რიცხვი და განვსაზღვროთ მათ შორის უდიდესი

ამოცანის პირობის მიხედვით უნდა გამოვიტანოთ ორი პასუხიდან ერთ-ერთი: თუ პირველი რიცხვი მეტია მეორეზე, მაშინ პირველი რიცხვი, თუ არა - მაშინ მეორე რიცხვი.

ქვემოთ მოყვანილია ამ ამოცანის გადაწყვეტის ორი ვარიანტი. პირველ ვარიანტში შედეგი გამოტანილია პირდაპირ ეკრანზე, ხოლო მეორე ვარიანტში, უდიდესი რიცხვი იწერება სხვა ცვლადში და შემდეგ ხდება მისი გამოტანა.

ვარიანტი 1

```c
#include <stdio.h>
 #include <conio.h>
    main()
{
float A, B;
printf ("input  A and B :");
scanf ( "%f%f", &A, &B );
 if ( A > B ) { printf ( "max= %f", A );
}
else
{ printf ( "max= %f", B ); }
getch();
}

```

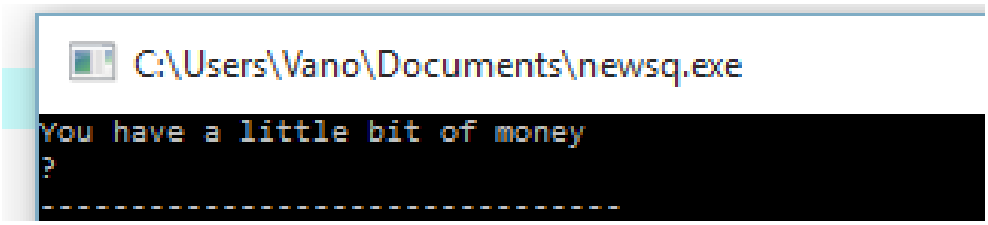
ვარიანტი 2

```c
#include <stdio.h>
 #include <conio.h>


main()
 { float A, B, Max;
printf("input A and B : ");
scanf ( "%f%f", &A, &B );
if ( A > B )
{
Max = A;
}
else
{ Max = B;
 }
 printf ( "max= %f", Max );
getch(); }

```

ბლოკებში **„თუ“** და **„წინააღმდეგ შემთხვევაში“** შესაძლებელია შედიოდეს ნებისმიერი სხვა ოპერატორი, მათ შორის პირობითი ოპერატორიც, ამ შემთხვევაში ოპერატორი **else** ეკუთვნის უახლოეს **if**-ს.

```
6  int main(int argc, char *argv[]) {
7      int A;
8
9      if ( A > 10 )
10      if ( A > 100 )
11            printf ( "You have a large amount of money\n");
12         else
13             printf ("You have enough money\n");
14                 else
15    printf ("You have a little bit of money\n?");
16
17     return 0;
18 }
```

პროგრამის შესრულების შედეგი



რთული პირობა

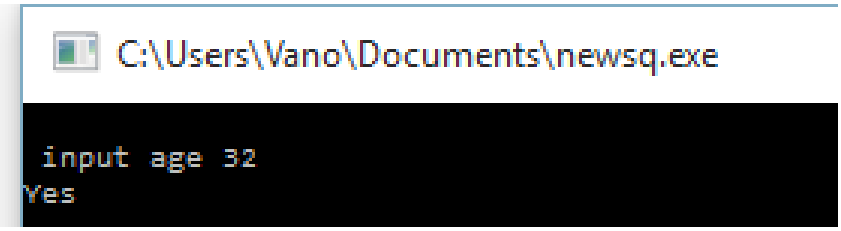
მარტივი პირობა შედგება ერთი დამოკიდებულებისაგან (მეტია, ნაკლებია და ა.შ.). ზოგჯერ საჭიროა პირობა, რომელშიც გაერთიანებულია ორი ან რამდენიმე მარტივი დამოკიდებულება.

მაგ. კომპანიას სჭირდება თანამშრომლები 25-დან 35 წლის ჩათვლით.

ამ შემთხვევაში პროგრამას ექნება სახე:

```c
int main(int argc, char *argv[]) {
    int age;
printf ( "\n input age" );
scanf ( "%d", &age );
if ( 25 <= age && age <= 40 )
 printf ( "Yes");
 else printf ( "No" );
getch();


    return 0;
}
```

პროგრამის შესრულების შედეგი



რთული პირობა შედგება ერთი ან რამდენიმე მარტივი დამოკიდებულებისაგან, რომლებიც ერთიანდებიან ლოგიკური ოპერაციის ნიშნის გამოყენებით.

ოპერაცია **“და”** მოითხოვს ერთდროულად პირობების შესრულებას

პირობა_1 && პირობა_2

ჭეშმარიტობის ცხრილი && ოპერაციის შემთხვევაში

პირობა_1	პირობა_2	პირობა_1 && პირობა_2
მცდარი (0)	მცდარი (0)	მცდარი (0)
მცდარი (0)	ჭეშმარიტი (1)	მცდარი (0)
ჭეშმარიტი (1)	მცდარი (0)	მცდარი (0)
ჭეშმარიტი (1)	ჭეშმარიტი (1)	ჭეშმარიტი (1)

ოპერაცია **„ან“** მოითხოვს ერთ-ერთი პირობის შესრულებას

პირობა_1 ან პირობა_2

ჭეშმარიტობის ცხრილი „ან“ ოპერაციის შემთხვევაში

პირობა_1	პირობა_2	პირობა_1 && პირობა_2
მცდარი (0)	მცდარი (0)	მცდარი (0)
მცდარი (0)	ჭეშმარიტი (1)	ჭეშმარიტი (1)
ჭეშმარიტი (1)	მცდარი (0)	ჭეშმარიტი (1)
ჭეშმარიტი (1)	ჭეშმარიტი (1)	ჭეშმარიტი (1)

რთულ პირობებში ზოგჯერ გამოიყენება ოპერაცია „**არა**“ - უარყოფა **! (პირობა)**
მაგ., შემდეგი ორი პირობა ერთმანეთის ტოლფასია:

A>B **! (A <= B)**

ლოგიკური ოპერაციების შესრულების პრიორიტეტები:

- ოპერაცია ფრჩხილებში,
- ოპერაცია „არა“
- ლოგიკური დამოკიდებულება **>, <, >/, <=, ==, !=**
- ოპერაცია „და“
- ოპერაცია „ან“

2.4.12.1. გადამრთველი switch

თუ ამორჩევა უნდა განხორციელდეს რამდენიმე ვარიანტიდან, შესაძლებელია გამოვიყენოთ რამდენიმე ჩადგმული **if** ოპერატორი, მაგრამ უფრო მოსახერხებელია სპეციალური **switch** ოპერატორის გამოყენება.

switch ოპერატორი შედგება სათაურისაგან და ოპერატორის ტანისაგან, რომელიც ფიგურულ ფრჩხილებშია მოქცეული.

სათაურში გასაღები სიტყვის **switch**-ის შემდეგ მრგვალ ფრჩხილებში მოცემულია ცვლადის სახელი (მთელი ან სიმბოლური ტიპის). ამ ცვლადის მნიშვნელობის მიხედვით ხდება ამორჩევა რამდენიმე ვარიანტს შორის.

ყოველ ვარიანტს შეესაბამება ჭდე **case,** რომლის შემდეგაც მოდის ამ ცვლადის ერთ-ერთი შესაძლო მნიშვნელობა და ორწერტილი; თუ ცვლადის მნიშვნელობა ემთხვევა ერთ-ერთ ჭდეს, მაშინ პროგრამა გადადის ამ ჭდეზე და ასრულებს მის მომდევნო ოპერატორებს.

ოპერატორი **break** გამოიყენება **switch** ოპერატორის სხეულიდან გამოსავლელად.

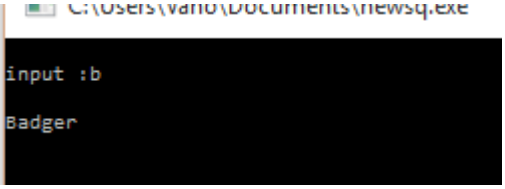
თუ ცვლადის მნიშვნელობა არ ემთხვევა არც ერთ ჭდეს, მაშინ პროგრამა გადადის ჭდეზე **default ;**

შესაძლებელია ერთ ოპერატორზე რამდენიმე ჭდის მითითება.

ამოცანა: შევადგინოთ პროგრამული კოდი, რომელიც უზრუნველყოფს ანბანის სიმბოლოს შეტანას და ამ ასოზე დაწყებული ცხოველის სახელის გამოტანას.

```c
#include <stdio.h>
#include <stdlib.h>



int main(int argc, char *argv[]) {
char c;
printf("\ninput :");
 scanf("%c", &c); // ???????? ???????
switch ( c ) // ????????? ?????????
{ case 'a': printf("\nAntelope");
      break;
case 'b': printf("\nBadger");
      break;
 case 'w': printf("\nWolf");
      break;
default: printf("\n I do not know");
}
 getch();


    return 0;
}
```

პროგრამის შესრულების შედეგი:



დავალება:

1. კლავიატურიდან შევიტანოთ იმ სტუდენტთა შეფასებები (A, B, C, D, E), რომლებმაც ჩააბარეს გამოცდა. თითოეული შეფასებისათვის დაბეჭდეთ ქულების შესაბამისი დიაპაზონი. შეტანა დაამთავრეთ, როდესაც შეგხვდებათ 'q' ან 'Q' სიმბოლო
   A-ს შეესაბამება 91-100 დიაპაზონი;
   B-ს შეესაბამება 81-90 დიაპაზონი;
   C-ს შეესაბამება 71-80 დიაპაზონი;
   D-ს შეესაბამება 61-70 დიაპაზონი;
   E-ს შეესაბამება 51-60 დიაპაზონი.
2. იპოვეთ და დაბეჭდეთ y ცვლადის მნიშვნელობა:

$$y = \begin{cases} x^2 + 1, & x > -2 \\ 3x + 1, & x \le -2 \end{cases}$$

3. შევადგინოთ პროგრამული კოდი, რომელიც უზრუნველყოფს კლავიატურიდან ციფრის შეტანისას, მისი შესაბამისი ტექსტური ინფორმაციის გამოტანას:

2.4.13. ციკლი

ციკლი - ეს არის მოქმედებების თანმიმდევრობა, რომელიც სრულდება რამდენიმეჯერ.

2.4.13.1. ციკლი ბიჯების ცნობილი რაოდენობით (for)

ხშირ შემთხვევაში ჩვენთვის წინასწარ ცნობილია (ან შეგვიძლია განვსაზღვროთ), რამდენჯერ უნდა შევასრულოთ გარკვეული ოპერაცია.

C-ში ციკლის უზრუნველყოფას ახორციელებს რამდენიმე ოპერატორი.

ციკლი for გამოიყენება მაშინ, როცა ოპერაციების რაოდენობა წინასწარ ცნობილია ან შესაძლებელია გამოითვალოს;

for-ის სტრუქტურა შეიძლება გამოვსახოთ შემდეგი სახით:

```
for (გამოსახულება1; გამოსხულება2; გამოსახულება3)
{
//. . . ციკლის ტანი
}
```

გამოსახულება1 ძირითადად გამოიყენება ცვლადის საწყისი მნიშვნელობის განსაზღვრისათვის;

გამოსახულება2 წარმოადგენს გამოსახულებას, რომელშიც გადმოცემულია პირობა, რა პირობითაც უნდა დატრიალდეს ციკლი;

გამოსახულება3 განსაზღვრავს ცვლადის ცვლილებას ყოველი ციკლის შემდეგ.

for ოპერატორის შესრულების თანმიმდევრობა:

1. სრულდება გამოსახულება 1, ანუ ცვლადს მიენიჭება საწყისი მნიშვნელობა;
2. მოწმდება გამოსახულება 2-ში მოცემული პირობა;
3. თუ პირობა ჭეშმარიტია, სრულდება ციკლის ტანი;
4. ცვლადის მნიშვნელობა იცვლება გამოსახულება 3-ის შესაბამისად და ხორციელდება გადასვლა მეორე პუნქტზე. ციკლი დატრიალდება მანამ, სანამ გამოსახულება 2-ში მოცემული პირობა არ გახდება მცდარი. ამის შემდეგ მართვა გადაეცემა for-ის მომდევნო ოპერატორს პროგრამაში.

განვიხილოთ რამდენიმე მარტივი მაგალითი:

ა)

```
#include <stdio.h>
 #include  <conio.h>
main()
{ int i; // ციკლის ცვლადის გამოცხადება
 for ( i = 1; i <= 10; i ++ ) //
 { // ციკლის დასაწყისი
printf("Hello"); // ციკლის ტანი
} // ციკლის დასასრული
```

```
getch(); }
```

ბ)

```
int main()
{ int i,b;
  for (i=1; i<10; i++) b=i*i;
  return 0;       }
```

ამ ფრაგმენტის მიხედვით ხდება 1-დან 10-მდე რიცხვების კვადრატების გამოთვლა.

არსებითია ის, რომ პირობის შემოწმება ხდება ციკლის დასაწყისში. ამიტომ, თუ პირობა არ სრულდება შესაძლებელია ციკლი საერთოდ არ შესრულდეს.

for-ში შესაძლებელია იყოს ერთდროულად რამდენიმე ოპერატორი, გამოყოფილი ერთმანეთისაგან მძიმის საშუალებით. მაგ. **for (i = 0, x = 1.; i < 10; i += 2, x *= 0.1){ ... }**

ციკლის ტანი ძირითადად მოთავსებულია ფიგურულ ფრჩხილებში.

თუ ციკლის ტანში არის მხოლოდ ერთი ოპერატორი, ფიგურული ფრჩხილები შეიძლება არც გამოვიყენოთ.

ციკლის ტანში შესაძლებელია შედიოდეს ნებისმიერი სხვა ოპერატორი, მათ შორის სხვა ციკლიც (ჩადგმული ციკლები).

განვიხილოთ მაგალითი, როდესაც ციკლის მართვისათვის გამოიყენება ერთდროულად ორი ცვლადი.

მაგ.

```
int main()
{  int top,  bot;
   char string[100],  temp;
   for ( top=0, bot=100 ; top < bot ; top++, bot--)
   {  temp=string[top];
      string[bot]=temp;
   }
   return 0;       }
```

ამ მაგალითში ციკლის მართვისათვის გამოიყენება ერთდროულად ორი ცვლადი top და bot. გამოსახულება1-ის და გამოსახულება3-ის პოზიციებზე გამოყენებულია ერთდროულად რამდენიმე გამოსახულება, რომლებიც ერთმანეთისაგან გამოყოფილია მძიმით და სრულდება მიმდევრობით.

for-ის გამოყენების კიდევ ერთ ვარიანტს წარმოადგენს უსასრულო ციკლი. ასეთი ციკლის ორგანიზაციისათვის შესაძლებელია გამოვიყენოთ ცარიელი პირობითი გამოსახულება, ხოლო ციკლიდან გამოსვლისათვის გამოიყენება დამატებითი პირობა ან ოპერატორი break.

მაგ.

```
for (;;)
{ ...
  ... break;
  ...
}
```

როგორც განხილული მაგალითიდან ჩანს, სინტაქსი საშუალებას გვაძლევს, რომ ოპერატორი იყოს ცარიელი.

განვიხილოთ რამდენიმე ალგორითმის პროგრამული რეალიზაციები for ოპერატორის გამოყენებით.

ა) პროგრამა უზრუნველყოფს 1-დან 10-მდე რიცხვების კვადრატების გამოტანას სვეტის სახით.

merlin.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
int i;
    for (i=1;i<=10;i++)
printf("%d\t%d\n",i,i*i);


    return 0;
}
```

შესრულების შედეგი:

```
C:\Dev-Cpp\for_c.exe
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
Press any key to continue . .
```

გ)

```
4  int main(int argc, char *argv[]) {
5
6
7  int i, j;
8      for (i=1,j=10;i<=j;i++,j--)
9      {
10 printf("i+j=%d+%d=%d\n",i,j,i+j);
11  }
12
13     return 0;
14 }
```

შესრულების შედეგი:

```
C:\Dev-Cpp\for_c.exe
i+j=1+10=11
i+j=2+9=11
i+j=3+8=11
i+j=4+7=11
i+j=5+6=11
Press any key to continue . . .
```

2.4.13.2. ციკლი პირობით (while)

ხშირ შემთხვევაში შეუძლებელია დავადგინოთ, რამდენჯერ უნდა შესრულდეს ოპერაცია, თუმცა შესაძლებელია განისაზღვროს პირობა, როდესაც ის უნდა დასრულდეს.

ყველაზე მარტივი ციკლის ორგანიზება ხდება **while** ციკლის ოპერატორის გამოყენებით:

while (ცვლადი ან პირობა)

{

მოქმედება, რომელიც უნდა შესრულდეს

}

აქ ციკლის ოპერატორი მუშაობს მანამ, სანამ ცვლადი ან პირობა, რომელიც ოპერატორის ფრჩხილებშია მოქცეული არ მიიღებს მნიშვნელობას **false.**

განვიხილოთ რამდენიმე მაგალითი:

1***)***

```
 int a=0;
int b=100;
while(a < b)
{a++;
}
```

ციკლის დასაწყისში მოწმდება პირობა (a < b) და თუ პირობა სრულდება, ციკლიც გრძელდება. რაც გამოიხატება იმაში, რომ ფიგურულ ფრჩხილებში მოთავსებული მოქმედება შესრულდება, ანუ ცვლადის მნიშვნელობა გაიზრდება 1-ით. ასე გაგრძელდება მანამ, სანამ ოპერატორში მოცემული პირობა არ დაირღვევა; მთვლელის დაყენება იმით არის ნაკარნახევი, რომ თუ არ იქნება მთვლელი, მაშინ ციკლი უსასრულოდ გაგრძელდება.

2) შევიტანოთ მთელი რიცხვი და განვსაზღვროთ რამდენი ციფრისგან შედგება ეს რიცხვი.

ამოცანის გადასაწყვეტად შემდეგი ალგორითმი გამოიყენება : რიცხვი უნდა გავყოთ 10-ზე (გაყოფის შედეგად ხდება ნაშთის უგულებელყოფა), ეს ოპერაცია გრძელდება მანამ, სანამ გაყოფის შედეგი არ იქნება 0.

სპეციალური ცვლადის გამოყენებით, რომელსაც მთვლელი ეწოდება, ვითვლით რამდენჯერ შესრულდა გაყოფა. გაყოფის ოპერაციების რაოდენობა რიცხვში ციფრების რაოდენობის ტოლი იქნება. ცხადია, რომ წინასწარ ვერ განვსაზღვრავთ რამდენჯერ შესრულდება გაყოფის ოპერაცია.

```
#include <stdio.h>
#include <stdlib.h>
 main()
 { int N; // ცვლადი, რომლეშიც შევიტანთ მოცემულ რიცხვს
 int count=0; // ცვლადი-მთვლელი
 printf ( "\n input N: " ); // მინიშნება
 scanf ( "%d", &N ); // N –ის შტანა კლავიატურიდან
while ( N > 0 ) // ციკლის პირობა  «სანამ N > 0»
{ // ციკლის დასაწყისი
 N /= 10; // ნაშთის უგულებელყოფა
 count ++; // ციფრების მთვლელის 1-ით გაზრდა
} // ციკლის დასასრული
 printf ( " In this number %d digit \n", count );
getch();
 }
```

3) ეკრანზე გამოვიტანოთ 10-დან 20-მდე რიცხვების კვადრატები. ეკრანზე შედეგი გამოვიდეს შემდეგი სახით, მაგ. 10*10=100

11*11=121

და ა.შ.

ყოველი რიცხვის კვადრატი გამოვიდეს ახალ სტრიქონზე.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

int k=10;
 while (k<=20)
 {
        printf("%d*%d=%d\n",k,k,k*k);
        k++;
        }

    return 0;
}
```

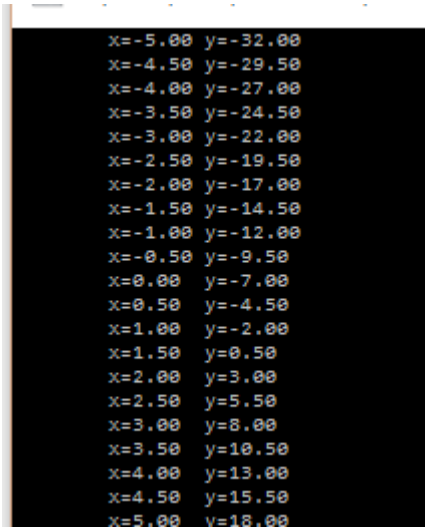
პროგრამის შესრულების შედეგი:

```
10*10=100
11*11=121
12*12=144
13*13=169
14*14=196
15*15=225
16*16=256
17*17=289
18*18=324
19*19=361
20*20=400

--------------------------------
Process exited after 0.0201 s
Press any key to continue . .
```

4) გამოვთვალოთ y=5*x-7 ფუნქციის მნიშვნელობა (-5;10) შუალედში 0.5-ის ტოლი ბიჯით.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  float x,y;
  x=-5;

  while (x<=10)
  {
     y=5*x-7;
      printf("\tx=%.2f\ty=%.2f\n",x,y);
      x=x+0.5;
        }


    return 0;
}
```

შესრულების შედეგი:



5) გამოვთვალოთ y ფუნქციის მნიშვნელობა (-5;10) შუალედში 1-ის ტოლი ბიჯით, შემდეგი პირობით:

თუ x<0 y=5*x-7, ყველა სხვა შემთხვევაში y=x+40.

```c
#include <stdio.h>
#include <stdlib.h>

/* run this program using the console pauser

int main(int argc, char *argv[]) {

    float x,y;
  x=-5;

  while (x<=10)
  {  if (x<0)
     y=5*x-7;
       else
       y=x+40;
      printf("\tx=%.2f\ty=%.2f\n",x,y);
      x=x+1;
        }


    return 0;
}
```

შესრულების შედეგი:

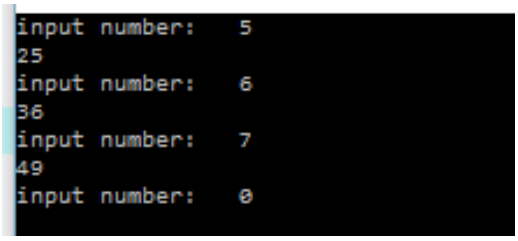
```
x=-5.00 y=-32.00
x=-4.00 y=-27.00
x=-3.00 y=-22.00
x=-2.00 y=-17.00
x=-1.00 y=-12.00
x=0.00  y=40.00
x=1.00  y=41.00
x=2.00  y=42.00
x=3.00  y=43.00
x=4.00  y=44.00
x=5.00  y=45.00
x=6.00  y=46.00
x=7.00  y=47.00
x=8.00  y=48.00
x=9.00  y=49.00
x=10.00 y=50.00
```

6) ვიპოვოთ შეტანილი რიცხვის კვადრატი, გამოთვლები განხორციელდეს მანამ, სანამ არ შევიტანთ 0-ს.

newsdaq.c

```c
#include <stdio.h>
#include <stdlib.h>

/* run this program using the console pa

int main(int argc, char *argv[]) {

int k;
  printf("input number:\t");
  scanf("%d", &k);
    while (k != 0)
    {
        printf("%d\n",k*k);
        printf("input number:\t");
         scanf("%d", &k);
          }


    return 0;
}
```

შესრულების შედეგი:



7) ვიპოვოთ შეტანილი რიცხვის გამყოფები

newsdaq.c

```c
#include <stdio.h>
#include <stdlib.h>

/* run this program using the console p

int main(int argc, char *argv[]) {
    int p;
    printf("input number:\t");
    scanf("%d", &p);
    int k=2;

    while (k<=p/2)
    {  if (p % k ==0)
         printf("%d\t", k);
          k++;
    }
    printf("%d\t", p);


    return 0;
}
```

შესრულების შედეგი:

```
input number:   54
2       3       6       9       18      27      54
--------------------------------
```

2.4.13.3. ციკლი do-while

do-while ციკლს ბოლო პირობიან ციკლის ოპერატორს უწოდებენ.

ოპერატორის ფორმატი:

```
 do
{
მოქმედებები
}
while ( /გამოსახულება1 / ) ;
```

გამოსახულება 1-ში უნდა მივუთითოთ ის პირობა, რომელიც უზრუნველყოფს ციკლის დატრიალებას;

ოპერატორის შესრულების თანმიმდევრობა:

პირობა მოწმდება ციკლის შესრულების შემდეგ;

ციკლის ტანი ნებისმიერ შემთხვევაში სრულდება ერთხელ მაინც, რამაც შეიძლება შეცდომამდე მიგვიყვანოს;

ასეთი კონსტრუქცია გამოიყენება მხოლოდ იმ შემთხვევაში, თუ ის ნამდვილად ამარტივებს შესასრულებელი ამოცანის ალგორითმს.

მაგალითი. პროგრამა უზრუნველყოფს შეტანილი რიცხვების შეკრებას. შეტანა ხორციელდება მანამ, სანამ არ შეიტანთ მნიშვნელობას 0-ს.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int sum = 0 , n ;
 do
 {
 printf("input n :\t");
 scanf("%d", &n);
 sum += n ;
 }
 while (n != 0 ) ;
printf("Sum:  %d\n", sum);


    return 0;
}
```

```
input n :       5
input n :       4
input n :       3
input n :       7
input n :       0
Sum:  19
```

დავალება:

1. იპოვეთ $y=8x^2+7$ ფუნქციის მნიშვნელობები (-1;-4) შუალედში 1-ის ბიჯით. ა) გამოიყენეთ for ოპერატორი; ბ) გამოიყენეთ while ოპერატორი;
2. ციკლის ოპერატორის გამოყენებით იპოვეთ:

- a-დან b-მდე რიცხვების ჯამი და ნამრავლი;
- 20-დან 40-მდე ყველა ლუწი რიცხვის ჯამი;
- 1-დან 100-მდე 3-ის ჯერადი რიცხვების ჯამი და რაოდენობა;

2.5. მიმთითებლები

მონაცემთა მისამართებთან სამუშაოდ C საკმაოდ მდიდარ შესაძლებლობებს გვთავაზობს. ყოველ ცვლადს და მუდმივას აუცილებლად გააჩნია სამი მახასიათებელი: სახელი, მნიშვნელობა და მისამართი. x ცვლადის მისამართს წარმოადგენს &x სიდიდე. საჭიროების შემთხვევაში მისი გამოტანა შეიძლება %d ან %p ფორმატით. სინამდვილეში, &x ოპერაცია (x –ის მისამართის აღება) გაცილებით ინფორმატიულია, რადგან მონაცემის მისამართი ინახავს ინფორმაციას მონაცემის ტიპის შესახებ. რადგან ეს საკითხი უაღრესად მნიშვნელოვანია და თანაბრად ეხება როგორც პროგრამულ, ასევე აპარატულ ასპექტს, ამიტომ მოკლედ გავეცნოთ კომპიუტერში მონაცემების დამისამართების ყველაზე გავრცელებულ სქემებს. x - სთვის ყველაზე ბუნებრივი მოდელი თითოეული სიმბოლოს (ბაიტის) დამისამართების საშუალებას იძლევა. ამას ეწოდება დამისამართება ბაიტების მიხედვით. მეხსიერებაში უფრო მსხვილი მონაცემის (მაგალითად, მთელი რიცხვის ან ათწილადის) მიერ დაკავებული ადგილის მისამართი, როგორც წესი, ემთხვევა მისი პირველი სიმბოლოს მისამართს.

ძალიან ხშირად აუცილებელია მონაცემის მისამართის დამახსოვრება. ეს ნიშნავს, რომ

თვითონ მისამართი უნდა განვიხილოთ მონაცემად. ეს სრულიად ბუნებრივია, რადგან, მისამართი ინახავს მონაცემის ტიპს. ფორმალურად, ამ მიზნით შემოდის მიმთითებლის (პოინტერის) ტიპი, რაც სხვა არაფერია, თუ არა კონკრეტული ტიპის მონაცემის მისამართი. პრაქტიკულად, მიმთითებელი ძალიან ამარტივებს ბევრ საკითხს, მათ შორის აღნიშვნებსაც.

როგორც აღვნიშნეთ, C ენაში მიმთითითებლები იგივეა რაც ცვლადები, იმ განსხვავებით, რომ ეს მიმთითითებლები **(Pointer)** ინახავენ ცვლადის, მასივის, ფუნქციის და ა.შ. მისამართს.

როგორც ვიცით, ყოველი გამოცხადებული ცვლადი, ოპერატიულ მეხსიერებაში იკავებს გარკვეულ ადგილს და გააჩნია შესაბამისი მისამართი, რომელიც უზრუნველყოფს წვდომას ამ ცვლადზე.

C ენაში მიმთითითებლიანი ცვლადის ზოგადი სტრუქტურა ასეთია:

<ცვლადი ტიპი> * ცვლადის სახელი;

განვიხილოთ მაგალითები:

გამოვაცხადოთ მთელი ტიპის ცვლადი **myvar;**

int myvar;

ამ ბრძანების შედეგად შეიქმნება ცვლადი, რომელიც განთავსდება ოპერატიულ მეხსიერებაში და მიენიჭება შესაბამისი მისამართი.

Int *Foo;

ამ ბრძანების შედეგად შეიქმნება მიმთითებელი Foo. მიმთითებელი ინახავს მისამართს, ე.ი. ***Foo**-ც შეინახავს მისამართს. **Foo** მიმთითებელში შევინახოთ **myvar** ცვლადის მისამართი:

Foo = &myvar;

& (ამპერსენდი) ნიშანი აღნიშნავს მისამართს.

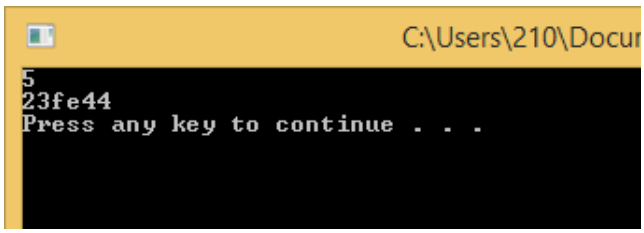
განვიხილოთ რამდენიმე მარტივი პროგრამა და გავაანალიზოთ შესრულების შედეგები:

1)

mer.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* run this program using the console p
5
6  int main(int argc, char *argv[]) {
7
8      int myvar;
9      int *Foo;
10
11     myvar=5;
12     Foo=&myvar;
13     printf("%d\n",myvar);
14     printf("%x\n",Foo);
15
16
17     system("pause");
18     return 0;
19 }
```

შესრულების შედეგი:



როგორც შესრულების შედეგიდან ჩანს, myvar ცვლადის მნისვნელობაა 5, ხოლო Foo-ში ჩაწერილია myvar ცვლადის მისამართი. კოდის მე- 12 სტრიქონიდან ჩანს, რომ Foo-ში იწერება myvar ცვლადის მისამართი.

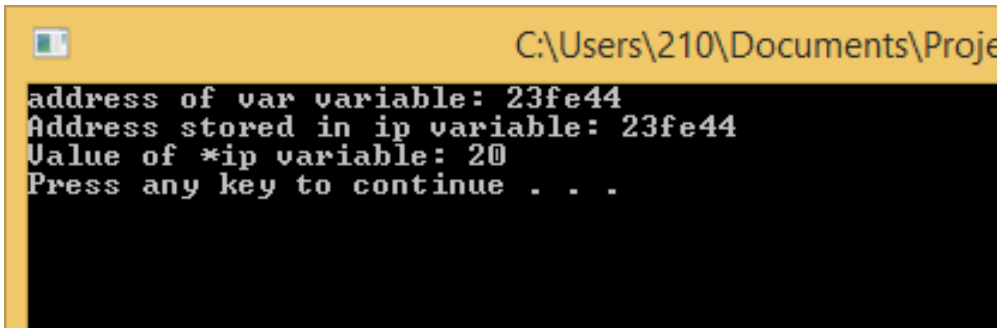
სპეციფიკატორი **"%x"** უზრუნველყოფს თექვსმეტობით სისტემაში რიცხვის გამოტანას. **%X** იყენებს მაღალი რეგისტრის ასოებს **(ABCDEF).** , ხოლო %x დაბალი რეგისტრის ასოებს **(abcdef).**

%p სპეციფიკატორი გამოიყენება მიმთითებლის გამოსატანად. გამოტანა შესაძლებელია განსხვავებული იყოს კომპილატორის და პლატფორმის მიხედვით.

2)

```
 4  /* run this program using the console pauser or add y
 5
 6  int main(int argc, char *argv[]) {
 7
 8  int var = 20;
 9  int *ip;
10  ip = &var;
11  printf("address of var variable: %x\n", &var);
12  //address stored in pointer variable
13  printf("Address stored in ip variable: %x\n",ip);
14  // access the value using the pointer
15  printf ("Value of *ip variable: %d\n", *ip);
16
17  system("pause");
18      return 0;
19  }
```

შესრულების შედეგი:

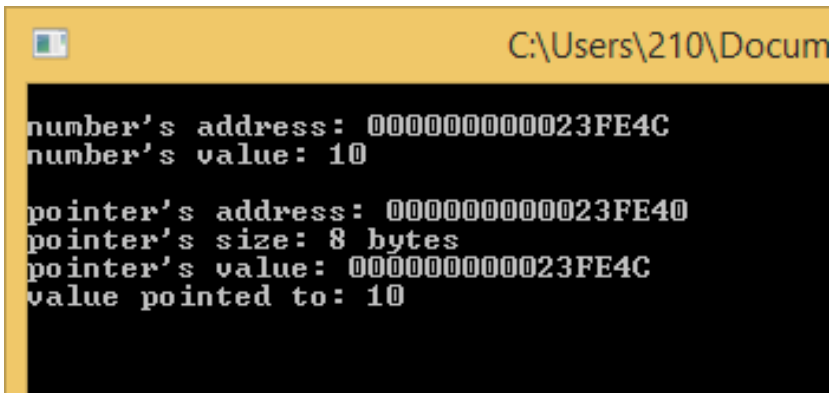


3)

mer.c Makefile.win

```c
#include <stdio.h>
#include <conio.h>
int main()
{int number = 0;
int *pointer = NULL;
 number = 10;
 printf("\nnumber's address: %p", &number);
   printf("\nnumber's value: %d\n\n", number);
     pointer = &number;
        printf("pointer's address: %p", &pointer);

        printf("\npointer's size: %d bytes", sizeof(pointer));
        printf("\npointer's value: %p", pointer);
         printf("\nvalue pointed to: %d\n", *pointer);
         getch();
         return 0;
}
```

შესრულების შედეგი:



sizeof ბრძანებას გამოაქვს მიმთითებლის მიერ დაკავებული მეხსიერების ზომა ბაიტებში.

2.6. მონაცემთა სტრუქტურებთან მუშაობა

2.6.1. მასივი

საბაზო ტიპების საფუძველზე შესაძლებელია შეიქმნას მონაცემთა ახალი ტიპები და მონაცემთა სტრუქტურები. მასივი არის ერთი და იგივე ტიპის მონაცემთა ერთობლიობა, რომელთაც გააჩნიათ ერთი სახელი.

წინა თემებიდან ოპერატორის მეშვეობით შეგვიძლია გამოვაცხადოთ ცვლადი, რომელიც მეხსიერებაში რაღაც ადგილს დაიკავებს, ყოველი შემდგომი ცვლადის გამოცხადებისას ჩვენ არა გვაქვს გარანტია, რომ ცვლადები რომლებიც გამოვაცხადეთ, ერთმანეთის მიყოლებით განლაგდებიან მეხსიერებაში და თუკი დაგვჭირდება მეხსიერებიდან ამ ცვლადების წაკითხვა ჩვენ სახელებით უნდა მივმართოთ მათ, სხვანაირად მათი მოძებნა შეუძლებელია. მასივი ამ მხრივ უნიკალურია, რადგანაც ყოველი ცვლადი, რომელიც მასივში იქნება, გარანტირებულად ერთმანეთის მიყოლებით იქნება მეხსიერებაში ჩაწერილი და მეხსიერებიდან შეიძლება ასევე მიმდევრობით წაკითხვა.

მასივი შეიძლება არსებობდეს ნებისმიერი ტიპის:

მაგ. **int mas [15];** // **int** არის ცვლადის ტიპი, რომელიც განსაზღვრავს, რომ მასივის თითოეულ წევრს ექნება 4 ბაიტის სიდიდე. აქ გამოყოფილი არის 15 წევრიანი მასივი (ანუ 15 არის მასივის სიგრძე) და თითოეული წევრის სიდიდე ოთხი ბაიტია;

როგორც ჩანს, მასივი შეიძლება განვსაზღვროთ, როგორც მოკლედ წარმოდგენილი ცვლადების ერთობლიობა, რომლებიც ინდექსირდებიან, თითოეულ მათგანზე მიმართვისათვის საკმარისია მივუთითოთ მასივის სახელი და ინდექსი კვადრატულ ფრჩხილებში. რადგან მასივის ელემენტები ჩვეულებრივი ცვლადებია, ამიტომ შესაძლებელია მათი მნიშვნელობების შეცვლა კონსოლიდანაც, ანუ შეგვიძლია მასივის რომელიმე ელემენტს მივანიჭოთ ნებისმიერი მნიშვნელობა, ისევე, როგორც ამას ვაკეთებდით ჩვეულებრივი ცვლადებისათვის.

მაგალითად ჩვენ მიერ გამოცხადებული მასივისათვის, ელემენტების ინიციალიზაცია შესაძლებელია შემდეგი სახით:

```
mas[2] = 108;
mas[0]=105;
masivi [3]=20;
masivi [9]=43;
masivi [15]=34; //ERROR!!!
```

უნდა შევნიშნოთ, რომ მასივის ელემენტების გადანომვრის ათვლა ხდება 0-დან, შესაბამისად ჩვენ ვერ გვექნება ელემენტი, რომლის ინდექსი იქნება 15.

მასივის გამოცხადების ფორმატი:

```
ტიპი დასახელება [კონსტანტა-გამოსახულება];
ტიპი დასახელება [];
```

დასახელება - ეს არის მასივის იდენტიფიკატორი.

კონსტანტა-გამოსახულება განსაზღვრავს მასივის ელემენტების რაოდენობას.

კონსტანტა-გამოსახულება მასივის გამოცხადებისას შესაძლებელია არ იყოს შემდეგ შემთხვევებში:

- თუ გამოცხადებისას მასივი ინიციალიზირებულია;

- მასივი გამოცხადებულია, როგორც ფუნქციის ფორმალური პარამეტრი;

- მასივი გამოცხადებულია, როგორც მიმმართველი მასივზე.

ერთგანზომილებიანი მასივის გამოცხადება შეიცავს ერთ კონსტანტა-გამოსახულებას; ორგანზომილებიანი - ორს, სამგანზომილებიანი - სამს და ა.შ.

```
int a[2][3];      /*   წარმოდგენილია მატრიცული სახით
                          a[0][0]  a[0][1] a[0][2]
                          a[1][0]  a[1][1] a[1][2]     */
double b[10];        /* double ტიპის 10 ელემენტისგან შემდგარი ვექტორი */
```

მასივის ელემენტების შეტანის ორგანიზაცია

მასივის ელემენტების შეტანა ხორციელდება ციკლის ოპერატორის გამოყენებით. ქვემოთ მოცემულ ფრაგმენტში განსაზღვრულია 10 ელემენტიანი მთელი ტიპის მასივი, სახელით **mas.**

```
int mas[10];
for(i=0;i<10;i++)
scanf("%d",&mas[i]);
```

მასივის ელემენტების გამოტანის ორგანიზაცია

```
for(i=0;i<10;i++)
printf ("%d\t",mas[i]);
```

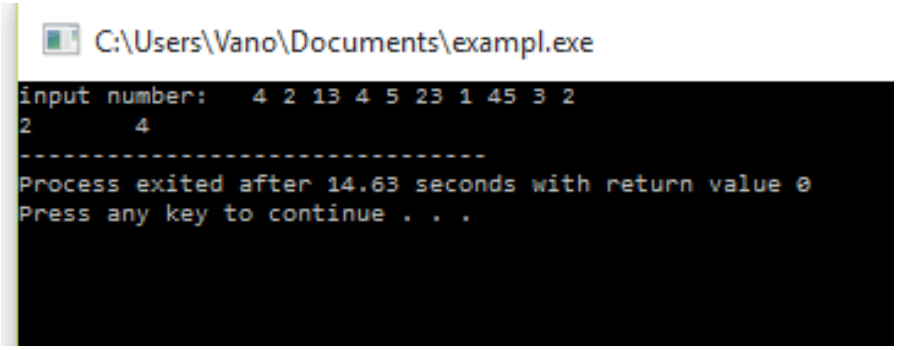
განვიხილოთ მასივებთან დაკავშირებული რამდენიმე ამოცანა :

1. **მოცემულია მასივი. იპოვეთ მასივში კენტი რიცხვების ჯამი და რაოდენობა.**

[*] newsdaq.c

```c
#include <stdio.h>
#include <stdlib.h>

/* run this program using the console pauser or add your

int main(int argc, char *argv[]) {
    int mas[10];
 int i,jami=0,raodenoba=0;

 for(i=0;i<10;i++)

 { printf ("m[%d]=\t",i);
  scanf("%d",&mas[i]);
     }

 for(i=0;i<10;i++)
 if(mas[i]%2==1)
 {
raodenoba++;
 jami=jami+mas[i];
                                    }
  printf ("raodenoba=%d\n jami=%d\n",raodenoba,jami);

    return 0;
}
```

ეს შედეგი არასწორია



2. მოცემულია რიცხვების თანმიმდევრობა. დათვალეთ ამ მიმდევრობაში რამდენჯერ შეიცვალა რიცხვების ნიშანი.

newsdaq.c

```c
#include <stdio.h>
#include <stdlib.h>

/* run this program using the console pauser or add

int main(int argc, char *argv[]) {
    int m[5];
int i;
for (i=0; i<5; i++)
{printf("m[%d]=\t",i);
scanf("%d",&m[i]);
}
int raod=0;
for (i=0; i<4; i++)
if(m[i]<=0  && m[i+1]>=0 || m[i]>=0 && m[i+1]<=0)
raod++;
printf("raodenoba=%d\n",raod);


    return 0;
}
```

```
m[0]=   5
m[1]=   3
m[2]=   -4
m[3]=   3
m[4]=   5
raodenoba=2
```

3. მოცემულია კვირის განმავლობაში სტუდენტთა ლექციაზე დასწრების შედეგები. არსებული მონაცემების მიხედვით იპოვეთ:
ა) სტუდენტთა საშუალო დასწრების რიცხვითი მნიშვნელობა;
ბ) იყო თუ არა შემთხვევა, როცა მთელი ჯგუფი არ გამოცხადდა ლექციაზე;

[*] newsdaq.c

```c
#include <stdio.h>
#include <stdlib.h>

/* run this program using the console pauser or

int main(int argc, char *argv[]) {
int m[5];
int i;
for (i=0; i<5; i++)
{printf("m[%d]=\t",i);
scanf("%d",&m[i]);
}
int jami=0,sashualo;
for (i=0; i<5; i++)
jami=jami+m[i];
sashualo=jami/6;

printf("sashualo=%d\n",sashualo);

for (i=0; i<5; i++)
if (m[i]==0)
printf("leqcia gacda me-%d  dges\n",i+1);

    return 0;
}
```

შესრულების შედეგი:

```
m[0]=   3
m[1]=   5
m[2]=   0
m[3]=   23
m[4]=   12
sashualo=7
leqcia gacda me-3  dges
```

4. იპოვეთ მასივის მაქსიმალური ელემენტი და მისი რიგითი ნომერი (ინდექსი);

newsdaq.c

```c
#include <stdio.h>
#include <stdlib.h>

/* run this program using the consol

int main(int argc, char *argv[]) {
int a[6];
int i;
for (i=0; i<6; i++)
scanf("%d",&a[i]);

int maxa=a[0], ind=0;

for (i=1; i<6; i++)

if (a[i]>maxa)
  { maxa=a[i];
    ind=i;
   }

  printf("max = %d\n",maxa);
  printf("N = %d\n",ind);


    return 0;
}
```

შესრულების შედეგი:

```
5
34
2
-4
36
4
max = 36
N = 4
```

დავალება:

1. მოცემულია რიცხვების თანმიმდევრობა. დათვალეთ ამ მიმდევრობაში დადებითი ელემენტების რაოდენობა და ჯამი;

2. მოცემულია რიცხვების თანმიმდევრობა. იპოვეთ რამდენი მეზობელი ერთმანეთის ტოლი რიცხვი არსებობს?

3. მოცემულია მატრიცა. იპოვეთ მინიმალური ელემენტი და მისი მდებარეობა. თუ ასეთი რიცხვი რამდენიმეა, მაშინ გამოიტანეთ ყველა ამ რიცხვის მდებარეობა (სტრიქონისა და სვეტის ნომერი)

მოცემულია კვირის განმავლობაში სტუდენტთა ლექციაზე დასწრების შედეგები. არსებული მონაცემების მიხედვით დაადგინეთ:

ა) სტუდენტთა დასწრების მაქსიმალური რიცხვითი მნიშვნელობა;

ბ) იყო თუ არა შემთხვევა, როცა ლექციაზე გამოცხადდა 5-ზე ნაკლები სტუდენტი?

2.6.2. სტრუქტურა

სტრუქტურა წარმოადგენს შედგენილ მონაცემთა ტიპს, რომელიც აიგება სხვა ტიპების საფუძველზე. მაშასადამე, სტრუქტურა არის სხვადასხვა ტიპის მონაცემთა ერთობლიობა. სტრუქტურაში გაერთიანებულ ცვლადებს სტრუქტურის ელემენტები, წევრები ან ველები ეწოდებათ. სტრუქტურაზე განაცხადი იწყება **struct** მომსახურე სიტყვით, შემდეგ მოიცემა სტრუქტურის დასახელება და ფიგურულ ფრჩხილებში მოთავსებული სტრუქტურის ელემენტები. მაგალითად, სტრუქტურაზე განაცხადი

```
sruct date
        { int day;
          int month;
          int year;
        };
```

აღწერს მონაცემთა ახალ ტიპს. სტრუქტურის დასახელებას – **date** – უწოდებენ სტრუქტურის ტეგს. **C**-ში სტრუქტურის ტეგი არის *ახალი ტიპის სახელი*, რომელიც გამოიყენება ამ ტიპის ცვლადის განაცხადში.

ჩვენს შემთხვევაში განსაზღვრულია მონაცემთა ახალი ტიპი **date.**

`struct date Days;` - ამ განაცხადის მიხედვით ცვლადი `Days` არის `date` ტიპის.

თუ მოცემულია სტრუქტურის ტიპის ცვლადი, მის ელემენტებზე გადასვლა სრულდება წერტილის ოპერატორის გამოყენებით, მაგალითად

Days. day=20;

Days.month=12;

Days.year=2000;

შესაძლებელია სტრუქტურაში სტრუქტურის განთავსება, მაგ.

```
struct man { char name[20], fam[20];
             struct date bd;
             int age;
           };
```

data ტიპი მოიცავს სამ ელემენტს: **day, month, year**, რომლებიც შეიცავენ მთელ მნიშვნელობებს **(int). man** სტრუქტურა მოიცავს ელემნტებს: **name, fam, bd** და **age.** პირველი ორი - **name[20]** და **fam[20]** - არის 20 ელემენტიანი სიმბოლური მასივები. ცვლადი bd წარმოდგენილია **data** შედგენილი ელემენტით (ჩადგმული სტრუქტურა). **age** ელემენტი შეიცავს მთელი ტიპის მნიშვნელობებს. წარმოებული ტიპის შესაბამისი ცვლადის განსაზღვრა შესაძლებელია შემდეგი სახით:

```
struct man man_[100];
```

ამ განსაზღვრით მოცემულია მასივი **man,** რომელიც შედგება **man** ტიპის **100** სტრუქტურისაგან.

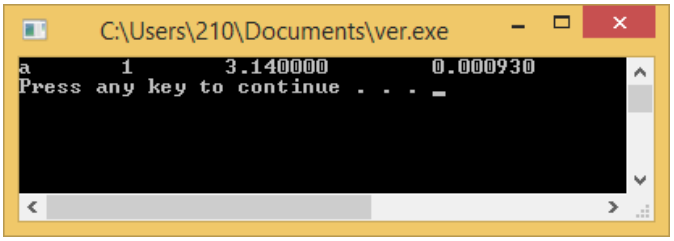
სტრუქტურის ცალკეულ ელემენტისთვის მიმართვა შესაძლებელია შემდეგი სახით:

```
man_[j].age = 19;
man_[j].bd.day = 24;
man_[j].bd.month = 2
man_[j].bd.year = 1987;
```

განვიხილოთ საილუსტრაციო მაგალითები:

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  struct Structure1 {
6    char c;
7    int i;
8    float f;
9    double d;
10 };
11 int main()
12 {
13     struct Structure1 s;
14    s.c = 'a';
15    s.i = 1;
16    s.f = 3.14;
17    s.d = 0.00093;
18   printf("%c\t%d\t%f\t%f\n", s.c,s.i,s.f,s.d);
19       system("pause");
20    return 0;
21 }
22
23
```

პროგრამის შესრულების შედეგი:



2.7. სამომხმარებლო ფუნქციების შემუშავება

ძალიან ხშირად პროგრამის გამართვის გასამარტივებლად აუცილებელი ხდება დიდი მოცულობის პროგრამის პატარა ნაწილებად დაყოფა. რეალური პროგრამები ათასობით სტრიქონისგან შედგება და შეუძლებელიცაა ასეთი პროგრამის შექმნა ნაწილებად დაყოფის გარეშე.

C საშუალებას იძლევა დავყოთ პროგრამის კოდი ნაწილებად, რომელთაც ფუნქციები ეწოდება. ფუნქციის განსაზღვრას გააჩნია შემდეგი სახე:

```
<დასაბრუნებელი მნიშვნელობის ტიპი> <ფუნქციის სახელი> (<ფუნქციის არგუმენტები>)
{
        return <გამოსახულება>;
}
```

შენიშვნა: ფუნქცია შესაძლებელია იყოს არგუმენტის გარეშე და არგუმენტით.

ფუნქციების ჩაწერა და გამართვა შესაძლებელია პროგრამის დანარჩენი კოდის გარეშე.

ფუნქციის არგუმენტები ეწოდება იმ მნიშვნელობებს, რომელიც შესაძლებელია გადაეცეს მას გამოძახების დროს.

დასაბრუნებლი მნიშვნელობა მიუთითებს შედეგს, რომელსაც აბრუნებს ფუნქცია მუშაობის დამთავრების შედეგად.

ფუნქციაში არგუმენტები და დასაბრუნებელი მნიშვნელობა არ არის აუცილებელი. თუ რომელიმე არ არსებობს, მაშინ მის ნაცვლად გამოიყენება გასაღები სიტყვა void. ე. ი. თუ არგუმენტების სიის ნაცვლად გამოიყენება სიტყვა void, ფუნქცია გამოძახებისას არ ღებულობს არანაირ არგუმენტებს. თუ ფუნქციის დასაბრუნებელი მნიშვნელობის ტიპია void, მაშინ გამომძახებელი პროგრამა ფუნქციისგან არანაირ მნიშვნელობას არ ღებულობს.

```
void swap(int a, int b)
     {
         int tmp = a;

         a = b;
         b = tmp;
     }
```

ფუნქციამ შეიძლება დააბრუნოს ნებისმიერი ტიპის მნიშვნელობა.

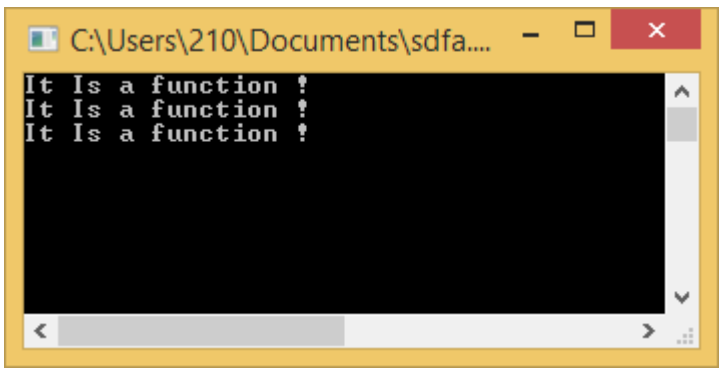
ინსტრუქცია return არ არის აუცილებელია void ფუნქციაში. თუ ის არ არის მითითებული, მაშინ პროგრამის ბლოკი მთავრდება „}“ ფრჩხილამდე მიღწევისას.

[*] sdga.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

/* run this program using the console p
void Write()
{
    printf( "It Is a function !\n" );
}

void main()
{
    Write();
    Write();
    Write();
    getch();

}
```

მოცემულია მაგალითი, რომელიც უზრუნველყოფს ძირითად პროგრამაში Write ფუნქციის გამოძახებას. ამოცანის შესრულების შედეგი იქნება „It Is a function !“ ტექსტი გამოტანილი კონსოლზე 3-ჯერ.

შესრულების შედეგი:



უნდა აღვნიშნოთ, რომ შესაძლებელია ერთი ფუნქციის გამოძახება მეორე ფუნქციით. ფუნქცია, რომელსაც გამოვიძახებთ, წინასწარ უნდა იყოს აღწერილი (გამომძახებელი კოდის ზემოთ).

მაგ.

```
#include <stdio.h>
#include <stdlib.h>
//პირველი ფუნქცია
void Pirveli()
{
  cout << "pirveli funqcia" << endl;
}
//მეორე ფუნქცია,
//რომელიც იძახებს პირველს
void Meore()
{
  cout << "pirveli funqciis gamodzaxeba" << endl;
  Pirveli();
}
//ძირითადი ფუნქცია, რომელსაც
//შეუძლია გამოიძახოს ყველა სხვა
//ფუნქცია
void main()
{
  Meore();
}
```

2.7.1. ფუნქცია არგუმენტებით

ფუნქცია არგუმენტების გარეშე გამოიყენება იშვიათად, რადგანაც ამ შემთხვევაში კავშირი ასეთ ფუნქციასთან ერთმხრივია, ანუ კავშირი ხორციელდება მხოლოდ დასაბრუნებელი მნიშვნელობით. ფუნქციის არგუმენტები საშუალებას იძლევა დამყარდეს ორმხრივი კავშირი: გადასაცემი პარამეტრებით და დასაბრუნებელი მნიშვნელობით.

ასეთი ფუნქციის აღწერის ფორმატი შემდეგი სახისაა:

```
ტიპი ფუნქციის სახელი (ტიპი პარამეტრი1, ტიპი პარამეტრი 2, ...)
{
ფუნქციის ტანი
}
```

მნიშვნელობის გადაცემა გამოძახებული ფუნქციიდან შესაძლებელია დაბრუნების **return** ოპერატორით, რომლის ჩაწერაც შემდეგი სახით ხორციელდება:

```
return გამოსახულება;
```

მაგ.

```
int f(int a, int b)
{
    if (a > b) { printf("max = %d\n", a); return a; }
    printf("max = %d\n", b); return b;
}
```

ამ ფუნქციის გამოძახება შესაძლებელია შემდეგი სახით:

```
c = f(15, 5);
c = f(d, g);
f(d, g);
```

C-ში ფუნქციის არგუმენტების გადაცემა ხდება მნიშვნელობის მიხედვით, რაც იმას ნიშნავს, რომ ფუნქცია ღებულობს ყოველი არგუმენტის დროებით ასლებს და არა მისამართს. ეს კი იმას ნიშნავს, რომ გამოძახებულ ფუნქციას არ შეუძლია შეცვალოს გამოძახებული ცვლადის მნიშვნელობა. თუმცა თუ ფუნქციას გადავცემთ არა ცვლადებს, არამედ მათ მისამართებს, მაშინ ცვლადების მნიშვნელობების შეცვლა შესაძლებელია:

მაგ.

```
void swap(int *a, int *b)
  {
      int tmp = *a;

      *a = *b;
      *b = tmp;
  }
```

ფუნქციის გამოძახება შესაძლებელია შემდეგი სახით: swap(&b, &c).

ამ შემთხვევაში ფუნქციას გადაეცემა **b** და **c** ცვლადების მისამართები, ხოლო პროგრამის ალგორითმის მიხედვით ამ ცვლადების მნიშვნელობები ერთმანეთს ადგილს გაუცვლის.

2.8. ფაილები

2.8.1. ფაილიდან ინფორმაციის მიღება

ფაილიდან ინფორმაციის მისაღებად, უპირველესად უნდა გავხსნათ ფაილი და უნდა გა-გვაჩნდეს ამ ფაილთან წვდომა.

C-ში ფაილზე მიმთითებელს გააჩნია ტიპი **FILE** და მისი გამოცხადება ხდება შემდეგი სახით:

FILE *myfile;

ფაილის გასახსნელად გამოიყენება ფუნქცია **fopen(),** რომელიც მოთავსებულია სათაურის ფაილში **stdio.h.** ამავე ფაილში არის გამოცხადებული ტიპი-სტრუქტურა **FILE.**

fopen() ფუნქციას გააჩნია ორი არგუმენტი. პირევლი არგუმენტი წარმოადგენს ფაილის დასახელებას და მისამართს, მეორე არგუმენტი ფაილთან მიმართვის რეჟიმს. ფაილის გახსნის პროცესი პროგრამაში გამოისახება შემდეგი ფორმით:

myfile = fopen ("hello.txt", "r");

მოცემულ ჩანაწერში **hello.txt** წარმოადგენს ფაილს, რომელიც ინახება იმავე მისამართზე, სადაც ინახება პროგრამული კოდის ფაილი (მიმდინარე საქაღალდე).

r კი ნიშნავს, რომ ფაილი იხსნება კითხვის რეჟიმში.

ფაილთან მიმართვის რეჟიმები:

‘r’ : ფაილის გახსნა წაკითხვის რეჟიმში . წაკითხვა ხორციელდება საწყისი პოზიციიდან;

‘r+’ : ფაილის გახსნა წაკითხვის და ჩაწერის რეჟიმში . წაკითხვა ხორციელდება საწყისი პოზიციიდან;

‘w’ : ფაილი გახდეს ნულოვანი სიგრძის ან გაიხსნას ფაილი ჩასაწერად. წაკითხვა ხორციელდება საწყისი პოზიციიდან;

‘w+’ : ფაილის გახსნა წაკითხვის და ჩაწერის რეჟიმში. თუ არ არსებობს, ფაილი იქმნება. წინააღმდეგ შემთხვევაში otherwise it is truncated. ხდება. წაკითხვა ხორციელდება საწყისი პოზიციიდან;

a’ : ფაილი გაიხსნას მონაცემების დამატების რეჟიმში (დამატება ფაილის ბოლოს), თუ ფაილი არ არსებობს, ის იქმნება. კურსორი დგება ფაილის ბოლოში.

‘a+’ : ფაილი იხსნება წაკითხვის და დამატების რეჟიმში. თუ ფაილი არ არსებობს, ის იქმნება. კურსორი დგება ფაილის ბოლოში.

ფაილთან მუშაობის დასრულების შემდეგ, ფაილი უნდა დაიხუროს, იმისათვის, რომ გათავისუფლდეს ბუფერი მონაცემებისაგან. ეს განსაკუთრებით მნიშვნელოვანია, იმ შემთხვევაში, როცა ფაილთან მუშაობის დასრულების შემდეგ პროგრამა აგრძელებს მუშაობას. ფაილის დახურვა ხორციელდება ფუნქციით **fclose().** პარამეტრის სახით მას გადაეცემა მიმთითებელი ფაილზე:

fclose(myfile);

პროგრამაში შესაძლებელია გაიხსნას ერთი ან რამდენიმე ფაილი. ასეთ შემთხვევაში ყოველ ფაილთან დაკავშირებულია საკუთარი ფაილური მიმთითებელი. თუმცა თუ დასაწყისში პროგრამა მუშაობს ერთ ფაილთან, ხოლო შემდეგ ხურავს მას, შესაძლებელია იგივე მიმთითებლის გამოყენება მეორე ფაილის გახსნისათვის.

2.8.2. ფაილიდან ინფორმაციის წაკითხვა და ჩაწერა

ფუნქცია fscanf()

ფუნქცია **fscanf()** ანალოგიური **scanf()** ფუნქციის, მაგრამ მისგან განსხვავებით ახორციელებს ფაილიდან ინფორმაციის ფორმატირებულ შეტანას. ფუნქციას გააჩნია შემდეგი პარამეტრები: ფაილური მიმთითებელი, ფორმატის სტრიქონი, მეხსიერების არის მისამართი მონაცემების ჩასაწერად:

fscanf (myfile, "%s%d", str, &a);

დავუშვათ, გვაქვს **fscanf.txt** ფაილი, რომელშიც არის შემდეგი ჩანაწერები:

```
apples 10 23.4
bananas 5 25.0
bread 1 10.3
```

იმისათვის, რომ წავიკითხოთ ეს ინფორმაცია ფაილიდან, შეგვიძლია დავწეროთ პროგრამა შემდეგი სახით:

```
#include <stdio.h>

main () {
       FILE *file;
       struct food {
              char name[20];
              unsigned qty;
              float price;
       };
       struct food shop[10];
       char i=0;

       file = fopen("fscanf.txt", "r");
```

```
        while (fscanf (file, "%s%u%f", shop[i].name, &(shop[i].qty),
&(shop[i].price)) != EOF) {
                printf("%s    %u    %.2f\n",    shop[i].name,    shop[i].qty,
shop[i].price);
                i++;
        }
}
```

ამ შემთხვევაში გამოცხადებულია სტრუქტურა და მასივის სტრუქტურა. ფაილის ყოველი სტრიქონი შეესაბამება მასივის ერთ ელემენტს; მასივის ელემენტი წარმოადგენს სტრუქტურას, რომელიც შეიცავს სტრიქონულ და ორ რიცხვით ველს. ციკლის ყოველი იტერაციის შედეგად წაკითხული იქნება ერთი სტრიქონი. როცა გვხვდება ფაილის დასასრული ფუნქცია **fscanf()** აბრუნებს მნიშვნელობას **EOF** და ციკლი სრულდება.

2.8.3. ტექსტურ ფაილში ინფორმაციის ჩაწერა

მონაცემების შეტანა, ასევე გამოტანაც შესაძლებელია იყოს სხვადასხვა სახის:

- ფორმატირებული გამოტანა. ფუნქცია **fprintf** (ფაილური მიმთითებელი, ფორმატის სტრიქონი, ცვლადი)

```
მაგ. file = fopen("fprintf.txt", "w");

        while (scanf ("%s%u%f", shop[i].name, &(shop[i].qty),
&(shop[i].price)) != EOF) {
                fprintf(file, "%s %u %.2f\n", shop[i].name, shop[i].qty,
shop[i].price);
                i++;
        }
```

- სტრიქონული გამოტანა. ფუნქცია **fputs** (***სტრიქონი, ფაილური მიმთითებელი***)

```
while (gets (arr) != NULL) {
                fputs(arr, file);
                fputs("\n", file);
        }
```

- სიმბოლოების თანმიმდევრობით გამოტანა. ფუნქცია **fputc()** ან **putc(*სიმბოლო, ფაილური მიმთითებელი*).**

```
while ((i = getchar()) != EOF)
            putc(i, file);
```

ჩავწეროთ ტექსტურ ფაილში ინფორმაცია

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7  char str_file[]="Line file ";
8  FILE* fp = fopen("my_file.txt","w");
9  if (fp != NULL)
10 {
11 printf("Recording information in the file ... \n");
12 int i;
13 for(i=0;i <strlen(str_file);i++)
14 putc(str_file[i],fp);
15 }
16 else printf("Can not open file for writing.\n");
17 fclose(fp);
18 return 0;
19 }
```

2.8.4. ინფორმაციის წაკითხვა ტექსტური ფაილიდან

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
char str_file[100];
FILE* fp = fopen("my_file.txt","r");
if(fp != NULL)
{
int i=0;
char ch;
while((ch = getc(fp)) != EOF)
str_file[i++]=ch;
str_file[i] = '\0';
printf(str_file);
}
else printf("Can not open file for writing\n");
fclose(fp);
return 0;
}
```

fprintf() და fscanf() ფუნქციების გამოყენება

```c
#include <stdio.h>
#include <stdlib.h>
struct tag_book
{
char name[100];
char author[100];
int year;
} books[2];

int main(void)
{
int i;
FILE * fp;
for(i=0;i < 2;i++)
{
scanf("%s",books[i].name);
scanf("%s",books[i].author);
scanf("%d",&books[i].year);
}

for(i=0;i < 2;i++)
{
puts(books[i].name);
puts(books[i].author);
printf("%d\n",books[i].year);
}

for(i=0;i < 2;i++)
fprintf(fp,"%s %s %d\n",books[i].name,books[i].author,
books[i].year);
fclose(fp);
fp = fopen("my_file.txt","r");
for(i=0;i < 2;i++)
fscanf(fp,"%s %s %d\n",books[i].name,books[i].author,
&books[i].year);
fclose(fp);
printf("-------------------------------------------------\n");
for(i=0;i < 2;i++)
{
puts(books[i].name);
puts(books[i].author);
printf("%d\n",books[i].year);
}
return 0;
}
```

მოცემული პროგრამის შესრულებისას წიგნების შესახებ ინფორმაცია შედის სტრუქტურის მასივში **books** და ეკრანზე გამოდის შეტანილი ინფორმაცია. შემდეგ იხსნება ფაილი **my_file.txt** ჩაწერის რეჟიმში, რომელშიც ინფორმაცია შემდეგი თანმიმდევრობით შედის:

დასახელება, ავტორი, გამოშვების წელი. რამდენადაც ამ შემთხვევაში წიგნების რაოდენობა 2-ის ტოლია, გამოსასვლელ ფაილში იქნება შემდეგი ინფორმაცია:

Gamzrdeli tsereteli 2014

Piligrimoba koelio 2000

შემდეგ, **my_file.txt** ფაილი იხსნება წაკითხვის რეჟიმში და **fscanf()** ფუნქციის დახმარებით ხდება სტრუქტურის ელემენტებში ინფორმაციის წაკითხვა. დასასრულს წაკითხული ინფორმაცია გამოდის ეკრანზე.

წარმოდგენილი მაგალითი გვიჩვენებს ინფორმაციის ფაილში სტრუქტურირებული ჩაწერის და წაკითხვის შესაძლებლობას. ყოველივე ეს საშუალებას გვაძლევს შედარებით მარტივად შევინახოთ სხვადასხვა ტიპის მონაცემი ფაილში შემდგომი გამოყენებისათვის.

დავალება

1. შეადგინეთ პროგრამული კოდი, რომელიც უზრუნველყოფს შემდეგი ალგორითმის შესრულებას:
მომხმარებელს შეაქვს ტექსტური ფაილის სახელი;
იხსნება შესაბამისი დასახელების მქონე ფაილი წაკითხვის რეჟიმში;
სრულდება ფაილში ჩაწერილი მონაცემების სიმბოლოების და სტრიქონების რაოდენობის დათვლა.

2. შეადგინეთ პროგრამული კოდი, რომელიც წაიკითხავს მონაცემებს ერთი ფაილიდან და ჩაწერს მეორე ფაილში.

შემაჯამებელი დავალებები :

1. დაწერეთ პროგრამული კოდი, რომელიც უზრუნველყოფს შემდეგი მოქმედებების შესრულებას:

ეკრანზე გამოვიდეს შეტყობინება:
წონა
შეიტანეთ წონა:

- ✓ ეკრანზე გამოვიდეს შეტყობინება:
  - სიმაღლე
  - შეიტანეთ სიმაღლე.
- ✓ ეკრანზე გამოვიდეს შეტყობინება
  - სქესი
  - შეიტანეთ სქესი
- ✓ გამოთვალეთ სტანდარტული წონა
  - თუ სქესი არის f მაშინ სიმაღლეს გამოკლებული 110;
  - თუ სქესი არის M მაშინ სიმაღლეს გამოკლებული 100;
- ✓ შეადარეთ შეტანილი წონის მნიშვნელობა სტანდარტულ წონას სქესის შესაბამისად:
  - თუ ტოლი აღმოჩნდა, გამოიტანეთ შეტყობინება: თქვენ სტანდარტული წონის ხართ.
  - თუ მეტი აღმოჩნდა, გამოთვალეთ რამდენით არის მეტი და გამოიტანეთ ეს შეტყობინებად;

- თუ ნაკლები აღმოჩნდა, გამოთვალეთ რამდენით არის ნაკლები და გამოიტანეთ ეს შეტყობინებად.

3. **დაწერეთ პროგრამული კოდი, რომელიც უზრუნველყოფს, ფაილში სატელეფონო ცნობარის ორგანიზებას და შეასრულებს შემდეგ ფუნქციებს:**

ა) ცნობარში აბონენტის გვარის და ტელეფონის ნომრის შეტანას;

ბ) აბონენტის გვარის მიხედვით ტელეფონის ნომრის ძებნას;

გ) ცნობარში აბონენტის გვარის და ტელეფონის ნომრის წაშლას

4. **დაწერეთ პროგრამული კოდი, რომელიც უზრუნველყოფს შემდეგი ალგორითმის გადაწყვეტას:**

საავტომობილო გზა დაყოფილია მონაკვეთებად და მოცემულია ამ მონაკვეთების შესაბამისი სიმაღლეები. დაადგინეთ საავტომობილო გზაზე რამდენი აღმართი შეგხვდებათ.

5. **დაწერეთ პროგრამული კოდი, რომელიც უზრუნველყოფს შემდეგი ალგორითმის გადაწყვეტას:**

მოცემულია ორნიშნა რიცხვებისაგან შედგენილი მიმდევრობა. თითოეული რიცხვი შეაბრუნეთ და იმავე თანმიმდევრობით დაბეჭდეთ.

ტესტი - სტრუქტურული დაპროგრამება

1. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
main()
{
        int x=5;
        printf("%d,%d,%d",x,x< <2,x>>2);
}
```

- (A) 1
- (B) 5,20,1
- (C) 0
- (D) კომპილაციის შეცდომა

2. რამდენჯერ შესრულდება ციკლში მოქცეული ოპერაცია?

```
main()
{
        int i;
        for(i=20, i=10; i<=20; i++)
        {
                printf("\n %d", i);
        }
}
```

- (A) 1
- (B) Run time Error
- (C) 11
- (D) კომპილაციის შეცდომა

3 :

რა დაიბეჭდება თუ a=10 და b = 20?

```
printf("%d",a==b);
```

- (A) 20
- (B) 10
- (C) 1
- (D) 0

4 :
რა დაიბეჭდება მოცემული პორგრამული კოდის შესრულების შედეგად:

```
void main()
{
        printf("%d",10?0?20?35:45:55:65);
}
```

- (A) 35
- (B) 45
- (C) 55
- (D) 65

5 : რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
void main(){
        float a;
        a=6.7;
        if(a==6.7)
        printf("A");
        else
        printf("B");
}
```

- (A) A
- (B) B
- (C) შეცდომა
- (D) არაფერი

6 : რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
#include "stdio.h"
extern int a=5;
main(){
        void fun();
        printf("\n a=%d",a);
        fun();
        return 0;
        }
        int a;
        void fun(){
```

```
        printf("\n in fun a=%d",a);
}
```

- (A) a=0 in fun a=5
- (B) a=5 in fun a=0
- (C) a=5 in fun a=5
- (D) error

.

6. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
void fun(auto int _){
        print("%d",_);
}
main(){
        fun(23);
        return 0;
}
```

- (A) 0
- (B) 5
- (C) error
- (D) 23

7. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
#include<stdio.h>
int main(){
int x;
x=10,20,30;
printf("%d",x);
return 0;
}
```

- (A) 10
- (B) 30
- (C) 30
- (D) 0

8. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
void main(){
        int a=1;
        while(a++<=1)
        while(a++<=2);
        printf("%d",a);
}
```

- (A) 1
- (B) 4
- (C) 5

(D) 6

9. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
void main(){
        int a=1;
        while(a++<=1)
        while(a++<=2);
        printf("%d",a);
}
```

(A) 2

(B) 3

(C) 4

(D) 5

10. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
void main(){
        int a;
        a=3+5*5+3;
        printf("%d",a);
}
```

(A) 43

(B) 64

(C) 31

(D) არცერთი ზემოთჩამოთვლილი

11. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
main()
{
   printf(3+"Proskills"+4);
}
```

(A) კომპილაციის შეცდომა

(B) skills

(C) kills

(D) ls

12. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
void main(){
        int i=10;
        printf("%d%d%d",++i, i++, ++i);
}
```

(A) 11 11 13

(B) 13 11 11

(C) 11 12 13

(D) 13 12 11

13. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
void main(){
        int a;
        a=100>90>80;
        printf("%d",a);
}
```

- (A) 0
- (B) 1
- (C) შეცდომა
- (D) არცერთი ზემოთჩამოთვლილი

14. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
void main(){
        int a,b,c,d;
        a=b=c=d=1;
        a=++b>1 || ++c>1 && ++d>1;
        printf("%d%d%d",a,b,c,d);
}
```

- (A) 1 1 2 1
- (B) 1 2 1 1
- (C) 1 2 2 1
- (D) 1 2 1 2

15. რა დაიბეჭდება მოცემული ოპერაციების შესრულების შედეგად:

```
#include<stdio.h>
int main(){
int a = 320;
char *ptr;
ptr =( char *)&a;
printf("%d ",*ptr);
return 0;
}
```

- (A) 2
- (B) 64
- (C) 320
- (D) არცერთი ზემოთჩამოთვლილი

3. ობიექტზე ორიენტირებული დაპროგრამება (Java)

შესავალი

Java ზოგადი დანიშნულების, ამავდროულად, მკაცრად (სტატიკურად) ტიპიზირებული, ობიექტზე ორიენტირებული დაპროგრამების ენაა.

მსოფლიო გლობალური ქსელის – ინტერნეტის ფართო გავრცელებამდე პროგრამების უმრავლესობის კომპილირება (დაპროგრამების ამა თუ იმ ენაზე წარმოდგენილი პროგრამის ტრანსლირება მანქანურ კოდში) ხდებოდა კონკრეტული პროცესორისა და ოპერაციული სისტემისათვის. ინტერნეტის განვითარებასთან ერთად გაჩნდა სხვადასხვა პროცესორებისა და ოპერაციული სისტემის მქონე კომპიუტერების დაკავშირების შესაძლებლობა, რომელმაც წინა პლანზე ერთი პლატფორმიდან მეორეზე პროგრამების ადვილად გადატანის პრობლემა წამოსწია. მის გადასაწყვეტად საჭირო გახდა ახალი ენის შემუშავება. სწორედ ასეთი ენა გახდა Java. იგი იმთავითვე შეიქმნა, როგორც პლატფორმისგან დამოუკიდებელი ენა. მას შეეძლო პლატფორმათაშორისი გადატანადი კოდის შექმნა, რაც მისი სწრაფი გავრცელების მიზეზი გახდა. გადატანადობა მიიღწევა პროგრამის საწყისი კოდის შუალედურ ენაში ტრანსლირების გზით, რომელსაც **ბაიტ-კოდი** (bytecode) ეწოდება. შემდეგ ეს შუალედური ენა სრულდება Java ვირტუალური მანქანის მიერ (**Java Virtual Machine, JVM**). შედეგად, Java-პროგრამა შეიძლება შესრულდეს ნებისმიერ პლატფორმაზე, რომელსაც Java ვირტუალური მანქანა აქვს.

საზოგადოდ, როდესაც პროგრამის კომპილირება შუალედურ ფორმაში ხდება, ხოლო მისი ინტერპრეტაცია წარმოებს ვირტუალური მანქანის მიერ, ეს იწვევს პროგრამის შესრულების პროცესის შენელებას იმ შემთხვევისგან განსხვავებით, როდესაც პროგრამა პირდაპირ კომპილირდება შესრულებად კოდში; მაგრამ Java ენის გამოყენების შემთხვევაში, პროგრამების წარმადობას შორის განსხვავება დიდი არ არის, რადგან ვირტუალური მანქანის კოდი მაქსიმალურად არის ოპტიმიზირებული და მისი გამოყენება JVM მანქანას საშუალებას აძლევს პროგრამები შეასრულოს იმაზე სწრაფად, ვიდრე მოსალოდნელია. მნიშვნელოვანია ის ფაქტიც, რომ Java ენის შემუშავებიდან სულ მიცრე დროში გაჩნდა ახალი ტექნოლოგია სახელწოდებით HotSpot. აღნიშნული ტექნოლოგია წარმოადგენს ვირტუალური მანქანის კოდის ე.წ. ოპერატიულ კომპილატორს (JustIn-Time - JIT). შესაბამისად, როდესაც JIT კომპილატორი JVM მანქანის შემადგენელი ნაწილია, ვირტუალური მანქანის ამა თუ იმ კოდის ფრაგმენტები ერთი მეორის მიყოლებით რეალურ დროში კომპილირდება შესრულებად კოდში.

დღეს Java ყველაზე გავრცელებული ენაა მოწყობილობათა რაოდენობის მიხედვით, რომლებიც Java პროგრამებს ასრულებენ და, აგრეთვე, ყველაზე პოპულარული ენაა იმ პროგრამისტების რაოდენობის მიხედვით, რომლებიც წერენ თავიანთ პროგრამებს. Java-მ მნიშვნელოვნად შეცვალა ციფრული ტექნოლოგია.

3.1. ობიექტზე ორიენტირებული დაპროგრამების ძირითადი პრინციპები

ობიექტზე ორიენტირებული დაპროგრამების ყველა ენა მოიცავს მექანიზმებს, რომლებიც ობიექტზე ორიენტირებული მოდელის რეალიზაციას გვიადვილებს. ამ მექანიზმებს წარმოადგენს: **ინკაფსულაცია**, **მემკვიდრეობითობა, პოლიმორფიზმი** და **აბსტრაქცია**. განვიხილოთ ეს კონცეფციები.

ინკაფსულაცია

ინკაფსულაცია წარმოადგენს მექანიზმს, რომელიც კოდს აკავშირებს იმ მონაცემებთან, რომელებზეც ის მანიპულირებს და ორივე კომპონენტს იცავს გარე ზემოქმედებისა და უნებართვო მიმართვებისგან. ინკაფსულაცია შეგვიძლია განვიხილოთ, როგორც დამცავი გარსი, რომელიც კოდს და მონაცემებს იცავს ამ გარსის გარეთ არსებული სხვა კოდების მიერ თავისუფალი წვდომისგან. გარსის შიდა კოდსა და მონაცემებზე წვდომა მკაცრად კონტროლირდება წინაწარ განსაზღვრული ინტერფეისით. რეალურ სამყაროსთან პარალელის გასავლებად (ან ანალოგის მისაღებად), განვიხილოთ ავტომობილის სიჩქარის გადაცემათა კოლოფი. ავტომობილის შესახებ, იგი ასეულობით ბიტი ისეთი ინფორმაციის ინკაფსულაციას ახდენს, როგორიცაა აჩქარება, იმ სავალი გზის ზედაპირის დახრილობა, რომელზეც მოძრაობა სრულდება და სიჩქარეების გადამრთველის მდგომარეობა. მომხმარებელს (მძღოლს) ამ რთულ ინკაფსულაციაზე ზემოქმედება მხოლოდ ერთი მეთოდით შეუძლია, კერძოდ, სიჩქარეთა გადამრთველის მოძრაობით. სიჩქარეთა კოლოფზე შეუძლებელია ვიმოქმედოთ, მაგალითად, მოხვევის ინდიკატორით ან შუშების გამწმენდი ბერკეტებით. ამგვარად, სიჩქარეთა გადამრთველი მკაცრად განსაზღვრული (სინამდვილეში, ერთადერთი) ინტერფეისია, რომელიც მოქმედებს სიჩქარეთა გადაცემების კოლოფზე. უფრო მეტიც, მასში მიმდინარე პროცესები გავლენას არ ახდენს კოლოფის გარეთ არსებულ ობიექტებზე. მაგალითად, ის არ ანთებს ავტომობილის ფარებს. ვინაიდან სიჩქარეების გადართვის ფუნქცია გადაცემათა კოლოფშია ინკაფსულირებული, ამიტომ ავტომობილების სხვადასხვა ათეულობით დამამზადებელმა აღნიშნული კოლოფის რეალიზება შეიძლება თავისი სურვილისამებრ მოახდინოს. თუმცა, მძღოლის გადმოსახედიდან, გადაცემათა ყველა კოლოფი ერთნაირად მუშაობს. ანალოგიური იდეა დაპროგრამებაშიც შეგვიძლია გამოვიყენოთ. ინკაფსულირებული კოდის უპირატესობა და

სიმძლავრე იმაში მდგომარეობს, რომ ყველასათვის ცნობილია, თუ როგორ მიმართოს მას და გამოიყენოს იგი რეალიზაციის ნიუანსების ცოდნის გარეშე.

Java ენაში ინკაფსულაციის საფუძველს კლასი წარმოადგენს. მართალია, კლასებს მოგვია-ნებით განვიხილავთ, მაგრამ ამ ეტაპზე სასურველია, მის მოკლე აღწერას გავეცნოთ. **კლასი** განსაზღვრავს სტრუქტურას და ქცევას (მონაცემებს და კოდს), რომელთაც ობიექტების ნაკრები ერთობლივად გამოიყენებს. კლასის ყოველი ობიექტი მოიცავს სტრუქტურასა და ქცევას, რომლებიც თავად კლასის მიერაა განსაზღვრული. კლასის ობიექტებს ხშირად, კლასის **ეგზემპლიარებს** უწოდებენ. ამგვარად, კლასი ლოგიკური კონსტრუქციაა, ხოლო ობიექტი – მისი ფიზიკური წარმოდგენა (განსახიერება).

კლასის შემუშავების დროს მომხმარებელი თავად განსაზღვრავს კოდს და მონაცემებს, რომლებიც კლასს ქმნიან. აღნიშნული ელემენტების ერთობლიობას კლასის **წევრები** ეწოდება. ზოგადად, კლასის მიერ განსაზღვრულ მონაცემებს ცვლადი–წევრები ან **ეგზემპლიარის ცვლადები** ეწოდება. კოდები, რომლებიც მონაცემებზე მოქმედებს, **მეთოდების** სახელწო-დებითაა ცნობილი. (Java ენაში არსებულ მეთოდებს C/C++ ენებში ფუნქციებს უწოდებენ). სწორად დაწერილ Java პროგრამებში მეთოდები განსაზღვრავს კლასის ცვლადი–წევრების გამოყენების გზებს. ეს ნიშნავს, რომ კლასის ქცევასა და ინტერფეისს კლასის მეთოდები განსაზღვრავს. კლასის ყოველი მეთოდი ან ცვლადი შეიძება იყოს **ღია** ან **დახურული** წვდომის. კლასის **ღია** ინტერფეისი წარმოადგენს ყველაფერ იმას, რაც შეიძლება იცოდნენ კლასის გარე მომხმარებლებმა. **დახურული** მონაცემები და მეთოდები წვდომადია მხოლოდ იმ კოდისთვის, რომელიც კლასის წევრს წარმოადგენს. შესაბამისად, ნებისმიერი სხვა კოდი, თუ ის კლასის წევრი არ არის, ვერ მიიღებს წვდომას კლასის დახურულ მონაცემებსა და მეთოდებზე. რადგან კლასის დახურული მონაცემები წვდომადია მხოლოდ იმავე კლასის ღია მეთოდების გამოყენებით, ამიტომ, ბუნებრივია, რომ კლასის დახურულ წევრებზე უნებართვო მიმართვები შეუძლებელია. ცხადია, ღია ინტერფეისი გააზრებულად უნდა დავაპროექტოთ, რომ საჭიროზე მეტად არ გავხსნათ კლასის შიდა მუშაობის ნიუანსები.

ამგვარად, **ინკაფსულაცია** კლასის რეალიზაციის დაფარვას წარმოადგენს. ის საშუალებას გვაძლევს კლასის ლოგიკა არა მხოლოდ შევცვალოთ ინტერფეისის შეუცვლელად, არამედ, ამავდროულად, დავიცვათ ობიექტები არასწორი გამოყენებისგან.

მემკვიდრეობითობა

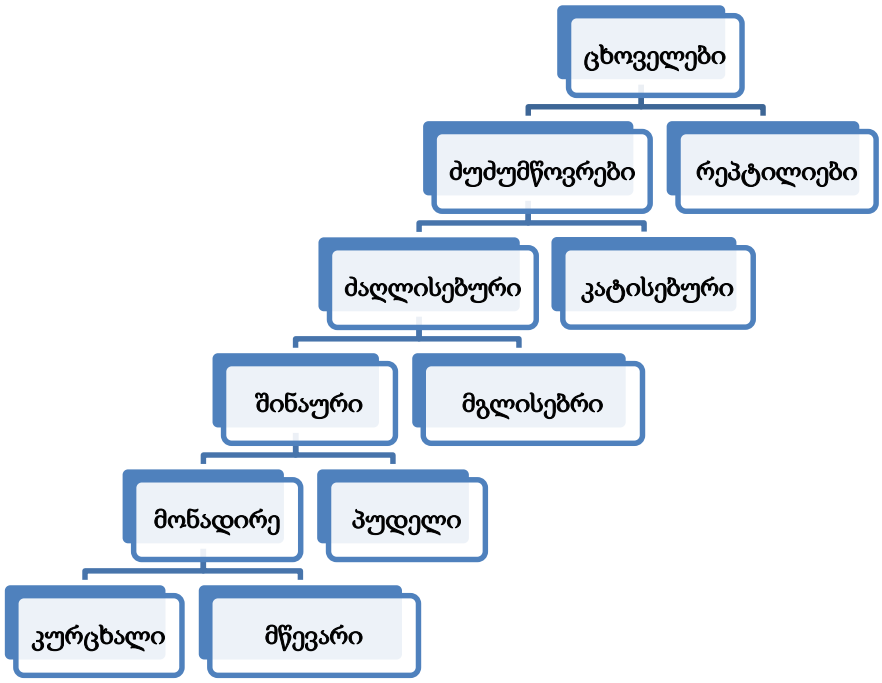
პროცესს, რომლის შედეგად ერთი ობიექტი იღებს მეორის თვისებებს, **მემკვიდრეობითობა** ეწოდება. ეს მნიშვნელოვანია, რადგან მემკვიდრეობითობა იერარქიული კლასიფიკაციის კონცეფციას უზრუნველყოფს. იერარქიის გამოყენების გარეშე თითოეული ობიექტისათვის ცხადად უნდა აღიწეროს ყველა მისი თვისება. მემკვიდრეობითობის წყალობით ობექტისათვის

უნდა აღიწეროს მხოლოდ ის თვისებები, რომლებიც განსხვავებულია, ანუ რაც მის უნიკალურობას განაპირობებს კლასის შიგნით. ობიექტს შეიძლება მემკვიდრეობით გადაეცეს საერთო ატრიბუტები თავისი მშობლიური ობიექტისაგან. უფრო დაწვრილებით განვიხილოთ ეს პროცესი მაგალითებზე.

როგორც წესი, ადამიანთა უმრავლესობა გარე სამყაროს აღიქვამს იერარქიულად ურთიერთდაკავშირებული ობიექტების სახით, მაგალითად, ცხოველები, ძუძუმწოვრები და ძაღლები. თუ საჭიროა ცხოველის ზოგადი, აბსტრაქტული აღწერა, მაშინ შეიძლება შემოვიღოთ შემდეგი ატრიბუტები: ზომები, ინტელექტის დონე, ჩონჩხის ტიპი. ცხოველებს ასევე ახასიათებთ გარკვეული მსგავსი (ერთნაირი) ქმედებებიც: ჭამა, სუნთქვა, ძილი. ჩამოთვლილი ატრიბუტები და ქცევები ფაქტიურად ცხოველთა **კლასის** აღწერას წარმოადგენს.

თუ საჭიროა ცხოველთა უფრო კონკრეტული კლასის აღწერა, მაგალითად ძუძუმწოვრების, აუცილებელია უფრო კონკრეტული ატრიბუტების მითითება, მაგალითად, კბილების ტიპი, სარძევე ჯირკვლები და სხვა. ამ აღწერას ცხოველების კლასის ქვეკლასი ვუწოდოთ. თავის მხრივ ძუძუმწოვრების კლასისათვის ცხოველების კლასი არის სუპერკლასი (მშობელი კლასი).

ვინაიდან ძუძუმწოვრები ესაა უფრო დაზუსტებულად განმარტებული ცხოველები, მათ მემკვიდროებით გადმოცემული ცხოველების ყველა ატრიბუტი აქვთ. კლასების იერარქიის ქვედა დონის ქვეკლასი მემკვიდრეობით იღებს ყოველი თავისი მშობელი კლასის ატრიბუტებს (ნახ. 57).



ნახ. 57 კლასების იერარქია

პოლიმორფიზმი

პოლიმორფიზმი (ბერძნულიდან „მრავალ ფორმას“ ნიშნავს) - ესაა თვისება, რომელიც საშუალებას იძლევა ერთიდაიგივე ინტერფეისი გამოყენებულ იქნეს მოქმედებათა საერთო კლასზე. კონკრეტული მოქმედება განისაზღვრება კონკრეტული სიტუაციის ხასიათის მიხედვით. განვიხილოთ მონაცემთა სტეკური ორგანიზება (ესაა სიის ტიპი როდესაც „ბოლო მოსული პირველი გადის“ LIFO). მონაცემების ასეთი ორგანიზებისათვის შეიძლება დაგვჭირდეს სამი ტიპის სტეკი. ერთი სტეკი გამოიყენებოდეს მთელი რიცხვებისათვის, მეორე - ნამდვილი ტიპის რიცხვებისათვის, მესამე - სიმბოლოებისათვის. თითოეული ამ სტეკის რეალიზაციის ალგორითმი ერთნაირია, იმისდა მიუხედავად, რომ თითოეული სხვადასხვა ტიპის მონაცემებს ინახავს. არაობიექტ-ორიენტირებული ენისათვის საჭირო იქნებოდა სამი სხვადასხვა ქვეპროგრამის დაწერა, თითოეულ მათგანს უნდა ჰქონოდა განსხვავებული სახელი. Java-ში, პოლიმორფიზმის წყალობით შესაძლებელია სტეკის ქვეპროგრამები ისე შევქმნათ, რომ ყველას ერთიდაიგივე სახელი ჰქონდეს.

უფრო ზოგადად პოლიმორფიზმის კონცეფციას ასეთი ფრაზით აღწერენ: „ერთი ინტერფეისი, რამდენიმე მეთოდი“. ეს ნიშნავს, რომ შესაძლებელია დავაპროექტოთ საერთო ინტერფეისი ურთიერთდაკავშირებული მოქმედებების ჯგუფისათვის. ასეთი მიდგომით შესაძლებელია პროგრამის სირთულის შემცირება, ვინაიდან ერთიდაიგივე ინტერფეისი საერთო მოქმედებების კლასის აღნიშვნისათვის გამოიყენება. კონკრეტული მოქმედების (ანუ მეთოდის) არჩევა თითოეული სიტუაციის მიხედვით ხდება და ამას კომპილატორი აკეთებს. პროგრამისტისთვის არაა აუცილებელი ამ არჩევანის ხელით გაკეთება. საკმარისია იცოდეს მხოლოდ საერთო ინტერფეისი და გამოიყენოს იგი.

თუ ძაღლებზე ანალოგიას გავნაგრძობთ, შეიძლება ვთქვათ, რომ ძაღლის ყნოსვა პოლიმორფული თვისებაა. თუ ძაღლი კატის სუნს იყნოსავს, იგი ყეფას დაიწყებს და გამოეკიდება მას. თუ ძაღლი საჭმლის სუნს იყნოსავს, მას დაეწყება ნერწყვის გამოყოფა და საჭმლის ჯამთან მივარდება. ორივე სიტუაციაში ერთიდაიგივე შეგრძნება - ყნოსვა მოქმედებს. განსხვავება ისაა, რა გამოსცემს ამ სუნს - ანუ მონაცემთა ტიპია განსხვავებული. ასეთი ზოგადი კონცეფცია Java-ში პროგრამის შიდა მეთოდების მიმართაც შესაძლებელია გამოყენებულ იქნეს.

ინკაფსულაციი, პოლიმორფიზმისა და მემკვიდრეობითობის ერთობლივად სწორად გამოყენების შემთხვევაში, ისინი ქმნიან დაპროგრამების გარემოს, რომელიც უზრუნველყოფს უფრო მდგრადი და მასშტაბირებადი პროგრამების დამუშავებას, ვიდრე პროცესზე ორიენტირებული მოდელის გამოყენება.

აბსტრაქცია

ობიექტზე ორიენტირებული დაპროგრამების მნიშვნელოვან ელემენტს **აბსტრაქცია** წარმოადგენს. ზოგადად, ადამიანისთვის დამახასიათებელია რთული მოვლენებისა და ობიექტების აბსტრაქციის გამოყენების გზით წარმოდგენა. მაგალითად, ადამიანები ავტომობილს წარმოიდგენენ არა როგორც ათი ათასი ცალკეული დეტალისგან შემდგარ ნაკრებს, არამედ, როგორც სრულიად განსაზღვრულ კონკრეტულ ობიექტს, რომელსაც თავისი უნიკალური ქცევა გააჩნია. სწორედ, აბსტრაქცია გვაძლევს საშუალებას, ვთქვათ, მაღაზიაში წასვლის დროს, არ ჩავუღრმავდეთ იმ დეტალების სირთულეს, რომლებისგანაც აწყობილია ავტომობილი. ამ შემთხვევაში, ჩვენ შეგვიძლია არ მივაქციოთ ყურადღება ძრავას ან სამუხრუჭო სისტემის ან ავტომობილის სხვა ნაწილების მუშაობის თანმიმდევრობას და ობიექტი განვიხილოთ და გამოვიყენოთ, როგორც ერთი მთლიანი.

ამდენად, ჩვენ უნდა შეგვეძლოს ობიექტის ზოგიერთი კონკრეტული დეტალისაგან აბსტრაგირება. აქ უდიდესი მნიშვნელობა აბსტრაქციის სწორი დონის შერჩევას ენიჭება, რადგან აბსტრაქციის ძალიან მაღალი დონე ობიექტის ზერელე აღწერას გვაძლევს, რაც საშუალებას არ მოგვცემს სწორად მოვახდინოთ ობიექტის ქცევის მოდელირება. აბსტრაქციის ძალიან დაბალი დონის შემთხვევაში, მოდელი საკმაოდ რთული გამოდის, ხდება მისი დეტალებით გადატვირთვა, რამაც საბოლოო ჯამში, შესაძლოა მისი გამოუსადეგარობა განაპირობოს.

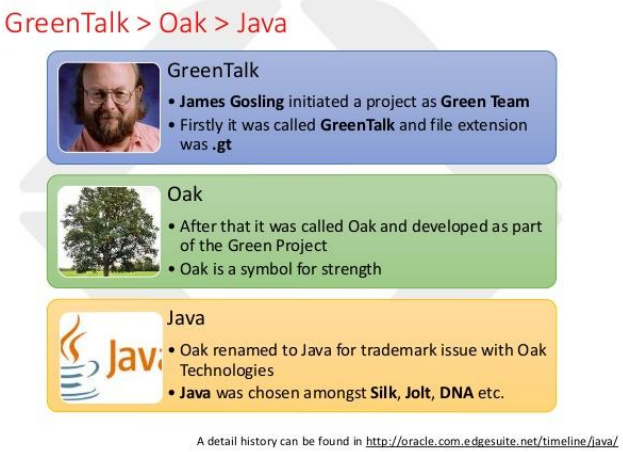
აბსტრაქციის გამოყენების მძლავრ საშუალებას იერარქიული კლასიფიკაციები წარმოადგენს. ის საშუალებას გვაძლევს გავამარტივოთ რთული სისტემების სემანტიკა (შინაარსი) მათი ფრაგმენტებად წარმოდგენის (დაყოფის) გზით, რომლებიც სამართავად უფრო ადვილი იქნება. მაგალითად, გარეგნულად ავტომობილი წარმოგვიდგება, როგორც ერთი ობიექტი, მაგრამ შიგ თუ ჩავიხედავთ, დავინახავთ, რომ ის რამდენიმე ქვესისტემისგან შედგება, კერძოდ: საჭის მართვის სისტემა, მუხრუჭები, აუდიოსისტემა, ღვედები, გამათბობელი, ნავიგატორი და სხვა. თვითოეული ქვესისტემა, თავის მხრივ, კიდევ უფრო სპეციფიური კვანძებისგანაა აგებული. მაგალითად, აუდიოსისტემა რადიომიმღებისგან, კომპაქტ დისკებისა და/ ან აუდიო კასეტების დამკვრელისგან შედგება. წარმოდგენილი მაგალითის არსი იმაში მდგომარეობს, რომ ავტომობილის (ან სხვა ნებისმიერი რთული სისტემის) რთული სტრუქტურა იერარქიული აბსტრაქციის გზით შეიძლება იქნეს აღწერილი.

რთული სისტემების იერარქიული აბსტრაქციები კომპიუტერული პროგრამების მიმართაც შეიძლება იქნეს გამოყენებული. ტრადიციულად, პროცესებზე ორიენტირებული პროგრამების მონაცემები აბსტრაქციის გამოყენებით შეიძლება მის შემადგენელ ობიექტებად გარდაიქმნას, ხოლო პროცესის მიმდევრობითი ეტაპები (ბიჯები) – შეტყობინების კოლექციად, რომლებიც ამ ობიექტებს შორის გადაიცემა. ამგვარად, ყოველი ობიექტი თავის უნიკალურ ქცევას აღწერს. ეს ობიექტები შეგვიძლია კონკრეტულ ელემენტებად ჩავთვალოთ, რომლებიც შეტყობინებებზე პასუხობენ. შეტყობინებები, თავის მხრივ, მიუთითებენ მათ ამა თუ იმ მოქმედების შესრულების აუცილებლობაზე. სწორედ, ამაში მდგომარეობს ობიექტზე ორიენტირებული დაპროგრამების არსი.

3.2. Java დაპროგრამების ენის შექმნის ისტორია

Java დაპროგრამების ენა მჭიდროდაა დაკავშირებული C++ ენასთან, რომელიც თავის მხრივ, C ენის მემკვიდრეა. შესაბამისად, Java ენის მნიშვნელოვანი ნაწილი მოიცავს დაპროგრამების ამ ორივე ენის შესაძლებლობებს. C ენისგან მან მემკვიდრეობით მიიღო მისი სინტაქსი, ხოლო ობიექტზე ორიენტრიებული დაპროგრამების მრავალი თვისება მასში C++ ენისგან იქნა გადმოტანილი.

Java-ს ისტორია 1990 წლის დეკემბრიდან იწყება, როდესაც ცნობილ კომპანია **Sun Microsystems, Inc**-ში მცირე ჯგუფის მიერ ინიცირებულ იქნა ეგრეთწოდებული „მწვანე პროექტი“ (**Green Project**). ამ პროექტის მიზანი იყო სამომხმარებლო ციფრული მოწყობილობების დაპროგრამების შესაძლებლობა, ისევე როგორც ეს კომპიუტერებზე შეიძლებოდა.



ნახ. 58 „მწვანე პროექტი“

“მწვანე პროექტის” ფარგლებში, ზემოთ აღნიშნულ კომპანიაში მომუშავე თანამშრომლების: ჯეიმს გოსლინგის (James Gosling), პატრიკ ნოტონის (Patrick Naughton), კრის ვორტის (Chris Warth), ედ ფრანკისა (Ed Frank) და მაიკ შერიდანის (Mike Sheridan) მიერ შეიქმნა დაპროგრამების ახალი ენა **Oak** (“მუხა“), რომელიც არ იყო დამოკიდებული კონკრეტული მოწყობილობის აგებულების დეტალებზე. Oak–ს შემდგომში სახელი შეუცვალეს, და მას **Java** დაერქვა.

განცხადება **Java ალფა** ვერსიის გამოსვლის შესახებ გაკეთდა 1995 წლის 23 მაისს, **Sun World** კონფერენციაზე. საბოლო ვერსიით **JDK 1.0 (Java Development Kit)** გამოვიდა 1996 წელს.

პირველი ფართო აღიარება Java ტექნოლოგიამ მიიღო 1997 წელს, როდესაც გამოვიდა **JDK**-ს შემდეგი, 1.1, ვერსია.

ნახ. 59 ჯეიმს გოსლინგი

რაოდენ უცნაურადაც არ უნდა მოგვეჩვენოს, Java დაპროგრამების ენის შექმნის საწყისი მიზეზი გახლდათ არა ინტერნეტი, არამედ პლატფორმისგან დამოუკიდებელი (არქიტექტურულად ნეიტრალური) დაპროგრამების ენის შემუშავება, რომელიც

პროგრამისტებს საშუალებას მისცემდა პროგრამული უზრუნველყოფა შეემუშავებინათ და დაენერგათ ისეთ საყოფაცხოვრებო ელექტრონულ ტექნიკაში, როგორიც იყო მიკროტალღური ღუმელები და დისტანციური მართვის მოწყობილობები.

ძნელი მისახვედრი არ არის, რომ კონტროლერების სახით დღესაც მრავალი სხვადასხვა ტიპის პროცესორი გამოიყენება, ხოლო ამ შემთხვევაში, C და C++ დაპროგრამების ენების (ისევე როგორც, დაპროგრამების მრავალი სხვა ენის) გამოყენების პრობლემა იმაში მდგომარეობს, რომ აღნიშნულ ენებზე დაწერილი პროგრამების კომპილირება მხოლოდ კონკრეტული პლატფორმებისთვის ხდება. თუმცა, აღსანიშნავია, რომ C++ ენაზე შემუშავებული პროგრამების კომპილირება შესაძლებელია პრაქტიკულად ნებისმიერი ტიპის პროცესორისთვის (ამისთვის საჭიროა C++ სრული კომპილატორის არსებობა), მაგრამ პრობლემა ის არის, რომ კომპილატორების შექმნა საკმაოდ დიდ მატერიალურ დანახარჯებთანაა დაკავშირებული და ამასთან, იგი მნიშვნელოვან დროს მოითხოვს, რამეთუ საკმაოდ შრომატევადი პროცესია. შესაბამისად, საჭირო გახდა შედარებით მარტივი და ეკონომიური თვალსაზრისით, უფრო ხელსაყრელი გადაწყვეტილების მიღება. სწორედ ასეთი გადაწყვეტილების მოძებნის მიზნით გოსლინგმა და მისმა თანამოაზრეებმა მუშაობა დაიწყეს პლატფორმისგან დამოუკიდებელი და გადატანითი თვისებების მქონე დაპროგრამების ენის შექმნაზე. ამ ძალისხმევამ ისინი Java ენის შემუშავებამდე მიიყვანა.

იმ პერიოდში, როდესაც Java ენის ძირითადი მახასიათებლები განისაზღვრებოდა, თავი იჩინა, მეორე და ამასთან, გაცილებით უფრო მნიშვნელოვანმა ფაქტორმა – მსოფლიო გლობალურმა ქსელმა (World Wide Web), რომელმაც გადამწყვეტი როლი შეასრულა Java დაპროგრამების ენის სპეციფიკის ჩამოყალიბებაში, რადგან თავის მხრივ, ისიც საჭიროებდა გადატანითი პროგრამების არსებობას.

ამგვარად, შეგვიძლია დავასკვნათ, რომ სწორედ ინტერნეტმა განაპირობა Java ენის ფართომასშტაბიანი წარმატება.

როგორც ზემოთ აღვნიშნეთ, Java–მ C და C++ დაპროგრამების ენებისგან მემკვიდრეობით მრავალი საინტერესო თვისება მიიღო. ეს სპეციალურად იქნა დაშვებული Java ენის შემქმნელების მიერ. მათ მშვენივრად იცოდნენ, რომ C ენის სინტაქსის გამოყენება და C++ ენის ობიექტზე ორიენტირებული თვისებების გამეორება მილიონობით პროგრამისტს დადებითად განაწყობდა Java–ს მიმართ და დროთა განმავლობაში ის პროფესიონალი პროგრამისტების ენა გახდებოდა.

ინტერნეტისთვის პროგრამების შემუშავების თვალსაზრისით, Java ისეთ ენად იქცა, როგორსაც თავის დროზე C წარმოადგენდა სისტემური დაპროგრამებისთვის. ხმამაღალ

ნათქვამად თუ არ ჩაგვითვლით, Java ის რევოლუციური ძალა გახდა, რომელმაც მსოფლიო შეცვალა.

Java-ს ძირითადი მახასიათებელი თვისებაა იყოს დამოუკიდებელი შესრულების გარემოს აგებულების დეტალებისაგან Java პლატფორმის ყველაზე ცნობილი დაპირების: „დაწერე ერთხელ, გაუშვი ყველგან“ – შესაბამისად. ასეთი მიდგომა, პირველ რიგში, ამარტივებს პროგრამის შემქმნელის შრომას. აღსანიშნავია, რომ კომპანია Sun Microsystems 2009–2010 წლებში Oracle–ის ორგანიზაციის მეირ იქნა შესყიდული და შესაბამისად, Java დღეს მისი კუთვნილებაა.

ნახ. 60 “Oracle” -ის კორპორაცია კალიფორნიაში

3.3. Java ენასთან დაკავშირებული ტერმინოლოგია

Java ენის შექმნისა და განვითარების ისტორიის განხილვა არ იქნებოდა სრულყოფილი Java–ს სპეციფიკური ტერმინოლოგიის აღწერის გარეშე. მართალია, ძირითად ფაქტორებს, რომელთაც Java–ს შემუშავება განაპირობეს, პროგრამების გადატანითობისა და უსაფრთხოების უზრუნველყოფა წარმოადგენდა, მაგრამ ენის საბოლოო ვერსიის ჩამოყალიბებაში დიდი როლი სხვა ფაქტროებმაც (შემდგომში Java–ს თვისებებმა) შეასრულა.

ახლა კი განვიხილოთ ის ტერმინები, რომლებიც უშუალოდ Java ენასთანაა დაკავშირებული და მოვახდინოთ მათი ანალიზი:

- სიმარტივე;
- უსაფრთხოება;
- გადატანითობა;
- ობიექტზე ორიენტირებულობა;
- მდგრადობა;
- მრავალნაკადიანობა;
- არქიტექტურული ნეიტრალობა;
- ინტერპრეტაციის უნარი;
- მაღალი წარმადობა;
- განაწილებული ხასიათი;
- დინამიური ხასიათი.

სიმარტივე

Java ენა თავდაპირველად შემუშავდა, როგორც პროფესიონალი პროგრამისტებისთვის მარტივად ასათვისებელი ენა, შემდგომში მისი ეფექტურად გამოყენების მიზნით. მათთვის, ვინც გარკვეულ ცოდნასა და გამოცდილებას ფლობს დაპროგრამებაში, Java ენის ათვისება დიდ სირთულეს არ წარმოადგენს. მათთვის კი, ვინც იცის ობიექტზე ორიენტირებული დაპროგ-რამების საბაზო კონცეფციები, Java–ს ათვისება კიდევ უფრო იოლია. C++–ზე მომუშავე გამოცდილი პროგრამისტებისთვის Java–ზე გადასვლა მინიმალურ ძალისხმევას საჭიროებს. სწორედ ამაში მდგომარეობს Java დაპროგრამების ენის სიმარტივე.

უსაფრთხოება

როგორც მომხმარებლებისთვისაა ცნობილი, ნებისმიერი „ჩვეულებრივი“ პროგრამის ჩატვირთვა გარკვეულ რისკთანაა დაკავშირებული, რადგან ჩასატვირთი კოდი შესაძლოა ვირუსებს შეიცავდეს. აღნიშნული პრობლემის არსი იმაში მდგომარეობს, რომ ვირუსს შეუძლია

თავისი „შავი“ საქმე შეასრულოს, რადგან მას სისტემურ რესურსებთან არასანქცირებული წვდომა გააჩნია. მაგალითად, როდესაც ვათვალიერებთ კომპიუტერის ლოკალური ფაილების სისტემის შიგთავსს, ვირუსულ პროგრამას შეუძლია ისეთი კონფიდენციალური ინფორმაციის მიღება, როგორიცაა საკრედიტო ბარათების ნომრები, პაროლები, საბანკო ანგარიშების მდგომარეობის შესახებ ინფორმაცია და სხვა. Java დაპროგრამების ენა უსაფრთხოებას უზრუნველყოფს, რამეთუ აპლეტი (Java პროგრამის განსაკუთრებული სახე) თავსდება Java–ს შესრულების გარემოში და საშუალებას არ აძლევს მას გააჩნდეს წვდომა კომპიუტერის ოპერაციული სისტემის სხვა ნაწილებთან. მრავალი ექსპერტის აზრით, სწორედ სისტემის უსაფრთხოების უზრუნველყოფა წარმოადგენს Java–ს ნოვატორულ ასპექტს.

გადატანითობა

გადატანითობა ინტერნეტის დამახასიათებელი განსაკუთრებული თვისებაა, რადგან გლობალური ქსელი აკავშირებს და აერთიანებს სხავდასხვა ტიპის კომპიუტერებსა და ოპერაციულ სისტემებს. გადატანითობა, აპლეტის შემთხვევაში, გულისხმობს იმ ფაქტს, რომ ერთი და იგივე აპლეტს შესაძლებლობა უნდა გააჩნდეს, რომ ჩაიტვირთოს და შესრულდეს ინტერნეტთან დაკავშირებულ არაერთ და სხვადასხვა ტიპის პროცესორზე, ოპერაციულ სისტემასა თუ ბრაუზერზე. აპლეტების განსხვავებული ვერსიების შექმნა სხვადასხვა კომპიუტერებისთვის, სავსებით არარაციონალურია. ერთი და იგივე კოდი ყველა კომპიუტერზე უნდა სრულდებოდეს. ამან განაპირობა გადატანითი შესრულებადი კოდის შესაქმენლად გარკვეული მექანიზმის შემუშვება.

ობიექტზე ორიენტირებულობა

Java დაპროგრამების ენისთვის დამახასიათებელია ობიექტებისადმი მკაფიო, პრაქტიკული და პრაგმატული მიდგომა. რადგან Java –მ მისი წინამორბედი ენებისგან (C++/C) მემკვიდრეობით ობიექტზე ორიენტირებული მრავალი თვისება მიიღო, ამიტომ მასში დაცულია ბალანსი ისეთ ორ მთავარ მოვლენას შორის, როგორიცაა კონცეფცია „პროგრამის ყველა კომპონენტი – ობიექტებია“ და პრაგმატული მოდელი „გზიდან ჩამომეცალე“. Java –ს ობიექტის მოდელი მარტივია და იოლად იძლევა გაფართოების საშუალებას, მაგრამ ამავდროულად, ისეთი ელე-მენტარული ტიპები, როგორიცაა მთელირცხვა მონაცემები, ობიექტებად არ განიხილება.

მდგრადობა

დღეს თანამედროვე პროგრამების მიმართ მოთხოვნები საკმაოდ გაიზარდა, რადგან ისინი საიმედოდ უნდა მუშაობდნენ ნებისმიერ ოპერაციულ სისტემაში. ამიტომ, მდგრადი პროგრამების შექმნა ერთ–ერთ უმთავრეს პრიორიტეტს წარმოადგენდა Java ენის შემუშავების დროს. საიმედოობის უზრუნველსაყოფად Java გარკვეულ შეზღუდვებს მოიცავს, რაც

პროგრამისტს აიძულებს პროგრამის შემუშავების საწყის ეტაპებზე მოახდინოს შეცდომების გამოვლენა და მათი აღმოფხვრა. რადგან Java მკაცრად ტიპიზირებული ენაა, ამიტომ კოდის შემოწმება მიმდინარეობს მისი კომპილაციის დროს, მაგრამ კოდი შესრულების პროცესშიც მოწმდება. ამიტომ Java–ში შეცდომების გამოვლენა და აღმოფხვრა საკმაოდ მარტივად ხორციელდება. სხვადასხვა სიტუაციებში კოდის განსაზღვრა Java–ს ერთ–ერთი მთავარი თვისებაა.

იმის საილუსტრაციოდ, თუ როგორ მიიღწევა Java პროგრამების მდგრადობა, განვიხილოთ შემდეგი შემთხვევები: მეხსიერების მართვასთან დაკავშირებული შეცდომები და განსაკუთრებული სიტუაციების არასწორი დამუშავება. დაპროგრამების ტრადიციულ გარემოში მეხსიერების მართვა საკმაოდ რთული და შრომატევადი ამოცანაა. მაგალითად, C და C++ დაპროგრამების ენებში პროგრამისტი ხელით ახდენს დინამიურად განაწილებული მეხსიერების რეზერვირებას და გამოთავისუფლებას, რაც ზოგჯერ პრობლემის წარმოქმნას განაპირობებს. ეს უკანასკნელი იმაში გამოიხატება, რომ ხშირად პროგრამისტს ავიწყდება რეზერვირებული მეხსიერების გამოთავისუფლება ან ცდილობს გაასუფთაოს მეხსიერების ის უბანი, რომელიც კოდის ამა თუ იმ ნაწილის მიერ ჯერ კიდევ გამოიყენება. აღნიშნული სიტუაციები Java–ში გამორიცხულია, რადგან ის ავტომატურად მართავს მეხსიერების რეზერვირებისა და გამოთავისუფლების პროცესებს. ფაქტიურად, მეხსიერების გამოთავისუფლება Java–ში მთლიანად ავტომატიზირებულია, რამეთუ ის მოიცავს „ნაგვის შეგროვების“ ფუნქციას გამოუყენებელი ობიექტების მიმართ. დაპროგრამების ტრადიციულ გარემოში განსაკუთრებული სიტუაციები იქმნება ისეთ შემთხვევებში, როგორიცაა: ნულზე გაყოფა, „ფაილი ვერ მოიძებნა“ და ა.შ., რომელთა მართვა რთული კონსტრუქციების გამოყენებით ხდება. Java აღნიშნულ პრობლემას მარტივად წყვეტს. ის გვთავაზობს განსაკუთრებული სიტუაციების დამუშავების ობიექტზე ორიენტირებულ მექანიზმს. კარგად დაწერილ Java პროგრამაში განსაკუთრებული სიტუაციების აღმოფხვრის პროცესი თავად პროგრამამ უნდა მართოს.

მრავალნაკადიანობა

Java ენა შემუშავებულ იქნა ინტერაქტიული ქსელური პროგრამების შექმნის საჭიროების პასუხად. აღნიშნული მიზნის მიღწევა Java–ში შესაძლებელია მრავალნაკადიანი პროგრამების საშუალებით, რომლებსაც ერთდროულად რამდენიმე მოქმედების შესრულება შეუძლიათ. სხვადასხვა პროცესების სინქრონიზაციის ამოცანა Java–ში სრულფასოვნადაა გადაჭრილი. მრავალნაკადიანი პროგრამების შემუშავებისადმი მიდგომა Java–ში მარტივადაა რეალიზებული, რაც პროგრამისტს საშუალებას აძლევს ყურადღება გაამახვილოს პროგრამის კონკრეტულ მოქმედებაზე და არა მრავალი ამოცანისგან შემდგარი ქვესისტემის შექმნაზე.

არქიტექტურული ნეიტრალობა

ერთ–ერთ ძირითად ამოცანას, რომელიც Java-ს შემქმნელების წინაშე იდგა, გრძელვადიანი და გადატანითი კოდის შემუშავება წარმოადგენდა. მთავარი პრობლემა, რაც Java–ს შემუშავების დროს პროგრამისტებს აფიქრებდათ, იყო იმ გარანტიის უქონლობა, რომ დღეს დაწერილი კოდი ხვალ წარმატებით შესრულდებოდა, თუნდაც, იმავე კომპიუტერზე. ოპერაციული სისტემები და პროცესორები მოდერნიზაციას განიცდიდნენ, ხოლო ძირითად სისტემურ რესურსებში განხორციელებულ ცვლილებებს შეეძლო პროგრამის არაქმედითუნარიანობა გამოეწვია. აღნიშნული სიტუაციის შეცვლის მიზნით Java–ს ავტორებმა ხისტი გადაწყვეტილება მიიღეს და მკაცრი მოთხოვნა წამოაყენეს როგორც Java ენის, ისე Java ვირტუალური მანქანის მიმართ. მათ მიზნად დაისახეს, რომ „პროგრამები შექმნილიყო მხოლოდ ერთხელ, ნებისმიერ გარემოში, ნებისმიერ დროს და სამუდამოდ“. აღნიშნულ მიზანს მათ გარკვეულწილად მიაღწიეს კიდეც.

ინტერპრეტაციის უნარი და მაღალი წარმადობა

დაპროგრმების ენა Java–ს კომპილატორი ასრულებს რა პროგრამების კომპილაციას შუალედური წარმოდგენით, რასაც ვირტუალური მანქანის კოდი ეწოდება, Java საშუალებას გვაძლევს შევიმუშავოთ მრავალპლატფორმიანი პროგრამები. აღნიშნული კოდი შესრულებადია ნებისმიერ სისტემაში, რომელიც ვირტუალური Java მანქანის რეალიზებას ახდენს. მრავალ-პლატფორმიანი გადაწყვეტილებების მიღების პირველმა მცდელობამ მიზანს მიაღწია პროგრამების შესრულების სწრაფქმედების შემცირების ხარჯზე; მაგრამ მოგვიანებით Java მანქანის ვირტუალური კოდი ისე იქნა დაპროექტებული, რომ JIT- კომპილაციის გამოყენების შედეგად მისი გარდასახვა მანქანაზე დამოკიდებულ კოდში შესაძლებელი გახდა მაღალი წარმადობით.

განაწილებული ხასიათი

Java ენა განკუთვნილია ინტერნეტის განაწილებული გარემოსთვის, რამეთუ ის TCP/IP „ოჯახის“ პროტოკოლების მხარდაჭერას უზრუნველყოფს. ფაქტიურად, URL მისამართის გამოყენებით რესურსთან მიმართვა დიდად არ განსხვავდება ფაილთან მიმართვისგან. Java–სთვის ასევე, დამახასიათებელია მეთოდების დისტანციური გამოძახება (Remote Method Invocation – RMI). ეს თვისება პროგრამებს საშუალებას აძლევს მეთოდები ქსელიდან იქნეს გამოძახებული.

დინამიური ხასიათი

Java პროგრამები ინფორმაციის მნიშვნელოვან მოცულობას შეიცავენ შესრულების დროის შესახებ, რომელიც გამოიყენება უფლებამოსილების შემოწმებისა და შესრულების პროცესში ობიექტებზე წვდომის მისაღებად. ეს საშუალებას გვაძლევს განვახორციელოთ კოდების უსაფრთხო და ტექნიკურად გამართლებული დაკავშირება. ეს უკანასკნელი კი, ძალზედ მნიშვნელოვანია Java გარემოს მდგრადობის თვალსაზრისით, რომელშიც ვირტუალური მანქანის კოდის მცირე ფრაგმენტები შეიძლება დინამიურად იქნეს განახლებული მოქმედ სისტემაში.

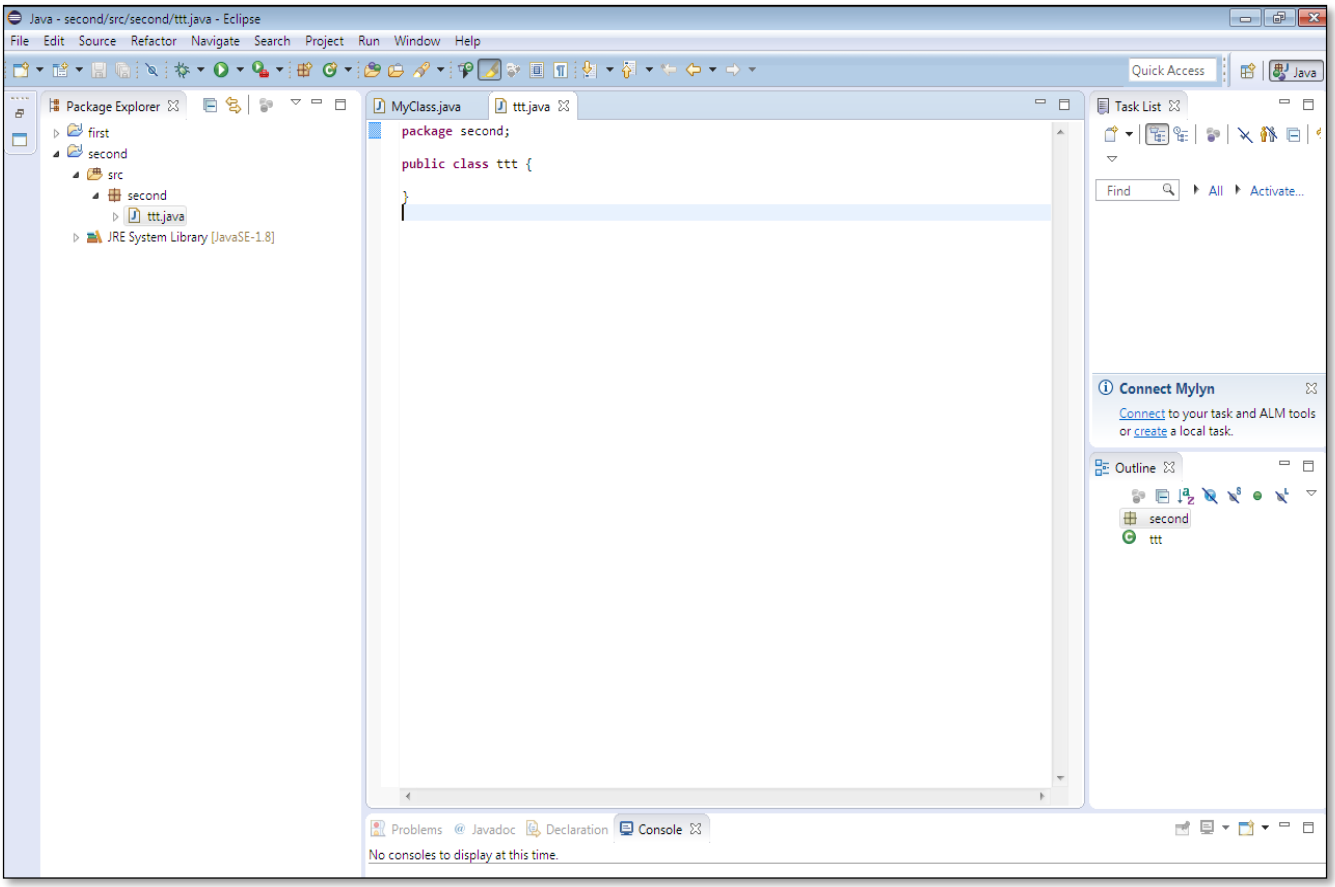
3.4. Java ენის ლექსიკა, ბაზისური ტიპები და ოპერაციები მათზე

პირველი პროგრამა

ტრადიციის მიხედვით, მრავალი სახელმძღვანელო იწყება უმარტივესი პროგრამით “Hello World!”. იმისათვის, რომ Java დაპროგრამების ენაზე წარმოვადგინოთ ეს მარტივი ტექსტი, სასურველია მივმართოთ eclipse ან NetBeans რედაქტორებს. ჩვენ eclipse რედაქტორს გამოვიყენებთ და წარმოგიდგენთ ამ რედაქტორში მუშაობის პრინციპს (ასევე, შეგიძლიათ იხილოთ შემდეგი ბმული

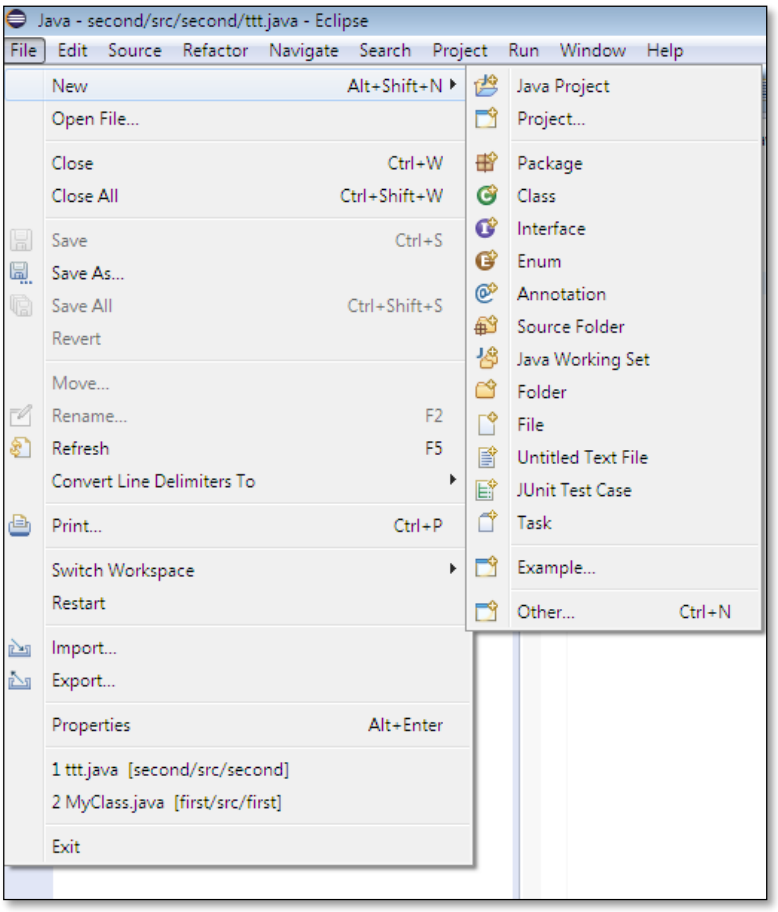
https://www.youtube.com/watch?v=SHIT5VkNrCg&index=4&list=PLFE2CE09D83EE3E28) .

რედაქტორის ჩატვირთვის შედეგად ეკრანზე გამოდის მე-61 სურათზე ნაჩვენები სამუშაო გარემო.



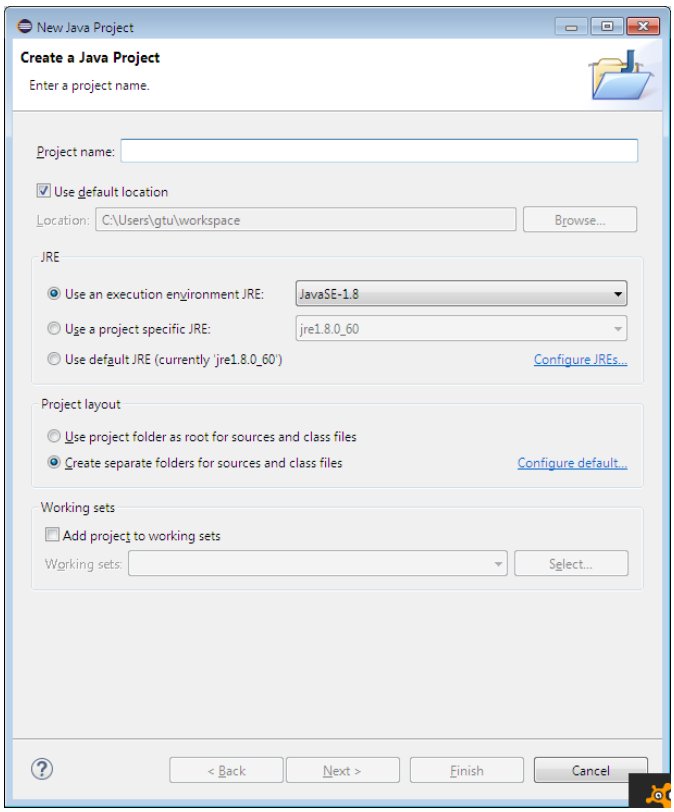
ნახ. 61 სამუშაო გარემო

აღნიშნულ გარემოში, ახალი პროექტის შესაქმნელად საჭიროა გავხსნათ File მენიუ და ჩამოთვლადი სიიდან ავირჩიოთ ფუნქცია New, შემდეგ კი - ქვეფუნქცია Java Project (იხილეთ ნახ. 62).

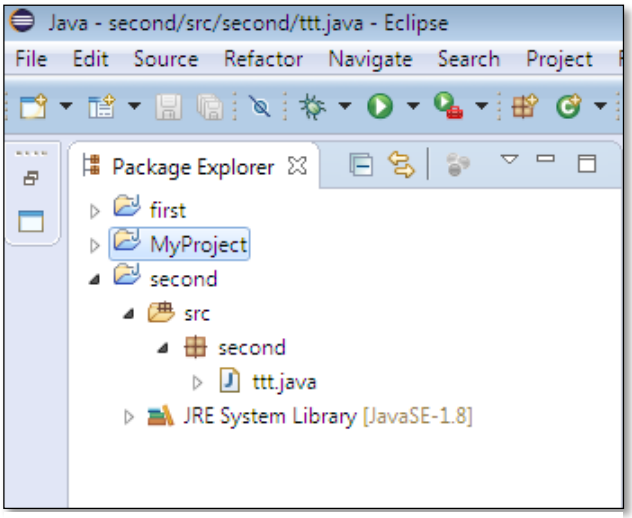


ნახ. 62

ზემოაღნიშნული პროცედურების შესრულების შედეგად, ეკრანზე გამოდის ახალი პროექტის შესაქმნელი დიალოგური ფანჯარა: New Java Project (იხილეთ მე-63 ნახაზი), სადაც ველში: Project name საჭიროა ავკრიფოთ ახალი პროექტის ჩვენთვის სასურველი სახელი, მაგალითად, MyProject. ამავე დიალოგურ ფანჯარაში Finish ღილაკზე დაჭერის შემდეგ, ეკრანზე ბრუნდება სამუშაო გარემო, სადაც მარცხენა მხარეს არსებული პროექტების ჩამონათვალში ჩვენ მიერ შექმნილი ახალი პროექტის სახელწოდება ჩანს (იხილეთ ნახ.64).

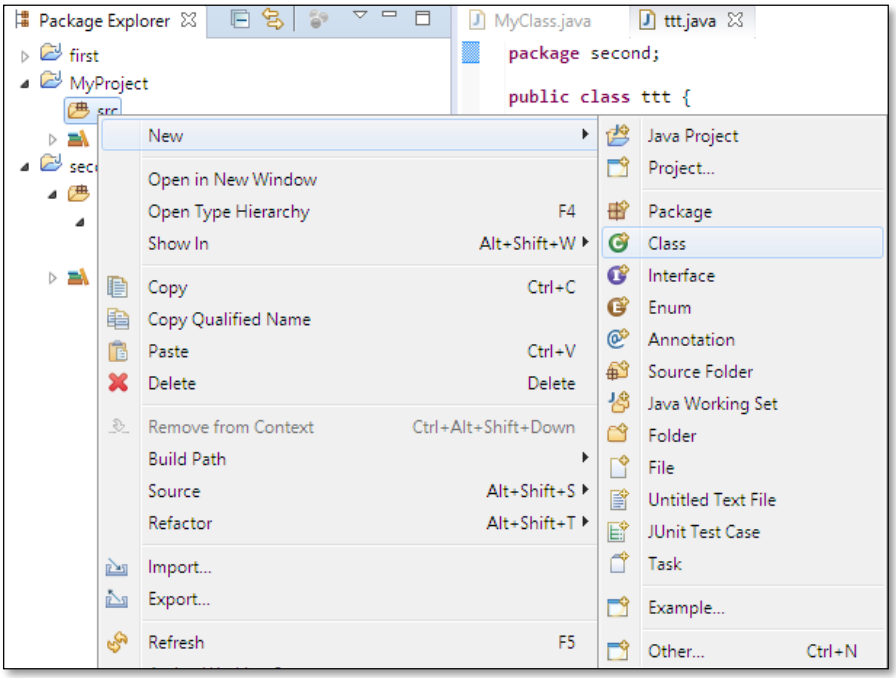


ნახ. 63

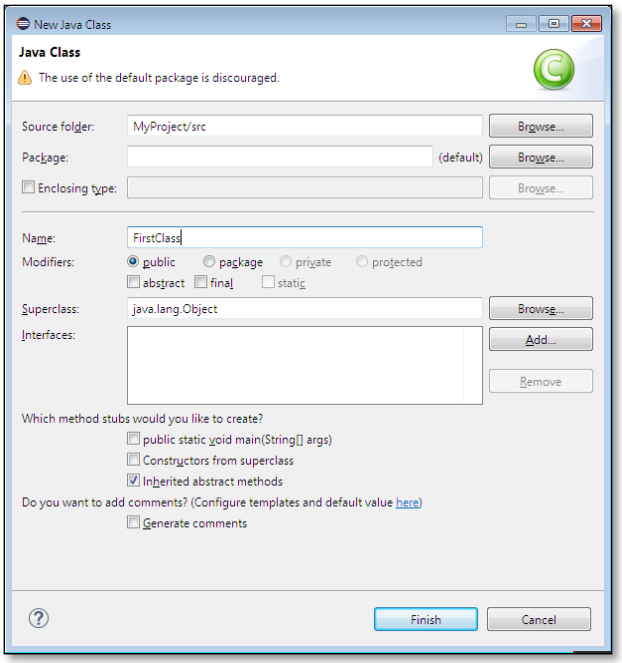


ნახ. 64

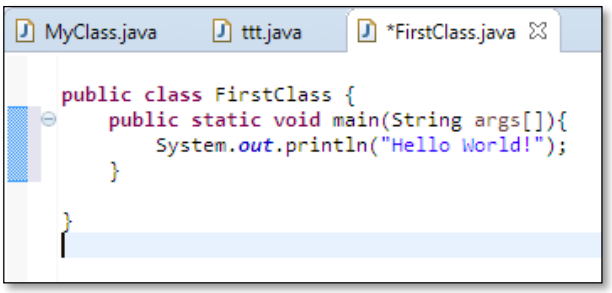
ვაწვებით რა, ჩვენ მიერ შექმნილი პროექტის სახლწოდებას მის მარცხნივ არსებულ ისარზე, პროექტი იხსნება და src ფუნქციაზე „მაუსის“ მარჯვენა ღილაკზე დაჭერით გამოსულ კონტექსტურ მენიუში ვირჩევთ ფუნქციას New, შემდეგ ქვეფუნქციას Class (იხილეთ ნახ. 65). აღნიშნული მოქმედებების შესრულების შედეგად ეკრანზე გამოდის ახალი კლასის შესაქმნელი დიალოგური ფანჯარა (იხილეთ ნახ. 66), სადაც Name ველში ვკრეფთ კლასის სახელს, მაგალითად, FirstClass; პროცესს ვასრულებთ Finish ღილაკზე დაჭერით. FirstClass.java გაფართოების მქონე ფაილში (სამუშაო გარემოში) ვკრეფთ ჩვენი პირველი პროგრამის ტექტს (იხილეთ ნახ.67)



ნახ. 65

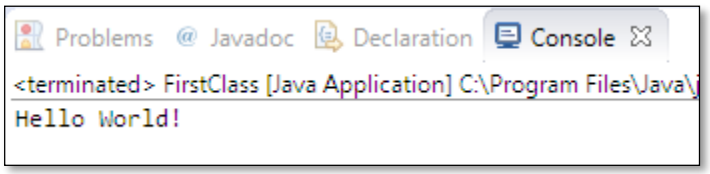


ნახ. 66



ნახ. 67

ჩვენი პროგრამის შესრულებაზე გაშვებას ვახდენთ ისნტრუმენტების ზოლზე არსებული Run ღილაკით. შედეგი ეკრანის ქვედა ნაწილში კონსოლზე გამოდის (იხილეთ ნახ.68).



ნახ. 68

ამ მარტივ მაგალითზეც შესაძლებელია შევამჩნიოთ ენის მთელი რიგი თავისებურებები:

- ყოველი პროგრამა წარმოადგენს ერთ ან რამოდენიმე **კლასს**. ჩვენ მაგალითში მხოლოდ ერთი კლასია (**class**).
- კლასის დასაწყისს აღნიშნავს დარეზერვებული სიტყვა **class**, რომელსაც მოსდევს პროგრამისტის მიერ შერჩეული კლასის კონკრეტული სახელი, ჩვენ შემთხვევაში **FirstClass**. ყველაფერი, რაც კლასს ეკუთვნის, ფიგურულ ფრჩხილებში იწერება და წარმოადგენს კლასის ტანს (**class body**).
- ყოველი მოქმედება (გამოთვლები) წარმოებს ინფორმაციის დამუშავების მეთოდების (**methods**) საშუალებით.
- მეთოდებს აქვთ განსხვავებული სახელები. ერთ-ერთი მეთოდის სახელი აუცილებლად უნდა იყოს **main**. ამ მეთოდით იწყება პროგრამის შესრულება. ჩვენ მარტივ მაგალითში ერთი მეთოდია, ამიტომ მისი სახელია **main**.
- როგორც წესი, მეთოდი შესრულების შედეგად იძლევა (ხშირად ვამბობთ: აბრუნებს **return**) ერთ მნიშვნელობას, რომლის ტიპი მეთოდის სახელის წინ მიეთითება. მეთოდმა შეიძლება არაფერი არ დააბრუნოს, როდესაც იგი პროცედურის როლს ასრულებს, როგორც ეს ჩვენ შემთხვევაშია. ასეთ დროს დაბრუნებული მნიშვნელობის ტიპის მაგივრად სიტყვა **void** იწერება.
- მეთოდის სახელის შემდეგ ფრჩხილებში თავსდება არგუმენტების ან პარამეტრების ჩამონათვალი, რომლებიც ერთმანეთისაგან მძიმეებით გამოიყოფა. ყოველი არგუმენტისათვის ეთითება მისი ტიპი და ცარიელი სიმბოლოს შემდეგ - სახელი. მაგალითში მხოლოდ ერთი არგუმენტია, ის სტრიქონული მასივის ტიპისაა. სიმბოლოების მასივი - ესაა Java-ში არსებული String ტიპი. კვადრატული ფრჩხილები მიუთითებს მასივს. მასივის სახელი ნებისმიერი დასაშვები იდენტიფიკატორი შეიძლება იყოს. მაგალითში არჩეულია **args**.
- მეთოდის დაბრუნებული ტიპის წინ შეიძლება ჩაწერილ იქნეს სხვადასხვა **მოდიფიკატორი** (**modifiers**). მაგალითში ორი მოდიფიკატორია: სიტყვა **public** აღნიშნავს, რომ ეს მეთოდი წვდომადია ყველასათვის; სიტყვა **static** უზრუნველყოფს **main()** მეთოდის გამოძახების შესაძლებლობას პროგრამის შესრულების დასაწყისში. ზოგადად, მოდიფიკატორები არაა აუცილებელი, მაგრამ **main()** მეთოდისათვის ისინი სავალდებულოა. მეთოდის სახელის

შემდეგ ყოველთვის ვწერთ ფრჩხილებს, რითაც ხაზს ვუსვამთ, რომ ეს მეთოდის სახელია და არა ჩვეულებრივი ცვლადის.

- მეთოდის მთელი შემცველობა წარმოადგენს მეთოდის ტანს (**method body**) და იწერება ფიგურულ ფრჩხილებში.

ერთადერთი მოქმედება, რაც **main()** მეთოდში ხდება, ესაა სხვა მეთოდის გამოძახება, რომელსაც რთული სახელი აქვს **System.out.println()** და მას გადაეცემა ერთი არგუმენტი, ტექსტური კონსტანტა **„Hello World!“.** ტექსტური კონსტანტა იწერება ორმაგ ბრჭყალებში, რაც გამყოფს წარმოადგენს და ტექსტის შემადგენლობაში არ შედის. **println()** მეთოდი თავის არგუმენტს გაიტანს გამავალ ნაკადში, რომელიც ჩვეულებრივ, ტექსტური ტერმინალის ეკრანთანაა დაკავშირებული. ტექსტის გამოტანის შემდეგ, კურსორი გადადის შემდეგი სტრიქონის დასაწყისში (ამას მიუთითებს დაბოლოება **ln**, სიტყვა **println –** შემოკლებით **print line**).

მნიშვნელოვანია აღინიშნოს, რომ Java-ს კომპილატორი განასხვავებს დიდ და პატარა ასოებს. პროგრამაში სიტყვები **String, System** უნდა დაიწყოს დიდი ასოებით, ხოლო **main-** პატარა ასოთი. ტექსტური კონსტანტის შიგნით კი მნიშვნელოვანი არაა პატარა ასოები იქნება გამოყენებული თუ დიდი, განსხვავება მხოლოდ ეკრანზე გამოჩნდება.

Java-ში პროგრამისტის მიერ შერჩეული სახელები შეიძლება ჩაიწეროს მისი შეხედულებისამებრ. კლასისთვის შეგვეძლო დაგვერქვა: **fisrstclass** ან **firstClass**, მაგრამ Java-პროგრამისტებს შორის მოქმედებს შეთანხმება, რომელსაც უწოდებენ „**Code Conventions for the Java Programming Language**“, რომელსაც შეიძლება გაეცნოთ მისამართზე: **java.sun.com/docs/codeconv/index.html**. ამ შეთანხმების ზოგიერთი პუნქტი ასეთია:

- კლასის სახელები დიდი ასოებით იწყება; თუ სახელი შედგება რამდენიმე სიტყვისაგან, ყოველი შემდეგი სიტყვა დიდი ასოთი უნდა იწყებოდეს;
- ცვლადებისა და მეთოდების სახელები პატარა ასოთი უნდა იწყებოდეს; თუ სახელი რამდენიმე სიტყვისაგან შედგება, ყოველი შემდეგი სიტყვა დიდი ასოთი უნდა იწყებოდეს;
- კონსტანტების სახელები იწერება მხოლოდ დიდი ასოებით; თუ სახელი რამდენიმე სიტყვისაგან შედგება, ყოველ სიტყვას შორის ხაზგასმის სიმბოლო უნდა ჩაიწეროს;

ამ წესების გათვალისწინება აუცილებელი არ არის, მაგრამ მათი დაცვა მნიშვნელოვნად აადვილებს პროგრამის კოდის კითხვადობას და მას Java-ს სტილს აძლევს.

სტილს განსაზღვრავს არამარტო სახელები, არამედ პროგრამის ტექსტის სტრიქონებად განლაგებაც. მაგალითად, ფიგურული ფრჩხილების განლაგება: გამხსნელი ფიგურული ფრჩხილი კლასის, მეთოდის სათაურის სტრიქონის ბოლოში ჩავწეროთ თუ ის გადავიტანოთ მის შემდეგ სტრიქონზე? ეს თითქოს უმნიშვნელო საკითხი ხშირად დავას იწვევს და

სხვადასხვა რედაქტორები, როგორც წესი, საშუალებას იძლევა მოხდეს ფიგურული ფრჩხილების დასმის სასურველი სტილის არჩევა.

კომპილატორისათვის დაპროგრამების სტილს მნიშვნელობა არ აქვს, იგი მთელ პროგრამას განიხილავს, როგორც ერთ, მთლიან სტრიქონს, მაგრამ სასურველია დავიცვათ აღნიშნული შეთანხმება.

რომელიმე რედაქტორში შექმნილი პროგრამის შენახვა უნდა მოხდეს ფაილში, რომლის გაფართოება იქნება **.java**, თუ ამ პირობას დავიცავთ, ოპერაციულ სისტემას გაუადვილდება ამ ფაილთან ასოცირებული პროგრამის გამოძახება. სასურველია ფაილს დაერქვას კლასის სახელი, ასოების რეგისტრის შენარჩუნებით, თუ ფაილში რამდენიმე კლასია, დაარქვით **main()** მეთოდის შემცველი კლასის სახელი.

Java თავისუფალი ფორმატის ენაა. ეს ნიშნავს, რომ პროგრამის დაწერისას რაიმე სპეციალური წესების დაცვა აბზაცების მიმართ საჭირო არ არის. ერთადერთი, აუცილებელი მოთხოვნაა, რომ ყოველ ლექსემას შორის, რომლებიც ოპერაციის სიმბოლოთი ან გამყოფით გამოყოფილი არაა, ჩასმული იყოს ერთი ცარიელი სიმბოლო მაინც.

Java-ში ცარიელ სიმბოლოდ ითვლება ჰარი (**space**), ტაბულაცია ან ახალი სტრიქონის სიმბოლო.

იდენტიფიკატორები

იდენტიფიკატორი კლასების, მეთოდების და ცვლადების დასახელებისათვის გამოიყენება. იდენტიფიკატორი შეიძლება იყოს დიდი და პატარა ასოების, ციფრების, ხაზგასმისა და დოლარის სიმბოლოს ნებისმიერი მიმდევრობა. იდენტიფიკატორი არ უნდა იწყებოდეს ციფრით, რათა კომპილატორს არ აერიოს რიცხვით კონსტანტებში. ისევ გავიმეოროთ, რომ Java დიდი და პატარა ასოების მიმართ მგრძნობიარეა. მაგალითად **VALUE** და **Value** სხვადასხვა იდენტიფიკატორებია. კორექტული იდენტიფიკატორების მაგალითებია:

```
AvgTemp
Count
a4
$test
this_is_ok
```

შემდეგი დასახელებები დაუშვებელია:

```
2count
high-temp
Not/ok
```

კომენტარები

პროგრამის ტექსტში შეიძლება ჩავსვათ კომენტარები, რომელსაც კომპილატორი არ გაითვალისწინებს. ისინი ძალიან მნიშვნელოვანია პროგრამის მიმდინარეობის განმარტებისათვის. პროგრამის გამართვისას შესაძლებელია ერთი ან რამდენიმე ოპერატორი გამოვრთოთ განხილვისგან, მათი კომენტარებად მონიშვნის გზით ანუ „დავაკომენტაროთ“.

Java-ში შესაძლებელია სამი ტიპის კომენტარის გამოყენება:

- კომენტარი იწყება ორი დახრილი ხაზის შემდეგ // (ხაზებს შორის არ უნდა იყოს ცარიელი სიმბოლო) და გრძელდება სტრიქონის ბოლომდე;
- კომენტარი იწყება დახრილი ხაზისა და ვარსკვლავის სიმბოლოთი /*, რომელიც შეიძლება გრძელდებოდეს რამდენიმე სტრიქონზე და მთავრდება ვარსკვლავით და დახრილი ხაზით */ (ამ სიმბოლოებს შორის ცარიელი სიმბოლო არ უნდა იყოს).
- კომენტარი იწყება დახრილი ხაზისა და ორი ცალი ვასკვლავისაგან /** და მთავრდება */ -ით. შეიძლება დაიკავოს რამდენიმე სტრიქონი, იგი მუშავდება სპეციალური პროგრამა **javadoc**-ის მიერ. კომენტარების ეს ტიპი თვითდოკუმენტირებისთვის გამოიყენება.

3.4.1. Java ენის ბაზისური ტიპები და ოპერაციები მათზე

Java დაპროგრამების ენაში მონაცემთა მრავალი ტიპი და მათი გამოყენების კიდევ უფრო მეტი წესი არსებობს. ამ წესების დაუცველობა დაფარულ შეცდომებს იწვევს, რომელთა აღმოჩენაც საკმაოდ ძნელი და შრომატევადი სამუშაოა.

საწყისი მონაცემების ყველა ტიპი, რომლის აღწერაც Java-შია შესაძლებელი, ორ ჯგუფად იყოფა: პრიმიტიული (ბაზისური) (**primitive types**) და აღწერითი (**reference types** მიმთითებელი, მისამართი, მაკავშირებელი) ტიპები.

აღწერითი ტიპები იყოფა მასივებად (**arrays**), კლასებად (**classes**) და ინტერფეისებად (**interfaces**).

პრიმიტიული სულ 8 ტიპია. ისინი შეიძლება დაიყოს **რიცხვით** (**numeric**), **სიმბოლურ** (**character**) და **ლოგიკურ** (**boolean** ბულის) ტიპებად.

რიცხვითი ტიპები თავის მხრივ იყოფა: მთელრიცხვა (**integer**) და ნამდვილ (**floating-point**) ტიპებად:

- **მთელრიცხვა.** ამ ჯგუფში შედის **byte, short, int, long** ტიპები, რომლებიც ნიშნიან მთელ რიცხვებს წარმოადგენს.
- **ნამდვილი (მცოცავმძიმიანი) რიცხვები.** ამ ჯგუფში შედის **float, double** ტიპები, ისინი ნიშნიან წილად რიცხვებს წარმოადგენენ, რომლებიც ათობითი წერტილის შემდეგ თანრიგების გარკვეული სიზუსტით მოიცემა.

- **სიმბოლოების** ტიპია **char**, რომელიც ენაში დაშვებულ ყველა სიმბოლოს წარმოადგენს, მაგალითად, ასოებს, ციფრებს, სასვენ ნიშნებს, სპეციალურ სიმბოლოებს და სხვა. ხშირად ამ ტიპს მთელრიცხვა ტიპს მიაკუთვნებენ და განიხილავენ, როგორც უნიშნო მთელ რიცხვს.
- **ლოგიკური** ანუ **ბულის** ტიპია **boolean**. ესაა სპეციალური ტიპი, რომელიც ჭეშმარიტი (**true**) ან მცდარი (**false**) მნიშვნელობის წარმოსადგენად გამოიყენება.

ეს ტიპები შეიძლება გამოყენებულ იქნეს იმ სახით, როგორც არიან განსაზღვრული ან პროგრამისტის მიერ საკუთარი, ახალი ტიპების შესაქმნელად. ამრიგად, ამ ტიპების საფუძველზე შეიძლება ყველა სხვა ტიპი შეიქმნას.

მერვე ცხრილში ნაჩვენებია **მთელი ტიპის** მონაცემების მიერ დაკავებული მეხსიერების ზომა, დასაშვები მნიშვნელობების დიაპაზონი და სტანდარტული მნიშვნელობა გამოცხადების შემდეგ.

ცხრილი 8

ტიპი	ბიტების რაოდენობა	დასაშვები დიაპაზონი	სტანდარტ. მნიშვნელობა
byte	8 (1 B)	-128÷127	0
short	16 (2 B)	-32768÷32767	0
int	32 (4 B)	-2147483648÷ 2147483647	0
long	64 (8 B)	-9223372036854775808 ÷ 9223372036854775807	0L

მე-9 ცხრილში ნაჩვენებია **მცოცავმძიმიანი ტიპის** მონაცემების მიერ დაკავებული მეხსიერების ზომა, დასაშვები მნიშვნელობების დიაპაზონი, სტანდარტული მნიშვნელობა გამოცხადების შემდეგ და სიზუსტე.

ცხრილი 9

ტიპი	ბიტების რაოდ.	დასაშვები დიაპაზონი	სტანდარტული მნიშვნელობა	სიზუსტე
double	64 (8 B)	4.9e-324÷1.8e+308	0.0D	17
float	32 (4 B)	1.4e-045÷3.4e+038	0.0F	7-8

Java-ში სიმბოლოების შესანახად **char** ტიპი გამოიყენება.

უნდა აღვნიშნოთ, რომ ეს ტიპი ალგორითმულ ენა **C**-შიც გამოიყენება, მაგრამ ისინი ეკვივალენტური არ არის. **C**-ში **char** - ესაა 8 ბიტიანი მთელრიცხვა ტიპი. **Java**-ში კი სიმბოლოს წარმოდგენისათვის **Unicode** სტანდარტი გამოიყენება, რომელიც სიმბოლოების საერთაშორისო ნაკრებს განსაზღვრავს. ის ყველა ცნობილი ენების სიმბოლოებს მოიცავს და ათობით სხვა-დასხვა სიმბოლოს უნიფიცირებულ ერთობლიობას წარმოადგენს.

კონსტანტები (ლიტერალები) ასევე შეიძლება დავყოთ რიცხვით, რომელიც თავის მხრივ იყოფა მთელრიცხვა და მცოცავმძიმიან (ნამდვილ) კონსტანტებად, სიმბოლურ, ლოგიკურ (ბულის) და სტრიქონულ კონსტანტებად.

ცვლადი Java პროგრამაში მონაცემთა შენახვის ძირითადი კომპონენტია. ის განი-საზღვრება: იდენტიფიკატორით, ტიპით და საწყისი მნიშვნელობით, რომელიც პრინციპში არასავალდებულოა. გარდა ამისა, ყველა ცვლადს აქვს განსაზღვრის არე (რომელიც სხვა ობიექტების მიმართ მის ხილვადობას განაპირობებს) და არსებობის დრო.

Java-ში ცვლადების გამოცხადება მათ გამოყენებამდე უნდა მოხდეს. ცვლადების გამოცხადების ძირითადი ფორმა ასე შეიძლება წარმოვადგინოთ:

```
<ტიპი> <იდენტიფიკატორი> [=<მნიშვნელობა>] [,<იდენტიფიკატორი> [=მნიშვნელობა] ...];
```

<ტიპი> - აქ უნდა ჩაიწეროს **Java**-ს ერთ–ერთი კონკრეტული ელემენტარული ტიპი ან კლასის სახელი ან ინტერფეისი (კლასებსა და ინტერფეისებს ქვემოთ განვიხილავთ).

<იდენტიფიკატორი> - აქ უნდა ჩაიწეროს ცვლადის სახელი.

ცვლადს შეიძლება თავიდანვე მივანიჭოთ საწყისი მნიშვნელობა (მოვახდინოთ მისი ინიციალიზაცია) = (უდრის) ნიშნისა და კონკრეტული მნიშვნელობის მითითებით. შესაძლებელია ინიციალიზაციისათვის გამოსახულების მითითებაც, ოღონდ მისი მნიშვნე-ლობის ტიპი უნდა ემთხვეოდეს ცვლადის ტიპს ან დაიყვანებოდეს ამ ტიპზე.

ერთი და იგივე ტიპის რამდენიმე ცვლადის ერთდროულად გამოცხადებისათვის შეიძლება გამოვიყენოთ სია, სადაც წევრები ერთმანეთისგან მძიმით იქნება გამოყოფილი.

მაგალითად:

```
int a, b, c; // გამოცხადებულია სამი int ტიპის ცვლადი:
int d = 3, e, f = 5; // სამი int ტიპის ცვლადი,
                     // ხდება d და f-ის ინიციალიზაცია
byte z = 22; // z ცვლადის ინიციალიზაცია
```

double pi = 3.14159; // pi ცვლადის ინიციალიზაცია

char c = `x`; // c ცვლადს ენიჭება `**x**`მნიშვნელობა

Java-ში შეიძლება გამოვყოთ ოპერაციების 4 ძირითადი ჯგუფი: არითმეტიკული ოპერაციები, ბიტური ოპერაციები, შედარების ოპერაციები და ლოგიკური ოპერაციები.

არითმეტიკული ოპერაციების შესრულება არ შეიძლება **boolean** ტიპის ცვლადებზე, მაგრამ შესაძლებელია **char** ტიპთან, ვინაიდან Java-ში ეს ტიპი, ფაქტიურად **int** ტიპის ქვესიმრავლეა.

მე-10 ცხრილში **არითმეტიკული ოპერაციები**ა წარმოდგენილი.

ცხრილი 10

ოპერაცია	აღწერა
+	შეკრება
-	გამოკლება
*	გამრავლება
/	გაყოფა
%	მოდულით გაყოფა
++	ინკრემენტი
+=	ავტოასოციური შეკრება
-=	ავტოასოციური გამოკლება
*=	ავტოასოციური გამრავლება
/=	ავტოასოციური გაყოფა
%=	ავტოასოციური მოდულით გაყოფა
--	დეკრემენტი

Java-ში შესაძლებელია განხორციელდეს გარკვეული ოპერაციები **long, int, short, char** და **byte** მთელრიცხვა ტიპის ბიტებზე.

მე-11 ცხრილში ეს **ბიტური ოპერაციები**ა წარმოდგენილი.

ცხრილი 11

ოპერაცია	აღწერა
~	ბიტური უნარული NOT (უარყოფა)
&	ბიტური AND (და)
|	ბიტური OR (ან)
^	ბიტური გამომრიცხავი OR (ან)

>>	მარჯვნივ ძვრა
>>>	მარჯვნივ ძვრა ნულებით შევსებით
<<	მარცხნივ ძვრა
&=	ავტოასოციური ბიტური AND
|=	ავტოასოციური ბიტური OR
^=	ავტოასოციური ბიტური გამომრიცხავი OR
>>=	ავტოასოციური მარჯვნივ ძვრა
>>>=	ავტოასოციური მარჯვნივ ძვრა ნულებით შევსებით
<<=	ავტოასოციური მარცხნივ ძვრა

შედარების ოპერაციებით ხდება ოპერანდების მნიშვნელობების ტოლობისა და მათი ერთმანეთის მიმართ მეტობის ან ნაკლებობის განსაზღვრა. მე-12 ცხრილში **შედარების ოპერაციები**ა წარმოდგენილი.

ცხრილი 12

ოპერაცია	**აღწერა**
==	ტოლია
!=	არ არის ტოლი
>	მეტია
<	ნაკლებია
>=	მეტია ან ტოლი
<=	ნაკლებია ან ტოლი

შედარების ოპერაციების შესრულების შედეგი ყოველთვის **boolean**-ის ტიპისაა. შედარების ოპერაციებს ხშირად იყენებენ პირობით და ციკლის ოპერატორებში.

Java-ში ნებისმიერი ტიპის მნიშვნელობების შედარება შესაძლებელია == და != ოპერაციებით. ყურადღება გაამახვილეთ, რომ ტოლობაზე შემოწმება აღინიშნება ორი „==“ ნიშნით (ერთი ნიშანი „=“ მინიჭებას აღნიშნავს). მეტობაზე (ან ნაკლებობაზე) შემოწმება დასაშვებია მხოლოდ რიცხვითი მნიშვნელობის მქონე ტიპებზე. ანუ მეტობის/ნაკლებობის ოპერაციის შესრულება შესაძლებელია მთელ ტიპებზე, ნამდვილ ტიპებზე ან სიმბოლურ ტიპებზე.

მე-13 ცხრილში **ბულის ლოგიკური ოპერაციები**ა ნაჩვენები.

ცხრილი 13

ოპერაცია	აღწერა
&	ლოგიკური AND (და)
|	ლოგიკური OR (ან)
^	ლოგიკური XOR (გამომრიცხავი ან)
||	მოკლე OR
&&	მოკლე AND
!	ლოგიკური უნარული NOT (არა)
&=	ავტოასოციური AND (მინიჭებით)
|=	ავტოასოციური OR (მინიჭებით)
^=	ავტოასოციური XOR (მინიჭებით)
==	ტოლია
!=	არ არის ტოლი

ბულის ლოგიკის ოპერაციები &, | და ^ ბულის ტიპის მნიშვნელობებზე მოქმედებენ ისეთნაირად, როგორც მთელრიცხვა მნიშვნელობების ბიტებზე. ბულის ოპერაცია ! ინვერტირებას უკეთებს ბულის მდგომარეობას: **!true == false** და **!false == true**. თითოეული ლოგიკური ოპერაციის შედეგი ნაჩვენებია მე-14 ცხრილში.

ცხრილი 14

A	B	A|B	A&B	A^B	~A
false	**false**	**false**	**false**	**false**	**true**
true	**false**	**true**	**false**	**true**	**false**
false	**true**	**true**	**false**	**true**	**true**
true	**true**	**true**	**true**	**false**	**false**

Java-ში განხორციელებულია ორი საინტერესო ბულის ოპერაცია, რომლის მსგავსი სხვა ენებში არ გვხვდება. ესაა **AND** და **OR** ოპერაციების მეორე ვერსიები, რომლებსაც **მოკლე ლოგიკურ ოპერაციებს** უწოდებენ. როგორც ზემოთ ნაჩვენები ცხრილიდან ჩანს, **OR** ოპერაციის შესრულების შედეგი არის **true**, როდესაც A-ს მნიშვნელობა არის **true**, იმისდა მიუხედავად, რა მნიშვნელობა აქვს B ცვლადს. ანალოგიურად, **AND** ოპერაციის შესრულებისას შედეგია **false**, თუ **A** ოპერანდის მნიშვნელობაა **false**, **B**-ს მნიშვნელობის მიუხედავად. || და && ოპერაციების გამოყენებისას | და & ოპერაციების მაგივრად, არ მოხდება მარჯვენა ოპერანდის გამოთვლა, თუ გამოსახულების მნიშვნელობის დადგენა შესაძლებელია მარტო მარცხენა ოპერანდის საშუალებით. ეს თვისება განსაკუთრებით მოსახერხებელია, როდესაც მარჯვენა ოპერანდის მნიშვნელობა დამოკიდებულია მარცხენა ოპერანდის მნიშვნელობაზე.

Java-ში რეალიზებულია სპეციალური სამ ოპერანდიანი (ტერნერული) ოპერაცია, რომელმაც ზოგჯერ, გარკვეულწილად **If-then-else** ოპერატორი (ამ ოპერატორს მომდევნო თავებში განვიხილავთ) შეიძლება ჩაანაცვლოს. ესაა ოპერაცია **„?:“,** რომლის ზოგადი ფორმა შემდეგია:

<გამოსახულება1> ? <გამოსახულება2> :<გამოსახულება3>

ზემოაღნიშნული ოპერაციის მუშაობის პრინციპი შემდეგში მდგომარეობს: თავდაპირველად მოწმდება **<გამოსახულება1>**, რომელიც თანადობის (შედარების) ან ლოგიკურ ოპერაციას წარმოადგენს, თუ მისი შედეგი ჭეშმარიტია, მაშინ შერულდება **<გამოსახულება2>** და არ შესრულდება **<გამოსახულება3>**; ხოლო თუ **<გამოსახულება1>** -ის შედეგი მცდარია, შესრულდება **<გამოსახულება3>** და არ შესრულდება **<გამოსახულება2>**.

ახლა კი განვიხილოთ წრფივი სტრუქტურის პროგრამის ამსახველი რამდენიმე მაგალითი.

მაგალითი 1. შევადგინოთ მთელრიცხვა და მცოცავმძიმიანი მონაცემების არითმეტიკის ამსახველი პროგრამა. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 69-ზე, ხოლო შესრულების შედეგი ნახ. 70-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magaliti1;
2  public class mag1 {
3      public static void main(String  args []){
4          // მთელრიცხვა არითმეტიკული ოპერაციები
5          System.out.println("მთელრიცხვა არითმეტიკა");
6           int a = 1 + 1;
7           int b =  a * 3;
8           int c = b / 4;
9           int d = c - a;
10          int e = -d;
11         System.out.println("a = " + a);
12         System.out.println("b = " + b);
13         System.out.println("c = " + c);
14         System.out.println("d ="+ d);
15         System.out.println("e  =" + e);
16         // double ტიპზე არითმეტიკული ოპერაციები
17         System.out.println("\n მცოცავმძიმიანი არითმეტიკა");
18         double da = 1 + 1;
19         double db = da * 3;
20         double dc = db / 4;
21         double dd = dc - a;
22         double de = -dd;
23         System.out.println("da = "+ da);
24         System.out.println("db = "+ db);
25         System.out.println("dc = " + dc);
26         System.out.println("dd = " + dd);
27         System.out.println("de = " + de);
28
29
30     }
31 }
32
```

ნახ. 69

შედეგი:

```
მთელრიცხვა არითმეტიკა
a = 2
b = 6
c = 1
d =-1
e  =1

მცოცავმძიმიანი არითმეტიკა
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

ნახ. 70

სასურველია დაწვრილებით გავეცნოთ ზოგიერთ არითმეტიკულ ოპერაციას.

მოდულით გაყოფის ანუ ნაშთის გამოთვლის ოპერაცია (%) აბრუნებს გაყოფის ოპერაციის ნაშთს. ეს ოპერაცია შეიძლება გამოყენებულ იქნეს როგორც მთელ ტიპებზე, ისე მცოცავმძიმიან რიცხვებზე.

მაგალითი 2. შევადგინოთ პროგრამა, რომელიც მოდულით გაყოფის ოპერაციის დემონსტრირებას ახდენს. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 71-ზე, ხოლო შესრულების შედეგი ნახ. 72-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magaliti1;
2  public class mag1 {
3      public static void main(String  args []){
4          int x=42;
5          double y = 42.25;
6          System.out.println("x mod 10 ="+x % 10);
7          System.out.println("y mod 10 ="+y % 10);
8
9      }
10 }
```

ნახ. 71

შედეგი:

```
x mod 10 =2
y mod 10 =2.25
```

ნახ. 72

Java-ში არსებობს სპეციალური ოპერაციები, რომლებიც არითმეტიკული და მინიჭების ოპრაციის გაერთიანებას წარმოადგენენ. დაპროგრამებაში ხშირად გვხვდება ამდაგვარი ოპერატორები:

a = a + 4;

Java-ში ეს ოპერატორი შეიძლება შემდეგი სახით ჩაიწეროს:

a += 4;

ოპერატორის ამ ვერსიაში გამოყენებულია **ავტოასოციური ოპერაცია** +=. ორივე ეს ოპერაცია ერთიდაიგივე მოქმედებას ასრულებს: ცვლადის მნიშვნელობას ზრდის 4-ით.

მეორე მაგალითი:

a = a % 2;

რომელიც ასე შეიძლება ჩაიწეროს:

a %= 2;

ამ შემთხვევაში %= ოპერაცია გამოითვლის ნაშთს და შედეგს მიანიჭებს ისევ **a** ცვლადს. ავტოასოციური ოპერაციები არსებობს ყველა ოროპერანდიანი არითმეტიკული ოპერაციებისათვის.

ამრიგად, ყველა ოპერატორი, რომელსაც აქვს ფორმა:

<ცვლადი> = <ცვლადი> <ოპერაცია> <გამოსახულება>;

შეიძლება ჩაიწეროს შემდეგნაირად:

<ცვლადი> <ოპერაცია>= გამოსახულება;

ავტოასოციური ოპერაციები ორ უპირატესობას იძლევა:

1. იგი ამცირებს შესატანი კოდის მოცულობას, ვინაიდან წარმოადგენს გრძელი ფორმის ერთგვარ „შემოკლებულ“ ვარიანტს;
2. შემსრულებელ გარემოში მათი რეალიზაცია უფრო ეფექტურია, ვიდრე ეკვივალენტური გრძელი ფორმის.

ამ მიზეზების გამო პროფესიონალურად შედგენილ **Java** პროგრამებში ავტოასოციური ოპერაციები ძალიან ხშირად გვხვდება.

მაგალითი 3. შევადგინოთ ავტოასოციური ოპერაციების ამსახველი პროგრამა. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 73-ზე, ხოლო შესრულების შედეგი ნახ. 74-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magaliti1;
2  public class mag1 {
3      public static void main(String  args []){
4          int a = 1;
5          int b = 2;
6          int c = 10;
7           a += 5;
8           b *= 4;
9           c %= 6;
10         System.out.println("a =  " + a);
11         System.out.println("b = "+ b);
12         System.out.println("c = " + c);
13
14
15     }
16 }
17
```

ნახ. 73

შედეგი:

```
a =  6
b = 8
c = 4
```

ნახ. 74

ოპერაცია **ინკრემენტი** აღინიშნება ასე: ++, ხოლო **დეკრემენტი** აღინიშნება - -. ინკრემენტის ოპერაცია ოპერანდის მნიშვნელობას ზრდის ერთით, ხოლო დეკრემენტის - ამცირებს ერთით. მაგალითად, ოპერატორი **x = x + 1**; ინკრემენტის ოპერაციით ასე ჩაიწერება: **x++;** ანალოგიურად, ოპერატორი **x = x – 1**; ეკვივალენტურია შემდეგი ოპერატორის: **x--;**

ეს ოპერატორები შეიძლება ჩაიწეროს, როგორც პოსტფიქტური ფორმით, როდესაც ოპერაციის სიმბოლო ჩაიწერება ოპერანდის შემდეგ, ისე პრეფიქსული ფორმით, ამ დროს ოპერაციის სიმბოლო წინ უსწრებს ოპერანდს. ზემოთ მოყვანილ მაგალითში ნებისმიერი ამ ფორმის გამოყენება ტოლფასია და ამიტომ არ აქვს მნიშვნელობა რომელს გამოვიყენებთ. მაგრამ, როდესაც ინკრემენტის/დეკრემენტის ოპერაცია წარმოადგენს სხვა უფრო რთული გამოსახულების ნაწილს, მაშინ მჟღავნდება მისი ჩაწერის ფორმის განსხვავება. პრეფიქსულ ფორმაში ოპერანდის მნიშვნელობა იზრდება ან მცირდება მნიშვნელობის გამოსახულებაში გამოყენებამდე. პოსტფიქსურ ფორმაში ოპერანდის მნიშვნელობა ჯერ გამოიყენება გამოსახულების გამოთვლაში, ხოლო ამის შემდეგ ოპერანდის მნიშვნელობა იცლება (იზრდება ან მცირდება იმისდა მიხედვით ++ თუ -- მითითებული). მაგალითად:

x=25; y=++x; ამ შემთხვევაში y-ის მნიშვნელობა გახდება 26, როგორც მოსალოდნელო იყო, ვინაიდან ოპერანდის მნიშვნელობის ერთით გაზრდა ხდება მანამ, სანამ **y**-ს მიენიჭება **x**-ის მნიშვნელობა. ამრიგად, **y=++x**; სტრიქონი ეკვივალენტურია შემდეგი ორი სტრიქონის:

x = x + 1; y = x; მაგრამ თუ ოპერატორებს ჩავწერთ ასე:

x=25; y=x++; მაშინ, **x** ცვლადის მნიშვნელობა ამოიღება ინკრემენტის ოპერაციის შესრულებამდე, ამიტომ **y** ცვლადის მნიშვნელობა ტოლი იქნება 25-ის. ცხადია, ორივე შემთხვევაში **x** ცვლადის მნიშვნელობა ერთით იზრდება და იქნება 26-ის ტოლი. შესაბამისად **y=x++;** სტრიქონის მნიშვნელობა ეკვივალენტურია შემდეგი ორი სტრიქონის:

y = x; x=x+1;

მაგალითი 4. შევადგინოთ ინკრემენტის ოპერაციის ამსახველი პროგრამა. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 75-ზე, ხოლო შესრულების შედეგი ნახ. 76-ზე. (ასევე შეგიძლიათ იხილოთ ბმული

https://www.youtube.com/watch?v=ydcTx6idTs0&list=PLFE2CE09D83EE3E28&index=9)

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magaliti1;
2  public class mag1 {
3      public static void main(String  args []){
4          int a = 1;
5          int b = 2;
6          int c;
7          int d;
8          c = ++b;
9          d = a++;
10         c++;
11         System.out.println("a = " + a);
12         System.out.println("b = " + b);
13         System.out.println("c = " + c);
14         System.out.println("d = " + d);
15
16
17
18     }
19 }
```

ნახ. 75

შედეგი:

```
a = 2
b = 3
c = 4
d = 1
```

ნახ. 76

მაგალითი 5. შევადგინოთ ბულის ლოგიკური ოპერაციების ამსახველი პროგრამა. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 77-ზე, ხოლო შესრულების შედეგი ნახ. 78-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magaliti1;
2  public class mag1 {
3      public static void main(String  args []){
4          boolean a = true;
5          boolean b = false;
6          boolean c = a | b;
7          boolean d = a & b;
8          boolean e = a ^ b;
9          boolean f = (!a&b) | (a& !b);
10         boolean g = !a;
11         System.out.println(" a = " + a);
12         System.out.println(" b = " + b);
13         System.out.println(" a | b   " + c);
14         System.out.println(" a & b " + d);
15         System.out.println(" a ^ b "+ e);
16         System.out.println("!a&b | a&!b = " + f);
17         System.out.println(" !a =" + g);
18     }
19 }
```

ნახ. 77

შედეგი:

```
 a = true
 b = false
 a | b  true
 a & b false
 a ^ b true
!a&b | a&!b = true
 !a =false
```

ნახ. 78

მაგალითი 6. შევადგინოთ ტერნერული ოპერაციის ამსახველი პროგრამა. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 79-ზე, ხოლო შესრულების შედეგი ნახ. 80-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magaliti1;
2  public class mag1 {
3      public static void main(String  args []){
4          int i, k;
5          i = 10;
6          k = i < 0 ?- i : i;  //  i  ცვლადის აბსოლუტური მნიშვნელობა
7          System.out.println("აბსოლუტური მნიშვნელობა ");
8          System.out.println(i +"-ის ტოლია " + k);
9          i = -10; k = i < 0 ? -i : i; //  i  ცვლადის აბსოლუტური მნიშვნელობა
10         System.out.print ("აბსოლუტური მნიშვნელობა ");
11         System.out.println (i + "-ის ტოლია " + k);
12
13     }
14 }
```

ნახ. 79

შედეგი:

```
აბსოლუტური მნიშვნელობა
10-ის ტოლია 10
აბსოლუტური მნიშვნელობა -10-ის ტოლია 10
```

ნახ. 80

<u>დავალება:</u>

1. *Java-დაპროგრამების ენაზე წარმოადგინეთ შემდეგი ტექსტის ამსახველი პროგრამა:*

   „მეცნიერებმა შექმნეს ვირუსული პროგრამების ახალი სახეობა, რომელიც კავშირს მოწყობილობებს შორის ადამიანის ყურისთვის მიუწვდომელი სიგნალებით ახორციელებს. ასე და ამგვარად, მოწყობილობა ცვლის პაროლებს და სხვა პირად ინფორმაციას. ვირუსი მხოლოდ ჩაშენებულ მიკროფონებსა და დინამიკებს იყენებს მობილურ მოწყობილობებში. სპეციალისტებმა შეძლეს გადაეცათ პაროლები და სხვა მცირე მოცულობის ინფორმაცია 20 მეტრის მანძილზე“.

2. *Java-დაპროგრამების ენაზე ჩაწერეთ ვარსკვლავებით ნაჩვენები მართკუთხა სამკუთხედის ამსახველი პროგრამა:*

```
*
* *
* * *
* * * *
* * * * *
```

3. *Java-დაპროგრამების ენაზე შეადგინეთ პროგრამა, რომელიც გამოთვლის მართკუთხედის პერიმეტრსა და ფართობს, სადაც მისი არამოპირდაპირე გვერდების სიგრძეებია: a=5 სმ და b=10 სმ.*
4. *Java-დაპროგრამების ენაზე შეადგინეთ პროგრამა, რომელიც გამოთვლის კვადრატის პერიმეტრსა და ფართობს, სადაც კვადრატის გვერდის სიგრძე a=2,5 სმ-ია.*
5. *Java-დაპროგრამების ენაზე შეადგინეთ პროგრამა, რომელიც თქვენ მიერ შეტანილ ნებისმიერ სამნიშნა რიცხვში გამოთვლის ციფრთა ჯამს.*
6. *Java-დაპროგრამების ენაზე ჩაწერეთ ვარსკვლავებით ნაჩვენები მართკუთხა სამკუთხედის ამსახველი პროგრამა:*

```
* * * * *
* * * *
* * *
* *
*
```

7. *Java-დაპროგრამების ენაზე შეადგინეთ პროგრამა, რომელიც თქვენ მიერ შეტანილ ნებისმიერ სამნიშნა რიცხვში გამოთვლის ციფრთა ნამრავლს.*

3.5. მმართველი სტურქტურები

დაპროგრამების ენებში მმართველი ოპერატორები გამოიყენება პროგრამის ბრძანებების მიმდევრობის შესრულების შესაცვლელად, გადასვლებისა და განშტოებების განსახორციელებლად. **Java** პროგრამაში მმართველი ოპერატორები შეიძლება დაიყოს შემდეგ კატეგორიებად: **არჩევის ოპერატორები, ციკლის ოპერატორები და გადასვლის ოპერატორები.**

არჩევის ოპერატორები საშუალებას იძლევა გამოსახულების მნიშვნელობის ან ცვლადის მდგომარეობის მიხედვით პროგრამაში არჩეულ იქნეს ბრძანებების შესრულების სხვადასხვა განშტოება.

ციკლის ოპერატორები საშუალებას იძლევა პროგრამაში ერთი ან რამდენიმე ოპერატორის შესრულება განმეორდეს.

გადასვლის ოპერატორებით შესაძლებელია პროგრამის არაწრფივად შესრულება. განვიხილოთ ყველა ეს ოპერატორი.

3.5.1. არჩევის ოპერატორები

Java-ში ორი არჩევის ოპერატორია რეალიზებული: **if** და **switch.**

if ოპერატორი Java პროგრამის განშტოების პირობითი შერჩევის ოპერატორია. ის პროგრამის შესრულების სამართავად გამოიყენება ორი სხვადასხვა მიმართულების შტოზე. ამ ოპერატორის ჩაწერის ზოგადი ფორმა შემდეგია:

if(<პირობა>) <ოპერატორი1>; [else <ოპერატორი2>;]

აქ თითოეული <ოპერატორი> წარმოადგენს ან რომელიმე ერთ ოპერატორს ან ერთ ბლოკში (ფიგურულ ფრჩხილებში) გაერთიანებულ რამდენიმე ოპერატორს. <პირობა> ესაა ნებისმიერი გამოსახულება, რომელიც **boolean** ტიპის შედეგს აბრუნებს. **else** ოპერატორის გამოყენება აუცილებელი არაა.

if ოპერატორის მუშაობის პრინციპი შემდეგია:

თუ <პირობა> ჭეშმარიტია, მაშინ პროგრამა ასრულებს <ოპერატორი1>-ს. წინააღმდეგ შემთხვევაში იგი ასრულებს <ოპერატორი2>-ს (თუ ეს ოპერატორი არსებობს); არცერთ შემთხვევაში პროგრამა არ შეასრულებს ორივე ოპერატორს. ყურადღება უნდა გავამახვილოთ, რომ **if** ან **else** საკვანძო სიტყვების შემდეგ შეიძლება მოდიოდეს, მხოლოდ ერთი ოპერატორი. თუ საჭიროა მეტი ოპერატორის მითითება, მაშინ ისინი უნდა ჩავსვათ ერთ ბლოკში (ფიგურულ ფრჩხილებში). ზოგი პროგრამისტი ფიგურულ ფრჩხილებს ერთი ოპერატორის დროსაც იყე-

ნებს. ეს შემდგომში ოპერატორების დამატებას აადვილებს. if ოპერატორის მოქმედების საილუსტრაციოდ განვიხილოთ რამდენიმე მაგალითი.

მაგალითი 1. შევადგინოთ პროგრამა, რომელიც განსაზღვრავს მაქსიმალურ მნიშვნელობას მთელი ტიპის ნებისმიერი სამი ცვლადის მნიშვნელობიდან.

ამოცანის გადასაწყვეტად შემოვიტანოთ მთელი ტიპის ოთხი ცვლადი: a, b, c, და max. დავუშვათ, რომ ამ ცვლადებიდან მაქსიმალური მნიშვნელობა a ცვლადს აქვს (თუმცა, დაშვება ნებისმერ ცვლადზე შეგვეძლო შეგვესრულებინა). ე.ი. max=a. შემდგომ შევამოწმოთ პირობა: if(max<=b), მაშინ max=b, წინააღმდეგ შემთხვევაში მაქსიმალური მნიშვნელობის მქონე ცვლადად a ცვლადი დარჩება. რადგან ორ ცვლადს შორის უდიდესი მოძებნილია, შეგვიძლია მისი მნიშვნელობა შევადაროთ c ცვლადს, ანუ შევამოწმოთ პირობა: max<=c? თუ პირობა ჭეშმარიტია, უდიდესი მნიშვნელობა c ცვლადს ქონია, წინააღმდეგ შემთხვევაში (თუ პირობა მცდარია), წინა ორ ცვლადს შორის მაქსიმალური აღმოჩნდება უდიდესი მნიშვნელობის მქონე. პროგრამის დასასრულს, max ცვლადში წარმოდგენილი იქნება ზემოაღნიშნული სამი ცვლადის მნიშვნელობებს შორის უდიდესი, რომელსაც კონსოლზე გამოვიტანთ. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 81-ზე, ხოლო შესრულების შედეგი ნახ. 82-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package maxi;
2  //სამ ცვლადს შორის უდიდესის განსაზღვრა
3  public class maximumi {
4      public static void main (String args[]){
5          int a=12, b=25, c=10, max;
6          max=a;
7          if(max<=b) max=b;
8          if(max<=c) max=c;
9          System.out.println("a=" + a);
10         System.out.println("b=" + b);
11         System.out.println("c=" + c);
12         System.out.println("max=" + max);
13     }
14 }
```

ნახ. 81

შედეგი:

```
a=12
b=25
c=10
max=25
```

ნახ. 82

მაგალითი 2. შევადგინოთ პროგრამა, რომელიც დაადგენს უნაშთოდ იყოფა თუ არა მთელი ტიპის ნებისმიერი სამნიშნა რიცხვი საკუთარ ციფრთა ჯამზე.

ამოცანის გადასაწყვეტად შემოვიტანოთ შემდეგი ცვლადები: a, b, c, s და x=345. ჩვენთვის უკვე ცნობილია, რომ მთელრიცხვა გაყოფის შედეგი მთელი რიცხვია, ხოლო მოდულით გაყოფის ოპერაცია (%) იძლევა გაყოფის შედეგად მიღებულ ნაშთს. შესაბამისად, გამოსახულება: a=x/100 ნიშნავს, რომ a=345/100=3 (a ცვლადში ინახება ასეულების აღმნიშვნელი ციფრი). ანალოგიურად, b=x/10%10, ანუ b=345/10%10=34%10=4 (b ცვლადში ინახება ათეულების აღმნიშვნელი ციფრი). c=x%10, ე.ი. c=345%10=5 (c ცვლადში ინახება ერთეულების აღმნიშვნელი ციფრი). s ცვლადში დავაგროვოთ რიცხვის შემადგენელი ციფრების ჯამი, ანუ s=a+b+c. ე.ი. s=3+4+5=12. ახლა შევამოწმოთ პირობა: x%s=0? ანუ x რიცხვის თავის ციფრთა ჯამზე გაყოფის შედეგად ნაშთი თუ ნოლის ტოლია, ე.ი. ეს რიცხვი უნაშთოდ იყოფა თავის ციფრთა ჯამზე. წინააღმდეგ შემთხვევაში - უნაშთოდ არ იყოფა. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 83-ზე, ხოლო შესრულების შედეგი ნახ. 84-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magalitebi;
2  //იყოფა თუ არა რიცხვი თავის ციფრთა ჯამზე
3  public class Struqturebi {
4      public static void main(String args[]){
5          int a, b, c, s, x=345;
6          System.out.println("x=" + x);
7          a=x/100;
8          b=x/10%10;
9          c=x%10;
10         s=a+b+c;
11         System.out.println("ციფრთა ჯამი =" + s);
12         if(x%s==0)
13         {
14             System.out.println("შედეგი=" + x%s);
15             System.out.println("რიცხვი უნაშთოდ იყოფა თავის ციფრთა ჯამზე");
16         }
17         else
18             System.out.println("რიცხვი უნაშთოდ  არ იყოფა თავის ციფრთა ჯამზე");
19     }
20
21 }
```

ნახ. 83

შედეგი:

```
x=345
ციფრთა ჯამი =12
რიცხვი უნაშთოდ  არ იყოფა თავის ციფრთა ჯამზე
```

ნახ. 84

პროგრამებში ხშირად გვხვდება ერთმანეთში ჩალაგებული **if** ოპერატორები. ამ შემთხვევაში უნდა გვახსოვდეს, რომ **else** ოპერატორი ყოველთვის უკავშირდება მის უახლოეს **if** ოპერატორს, რომელიც იმავე ბლოკში მდებარეობს და ჯერ არაა დაკავშირებული სხვა **else** ოპერატორთან. მაგალითად:

```
if(i == 10) {
if(j < 20) a =b;
if(k > 100) c = d; // ეს if ოპერატორი
else a = c; // დაკავშირებულია ამ else-სთან
}
else a = d; // ეს else ოპერატორი დაკავშირებულია if(i == 10)-თან.
```

როგორც კომენტარებიდანაც ჩანს, ბოლო **else** ოპერატორი არ არის დაკავშირებული **if(j<20)** ოპერატორთან, ვინაიდან ის არ იმყოფება იმავე ბლოკში (მიუხედავად იმისა, რომ **if**-ის უახლოესი ოპერატორია, რომელთანაც **else** ოპერატორი ჯერ არაა დაკავშირებული). ბლოკში ბოლო **else** ოპერატორი დაკავშირებულია **if(k > 100)** ოპერატორთან, ვინაიდან იგი უახლოესია ბლოკის შიგნით.

დაპროგრამებაში გავრცელებულია კონსტრუქცია, რომელიც ჩალაგებული **if** ოპერატორების მიმდევრობისაგან აიგება, მას მრავალრგოლიან **if–else-if** სტრუქტურას უწოდებენ და შემდეგი სახე აქვს:

```
if (<პირობა>)
<ოპერატორი>;
else if (<პირობა>)
<ოპერატორი>;
 else if (<პირობა>)
<ოპერატორი>;
else
<ოპერატორი>;
```

if ოპერატორი სრულდება მიმდევრობით, ზემოდან ქვემოთ. როგორც კი **if** ოპერატორის მმართავი ერთ-ერთი პირობა გახდება **true** (ჭეშმარიტი), პროგრამა შეასრულებს ამ **if** ოპერატორთან დაკავშირებულ ოპერატორს და გამოტოვებს დანარჩენ მრავალრგოლიანი სტრუქტურის ნაწილს. თუ არცერთი პირობა არ სრულდება, პროგრამა შეასრულებს ბოლო, **else** ოპერატორს. ე.ი., თუ ყველა სხვა შემოწმების პირობა იძლევა მცდარ შედეგს, პროგრამა ასრულებს ბოლო **else** ოპერატორს. თუ ბოლო **else** ოპერატორი მითითებული არ არის და ყველა წინა

შემოწმების შედეგია **false** (მცდარი), მაშინ პროგრამა ამ კონსტრუქციიდან არცერთ მოქმედებას არ შეასრულებს.

მაგალითი 3. შევადგინოთ პროგრამა, რომელიც რიცხვის ნიშანს განსაზღვრავს.

ამოცანის გადასაწყვეტად, შემოვიტანოთ x ცვლადი, თუ მისი მნიშვნელობა 0-ზე მეტი აღმოჩნდება, ცხადია, x დადებითი რიცხვია, თუ მისი მნიშვნელობა ნოლზე ნაკლები აღმოჩნდება, x უარყოფითი რიცხვია და თუ ორივე პირობა მცდარია, ე.ი. x=0.

აქამდე პროგრამებში ცვლადებს მინიჭების ოპერაციით ვაძლევდით კონკრეტულ მნიშვნელობებს. ამჯერად, ცვლადებზე საწყისი მნიშვნელობების მინიჭება პროგრამის შესრულებაზე გაშვების დროს განვახორციელოთ. ამისათვის საჭიროა, ჩვენს პროგრამაში კლასის აღწერამდე მოვახდინოთ საჭირო კლასის იმპორტირება შემდეგი ბრძანების გამოყენების გზით: import java.util.Scanner; შემდეგ კლასში შევქმნათ Scanner კლასის ობიექტი: Scanner ob1=new Scanner(System.in); ბრძანებით. და ბოლოს, x ცვლადს კონკრეტული მნიშვნელობა მივანიჭოთ ბრძანებით: x=ob1.nextInt(); ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 85-ზე, ხოლო შესრულების შედეგი ნახ. 86-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magalitebi;
2  import java.util.Scanner;
3  //რიცხვის ნიშნის განსაზღვრა
4  public class Struqturebi {
5      public static void main(String args[]){
6          int x;
7          Scanner ob1=new Scanner(System.in);
8          System.out.print("x=");
9          x=ob1.nextInt();
10         if(x>0)
11             System.out.print("x დადებითი რიცხვია");
12         else
13         {
14             if(x<0)
15                 System.out.print("x უარყოფითი რიცხვია");
16             else
17                 System.out.print("x=0");
18         }
19     }
20
21 }
```

ნახ. 85

შედეგი:

```
x=-30
x უარყოფითი რიცხვია
```

ნახ. 86

switsh ოპერატორით წარმოებს პროგრამის შესრულების განშტოება. განშტოება ხდება მმართავი გამოსახულების გამოთვლის შედეგად მიღებული მნიშვნელობის მიხედვით. **switch** ოპერატორის ზოგადი ფორმა შემდეგია:

```
switch (<გამოსახულება>) {
case <მნიშვნელობა1>:
// ოპერატორების მიმდევრობა
 break;
 case <მნიშვნელობა 2>:
// ოპერატორების მიმდევრობა
break;
 case <მნიშვნელობა N>:
// ოპერატორების მიმდევრობა
 break;
default:
// ოპერატორები
}
```

<გამოსახულება> უნდა იყოს **byte, short, int** ან **char** ტიპის. <მნიშვნელობა>-ში მითითებული ყოველი ტიპი <გამოსახულება>-ში მითითებული ტიპის თანადი (თავსებადი) ტიპი უნდა იყოს. **case**-ს თითოეული <მნიშვნელობა> უნიკალური კონსტანტა (მუდმივა და არა ცვლადი) უნდა იყოს. ასევე არაა დაშვებული **case**-ის მნიშვნელობების დუბლირება.

switch ოპერატორის მუშაობის პრინციპი შემდეგია:

<გამოსახულება>-ის მნიშვნელობა დარდება **case** ოპერატორებში მითითებულ თითოეულ კონსტანტას. თუ მოხდება რომელიმესთან თანხვედრა, მაშინ შესრულება გრძელდება ამ **case**-ის შემდეგ ჩაწერილი ოპერატორიდან. თუ არცერთი კონსტანტის მნიშვნელობა არ დაემთხვევა <გამოსახულების> მნიშვნელობას, მაშინ პროგრამა ასრულებს **default** ოპერატორის შემდეგ ჩაწერილ ოპერატორებს. თუმცა, ამ ოპერატორის გამოყენება აუცილებელი არაა. **default** ოპერატორის არარსებობის და გამოსახულების არცერთ კონსტანტაზე დამთხვევის შემთხვევაში, პროგრამა **switch** კონსტრუქციაში არაფერს ასრულებს. პროგრამის შესრულება გრძელდება **switch** კონსტრუქციის შემდეგი ოპერატორიდან.

break ოპერატორი, რომელიც **switch** ოპერატორის შიგნითაა მითითებული, გამოიყენება ოპერატორების შესრულების მიმდევრობის წყვეტისათვის. როგორც კი პროგრამა **break** ოპერატორთან მივა, შესრულება გაგრძელდება მთლიანი **switch** ოპერატორის შემდეგი ოპერატორიდან. იგი ფაქტიურად ახორციელებს **switch** ოპერატორიდან „გამოსვლას“.

მაგალითი 4. შევადგინოთ პროგრამა, რომელიც **switch** ოპერატორის გამოყენებით გამოთვლის გამოსახულებათა შემდეგი სისტემის მნიშვნელობას: $y = \begin{cases} x+5, & \text{თუ } x=2 \\ x-10, & \text{თუ } x=3 \\ 2*x, & \text{თუ } x=4 \end{cases}$

წარმოდგენილ ამოცანაში თუ x=2, მაშინ y ფუნქცია პირველი ფორმულით გამოითვლება: y=x+5. თუ x=3, მაშინ მეორე ფორმულით: y=x-10, ხოლო თუ x=4, - ბოლო ფორმულით: y=2*x. თუ x ცვლადის მნიშვნელობა არცერთ ჭდეს (case) არ დაემთხვევა, პროგრამა გამოიტანს შეტყობინებას, რომ „ამონახსნი არ გვაქვს“. ამ უკანასკნელს default ჭდე უზრუნველყოფს.

აქ switch ოპერატორის მმართველი გამოსახულების როლში x ცვლადი გვევლინება, რომელიც ამავდროულად, მოცემული ფუნქციის არგუმენტია.

ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 87-ზე, ხოლო შესრულების შედეგი ნახ. 88-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magalitebi;
2  import java.util.Scanner;
3  //გამოსახულებათა სისტემის ამოხსნა
4  public class Struqturebi {
5      public static void main(String args[]){
6          int x, y;
7          Scanner ob1=new Scanner(System.in);
8          System.out.print("x=");
9          x=ob1.nextInt();
10         switch(x){
11         case 2: y=x+5;  System.out.println ("y=" +y); break;
12         case 3: y=x-10;  System.out.println ("y=" +y); break;
13         case 4: y=2*x; System.out.println ("y=" +y); break;
14         default:
15             System.out.println("ამონახსნი არ გვაქვს");
16         }
17     }
18
19 }
```

ნახ. 87

შედეგი:

```
x=4
y=8
```

ნახ. 88

ზოგჯერ საჭიროა რამდენიმე **case** ოპერატორის გამოყენება **break** ოპერატორით გამოყოფის გარეშე. განვიხილოთ შესაბამისი **მაგალითი**.

მაგალითი 4. შევადგინოთ პროგრამა, რომელიც month ცვლადის რიცხვითი მნიშვნელობის მიხედვით განსაზღვრავს თუ წელიწადის (season) რა დროა. კერძოდ, თუ month=12, 1 ან 2-ს, ე.ი ზამთარია. თუ month=3, 4 ან 5-ს, - გაზაფხულია, თუ month=6, 7 ან 8-ს, - ზაფხული, ხოლო თუ month=9, 10 ან 11-ს, - შემოდგომა. ზემოთ წარმოდგენილი რიცხვებისგან განსხვავებული რიცხვის შეტანის შემთხვევაში, პროგრამა გამოიტანს შეტყობინებას: „არარსებული თვე“. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 89-ზე, ხოლო შესრულების შედეგი ნახ. 90-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package magalitebi;
2  import java.util.Scanner;
3  //წელიწადის დროის განსაზღვრა
4  public class Struqturebi {
5      public static void main(String args[]){
6          int month, y;
7          Scanner ob1=new Scanner(System.in);
8          System.out.print("month=");
9          month=ob1.nextInt();
10         String season;
11         switch (month) {
12         case 12:
13         case 1:
14         case 2:
15           season = "ზამთარი";
16         break;
17         case 3:
18         case 4:
19         case 5:
20           season = "გაზაფხული";
21         break;
22         case 6:
23         case 7:
24         case 8:
25           season = "ზაფხული";
26         break;
27         case 9:
28         case 10:
29         case 11:
30           season = "შემოდგომა";
31         break;
32         default:
33           season = "არარსებული თვე";
34         }
35         System.out.println("მე-"  + month + " თვე არის "  + season + ".");
36         }
37
38     }
```

ნახ. 89

შედეგი:

```
month=12
მე-12 თვე არის ზამთარი.
```

ნახ. 90

არჩევის ოპერატორებთან დაკავშირებული ვიდეო მასალა შეგიძლიათ იხილოთ ბმულზე: https://www.thenewboston.com/videos.php?cat=31 ვიდეო-ფაილები 10, 11, 12, 19.

3.5.2. ციკლის ოპერატორები

როდესაც პროგრამული კოდის ეს თუ ის ფრაგმენტი რამდენჯერმე მეორდება, აღნიშნულ პროცესს პროგრამისტები **ციკლს** უწოდებენ. Java-ში არსებობს მარტივი ციკლის სამი სახე: **while**, **do/while** და **for**.

while ოპერატორი პროგრამებში ხშირად გამოიყენება. ის ციკლის ტანში არსებულ ოპერატორს, ან ოპერატორთა ჯგუფს იმეორებს მანამდე, სანამ ციკლის პირობა მცდარი არ გახდება. **while** ციკლის ოპერატორის ჩაწერის ფორმა შემდეგია:

```
while (<პირობა>) {
 //ციკლის ტანი;
}
```

<პირობა> შეიძლება იყოს ბულის ტიპის ნებისმიერი გამოსახულება. აღნიშნული ციკლი მუშაობს მანამ, სანამ <პირობა> ჭეშმარიტია. როგორც კი პირობა აღმოჩნდება მცდარი, პროგრამის მუშაობა გრძელდება ციკლის ტანის გარეთ არსებული მომდევნო ოპერატორიდან.

ციკლის ტანის აღმნიშვნელი ფიგურული ფრჩხილების ({ }) გამოყენება აუცილებელია, თუ მასში ერთზე მეტი ოპერატორია მოთავსებული. რადგან **while** ციკლი თავის პირობით გამოსახულებას ამოწმებს ციკლის დასაწყისში, ამიტომ, თუ გამოსახულების შედეგი მცდარია, ციკლი თავიდანვე არ შესრულდება.

მაგალითი 5. შევადგინოთ პროგრამა, რომელიც ევკლიდეს ალგორითმის საფუძველზე განსაზღვრავს ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფისა და უმცირესი საერთო ჯერადის მნიშვნელობებს (აღნიშნული ალგორითმი წინა თავებშია განხილული). ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 91-ზე, ხოლო შესრულების შედეგი ნახ. 92-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
package magalitebi;
import java.util.Scanner;
//ევკლიდეს ალგორითმი
public class MyWork {
    public static void main(String args []){
        long a, b, s;
        Scanner ob1=new Scanner(System.in);
        System.out.print("შეიტანეთ a ცვლადის მნიშვნელობა ");
        a=ob1.nextInt();
        System.out.print("შეიტანეთ b ცვლადის მნიშვნელობა ");
        b=ob1.nextInt();
        s=a*b;
        while(a!=b){
            if(a>b) a-=b;
            else b-=a;
        }
        System.out.println("უდიდესი საერთო გამყოფი= " + a);
        System.out.println("უდიდესი საერთო ჯერადი= " + s/a);
    }
}
```

ნახ. 91

შედეგი:

```
შეიტანეთ a  ცვლადის მნიშვნელობა 35
შეიტანეთ b  ცვლადის მნიშვნელობა 60
უდიდესი საერთო გამყოფი=  5
უდიდესი საერთო ჯერადი=  420
```

ნახ. 92

პროგრამის წერისას, ზოგჯერ იქმენა სიტუაციები, როდესაც საჭიროა ციკლის ტანის ერთხელ მაინც შესრულება, მაშინაც კი, თუ ციკლის მმართველი გამოსახულების შედეგი მცდარია. ამ შემთხვევაში **do-while** ციკლი გამოიყენება, რომლის ჩაწერის ფორმა შემდეგია:

do {

//ციკლის ტანი;

} while(<პირობა>);

ამგვარად, **do-while** ციკლში, ჯერ მიყოლებით სრულდება ციკლის ტანში არსებული ოპერატორები და ბოლოს მომწმდება <პირობა>. ციკლი მუშაობს მანამ, სანამ პირობა ჭეშმარიტია. როგორც კი პირობა ხდება მცდარი, ციკლური პროცესი წყდება და პროგრამის შესრულება გრძელდება ციკლის ტანის გარეთ არსებული მომდევნო ოპერატორიდან. <პირობის> შედეგი, ამ შემთხვევაშიც, ბულის ტიპის უნდა იყოს.

მაგალითი 6. შევადგინოთ პროგრამა, რომელიც **y=*ax*+*b*** გამოსახულების მიხედვით წარმოგვიდგენს **y** ფუნქციისა და ***x*** არგუმენტის მნიშვნელობათა ცხრილს, სადაც ***x*∈[1;10]** ინტერვალში H=1,5-ის ტოლი ბიჯით იცვლება. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 93-ზე, ხოლო შესრულების შედეგი ნახ. 94-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

ნახ.93

```
package magalitebi;
import java.util.Scanner;
//სისტემის ამოხსნა
public class MyWork {
    public static void main(String args []){
        double y, x=1, a, b, H=1.5;
        Scanner ob1=new Scanner(System.in);
        System.out.print("შეიტანეთ a  ცვლადის მნიშვნელობა ");
        a=ob1.nextInt();
        System.out.print("შეიტანეთ b  ცვლადის მნიშვნელობა ");
        b=ob1.nextInt();
        do{
            y=a*x+b;
            System.out.println("x=" +x + "\ty=" +y);
            x+=H;
        }while(x<=10);

    }
}
```

შედეგი:

```
შეიტანეთ a  ცვლადის მნიშვნელობა 1
შეიტანეთ b  ცვლადის მნიშვნელობა 2
x=1.0   y=3.0
x=2.5   y=4.5
x=4.0   y=6.0
x=5.5   y=7.5
x=7.0   y=9.0
x=8.5   y=10.5
x=10.0  y=12.0
```

ნახ. 94

Java-ში არსებობს **for** ციკლის ორი ფორმა. პირველი - ტრადიციული ფორმა, რომელიც თავიდანვე იქნა რეალიზებული ენაში. მეორე - ახალი ფორმა **for-each**.

ტრადიცული **for** ციკლის ოპერატორის ზოგად ფორმას შემდეგი სახე აქვს:

```
for(<ინიციალიზაცია>; <პირობა>; <იტერაცია>) {

// ციკლის ტანი;

}
```

თუ ციკლში მხოლოდ ერთი ოპერატორის განმეორება ხდება, შესაძლებელია ფიგურული ფრჩხილების გამოტოვება.

for ციკლის მუშაობის პრინციპი შემდეგია: ციკლის პირველ გაშვებაზე, პროგრამა ასრულებს ციკლის ინიციალიზაციის ნაწილს. ზოგადად, ესაა გამოსახულება, რომელიც საწყის მნიშვნელობას ანიჭებს ციკლის მმართველ ცვლადს, რომელიც, ამავდროულად, ციკლის მთვლელის როლში გვევლინება. აღსანიშნავია, რომ ინიციალიზაციის გამოსახულების შესრულება ხდება მხოლოდ ერთხელ! შემდეგ პროგრამა გამოითვლის <პირობას>, რომელიც ბულის ტიპის გამოსახულება უნდა იყოს. როგორც წესი, მმართველი ცვლადი დარდება მიზნობრივ მნიშვნელობას. თუ ბულის ტიპის გამოსახულება ჭეშმარიტია, პროგრამა ასრულებს ციკლის ტანს. წინააღმდეგ შემთხვევაში, ციკლის შესრულება წყდება. ციკლის პირობის ჭეშმარიტების შემთხვევაში, ციკლის ტანის შესრულების ყოველი გავლის შემდეგ სრულდება იტერაციული ნაწილი. როგორც წესი, ესაა გამოსახულება, რომელიც ზრდის ან ამცირებს ციკლის მთვლელის მნიშვნელობას. შემდეგ პროგრამა იმეორებს ციკლს: ყოველი გავლის წინ ჯერ ხდება პირობითი გამოსახულების გამოთვლა, შემდეგ სრულდება ციკლის ტანი და იტერაციული ნაწილი. პროცესი გრძელდება მანამ, სანამ ციკლის პირობა არ გახდება მცდარი.

მაგალითი 7. შევადგინოთ პროგრამა, რომელიც ითვლის შემდეგი გამოსახულების მნიშვნელობას:

$s = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$; სადაც x=1, ხოლო n=10.

დასმული ამოცანის ამოხსნის მიზნით შემოვიტანოთ შემდეგი საწყისი მნიშვნელობების მქონე ცვლადები: p=1 ცვლადი, რომელიც გამოსახულებაში არსებული x ცვლადის (მრიცხველის) ხარისხებს p=p*x ფორმულით დააგროვებს; F=1 ცვლადი, რომელიც ნატურალური რიცხვების ფაქტორიალების მნიშვნელობებს გამოითვლის ფორმულით: F=F*i; S=1 ცვლადი, რომელსაც მივანიჭეთ მოცემული მწკრივის პირველი წევრის მნიშვნელობა და მწკრივის წევრთა ჯამს დააგროვებს ფორმულით: S=S+P/F; n=10 ცვლადი, რომელიც მიუთითებს მწკრივის წევრების რაოდენობას და for ციკლის მმართველი პარამეტრი (ციკლის მთვლელი) i. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 95-ზე, ხოლო შესრულების შედეგი ნახ. 96-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
package magalitebi;
//მწკრივის წევრთა ჯამის გამოთვლა
public class MyWork {
    public static void main(String args []){
        double x=1, F=1, S=1, P=1, n=10;
        for(int i=1; i<n; i++){
            P*=x;
            F*=i;
            S+=P/F;
        }
        System.out.println("S=" + S);

      }
    }
```

ნახ. 95

შედეგი:

```
S=2.7182815255731922
```

ნახ. 96

for ციკლი წარმოადგენს ციკლის ერთადერთ სახეს, სადაც შესაძლებელია ციკლის მმართველი პარამეტრის (მთვლელის) ინიციალიზაცია მოვახდინოთ ციკლშივე, ცხადია, თუ აღნიშნული პარამეტრი პროგრამის სხვა ნაწილში არ გამოიყენება.

მმართველი ცვლადის ციკლის ტანში გამოცხადების შემთხვევაში, უნდა გვახსოვდეს, რომ ამ ცვლადის განსაზღვრის არე და არსებობის დრო მთლიანად ემთხვევა **for** ციკლის განსაზღვრის არესა და არსებობის დროს. ანუ, ცვლადის განსაზღვრის არე შემოფარგლულია **for**

ციკლით და მის გარეთ ცვლადი წყვეტს არსებობას. თუ მმართველი ცვლადი პროგრამის სხვა ადგილებშიცაა საჭირო, ის ციკლის შიგნით არ უნდა გამოცხადდეს.

თანამედროვე დაპროგრამების ენებში სულ უფრო დიდ გამოყენებას ჰპოვებს ციკლების **for-each** (თითოეულისათვის) კონცეფცია. ასეთი ციკლის დანიშნულებაა მკაცრი თანმიმდევრობით მოხდეს განმეორებადი მოქმედებები ობიექტების კოლექციის მიმართ, რომლებიც ვთქვათ, მასივადაა წარმოდგენილი.

for-each ციკლის ზოგადი ფორმა შემდეგია:

for (<ტიპი> <იტერაციული ცვლადი> : <კოლექცია>) <ოპერატორების ბლოკი>

<ტიპი> - ესაა კოლექციაში შემავალ მონაცემთა ტიპი, ხოლო <იტერაციული ცვლადი> – იტერაციული ცვლადის სახელი, რომელიც კოლექციის პირველი ელემენტიდან ბოლომდე მიმდევრობით იღებს მნიშვნელობებს.

<კოლექცია> იმ კოლექციას წარმოადგენს, რომლის მიხედვითაც ციკლი ხორციელდება.

ამრიგად, პროგრამა ციკლის თითოეულ იტერაციაზე ამოიღებს კოლექციის შემდგომ ელემენტს და ინახავს მას ცვლადში <იტერაციული ცვლადი>. ციკლი გრძელდება მანამ, სანამ არ იქნება მიღებული იტერაციის ყველა ელემენტი.

ვინაიდან იტერაციული ელემენტი კოლექციიდან იღებს მნიშვნელობებს, ამიტომ, <ტიპი> უნდა ემთხვეოდეს (ან თავსებადი იყოს) კოლექციაში არსებული ელემენტების ტიპს. საჭიროა ხაზი გავუსვათ **for-each** ციკლის ერთ მნიშვნელოვან გარემოებას. მისი იტერაციული ცვლადი წარმოადგენს ცვლადს, განკუთვნილს „მხოლოდ კითხვისთვის“. **for-each** ციკლის გამოყენების ნიმუშებს წინამდებარე სახელმძღვანელოს შემდომ თავებში შემოგთავაზებთ.

სხვა ენების ანალოგიურად, Java-ში დასაშვებია ერთმანეთში ჩალაგებული ციკლების ორგანიზება. ასეთ შემთხვევაში, ციკლები შემდეგნაირად მუშაობს: თავდაპირველად, გარე ციკლის მთვლელი იღებს საწყის მნიშვნელობას და შემდეგ მუშაობას იწყებს შიდა ციკლი, სანამ მისი (შიდა ციკლის) პირობა ჭეშმარიტია. შიდა ციკლის პირობის დარღვევის შემდეგ, ვუბრუნდებით გარე ციკლს, სადაც მიმდინარეობს მისი მთვლელის მნიშვნელობის ცვლილება, ციკლის პირობის შემოწმება და თუ ის ჭეშმარიტი აღმოჩნდება, ხელახლა იწყებს მუშაობას შიდა ციკლი. და ა.შ., სანამ გარე ციკლის პირობა არ აღმოჩნდება მცდარი. ამ შემთხვევაში, ციკლური პროცესები წყდება და მართვა გადაეცემა გარე ციკლის ტანის გარეთ არსებულ მომდევნო ოპერატორს პროგრამაში.

მაგალითი 8. შევადგინოთ პროგრამა, რომელიც ჩალაგებული (რთული ციკლების) გამოყენებით წარმოადგენს მართკუთხა სამკუთხედის ამსახველ ჰისტროგრამას.

ამოცანის გადაწყვეტის მიზნით, გარე ციკლის მთვლელი i ითვლის სტრიქონების რაოდენობას და ასევე, ასრულებს ახალ სტრიქონზე გადასვლის ბრძანებებს, ხოლო შიდა ციკლის

საშუალებით ადგილი აქვს სტრიქონების მიხედვით „*“ (ვარსკვლავის) სიმბოლოების ეკრანზე გამოტანას. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 97-ზე, ხოლო შესრულების შედეგი ნახ. 98-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
package magalitebi;
//პისტროგრამის ბეჭდვა
public class MyWork {
    public static void main(String args []){
        for(int i=1; i<=10; i++){
            for(int k=i; k<=10; k++)
                System.out.print("* ");
        System.out.println();
        }

    }
}
```

ნახ. 97

შედეგი:

```
* * * * * * * * * *
* * * * * * * * *
* * * * * * * *
* * * * * * *
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

ნახ. 98

break ოპერატორი ციკლური პროცესის შეწყვეტას განაპირობებს. შესაბამისად, ციკლში არსებული ბრძანებები აღარ სრულდება და მართვა ციკლის ტანის გარეთ არსებულ მომდევნო ოპერატორს გადაეცემა პროგრამაში.

მაგალითი 9. შევადგინოთ პროგრამა, რომელმაც, წესით 2-დან დაწყებული 50-ის ჩათვლით კონსოლზე ყველა ლუწი რიცხვი უნდა გამოიტანოს, მაგრამ თუ ციკლის მთვლელის მნიშვნელობა აღმოჩნდება 20-ის ტოლი, ციკლური პროცესი უნდა შეწყდეს და შესაბამისად, კონსოლზე მხოლოდ 2-დან 20-მდე ლუწი რიცხვები გამოიტანება.

ცხადია, ამოცანის გადაწყვეტა break ოპერატორის გამოყენებას საჭიროებს, ხოლო ლუწი რიცხვების მისაღებად ყოველ იტერაციაზე საკმარისია მთვლელის საწყისი მნიშვნელობის 2-ით

გაზრდა. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 99-ზე, ხოლო შესრულების შედეგი ნახ. 100-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
package magalitebi;
//ლუწი რიცხვების ბეჭდვა
public class MyWork {
    public static void main(String args []){

            for(int i=2; i<=50; i+=2){

            if(i==20) break;
                System.out.print(i + " ");

            }
        }
    }
```

ნახ. 99

შედეგი:

```
2 4 6 8 10 12 14 16 18
```

ნახ. 100

ზოგჯერ, კონკრეტულ იტერაციაში საჭიროა ციკლის გაგრძელება მის ტანში დანარჩენი ოპერატორების შესრულების გარეშე მოხდეს. ფაქტიურად, ესაა ციკლის ტანიდან მის ბოლოში გადასვლა. აღნიშნული მოქმედების შესასრულებლად ოპერატორი **continue** გამოიყენება. **while** და **do-while** ციკლებში **continue** ოპერატორი მართვის გადაცემას უშუალოდ ციკლის შესრულების პირობაზე იწვევს. **for** ციკლში მართვა ციკლის თავში იტერაციულ ნაწილს გადაეცემა, ხოლო შემდეგ პირობით ნაწილს. სამივე ამ ციკლში ნებისმიერი შუალედური ოპერაცია გამოიტოვება.

მაგალითი 10. შევადგინოთ პროგრამა, რომელიც **continue** ოპერატორის გამოყენებით ერთიდან 15-ის ჩათვლით დაბეჭდავს ყველა კენტ რიცხვს 5-ის 7-ის და 11-ის გარდა. პროგრამის რეალიზაციაში მოკლე ლოგიკური „ან“ (||) ოპერაციები გამოვიყენეთ საჭირო პირობების გაერთიანების მიზნით. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 101-ზე, ხოლო შესრულების შედეგი ნახ. 102-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
package magalitebi;
//კენტი რიცხვების ბეჭდვა
public class MyWork {
    public static void main(String args []){

            for(int i=1; i<=15; i+=2){

            if(i==5 || i==7 || i==11) continue;
                System.out.print(i + " ");

            }
        }
    }
```

ნახ. 101

შედეგი:

```
1 3 9 13 15
```

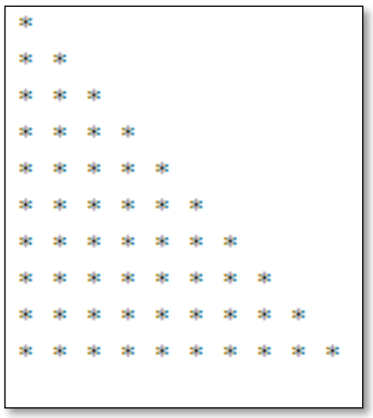
ნახ. 102

ციკლის ოპერატორებთნ დაკავშირებული ვიდეო მასალა შეგიძლიათ იხილოთ ბმულზე: https://www.thenewboston.com/videos.php?cat=31 ვიდეო-ფაილები: 13, 22, 24

დავალება:

1. შეადგინეთ პროგრამა, რომელიც double ტიპის ნებისმიერ სამ ცვლადს შორის განსაზღვარვს მინიმალურს.
2. შეადგინეთ პროგრამა, რომელიც მთელი ტიპის ნებისმიერ სამნიშნა რიცხვში გამოთვლის ციფრთა ნამრავლს, კონსოლზე გამოიტანს შედეგს და შეტყობინებას ლუწია თუ კენტი მიღებული შედეგი.

3. ციკლის ნებისმიერი ოპერატორის გამოყენებით შეადგინეთ პირველი 15 ნატურალური რიცხვის ჯამის, ნამრავლისა და საშუალო არითმეტიკულის გამოთვლის პროგრამა.
4. შეადგინეთ შემდეგი მწკრივის წევრების ჯამის გამოთვლის პროგრამა: $s = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{100}$
5. შეადგინეთ $y = x^2 + 5$ ფუნქციის მნიშვნელობათა ცხრილის გამოთვლის პროგრამა, სადაც x არგუმენტი [1; 10] ინტერვალში h=0,5 ბიჯით იცვლება.
6. შეადგინეთ პროგრამა, რომელიც წარმოადგენს ყველა სამნიშნა რიცხვს, რომელიც უნაშთოდ იყოფა თავის ციფრთა ჯამზე.
7. ჩალაგებული (რთული) ციკლების გამოყენებით შეადგინეთ ნახ.103-ზე მოცემული ჰისტროგრამის ამსახველი პროგრამა:



ნახ. 103

3.6. მასივები

მასივი წარმოადგენს ერთი და იმავე ტიპის მონაცემთა ერთობლიობას, რომლებზეც მიმართვა საერთო სახელის გამოყენებით ხდება. Java ენაში გამოიყენება როგორც ერთ, ისე მრავალგანზომილებიანი ნებისმიერი ტიპის მასივები. მასივის კონკრეტულ ელემენტზე მიმართვა ხდება მისი ინდექსის საშუალებით. Java-ში მასივის ელემენტების ინდექსაცია ნულიდან იწყება. ინდექსი მთელი ტიპის მონაცემია, ან გამოსახულება, რომლის შედეგი მთელი ტიპისაა. მასივის ელემენტების რაოდენობა განსაზღვრავს მის **ზომას,** ხოლო ინდექსების რაოდენობა – **განზომილებას**.

3.6.1. ერთგანზომილებიანი მასივები

ერთგანზომილებიანი მასივის შესაქმნელად ჯერ უნდა შეიქმნას საჭირო ტიპის მასივის ცვლადი.

ერთგანზომილებიანი მასივის გამოცხადების ზოგადი ფორმა შემდეგია:

<ტიპი> <მასივის სახელი>[ ];

აქ <ტიპი> წარმოადგენს მასივის შემადგენელი ელემენტების ტიპს. მაგალითად, შემდეგი ოპერატორით ხდება **array** სახელის მქონე მასივის გამოცხადება, რომლის ელემენტები **int** ტიპისაა:

```
int array[ ];
```

ამ გამოცხადებით ჯერ მასივი არ შექმნილა. **array** მასივის ფაქტიური მნიშვნელობა ჯერჯერობით არის **null**, რომელიც წარმოადგენს მასივს მნიშვნელობის გარეშე. **array** მასივის ცვლადის რეალური ფიზიკური მთელი რიცხვების მასივთან დასაკავშირებლად საჭიროა **new** ოპერაციით მეხსიერების გამოყოფა და მისი მისამართის ამ ცვლადზე მინიჭება.

მასივის გამოცხადების სრულყოფილი ზოგადი ფორმა შემდეგია:

<მასივის სახელი> = new <ტიპი>[<ზომა>] ;

<ტიპი> აღწერს დარეზერვებული მონაცემების მეხსიერების ტიპს. <ზომა> მიუთითებს მასივში შემავალი ელემენტების რაოდენობას, ხოლო <მასივის სახელი> მასივთან დაკავშირებული ცვლადია. ანუ, **new** ოპრაციით მეხსიერების გამოსაყოფად საჭიროა ელემენტების ტიპისა და რაოდენობის მითითება. მასივის ელემენტებისათვის **new** ოპერაციით გამოყოფილი მეხსიერების ინიციალება მოხდება ნულოვანი მნიშვნელობებით.

მაგალითად, ჩანაწერით:

```
array=new int [10];
```

ხდება 10 ელემენტიანი მასივისათვის მეხსიერების რეზერვირება და მისი დაკავშირება **array** მასივის სახელთან.

ამრიგად, მასივის შექმნა ორსაფეხურიანი პროცესია. ჯერ ცხადდება საჭირო ტიპის მასივის ცვლადი, ხოლო შემდეგ, **new** ოპერაციით გამოიყოფა მასივის შესანახი მეხსიერება და მისი მისამართი ენიჭება მასივის ცვლადს. შესაბამისად, Java-ში ყველა მასივი დინამიურად იქმნება.

მასივის გამოცხადებისა და შექმნის შემდეგ, შესაძლებელია მის თითოეულ ელემენტს მივმართოთ ინდექსით, რომელიც კვადრატულ ფრჩხილებში უნდა იყოს მოთავსებული. მასივის ინდექსაცია 0-დან იწყება.

მასივის ცვლადის გამოცხადება შეიძლება მისთვის მეხსიერების გამოყოფასთან გავაერთიანოთ:

```
int array[ ] = new int[10];
```

ასევე შესაძლებელია მასივის ინიციალიზაცია მისი გამოცხადებისას. ეს პროცესი ჰგავს ცვლადის ინიციალიზაციის პროცესს. **მასივის ინიციალიზატორი** წარმოადგენს ერთმანეთისაგან მძიმით გამოყოფილი გამოსახულებების სიას, რომელიც ფიგურულ ფრჩხილებშია მოთავსებული. მძიმით გამოყოფილია მასივის ელემენტების მნიშვნელობები. მასივი ავტომატურად იქმნება ისეთი ზომის, რომ მასში ჩაეტიოს ინიციალიზატორში მითითებული ყველა ელემენტი. ამ დროს **new** ოპერატორის გამოყენება საჭირო არ არის.

მაგალითი 1. შევადგინოთ პროგრამა, რომელიც int a[10] მასივში წრფივი ძებნის მეთოდით მოძებნის საჭირო მნიშვნელობის ელემენტებს და დათვლის მათ რაოდენობას.

აღნიშნული მეთოდი წინამდებარე სახელმძღვანელოს 1.4. თავშია განხილული. ის ითვალისწინებს მასივის ყოველი ელემენტის მნიშვნელობის ციკლურად შედარებას საძიებელი მნიშვნელობის მქონე ცვლადთან და მნიშვნელობების თანხვედრის შემთხვევაში გვაძლევს მოძებნილი ელემენტის ინდექსს. რადგან ამოცანა მოითხოვს საჭირო ელემენტების რაოდენობის დათვლასაც, ამიტომ პროგრამაში დამატებით *n* ცვლადია შემოტანილი საწყისი ნულოვანი მნიშვნელობით და ის ყოველ ჯერზე ერთით იზრდება, როცა მასივის ელემენტის მნიშვნელობა ემთხვევა საძიებელი ცვლადის მნიშვნელობას. თუ მასივში ასეთი ელემენტი არ არსებობს, პროგრამას შესაბამისი შეტყობინება გამოაქვს კონსოლზე.

ვინაიდან პროგრამაში მასივის ზომა (ელემენტების რაოდენობა) წინასწარ არ დავაფიქსირეთ, ამიტომ ციკლურ პროცესში გამოყენებულ იქნა length თვისება, რომელიც მასივის რეალური ელემენტების რაოდენობას თავად განსაზღვრავს. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 104-ზე, ხოლო შესრულების შედეგი ნახ. 105-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```java
package masivebi;
//წრფივი ძებნის მეთოდი
public class Masivi {
    public static void main(String args[]){
        int a[]={12, 10, 23, 34, 45, 10, 34, 10, 35, 10};
        int key=10;  //საძიებელი მნიშვნელობის ცვლადი
        int s=0, n=0;  // n ითვლის საჭირო ელემენტების რაოდენობას
        for(int i=0; i<a.length; i++){
            if(a[i]==key){
                n++;
                s++;
                System.out.println("საძიებელი ელემენტის ინდექსი=" +i);
            }
        }
            if(s==0)
                System.out.println("საჭირო მნიშვნელობის  ელემენტი მასივში არ გვაქვს");
            else
                System.out.println("საძიებელი ელემენტების რაოდენობა=" + n);

    }

}
```

ნახ. 104

შედეგი:

```
საძიებელი ელემენტის ინდექსი=1
საძიებელი ელემენტის ინდექსი=5
საძიებელი ელემენტის ინდექსი=7
საძიებელი ელემენტის ინდექსი=9
საძიებელი ელემენტების რაოდენობა=4
```

ნახ. 105

მაგალითი 2. შევადგინოთ პროგრამა, რომელიც int a[10] მასივის ელემენტებს შემთხვევითი რიცხვების გენერირების გზით მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს და „ჩაძირვის“ მეთოდით დაახარისხებს მათ კლებადობით. დაბეჭდავს საწყის და დახარისხებულ მასივებს. დახარისხების აღნიშნული მეთოდი წინამდებარე სახელმძღვანელოს 1.4.4. თავშია განხილული. ვინაიდან, ამოცანაში მოითხოვება, რომ მასივის ელემენტებს შემთხვევითი რიცხვების გენერირების გზით მივანიჭოთ მნიშვნელობები, ამიტომ პროგრამაში აუცილებელია Random კლასის ჩართვა ბრძანებით: import java.util.Random და შემდგომ ამავე კლასის ობიექტის შექმნა ბრძანებით: Random ob1=new Random(); ხოლო მასივის ელემენტებისთვის სასურველი რიცხვითი დიაპაზონიდან (ჩვენ შემთხვევაში 0-20) მნიშვნელობების მისანიჭებლად, ციკლურად შემდეგი ბრძნების გამოყენება: a[i]=(int)(20*ob1.nextDouble()); ბრძანებაში int გულისხმობს მთელრიცხვა მნიშვნელობების მინიჭებას. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 106-ზე, ხოლო შესრულების შედეგი ნახ. 107-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package masivebi;
2  import java.util.Random;
3  //"ჩამირვის" მეთოდი
4  public class Masivi {
5      public static void main(String args[]){
6          int a[]=new int[10];
7          int t; //ცვლადი, რომელსაც ელემენტების მნიშვნელობათა შესაცვლელად ვიყენებთ
8          Random ob1=new Random();
9          System.out.println("საწყისი მასივი:");
10         for(int i=0; i<a.length; i++){
11             a[i]=(int)(20*ob1.nextDouble());
12             System.out.print(a[i] +" ");
13         }
14         for(int k=1; k<a.length; k++)  //ეტაპების თვლა
15             for(int i=0; i<a.length-1; i++)
16                 if(a[i]<a[i+1]){
17                     t=a[i];
18                     a[i]=a[i+1];
19                     a[i+1]=t;
20                     }
21         System.out.println();
22         System.out.println("დახარისხებული მასივი:");
23         for(int i=0; i<a.length; i++)
24             System.out.print(a[i] +" ");
25              System.out.println();
26     }
27 }
```

ნახ. 106

შედეგი:

```
საწყისი მასივი:
4 2 9 16 13 7 8 7 0 4
დახარისხებული მასივი:
16 13 9 8 7 7 4 4 2 0
```

ნახ. 107

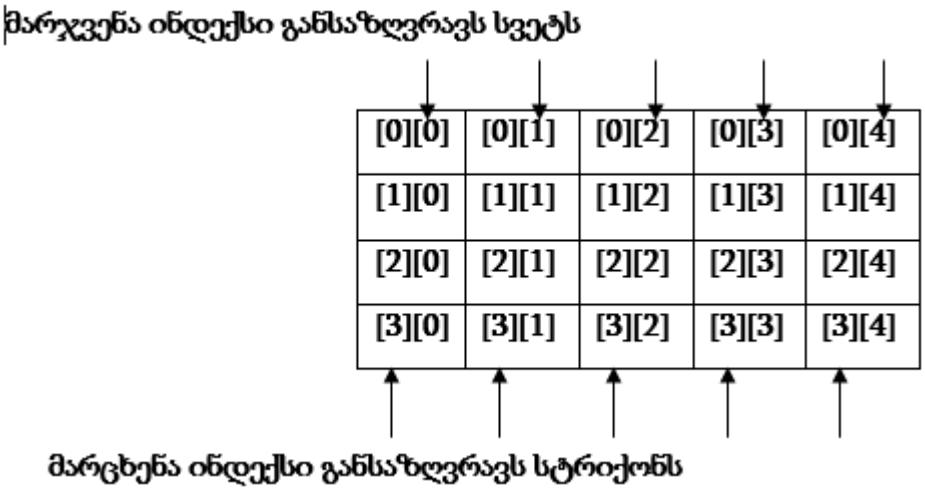
საჭიროა აღინიშნოს, რომ პროგრამის ყველა გაშვებაზე სხვადასხვა შედეგები მიიღება, რადგან მოცემულ დიაპაზონში (0-20) შემთხვევითი რიცხვები თანაბარი ალბათობით გამომუშავდება.

3.6.2. მრავალგანზომილებიანი მასივები

Java-ს მრავალგანზომილებიანი მასივი წარმოადგენს მასივების მასივს. ის მოქმედებს ჩვეულებრივი მრავალგანზომილებიანი მასივის ანალოგიურად. თუმცა, მას გარკვეული თავისებურებაც ახასიათებს. მრავალგანზომილებიანი მასივის თითოეული დამატებითი ინდექსის მისათითებლად იყენებენ კვადრატული ფრჩხილების ცალკე წყვილს. მაგალითად, ორგანზომილებიანი მასივი შეიძლება შემდეგი სახით გამოვაცხადოთ:

```
int twoD[][] = new int[4][5];
```

ეს ოპერატორი 4x5 განზომილებიანი მასივისთვის გამოყოფს მეხსიერებას. მასივის ლოგიკური ორგანიზება ნაჩვენებია 108-ე სურათზე.



ნახ. 108

შესაძლებელია მრავალგანზომილებიანი მასივების ინიციალიზაცია. ამისათვის საჭიროა თითოეული ინიციალიზატორი ჩავსვათ ცალკე ფიგურული ფრჩხილების წყვილში. მაგალითად, პროგრამის შემდეგი ფრაგმენტი ქმნის მატრიცას, სადაც ყოველი ელემენტი წარმოადგენს თავისი სტრიქონისა და სვეტის ინდექსების ნამრავლს.

```
class Matrix {
public static void main(String args[]) {
double m[] [] = {
{ 0*0, 1*0, 2*0, 3*0 },
{ 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 },
{ 0*3, 1*3, 2*3, 3*3 }
} ;  } }
```

მასივების გამოცხადებისათვის შეიძლება გამოყენებულ იქნეს მეორე ფორმა:

<ტიპი> [ ] <ცვლადის სახელი>;

ამ ფორმაში კვადრატული ფრჩხილები მიეთითება ტიპის გამოცხადების შემდეგ და არა მასივის ცვლადის შემდეგ. მაგალითად, მასივის შემდეგი ორი გამოცხადება ეკვივალენტურია:

```
int al[] = new int[3];
int [] a1= new int[3];
```

ქვემოთ ნაჩვენები ორი გამოცხადებაც ეკვივალენტურია:

```
char twod1[] [] = new char[3] [4];
char[] [] twod1 = new char[3] [4];
```

გამოცხადების ეს მეორე ფორმა მოსახერხებელია, როდესაც ხდება რამდენიმე მასივის ერთდროული გამოცხადება.

მაგალითად, გამოცხადება

int[] nums, nums2, nums3; // იქმნება 3 მასივი

ქმნის **int** ტიპის მასივების სამ სახელს. იგი ეკვივალენტურია შემდეგი გამოცხადების:

int nums[], nums2[], nums3[]; // იქმნება 3 მასივი.

მაგალითი 5.3. შევადგინოთ პროგრამა, რომელიც int a[5][5] კვადრატული მატრიცის ელემენტებს შემთხვევითი რიცხვების გენერირების გზით მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს, მოახდენს მატრიცის ტრანსპონირებას და დაბეჭდავს საწყის და ტრანსპონირებულ მატრიცებს.

გვინდა შევნიშნოთ, რომ კვადრატულ მატრიცაში სტრიქონებისა და სვეტების რაოდენობა ერთმანეთის ტოლია და შესაბამისად, მხოლოდ კვადრატული მატრიცის ტრანსპონირებაა შესაძლებელი. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 109-ზე, ხოლო შესრულების შედეგი ნახ. 110-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package masivebi;
2  import java.util.Random;
3  //მატრიცის ტრანსპონირება
4  public class Masivi {
5      public static void main(String args[]){
6          int a[][]=new int[5][5];
7          int t; //ცვლადი, რომელსაც ელემენტების მნიშვნელობათა შესაცვლელად ვიყენებთ
8          Random ob1=new Random();
9          System.out.println("საწყისი მატრიცა:");
10         for(int i=0; i<a.length; i++){
11             for(int j=0; j<a.length; j++){
12             a[i][j]=(int)(30*ob1.nextDouble());
13             System.out.print(a[i][j] +"\t");
14         }
15             System.out.println();
16         }
17             for(int i=0; i<a.length; i++){
18                 for(int j=i+1; j<a.length; j++){
19                 t=a[i][j];
20                 a[i][j]=a[j][i];
21                 a[j][i]=t;
22                     }}
23         System.out.println();
24         System.out.println("ტრანსპონირებული მატრიცა:");
25         for(int i=0; i<a.length; i++){
26             for(int j=0; j<a.length; j++)
27             System.out.print(a[i][j] +"\t");
28              System.out.println();
29         }
30     }
31 }
```

ნახ. 109

შედეგი:

```
საწყისი მატრიცა:
2       2       17      8       19
6       7       4       26      29
18      25      22      25      23
9       24      17      6       10
5       21      29      8       23

ტრანსპონირებული მატრიცა:
2       6       18      9       5
2       7       25      24      21
17      4       22      17      29
8       26      25      6       8
19      29      23      10      23
```

ნახ. 110

მაგალითი 5. for ციკლის **for-each** სტილში გამოყენებით შევადგინოთ პროგრამა, რომელიც int a[10] მასივის ელემენტებს შემთხვევითი რიცხვების გენერირების გზით მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს, გამოთვლის მასივის ელემენტების ჯამს, ნამრავლსა და საშუალო არითმეტიკულ მნიშვნელობებს.

შევნიშნავთ, რომ ასეთი ციკლის შემთხვევაში საჭირო აღარაა მასივის ინდექსების ხელით მიწერა და ციკლის მთვლელის გამოყენება, რომლისთვისაც საწყისი და საბოლოო მნიშვნელობები უნდა მიგვეთითებინა. აქ პროგრამა ავტომატურად შეასრულებს ციკლს მასივის ყველა ელემენტზე და მიმდევრობით ამოიღებს თითოეული ელემენტის მნიშვნელობას, პირველიდან ბოლო წევრამდე.

ასევე უნდა აღვნიშნოთ, რომ მასივის ელემენტების ჯამის გამოსათვლელად პროგრამაში შემოტანილია S ცვლადი საწისი ნულოვანი მნიშვნელობით და ნამრავლის გამოსათვლელად - P ცვლადი საწყისი ერთის ტოლი მნიშვნელობით. მასივის ელემენტებს აქაც შემთხვევითი რიცხვების გენერირების გზით ენიჭება მთელი ტიპის შემთხვევითი მნიშვნელობები და შესაბამისად, პროგრამის შესრულებაზე გაშვებისას ყოველ ჯერზე განსხვავებული შედეგები მიიღება. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 111-ზე, ხოლო შესრულების შედეგი ნახ. 112-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package masivebi;
2  import java.util.Random;
3  //for-each ციკლის გამოყენება
4  public class Masivi {
5      public static void main(String args[]){
6          int a[]=new int [10];
7          double s=0;
8          long p=1;
9          Random ob1=new Random();
10         System.out.println("საწყისი მასივი:");
11         for(int i=0; i<a.length; i++)
12             a[i]=(int)(15*ob1.nextDouble());
13         for(int x : a){
14             System.out.print(x + "\t");
15             s+=x;
16             p*=x;
17         }
18         System.out.println();
19         System.out.println("ჯამი=" + s);
20         System.out.println("ნამრავლი=" + p);
21         System.out.println("საშუალო არითმეტიკული=" + s/10);
22     }
23 }
24
```

ნახ. 111

შედეგი:

```
საწყისი მასივი:
11      2       2       1       6       11      2       2       2       7
ჯამი=46.0
ნამრავლი=162624
საშუალო არითმეტიკული=4.6
```

ნახ. 112

ერთ და მრავალგანზომილებიან მასივებთან დაკავშირებული ვიდეო-მასალა შეგიძლიათ იხილოთ ბმულზე: https://www.thenewboston.com/videos.php?cat=31 ვიდეო-ფაილები: 27-30, 34.

დავალება

1. შეადგინეთ პროგრამა, რომელიც int b[12] მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს, ცალ-ცალკე გამოთვლის მასივის კენტი და ლუწი მნიშვნელობის ელემენტების ნამრავლებს და მიღებულ შედეგებს ერთმანეთს შეადარებს (შენიშვნა: მასივის კენტი და ლუწი მნიშვნელობის ელემენტების განსაზღვრისთვის ციკლურად შეამოწმეთ ყველა ელემენტი მოდულით გაყოფის ოპერაციის გამოყენებით).

2. შეადგინეთ პროგრამა, რომელიც int a[10] მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს, კომბინირებული მეთოდით დაახარისხებს მასივის ელემენტებს ზრდადობით და დაბეჭდავს საწყის და დახარისხებულ მასივებს.
3. შეადგინეთ პროგრამა, რომელიც int c[14] მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს და ბინარული ძებნის მეთოდით განსაზღვრავს საჭირო ელემენტის ინდექსს (გაითვალისწინეთ, რომ აღნიშნული მეთოდი მასივის წინასწარ დახარისხებას მოითხოვს).
4. შეადგინეთ პროგრამა, რომელიც int a[4][5] მატრიცის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს, დაახარისხებს მატრიცის ელემენტებს სტრიქონების, შემდეგ სვეტების მიხედვით ზრდადობით და დაბეჭდავს საწყის და დახარისხებულ მატრიცებს.
5. შეადგინეთ პროგრამა, რომელიც int b[5][6] მატრიცის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს, გამოთვლის მატრიცის ელემენტების კვადრატების ჯამის მნიშვნელობებს სვეტების მიხედვით, განსაზღვრავს მიღებულ შედეგებს შორის უდიდესს და დაბეჭდავს: საწყის მატრიცას, მიღებულ შედეგებს და უდიდესი მნიშვნელობის შედეგის მომცემ ელემენტებს.
6. შეადგინეთ პროგრამა, რომელიც int a[10] და int b[10] მასივების ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს, გამოთვლის მათ სკალარულ ნამრავლს და დაბეჭდავს საწყის მასივებს და მიღებულ შედეგს (გაითვალისწინეთ, რომ ორი მასივის სკალარული ნამრავლი ამ მასივების შესაბამისი ელემენტების ნამრავლთა ჯამია და ის კონკრეტული რიცხვია).

3.7. კლასები

3.7.1. კლასის ზოგადი ფორმა

კლასი Java-ს ცენტრალური კომპონენტია. რადგან კლასი განსაზღვრავს ობიექტის ფორმასა და არსს, შესაბამისად, იგი ლოგიკურ კონსტრუქციას წარმოადგენს, რომლის საფუძველზეც აგებულია თავად Java დაპროგრამების ენა. ნებისმიერი კონცეფცია, რომელიც უნდა იქნეს რეალიზებული Java პროგრამაში, საჭიროა განთავსდეს კლასის შიგნით.

კლასის უმნიშვნელოვანესი თვისება იმაში გამოიხატება, რომ იგი განსაზღვრავს მონაცემთა ახალ ტიპს. განსაზღვრის შემდეგ მონაცემთა ახალი ტიპი ობიექტების შესაქმნელად შეიძლება გამოვიყენოთ.

ამგვარად, კლასი წარმოადგენს ობიექტის შაბლონს, ხოლო თავად ობიექტი – კლასის ეგზემპლიარს. შესაბამისად, რადგან ობიექტი განიხილება, როგორც კლასის ეგზემპლიარი, ამიტომ ტერმინები: **ობიექტი** და **ეგზემპლიარი** ერთმანეთის სინონიმებია.

კლასის განსაზღვრა მოიცავს მისი კონკრეტული ფორმისა და არსის გამოცხადებას. ამ დროს აღიწერება კლასის მონაცემები და კოდი, რომელიც მათზე მოქმედებს. შესაძლებელია მარტივი კლასი შეიცავდეს მხოლოდ მონაცემებს ან მხოლოდ კოდს, მაგრამ რეალურ პროგრამებში კლასების უმრავლესობა ორივე კომპონენტს შეიცავს.

კლასის გამოცხადებას ემსახურება საკვანძო სიტყვა **class**. კლასის განსაზღვრის გამარტივებულ ზოგად ფორმას შემდეგი სახე აქვს:

```
class <კლასის სახელი> {
<ტიპი> ეგზემპლიარის _ცვლდი 1;
<ტიპი> ეგზემპლიარის _ცვლდი 2;
// ...
<ტიპი> ეგზემპლიარის _ცვლდი N;
<ტიპი> <მეთოდის სახელი 1>(<პარამეტრების სია>) {
// მეთოდის ტანი
}
<ტიპი> <მეთოდის სახელი 2>(<პარამეტრების სია>) {
// მეთოდის ტანი
}
 // ...
<ტიპი> <მეთოდის სახელი N>(<პარამეტრების სია>) {
// მეთოდის ტანი
}
}
```

კლასში განსაზღვრულ მონაცემებს ან ცვლადებს, კლასის ეგზემპლიარის ცვლადებს უწოდებენ. კოდი თავსდება მეთოდის შიგნით. კლასში გამოცხადებულ მეთოდებსა და ცვლადებს

ერთად **კლასის წევრები** ეწოდება. კლასების უმრავლესობაში ეგზემპლიარის ცვლადებზე მოქმედებები და მიმართვები ამავე კლასში აღწერილი მეთოდების საშუალებით ხდება.

3.7.2. კლასის ობიექტის შექმნა და კონსტრუქტორები

როგროც ზემოთ აღვნიშნეთ, კლასის შექმნით ჩვენ მონაცემთა ახალ ტიპს ვქმნით, რომელიც მოცემული ტიპის ობიექტების გამოსაცხადებლად შეგვიძლია გამოვიყენოთ. კლასის ობიექტების შექმნა ორსაფეხუროვან პროცესს წარმოადგენს. თავდაპირველად, აუცილებელია კლასის ტიპის ცვლადის გამოცხადება. აღნიშნული ცვლადი ობიექტს არ განსაზღვრავს. ის წარმოგვიდგენს მხოლოდ ცვლადს, რომელსაც ობიექტზე მიმართვა შეუძლია. შემდეგ აუცილებელია ნამდვილი ობიექტის ფიზიკური ასლის შექმნა და მისი მინიჭება მოცემულ ცვლადზე. ეს უკანასკნელი **new** ოპერატორის საშუალებით ხორციელდება. ამ ოპერატორის ჩაწერის ზოგადი ფორმა შემდეგია:

```
კლასის _ ცვლადი=new კლასის_ სახელი();
```

აქ **კლასის_ ცვლადი** წარმოადგენს შესაქმნელი კლასის ცვლადს. **კლასის_ სახელი** კი იმ კლასის სახელია, რომლის დაკონკრეტებასაც ვახდენთ. კლასის სახელი, რომელსაც მრგვალი ფრჩხილები მოსდევს, მიუთითებს მოცემული კლასის **კონსტრუქტორზე**. ეს უკანასკნელი განსაზღვრავს მოქმედებებს, რომლებიც კლასის ობიექტის შექმნის დროს სრულდება. კონსტრუქტორები ყველა კლასის მნიშვნელოვანი ნაწილია და მათ მრავალი საინტერესო ატრიბუტი გააჩნიათ. კლასის კონსტრუქტორს აუცილებელია იგივე სახელი ჰქონდეს, რაც კლასს. მისი დანიშნულებაა კლასის მონაცემების ინიციალება. კლასის ობიექტის შექმნისთანავე კი, ავტომატურად ხდება კონსტრუქტორის გამოძახება და შესაბამისად, ავტომატურად სრულდება მასში არსებული ბრძანებები. რეალურ პროგრამებში კლასების უმრავლესობა თავიანთ კონსტრუქტორებს ცხადად განსაზღვრავს კლასის აღწერის შიგნით. ხოლო, თუ კონსტრუქტორი ცხადი სახით მითითებული არ არის, მაშინ Java კლასს ავტომატურად უზრუნველყოფს სტანდარტული კონსტრუქტორით.

3.7.3. მეთოდები

როგორც წესი, კლასები ორი ელემენტისგან შედგება: ეგზემპლიარის ცვლადებისა და მეთოდებისგან.

მეთოდის გამოცხადების ზოგადი ფორმა შემდეგია:

```
ტიპი სახელი(პარამეტრების _ სია){
//მეთოდის ტანი
}
```

ტიპი აქ მიუთითებს მეთოდის მიერ დასაბრუნებელი მონაცემების ტიპზე. ის შეიძლება იყოს ნებისმიერი დასაშვები ტიპი, მათ შორის პროგრამისტის მიერ შექმნილი კლასის ტიპიც. თუ მეთოდი შედეგს აბრუნებს, მაშინ ის აუცილებლად უნდა შეიცავდეს ერთს მაინც **return** ოპერატორს, რომელიც იმავე ტიპის შედეგს უნდა აბრუნებდეს, რომელი ტიპის მითითებითაც იწყება მეთოდის აღწერა. თუ მეთოდი მნიშვნელობას არ აბრუნებს, მაშინ საკვანძო სიტყვა **void** გამოიყენება.

სახელი წარმოადგენს მეთოდის სახელს. ის ნებისმიერი დასაშვები იდენტიფიკატორია, გარდა იმ სახელებისა, რომლებიც სხვა ელემენტების მიერ უკვე გამოყენებულია მიმდინარე ხედვის არეში.

პარამეტრების _ სია წარმოადგენს „ტიპი–იდენტიფიკატორი“ წყვილების მიმდევრობას, რომლებიც მძიმეებითაა გამოყოფილი. შინაარსობრივად, პარამეტრები ცვლადებია, რომლებიც იმ არგუმენტების მნიშვნელობებს იღებენ, რაც მეთოდს გადაეცემა მისი გამოძახების დროს. თუ მეთოდს პარამეტრები არ გააჩნია, მაშინ პარამეტრების სია ცარიელია და გამოძახებისას, შესაბამისად, ის არც არგუმენტებს შეიცავს.

მეთოდები, რომელთაც დაბრუნებული მნიშვნელობის ტიპად **void** არ უწერიათ, გამომძახებელ პროცედურას მნიშვნელობას **return** ოპერატორის საშუალებით უბრუნებენ. ამ უკანასკნელის ჩაწერის სინტაქსი შემდეგია:

return მნიშვნელობა;

საჭიროა გვესმოდეს, რომ, როდესაც ეგზემპლიარის ცვლადზე მიმართვა სრულდება კოდით, რომელიც არ წარმოადგენს იმ კლასის შემადგენელ ნაწილს, სადაც ეგზემპლიარის ცვლადია განსაზღვრული, აუცილებელია ობიექტის მითითება წერტილის ოპერატორის გამოყენებით; მაგრამ, როდესაც მიმართვა წარმოებს კოდით, რომელიც იმავე კლასის ნაწილია, რომელშიც ეგზემპლიარის ცვლადია განსაზღვრული, ცვლადზე მიმართვა უშუალოდ სრულდება. ეს წესები მეთოდების მიმართაც გამოიყენება. მნიშვნელოვანია შეგვეძლოს ორი ტერმინის: **პარამეტრის** და **არგუმენტის** განსხვავება.

პარამეტრი მეთოდის მიერ განსაზღვრული ცვლადია, რომელიც მნიშვნელობას იღებს მეთოდის გამოძახების დროს. **არგუმენტი** კი მნიშვნელობაა, რომელიც მეთოდს მისი გამოძახების დროს გადაეცემა.

კლასის ყოველი ეგზემპლიარის შექმნის დროს კლასის ცვლადების ინიციალება საკმაოდ დამღლელი პროცესია. გაცილებით მარტივი და მოსახერხებელია, როცა კლასის ცვლადები ობიექტის პირველი შექმნისთანავე იღებენ მნიშვნელობებს. ვინაიდან, ინიციალიზაციის მოთხოვნა ხშირია, ამიტომ Java ენა ობიექტებს უფლებას აძლევს თავად შეასრულონ ეს პროცესი მათი

(ობიექტების) შექმნის დროს. ეს ავტომატური ინიციალიზაცია, როგორც უკვე აღვნიშნეთ, **კონსტრუქტორის** საშუალებით ხორციელდება.

კონსტუქტორი ობიექტის ინიციალებას უშუალოდ მისი შექმნის დროს ახდენს. მისი სახელი იმ კლასის სახელს ემთხვევა, რომელშიც ის განსაზღვრულია, ხოლო სინტაქსი – მეთოდის სინტაქსის ანალოგიურია. ობიექტის შექმნის შემდეგ კონსტრუქტორი ავტომატურად გამოიძახება **new** ოპერატორის შესრულების დამთავრებამდე. ერთი შეხედვით, კონსტრუქტორები ცოტა უცნაურად გამოიყურება, რადგან მათ არ გააჩნიათ დასაბრუნებელი შედეგის ტიპი და არც საკვანძო სიტყვა **void**-ს გამოიყენებენ.

მაგალითი 1. მომხმარებლის მიერ შექმნილი კლასის გამოყენებით შევადგინოთ პროგრამა, რომელიც ერთიდან 10-ის ჩათვლით ყველა ნატურალური რიცხვის ფაქტორიალებს გამოთვლის. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 113-ზე, ხოლო შესრულების შედეგი ნახ. 114-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package klasebi;
2  //ფაქტორიალის გამოთვლა
3  class facto{
4      long F;
5      facto(){   //კლასის უპარამეტრებო კონსტრუქტორი
6          F=1;
7      }
8   void method1(){   //ფაქტორიალის გამოთვლის მეთოდი
9       for(int i=1; i<=10; i++){
10          F*=i;
11          System.out.println(i + "!=" + F);
12      }
13   }
14  }
15  public class nimushebi {
16      public static void main(String args[]){
17          facto ob1=new facto();
18          ob1.method1();
19      }
20
21  }
```

ნახ. 113

შედეგი:

```
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5040
8!=40320
9!=362880
10!=3628800
```

ნახ. 114

მაგალითი 2. მომხმარებლის მიერ შექმნილი კლასის გამოყენებით შევადგინოთ პროგრამა, რომელიც int a[10] მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს. კლასის კონსტრუქტორის საშუალებით მოახდენს კლასის ეგზემპლიარის ცვლადების ინიციალებას. კლასის მეთოდებით ცალ-ცალკე გამოთვლის მასივის კენტი და ლუწი მნიშვნელობის ელემენტების ჯამის მნიშვნელობებს და მიღებულ შედეგებს ერთმანეთს შეადარებს.

მინიშნება: ამოცანის გადაწყვეტის დროს გათვალისწინებულ იქნა ის გარემოება, რომ ნებისმიერი ლუწი მთელი რიცხვის 2-ზე გაყოფის შედეგად ნაშთი ნოლის ტოლი მიიღება, ხოლო კენტი რიცხვის შემთხვევაში - ერთის.

ქვემოთ წარმოდგენილია დასმული ამოცანის შედეგები (ნახ.115) და შემდეგ კი ამოცანის პროგრამული რეალიზაცია (ნახ.116).

შედეგები:

```
მიმდინარე მასივი:
3       4       9       11      1       8       0       11      0       9
ლუწების ჯამი=12
კენტების ჯამი=44
კენტების ჯამი მეტია ლუწების ჯამზე.
```

ნახ. 115

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package klasebi;
2  import java.util.Random;
3  //მასივები კლასებში
4  class masivi{
5      int a[]=new int[10];
6      int s1, s2;
7      masivi()
8  {
9      s1=0; s2=0;
10     Random ob1=new Random();
11     System.out.println("მიმდინარე მასივი:");
12     for(int i=0; i<a.length; i++){
13         a[i]=(int)(12*ob1.nextDouble());
14         System.out.print(a[i] + "\t");
15     }
16 }
17     int method1(){ //მასივის ლუწი მნიშვნელობის ელემენტების ჯამის გამოთვლის მეთოდი
18         for(int i=0; i<a.length; i++)
19             if(a[i]%2==0)
20                 s1+=a[i];
21         return s1;
22     }
23     int method2(){ //მასივის კენტი მნიშვნელობის ელემენტების ჯამის გამოთვლის მეთოდი
24         for(int i=0; i<a.length; i++)
25             if(a[i]%2==1)
26                 s2+=a[i];
27         return s2;
28     }
29     void method3(int s1, int s2){ //შედეგების შედარების მეთოდი
30         if(s1>s2)
31             System.out.println("ლუწების ჯამი მეტია კენტების ჯამზე.");
32             else if(s2>s1)
33                 System.out.println("კენტების ჯამი მეტია ლუწების ჯამზე.");
34             else
35                 System.out.println("შედეგები ერთმანეთის ტოლია.");
36     }
37 } // კლასის აღწერის დასასრული
38 public class nimushebi {
39     public static void main(String args[]){
40         masivi ob2=new masivi();
41         int res1, res2;
42         res1=ob2.method1();
43         res2=ob2.method2();
44         System.out.println();
45         System.out.println("ლუწების ჯამი=" + res1);
46         System.out.println("კენტების ჯამი=" + res2);
47         ob2.method3(res1, res2);}}
```

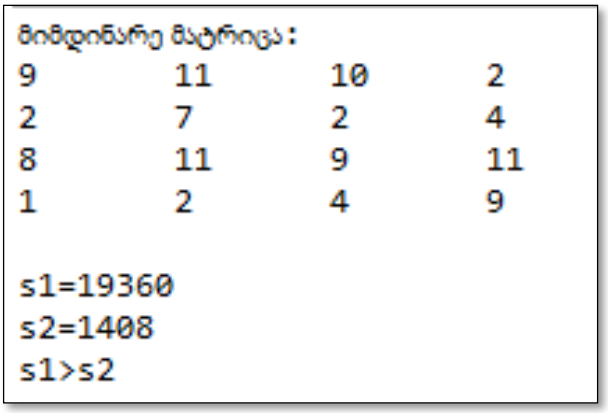
ნახ. 116

მაგალითი 3. მომხმარებლის მიერ შექმნილი კლასის გამოყენებით შევადგინოთ პროგრამა, რომელიც int a[4][4] მატრიცის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს. კლასის კონსტრუქტორის საშუალებით მოახდენს კლასის ეგზემპლიარის ცვლადების ინიციალებას. კლასის მეთოდებით ცალ-ცალკე გამოთვლის მატრიცის მთავარი დიაგონალის ზემოთ და ქვემოთ არსებული ელემენტების ნამრავლებს და მიღებულ შედეგებს ერთმანეთს შეადარებს.

მინიშნება: ამოცანის გადაწყვეტის დროს გათვალისწინებულ იქნა ის გარემოება, რომ კვადრატული მატრიცის მთავარი დიაგონალის ზემოთ არსებული ელემენტების სტრიქონების ინდექსების მნიშვნელობები ნაკლებია ამავე ელემენტების სვეტების ინდექსების მნიშვნელობებზე, ხოლო მთავარი დიაგონლის ქვემოთ განლაგებული ელემენტების შემთხვევაში კი პირიქით - სტრიქონების ინდექსების მნიშვნელობები მეტია სვეტების ინდექსების მნიშვნელობებზე.

ქვემოთ წარმოდგენილია დასმული ამოცანის ამოხსნის შედეგები (ნახ.117) და შემდეგ კი ამოცანის პროგრამული რეალიზაცია (ნახ.118).

შედეგები:



ნახ. 117

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package klasebi;
2  import java.util.Random;
3  //მატრიცები კლასებში
4  class masivi{
5      int a[][]=new int[4][4];
6      long s1, s2;
7      masivi(){
8      s1=1; s2=1;
9      Random ob1=new Random();
10     System.out.println("მიმდინარე მატრიცა:");
11     for(int i=0; i<a.length; i++){
12         for(int j=0; j<a.length; j++){
13         a[i][j]=(int)(12*ob1.nextDouble());
14         System.out.print(a[i][j] + "\t");}
15         System.out.println();}}
16     long method1(){ //მთავარი დიაგონალის ზემოთ არსებული ელემენტების ნამრავლი
17         for(int i=0; i<a.length; i++)
18             for(int j=0; j<a.length; j++)
19             if(i<j)
20                 s1*=a[i][j];
21         return s1;}
22     long method2(){ //მთავარი დიაგონალის ქვემოთ  არსებული ელემენტების ნამრავლი
23         for(int i=0; i<a.length; i++)
24             for(int j=0; j<a.length; j++)
25             if(i>j)
26                 s2*=a[i][j];
27         return s2;}
28     void method3(long s1, long s2){ //შედეგების შედარების მეთოდი
29         if(s1>s2)
30             System.out.println("s1>s2");
31             else if(s2>s1)
32                 System.out.println("s2>s1");
33             else
34                 System.out.println("s1=s2");}
35 } // კლასის აღწერის დასასრული
36 public class nimushebi {
37     public static void main(String args[]){
38         masivi ob2=new masivi();
39         long res1, res2;
40         res1=ob2.method1();
41         res2=ob2.method2();
42         System.out.println();
43         System.out.println("s1=" + res1);
44         System.out.println("s2=" + res2);
45         ob2.method3(res1, res2);}}
```

ნახ. 118

კლასებთან დაკავშირებული ვიდეო-მასალა შეგიძლიათ იხილოთ ბმულზე: https://www.thenewboston.com/videos.php?cat=31 ვიდეო-ფაილები: 14-17.

3.7.4. this საკვანძო სიტყვის გამოყენება

ზოგჯერ საჭიროა, რომ მეთოდმა მის გამომძახებელ ობიექტს მიმართოს. ამ მიზნით Java–ში განსაზღვრულია **this** საკვანძო სიტყვა. ის შეიძლება ნებისმიერი მეთოდის შიგნით მიმდინარე ობიექტზე მიმართვისათვის გამოვიყენოთ.

ასევე ცნობილია, რომ Java ენაში ერთი და იგივე ხედვის არეში დაუშვებელია ერთი და იმავე სახელის ორი ლოკალური ცვლადის გამოცხადება. საჭიროა შევნიშნოთ, რომ შესაძლებელია ლოკალური ცვლადების, მათ შორის მეთოდის ფორმალური პარამეტრების არსებობა, რომელთა სახელებიც ემთხვევა კლასის ეგზემპლიარის ცვლადების სახელებს. მაგრამ, როდესაც ლოკალური ცვლადის სახელი ემთხვევა ეგზემპლიარის ცვლადის სახელს, მაშინ ლოკალური ცვლადი ფარავს ეგზემპლიარის ცვლადს. რადგან **this** საკვანძო სიტყვა საშუალებას გვაძლევს

უშუალოდ მივმართოთ ობიექტს, ამიტომ მისი გამოყენება სახელების სივრცის ნებისმიერი კონფლიქტის გადასაწყვეტად შეგვიძლია, მათ შორის ისეთი კონფლიქტების, რომელიც ეგზემპლიარის ცვლადებს და ლოკალურ ცვლადებს უკავშირდება.

ზემოთ თქმულის საილუსტრაციოდ განვიხილოთ **მაგალითი 4.** მომხმარებლის მიერ შექმნილი კლასის და **this** საკვანძო სიტყვის გამოყენებით შევადგინოთ მართკუთხა პარალელეპიპედის მოცულობის გამოსათვლელი პროგრამა. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 119-ზე, ხოლო შესრულების შედეგი ნახ. 120-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package klasebi;
2  //this საკვანძო სიტყვის გამოყენება
3  class student{
4      int a, b, c;
5      student(int a, int b, int c){ //კლასის კონსტრუქტორი
6          this.a=a;
7          this.b=b;
8          this.c=c;
9       }
10     int method1(){ //მართკუთხა პარალელეპიპედის მოცულობის გამოთვლის მეთოდი
11         return a*b*c;
12     }
13 }
14 public class nimushebi {
15     public static void main(String args[]){
16     student ob1=new student(2, 3, 4);
17     System.out.println("მოცულობა=" + ob1.method1());
18     }
19 }
```

ნახ. 119

შედეგი:

```
მოცულობა=24
```

ნახ. 120

დავალება

1. მომხმარებლის მიერ შექმნილი კლასის გამოყენებით შეადგინეთ პროგრამა, რომელიც int b[15] მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს. კლასის კონსტრუქტორის საშუალებით მოახდენს კლასის ეგზემპლიარის ცვლადების ინიციალებას. კლასის მეთოდებით ცალ-ცალკე გამოთვლის მასივის კენტი და ლუწინდექსიანი ელემენტების საშუალო არითმეტიკულ მნიშვნელობებს და მიღებულ შედეგებს ერთმანეთს შეადარებს.
2. მომხმარებლის მიერ შექმნილი კლასის გამოყენებით შეადგინეთ პროგრამა, რომელიც int a[5][5] მატრიცის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს. კლასის კონსტრუქტორის საშუალებით მოახდენს კლასის ეგზემპლიარის ცვლადების ინიციალებას. კლასის მეთოდებით ცალ-ცალკე გამოთვლის მატრიცის არამთავარი დიაგონალის ზემოთ და ქვემოთ არსებული ელემენტების კვადრატების ჯამის მნიშვნელობებს და მიღებულ შედეგებს ერთმანეთს შეადარებს.
3. მომხმარებლის მიერ შექმნილი კლასის და **this** საკვანძო სიტყვის გამოყენებით შეადგინეთ სამკუთხედის ფართობის გამოსათვლელი პროგრამა (დაიცავით სამკუთხედების აგების წესი).
4. მომხმარებლის მიერ შექმნილი კლასის გამოყენებით შეადგინეთ პროგრამა, რომელიც int D[13] მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს. კლასის კონსტრუქტორის საშუალებით მოახდენს კლასის ეგზემპლიარის ცვლადების ინიციალებას. კლასის მეთოდების საშუალებით დაახარისხებს მასივის ელემენტებს კლებადობით „მარტივი გადანაცვლების“ მეთოდით და დაბეჭდავს საწყის და დახარისხებულ მასივებს.
5. მომხმარებლის მიერ შექმნილი კლასის გამოყენებით შეადგინეთ პროგრამა, რომელიც int a[3][5] მატრიცის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს. კლასის კონსტრუქტორის საშუალებით მოახდენს კლასის ეგზემპლიარის ცვლადების ინიციალებას. კლასის მეთოდებით გამოთვლის მატრიცის ელემენტების ნამრავლებს სვეტების მიხედვით და მიღებულ შედეგებს შორის განსაზღვრავს უმცირესს.

3.8. კლასის მეთოდების და კონსტრუქტორების გადატვირთვა

3.8.1. მეთოდების გადატვირთვა

დაპროგრამების ენა Java უფლებას გვაძლევს ერთსა და იმავე კლასში განვსაზღვროთ ერთი და იგივე სახელის მქონე ორი ან მეტი მეთოდი, მაგრამ მხოლოდ იმ შემთხვევაში, თუ მეთოდებს განსხვავებული პარამეტრები გააჩნიათ. აღნიშნულ შემთხვევაში მეთოდებს **გადატვირთულს,** ხოლო თავად პროცესს კი – **მეთოდების გადატვირთვას** უწოდებენ.

გადატვირთული მეთოდის გამოძახების დროს საჭირო ვერსიის განსაზღვრისათვის Java იყენებს მეთოდის პარამეტრების ტიპს და/ან მათ რაოდენობას. შესაბამისად, გადატვირთული მეთოდები ერთმანეთისგან პარამეტრების ტიპებით და /ან მათი რაოდენობით უნდა განსხვავდებოდეს. მართალია, გადატვირთულ მეთოდებს შესაძლოა განსხვავებული დასაბრუნებელი შედეგის ტიპი ქონდეთ, მაგრამ დასაბრუნებელი შედეგის ტიპი არ არის საკმარისი მეთოდის ორი ან მეტი განსხვავებული ვერსიის განსაზღვრისთვის. გადატვირთული მეთოდის გამოძახების დროს Java-ს შესრულების გარემო ამუშავებს მეთოდის იმ ვერსიას, რომლის პარამეტრები გამოძახებული მეთოდის არგუმენტებს შეესაბამება.

გადატვირთული მეთოდის დემონსტრირების მიზნით განვიხილოთ შემდეგი **მაგალითი 1.** შევადგინოთ პროგრამა, რომელიც მეთოდის გადატვირთვის გზით ორ ცვლადს მნიშვნელობებს უცვლის (ერთმანეთის მნიშვნელობებს ანიჭებს).

ქვემოთ წარმოდგენილია დასმული ამოცანის შედეგები (ნახ.121) და შემდეგ კი ამოცანის პროგრამული რეალიზაცია (ნახ.122).

შედეგი:

```
ცვლადების საწყისი მნიშვნელობებია:
a=25    b=70
ცვლადების  მნიშვნელობები პირველი მეთოდის გამოძახების შემდეგ:
a=70    b=25
ცვლადების  მნიშვნელობები მეორე მეთოდის გამოძახების შემდეგ:
a=70    b=25
```

ნახ. 121

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package klasebi;
2  //პოლიმორფიზმი
3  class student{
4      int a, b;
5      student(int a, int b){ //კლასის კონსტრუქტორი
6          this.a=a;
7          this.b=b;
8       }
9      void method(){ //ცვლადების მნიშვნელობების ბეჭდვა
10         System.out.println("a=" + a +  "\tb=" + b);
11     }
12     void method1(){ //ცვლადებს შორის  მნიშვნელობების გაცვლის მეთოდი-1
13         int t=a;
14         a=b;
15         b=t;
16     }
17     void method1(int a, int b){ //ცვლადებს შორის  მნიშვნელობების გაცვლის მეთოდი-2
18         a+=b;
19         b=a-b;
20         a=a-b;
21     }
22 }
23 public class nimushebi {
24     public static void main(String args[]){
25     student ob1=new student(25,70);
26     System.out.println("ცვლადების საწყისი მნიშვნელობებია:");
27     ob1.method();
28     ob1.method1();
29     System.out.println("ცვლადების  მნიშვნელობები პირველი მეთოდის გამოძახების შემდეგ:");
30     ob1.method();
31     ob1.method1(25, 70);
32     System.out.println("ცვლადების  მნიშვნელობები მეორე მეთოდის გამოძახების შემდეგ:");
33     ob1.method();
34     }
35 }
```

ნახ. 122

3.8.2. კონსტრუქტორების გადატვირთვა

სხვა მეთოდების მსგავსად, შესაძლებელია კონსტრუქტორების გადატვირთვაც. აღნიშნულის საილუსტრაციოდ განვიხილოთ **მაგალითი 2.** შევადგინოთ პროგრამა, რომელიც კლასის კონსტრუქტორის გადატვირთვის გამოყენებით გამოთვლის ტრაპეციის ფართობს. ქვემოთ წარმოდგენილია დასმული ამოცანის ამოხსნის შედეგი (ნახ.123) და შემდგომ მისი პროგრამული რეალიზაცია (ნახ.124).

შედეგი:

```
a=3.0   b=5.0   h=7.0
ფართობი-1=28.0
a=5.0   b=15.0  h=7.0
ფართობი -2=70.0
```

ნახ. 123

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package klasebi;
2  //პოლიმორფიზმი
3  class student{
4      double a, b, h;
5      student(double a, double b, double h){ //კლასის კონსტრუქტორი
6          this.a=a;
7          this.b=b;
8          this.h=h;}
9      student(){ //გადატვირთული კონსტრუქტორი
10         a=5;
11         b=a+10;
12         h=7;}
13
14     void method(){ // მნიშვნელობების ბეჭდვა
15         System.out.println("a=" + a +  "\tb=" + b + "\th=" + h);
16     }
17     double method1(){ //ტრაპეციის ფართობის გამოთვლის  მეთოდი
18     return ((a+b)/2*h);}
19 }
20 public class nimushebi {
21     public static void main(String args[]){
22     student ob1=new student(3, 5, 7);
23     student ob2=new student();
24     double res;
25     ob1.method();
26     res=ob1.method1();
27     System.out.println("ფართობი-1=" + res);
28     ob2.method();
29     res=ob2.method1();
30     System.out.println("ფართობი -2=" + res);
31     }
32 }
```

ნახ. 124

მოცემულ მაგალითში, პირველ შემთხვევაში ეგზემპლიარის ცვლადებზე მნიშვნელობების მისანიჭებლად კონსტრუქტრის მიერ გამოყენებულ იქნა **this** საკვანძო სიტყვა, ხოლო მეორე შემთხვევაში - გარკვეული ფორმულები.

[დავალება]

1. *შეადგინეთ პროგრამა, რომელიც მეთოდის გადატვირთვის გზით მთელი ტიპის ნებისმიერ სამ ცვლადს შორის განსაზღვრავს შუალედური მნიშვნელობის ცვლადს.*
2. *შეადგინეთ პროგრამა, რომელიც მეთოდის გადატვირთვის გზით, შემდეგი ფორმულის* $S = 2\pi RH$ *საფუძველზე, გამოთვლის ცილინდრის გვერდითი ზედაპირის ფართობს.*
3. *შეადგინეთ პროგრამა, რომელიც კლასის კონსტრუქტორის გადატვირთვის გზით,* $S = \pi Rl$ *ფორმულის საფუძველზე, გამოთვლის კონუსის გვერდითი ზედაპირის ფართობს.*
4. *შეადგინეთ პროგრამა, რომელიც კლასის კონსტრუქტორის გადატვირთვის გზით მთელი ტიპის ნებისმიერ ოთხნიშნა რიცხვში განსაზღვრავს საკუთარი ციფრების კუბების ჯამს.*

3.9. რეკურსიული მეთოდები

რეკურსია არის ატრიბუტი, რომელიც მეთოდს საკუთარი თავის გამოძახების უფლებას აძლევს. ასეთ დროს, თავად მეთოდს **რეკურსიული მეთოდი** ეწოდება.

რეკურსიის კლასიკურ მაგალითს არაუარყოფითი მთელი რიცხვის ფაქტორიალის გამოთვლა წარმოადგენს. N რიცხვის ფაქტორიალი ერთიდან N-მდე მთელი რიცხვების ნამრავლია. მაგალითად 5–ის ფაქტორიალი უდრის 1*2*3*4*5 ანუ 120–ს.

როდესაც მეთოდი საკუთარი თავის გამოძახებას ახდენს, ახალი ლოკალური ცვლადებისა და პარამეტრებისთვის სტეკში გამოიყოფა ადგილი და მეთოდის კოდი ამ ახალი საწყისი მნიშვნელობებით სრულდება. რეკურსიული მეთოდის ყოველი გამოძახებისას დაბრუნებული ძველი ლოკალური ცვლადები და პარამეტრები სტეკიდან იშლება და შესრულება მეთოდის შიგნით გამოძახების მომენტიდან გრძელდება. ფაქტიურად, რეკურსიული მეთოდები ტელესკოპის მსგავს მოქმედებებს ასრულებენ.

კომპიუტერული რესურსების გადატვირთვის გამო, რაც მეთოდების დამატებით გამოძახებებს უკავშირდება, მრავალი ქვეპროგრამის რეკურსიული ვერსია შესაძლებელია უფრო ნელა შესრულდეს, ვიდრე მათი იტერაციული ანალოგები. მეთოდზე მრავალჯერადმა მიმართვამ შეიძლება სტეკის გადავსება გამოიწვიოს. ვიცით, რომ პარამეტრები და ლოკალური ცვლადები ამ დროს სტეკში ინახება და ამასთან, ყოველი ახალი გამოძახება ამ მონაცემების ახალ ასლებს წარმოშობს. ეს კი დიდ ალბათობას ქმნის იმისას, რომ სტეკი გადაივსოს. რეკურსიული მეთოდების ძირითადი უპირატესობა იმაში მდგომარეობს, რომ მათი გამოყენება იმ ალგორითმების ვერსიების შესაქმნელადაა შესაძლებელი, რომლებიც რეკურსიული გზით უფრო ნათლად გამოისახება, ვიდრე მათი იტერაციული ანალოგები. რეკურსიული მეთოდების გამოყენებისას უნდა ვიზრუნოთ, რომ პროგრამაში სადმე აუცილებლად იქნეს გამოყენებული **if** ოპერატორი, რომელიც რეკურსიული მეთოდიდან შედეგის დაბრუნებას რეკურსიული გამოძახების შესრულების გარეშე ახორციელებს. წინააღმდეგ შემთხვევაში, ჩვენ უსასრულო რეკურსიას მივიღებთ.

ახლა კი რეკურსიის ის მაგალითები განვიხილოთ, რომელთა შესაბამისი ალგორითმები წინამდებარე სახელმძღვანელოში ბლოკ-სქემების სახით უკვე წარმოვადგინეთ.

მაგალითი 1. შევადგინოთ პროგრამა, რომელიც რეკურსიული მეთოდის საშუალებით ერთიდან შვიდის ჩათვლით მთელი რიცხვების ფაქტორიალების მნიშვნელობებს ითვლის. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 125-ზე, ხოლო შესრულების შედეგი ნახ. 126-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package recursia;
2  //ფაქტორიალის გამოთვლა
3  class student{
4      long facto(long n){ //ფაქტორიალის გამოთვლის რეკურსიული მეთოდი
5          if(n==0 || n==1)
6              return 1;
7          else
8              return n*facto(n-1);
9      }
10 }
11 public class Recursion {
12     public static void main(String args[]){
13         student ob1=new student(); //student კლასის ობიექტის შექმნა
14         for(int i=0; i<=7; i++)
15             System.out.println(i + "!=" + ob1.facto(i));
16     }
17 }
```

ნახ. 125

შედეგი:

```
0!=1
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5040
```

ნახ. 126

მაგალითი 2. შევადგინოთ პროგრამა, რომელიც რეკურსიული გზით გამოთვლის ფიბონაჩის რიცხვითი მიმდევრობის პირველი n რაოდენობის წევრების მნიშვნელობებს. აღნიშნული მიმდევრობა იწყება ნულითა და ერთით და ყოველი მომდევნო წევრი წინა ორი წევრის ჯამის ტოლია. ფიბონაჩის რიცხვით მიმდევრობას აქვს ქვემოთ წარმოდგენილი სახე:

0 1 1 2 3 5 8 13 21 34 55...

და ის ბუნებაში სპირალის ფორმას აღწერს. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 127-ზე, ხოლო შესრულების შედეგი ნახ. 128-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package recursia;
2  //ფიბონაჩის რიცხვითი მიმდევრობის პირველი 11  წევრის მნიშვნელობის გამოთვლა
3  class student{
4      long fibonacci(long n){ // რეკურსიული მეთოდი
5          if(n==0 || n==1)
6              return n;
7          else
8              return fibonacci(n-1)+fibonacci(n-2);
9      }
10 }
11 public class Recursion {
12     public static void main(String args[]){
13         student ob1=new student(); //student კლასის ობიექტის შექმნა
14         for(int i=0; i<11; i++)
15             System.out.print(ob1.fibonacci(i) + "\t");
16             System.out.println("\nFinish");
17     }
18 }
```

ნახ. 127

შედეგი:

```
0       1       1       2       3       5       8       13      21      34      55
Finish
```

ნახ. 128

მაგალითი 7.3. შევადგინოთ პროგრამა, რომელიც რეკურსიული გზით ახარისხების ფუნქციას წარმოგვიდგენს.

ნახ. 129-ზე ნაჩვენებია დასმული ამოცანის ამოხსნის შედეგები, ხოლო ნახ.130 მისი კომპიუტერული რეალიზაცია.

შედეგი:

```
2^10=1024
```

ნახ. 129

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package recursia;
2  //ახარისხების ოპერაციის რეკურსიული გზით წარმოდგენა
3  class student{
4      int a, x;
5      student(int a, int x){ //კლასის პარამეტრებიანი კონსტრუქტორი
6          this.a=a;
7          this.x=x;
8      }
9      long axarisxeba(int a, int x){ // ახარისხების რეკურსიული მეთოდი
10         if(x==0)
11             return 1;  //ნებისმიერი რიცხვი 0 ხარისხში ერთის ტოლია
12         if(x==1)
13             return a;  //ნებისმიერი რიცხვი 1 ხარისხში თავისი თავის ტოლია
14         else
15             return a*axarisxeba(a, x-1);
16     }
17 }
18 public class Recursion {
19     public static void main(String args[]){
20         int a=2, x=10;
21         student ob1=new student(a, x); //student კლასის ობიექტის შექმნა
22             System.out.println(a + "^" + x + "=" + ob1.axarisxeba(a, x));
23     }
24 }
```

ნახ. 130

რეკურსიულ მეთოდებთან დაკავშირებული ვიდეო მასალა შეგიძლიათ იხილოთ ბმულ-ზე:https://www.youtube.com/watch?v=WOUpjal8ee4&list=PLGwb7xZHg-oMv1pOlTHAqAEjw0EPALzlL ვიდეო-ფაილი 24.

დავალება

1. *შეადგინეთ პროგრამა, რომელიც რეკურსიული მეთოდის გამოყენებით განსაზღვრავს ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფის მნიშვნელობას.*
2. *შეადგინეთ პროგრამა, რომელიც რეკურსიული მეთოდის გამოყენებით გამოთვლის a=5-დან b=20-მდე მთელი ტიპის რიცხვითი მონაცემების ჯამს.*
3. *შეადგინეთ პროგრამა, რომელიც რეკურსიული მეთოდის გამოყენებით გამოთვლის m=1-დან n=10-მდე მთელი ტიპის რიცხვითი მონაცემების ნამრავლს.*
4. *შეადგინეთ პროგრამა, რომელიც რეკურსიული მეთოდების გამოყენებით გამოთვლის $y = a^x$ გამოსახულების მნიშვნელობას, თუ a და x კენტი რიცხვებია, წინააღმდეგ შემთხვევაში $y = a! - x$ გამოსახულების მნიშვნელობას.*
5. *შეადგინეთ პროგრამა, რომელიც რეკურსიული მეთოდის გამოყენებით გამოთვლის ფიბონაჩის რიცხვითი მიმდევრობის მე-15 წევრის მნიშვნელობას.*

3.10. კლასის წევრებზე წვდომის მართვა და კლასის სტატიკური წევრები

ცნობილია, რომ ინკაფსულაცია მონაცემებს კოდთან აკავშირებს. ამას გარდა, ის კიდევ ერთ მნიშვნელოვან ატრიბუტს წარმოგვიდგენს: **წვდომის მართვას**, ანუ ინკაფსულაცია საშუალებას გვაძლევს ვმართოთ, თუ პროგრამის რა ნაწილებს ექნებათ წვდომა კლასის წევრებზე. მაგალითად, თუ კლასის მონაცემებზე წვდომა ექნებათ მხოლოდ მკაცრად განსაზღვრულ მეთოდების ნაკრებს, მაშინ თავიდან ავიცილებთ აღნიშნული მონაცემების „ბოროტად“ გამოყენებას. ამგვარად, როდესაც კლასი სწორადაა რეალიზებული, ის ქმნის ერთგვარ „შავ ყუთს“, რომლის შიდა მექანიზმი დაცულია დაზიანებისგან და ამასთან, ის შეგვიძლია გამოვიყენოთ.

3.10.1. კლასის წევრებზე წვდომის მართვა

კლასის წევრებზე წვდომას განსაზღვრავს **წვდომის მოდიფიკატორები**, რომლებიც მათ გაცხადებას თან ახლავს. Java ენის წვდომის მოდიფიკატორებია: **public (ღია), private (დახურული)** და **protected (დაცული).** Java ასევე განსაზღვრავს სტანდარტული წვდომის დონესაც (მოდიფიკატორის ცხადი სახით მიუთითებლობის შემთხვევაში).

protected (დაცული) მოდიფიკატორი მხოლოდ მემკვიდრეობითობის შემთხვევაში გამოიყენება.

ახლა კი შევუდგეთ **public** (ღია) და **private** (დახურული) მოდიფიკატორების განსაზღვრას. როდესაც კლასის წევრზე **public** წვდომის მოდიფიკატორია გამოყენებული, ის ნებისმიერი სხვა კოდისთვის წვდომადი ხდება. როდესაც კლასის წევრი მითითებულია, როგორც **private**, ის წვდომადია მხოლოდ იმავე კლასის სხვა წევრებისთვის. ახლა, თქვენთვის გასაგები და ნათელი გახდა, თუ რატომ უსწრებს წინ **main()** მეთოდს **public** მოდიფიკატორი. აღნიშნულ მეთოდს იძახებს კოდი, რომელიც განთავსებულია მოცემული პროგრამის გარეთ.

წვდომის მოდიფიკატორის მიუთითებლობის შემთხვევაში, კლასის წევრი საკუთარი პაკეტის ფარგლებში (შიგნით) ღიად ითვლება, მაგრამ ის მიუწვდომელია კოდისთვის, რომელიც ამ პაკეტის ფარგლებს გარეთაა წარმოდგენილი. როგორც წესი, სასურველია კლასის მონაცემებზე თავისუფალი წვდომა შევზღუდოთ და ეს უკანასკნელი მხოლოდ კლასის მეთოდების საშუალებით განვახორციელოთ. მოდიფიკატორი კლასის წევრის ტიპის ყველა სპეციფიკაციას წინ უსწრებს, ანუ კლასის წევრის გაცხადების ოპერატორი წვდომის მოდიფიკატორით უნდა იწყებოდეს. მაგალითად:

```
public int x;
private double y;
private int myMethod( ){ //… }
```

იმისათვის, რომ გასაგები გახდეს ღია და დახურული წვდომის გამოყენების გავლენა, განვიხილოთ შემდეგი **მაგალითი 1.** შევადგინოთ პროგრამა, რომელიც მთელრიცხვა int a[10] მასივის ელემენტებს მიანიჭებს ნებისმიერ მნიშვნელობებს და განსაზღვრავს მასივის უდიდესი და უმცირესი მნიშვნელობის ელემენტებს და მათ ინდექსებს. კლასის წევრებზე გამოვიყენოთ წვდომის მოდიფიკატორები.

ნახ.131-ზე წარმოდგენილია დასმული ამოცანის გადაწვეტის შედეგი და შემდგომ, ნახ.132-ზე მისი პროგრამული რეალიზაცია.

შედეგი:

```
საწყისი მასივი:
4       21      3       8       22      12      16      19      16      16
მაქსიმალური მნიშვნელობა=22
მაქსიმალური ელემენტის ინდექსი=4
მინიმალური მნიშვნელობა=3
მინიმალური ელემენტის ინდექსი=2
```

ნახ. 131

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package masivebi;
2  import java.util.Random;
3  //წვდომის მოდიფიკატორების გამოყენება
4  class student{
5      public int a[]=new int[10];
6      private int maxi, mini, k, p;
7      student(){
8      Random rand=new Random();
9      for(int i=0; i<a.length; i++){
10         a[i]=(int)(25*rand.nextDouble()); //შემთხვევითი რიცხვების გენერირება
11         System.out.print(a[i] + "\t");}}
12     public void method1(){ //მაქსიმალური ელემენტის და მისი ინდექსის განსაზღვრა
13         maxi=a[0]; k=0;
14         for(int i=0; i<a.length; i++)
15             if(maxi<=a[i]){
16                 maxi=a[i];
17                 k=i;
18             }
19         System.out.println("\nმაქსიმალური მნიშვნელობა=" + maxi);
20         System.out.println("მაქსიმალური ელემენტის ინდექსი=" + k);
21     }
22     public void method2(){ //მინიმალური ელემენტის და მისი ინდექსის განსაზღვრა
23         mini=a[0]; p=0;
24         for(int i=0; i<a.length; i++)
25             if(mini>=a[i]){
26                 mini=a[i];
27                 p=i;}
28         System.out.println("მინიმალური მნიშვნელობა=" + mini);
29         System.out.println("მინიმალური ელემენტის ინდექსი=" + p);}}
30 public class Masivi {
31     public static void main(String args[]){
32         System.out.println("საწყისი მასივი:");
33         student object1=new student();
34         object1.method1();
35         object1.method2();
36         //object1.k; დაუშვებელი ოპერაციაა!
37     }}
```

ნახ. 132

კლასის წევრებზე წვდომის მართვის ნიმუშები შეგიძლიათ იხილოთ ბმულზე: https://www.thenewboston.com/videos.php?cat=31 ვიდეო-ფაილი: 38.

3.10.2. კლასის სტატიკური წევრები

ზოგ შემთხვევაში, სასურველია განვსაზღვროთ კლასის წევრი, რომელიც მოცემული კლასის ნებისმიერი ობიექტისგან დამოუკიდებლად გამოიყენება. კლასის ასეთი წევრის შესაქმნელად აუცილებელია მის გამოცხადებას წინ დავურთოთ **static** საკვანძო სიტყვა. როდესაც კლასის წევრი ცხადდება როგორც სტატიკური (**static**), ის მისაწვდომი ხდება თავისი კლასის ნებისმიერი ობიექტის შექმნამდე. სტატიკური შესაძლებელია იყოს როგორც მეთოდები, ასევე ცვლადები. სტატიკური წევრის ყველაზე თვალსაჩინო მაგალითს **main()** მეთოდი წარმოადგენს. აღნიშნული მეთოდი სტატიკურია, რადგან ის ნებისმიერი ობიექტის შექმნამდე უნდა იქნეს გაცხადებული.

სტატიკური მეთოდების გამოყენების დროს შემდეგი შეზღუდვები უნდა გავითვალისწინოთ:

- მათ უშუალოდ, მხოლოდ სხვა სტატიკური მეთოდების გამოძახება შეუძლიათ;
- უშუალო წვდომა მათ მხოლოდ სტატიკურ ცვლადებზე გააჩნიათ;
- მათ არ შეუძლიათ this და super ტიპის წევრებზე მიმართვა.

სტატიკური მეთოდები და ცვლადები, თავიანთი კლასის გარეთ შეგვიძლია ობიექტებისგან დამოუკიდებლად გამოვიყენოთ. ამისათვის საკმარისია კლასის სახელის მითითება, რომელსაც წერტილის ოპერატორი უნდა მოყვებოდეს. მაგალითად, თუ სტატიკური მეთოდი კლასის გარეთ გვსურს გამოვიძახოთ, საჭიროა შემდეგი ზოგადი ფორმის გამოყენება:

კლასის_სახელი.მეთოდი();

აქ კლასის_სახელის ქვეშ იმ კლასის სახელი იგულისხმება, რომელშიც სტატიკური მეთოდია გამოცხადებული. ანალოგიურად ვიქცევით სტატიკური ცვლადების შემთხვევაშიც. სწორედ, ამ გზით ხდება Java-ში გლობალური მეთოდებისა და ცვლადების მმართველი ვერსიების რეალიზება.

ზემოაღნიშნულის საილუსტრაციოდ განვიხილოთ **მაგალითი 2.** შევადგინოთ პროგრამა, რომელიც კლასის სტატიკური მეთოდის საშუალებით ერთიდან 10-ის ჩათვლით რიცხვითი მონაცემების გამრავლების ტაბულას წარმოგვიდგენს. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 133-ზე, ხოლო შესრულების შედეგი ნახ. 134-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package masivebi;
2  //გამრავლების ტაბულა
3  class student{
4      student(){
5          System.out.println("გამრავლების ტაბულა:");
6      }
7      static void method1(){   //სტატიკური მეთოდი
8          for(int i=1; i<=10; i++)
9              System.out.print("\t" + i);
10         System.out.println();
11         for(int i=1; i<=10; i++){
12             System.out.print(i + "\t");
13             for(int j=1; j<=10; j++){
14                 System.out.print(i*j + "\t");
15             }
16             System.out.println();
17         }
18       }
19     }
20 public class Masivi {
21     public static void main(String args[]){
22         student.method1(); //სტატიკური მეთოდის გამოძახება ობიექტის შექმნამდე
23      }
24     }
```

ნახ. 133

შედეგი:

	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

ნახ. 134

კლასის სტატიკურ წევრებთან დაკავშირებული ვიდეო მასალა შეგიძლიათ იხილოთ ბმულზე: https://www.thenewboston.com/videos.php?cat=31 ვიდეო-ფაილი 46.

დავალება

1. შეადგინეთ პროგრამა, რომელიც კლასის სტატიკური მეთოდის გამოყენებით 0-დან 10000-მდე იპოვის ყველა ისეთ რიცხვს, რომლის ციფრების ჯამი ამავე ციფრების ნამრავლის ტოლი იქნება.
2. შეადგინეთ პროგრამა, რომელიც კლასის სტატიკური მეთოდის გამოყენებით ერთიდან 10000-მდე იპოვის ყველა წესიერ რიცხვს, რაც გულისხმობს იმ ფაქტს, რომ რიცხვის გამყოფების ჯამი ამავე რიცხვის ტოლი უნდა იყოს.
3. შეადგინეთ პროგრამა, რომელიც კლასის წევრებზე წვდომის მოდიფიკატორების გამოყენებით 20-ელემენტიან მთელრიცხვა მასივში გამოთვლის კენტინდექსიანი და ამავდროულად, კენტი მნიშვნელობის მქონე ელემენტების საშუალო არითმეტიკულს.
4. შეადგინეთ პროგრამა, რომელიც კლასის წევრებზე წვდომის მოდიფიკატორების გამოყენებით 4x4 განზომილების მქონე კვადრატულ მატრიცაში ცალ-ცალკე დათვლის კენტი და ლუწი მნიშვნელობის ელემენტების რაოდენობას და მიღებულ შედეგებს ერთმანეთს შეადარებს.
5. შეადგინეთ პროგრამა, რომელიც კლასის სტატიკური მეთოდის გამოყენებით ჰერონის ფორმულის საფუძველზე გამოთვლის სამკუთხედის ფართობს (დაიცავით სამკუთხედის აგების წესი, რაც გულისხმობს იმ ფაქტს, რომ სამკუთხედის ნებისმიერი ორი გვერდის ჯამი მესამეზე მეტი უნდა იყოს. კვადრატული ფესვის წარმოსადგენად (რაც ფართობის ფორმულაში დაგჭირდებათ) გამოიყენეთ მათემატიკური ფუნქცია შემდეგი ჩანაწერის გათვალისწინებით: S=Math.sqrt(p*(p-a)*(p-b)*(p-c)); სადაც p - სამკუთხედის ნახევარპერიმეტრია, ხოლო a, b და c - სამკუთხედის გვერდები, S კი - ფართობი.

3.11. მემკვიდრეობითობა

3.11.1. მემკვიდრეობითობის არსი.

კლასის წევრებზე წვდომა მემკვიდროებითობის დროს

ობიექტზე ორიენტირებული დაპროგრამების ერთ–ერთ ფუნდამენტურ ცნებას **მემკვიდრეობითობა** წარმოადგენს. ის საშუალებას გვაძლევს იერარქიული კლასიფიკაციები შევქმნათ. მემკვიდრეობითობის გამოყენებით შეიძლება საერთო კლასი შეიქმნას, რომელიც ერთმანეთთან დაკავშირებული ელემენტების ნაკრების საერთო მახასიათებლებს განსაზღვრავს. შემდგომ ეს კლასი მემკვიდრეობით შეიძლება მიიღოს გაცილებით უფრო სპეციალიზირებულმა კლასებმა, რომელთაგან თვითოეულ მათგანს თავისი უნიკალური მახასიათებლების დამატება შეეძლება. Java-ს ტერმინოლოგიაში მშობელ კლასს **სუპერ კლასი** ეწოდება, ხოლო მემკვიდრე კლასს კი – **ქვეკლასი**. შესაბამისად, ქვეკლასი სუპერ კლასის სპეციალიზირებული ვერსიაა. ის მემკვიდრეობით იღებს სუპერ კლასის მიერ განსაზღვრულ ყველა წევრს და უმატებს საკუთარ, უნიკალურ ელემენტებს.

მემკვიდრეობითობის განსახორციელებლად საკმარისია ერთი კლასის განსაზღვრა მეორე კლასში მოვათავსოთ **extends** საკვანძო სიტყვის გამოყენებით.

ქვეკლასის გამოცხადების ზოგადი ფორმა შემდეგია:

```
class ქვეკლასის_სახელი extends სუპერ კლასის_სახელი{
//კლასის ტანი; }
```

ყოველ ქვეკლასს მხოლოდ ერთი სუპერკლასი უნდა მიეთითოს. Java-ში დაუშვებელია რამდენიმე სუპერკლასისგან ერთი ქვეკლასის მემკვიდრეობა.

შესაძლებელია მემკვიდრეობითობის იერარქიის შექმნა, სადაც ქვეკლასი სხვა ქვეკლასის სუპერკლასი ხდება. მართლია, ქვეკლასი სუპერკლასის ყველა წევრს თავის თავში მოიცავს, მაგრამ მას არ აქვს წვდომა სუპერკლასის დახურულ (**private**) წევრებზე.

მემკვიდრეობითობის უპირატესობა იმაში მდგომარეობს, რომ ობიექტების ნაკრების საერთო ატრიბუტების განმსაზღვრელი სუპერკლასის შექმნისთანავე, ის შეგვიძლია ნებისმიერი რაოდენობის სპეციალიზირებული კლასების შესაქმნელად გამოვიყენოთ. ამასთან, ყოველი ქვე-კლასი ზუსტად განსაზღვრავს საკუთარ კლასიფიკაციას. ყოველ ქვეკლასს შეუძლია საკუთარი უნიკალური ატრიბუტების დამატება და სწორედ, ამაში მდგომარეობს მემკვიდრეობითობის არსებითი მნიშვნელობა.

მაგალითი 9.1. შევადგინოთ პროგრამა, რომელიც კლასების მემკვიდრეობითობის გამოყენების გზით int a[10] ელემენტიან მასივში გამოთვლის ელემენტების ჯამს, ნამრავლსა და საშუალო არითმეტიკულ მნიშვნელობას. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 135-ზე, ხოლო შესრულების შედეგი ნახ. 136-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package memkvidrerobitoba;
2  import java.util.Random;
3  //მემკვიდრეობითობა
4  class student{
5      private int s; //დახურულ s  ცვლადზე წვდომა არ აქვს მემკვიდრე კლასის წევრებს
6      public double p;
7      public int a[]=new int[10];
8      student(){
9          s=0; p=1;
10         Random rand=new Random();
11         for(int i=0; i<a.length; i++){
12             a[i]=(int)(20*rand.nextDouble());
13             System.out.print(a[i] + "\t");
14         }
15     }
16     public void method1(){
17         for(int x :a)  // for ციკლის გამოყენება for-each სტილში
18             s+=x;
19         System.out.println("\nჯამი=" + s);
20         System.out.println("საშუალო არითმეტიკული=" + s/10.);
21     }}
22     class student1 extends student{ //მემკვიდრე კლასის შექმნა
23         public void method2(){
24             for(int x :a)
25                 p*=x;
26             System.out.println("ნამრავლი=" + p);
27         }
28        }
29 public class Inher {
30     public static void main(String args []){
31         System.out.println("საწყისი მასივი:");
32         student1 ob1=new student1(); //მემკვიდრე კლასის ობიექტის შექმნა
33         ob1.method1();
34         ob1.method2();
35     }}
36
```

ნახ. 135

შედეგი:

```
საწყისი მასივი:
14      13      18      17      19      16      17      12      3       10
ჯამი=139
საშუალო არითმეტიკული=13.9
ნამრავლი=1.0361385216E11
```

ნახ. 136

3.11.2. super საკვანძო სიტყვის გამოყენება

რიგ შემთხვევებში საჭირო ხდება ისეთი სუპერკლასის შექმნა, რომლის რეალიზაციის დეტალები წვდომადია მხოლოდ მისთვის (ანუ ადგილი აქვს მონაცემთა დახურული წევრების გამოყენებას). ასეთ დროს ქვეკლასი დამოუკიდებლად (უშუალოდ) ვერ მიმართავს დახურულ ცვლადებს და ვერ ახდენს მათ ინიციალებას. რადგან ინკაფსულაცია ობიექტზე ორიენტირებული დაპროგრამების ერთ–ერთი მთავარი ატრიბუტია, გასაკვირი არ არის, რომ Java ამ პრობლემის გადაწყვეტას გვთავაზობს. ყველა შემთხვევაში, როდესაც ქვეკლასმა უშუალოდ უნდა მიმართოს თავის სუპერკლასს, შეგვიძლია **super** საკვანძო სიტყვა გამოვიყენოთ. მას ორი ზოგადი ფორმა გააჩნია. პირველი გამოიყენება სუპერკლასის კონსტრუქტორის გამოსაძახებლად, ხოლო მეორე – სუპერკლასის იმ წევრზე მიმართვისათვის, რომელიც ქვეკლასის წევრის მიერაა დაფარული. განვიხილოთ ორივე ფორმა.

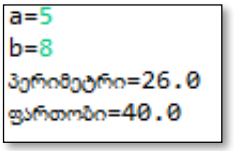
ქვეკლასს შეუძლია თავისი სუპერკლასის მიერ განსაზღვრული კონსტრუქტორის გამოძახება **super** საკვანძო სიტყვის შემდეგი ფორმის გამოყენებით:

```
super(არგუმენტების_სია);
```

აქ **არგუმენტების_სია** განსაზღვრავს ნებისმიერ არგუმენტს, რომელიც სუპერკლასის კონსტრუქტორს ესაჭიროება. **super()** მეთოდის გამოძახება ყოველთვის პირველ ოპერატორს უნდა წარმოადგენდეს, რომელიც ქვეკლასის კონსტრუქტორში სრულდება.

super() მეთოდის საილუსტრაციოდ განვიხილოთ **მაგალითი 2.** შევადგინოთ პროგრამა, რომელიც კლასების მემკვიდრეობითობის გამოყენებით გამოთვლის მართკუთხედის პერიმეტრსა და ფართობს. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 138-ზე, ხოლო შესრულების შედეგი ნახ. 137-ზე.

შედეგი:



ნახ. 137

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package memkvidrerobitoba;
2  import java.util.Scanner;
3  //მართკუთხედის პერიმეტრის და ფართობის გამოთვლა
4  class student{
5       double a;
6      student(double a){
7          this.a=a;
8      }
9      }
10     class student1 extends student{
11         double b;
12         student1(double a, double b){
13             super(a); //super საკვანძო სიტყვის გამოყენების პირველი ფორმა
14             this.b=b;
15         }
16         void method1(){
17             double p;
18             p=2*(a+b);
19             System.out.println("პერიმეტრი=" + p);
20         }
21         void method2(){
22             double s;
23             s=a*b;
24             System.out.println("ფართობი=" + s);}}
25 public class Inher {
26     public static void main(String args []){
27         double a, b;
28         Scanner ob1=new Scanner(System.in);
29         System.out.print("a=");
30         a=ob1.nextDouble();
31         System.out.print("b=");
32         b=ob1.nextDouble();
33         student1 ob2=new student1(a, b); //მემკვიდრე კლასის ობიექტის შექმნა
34         ob2.method1();
35         ob2.method2();
36     }}
```

ნახ. 138

განვიხილოთ super() მეთოდის გამოყენების ძირითადი კონცეფციები. როდესაც ქვეკლასი super() მეთოდს იძახებს, ამ დროს ის უშუალოდ თავისი სუპერკლასის კონსტრუქტორის გამოძახებას ახდენს. ამგვარად, super() მეთოდი ყოველთვის მიმართავს სუპერკლასს, რომელიც კლასების იერარქიულ სტრუქტურაში უშუალოდ გამომძახებელი კლასის ზემოთ მდებარეობს. ეს სამართლიანია მრავალდონიანი იერარქიის შემთხვევაშიც. ამასთან, super() მეთოდი ყოველთვის პირველი შესასრულებელი ოპერატორი უნდა იყოს ქვეკლასის კონსტრუქტორში.

super საკვანძო სიტყვის გამოყენების მეორე ფორმა მსგავსია this საკვანძო სიტყვის გამოყენების, იმ განსხვავებით, რომ super საკვანძო სიტყვა ყოველთვის მიმართავს იმ ქვეკლასის სუპერკლასს, რომელშიც ის გამოიყენება.

super საკვანძო სიტყვის მეორე ზოგად ფორმას შემდეგი სახე აქვს:

super.წევრი;

აქ **წევრის** ქვეშ იგულისხმება მეთოდი ან ეგზემპლიარის ცვლადი. super საკვანძო სიტყვის გამოყენების მეორე ფორმა იმ სიტუაციებში გვხვდება, როდესაც ქვეკლასის წევრების სახელები ფარავენ სუპერკლასის იმავე სახელის მქონე წევრებს.

ზემოაღნიშნულის საილუსტრაციოდ განვიხილოთ super საკვანძო სიტყვის გამოყენების მეორე ფორმის ამსახველი **მაგალითი 3**, რომელშიც A სუპერ კლასსა და B ქვეკლასში არსებულ ერთსა და იმავე სახელის მქონე i ცვლადზე მიმართვა შესრულებულია super საკვანძო სიტყვის გამოყენებით და მის გარეშე. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 139-ზე, ხოლო შესრულების შედეგი ნახ. 140-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package memkvidrerobitoba;
2
3  //super საკვანძო სიტყვის გამოყენების მეორე ფორმა
4  class A{
5      int i;
6       }
7      class B extends A{
8          int i;
9          B(int a, int b){
10             super.i=a; //მიმართვა A სუპერ კლასის i ცვლადზე
11             i=b;       //მიმართვა B ქვეკლასის i ცვლადზე
12         }
13         void show(){
14             System.out.println("i სუპერ კლასში=" + super.i);
15             System.out.println("i ქვეკლასში=" + i);
16         }
17         }
18 public class Inher {
19     public static void main(String args []){
20       B object1=new B(5, 10);
21       object1.show();
22     }}
```

ნახ. 139

შედეგი:

```
i სუპერ კლასში=5
i ქვეკლასში=10
```

ნახ. 140

ზემოთ წარმოდგენილ მაგალითში მართალია, B კლასის i ცვლადი ეგზემპლიარის მიერ დაფარულია A სუპერ კლასის i ცვლადი, მაგრამ super საკვანძო სიტყვა საშუალებას გვაძლევს წვდომა ვიქონიოთ სუპერკლასში განსაზღვრულ i ცვლადზე.

3.11.3. მრავალდონიანი იერარქიის შექმნა

აქამდე ჩვენ კლასების მარტივ იერარქიას ვიყენებდით, რომელიც მხოლოდ სუპერკლასისა და ქვეკლასისგან შედგებოდა. თუმცა, შეგვიძლია შევქმნათ იერარქიები, რომლებიც მემკვიდრეობითობის დონეების ნებისმიერ რაოდენობას მოიცავს. როგორც ვიცით, ქვეკლასი შეიძლება სხვა ქვეკლასის სუპერკლასის როლში წარმოვადგინოთ. ზოგადად, C კლასი შეიძლება იყოს B კლასის ქვეკლასი, რომელიც თავის მხრივ, A კლასის ქვეკლასია. მსგავს სიტუაციებში

ყოველი ქვეკლასი მემკვიდრეობით იღებს ყველა მისი სუპერკლასის მახასიათებლებს. ანუ C კლასი მემკვიდრეობით მიიღებს A და B კლასების ყველა მახასიათებელს.

მრავალდონიანი იერარქიის საილუსტრაციოდ განვიხილოთ **მაგალითი 4.** შევადგინოთ პროგრამა, რომელიც კლასების იერარქიული სტრუქტურის გამოყენებით სამკუთხედის აგების წესის დაცვით ჰერონის ფორმულის საფუძველზე სამკუთხედის ფართობს ითვლის.

ნახ. 141-ზე წარმოდგენილია დასმული ამოცანის ამოხსნის შედეგი, ხოლო ნახ.142-ზე სურათზე შესაბამისი პროგრამის კომპიუტერული რეალიზაცია.

შედეგი:

```
a=4
b=5
c=7
p/2=8.0
S=9.797958971132712
```

ნახ. 141

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package memkvidrerobitoba;
2  import java.util.Scanner;
3  //სამკუთედის ფართობის გამოთვლა
4  class student{  //სუპერ კლასი
5      double a;
6      student(double a){
7          this.a=a;}
8       }
9      class student1 extends student{ //student კლასის მემკვიდრე კლასი
10         double b;
11         student1(double a, double b){
12             super(a);
13             this.b=b;}
14     }
15         class student2 extends student1{ // student1 კლასის მემკვიდრე კლასი
16             double c;
17             student2(double a, double b, double c){
18                 super(a, b);
19                 this.c=c;}
20             void method1(){
21                 double p, s;
22                 if(a+b>c && a+c>b && b+c>a){
23                     p=(a+b+c)/2;
24                     s=Math.sqrt(p*(p-a)*(p-b)*(p-c));
25                     System.out.println("p/2=" + p);
26                     System.out.println("S=" + s);}
27                 else
28                 System.out.println("სამკუთხედის აგების წესი დარღვეულია.");}}
29 public class Inher {
30     public static void main(String args []){
31         double a, b, c;
32         Scanner object1=new Scanner(System.in);
33         System.out.print("a=");
34         a=object1.nextDouble();
35         System.out.print("b=");
36         b=object1.nextDouble();
37         System.out.print("c=");
38         c=object1.nextDouble();
39       student2 object2=new student2(a, b, c);
40       object2.method1();
41     }}
```

ნახ. 142

მემკვიდრეობითობასთან დაკავშირებული ვიდეო-მასალა შეგიძლიათ იხილოთ ბმულზე: https://www.thenewboston.com/videos.php?cat=31 ვიდეო-ფაილი 49.

დავალება

1. *შეადგინეთ პროგრამა, რომელიც კლასების მემკვიდრეობითობის გამოყენების გზით int b[15] ელემენტიან მასივში გამოთვლის ლუწი მნიშვნელობის ელემენტების ჯამიდან კვადრატული ფესვის მნიშვნელობას.*
2. *შეადგინეთ პროგრამა, რომელიც კლასების მემკვიდრეობითობის გამოყენების გზით int c[17] ელემენტიან მასივში გამოთვლის კენტინდექსიანი ელემენტების საშუალო არითმეტიკულ მნიშვნელობას.*
3. *შეადგინეთ პროგრამა, რომელიც კლასების მემკვიდრეობითობის გამოყენების გზით int a[7][7] კვადრატულ მატრიცაში გამოთვლის ელემენტების კვადრატების ჯამის მნიშვნელობებს.*
4. *შეადგინეთ პროგრამა, რომელიც კლასების მემკვიდრეობითობის გამოყენების გზით int a[4][4] კვადრატულ მატრიცაში გამოთვლის 10-ზე მეტი მნიშვნელობის მქონე ელემენტების ნამრავლს (მატრიცის ელემენტებს თავად მიანიჭეთ ნებისმიერი მნიშვნელობები).*
5. *შეადგინეთ პროგრამა, რომელიც კლასების იერარქიულ სტრუქტურაში სუპერკლასების კონსტრუქტორების გამოძახების მიზნით super() მეთოდს გამოიყენებს და 15 ელემენტიან მთელრიცხვა მასივში გამოთვლის მაქსიმალური და მინიმალური მნიშვნელობის ელემენტების ნამრავლს.*
6. *შეადგინეთ პროგრამა, რომელიც კლასების იერარქიულ სტრუქტურაში სუპერკლასების კონსტრუქტორების გამოძახების მიზნით super() მეთოდს გამოიყენებს და 12 ელემენტიან მთელრიცხვა მასივში გამოთვლის ლუწი მნიშვნელობის ელემენტების საშუალო არითმეტიკულს და კენტი მნიშვნელობის ელემენტების ჯამს.*
7. *შეადგინეთ პროგრამა, რომელიც კლასების იერარქიულ სტრუქტურაში სუპერკლასების კონსტრუქტორების გამოძახების მიზნით super() მეთოდს გამოიყენებს და 4x4 განზომილებიან მატრიცაში ცალ-ცალკე გამოთვლის მთავარ და არამთავარ დიაგონალებზე არსებული ელემენტების ნამრავლებს.*

3.12. მეთოდების ხელახალი განსაზღვრა. final საკვანძო სიტყვის გამოყენება

3.12.1. მეთოდების ხელახალი განსაზღვრა

როდესაც კლასების იერარქიაში ქვეკლასის მეთოდის სახელი და ტიპის სიგნატურა ემთხვევა სუპერკლასის მეთოდის ატრიბუტებს, ამბობენ, რომ ქვეკლასის მეთოდი **ხელახლა განსაზღვრავს** სუპერკლასის მეთოდს.

როდესაც ხელახლა განსაზღვრული მეთოდი გამოიძახება თავისი ქვეკლასიდან, ის ყოველთვის მიმართავს ქვეკლასის მიერ განსაზღვრულ მეთოდის ვერსიას. სუპერკლასის მიერ განსაზღვრული მეთოდი კი იფარება.

ზემოაღნიშნულის საილუსტრაციოდ განვიხილოთ **მაგალითი10.1**. შევადგინოთ პროგრამა, რომელიც 12-ელემენტიან მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს და გამოთვლის მასივის 10-ზე მეტი მნიშვნელობის ელემენტების ნამრავლს სუპერ კლასის method1() მეთოდით და 10-ზე ნაკლები მნიშვნელობის ელემენტების კვადრატების ჯამს ქვეკლასის method1() მეთოდით. ვინაიდან კლასების იერარქიაში ორი ერთი და იმავე სიგნატურის მქონე მეთოდი გვაქვს გამოყენებული, ამიტომ აღნიშნულ მაგალითში ქვეკლასის ობიექტის მიერ მეთოდზე მიმართვისას ხდება მისივე კლასში ხელახლა განსაზღვრული მეთოდის გამოძახება და არა სუპერკლასის method1() მეთოდის, რამეთუ, ეს უკანასკნელი დაფარულია. შედეგად ვიღებთ მხოლოდ 10-ზე ნაკლები მნიშვნელობის ელემენტების კვადრატების ჯამს.

დასმული ამოცანის ამოხსნის შედეგები ნახ.143-ზეა წარმოდგენილი, ხოლო მისი პროგრამული რეალიზაცია ნახ. 144.

შედეგი:

```
3       13      3       3       5       8       8       4       11      4       10      14
კვადრატების ჯამი=212.0
```

ნახ. 143

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package override1;
2  import java.util.Random;
3  //მეთოდების ხელახალი განსაზღვრა
4  class student{
5      int a[]=new int [12];
6      student(){
7          Random rand=new Random();
8          for(int i=0; i<a.length; i++){
9              a[i]=(int)(15*rand.nextDouble());
10             System.out.print(a[i] + "\t");
11         }
12     }
13     void method1(){
14         double s=1;
15         for(int i=0; i<a.length; i++)
16             if(a[i]>10) s*=a[i];
17         System.out.println("\nნამრავლი="  + s);
18     }
19 }
20 class student1 extends student{
21     student1(){
22         super();}
23         void method1(){
24             double s=0;
25             for(int i=0; i<a.length; i++)
26             if(a[i]<10) s+=Math.pow(a[i], 2);
27         System.out.println("\nკვადრატების ჯამი="  + s);
28         }
29     }
30 public class Clasebi {
31     public static void main(String args []){
32         student1 ob1=new student1();
33         ob1.method1();
34     }}
35
```

ნახ. 144

თუ გვსურს, წვდომა გვქონდეს სუპერკლასში განსაზღვრულ მეთოდის ვერსიაზე, საჭიროა **super** საკვანძო სიტყვას მივმართოთ.

აღნიშნულის საილუსტრაციოდ, შეცვლილი სახით წარმოვადგინოთ 10.1 ამოცანის გადაწყვეტის ზემოთ განხილული პროგრამა. აქ მეთოდი super.method1() იძახებს სუპერკლასში განსაზღვრულ method1() მეთოდის ვერსიას.

პროგრამის შესრულებაზე გაშვების შედეგად დასმული ამოცანის ამოხსნისას ვიღებთ, როგორც 10-ზე მეტი მნიშვნელობის მქონე ელემენტების ნამრავლს, ისე 10-ზე ნაკლები მნიშვნელობის მქონე ელემენტების კვადრატების ჯამს. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 145-ზე, ხოლო შესრულების შედეგი ნახ. 146-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package override1;
2  import java.util.Random;
3  //მეთოდების ხელახალი განსაზღვრა
4  class student{
5      int a[]=new int [12];
6      student(){
7          Random rand=new Random();
8          for(int i=0; i<a.length; i++){
9              a[i]=(int)(15*rand.nextDouble());
10             System.out.print(a[i] + "\t");
11         }
12     }
13     void method1(){
14         double s=1;
15         for(int i=0; i<a.length; i++)
16             if(a[i]>10) s*=a[i];
17         System.out.println("\nნამრავლი="  + s);
18     }
19 }
20 class student1 extends student{
21     student1(){
22         super();}
23         void method1(){  //method1() მეთოდის ხელახალი განსაზღვრა
24         super. method1();  //მიმართვა სუპერ კლასის metdo1() მეთოდზე
25             double s=0;
26             for(int i=0; i<a.length; i++)
27             if(a[i]<10) s+=Math.pow(a[i], 2);
28         System.out.println("\nკვადრატების ჯამი="  + s);
29         }
30     }
31 public class Clasebi {
32     public static void main(String args []){
33         student1 ob1=new student1();
34         ob1.method1();
35     }}
```

ნახ. 145

შედეგი:

```
6       6       13      9       1       5       1       4       3       10      11      6
ნამრავლი=143.0

კვადრატების ჯამი=241.0
```

ნახ. 146

მეთოდის ხელახალი განსაზღვრა მხოლოდ იმ შემთხვევაში ხდება, როდესაც ორი მეთოდის სახელები და სიგნატურა **იდენტურია.** წინააღმდეგ შემთხვევაში, მეთოდები **გადატვირთულს** წარმოადგენს.

3.12.2. final საკვანძო სიტყვის გამოყენება

final საკვანძო სიტყვის გამოყენების სამი საშუალება არსებობს.

- ის შეიძლება გამოვიყენოთ სახელობითი კონსტანტის ექვივალენტის შესაქმნელად;
- მეთოდის ხელახლა განსაზღვრის აკრძალვის მიზნით;
- მემკვიდრეობითობის შეწყვეტის მიზნით.

ველი შეიძლება გამოცხადებულ იქნეს როგორც **final** (დამამთავრებელი). ეს უკანასკნელი საშუალებას გვაძლევს შევაჩეროთ ცვლადის მნიშვნელობის ცვლილება და ფაქტიურად, ის კონსტანტად (მუდმივ სიდიდედ) ვაქციოთ. ეს გულისხმობს, რომ ფინალური ველი მისი გაცხადების დროს უნდა იქნეს ინიციალებული. მაგალითად:

```
final int FILE_NEW=1;
final int FILE_OPEN=2;
final int FILE_SAVE=3;
final int FILE_NEWSAVEAS=4;
final int FILE_QUIT=5;
```

ამგვარად, პროგრამის სხვა დანარჩენ ნაწილებს შეუძლიათ **FILE_OPEN** ცვლადის მსგავსად, ზემოთ წარმოდგენილი ცვლადები გამოიყენონ როგორც მუდმივები, მათი მნიშვნელობების ცვლილების რისკის გარეშე. Java–დაპროგრამების პრაქტიკაში მიღებულია, რომ ფინალური ველების იდენტიფიკატორები ჩაიწეროს დიდი ასოებით, როგორც ეს ზემოთ გვაქვს წარმოდგენილი.

გარდა ველებისა, final საკვანძო სიტყვა შეიძლება გამოვიყენოთ მეთოდის პარამეტრებისა და ლოკალური ცვლადების გამოცხადებისას. ასეთ დროს მეთოდის ფარგლებში პარამეტრის მნიშვნელობის ცვლილება შეუძლებელია, ხოლო ლოკალური ცვლადის შემთხვევაში, მასზე მნიშვნელობის მინიჭება მხოლოდ ერთხელ ხდება შესაძლებელი.

მართალია, მეთოდების ხელახალი განსაზღვრა Java დაპროგრამების ერთ–ერთი მძლავრი საშუალებაა, მაგრამ არის შემთხვევები, როდესაც ამ პროცესს სასურველია თავი ავარიდოთ. იმისათვის, რომ ავკრძალოთ მეთოდის ხელახალი განსაზღვრა, საჭიროა მისი გაცხადების წინ final საკვანძო სიტყვა მივუთითოთ. ამგვარად, მეთოდები, რომლებიც გამოცხადებულია final საკვანძო სიტყვით, არ შეიძლება ხელახლა განისაზღვროს. პროგრამული კოდის შემდეგი ფრაგმენტი final საკვანძო სიტყვის გამოყენების ამ შემთხვევის ილუსტრირებას ახდენს.

```
class A{
final void meth{
System.out.println("ეს არის final მეთოდი.");
}}
```

```
class B extends A{
void meth(){  //შეცდომაა!!! ეს მეთოდი არ შეიძლება ხელახლა განისაზღვროს.
System.out.println("დაუშვებელია!");
```

რადგან meth() მეთოდი გამოცხადებულია, როგორც final, ამიტომ B კლასში მისი ხელახალი განსაზღვრა შეუძლებელია.

ზოგჯერ final საკვანძო სიტყვით გამოცხადებული მეთოდები პროგრამის წარმადობას უწყობს ხელს. კომპილატორს შეუძლია ასეთი მეთოდების გამოძახების ჩასმა, რამეთუ მისთვის „ცნობილია", რომ ქვეკლასის მიერ მათი ხელახალი განსაზღრა ვერ მოხდება.

ხშირად, მცირე ზომის ფინალური მეთოდის გამოძახების დროს, Java კომპილატორს შეუძლია ვირტუალური მანქანის ქვეპროგრამის კოდი უშუალოდ განათავსოს გამოძახებული მეთოდის კომპილირებულ კოდში და ამით მნიშვნელოვნად შეამციროს სისტემური რესურსების ის დანახარჯები, რომლებიც მეთოდის გამოძახებას უკავშირდება. ფინალური მეთოდების განთავსება გამომძახებელ კოდში – მხოლოდ პოტენციურ შესაძლებლობას წარმოადგენს. როგორც წესი, Java მეთოდების გამოძახებას დინამიურად წყვეტს, შესრულების დროს. ასეთ მიდგომას **გვიან დაკავშირებას** უწოდებენ. რადგან ფინალური მეთოდები ხელახლა არ განისაზღვრება, ამიტომ ასეთ მეთოდებზე მიმართვა კომპილაციის დროსაა შესაძლებელი, რასაც **ადრეულ დაკავშირებას** უწოდებენ.

ზოგჯერ საჭირო ხდება კლასებში მემკვიდრეობითობის შეწყვეტა. ამ მიზნით კლასის გამოცხადების წინ ასევე, final საკვანძო სიტყვა გამოიყენება. როდესაც კლასს ვაცხადებთ, როგორც ფინალურს, ასევე არაცხადი სახით ვახდენთ მისი ყველა მეთოდის ფინალურ მეთოდად გამოცხადებას.

ქვემოთ წარმოდგენილია ფინალური კლასის ნიმუში:

```
final class A{
//…
}
class B extends A{ //შეცდომაა! A კლასს რ შეიძლება ქვეკლასი გააჩნდეს!
//…
}
```

როგორც კომენტარიდან ჩანს, B კლასი არ შეიძლება წარმოებული იყოს A კლასისგან, რადგან ეს უკანასკნელი (A კლასი) გამოცხადებულია, როგორც ფინალური კლასი.

მოცემულ თავთან დაკავშირებული ვიდეო-მასალა შეგიძლიათ იხილოთ ბმულზე: https://www.thenewboston.com/videos.php?cat=31 ვიდეო-ფაილი: 48.

დავალება

1. *შეადგინეთ პროგრამა, რომელიც მეთოდის ხელახალი განსაზღვრის გზით 6x6 კვადრატულ მატრიცაში შესაბამისი მეთოდის გამოყენებით სტრიქონების მიხედვით განსაზღვრავს ჯერ მაქსიმალური მნიშვნელობის ელემენტებს და შემდეგ იმავე მეთოდის ხელახლა განსაზღვრული ვერსიის გამოყენებით - მინიმალური მნიშვნელობის ელემენტებს.*
2. *შეადგინეთ პროგრამა, რომელიც მეთოდის ხელახალი განსაზღვრის გზით 4x4 კვადრატულ მატრიცაში შესაბამისი მეთოდის გამოყენებით გამოთვლის ჯერ მატრიცის არამთავარი დიაგონალის ზემოთ განლაგებული ელემენტების საშუალო არითმეტიკულ მნიშვნელობას და შემდეგ იმავე მეთოდის ხელახლა განსაზღვრული ვერსიის გამოყენებით - არამთავარი დიაგონალის ქვემოთ განლაგებული ელემენტების საშუალო არითმეტიკულ მნიშვნელობას.*
3. *შეადგინეთ პროგრამა, რომელიც მეთოდის ხელახალი განსაზღვრის გზით 10-ელემენტიან მთელრიცხვა მასივში შესაბამისი მეთოდის გამოყენებით გამოთვლის ჯერ ლუწი მნიშვნელობის ელემენტების ნამრავლს და შემდეგ იმავე მეთოდის ხელახლა განსაზღვრული ვერსიის გამოყენებით - კენტი მნიშვნელობის ელემენტების ნამრავლს.*

3.14. აბსტრაქტული კლასები. მათი გამოყენების ნიმუშები

3.14.1. აბსტრაქტული კლასები

ზოგ შემთხვევაში საჭიროა განისაზღვროს სუპერკლასი, რომელიც ყოველი მეთოდის სრული რეალიზაციის წარმოდგენის გარეშე ახდენს გარკვეული აბსტრაქციის სტრუქტურის გამოცხადებას. ანუ, გვიწევს ისეთი სუპერკლასის შექმნა, რომელიც განსაზღვრავს მხოლოდ განზოგადებულ ფორმას, რასაც მისი ყველა ქვეკლასი გარკვეული დეტალების დამატებით ერთობლივად გამოიყენებს. ასეთი კლასი იმ მეთოდების არსს განსაზღვრავს, რომელთა რეალიზაცია ქვეკლასებმა უნდა მოახდინოს. მსგავსი სიტუაცია იქმნება მაშინ, როდესაც სუპერ კლასს არ ძალუძს მეთოდის სრულყოფილი რეალიზაციის შემუშავება. იმისათვის, რომ მოვითხოვოთ, რომ გარკვეული მეთოდები ხელახლა განისაზღვროს ქვეკლასის მიერ, შეგვიძლია **abstract** ტიპის მოდიფიკატორი გამოვიყენოთ. ზოგჯერ ასეთ მეთოდებს ქვეკლასის კომპეტენციაში მყოფს უწოდებენ, რადგან სუპერკლასში მათი არანაირი რეალიზაცია გათვალისწინებული არ არის.

ამგვარად, ქვეკლასმა ხელახლა უნდა განსაზღვროს ეს მეთოდები, რადგან სუპერკლასში განსაზღვრული ამ მეთოდების ვერსიის გამოყენების უფლება მას არ აქვს. აბსტრაქტული მეთოდის გამოცხადების ზოგადი ფორმა შემდეგია:

abstract ტიპი სახელი(პარამეტრების_სია);

როგორც ხედავთ, აღნიშნული ფორმა მეთოდის ტანს არ მოიცავს. ნებისმიერი კლასი, რომელიც ერთ ან რამდენიმე აბსტრაქტულ მეთოდს შეიცავს, ასევე **აბსტრაქტულ კლასად** უნდა იყოს გამოცხადებული. ამისათვის საკმარისია კლასის გამოცხადებისას class საკვანძო სიტყვას წინ უსწრებდეს **abstract** მოდიფიკატორი. აბსტრაქტული კლასი არ შეიძლება შეიცავდეს ობიექტებს. ანუ, აბსტრაქტული კლასის უშუალო კონკრეტიზაცია new ოპერატორით დაუშვებელია. ასეთი ობიექტები უსარგებლო იქნებოდა, რადგან აბსტრაქტული კლასი სრულყოფილად არ არის განსაზღვრული. ასევე, დაუშვებელია აბსტრაქტული კონსტრუქტორებისა და აბსტრაქტული სტატიკური მეთოდების გამოცხადება. აბსტრაქტული კლასის ნებისმიერმა ქვეკლასმა ან უნდა მოახდინოს თავისი აბსტრაქტული სუპერ კლასის ყველა მეთოდის რეალიზება, ან თავადაც უნდა გამოცხადდეს, როგორც აბსტრაქტული კლასი.

3.14.2. აბსტრაქტული კლასების გამოყენების ნიმუშები

მაგალითი 1. შევადგინოთ პროგრამა, რომელიც აბსტრაქტული კლასის რეალიზების გზით ერთიდან *n*-მდე ყველა წესიერ რიცხვს (ანუ რიცხვს, რომლის გამყოფების ჯამი ამავე რიცხვის ტოლია) განსაზღვრავს. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 147-ზე, ხოლო შესრულების შედეგი ნახ. 148-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package abstractdemo;
2  abstract class student{ //აბსტრაქტული კლასის გამოცხადება
3      abstract void method1(); //აბსტრაქტული მეთოდის გამოცხადება
4  }
5  class student1 extends student{
6      void method1(){    //აბსტრაქტული მეთოდის რეალიზება ქვეკლასში
7          for(int n=1; n<10000; n++){
8              int sum=0, k=1;
9              while(k<n){
10                 if(n%k==0) sum+=k;
11                 k++;
12             }
13             if(n==sum) System.out.println("n=" + n);}}}
14   public class Abstr {
15       public static void main(String args []){
16           student1 ob1=new student1();
17           ob1.method1();}}
```

ნახ. 147

შედეგი:

```
n=6
n=28
n=496
n=8128
```

ნახ. 148

მაგალითი 2. შევადგინოთ პროგრამა, რომელიც აბსტრაქტული კლასის რეალიზების გზით ნებისმიერი მთელი დადებითი რიცხვის ყველა გამყოფს განსაზღვრავს. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 149-ზე, ხოლო შესრულების შედეგი ნახ. 150-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package abstractdemo;
2  //რიცხვის გამყოფების განსაზღვრა
3  abstract class student{ //აბსტრაქტული კლასის გამოცხადება
4      abstract void method1(); //აბსტრაქტული მეთოდის გამოცხადება
5  }
6  class student1 extends student{
7      void method1(){     //აბსტრაქტული მეთოდის რეალიზება ქვეკლასში
8          System.out.println("1024-ის გამყოფებია:");
9          int num=1024;
10         int half=num/2;
11         int div=2;
12         while(div<=half){
13             if(num%div==0)
14                 System.out.print(div + "\t");
15             div++;}}}
16   public class Abstr {
17       public static void main(String args []){
18           student1 ob1=new student1();
19           ob1.method1();
20           System.out.println(); }}
```

ნახ. 149

შედეგი:

```
1024-ის გამყოფებია:
2       4       8       16      32      64      128     256     512
```

ნახ. 150

აბსტრაქტულ კლასებთან დაკავშირებული ვიდეო-მასლა შეგიძლიათ იხილოთ ბმულზე: https://www.youtube.com/watch?v=Hl-zzrqQoSE&list=PLFE2CE09D83EE3E28 ვიდეო-ფაილი 58.

დავალება

1. *შეადგინეთ პროგრამა, რომელიც აბსტრაქტული კლასის რეალიზების გზით გამრავლების ტაბულას კონსოლზე გამოიტანს.*
2. *შეადგინეთ პროგრამა, რომელიც აბსტრაქტული კლასის რეალიზების გზით ერთიდან 1000-მდე ყველა მარტივ რიცხვს კონსოლზე გამოიტანს.*
3. *შეადგინეთ პროგრამა, რომელიც აბსტრაქტული კლასის რეალიზების გზით ორი ნატურალური რიცხვის უმცირესი საერთო ჯერადის მნიშვნელობას განსაზღვრავს.*

3.15. პაკეტები და ინტერფეისები

ახლა განვიხილოთ Java ენის ორი მნიშვნელოვანი ნოვატორული კონცეფცია: **პაკეტები** და **ინტერფეისები.**

პაკეტები კლასების კონტეინერებია. მაგალითად, პაკეტი საშუალებას გვაძლევს შევქმნათ List სახელის მქონე კლასი და ეს უკანასკნელი შევინახოთ ცალკე პაკეტში ისე, რომ არ ვიფიქროთ შესაძლო კონფლიქტზე სხვა List კლასთან, რომელიც სხვაგანაა შენახული. პაკეტები, როგორც წესი, იერარქიულ სტრუქტურაში ინახება და ახალი კლასების განსაზღვრისას ცხადი სახით ხდება მათი იმპორტირება.

საკვანძო სიტყვა **interface** საშუალებას გვაძლევს ინტერფეისი მთლიანად გამოვყოთ მისი რეალიზაციისგან. ზემოაღნიშნული საკვანძო სიტყვის გამოყენებით შეგვიძლია მივუთითოთ იმ მეთოდების ნაკრები, რომლის რეალიზებას ერთი ან რამდენიმე კლასი განახორციელებს. ინტერფეისები აბსტრაქტული კლასების მსგავსია, თუმცა მათ დამატებითი შესაძლებლობებიც გააჩნიათ: ერთ კლასს ერთზე მეტი ინტერფეისის რეალიზება შეუძლია. და პირიქით, კლასი მემკვიდრეობით შეიძლება მიღებულ იქნეს მხოლოდ ერთი სუპერკლასისგან (აბსტრაქტულის ან არააბსტრაქტულისგან).

3.15.1. პაკეტების განსაზღვრა

Java ენაში გამოყენებულია სახელების სივრცის განცალკევების მექანიზმი და მას **პაკეტი** წარმოადგენს. ის გამოიყენება, როგორც სახელების მინიჭების, ასევე როგორც ხედვის არის მართვის მექანიზმი. პაკეტის შიგნით შესაძლებელია კლასების განსაზღვრა, რომლებიც ამ პაკეტის გარეთ არსებული კოდისთვის არ არის წვდომადი. ასევე, შესაძლებელია კლასის წევრების განსაზღვრა, რომლებიც ხილვადია მხოლოდ იმავე პაკეტის წევრებისთვის. აღნიშნული მექანიზმი კლასებს საშუალებას აძლევს ერთმანეთის შესახებ ფლობდნენ სრულ ინფორმაციას, მაგრამ დანარჩენი „გარე სამყაროსთვის“ ეს ინფორმაცია არ იყოს ცნობილი.

პაკეტის შექმნა მარტივი ამოცანაა. ამისათვის საკმარისია Java-ს საწყის ფაილში პირველ ოპერატორად **package** ბრძანების ჩასმა. ნებისმიერი კლასი, რომელიც გაცხადებული იქნება ამ ფაილში, აღნიშნულ პაკეტს მიეკუთვნება. package ოპერატორი განსაზღვრავს სახელების სივრცეს, რომელშიც კლასები ინახება. თუ package ოპერატორი არ გამოიყენება, მაშინ კლასების სახელები თავსდება სტანდარტულ პაკეტში (სახელის გარეშე). მცირე ზომის პროგრამებისთვის სტანდარტული პაკეტის გამოყენება სავსებით დასაშვები და მისაღებია, მაგრამ რეალური პროგრამული უზრუნველყოფის შემუშავებისთვის არ გამოდგება. უმეტეს შემთხვევაში, კოდისთვის პაკეტის განსაზღვრა საჭირო გახდება.

package ოპერატორს შემდეგი ზოგადი ფორმა გააჩნია:

```
package პაკეტი;
```

აქ **პაკეტის** ქვეშ მისი სახელი იგულისხმება. მაგალითად, ქვემოთ წარმოდგენილი ოპერატორი ქმნის MyPackage სახელის მქონე პაკეტს:

```
package MyPackage;
```

პაკეტების შენახვის მიზნით Java ფაილური სისტემის კატალოგებს იყენებს. ყურადღება უნდა მივაქციოთ იმ ფაქტს, რომ ამ დროს დიდი მნიშვნელობა ენიჭება სიმბოლოების რეგისტრს, ხოლო კატალოგის სახელი ზუსტად უნდა ემთხვეოდეს პაკეტის სახელს. ერთი და იგივე package ოპერატორი შესაძლებელია რამდენიმე ფაილში არსებობდეს. აღნიშნული ოპერატორი უბრალოდ იმ პაკეტზე მიუთითებს, რომელსაც მოცემულ ფაილში განსაზღვრული კლასები მიეკუთვნება. რეალურ პროგრამებში გამოყენებული პაკეტების უმრავლესობა მრავალ ფაილშია განაწილებული.

Java ენა უფლებას გვაძლევს შევქმნათ პაკეტების იერარქია. ამ მიზანს წერტილის ოპერატორი ემსახურება. მრავალდონიანი პაკეტის ოპერატორის ჩაწერის ზოგადი ფორმა შემდეგია:

```
package პაკეტი1[.პაკეტი2 [.პაკეტი3]] ;
```

პაკეტების იერარქია წარმოდგენილი უნდა იყოს Java-ს დამუშავების ფაილურ სისტემაში. მაგალითად, Windows გარემოში პაკეტი, რომელიც გამოცხადებულია **package java.awt.image** სახით, საჭიროა ინახებოდეს **java\awt\image** კატალოგში. პაკეტის სახელი არ შეიძლება შევცვალოთ, თუ არ შევცვლით იმ კატალოგის სახელს, რომელშიც კლასები ინახება.

მაგალითი 1. შევადგინოთ პროგრამა ერთი მარტივი პაკეტის გამოყენებით, რომლის სახელია Paketebi და ამ სახელის მქონე კატალოგში მოვათავსოთ ფაილი MyRes.java. შემდეგ კი ამოვხსნათ $ax + b = 0$ წრფივი განტოლება a და b კოეფიციენტების ნებისმიერი მნიშვნელობების დროს, რაც შემდეგი გარემოებების გათვალისწინებას გულისხმობს:

- თუ a=0, მაშინ x=-b/a;
- თუ a=0 და b=0, ამონახსნთა უსასრულო სიმრავლე გვაქვს;
- თუ a=0 და b≠0, ამონახსნი არ გვაქვს.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package paketebi;
2  import java.util.Scanner;
3  //წრფივი განტოლების ამოხსნა
4  class Resursebi{
5      double a, b;
6      Resursebi(double a, double b){
7          this.a=a;
8          this.b=b;
9      }
10     void gamotvla()
11     {
12         if(a!=0){
13             double x=-b/a;
14             System.out.println("x=" + x);
15         }
16         else if (b==0) System.out.println("ნებისმიერი ამონახსნი გვაკმაყოფილებს.");
17         else System.out.println("ამონახსნი არ გვაქვს.");
18     }
19 }
20 public class MyRes {
21  public static void main(String args []){
22      Scanner ob1=new Scanner(System.in);
23      double a, b;
24      System.out.print("a=");
25      a=ob1.nextDouble();
26      System.out.print("b=");
27      b=ob1.nextDouble();
28      Resursebi ob2=new Resursebi(a, b);
29      ob2.gamotvla();
30  }
31 }
```

ნახ. 151

შედეგი:

```
a=0
b=7
ამონახსნი არ გვაქვს.
```

ნახ. 152

როგორც ვხედავთ, MyRes კლასი Paketebi პაკეტის ნაწილია, რაც ნიშნავს, რომ მისი დამოუკიდებლად შესრულება არ შეიძლება.

3.15.2. წვდომის დაცვა

ჩვენ უკვე განვიხილეთ Java-ში არსებული წვდომის მართვის მექანიზმის სხვადასხვა ასპექტი და მისი მოდიფიკატორები. მაგალითად, ვიცით, რომ კლასის დახურულ წევრზე წვდომა შესაძლებელია მხოლოდ ამავე კლასის სხვა წევრებისთვის. პაკეტები წვდომის მართვას კიდევ ერთ განზომილებას უმატებენ.

კლასები და პაკეტები ერთდროულად წარმოადგენენ ინკაფსულაციისა და სახელების სივრცეების, ცვლადებისა და მეთოდების ხედვის არეების შენახვის საშუალებებს. პაკეტები ასრულებენ კლასების და სხვა დაქვემდებარებული პაკეტების კონტეინერების როლს. კლასები მონაცემებისა და კოდის კონტეინერებია. კლასი Java-ს აბსტრაქციის უმცირესი ერთეულია. კლასებისა და პაკეტების ურთიერთდამოკიდებულებიდან გამომდინარე, Java განსაზღვრავს კლასის წევრების ხილვადობის ოთხ კატეგორიას.

- ქვეკლასები ერთ პაკეტში;
- კლასები ერთ პაკეტში, რომლებიც არ წარმოადგენენ ქვეკლასებს;
- ქვეკლასები სხვადასხვა პაკეტებში;
- კლასები, რომლებიც არ არიან განთავსებული ერთ პაკეტში და არც ქვეკლასებს წარმოადგენენ.

წვდომის სამი მოდიფიკატორი: public, private, protected წარმოგვიდგენენ მრავალფეროვან საშუალებებს წვდომის მრავალი დონის შესაქმნელად, რაც აღნიშნული კატეგორიებისთვის აუცილებელია. მათ შორის ურთიერთკავშირი ქვემოთ ნაჩვენებ მე-15-ე ცხრილშია წარმოდგენილი.

ერთი შეხედვით, წვდომის მართვის მექანიზმი შესაძლოა რთულად მოგვეჩვენოს, მაგრამ, მიგვაჩნია, რომ შემდეგი მოსაზრებები მის აღქმაში ხელს შეგიწყობთ. ნებისმიერი კომპონენტი გამოცხადებული როგორც public, წვდომადია ნებისმიერი კოდიდან. ნებისმიერი კომპონენტი გამოცხადებული როგორც private, არ არის ხილვადი კომპონენტებისთვის, რომლებიც განთავსებულია ამ კომპონენტის კლასის გარეთ. თუ წევრს ცხადი სახით არ აქვს მითითებული წვდომის მოდიფიკატორი, ის ხილვადია ქვეკლასებისთვის და მოცემულ პაკეტში არსებული სხვა კლასებისთვის. წვდომის ეს დონე სტანდარტულად (მიუთითებლობის შემთხვევაში) გამოიყენება. თუ საჭიროა, რომ ელემენტი ხილვადი იყოს მისი მიმდინარე პაკეტის გარეთ, მაგრამ კლასებისთვის, რომლებიც მოცემული კლასის უშუალო ქვეკლასებს წარმოადგენენ, საჭიროა ელემენტი protected მოდიფიკატორის გამოყენებით გაცხადდეს.

	private	მოდიფიკატორის გარეშე	protected	public
ერთიდა იგივე კლასი	კი	კი	კი	კი
იმავე პაკეტის კლასის ქვეკლასი	არა	კი	კი	კი
ამავე პაკეტის კლასი, რომელიც ქვეკლასს არ წარმოაგდენს	არა	კი	კი	კი
სხვა პაკეტის კლასის ქვეკლასი	არა	არა	კი	კი
სხვა პაკეტის კლასი, რომელიც არ წარმოადგენს მოცემული პაკეტის კლასის ქვეკლასს	არა	არა	არა	კი

მე-15-ე ცხრილში წარმოდგენილი წვდომის წესები მხოლოდ კლასის წევრებზე გამოიყენება. კლასისთვის, რომელიც არ წარმოადგენს ჩადგმულ კლასს, შეიძლება მიეთითოს მხოლოდ წვდომის ორი შესაძლო დონიდან ერთ-ერთი: სტანდარტული (მიუთითებლობის შემთხვევაში) და public. როდესაც კლასი გამოცხადებულია, როგორც public, ის წვდომადია ნებისმიერი სხვა კოდისთვის. თუ კლასს წვდომის სტანდარტული დონე აქვს (განსაზღვრული მიუთითებლობის შემთხვევაში), ის წვდომადია მხოლოდ მოცემული პაკეტის შიგნით არსებული კოდისთვის. როდესაც კლასი ღიაა, ის ერთადერთ ღია კლასს უნდა წარმოადგენდეს, რომელიც ფაილშია გამოცხადებული და ფაილის სახელი კლასის სახელს უნდა ემთხვეოდეს.

3.15.3. წვდომის დაცვის მაგალითი

ახლა განვიხილოთ **მაგალითი 2**, რომელიც წვდომის მართვის მოდიფიკატორთა ყველა კომბინაციას წარმოგვიდგენს. ის შეიცავს ორ პაკეტს და ხუთ კლასს. საჭიროა გვახსოვდეს, რომ ორი სხვადასხვა პაკეტის კლასები უნდა ინახებოდეს კატალოგებში, რომელთა სახელები შესაბამისი პაკეტების სახელებს უნდა ემთხვეოდეს. ჩვენს შემთხვევაში – p1 და p2.

პირველი პაკეტის საწყისი ფაილი განსაზღვრავს სამ კლასს: Protection, Derived და SamePackage. პირველი კლასი განსაზღვრავს int ტიპის ოთხ ცვლადს. ამათგან, n ცვლადი სტანდარტული დაცვითაა გამოცხადებული, n_pri - როგორც private, n_pub - როგორც public, n_pro - როგორც protected.

აღნიშნულ მაგალითში ყველა სხვა კლასი შეეცდება მიმართოს მოცემული კლასის ეგზემპლიარის ცვლადებს. პროგრამაში, ის სტრიქონები, სადაც წვდომა დაუშვებელია, კომენტარების სახითაა გაფორმებული.

Derived მეორე კლასი ამავე p1 პაკეტის Protection კლასის ქვეკლასია. ის Derived კლასს საშუალებას აძლევს წვდომა ქონდეს Protection კლასის ყველა ცვლადთან, გარდა n_pri ცვლადისა, რომელიც გამოცხადებულია, როგორც დახურული (private).

SamePackage მესამე კლასი არ წარმოადგენს Protection კლასის ქვეკლასს, მაგრამ ის იმავე პაკეტშია განთავსებული და წვდომა გააჩნია Protection კლასის ყველა ცვლადთან, გარდა n_pri ცვლადისა.

Protection.java ფაილის კოდი შემდეგია:

```
1  package P1;
2  public class Protection {
3  int n=1;
4  private int n_pri=2;
5  protected int n_pro=3;
6  public int n_pub=4;
7  public Protection(){
8      System.out.println("სუპერ კლასის კონსტრუქტორი.");
9      System.out.println("n=" + n);
10     System.out.println("n_pri=" + n_pri);
11     System.out.println("n_pro=" + n_pro);
12     System.out.println("n_pub=" + n_pub);
13 }
14 }
```

ნახ. 153

Derived.java ფაილის კოდი შემდეგია:

```
1  package P1;
2   class Derived extends Protection{
3       Derived(){
4           System.out.println("ქვეკლასის კონსტრუქტორი");
5           System.out.println("n=" + n);
6          //System.out.println("n_pri=" + n_pri);
7              System.out.println("n_pro=" + n_pro);
8              System.out.println("n_pub=" + n_pub);
9       }
10 }
```

ნახ. 154

SamePackage.java ფაილის კოდი შემდეგია:

```
package P1;
 class SamePackage {
     SamePackage(){
         Protection p=new Protection();
         System.out.println("ამავე კლასის კონსტრუქტორი");
         System.out.println("n=" + p.n);
         //class only
         //System.out.println("n_pri=" + p.n_pri);
                System.out.println("n_pro=" + p.n_pro);
                System.out.println("n_pub=" + p.n_pub);
     }
}
```

ნახ. 155

ქვემოთ წარმოდგენილია მეორე (p2) პაკეტის საწყისი კოდი. მასში ორი კლასია განსაზღვრული. Protection2, რომელიც Protection კლასის ქვეკლასია (p1 პაკეტიდან) და OtherPackage კლასი, რომელსაც წვდომა მხოლოდ n_pub ცვლადთან აქვს, რადგან იგი გამოცხადებულია, როგორც public. Protection2 კლასს კი წვდომა გააჩნია Protection კლასის ყველა ცვლადთან, გარდა n_pri და n ცვლადებისა. ეს უკანასკნელი (n ცვლადი) სტანდარტული დაცვით იყო გამოცხადებული, რომელზე წვდომა გააჩნია მხოლოდ მოცემული კლასს ან მოცემულ პაკეტს და არ აქვთ წვდომა ქვეკლასებს სხვა პაკეტებიდან.

Protection2.java ფაილის კოდი შემდეგია:

```
package P2;
class Protection2 extends P1.Protection{
    Protection2(){
        System.out.println("მეორე პაკეტის მემკვიდრე კონსტრუქტორი.");
        //System.out.println("n=" + n);
        //System.out.println("n_pri=" + n_pri);
        System.out.println("n_pro=" + n_pro);
        System.out.println("n_pub=" + n_pub);
    }
}
```

ნახ. 156

OtherPackage.java ფაილის კოდი შემდეგია:

```
package P2;
class OtherPackage {
    OtherPackage(){
        P1.Protection p=new P1.Protection();
        System.out.println("მეორე პაკეტის კონსტრუქტორი.");
        //System.out.println("n=" + p.n);
        //System.out.println("n_pri=" + p.n_pri);
        //System.out.println("n_pro=" + p.n_pro);
        System.out.println("n_pub=" + p.n_pub);
    }
}
```

ნახ. 157

ზემოთ წარმოდგენილი ორი პაკეტის მუშაობის შესამოწმებლად გამოვიყენოთ ორი ფაილი. პირველი პაკეტის შემმოწმებელი ფაილის კოდი შემდეგია:

```
1  package P1;
2  public class Demo {
3      public static void main(String [] args){
4          Protection ob1=new Protection();
5          Derived ob2=new Derived();
6          SamePackage ob3=new SamePackage();
7      }
8  }
```

ნახ. 158

მეორე პაკეტის შემმოწმებელი ფაილის კოდი შემდეგია:

```
1  package P2;
2  public class Demo {
3      public static void main(String [] args){
4          OtherPackage ob1=new OtherPackage();
5          Protection2 ob2=new Protection2();
6          }
7       }
```

ნახ. 159

პირველი პაკეტის შესრულების შედეგი:

```
ხაბაზო კლასის კონსტრუქტორი.
n=1
n_pri=2
n_pro=3
n_pub=4
ხაბაზო კლასის კონსტრუქტორი.
n=1
n_pri=2
n_pro=3
n_pub=4
ქვეკლასის კონსტრუქტორი
n=1
n_pro=3
n_pub=4
ხაბაზო კლასის კონსტრუქტორი.
n=1
n_pri=2
n_pro=3
n_pub=4
ამავე კლასის კონსტრუქტორი.
n=1
n_pro=3
n_pub=4
```

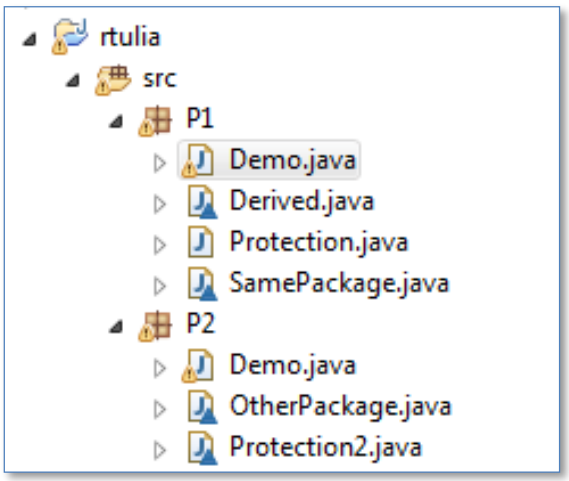
ნახ. 160

მეორე პაკეტის შესრულების შედეგი:

```
საბაზო კლასის კონსტრუქტორი.
n=1
n_pri=2
n_pro=3
n_pub=4
მეორე პაკეტის მემკვიდრე კონსტრუქტორი.
n_pro=3
n_pub=4
საბაზო კლასის კონსტრუქტორი.
n=1
n_pri=2
n_pro=3
n_pub=4
მეორე პაკეტის კონსტრუქტორი.
n_pub=4
```

ნახ. 161

ჩვენ მიერ შექმნილი და ზემოთ განხილული პაკეტებისა და მათში შემავალი კლასების ზოგადი სტრუქტურა წარმოდგენილია ნახ. 165-ზე.



ნახ. 162

3.15.4. პაკეტების იმპორტირება

პაკეტები სხვადასხვა კლასების ერთმანეთისგან იზოლაციის ეფექტურ მექანიზმს გვთავაზობენ. სწორედ ამიტომ, Java-ში ყველა ჩადგმული კლასი პაკეტებში ინახება. არცერთი ძირითადი კლასი Java-ში მიუთითებლობის შემთხვევაში არ ინახება სტანდარტულ პაკეტში.

ყველა სტანდარტული კლასი გარკვეული სახელის მქონე პაკეტშია განთავსებული. რადგან პაკეტებში არსებული კლასები სრულფასოვნად უნდა იყოს განსაზღვრული მათი პაკეტების სახელებით, ამიტომ პაკეტის გზის ამსახველი წერტილებით გამოყოფილი გრძელი სახელი ყოველი კლასისთვის, საკმაოდ დიდი შეიძლება აღმოჩნდეს. განსაზღვრული კლასები ან თუნდაც მთელი პაკეტი, ხილვადი რომ გავხადოთ, Java-ში გამოყენებულია **import** ოპერატორი. კლასის იმპორტირების შემდეგ, მასზე მიმართვა უშუალოდ კლასის სახელის მითითებით შეიძლება განვახორციელოთ. import ოპერატორი მხოლოდ პროგრამისტების მოსახერხებელ მუშაობას ემსახურება და ტექნიკური თვალსაზრისით, დასრულებული Java პროგრამის შესაქმნელად მისი გამოყენება აუცილებელი არ არის. მაგრამ, თუ პროგრამულ უზრუნველყოფაში გვიწევს რამდენიმე ათეულობით კლასზე მიმართვა, import ოპერატორი მნიშვნელოვნად ამცირებს შესატანი კოდის მოცულობას.

Java პროგრამის საწყის ფაილში import ოპერატორები უშუალოდ package (თუ ასეთი არსებობს) ოპერატორებს უნდა მოყვებოდეს და, კლასების ნებისმიერ განსაზღვრას წინ უნდა უსწრებდეს. import ოპერატორის ჩაწერის ზოგადი ფორმა შემდეგია:

import პაკეტი1 [.პაკეტი2]. (კლასის_სახელი | *);

აღნიშნულ ფორმაში **პაკეტი1** ზედა დონის პაკეტის სახელია, **პაკეტი2** - გარე პაკეტის შიგნით დაქვემდებარებული პაკეტის სახელი, რომელიც „წერტილის“ (.) სიმბოლოთია გამოყოფილი. პაკეტების ჩადგმის (ჩალაგების) სიღრმე პრაქტიკულად შეუზღუდავია, გარდა ფაილური სისტემისა და ბოლოს, **კლასის_სახელი** შეგვიძლია ჩავწეროთ ცხადად ან გამოვიყენოთ „ვარსკვლავის“ (*) სიმბოლო, რომელიც Java კომპილატორს მთელი პაკეტის იმპორტირების აუცილებლობაზე მიუთითებს. ქვემოთ წარმოდგენილი ჩანაწერი აღნიშნული ოპერატორის ორივე ფორმის დემონსტრირებას ახდენს:

```
import java.util.Date;
import java.io.*;
```

Java სისტემის ყველა სტანდარტული კლასი Java პაკეტში ინახება. ენის ძირითადი ფუნქციები Java პაკეტში არსებულ Java.lang პაკეტშია მოთავსებული. როგორც წესი, ჩვენ საჭირო კლასებისა და პაკეტების იმპორტირებას ვახდენთ. მაგრამ, რადგან Java სისტემა უსარგებლოა იმ მრავალი ფუნქციის გარეშე, რაც Java.lang პაკეტშია განსაზღვრული, ამიტომ კომპილატორი არაცხადი სახით ახდენს ყველა პროგრამაში მის იმპორტირებას. ეს შემდეგი სტრიქონის გამოყენების ეკვივალენტურია ყოველ პროგრამაში:

```
import java.lang.*;
```

პაკეტების სრული იერარქიის მითითებით კლასის სრულად განსაზღვრული სახელი შეგვიძლია გამოვიყენოთ ყველგან, სადაც კლასის სახელი დასაშვებია. მაგალითად, კოდის შემდეგ ფრაგმენტში იმპორტის ოპერატორია გამოყენებული:

```
import java.util.*;
class MyDate extends Date{
}
```

import ოპერატორის გარეშე იგივე მაგალითი შემდეგნაირად გამოიყურება:

```
class MyDate extends java.util.Date{
}
```

ბოლო ვერსიაში **Date** ობიექტი სრულადაა განსაზღვრული.

პაკეტის იმპორტირების დროს, შესაბამის კოდში (სადაც პაკეტის იმპორტი ხორციელდება) არსებული კლასებისთვის, რომლებიც იმპორტირებული პაკეტის კლასების ქვეკლასებს არ წარმოადგენენ, წვდომადია იმპორტირებული პაკეტის მხოლოდ public მოდიფიკატორით გაცხადებული ელემენტები.

ზემოთ თქმულის საილუსტრაციოდ განვიხილოთ შემდეგი **მაგალითი 12.3**, რომელიც პაკეტის იმპორტირების დემონსტრირებას ახდენს. თავად პროგრამაში მომხმარებლის სახელი და ბალანსის ოდენობა კონსოლზე გამოდის ნებისმიერ შემთხვევაში, ხოლო თუ ბალანსის მნიშვნელობა 0-ზე ნაკლებია, კონსოლზე ასევე იბეჭდება ისრის (--->) გამოსახულება.

პროგრამის კომპიუტერული რეალიზება (პაკეტის შექმნა):

```
1  package mypack;
2
3  public class Balan {
4          String name;
5          double bal;
6          public Balan(String n, double b){
7              name=n; bal=b;
8          }
9          public void show(){
10             if(bal<0)
11                 System.out.print("--->");
12             System.out.println(name + "$" + bal);
13         }
14     }
```

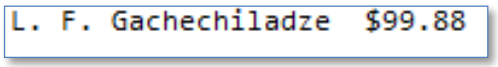
ნახ. 163

პროგრამის კომპიუტერული რეალიზება (პაკეტის იმპორტირება):

```
1  package dfsdf;
2  import mypack.Balan;
3  public class TestBalance {
4  public static void main(String [] args){
5      Balan ob1=new Balan("L. F. Gachechiladze  ", 99.88);
6      ob1.show();
7  }}
```

ნახ. 164

შედეგი:



ნახ. 165

3.15.5. ინტერფეისები

interface საკვანძო სიტყვის გამოყენება საშუალებას გვაძლევს კლასის ინტერფეისი სრულად გამოვყოთ მისი რეალიზაციისგან. სხვა სიტყვებით რომ ვთქვათ, **interface** საკვანძო სიტყვის გამოყენებით შეგვიძლია მივუთითოთ ის მოქმედებები, რაც კლასმა უნდა შეასრულოს, მაგრამ არა ის, თუ როგორ შეასრულოს ეს მოქმედებები. სინტაქსური თვალსაზრისით, ინტერფეისები კლასების ანალოგიურია, მაგრამ ისინი არ შეიცავენ ეგზემპლიარის ცვლადებს, ხოლო მათი მეთოდების გამოცხადება არ მოიცავს მეთოდების ტანს. პრაქტიკაში ეს ნიშნავს, რომ ინტერფეისები ისე შეგვიძლია გამოვაცხადოთ, რომ მათი რეალიზაცია არ იქნეს მითითებული. ინტერფეისის განსაზღვრის შემდეგ, მისი რეალიზება კლასების ნებისმიერ რაოდენობას შეუძლია. ამას გარდა, ერთ კლასს ინტერფეისების ნებისმიერი რაოდენობის რეალიზება შეუძლია.

ინტერფეისის რეალიზებისთვის კლასმა უნდა შექმნას ინტერფეისის მიერ განსაზღვრული მეთოდების სრული ნაკრები. **interface** საკვანძო სიტყვა საშუალებას გვაძლევს სრულად გამოვიყენოთ პოლიმორფიზმის კონცეფცია: „ერთი ინტერფეისი, მრავალი მეთოდი“. რადგან ინტერფეისების იერარქია არ ემთხვევა კლასების იერარქიას, ამიტომ კლასებს, რომლებიც კლასების იერარქიაში ერთმანეთთან დაკავშირებული არ არიან, შეუძლიათ ერთი და იმავე ინტერფეისის რეალიზება.

ინტერფეისის განსაზღვრა გარკვეულწილად კლასის განსაზღვრის მსგავსია. ინტერფეისის გამარტივებულ ფორმას შემდეგი სახე აქვს:

```
წვდომის მოდიფიკატორი interface სახელი{
დასაბრუნებელი შედეგის_ ტიპი მეთოდი1_სახელი(პარამეტრების_სია);
დასაბრუნებელი შედეგის_ ტიპი მეთოდი2_სახელი(პარამეტრების_სია);
ტიპი ცვლადი1_სახელი=მნიშვნელობა;
ტიპი ცვლადი2_სახელი=მნიშვნელობა;
//...
დასაბრუნებელი შედეგის_ ტიპი მეთოდიN_სახელი(პარამეტრების_სია);
ტიპი ცვლადიN_სახელი=მნიშვნელობა;}
```

თუ ინტერფეისის განსაზღვრა წვდომის არანაირ მოდიფიკატორს არ შეიცავს, მაშინ სტანდარტული წვდომა გამოიყენება და ინტერფეისი წვდომადი ხდება მხოლოდ იმ პაკეტის სხვა წევრებისთვის, რომელშიც თავად ინტერფეისია განსაზღვრული. თუ ინტერფეისი გამოცხადებულია, როგორც public, მაშინ მისი გამოყენება ნებისმიერ სხვა კოდს შეუძლია. ასეთ შემთხვევაში, ეს უნდა იყოს ფაილში გამოცხადებული ერთადერთი ღია ინტერფეისი და ფაილის სახელი უნდა ემთხვეოდეს ინტერფეისის სახელს.

ინტერფეისის ზემოაღნიშნულ განსაზღვრაში **სახელი**–ს ქვეშ ინტერფეისის სახელი იგულისხმება. ის ნებისმიერი დასაშვები იდენტიფიკატორია. ყურადღება მიაქციეთ იმ ფაქტს, რომ გამოცხადებული მეთოდები ტანს არ მოიცავენ. მათი გამოცხადება სრულდება პარამეტრების სიით, რომელსაც „წერტილ–მძიმის“ სიმბოლო მოსდევს. ერთი შეხედვით, ეს აბსტრაქტული მეთოდებია. კლასი, რომელიც ახდენს ინტერფეისის რეალიზებას, ამავდროულად, ინტერფეისში გამოცხადებული ყველა მეთოდის რეალიზებას უნდა ახორციელებდეს.

ინტერფეისის გამოცხადება ცვლადების გამოცხადებასაც მოიცავს. ეს ცვლადები არაცხადი სახით განიხილება, როგორც final და static, ანუ კლასი, რომელიც ინტერფეისის რეალიზებას ახდენს, ამ ცვლადების მნიშვნელობებს ვერ ცვლის. ამას გარდა, ცვლადები ინიციალებული უნდა იყოს. ყველა მეთოდი და ცვლადი არაცხადი სახით განიხილება, როგორც public.

ქვემოთ ნაჩვენებია ინტერფეისის განსაზღვრის მაგალითი. მასში მარტივი ინტერფეისია გამოცხადებული, რომელიც მოიცავს მხოლოდ ერთ callback() მეთოდს ერთი მთელრიცხვა პარამეტრით.

```
interface Callback{
void callback(int param);
}
```

ინტერფეისებთან დაკავშირებული ვიდეო მასალა შეგიძლიათ იხილოთ ბმულზე: https://www.youtube.com/watch?v=WOUpjal8ee4&list=PLGwb7xZHg-oMv1pOlTHAqAEjw0EPALzlL ვიდეო-ფაილი 62.

3.15.6. ინტერფეისების რეალიზება

როგორც კი ინტერფეისი განისაზღვრება, შესაძლებელია მისი რეალიზება ერთი ან რამდენიმე კლასის მიერ. ინტერფეისის რეალიზაციის მიზნით კლასის განსაზღვრაში საჭიროა ჩავრთოთ **implements** კონსტრუქცია, ხოლო შემდეგ შევქმნათ ინტერფეისის მიერ განსაზღვრული მეთოდები. კლასის ჩაწერის ზოგადი ფორმა, რომელიც **implements** კონსტრუქციას მოიცავს შემდეგია:

წვდომის მოდიფიკატორი class კლასის_სახელი[extends სუპერკლასი]

[implements ინტერფეისი [, ინტერფეისი...]]{

//კლასის ტანი

}

თუ კლასი ერთზე მეტი ინტერფეისის რეალიზებას ახდენს, მაშინ ინტერფეისების სახელები ერთმანეთისგან მძიმეებით გამოიყოფა. როდესაც კლასი ორი ინტერფეისის რეალიზებას ახორციელებს, რომლებშიც ერთი და იგივე მეთოდია გამოცხადებული, მაშინ ეს მეთოდი ნებისმიერი ინტერფეისის კლიენტების მიერ იქნება გამოყენებული. ამავდროულად, მეთოდები, რომლებიც ინტერფეისის რეალიზებას ახდენენ, გამოცხადებულ უნდა იქნეს, როგორც public. ამასთან, მეთოდების სიგნატურები კლასსა და ინტერფეისში სრულად უნდა შეესაბამებოდეს ერთმანეთს.

განვიხილოთ კლასი, რომელიც ზემოთ წარმოდგენილი Callback ინტერფეისის რეალიზებას ახდენს.

```
class Client implements Callback{
public void callback(int p){
System.out.println("callback მეთოდი გამოძახებულია" + p + "მნიშვნელობით");
}}
```

მიაქციეთ ყურადღება იმ ფაქტს, რომ callback() მეთოდი გამოცხადებულია წვდომის public მოდიფიკატორის გამოყენებით. გახსოვდეთ, რომ ინტერფეისის მეთოდის რეალიზების დროს, მეთოდი უნდა გამოცხადდეს, როგორც public.

სავსებით დასაშვებია და საკმარისად გავრცელებულია სიტუაცია, როდესაც კლასები, რომლებიც ახდენენ ინტერფეისის რეალიზებას, ხშირად განსაზღვრავენ საკუთარ დამატებით წევრებს. მაგალითად, Client კლასის შემდეგი ვერსია ახდენს callback() მეთოდის რეალიზებას და ამატებს nonInfaceMeth() მეთოდს.

```
class Client implements Callback{
public void callback(int p){
System.out.println("callback მეთოდი გამოძახებულია" + p + "მნიშვნელობით");
}
void nonInfaceMath(){
System.out.println("ინტერფეისის მარეალიზებელ კლასებს" +
                                „შეუძლიათ სხვა წევრების განსაზღვრაც"); }}
```

3.15.7. რეალიზაციებზე წვდომა ინტერფეისებზე მიმართვის გზით

ცვლადები შეიძლება გამოვაცხადოთ, როგორც ობიექტებზე მიმართვები, რომლებიც იყენებენ ინტერფეისის და არა კლასის ტიპს. ასეთი ცვლადის საშუალებით მიმართვა შეგვიძლია განვახორციელოთ ნებისმიერი კლასის ნებისმიერ ეგზემპლიარზე, რომელიც ინტერფეისის რეალიზებას ახორციელებს. ასეთი მიმართვის გამოყენებით, მეთოდის გამოძახებისას საჭირო ვერსიის შერჩევა მოხდება ინტერფეისის იმ კონკრეტულ ეგზემპლიარზე დამოკიდებულებით, რომელზეც მიმართვა ხორციელდება. ამაში გამოიხატება ინტერფეისების განსაკუთრებულობა. საჭირო მეთოდის ძებნა დინამიურად, პროგრამის შესრულების დროს ხორციელდება, რაც საშუალებას გვაძლევს კლასები იმ კოდზე გვიან შევქმნათ, რომელიც მეთოდების გამოძახებას ახდენს.

ქვემოთ წარმოდგენილ მაგალითში callback() მეთოდი ინტერფეისის მიმართვითი ცვლადის საშუალებით გამოიძახება.

```
class TestIface{
public static void main(String args []){
Callback c=new Client();
c.callback(42);
}}
```

პროგრამის შესრულების შედეგი:

callback მეთოდი გამოძახებულია 42 მნიშვნელობით.

ყურადღება მიაქციეთ იმ ფაქტს, რომ მართალია c ცვადი გამოცხადებულ იქნა Callback ინტერფეისის ტიპით, მაგრამ მას მიენიჭა Client კლასის ეგზემპლიარი. c ცვადი ცხადია, შეგვიძლია გამოვიყენოთ callback() მეთოდზე წვდომისათვის, მაგრამ Client კლასის სხვა წევრებზე წვდომა მას არ გააჩნია. ინტერფეისის მიმართვითი ცვლადი ინფორმაციას ფლობს მხოლოდ იმ მეთოდებზე, რომლებიც მხოლოდ მისი interface გამოცხადების დროსაა აღწერილი. ამგვარად, c ცვლადს ვერ გამოვიყენებთ nonInfaceMeth() მეთოდზე მიმართვისათვის, რადგან ის გამოცხადებულია Client კლასის და არა Callback კლასის მიერ.

ზემოთ წარმოდგენილი პროგრამის სრულ ვერსიას შემდეგი სახე აქვს:

```
package testiface;
public interface Callback {
    void callback(int param);
}
class Client implements Callback{
    public void callback(int p){
      System.out.println("callback მეთოდი გამოძახებულია " + p + "მნიშვნელობით.");
    }
    void nonInfaceMath(){
     System.out.println("ინტერფეისის მარეალიზებელ კლასებს" + "შეუძლიათ სხვა ფუნქციის განსაზღვრაც.");
    }
}
```

ნახ. 165

```
package testiface;

public class TestIface {
    public static void main(String args []){
        Callback c=new Client();
        c.callback(42);
    }

}
```

ნახ. 166

შედეგი:

```
callback მეთოდი გამოძახებულია 42 მნიშვნელობით
```

ნახ. 167

თუმცა ზემოთ წარმოდგენილი მაგალითი ფორმალურად გვიჩვენებს, თუ ინტერფეისის მიმართვით ცვლადს როგორ შეუძლია რეალიზაციის ობიექტზე მიიღოს წვდომა, მაგრამ ის არ ახდენს ამ მიმართვის პოლიმორფული შესაძლებლობების დემონსტრირებას. ზემოთ თქმულის საილუსტრაციოდ განვიხილოთ Callback ინტერფეისის მეორე რეალიზაცია.

class AnotherClient implements Callback{

public void callback(int p){

System.out.println(“callback მეთოდის კიდევ ერთი ვერსია“);

System.out.println(“p კვადრატში=“ + (p*p));

}}

ახლა კი კლასის მუშაობა შევამოწმოთ და წარმოვადგინოთ წინა პროგრამის კიდევ ერთი ვერსია:

```
package testiface;
public interface Callback {
    void callback(int param);
}
class Client implements Callback{
    public void callback(int p){
      System.out.println("callback მეთოდი გამოძახებულია " + p + "მნიშვნელობით.");
    }
    void nonInfaceMath(){
     System.out.println("ინტერფეისის მარეალიზებელ კლასებს" + "შეუძლიათ სხვა რევრების განსაზღვრაც.");
    }}
    class AnotherClient implements Callback{
        public void callback(int p){
        System.out.println("callback მეთოდის კიდევ ერთი ვერსია");
        System.out.println("p კვადრატში=" + (p*p));
        }}
```

ნახ. 168

```
package testiface;

public class TestIface {
    public static void main(String args []){
        Callback c=new Client();
        AnotherClient ob= new AnotherClient();
        c.callback(42);
        c=ob;
        c.callback(42);
    }

}
```

ნახ. 169

შედეგი:

```
callback მეთოდი გამოძახებულია 42 მნიშვნელობით
callback მეთოდის კიდევ ერთი ვერსია
p კვადრატში=1764
```

ნახ. 170

როგორც ხედავთ, callback() მეთოდის გამოძახების ეს ვერსია განისაზღვრება იმ ობიექტის ტიპით, რომელსაც c ცვლადი შესრულების დროს მიმართავს.

თუ კლასი მოიცავს ინტერფეისს, რომლის ყველა მეთოდის რეალიზებას ის არ ახდენს, მაშინ კლასი უნდა გამოცხადდეს, როგორც აბსტრაქტული (abstract).

abstract class Incomplete implements Callback{

int a, b;

void show(){

System.out.println(a + “ “ + b);

}

// …}

წარმოდგენილ ფრაგმენტში Incomplete კლასი არ ახდენს callback() მეთოდის რეალიზებას. ამიტომ, ის გამოცხადებულია როგორც აბსტრაქტული კლასი. შესაბამისად, ნებისმიერი კლასი, რომელიც მემკვიდრეობით იღებს Incomplete კლასს, უნდა ახდენდეს callback() მეთოდის რეალიზებას, ან უნდა გამოცხადდეს, როგორც აბსტრაქტული.

<u>დავალება:</u>

1. *შეადგინეთ პროექტი, რომელიც მოიცავს ორ პაკეტს, სახელებით Pack1 და Pack2. პირველ პაკეტში განათავსეთ ორი კლასი. პირველ კლასში გამოთვალეთ კვადრატის პერიმეტრი, ხოლო მეორეში - მართკუთხედის პერიმეტრი. მეორე პაკეტშიც ორი კლასი განათავსეთ. პირველ კლასში გამოთვალეთ კვადრატის ფართობი, ხოლო მეორეში - მართკუთხედის ფართობი. ორივე პაკეტის მუშაობა შეამოწმეთ ორი ფაილის საშუალებით.*
2. *შეადგინეთ პროგრამა, რომელიც მოიცავს ინტერფეისს სახელწოდებით Face. ამ უკანასკნელმა შესაბამისი მეთოდის საშუალებით კონსოლზე უნდა გამოიტანოს n რაოდენობის რიცხვებიდან ყველა რიცხვი, რომელთა პირველი და ბოლო ციფრები ერთმანეთის ტოლია. ინტერფეისის რეალიზება მოახდინეთ Student სახელის მქონე კლასის გამოყენებით.*

3.16. Java ენის კლასები

3.16.1. String კლასი

String კლასი Java–ს კლასების ბიბლიოთეკის ყველაზე ხშირად გამოყენებადი კლასია. ამის მიზეზი ის გახლავთ, რომ სტრიქონები დაპროგრამების უმნიშვნელოვანესი ელემენტია.

პირველ რიგში, საჭიროა გავაცნობიეროთ, რომ ნებისმიერი სტრიქონი String კლასის ობიექტს წარმოადგენს. სტრიქონული კონსტანტებიც კი ამავე კლასის ობიექტებია. მაგალითად, ოპერატორში:

System.out.println("ესეც String კლასის ობიექტია");

სტრიქონი – **„ესეც String კლასის ობიექტია"** – რეალურად წარმოადგენს String კლასის ობიექტს.

მეორეს მხრივ, String კლასის ობიექტები უცვლელია, რაც იმას გულისხმობს, რომ აღნიშნული კლასის ობიექტის შექმნის შემდეგ, მისი შიგთავსის ცვლილება შეუძლებელია. ყოველივე ზემოთ თქმული შეიძლება სერიოზულ შეზღუდვად მოგვეჩვენოს, მაგრამ ორი მიზეზიდან გამომდინარე, რეალურად სხვა სურათი იქმნება:

- თუ სტრიქონის შეცვლაა საჭირო, ყოველთვის შეიძლება ახალი სტრიქონის შექმნა, რომელიც ცვლილებებს მოიცავს;
- Java-ში განსაზღვრულია StringBuffer კლასი, რომელიც String კლასის თანასწორია და ის საშუალებას იძლევა შევცვალოთ სტრიქონები, რაც თავის მხრივ, უფლებას გვაძლევს ყველა ჩვეულებრივი მანიპულაცია განვახორციელოთ სტრიქონებზე.

სტრიქონების შექმნის მრავალი გზა არსებობს. მათ შორის ყველაზე მარტივია შემდეგი სახის ოპერატორის გამოყენება:

String myString="ტექსტური სტრიქონი";

String კლასის ობიექტის შექმნისთანავე მისი გამოყენება ყველა სიტუაციაშია შესაძლებელი, რომელშიც სტრიქონებთან მუშაობაა დასაშვები. მაგალითად, შემდეგი ოპერატორი myString ობიექტის შიგთავსს წარმოგვიდგენს:

System.out.println(myString);

String კლასის ობიექტებისთვის Java-ში განსაზღვრულია „+" ოპერატორი, რომელიც ორი სტრიქონის გაერთიანებას ასრულებს. მაგალითად, შემდეგი ოპერატორი:

String myString="მე" + „მომწონს" + „Java.";

myString ცვლადს ანიჭებს შემდეგი სტრიქონის მნიშვნელობას: „მე მომწონს Java".

განვიხილოთ, ზემოთ წარმოდგენილი კონცეფციების ამსახველი **მაგალითი 1**, რომელიც ორი სტრიქონის გაერთიანებას ახდენს და კონსოლზე საწყისი და მიღებული სტრიქონები გამოაქვს.

პროგრამის კომპიუტერული რეალიზაცია:

```java
//ორი სტრიქონის გაერთიანება
public class Student {
    public static void main(String args[]){
        String ob1="მე ქართველი ვარ ";
        String ob2="და მაშასადამე, ვარ ევროპელი";
        String ob3=ob1+ob2;
        System.out.println(ob1);
        System.out.println(ob2);
        System.out.println(ob3);
    }

}
```

ნახ. 171

შედეგი:

```
მე ქართველი ვარ
და მაშასადამე, ვარ ევროპელი
მე ქართველი ვარ და მაშასადამე, ვარ ევროპელი
```

ნახ. 172

String კლასი რამდენიმე მეთოდს მოიცავს. გავეცნოთ ზოგიერთ მათგანს. **equals()** მეთოდით შესაძლებელია ორი სტრიქონის ტოლობაზე შემოწმება. **length()** მეთოდი სტრიქონის სიგრძეს განსაზღვრავს. **charAt()** მეთოდით მითითებული ინდექსით სიმბოლოს მიღებაა შესაძლებელი. ქვემოთ წარმოდგენილია ამ მეთოდების ჩაწერის ზოგადი ფორმები:

boolean aquals(მეორე სტრიქონი);

int length();

char charAt(ინდექსი);

მაგალითი 2. შევადგინოთ პროგრამა, რომელიც ახდენს სტრიქონების ტოლობაზე შემოწმებას, მათი სიგრძის განსაზღვრას და მითითებული ინდექსით სტრიქონიდან შესაბამისი სიმბოლოს ამოღებას.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  //ოპერაციები სტრიქონებზე
2  public class Student {
3      public static void main(String args[]){
4          String ob1="ჩვენ ვსწავლობთ Java დაპროგრამების ენას ";
5          String ob2="დაპროგრამება საინტერესოა";
6          String ob3=ob1;
7          System.out.println("პირველი სტრიქონია: " + ob1);
8          System.out.println("მეორე სტრიქონია: " + ob2);
9          System.out.println("მესამე სტრიქონია: " + ob3);
10         System.out.println("პირველი სტრიქონის სიგრძე: " + ob1.length());
11         System.out.println("მეორე სტრიქონის სიგრძე: " + ob2.length());
12         System.out.println("მე-7 ინდექსის მქონე სიმბოლო მეორე სტრიქონში: " + ob2.charAt(7));
13         if(ob1.equals(ob2))
14             System.out.println("პირველი და მეორე სტრიქონები ერთნაირია.");
15         else System.out.println("პირველი და მეორე სტრიქონები განსხვავებულია.");
16         if(ob1.equals(ob3))
17             System.out.println("პირველი და მესამე სტრიქონები ერთნაირია.");
18         else System.out.println("პირველი და მესამე სტრიქონები განსხვავებულია.");
19     }}
```

ნახ. 173

შედეგი:

```
პირველი სტრიქონია: ჩვენ ვსწავლობთ Java დაპროგრამების ენას
მეორე სტრიქონია: დაპროგრამება საინტერესოა
მესამე სტრიქონია: ჩვენ ვსწავლობთ Java დაპროგრამების ენას
პირველი სტრიქონის სიგრძე: 39
მეორე სტრიქონის სიგრძე: 24
მე-7 ინდექსის მქონე სიმბოლო მეორე სტრიქონში: ა
პირველი და მეორე სტრიქონები განსხვავებულია.
პირველი და მესამე სტრიქონები ერთნაირია.
```

ნახ. 174

3.16.2. Math კლასი

Math კლასი შეიცავს ყველა იმ მცოცავმძიმიანი არგუმენტების შემცველ ფუნქციას, რომელიც გეომეტრიასა და ტრიგონომეტრიაში გამოიყენება. მე-16-ე ცხრილში წარმოდგენილია ეს ფუნქციები.

ცხრილი 16 Math კლასის ფუნქციები

მეთოდი	აღწერა
static double sin(double arg)	აბრუნებს arg კუთხის სინუსს (არგუმენტი რადიანებში მოიცემა)
static double cos(double arg)	აბრუნებს arg კუთხის კოსინუსს (არგუმენტი რადიანებში მოიცემა)

static double tan(double arg)	აბრუნებს arg კუთხის ტანგენსს (არგუმენტი რადიანებში მოიცემა)
static double asin(double arg)	აბრუნებს კუთხეს (რადიანებში), რომლის სინუსი arg პარამეტრით არის მითითებული
static double acos(double arg)	აბრუნებს კუთხეს (რადიანებში), რომლის კოსინუსი arg პარამეტრით არის მითითებული
static double atan(double arg)	აბრუნებს კუთხეს (რადიანებში), რომლის ტანგენსი arg პარამეტრით არის მითითებული
static double atan2(double x, double y)	აბრუნებს კუთხეს (რადიანებში), რომლის ტანგენსია x / y
static double exp(double arg)	აბრუნებს e^{arg} ფუნქციის მნიშვნელობას (სადაც e=2.72 ნატურალური ლოგარითმის ფუძეა)
static double log(double arg)	აბრუნებს ln(arg) ნატურალური ლოგარითმის მნიშვნელობას
static double pow(double y, double x)	აბრუნებს y^x გამოსახულების მნიშვნელობას.
static double sqrt(double arg)	აბრუნებს კვადრატულ ფესვს arg-დან
static double abs(double arg)	აბრუნებს arg-ის აბსოლუტურ მნიშვნელობას
static double ceil(double arg)	აბრუნებს უმცირეს მთელ რიცხვს, რომელიც arg არგუმენტზე მეტია ან ტოლი
static double floor(double arg)	აბრუნებს უდიდეს მთელ რიცხვს, რომელიც arg არგუმენტზე ნაკლებია ან ტოლი
static double max(double x, double y)	აბრუნებს *x* და *y* შორის უდიდესს
static double min(double x, double y)	აბრუნებს *x* და *y* შორის უმცირესს
static double rint(double arg)	აბრუნებს arg-ის უახლოეს მთელ რიცხვს
static int round(float arg)	აბრუნებს arg-ის მნიშვნელობას დამრგვალებულს უახლოეს int (მთელ) რიცხვამდე
static long round(double arg)	აბრუნებს arg-ის მნიშვნელობას დამრგვალებულს უახლოეს long (მთელ) რიცხვამდე
static double random()	აბრუნებს ფსევდო-შემთხვევით რიცხვს 0-დან 1-მდე
static double toRadians(double angle)	გრადუსი რადიანებში გადაყავს
static double toDegrees(double angle)	რადიანები გრადუსში გადაყავს

მე-16-ე ცხრილში წარმოდგენილ ყველა მათემატიკურ ფუნქციას (მეთოდს) წინ უსწრებს კლასის სახელი Math, რომელსაც მოყვება წერტილის ოპერაცია და შემდგომ მეთოდის სახელი. ეს ბუნებრივიცაა, რადგან ზემოთ წარმოდგენილი ყველა მეთოდი სტატიკურია.

მაგალითი 3. შევადგინოთ პროგრამა, რომელიც გამოთვლის შემდეგი გამოსახულებების მნიშვნელობებს: $y = a^x + e^{x^2}$, $z = \sqrt{x^2 + a}$, $d = |ax - b| + sinx$.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  import java.util.Scanner;
2  //მათემატიკური მეთოდები
3  public class Student {
4      public static void main(String args[]){
5          double x, y, z, d, a, b;
6          Scanner ob1=new Scanner(System.in);
7          System.out.print("a=");
8          a=ob1.nextDouble();
9          System.out.print("b=");
10         b=ob1.nextDouble();
11         System.out.print("x=");
12         x=ob1.nextDouble();
13         y=Math.round(Math.pow(a,x)+Math.exp(Math.pow(x,2)));
14         z=Math.round(Math.sqrt(Math.pow(x, 2)+a));
15         d=Math.abs(a*x-b)+ Math.sin(x);
16         System.out.println("y=" + y);
17         System.out.println("z=" + z);
18         System.out.println("d=" + d);
19     }}
```

ნახ. 175

შედეგი:

```
a=1
b=2
x=3
y=8104.0
z=3.0
d=1.1411200080598671
```

ნახ. 176

3.16.3. Vector კლასი

java.util (უტილიტები) პაკეტი შეიცავს ინტერფეისებისა და კლასების დიდ არჩევანს, რომლებიც უზრუნველყოფენ ფუნქციონალური შესაძლებლობების დიდ დიაპაზონს. ძირითადი შესაძლებლობებია: ფსევდოშემთხვევითი რიცხვების გენერირება, თარიღითა და დროით მანიპულირება, გამონაკლისებზე დაკვირვება, ბიტურ ნაკრებებზე მანიპულირება, სტრიქონების სინტაქსური ანალიზი და სხვა. ეს ერთ-ერთი ყველაზე ხშირად გამოყენებული პაკეტია.

ზემოაღნიშნული პაკეტი შეიცავს დაპროგრამების თანამედროვე ტექნოლოგიის ერთ-ერთ შემადგენელ ნაწილს – **კოლექციას**, რომელიც რაიმე პრინციპით ობიექტების დაჯგუფებას წარმოადგენს. java-ში კოლექციის სტრუქტურის საშუალებით ხდება ობიექტთა ჯგუფების დამუშავების სტანდარტიზაცია. მონაცემთა ჯგუფების შესანახად და მანიპულირებისათვის java-ს ამ პაკეტში რეალიზებულია სპეციალური ინტერფეისი Enumeration და კლასები: Vector, Stack, Hashtable და სხვა.

Vector კლასი რეალიზაციას უკეთებს დინამიურ მასივს. მოცემული კლასის კონსტრუქტორებია:

```
Vector()
Vector(int size)
Vector(int size, int incr)
```

პირველი ფორმა ქმნის ვექტორს, რომელის საწყისი ზომა 10-ის ტოლია. მეორე ფორმა აყალიბებს ვექტორს, რომლის საწყისი ზომაა – size. მესამე ფორმა – ესაა ვექტორი, რომლის საწყისი ზომაა size, ხოლო შემდგომი ნაზრდი incr. ნაზრდი ელემენტების რაოდენობაა, რომელიც ვექტორს ყოველთვის დაემატება, როცა საჭიროა ვექტორის ზომის გაზრდა.

ყველა ვექტორი თავის არსებობას კონსტრუქტორით განსაზღვრული ზომით იწყებს. თუ ეს საწყისი ზომა ამოიწურა, მაგრამ კომპონენტის დამატება კვლავ გვსურს, ვექტორს ავტომატურად გამოეყოფა მეხსიერება ნაზრდის მნიშვნელობის შესაბამისად და კომპონენტის დამატება ამ არეში მოხდება. თუ ნაზრდის მნიშვნელობა მითითებული არ გვაქვს, მაშინ ვექტორის ზომა ორმაგდება.

Vector კლასში განსაზღვრულია დაცული (protected) მონაცემთა ველები (კომპონენტები):

```
protected int capacityIncrement;
protected int elementCount;
protected Object elementData[];
```

ნაზრდის მნიშვნელობა capacityIncrement ცვლადში ინახება. ვექტორში ელემენტების მიმდინარე რაოდენობა elementCount ცვლადში ინახება. ვექტორის კომპონენეტ-ობიექტების მასივი elementData[]-ში ინახება.

Vector კლასში განსაზღვრული მეთოდები მე-17-ე ცხრილშია წარმოდგენილი.

ცხრილი 17. Vector კლასის მეთოდები

მეთოდი	აღწერა
Final void addElement(Object element)	Element ობიექტი დაემატება ვექტორს
Final int capacity()	აბრუნებს ვექტორის ელემენტების რაოდენობას
Object clone()	აბრუნებს გამომძახებელი ვექტორის კლონს (დუბლიკატს)
Final boolean contains(Object element)	აბრუნებს true (ჭეშმარიტ) შედეგს, თუ element შედის ვექტორში, false - წინააღმდეგ შემთხვევაში
Final void copyInto(Object array[])	გამომძახებელი ვექტორის ელემენტები კოპირდება array მასივში
Final Object elementAt(int index)	აბრუნებს index პოზიციაში განლაგებულ ელემენტს (ობიექტს)
Final Enumeration elements()	აბრუნებს ვექტორის ელემენტების ჩამონათვალს
Final void ensureCapacity(int size)	აყენებს ვექტორის ზომას size-ის ტოლად
final Object firstElement()	აბრუნებს ვექტორის პირველ ელემენტს
Final int indexOf(Object element)	აბრუნებს გამომძახებელ ვექტორში element-ობიექტის პირველი განლაგების ინდექსს, თუ ელემენტი ვექტორში არ იმყოფება – ბრუნდება -1
Final int indexOf(Object element , int start)	აბრუნებს გამომძახებელ ვექტორში element-ობიექტის პირველი განლაგების ინდექსს, დაწყებული start ინდექსიდან. თუ ელემენტი ვექტორში არ იმყოფება – ბრუნდება -1
Final void insertElementAt(Object element, int index)	გამომძახებელ ვექტორში index პოზიციაში ჩაამატებს element-ს
final boolean isEmpty()	აბრუნებს true-ს, თუ ვექტორი ცარიელია და false-ს თუ ვექტორში ერთი ან მეტი ელემენტია
Final Object lastElement()	აბრუნებს ვექტორის ბოლო ელემენტს
Final int lastIndexOf(Object element)	აბრუნებს element-ობიექტის ბოლო განლაგების ინდექსს. თუ ელემენტი ვექტორში არ იმყოფება ბრუნდება -1

Final int lastIndexOf(Object element, int start)	აბრუნებს element-ობიექტის ბოლო განლაგების ინდექსს დაწყებული start ინდექსიდან. თუ ელემენტი ვექტორში არ იმყოფება ბრუნდება -1
Final void removeAllElements()	ვექტორიდან ამოაგდებს ყველა ელემენტს. ამის შემდეგ ვექტორის სიგრძე 0-ის ტოლი ხდება
Final boolean removeElement(Object element)	ვექტორიდან ამოაგდებს element-ობიექტს. თუ რამოდენიმე ასეთი ელემენტია, ამოვარდება პირველი. თუ ამოგდება წარმატებით დამთავრდა,ბრუნდება true, წინააღმდეგ შემთხვევაში - false
Final void removeElementAt(int index)	ამოაგდებს index პოზიციის ელემენტს
Final void setElementAt(Object element, int index)	Element-ობიექტი ვექტორში მოთავსდება index პოზიციაში
Final void setSize(int size)	ვექტორში ელემენტების რაოდენობა ხდება size-ის ტოლი. თუ ახალი ზომა ძველზე მცირეა, ელემენტები იკარგება. თუ ახალი ზომა მეტია, ემატება null-ელემენტები
Final int size()	ვექტორში აბრუნებს ელემენტების რაოდენობას
String toString()	აბრუნებს ვექტორის სტრიქონულ ექვივალენტს
Final void trimToSize()	აყენებს იმ ზომას, რამდენი ელემენტიცაა ახალ ვექტორში

მაგალითი 4. შევადგინოთ პროგრამა, რომელიც Vector კლასის მეთოდების დემონსტრირებას ახორციელებს. კერძოდ, განსაზღვრავს ვექტორის საწყის ზომას, მოცულობას, ამატებს მასში ელემენტებს, კონსოლზე გამოაქვს პირველი და ბოლო ელემენტების მნიშვნელობები და ბეჭდავს ვექტორს საბოლოო სახით.

დასმული ამოცანის ამოხსნის პროგრამული რეალიზაცია 177-ე ნახაზზეა წარმოდგენილი, ხოლო შედეგები -178-ე.

შედეგები:

```
საწყისი ზომა : 0
საწყისი მოცულობა : 3
მოცულობა 4-ის დამატების შემდეგ :  5
მიმდინარე მოცულობა :  5
მიმდინარე მოცულობა :  7
მიმდინარე მოცულობა :  9
პირველი ელემენტი :  1
ბოლო ელემენტი :  12
ვექტორი შეიცავს 3

ვექტორის ელემენტები :
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

ნახ. 177

პროგრამის კომპიუტერული რეალიზაცია:

```
1  import java.util.*;
2  public class VectorDemo {
3      public static void main(String arg[]){
4      Vector v = new Vector(3, 2);
5
6      System.out.println("საწყისი ზომა:" + v.size());
7      System.out.println("საწყისი მოცულობა:" + v.capacity());
8
9      v.addElement(new Integer(1));   //ვექტორში ელემენტების დამატება
10     v.addElement(new Integer(2));
11     v.addElement(new Integer(3));
12     v.addElement(new Integer(4));
13
14     System.out.println("მოცულობა 4-ის დამატების შემდეგ:  " + v.capacity());
15
16     v.addElement(new Double(5.45));
17     System.out.println("მიმდინარე მოცულობა:  " + v.capacity());
18
19     v.addElement(new Double(6.08));
20     v.addElement(new Integer(7));
21     System.out.println("მიმდინარე მოცულობა:  " + v.capacity());
22
23     v.addElement(new Float(9.4));
24     v.addElement(new Integer(10));
25     System.out.println("მიმდინარე მოცულობა:  " + v.capacity());
26
27     v.addElement(new Integer(11));
28     v.addElement(new Integer(12));
29     System.out.println("პირველი ელემენტი:  " +(Integer) v.firstElement());
30     System.out.println("ბოლო ელემენტი:  " +(Integer) v.lastElement());
31
32     if(v.contains(new Integer(3))) System.out.println("ვექტორი შეიცავს 3 "); //შეიცავს თუ არა ვექტორი 3-ს
33
34     Enumeration vEnum = v.elements(); //Enumeration ინტერფეისი განსაზღვრავს მეთოდებს
35     System.out.println("\nვექტორის ელემენტები:  " );
36     while(vEnum.hasMoreElements())System.out.print(vEnum.nextElement() + " ");
37
38 }
39 }
```

ნახ. 178

3.16.4. Stack კლასი

Stack კლასი Vector კლასის ქვეკლასია, რომელიც რეალიზაციას უკეთებს LIFO (Last – In, First – Out --- ბოლო შესული პირველი გამოდის) სტანდარტულ სტეკს.

ამ კლასისათვის მხოლოდ ერთი ცარიელი კონსტრუქტორია განსაზღვრული, რომელიც ცარიელ სტეკს ქმნის. Stack კლასი შეიცავს ყველა იმ კლასს, რომლებიც Vector კლასშია

განსაზღვრული და თავის მხრივ ამატებს მეთოდებს, რომლებიც მე-18 ცხრილშია წარმოდგენილი.

ცხრილი 18. Stack კლასის მეთოდები

მეთოდი	აღწერა
Boolean empty()	აბრუნებს true-ს, თუ სტეკი ცარიელია, false-ს თუ სტეკი ელემენტებს შეიცავს.
Object peek()	აბრუნებს სტეკის ზედა ელემენტს, მაგრამ სტეკიდან არ აგდებს.
Object pop()	ელემენტის ამოღება და დაბრუნება სტეკიდან.
Object push(Object element)	element ობიექტის სტეკში მოთავსება.
Int search(Object element)	element ობიექტს ეძებს სტეკში. პოვნის შემთხვევაში აბრუნებს მის ინდექსს სტეკის დასაწყისიდან (თავიდან). წინააღმდეგ შემთხვევაში აბრუნებს -1-ს.

მაგალითი 5. შევადგინოთ პროგრამა, რომელიც Stack კლასის მეთოდების დემონსტრირებას ახორციელებს. კერძოდ, სტეკში ათავსებს ელემენტებს, შემდგე სტეკს ათავისუფლებს და კონსოლზე გამოაქვს როგორც თითოეული მეთოდის შესრულების, ასევე საბოლოო შედეგები.

დასმული ამოცანის ამოხსნის პროგრამული რეალიზაცია 179-ე სურათზეა წარმოდგენილი, ხოლო შედეგები -180-ე სურათზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  import java.util.Stack;
2  import java.util.EmptyStackException;
3  public class StackDemo {
4      static void showpush(Stack st, int a)
5        {
6          st.push(new Integer(a));
7          System.out.println("push(" + a + ")");
8          System.out.println("stack: " + st);}
9        static void showpop(Stack st)
10       {
11         System.out.print("pop -> ");
12         Integer a = (Integer) st.pop();
13         System.out.println(a);
14         System.out.println("stack: " + st);}
15       public static void main(String args[])
16       {
17         Stack st = new Stack();
18         System.out.println("stack: " + st);
19         showpush(st, 55);
20         showpush(st, 77);
21         showpush(st, 99);
22         showpop(st);
23         showpop(st);
24         showpop(st);
25         try
26         {
27           showpop(st);
28         }
29         catch(EmptyStackException e)
30         {
31           System.out.println("empty stack");}}}
32
```

ნახ. 179

შედეგი:

```
stack: []
push(55)
stack: [55]
push(77)
stack: [55, 77]
push(99)
stack: [55, 77, 99]
pop -> 99
stack: [55, 77]
pop -> 77
stack: [55]
pop -> 55
stack: []
pop -> empty stack
```

ნახ. 180

3.16.5. Hashtable კლასი

Hashtable კლასს შეუძლია წყვილის გასაღები/მნიშვნელობის ჰეშ-ცხრილში შენახვა. Hashtable-ზე მიმართვისას ვუთითებთ ობიექტს, რომელიც გამოიყენება როგორც გასაღები, და მნიშვნელობა, რომელიც ამ გასაღებთან გვინდა დავაკავშიროთ. შემდეგ ხდება გასაღების

ჰეშირება და მიღებული ჰეშ-კოდი გამოიყენება როგორც ინდექსი (ელემენტის ნომერი) რომლითაც მნიშვნელობა ცხრილის შიგნით ინახება.

მოცემული კლასის კონსტრუქტორიებია:

```
Hashtable()
Hashtable(int size)
Hashtable(int size, float fillRatio)
```

პირველი ვერსია – ესაა სტანდარტული კონსტრუქტორი პარამეტრების გარეშე. მეორე – ქმნის ჰეშ-ცხრილს, რომელის საწყისი ზომაა size. მესამე – ქმნის size ზომის ჰეშ-ცხრილს და შევსების კოეფიციენტი არის fillRatio. ამ კოეფიციენტის მნიშვნელობა 0.0-დან 1.0-მდე რიცხვით დიაპაზონს მოიცავს. იგი მიუთითებს, რამდენად შეიძლება იყოს შევსებული ჰეშ-ცხრილი სანამ ის თავის ზომას გაზრდის. როდესაც ელემენტების რაოდენობა მეტია ვიდრე ჰეშ-ცხრილის ტევადობის ნამრავლი შევსების კოეფიციენტზე, ჰეშ-ცხრილი ფართოვდება. თუ შევსების კოეფიციენტი მითითებული არ არის, მაშინ გამოიყენება კოეფიციენტი 0.75. ამ კლასის მეთოდები მე-19 ცხრილშია წარმოდგენილი:

ცხრილი 19. Hashtable კლასის მეთოდები

მეთოდი	აღწერა
Void clear()	ასუფთავებს ჰეშ-ცხრილს
Object clone()	აბრუნებს გამომძახებლის კლონს
Boolean contains(Object value)	აბრუნებს true-ს (ჭეშმარიტ მნიშვნელობას), თუ ჰეშ-ცხრილში არსებობს value-ს ტოლი მნიშვნელობა, წინააღმდეგ შემთხვევაში აბრუნებს false-ს (მცდარ მნიშვნელობას)
Boolean containsKey(Object key)	აბრუნებს true-ს (ჭეშმარიტ მნიშვნელობას), თუ ჰეშ-ცხრილში არსებობს key-ს ტოლი გასაღები, წინააღმდეგ შემთხვევაში აბრუნებს false-ს (მცდარ მნიშვნელობას)
Boolean containsValue(Object value)	აბრუნებს true-ს (ჭეშმარიტ მნიშვნელობას), თუ ჰეშ-ცხრილში არსებობს value-ს ტოლი მნიშვნელობა, წინააღმდეგ შემთხვევაში აბრუნებს false-ს (მცდარ მნიშვნელობას)

Enumeration elements()	აბრუნებს ჰეშ-ცხრილის შემცველი ელემენტების ჩამონათვალს
Object get(Object key)	აბრუნებს ობიექტს, რომელიც key გასაღებთან არის დაკავშირებული. გასაღების არარსებობის შემთხვევაში აბრუნებს null-ს.
Boolean isEmpty()	აბრუნებს true-ს (ჭეშმარიტ მნიშვნელობას), თუ ჰეშ-ცხრილი ცარიელია, წინააღმდეგ შემთხვევაში აბრუნებს false-ს (მცდარ მნიშვნელობას)
Enumeration keys()	აბრუნებს ჰეშ-ცხრილში შემავალი გასაღებების ჩამონათვალს
Object put(Object key, Object value)	value მნიშვნელობას key გასაღებით მოათავსებს ჰეშ-ცხრილში. აბრუნებს null-ს, თუ ასეთი გასაღები ჰეშ-ცხრილში არ არსებობს. თუ ასეთი გასაღები ჰეშ-ცხრილში არსებობდა – აბრუნებს ამ გასაღებთან დაკავშირებულ ძველ მნიშვნელობას
Void rehash()	ზრდის ჰეშ-ცხრილის ზომას და ყველა გასაღებს გადაანაწილებს თავიდან
Object remove(Object key)	ამოაგდებს ჰეშ-ცხრილიდან key გასაღებსა და მის შესაბამის მნიშვნელობას. აბრუნებს გასაღებთან დაკავშირებულ მნიშვნელობას. ასეთი გასაღების არარსებობის შემთხვევაში - აბრუნებს null-ობიექტს
Int size()	აბრუნებს ჰეშ-ცხრილის ელემენტების რაოდენობას
String toString()	აბრუნებს ჰეშ-ცხრილის სტრიქონულ ეკვივალენტს

მაგალითი 6. შევადგინოთ პროგრამა, რომელიც Hashtable კლასის მეთოდების დემონსტრირებას ახორციელებს. კერძოდ, ავსებს ჰეშ-ცხრილს გასაღებებითა და მნიშვნელობებით, შემდგომ ახდენს მათ კონსოლზე გამოტანას, ერთ-ერთი მნიშვნელობის 1000-ით გაზრდას და ახალი შედეგის დაფიქსირებას.

პროგრამის კომპიუტერული რეალიზაცია:

```
package koleqciebi;
import java.util.*;
//ჩეკბრიღები
public class HTDemo {
    public static void main(String arg[])
    {

    Hashtable balance = new Hashtable(); //კლასის ობიექტის შექმნა
    Enumeration names;
    String str;
    double bal;

balance.put("John Doe", new Double(3434.34)); //ჩეკბრიღის შევსება გასაღებებითა და მნიშვნელობებით
balance.put("Tom Smith", new Double(123.22));
balance.put("Jane Baker", new Double(1378.00));
balance.put("Tod Hall", new Double(99.22));
balance.put("Ralph Smith", new Double(-19.34));

names = balance.keys();
while(names.hasMoreElements())
{
str = (String) names.nextElement();                    //გასაღებებისა და მნიშვნელობების ბეჭდვა
System.out.println(str + ": "+balance.get(str));
}
bal = ((Double)balance.get("John Doe")).doubleValue();
balance.put("John Doe", new Double(bal + 1000)); // ბალანსის 1000-ით გაზრდა
System.out.println("ახალი ბალანსი John Doe: " + balance.get("John Doe"));
}
}
```

ნახ. 181

შედეგი:

```
Tod Hall: 99.22
Ralph Smith: -19.34
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22
ახალი ბალანსი John Doe: 4434.34
```

ნახ. 182

3.16.6. StringTokenizer კლასი

ამა თუ იმ ტექსტის დასამუშავებლად ხშირად სტრიქონის სინტაქსური ანალიზია საჭირო. სინტაქსური ანალიზი (parsing) ნიშნავს ტექსტის დისკრეტულ ნაწილებად ანუ ლექსემებად (tokens) დაყოფას. StringTokenizer კლასს უწოდებენ ლექსიკურ ანალიზატორს ან სკანერს. იგი Enumeration ინტერფეისს უკეთებს რეალიზაციას.

StringTokenizer კლასის გამოსაყენებლად საჭიროა მივუთითოთ განსახილველი სტრიქონი და გამყოფების შემცველი სტრიქონი. გამყოფები – ეს სიმბოლოებია, რომლებიც ერთმანეთისაგან ყოფენ ლექსემებს. მაგალითად “, ; :” გამყოფებად აცხადებს მძიმეს, წერტილმძიმეს, ორწერტილს. სტანდარტულ გამყოფებად ითვლება: ჰარი (space), ტაბულაციის სიმბოლო, ახალი სტრიქონის დასაწყისის სიმბოლო (LF), ახალ სტრიქონზე გადასვლის სიმბოლო (CR).

StringTokenizer კლასის კონსტრუქტორებია:

StringTokenizer(String str)
StringTokenizer(String str, String delimiters)
StringTokenizer(String str, String delimiters, boolean delimAsToken)

ყველა ვერსიაში str – გასაანალიზებელი სტრიქონია. პირველ ვერსიაში გამყოფებად სტანდარტული გამყოფები გამოიყენება. მეორე და მესამე ვერსიებში ეს გამყოფები delimiters პარამეტრში უნდა იყოს მითითებული. მესამე ვერსიაში delimAsToken თუ ჭეშმარიტია (true-ს ტოლია), მაშინ გამყოფები ბრუნდება როგორც ლექსემები, წინააღმდეგ შემთვევაში - გამყოფები ლექსემებად არ ბრუნდება.

StringTokenizer კლასის მეთოდები მე-20 ცხრილშია წარმოდგენილი:

ცხრილი 20. StringTokenizer კლასის მეთოდები

მეთოდი	აღწერა
Int countTokens()	მიმდინარე გამყოფების საშუალებით მეთო-დი ლეკსემების რაოდენობას აბრუნებს
Boolean hasMoreElements()	აბრუნებს true-ს, თუ სტრიქონში ერთი ან მეტი ლექსემა დარჩა და აბრუნებს false-ს, თუ არცერთი არ დარჩა
Boolean hasMoreTokens()	აბრუნებს true-ს, თუ სტრიქონში ერთი ან მეტი ლექსემა დარჩა და აბრუნებს false-ს, თუ არცერთი არ დარჩა
Object nextElement()	აბრუნებს მომდევნო ლექსემას, როგორც Object-კლასის ობიექტს
String nextToken()	აბრუნებს მომდევნო ლექსემას, როგორც String - კლასის ობიექტს
String nextToken(String delimiters)	აბრუნებს მომდევნო ლექსემას, როგორც String - კლასის ობიექტს და გამყოფების სტრიქონად აყენებს delimiters

მაგალითი 7. შევადგინოთ პროგრამა, რომელიც StringTokenizer კლასის მეთოდების დემონსტრირებას ახორციელებს. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 183-ზე, ხოლო შესრულების შედეგი ნახ. 184-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
package koleqciebi;
import java.util.StringTokenizer;
public class STDemo {
static String in = "title=The Java Handbook:" + "author=Patrick Naughton:" +
                   "isbn=0-07-882199-1:" + "ean=9 780078 821998:" +
                   "email=naughton@starwave. corn";
          public static void main(String args[])
          {
            StringTokenizer st = new StringTokenizer(in, "=:");
            while(st.hasMoreTokens())
            {
              String key = st.nextToken();
              String val = st.nextToken();
              System.out.println(key + "\t" + val);
            }
          }
         }
```

ნახ. 183

შედეგი:

```
title   The Java Handbook
author  Patrick Naughton
isbn    0-07-882199-1
ean     9 780078 821998
email   naughton@starwave. corn
```

ნახ. 184

3.16.7. ArrayList კლასი

ArrayList კლასი დინამიურ მასივებს მოიცავს, რომელთა ზომა საჭიროების შემთხვევაში შეიძლება გაიზარდოს. მოცემული კლასის გამოცხადების სინტაქსი შემდეგია:

class ArrayList <E>

სადაც E პარამეტრი შესანახი ობიექტების ტიპზე მიუთითებს. სტანდარტულ მასივებს Java-ში ფიქსირებული სიგრძე აქვთ, რაც იმას ნიშნავს, რომ მასივის შექმნის შემდეგ მისი შეცვლა (გაზრდა ან შემცირება) შეუძლებელია. გამოდის, რომ ასეთ დროს წინასწარ უნდა ვიცოდეთ, თუ რა რაოდენობის ელემენტი გვჭირდება მასივში, რაც რიგ შემთხვევებში შეუძლებელია, რადგან პროგრამის შესრულებაზე გაშვებამდე ჩვენ ხშირად არც კი ვიცით რამდენად დიდი ზომის მასივი დაგვჭირდება. სწორედ მსგავსი სიტუაციის თავიდან ასაცილებლად (ან უფრო სწორად, გამოსასწორებლად) კოლექციის ინფრასტრუქტურაში ArrayList კლასია განსაზღვრული. მოცემული კლასის ობიექტი დინამიურად შეიძლება გაიზარდოს ან შემცირდეს. მასივები-სიები

(ArrayList) გარკვეული საწყისი ზომით იქმნება და როდესაც ეს ზომა არასაკმარისი ხდება, კოლექცია ავტომატურად იზრდება. ობიექტების წაშლის შემთხვევაში კი კოლექცია მცირდება.

ArrayList კლასს შემდეგი კონსტრუქტორები გააჩნია:

ArrayList()
ArrayList(Collection <? Extends E> c)
ArrayList(int ტევადობა)

პირველი კონსტრუქტორი ცარიელ მასივს ქმნის. მეორე წარმოგვიდგენს მასივს, რომელიც c კოლექციის ელემენტებით არის ინიციალებული. მესამე კონსტრუქტორი კი ქმნის მასივს, რომელსაც საწყისი ტევადობა გააჩნია. მასივში ელემენტების დამატების დროს მისი ტევადობა ავტომატურად იზრდება.

განვიხილოთ ArrayList კლასის სადემონსტრაციო **მაგალითი 8**, სადაც String კლასის ობიექტებისთვის იქმნება მასივი-სია, რომელსაც შემდგომ რამდენიმე სტრიქონი ემატება. შესაბამისად, ადგილი აქვს ახალი მასივი-სიის კონსოლზე გამოტანას, რის შემდეგაც იშლება გარკვეული ელემენტები და სახეშეცვლილი სია კვლავ კონსოლზე გამოდის.

ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 185-ზე, ხოლო შესრულების შედეგი ნახ. 186-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
package koleqciebi;
import java.util.*;
//arrayList კლასი
public class ArrayListDemo {
    public static void main(String args []){
    //მასივი-სიის შექმნა
    ArrayList <String> a1 = new ArrayList <String> ();
    System.out.println("a1 მასივის საწყისი ზომა : " + a1.size());
    //ელემენტების დამატება
    a1.add("A");
    a1.add("B");
    a1.add("C");
    a1.add("D");
    a1.add("F");
    a1.add(1, "A2"); //ინდექსით პირველი ელემენტის ადგილას A2-ის დამატება
    System.out.println("შევსების შემდეგ a1 მასივის ზომა: " + a1.size());
    //სიის ეკრანზე გამოტანა
    System.out.println("a1-მასივის შიგთავსი: " + a1);
    //ელემენტების ამოშლა
    a1.remove("F");// მნიშვნელობით ელემენტის წაშლა (იშლება F)
    a1.remove(2); //ინდექსით მეორე ელემენტის წაშლა (იშლება B)
    System.out.println("a1 მასივის ზომა ელემენტების ამოშლის შემდეგ: " + a1.size());
    System.out.println("a1-მასივის შიგთავსი: " + a1);
    }
}
```

ნახ. 185

შედეგი:

```
a1 მასივის საწყისი ზომა : 0
შევსების შემდეგ a1 მასივის ზომა: 6
a1-მასივის შიგთავსი: [A, A2, B, C, D, F]
a1 მასივის ზომა ელემენტების ამოშლის შემდეგ: 4
a1-მასივის შიგთავსი: [A, A2, C, D]
```

ნახ. 186

ArrayList კლასთან მუშაობის დროს ზოგჯერ საჭირო ხდება ჩვეულებრივი მასივის მიღება, რომელიც სიის ელემენტებს შეიცავს. ამის განხორციელება Collection ინტერფეისში განსაზღვრული ToArray() მეთოდის გამოძახებითაა შესაძლებელი. არსებობს ToArray() მეთოდის ორი ვერსია:

object[] toArray()
<T> T[] toArray (T მასივი [])

პირველი ვერსია object კლასის ობიექტების მასივს აბრუნებს, ხოლო მეორე - T ტიპის ელემენტებისგან შემდგარ მასივს. როგორც წესი, მეორე ფორმა უფრო მოსახერხებელია, რამეთუ, ის მასივის სწორი ტიპის დაბრუნებას ახდენს.

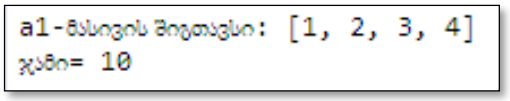
მაგალითი 9. წარმოვადგინოთ ArrayList კლასიდან ჩვეულებრივი მასივის მიღების ამსახველი პროგრამა სადაც მიღებულ მასივში წარმოებს ელემენტების ჯამის გამოთვლა. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 187-ზე, ხოლო შესრულების შედეგი ნახ. 188-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
package koleqciebi;
import java.util.*;
//ArrayList მასივში გარდასახვა
public class ArrayListtoArrayDemo {
    public static void main(String args[]){
        ArrayList <Integer> a1= new ArrayList <Integer>();
        //მასივ-სიაში ელემენტების დამატება
        a1.add(1);
        a1.add(2);
        a1.add(3);
        a1.add(4);
        System.out.println("a1-მასივის შიგთავსი: " + a1);
        //მასივის მიღება
        Integer a[]=new Integer[a1.size()];
        a=a1.toArray(a);
        int sum=0;
        //მასივი ელემენტების ჯამის გამოთვლა
        for(int i : a)
            sum+=i;
        System.out.println("ჯამი= " + sum);
    }
}
```

ნახ. 187

შედეგი:



ნახ. 188

კოლექციებთან დაკავშირებული ვიდეო მასალა შეგიძლიათ იხილოთ ბმულზე: https://www.youtube.com/watch?v=WOUpjal8ee4&list=PLsyeobzWxl7oZ-fxDYkOToURHhMuWD1BK ვიდეო-ფაილი 94.

3.16.8. TreeSet კლასი

TreeSet კლასი ქმნის კოლექციას, რომელიც ელემენტების შესანახად ხეს იყენებს. ობიექტები მასში ზრდადობის მიხედვით დახარისხებული ინახება. ელემენტებზე წვდომა აქ იმდენად სწრაფად მიმდინარეობს, რომ კლასი TreeSet საუკეთესო არჩევანს წარმოადგენს დიდი მოცულობის დახარისხებული ინფორმაციის შენახვისათვის.

TreeSet კლასის გამოცხადებას შემდეგი სახე აქვს:

class TreeSet <E>

სადაც E პარამეტრი იმ ობიექტების ტიპზე მიუთითებს, რომლებიც გროვაში უნდა იქნეს შენახული.

TreeSet კლასს შემდეგი კონსტრუქტორები აქვს:

```
TreeSet()
TreeSet(Collection<? Extends E> c)
TreeSet(Comparator<? super E> კომპარატორი)
TreeSet(SortedSet <E> ss)
```

ამათგან პირველი ფორმა ცარიელ გროვას (ხეს) ქმნის, რომელიც ელემენტებს ზრდადობის მიხედვით დაახარისხებს. მეორე ფორმა ნაკრებ-ხეს ქმნის, რომელიც c კოლექციის ელემენტებს შეიცავს. მესამე ფორმა ცარიელ გროვას ქმნის, რომელშიც ელემენტები კომპარატორის მიერ იქნება დახარისხებული. ეს უკანასკნელი კომპარატორი - პარამეტრში უნდა იყოს მითითებული. მეოთხე ფორმა ქმნის ნაკრებ-ხეს, რომელლიც ss-დან შეიცავს ელემენტებს.

მაგალითი 13.10. შევადგინოთ პროგრამა, რომელიც TreeSet კლასის გამოყენების დემონსტრირებას ახდენს. შედეგი 189-ე სურათზეა წარმოდგენილი, ხოლო პროგრამული რეალიზაცია 190-ე სურათზე.

შედეგი:

```
[A, B, C, D, E, F]
```

ნახ. 189

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package koleqciebi;
2  import java.util.*;
3  //ხეები
4  public class TreeSetDemo {
5      public static void main(String args []){
6          //TreeSet კლასის ობიექტის შექმნა
7          TreeSet<String> ts=new TreeSet<String>();
8          ts.add("D");
9          ts.add("A");
10         ts.add("C");
11         ts.add("B");
12         ts.add("F");
13         ts.add("E");
14         System.out.println(ts);
15     }
16
17 }
```

ნახ. 190

დავალება:

1. *შეადგინეთ პროგრამა, რომელიც მოახდენს ნებისმიერი ორი სტრიქონის კონკატენაციას (გაერთიანებას), განსაზღვრავს საწყისი და მიღებული სტრიქონების სიგრძეს და ყოველი სტრიქონიდან მითითებული ინდექსით ამოიღებს საჭირო სიმბოლოებს.*
2. *შეადგინეთ პროგრამა, რომელიც Math კლასის მეთოდების გამოყენებით ამოხსნის განტოლებათა შემდეგ სისტემას:* $y = \begin{cases} \sqrt{x^2 + 15}, & x < 0 \\ cosx + logx, & x \geq 0 \end{cases}$

   x პარამეტრის მნიშვნელობა პროგრამის შესრულებაზე გაშვების დროს შეიტანეთ.
3. *შეადგინეთ პროგრამა, რომელიც ვექტორს შეავსებს შვიდი ელემენტით. კონსოლზე გამოიტანს პირველი და ბოლო ელემენტის მნიშვნელობებს და ვექტორის ზომას.*
4. *შეადგინეთ პროგრამა, რომელიც სტეკს შეავსებს ათი ელემენტით. დაბეჭდავს საწყის სტეკს. შემდეგ გაასუფთავებს მას და შეამოწმებს, არის თუ არა სტეკი ცარიელი.*
5. *შეადგინეთ პროგრამა, რომელიც ჰეშ-ცხრილს შეავსებს String ტიპის გასაღებებითა და int ტიპის მნიშვნელობებით, კონსოლზე გამოიტანს საწყის ჰეშ-ცხრილს, შემდეგ ყოველ მნიშვნელობას გაზრის 5-ით და წარმოადგენს სახეშეცვლილ ჰეშ-ცხრილს.*
6. *შეადგინეთ პროგრამა, რომელიც ნებისმიერ სტრიქონს ლექსემებად დაყოფს.*
7. *შეადგინეთ პროგრამა, რომელიც ArrayList კლასიდან (რომელიც 10 ელემენტს მოიცავს) მიიღებს ჩვეულებრივ მასივს, დაბეჭდავს მის შიგთავსს და გამოთვლის მასივის ელემენტების საშუალო არითმეტიკულ მნიშვნელობას.*
8. *შეადგინეთ პროგრამა, რომლიც გროვას შეავსებს სტრიქონული ტიპის 12 მონაცემით და წარმოადგენს მათ ზრდადობით დახარისხებულს.*

3.17. ვიზუალური პროგრამის შექმნა

3.17.1. კლასი Applet

აპლეტი მცირე ზომის პროგრამაა, რომელიც ინტერნეტ-სერვერზე მდებაროებს, ავტომატურად ინსტალირდება და მისი შესრულებაზე გაშვება ვებ-დოკუმენტის ნაწილის სახით ხდება.

განვიხილოთ აპლეტის მარტივი **მაგალითი 1**:

```
import java.awt.*;
import java.util.applet.*;
public class SimpleApplet extends Applet{
public void paint(Graphics g){
g.drawString("Applet",20, 20);}}
```

ზემოთ წარმოდგენილი აპლეტი ორ import ოპერატორს მოიცავს, რომელთაგან პირველი ახდენს Abstract Window Toolkit (AWT) ბიბლიოთეკის კლასების იმპორტირებას, ხოლო მეორე - applet პაკეტის იმპორტირებას, რომელიც Applet კლასს შეიცავს. ყველა აპლეტი, რომელსაც ჩვენ შევქმნით, Applet კლასის ქვეკლასი უნდა იყოს. ჩვენ შემხთვევაში, ასეთ ქვეკლასს SimpleApplet კლასი წარმოადგენს. ამავე კლასში paint() მეთოდია აღწერილი, რომელიც AWT ბიბლიოთეკაშია განსაზღვრული და აპლეტის მიერ ხელახლა უნდა განისაზღვროს. paint() მეთოდი Graphics ტიპის ერთ პარამეტრს შეიცავს, რომელიც გრაფიკული კონტექსტის მომცველია და იმ გრაფიკულ გარემოს აღწერს, რომელშიც აპლეტი მუშაობს. paint() მეთოდში ადგილი აქვს drawString() მეთოდის გამოძახებას. ამ უკანასკნელს სტრიქონი გამოაქვს x, y კოორდინატებით მითითებულ პოზიციაში. მისი ჩაწერის ფორმაა:

```
drawString(String შეტყობინება, int x, int y)
```

აქ „შეტყობინება“ ის სტრიქონია, რომლის გამოტანა x, y პოზიციიდან იწყება. Java-ფანჯარაში ეკრანის მარცხენა ზედა კუთხის კოორდინატებია: 0, 0. ჩვენ შემთხვევაში, აპლეტში წარმოდგენილი სტრიქონი: "Applet" იწყება 20, 20 პოზიციიდან.

ყურადღება მიაქციეთ იმ ფაქტს, რომ აპლეტს არ გააჩნია main() მეთოდი. შესაბამისად აპლეტის შესრულებაზე გაშვება სხვაგვარად ხდება.

აპლეტის გაშვების ორი გზა არსებობს:

1. აპლეტის შესრულება Java-ს შესაბამისი ბრაუზერიდან;
2. აპლეტების დათვალიერების ისეთი საშუალების გამოყენება, როგორიცაა სტანდარტული ინსტრუმენტი appletwiever. ეს აპლეტის შემოწმების ყველაზე მარტივი გზაა.

ჩვენ მიერ შექმნილი აპლეტის გასაშვებად, საჭიროა შემდეგი ტექსტის კომენტარის სახით პროგრამის დასაწყისში ჩაწერა:

```
/*
<applet code="SimpleApplet" width=200, height=60>
```

```
</applet>
*/
```

width და height პარამეტრები იმ არის ზომებზე მიუთითებენ, რომელსაც აპლეტი გამოიყენებს.

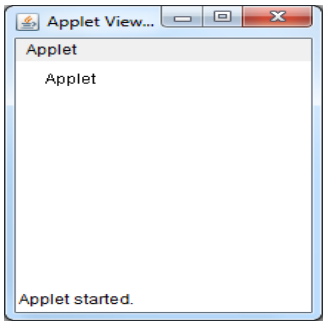
ახლა კი ზემოთ აღწერილი პროცედურების გამოყენებით, წარმოვადგინოთ აპლეტის სრულყოფილი კოდი, გავუშვათ ის Java ფაილიდან და ვნახოთ მიღებული შედეგი.

პროგრამის კომპიუტერული რეალიზაცია

```
1  package apletebi;
2  import java.awt.*;
3  import java.applet.*;
4  /*
5  <applet code="SampleApplet" width=200 height=60>
6  </applet>
7  */
8  public class SimpleApplet extends Applet{
9      public void paint(Graphics g){
10         g.drawString("Applet", 20, 20);
11     }
12 }
```

ნახ. 191

შედეგი:



ნახ.192

აპლეტის შაბლონი

ტრივიალური აპლეტების გარდა, ყველა აპლეტი მეთოდების ხელახალ განსაზღვრას აწარმოებს. მათ შორის ოთხი მეთოდი: init(), start(), stop() და destroy() ყველა აპლეტის მიმართ გამოიყენება. ისინი Applet კლასში განსაზღვრული მეთოდებია.

აპლეტის შაბლონის ზოგადი სახე შემდეგია:

```
import java.awt.*;
import java.util.applet.*;
```

```
/*
<applet code="AppletSkel" width=300, height=100>
</applet>
*/
public class AppletSkel extends Applet{
// init() მეთოდი პირველ რიგში გამოიძახება
public void init(){
//ინიციალიზაცია;
}
// init() მეთოდის შემდეგ start() მეთოდი გამოიძახება
public void start(){
//აპლეტის გაშვება;
}
//აპლეტის შეჩერების მიზნით გამოიძახება stop() მეთოდი
public void stop(){
//შესრულების შეჩერება
}
//აპლეტის განადგურებამდე ბოლო შესრულებადი მეთოდია destroy()
public void destroy(){
//ასრულებს დამამთავრებელ მოქმედებებს
}
//აპლეტის ფანჯრის აღსადგენად pain() მეთოდი გამოიძახება
public void paint(Graphics g){
//ფანჯრის შიგთავსის ხელახალი გადაწერა
}
```

აპლეტის ჩატვირთვისას თანმიმდევრობით გამოიძახება მეთოდები: init(), start(), paint(); ხოლო აპლეტის მუშაობის შეწყვეტის შემთხვევაში მეთოდები - stop() და destroy().

init() მეთოდში ადგილი აქვს ცვლადების ინიციალებას. აპლეტის შესრულების პროცესში ეს მეთოდი მხოლოდ ერთხელ გამოიძახება.

init() მეთოდის შემდეგ start() მეთოდი გამოიძახება. ეს უკანასკნელი ყოველთვის გამოიძახება, როდესაც აპლეტის შემცველი HTML დოკუმენტი ეკრანზე გამოისახება.

ყოველ ჯერზე, როდესაც აპლეტის გამოტანა ხელახლა უნდა დაიწეროს, paint() მეთოდი გამოიძახება.

stop() მეთოდი გამოიძახება მაშინ, როდესაც HTML დოკუმენტი, რომელიც აპლეტს შეიცავს, ტოვებს ვებ-ბრაუზერს.

როდესაც გარემო განსაზღვრავს, რომ აპლეტისგან კომპიუტერის ოპერატიული მეხსიერება უნდა გასუფთავდეს (აპლეტი წაიშალოს), destroy() მეთოდი გამოიძახება. აღნიშნული მეთოდის გამოძახებას ყოველთვის წინ უსწრებს stop() მეთოდის გამოძახება.

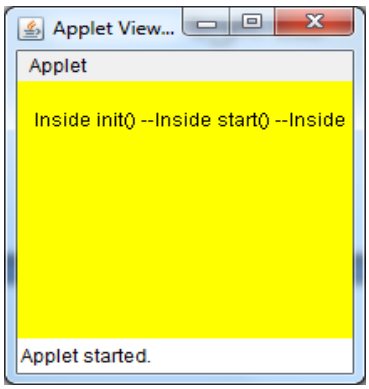
ახლა კი განვიხილოთ **მაგალითი 2**, სადაც იქმნება მარტივი აპლეტი, რომელზეც გარკვეული ფერის ფონია დართული და გამოიტანება სტრიქონი შეტყობინების სახით. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 193-ზე, ხოლო შესრულების შედეგი ნახ. 194-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package apletebi;
2  import java.awt.*;
3  import java.applet.*;
4  /*
5  <applet code="Sample" width=300 height=50>
6  </applet>
7  */
8  public class MyApplet extends Applet{
9      String msg;
10     public void init(){
11         setBackground(Color.blue);
12         setBackground(Color.yellow);
13         msg="Inside init() --";
14     }
15     public void start(){
16         msg+="Inside start() --";
17     }
18     public void paint(Graphics g){
19         msg+="Inside paint().";
20         g.drawString(msg, 10, 30);
21 }
22     }
```

ნახ. 193

შედეგი:



ნახ. 194

3.17.2. ხდომილების დამუშავება

გრაფიკული ინტერფეისის მქონე პროგრამებში ღილაკზე დაჭერა ზოგჯერ რაიმე მოქმედების ინიცირებას არ იწვევს, ზოგჯერ კი რაღაც მოქმედების წამოწყებას აქვს ადგილი. განვიხილოთ ხდომილებების დამუშავების მოდელი. მომხმარებელთან ურთიერთობისთვის ცხადია, ეკრანზე მხოლოდ კომპონენტების განლაგება საკმარისი არ არის. AWT კომპონენტების ნაკრების ხდომილებების დამუშავების მოდელში ყოველი კომპონენტი ერთ ან რამდენიმე დამკვირვებელთან ასოცირდება. როდესაც კომპონენტზე რაიმე ხდომილება წარმოიქმნება, ეს ფაქტი

ყველა დამკვირვებელს ეცნობება. დამკვირვებელი, ფაქტობრივად, ის ობიექტია, რომელიც ხდომილებით „დაინტერესებულია“. AWT კომპონენტების დამკვირვებლებს **მსმენელებს** (Listener) უწოდებენ. მათ უნდა განახორციელონ ცარიელი ინტერფეისის - java.util.EventListener - ის რეალიზება და თითქმის ყველა შემთხვევაში, მსმენელი - ქვეკლასის ინტერფეისი, რომელსაც ერთი მეთოდი მაინც უნდა ჰქონდეს. ყოველი ხდომილება java.util.EventObject კლასის ქვეკლასს წარმოადგენს. მნიშვნელოვანია განისაზღვროს, თუ კომპონენტის რომელ ხდომილებებთანაა ასოცირებული კონკრეტული მსმენელი. განვიხილოთ შემთხვევა, როდესაც ხდება Button ღილაკის არჩევა. ამ მოქმედებაზე პასუხისმგებელია ხდომილება, რომლის გენერირებაც მომხმარებლის მიერ Button ღილაკის არჩევისას ხდება. თუ რომელიმე ობიექტი ამ ხდომილებით არის დაინტერესებული, ის Button ღილაკთან უნდა დარეგისტრირდეს. ActionEvent ხდომილებისათვის რეგისტრაცია ActionListener-ის ფორმით ხდება. მათ შორის სახელები შე-თანხმებულია შემდეგნაირად: ყოველი ABCEvent ფორმის ხდომილებისთვის ასოცირებულ მსმენელად უნდა წარდგეს ABCListener მსმენელი, სადაც ABC-ის მაგივრად კონკრეტული ხდომილების ტიპი უნდა იქნეს მითითებული.

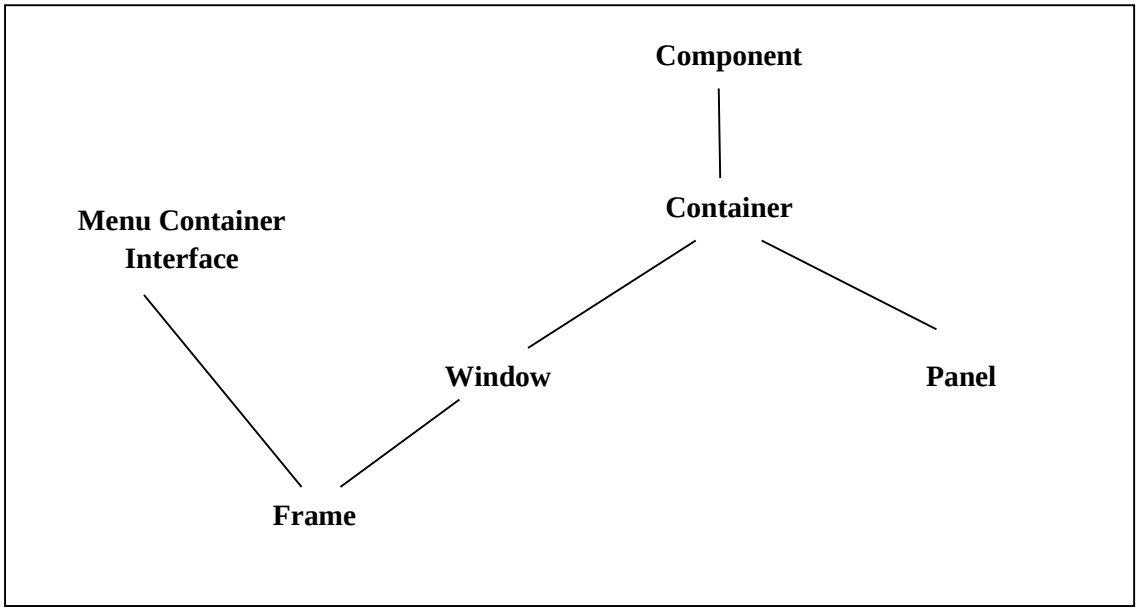
რეგისტრაცია addActionListener მეთოდის გამოძახებისას ხდება. ყოველ დარეგისტრირებულ რეალიზატორს ActionEvent ხდომილების გენერაციის შემთხვევაში ეცნობება ეს ფაქტი. ერთ ხდომილებას რამდენიმე დამკვირვებელი შეიძლება ჰყავდეს. მთელი ეს პროცესი შემდეგი მიმდევრობით შეგვიძლია წარმოვადგინოთ:

- მოცემული ხდომილებისთვის განისაზღვროს კლასი, რომელიც რეალიზაციას გაუკეთებს მასთან დაკავშირებულ ინტერფეისს. კლასის აღწერას უნდა დაემატოს „implements ABCListener” ან შეიქმნას ახალი კლასი, რომელიც ამ ინტერფეისის რეალიზებას განა-ხორციელებს.
- კლასის აღწერაში მხოლოდ „implements ABCListener”-ის დამატება საკმარისი არ არის. საჭიროა ABCListener ინტერფეისის ყველა მეთოდის რეალიზაცია. ზოგ მსმენელს (მაგალითად, ActionListener) აქვს მხოლოდ ერთი მეთოდი, ხოლო სხვებს (მაგალითად, WindowListener - ის ფანჯრის გადაადგილებისას ან დახურვისას გამოიყენება) რამდენიმე.
- ინტერფეისის რეალიზაციის განსაზღვრის შემდეგ უნდა შეიქმნას რეალიზაციის ეგზემპლიარი და მისი ასოცირება კომპონენტთან უნდა მოხდეს. მხოლოდ ამის შემდეგ შეეძლება მსმენელს მიიღოს შეტყობინება რაიმე ხდომილების წარმოქმნის შესახებ.

3.17.3. AWT ბიბლიოთეკა

AWT (Abstract Window Toolkit – აბსტრაქტული ფანჯარის ინსტრუმენტები) ბიბლიოთეკა მრავალრიცხოვან კლასებსა და მეთოდებს შეიცავს, რომელთა საშუალებით შესაძლებელია ფანჯრების შექმნა და მათი მართვა. AWT –ს კლასები მოთავსებულია java.awt პაკეტში. ეს პაკეტი ორგანიზებულია იერარქიულად, ამიტომ გასაგებად და გამოსაყენებლად ადვილია.

AWT განსაზღვრავს ფანჯრებს, რომლებსაც კლასთა იერარქიის მიხედვით, ყოველი დონე ამატებს ფუნქციონალურ შესაძლებლობებსა და სპეციფიკას. 195-ე სურათზე წარმოდგენილია ზოგიერთი ძირითადი კლასის (კერძოდ, Panel და Frame) იერარქია.



ნახ. 195 Panel და Frame კლასების იერარქია

Component კლასი AWT- იერარქიაში ყველაზე ზედა კლასია. ის აბსტრაქტულ კლასს წარმოადგენს და მასში ინკაფსულირებულია ვიზუალური კომპონენტების ყველა თვისება. მომხმარებელთან ინტერფეისისათვის საჭირო ყველა ელემენტი, რომელიც ეკრანზე გამოისახება და რომლებთანაც მომხმარებელი აწარმოებს ურთიერთობას, Component-ის ქვეკლასს წარმოად-გენს. ამ კლასში ასზე მეტი public (ღია წვდომის) მეთოდია, რომელთა დანიშნულებაა:

- კლავიატურიდან ან „მაუსიდან“ ინფორმაციის შეტანა;
- ფანჯრების პოზიციონირება ან ზომების ცვლილება;
- მოვლენების (ხდომილებების) მართვა;
- წინა პლანისა და ფონის მიმდინარე ფერების დამახსოვრება;
- არჩეული ტექსტური შრიფტის ტიპის, ფერის, ზომისა და შრიფტის სხვა პარამეტრების დამახსოვრება;

- ფანჯარაში გამოტანილი ელემენტების ხელახალი გადახატვა და სხვა.

Container კლასი Component-ის ქვეკლასს წარმოადგენს. ის დამატებით მეთოდებს შეიცავს, რომლებიც საშუალებას გვაძლევს მასში სხვა Component-ობიექტები ჩავდოთ. Container კლასის შიგნით შესაძლებელია თვით ამ კლასის ობიექტების მოთავსებაც. კონტეინერი მართავს ნების-მიერი მასში შემავალი კომპონენტის განლაგებასა და მის პოზიციონირებას. ამას ის სხვადასხვა კომპონირების (განლაგების) მენეჯერების საშუალებით ასრულებს.

Panel კლასი Container კლასის კონკრეტული ქვეკლასია. ის ახალ მეთოდებს არ ამატებს. ის Container კლასის რეალიზაციას წარმოადგენს. Panel კლასის ობიექტი – ეს არის ფანჯარა, რომელსაც არ აქვს: სათაურის არე, მენიუს სტრიქონი და გარე ბორდიური. Panel-ობიექტს სხვა კომპონენტები add() მეთოდის საშუალებით შეიძლება დავუმატოთ. მას შემდეგ რაც Panel-ობიექტს ელემენტები დაემატება setLocation(), setSize() და setBounds() მეთოდების საშუალებით, რომლებიც Component კლასშია განსაზღვრული, შეგვიძლია მათი პოზიციონირება, ზომების შეცვლა.

Window კლასი ზედა დონის ფანჯარას ქმნის. ასეთი ფანჯარა არ შედის არცერთ სხვა ობიექტში. ის უშუალოდ სამუშაო მაგიდაზე მდებარეობს. ჩვენ გამოვიყენებთ Window-კლასის ქვეკლასს - Frame კლასს.

Frame კლასში ინკაფსულირებულია ფანჯარა, რომელიც Window კლასის ქვეკლასს წარმოადგენს და რომელსაც აქვს სასათაურო სტრიქონი, მენიუს სტრიქონი, კუთხეები, ფანჯრის ზომები, რაც შეიძლება შეიცვალოს.

ყველაზე უფრო ხშირად ჩვენ Frame-ის შვილობილ ფანჯრის ტიპს გამოვიყენებთ. Frame – ესაა სტანდარტული სტილის ფანჯარა. მას კონსტრუქტორის ორი ფორმა აქვს:

```
Frame()
Frame(String <სათაური>)
```

პირველი ფორმა სტანდარტულ ფანჯარას ქმნის, რომელსაც სათაური არ გააჩნია, ხოლო მეორე ფორმით შექმნილი ფანჯრის სათაური იქნება <სათაური>-ში მითითებული ტექსტი. როგორც კონსტრუქტორიდან ჩანს, ფანჯრის საწყისი ზომის დაყენება არ შეიძლება, ამიტომ ფანჯრის შექმნის შემდეგ ფანჯრის ზომა ჩვენთვის საჭირო მნიშვნელობებით უნდა შევცვალოთ.

Frame-ფანჯრებთან სამუშაოდ სხვადასხვა მეთოდია განსაზღვრული. განვიხილოთ ეს მეთოდები:

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

ფანჯრის ახალი ზომის დაყენება ხდება newWidth (სიგანე) და newHeight (სიმაღლე) პარამეტრებით, ან width და height ველებით, რომლებიც Dimension კლასის ობიექტშია

განსაზღვრული და კონსტრუქტორს გადაეცემა newSize პარამეტრით. ზომები პიქსელებში უნდა იყოს მითითებული.

Dimension getSize() მეთოდი ფანჯრის მიმდინარე ზომის მისაღებად გამოიყენება. ის სიგანესა და სიმაღლეს აბრუნებს Dimension კლასის ობიექტის width და height ველებში.

void setVisible(boolean visibleFlag) მეთოდი. ფრეიმ-ფანჯრის შექმნის შემდეგ ის უჩინარი რჩება მანამ, სანამ setVisible() – მეთოდს არ გამოვიძახებთ. ფანჯარა ხილვადი ხდება, თუ ამ მეთოდის პარამეტრი მნიშვნელობად true-ს იღებს, წინააღმდეგ შემთხვევაში ფანჯარა არ ჩანს (უჩინარია).

void setTitle(String newTitle) მეთოდით შესაძლებელია ფანჯრის სათაურის შეცვლა. **newTitle** – ეს ფანჯრის ახალი სათაურია.

შესაძლებელია ფრეიმ-ფანჯრის შექმნა, უბრალოდ, Frame-ობიექტის შექმნით, მაგრამ ასეთი ფანჯრის შემთხვევაში არ შეგვეძლება მივიღოთ და დავამუშაოთ ხდომილებები, რომლებიც ფანჯრის შიგნით წარმოიშვება ან გამოვიტანოთ ფანჯარაში ინფორმაცია. ამიტომ, Frame კლასის ქვეკლასი უნდა შევქმნათ და ამ ქვეკლასში მოვახდენოთ Frame კლასში განსაზღვრული ხდომილებების დამმუშავებელი მეთოდების ხელახალი გამოცხადება (გადაფარვა).

Frame კლასს გადაეცემა თავისი სუპერკლასის ყველა შესაძლებლობა. ეს იმას ნიშნავს, რომ ფრეიმ-ფანჯრის მართვა ისევეა შესაძლებელი, როგორც ჩვეულებრივი მთავარი ფანჯრის. მაგალითად, შეიძლება გადავტვირთოთ paint() მეთოდი, რათა ფანჯარაში აისახოს გამოტანა, გადაიფაროს repaint() მეთოდი, რათა აღდგეს ფანჯარა, ასევე შესაძლებელია ყველა ხდომილების დამამუშავებლის გადაფარვა. ყოველთვის როდესაც ფანჯარაში რაიმე მოვლენა (ხდომილება) ხდება, ამ ხდომილებასთან დაკავშირებული დამამუშავებელი მეთოდი გამოიძახება.

აპლეტის ახალი ფრეიმ-ფანჯრის შექმნა საკმაოდ მარტივია. თავდაპირველად Frame კლასის ქვეკლასი იქმნება. შემდეგ ხელახლა განისაზღვრება აპლეტის ნებისმიერი სტანდარტული მეთოდი: როგორებიცაა: init(), start(), stop() იმისათვის, რომ საჭიროების შემთხვევაში გამოვაჩინოთ ან დავფაროთ ფრეიმ-ფანჯარა. ბოლოს WindowClosing() მეთოდის რეალიზებას აქვს ადგილი. Frame კლასის ქვეკლასის განსაზღვრისთანავე შესაძალებელია მისი ობიექტის შექმნა. ეს უკანასკნელი ფრეიმ-ფანჯრის შექმნას განაპირობებს, თუმცა ფანჯარა თავიდან ეკრანზე არ გამოჩნდება. setVisible() მეთოდის გამოყენებით ფანჯრის ეკრანზე ჩვენება შესაძლებელი გახდება. შექმნილ ფანჯარას სტანდარტული ზომები: სიმაღლე და სიგანე გააჩნია. ფანჯრის ზომები setSize() მეთოდის საშუალებთ შეგვიძლია დავაყენოთ.

მაგალითი 3. შევქმნათ Frame კლასის ქვეკლასი სახლწოდებით SampleFrame. აღნიშნული ფანჯრის კლასის ეგზემპლიარი AppletFrame კლასის init() მეთოდში იქმნება. SampleFrame კლასი Frame კლასის კონსტრუქტორს იძახებს, რაც საშუალებას გვაძლევს მივიღოთ სტანდარტული

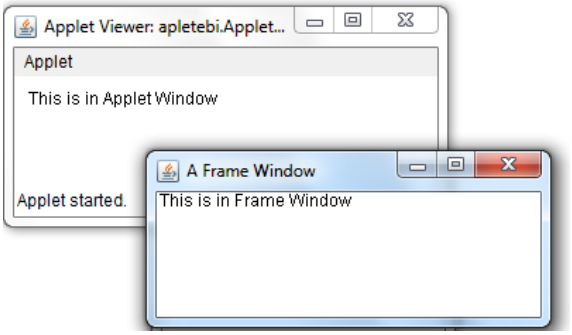
ფრეიმ-ფანჯარა სათაურით, რომელიც title პარამეტრშია გადაცემული. აღნიშნულ მაგალითში start() და stop() მეთოდები ხელახლა განისაზღვრება, რაც ფანჯრის გამოჩენას და დაფარვას უზრუნველყოფს.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package apletebi;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.applet.*;
5  /*
6  <applet code="AppletFrame" width=300 height=50>
7  </applet>
8  */
9   class SampleFrame extends Frame {
10     SampleFrame(String title){
11     super(title);
12     MyWindowAdapter adapter=new MyWindowAdapter(this);
13     addWindowListener(adapter);
14 }
15     public void paint(Graphics g){
16         g.drawString("This is in Frame Window", 10, 40);
17     }
18 }
19
20 class MyWindowAdapter extends WindowAdapter{
21     SampleFrame sampleFrame;
22 public MyWindowAdapter(SampleFrame sampleFrame){
23     this.sampleFrame=sampleFrame;
24 }
25 public void windowClosing(WindowEvent we){
26     sampleFrame.setVisible(false);
27 }
28 }
29 public class AppletFrame extends Applet{
30     Frame f;
31     public void init(){
32         f=new SampleFrame("A Frame Window");
33         f.setSize(250, 250);
34         f.setVisible(true);
35     }
36     public void start(){
37         f.setVisible(true);
38     }
39     public void stop(){
40         f.setVisible(false);
41     }
42     public void paint(Graphics g){
43         g.drawString("This is in Applet Window", 10, 20);
44 }}
```

ნახ. 196

შედეგი:



ნახ. 197

3.17.4. ტექსტური ჭდეები

ყველაზე მარტივი მართვის ელემენტია ჭდე (Label). ტექსტური ჭდე არის Label კლასის ობიექტი, რომელიც შეიცავს სტრიქონს, რომელსაც ის ფანჯარაში გამოსახავს. ჭდე პასიური ელემენტია და მომხმარებელთან არავითარი ურთიერთობა არ შეუძლია დაამყაროს. მისი კონსტრუქტორებია:

```
Label()
Label(String str)
Label(String str, int how)
```

პირველი ფორმა ცარიელ ჭდეს ქმნის, მეორე str სტრიქონიან ჭდეს, რომელიც მარცხნივაა მოთავსებული. მესამე ფორმა წარმოშობს ჭდეს, რომელიც ინიციალიზირებულია str სტრიქონით და how პარამეტრში მითითებული მნიშვნელობის მიხედვით არის გასწორებული. how პარამეტრის მნიშვნელობა უნდა იყოს ერთ-ერთი ამ სამი კონსტანტიდან: Label.LEFT, Label.RIGTH, Label.CENTER.

ტექსტი ჭდეში შეიძლება დავაყენოთ ან შევცვალოთ setText() მეთოდით. getText() მეთოდით შესაძლებელია ჭდის მიმდინარე მნიშვნელობის წაკითხვა. ამ მეთოდების ფორმატი შემდეგია:

```
void setText(String str)
String getText()
```

setAlignment() მეთოდის გამოძახებით, შესაძლებელია სტრიქონის გასწორება ჭდის არეს ფარგლებში. მიმდინარე გასწორების მისაღებად გამოიყენება getAlignment() მეთოდი. ამ მეთოდების ფორმატია:

```
void setAlignment(int how)
int getAlignment()
```

სადაც how ზემოთ აღწერილი გასწორების კონსტანტებია.

3.17.5. ღილაკები (Button)

ყველაზე ხშირად გამოყენებად მართვის ელემენტს ღილაკი წარმოადგენს. ესაა კომპონენტი, რომელიც შეიცავს ტექსტურ ჭდეს და იწვევს ხდომილებას, როცა მასზე აწვებიან. ის Button კლასის ობიექტს წარმოადგენს. ამ კლასში ორი კონსტრუქტორია განსაზღვრული:

Button()

Button(String str)

პირველი ვერსია ცარიელ ღილაკს ქმნის, მეორე – ღილაკი ტექსტური ჭდითაა.

ღილაკის შექმნის შემდეგ შესაძლებელია ჭდის დაყენება (satLabel()) ან ჭდის მიმდინარე მნიშვნელობის წაკითხვა(getLabel()). ამ მეთოდების ფორმატებია:

void setLabel(String str)

String getlabel()

ღილაკების დამუშავება

ყოველთვის როდესაც ღილაკზე დაჭერა ხდება, action-ხდომილება გენერირდება. ის იმ მომსმენ ბლოკებს ეგზავნება, რომლებიც დარეგისტრირდნენ, როგორც ამ კომპონენტში წარმოქმნილი ხდომილებების შესახებ შეტყობინების მიმღებები. დარეგისტრირება addTypeListener() მეთოდით ხდება, რომელსაც ზოგადად შემდეგი ფორმა აქვს:

public void addTypeListener(TypeListener el)

სადაც Type ხდომილების სახელია, ხოლო el ხდომილების მომსმენ ბლოკზე კავშირი. მაგალითად, კლავიატურით გამოწვეული ხდომილების მომსმენი ბლოკის დასარეგისტრირებელი სახელია addKeyListener(), „მაუსის“ მოძრაობის ხდომილების მომსმენი ბლოკის დასარეგისტრირებელი სახელია addMouseMotionListener().

თითოეული მოსმენის ბლოკი რეალიზაციას უკეთებს ActionListener ინტერფეისს. ეს ინტერფეისი actionPerformed() მეთოდს განსაზღვრავს, რომელიც ხდომილების წარმოშობის შემთხვევაში გამოიძახება. ამ მეთოდს არგუმენტად ActionEvent ობიექტი გადაეცემა. ის შეიცავს კავშირს იმ ღილაკთან, რომელმაც ხდომილება წარმოშვა და აგრეთვე, კავშირს იმ სტრიქონთან, რომელიც ღილაკის ჭდეს წარმოადგენს. ღილაკის იდენტიფიკაციისათვის (გამოცნობისათვის) შეიძლება ორივე ეს კავშირი იქნეს გამოყენებული.

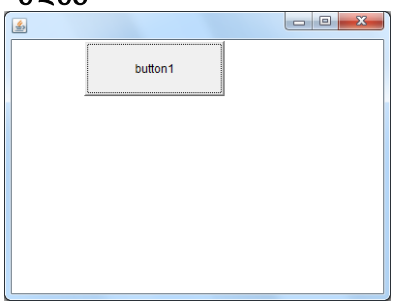
მაგალითი 4. აქ ნაჩვენებია ღილაკზე მაუსით დაწკაპუნების შემთხვევაში წარმოქმნილი ხდომილების დამუშავებისათვის საჭირო მექანიზმი. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 198-ზე, ხოლო შესრულების შედეგი ნახ. 199-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package components;
2  import java.awt.*;
3  import javax.swing.*;
4  import java.awt.event.*;
5  public class Frame2 extends Frame
6  {
7    Button button1 = new Button();
8    public Frame2()
9    {
10     try
11     {
12       jbInit();
13     }
14     catch(Exception ex)
15     {
16       ex.printStackTrace();
17     }}
18   void jbInit() throws Exception
19   {
20     button1.setLabel("button1");
21     button1.setBounds(new Rectangle(83, 31, 145, 57));
22     button1.addActionListener(new Frame2_button1_actionAdapter(this));
23     this.setLayout(null);
24     this.add(button1, null);}
25   public static void main(String[] args)
26   {
27     Frame2 frame2 = new Frame2();
28     frame2.setSize(400,300);
29     frame2.setVisible(true) ;}
30   void button1_actionPerformed(ActionEvent e)
31   {
32
33   }
34 }
35 class Frame2_button1_actionAdapter implements java.awt.event.ActionListener
36 {
37   Frame2 adaptee;
38
39   Frame2_button1_actionAdapter(Frame2 adaptee)
40   {
41     this.adaptee = adaptee;
42   }
43   public void actionPerformed(ActionEvent e)
44   {
45     adaptee.button1_actionPerformed(e);
46   }}
```

ნახ. 198

შედეგი:

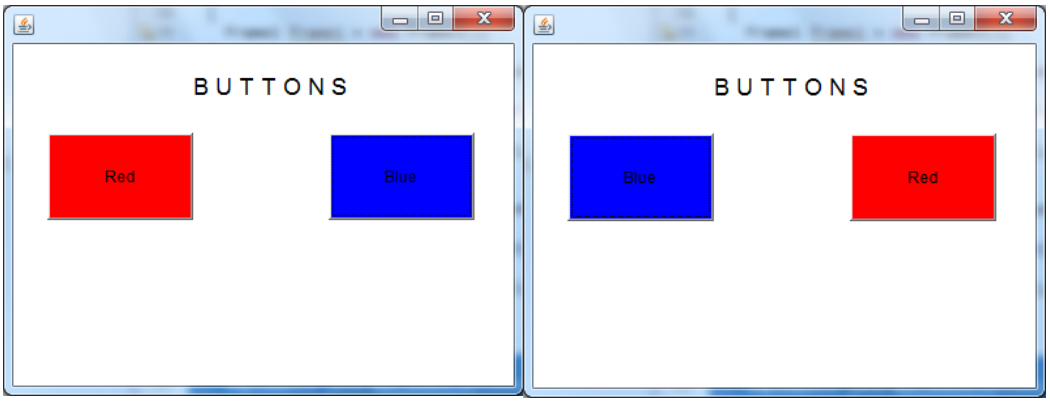


ნახ. 199

მაგალითში 5. ფრეიმში ორი ღილაკია გამოტანილი, წითელი და ლურჯი, შესაბამისი წარწერებით. წითელზე დაჭერით, მისი ფერი და შესაბამისი წარწერა უნდა შეიცვალოს ლურჯით, ხოლო ლურჯი ღილაკის ფერი და წარწერა – წითლით. შესაბამისად ლურჯ ღილაკზე დაჭერით იგი უნდა გაწითლდეს, ხოლო წითელი უნდა გალურჯდეს.

დასმული ამოცანის შედეგი მე-200 ნახაზზეა წარმოდგენილი, ხოლო პროგრამული რეალიზაცია - 201-ე ნახაზზე.

შედეგი:



ნახ. 200

პროგრამის კომპიუტერული რეალიზაცია:

```
package components;
import java.awt.*;
import java.awt.event.*;
public class Frame1 extends Frame
{
  Label label1 = new Label();
  Button button1 = new Button();
  Button button2 = new Button();
  public Frame1()
  {
    try
    {      jbInit(); }
    catch(Exception ex)
    { ex.printStackTrace();  }}
  void jbInit() throws Exception
  {
    label1.setAlignment(Label.CENTER);
    label1.setFont(new java.awt.Font("Times New Roman", 0, 18));
    label1.setText("B U T T O N S");
    label1.setBounds(new Rectangle(40, 37, 330, 51));
    this.setLayout(null);
```

```
    button1.setBackground(Color.red);
    button1.setLabel("Red");
    button1.setBounds(new Rectangle(34, 98, 112, 67));
    button1.addActionListener(new Frame1_button1_actionAdapter(this));
    button2.setBackground(Color.blue);
    button2.setLabel("Blue");
    button2.setBounds(new Rectangle(250, 98, 112, 67));
    button2.addActionListener(new Frame1_button2_actionAdapter(this));
    this.setVisible(true);
    this.setSize(400, 300);
    this.add(button2, null);
    this.add(button1, null);
    this.add(label1, null);}
public static void main(String[] args)
  {
    Frame1 frame1 = new Frame1();
  }
  void button1_actionPerformed(ActionEvent e)
  {
    button1.setBackground(Color.blue);
    button1.setLabel("Blue");
    button2.setBackground(Color.red);
    button2.setLabel("Red");}
  void button2_actionPerformed(ActionEvent e)
  {
    button2.setBackground(Color.blue);
    button2.setLabel("Blue");
    button1.setBackground(Color.red);
    button1.setLabel("Red");
  }}
```

ნახ. 201

3.17.6. ტექსტური ველი (TextFields)

TextField კლასი ერთსტრიქონიანი არეს წარმოადგენს, რომელშიც ტექსტის შეტანა და რედაქტირებაა შესაძლებელი (მაუსით ან კლავიატურის ღილაკების კომბინაციით ტექსტის მონიშვნა, კოპირება, ამოჭრა, ჩაკერება). ეს კლასი TextComponent კლასის ქვეკლასია. TextField კლასის კონსტრუქტორებია:

TextField()

TextField(int numChars)

TextField(String str)

TextField(String str, int numChars)

პირველი ფორმა ქმნის სტანდარტულ ტექსრურ ველს. მეორე – numChars რაოდენობის სიმბოლოების ტექსტურ ველს. მესამე – ტექსტურ ველს str სტრიქონით. მეოთხე – ტექსტურ ველში იწერება str სტრიქონი და ველის სიგრძე numChars სიმბოლოების რაოდენობა.

ტექსტური ველში ჩაწერილი სტრიქონის მისაღებად getText() მეთოდს ვიყენებთ, ხოლო ამ ველში სტრიქონის ჩასაწერად setText() მეთოდს. ამ მეთოდების ზოგადი ფორმებია:

```
String getText()
void setText(String str)
```

მომხმარებელს შეუძლია ტექსტური ველის ნაწილის ამორჩევა (მონიშვნა). select() მეთოდით შესაძლებელია ტექსტის ნაწილის არჩევა პროგრამულად. getSelectedText() მეთოდით შეიძლება მიმდინარე არჩეული ტექსტი მივიღოთ. ამ მეთოდების ფორმატებია:

```
String getSelectedText()
void select(int startIdex, int endIndex)
```

getSelectedText() მეთოდი მონიშნულ ტექსტს აბრუნებს, ხოლო select() მეთოდი სიმბოლოებს ნიშნავს დაწყებული startIdex-დან დამთავრებული endIndex-1 –მდე.

setEditable() მეთოდის გამოძახებით, შეიძლება ვმართოთ ის ფაქტი, ტექსტური ველი მომხმარებლის მიერ იყოს თუ არა რედაქტირებადი (ტექსტში ცვლილებების შეტანის საშუალება ჰქონდეს თუ არა მომხმარებელს). isEditable() მეთოდით შეიძლება გავიგოთ რედაქტირებადია თუ არა მოცემული ველი. ამ მეთოდების ფორმატებია:

```
boolean isEditable()
void setEditable(boolean canEdit)
```

isEditable() მეთოდი აბრუნებს true-ს, თუ ტექსტი რედაქტირებადია, false-ს საწინააღმდეგო შემთხვევაში. setEditable() მეთოდში თუ canEdit ტოლია true-ს, მაშინ შესაძლებელია ველის რედაქტირება, false-ის შემთხვევაში – არ შეიძლება.

თუ მომხმარებელს ტექსტი შეაქვს ველში, რომელიც საიდუმლო ველში (პაროლის შემთხვევაში) არ უნდა აისახოს, მაშინ შეტანილი სიმბოლოების გამოჩენა ეკრანზე უნდა გამოირთოს setEchoChar() მეთოდის გამოძახებით. ამ მეთოდით განისაზღვრება ერთი სიმბოლო, რომელიც ველში თითოეული სიმბოლოს აკრეფის შემთხვევაში გამოვა. ამრიგად, ფაქტიურად შეტანილი სიმბოლოების მაგივრად ეკრანზე ერთიდაიგივე სიმბოლო გამოჩნდება. ხშირად ასეთ სიმბოლოდ '*' (ვარსკვლავს) ირჩევენ ხოლმე. echoCharIsSet() მეთოდით შეიძლება შემოწმდეს არის თუ არა ველი ასეთ რეჟიმში. getEchoChar() მეთოდით შეიძლება ექო-სიმბოლოს ამოღება. ამ მეთოდების ფორმატებია:

```
void setEchoChar(char ch)
```

boolean echoCharIsSet()

char getEchoChar()

აქ ch-ით აღნიშნულია ექო-სიმბოლო, რომელიც ეკრანზე აისახება.

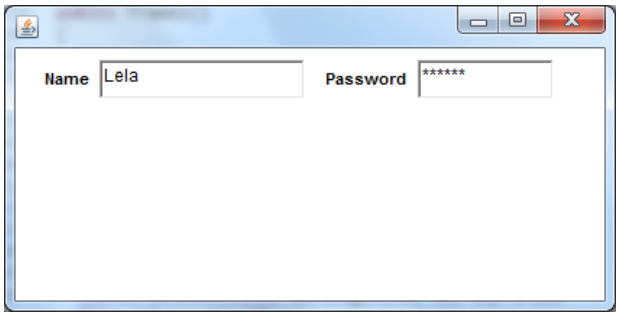
ვინაიდან ტექსტური ველები თავის საკუთარ რედაქტირების ფუნქციებს აწარმოებენ, პროგრამა არ გამოეხმაურება ინდივიდუალურ key-ხდომილებას (კლავიატურის კლავიშზე ხელის დაჭერას), რომლებიც ტექსტურ ველში წარმოიშვება. თუმცა, შესაძლებელია <Enter> კლავიშზე დაჭერისაგან წარმოქმნილი ხდომილების დამუშავება. <Enter> კლავიშზე დაჭერის შემთხვევაში action-ხდომილება გენერირდება (წარმოიქმნება).

მაგალითში 7. ორი რედაქტირებადი ველია (textField1, textField2), რომლესაც წინ წამძღვარებული აქვს label1 ტექსტური ჭდე (Name) და label2 ტექსტური ჭდე (Password). textField2 ტექსტური ველში აკრეფილი სიმბოლოების მაგივრად აისახება '*' (ვარსკვლავის) სიმბოლო. კურსორის ნებისმიერ ტექსტურ არეში განლაგებისას <Enter>-კლავიშზე ხელის დაჭერით:

- label3-ში გამოდის Name-ში შეტანილი ტექსტი;
- label4-ში გამოდის Name-ში მონიშნული ტექსტი;
- label5-ში გამოდის Password-ში აკრეფილი რეალური ტექსტი, რომელიც პირდაპირ ეკრანზე არ ჩანს.

დასმული ამოცანის შედეგი ნახ. 202-ზეა წარმოდგენილი, ხოლო პროგრამული რეალიზაცია - ნახ. 203-ზე.

შედეგი



ნახ. 202

პროგრამის კომპიუტერული რეალიზაცია:

```
package components;
import java.awt.*;
import java.awt.event.*;
public class Frame1  extends Frame
{
  Label label1 = new Label();
  TextField textField1 = new TextField();
  Label label2 = new Label();
  TextField textField2 = new TextField();
  Label label3 = new Label();
  Label label4 = new Label();
  Label label5 = new Label();
  public Frame1()
  {
    try
    {      jbInit();  }
    catch(Exception ex)
    {
      ex.printStackTrace();
    }
  }
  void jbInit() throws Exception
  {
    label1.setFont(new java.awt.Font("Dialog", 1, 11));
    label1.setText("Name");
    label1.setBounds(new Rectangle(26, 38, 38, 27));
    this.setLayout(null);
    this.setVisible(true);
    this.setSize(400, 300);
    textField1.setText("");
    textField1.setBounds(new Rectangle(66, 38, 142, 27));
    textField1.addActionListener(new
Frame1_textField1_actionAdapter(this));
    //textField1.addActionListener(new
Frame1_textField1_actionAdapter(this));
    label2.setFont(new java.awt.Font("Dialog", 1, 11));
    label2.setText("Password");
    label2.setBounds(new Rectangle(220, 38, 60, 27));
    textField2.setEchoChar('*');
    textField2.setText("");
    textField2.setBounds(new Rectangle(286, 38, 94, 27));
    textField2.addActionListener(new
Frame1_textField2_actionAdapter(this));
    label3.setText("");
    label3.setBounds(new Rectangle(34, 88, 250, 46));
    label4.setText("");
    label4.setBounds(new Rectangle(34, 134, 250, 46));
    label5.setText("");
    label5.setBounds(new Rectangle(34, 188, 250, 46));
    this.add(label3, null);
    this.add(label4, null);
    this.add(label5, null);
    this.add(textField1, null);
    this.add(label1, null);
    this.add(label2, null);
    this.add(textField2, null);
  }
```

```
public static void main(String[] args)
  {
    Frame1 frame1 = new Frame1();
  }
  void textField1_actionPerformed(ActionEvent e)
  {
    metReturn(e);
  }
  void textField2_actionPerformed(ActionEvent e)
  {
    metReturn(e);
  }
  void metReturn(ActionEvent e)
  {
    String str = e.getActionCommand() ;
    System.out .println(str) ;
    String name = textField1.getText() ;
    label3.setText(name) ;
    String selectedName = textField1.getSelectedText() ;
    label4.setText(selectedName) ;
    String password = textField2.getText() ;
    label5.setText(password) ;
  }
}
class Frame1_textField1_actionAdapter implements
java.awt.event.ActionListener
{
  Frame1 adaptee;
  Frame1_textField1_actionAdapter(Frame1 adaptee)
  {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e)
  {
    adaptee.textField1_actionPerformed(e);
  }
}
class Frame1_textField2_actionAdapter implements
java.awt.event.ActionListener
{
  Frame1 adaptee;
  Frame1_textField2_actionAdapter(Frame1 adaptee)
  {
    this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e)
  {
    adaptee.textField2_actionPerformed(e);
  }
}
```

бзв. 203

ტექსტური არე (TextArea)

AWT პაკეტი შეიცავს მარტივი მრავალსტრიქონიანი რედაქტორის კლასს TextArea. ამ კლასის კონსტრუქტორებია:

```
TextArea()
TextArea(int numLines, int numChars)
TextArea(String str)
TextArea(String str, int numLines, int numChars)
TextArea(String str, int numLines, int numChars, int sBars)
```

აქ numLines ტექსტური არეს სიმაღლეს განსაზღვრავს; numChars – მის სიგანეს (სიმბოლოებში); str – საწყის ტექსტს; მეხუთე ფორმაში შესაძლებელია განვსაზღვროთ, მართვის ელემენტს ჰქონდეს თუ არა "ლიფტები" (ჰორიზონტალური ან ვერტიკალური სქროლინგის ხაზები). sBars-მა ჩამოთვლილიდან ერთ-ერთი მნიშვნელობა უნდა მიიღოს:

- SCROLLBARS_BOTH
- SCROLLBARS_HORIZONTAL_ONLY
- SCROLLBARS_NONE
- SCROLLBARS_VERTICAL_ONLY

TextArea კლასი TextComponent კლასის ქვეკლასია. ამიტომ ის უზრუნველყოფს ჩვენ მიერ უკვე აღწერილ მეთოდებს: getText(), setText(), getSelectedText(), select(), isEditable(), setEditable().

TextArea კლასში დამატებულია სხვა მეთოდებიც. ესენია:

```
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
```

append() მეთოდი str სტრიქონს მიმდინარე ტექსტის ბოლოში ამატებს. insert() –str სტრიქონს index პოზიციაში ჩაამატებს. replaceRange() მეთოდი – str ტექსტით შეცვლის სიმბოლოებს დაწყებული startIndex პოზიციიდან, დამთავრებული endIndex-1 –მდე.

მაგალითი 8. შევადგინოთ პროგრამა, რომელშიც TextArea კლასის ელემენტი იქმნება.

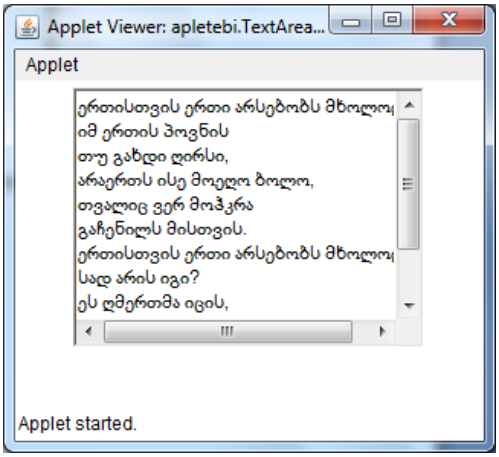
დასმული ამოცანის შედეგი ნახ. 205-ზეა წარმოდგენილი, ხოლო პროგრამული რეალიზაცია - ნახ. 204-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package apletebi;
2  import java.awt.*;
3  import java.applet.*;
4  /*
5  <applet code="TextAreaDemo" width=300 height=250>
6  </applet>
7  */
8  public class TextAreaDemo extends Applet{
9      public void init(){
10         String val=
11 "ერთისთვის ერთი არსებობს მხოლოდ\n" +
12 "იმ ერთის პოვნის\n" +
13 "თუ გახდი ღირსი,\n" +
14 "არაერთს ისე მოელო ბოლო, \n" +
15 "თვალიც ვერ მოჰკრა\n" +
16 "გაჩენილს მისთვის.\n" +
17 "ერთისთვის ერთი არსებობს მხოლოდ,\n" +
18 "სად არის იგი?\n" +
19 "ეს ღმერთმა იცის,\n" +
20 "იმ ერთის ხილვის, \n" +
21 "იმ ერთის პოვნის, \n" +
22 "ათასში ერთი გამხდარა ღირსი!";
23 TextArea text=new TextArea(val, 10, 30);
24 add(text);
25     }
26 }
```

ნახ. 204

შედეგი:



ნახ. 205

3.17.7. List ტიპის სია

List კლასი უზრუნველყოფს კომპაქტური მრავალელემენტიანი სიის ორგანიზებას. სიაში შესაძლებელია რამოდენიმე არჩევანის ერთდროული მონიშვნა. თუ შესაძლო არჩევანის ჩამონათვალი კომპონენტის გამოსაჩენ არეში არ ეტევა, შეიძლება გამოვიყენოთ ამ კომპონენტის "ლიფტი" (ჰორიზონტალური ან ვერტიკალური სქროლინგის ხაზები). List კლასის კონსტრუქტორებია:

List();

List(int numRows);

List(int numRows, boolean multipleSelect)

პირველი ფორმა ქმნის List ელემენტს, რომელსაც მხოლოდ ერთი ელემენტის არჩევა შეუძლია. მეორე ფორმაში numRows პარამეტრის მნიშვნელობა განსაზღვრავს, რამდენი სტრიქონი ჩანდეს სიაში (დანარჩენები შეიძლება გამოჩნდეს თუ გადავფურცლავთ "ლიფტით"). მესამე ფორმაში, თუ multipleSelect პარამეტრი ჭეშმარიტია, შესაძლებელია მომხმარებელმა ერთი ან მეტი სიის ელემენტი ერთდროულად აირჩიოს. თუ მცდარია, მხოლოდ ერთი ელემენტის არჩევაა შესაძლებელი.

სიაში ასარჩევი ელემენტის დასამატებლად add() მეთოდი გამოიყენება, რომელსაც შემდეგი ფორმები აქვს:

void add(String name)

void add((String name, int index)

სადაც name – იმ ელემენტის სახელია, რომელიც სიას ემატება. პირველი ფორმა ელემენტს სიის ბოლოში ამატებს. მეორე - ელემენტებს index პარამეტრით განსაზღვრულ ადგილზე ამატებს. ინდექსირება 0-დან იწყება. თუ სტრიქონის დამატება სიის ბოლოს გვსურს, ინდექსად -1 უნდა მივუთითოთ.

იმ სიებში, სადაც მხოლოდ ერთი ელემენტის არჩევა შეიძლება, getSelectedItem() და getSelectedIndex() მეთოდებით შეგვიძლია განვსაზღვროთ ახლა, ამ მომენტში რომელი ელემენტია არჩეული. აღნიშნული მეთოდების ფორმატებია:

String getSelectedItem()

int getSelectedIndex()

მეთოდი getSelectedItem() არჩეული ელემენტის სახელის შესაბამის სტრიქონს აბრუნებს. თუ ერთზე მეტი ელემენტია არჩეული, ან არცერთი ელემენტი არ არის არჩეული, მეთოდი null-ს აბრუნებს. getSelectedIndex() მეთოდი არჩეული ელემენტის ინდექსს აბრუნებს. პირველი ელემენტის ინდექსია – 0. თუ ერთზე მეტი ელემენტია არჩეული ან არცერთი ელემენტი ჯერ არ არის არჩეული, აბრუნებს – "-1"-ს.

სიებში, რომლებიც მრავლობითი მონიშვნის საშუალებას იძლევიან, getSelectedItems() და getSelectedIndexes() მეთოდები უნდა გამოვიყენოთ. მათი ჩაწერის ფორმატებია:

String[ ] getSelectedItems()

int [ ] getSelectedIndexes()

მეთოდი getSelectedItems() არჩეული ელემენტების სახელების მასივს აბრუნებს. getSelectedIndexes() მეთოდი, კი არჩეული ელემენტების ინდექსების მასივს აბრუნებს.

სიაში ელემენტების რაოდენობის გასაგებად getItemCount() მეთოდი გამოიყენება. select() მეთოდით მიმდინარე ელემენტის არჩეულად დაყენება, მონიშვნაა შესაძლებელი.

ამ მეთოდების ჩაწერის ფორმატებია:

```
int getItemCount()
void select(int index)
```

სადაც index მოსანიშნი ელემენტის ინდექსია (მისი მნიშვნელობა 0-დან იწყება).

თუ ინდექსი ვიცით, getItem() მეთოდით შეგვიძლია მივიღოთ მასთან დაკავშირებული ელემენტის სახელი. getItem() მეთოდის ჩაწერის ფორმატია:

```
String getItem(int index)
```

სადაც index – საჭირო ელემენტის ინდექსია.

List-მოვლენების (list events) დასამუშავებლად, რეალიზაცია ActionListener ინტერფეისს უნდა გავუკეთოთ. ყოველთვის როდესაც სიის ელემენტზე ორმაგი დაწკაპუნება მოხდება, ActionEvent ობიექტი გენერირდება. მისი getActionCommand() მეთოდი შეიძლება ახლად არჩეული ელემენტის სახელის ამოსაღებად გამოვიყენოთ. გარდა ამისა, ყოველთვის როდესაც ელემენტის არჩევა ხდება ან პირიქით, არჩეულის მოხსნა („მაუსის“ ერთმაგი დაწკაპუნებით), ItemEvent ობიექტი გენერირდება. მისი getStateChange() მეთოდი შეიძლება გამოვიყენოთ, იმ მიზნით, რომ დავადგინოთ რამ გამოიწვია მოვლენა – ელემენტის არჩევამ, თუ არჩევის მოხსნამ. getItemSelectable() მეთოდი აბრუნებს იმ ობიექტზე კავშირს, რომელმაც ეს მოვლენა წარმოშვა.

3.17.8. Choice ტიპის სია

კლასი Choice ჩამოშლადი ელემენტების სიის ორგანიზებისათვის გამოიყენება, რომლისგანაც ერთ-ერთის არჩევა შეგვიძლია. ამრიგად, Choice კომპონენტს მენიუს ფორმა აქვს. არააქტიურ მდგომარეობაში Choice კომპონენტი მხოლოდ იმდენ ადგილს იკავებს, რაც მიმდინარედ არჩეული ელემენტის საჩვენებლად არის საჭირო. როდესაც მომხმარებელი მასზე „მაუსით“ დააწკაპუნებს, ელემენტების სრული სია ჩამოიშლება და ახალი არჩევანის გაკეთება ხდება შესაძლებელი. სიაში თითოეული ელემენტი სტრიქონს წარმოადგენს, რომელიც განთავსებულია მარცხნივ და სიაში იმ მიმდევრობით გამოჩნდება, რა მიმდევრობითაც ისინი დაემატნენ Choice კომპონენტს. ამ კლასს მხოლოდ სტანდარტული, ცარიელი კონსტრუქტორი აქვს, რომელიც ცარიელ სიას ქმნის. მისი ჩაწერის ფორმაა:

```
Choice()
```

ასარჩევი ელემენტის სიაში ჩასამატებლად addItem() ან add() მეთოდები გამოიყენება. ამ მეთოდების სიგნატურებია:

```
void addItem(String name)
```

void add(String name)

სადაც name – დასამატებელი ელემენტის სახელია. ელემენტები სიას იმ მიმდევრობით ემატება, რა მიმდევრობითაც მოხდა add() და addItem() მეთოდების შესრულება.

მოცემულ მომენტში არჩეული ელემენტის განსაზღვრის მიზნით getSelectedItem() ან getSelectedIndex() მეთოდები შეგვიძლია გამოვიძახოთ. მათი ფორმატია:

String getSelectedItem()

int getSelectedIndex()

მეთოდი getSelectedItem() – აბრუნებს სტრიქონს, რომელიც ელემენტის სახელს შეიცავს, ხოლო getSelectedIndex() – ელემენტის ინდექსს (ნომერს). პირველი ელემენტის ინდექსია 0.

სიაში ელემენტთა რაოდენობის მისაღებად getItemCount() მეთოდი უნდა გამოვიძახოთ. არჩეული ელემენტის მიმდინარედ დასაყენებლად select() მეთოდს მივმართავთ, რომელსაც არგუმენტად ან 0-დან დაწყებული ინდექსი ექნება ან სტრიქონი, რომელიც სიის ერთ-ერთი ელემენტის სახელს ემთხვევა. ამ მეთოდების ფორმატებია:

int getItemCount()

void select(int index)

void select(String name)

თუ ვიცით ინდექსი, getItem() მეთოდით შეგვიძლია მივიღოთ მასთან დაკავშირებული ელემენტის სახელი. getItem() მეთოდის ჩაწერის ფორმატია:

String getItem(int index)

სადაც index საჭირო ელემენტის ინდექსია.

ყოველთვის, როდესაც Choice-სიის ელემენტის არჩევა ხდება, Item მოვლენა (ხდომილება) გენერირდება. ის ამ კომპონენტთან დარეგისტრირებულ მომსმენ ბლოკებს ეგზავნება. ყოველი მომსმენი ბლოკი ItemListener ინტერფეისს უკეთებს რეალიზაციას. ეს მეთოდი itemStateChanged() მეთოდს განსაზღვრავს, რომელსაც არგუმენტად ItemEvent ობიექტი გადაეცემა.

მაგალითი 9. წარმოგვიდგენს პროგრამას, რომელსაც მომხამრებლის მიერ ლარებში შეტანილი თანხა გადაყავს დოლარში, ევროსა და რუსულ რუბლში.

დასმული ამოცანის შედეგი ნახ. 206-ზეა წარმოდგენილი, ხოლო პროგრამული რეალიზაცია - ნახ. 207-ზე.

შედეგი:

ნახ. 206

პროგრამის კომპიუტერული რეალიზაცია:

```
package components;
import java.awt.*;
import java.awt.event.*;
public class Frame3 extends Frame
{
  List sVal = new List();
  TextField tfLari = new TextField();
  Label label1 = new Label();
  Button bKon = new Button();
  TextField tfKV = new TextField();
  Label lKV = new Label();
  Label inf = new Label();
  double [] kursebi = {2.45,0.07,2.7};
  public Frame3()
  {
    try
    {
      jbInit();
    }
    catch(Exception ex)
    {
      ex.printStackTrace();
    } }
  void jbInit() throws Exception
    {this.setLayout(null);
    sVal.setBounds(new Rectangle(22, 180, 158, 103));
    sVal.addActionListener(new Frame3_sVal_actionAdapter(this));
    tfLari.setText("");
    tfLari.setBounds(new Rectangle(22, 116, 96, 34));
    label1.setFont(new java.awt.Font("Dialog", 0, 16));
    label1.setText("Lari");
    label1.setBounds(new Rectangle(127, 116, 37, 35));
    bKon.setFont(new java.awt.Font("Dialog", 1, 16));
    bKon.setLabel("Konvertireba");
    bKon.setBounds(new Rectangle(176, 117, 114, 33));
    bKon.addActionListener(new Frame3_bKon_actionAdapter(this));
    tfKV.setText("");
    tfKV.setBounds(new Rectangle(294, 117, 86, 36));
    lKV.setFont(new java.awt.Font("Dialog", 1, 16));
    lKV.setBounds(new Rectangle(394, 117, 49, 35));
    inf.setText("");
    inf.setBounds(new Rectangle(210, 184, 178, 94));
    this.add(sVal, null);
    sVal.add("Dolari") ;
    sVal.add("Rubli") ;
    sVal.add("Evro") ;
    this.add(tfLari, null);
    this.add(label1, null);
    this.add(bKon, null);
    this.add(lKV, null);
    this.add(tfKV, null);
    this.add(inf, null); }
```

```
public static void main(String[] args)
  {
    Frame3 frame3 = new Frame3();
    frame3.setVisible(true) ;
    frame3.setSize(400,400) ;
  }
  void sVal_actionPerformed(ActionEvent e)
  {
    System.out .print(e.toString() ) ;
    //l1.setText(sVal.getSelectedItem() ) ;
    //list1.remove(0) ;
  }
  void bKon_actionPerformed(ActionEvent e)
  {
    double dLari = Double.parseDouble(tfLari.getText() ) ;
    if(sVal.getSelectedIndex() == -1)
    {
      inf.setText("airchie valuta!!!") ;
    }
    else
    {
      double k = kursebi[ sVal.getSelectedIndex()];
      double dVal = dLari / k;

      tfKV.setText(Double.toString(dVal) ) ;
      lKV.setText(sVal.getSelectedItem() ) ;
      inf.setText("") ;
    }
  }
}
class Frame3_sVal_actionAdapter implements
java.awt.event.ActionListener
{
  Frame3 adaptee;  Frame3_sVal_actionAdapter(Frame3 adaptee)
  { this.adaptee = adaptee;}
  public void actionPerformed(ActionEvent e)
  { adaptee.sVal_actionPerformed(e); }
}
class Frame3_bKon_actionAdapter implements
java.awt.event.ActionListener
{
  Frame3 adaptee;
  Frame3_bKon_actionAdapter(Frame3 adaptee)
  { this.adaptee = adaptee;}
  public void actionPerformed(ActionEvent e)
  {
    adaptee.bKon_actionPerformed(e);
  }
}
```

ბეჭ. 207

3.17.9. მონიშვნის "ალმები" (Checkbox)

მონიშვნის ალამი მართვის ელემენტია, რომელიც რაიმე რეჟიმის, პარამეტრის ან ოფციის ჩასართავად/გამოსართავად გამოიყენება. ვიზუალურად Checkbox პატარა ოთხკუთხედს წარმოადგენს, რომელიც შეიძლება მომნიშვნელ მარკერს "√" შეიცავდეს. ყოველ ალამს ტექსტური ჭდე შეესაბამება, რომელიც აღწერს, თუ რომელ ოფციას წარმოადგენს ეს ალამი. მისი მდგომარეობა იცვლება მაუსის დაწკაპუნებით (შესაძლებელია კლავიატურიდანაც შეიცვალოს). ალმები შეიძლება გამოყენებულ იქნეს როგორც დამოუკიდებლად - ინდივიდუალურად, ასევე რომელიმე ჯგუფის ნაწილად. ალმები Checkbox კლასის ობიექტებს წარმოადგენენ. მათი კონსტრუქტორებია:

Checkbox()

Checkbox(String str)

Checkbox(String str, boolean on)

Checkbox(String str, boolean on, CheckboxGroup cbGroup)

Checkbox(String str, CheckboxGroup cbGroup, boolean on)

პირველი ფორმა ქმნის ალამს, რომლის ჭდეც თავიდან ცარიელია. ალმის მდგომარეობა ამ დროს – "გამორთულია"(off);

მეორე ფორმა ქმნის ალამს, რომლის ჭდეც იქნება str. ალამის მდგომარეობა აქაც "გამორთულია" (off);

მესამე ფორმა საშუალებას იძლევა ალამი დავაყენოთ ჩვენთვის საჭირო საწყის მდგომარეობაში. თუ on პარამეტრის მნიშვნელობაა true, შეიქმნება ალამი საწყისი მნიშვნელობით "ჩართულია"(on). თუ on პარამეტრის მნიშვნელობაა false, შექმნება ალამი, რომელიც არ იქნება მონიშნული "√" მარკერით (ე.ი. "off").

ალმის მიმდინარე მდგომარეობის მისაღებად getState() მეთოდი გამოიყენება. ალმის მდგომარეობის დასაყენებლად - setState() მეთოდი. getLabel() მეთოდის საშუალებით შეგვიძლია ალამთან დაკავშირებული ჭდე მივიღოთ.

ყოველ ჯერზე, როდესაც ალამი ფიქსირდება ან იხსნება, ელემენტის შესაბამის მოვლენას (ხდომილებას) აქვს ადგილი. ამის შესახებ ინფორმაცია ყველა იმ მსმენელს გადაეცემა, რომელიც მოცემულ კომპონენტთან დაკავშირებული ხდომილების მიღების თაობაზე წინასწარ დარეგისტრირდა. ყოველი მსმენელი ახდენს ItemListener ინტერფეისის რეალიზებას, რომელიც განსაზღვრავს itemStateChanged() მეთოდს. ItemEvent კლასის ობიექტი აღნიშნულ მეთოდს პარამეტრის სახით გადაეცემა. ის შეიცავს მოვლენასთან დაკავშირებულ ინფორმაციას.

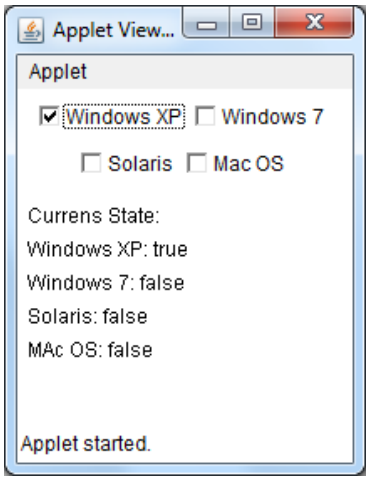
განვიხილოთ **მაგალითი 10**, სადაც ოთხი ალამი იქმნება. პირველი ალამი თავიდანვე დაყენებულია. ყოველი ალმის მდგომარეობა, ისვე, როგორც ცვლილებება, ეკრანზე აისახება.

პროგრამის კომპიუტერული რეალიზაცია:

```
package apletebi;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckBoxDemo" width=240 height=200>
</applet>
*/
public class CheckBoxDemo extends Applet implements ItemListener {
    String msg="";
    Checkbox winXP, win7, solaris, mac;
    public void init(){
        winXP=new Checkbox("Windows XP", null, true);
        win7=new Checkbox("Windows 7");
        solaris=new Checkbox("Solaris");
        mac=new Checkbox("Mac OS");
        add(winXP);
        add(win7);
        add(solaris);
        add(mac);
        winXP.addItemListener(this);
        win7.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }
     public void itemStateChanged(ItemEvent ie){
         repaint();
     }
     public void paint(Graphics g){
         msg="Currens State: ";
         g.drawString(msg, 6, 80);
         msg="Windows XP: " + winXP.getState();
         g.drawString(msg, 6, 100);
         msg="Windows 7: " + win7.getState();
         g.drawString(msg, 6, 120);
         msg="Solaris: " + solaris.getState();
         g.drawString(msg, 6, 140);
         msg="MAc OS: " + mac.getState();
         g.drawString(msg, 6, 160);
     }
}
```

ნახ. 208

შედეგი:



ნახ. 209

3.17.10. CheckboxGroup კლასი

ჩვენ შეგვიძლია ალმების ისეთი ნაკრების შევქმნათ, რომელშიც ერთსა და იმავე დროს მხოლოდ ერთი ალმის დაყენებაა შესაძლებელი. ასეთ ალმებს **გადამრთველებს** უწოდებენ. იმისათვის, რომ ურთიერთდაკავშირებული ღილაკების ნაკრები შევქმნათ, პირველ რიგში საჭიროა იმ ჯგუფის განსაზღვრა, რომელსაც ისინი ეკუთვნიან და ალმების შექმნისას ამ ჯგუფის მითითება.

იმისათვის, რომ გავიგოთ, თუ ჯგუფში რომელი ალამია დაყენებული, საჭიროა getSelectedCheckbox() მეთოდის გამოძახება, ხოლო ალმის დაყენების მიზნით - setSelectedCheckbox() მეთოდის გამოძახება.

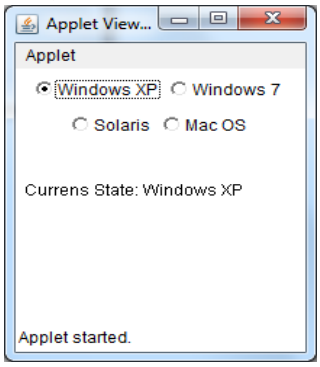
განვიხილოთ **მაგალითი 11**, რომელიც ახდენს ჯგუფში გაერთიანებული ალმების მუშაობის დემონსტრირებას. ალგორითმის პროგრამული რეალიზაცია მოცემულია ნახ. 210-ზე, ხოლო შესრულების შედეგი ნახ. 211-ზე.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package apletebi;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.applet.*;
5  /*
6  <applet code="CBGroup" width=254 height=200>
7  </applet>
8  */
9  public class CBGroup extends Applet implements ItemListener {
10     String msg="";
11     Checkbox winXP, win7, solaris, mac;
12     CheckboxGroup cbg;
13     public void init(){
14         cbg=new CheckboxGroup();
15         winXP=new Checkbox("Windows XP", cbg, true);
16         win7=new Checkbox("Windows 7", cbg, false);
17         solaris=new Checkbox("Solaris", cbg, false);
18         mac=new Checkbox("Mac OS", cbg, false);
19         add(winXP);
20         add(win7);
21         add(solaris);
22         add(mac);
23         winXP.addItemListener(this);
24         win7.addItemListener(this);
25         solaris.addItemListener(this);
26         mac.addItemListener(this);
27     }
28     public void itemStateChanged(ItemEvent ie){
29          repaint();
30      }
31     public void paint(Graphics g){
32          msg="Currens State: ";
33          msg+=cbg.getSelectedCheckbox().getLabel();
34          g.drawString(msg, 6, 100);
35      }
36    }
```

ნახ.210

შედეგი:



ნახ. 211

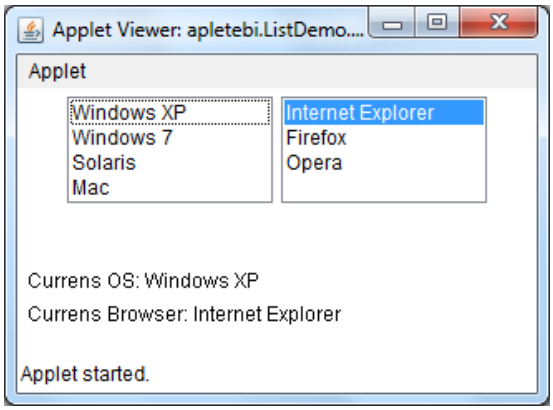
3.17.11.სიების დამუშავება

სიებთან დაკავშირებული ხდომილებების დასამუშავებლად საჭიროა ActionListener ინტერფეისის რეალიზება. List კლასის მართვის ელემენტზე „მაუსის“ ორჯერ დაწკაპუნებისას ActionEvent კლასის ობიექტი იქმნება. მისი getActionCommand() მეთოდი შეგვიძლია ახლადარჩეული პუნქტის სახელის მისაღებად გამოვიყენოთ. სიის ამათუიმ პუნქტის ამორჩევის ან გაუქმების შემთხვევაში („მაუსის“ ერთხელ დაწკაპუნების დროს) ItemEvent კლასის ობიექტი იქმნება, ხოლო მისი getStateChanged() მეთოდი შეგვიძლია იმ თვალსაზრისით გამოვიყენოთ, რომ გავიგოთ, თუ რით იყო გამოწვეული ესა თუ ის მოვლენა - პუნქტის არჩევით თუ გაუქმებით. getItemSelectable() მეთოდი იმ ობიექტზე მიმართვას აბრუნებს, რომლის ინიცირებითაც ადგილი ქონდა ამა თუ იმ მოვლენას.

მაგალითი 12. წარმოვადგინოთ List კლასის კომპონენტების მუშაობის სადემონსტრაციო პროგრამა.

დასმული ამოცანის შედეგი ნახ. 212-ზეა წარმოდგენილი, ხოლო პროგრამული რეალიზაცია - ნახ. 213-ზე.

შედეგი:



ნახ. 212

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package apletebi;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.applet.*;
5  /*
6  <applet code="ListDemo" width=300 height=180>
7  </applet>
8  */
9  public class ListDemo extends Applet implements ActionListener {
10     List os, browser;
11     String msg="";
12     public void init(){
13         os=new List(4, true);
14         browser=new List(4, false);
15         os.add("Windows XP");   //OS სიაში ელემენტების ჩამატება
16         os.add("Windows 7");
17         os.add("Solaris");
18         os.add("Mac");
19         browser.add("Internet Explorer");//Browser სიაში ელემენტების ჩამატება
20         browser.add("Firefox");
21         browser.add("Opera");
22         browser.select(1);
23         //სიების ფანჯარაში დამატება
24         add(os);
25         add(browser);
26         //ხდომილებებზე რეგისრაცია
27         os.addActionListener(this);
28         browser.addActionListener(this);}
29     public void actionPerformed(ActionEvent ae){
30         repaint();}
31      public void paint(Graphics g){
32          int idx[];
33          msg="Currens OS: ";
34          idx=os.getSelectedIndexes();
35          for(int i=0; i<idx.length; i++)
36              msg+=os.getItem(idx[i]) + "   ";
37              g.drawString(msg, 6, 120);
38              msg="Currens Browser: ";
39              msg+=browser.getSelectedItem();
40              g.drawString(msg, 6, 140); } }
```

ნახ. 213

3.17.12.ჰორიზონტალური და ვერტიკალური

სქროლინგის (გადაფურცვლის) ზოლების მართვა

სქროლინგის ზოლები უწყვეტი მნიშვნელობების ასარჩევად გამოიყენება წინასწარ მოცემული მინიმალური და მაქსიმალური მნიშვნელობების დიაპაზონის არეში. ისინი შეიძლება ორიენტირებული იყოს ჰორიზონტალურად და ვერტიკალურად. რეალურად, სქროლინგის ხაზი (ზოლი) რამდენიმე ცალკეული ნაწილისგან შედგება. მისი ორივე ბოლო ისრით ბოლოვდება, რომელზე დაწკაპუნებაც სქროლინგის ხაზის მიმდინარე მნიშვნელობის ერთი ერთეულით გადაადგილებას იწვევს ისრის მიმართულებით. სქროლინგის ხაზს ასევე გააჩნია მცოცი,

რომლის გადააგილება მომხმარებელს შეუძლია მინიმალურ და მაქსიმალურ მნიშვნელობებს შორის, რის შემდეგაც სქროლინგის ხაზი ახალ მნიშვნელობას დააფიქსირებს.

სქროლინგის ზოლები Scrollbar კლასის მიერ არის ინკაფსულირებული, რომელიც შემდეგ კონსტრუქტორებს განსაზღვრავს:

```
Scrollbar()
Scrollbar(int სტილი)
Scrollbar(int სტილი, int საწყისი მნიშვნელობა, int მცოცის ზომა, int მინ, int მაქს)
```

პირველი კონსტრუქტორი ვერტიკალური სქროლინგის ხაზს ქმნის. მეორე და მესამე ვარიანტებში შესაძლებელია სქროლინგის ხაზის ოპრიენტაციის მიცემა. თუ პარამეტრი „სტილი“ შეიცავს Scrollbar.VERTICAL, ვერტიკალური სქროლინგის ხაზი იქმნება, ხოლო თუ მისი მნიშვნელობაა Scroll.HORIZONTAL - ჰორიზონტალური სქროლინგის ხაზი. მესამე შემთხვევაში სქროლინგის ხაზის საწყის მდგომარეობას პარამეტრი „საწყისი მნიშვნელობა“ განსაზღვრავს, ხოლო ერთეულების რაოდენობა „მცოცის ზომა“ პარამეტრით ფიქსირდება. სქროლინგის ხაზის მინიმალური და მაქსიმალური მნიშვნელობები „მინ“ და“მაქს“ პარამეტრებს გადაეცემა.

თუ თქვენ სქროლინგის ხაზს ზემოთ წარმოდგენილი პირველი ან მეორე კონსტრუქტორის გამოყენებით ქმნით, პირველ რიგში setValue() მეთოდით სქროლინგის ხაზის პარამეტრები უნდა განსაზღვროთ. აღნიშნული მეთოდის ჩაწერის ფორმაა:

```
void setValue(int საწყისი მნიშვნელობა, int მცოცის ზომა, int მინ, int მაქს)
```

სქროლინგის ხაზის მიმდინარე მნიშვნელობის გასაგებად getValue() მეთოდი გამოიძახეთ. მიმდინარე მნიშვნელობის ასაწყობად setValue() მეთოდს მიმართეთ. ეს ორივე მეთოდი შემდეგი ფორმებით ხასიათდება:

```
int getValue()
void setValue(int ახალი მნიშვნელობა)
```

აქ პარამეტრი „ახალი მნიშვნელობა“ სქროლინგის ხაზის ახალ მნიშვნელობას განსაზღვრავს.

getMinimum() და getMaximum() მეთოდები მინიმალურ და მაქსიმალურ მნიშვნელობებს იძლევა. მათი ჩაწერის ფორმებია:

```
int getMinimum()
int getMaximum()
```

ეს მეთოდები შესაბამის მნიშვნელობას აბრუნებენ. სტანდარტულად გადაადგილება ერთი ერთეულით ხდება ზემოთ ან ქვემოთ. ამ ერთეულის მნიშვნელობის შეცვლა setUnitIncrement() მეთოდით შეიძლება. გადაფურცვლა სტანდარტულად ათი ერთეულით წარმოებს ზემოთ ან ქვემოთ, თუმცა ამ სიდიდის შეცვლაც არის შესაძლებელი setBlockIncrement() მეთოდის გამოყენებით. აღნიშნული მეთოდების ჩაწერის ფორმებია:

void setUnitIncrement(int ახალი ინკრემენტი)

void setBlockIncrement(int ახალი ინკრემენტი)

სქროლინგის ხაზებთან დაკავშირებული ხდომილებების დასამუშავებლად აუცილებელია AdjustmentListener ინტერფეისის რეალიზება. როგორც კი მომხმარებელი სქროლინგის ხაზთან მუშაობას იწყებს, წარმოიქმნება AdjustmentEvent კლასის ობიექტი. მისი getAdjustmentType() მეთოდი საჭირო პარამეტრებს განსაზღვრავს.

21-ე ცხრილში წარმოდგენილია სქროლინგის ხაზის აწყობის ხდომილებები.

ცხრილი 21 სქროლინგის ხაზის აწყობის ხდომილებები

ხდომილება	აღწერა
BLOCK_DECREMENT	შესრულდა ქვემოთ გადაფურცვლის ხდომილება
BLOCK_INCREMENT	შესრულდა ზემოთ გადაფურცვლის ხდომილება
TRACK	შესრულდა აბსოლუტური გადაადგილების ხდომილება
UNIT_ DECREMENT	დაჭერილია ფურცლის ერთი სტრიქონით ქვემოთ გადაადგილების ღილაკი
UNIT_ INCREMENT	დაჭერილია ფურცლის ერთი სტრიქონით ზემოთ გადაადგილების ღილაკი

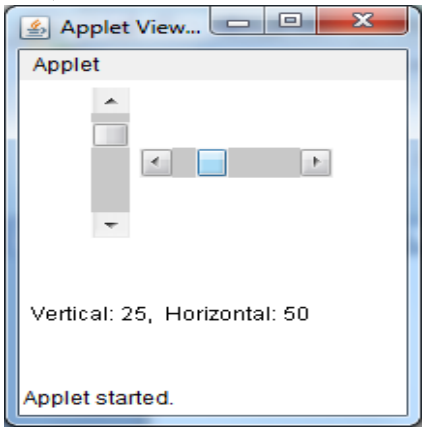
მაგალითი 13. შევადგინოთ ვერტიკალური და ჰორიზონტალური სქროლინგის ხაზების შექმნის კოდი. კურსორის გადაადგილებისას ყოველი ხდომილების კოორდინატი სქროლინგის მიმდინარე მნიშვნელობის გასაახლებლად გამოვიყენოთ. პროგრამაში მეთოდი setPreferredSize() სქროლინგის ხაზების ზომის დასაყენებლად გვჭირდება.

პროგრამის კომპიუტერული რეალიზაცია:

```
1  package apletebi;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.applet.*;
5  /*
6  <applet code="SBDemo" width=300 height=200>
7  </applet>
8  */
9  public class SBDemo extends Applet implements AdjustmentListener, MouseMotionListener  {
10     String msg="";
11     Scrollbar vertSB, horizSB;
12     public void init(){
13         int width=Integer.parseInt(getParameter("width"));
14         int height=Integer.parseInt(getParameter("height"));
15         vertSB=new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0,height );
16         vertSB.setPreferredSize(new Dimension(20, 100));
17         horizSB=new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0,width);
18         horizSB.setPreferredSize(new Dimension(100, 20));
19         add(vertSB);
20         add(horizSB);
21         //რეგისტრაცია ხდომილებებზე
22         vertSB.addAdjustmentListener(this);
23         horizSB.addAdjustmentListener(this);
24         addMouseMotionListener(this);}
25     public void adjustmentValueChanged(AdjustmentEvent ae){
26         repaint();}
27     //სქროლინგის ზაზების განსლება
28     public void mouseDragged(MouseEvent me)
29     {
30         int x=me.getX();
31         int y=me.getY();
32         vertSB.setValue(y);
33         horizSB.setValue(x);
34         repaint();}
35     public void mouseMoved(MouseEvent me){
36
37 }
38     public void paint(Graphics g){
39         msg="Vertical: " + vertSB.getValue();
40         msg+=",  Horizontal: " + horizSB.getValue();
41         g.drawString(msg, 6, 160);
42         //მიმდინარე პოზიციის ჩვენება
43         g.drawString("*", horizSB.getValue(),  vertSB.getValue());}}
```

ნახ. 214

შედეგი:



ნახ.215

დავალება:

1. *შექმენით აპლეტი, რომელიც ეკრანზე გამოიტანს სტრიქონს „აპლეტი“ და წარმოდგენილი სტრიქონი დაიწყება 20, 30 პოზიციიდან.*
2. *შექმენით მარტივი გარკვეული ფონის ფერის (მაგალითად, წითელი) აპლეტი, რომელიც ეკრანზე გამოიტანს სტრიქონს „ჩემი პირველი აპლეტი“ პოზიციიდან 15, 35.*
3. *შექმენით აპლეტი, რომელიც ეკრანზე წარმოგვიდგენს სტანდარტულ ფრეიმ-ფანჯარას სათაურით „ფრეიმ-ფანჯარა“. გამოიყენეთ start() და stop() მეთოდები (ტელახლა განსაზღვრეთ) ფრეიმ-ფანჯრის გამოჩენისა და დაფარვის მიზნით.*
4. *შექმენით აპლეტი, რომელიც ფრეიმ-ფანჯარაში ორ ღილაკს გამოიტანს (წითელს და ყვითელს). ღილაკებს ფერების მიხედვით მიეცით სახლწოდებები და „მაუსით“ დაწკაპუნებისას თითოეულ ღილაკზე წარწერა ღილაკის ფერის შესაბამისად შეცვალეთ.*
5. *შექმენით აპლეტი, რომელიც მოიცავს TextArea კლასის ელემეტს და ნებისმიერ ტექსტს (თუნდაც რაიმე ლექსს) გამოიტანს ტექსტურ ველში სქროლინგის (გადაფურცვლის) ხაზების გამოყენების გათვალისწინებით.*
6. *შექმენით აპლეტი, რომელიც მიმდინარე კურსის გათვალისწინებით ლარს გადაიყვანს ევროსა და დოლარში. ელემენტების არჩევა Choice სიიდან განახორციელეთ.*
7. *შექმენით აპლეტი, რომელიც სამ ალამს მოიცავს. ამათგან მეორე ალამი თავიდანვე ჩართული უნდა იყოს. წარწერები ალმების გვერდით ჩაწერეთ: „ჰიმნი“, „გერბი“ და „დროშა“. ყოველი ალმის მდგომარეობა, ისვე, როგორც ცვლილებება, ეკრანზე გამოსახეთ.*
8. *მე-7 დავალებაში წარმოდგენილი ამოცანა შეცვალეთ შემდეგი სახით, კერძოდ, ალმების ნაცვლად გადამრთველი ღილაკები გამოიყენეთ.*
9. *შექმენით აპლეტი, რომელიც ორ სიას მოიცავს. პირველი სია ოპერაციულ სისტემებს წარმოადგენს, ხოლო მეორე ბრაუზერებს. სიებიდან ელემენტების ამორჩევისას ფრეიმ-ფანჯარაში გამოსახეთ არჩეული ელემენტების სახელწოდებები.*
10. *შექმენით აპლეტი, რომელიც სტანდარტულ ვერტიკალურ და ჰორიზონტალურ სქროლინ-გებს (გადაფურცვლის ზოლებს) მოიცავს. სქროლინგებზე შესრულებული ნებისმიერი ცვლილება ფრეიმ-ფანჯარაში წარმოადგინეთ რიცხვითი მნიშვნელობების სახით.*

4. მონაცემთა ბაზების პროექტირება და მართვა

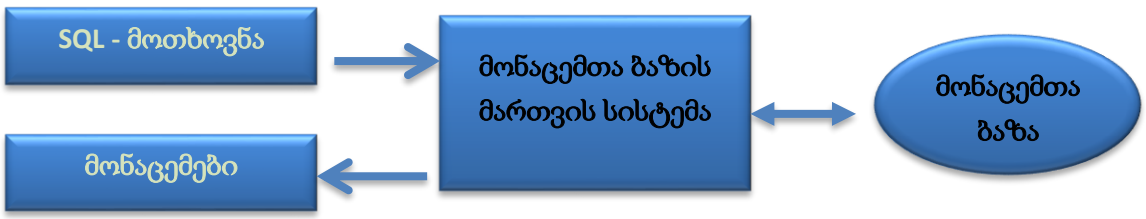
4.1. მონაცემთა ბაზის შექმნა

მონაცემთა ბაზა ეს არის გარკვეული საგნობრივი სფეროს ობიექტების შესახებ მონაცემების მოწესრიგებული ერთობლიობა. საგნობრივი სფერო შეიძლება იყოს სკოლა, უნივერსიტეტი, საავადმყოფო, ფირმა და ა.შ. ობიექტი კი შესაბამისად, მოსწავლე, სტუდენტი, ავადმყოფი, თანამშრომელი და ა.შ. მონაცემთა ბაზაში ეს ინფორმაცია სისტემატიზირებულია ისეთი სახით, რომ შესაძლებელი იყოს მათი მოძიება და ელექტრონულად დამუშავება.

მონაცემთა ბაზასთან მუშაობა მონაცემთა ბაზის მართვის სისტემის საშუალებით შეიძლება. მონაცემთა ბაზის მართვის სისტემა ენობრივი და პროგრამული საშუალებების ერთობლიობაა, რომლის საშუალებითაც იქმნება მონაცემთა ბაზა და სრულდება სხვადასხვა მანიპულაციები (დამატება, რედაქტირება, წაშლა, დაცვა და ა.შ.) ბაზის ობიექტებზე. ობიექტებთან წვდომა ხორციელდება სპეციალური ენის SQL -ის საშუალებით.

SQL - ეს არის სტრუქტურირებული მოთხოვნების ენა, რომლის ძირითად ამოცანას წარმოადგენს მონაცემთა ბაზაში ინფორმაციის ჩაწერა და ბაზიდან ინფორმაციის წაკითხვა.

მონაცემთა ბაზასთან მუშაობის მარტივი სქემა შესაძლებელია წარმოვადგინოთ ნახ.216-ზე მოცემული სახით:



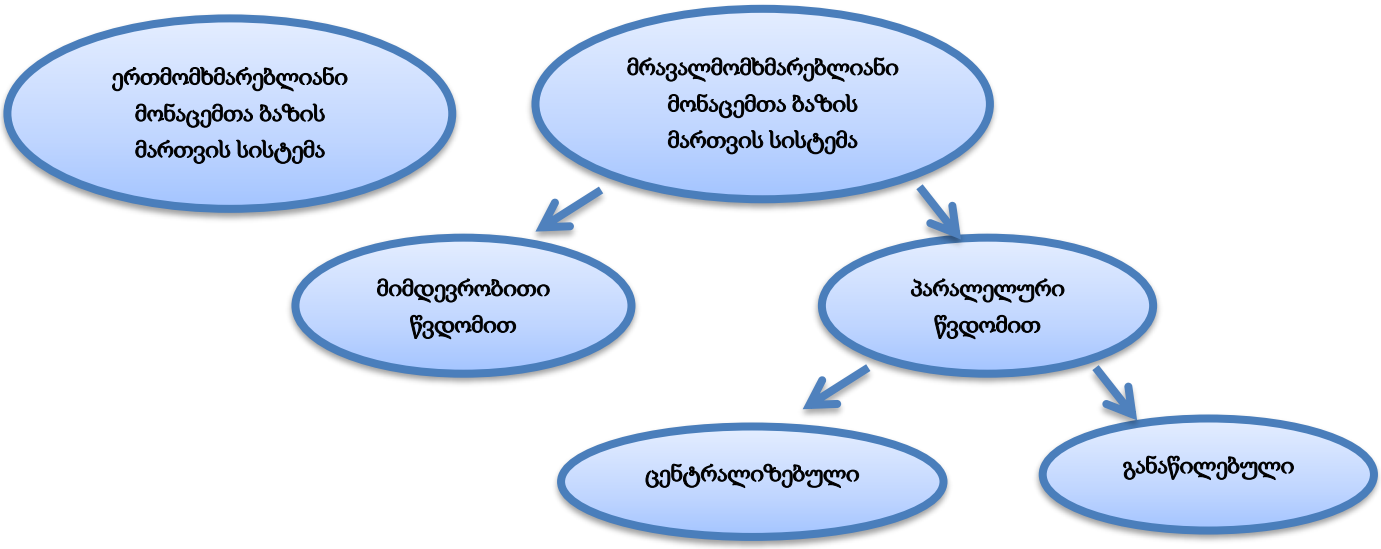
ნახ. 496 მონაცემთა ბაზასთან მუშაობის სქემა

მონაცემთა ბაზის მართვის სისტემები გამოყენების თვალსაზრისის მიხედვით იყოფა ერთმომხმარებლიან და მრავალმომხმარებლიან სისტემებად.

ერთმომხმარებლიანი სისტემა გამოიყენება პერსონალურ კომპიუტერზე მონაცემთა ბაზის შექმნისა და გამოყენებისათვის. ერთმომხმარებლიანი მონაცემთა ბაზის მართვის სისტემებია: Microsoft Visual FoxPro, Access.

მრავალმომხმარებლიანი სისტემა გამოიყენება ქსელში ჩართული კომპიუტერების ერთიანი მონაცემთა ბაზის სამართავად. მრავალმომხმარებლიანი მონაცემთა ბაზის მართვის სისტემებია: MS SQL Server, Oracle, MySQL

გამოყენების თვალსაზრისით მართვის სისტემების დაყოფის სქემა შესაძლებელია წარმოვადგინოთ შემდეგი სახით:



ნახ. 217 განაწილების თვალსაზრისით მართვის სისტემების დაყოფის სქემა

მონაცემთა განაწილებული ბაზა რამდენიმე ნაწილისაგან შედგება, რომლებიც მოთავსებულია კომპიუტერული ქსელის სხვადასხვა კომპიუტერზე (სერვერზე).

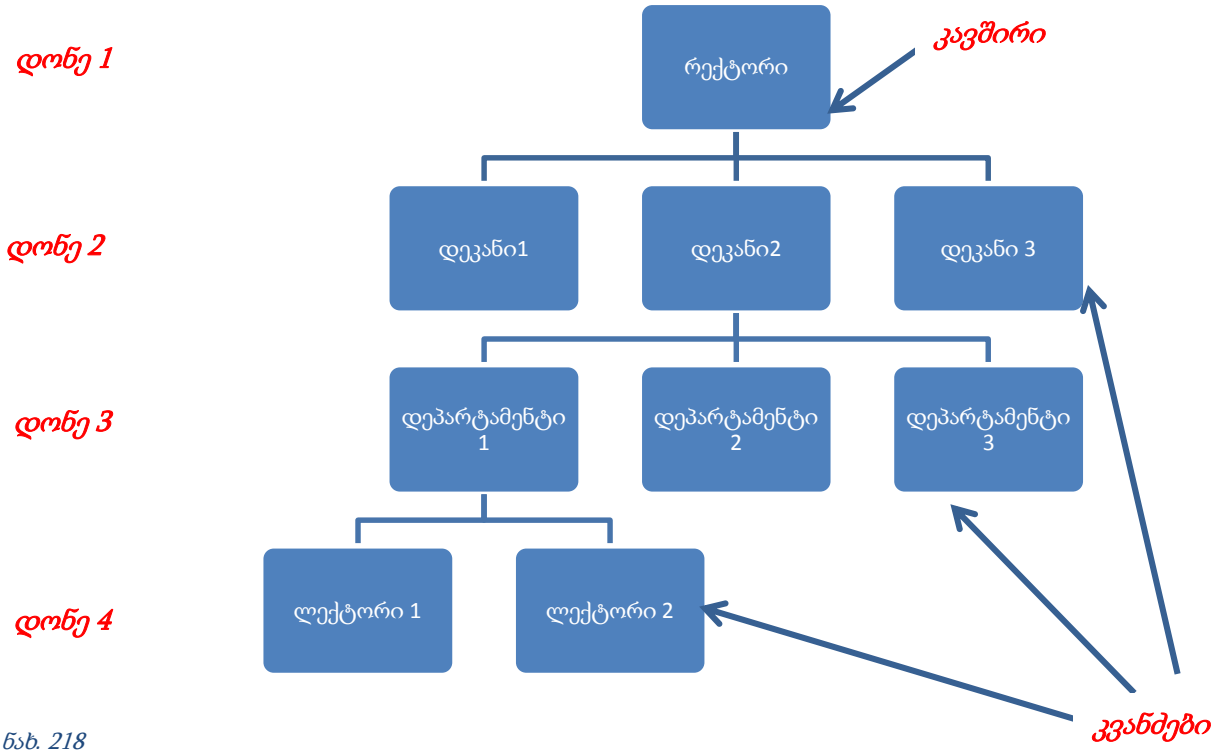
მონაცემთა ცენტრალიზებული ბაზა მოთავსებულია ერთ კომპიუტერზე, რომელიც შეიძლება იყოს ჩართული ლოკალურ ქსელში და შესაბამისად, შესაძლებელი იქნება სხვა კომპიუტერების მხრიდან ამ ბაზასთან მიმართვა. მონაცემთა ცენტრალიზებული ბაზა არქიტექტურის მიხედვით ორგვარია: ფაილ- სერვერი და კლიენტ-სერვერი.

4.1.1. მონაცემთა ბაზის სტრუქტურა

მონაცემთა ბაზების კლასიფიცირება შეიძლება მოვახდინოთ მონაცემების მოდელების მიხედვით. არსებობს მონაცემების იერარქიული, ქსელური და რელაციური მოდელები. ამჟამად ყველაზე გავრცელებულია მონაცემთა ბაზის რელაციური მოდელი.

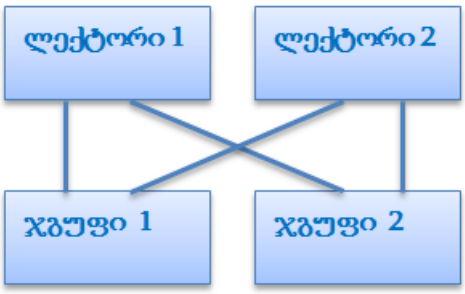
4.1.1.1. მონაცემების იერარქიული მოდელი

მონაცემების იერარქიულ მოდელში ყოველი ობიექტი შეიცავს უფრო დაბალი დონის რამდენიმე ობიექტს. ასეთი ობიექტები ერთმანეთთან ქმნიან ურთიერთობას: წინაპარი (ობიქტი, რომელიც უფრო ახლოს არის ფესვებთან) და მემკვიდრე (უფრო დაბალი დონის ობიექტი).



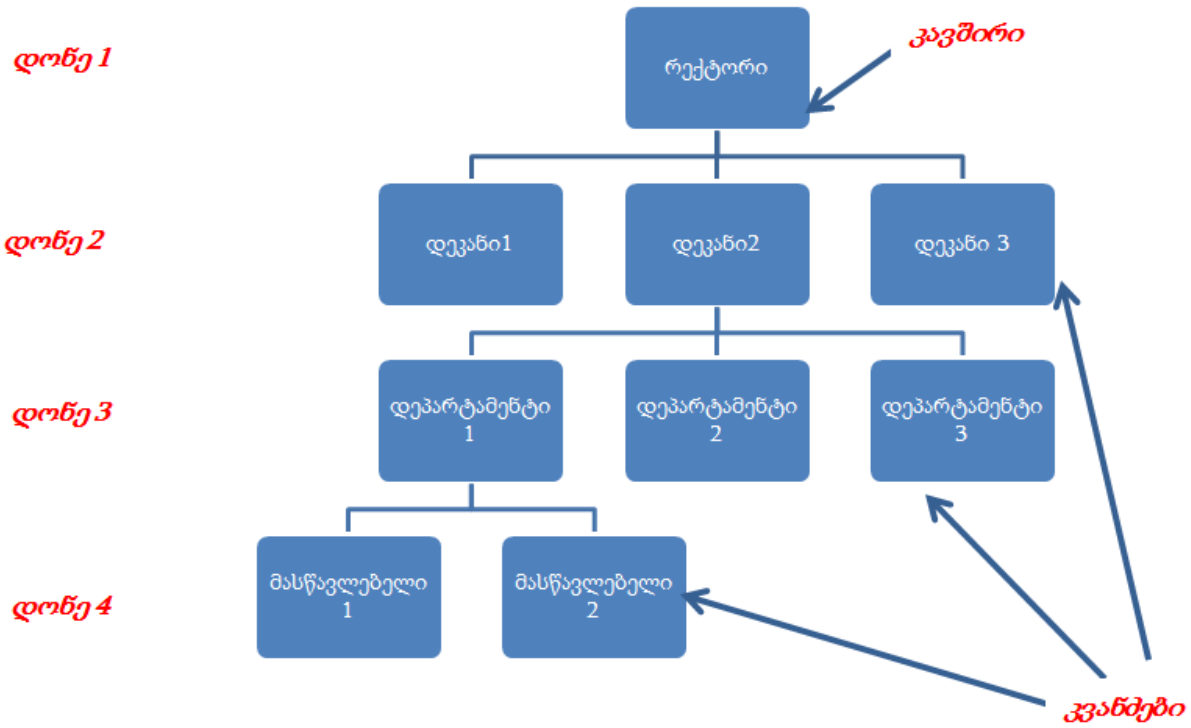
ნახ. 218

როგორც სტრუქტურიდან ჩანს ერთ დეპარტამენტში შეიძლება მუშაობდეს რამდენიმე ლექტორი. ასეთ კავშირს ეწოდება კავშირი „ერთი მრავალთან“ (ერთი დეპარტამენტი - მრავალი ლექტორი). თუ ჩვენ ამ სტრუქტურას დავამატებთ სტუდენტთა ჯგუფს, მაშინ დაგვჭირდება კავშირი „მრავალი მრავალთან“.



ნახ. 219 კავშირი მრავალი-მრავალთან

ერთ ლექტორს შეუძლია მუშაობა რამდენიმე ჯგუფთან, ხოლო ერთ ჯგუფს შეიძლება ასწავლიდეს სხვადასხვა ლექტორი. ასეთი კავშირის დამყარება იერარქიულ მოდელში შეუძლებელია, კავშირი შესაძლებელია იყოს მხოლოდ ერთ მაღალი დონის კვანძთან. ეს იერარქიული მოდელის ერთ-ერთი უარყოფითი მხარეა.



ნახ. 220

4.1.1.2. მონაცემების ქსელური მოდელი

ქსელური მოდელი ეს არის იერარქიული სტრუქტურის გაფართოება. ის ემყარება გრაფთა თეორიას. ამ მოდელში არსებობს კავშირი „მრავალი მრავალთან“. ქსელური სტრუქტურა საშუალებას გვაძლევს იერარქიულ სტრურტურაში განხილულ მაგალითს დავამატოთ ჯგუფები. ქსელური მოდელის გამოყენება საკმაოდ რთულია და ეს მისი უმთავრესი ნაკლოვანებაა.

4.1.1.3. მონაცემების რელაციური მოდელი

რელაციურ მოდელში მონაცემები წარმოდგენილია ორგანზომილებიანი ცხრილის სახით. ყოველი ცხრილი შედგება სვეტებისაგან და სტრიქონებისაგან. სვეტებს ვუწოდებთ ველებს (Fields), ხოლო სტრიქონებს - ჩანაწერებს (records). რელაციურ მონაცემთა ბაზის ცხრილებს შემდეგი ძირითადი თვისებები გააჩნიათ:

1. ცხრილში შეუძლებელია იყოს ორი ერთიდაიგივე სტრიქონი;
2. ცხრილში შესაძლებელია არ იყოს არცერთი სტრიქონი, მაგრამ ერთი სვეტი მაინც უნდა იყოს;
3. სვეტს გააჩნია უნიკალური სახელი და მისი მნიშვნელობები აუცილებლად იქნება ერთიდაიგივე ტიპის (რიცხვი, ტექსტი, თარიღი ...)
4. ყოველი სვეტისა და სტრიქონის გადაკვეთაზე შესაძლებელია იყოს მხოლოდ ერთი მნიშვნელობა. ცხრილები, რომლებიც აკმაყოფილებენ ამ პირობებს, წარმოადგენენ ნორმალიზებულ ცხრილებს

4.1.1.2. მონაცემთა ბაზების ობიექტები:

ცხრილები (tables) - ორგანზომილებიანი მატრიცებია, რომლებშიც უშუალოდ მონაცემები ინახება;

შენახული პროცედურები (stored procedures) - Transact_SQL-ის ბრძანებების ნაკრებია, შენახული გარკვეული სახელით. ამრიგად, ნაცვლად იმისა, რომ შევინახოთ ხშირად გამოყენებული მოთხოვნა, კლიენტს შეუძლია მიმართოს შესაბამის შენახულ პროცედურას, რაც უზრუნველყოფს პროგრამის მაღალ წარმადობას, რამდენადაც მოცემული მოთხოვნის ანალიზი მხოლოდ ერთხელ ხდება და ამით მცირდება ტრაფიკი სერვერსა და კლიენტს შორის.

ტრიგერები (triggers) - სპეციალური შენახული პროცედურებია, რომელთა გააქტიურება განსაზღვრული მოვლენის მოხდენის დროს ხდება. მაგ. ცხრილში მონაცემების შეცვლა.

წარმოდგენები (views) - ვირტუალური ცხრილებია, რომლებიც წარმოიქმნება მოთხოვნების შესრულების შედეგად. ისინი საშუალებას გვაძლევენ ამორჩევის შედეგთან ვიმუშაოთ როგორც ცხრილთან;

ინდექსები (indexes) - სტრუქტურებია, რომლებიც ზრდიან მონაცემებთან მუშაობის მწარმოებლურობას;

მომხმარებლების ტიპები (user-defined data types) - მომხმარებლების მიერ შექმნილი მონაცემთა ტიპებია;

მომხმარებლების ფუნქციები (user-defined funqtions) - მომხმარებლების მიერ შექმნილი ფუნქციებია;

გასაღებები (keys) - მთლიანობის შეზღუდვის ერთ-ერთი სახეა, რომელიც უზრუნველყოფს მონაცემების მიმართვით მთლიანობას;

მთლიანობის შეზღუდვები (constraints) - ობიექტებია, რომლებიც უზრუნველყოფენ მონაცემების ლოგიკურ მთლიანობას და არ არსებობენ ცხრილებისაგან დამოუკიდებლად;

მნიშვნელობები გულისხმობის პრინციპით (defaults) - მონაცემთა ბაზის ობიექტებია, რომლებიც შეგვიძლია მივუთითოთ უშუალოდ ცხრილის სტრუქტურაში ან მომხმარებლების მიერ განსაზღვრულ ტიპებში.

მონაცემები სერვერზე მონაცემთა ბაზებში ინახება. მონაცემთა ბაზის სტრუქტურა შეიძლება ორი კუთხით განვიხილოთ: **ლოგიკური და ფიზიკური.**

მონაცემთა ბაზის **ლოგიკური სტრუქტურა** მოიცავს ცხრილების სტრუქტურას, მათ შორის კავშირებს, მომხმარებლებლების სიას, შენახულ პროცედურებს და მონაცემთა ბაზის სხვა ობიექტებს.

მონაცემთა ბაზის ფიზიკური სტრუქტურა მოიცავს მონაცემთა ბაზის ფაილებისა და ტრანზაქციების ჟურნალის აღწერას, მათ საწყის ზომას, ნაზრდის ზომას, მაქსიმალურ ზომას, კონფიგურირების პარამეტრებს და ა.შ.

SQL - ბრძანებების შესარულებლად გამოიყენება MySQL სერვერის მრავალრიცხოვანი კლიენტური დანართები, რომელთა შორის საყურადღებოა MySQL AB კომპანიის მიერ შემუშავებული უტილიტები: ბრძანებითი სტრიქონის უტილიტა mysql და გრაფიკული უტილიტა MySQL Query Browser. MySQL Query Browser -ის კომპონენტები თვალსაჩინოდ არიან წარმოდგენილი. მონაცემთა ბაზასთან მუშაობის პროცესში ამ უტილიტას დახმარებით

უშუალოდ არის შესაძლებელი მონაცემთა რედაქტირება (UPDATE ოპერატორის გარეშე), მოთხოვნებთან მუშაობა, მათი შენახვა ფაილში, შედეგების ექსპორტირება და მრავალი სხვა, მაგრამ მანამდე განვიხილოთ ბრძანებით სტრიქონში მუშაობის საკითხები.

MySQL მონაცემთა ბაზის სერვერი შეიცავს მონაცემთა ბაზების სიმრავლეს (ან სქემებს). თითოეული მონაცემთა ბაზა შედგება ერთი ან რამდენიმე ცხრილისაგან. ცხრილი შედგება ველებისა და ჩანაწერებისაგან.

SHOW DATABASES ბრძანების საშუალებით ჩვენ შეგვიძლია გამოვიტანოთ სერვერზე არსებული მონაცემთა ბაზები.

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
........
```

მონაცემთა ბაზები "MySQL", "INFORMATION_SCHEMA" და "performance_schema" სისტემური მონაცემთა ბაზებია, რომლებიც გამოიყენება MySQL-ში. მონაცემთა ბაზა "test" უზრუნველყოფს ინსტალაციის ტესტირების დროს.

განვიხილოთ მარტივი მაგალითი: მონაცემთა ბაზა პროდუქციის რეალიზაციისათვის. პროდუქციის რეალიზაციის მონაცემთა ბაზა, როგორც წესი შედგება მრავალი ცხრილისაგან, მაგ. პროდუქტების, კლიენტების, მომწოდებლების, შეკვეთების, გადახდების, თანამშრომლების და ა.შ. მონაცემთა ბაზას დავარქვათ სახელი „southwind„ და დავიწყოთ მონაცემთა ბაზის შექმნა ცხრილით „პროდუქტები“. დავუშვათ, რომ ჩვენი ცხრილი შედგება შემდეგი ველებისაგან (ცხრილი 22) და ჩანაწერებისაგან (ცხრილი 23):

ცხრილი 22 products ცხრილის ველები და შესაბამისი ტიპები

სვეტის დასახელება	მონაცემთა ტიპი
ProductID	INT
productCode	CHAR(3)
name	VARCHAR(30)
quantity	INT
price	DECIMAL(10,2)

ცხრილი 23 products ცხრილის ჩანაწერები

Database: southwind Table: products				
productID INT	productCode CHAR(3)	name VARCHAR(30)	quantity INT	price DECIMAL(10,2)
1001	PEN	Pen Red	5000	1.23
1002	PEN	Pen Blue	8000	1.25
1003	PEN	Pen Black	2000	1.25
1004	PEC	Pencil 2B	10000	0.48
1005	PEC	Pencil 2H	8000	0.49

მონაცემთა ბაზის შექმნა - CREATE DATABASE

მონაცემთა ბაზის შექმნისათვის სრულდება ბრძანება:

CREATE DATABASE <მონაცემთა ბაზის სახელი>;

```
mysql> CREATE DATABASE southwind;
Query OK, 1 row affected (0.03 sec)
```

შექმნილი მონაცემთა ბაზის დათვალიერება:

SHOW CREATE DATABASE

CREATE DATABASE ბრძანება ზოგიერთ მნიშვნელობას გულისხმობის პრინციპით ღებულობს. ჩვენ შეგვიძლია გამოვიყენოთ ბრძანება SHOW CREATE DATABASE <მონაცემთა ბაზის სახელი>, იმისათვის რომ დავათვალიეროთ ეს მნიშვნელობები.

```
mysql> CREATE DATABASE IF NOT EXISTS southwind;

mysql> SHOW CREATE DATABASE southwind \G
*************************** 1. row ***************************
       Database: southwind
Create Database: CREATE DATABASE `southwind` /*!40100 DEFAULT CHARACTER SET latin1 */
```

ცხრილის შექმნა

ჩვენ შეგვიძლია შევქმნათ მონაცემთა ბაზის ცხრილი CREATE TABLE <*ცხრილის სახელი*> ბრძანების გამოყენებით.

შევქმნათ ცხრილი “products” ჩვენი “southwind” მონაცემთა ბაზისათვის.

```
mysql> DROP DATABASE IF EXISTS southwind;
Query OK, 1 rows affected (0.31 sec)

-- Create the database "southwind"
mysql> CREATE DATABASE southwind;
Query OK, 1 row affected (0.01 sec)

-- Show all the databases in the server
--   to confirm that "southwind" database has been created.
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| southwind          |
| ......             |

-- Set "southwind" as the default database so as to reference its table directly.
mysql> USE southwind;
Database changed

-- Show the current (default) database
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| southwind  |
+------------+

-- Show all the tables in the current database.
-- "southwind" has no table (empty set).
```

```
-- Show all the tables in the current database.
-- "southwind" has no table (empty set).
mysql> SHOW TABLES;
Empty set (0.00 sec)

-- Create the table "products". Read "explanations" below for the column defintions
mysql> CREATE TABLE IF NOT EXISTS products (
         productID    INT UNSIGNED  NOT NULL AUTO_INCREMENT,
         productCode  CHAR(3)       NOT NULL DEFAULT '',
         name         VARCHAR(30)   NOT NULL DEFAULT '',
         quantity     INT UNSIGNED  NOT NULL DEFAULT 0,
         price        DECIMAL(7,2)  NOT NULL DEFAULT 99999.99,
         PRIMARY KEY  (productID)
       );
Query OK, 0 rows affected (0.08 sec)

-- Show all the tables to confirm that the "products" table has been created
mysql> SHOW TABLES;
+---------------------+
| Tables_in_southwind |
+---------------------+
| products            |
+---------------------+

-- Describe the fields (columns) of the "products" table
mysql> DESCRIBE products;
+-------------+------------------+------+-----+------------+----------------+
| Field       | Type             | Null | Key | Default    | Extra          |
+-------------+------------------+------+-----+------------+----------------+
| productID   | int(10) unsigned | NO   | PRI | NULL       | auto_increment |
| productCode | char(3)          | NO   |     |            |                |
| name        | varchar(30)      | NO   |     |            |                |
| quantity    | int(10) unsigned | NO   |     | 0          |                |
| price       | decimal(7,2)     | NO   |     | 99999.99   |                |
+-------------+------------------+------+-----+------------+----------------+

-- Show the complete CREATE TABLE statement used by MySQL to create this table
mysql> SHOW CREATE TABLE products \G
*************************** 1. row ***************************
       Table: products
Create Table:
CREATE TABLE `products` (
  `productID`    int(10) unsigned  NOT NULL AUTO_INCREMENT,
  `productCode`  char(3)           NOT NULL DEFAULT '',
  `name`         varchar(30)       NOT NULL DEFAULT '',
  `quantity`     int(10) unsigned  NOT NULL DEFAULT '0',
  `price`        decimal(7,2)      NOT NULL DEFAULT '99999.99',
  PRIMARY KEY (`productID`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

შესრულებული კოდის აღწერა:

- ჩვენ ვქმნით ცხრილში 5 ველს:

- productID, productCode, name, quantity და price.
- ველებში შერჩული გვაქვს მონაცემთა შემდეგი ტიპები:
- productID - INT UNSIGNED - არაუარყოფითი მთელი რიცხვები;
- productCode - CHAR(3) - ანბანურ-ციფრული სტრიქონი, ფიქსირებული 3 სიმბოლოთი.
- name - VARCHAR(30) - სტრიქონული ცვლადი, სიგრძით 30 სიმბოლომდე.
- productCode ცვლადისათვის ჩვენ ვიყენებთ ფიქსირებულ სიგრძეს, რამდენადაც ვთვლით, რომ productCode ცვლადი უნდა შეიცავდეს ზუსტად 3 სიმბოლოს.
- Quantity - INT UNSIGNED - არაუარყოფითი მთელი რიცხვები;
- Price - DECIMAL(10,2) - წილადი რიცხვი, წილად ნაწილში 2 ციფრის სიზუსტით.
- ატრიბუტი "NOT NULL" ნიშნავს, რომ სვეტი არ შეიძლება შეიცავდეს მნიშვნელობას NULL-ს.
- productID ველს ჩვენ ასევე მივანიჭეთ პირველადი გასაღები სვეტის თვისება, რომელიც უზრუნველყოფს იმას, რომ სვეტის თითოეული მნიშვნელობა იქნება უნიკალური.
- productID სვეტისთვის ასევე შევარჩიეთ AUTO_INCREMENT, საწყისი მნიშვნელობით 1.

ჩანაწერების შეტანა ცხრილში

INSERT INTO -ს სინტაქსი

```
INSERT INTO ცხრილის სახელი VALUES (პირველი სვეტის მნიშვნელობა, მეორე სვეტის მნიშვნელობა ..., ბოლო სვეტის მნიშვნელობა)... ყველა სვეტი
```

შესაძლებელია რამდენიმე სტრიქონის მნიშვნელობების შეტანა ერთი ბრძანებით:

```
INSERT INTO ცხრილის სახელი VALUES
    (პირველი სტრიქონის პირველი სვეტის მნიშვნელობა...,  პირველი სტრ. ბოლო სვეტის მნიშვნელობა),
    (მეორე სტრიქონის პირველი სვეტის მნიშვნელობა...,  მეორე სტრ. ბოლო სვეტის მნიშვნელობა),
    ...
```

შევიტანოთ მონაცემები ჩვენს ცხრილში. PRODUCTID სვეტის საწყის მნიშვნელობად განვსაზღვროთ 1001 და გამოვიყენოთ AUTO_INCREMENT დანარჩენი ჩანაწერებისათვის.

```
-- Insert a row with all the column values
mysql> INSERT INTO products VALUES (1001, 'PEN', 'Pen Red', 5000, 1.23);
Query OK, 1 row affected (0.04 sec)

-- Insert multiple rows in one command
-- Inserting NULL to the auto_increment column results in max_value + 1
mysql> INSERT INTO products VALUES
         (NULL, 'PEN', 'Pen Blue',  8000, 1.25),
         (NULL, 'PEN', 'Pen Black', 2000, 1.25);
Query OK, 2 rows affected (0.03 sec)
Records: 2  Duplicates: 0  Warnings: 0

-- Insert value to selected columns
-- Missing value for the auto_increment column also results in max_value + 1
mysql> INSERT INTO products (productCode, name, quantity, price) VALUES
         ('PEC', 'Pencil 2B', 10000, 0.48),
         ('PEC', 'Pencil 2H', 8000, 0.49);
Query OK, 2 row affected (0.03 sec)

-- Missing columns get their default values
mysql> INSERT INTO products (productCode, name) VALUES ('PEC', 'Pencil HB');
Query OK, 1 row affected (0.04 sec)

-- 2nd column (productCode) is defined to be NOT NULL
mysql> INSERT INTO products values (NULL, NULL, NULL, NULL, NULL);
ERROR 1048 (23000): Column 'productCode' cannot be null

-- Query the table
mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+------------+
| productID | productCode | name      | quantity | price      |
+-----------+-------------+-----------+----------+------------+
|      1001 | PEN         | Pen Red   |     5000 |       1.23 |
|      1002 | PEN         | Pen Blue  |     8000 |       1.25 |
|      1003 | PEN         | Pen Black |     2000 |       1.25 |
|      1004 | PEC         | Pencil 2B |    10000 |       0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |       0.49 |
|      1006 | PEC         | Pencil HB |        0 | 9999999.99 |
+-----------+-------------+-----------+----------+------------+
6 rows in set (0.02 sec)
```

წავშალოთ ბოლო სტრიქონი

```
-- Remove the last row
mysql> DELETE FROM products WHERE productID = 1006;
```

მოთხოვნები მონაცემთა ბაზისათვის - SELECT

მოთხოვნების შექმნა ყველაზე ხშირად გამოყენებული, ყველაზე მნიშვნელოვანი და რთული ამოცანაა:

SELECT ბრძანებას შემდეგი სინტაქსი გააჩნია:

```
-ყველა სვეტის შესაბამისი სტრიქონების სია
SELECT სვეტი1 სახელი, სვეტი2 სახელი,  ... FROM ცხრილის სახელი

--  ყველა სვეტის შესაბამისი სტრიქონების სია .  * წარმოადგენს ნიღაბს, რომელიც მიუთითებს ყველა ველს
SELECT * FROM ცხრილის სახელი

-- იმ სტრიქონების შესაბამისი სვეტის მონაცემების გამოტანა, რომლებიც აკმაყოფილებენ პირობას (WHERE-ში მითითებული პირობა)
SELECT სვეტი1 სახელი, სვეტი2 სახელი,  ... FROM ცხრილის სახელი WHERE პირობა
SELECT * FROM ცხრილისსახელი  WHERE პირობა
```

მაგალითად,

```
-- List all rows for the specified columns
mysql> SELECT name, price FROM products;
+-----------+-------+
| name      | price |
+-----------+-------+
| Pen Red   |  1.23 |
| Pen Blue  |  1.25 |
| Pen Black |  1.25 |
| Pencil 2B |  0.48 |
| Pencil 2H |  0.49 |
+-----------+-------+
5 rows in set (0.00 sec)

-- List all rows of ALL the columns. The wildcard * denotes ALL columns
mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.25 |
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.49 |
+-----------+-------------+-----------+----------+-------+
5 rows in set (0.00 sec)
```

შედარების ოპერატორები

INT, DECIMAL, FLOAT ტიპის მონაცემებისათვის შეგვიძლია გამოვიყენოთ შემდეგი შედარების ოპერატორები:

= - ტოლია;

< > ან != - არ უდრის;

>- მეტია;

<-ნაკლებია;

>= - მეტია ან ტოლია;

<=- ნაკლ;ებია ან ტოლია

მაგ. `price > 1.0`, `quantity <= 500`.

```
mysql> SELECT name, price FROM products WHERE price < 1.0;
+-----------+-------+
| name      | price |
+-----------+-------+
| Pencil 2B |  0.48 |
| Pencil 2H |  0.49 |
+-----------+-------+
2 rows in set (0.00 sec)

mysql> SELECT name, quantity FROM products WHERE quantity <= 2000;
+-----------+----------+
| name      | quantity |
+-----------+----------+
| Pen Black |     2000 |
+-----------+----------+
1 row in set (0.00 sec)
```

შედარების ოპერატორები ასევე გამოიყენება სტრიქონული ტიპის მნიშვნელობებისათვის. მაგ. `productCode = 'PEC'`

```
mysql> SELECT name, price FROM products WHERE productCode = 'PEN';
                                      -- String values are quoted
+-----------+-------+
| name      | price |
+-----------+-------+
| Pen Red   |  1.23 |
| Pen Blue  |  1.25 |
| Pen Black |  1.25 |
+-----------+-------+
3 rows in set (0.00 sec)
```

LIKE და NOT LIKE ინსტრუქცია

სტრიქონული ცვლადებისათვის „=“ და „<>“ ოპერატორების გარდა გამოიყენება ოპერატორი LIKE და OR LIKE ოპერატორი. სიმბოლო ‘-‘ აღნიშნავს ერთ ნებისმიერ სიმბოლოს, ხოლო სიმბოლო ‘%’ - მიუთითებს სიმბოლოთა ნებისმიერ რაოდენობას, მათ შორის ნულსაც.

მაგ.

```
-- "name" begins with 'PENCIL'
mysql> SELECT name, price FROM products WHERE name LIKE 'PENCIL%';
+-----------+-------+
| name      | price |
+-----------+-------+
| Pencil 2B |  0.48 |
| Pencil 2H |  0.49 |
+-----------+-------+

-- "name" begins with 'P', followed by any two characters,
--   followed by space, followed by zero or more characters
mysql> SELECT name, price FROM products WHERE name LIKE 'P__ %';
+-----------+-------+
| name      | price |
+-----------+-------+
| Pen Red   |  1.23 |
| Pen Blue  |  1.25 |
| Pen Black |  1.25 |
+-----------+-------+
```

არითმეტიკული ოპერატორები

ოპერატორი	აღწერა
+	შეკრება
-	გამოკლება
*	გამრავლება
/	გაყოფა
DIV	მთელრიცხვა გაყოფა
%	ნაშთის გამოყოფა

ლოგიკური ოპერატორები - AND (და), OR (ან) , NOT (უარყოფა), XOR

მაგ.

```
mysql> SELECT * FROM products WHERE quantity >= 5000 AND name LIKE 'Pen %';
+-----------+-------------+----------+----------+-------+
| productID | productCode | name     | quantity | price |
+-----------+-------------+----------+----------+-------+
|      1001 | PEN         | Pen Red  |     5000 |  1.23 |
|      1002 | PEN         | Pen Blue |     8000 |  1.25 |
+-----------+-------------+----------+----------+-------+

mysql> SELECT * FROM products WHERE quantity >= 5000 AND price < 1.24 AND name LIKE 'Pen %';
+-----------+-------------+---------+----------+-------+
| productID | productCode | name    | quantity | price |
+-----------+-------------+---------+----------+-------+
|      1001 | PEN         | Pen Red |     5000 |  1.23 |
+-----------+-------------+---------+----------+-------+

mysql> SELECT * FROM products WHERE NOT (quantity >= 5000 AND name LIKE 'Pen %');
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.49 |
+-----------+-------------+-----------+----------+-------+
```

ოპერატორები `BETWEEN` (მოთავსებულია მნიშვნელობებს შორის) , `NOT BETWEEN` (არ არის მოთავსებული მოცემულ მნიშვნელობებს შორის)

მაგ.

```
mysql> SELECT * FROM products
       WHERE (price BETWEEN 1.0 AND 2.0) AND (quantity BETWEEN 1000 AND 2000);
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
+-----------+-------------+-----------+----------+-------+
```

ოპერატორები `NULL` (გამოტოვებული მნიშვნელობა), `IS NOT NULL`(არარის ნული)

NULL არის სპეციფიური მნიშვნელობა, რომელიც მიუთითებს იმაზე, რომ მნიშვნელობა არ არსებობს ან გამოტოვებულია.

მაგ.

```
mysql> SELECT * FROM products WHERE productCode IS NULL;
Empty set (0.00 sec)
```

```
SELECT * FROM products WHERE productCode = NULL;
-- This is a common mistake. NULL cannot be compared.
```

ჩანაწერების დალაგება (სორტირება) ORDER BY

ORDER BY ბრძანების სინტაქსი შემდეგი სახისაა:

```
SELECT ... FROM ცხრილის სახელი
WHERE კრიტერიუმი
ORDER BY სვეტი A ASC|DESC, სვეტი B ASC|DESC, ...
```

ASC - ზრდადობის მიხედვით (ანბანის მიხედვით A-Z);

DESC - კლებადობისს მიხედვით (ანბანის უკუთანმიმდევრობით Z-A).

სვეტი A , სვეტი B – სვეტის დასახელებები, რომელთა მ,ნიშვნელობების სორტირებაც არის დაგეგმილი

მაგ.

```
-- Order the results by price in descending order
mysql> SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1002 | PEN         | Pen Blue  |     8000 |  1.25 |
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
+-----------+-------------+-----------+----------+-------+

-- Order by price in descending order, followed by quantity in ascending (default) order
mysql> SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC, quantity;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1003 | PEN         | Pen Black |     2000 |  1.25 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.25 |
|      1001 | PEN         | Pen Red   |     5000 |  1.23 |
+-----------+-------------+-----------+----------+-------+
```

მონაცემების ამოღება შემთხვევითი წესით - RAND()

```
mysql> SELECT * FROM products ORDER BY RAND();
```

ბრძანება LIMIT (ჩანაწერების ლიმიტირებული რაოდენობის გამოტანა)

LIMIT გამოიყენება განსაზღვრული რაოდენობის ჩანაწერების გამოსატანად,

მაგ. 2 სტრიქონის გამოტანა

```
mysql> SELECT * FROM products ORDER BY price LIMIT 2;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1004 | PEC         | Pencil 2B |    10000 |  0.48 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.49 |
+-----------+-------------+-----------+----------+-------+
```

4.3. მონაცემთა რედაქტირება - განახლება

მონაცემთა რედაქტირებისთვის გამოიყენება ბრძანება `UPDATE ... SET`, რომელსაც შემდეგი სინტაქსი გააჩნია:

```
UPDATE ცხრილის სახელი SET სვეტის სახელი = {მნიშვნელობა|NULL|DEFAULT}, ...
WHERE პირობა
```

მაგ.

```
-- Increase the price by 10% for all products
mysql> UPDATE products SET price = price * 1.1;

mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1001 | PEN         | Pen Red   |     5000 |  1.35 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.38 |
|      1003 | PEN         | Pen Black |     2000 |  1.38 |
|      1004 | PEC         | Pencil 2B |    10000 |  0.53 |
|      1005 | PEC         | Pencil 2H |     8000 |  0.54 |
+-----------+-------------+-----------+----------+-------+

-- Modify selected rows
mysql> UPDATE products SET quantity = quantity - 100 WHERE name = 'Pen Red';

mysql> SELECT * FROM products WHERE name = 'Pen Red';
+-----------+-------------+---------+----------+-------+
| productID | productCode | name    | quantity | price |
+-----------+-------------+---------+----------+-------+
|      1001 | PEN         | Pen Red |     4900 |  1.35 |
+-----------+-------------+---------+----------+-------+

-- You can modify more than one values
mysql> UPDATE products SET quantity = quantity + 50, price = 1.23 WHERE name = 'Pen Red';

mysql> SELECT * FROM products WHERE name = 'Pen Red';
+-----------+-------------+---------+----------+-------+
| productID | productCode | name    | quantity | price |
+-----------+-------------+---------+----------+-------+
|      1001 | PEN         | Pen Red |     4950 |  1.23 |
+-----------+-------------+---------+----------+-------+
```

სტრიქონის წაშლა - DELETE FROM

ბრძანების სინტაქსი:

```
DELETE FROM ცხრილის სახელი
-- ერთი სტრიქონის წაშლა , რომელიც აკმაყოფილებს where-ში მითითებულ პირობას
DELETE FROM ცხრილის სახელი  WHERE პირობა
```

მაგ.

```
mysql> DELETE FROM products WHERE name LIKE 'Pencil%';
Query OK, 2 row affected (0.00 sec)

mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1001 | PEN         | Pen Red   |     4950 |  1.23 |
|      1002 | PEN         | Pen Blue  |     8000 |  1.38 |
|      1003 | PEN         | Pen Black |     2000 |  1.38 |
+-----------+-------------+-----------+----------+-------+

-- Use this with extreme care, as the deleted records are irrecoverable!
mysql> DELETE FROM products;
Query OK, 3 rows affected (0.00 sec)

mysql> SELECT * FROM products;
Empty set (0.00 sec)
```

4.4. მონაცემების ექსპორტი და იმპორტი

ბრძანება `LOAD DATA LOCAL INFILE ... INTO TABLE ...`
მონაცემების იმპორტისათვის, შევქმნათ ტექსტური ფაილი შემდეგი ჩანაწერებით:

```
\N,PEC,Pencil 3B,500,0.52
\N,PEC,Pencil 4B,200,0.62
\N,PEC,Pencil 5B,100,0.73
\N,PEC,Pencil 6B,500,0.47
```

შევინახოთ ფაილი დასახელებით: `products_in.csv` კომპიუტერის მეხსიერებაში, მაგ. D დისკზე Myproject საქაღალდეში

განვახორციელოთ ამ მონაცემების იმპორტი products ცხრილში.

```
mysql> LOAD DATA LOCAL INFILE 'd:/myProject/products_in.csv' INTO TABLE products
         COLUMNS TERMINATED BY ','
         LINES TERMINATED BY '\r\n';
```

```
mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+
| productID | productCode | name      | quantity | price |
+-----------+-------------+-----------+----------+-------+
|      1007 | PEC         | Pencil 3B |      500 |  0.52 |
|      1008 | PEC         | Pencil 4B |      200 |  0.62 |
|      1009 | PEC         | Pencil 5B |      100 |  0.73 |
|      1010 | PEC         | Pencil 6B |      500 |  0.47 |
+-----------+-------------+-----------+----------+-------+
```

მონაცემების იმპორტი ტექსტურ ფაილში ხორციელდება ბრძანებით:
SELECT ... INTO OUTFILE ...

მაგ.

```
mysql> SELECT * FROM products INTO OUTFILE 'd:/myProject/products_out.csv'
         COLUMNS TERMINATED BY ','
         LINES TERMINATED BY '\r\n';
```

4.5. ცხრილის სტრუქტურის შეცვლა

ALTER TABLE. ცხრილის სტრუქტურის შეცვლა

ცხრილის სტრუქტურის შეცვლისათვის გამოიყენება ბრძანება ALTER TABLE

სვეტის დამატება: ALTER TABLE <ცხრილის სახელი> ADD <სვეტის სახელი> <სვეტის ტიპი> [<სვეტისთვისებები>] [FIRST ან AFTER <წინამორბედი სვეტის სახელი>];

მაგალითად, ALTER TABLE Products ADD id BIGINT AUTO_INCREMENT, ADD PRIMARY KEY (id);

• პირველადი გასაღების დამატება: ALTER TABLE <ცხრილის სახელი>> ADD [CONSTRAINT <გასაღების სახელი>] PRIMARY KEY (<სვეტების სია>);

მაგალითად, ALTER TABLE Orders ADD PRIMARY KEY (id);

• გარე გასაღების დამატება:

ALTER TABLE <ცხრილის სახელი> ADD [CONSTRAINT <გარე გასაღების სახელი>] FOREIGN KEY [<ინდექსის სახელი>] (<სვეტების სია>) REFERENCES <მშობელი ცხრილის სახელი> (<მშობელი ცხრილის პირველადი გასაღების სვეტების სია>) [<კავშირის მთლიანობის მხარდაჭერის წესები>];

მაგალითად, ALTER TABLE Orders ADD FOREIGN KEY (customer_id) REFERENCES Customers (id) ON DELETE RESTRICT ON UPDATE CASCADE);

• ჩვეულებრივი ინდექსის დამატება:

ALTER TABLE <ცხრილისსახელი> ADD INDEX [<ინდექსის სახელი>] (<სვეტების სია>);

უნიკალური ინდექსის დამატება:

ALTER TABLE <ცხრილის სახელი> ADD [CONSTRAINT <შეზღუდვის სახელი>] UNIQUE (<სვეტების სია>);

სრულტექსტიანი ინდექსის დამატება:

ALTER TABLE <ცხრილის სახელი> ADD FULLTEXT [<ინდექსის სახელი>] (<სვეტების სია>);
• ცხრილის სვეტის შეცვლა: ALTER TABLE <ცხრილის სახელი> CHANGE <სვეტის ძველი სახელი

> <სვეტის ახალი სახელი> <სვეტის ახალი ტიპი> [<სვეტის თვისებები>] [FIRST ან AFTER <წინამდებარე სვეტის სახელი>];

სვეტისათვის AUTO_INCREMENT თვისების მინიჭებისათვის აუცილებელია ან ერთდროულად ან წინასწარ შეიქმნას ამ სვეტის ინდექსი.

გადარქმევის გარეშე სვეტის აღწერის შეცვლა:

ALTER TABLE <ცხრილის სახელი> MODIFY <სვეტის სახელი> <სვეტის ახალი ტიპი> [<სვეტის თვისებები>] [FIRST ან AFTER <წინამდებარე სვეტის სახელი>];

სვეტისათვის მნიშვნელობის უსიტყვოდ დადგენა:

ALTER TABLE <ცხრილის სახელი> ALTER <სვეტის სახელი> SET DEFAULT <მნიშვნელობა უსიტყვოდ>;

მაგალითად, ALTER TABLE Products

ALTER warehouse SET DEFAULT 'Склад № 1';

სვეტისათვის მნიშვნელობის უსიტყვოდ ამოგდება:

ALTER TABLE <ცხრილის სახელი> ALTER <სვეტის სახელი> DROP DEFAULT; მაგალითად, ALTER TABLE Products ALTER warehouse DROP DEFAULT;

• ცხრილიდან სვეტის ამოდგება:

ALTER TABLE <ცხრილის სახელი> DROP <სვეტის სახელი>; მაგალითად, ALTER TABLE Products DROP warehouse;

• პირველადი გასაღების ამოგდება:

ALTER TABLE <ცხრილის სახელი> DROP PRIMARY KEY;

მაგალითად, ALTER TABLE Orders DROP PRIMARY KEY;

• გარე გასაღების ამოგდება: ALTER TABLE <ცხრილის სახელი> DROP FOREIGN KEY <გარე გასაღების სახელი>;

მაგალითად, ALTER TABLE Orders DROP FOREIGN KEY orders_ibfk_1;

• ინდექსის ამოგდება

ALTER TABLE <ცხრილის სახელი> DROP INDEX <ინდექსის სახელი>; მაგალითად, ALTER TABLE Customers DROP INDEX name;

• MyISAM ტიპის ცხრილში არაუნიკალური ინდექსების ჩართვა და გამორთვა:

ALTER TABLE <ცხრილის სახელი> DISABLE KEYS; ALTER TABLE <ცხრილის სახელი> ENABLE KEYS;

• ცხრილის სახელის გადარქმევა:

ALTER TABLE <ცხრილის სახელი> RENAME <ცხრილის ახალი სახელი>;

• ცხრილში სტრიქონების დალაგება:

ALTER TABLE <ცხრილის სახელი> ORDER BY <1 სვეტის სახელი> [ASC ან DESC], [<2 სვეტის სახელი2> [ASC ან DESC],...];

InnoDB ტიპის ცხრილებში, რომლებსაც გააჩნიათ პირველადი გასაღები ან უნიკალური ინდექსი, დაუშვებელია მნიშვნელობა NOT NULL UNIQUE.

• ცხრილის ოპციური თვისებების მოცემა ხდება ბრძანებით: ALTER TABLE <ცხრილის სახელი> <ცხრილის ოპციური თვისება>;

მაგალითად, ცხრილის ტიპის შეცვლა შესაძლებელია ბრძანებით: ALTER TABLE <ცხრილის სახელი> ENGINE <ცხრილის ახალი ტიპი>;

• კოდირებისა და სიმბოლური მნიშვნელობების შედარების წესის ცვლილებისათვის გამოიყენება შემდეგი ბრძანება: ALTER TABLE <ცხრილის სახელი> CHARACTER SET <კოდირების სახელი> [COLLATE <შედარების წესის სახელი>];

მაგალითად, ALTER TABLE Products CHARACTER SET cp1251;

4.6.1. რელაციური კავშირი ერთი - მრავალთან

ცხრილებს შორის რეალციური კავშირის დასამყარებლად შევქმნათ ჩვენი მონაცემთა ბაზისათვის მეორე ცხრილი suppliers, ცხრილი 23-ში მითითებული ველებით და შესაბამისი ტიპებით, ხოლო ცხრილს products დავამატოთ სვეტი supplerID (ცხრილი 24)

ცხრილი 24

Database: southwind Table: suppliers		
supplierID INT	**name VARCHAR(3)**	**phone CHAR(8)**
501	ABC Traders	88881111
502	XYZ Company	88882222
503	QQ Corp	88883333

ცხრილი 25

Database: southwind Table: products					
productID INT	**product Code CHAR(3)**	**name VARCHAR(30)**	**quantity INT**	**price DECIMAL(10,2)**	**supplierID INT (Foreign Key)**
2001	PEC	Pencil 3B	500	0.52	**501**
2002	PEC	Pencil 4B	200	0.62	**501**
2003	PEC	Pencil 5B	100	0.73	**501**
2004	PEC	Pencil 6B	500	0.47	**502**

```
mysql> USE southwind;

mysql> DROP TABLE IF EXISTS suppliers;

mysql> CREATE TABLE suppliers (
         supplierID  INT UNSIGNED  NOT NULL AUTO_INCREMENT,
         name        VARCHAR(30)   NOT NULL DEFAULT '',
         phone       CHAR(8)       NOT NULL DEFAULT '',
         PRIMARY KEY (supplierID)
       );

mysql> DESCRIBE suppliers;
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| supplierID | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| name       | varchar(30)      | NO   |     |         |                |
| phone      | char(8)          | NO   |     |         |                |
+------------+------------------+------+-----+---------+----------------+

mysql> INSERT INTO suppliers VALUE
          (501, 'ABC Traders', '88881111'),
          (502, 'XYZ Company', '88882222'),
          (503, 'QQ Corp', '88883333');

mysql> SELECT * FROM suppliers;
+------------+-------------+----------+
| supplierID | name        | phone    |
+------------+-------------+----------+
|        501 | ABC Traders | 88881111 |
|        502 | XYZ Company | 88882222 |
|        503 | QQ Corp     | 88883333 |
+------------+-------------+----------+
```

ჩვენს ცხრილს დავამატოთ სვეტი supplierID

```
mysql> ALTER TABLE products
       ADD COLUMN supplierID INT UNSIGNED NOT NULL;
Query OK, 4 rows affected (0.13 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> DESCRIBE products;
+-------------+------------------+------+-----+------------+----------------+
| Field       | Type             | Null | Key | Default    | Extra          |
+-------------+------------------+------+-----+------------+----------------+
| productID   | int(10) unsigned | NO   | PRI | NULL       | auto_increment |
| productCode | char(3)          | NO   |     |            |                |
| name        | varchar(30)      | NO   |     |            |                |
| quantity    | int(10) unsigned | NO   |     | 0          |                |
| price       | decimal(10,2)    | NO   |     | 9999999.99 |                |
| supplierID  | int(10) unsigned | NO   |     | NULL       |                |
+-------------+------------------+------+-----+------------+----------------+
```

რელაციური კავშირისათვის შევქმნათ გარე გასაღები supplerID სვეტისათვის

```
mysql> UPDATE products SET supplierID = 501;

-- Add a foreign key constrain
mysql> ALTER TABLE products
       ADD FOREIGN KEY (supplierID) REFERENCES suppliers (supplierID);

mysql> DESCRIBE products;
+-------------+------------------+------+-----+------------+----------------+
| Field       | Type             | Null | Key | Default    | Extra          |
+-------------+------------------+------+-----+------------+----------------+
  ......
| supplierID  | int(10) unsigned | NO   | MUL |            |                |
+-------------+------------------+------+-----+------------+----------------+

mysql> UPDATE products SET supplierID = 502 WHERE productID  = 2004;
  -- Choose a valid productID
```

```
mysql> SELECT * FROM products;
+-----------+-------------+-----------+----------+-------+------------+
| productID | productCode | name      | quantity | price | supplierID |
+-----------+-------------+-----------+----------+-------+------------+
|      2001 | PEC         | Pencil 3B |      500 |  0.52 |        501 |
|      2002 | PEC         | Pencil 4B |      200 |  0.62 |        501 |
|      2003 | PEC         | Pencil 5B |      100 |  0.73 |        501 |
|      2004 | PEC         | Pencil 6B |      500 |  0.47 |        502 |
+-----------+-------------+-----------+----------+-------+------------+
```

ამორჩევა SELECT … JOIN

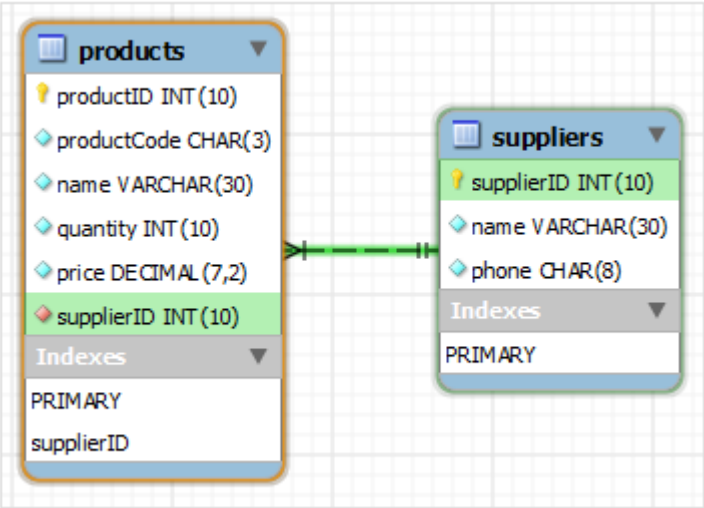
SELECT ბრძანებით შესაძლებელია მივიღოთ მოთხოვნა ორი ურთიერთდაკავშირებული ცხრილის საფუძველზე. მაგ. იმისათვის, რომ მივიღოთ მოთხოვნა, რომელშიც მოცემული იქნება პროდუქტის დასახელებები (ცხრილი products) და შესაბამისი მომწოდებლების სახელები (ცხრილი suppliers), ჩვენ ეს ცხრილები საერთო სვეტის SupplierID-ის მიხედვით უნდა დავაკავშიროთ.

```
mysql> SELECT products.name, price, suppliers.name
       FROM products
          JOIN suppliers ON products.supplierID = suppliers.supplierID
       WHERE price < 0.6;
+-----------+-------+-------------+
| name      | price | name        |
+-----------+-------+-------------+
| Pencil 3B |  0.52 | ABC Traders |
| Pencil 6B |  0.47 | XYZ Company |
+-----------+-------+-------------+
```

```
mysql> SELECT products.name, price, suppliers.name
       FROM products, suppliers
       WHERE products.supplierID = suppliers.supplierID
          AND price < 0.6;
+-----------+-------+-------------+
| name      | price | name        |
+-----------+-------+-------------+
| Pencil 3B |  0.52 | ABC Traders |
| Pencil 6B |  0.47 | XYZ Company |
+-----------+-------+-------------+
```

ან შემდეგი სახით:

```
mysql> SELECT products.name AS `Product Name`, price, suppliers.name AS `Supplier Name`
       FROM products
          JOIN suppliers ON products.supplierID = suppliers.supplierID
       WHERE price < 0.6;
+--------------+-------+---------------+
| Product Name | price | Supplier Name |
+--------------+-------+---------------+
| Pencil 3B    |  0.52 | ABC Traders   |
| Pencil 6B    |  0.47 | XYZ Company   |
+--------------+-------+---------------+

-- Use aliases for table names too
mysql> SELECT p.name AS `Product Name`, p.price, s.name AS `Supplier Name`
       FROM products AS p
          JOIN suppliers AS s ON p.supplierID = s.supplierID
       WHERE p.price < 0.6;
```



ნახ. 221 კავშირი ერთი-მრავალთან

4.6.2. რელაციური კავშირი მრავალი - მრავალთან

დავუშვათ, რომ პროდუქციას გააჩნია რამდენიმე მომწოდებელი, ხოლო მომწოდებელი გვაწვდის რამდენიმე სახეობის პროდუქციას. ამ შემთხვევაში კავშირი „მრავალი-მრავალთან“ უნდა განხორციელდეს. აქ ჩვენ არ შეგვიძლია SupplerID სვეტი ჩავსვათ products ცხრილში, რამდენადაც ჩვენ ვერ განვსაზღვრავთ მომწოდებლების რაოდენობას, ასევე მომწოდებლების ცხრილში ვერ ჩავსვამთ productid ველს, იმის გამო, რომ ვერც ამ შემთხვევაში ვერ განვსაზღვრავთ პროდუქციის რაოდენობას.

ამ პრობლემის გადასაჭრელად, აუცილებელია შეიქმნას ახალი ცხრილი, როგორც მაკავშირებელი ცხრილი. ასეთ ცხრილს ჩვენი ამოცანისათვის დავარქვათ products_suppliers.

ჩვენი მონაცემთა ბაზის ცხრილებს უნდა ჰქონდეთ ცხრილი 26 , ცხრილი 27 და ცხრილი 28 -ში მოცემული სვეტების შესაბამისი სტრუქტურა.

ცხრილი 26

Database: southwind Table: products_suppliers	
productID INT (Foreign Key)	supplierID INT (Foreign Key)
2001	501
2002	501
2003	501
2004	502
2001	503

ცხრილი 27

Database: southwind Table: suppliers		
supplierID **INT**	**name** **VARCHAR(30)**	**phone** **CHAR(8)**
501	ABC Traders	88881111
502	XYZ Company	88882222
503	QQ Corp	88883333

ცხრილი 28

atabase: southwind Table: products				
productID **INT**	**productCode** **CHAR(3)**	**name** **VARCHAR(30)**	**quantity** **INT**	**price** **DECIMAL(10,2)**
2001	PEC	Pencil 3B	500	0.52
2002	PEC	Pencil 4B	200	0.62
2003	PEC	Pencil 5B	100	0.73
2004	PEC	Pencil 6B	500	0.47

შევქმნათ ცხრილი products_suppliers. ცხრილის ორ ველს გააჩნია პირველადი გასაღები სვეტის თვისება productID და supplierID, მათი კომბინაცია მოახდენს თითოეული სტრიქონის იდენტიფიცირებას. ორი გარე გასაღები განისაზღვრება იმისათვის, რომ დამყარდეს კავშირი ორ მშობელ ცხრილთან.

```
mysql> CREATE TABLE products_suppliers (
         productID   INT UNSIGNED  NOT NULL,
         supplierID  INT UNSIGNED  NOT NULL,
                     -- Same data types as the parent tables
         PRIMARY KEY (productID, supplierID),
                     -- uniqueness
         FOREIGN KEY (productID)  REFERENCES products  (productID),
         FOREIGN KEY (supplierID) REFERENCES suppliers (supplierID)
       );

mysql> DESCRIBE products_suppliers;
+------------+------------------+------+-----+---------+-------+
| Field      | Type             | Null | Key | Default | Extra |
+------------+------------------+------+-----+---------+-------+
| productID  | int(10) unsigned | NO   | PRI | NULL    |       |
| supplierID | int(10) unsigned | NO   | PRI | NULL    |       |
+------------+------------------+------+-----+---------+-------+

mysql> INSERT INTO products_suppliers VALUES (2001, 501), (2002, 501),
       (2003, 501), (2004, 502), (2001, 503);
-- Values in the foreign-key columns (of the child table) must match
--   valid values in the columns they reference (of the parent table)

mysql> SELECT * FROM products_suppliers;
+-----------+------------+
| productID | supplierID |
+-----------+------------+
|      2001 |        501 |
|      2002 |        501 |
|      2003 |        501 |
|      2004 |        502 |
|      2001 |        503 |
+-----------+------------+
```

წავშალოთ სვეტი SupplerID products ცხრილიდან (ეს სვეტი დამატებული იყო იმისათვის, რომ მიგვეღო რელაციური კავშირი ერთი-მრავალთან. მრავალი-მრავალთან კავშირის უზრუნველსაყოფად ის საჭირო არ არის) .

ამ სვეტის წაშლამდე საჭიროა მოიხსნას გარე გასაღები სვეტის თვისება, რომელიც ამ სვეტს გააჩნია. MYSQL-ში ამისათვის უნდა ვიცოდეთ შეზღუდვების სახელი, რომელიც გენერირებულ იქნა სისტემის მიერ. შეზღუდვების სახელის დასადგენად გამოვიყენოთ ინსტრუქცია SHOW CREATE TABLE products და მივიღოთ ინფორმაცია გარე გასაღების შეზღუდვების შესახებ წინადადებაში „CONSTRAINT constraint_name FOREIGN KEY ....", გასაღები სვეტის თვისების მოხსნა ასევე შეგვიძლია ინსტრუქციით: "ALTER TABLE products DROP FOREIGN KEY constraint_name".

```
mysql> SHOW CREATE TABLE products \G
Create Table: CREATE TABLE `products` (
  `productID`   int(10) unsigned  NOT NULL AUTO_INCREMENT,
  `productCode` char(3)           NOT NULL DEFAULT '',
  `name`        varchar(30)       NOT NULL DEFAULT '',
  `quantity`    int(10) unsigned  NOT NULL DEFAULT '0',
  `price`       decimal(7,2)      NOT NULL DEFAULT '99999.99',
  `supplierID`  int(10) unsigned   NOT NULL DEFAULT '501',
  PRIMARY KEY (`productID`),
  KEY `supplierID` (`supplierID`),
  CONSTRAINT `products_ibfk_1` FOREIGN KEY (`supplierID`)
     REFERENCES `suppliers` (`supplierID`)
) ENGINE=InnoDB AUTO_INCREMENT=1006 DEFAULT CHARSET=latin1

mysql> ALTER TABLE products DROP FOREIGN KEY products_ibfk_1;

mysql> SHOW CREATE TABLE products \G
```

ეხლა უკვე შეგვიძლია SupplierID სვეტის წაშლა :

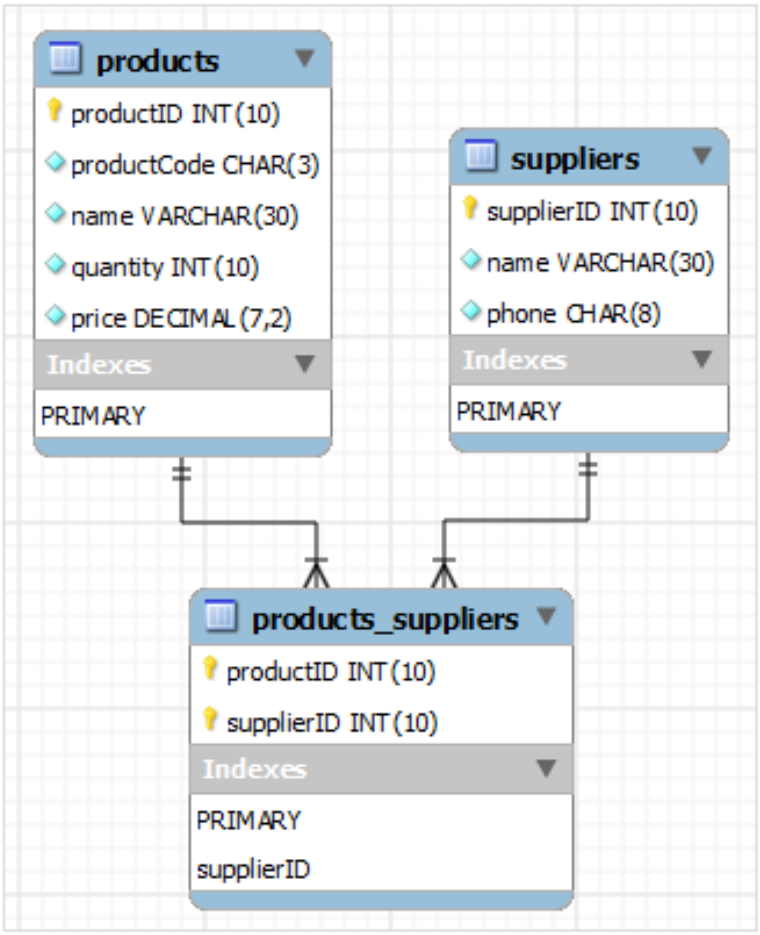
```
mysql> ALTER TABLE products DROP supplierID;

mysql> DESC products;
```

მოთხოვნები

სამი ცხრილის ბაზაზე მოთხოვნების მიღება ანალოგიურად შევიძლია SELECT with JOIN ინსტრუქციის გამოყენებით.

```
mysql> SELECT products.name AS `Product Name`, price, suppliers.name AS `Supplier Name`
       FROM products_suppliers
          JOIN products  ON products_suppliers.productID = products.productID
          JOIN suppliers ON products_suppliers.supplierID = suppliers.supplierID
       WHERE price < 0.6;
+--------------+-------+---------------+
| Product Name | price | Supplier Name |
+--------------+-------+---------------+
| Pencil 3B    |  0.52 | ABC Traders   |
| Pencil 3B    |  0.52 | QQ Corp       |
| Pencil 6B    |  0.47 | XYZ Company   |
+--------------+-------+---------------+

-- Define aliases for tablenames too
mysql> SELECT p.name AS `Product Name`, s.name AS `Supplier Name`
       FROM products_suppliers AS ps
          JOIN products AS p ON ps.productID = p.productID
          JOIN suppliers AS s ON ps.supplierID = s.supplierID
       WHERE p.name = 'Pencil 3B';
+--------------+---------------+
| Product Name | Supplier Name |
+--------------+---------------+
| Pencil 3B    | ABC Traders   |
| Pencil 3B    | QQ Corp       |
+--------------+---------------+

-- Using WHERE clause to join (legacy and not recommended)
mysql> SELECT p.name AS `Product Name`, s.name AS `Supplier Name`
       FROM products AS p, products_suppliers AS ps, suppliers AS s
       WHERE p.productID = ps.productID
          AND ps.supplierID = s.supplierID
          AND s.name = 'ABC Traders';
+--------------+---------------+
| Product Name | Supplier Name |
+--------------+---------------+
| Pencil 3B    | ABC Traders   |
| Pencil 4B    | ABC Traders   |
| Pencil 5B    | ABC Traders   |
+--------------+---------------+
```

ნახ.222-ზე მოცემულია დიაგრამა, რომელიც თვალსაჩინოდ ასახავს ცხრილებს შორის კავშირს.



ნახ. 222

წარმოდგენები

წარმოდგენა, როგორც ვიცით, ეს არის ვირტუალური ცხრილი, რომელიც არ შეიცავს არანაირ ფიზიკურ მონაცემს. ის გამოიყენება, როგორც მონაცემების დათვალიერების ერთ-ერთი ალტერნატიული საშუალება.

```
-- Define a VIEW called supplier_view from products, suppliers and products_suppliers tables
mysql> CREATE VIEW supplier_view
       AS
       SELECT suppliers.name as `Supplier Name`, products.name as `Product Name`
       FROM products
          JOIN suppliers ON products.productID = products_suppliers.productID
          JOIN products_suppliers ON suppliers.supplierID = products_suppliers.supplierID;

-- You can treat the VIEW defined like a normal table
mysql> SELECT * FROM supplier_view;
+---------------+--------------+
| Supplier Name | Product Name |
+---------------+--------------+
| ABC Traders   | Pencil 3B    |
| ABC Traders   | Pencil 4B    |
| ABC Traders   | Pencil 5B    |
| XYZ Company   | Pencil 6B    |
+---------------+--------------+

mysql> SELECT * FROM supplier_view WHERE `Supplier Name` LIKE 'ABC%';
+---------------+--------------+
| Supplier Name | Product Name |
+---------------+--------------+
| ABC Traders   | Pencil 3B    |
| ABC Traders   | Pencil 4B    |
| ABC Traders   | Pencil 5B    |
+---------------+--------------+
```

```
mysql> DROP VIEW IF EXISTS patient_view;

mysql> CREATE VIEW patient_view
       AS
       SELECT
          patientID AS ID,
          name AS Name,
          dateOfBirth AS DOB,
          TIMESTAMPDIFF(YEAR, dateOfBirth, NOW()) AS Age
       FROM patients
       ORDER BY Age, DOB;

mysql> SELECT * FROM patient_view WHERE Name LIKE 'A%';
+------+---------+------------+------+
| ID   | Name    | DOB        | Age  |
+------+---------+------------+------+
| 1003 | Ali     | 2011-01-30 |    1 |
| 1001 | Ah Teck | 1991-12-31 |   20 |
+------+---------+------------+------+

mysql> SELECT * FROM patient_view WHERE age >= 18;
+------+---------+------------+------+
| ID   | Name    | DOB        | Age  |
+------+---------+------------+------+
| 1001 | Ah Teck | 1991-12-31 |   20 |
+------+---------+------------+------+
```

```
mysql> CREATE TABLE accounts (
          name     VARCHAR(30),
          balance  DECIMAL(10,2)
       );

mysql> INSERT INTO accounts VALUES ('Paul', 1000), ('Peter', 2000);
mysql> SELECT * FROM accounts;
+-------+---------+
| name  | balance |
+-------+---------+
| Paul  | 1000.00 |
| Peter | 2000.00 |
+-------+---------+

-- Transfer money from one account to another account
mysql> START TRANSACTION;
mysql> UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';
mysql> UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';
mysql> COMMIT;     -- Commit the transaction and end transaction
mysql> SELECT * FROM accounts;
+-------+---------+
| name  | balance |
+-------+---------+
| Paul  |  900.00 |
| Peter | 2100.00 |
+-------+---------+

mysql> START TRANSACTION;
mysql> UPDATE accounts SET balance = balance - 100 WHERE name = 'Paul';
mysql> UPDATE accounts SET balance = balance + 100 WHERE name = 'Peter';
mysql> ROLLBACK;    -- Discard all changes of this transaction and end Transaction
mysql> SELECT * FROM accounts;
+-------+---------+
| name  | balance |
+-------+---------+
| Paul  |  900.00 |
| Peter | 2100.00 |
+-------+---------+
```

დავალება

დავალება 1

- ***შექმენით მონაცემთა ბაზა სახელით: dbstud_Lessons;***
  - შექმენით ცხრილი: Student მითითებული ველებით:
    - kodStud - სტუდენტის კოდი - მთელი ტიპის მონაცემი;
    - gvari - სტუდენტის გვარი - სტრიქონული ცვლადი, მაქსიმუმ 20 სიმბოლო;
    - saxeli - სტუდენტის სახელი - მაქსიმუმ 15 სიმბოლო;
    - gr_nomber - ჯგუფის ნომერი - მთელი ტიპის ცვლადი
    - kodStud-ს მიანიჭეთ გასაღები სვეტის თვისება;

  - ***შექმენით ცხრილი Ped, გასაღები სვეტით Kodped და ველებით***
    - Gvped - მასწავლებლის გვარი - სტრიქონული ცვლადი, მაქსიმუმ 20 სიმბოლო;
    - SaxPed -- მასწავლებლის სახელი - სტრიქონული ცვლადი, მაქსიმუმ 15 სიმბოლო;
    - Kaf, tanam, xarisxi - კათედრა, თანამდებობა, სამეცნიერო ხარისხი - სტრიქონული ცვლადი, მაქსიმუმ 20 სიმბოლო;

  - ***შექმენით ცხრილი sagani, გასაღები სვეტით Kodsag და ველებით***
    - Sag_sax - საგნის დასახელება- სტრიქონული ცვლადი, მაქსიმუმ 20 სიმბოლო;
    - Hours - დრო - მთელი ტიპის ცვლადი;
    - Gamocda - გამოცდის მაქსიმალური შეფასება - მთელი ტიპის ცვლადი;
    - Semestr - სემესტრი - მთელი ტიპის ცვლადი;
    - Kodped - მასწავლებლის კოდი - მთელი ტიპის ცვლადი;

შეიტანეთ მონაცემები ცხრილებში
ა) შეიტანეთ ბრძანების გამოყენებით რამდენიმე ჩანაწერი;
ბ) შექმენით ტექსტური ფაილი, შეიტანეთ შესაბამისი ჩანაწერები და გადმოიტანეთ ფაილიდან ცხრილებში;

Student ცხრილს დაამატეთ ახალი სვეტი address - მისამართი

მოთხოვნები

- გამოიტანეთ ყველა მონაცემი Student ცხრილიდან;
- გამოიტანეთ Student ცხრილიდან მხოლოდ gvari და saxeli სვეტის ჩანაწერები;
- გამოიტანეთ Student ცხრილიდან gvari და saxeli სვეტის ჩანაწერები სორტირებული ანბანის მიხედვით;

- გამოიტანეთ Student ცხრილიდან მხოლოდ პირველი ორი ჩანაწერი;
- გამოიტანეთ Student ცხრილიდან მხოლოდ ის ჩანაწერები, რომელთა სახელიც იწყება „G” ასოზე;
- გამოიტანეთ Student ცხრილიდან მხოლოდ ის ჩანაწერები, რომელთა ჯგუფის ნომერია 2 ან 3;

დავალება 2.

- Company ცხრილი შეიცავს ინფორმაციას სარეკლამო კომპანიების შესახებ.
  - კერძოდ: კომპანიის დასახელებას, ანგარიშის ნომერს, ვებ-გვერდის მისამართს, On_Line შეკვეთების შესაძლებლობა/შეუძლებლობას;
- Advert ცხრილში აღწერილია სარეკლამო კომპანიების მიერ მიღებული შეკვეთების შესახებ ინფორმაცია.
  - კერძოდ: რეკლამის კატეგორია, შეკვეთის თარიღი, ფასი და შესრულების ვადა.

1. შექმენით ცხრილი - Company, შემდეგი ველებით, ველების ტიპებითა და აღწერით:

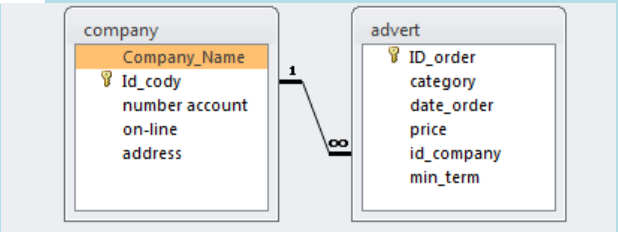
Field Name	Data Type	Description
Company Name	ტექსტური	კომპანიის დასახელება
Number_account	ტექსტური	ანგარიშის ნომერი
On_line	ლოგიკური	
Id_Cody	მთელი რიხვი	
Address	ლინკი	მისამართი

2. ველს Id_Cody განუსაზღვრეთ პირველადი გასაღები სვეტის თვისება.

3. შექმენით ცხრილი - Advert, შემდეგი ველებით, ველების ტიპებითა და აღწერით:

Field Name	Data Type	Description
ID_order	მთელი რიცხვი	
Category	ტექსტი	რეკლამის ტიპი
Date_Order	თარიღი	შეკვეთის თარიღი
Price	ნამდვილი რიცხვი	ფასი
Id_Company	მთელი რიცხვი	
Min_term	მთელი რიცხვი	შესრულების მინიმალური ვადა

განახორციელეთ რელაციური კავშირი ერთი-მრავალთან ნახაზის შესაბამისად.



ცხრილის ველები შეავსეთ შემდეგი ჩანაწერებით

Company_N	Id_cody	number acc	on-line	address
NEW LINE	25	QW342 567	☑	tbilisi
GZA	26	LP456 378	☐	rustavi
FIRST	27	QD456 878	☑	rustavi
SARKE	34	NK456 342	☑	Tbilisi
STAR	98	RT267 123	☑	tbilisi
			☐	

ID_order	category	date_order	price	id_company	min_term
1	internet	18-მარ-13	345,00 Lari	sarke	15
2	radio	06-მარ-13	125,00 Lari	new line	14
3	television	08-მარ-13	350,00 Lari	first	10
4	outdoor	12-მარ-13	348,00 Lari	gza	16
5	television	05-მარ-13	624,00 Lari	first	19
6	radio	18-მარ-13	683,00 Lari	sarke	20
7	radio	20-მარ-13	2 345,00 Lari	sarke	19
8	radio	21-მარ-13	2 300,00 Lari	gza	15
9	radio	13-მარ-13	500,00 Lari	first	15
10	television	16-მარ-13	450,00 Lari	first	18

4. შექმენით მოთხოვნა advert ცხრილის საფუძველზე, რომელშიც გამოტანილი იქნება რეკლამის ტიპი, შეკვეთის თარიღი და ფასი.

   შექმენით ახალი გამოთვლადი სვეტი, რომელიც დაადგენს (გამოითვლის) შესრულების ახალ ვადებს - შესრულების მინიმალური ვადა გაზარდეთ 5-ით. ველს დაარქვით სახელი: NewTerm;
   მიღებულ შედეგები Table-ში წარმოადგინეთ ფულადი ფორმატით.
   მოთხოვნა შეინახეთ სახელით: Query_Term

5. შექმენით მოთხოვნა, რომელშიც მოცემული იქნება შემდეგი ინფორმაცია: კომპანიის დასახელება, მისამართი, შეკვეთის ორდერი, ფასი და შეკვეთის ვადა.

   გამოიტანეთ მხოლოდ ის მონაცემები, რომლებიც აკმაყოფილებენ შემდეგ პირობებს: შეკვეთის თარიღი ნაკლებია 20 მარტზე და შეკვეთის ხანგრძლივობა მეტია 10-ზე;
   მოთხოვნა შეინახეთ სახელით: Query_date_term

5. პარალელური დაპროგრამება

პარალელური დაპროგრამება თანამედროვე დაპროგრამების ერთ-ერთი მნიშვნელოვანი შემადგენელი ნაწილია. ვინაიდან ის გარკვეულწილად ითვალისწინებს განსაკუთრებულ სიტუაციებს (გამონაკლისებს), ამიტომ წინამდებარე სახელმძღვანელოში აღნიშნული საკითხის განხილვა მიზანშეწონილად ჩავთვალეთ.

5.1. განსაკუთრებული სიტუაციების (გამონაკლისების) დამუშავება

გამონაკლისი არასტანდარტული სიტუაციაა, რომელიც კოდის შესრულების დროს წარმოიქმნება. სხვა სიტყვებით, რომ ვთქვათ, გამონაკლისი შეცდომაა, რომელიც პროგრამის შესრულების დროს იჩენს თავს. დაპროგრამების ენებში, სადაც გამონაკლისების, ანუ განსაკუთრებული სიტუაციების დამუშავება ავტომატურად არ ხდება, შეცდომები „ხელით“ უნდა შესწორდეს პროგრამისტის მიერ, რაც საკმაოდ დამღლელი და შრომატევადი პროცესია. გამონაკლისების დამუშავების თვალსაზრისით, Java ამ პრობლემებისგან გვათავისუფლებს და მათი მართვა ობიექტზე ორიენტირებულ სამყაროში გადააქვს.

5.1.1. განსაკუთრებული სიტუაციების დამუშავების საფუძვლები

Java–ში გამონაკლისი არის ობიექტი, რომელიც იმ გამონაკლის (შეცდომითი ხასიათის) სიტუაციას აღწერს, რაც პროგრამული კოდის ამა თუ იმ ფრაგმენტში წარმოიქმნება. ასეთი სიტუაციის შექმნის დროს, შეცდომის გამომწვევ მეთოდში იქმნება და გადაიცემა ობიექტი, რომელიც გამონაკლისს წარმოადგენს. ასეთ შემთხვევაში, მეთოდი ან თავად, დამოუკიდებლად ახდენს გამონაკლისის დამუშავებას ან უბრალოდ გაატრებს მას. ორივე შემთხვევაში, გამონაკლისი მაინც იქნება გარკვეულ დროში აღმოჩენილი და დამუშავებული.

განსაკუთრებული სიტუაციების დამუშავება Java-ში ხუთი საკვანძო სიტყვით იმართება: **try, catch, throw, throws** და **finally**. მათი მუშაობის ზოგადი პრინციპი შემდეგია. პროგრამის ოპერატორები, რომელთა შემოწმება გვსურს გამონაკლისზე, თავსდება **try** ბლოკში. თუ აღნიშნულ ბლოკში ადგილი აქვს განსაკუთრებულ სიტუაციას, ის გამონაკლისის სახით იქმნება და გადაიცემა. ჩვენს პროგრამულ კოდს შეუძლია ამ გამონაკლისის დაჭერა (**catch** ბლოკის გამოყენებით) და გარკვეული საშუალებით დამუშავება. სისტემური გამონაკლისები ავტომატურად გადაეცემა Java-ს შესრულების დროის სისტემას, ხოლო გამონაკლისის „ხელით“ გადასაცემად **throw** საკვანძო სიტყვა გამოიყენება. ნებისმიერი გამონაკლისი, რომელიც იქმნება და გადაიცემა მეთოდში, მის ინტერფეისში უნდა იქნას მითითებული **throws** საკვანძო სიტყვის გამოყენებით. ნებისმიერი კოდი, რომელიც აუცილებლად უნდა შესრულდეს **try** ბლოკის დასრულების შემდეგ, თავსდება **finally** ბლოკში.

ქვემოთ წარმოდგენილია განსაკუთრებული სიტუაციების (გამონაკლისების) დამუშავების ბლოკის ზოგადი ფორმა:

```
try {
// კოდის ბლოკი, რომელშიც გამონაკლისი მოჭმდება
}
catch(გამონაკლისის_ტიპი1 exOb ){
  //გამონაკლისის_ტიპი1 ტიპის გამონაკლისების დამუშავება
}
catch(გამონაკლისის_ტიპი2 exOb ){
  //გამონაკლისის_ტიპი2 ტიპის გამონაკლისების დამუშავება
}
//...
finally {
// კოდის ბლოკი, რომელიც try ბლოკის დასრულების შემდეგ უნდა შესრულდეს
}
```

აქ **გამონაკლისის_ტიპი**–ს ქვეშ მიმდინარე გამონკლისის ტიპი იგულისხმება.

გვინდა აღვნიშნოთ, რომ JDK7-ის კომპლექტში დამატებულია try ოპერატორის ახალი ფორმა, რომელიც რესურსების ავტომატურ მართვას უზრუნველყოფს. ამ ფორმას try–რესურსებით ეწოდება და მას ფაილებთან დაკავშირებულ შემდეგ თავებში განვიხილავთ, რადგან ფაილები ყველაზე ხშირად გამოყენებად რესურსებს წარმოადგენს.

5.1.2. გამონაკლისების ტიპები

გამონაკლისთა ყველა ტიპი **Trowable** ჩადგმული კლასის ქვეკლასს წარმოადგენს. ანუ, იგი გამონაკლისთა კლასების იერარქიის ყველაზე მაღალ საფეხურზეა (მწვერვალზეა) განთავსებული. უშუალოდ Trowable კლასის ქვეკლასად ორი კლასი მოიაზრება, რომლებიც ყველა გამონაკლის სიტუაციას ორ ცალკეულ შტოდ ყოფს. ერთ შტოს **Exception** კლასი „უდგას სათავეში“. ეს კლასი იმ გამონაკლისი პირობებისთვის გამოიყენება, რომლებიც სამომხმარებლო პროგრამამ უნდა დაიჭიროს. ეს ამავდროულად, არის კლასი, რომლისგანაც მემკვიდრეობით უნდა მივიღოთ საკუთარი ქვეკლასები გამონაკლისი სიტუაციების საკუთარი ტიპების შექმნის დროს. Exception კლასი შეიცავს მნიშვნელოვან ქვეკლასს სახელწოდებით RuntimeException. აღნიშნული ტიპის გამონაკლისი სიტუაციები ავტომატურად განისაზღვრება იმ პროგრამებისთვის, რომლებსაც ჩვენ ვწერთ, და მოიცავენ ისეთ შეცდომებს, როგორიცაა ნულზე გაყოფა და მასივების მცდარი ინდექსაცია.

მეორე შტო Error კლასიდან იწყება. ის იმ გამონაკლისებს განსაზღვრავს, რომელთა წარმოქმნას პროგრამების ნორმალური შესრულების დროს არ ველით. Error ტიპის გამონაკლისები თავად Java-ს დაპროგრამების გარემოში მიმდინარე შეცდომების აღმოსაჩენად გამოიყენება. ასეთი შეცდომის ნიმუშს სტეკის გადავსება წარმოადგენს. Error ტიპის გამონაკლისები, როგორც წესი, კატასტროფული სიტუაციების პასუხად წარმოიქმნება, რომელთა დამუშავება ჩვენი პროგრამების მიერ შეუძლებელია.

მანამდე, სანამ გაიგებდეთ, თუ როგორ დაამუშავოთ საკუთარ პორგრამებში გამონაკლისები, სასურველია, ნახოთ, თუ რა ხდება მაშინ, როცა მათ არ ამუშავებთ. ქვემოთ წარმოდგენილ მცირე ზომის პროგრამაში სპეციალურად არის დაშვებული ნულზე გაყოფის შეცდომა.

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
public class Excep1 {
	public static void main(String ars[]){
		int d=0;
		int a=45/d;
	}
```

ნახ. 223

შედეგი:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at exc.Excep1.main(Excep1.java:5)
```

ნახ.224

როდესაც Java სისტემა ნულზე გაყოფის მცდელობას აღმოაჩენს, ის გამონაკლისი შემთხვევის ახალ ობიექტს ქმნის და შემდეგ მას გადასცემს. ეს უკანასკნელი Excep1 კლასის შესრულების შეწყვეტას ახდენს, რადგან როგორც კი გამონაკლისი სიტუაცია გადაიცემა, მას იჭერს გამონაკლისი სიტუაციების დამამუშავებელი, რომელმაც მას სასწრაფოდ რამე უნდა მოუხერხოს. მოცემულ მაგალითში ჩვენ ამ განსაკუთრებული სიტუაციის არანაირი საკუთარი დამამუშავებელი არ გამოგვიყენებია, შესაბამისად, იგი Java-ს სტანდარტულმა გადამამუშავებელმა დაიჭირა. ნებისმიერი განსაკუთრებული სიტუაცია, რომელსაც ჩვენი პროგრამა არ დაიჭერს, საბოლოდ Java-ს სტანდარტული დამამუშავებლის მიერ იქნება დაჭერილი და დამუშავებული. სტანდარტული დამამუშავებელი წარმოგვიდგენს სტრიქონს, რომელიც განსაკუთრებულ სიტუაციას აღწერს, გამოიტანს სტეკის ტრასირებას განსაკუთრებული სიტუაციის წარმოქმნის წერტილიდან და პროგრამის შესრულებას შეწყვეტს. სწორედ, ეს გამონაკლისი შემთხვევაა წარმოდგენილი ზემოთ ნაჩვენებ (პროგრამის შესრულების შედეგი) კოდში. ყურადღება მიაქციეთ იმ ფაქტს, რომ კლასის სახელი Excep1, main() მეთოდის სახელი, ფაილის სახელი Excep1.java

და სტრიქონის ნომერი 5 სტეკის ტრასირებაშია ჩართული. ასევე, ყურადსაღებია ისიც, რომ გადაცემული გამონაკლისი სიტუაცია წარმოადგენს Exception კლასის ქვეკლასს სახელწოდებით ArithmeticException, რომელიც უფრო ზუსტად აღწერს წარმოქმნილი შეცდომის ტიპს.

სტეკის ტრასირება ყოველთვის გვიჩვენებს მეთოდების გამოძახების იმ თანმიმდევრობას, რომელმაც შეცდომამდე მიგვიყვანა.

ახლა ვნახოთ წინა პროგრამის მეორე ვერსია (ნახ.225), რომელიც იგივე შეცდომას, ოღონდ სხვა მეთოდში (და არა main() მეთოდში) წარმოგვიდგენს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
public class Excep1 {
	static void subroutine(){
		int d=0;
		int a=45/d;
	}
	public static void main(String ars[]){
		Excep1.subroutine();
	}
}
```

ნახ. 225

შედეგი:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at exc.Excep1.subroutine(Excep1.java:5)
	at exc.Excep1.main(Excep1.java:8)
```

ნახ. 226

აქ უკვე გამოძახებათა მთელი სტეკია წარმოდგენილი. როგორც ხედავთ, ბოლო სტრიქონში ნაჩვენებია main() მეთოდის სტრიქონი 8, რომელშიც ადგილი აქვს subroutine() მეთოდის გამოძახებას, რაც მე–5 სტრიქონში იწვევს შეცდომას. სტეკის ტრასირება პროგრამის გამართვის თვალსაზრისით, ძალზე მოსახერხებელია, რადგან ის სრულ თანმიმდევრობას გვიჩვენებს იმ გამოძახებებისა, რამაც შეცდომამდე მიგვიყვანა.

5.1.3. try და catch ბლოკების გამოყენება

მართალია, Java სისტემის გამონაკლისების სტანდარტული დამმუშავებელი პროგრამის გამართვის თვალსაზრისით საკმაოდ მოხერხებულია, მაგრამ როგორც წესი, მომხმარებელს თავად სურს საკუთარ პროგრამებში წარმოქმნილი განსაკუთრებული სიტუაციების დამუშავება. ეს უკანასკნელი ორ უპირატესობას გვაძლევს. პირველ რიგში, საშუალება გვაქვს შეცდომა გამოვასწოროთ, ხოლო მეორეს მხრივ, პროგრამის შესრულების ავტომატურ შეწყვეტას ადგილი აღარ აქვს. ცხადია, მომხმარებელთა უმრავლესობა ყოველთვის უკმაყოფილო იქნება, თუ ჩვენი პროგრამა გაჩერებას და სტეკის ტრასირებას შეუდგება შეცდომის ყოველი წარმოქმნის შემთხვევაში. თუმცა, ამ საკითხის გამოსწორება საკმაოდ მარტივადაა შესაძლებელი. ამისათვის საკმარისია, პროგრამული კოდის ის ფრაგმენტი, რომლის შემოწმებაც გვსურს, მოვათავსოთ try ბლოკში, რომლის შემდეგ catch კონსტრუქციას უნდა მივმართოთ, რომელიც განსაკუთრებული სიტუაციის ტიპს უთითებს. იმისათვის, რომ ვნახოთ, თუ რამდენად მარტივად კეთდება ეს ყოველივე, ქვემოთ წარმოდგენილ პროგრამაში try ბლოკი catch კონსტრუქციასთან ერთად არის ჩართული. ეს უკანასკნელი ArithmeticException ტიპის გამონაკლისის დამუშავებას ახდენს, რომელიც ნულზე გაყოფის მცდელობის შედეგად წარმოიქმნება.

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
public class Excep1 {
	public static void main(String ars[]){
		int d, a;
		try{ //კოდის მონიტორინგის ბლოკი
			d=0;
			a=45/d;
			System.out.println("შედეგი არ გამოიტანება");
		}
		catch(ArithmeticException e){
			System.out.println("0-ზე გაყოფა");
		}
		System.out.println("catch ოპერატორის შემდეგ");

	}
}
```

ნახ. 227

შედეგი:

```
0-ზე გაყოფა
catch ოპერატორის შემდეგ
```

ნახ. 228

ყურადღება მიაქციეთ იმ ფაქტს, რომ println() მეთოდი try ბლოკის შიგნით არასდროს შესრულდება. როგორც კი განსაკუთრებული სიტუაცია გადაიცემა, try ბლოკიდან მართვა catch ბლოკს გადაეცემა. ანუ, სტრიქონი: „შედეგი არ გამოიტანება“, კონსოლზე არ გამოისახება. catch ბლოკის შესრულების შემდეგ პროგრამაში მართვა try/ catch ბლოკის მომდევნო სტრიქონს გადაეცემა.

try და catch ოპერატორები ერთიან კვანძს შეადგენენ. catch ბლოკის მოქმედების არე არ ვრცელდება იმ ოპერატორებზე, რომლებიც try ოპერატორის წინაა განთავსებული. catch ოპერატორი ვერ იჭერს სხვა try ოპერატორის მიერ გადაცემულ განსაკუთრებულ სიტუაციას (გარდა ჩალაგებული try კონსტრუქციების შემთხვევებისა, რომელთაც მოგვიანებით განვიხილავთ). try ბლოკის მიერ დაცული ოპერატორები ფიგურულ ფრჩხილებში უნდა იქნეს მოთავსებული, ანუ ისინი ბლოკის შიგნით უნდა მდებარეობდეს. პროგრამის ცალკეულ ოპერატორზე try ოპერატორს ვერ გამოვიყენებთ.

სწორად აგებული catch ოპერატორების მიზანი განსაკუთრებული სიტუაციების აღმოფხვრა და პროგრამის შესრულების გაგრძელებაა.

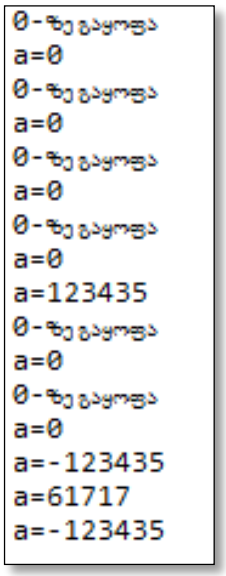
მაგალითი 1. შევადგინოთ პროგრამა, რომელშიც for ციკლის ყოველი იტერაცია ორ შემთხვევით რიცხვს იღებს. ეს ორი რიცხვი ერთმანეთზე იყოფა, ხოლო მიღებულ შედეგზე კი იყოფა რიცხვი 12345. საბოლოო შედეგი a ცვლადში თავსდება. თუ გაყოფის რომელიმე ოპერაცია ნულზე გაყოფის შეცდომას იწვევს, მას სათანადო ბლოკი იჭერს, a ცვლადის მნიშვნელობა ნულს უტოლდება და პროგრამის შესრულება გრძელდება.

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
import java.util.Random;
public class Excep1 {
	public static void main(String args[]){
		int a=0, b=0, c=0;
		Random ob=new Random();
		for(int i=0; i<10; i++){
			try{
				b=ob.nextInt();
				c=ob.nextInt();
				a=123435/(b/c);
			}
			catch(ArithmeticException e){
				System.out.println("0-ზე გაყოფა");
					a=0;  //a-ს განულება და მუშაობის გაგრძელება
			}
			System.out.println("a=" + a);}}}
```

ნახ. 229

შედეგი:



ნახ. 230

5.1.4 გამონაკლისი სიტუაციების აღწერის გამოსახვა

Throwable კლასი ხელახლა განსაზღვრავს Object კლასში განსაზღვრულ toString() მეთოდს. ამგვარად, ის ახდენს იმ სტრიქონის დაბრუნებას, რომელიც გამონაკლისი სიტუაციის აღწერას შეიცავს. println() მეთოდის საშუალებით ჩვენ შეგვიძლია ამ აღწერის კონსოლზე გამოტანა. ამისათვის საკმარისია, println() მეთოდს არგუმენტის სახით გადავცეთ გამონაკლისი სიტუაცია. წინა პროგრამაში (მაგალითი 1) წარმოდგენილი catch ბლოკი შეგვიძლია შევცვალოთ. სახეშეცვლილი პროგრამის შესრულების შედეგი ნახ. 231-ზეა წარმოდგენილი, ხოლო მისი კომპიუტერული რეალიზაცია - ნახ. 232-ზე.

შედეგი:

ნახ. 231

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
import java.util.Random;
public class Excep1 {
	public static void main(String args[]){
		int a=0, b=0, c=0;
		Random ob=new Random();
		for(int i=0; i<10; i++){
			try{
				b=ob.nextInt();
				c=ob.nextInt();
				a=123435/(b/c);
			}
			catch(ArithmeticException e){
				System.out.println("განსაკუთრებული სიტუაცია" + e);
					a=0;  //a-ს განულება და მუშაობის გაგრძელება
			}
			System.out.println("a=" + a);}}}
```

ნახ. 232

5.1.4.1. მრავალჯერადი catch ოპერატორები

არის შემთხვევები, როდესაც პროგრამული კოდის ერთი და იგივე ფრაგმენტი ერთზე მეტ განსაკუთრებულ სიტუაციას მოიცავს. აღნიშნულ შემთხვევაში ჩვენ შეგვიძლია ორი ან მეტი catch ოპერატორი გამოვიყენოთ, რომელთაგან თვითოეული მათგანი თავისი ტიპის შესაბამის გამონაკლისს დაიჭერს. განსაკუთრებული სიტუაციის გადაცემის დროს ყოველი catch ოპერატორი თანმიმდევრობით მოწმდება და მათგან პირველი ის catch ოპერატორი სრულდება, რომლის ტიპიც გამონაკლისს შეესაბამება. catch ოპერატორებიდან ერთ–ერთის შესრულების შემდეგ, დანარჩენი catch ოპერატორები გამოიტოვება და პროგრამის შესრულება იმ ადგილიდან გრძელდება, რომელიც try/catch ბლოკს მოყვება.

მაგალითი 2. შევადგინოთ პროგრამული კოდი, სადაც ორი სხვადასხვა ტიპის განსაკუთრებული სიტუაციის დაჭერა ხდება (ნახ.233).

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
public class Excep1 {
	public static void main(String args[]){
		try{
			int a=args.length;
			System.out.println("a=" + a);
			int b=45/a;
			int c[]={1};
			c[45]=99;
		}catch(ArithmeticException e){
			System.out.println("0-ზე გაყოფა" + e);
		}catch(ArrayIndexOutOfBoundsException e){
			System.out.println("მასივის ინდექსის შეცდომა" + e);
		}
		System.out.println("try/catch ბლოკის შემდგომი ბრძანება");
	}}
```

ნახ. 233

შედეგი:

```
a=0
0-ზე გაყოფაjava.lang.ArithmeticException: / by zero
try/catch ბლოკის შემდგომი ბრძანება
```

ნახ. 234

ეს პროგრამა იძახებს ნულზე გაყოფის განსაკუთრებულ სიტუაციას, რადგან ამ შემთხვევაში a ცვლადის მნიშვნელობა ნულის ტოლია. მაგრამ, იგივე პროგრამა გაყოფის ოპერაციას შეასრულებს, თუ a ცვლადს ნულზე მეტ მნიშვნელობას მივანიჭებთ. თუმცა ამ პროგრამაში ახლა სხვა განსაკუთრებული სიტუაცია შეიქმნება ArrayIndexOutOfBoundsException, რადგან მთელრიცხვა c მასივის სიგრძე ერთის ტოლია, ხოლო ჩვენ ვცდილობთ მასივის c[45] ელემენტს მივანიჭოთ მნიშვნელობა.

ზემოთ წარმოდგენილის პროგრამის მეორე ვერსიას ნახ.235-ზე ნაჩვენები სახე აქვს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
public class Excep1 {
	public static void main(String args[]){
		try{
			int a=1;
			System.out.println("a=" + a);
			int b=45/a;
			int c[]={1};
			c[45]=99;
		}catch(ArithmeticException e){
			System.out.println("0-ზე გაყოფა" + e);
		}catch(ArrayIndexOutOfBoundsException e){
			System.out.println("მასივის ინდექსის შეცდომა" + e);
		}
		System.out.println("try/catch ბლოკის შემდგომი ბრძანება");
	}}
```

ნახ. 525

შედეგი:

```
a=1
მასივის ინდექსის შეცდომაjava.lang.ArrayIndexOutOfBoundsException: 45
try/catch ბლოკის შემდგომი ბრძანება
```

ნახ. 236

5.1.5. ჩადგმული try ოპერატორები

try ოპერატორები შესაძლოა ერთმანეთში იყოს ჩადგმული. რაც ნიშნავს, რომ try ოპერატორი შეიძლება სხვა try ოპერატორის ბლოკში განთავსდეს. ყოველ ჯერზე, როდესაც მართვა try ოპერატორს გადაეცემა, შესაბამისი განსაკუთრებული სიტუაცია სტეკში თავსდება. თუ ჩადგმულ try ოპერატორს ამ განსაზღვრული განსაკუთრებული სიტუაციის დამამუშავებელი catch ოპერატორი არ გააჩნია, სტეკი „იხსნება“ და შესაბამისობაზე მოწმდება გარე try ბლოკის catch ოპერატორი. ასე გაგრძელდება მანამ, სანამ შესაბამისი catch ოპერატორი არ მოიძებნება ან სანამ არ შემოწმდება ჩალაგებული try ოპერატორების ყველა დონე.

მაგალითი 3. წარმოგვიდგენს ჩალაგებული (ჩადგმული) try ოპერატორების ამსახველ პროგრამას.

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
public class Exc2 {
	public static void main(String args[]){
		try{
			int a=args.length;
			int b=42/a;
			System.out.println("a=" + a);
			try{// ჩადგმული try ვლოკი
				if(a==1) a=a/(a-a); //0-ზე გაყოფა
				if(a==2){
				int c[]={1};
				c[42]=99; } //მასივის საზღვრებს გარეთ გასვლის
განსაკ. სიტუაცია
			}catch(ArrayIndexOutOfBoundsException e){
				System.out.println("ინდექსი მასივის საზღვრებს
გარეთაა" + e);
			}
		}catch(ArithmeticException e){
			System.out.println("0-ზე გაყოფა" +e);}}}
```

ნახ. 237

როგორც ზემოთ წარმოდგენილი პროგრამიდან ჩანს, აქ ერთი try ბლოკი მეორეშია ჩადგმული. პროგრამა კი შემდეგნაირად მუშაობს. როდესაც მას ბრძანების ხაზის არგუმენტების გარეშე გავუშვებთ, გარე try ბლოკი ქმნის 0–ზე გაყოფის განსაკუთრებულ სიტუაციას. თუ პროგრამას გავუშვებთ შესაბამისი ერთი არგუმენტით, 0–ზე გაყოფის განსაკუთრებული სიტუაცია შიდა try ბლოკს გადაეცემა. მაგრამ, რადგან შიდა try ბლოკი არ ახდენს ამ ტიპის განსაკუთრებული სიტუაციის დამუშავებას, ამიტომ იგი უბრუნდება გარე try ბლოკს, რომელიც მას ამუშავებს. თუ პროგრამას ორი არგუმენტი გადაეცემა, მაშინ მასივის ინდექსის საზღვრებს გარეთ გასვლის განსაკუთრებული სიტუაცია იქმნება შიდა try ბლოკში.

პროგრამის შესრულებაზე გაშვების ერთ–ერთი შესაძლო შედეგი ნახ.238-ზეა წარმოდგენილი.

შედეგი:

```
0-ზე გაყოფაjava.lang.ArithmeticException: / by zero
```

ნახ. 238

5.1.6. throw და throws ოპერატორები

აქამდე, ჩვენ Java-ს შესრულების სისტემის მიერ გადაცემული განსაკუთრებული სიტუაციების დაჭერას ვახდენდით. თუმცა არსებობს შესაძლებლობა, რომლითაც განსაკუთრებული სიტუაციები უშუალოდ ჩვენი პროგრამიდან შეგვიძლია გადავცეთ ცხადი სახით. სწორედ, ამ მიზანს ემსახურება **throw** ოპერატორი, რომლის ჩაწერის ზოგადი ფორმა შემდეგია:

throw Throwable_ეგზემპლიარი;

აქ **Throwable_ეგზემპლიარი** უნდა წარმოადგენდეს **Throwable** კლასის ობიექტს ან **Throwable** კლასის ქვეკლასს. მონაცემთა ისეთი ელემენტარული ტიპები, როგორიცაა int ან char, ან **Throwable** კლასისგან განსხვავებული კლასები, მაგალითად: String და Object, განსაკუთრებული სიტუაციებისთვის არ შეიძლება გამოყენებულ იქნეს.

არსებობს **Throwable** კლასის ობიექტის მიღების ორი შესაძლებლობა: catch ოპერატორში პარამეტრის გამოყენება ან new ოპერატორით ობიექტის შექმნა.

შესრულების ნაკადი უშუალოდ throw ოპერატორის შემდეგ ჩერდება და შესაბამისად, მომდევნო ოპერატორები არ სრულდება. ამ დროს მოიძებნება უახლოესი დახურული try ბლოკი, რომელიც განსაკუთრებული სიტუაციის შესაბამისი ტიპის catch ოპერატორს შეიცავს. თუ შესაბამისობა მოიძებნება, მართვა ამ ოპერატორს გადაეცემა. წინააღმდეგ შემთხვევაში, მოწმდება შემდეგი გარე try ბლოკი და ა.შ. თუ ვერ მოხერხდება განსაკუთრებული სიტუაციის ტიპის შესაბამისი catch ოპერატორის მოძებნა, მაშინ განსაკუთრებული სიტუაციების სტანდარტული დამამუშავებელი პროგრამის შესრულებას წყვეტს.

მაგალითი 4. შევადგინოთ პროგრამა, რომელიც ქმნის და გადაცემს განსაკუთრებულ სიტუაციას, ხოლო დამამუშავებელი, რომელიც მას იჭერს, ხელახლა გადასცემს მას გარე დამამუშავებელს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
public class Exc2 {
	static void demoproc(){
		try{
			throw new NullPointerException("demo");
		}catch(NullPointerException e){
			System.out.println("დაჭრილია demoproc - ის შიგნით");
			throw e;
		}
	}
	public static void main(String args[]){
	try{
	  demoproc();
	}catch(NullPointerException e){
	  System.out.println("ხელახალი დაჭერა  " +e);
	}}}
```

ნახ. 239

შედეგი:

```
დაჭრილია demoproc - ის შიგნით
ხელახალი დაჭერა java.lang.NullPointerException: demo
```

ნახ. 240

ეს პროგრამა ერთი და იმავე შეცდომის დამუშავების ორ საშუალებას მოიცავს. თავდა-პირველად, main() მეთოდი აყენებს განსაკუთრებული სიტუაციის კონტექსტს, ხოლო შემდეგ demoproc() მეთოდს იძახებს, რომელიც განსაკუთრებული სიტუაციის დამუშავების სხვა კონტექსტს წარმოგვიდგენს და ახდენს NullPointerException განსაკუთრებული სიტუაციის ახალი ეგზემპლიარის მყისიერად გადაცემას, რომლის დაჭერა შემდეგ სტრიქონში ხორციელდება. შემდეგ, განსაკუთრებული სიტუაცია ხელახლა გადაიცემა. წარმოდგენილი პროგრამა იმის დემონსტრირებასაც ახდენს, თუ როგორ შევქმნათ Java-ს სტანდარტული გამონაკლისების საკუთარი ობიექტები. ყურადღება მიაქციეთ შემდეგ სტრიქონს:

throw new NullPointerException(“demo”);

აქ new ოპერატორი NullPointerException გამონაკლისის ეგზემპლიარის შესაქმნელად გამო-იყენება. Java-ში ჩადგმულ ბევრ განსაკუთრებულ სიტუაციას, სულ მცირე, ორი კონსტრუქტორი მაინც გააჩნია: პარამეტრების გარეშე და სტრიქონული პარამეტრით. მეორე ფორმის გამოყენების დროს, არგუმენტი იმ სტრიქონზე მიუთითებს, რომელიც განსაკუთრებულ სიტუაციას აღწერს. ეს სტრიქონი ცხადი სახით ჩანს მაშინ, როდესაც ობიექტი print() ან println() მეთოდების

არგუმენტად გამოიყენება. ის, ასევე, შეგვიძლია მივიღოთ getMessage() მეთოდის გამოძახებითაც, რომელიც Throwable კლასშია განსაზღვრული.

თუ მეთოდი იწვევს განსაკუთრებულ სიტუაციას, რომლის დამუშავებასაც ის თავად არ ახდენს, მაშინ ამ გამონაკლისის დამუშავება მეთოდის გამომძახებელმა კოდმა უნდა შეძლოს. ამ მიზნით მეთოდის აღწერას **throws** კონსტრუქცია ემატება. ეს უკანასკნელი წარმოგვიდგენს იმ განსაკუთრებული სიტუაციების ტიპებს, რომლებიც მეთოდმა შეიძლება გამოიწვიოს. ეს აუცილებელია ყველა გამონაკლსის შემთხვევაში, გარდა Error, RuntimeException ტიპის გამონაკლისებისა ან მათი ქვეკლასებისა. ყველა სხვა დანარჩენი გამონაკლისი, რაც მეთოდის მიერ შეიძლება გადაცემულ იქნეს, throws კონსტრუქციაში უნდა გამოცხადდეს. წინააღმდეგ შემთხვევაში კომპილაციის შეცდომას მივიღებთ.

throws ოპერატორის შემცველი მეთოდის გამოცხადების ზოგადი ფორმა შემდეგია:

```
ტიპი მეთოდის_სახელი(პარამეტრების სია) throws გამონაკლისების სია
{
  //მეთოდის ტანი;
}
```

გამონაკლისების სია აქ მძიმეებით გამოყოფილი იმ განსაკუთრებული სიტუაციების სიაა, რომელიც მეთოდმა შეიძლება გადასცეს.

5.1.7. finally ოპერატორი

როდესაც განსაკუთრებული სიტუაცია გადაცემულია, მეთოდის შესრულება არაწრფივად მიმდინარეობს, რაც მასში მართვის ნორმალური ნაკადის ცვლილებას იწვევს. იმის მიხედვით, თუ როგორ არის მეთოდი დაწერილი, არსებობს მართვის ნაადრევი დაბრუნების შესაძლებლობა. ზოგიერთ მეთოდში ამან სერიოზული პრობლემა შეიძლება გამოიწვიოს. მაგალითად, თუ მეთოდი შესვლისას ხსნის ფაილს, ხოლო გამოსვლისას მას ხურავს, ბუნებრივია, ჩვენ არ გვექნება სურვილი ფაილის დამხურავი კოდი გამოტოვებულ იქნეს განსაკუთრებული სიტუაციის დამამუშავებელი მექანიზმის გამოყენების გამო. მსგავს სიტუაციებში პრობლემის აღმოფხვრის მიზნით საკვანძო სიტყვა **finally** გამოიყენება. ის ქმნის კოდის ბლოკს, რომელიც try/catch ბლოკის დასრულების შემდეგ შესრულდება. finally ბლოკი ყოველთვის შესრულედება, მიუხედავად იმისა, განსაკუთრებული სიტუაციის გადაცემა ხდება თუ არა. როდესაც განსაკუთრებული სიტუაცია გადაცემულია, finally ბლოკი მაშინაც კი შესრულდება, თუ არცერთი catch ბლოკი არ

შეესაბამება განსაკუთრებულ სიტუაციას. ნებისმიერ მომენტში, როდესაც მეთოდი შეეცდება მართვა დაუბრუნოს try/catch ბლოკიდან გამომძახებელ კოდს (დაუმუშავებელი გამონაკლისის, ან return ოპერატორის ცხადი სახით გამოყენებით), finally ბლოკი მეთოდიდან მართვის დაბრუნებამდე შესრულედება. ეს მოსახერხებელია ფაილთა დესკრიპტორების დასახურად ან სხვა რესურსების გამოსათავისუფლებლად, რომლებიც მეთოდის დასაწყისში იქნა მიღებული და მართვის დაბრუნებამდე უნდა გამოთავისუფლდეს. ზოგადად, finally ოპერატორის გამოყენება სავალდებულო არ არის. თუმცა ყოველი try ოპერატორი ითხოვს სულ ცოტა ერთ catch ან finally ოპერატორს.

მაგალითი 5. შევადგინოთ პროგრამა, რომელშიც სამი მეთოდია გამოყენებული. ისინი მართვის დაბრუნებას სხვადასხვა საშუალებებით ახდენენ, მაგრამ არცერთი არ ახდენს finally ბლოკის შესრულების გამოტოვებას.

აღნიშნულ პროგრამაში procA() მეთოდი შესრულებას ნაადრევად წყვეტს try ბლოკში, გადასცემს რა მას განსაკუთრებულ სიტუაციას. მიუხედავად ამისა, finally ბლოკი მაინც სრულდება.

procB() მეთოდში try ბლოკში მართვის დაბრუნება return ოპერატორით ხორციელდება. აქ finally ბლოკი procB() მეთოდიდან მართვის დაბრუნებამდე სრულდება.

procC() მეთოდში try ბლოკი ნორმალურად სრულდება, შეცდომების გარეშე. მიუხედავად ამისა, finally ბლოკი მაინც სრულდება.

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
public class Exc2 {
	//გამონაკლისის მეთოდიდან გადაცემა
	static void procA(){
		try{
		System.out.println("procA მეთოდის შიგნით");
		throw new RuntimeException("demo");
	}finally{
		System.out.println("finally procA ბლოკი");
	}}
//try ბლოკში მართვის დაბრუნება
	static void procB(){
		try{
		System.out.println("procB მეთოდის შიგნით");
		return;
		}finally{
			System.out.println("finally procB
ბლოკი");
		}}
	//try ბლოკის ნორმალური შესრულება
	static void procC(){
		try{
		System.out.println("procC მეთოდის შიგნით");
		return;
		}finally{
			System.out.println("finally procC
ბლოკი");
		}}
	 public static void main(String args[]){
	try{
	  procA();
	}catch(Exception e){
	  System.out.println("გამონაკლისი დაჭერილია");
	}
	procB();
	procC();
	}}
```

ნახ. 241

შედეგი:

```
procA მეთოდის შიგნით
finally procA ბლოკი
გამონაკლისი დაჭერილია
procB მეთოდის შიგნით
finally procB ბლოკი
procC მეთოდის შიგნით
finally procC ბლოკი
```

ნახ. 242

Java–ს ჩადგმული გამონაკლისები

java.lang სტანდარტულ პაკეტში განსაკუთრებული სიტუაციების რამდენიმე კლასია განსაზღვრული. მათი უმრავლესობა სტანდარტული RuntimeException ტიპის ქვეკლასებს წარმოადგენს. აღნიშნული გამონაკლისების მოთავსება throws მეთოდის სიაში საჭირო არ არის. მათ **შეუმოწმებელი გამონაკლისები** ეწოდება, რადგან კომპილატორი ამ გამონაკლისების მეთოდის მიერ დამუშვების ან გადაცემის ფაქტს არ ამოწმებს. java.lang პაკეტში განსაზღვრული არაშემოწმებადი გამონაკლისები 29-ე ცხრილშია წარმოდგენილი. ხოლო 30-ე ცხრილში java.lang პაკეტში განსაზღვრული ის გამონაკლისებია ნაჩვენები, რომლებიც throws მეთოდების სიაში უნდა განთავსდეს, რამეთუ ეს მეთოდები ქმნიან, თუმცა დამოუკიდებლად ვერ ახდენენ მათ დამუშავებას. ასეთ გამონაკლისებს **შემოწმებადი გამონაკლისები** ეწოდება.

ცხრილი 29 java.lang პაკეტში განსაზღვრული RuntimeException კლასის შეუმოწმებელი ქვეკლასები

გამონაკლისი	აღწერა
ClassNotFoundException	კლასი ვერ მოიძებნა
ClassNotSupportedException	იმ ობიექტის კლონირების მცდელობა, რომელიც Cloneable ინტერფეისის რეალიზებას არ ახდენს
IllegalAccessException	კლასზე წვდომა დაუშვებელია
InstantiationException	აბსტრაქტული კლასის ან ინტერფეისის ობიექტის შექმნის მცდელობა
InterruptedException	ერთი ნაკადი შეწყვეტილია მეორე ნაკადის მიერ
NoSuchFieldException	მოთხოვნილი ველი არ არსებობს
NoSuchMethodException	მოთხოვნილი მეთოდი არ არსებობს
ReflectiveOpertaionException	რეფლექსიასთან დაკავშირებული გამონაკლისების სუპერკლასი (დამატებულია JDK7-ში)

ცხრილი 30 java.lang პაკეტში განსაზღვრული შემოწმებადი გამონაკლისები

გამონაკლისი	აღწერა
ArithmeticException	არითმეტიკული შეცდომა, როგორიცაა ნულზე გაყოფა
ArrayIndexOutOfBoundsException	ინდექსის გასვლა მასივის საზღვრებს გარეთ
ArrayStoreException	მასივის ელემენტზე არათავსებადი ტიპის ობიექტის მინიჭება
ClassCastException	არასწორი გარდასახვა
EnumConstantNotPresentException	ჩამოთვლადი სიის განუსაზღვრელი მნიშვნელობის გამოყენების მცდელობა
IllegalArgumentException	მეთოდის გამოძახებისას არასწორი არგუმენტია გამოყენებული
IllegalMonitorStateException	მონიტორინგის არასწორი ოპერაცია, როგორიცაა დაუბლოკავი ნაკადის მოლოდინი
IllegalStateException	გარემო ან პროგრამული უზრუნველყოფა არაკორექტულ მდგომარეობაშია
IllegalThreadStateException	მოთხოვნილი ოპერაცია ნაკადის მიმდინარე მდგომარეობის არათავსებადია
IndexOutOfBoundsException	ინდექსის გარკვეული ტიპი გასცდა დასაშვებ საზღვრებს
NegativeArraySizeException	შექმნილია უარყოფითი ზომის მასივი
NullPointerException	ცარიელი მიმართვის არასწორი გამოყენება
NumberFormatException	სტრიქონის არასწორი გარდასახვა რიცხვით ფორმატში
SecurityException	უსაფრთხოების დარღვევის მცდელობა
StringIndexOutOfBounds	სტრიქონის საზღვრებს გარეთ ინდექსის გამოყენების მცდელობა
TypeNotPresentException	ტიპი არ მოიძებნა
UnsupportedOperationException	აღმოჩენილია მიუღებელი ოპერაცია

5.1.8. გამონაკლისების სამი ახალი საშუალება JDK 7-ში

JDK 7–ის გამონაკლისთა სისტემის კომპლექტს არც თუ ისე დიდი ხნის წინ სამი ძალზე საინტერესო და მნიშვნელოვანი საშუალება დაემატა. პირველი მათგანი ისეთი რესურსის გამოთავისუფლების ავტომატურ პროცესს ახორციელებს, როგორიცაა, ფაილი, რომელიც საჭირო აღარ არის. ის ემყარება try ოპერატორის გაფართოებულ ფორმას, რომელსაც try-რესურსებიან ოპერატორს უწოდებენ. მეორე საშუალებას მულტი–დამამუშავებელი (multi-catch) ეწოდება, ხოლო მესამეს - ფინალური გამეორებითი გადაცემა (final rethrow), რომელსაც მეორენაირად უფრო ზუსტ გამეორებით გადაცემასაც (more precise rethrow) უწოდებენ. ბოლო ორ სიახლეს ახლა განვიხილავთ.

მულტიდამამუშავებელი ერთსა და იმავე catch ოპერატორში საშუალებას გვაძლევს რამდენიმე გამონაკლისი დავამუშავოთ. სავსებით ნორმალური სიტუაციაა, როდესაც დამამუშავებლები სხვადასხვა ტიპის გამონაკლისების დამუშავების მიზნით ერთნაირ კოდს იყენებენ, მიუხედავად იმისა, რომ ისინი სხვადასხვა გამონაკლისებს შეესაბამება. ამ შემთხვევაში სხვადასხვა ტიპის გამონაკლისების ინდივიდუალური დამუშავების ნაცვლად თქვენ ერთი catch ბლოკი კოდის დუბლირების გარეშე შეგიძლიათ გამოიყენოთ ყველა გამონაკლისის დასამუშავებლად. იმისათვის, რომ ასეთ მულტიდამამუშავებლს მიმართოთ, საკმარისია catch ოპერატორში გამონაკლისის ყოველი ტიპი OR ოპერატორით გამოყოთ. მულტიდამამუშავებლის ყოველი პარამეტრი არაცხადი სახით, მაგრამ მაინც ფინალურია (სურვილის შემთხვევაში შეგიძლიათ ცხადად მიუთითოთ final საკვანძო სიტყვა, თუმცა ამის გაკეთება აუცილებლობას არ წარმოადგენს). რადგან მულტიდამამუშავებლის ყოველი პარამეტრი არაცხადად ფინალურია, ამიტომ მათ ახალი მნიშვნელობები არ შეგვიძლია მივანიჭოთ.

catch ოერატორს, რომელიც მულტიდამამუშავებელს ArithmeticException და ArrayIndexOutOfBoundsException ტიპის გამონაკლისების დასამუშვებლად იყენებს, შემდეგი სახე აქვს:

```
catch(ArithmeticException | ArrayIndexOutfBoundsException e)  {
```

მაგალითი 6. შევადგინოთ მულტიდამამუშავებლის დემონსტრირების ამსახველი პროგრამა. ის ArithmeticException ტიპის გამონაკლისს ნულზე გაყოფის მცდელობის შემთხვევაში ქმნის. თუ პროგრამაში კომენტარად აქცევთ გაყოფის ოპერატორს, ხოლო მომდევნო სტრიქონს მოხსნით კომენტარს, მაშინ ArrayIndexOutOfBoundsException ტიპის გამონაკლისი შეიქმნება. ორივე გამონაკლისი კი ერთი catch ოპერატორის მიერ მუშავდება.

პროგრამის კომპიუტერული რეალიზაცია:

```
package exc;
public class Exc2 {
       public static void main(String args[]){
      int a=10, b=0;
      int vals[]={1, 2, 3};
      try{
        int res=a/b; //იქმნება ArithmeticException ტიპის გამონაკლისი
        //vals[10]=19; იქმნება ArraIndexOutOfBoundsException ტიპის
გამონაკლისი
        // ეს catch ოერატორი ორივე ტიპის გამონაკლისის დამუშავებას ახდენს.
      }catch(ArithmeticException | ArrayIndexOutOfBoundsException e){
        System.out.println("მუშავდება გამონაკლისი: " + e);
      }
         System.out.println("იბეჭდება მულტიდამამუშავებლის შემდეგ");
      ``
```

ნახ. 243

შედეგი:

```
მუშავდება გამონაკლისი: java.lang.ArithmeticException: / by zero
იბეჭდება მულტიდამამუშავებლის შემდეგ
```

ნახ. 244

პროგრამის სახეშეცვლილი ვერსია:

```
package exc;
public class Exc2 {
       public static void main(String args[]){
      int a=10, b=0;
      int vals[]={1, 2, 3};
      try{
        // int res=a/b; //იქმნება ArithmeticException ტიპის გამონაკლისი
        vals[10]=19; //იქმნება ArraIndexOutOfBoundsException ტიპის გამონაკლისი
        // ეს catch ოერატორი ორივე ტიპის გამონაკლისის დამუშავებას ახდენს.
      }catch(ArithmeticException | ArrayIndexOutOfBoundsException e){
        System.out.println("მუშავდება გამონაკლისი: " + e);
      }
         System.out.println("იბეჭდება მულტიდამამუშავებლის შემდეგ");
      }}
```

ნახ. 245

უფრო ზუსტი განმეორებითი გადაცემის საშუალება გამონაკლისთა იმ ტიპებს უკავშირდება, რომლებიც ხელმეორედ გადაიცემა იმ შემოწმებადი გამონაკლისების სახით, რომლებიც გადაცემის try ბლოკს უკავშირდება და არ მუშავდება catch ოპერატორის მიერ. ისინი, როგორც წესი, პარამეტრის ქვეტიპს ან სუპერტიპს წარმოადგენენ. ამ ბოლო ახალი საშუალების გამოყენების საჭიროება მართალია, იშვიათად იქმნება, თუმცა ამით სარგებლობა დღეს უკვე შესაძლებელია. უფრო ზუსტი განმეორებითი გადაცემის გამოყენების შემთხვევაში, catch ოპერატორის პარამეტრი ფაქტიურად ფინალური უნდა იყოს, რაც იმას ნიშნავს, რომ catch

ბლოკში მას ან არ უნდა მიენიჭოს ახალი მნიშვნელობა, ან ის ცხადი სახით უნდა გამოცხადდეს როგორც final (ფინალური).

დასასრულს, გვინდა აღვნიშნოთ, რომ გამონაკლისების (განსაკუთრებული სიტუაციების) დამუშავება რთული პროგრამების მართვის მძლავრ მექანიზმს წარმოადგენს. try, throw და catch ოპერატორები თქვენს პროგრამულ ლოგიკაში არსებული შეცდომებისა და განსაკუთრებული კრიტიკული სიტუაციების დამუშავების მარტივ საშუალებებს განეკუთვნება.

ობიექტზე ორიენტირებული დაპროგრამების სხვა ენებისგან განსხვავებით, როცა მეთოდის შესრულება შეიძლება წარუმატებლად დასრულდეს (შეწყდეს), Java-ში ის გამონაკლისს გადასცემს. ეს კი ყველაზე მარტივი საშუალებაა გავუმკლავდეთ შეცდომითი ხასიათის სიტუაციებს.

დავალება:

1. **განმარტეთ finally ბლოკის მუშაობის მექანიზმი.**
2. **განმარტეთ შეუმოწმებელი გამონაკლისების არსი.**
3. **შეადგინეთ პროგრამა, რომელშიც while ციკლის ყოველ იტერაციაზე (მათი რიცხვი 5-ის ტოლია) ორ შემთხვევით რიცხვს იღებს. ეს ორი რიცხვი ერთმანეთზე იყოფა, ხოლო მიღებულ შედეგზე იყოფა რიცხვი 589. საბოლოო შედეგი *x* ცვლადში მოათავსეთ. მოახდინეთ ნოლზე გაყოფის შეცდომის დაჭერა და დამუშავება (მიმართეთ try/catch ოპერატორების ბლოკებს და კონსოლზე გამოიტანეთ გამონაკლისი სიტუაციის აღწერა).**
4. **შეადგინეთ პროგრამა, რომელშიც ორი სხვადასხვა ტიპის (ნოლზე გაყოფისა და მასივის ინდექსის საზღვრებს გარეთ გასვლის) განსაკუთრებული სიტუაციის დაჭერას განახორციელებთ.**
5. **შეადგინეთ მულტიდამამუშავებლის მოქმედების ამსახველი პროგრამა. განმარტეთ მისი მუშაობის მექანიზმის უპირატესობები.**

5.2. მრავალნაკადური დაპროგრამება

დაპროგრამების სხვა ენებისგან განსხვავებით, Java გვთავაზობს მრავალნაკადური დაპროგრამების ჩადგმულ შესაძლებლობებს. ზოგადად, მრავალნაკადიანი პროგრამა მოიცავს ორ ან რამდენიმე ნაწილს, რომლებიც ერთდროულად შეიძლება შესრულდეს. ასეთი პროგრამის ყოველ ნაწილს **ნაკადი** (**thread**) ეწოდება და ყოველი ნაკადი შესრულების ცალკეულ გზას მიუთითებს. სხვა სიტყვებით რომ ვთქვათ, მრავალნაკადიანობა - მრავალამოცანიანობის სპეციალიზირებული ფორმაა.

მრავალამოცანიანობის ორი განსხვავებული ტიპი არსებობს: მრავალამოცანიანობა, დამყარებული პროცესებზე და მრავალამოცანიანობა, დამყარებული ნაკადებზე. მნიშვნელოვანია, მათ შორის არსებული განსხვავების ცოდნა. პროცესი, თავისი არსით, შესრულებადი პროგრამაა. ანუ, პროცესებზე დაფუძნებული მრავალამოცანიანობა ეს ის საშუალებაა, რომელიც თქვენს კომპიუტერს შესაძლებლობას აძლევს ერთდროულად ორი ან მეტი პროგრამა შეასრულოს. ასე, მაგალითად: პროცესებზე დაფუძნებული მრავალამოცანიანობა საშუალებას გაძლევთ გაუშვათ Java-კომპილატორი იმ დროს, როდესაც თქვენ სარგებლობთ ვთქვათ, ტექსტური რედაქტორით, ან ნახულობთ ვებ-გვერდს. ამ შემთხვევაში, პროგრამა კოდის უმცირეს ელემენტს წარმოადგენს, რომლის მართვა ოპერაციული სისტემის დამგეგმარებელს შეუძლია.

ნაკადური მრავალამოცანიანობის გარემოში მმართველი კოდის უმცირეს ელემენტს ნაკადი წარმოადგენს. ეს კი ნიშნავს, რომ ერთმა პროგრამამ ორი ან მეტი ამოცანა შეიძლება ერთდროულად შეასრულოს. მაგალითად, ტექსტური რედაქტორი შეიძლება ტექსტის ფორმატირებას ახდენდეს და ამავდროულად, ადგილი ქონდეს მის საბეჭდ მოწყობილობაზე ამობეჭდვას. ეს ორი მოქმედება ერთდროულად ორი ცალკეული ნაკადის მიერ სრულდება. მაშასადამე, პროცესებზე დამყარებულ მრავალამოცანიანობას საქმე „მთლიან სურათთან“ აქვს, ხოლო ნაკადურ მრავალამოცანიანობას - დეტალებთან. პროცესები - მძიმეწონიანი ამოცანებია, რომელთაგან თვითოეული საკუთარ მისამართის სივრცეს მოითხოვს. პროცესთაშორისი კომუნიკაციები ძვირადღირებული და შეზღუდულია. ერთი პროცესის კონტექსტიდან მეორეზე გადართვა, ასევე, ძვირი ჯდება. ამ მხრივ, ნაკადები გაცილებით მარტივია. ისინი ერთობლივად იყენებენ ერთსა და იმავე სამისამართო სივრცეს და ერთსა და იმავე მძიმეწონიან პროცესს. ნაკადებს შორის კომუნიკაცია ეკონომიურია და მათ შორის კონტექსტის გადართვაც დაბალი ღირებულებით ხასიათდება. მიუხედავად იმისა, რომ Java-პროგრამები პროცესებზე დაფუძნებულ მრავალამოცანიან გარემოში გამოიყენება, მრავალამოცანიანობის ეს ფორმა Java-საშუალებებით მაინც არ კონტროლდება, ხოლო მრავალნაკადიანი მრავალამოცანიანობა კი Java-საშუალებებით კონტროლირებადია.

მრავალნაკადურობა საშუალებას გვაძლევს ვწეროთ ეფექტური პროგრამები, რომლებშიც სისტემის პროცესორის დასაშვები სიმძლავრე მაქსიმალურად იქნება გამოყენებული. მრავალნაკადურობის კიდევ ერთ უპირატესობას მოლოდინის დროის მინიმუმამდე დაყვანა წარმოადგენს. ეს განსაკუთრებით მნიშვნელოვანია იმ ინტერაქტიული ქსელური გარემოსთვის, რომელშიც Java მუშაობს. ასე, მაგალითად: ქსელში მონაცემთა გადაცემის სიჩქარე გაცილებით დაბალია იმ სიჩქარეზე, რომლითაც კომპიუტერი ახდენს მათ დამუშავებას. ლოკალური ფაილური სისტემის წაკითხვა და ჩაწერაც გაცილებით ნელა მიმდინარეობს, ვიდრე მათი დამუშავების ტემპი პროცესორში. და რა თქმა უნდა, მომხმარებელს გაცილებით ნელა შეაქვს მონაცემები კლავიატურიდან, ვიდრე მათ თავად კომპიუტერი ამუშავებს. ერთნაკადიან გარემოში თქვენი პროგრამა იძულებულია დაელოდოს გარკვეული ამოცანების დასრულებას, რათა შემდეგი ამოცანის შესრულებაზე გადავიდეს. მრავალნაკადიანობა კი ხელს უწყობს შეყოვნების დროის შემცირებას, რამეთუ აქ შესაძლებელია სხვა ნაკადები შესრულდეს, სანამ ერთი მოლოდინის რეჟიმშია.

5.2.1. Java - ნაკადების მოდელი

ფაქტიურად, java ნაკადებს იყენებს იმისთვის, რომ უზრუნველყოს შესრულების მთელი გარემოს ასინქრონულობა. მრავალნაკადური გარემოს არსი შედარების შედეგად გაცილებით იოლი გასაგებია. ერთნაკადიანი სისტემები იყენებენ მიდგომას, რომელსაც **მოვლენათა ცილკს (გამოკითხვით)** უწოდებენ. აღნიშნულ მოდელში მართვის ერთადერთი ნაკადი უსასრულო ციკლში სრულდება, რომელიც ერთადერთი რიგის გამოკითხვას ახდენს იმ მიზნით, რომ მიიღოს გადაწყვეტილება, თუ რა განახორციელოს შემდგომ. როგორც კი გამოკითხვის მექანიზმი დააბრუნებს, ვთქვათ, სიგნალს, იმის შესახებ, რომ ქსელური ფაილი წასაკითხად მზადაა, მოვლენათა ციკლი მართვას გადასცემს მოვლენათა შესაბამის დამამუშავებელს. მანამდე, სანამ ის მართვას არ დააბრუნებს, პროგრამაში ვერაფერი შესრულდება. ამან კი შესაძლოა გამოიწვიოს პროგრამის ერთი ნაწილის მეორეზე დომინირება და არ მისცეს საშუალება დამუშავდეს ნებისმიერი სხვა მოვლენა. საზოგადოდ, ერთნაკადიან გარემოში, როდესაც ამა თუ იმ რესურსის მოლოდინის მიზეზით ადგილი აქვს ნაკადის ბლოკირებას (ანუ შესრულების პროცესს დროებით აჩერებს), მთელი პროგრამის შესრულება ჩერდება.

მრავალნაკადიანობის უპირატესობა იმაში მდგომარეობს, რომ მასში ციკლური გამოკითხვის ძირითადი მექანიზმი გამორიცხულია. აქ ნაკადი შეიძლება შეჩერებულ იქნეს პროგრამის სხვა ნაწილების შეჩერების გარეშე. მრავალნაკადიანობა ანიმაციის ციკლებს უფლებას აძლევს მეზობელი კადრების ჩვენებისას „წამით ჩაიძინონ“, ისე, რომ მთელი სისტემის მუშაობა არ

შეჩერდეს. როდესაც Java-პროგრამაში ნაკადის ბლოკირებას აქვს ადგილი, მაშინ ჩერდება მხოლოდ ერთადერთი ბლოკირებული ნაკადი, დანარჩენი ნაკადების შესრულება კი გრძელდება.

ბოლო რამდენიმე წელია, რაც მრავალბირთვიანი სისტემები ჩვეულებრივ მოვლენად იქცა. თუმცა, ერთბირთვიანი სისტემები ჯერ კიდევ ფართოდ გავრცელებულია და გამოიყენება. მთავარია გვესმოდეს, რომ Java-ს მრავალნაკადიანი საშუალებები ორივე ტიპის სისტემაში მუშაობს. ერთბირთვიან სისტემაში ერთდროულად შესრულებადი რამდენიმე ნაკადი პროცესორს ერთობლივად იყენებს და ყოველი ნაკადისთვის პროცესორული დროის გარკვეულ სექტორს იღებს. შესაბამისად, ერთბირთვიან სისტემაში ორი ან მეტი ნაკადი ფაქტიურად ერთდროულად არ სრულდება. ისინი პროცესორული დროის გამოყენების საკუთარ რიგს ელოდებიან. მაგრამ მრავალბირთვიან სისტემებში ორი ან მეტი ნაკადი ფაქტიურად ერთდროულად შეიძლება შესრულდეს. უმეტეს შემთხვევაში, ამან შეიძლება პროგრამის ეფექტურობისა და გარკვეული ოპერაციების შესრულების სიჩქარის ზრდა გამოიწვიოს.

ნაკადი რამდენიმე მდგომარეობაში არსებობს. მათი ზოგადი აღწერა შემდეგია: ნაკადი შეიძლება **სრულდებოდეს.** ის შეიძლება **მზად იყოს შესრულებისთვის,** როგორც კი ცენტრალური პროცესორის დროს მიიღებს. მუშა ნაკადი შეიძლება **შეჩერებულ** იქნეს, რაც დროებით წყვეტს მის აქტიურობას. შეჩერებული ნაკადის შესრულება შესაძლებელია **აღდგენილ** იქნეს, რაც მას უფლებას მისცემს გააგრძელოს მუშაობა იმ ადგილიდან, საიდანაც მისი შეჩერება მოხდა. შესაძლოა ნაკადის **ბლოკირება,** როდესაც ის გარკვეულ რესურსს ელოდება. ნებისმიერ მომენტში ნაკადი შეიძლება **შეწყდეს,** რაც მისი შესრულების მყისიერ შეჩერებას იწვევს. ერთხელ შეწყვეტილი ნაკადის აღდგენა კი შეუძლებელია.

5.2.2. ნაკადების პრიორიტეტები

Java ყოველ ნაკადს პრიორიტეტს ანიჭებს, რომელიც მოცემული ნაკადის სხვა ნაკადებისადმი ქცევას განსაზღვრავს. ნაკადების პრიორიტეტები მთელი ტიპის რიცხვებით მოიცემა, რომლებიც ერთი ნაკადის უპირატეს პრიორიტეტს განაპირობებს სხვა ნაკადებთან შედარებით. პრიორიტეტის მნიშვნელობა, თავის მხრივ, გარკვეულ აზრს არ მოიცავს, რამეთუ, მაღალი პრიორიტეტის მქონე ნაკადი დაბალი პრიორიტეტის მქონე ნაკადთან შედარებით სწრაფად არ სრულდება, მით უმეტეს მაშინ, როდესაც მოცემულ მომენტში ეს უკანასკნელი ერთადერთ შესრულებად ნაკადს წარმოადგენს. როგორც წესი, ნაკადის პრიორიტეტი ერთი შესრულებადი ნაკადიდან მეორეზე გადართვის გადაწყვეტილების მისაღებად გამოიყენება. ამ მოვლენას **კონტექსტის გადართვას** უწოდებენ. წესები, რომლებიც განსაზღვრავს, თუ როდის უნდა განხორციელდეს კონტექსტის გადართვა, საკმაოდ მარტივია.

- ნაკადს შეუძლია მართვა ნებაყოფლობით დათმოს. ამისთვის საკმარისია ნაკადის შეჩერება ან ბლოკირება. ასეთი სცენარის შემთხვევაში, მოწმდება დანარჩენი ნაკადები და პროცესორის რესურსები მაქსიმალური პრიორიტეტის მქონე იმ ნაკადს გადაეცემა, რომელიც მზადაა შესასრულებლად.
- ნაკადი შეიძლება შეწყდეს სხვა, გაცილებით მეტი პრიორიტეტის მქონე ნაკადის მიერ. აღნიშნულ შემთხვევაში, დაბალი პრიორიტეტის მქონე ნაკადი უბრალოდ შეჩერებული იქნება მაღალი პრიორიტეტის მქონე ნაკადის მიერ, მიუხედავად იმისა, თუ რა როლს (ფუნქციას) ასრულებს იგი. ძირითადად კი, მაღალი პრიორიტეტის მქონე ნაკადი სრულდება მაშინ, როდესაც მას ეს „მოესურვება“. აღნიშნულ მოვლენას **შემავიწროებელი მრავალ-ამოცანიანობა** (ან მრავალამოცანიანობა პრიორიტეტებით) ეწოდება.

იმ შემთხვევაში, როდესაც ერთნაირი პრიორიტეტის მქონე ორი ნაკადი პროცესორის ციკლზე „პრეტენზიას გამოთქვამს“, სიტუაცია რთულდება. ისეთი ოპერაციული სისტემებისთვის, როგორიცაა Windows, ერთნაირი პრიორიტეტის მქონე ნაკადები ციკლურ რეჟიმში დროს იყოფენ. სხვა ტიპის ოპერაციულ სისტემებში კი ერთნაირი პრიორიტეტის მქონე ნაკადები იძულებით გადასცემენ მართვას თავიანთ „ნათესავებს“. თუ ასე არ ხდება, სხვა ნაკადების გაშვებას ადგილი არ აქვს.

5.2.3. სინქრონიზაცია

რადგან მრავალნაკადიანობა ჩვენს პროგრამებს ასინქრონული ქცევის შესაძლებლობას აძლევს, ცხადია, უნდა არსებობდეს საშუალება, რომელიც საჭიროების შემთხვევაში უზრუნველყოფს სინქრონიზაციას. მაგალითად, თუ გვსურს, რომ ორი ნაკადი ურთიერთქმედებდეს და ერთდროულად იყენებდეს მონაცემთა ისეთ რთულ სტრუქტურას, როგორიცაა მაგალითად, დაკავშირებული სიები, მაშინ ჩვენ გვჭირდება საშუალება, რომელიც ამ ნაკადებს შორის კონფლიქტს შეწყვეტს. ანუ, საჭირო ხდება ერთ ნაკადში მონაცემთა ჩაწერის შეწყვეტა, მაშინ, როდესაც სხვა ნაკადი მისი კითხვითაა დაკავებული. ამ მიზნით Java-ში რეალიზებულია საკმაოდ ელეგანტური ხერხი პროცესთა შორისი სინქრონიზაციის ძველი მოდელიდან, კერძოდ - **მონიტორი.** ეს უკანასკნელი წარმოადგენს მმართველ მექანიზმს, რომლის რეალიზება პირველად ჩარლზ ენტონი რიჩარდ ხოარდმა განახორციელა. მონიტორი შეიძლება აღვიქვათ, როგორც საკმაოდ მცირე ზომის ყუთი, რომელიც დროის ერთეულში მხოლოდ ერთ ნაკადს იღებს. როგორც კი ნაკადი შევა მონიტორში, სხვა ნაკადები უნდა დაელოდონ, სანამ იგი არ დატოვებს მონიტორს. ამგვარად, მონიტორი შეიძლება გამოყენებულ იქნეს, როგორც დაცვა იმ შემთხვევისათვის, როდესაც რამდენიმე ნაკადის მიერ ერთობლივად გამოყენებადი რესურსების

ერთდროულად მოხმარების მცდელობას ექნება ადგილი. ანუ, რესურსების გამოყენება აქ ხორციელდება არაუმეტეს ერთი ნაკადისა.

მრავალნაკადიანი სისტემების უმრავლესობა მონიტორებს იყენებს როგორც ობიექტებს, რომელთა მიღება და მართვა (მანიპულირება) შეუძლია ჩვენს პროგრამებს. Java გაცილებით „სუფთა“ გადაწყვეტილებას გვთავაზობს. ცალკეული კლასი „Monitor“ კლასის სახით აქ არ არსებობს. ნაცვლად ამისა, ყოველ ობიექტს საკუთარი არაცხადი მონიტორი გააჩნია, რომელში შესვლაც ავტომატურად ხორციელდება, როდესაც ადგილი აქვს ობიექტის სინქრონიზირებული მეთოდის გამოძახებას. როდესაც ნაკადი სინქრონიზირებული მეთოდის შიგნითაა, არცერთ სხვა ნაკადს არ შეუძლია ამ ობიექტის არცერთი სინქრონიზირებული მეთოდის გამოძახება. ეს საშუალებას გვაძლევს ვწეროთ საკმაოდ მკაფიო და მოკლე მრავალნაკადიანი კოდი, რამეთუ სინქრონიზაციის მხარდაჭერა ჩადგმულია Java დაპროგრამების ენაში.

მას შემდეგ, რაც დაყოფთ პროგრამას ცალკეულ ნაკადებად, თქვენ უნდა განსაზღვროთ, თუ რა სახით ექნებათ ამ ნაკადებს ერთმანეთთან ურთიერთობა. დაპროგრამების ზოგიერთი ენის შემთხვევაში ნაკადებს შორის ურთიერთქმედების დასამყარებლად, თქვენ იძულებული ხდებით დამოკიდებული იყოთ კონკრეტულ ოპერაციულ სისტემაზე. მსგავსი ენებისგან განსხვავებით, Java გვთავაზობს ორი ან მეტი ნაკადის ერთმანეთთან ურთიერთობის საკმაოდ მკაფიო და ეკონომიურ საშუალებას - წინასწარ განსაზღვრული იმ მეთოდების გამოძახებას, რომლებსაც ობიექტები ფლობენ. Java-ს შეტყობინებათა სისტემა ნაკადს უფლებას აძლევს შევიდეს ობიექტის სინქრონიზირებულ მეთოდში და დაელოდოს, სანამ რომელიმე სხვა ნაკადი არ მისცემს შეტყობინებას მისი (მეორე ნაკადის) მოსვლის თაობაზე.

5.2.4. Thread კლასი და Runnable ინტერფეისი

Java-ს მრავალნაკადიანი სისტემა, მისი მეთოდები და Runnable ინტერფეისი ჩადგმულია **Thread** კლასში. აღნიშნული კლასი შესრულების ნაკადის ინკაფსულირებას ახდენს. რადგან თქვენ არ გაქვთ საშუალება პირდაპირ მიმართოთ მუშა ნაკადის არამატერიალურ (არაფიზიკურ) მდგომარეობას, შესაბამისად, საქმე გაქვთ მის მოადგილესთან (proxy), რომელიც Thread კლასის ეგზემპლიარს წარმოადგენს. ახალი ნაკადის შექმნის მიზნით, თქვენმა პროგრამამ ან უნდა გააფართოვოს Thread კლასი, ან მოახდინოს Runnable ინტერფეისის რეალიზება.

Thread კლასი განსაზღვრავს რამდენიმე მეთოდს, რომლებიც ნაკადების მართვას უწყობს ხელს. ზოგიერთი მათგანი მე-31-ე ცხრილშია წარმოდგენილი.

ცხრილი 31. Thread კლასის ნაკადების მართვის მეთოდები

მეთოდი	დანიშნულება
getName	ნაკადის სახელის მიღება
getPriority	ნაკადის პრიორიტეტის მიღება
isAlive	განსაზღვრა, სრულდება თუ არა ნაკადი
join	ნაკადის დასრულების მოლოდინი
run	ნაკადის შემავალი წერტილი
sleep	მოცემული დროით ნაკადის შესრულების შეჩერება
start	run() მეთოდით ნაკადის გამოძახება

აქამდე ყველა მაგალითში ჩვენ მართვის ერთადერთ ნაკადს ვიყენებდით. ახლა კი განვიხილოთ, თუ როგორ უნდა გამოვიყენოთ Thread კლასი და Runnable ინტერფეისი ნაკადების შექმნისა და მართვის მიზნით, დაწყებული მთავარი ნაკადით, რომელიც ყველა Java პროგრამაშია.

5.2.5. მთავარი ნაკადი

როდესაც Java პროგრამა იტვირთება, მომენტალურად იწყებს შესრულებას ერთი ნაკადი. როგორც წესი, მას პროგრამის **მთავარ ნაკადს** (main thread) უწოდებენ, რადგან ეს ის ნაკადია, რომელიც ჩვენს პროგრამასთან ერთად გაიშვება. მთავარი ნაკადი მნიშვნელოვანია ორი მიზეზის გამო.

- ეს არის ნაკადი, რომლისგანაც წარმოიქმნება ყველა „შვილობილი“ ნაკადი.
- ხშირ შემთხვევაში ის უკანასკნელი ნაკადი უნდა იყოს, რომელიც შესრულების პროცესს ამთავრებს, რადგან ის სხვადასხვა დამამთავრებელ მოქმედებებს იღებს.

მიუხედავად იმისა, რომ მთავარი ნაკადი ავტომატურად იქმნება პროგრამის გაშვების დროს, მისი მართვა Thread კლასის ობიექტის საშუალებით შესაძლებელია. ამისათვის საკმარისია მასზე მიმართვა მივიღოთ currentThread() მეთოდის გამოყენებით, რომელიც Thread კლასის ღია სტატიკურ (public static) მეთოდს წარმოადგენს. მისი ჩაწერის ზოგადი ფორმა შემდეგია:

```
static Thread currentThread()
```

აღნიშნული მეთოდი მიმართვას აბრუნებს ნაკადზე, რომლიდანაც ის იქნა გამოძახებული. მივიღებთ რა, მიმართვას მთავარ ნაკადზე, მისი მართვა ჩვენ ისევე შეგვეძლება როგორც ნებისმიერი სხვა ნაკადის.

ზემოაღნიშნულის საილუსტრაციოდ **მაგალითი 7** განვიხილოთ.

პროგრამის კომპიუტერული რეალიზაცია:

```
package thread;
public class threadDemo {
public static void main(String args[]){
	Thread t=Thread.currentThread();
	System.out.println("მიმდინარე ნაკადი: " + t);
	//ნაკადის სახლის შეცვლა
	t.setName("ჩემი ნაკადი: ");
	System.out.println("სახელის შეცვლის შემდეგ: " + t);
	try{
		for(int n=5; n>0; n--){
			System.out.println(n);
			Thread.sleep(1000);
		}
	}catch(InterruptedException e){
		System.out.println("მთავარი ნაკადი შეწყვეტილია.");}}}
```

ნახ. 246

შედეგი:

```
მიმდინარე ნაკადი: Thread[main,5,main]
სახელის შეცვლის შემდეგ: Thread[ჩემი ნაკადი: ,5,main]
5
4
3
2
1
```

ნახ. 247

წარმოდგენილ პროგრამაში მიმდინარე (ჩვენ შემთხვევაში მთავარ) ნაკადზე მიმართვა currentThread() მეთოდით ხორციელდება და ეს მიმართვა t ლოკალურ ცვლადში ინახება. შემდგომ პროგრამა ნაკადის შესახებ ინფორმაციას წარმოგვიდგენს. პროგრამა setName() მეთოდს ნაკადის შიდა სახელწოდების შესაცვლელად იძახებს. ამის შემდეგ ნაკადის შესახებ ინფორმაცია კონსოლზე კვლავ გამოიტანება. შემდგომ ციფრები ხუთიდან დაწყებული კლებადობით ციკლურად გამოიტანება ყოველი სტრიქონის შემდეგ ერთი წამის დაყოვნებით. პაუზის ორგანიზება sleep() მეთოდის გამოძახებით ხდება. აღნიშნული მეთოდის არგუმენტი შეყოვნების პერიოდს მილიწამებში იძლევა. ყურადღება მიაქციეთ try/catch ბლოკს ციკლის გარშემო. sleep() მეთოდს Thread კლასში InterruptedException გამონაკლისის გადაცემა შეუძლია. ეს მაშინ შეიძლება მოხდეს, თუ სხვა რომელიმე ნაკადი შეეცდება ამ „მძინარე“ ნაკადის შესრულების შეწყვეტას. პროგრამას უბრალოდ შეყტობინების გამოტანა შეუძლია ნაკადის შეწყვეტის შემთხვევაში. t ცვლადი println() მეთოდს არგუმენტის სახით გადაეცემა. თანმიმდევრობით ის წარმოგვიდგენს ნაკადის სახელს, მის პრიორიტეტს და ჯგუფის სახელს. სტანდარტულად, მიუთითებლობის

შემთხვევაში, მთავარი ნაკადის სახელია main. მისი პრიორიტეტი ხუთის ტოლია, რაც სტანდარტულ (მიუთითებლობის შემთხვევაში) მნიშვნელობას წარმოადგენს. main ამავდროულად იმ ნაკადის ჯგუფის სახელია, რომელსაც მიმდინარე ნაკადი მიეკუთვნება. ნაკადების ჯგუფი მონაცემთა სტრუქტურას წარმოადგენს, რომელიც ნაკადების ნაკრების მდგომარეობას მართავს მთლიანობაში. ნაკადის სახელის შეცვლის შემდეგ, t ცვლადი კვლავ გამოიტანება კონსოლზე და ამჯერად, ნაკადის ახალი სახელი იწერება.

პროგრამაში გამოყენებულ Thread კლასის მეთოდებს უკეთ გავეცნოთ. sleep() მეთოდი ნაკადს აიძულებს მითითებული მილიწამებით შეაჩეროს შესრულება. მისი ჩაწერის ზოგადი ფორმა შემდეგია:

static void sleep(long მილიწამები) throws InterruptedException

მილიწამების რაოდენობა, რომლის განმავლობაშიც უნდა შეჩერდეს ნაკადის შესრულება „მილიწამები“-ს პარამეტრს გადაეცემა. აღნიშნულ მეთოდს შეუძლია InterruptedException გამონაკლისის გადაცემა.

sleep() მეთოდს ჩაწერის მეორე ფორმაც გააჩნია:

static void sleep(long მილიწამები, long ნანოწამები) throws InterruptedException

მეთოდის ამ ფორმას პერიოდის მიცემა მილიწამებსა და ნანოწამებში შეუძლია.

ნაკადის სახელის დასაყენებლად პროგრამაში setName() მეთოდია გამოყენებული. ნაკადის სახელის მიღება კი getName() მეთოდის გამოძახებით არის შესაძლებელი. ეს მეთოდები Thread კლასის წევრებია და შემდეგი სახით აღიწერება:

final void setName(String ნაკადის სახელი)

final String getName()

ზოგადად შეგვიძლია აღვნიშნოთ, რომ თქვენ ნაკადს Thread კლასის ობიექტის რეალიზებით ქმნით. ამ მიზნის განსახორციელებლად Java-ში ორი საშუალება არსებობს:

- Runnable ინტერფეისის რეალიზება.
- Thread კლასის გაფართოება.

5.2.6. Runnable ინტერფეისის რეალიზება

ნაკადის შექმნის ყველაზე მარტივ საშუალებას იმ კლასის გამოცხადება წარმოადგენს, რომელიც Runnable ინტერფეისის რეალიზებას ახდენს. შესაბამისად, ნაკადი ნებისმიერი ობიექტით შეგვიძლია შევქმნათ, რომელიც Runnable ინტერფეისის რეალიზებას განახორციელებს.

Runnable ინტერფეისის რეალიზებისთვის კლასმა მხოლოდ ერთი run() მეთოდი უნდა გამოაცხადოს, რომლის ჩაწერის ფორმა შემდეგია:

public void run()

run() მეთოდის შიგნით ჩვენ განვსაზღვრავთ კოდს, რომელიც ახალ ნაკადს შეადგენს. run() მეთოდს სხვა მეთოდის გამოძახება, სხვა კლასების გამოყენება და ცვლადების აღწერა ისევე შეუძლია, როგორც მთავარ ნაკადს.

მას შემდეგ, რაც Runnable ინტერფეისის მარეალიზებელი კლასი შეიქმნება, საჭიროა Thread ტიპის ობიექტის შექმნა ამ კლასიდან. Thread კლასში რამდენიმე კონსტრუქტორია განსაზღვრული. ის, რომელიც ჩვენ შემთხვევაში გამოიყენება, შემდეგი ფორმის მქონეა:

Tread(Runnable ნაკადის ობიექტი, String ნაკადის სახელი)

წარმოდგენილ კონსტრუქტორში „ნაკადის ობიექტი“ იმ კლასის ეგზემპლიარია, რომელიც რეალიზებას უკეთებს Runnable კლასს. ის განსაზღვრავს, თუ სად იწყება ნაკადის შესრულება. ახალი ნაკადის სახელი „ნაკადის სახელი“ - პარამეტრს გადაეცემა.

ახალი ნაკადის შექმნის შემდეგ, სანამ ჩვენ Tread კლასში გამოცხადებულ start() მეთოდს არ გამოვიძახებთ, ნაკადი არ გაეშვება. start() მეთოდის ჩაწერის ფორმა შემდეგია:

void start()

მაგალითი 8. განვიხილოთ პროგრამა, რომელიც ნაკადს ქმნის და შემდგომ შესრულებაზე უშვებს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package thread;
class NewThread implements Runnable{
	Thread t;
	NewThread(){
		//ახალი ნაკადის შექმნა
		t=new Thread(this, "სადემონსტრაციო ნაკადი");
		System.out.println("შვილობილი ნაკადი შექმნილია:" + t);
		t.start();  //ნაკადის გაშვება
		}
	//მეორე ნაკადის შესვლის წერტილი
	public void run(){
		try{
			for(int i=5; i>0; i--){
				System.out.println("შვილობილი ნაკადი : " + i);
				Thread.sleep(500);
			}
		}catch(InterruptedException e){
			System.out.println("შვილობილი ნაკადი შეწყვეტილია.");
		}
		System.out.println("შვილობილი ნაკადი დასრულებულია");}}
public class ThreadDemo {
	public static void main(String args[]){
		new NewThread(); //ახალი ნაკადის შექმნა
		try{
			for(int i=5; i>0; i--){
				System.out.println("მთავარი ნაკადი : " + i);
				Thread.sleep(1000);
			}
		}catch(InterruptedException e){
			System.out.println("მთავარი ნაკადი შეწყვეტილია.");
		}
		System.out.println("მთავარი ნაკადი დასრულებულია"); }}
```

ნახ. 248

```
შვილობილი ნაკადი შექმნილია:Thread[სადემონსტრაციო ნაკადი,5,main]
მთავარი ნაკადი : 5
შვილობილი ნაკადი : 5
შვილობილი ნაკადი : 4
მთავარი ნაკადი : 4
შვილობილი ნაკადი : 3
შვილობილი ნაკადი : 2
მთავარი ნაკადი : 3
შვილობილი ნაკადი : 1
შვილობილი ნაკადი დასრულებულია
მთავარი ნაკადი : 2
მთავარი ნაკადი : 1
მთავარი ნაკადი დასრულებულია
```

ნახ. 249

წარმოდგენილ პროგრამაში NewThread() კონსტრუქტორში Thread კლასის ახალი ობიექტი იქმნება:

```
t=new Thread(this, "სადემონსტრაციო ნაკადი");
```

აქ this ობიექტის პირველ არგუმენტად გადაცემა იმას ნიშნავს, რომ ჩვენ გვსურს, this ობიექტის run() მეთოდი ახალმა ნაკადმა გამოიძახოს. შემდგომ start() მეთოდი გამოიძახება, რის შედეგადაც run() მეთოდიდან ხდება ნაკადის შესრულებაზე გაშვება. ეს უკანასკნელი შვილობილი ნაკადის for ციკლის გაშვებას იწვევს. start() მეთოდის გამოძახების შემდეგ NewTread() კონსტრუქტორი მართვას main() მეთოდს უბრუნებს. რეოდესაც მთავარი ნაკადი მუშობას აგრძელებს, ის თავის for ციკლში შედის. ამის შემდეგ, ორივე ნაკადი პარალელურად სრულდება და ერთბირთვიან სისტემაში პროცესორის რესურსებს ისინი ერთობლივად იყენებენ საკუთარი ციკლების დასრულებამდე. ცხადია, ჩვენი შედეგები შესაძლოა განსხვავებული იყოს თქვენ მიერ ამავე პროგრამის შესრულებაზე გაშვებით მიღებული შედეგებისგან (ეს შესრულების კონკრეტულ გარემოზეა დამოკიდებული).

როგორც უკვე აღვნიშნეთ, მრავალნაკადიან პროგრამაში მთავარი ნაკადი ყველაზე ბოლოს უნდა დასრულდეს. ფაქტიურად, ზოგიერთი ძველი ვირტუალური Java მანქანისთვის (JVM), თუ მთავარი ნაკადი შვილობილ ნაკადებზე ადრე დასრულდა, შესაძლებელია Java-ს შესრულების გარემო „დაეკიდოს“. ჩვენი პროგრამა გარანტიას გვაძლევს, რომ მთავარი ნაკადი ბოლოს დასრულდება, რადგან ციკლის იტერაციებს შორის მას 1000 მილიწამი „ძინავს“, ხოლო შვილობილ ნაკადს - მხოლოდ 500 მილიწამი. ეს კი შვილობილ ნაკადს აიძულებს მთავარ ნაკადზე ადრე დასრულდეს.

5.2.7. Thread კლასის გაფართოება

ნაკადის შექმნის კიდევ ერთი გზა იმ კლასის გამოცხადებაა, რომელიც Thread კლასს აფართოებს და შემდეგ საკუთარ ეგზემპლიარს ქმნის. ეს კლასი ვალდებულია ხელახლა განსაზღვროს run() მეთოდი, რომელიც ახალი ნაკადის შესვლის წერტილს წარმოადგენს. ახალი ნაკადის შესრულებაზე გაშვების მიზნით, მან ასევე, უნდა გამოიძახოს start() მეთოდი.

ქვემოთ წარმოგიდგენთ მე-8 მაგალითის შესაბამისი პროგრამის სახეშეცვლილ ვერსიას Thread კლასის გაფართოების გამოყენებით.

პროგრამის კომპიუტერული რეალიზაცია:

```
 package thread;
class NewThread extends Thread{
 Thread t;
 NewThread(){
        //ახალი ნაკადის შექმნა
        super("სადემონსტრაციო ნაკადი");
        System.out.println("შვილობილი ნაკადი :" + this);
        start();  //ნაკადის გაშვება
        }
 //მეორე ნაკადის შესვლის წერტილი
 public void run(){
        try{
               for(int i=5; i>0; i--){
                      System.out.println("შვილობილი ნაკადი : " + i);
                      Thread.sleep(500);
               }
        }catch(InterruptedException e){
               System.out.println("შვილობილი ნაკადი შეწყვეტილია.");
        }
        System.out.println("შვილობილი ნაკადი დასრულებულია");}}
   public class ThreadDemo {
  public static void main(String args[]){
        new NewThread(); //ახალი ნაკადის შექმნა
        try{
               for(int i=5; i>0; i--){
                      System.out.println("მთავარი ნაკადი : " + i);
                      Thread.sleep(1000);
               }
        }catch(InterruptedException e){
               System.out.println("მთავარი ნაკადი შეწყვეტილია.");
        }
        System.out.println("მთავარი ნაკადი დასრულებულია"); }}
```

ნახ. 250

შედეგი:

```
შვილობილი ნაკადი :Thread[სადემონსტრაციო ნაკადი,5,main]
მთავარი ნაკადი : 5
შვილობილი ნაკადი : 5
შვილობილი ნაკადი : 4
მთავარი ნაკადი : 4
შვილობილი ნაკადი : 3
შვილობილი ნაკადი : 2
მთავარი ნაკადი : 3
შვილობილი ნაკადი : 1
შვილობილი ნაკადი დასრულებულია
მთავარი ნაკადი : 2
მთავარი ნაკადი : 1
მთავარი ნაკადი დასრულებულია
```

ნახ. 251

ორივე პროგრამის შესრულების შედეგები ერთიდაიგივეა. როგორც ხედავთ, შვილობილი ნაკადი აქ NewThread კლასის (რომელიც Thread კლასის მემკვიდრეა) ობიექტის კონსტრუირების დროს იქმნება.

ყურადღება მიაქციეთ NewThread კლასში არსებულ super() მეთოდს. ის Thread() კონსტრუქტორის შემდეგ ფორმას იძახებს:

public thread(String ნაკადის სახელი)

შესაძლოა, გაგიჩნდეთ შეკითხვა, თუ რატომ გვთავაზობს Java შვილობილი ნაკადების შექმნის ორ გზას და რომელია უკეთესი. ამ კითხვებზე პასუხები ურთიერთდაკავშირებულია. Thread კლასი რამდენიმე მეთოდს განსაზღვრავს, რომლებიც წარმოებულ კლასში შეიძლება ხელახლა განისაზღვროს. ამ მეთოდებიდან run() მეთოდის ხელახალი განსაზღვრა აუცილებლად უნდა მოხდეს. ცხადია, ეს მეთოდი Runnable ინტერფეისის რეალიზების დროსაც გვჭირდება. Java ენის ბევრ პროგრამისტს მიაჩნია, რომ კლასები მხოლოდ მაშინ უნდა გაფართოვდეს, თუ ადგილი ექნება მათ სრულყოფას ან მოდიფიცირებას. ამიტომ, თუ თქვენ Thread კლასის სხვა მეთოდების ხელახალ განსაზღვრას არ ახდენთ, უმჯობესია Runnable ინტერფეისის რეალიზება განახორციელოთ.

საბოლოო ჯამში, თუ რომელ მიდგომას (გზას) მიმართავთ, თქვენზეა დამოკიდებული.

5.2.8. ნაკადების სიმრავლის შექმნა

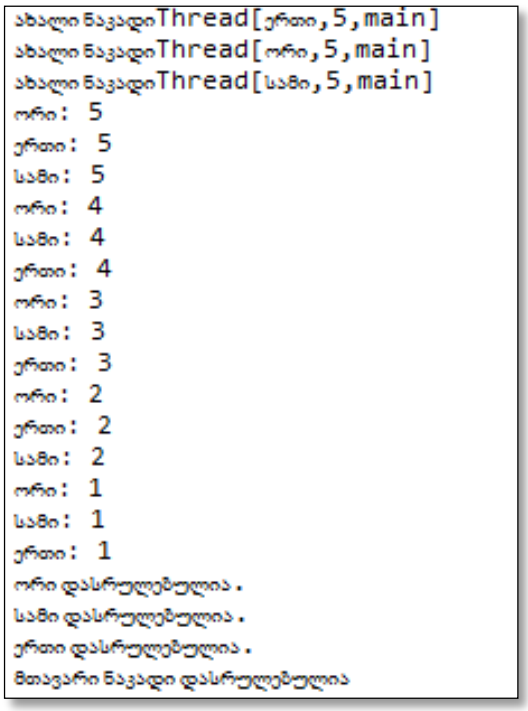
აქამდე ჩვენ მხოლოდ ორ ნაკადს ვიყენებდით: მთავარს და შვილობილს. რეალურად კი, ჩვენმა პროგრამამ შესაძლოა იმდენი ნაკადი შექმნას, რამდენიც საჭიროა.

მაგალითი 9. შევადგინოთ პროგრამა, რომელშიც სამი შვილობილი ნაკადი იქმნება.

პროგრამის კომპიუტერული რეალიზაცია:

```
package thread;
class NewThread implements Runnable{
       String name;  //ნაკადის სახელი
       Thread t;
       NewThread(String threadname){
              name=threadname;
              t=new Thread(this, name);
              System.out.println("ახალი ნაკადი" + t);
              t.start();  //ნაკადის გაშვება
       }
       // ნაკადის შესვლის წერტილი
       public void run(){
              try{
                     for(int i=5; i>0; i--){
                           System.out.println(name + ": " + i);
                           Thread.sleep(1000);}
              }catch(InterruptedException e){
                     System.out.println(name + " შეწყვეტილია.");}
              System.out.println(name + " დასრულებულია.");}}
public class MultiTreadDemo {
 public static void main(String args[]){
        new NewThread("ერთი"); //ნაკადების გაშვება
        new NewThread("ორი");
        new NewThread("სამი");
        try{
               // სხვა ნაკადების დასრულების მოლოდინი
               Thread.sleep(10000);
        }catch(InterruptedException e){
               System.out.println("მთავარი ნაკადი შეწყვეტილია");}
            System.out.println("მთავარი ნაკადი დასრულებულია"); }}
```

ნახ. 252



ნახ. 253

როგორც ხედავთ, გაშვების შემდეგ, სამივე შვილობილი ნაკადი ერთობლივად იყენებს ცენტრალური პროცესორის რესურსს. ყურადღება მიაქციეთ sleep(10000) მეთოდის main() მეთოდში გამოძახებას. ეს მთავარ ნაკადს აიძულებს 10 წამის განმავლობაში „ჩაიძინოს“ და გარანტირებულად ბოლოს (ყველა ნაკადის დასრულების შემდეგ) დასრულდეს.

5.2.9. isAlive() და join() მეთოდების გამოყენება

ზოგჯერ საჭიროა, რომ მთავარი ნაკადი ყველაზე ბოლოს დასრულდეს. აქამდე განხილულ პროგრამებში ამის მიღწევას main() მეთოდიდან sleep() მეთოდის გამოძახებით ვახდენდით, თანაც ისეთი შეყოვნებით, რომელიც გარანტიას იძლეოდა, რომ ყველა შვილობილი ნაკადი მთავარ ნაკადზე ადრე დასრულდებოდა; მაგრამ ეს არადამაკმაყოფილებელი გადაწყვეტილებაა, რომელიც სერიოზულ შეკითხვას ბადებს: საიდან შეიძლება ერთმა ნაკადმა იცოდეს, რომ მეორე დასრულდა? საბედნიეროდ, ამ კითხვაზე პასუხის გასაცემად Thread კლასი გვაძლევს საშუალებას.

არსებობს ნაკადის დასრულების განსაზღვრის ორი საშუალება. პირველ რიგში, ნაკადისთვის isAlive() მეთოდი შეგვიძლია გამოვიძახოთ. აღნიშნული მეთოდი Thread კლასშია განსაზღვრული და ჩაწერის შემდეგი ზოგადი ფორმა აქვს:

```
final Boolean   isAlive()
```

isAlive() მეთოდი ჭეშმარიტ (true) მნიშვნელობას აბრუნებს, თუ კი ნაკადი, რომლისთვისაც ის იქნა გამოძახებული, ჯერ კიდევ შესრულების პროცესშია. წინააღმდეგ შემთხვევაში, ის მცდარ (false) მნიშვნელობას აბრუნებს.

მეორეს მხრივ, არსებობს მეთოდი (რომელსაც უფრო ხშირად გამოვიყენებთ), რომელიც ყოველთვის იმ ნაკადის დასრულებას ელოდება, რომლისთვისაც ის გამოძახებულ იქნა. ეს join() მეთოდია, რომლის ჩაწერის ზოგადი ფორმაა:

final void join() throws InterruptedException

აღნიშნული მეთოდის სახელი იმ კონცეფციას წარმოგვიდგენს, რომ გამოძახებული ნაკადი ელოდება, თუ როდის შეუერთდება მას მითითებული ნაკადი. join() მეთოდის სხვა ფორმები საშუალებას გვაძლევს დავაფიქსიროთ დროის მაქსიმალური პერიოდი, რომლის განმავლობაშიც მითითებული ნაკადის დასრულებას დაველოდებით.

ქვემოთ წარმოდგენილია მე-9 მაგალითის გაუმჯობესებული ვერსია join() მეთოდის გამოყენებით, იმის გარანტად, რომ მთავარი ნაკადი ყველაზე ბოლოს დასრულდება. აქვე ადგილი აქვს isAlive() მეთოდის გამოყენებას.

პროგრამის კომპიუტერული რეალიზაცია:

```
package thread;
class NewThread implements Runnable{
        String name;  //ნაკადის სახელი
        Thread t;
        NewThread(String threadname){
                name=threadname;
                t=new Thread(this, name);
                System.out.println("ახალი ნაკადი" + t);
                t.start();  //ნაკადის გაშვება
        }
        // ნაკადის შესვლის წერტილი
        public void run(){
                try{
                        for(int i=5; i>0; i--){
                                System.out.println(name + ": " + i);
                                Thread.sleep(1000);}
                }catch(InterruptedException e){
                        System.out.println(name + " შეწყვეტილია.");}
                System.out.println(name + " დასრულებულია.");}}
```

```
public class MultiTreadDemo {
 public static void main(String args[]){
        NewThread ob1=new NewThread("ერთი");
        NewThread ob2=new NewThread("ორი");
        NewThread ob3=new NewThread("სამი");
        System.out.println("პირველი ნაკადი გაშვებულია" +
ob1.t.isAlive());
        System.out.println("მეორე  ნაკადი გაშვებულია" + ob2.t.isAlive());
        System.out.println("მესამე  ნაკადი გაშვებულია" + ob3.t.isAlive());
        try{
                // სხვა ნაკადების დასრულების მოლოდინი
                System.out.println("ნაკადების დასრულების მოლოდინი");
                ob1.t.join();
                ob2.t.join();
                ob3.t.join();
        }catch(InterruptedException e){
                System.out.println("მთავარი ნაკადი შეწყვეტილია");}
        System.out.println("პირველი ნაკადი გაშვებულია" +
ob1.t.isAlive());
        System.out.println("მეორე  ნაკადი გაშვებულია" + ob2.t.isAlive());
        System.out.println("მესამე  ნაკადი გაშვებულია" + ob3.t.isAlive());
         System.out.println("მთავარი ნაკადი დასრულებულია"); }}
```

ნახ. 254

შედეგი:

```
ახალი ნაკადიThread[ერთი,5,main]
ახალი ნაკადიThread[ორი,5,main]
ერთი: 5
ახალი ნაკადიThread[სამი,5,main]
ორი: 5
პირველი ნაკადი გაშვებულიაtrue
მეორე ნაკადი გაშვებულიაtrue
მესამე ნაკადი გაშვებულიაtrue
ნაკადების დასრულების მოლოდინი
სამი: 5
ორი: 4
ერთი: 4
სამი: 4
ერთი: 3
ორი: 3
სამი: 3
ერთი: 2
ორი: 2
სამი: 2
ერთი: 1
ორი: 1
სამი: 1
ორი დასრულებულია.
ერთი დასრულებულია.
სამი დასრულებულია.
პირველი ნაკადი გაშვებულიაfalse
მეორე ნაკადი გაშვებულიაfalse
მესამე ნაკადი გაშვებულიაfalse
მთავარი ნაკადი დასრულებულია
```

ნახ. 255

5.2.10. ნაკადების პრიორიტეტების დაყენება და მნიშვნელობის მიღება

ნაკადების დამგეგმავი ნაკადების პრიორიტეტებს იმ გადაწყვეტილების მისაღებად იყენებს, თუ რომელ ნაკადს მისცეს „მუშაობის უფლება“. თეორიულად, მაღალი პრიორიტეტის მქონე ნაკადები პროცესორის უფრო მეტ დროს იღებენ, ვიდრე დაბალი პრიორიტეტის ნაკადები. პრაქტიკულად, პროცესორული დროის მოცულობა, რომელსაც ნაკადი იღებს, ამ უკანასკნელის პრიორიტეტის გარდა, მრავალ ფაქტორზეა დამოკიდებული (მაგალითად, როგორ ახდენს მრავალამოცანიანობის რეალიზებას ოპერაციული სისტემა). ამასთან, მაღალპრიორიტეტულმა ნაკადმა შეიძლება დაბალპრიორიტეტული ნაკადი გამოაძევოს კიდეც. მაგალითად, იმ შემთხვევაში, როდესაც დაბალპრიორიტეტული ნაკადი მუშაობს, ხოლო მაღალი პრიორიტეტის მქონე ნაკადი შეწყვეტილი მუშაობის გაგრძელებას აპირებს.

თეორიულად, თანაბარი პრიორიტეტის ნაკადებმა ცენტრალურ პროცესორზე თანაბარი წვდომა უნდა იქონიონ. რეალურად კი, უსაფრთხოების მიზნებიდან გამომდინარე, თანაბარი პრიორიტეტის ნაკადებმა მართვა სხვადასხვა ხარისხით უნდა მიიღონ.

ნაკადის პრიორიტეტის დასაყენებლად Thread კლასის setPriority() მეთოდი გამოიყენება, რომლის ჩაწერის ზოგადი ფორმა შემდეგია:

final void setPriority(დონე)

პარამეტრი „დონე“ გამოძახებული ნაკადის პრიორიტეტის ახალი დონის განსაზღვრას გულისხმობს. მისი მნიშვნელობა MIN_PRIORITY-დან MAX_ PRIORITY დიაპაზონის საზღვრებ-ში აიღება. ეს მნიშვნელობები, დღესდღეობით, შესაბამისად: 1-ის და 10-ის ტოლია. იმისათვის, რომ ნაკადს სტანდარტული პრიორიტეტი დავუბრუნოთ, საკმარისია NORM_ PRIORITY-ის მითითება (მისი მნიშვნელობა 5-ის ტოლია). ეს პრიორიტეტები სტატიკური ფინალური (static final) ცვლადების სახით Thread კლასშია განსაზღვრული.

ნაკადის მიმდინარე პრიორიტეტის მნიშვნელობის მისაღებად Thread კლასში განსაზღვრულ getPriority() მეთოდს ვიძახებთ, რომლის ჩაწერის ფორმაა:

final int getPriority()

დავალება

1. *განმარტეთ მრავალამოცანიანობის ტიპები.*
2. *განმარტეთ კონტექსტის გადართვის მოვლენა.*
3. *შეადგინეთ პროგრამა, რომელიც მარტივ ნაკადს ქმნის და შემდეგ უშვებს მას შესრულებაზე.*
4. *განმარტეთ ნაკადთაშორისი კონფლიქტის არსი.*
5. *შეადგინეთ პროგრამა, რომელიც კონსოლზე გამოიტანს ნაკადის საწყის სახელს, მის ახალ სახელს და სამ მილიწამიანი ინტერვალით დაბეჭდავს ერთიდან 10-ის ჩათვლით ნატურალური რიცხვების კვადრატების მნიშვნელობებს.*
6. *შეადგინეთ სამი შვილობილი ნაკადის შემცველი პროგრამა, რომელიც გვიჩვენებს მათი შესრულების თანმიმდევრობას.*

5.3. ნაკადების სინქრონიზაციის უზრუნველყოფა

5.3.1. სინქრონიზირებული მეთოდების გამოყენება

როდესაც ორ ან მეტ ნაკადს ერთსა და იმავე ერთობლივად გამოსაყენებელ რესურსთან აქვს წვდომა, იქმნება მოთხოვნა იმისა, რომ ერთი და იგივე დროს რესურსი მხოლოდ ერთმა ნაკადმა გამოიყენოს. ამ მოთხოვნის უზრუნველყოფის პროცესს **სინქრონიზაცია** ეწოდება. სინქრონიზაციის გასაღებს **მონიტორის** კონცეფცია წარმოადგენს. მონიტორი ობიექტია, რომელიც ურთიერთგამომრიცხავი ბლოკირების (mutually exclusive block - mutex) როლში გამოიყენება. მას მეორენაირად, **მუტექსს** უწოდებენ. ერთსა და იმავე დროს მონიტორს მხოლოდ ერთი ნაკადი ფლობს. როდესაც ნაკადი ბლოკირებას მოითხოვს, ამბობენ, რომ ის მონიტორში შედის. ყველა სხვა ნაკადი, რომელიც დაბლოკილ მონიტორში შესვლას შეეცდება, შეჩერებული იქნება მანამდე, სანამ პირველი ნაკადი მონიტორიდან არ გამოვა. ყველა დანარჩენ ნაკადზე ვამბობთ, რომ ისინი მონიტორს ელოდებიან. ნაკადს, რომელიც მონიტორს ფლობს, „სურვილის შემთხვევაში“, ხელახლა შეუძლია მასში შესვლა.

Java-ში სინქრონიზაცია საკმაოდ მარტივია, რადგან ყველა ობიექტს გააჩნია საკუთარი, (მასთან ასოცირებული) არაცხადი მონიტორი. ობიექტის მონიტორში შესასვლელად საკმარისია synchronized საკვანძო სიტყვით მოდიფიცირებული მეთოდი უბრალოდ გამოვიძახოთ. როდესაც ნაკადი სინქრონიზირებულ მეთოდშია, ყველა სხვა ნაკადი, რომელიც მის გამოძახებას იმავე ეგზემპლიარში შეეცდება, მოლოდინის რეჟიმში აღმოჩნდება. იმისათვის, რომ მონიტორიდან გამოვიდეს და მართვა მომლოდინე ნაკადს გადასცეს, მონიტორის მფლობელი სინქრონიზირებული მეთოდიდან უბრალოდ მართვას აბრუნებს.

სინქრონიზაციის არსის უკეთ გასაგებად განვიხილოთ **მაგალითი 10.** ის წარმოგვიდგენს პროგრამას, რომელიც სამი მარტივი კლასისგან შედგება. პირველი მათგანი - Callme კლასი call() მეთოდს შეიცავს. აღნიშნული მეთოდი String კლასის msg პარამეტრს იღებს და ცდილობს msg სტრიქონი კვადრატულ ფრჩხილებში გამოიტანოს. გახსნილი კვადრატული ფრჩხილისა და msg სტრიქონის გამოტანის შემდეგ call() მეთოდი იძახებს Thread.sleep(1000) მეთოდს, რომელიც მიმდინარე ნაკადს ერთი წამით აჩერებს.

მეორე Caller კლასის კონსტრუქტორი Callme და String კლასების ეგზემპლიარებზე მიმართვას იღებს. ეს უკანასკნელნი, შესაბამისად, target და msg ცვლადებში ინახება. კონსტრუქტორი ასევე, ახალ ნაკადს ქმნის, რომელსაც run() მეთოდი იძახებს. ნაკადი მაშინვე იტვირთება. Caller კლასის run() მეთოდი Caller კლასის target ეგზემპლიარისთვის იძახებს call() მეთოდს და გადასცემს მას msg სტრიქონს. Synch კლასი ქმნის Callme კლასის ერთადერთ ეგზემპლიარს და Caller კლასის სამ ეგზემპლიარს. ყოველ მათგანს შეტყობინების უნიკალური სტრიქონი გააჩნია. Callme კლასის ერთი ეგზემპლიარი ყოველ Caller კონსტრუქტორს გადაეცემა. და ბოლოს, ამ

ყველაფრის გათვალისწინებით, ჩავწერთ პროგრამას, რომელიც არ იქნება სინქრონიზებული და სინქრონიზაციის ელემენტებით მის სრულყოფას შემდგომში შევეცდებით.

პროგრამის კომპიუტერული რეალიზაცია:

```
package tread;
class Callme{
	void call(String msg){
		System.out.print("[" + msg);
		try{
			Thread.sleep(1000);
		}catch(InterruptedException e){
				System.out.println(" შეწყვეტილია.");}
			System.out.println("]");    }}
class Caller implements Runnable{
	String msg;
	Callme target;
	Thread t;
	public Caller(Callme targ, String s){
		target=targ;
		msg=s;
		t=new Thread(this);
		t.start();}
	public void run(){
		target.call(msg);
	}}
public class Synch {
	public static void main(String args[]){
		Callme target=new Callme();
		Caller ob1=new Caller(target, "კეთილი იყოს თქვენი მობრძანება ");
		Caller ob2=new Caller(target, "სინქრონიზირებულ ");
		Caller ob3=new Caller(target, "სამყაროში");
		//ნაკადის დასრულების მოლოდინი
		try{
			ob1.t.join();
			ob2.t.join();
			ob3.t.join();
		 }catch(InterruptedException e){
			System.out.println("შეწყვეტილია"); }}}
```

ნახ. 256

შედეგი:

```
[სინქრონიზირებულ [სამყაროში[კეთილი იყოს თქვენი მობრძანება ]
]
]
```

ნახ. 257

როგორც ვხედავთ, sleep() მეთოდის გამოძახებით, call() მეთოდი სხვა ნაკადის შესრულებაზე გადაერთვება. ეს კი, შეტყობინებათა სამივე სტრიქონის შერეულ გამოტანას განაპირობებს. ასეთ მდგომარეობას ნაკადთაშორის **კონფლიქტს** უწოდებენ. ჩვენ შემთხვევაში, სამი ნაკადი ეჯიბრება ერთმანეთს.

ჩვენი პროგრამის გასასწორებლად საჭიროა call() მეთოდისადმი წვდომა სერიალიზირებული გავხადოთ, რაც იმას ნიშნავს, რომ ერთსა და იმავე დროს ამ მეთოდზე წვდომის უფლება მხოლოდ ერთ ნაკადს უნდა მივცეთ. ამისათვის საკმარისია call() მეთოდს წინ საკვანძო სიტყვა synchronized დავურთოთ:

```
synchronized void call(String msg){
…… }
```

ზემოთ აღწერილი ცვლილების გათვალისწინებით, პროგრამის შესრულების შედეგი მიიღებს ნახ.258-ზე წარმოდგენილ სახეს.

შედეგი:

```
[სინქრონიზირებულ ]
[სამყაროში]
[კეთილი იყოს თქვენი მობრძანება ]
```

ნახ. 258

5.3.2. synchronized ოპერატორი

მართალია, ჩვენს კლასებში სინქრონიზირებული მეთოდების გამოყენება სინქრონიზაციის მარტივი და ეფექტური საშუალებაა, მაგრამ ის ყოველთვის არ მუშაობს. ამ ფაქტის გასაგებად შემდეგი შემთხვევა განვიხილოთ. დავუშვათ, გვსურს სინქრონიზირება გავუკეთოთ კლასის იმ ობიექტებზე წვდომას, რომლებიც მრავალნაკადიანი წვდომისთვის არ იყო განკუთვნილი. ეს ნიშნავს, რომ კლასი სინქრონიზირებულ მეთოდებს არ იყენებს. უფრო მეტიც, დავუშვათ, კლასი ჩვენ მიერ არ არის დაწერილი და არ გვაქვს მის საწყის კოდთან წვდომა. მაშასადამე, synchronized საკვანძო სიტყვას კლასის შესაბამის მეთოდებს ჩვენ ვერ დავურთავთ. როგორ არის შესაძლებელი ასეთი კლასის ობიექტებზე წვდომის სინქრონიზაცია? საბედნიეროდ, ამ პრობლემის საკმაოდ მარტივი გადაწყვეტა არსებობს: ასეთი კლასის მეთოდების გამოძახებას ჩვენ synchronized ბლოკში ჩავრთავთ.

synchronized ოპერატორის ზოგადი ფორმა შემდეგია:

```
synchronized(ობიექტი) {
//სინქრონიზაციას დაქვემდებარებული ოპერატორები
}
```

აქ „ობიექტი“ სინქრონიზირებულ ობიექტზე მიმართვას წარმოადგენს. synchronized ბლოკი იმის გარანტიას იძლევა, რომ ობიექტის მეთოდის გამოძახება მხოლოდ მაშინ მოხდება, როდესაც მიმდინარე ნაკადი ობიექტის მონიტორში წარმატებით შევა.

ქვემოთ წარმოდგენილია მე-10 მაგალითის ამოხსნის ალტერნატიული ვერსია, რომელიც run() მეთოდში სინქრონიზირებული ბლოკის გამოყენებას მოიცავს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package tread;
class Callme{
        void call(String msg){
              System.out.print("[" + msg);
              try{
                    Thread.sleep(1000);
              }catch(InterruptedException e){
                          System.out.println("InterUpted");}
                   System.out.println("]");    }}
class Caller implements Runnable{
       String msg;
       Callme target;
       Thread t;
       public Caller(Callme targ, String s){
              target=targ;
              msg=s;
              t=new Thread(this);
              t.start();}
       // call() მეთოდის სინქრონიზირებული გამოძახებები
       public void run() {
              synchronized (target) {  // სინქრონიზირებული ბლოკი
              target.call(msg); }}}
```

```
class Synch {
       public static void main(String args[]){
              Callme target=new Callme();
              Caller ob1=new Caller(target, "კეთილი იყოს თქვენი მობრძანება
");
              Caller ob2=new Caller(target, "სინქრონიზირებულ ");
              Caller ob3=new Caller(target, "სამყაროში");
              //ნაკადის დასრულების მოლოდინი
              try{
                     ob1.t.join();
                     ob2.t.join();
                     ob3.t.join();
               }catch(InterruptedException e){
                     System.out.println("შეწყვეტილია"); }}}
```

ნახ. 259

აქ call() მეთოდი synchronized საკვანძო სიტყვით მოდიფიცირებული არ არის. ამის ნაცვლად, Caller კლასის run() მეთოდში synchronized ოპერატორია გამოყენებული. ეს საშუალებას გვაძლევს იგივე კორექტული შედეგი მივიღოთ, რაც წინა შემთხვევაში, რამეთუ ყოველი ნაკადი თავის წინა ნაკადის დასრულებას ელოდება.

5.3.3. ნაკადთაშორისი კომუნიკაციები

ობიექტების არაცხადი მონიტორების გამოყენება უდავოა, რომ Java-ს მძლავრ საშუალებას წარმოადგენს, თუმცა ნაკადების კონტროლის უკეთეს დონეს ნაკადთაშორისი კომუნიკაციების საშუალებით შეგვიძლია მივაღწიოთ. ნაკადების კიდევ ერთი უპირატესობა იმაში მდგომარეობს, რომ ისინი ე.წ. „გამოკითხვას“ გამორიცხავენ. როგორც წესი, გამოკითხვა ციკლის სახით არის რეალიზებული, რომელიც ამა თუ იმ დონის პერიოდული შემოწმებისთვის გამოიყენება. როგორც კი პირობა აღმოჩნდება ჭეშმარიტი, განსაზღვრული მოქმედება სრულდება. ეს კი პროცესორის დროის გახარჯვას იწვევს. მაგალითისთვის კლასიკური პრობლემა განვიხილოთ, როდესაც ერთი ნაკადი გარკვეულ მონაცემებს ქმნის, ხოლო მეორე - მათ იღებს. უფრო მეტი თვალსაჩინოებისთვის დავუშვათ, რომ მონაცემთა მომწოდებელი ელოდება, თუ როდის დაასრულებს მუშაობას მომხმარებელი, რომ შემდგომ ახალი მონაცემები შექმნას. გამოკითხვის სისტემებში მონაცემთა მომხმარებელი პროცესორის არაერთ ციკლს ხარჯავს მომწოდებლის მონაცემთა მოლოდინში. როგორც კი მომწოდებელი მუშაობას ასრულებს, მონაცემთა მომხმარებლის მუშაობის დასრულების მოლოდინში ის იწყებს გამოკითხვას, რაც პროცესორის ციკლების გახარჯვას იწვევს. ცხადია, აღნიშნული სიტუაცია არასასურველია.

გამოკითხვის თავიდან ასაცილებლად Java ნაკადთაშორისი კომუნიკაციის საკმაოდ ელეგანტურ მექანიზმს იყენებს wait(), notify() და notifyAll() მეთოდების საშუალებით. ეს მეთოდები Object კლასში რეალიზებულია როგორც ფინალური და შესაბამისად, წვდომადია ყველა კლასისთვის. სამივე მეთოდი მხოლოდ სინქრონიზებული კონტექსტიდან გამოიძახება. კომპიუტერული მეცნიერების თვალთახედვით, ეს მეთოდები კონცეპტუალურად საკმაოდ რთულია, თუმცა მათი გამოყენების წესები მარტივია.

- wait() მეთოდი გამომძახებელ ნაკადს აიძულებს გასცეს მონიტორი და შესრულება მანამდე შეაჩეროს, სანამ რომელიმე სხვა ნაკადი იმავე მონიტორში არ შევა და არ გამოიძახებს notify() მეთოდს.
- notify() მეთოდი იმ ნაკადის მუშაობას აღადგენს, რომელმაც wait() მეთოდი იმავე ობიექტში გამოიძახა.

- notifyAll() მეთოდი კი ყველა იმ ნაკადის მუშაობას აღადგენს, რომელთაც wait() მეთოდი იმავე ობიექტში გამოიძახეს. ნაკადებიდან წვდომა მხოლოდ ერთს ეძლევა.

  ეს მეთოდები Object კლასში შემდეგი სახით არის განსაზღვრული:

```
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

არსებობს wait() მეთოდის დამატებითი ფორმები, რომლებიც მოლოდინის დროის მითითების საშუალებას გვაძლევს. მართალია, wait() მეთოდი მანამდე იცდის, სანამ notify() ან notifyAll() მეთოდები არ იქნება გამოძახებული, მაგრამ არსებობს ალბათობა იმისა, რომ იშვიათ შემთხვევებში მომლოდინე ნაკადი შეიძლება ყალბი სიგნალით აღდგეს. ამასთან, მომლოდინე ნაკადი notify() ან notifyAll() მეთოდების გამოძახების გარეშე აღდგება. ამ შემთხვევის გამო, რომლის ალბათობა საკმაოდ მცირეა, Oracle რეკომენდაციას გვაძლევს, wait() მეთოდის გამოძახებები შევასრულოთ ციკლში, რომელიც ამოწმებს პირობას, რის მიხედვითაც ნაკადი მოლოდინის რეჟიმშია.

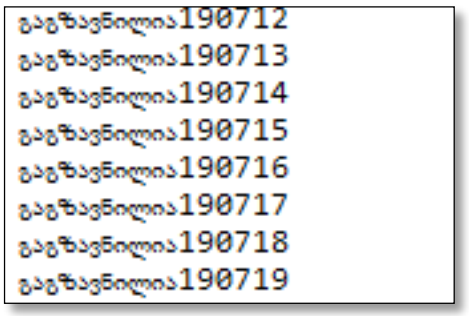
მაგალითი 11. განვიხილოთ მარტივი შემთხვევა, სადაც „მომწოდებელი/მომხმარებლის“ ამოცანა არაკორექტულად არის რეალიზებული. პროგრამა ოთხ კლასს მიოიცავს: Q - რიგი, რომლის სინქრონიზებაც უნდა მოვახდინოთ; Producer - ნაკადი-ობიექტი, რომელიც რიგის ელემენტებს ქმნის, Consumer - ნაკადი-ობიექტი, რომელიც რიგის ელემენტებს იღებს და PC - მცირე კლასი, რომელიც Q, Producer და Consumer კლასების ობიექტებს ქმნის.

პროგრამის კომპიუტერული რეალიზაცია:

```
package tread;
//არასწორი რეალიზაცია
class Q{
	int n;
	synchronized int get(){
		System.out.println("მიღებულია" + n);
		return n;
	}
	synchronized void put(int n){
		this.n=n;
		System.out.println("გაგზავნილია" + n);
	}}
  class Producer implements Runnable{
	  Q q;
	  Producer(Q q){
		this.q=q;
		new Thread(this, "მომწოდებელი").start();
	  }
	  public void run(){
		int i=0;
		while(true){
			q.put(i++);
		}}}
  class Consumer implements Runnable{
	  Q q;
	  Consumer(Q q){
		this.q=q;
		new Thread(this, "მომხმარებელი").start();
	  }
	  public void run(){
		while(true){
			q.get();
	  }}}

public class PC {
	public static void main(String args[]){
		Q q=new Q();
		new Producer(q);
		new Consumer(q);
System.out.println("პროგრამის გასაჩერებლად Ctr+c კლავიშების კომბინაციას
დააწექით"); }}
```

ნახ. 260

შედეგი (არასწორი):



ნახ. 261

პროცესორისა და პროგრამის ჩატვირთვის სისწრაფიდან გამომდინარე, თქვენ შესაძლოა სხვა შედეგი მიიღოთ.

Java ენაზე ამ პროგრამის სწორად ჩაწერის ერთადერთი საშუალებაა სიგნალების ორივე მიმართულებით გადასაგზავნად wait() და notify() მეთოდების გამოყენება.

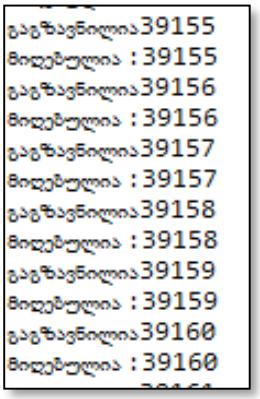
პროგრამის კომპიუტერული რეალიზაცია:

```
package tread;
// ამოცანის სწორი რეალიზაცია
class Q{
       int n;
       boolean valueSet=false;
       synchronized int get(){
              while(!valueSet)
              try{
                     wait();
              }catch(InterruptedException e){
                     System.out.println("InterUptedException  დაჭერილია"); }
              System.out.println("მიღებულია :"  + n);
              valueSet=false;
              notify();
              return n;}

       synchronized void put(int n){
              while(valueSet)
                     try{
                            wait();
                     }catch(InterruptedException e){
                     System.out.println("InterUptedException  დაჭერილია"); }
              this.n=n;
              valueSet=true;
              System.out.println("გაგზავნილია" + n);
              notify();    }}


         public void run(){
                int i=0;
                while(true){
                       q.put(i++);
                }}}
```

```
class Producer implements Runnable{
        Q q;
        Producer(Q q){
               this.q=q;
               new Thread(this, "მომწოდებელი").start();
        }
        public void run(){
               int i=0;
               while(true){
                      q.put(i++);
               }}}
  class Consumer implements Runnable{
        Q q;
        Consumer(Q q){
               this.q=q;
               new Thread(this, "მომხმარებელი").start();
        }
        public void run(){
               while(true){
                      q.get();
        }}}
public class PC {
       public static void main(String args[]){
             Q q=new Q();
             new Producer(q);
             new Consumer(q);
System.out.println("პროგრამის გასაჩერებლად Ctr+c კლავიშების კომბინაციას დააწექით");
}}
```

ნახ. 262

შედეგი:



ნახ. 263

წარმოდგენილ პროგრამაში wait() მეთოდი get() მეთოდიდან გამოიძახება. ეს ნაკადის მუშაობას მანამდე აჩერებს, სანამ Producer კლასის ობიექტი არ გვაცნობებს, რომ მონაცემები წაკითხულია. მონაცემთა მიღების შემდეგ get() მეთოდი notify() მეთოდს იძახებს. ეს კი Producer კლასის ობიექტს ატყობინებს, რომ ყველაფერი წესრიგშია და რიგში მონაცემთა შემდეგი ელემენტის მოთავსება შესაძლებელია. put() მეთოდში wait() მეთოდი ნაკადის შესრულებას

მანამდე აჩერებს, სანამ Consumer კლასის ობიექტი რიგიდან ელემენტს არ ამოიღებს. როდესაც ნაკადის შესრულება აღსდგება, მონაცემთა შემდეგი ელემენტი მოთავსდება რიგში და notify() მეთოდი გამოიძახება. ეს Consumer კლასის ობიექტს ატყობინებს, რომ მას ელემენტის რიგიდან ამოღება შეუძლია. ამჯერად სინქრონიზაცია კორექტულად მუშაობს.

5.3.4. ურთიერთბლოკირება

შეცდომების განსაკუთრებულ ტიპს, რომელიც Java-ში მრავალამოცანიანობას უკავშირდება, უნდა შევეცადოთ თავი ავარიდოთ. ასეთ განსაკუთრებულ შემთხვევას ნაკადების ურთიერთ-ბლოკირება (deadlock) წარმოადგენს, რომელიც თავს მაშინ იჩენს, როდესაც ნაკადები სინ-ქრონიზირებულ ობიექტებზე ციკლურად დამოკიდებული არიან. დავუშვათ, რომ ერთი ნაკადი X ობიექტის მონიტორში შედის, ხოლო მეორე - Y ობიექტის მონიტორში. თუ X ობიექტის ნაკადი შეეცდება გამოიძახოს Y ობიექტის ნებისმიერი სინქრონიზებული მეთოდი, ის აუცილებლად დაიბლოკება. საწინააღმდეგო შემთხვევაში, თუ Y ობიექტის ნაკადი შეეცდება გამოიძახოს X ობიექტის ნებისმიერი სინქრონიზებული მეთოდი, მაშინ ნაკადს უსასრულო დროით მოუწევს ლოდინი, რადგან X ობიექტზე წვდომის მისაღებად მან საკუთარი ბლოკი უნდა მოხსნას Y ობიექტიდან იმისთვის, რომ პირველმა ნაკადმა მუშაობა შეძლოს. ურთიერთბლოკირება იმ შეცდომას წარმოადგენს, რომლის გამართვა საკმაოდ რთულია შემდეგი ორი მიზეზის გამო:

- ზოგადად, ის საკმაოდ იშვიათია, თუმცა ადგილი აქვს, როდესაც ორი ნაკადის შესრულება ზუსტად ემთხვევა დროში.
- მან შესაძლოა თავი მაშინ იჩინოს, როდესაც ამ პროცესში მონაწილეობას ორზე მეტი ნაკადი და ორზე მეტი სინქონიზებული ობიექტი იღებს.

ნაკადების ურთიერთბლოკირების საილუსტრაციოდ განვიხილოთ **მაგალითი 12**. შევადგი-ნოთ პროგრამა, რომელიც ორ A და B კლასს ქმნის შესაბამისად foo() და bar() მეთოდებით. სხვა კლასის მეთოდის გამოძახების მცდელობისას ადგილი აქვს მათ შეჩერებას. Deadlock სახელის მქონე მთავარი კლასი A და B კლასების ეგზემპარებს ქმნის და შემდეგ მეორე ნაკადს უშვებს, რომელიც ურთიერთბლოკირების მდგომარეობას აყენებს. foo() და bar() მეთოდები sleep() მეთოდს ურთიერთბლოკირების სტიმულირებისთვის იყენებენ.

პროგრამის კომპიუტერული რეალიზაცია:

```
package thread;
class A{
	synchronized void foo(B b){
	String name=Thread.currentThread().getName();
	System.out.println(name + "შევიდა მეთოდში A.foo");
	try{
		Thread.sleep(1000);
	}catch(Exception e){
		System.out.println("A შეწყვეტილია");
	}
	System.out.println(name + "ცდილობს გამოიძახოს B.last()");
	b.last();
	}
	synchronized void last(){
		System.out.println("A.last-ში");
	}
}
class B{
	synchronized void bar(A a){
	String name=Thread.currentThread().getName();
	System.out.println(name + "შევიდა მეთოდში B.bar");
	try{
		Thread.sleep(1000);
	}catch(Exception e){
		System.out.println("B შეწყვეტილია");
	}
	System.out.println(name + "ცდილობს გამოიძახოს A.last()");
	a.last();
	}
	synchronized void last(){
		System.out.println("A.last-ში");}}
class Deadlock implements Runnable {
	 A a= new A();
	 B b= new B();
	 Deadlock(){
Thread.currentThread().setName("MainThread");
		 Thread t=new Thread(this,"RacingThread");
		 t.start();
		 a.foo(b); //ნაკადის შიგნით ბლოკირების მიღება
		 System.out.println("უკან მთავარ ნაკადში");
	 }
```

```
public void run(){
                b.bar(a); //სხვა ნაკადში b-ს ბლოკირების მიღება
                System.out.println("უკან სხვა  ნაკადში");
        }
        public static void main(String args[]){
                new Deadlock();
        }
}
```

ნახ. 264

შედეგი:

```
MainThreadშევიდა მეთოდში A.foo
RacingThread შევიდა მეთოდში B.bar
MainThread ცდილობს გამოიძახოს B.last()
RacingThread ცდილობს გამოიძახოს A.last()
```

ნახ. 265

რადგან ზემოთ წარმოდგენილი პროგრამა ბლოკირებულია, მის დასასრულებად მოგიწევთ Ctr+C კლავიშების კომბინაციის გამოყენება. ეს პროგრამა არასდროს დასრულდება. ამ მაგალითიდან შეგვიძლია დავასკვნათ, თუ თქვენი მრავალამოცანიანი პროგრამა „დაეკიდა“, პირველ რიგში, ურთიერთბლოკირების შესაძლებლობა უნდა შეამოწმოთ.

5.3.5. ნაკადების დროებით შეჩერება, აღდგენა და გაჩერება

ზოგჯერ ნაკადების შესრულების შეჩერების აუცილებლობა წარმოიქმნება. მაგალითად, რომელიმე ნაკადი შესაძლოა მიმდინარე დროს საჩვენებლად იქნეს გამოყენებული. თუ მომხმარებელს საათი არ ჭირდება, ეს ნაკადი შეიძლება შევაჩეროთ. ნებისმიერ შემთხვევაში, ნაკადის შეჩერება მარტივი საქმეა. შეჩერებული ნაკადის შესრულების აღდგენაც პრობლემას არ წამროადგენს.

ნაკადების შეჩერებისა და აღდგენისსათვის Thread კლასის შესაბამისი მეთოდები: suspend() და resume() გამოიყენება. მათი ჩაწერის ზოგადი ფორმა შემდეგია:

final void suspend()

final void resume()

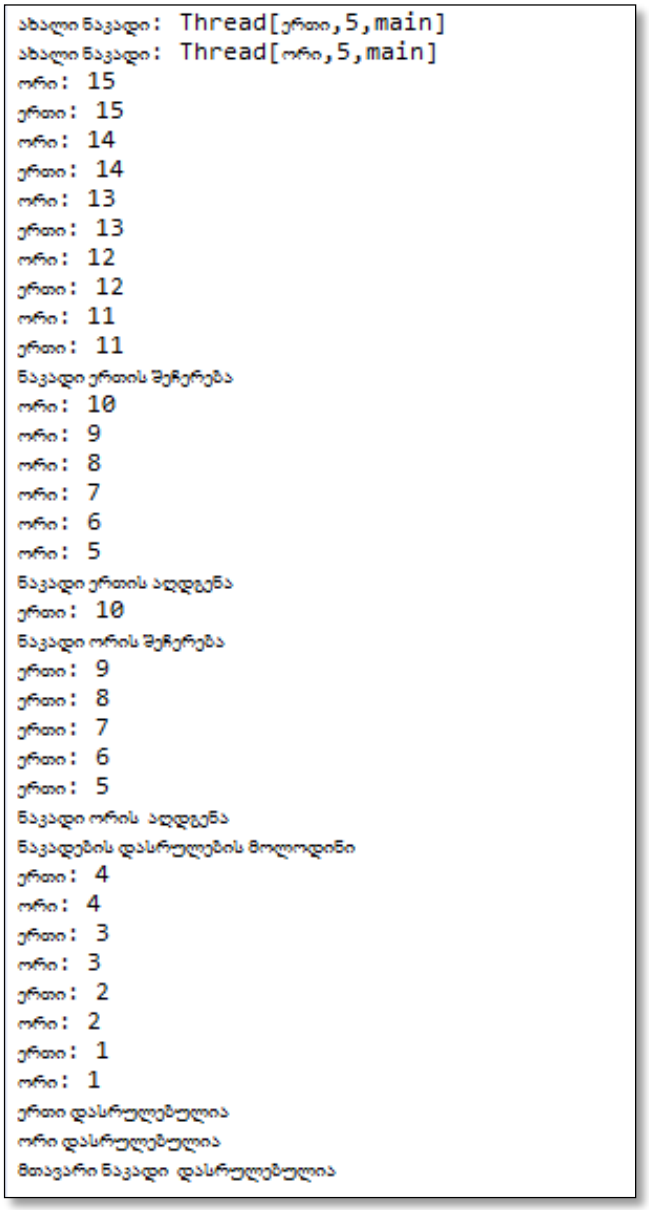
მაგალითი 13. შევადგინოთ suspend() და resume() მეთოდების გამოყენების სადემონსტრაციო პროგრამა.

პროგრამის კომპიუტერული რეალიზაცია:

```
package thread;
class NewThread implements Runnable{
	String name;  //ნაკადის სახელი
	Thread t;
	NewThread(String threadname){
		name=threadname;
		t=new Thread(this, name);
		System.out.println("ახალი ნაკადი: " + t);
		t.start();  } //ნაკადის გაშვება
	//ნაკადის შესვლის წერტილი
	public void run(){
		try{
			for(int i=15; i>0; i--){
				System.out.println(name + ": " + i);
				Thread.sleep(200);}
		}catch(InterruptedException e){
			System.out.println(name + " შეწყვეტილია"); }
		System.out.println(name + " დასრულებულია"); }}
public class SuspendResume {
	public static void main(String [] args){
		NewThread ob1=new NewThread("ერთი");
		NewThread ob2=new NewThread("ორი");
		try{
			Thread.sleep(1000);
			ob1.t.suspend();
			System.out.println("ნაკადი ერთის შეჩერება");
			Thread.sleep(1000);
			ob1.t.resume();
			System.out.println("ნაკადი ერთის აღდგენა");
			ob2.t.suspend();
			System.out.println("ნაკადი ორის შეჩერება");
			Thread.sleep(1000);
			ob2.t.resume();
			System.out.println("ნაკადი ორის  აღდგენა");
		}catch(InterruptedException e){

System.out.println("მთავარი ნაკადი  შეწყვეტილია"); }

		//ნაკადების დასრულების მოლოდინი
		try{
			System.out.println("ნაკადების დასრულების
მოლოდინი");
			ob1.t.join();
			ob2.t.join();
		}catch(InterruptedException e){
			System.out.println("მთავარი ნაკადი  შეწყვეტილია"); }
			System.out.println("მთავარი ნაკადი  დასრულებულია");
}
```

ნახ. 266

შედეგი:



ნახ. 267

თქვენ შესაძლოა, განსხვავებული შედეგები მიიღოთ, რადგან ეს პროცესორის სიჩქარესა და ჩატვირთვაზეა დამოკიდებული.

Thread კლასი stop()მეთოდს განსაზღვრავს, რომლიც ნაკადს აჩერებს. მისი სიგნატურაა:

```
final void stop()
```

შეჩერებული ნაკადის resume() მეთოდით აღდგენა აღარ ხდება.

5.3.6. ნაკადის მდგომარეობის მიღება

როგორც უკვე აღვნიშნეთ, ნაკადი რამდენიმე სხვადასხვა მდგომარეობაში შეიძლება იმყოფებოდეს. ნაკადის მიმდინარე მდგომარეობა Thread კლასში განსაზღვრული getState() მეთოდით შეგვიძლია მივიღოთ, რომლის ზოგადი ფორმაა:

```
Thread. State getState()
```

მეთოდი Thread. State ტიპის მნიშვნელობას აბრუნებს, რაც გამოძახების მომენტში ნაკადის მდგომარეობაზე მიუთითებს.

მნიშვნელობები, რაც getState() მეთოდმა შეიძლება დააბრუნოს, 32-ე ცხრილშია წარმოდგენილი.

Thread კლასის ეგზემპარისა და getState() მეთოდის გამოყენებით, ჩვენ ნაკადის მდგომარეობის მიღება შეგვძლია. მაგალითად, ქვემთ წარმოდგენილი კოდი განსაზღვრავს, thrd სახელის მქონე ნაკადი იმყოფება თუ არა RUNNABLE მდგომარეობაში getState() მეთოდის გამოძახების დროს.

```
Thread.State st = thrd. getState();
if(ts = =  Thread.State.RUNNABLE) //…
```

ცხრილი 32 getState() მეთოდის მიერ დაბრუნებული მნიშვნელობები

მნიშვნელობა	მდგომარეობა
BLOCKED	ნაკადმა შესრულება შეაჩერა, რადგან დაბლოკვის მოლოდინშია
NEW	ნაკადს ჯერ შესრულება არ დაუწყია
RUNNABLE	ნაკადი ახლა სრულდება ან დაიწყებს შესრულებას, როგორც კი პროცესორზე მიიღებს წვდომას
TERMINATED	ნაკადმა დაასრულა შესრულება
TIMED_WAITING	ნაკადმა შესრულება გარკვეული დროით შეაჩერა, მაგალითად, sleep() მეთოდის გამოძახების შემდეგ. ამ მდგომარეობაში ნაკადი wait() და join() მეთოდების გამოძახების შედეგადაც გადადის.
WAITING	ნაკადმა შესრულება შეაჩერა, რადგან ის გარკვეულ მოქმედებას ელოდება. მაგალითად, wait() და join() მეთოდების ვერსიების გამოძახებას, რომლებიც მოლოდინის რეჟიმს წყვეტენ.

იმისათვის, რომ Java-ს მრავალნაკადური შესაძლებლობები ეფექტურად გამოიყენოთ, თქვენ უნდა ისწავლოთ არა „მიმდევრობით“, არამედ „პარალელურად“ აზროვნება. მაგალითად, როდესაც პროგრამაში ორი ქვესისტემა გაქვთ, რომლებიც შეიძლება ერთდროულად შესრულედეს, გააფორმეთ ისინი ცალკეული ნაკადების სახით. ეს თქვენს პროგრამებს ეფექტურს

გახდის. მაგრამ, თუ ძალიან ბევრ ნაკადს შექმნით, ეს თქვენი პროგრამის წარმადობაზე უარყოფითად აისახება. ძალიან ბევრი ნაკადის შემთხვევაში პროცესორის დიდი დრო კონტექსტებზე გადართვას დაეთმობა და არა თქვენი პროგრამის შესრულებას.

დავალება:

1. *განმარტეთ მუტექსის არსი.*
2. *განმარტეთ ნაკადთაშორისი კომუნიკაციის მექანიზმი.*
3. *შეადგინეთ synchronized ოპერატორის გამოყენების სადემონსტრაციო პროგრამა.*
4. *შეადგინეთ suspend() და resume() მეთოდების გამოყენების სადემონსტრაციო პროგრამა, რომელიც run() მეთოდში ერთიდან 10-მდე მთელრიცხვა მნიშვნელობების კვადრატებს გამოიტანს კონსოლზე ორი სხვადასხვა ნაკადის არსებობის შემთხვევაში.*
5. *შეადგინეთ ორი X და Y ნაკადის ურთიერთბლოკირების ამსახველი პროგრამა.*

5.4. ფაილურ სისტემასთან მუშაობა

5.4.1. File კლასი

მართალია java.io პაკეტში განსაზღვრული კლასების უმრავლესობა ნაკადებთან მუშაობს, მაგრამ File კლასი ამას არ აკეთებს. მას საქმე უშუალოდ ფაილებთან და ფაილურ სისტემასთან აქვს. მაშასადამე, File კლასი არ მიუთითებს, თუ როგორ ხდება ფაილებიდან ინფორმაციის ამოღება და შენახვა. ის თავად ფაილების თვისებებს აღწერს. File კლასის ობიექტი ინფორმაციის მისაღებად და მისი მანიპულირებისთვის გამოიყენება. ეს ინფორმაცია დისკურ ფაილთან ასოცირდება, ისეთთან როგორიცაა, კატალოგთან წვდომა, კატალოგისკენ მიმავალი გზა, ქვეკატალოგების იერარქიაში ნავიგაცია და სხვა. File კლასი მრავალ პროგრამაში მონაცემთა პირველწყარო და დანიშნულების ადგილია.

ფაილები შეიძლება დავყოთ ორობით (Binnary) და ტექსტურ ფაილებად. ისინი შეღწვადობის (წვდომის) ორი თვისებით გამოირჩევიან: მიმდევრობითი და პირდაპირი. ფაილები შეიძლება გაიხსნას მხოლოდ წაკითხვისთვს, ჩაწერისთვის და დამატებისთვის.

Java-ში მიმდევრობითი ფაილის ცნების განზოგადება მოხდა და ამ განზოგადებულ ფაილს ნაკადები ქვია. ნაკადებსა და ფაილებზე მანიპულირების ყველა კლასი java.io პაკეტშია მოთავსებული.

Java-ში კატალოგი განიხილება როგორც File კლასის ობიექტი ერთადერთი დამატებითი თვისებით - ფაილთა სახელების სიით, რომელიც list() მეთოდის საშუალებით მიიღება.

File კლასის ობიექტების შესაქმნელად შემდეგი კონსტრუქტორები გამოიყენება:

```
File(String გზა კატალოგისკენ)
File(String  გზა კატალოგისკენ, String ფაილის სახელი)
File(File კატალოგის ობიექტი, String ფაილის სახელი)
File(URI ობიექტი URI)
```

აქ პარამეტრი „გზა კატალოგისკენ“ ფაილისკენ მიმავალ გზას გულისხმობს. „ფაილის სახელი“ ფაილის ან კატალოგის სახელია; „კატალოგის ობიექტი“ – File კლასის ობიექტი, ხოლო “URI ობიექტი“ წარმოადგენს ობიექტს URI, რომელიც ფაილს აღწერს.

შევქმნათ სამი f1, f2 და f3 ფაილები. File კლასის პირველი ობიექტი ერთადერთი არგუმენტით - ფაილისკენ მიმავალი გზით იქმნება. მეორე - ორ არგუმენტს მოიცავს: ფაილისკენ მიმავალ გზას და ფაილის სახელს. მესამე წარმოგივგდენს f1 ფაილისთვის მინიჭებულ გზას და ფაილის სახელს. f3 ფაილის ობიექტი კი იმავე ფაილს მიმართავს, რომელსაც f2.

```
File f1 = new File(“/”)
File f2 = new File(“/”, “autoexec.bat”)
File f3 = new File(f1, “autoexec.bat”)
```

File კლასი მრავალ მეთოდს განსაზღვრავს, რომლებიც File კლასის ობიექტის სტანდარტულ თვისებებს წარმოგვიდგენს. მაგალითად, getName() მეთოდი ფაილის სახელს აბრუნებს; getParent() მეთოდი - მშობელი კატალოგის სახელს, ხოლო exists() მეთოდი ფაილის არსებობის შემთხვევაში - true (ჭეშმარიტ) მნიშვნელობას, წინააღმდეგ შემთხვევაში (ფაილის არარსებობის დროს) - false (მცდარ) მნიშვნელობას. File კლასი არ არის სიმეტრიული. მასში რამდენიმე მეთოდი არსებობს, რომლებიც საშუალებას გვაძლევს უბრალო ფაილური ობიექტის თვისება შევამოწმოთ. 33-ე ცხრილში წარმოდგენილია File კლასის ძირითადი მეთოდები.

ცხრილი 33. File კლასის ძირითდი მეთოდები

მეთოდი	აღწერა
boolean exists( )	ამოწმებს ფაილი არსებობს დისკზე თუ არა
boolean createNewFile( )	ქმნის ახალ ფაილს
boolean delete( )	შლის (ანადგურებს) არსებულ ფაილს
void deleteOnExit( )	შლის ფაილს, როდესაც პროგრამა მუშაობას ამთავრებს
String getAbsolutePath()	იძლევა ფაილის აბსოლუტურ მისამართს
String getName( )	იძლევა ფაილის ან კატალოგის სახელს
boolean isDirectory( )	ამოწმებს არის თუ არა კატალოგი
boolean isFile( )	ამოწმებს არის თუ არა ფაილი
long length( )	იძლევა ფაილის სიგრძეს ბაიტებში
String list()	იძლევა იმ ფაილების და კატალოგების დასახელებების სიას, რომლებიც აქტიურ კატალოგშია მოთავსებული
boolean mkdir( )	ქმნის კატალოგს
boolean renameTo(File)	ფაილს უცვლის სახელს

მაგალითი 14. შევადგინოთ პროგრამა, რომელიც File კლასის მეთოდების დემონსტრირებას ახდენს. აქ იგულისხმება, რომ ფესვურ კატალოგში java სახელის მქონე კატალოგი არსებობს და რომ ის COPYRIGHT ფაილს შეიცავს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package fileswork;
import java.io.File;
public class FileDemo {
	static void p(String s){
		System.out.println(s);
	}
	public static void main(String [] args){
		File f1=new File("/java/COPYRIGHT.txt");
		p("ფაილის სახელი: " + f1.getName());
		p("გზა: " + f1.getPath());
		p("აბსოლუტური გზა: " + f1.getAbsolutePath());
		p("მშობელი კატალოგი: " + f1.getParent());
		p(f1.exists() ? "ფაილი არსებობს" : "ფაილი არ არსებობს");
		p(f1.canWrite() ? "ფაილი წვდომადია ჩაწერისთვის" : "ფაილი წვდომადი არ
არის  ჩაწერისთვის");
		p(f1.canRead() ? "ფაილი წვდომადია კითხვისთვის" : "ფაილი წვდომადი არ
არის  კითხვისთვის");
		p(f1.isDirectory() ? "წარმოადგენს კატალოგს" : "არ წარმოადგენს
კატალოგს");
		p(f1.isFile() ? "ჩვეულებრივი ფაილია" : "შესაძლოა სახელობითი
კატალოგია");
		p(f1.isAbsolute() ? "აბსოლუტურია" : "არ არის აბსოლუტური");
		p("მოდიფიკაციის დრო: " + f1.lastModified());
		p("ფაილის ზომა: " + f1.length() + " ბაიტი");
	}
}
```

ნახ. 268

```
ფაილის სახელი: COPYRIGHT.txt
გზა: \java\COPYRIGHT.txt
აბსოლუტური გზა:
C:\java\COPYRIGHT.txt
მშობელი კატალოგი: \java
ფაილი არსებობს
ფაილი წვდომადია ჩაწერისთვის
ფაილი წვდომადია კითხვისთვის
არ წარმოადგენს კატალოგს
ჩვეულებრივი ფაილია
არ არის აბსოლუტური
მოდიფიკაციის დრო:
1445700309276
ფაილის ზომა: 94 ბაიტი
```

ნახ. 269

წარმოდგენილ პროგრამაში isFile() მეთოდი აბრუნებს true მნიშვნელობას, თუ ის ფაილთან გამოიძახება და აბრუნებს false მნიშვნელობას, თუ გამოძახება კატალოგთან ხდება. isFile() მეთოდი false მნიშვნელობას სპეციალური ფაილების შემთხვევაშიც აბრუნებს, როგორებიცაა

მოწყობილობის დრაივერები, სახელობითი არხები. ამიტომ ეს მეთოდი იმის გარანტადაც გამოიყენება, რომ მოცემული ფაილი იქცევა როგორც ჩვეულებრივი ფაილი.

isAbsolute() მეთოდი true მნიშვნელობას აბრუნებს, თუ ფაილს აბსოლუტური გზა აქვს, ფარდობითის შემთხვევაში აბრუნებს false-ს.

File კლასი შეიცავს renameTo() მეთოდს, რომლის ჩაწერის ზოგადი ფორმა შემდეგია:

boolian renameTo(File ახალი სახელი)

აქ პარამეტრი „ახალი სახელი“ ფაილის სახელია, რომელიც File კლასის გამომძახებელი ობიექტის ახალ სახელად იქცევა. მეთოდი true მნიშვნელობას წარმატების შემთხვევაში აბრუნებს. თუ ფაილის სახელის გადარქმევა შეუძლებელია (ვთქვათ, ასეთი სახელის მქონე ფაილის არსებობის გამო), აბრუნებს false მნიშვნელობას.

File კლასი შეიცავს delete() მეთოდს, რომელიც დისკურ ფაილს შლის (ანადგურებს). მისი ჩაწერის ზოგადი ფორმაა:

boolian delete()

ამ მეთოდით კატალოგის წაშლაც შეიძლება, ოღონდ იმ შემთხვევაში, თუ ის ცარიელია. მეთოდი true მნიშვნელობას აბრუნებს, თუ ფაილის წაშლა წარმატებით დასრულდა. წინააღმდეგ შემთხვევაში - აბრუნებს false მნიშვნელობას.

34-ე ცხრილში წარმოდგენილია File კლასის სასარგებლო (საჭირო) მეთოდები.

ცხრილი 34. File კლასის საჭირო მეთოდები

მეთოდი	აღწერა
void deleteOnExit()	Java ვირტუალური მანქანის მუშაობის დასრულებისთანავე შლის (ანადგურებს) ფაილს
long getFreeSpace()	აბრუნებს საცავის წვდომადი თავისუფალი ბაიტების რაოდენობას
long getTotalSpace()	აბრუნებს საცავის ტევადობას
long getUsableSpace()	აბრუნებს გამოყენებისთვის გამოსადეგი თავისუფალი ბაიტების რაოდენობას
boolean isHidden()	აბრუნებს true მნიშვნელობას თუ ფაილი დაფარულია, წინააღმდეგ შემთხვევაში აბრუნებს false-ს
boolean setLastModified(long მილიწამები)	ფაილისთვის დროითი ჭდის მილიწამებში დაყენება, რომელიც მილიწამების იმ რაოდენობას გვიჩვენებს, რაც 1970 წლის პირველი იანვრიდან არის გასული
boolean setReadOnly()	ფაილი მხოლოდ კითხვისთვის ხდება წვდომადი

5.4.2. კატალოგები

კატალოგი File კლასის ობიექტია, რომელიც სხვა ფაილებისა და კატალოგების სიას შეიცავს. File კლასის ობიექტის შექმნის შემდეგ, რომელიც კატალოგს წარმოადგენს, isDirectory() მეთოდი true მნიშვნელობას აბრუნებს. ამ შემთხვევაში, აღნიშნული ობიექტისთვის list() მეთოდი შეგვიძლია გამოვიძახოთ იმ მიზნით, რომ კატალოგში არსებული ფაილებისა და სხვა კატალოგების ჩამონათვალი ვნახოთ. აღნიშნულ მეთოდს ჩაწერის ორი ფორმა აქვს. პირველი:

String [] list()

ფაილების სია აქ String კლასის ობიექტების მასივის სახით ბრუნდება.

მაგალითი 15. შევადგინოთ პროგრამა, რომელიც list() მეთოდის გამოყენებით კატალოგის შიგთავსის დათვალიერების საშუალებას მოგვცემს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package fileswork;
import java.io.File;
public class DirList {
	public static void main(String [] args){
		String dirname= "/java";
		File f1=new File(dirname);
		if(f1.isDirectory()){
			System.out.println("კატალოგი " + dirname);
			String s[]=f1.list();
			for(int i=0; i<s.length; i++){
				File f=new File(dirname + "/" + s[i]);
				if(f.isDirectory()){
					System.out.println(s[i] + " კატალოგია");
				}
				else{
					System.out.println(s[i] + " ფაილია");}}}
			 else{
			System.out.println(dirname + "არ არის კატალოგი");}}}
```

ნახ. 270

შედეგი:

```
კატალოგი /java
COPYRIGHT.txt ფაილია
Ece კატალოგია
MyJava კატალოგია
```

ნახ. 271

ცხადია, პროგრამის რეალიზაციის ჩვენ მიერ მიღებული შედეგები განსხვავებული იქნება თქვენ მიერ მიღებული შედეგებისგან, რადგან ძალზე დიდია ალბათობა იმისა, რომ კატალოგის შიგთავსი განსხვავებული გვქონდეს.

5.4.3. FilenameFilter ინტერფეისის გამოყენება

ზოგჯერ აუცილებელია list() მეთოდის მიერ დასაბრუნებელი ფაილების რაოდენობა შევზღუდოთ და მხოლოდ ის ფაილები დავაბრუნოთ, რომლებიც სახელების გარკვეულ შაბლონს ან ფილტრს შეესაბამება. ამ მიზნით list() მეთოდის მეორე ფორმა გამოიყენება, რომლის ჩაწერის ზოგადი სახე შემდეგია:

String [] list (FilenameFilter FF ობიექტი)

აქ "FF ობიექტი" კლასის ობიექტია, რომელიც FilenameFilter ინტერფეისის რეალიზებას ახდენს. აღნიშნული ინტერფეისი მხოლოდ ერთ accept() მეთოდს განსაზღვრავს, რომლის ჩაწერის ზოგადი ფორმაა:

boolean accept(File კატალოგი, String ფაილის სახელი)

accept() მეთოდი true მნიშვნელობას აბრუნებს კატალოგის იმ ფაილებისთვის, რომლებიც სიაში უნდა ჩაემატოს (ანუ რომლებიც არგუმენტს - „ფაილის სახელი" შეესაბამებიან) და false მნიშვნელობას - იმ ფაილებისთვის, რომლებიც სიიდან უნდა ამოვიღოთ.

მოვახდინოთ წინა პროგრამის (მაგალითი 15) მოდიფიცირება და ფაილების გაფილტვრა list() მეთოდის გამოყენებით. კერძოდ, პროგრამა აქტიური კატალოგიდან მხოლოდ txt (ტექსტურ) გაფართოების ფაილებს გამოიტანს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package fileswork;
import java.io.*;
public class OnlyExt implements FilenameFilter{
	String ext;
	public  OnlyExt(String ext){
		this.ext="." + ext;
	}
	public boolean accept(File dir, String name){
		return name.endsWith(ext);
	}
}
```

```
package fileswork;
import java.io.*;
class DirListOnly {
	public static void main(String [] args){
		String dirname= "/java";
		File f1=new File(dirname);
		FilenameFilter only=new OnlyExt("txt");
		String s[]=f1.list(only);
		for(int i=0; i<s.length; i++){
			System.out.println(s[i]); }}}
```

ნახ. 272

შედეგი:

```
COPYRIGHT.txt
Lulu.txt
MYfile.txt
```

ნახ. 273

File კლასს კიდევ ორი საინტერესო მეთოდი აქვს. ესენია: mkdir() და mkdirs(). მათგან პირველი mkdir() კატალოგს ქმნის და წარმატების შემთხვევაში true მნიშვნელობას აბრუნებს, ხოლო წინააღმდეგ შემთხვევაში - flase მნიშვნელობას. მაშინ, როდესაც კატალოგის გზა ჯერ კიდევ არ არის შექმნილი, თავად კატალოგის შესაქმენლად mkdirs() მეთოდი გამოიყენება. ის ქმნის არა მხოლოდ კატალოგს, არამედ მის „მშობლებსაც“.

5.4.4. ფაილების ჩაწერა, წაკითხვა და დახურვა

Java ენა ფაილების წაკითხვისა და ჩაწერის ძალიან დიდ შესაძლებლობებს გვაძლევს კლასებისა და მეთოდების გამოყენების თვალსაზრისით.

FileInputStream და FileOutputStream ის კლასებია, რომლებიც ფაილებთან დაკავშირებულ ბაიტის ტიპის ნაკადებს ქმნიან. ფაილის გახსნის მიზნით, თქვენ ამ კლასებიდან ერთ-ერთის ობიექტი უნდა შექმნათ მხოლოდ, სადაც არგუმენტის სახით კლასის კონსტრუქტორს ფაილის სახელი უნდა გადასცეთ. ამ კონსტრუქტორების ჩაწერის ზოგადი ფორმები შემდეგია:

FileInputStream(String ფაილის სახელი) throws FileNotFoundException

FileOutputStream(String ფაილის სახელი) throws FileNotFoundException

აქ არგუმენტი „ფაილის სახელი“ იმ ფაილის სახელს გულისხმობს, რომლის გახსნაც გსურთ. თუ შემავალი ნაკადის შექმნისას ფაილი არ არსებობს, ხდება FileNotFoundException გამონაკლისის გადაცემა. გამოსასვლელი ნაკადების შემთხვევაშიც, თუ ფაილი ვერ იხსნება ან ვერ

იქმნება, იგივე FileNotFoundException გამონაკლისი გადაიცემა. გამონაკლისების ეს კლასი IOException კლასისგან არის წარმოებული. როდესაც გამოსასვლელი ფაილი გახსნილია, უკვე არსებული იმავე სახელის მქონე ფაილი ნადგურდება.

ფაილთან მუშაობის დასრულებისას, მისი დახურვა აუცილებელია. ამ მიზნით FileInput-Stream და FileOutputStream კლასებში განსაზღვრული close() მეთოდი გამოიყენება, რომლის ჩაწერის ზოგადი ფორმაა:

```
void close() throws IOException
```

ფაილის დახურვა მისთვის გამოყოფილი სისტემური რესურსების გამოთავისუფლებას იწვევს. შემდგომ ეს რესურსები სხვა ფაილებისთვის შეგვიძლია გამოვიყენოთ.

როდესაც ფაილი აღარ არის საჭირო და მისი დახურვა გვსურს, ორი მიდგომიდან ერთ-ერთი შეგვიძლია გამოვიყენოთ. კერძოდ, პირველი - ტრადიციული მიდგომაა, როდესაც ფაილის დასახურად close() მეთოდს ცხადი სახით ვიძახებთ. მეორე მიდგომა, შედარებით ახალია და ის ავტომატურად ხურავს ფაილს, როდესაც ის საჭირო აღარ არის. ეს უკანასკნელი (მეორე მიდგომა) რესურსებიანი try ოპერატორის გამოყენებას გულისხმობს. ამ შემთხვევაში close() მეთოდი ცხადი სახით არ გამოიძახება.

ფაილის წასაკითხად FileInputStream კლასის read() მეთოდს მივმართავთ, რომლის ჩაწერის ზოგადი ფრომა შემდეგია:

```
int read() throws IOExcepton
```

აღნიშნული მეთოდის ყოველ გამოძახებაზე ფაილიდან ერთადერთი ბაიტის წაკითხვა და მთელრიცხვა მნიშვნელობის სახით დაბრუნება ხდება. ფაილის დასრულებისას მეთოდი -1-ის ტოლ მნიშვნელობას აბრუნებს და მეთოდს IOExcepton გამონაკლისის გადაცემაც შეუძლია.

მაგალითი 16. შევადგინოთ პროგრამა, რომელიც იმ ფაილის შეტანას და შიგთავსის გამო-ტანას ასრულებს, რომელშიც ტექსტი ASCII კოდებშია წარმოდგენილი.

ნახ.274-ზე ნაჩვენებ პროგრამაში TEST.TXT ფაილის შიგთავსი გამოიტანება, თუმცა თქვენ სხვა ფაილის შიგთავსის ნახვაც შეგიძლიათ, თუ მიუთითებთ სხვა ფაილის სახელს და გაფართოებას.

პროგრამის კომპიუტერული რეალიზაცია:

```
package fileswork;
/*
 java ShowFile TEST.TXT
 */
import java.io.*;
public class ShowFile {
	public static void main(String args[]){
		int i;
		FileInputStream fin;
		//პირველ რიგში დავრწმუნდეთ, რომ ფაილის სახელი მითითებულია
		if(args.length != 1){
			System.out.println("გამოიყენება:  java ShowFile TEST.TXT");
		return; }
		//ფაილის გახსნის მცდელობა
		try{
			fin=new FileInputStream(args[0]);
		}catch(FileNotFoundException e){
			System.out.println("არ შემიძლია ფაილის გახსნა");
			return;
		}
		//ახლა ფაილი გახსნილია და მზადაა წაკითხვისთვის
		//შემდეგი კოდი კითხულობს სიმბოლოებს,  სანამ EOF არ შეხვდება
		try{
			do{
				i=fin.read();
				if(i != -1) System.out.print((char) i);
			}while(i != -1);
		}catch(IOException e){
			System.out.println("ფაილის წაკითხვის შეცდომა");
		}
		//ფაილის დახურვა
		try{
			fin.close();
		}catch(IOException e){
			System.out.println("ფაილის დახურვის შეცდომა"); }}}
```

ნახ. 274

ზოგჯერ სასურველია, რომ პროგრამის ყველა ის ნაწილი, სადაც ადგილი აქვს ფაილის გახსნას და მის შიგთავსზე წვდომას, ერთ try ბლოკში მოვათავსოთ. შემდგე კი finally ბლოკი გამოვიყენოთ ფაილის დასახურად.

ზემოთ თქმულის გათვალისწინებით მე-16 მაგალითის შესაბამისი პროგრამა მოდიფიცირებული სახით შეგვიძლია წარმოვადგინოთ.

პროგრამის კომპიუტერული რეალიზაცია:

```
package fileswork;
/*
 java ShowFile TEST.TXT
 */
import java.io.*;
public class ShowFile {
	public static void main(String args[]){
		int i;
		FileInputStream fin=null;
		//პირველ რიგში დავრწმუნდეთ, რომ ფაილის სახელი მითითებულია
		if(args.length != 1){
			System.out.println("გამოიყენება:  java ShowFile TEST.TXT");
		return; }
		//შემდეგი კოდი ხსნის ფაილს, კითხულობს სიმბოლოებს და finally ბლოკში
ხურავს ფაილს
		try{
			fin=new FileInputStream(args[0]);
			do{
				i=fin.read();
				if(i != -1) System.out.print(i);
			}while(i != -1);
		}catch(FileNotFoundException e){
			System.out.println("ფაილი ვერ მოიძებნა");
			}catch(IOException e){
				System.out.println("დაშვებულია შეტანა-გამოტანის (I/O)
შეცდომა");
			}finally{
				//ფაილი იხურება ნებისმიერ შემთხვევაში
				try{
					if(fin != null) fin.close();
		}catch(IOException e){
			System.out.println("ფაილის დახურვის შეცდომა"); }}}}
```

ნახ. 275

ფაილში მონაცემების შესატანად (ჩასაწერად) fileOutputStrem კლასის write() მეთოდი გამოიყენება, რომლის ჩაწერის უმარტივესი ფორმა შემდეგია:

void write(ბაიტის მნიშვნელობა) throws IOException

ეს მეთოდი ფაილში წერს ბაიტს, რომელიც „ბაიტის მნიშვნელობა“ პარამეტრით გადაიცემა. მართალია, ეს პარამეტრი მთელრიცხვა მნიშვნელობით აღიწერება, მაგრამ ფაილში მხოლოდ მისი უმცირესი რვა ბიტი იწერება. ჩაწერის დროს, შეცდომის შემთხვევაში, IOException ტიპის გამონაკლისი გადაიცემა.

მაგალითი 17. შევადგინოთ პროგრამა, რომელიც write() მეთოდის გამოყენებით ერთი ფაილის (ჩვენ შემთხვევაში FIRST.TXT) მეორეში ფაილში (ჩვენ შემთხვევაში SECOND.TXT) კოპირებას მოახდენს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package fileswork;
import java.io.*;
public class CopyFile {
	public static void main(String args[]){
		int i;
		FileInputStream fin=null;
		FileOutputStream fout=null;
		//დავრწმუნდეთ, რომ ორივე ფაილის სახელი მითითებულია
		if(args.length != 2){
			System.out.println("CopyFile FIRST.TXT SECOND.TXT");
		return;
	}
	//ფაილის კოპირება
	try{
		//ფაილების გახსნის მცდელობა
		fin=new FileInputStream(args[0]);
		fout=new FileOutputStream(args[1]);
		do{
			i=fin.read();
			if(i != -1) fout.write(i);
		}while(i != -1);
		}catch(IOException e){
			System.out.println("დაშვებულია შეტანა-გამოტანის (I/O) შეცდომა");
		}finally{
			//ფაილი იხურება ნებისმიერ შემთხვევაში
			try{
				if(fin != null) fin.close();
			}catch(IOException e2){
				System.out.println("შეტანის ფაილის დახურვის შეცდომა"); }
			try{
				if(fout != null) fout.close();
			}catch(IOException e2){
				System.out.println("გამოტანის  ფაილის დახურვის შეცდომა");
			}}}}
```

ნახ. 276

5.4.5. ფაილის ავტომატური დახურვა

აქამდე ამ მიზნით close() მეთოდს ცხადი სახით ვიძახებდით. თუმცა დღეს Java-ში რესურსების გამოთავისუფლების ავტომატური საშუალება არსებობს და იგი try ოპერატორის გაუმჯობესებული ვერსიის გამოყენებას ემყარება, რომლის ჩაწერის ზოგადი ფორმა შემდეგია:

```
try(რესურსის სპეციფიკაცია)
{
//რესურსის გამოყენება
}
```

აქ „რესურსის სპეციფიკაცია“ ოპერატორია, რომელიც აღწერს და ინიციალებას უკეთებს ისეთ რესურსს, როგორსაც ფაილური ნაკადი წარმოადგენს. try ბლოკის დასრულებისას, რესურსი ავტომატურად თავისუფლდება. ფაილის შემთხვევაში, ეს ნიშნავს, რომ ფაილი ავტომატურად

იხურება. ასე რომ, close() მეთოდის ცხადი სახით გამოძახების არანაირი საჭიროება აქ არ არსებობს. ცხადია, try ოპერატორის ეს ფორმა, რომელსაც try-რესურსებით ეწოდება, შეიძლება finally და catch დირექტივებსაც მოიცავდეს.

ამგვარად, try ოპერატორის ეს გაუმჯობესებული ვარიანტი შეიძლება გამოვიყენოთ ნაკადებთან, მათ შორის, ფაილურ ნაკადებთან სამუშაოდ.

მაგალითი 18. შევადგინოთ პროგრამა, რომელიც ფაილის ავტომატურ დახურვას ასრულებს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package fileswork;
import java.io.*;
public class ShowFile2 {
	public static void main(String args[]){
		int i;
		//პირველ რიგში დავრწმუნდეთ, რომ ფაილის სახელი მითითებულია
		if(args.length != 1){
			System.out.println("გამოიყენება: ShowFile TEST.TXT");
		return; }
```

```
//შემდეგი კოდი რესურსებიან try ოპერატორს იყენებს
		//იმისათვის, რომ ფაილი გახსნას და try ბლოკის დასრულებისას
ავტომატურად დახუროს
		try(FileInputStream fin=new FileInputStream(args[0])){
			do {
				i=fin.read();
				if(i != -1) System.out.print(i);
			} while(i != -1);
		}catch(FileNotFoundException e){
			System.out.println("ფაილი ვერ მოიძებნა");
	}catch(IOException e){
		System.out.println("დაშვებულია შეტანა-გამოტანის (I/O) შეცდომა");
}}}
```

დავალება:

1. *შეადგინეთ პროგრამა, რომელიც File კლასის მეთოდების გამოყენებით კონსოლზე გამოიტანს თქვენთვის საინტერესო ფაილის სახელს, გზას, ინფორმაციას იმის შესახებ, წვდომადია თუ არა ფაილი კითხვისთვის, ჩაწერისთვის, არის თუ არა ის ჩვეულებრივი ფაილი და წარმოგვიდგენს ფაილის ზომას.*
2. *შეადგინეთ პროგრამა, რომელიც თქვენთვის სასურველი კატალოგის შიგთავსს list() მეთოდის გამოყენებით გამოიტანს კონსოლზე.*
3. *შეადგინეთ პროგრამა, რომელიც Pictures კატალოგიდან მხოლოს .jpg გაფართოების ფაილებს გამოიტანს კონსოლზე.*
4. *შეადგინეთ პროგრამა, რომელიც განახორციელებს თქვენთვის სასურველი ფაილის შეტანას და მისი შიგთავსის გამოტანას.*
5. *შეადგინეთ პროგრამა, რომელიც თქვენთვის სასურველი ორი ფაილიდან ერთ-ერთის კოპირებას მოახდენს მეორე ფაილში.*
6. *შეადგინეთ პროგრამა, რომელიც თქვენთვის სასურველი ფაილის ავტომატურ დახურვას გაუმჯობესებული რესურსებიანი try ოპერატორის გამოყენებით განახორციელებს.*

5.5. შემავალ და გამავალ ნაკადებთან მუშაობა

Java პროგრამები შეტანა-გამოტანის ნაკადებს ქმნიან. ნაკადი (Stream) აბსტრაქციაა, რომელიც „ბადებს“ ან იღებს ინფორმაციას. ფიზიკურ მოწყობილობასთან ნაკადი java-ს შეტანა-გამოტანის სისტემით არის დაკავშირებული. ნაკადების რეალიზებას Java ენა იმ კლასების გამოყენებით ახდენს, რომლებიც java.io პაკეტშია განსაზღვრული.

Java ნაკადის ორ ტიპს განსაზღვრავს: ბაიტის ტიპის და სიმბოლურს. ბაიტური ნაკადები ბაიტების შეტანა-გამოტანის მართვისთვის მოხერხებულ საშუალებებს გვთავაზობს. ეს ნაკადები ბინარული მონაცემების ჩაწერისა და წაკითხვისთვის გამოიყენება. სიმბოლური ნაკადები კი სიმბოლოების შეტანა-გამოტანის მართვის საუკეთესო საშუალებებს გვთავაზობენ. ისინი Unicode კოდირებას იყენებენ და ზოგჯერ სიმბოლური ნაკადები ბაიტის ტიპის ნაკადებზე უფრო ეფექტურია.

5.5.1. ბაიტის ტიპის ნაკადების კლასები

ბაიტის ტიპის ნაკადები კლასების ორ იერარქიაშია განსაზღვრული. იერარქიის თავში InputStream და OutputStream აბსტრაქტული კლასებია მოთავსებული. ყოველ ამ აბსტრაქტულ კლასს რამდენიმე რეალური ქვეკლასი აქვს. java.io პაკეტში არსებული ბაიტის ტიპის კლასები 35-ე ცხრილშია წარმოდგენილი. გახსოვდეთ, რომ პროგრამებში ნაკადების კლასების გამოსაყენებლად აუცილებელია java.io პაკეტის იმპორტირება.

ცხრილი 35 ბაიტის ტიპის ნაკადების კლასები java.io პაკეტიდან

ნაკადის კლასი	დანიშნულება
BufferedInputStream	ბუფერიზებული შემავალი ნაკადი
BufferedOutputStream	ბუფერიზებული გამომავალი ნაკადი
ByteArrayInputStream	შემავალი ნაკადი, რომელიც მასივიდან კითხულობს ბაიტს
ByteArrayOutputStream	გამომავალი ნაკადი, რომელიც მასივში წერს ბაიტს
DataInputStream	შემავალი ნაკადი, რომელიც მოიცავს Java-ს სტანდარტული ტიპის მონაცემების წაკითხვის მეთოდებს
DataOutputStream	გამომავალი ნაკადი, რომელიც მოიცავს Java-ს სტანდარტული ტიპის მონაცემების ჩაწერის მეთოდებს
FileInputStream	ფაილიდან წამკითხავი შემავალი ნაკადი
FileOutputStream	ფაილში ჩამწერი გამომავალი ნაკადი
FilterInputStream	InputStream კლასის რეალიზაცია
FilterOutputStream	OutputStream კლასის რეალიზაცია

InputStream	შეტანის ნაკადის აღმწერი აბსტრაქტული კლასი
ObjectInputStream	შემავალი ნაკადი ობიექტებისთვის
ObjectOutputStream	გამომავალი ნაკადი ობიექტებისთვის
OutputStream	გამოტანის ნაკადის აღმწერი აბსტრაქტული კლასი
PipedInputStream	შემავალი არხი (მაგალითად, პროგრამათაშორისი)
PipedOutputStream	გამომავალი არხი
PrintStream	print() და println() მეთოდების მომცველი გამომავალი ნაკადი
PushbackInputStream	შემავალ ნაკადში ერთბაიტიანი დაბრუნების მხარდამჭერი შემავალი ნაკადი
SequenceInputStream	შემავალ ნაკადი, რომელიც წარმოადგენს ორი ან მეტი შემავალი ნაკადის კომბინაციას (ეს ნაკადები ერთდოულად იკითხება. ერთი მეორის შემდეგ)

InputStream და OutputStream აბსტრაქტული კლასები რამდენიმე მნიშვნელოვან მეთოდს განსაზღვრავენ, რომლებიც ნაკადების სხვა კლასების რეალიზებას ახდენენ. ესენია read() და write() მეთოდები, რომლებიც შესაბამისად, კითხულობენ და წერენ მონაცემთა ბაიტებს. ორივე მეთოდი აბსტრაქტულია და მემკვიდრე კლასებში ადგილი აქვს მათ ხელახლა განსაზღვრას.

5.5.2. სიმბოლური ნაკადების კლასები

სიმბოლური ნაკადებიც კლასებიც ორ იერარქიაშია განსაზღვრული. მათ სათავეში ორი აბსრაქტული კლასი: Reader და Writer უდგას. სიმბოლური ნაკადების კლასები 36-ე ცხრილშია წარმოდგენილი.

ცხრილი 36 სიმბოლური ტიპის ნაკადების კლასები java.io პაკეტიდან

ნაკადის კლასი	დანიშნულება
BufferedReader	ბუფერიზებული შემავალი სიმბოლური ნაკადი
BufferedWriter	ბუფერიზებული გამომავალი სიმბოლური ნაკადი
CharArrayReader	შემავალი ნაკადი, რომელიც სიმბოლური მასივიდან სიმბოლოს კითხულობს
CharArrayWriter	გამომავალი ნაკადი, რომელიც სიმბოლოს წერს სიმბოლურ მასივში
FileReader	ფაილის წამკითხავი შემავალი ნაკადი
FileWriter	ფაილში ჩამწერი გამომავალი ნაკადი
FilterReader	გამფილტრავი „მკითხველი“

FilterWriter	გამფილტრავი „მწერალი“
InputStreamReader	შემავალი ნაკადი, რომელიც ბაიტების სიმბოლოებში ტრანსლირებას ახდენს
LineNumber Reader	სტრიქონების მთვლელი შემავალი ნაკადი
OutputStreamWriter	გამომავალი ნაკადი, რომელიც ბაიტების სიმბოლოებში ტრანს-ლირებას ახდენს
PipedReader	შემავალი არხი
PipedWriter	გამომავალი არხი
PrintWriter	print() და println() მეთოდების შემცველი გამომავალი ნაკადი
PushbackReader	შემავალი ნაკადი, რომელიც ნებას გვრთავს სიმბოლო დავაბრუნოთ უკან, ნაკადში
Reader	სიმბოლური შეტანის აღმწერი აბსტრაქტული კლასი
StringReader	სტრიქონიდან წამკითხავი შემავალი ნაკადი
StringWriter	სტრიქონში ჩამწერი გამომავალი ნაკადი
Writer	სიმბოლური გამოტანის აღმწერი აბსტრაქტული კლასი

Read და Writer ბასტრაქტული კლასები რამდენიმე მნიშვნელოვან მეთოდს განსაზღვრავენ, რომლებიც ნაკადების სხვა კლასების რეალიზებას ახდენენ. ესენია read() და write() მეთოდები, რომლებიც შესაბამისად, კითხულობენ და წერენ სიმბოლურ მონაცემებს. ეს მეთოდები მემკვიდრე კლასებში ხელახლა განისაზღვრება.

5.5.3. წინასწარ განსაზღვრული ნაკადები

ცნობილია, რომ Java ენის ყველა პროგრამა ავტომატურად ახდენს java.lang პაკეტის იმპორტირებას. ამ პაკეტში System კლასია განსაზღვრული, რომელიც გარდა მეთოდებისა შეიცავს წინასწარ განსაზღვრულ სამ ნაკადურ ცვლადს. კერძოდ: in, out და err. ეს ცვლადები System კლასში გაცხადებულია როგორც public, static და final. ეს კი ნიშნავს, რომ ისინი System კლასის ობიექტზე მიმართვის გარეშე შეიძლება გამოყენებულ იქნეს თქვენი პროგრამის ნების-მიერი ნაწილის მიერ.

System.out ცვლადი მიმართავს სტანდარტულ გამომავალ ნაკადს. მიუთითებლობის შემთხვევაში, ეს კონსოლია. System.in ცვლადი სტანდარტულ შემავალ ნაკადს მიმართავს, რომელიც მიუთითებლობის შემთხვევაში, ასევე კონსოლს წარმოადგენს.

System.err ცვლადი შეცდომების სტანდარტულ ნაკადს მიმართავს, რომელიც მიუთითებ-ლობის შემთხვევაში, აქაც კონსოლს წარმოადგენს.

System.in ცვლადი InputStream კლასის ობიექტია, System.out და System.err ცვლადები - PrintStream კლასის ობიექტები. ისინი ბაიტის ტიპის ნაკადებია, თუმცა ჩვეულებრივ, კონსოლიდან სიმბოლოების წაკითხვის ან კონსოლზე სიმბოლოების გამოტანის მიზნით გამოიყენებიან.

კონსოლური შეტანა java ენაში System.in ნაკადის კითხვით ხორციელდება. კონსოლზე მიერთებული სიმბოლური ნაკადის მისაღებად System.in ნაკადი BufferedReader კლასის ობიექტის შიგნით განათავსეთ. აღნიშნული კლასის ობიექტი ბუფერიზებულ შემავალ ნაკადს უზრუნველყოფს. მისი ხშირად გამოყენებადი კონსტრუქტორის ზოგად ფორმაა:

BufferedReader(Reader შეტანის მთვლელი)

აქ „შეტანის მთვლელი“ ნაკადია, რომელიც BufferedReader კლასის ეგზემპლიარს უკავშირდება. Reader აბსტრაქტული კლასია. მის ერთ-ერთ კონკრეტულ მემკვიდრეს InputStreamReader კლასი წარმოადგენს, რომელიც ბაიტებს სიმბოლოებში გარდასახავს. System.in ნაკადთან დაკავშირებული InputStreamReader კლასის ობიექტის მისაღებად შემდეგი კონსტრუქტორი გამოიყენება:

InputStreamReader(InputStream შეტანის ნაკადი)

რადგან System.in ცვლადი InputStream კლასის ობიექტს მიმართავს, ამიტომ ის „შეტანის ნაკადის“ პარამეტრად უნდა იქნეს გამოყენებული. ამ ყველაფრის გათვალისწინებით, კლავიატურასთან დაკავშირებული BufferedReader კლასის ობიექტის შესაქმენლად შემდეგი კოდი გამოიყენება:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

ამ ოპერატორის შერულების შემდეგ br ობიექტი სიმბოლურ ნაკადს წარმოადგენს, რომელიც კონსოლს System.in ნაკადით უკავშირდება.

BufferedReader კლასის ობიექტიდან სიმბოლოს წასაკითხად read() მეთოდი გამოიყენება, რომლის ჩაწერის ზოგადი ფორმაა:

```
int read() throws IOException
```

read() მეთოდის ყოველ გამოძახებაზე, ის სიმბოლოს შემავალი ნაკადიდან კითხულობს და მთელრიცხვა მნიშვნელობის სახით აბრუნებს. ნაკადის დასასრულის მიღწევის შემთხვევაში კი -1-ის ტოლი მნიშვნელობა ბრუნდება. მეთოდს IOException გამონაკლისის გადაცემაც შეუძლია.

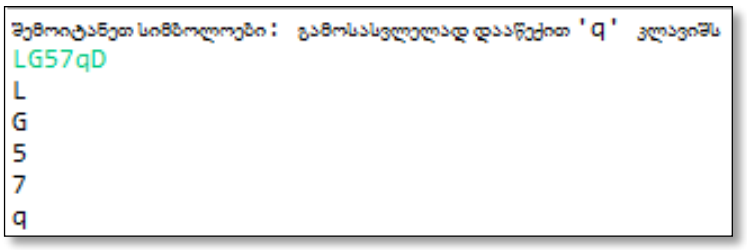
მაგალითი 19. შევადგინოთ პროგრამა, რომელიც read() მეთოდის გამოყენებით კონსოლიდან სიმბოლოებს მანამდე კითხულობს, სანამ მომხმარებელი „q“ ასოს არ შეიტანს. ყურადღება მიაქციეთ იმ ფაქტს, რომ შეტანა-გამოტანის ნებისმიერი გამონაკლისი, რომელიც შეიძლება შეიქმნას, უბრალოდ main() მეთოდს გადაეცემა.

პროგრამის კომპიუტერული რეალიზაცია:

```
package tread;
import java.io.*;
//სიმბოლოების კონსოლიდან წაკითხვა
public class BRRead {
	public static void main(String args[]) throws IOException{
		char c;
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
System.out.println("შემოიტანეთ სიმბოლოები: გამოსასვლელად დააწექით
'q' კლავიშს");
		//სიმბოლოების წაკითხვა
		do{
			c=(char)br.read();
			System.out.println(c);
		}while(c!='q');
	}
}
```

ნახ. 278

შედეგი:



ნახ. 279

კლავიატურიდან სტრიქონის წასაკითხად readLine() მეთოდს ვიყენებთ, რომელიც BufferedReader კლასის წევრია. მისი ჩაწერის ზოგადი ფორმა შემდეგია:

String readLine() throws IOException

როგორც ხედავთ, ის String კლასის ობიექტს აბრუნებს.

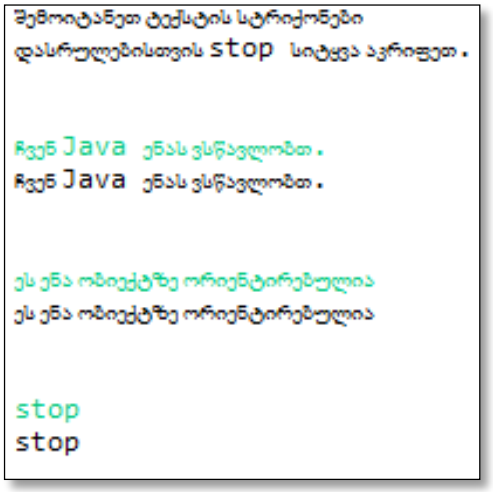
მაგალითი 20. შევადგინოთ პროგრამა, რომელიც კონსოლიდან კითხულობს და ტექსტის სტრიქონებს მანამდე წარმოგვიდგენს, სანამ სიტყვა „stop”-ს არ შეიყვანთ.

პროგრამის კომპიუტერული რეალიზაცია:

```
package tread;
import java.io.*;
//სიმბოლოების კონსოლიდან წაკითხვა
public class BRRead {
	public static void main(String args[]) throws IOException{
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		String str;
		System.out.println("შემოიტანეთ ტექსტის სტრიქონები");
		System.out.println("დასრულებისთვის stop სიტყვა
აკრიფეთ.");
		do{
			str=br.readLine();
			System.out.println(str);
		}while(!str.equals("stop")); }}
```

ნახ. 280

შედეგი:



ნახ. 281

რადგან PrintStream კლასი OutputStream კლასიდან გამომდინარეობს და გამომავალ ნაკადს აღწერს, შესაბამისად ის დაბალი დონის write() მეთოდის რეალიზებასაც ახდენს.

PrintStream კლასში განსაზღვრული write() მეთოდის ჩაწერის უმარტივესი ფორმა შემდეგია:

void write(int ბაიტის მნიშვნელობა)

ეს მეთოდი „ბაიტის მნიშვნელობა“ პარამეტრში გადაცემული ბაიტის ჩაწერას ასრულებს. მართალია, ეს პარამეტრი გამოცხადებულია როგორც მთელრიცხვა, მაგრამ ჩაწერა მხოლოდ მისი უმცროსი 8 ბიტის ხდება.

მაგალითი 21. შევადგინოთ პროგრამა, რომელიც write() მეთოდს ეკრანზე 'D' სიმბოლოს გამოსატანად გამოიყენებს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package tread;
import java.io.*;
public class BRRead {
	public static void main(String args[]){
		int b;
		b='D';
		System.out.write(b);
		System.out.write('\n');
	}
}
```

ნახ.282

ბუნებრივია, ამ პროგრამის შესრულების შედეგს „D“ ასო წარმოადგენს.

PrintWriter კლასი რამდენიმე კონსტრუქტორს განსაზღვრავს. მათგან ერთ-ერთის ჩაწერის ზოგადი ფორმა შემდეგია:

PrintWriter(OutputStream გამოტანის ნაკადი, boolean ახალი სტრიქონის შემთხვევაში ჩამოგდება)

აქ „გამოტანის ნაკადი“ OutputStream კლასის ობიექტია, ხოლო „ახალი სტრიქონის შემთხვევაში ჩამოგდება“ ბუფერის მართვას გულისხმობს. კერძოდ, თუ მისი მნიშვნელობა ჭეშმარიტია (true), მაშინ ბუფერის ავტომატური ჩამოგდება ხორციელდება.

PrintWriter კლასის ობიექტით კონსოლზე დაწერის მიზნით კლასის კონსტრუქტორში გამოსასვლელი ნაკადის სახით System.out ნაკადი მიუთითეთ.

კონსოლურ გამოტანასთან დაკავშირებული PrintWriter კლასის ობიექტის შესაქმნელად შემდეგი კოდი გამოიყენება:

PrintWriter pw = new PrintWriter(System.out, true);

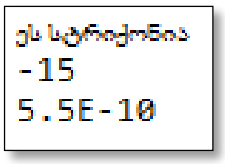
მაგალითი 22. შევადგინოთ პროგრამა, რომელიც PrintWriter კლასის გამოყენების დემონსტრირებას მოახდენს.

პროგრამის კომპიუტერული რეალიზაცია:

```
ackage tread;
import java.io.*;
public class BRRead {
	public static void main(String args[]){
	PrintWriter pw = new PrintWriter(System.out, true);
		pw.println("ეს სტრიქონია");
		int i=-15;
		pw.println(i);
		double d=5.5e-10;
		pw.println(d); } }
```

ნახ. 283

შედეგი:



ნახ.284

5.5.4. InputStream და OutputStream კლასები

როგორც უკვე აღვნიშნეთ InputStream კლასი აბსტრაქტულია, რომელიც Java-ს ბაიტის ტიპის ნაკადური შეტანის მოდელს განსაზღვრავს. ის AutoCloseable და Closeable ინტერფეისების რეალიზებას ახდენს. შეტანა-გამოტანის შეცდომების შემთხვევაში ამ კლასის მეთოდების უმრავლესობა IOException ტიპის გამონაკლისს გადასცემს.

InputStream კლასში განსაზღვრული მეთოდები 37-ე ცხრილშია წარმოდგენილი.

ცხრილი 37. InputStream კლასში განსაზღვრული ზოგიერთი მეთოდი

მეთოდი	აღწერა
int available()	აბრუნებს შეტანის ბაიტების რაოდენობას, რომლებიც მოცემულ მომენტში წვდომადია კითხვისთვის
void close()	კეტავს (ხურავს) შეტანის წყაროს. წაკითხვის შემდგომი მცდელობები IOException ტიპის გამონაკლისს გადასცემენ
boolean markSupported()	აბრუნებს ჭეშმარიტ (true) მნიშვნელობას, თუ mark() და reset() მეთოდებს გამომძახებელი ნაკადის მხარდაჭერა აქვთ
int read()	ნაკადში წვდომადი ბაიტის მთელრიცხვა მნიშვნელობას აბრუნებს. ფაილის დასასრულს -1 ბრუნდება
void reset()	შესასვლელ მიმთითებელს აბრუნებს წინანდელ მდგომარეობაში

OutputStream კლასი აბსტრაქტულია, რომელიც ბაიტის ტიპის ნაკადურ გამოტანას განსაზღვრავს. ის AutoCloseable, Closeable და Flushable ინტერფეისების რეალიზებას ახდენს. აღნიშნული კლასის მეთოდების უმრავლესობა void მოდიფიკაციის მქონეა და შეტანა-გამოტანის შეცდომების შემთხვევაში IOException ტიპის გამონაკლისს გადასცემს.

OutputStream კლასის მეთოდები 38-ე ცხრილშია წარმოდგენილი.

ცხრილი 38. OutputStream კლასში განსაზღვრული ზოგიერთი მეთოდი

მეთოდი	აღწერა
int close()	კეტავს გამოსასვლელ ნაკადს. ჩაწერის შემდგომი მცდელობები IOException ტიპის გამონაკლისს გადასცემენ
void flush()	ახდენს გამოსასვლელი მდგომარეობის ფინალიზირებას. ასუფთავებს ყველა ბუფერს (გამოტანის ბუფერებს)
void write(int b)	გამოსასვლელ ნაკადში ერთადერთ ბაიტს წერს
void write(byte ბუფერი[])	გამოსასვლელ ნაკადში წერს ბაიტების მასივს

ბაიტების გაფილტვრადი ნაკადები

ბაიტების გაფილტვრადი ნაკადები შემავალ და გამავალ ნაკადებთან დაკავშირებული დამატებითი ფუნქციონალური შესაძლებლობების მქონე ნაკადებია. ისინი იმ მეთოდებისთვისაა წვდომადი, რომლებიც მოელიან ნაკადს, რომელიც თავის მხრივ, გაფილტვრადი ნაკადის სუპერ კლასს წარმოადგენს. ასეთ შესაძლებლობებს ბუფერიზაცია, სიმბოლური და საბაზო მონაცემების გარდასახვა და სხვა წარმოადგენს. ბაიტების გაფილტვრადი ნაკადებია FilterInputStream და FilterOutputStream, რომელთა კონსტრუქტორების ჩაწერის ზოგადი ფორმები შემდეგია:

```
FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)
```

ამ კლასებში წარმოდგენილი მეთოდები InputStream და OutputStream კლასების მეთოდების იდენტურია.

ბაიტის ტიპის ნაკადებისთვის ბუფერიზირებული ნაკადები გაფილტვრადი ნაკადების კლასს აფართოებენ, მეხსიერებაში უმატებენ რა მას ბუფერს. ეს ბუფერი Java-ს უფლებას აძლევს შეტანა-გამოტანის ოპერაციები ყოველ ჯერზე ერთ ბაიტზე მეტი ოდენობით შეასრულოს. ბუფერზე წვდომის გამო შესაძლებელია ნაკადის გამოტოვება, მარკირება და ჩამოგდება. ბუფერი-ზირებული ბაიტის ტიპის ნაკადები BufferedInputStream და BufferedOutputStream კლასებს მიეკუთვნება.

შეტანა-გამოტანის ბუფერიზაცია პროგრამის წარმადობის ოპტიმიზაციის საკმაოდ გავრცე-ლებული საშუალებაა. BufferedInputStream კლასს ორი კონსტრუქტორი გააჩნია, რომელთა ჩაწერის ზოგადი ფორმებია:

```
BufferedInputStream(InputStream შემავალი ნაკადი)
BufferedInputStream(InputStream შემავალი ნაკადი, int ბუფერის ზომა)
```

პირველი ფორმა ბუფერიზირებულ ნაკადს ქმნის, რომელიც ბუფერის სტანდარტულ ზომას იყენებს. მეორე ფორმის შემთხვევაში, ბუფერის ზომა პარამეტრში „ბუფერის ზომა“ ეთითება. ბუფერის არასავალდებულო ზომა ძირითადად, ოპერაციულ სისტემაზე, მეხსიერების მოცულობასა და კომპიუტერის კონფიგურაციაზეა დამოკიდებული. ზოგადად, შეტანა-გამოტანის ნაკადისთვის ბუფერის ზომა 8192 ბაიტი ან ნაკლებიც, სავსებით საკმარისია. ამგვარად, დაბალი დონის სისტემას მონაცემთა ბლოკის წაკითხვა დისკიდან ან თუნდაც, ქსელიდანაც შეუძლია და შემდომ შედეგებს თქვენს ბუფერში შეინახავს.

მაგალითი 23. შევადგინოთ პროგრამა, სადაც ადგილი აქვს შემდეგი სიტუაციის მოდელირებას: განიხილება ნაკადი, რომელიც HTML კონსტრუქციას (რაც საავტორო უფლებების სიმბოლოზე მიუთითებს) მოიცავს. მიმართვა &-ის ნიშნით იწყება და „ ; “ -ით სრულდება. შეტანის ნიმუში ორ &-ს მოიცავს, იმისათვის რომ ვაჩვენოთ, თუ როდის მუშაობს (ან არ მუშაობს) reset() მეთოდი.

პროგრამის კომპიუტერული რეალიზაცია:

```
package thread;
import java.io.*;
//ბუფერიზირებული შეტანის გამოყენება
public class BufferedInputStramDemo {
	public static void main(String args[]){
		String s="This is a &copy; copyright symbol:" + "but this is
&copy not.\n";
		byte buf[]=s.getBytes();
		ByteArrayInputStream in=new ByteArrayInputStream(buf);
		int c;
		boolean marked=false;
		try(BufferedInputStream f = new BufferedInputStream(in))
		{
		while((c=f.read())!=-1){
		switch(c){
		case '&':
		if(!marked){
		f.mark(32);
		marked=true;
			}
		else{
		marked=false;
			}
		break;
		case ';':
		if(marked){
		marked=false;
		System.out.print("(c)");
				}
		else
		System.out.print((char) c);
		break;
```

```
case ' ':
       if(marked){
       marked=false;
       f.reset();
       System.out.print("&");
                              }
       else System.out.print((char) c);
       break;
       default:
              if(!marked)
                     System.out.print((char) c);
                     break;}}

}catch(IOException e){
       System.out.println("I/O Error:" + e); }}}
```

ნახ. 285

შედეგი:

```
This is a (c) copyright symbol:but this is &copy not.
```

ნახ. 286

ყურადღება მიაქციეთ იმ ფაქტს, რომ ზემოთ წარმოდგენილ პროგრამაში mark(32) მეთოდი გამოიყენება, რაც შემდეგი 32 ბაიტის წაკითხვისთვის საჭირო ჭდეს ინახავს.

BufferedOutputStream კლასი OutputStream კლასის ნებისმიერი კლასის მსგავსია. განმასხვავებელი მხოლოდ flush() მეთოდია, რომელიც ბუფერიზებულ ნაკადში მონაცემების ჩაწერას უზრუნველყოფს. Java-ს გამოტანის ბუფერები პროგრამის წარმადობის ასამაღლებლად გამოიყენება. ამ კლასის ორი წვდომადი კონსტრუქტორის ჩაწერის ზოგადი ფორმები შემდეგია:

BufferedOutputStream(OutputStream გამომავალი ნაკადი)

BufferedOutputStream(OutputStream გამომავალი ნაკადი, int ბუფერის ზომა)

პირველი ფორმა ბუფერიზირებულ ნაკადს ქმნის, რომელიც ბუფერის სტანდარტულ ზომას იყენებს. მეორე ფორმის შემთხვევაში, ბუფერის ზომა პარამეტრით „ბუფერის ზომა“ გადაიცემა.

5.5.5. Reader და Writer კლასები

როგორც უკვე აღვნიშნეთ, სიმბოლური ნაკადების იერარქიის სათავეში Reader და Writer აბსტრაქტული კლასებია დგას. Reader კლასი java-ში სიმბოლურ ნაკადურ შეტანას განსაზღვრავს. ის ახდენს AutoCloseable, Closeable და Readable ინტერფეისების რეალიზებას. კლასის ყველა მეთოდი, markSupported() მეთოდის გარდა, შეცდომის შემთხვევაში, IOException ტიპის გამონაკლისს გადასცემს.

Reader კლასის მეთოდები 39-ე ცხრილშია წარმოდგენილი.

ცხრილი 39. Reader კლასის მეთოდები

მეთოდი	აღწერა
abstract void close ()	კეტავს შემავალ ნაკადს. წაკითხვის ყოველი მომდევნო მცდელობა IOException ტიპის გამონაკლისს გადასცემს.
void mark (int სიმბოლოების რაოდენობა)	ჭდეს ათავსებს გამომავალი ნაკადის მიმდინარე პოზიციაში, რომელიც კორექტულია მანამ, სანამ „სიმბოლოების რაოდენობა“ პარამეტრში მითითებული რაოდენობის სიმბოლოები არ იქნება წაკითხული.
boolean marcSupported ()	აბრუნებს ჭეშმარიტ მნიშვნელობას, თუ mark() და reset() მეთოდებს ნაკადის მხარდაჭერა აქვთ.
int read ()	აბრუნებს შემავალი ნაკადის მომდევნო წვდომადი სიმბოლოს მთელრიცხვა წარმოდგენას. ფაილის დასასრულის მიღწევისას ბრუნდება -1.
int read (char ბუფერი[])	„ბუფერი.lenght” სიმბოლომდე სიმბოლოების წაკითხვას ცდილობს ბუფერში და აბრუნებს წარმატებით წაკით-ხული სიმბოლოების რაოდენობას. ფაილის დასასრულს -1-ს აბრუნებს.
int read (charBufeer ბუფერი)	ბუფერში ცდილობს სიმბოლოების წაკითხვას და აბრუნებს წარმატებით წაკითხული სიმბოლოების ფაქ-ტიურ რაოდენობას. ფაილის დასასრულს -1-ს აბრუნებს.
boolean ready ()	აბრუნებს true (ჭეშმარიტ) მნიშვნელობას, თუ შემდეგ მო-თხოვნას ლოდინი არ მოუწევს. წინააღმდეგ შემთხვევაში აბრუნებს false (მცდარ) მნიშვნელობას.
void reset ()	შეტანის მიმთითებელს ადრე დაფიქსირებულ პო-ზიციაში აბრუნებს
long skeep (long სიმბოლოების რაოდენობა)	გამოტოვებს „სიმბოლოების რაოდენობის“ ტოლ სიმ-ბოლოებს და აბრუნებს რეალურად გამოტოვებული სიმ-ბოლოების რაოდენობას.

Writer კლასი ასევე აბსტრაქტულია, რომელიც სიმბოლურ ნაკადურ გამოტანას განსაზღვ-რავს. ის ახდენს AutoCloseable, Closeable, Flushable და Appendable ინტერფეისების რეალიზებას. შეცდომის შემთხვევაში, აღნიშნული კლასის ყველა მეთოდი IOException ტიპის გამონაკლისს გადასცემს. Writer კლასის მეთოდები მე-40-ე ცხრილშია წარმოდგენილი.

ცხრილი 40. Writer კლასის მეთოდები

მეთოდი	აღწერა
Writer append (char სიმბოლო)	სიმბოლოს ამატებს გამომავალი ნაკადის ბოლოს. მიმართვას გამომძახებელ ნაკადზე აბრუნებს.
Writer append (charSecquence სიმბოლოები)	სიმბოლოებს ამატებს გამომავალი ნაკადის ბოლოს. მიმართვას გამომძახებელ ნაკადზე აბრუნებს.
Writer append (charSecquence სიმბოლოები, int დასაწყისი, int დასასრული)	„დასაწყისი“ და „დასასრული“ პარამეტრებით მითითებული დიაპაზონით სიმბოლოებს ამატებს გამომავალი ნაკადის ბოლოს. მიმართვას გამომძახებელ ნაკადზე აბრუნებს.
abstract void close ()	კეტავს გამომძახებელ ნაკადს. ჩაწერის ყოველი მომდევნო მცდელობა IOException ტიპის გამონაკლისს გადასცემს.
abstract void flush ()	გამომავალი მდგომარეობის ფინალიზირებას ისე ახდენს, რომ ყველა ბუფერი გასუფთავდეს.
void write (int სიმბოლო)	გამომავალ ნაკადში ერთადერთ სიმბოლოს წერს (მხოლოდ უმცირესი 16 ბიტი იწერება)
void write (char ბუფერი [])	გამომავალ ნაკადში სიმბოლოების სრულ მასივს წერს.
void write (String სტრიქონი)	გამომავალ ნაკადში პარამეტრში „სტრიქონი“ მითითებულ სტრიქონს წერს.

6. ქსელური პროგრამირება Java ენაზე

ცნობილია, რომ Java პრაქტიკულად პროგრამირების სინონიმია ინტერნეტისთვის. ამის მრავალი მიზეზი არსებობს, რომელთაგან ერთ-ერთი უსაფრთხო, მრავალპლატფორმიანი და გადატანითი კოდის შექმნაა. მაგრამ, ყველაზე მნიშვნელოვანი მიზეზი, რის გამოც Java ქსელური პროგრამირების ერთ-ერთ ბრწყინვალე ენად ითვლება, იმ კლასებშია ჩადებული, რომლებიც java.net პაკეტშია განსაზღვრული.

უზრუნველყოფა კლიენტი/სერვერი კომპიუტერს იყენებს, რომელიც სპეციალურ სერვერ-პროგრამას ასრულებს და ის კლიენტ-პროგრამებს სხვადასხვა მომსახურებას სთავაზობს. კლიენტი არის პროგრამა, რომელიც სერვერიდან იღებს მომსახურებას. კლიენტი კავშირს ამყარებს სერვერთან და უგზავნის მას მოთხოვნას. სერვერი კლიენტების „მოსმენას“ აწარმოებს, ხოლო კლიენტთან დაკავშირების შემდეგ იღებს და ასრულებს მოთხოვნებს.

მოთხოვნის შესრულების შედეგი შეიძლება სერვერიდან დაუბრუნდეს კლიენტს. მოთხოვნები და შეტყობინებები ჩანაწერებია, რომელთა სტრუქტურა პროტოკოლებით განისაზღვრება. ამ პროტოკოლებს განვიხილავთ მიმდინარე სახელმძღვანელოში.

6.1. ქსელთან მუშაობის საფუძვლები

Java-ს ქსელურ მხარდაჭერას საფუძვლად **სოკეტის** (socket) კონცეფცია უდევს. ქსელის საბოლოო წერტილის იდენტიფიცირებას სოკეტი ახდენს. სოკეტის პარადიგმა გასული საუკუნის 80-ან წლებში 4.2 BSD Berkley UNIX ვერსიაში გამოჩნდა. სწორედ ამ მიზეზით გამოიყენება ტერმინი **ბერკლის სოკეტი.**

სოკეტები თანამედროვე ქსელების საფუძვლებია, რადგან სოკეტი ცალკეულ კომპიუტერს უფლებას აძლევს ერთდროულად გაუწიოს მომსახურება როგოც მრავალ კლიენტს, ასევე სხვადასხვა ტიპის არაერთ ინფორმაციას. ეს **პორტის** (port) გამოყენების ხარჯზე მიიღწევა, რომელიც განსაზღვრულ კომპიუტერზე ნუმირებულ სოკეტს წარმოადგენს. ამბობენ, რომ სერვერული პროცესი პორტს მანამდე „უსმენს“, სანამ კლიენტი არ დაუკავშირდება მას. სერვერს მრავალი კლიენტის მიღება შეუძლია, რომლებიც პორტის ერთსა და იმავე ნომერთან არიან ჩართულები, თუმცა ყოველი სეანსი უნიკალურია.

სოკეტური კომუნიკაციები განსაზღვრული პროტოკოლით სრულდება. **ინტერნეტ პროტოკოლი** (Internet Protocol - IP) დაბალი დონის მქონე მარშრუტის მიმცემი პროტოკოლია, რომელიც მონაცემებს პაკეტებად ყოფს და განსაზღვრულ მისამართზე ქსელით აგზავნის, რაც

ყველა პაკეტის მიღებას საჭირო მისამართზე ვერ უზრუნველყოფს. **გადაცემების მართვის პროტოკოლი** (Transmission Control Protocol - TCP) უფრო მაღალი დონის პროტოკოლს წარმოადგენს, რომელიც ამ პაკეტების საიმედო შეკრებას, დახარისხებას და ხელახლა გადაგზავნას უზრუნველყოფს, რაც მონაცემთა საიმედო მიღებას განაპირობებს. არსებობს კიდევ ერთი პროტოკოლი - **სამომხმარებლო დეიტაგრამების პროტოკოლი** (User Datagram Protocol - UDP), რომელიც უშუალოდ TCP პროტოკოლის შემდეგ დგას. ის პაკეტების სწრაფი და საიმედო ტრანსპორტირების უზრუნველსაყოფად გამოიყენება. UDP ვირტუალურ კავშირს არ ამყარებს და ვერც მონაცემთა მიღების „გარანტიას“ იძლევა. გამგზავნი აღნიშნულ მისამართზე უბრალოდ აგზავნის პაკეტებს და თუ გაგზავნილი ინფორმაცია დაზიანდა ან მან ვერ მიაღწია დანიშნულების ადგილს, ამას გამგზავნი ვერც გაიგებს. UDP-ს ერთადერთი უპირატესობა მონაცემთა გადაცემის მაღალი სიჩქარეა. აღნიშნული პროტოკოლი აუდიო და ვიდეო-სიგნალების ტრანსლაციის დროს გამოიყენება, რადგან მონაცემთა მცირე რაოდენობის დაკარგვა მთელი ინფორმაციის სერიოზულ დამახინჯებას არ იწვევს.

UDP პროტოკოლით მონაცემები პაკეტებით გადაიცემა. ამ შემთხვევაში UDP-ს პაკეტს DatagramPacket კლასის ობიექტი წარმოადგენს. ეს კლასი გადასაცემ მონაცემებს ბაიტების მასივის სახით წარმოადგენს.

TCP/IP პროტოკოლების სტეკში შემდეგი პროტოკოლები გამოიყენება:

HTTP – Hypertext Transfer Protocol (WWW);

NNTP – Network News Transfer Protocol (სიახლეების ჯგუფები);

SMTP – Simple Mail Transfer Protocol (საფოსტო გზავნილი);

POP3 – Post Office Protocol (ფოსტის წაკითხვა სერვერიდან);

FTP – File Transfer Protocol (ფაილების გადაცემის პროტოკოლი).

ქსელში ჩართულ ყოველ კომპიუტერს TCP/IP პროტოკოლის მიხედვით უნიკალური IP-მისამართი გააჩნია, რომელიც იდენტიფიცირებისა და კავშირის დასამყარებლად გამოიყენება. ის 32-ბიტიან რიცხვს წარმოადგენს, რომელიც, როგორც წესი, ოთხი რიცხვის სახითაა ჩაწერილი და გამოყოფილია წერტილებით. თითოეული მათგანი 0-დან 255-მდე რიცხვით დიაპაზონში იცვლება. IP-მისამართი შეიძლება იყოს დროებითი და დინამიურად გამოიყოს ყოველი დაკავშირების დროს ან იყოს მუდმივი. IP-მისამართები შიდა ქსელურ სისტემებში გამოიყენება.

კავშირის დამყარებისთანავე მაღალი დონის პროტოკოლი გამოიყენება, რაც გამოსაყენებელ პორტზეა დამოკიდებული. TCP/IP პროტოკოლი სპეციფიური პროტოკოლებისთვის პირველი 1024 პორტის რეზერვირებას ახდენს. მაგალითად, პორტი №21 FTP პროტოკოლისთვისაა განკუთვნილი, №23 - Telnet პროტოკოლისთვის, №25 - ელექტრონული ფოსტისთვის, №80 – HTTP პროტოკოლისთვის, №119 – netnews-თვის და ა.შ. ყოველი პროტოკოლი კლიენტის

პორტთან ურთიერთობის სახეს განსაზღვრავს. მაგალითად, HTTP პროტოკოლი სერვერების და ვებ-ბრაუზერების მიერ ჰიპერ-ტექსტისა და გრაფიკული გამოსახულებების გადასაცემად გამოიყენება. ის ვებ-სერვერების მიერ წარმოდგენილი ინფორმაციის ნახვის საკმაოდ მარტივი პროტოკოლია. ვნახოთ, თუ როგორ მუშაობს ეს პროტოკოლი. როდესაც კლიენტი HTTP სერვერიდან გამოითხოვს ფაილს, ადგილი აქვს წინასწარ განსაზღვრულ პორტზე გარკვეულ ფორმატში ფაილის სახელის გაგზავნას და მისი შიგთავსის წაკითხვას. სერვერი ასევე კოდის მდგომარეობის შესახებ იძლევა შეტყობინებას, იმისთვის, რომ კლიენტს აცნობოს იყო თუ არა მოთხოვნა დამუშავებული და რა მიზეზით.

ინტერნეტის მნიშვნელოვან კომპონენტს **მისამართი** წარმოადგენს. ყოველ კომპიუტერს ინტერნეტში საკუთარი მისამართი გააჩნია. ის რიცხვს წარმოადგენს, რომელიც ინტერნეტში უნიკალურად ახდენს ყოველი კომპიუტერის იდენტიფიცირებას. ისევე როგორც IP-მისამართი აღწერს ქსელურ იერარქიას, ინტერნეტ-მისამართის სახელი, რომელსაც **დომენურ** სახელს უწოდებენ, სახელების სივრცეში კომპიუტერის ადგილმდებარეობას წარმოგვიდგენს. მაგალითად, მისამართი: www.Herbschildt.com მიეკუთვნება com დომენს, (რომელიც აშშ-ს კომერციული საიტებისთვისაა დარეზერვებული), გააჩნია სახელი Herbschildt (კომპანიის სახელის მიხედვით), ხოლო www იმ სერვერის იდენტიფიცირებას ახდენს, რომელიც ვებ-მოთხოვნებს ამუშავებს. ინტერნეტის დომენური სახელი IP-მისამართს დომენური სახელების მომსახურების (Domain Name Service - DNS) საშუალებით უკავშირდება. ეს მომხმარებლებს უფლებას აძლევს დომენური სახელებით იმუშაონ, მაშინ როდესაც ინტერნეტი IP-მისამართებით ოპერირებს.

6.1.1. ქსელური კლასები და ინტერფეისები

Java ენის მხარდაჭერა აქვს როგორც TCP/IP პროტოკოლს, ასევე TCP პროტოკოლების ოჯახს და UDP პროტოკოლს. TCP პროტოკოლი ქსელში მონაცემთა საიმედო ნაკადური შეტანა-გამოტანისთვის გამოიყენება, ხოლო UDP პროტოკოლი დეიტაგრამების გადაცემის სწრაფი მოდელის ფუნქციონირებას უზრუნველყოფს.

java.net პაკეტში არსებული კლასები 41-ე ცხრილშია წარმოდგენილი.

ცხრილი 41. java.net პაკეტის კლასები

Authenticator	Inet6Address	ServerSocket
CacherRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress	SockeImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager	MulticastSocket	StandartSocketOption
DatagramPacket	NetPermission	URI
DatagramSocket	NetworkInterface	URL
DatagramSocketImpl	PasswordAuthentication	URLClassLoader
HttpCookie	Proxy	URLConnection
HttpURLConnection	ProxySelector	URLDecoder
IDN	ResponseCache	URLEncoder
Inet4Address	SecureCacheResponse	URLStreamHandler

java.net პაკეტის ინტერფეისები კი 42-ე ცხრილშია წარმოდგენილი.

ცხრილი 42. java.net პაკეტის ინტერფეისები

ContentHandlerFactory	FileNameMap	SocketOptions
CookiePolicy	ProtocolFamily	URLStreamHandlerFactory
CookieStore	SocketImplFactory	
DatagramSocketImplFactory	SocketOption	

InetAddress კლასი როგორც რიცხვითი IP-მისამართის, ასევე მისი დომენური სახელის ინკაფსულირებისთვის გამოიყენება. თქვენ ურთიერთობთ კლასთან, იყენებთ რა IP-ჰოსტის სახელს. IP-მისამართთან შედარებით, ეს გაცილებით მარტივია.

InetAddress კლასს კონსტრუქტორი ცხადი სახით არ გააჩნია. ამ კლასის ობიექტის შე-საქმნელად საჭიროა factory method - მეთოდის გამოყენება, რაც გულისხმობს იმ ფაქტს, რომ

არსებობს მარტივი შეთანხმება, რომლის შესაბამისად, კლასის სტატიკური მეთოდები ამავე კლასის ეგზემპლიარს აბრუნებენ. ეს კონსტრუქტორის გადატვირთვის ნაცვლად გამოიყენება, როდესაც მეთოდების უნიკალური სახელების სიმრავლე შედეგს უფრო ნათელს ხდის. ქვემოთ წარმოდგენილია InetAddress კლასის ხშირად გამოყენებადი მეთოდები:

static InetAddress getLocalHost () throws UnknownHostException

static InetAddress getByName (String ჰოსტის სახელი) throws UnknownHostException

static InetAddress [ ] getAllByName (String ჰოსტის სახელი) throws UnknownHostException

getLocalHost() მეთოდი აბრუნებს InetAddress კლასის ობიექტს, რომელიც ლოკალურ ჰოსტს წარმოადგენს. getByName() მეთოდი აბრუნებს InetAddress კლასის ობიექტს, რომელსაც ჰოსტის სახელი პარამეტრად გადაეცა. თუ ეს მეთოდები ჰოსტის სახელს ვერ იღებენ, ისინი UnknownHostException ტიპის გამონაკლისს გადასცემენ.

როდესაც ინტერნეტში რამდენიმე კომპიუტერის წარმოსადგენად ერთი სახელი გამოიყენება, ეს ჩვეულებრივი მოვლენაა. getAllByName() მეთოდი InetAddress კლასის მასივს აბრუნებს, რომელიც ყველა იმ მისამართს წარმოგვიდგენს, რომლებშიც კონკრეტული სახელი გარდა-იქმნება. თუ მეთოდი ვერ ახერხებს სახელის თუნდაც ერთ მისამართზე გარდაქმნას, მსგავსად წინა მეთოდებისა, ისიც UnknownHostException ტიპის გამონაკლისს გადასცემს. InetAddress კლასი მოიცავს getByAddress() მეთოდსაც, რომელიც IP-მისამართს იღებს და InetAddress კლასის ობიექტს აბრუნებს.

მაგალითი 1. შევადგინოთ პროგრამა, რომელსაც კონსოლზე გამოაქვს ლოკალური კომპიუ-ტერისა და ორი ინტერნეტ-საიტის სახელები და მისამართები.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.net.*;
public class IntelAddressDemo {
	public static void main(String args[])throws
UnknownHostException{
		InetAddress Address=InetAddress.getLocalHost();
		System.out.println(Address);
		Address=InetAddress.getByName("www.gtu.ge");
		System.out.println(Address);
		InetAddress
SW[]=InetAddress.getAllByName("www.nba.com");
		for(int i=0; i<SW.length; i++)
			System.out.println(SW[i]);
	}
}
```

ნახ. 287

შედეგი:

```
Lela-PC/192.168.0.101
www.gtu.ge/109.205.46.3
www.nba.com/82.166.201.210
www.nba.com/82.166.201.208
```

ნახ. 288

კოდი, რომელსაც შედეგის სახით საკუთარ კომპიუტერზე იხილავთ, ცხადია, ჩვენ მიერ მიღებული შედეგისგან განსხვავებული იქნება.

6.1.2. ეგზემპლიარის მეთოდები

InetAddress კლასი სხვა მეთოდებსაც მოიცავს, რომლებიც ობიექტებთან სამუშაოდ გამოიყენება. ეს მეთოდები 43-ე ცხრილშია წარმოდგენილი.

ცხრილი 43. InetAddress კლასის მეთოდები

მეთოდი	აღწერა
boolean equals (Object სხვა)	აბრუნებს true (ჭეშმარიტ) მნიშვნელობას, თუ ობიექტს ის ინტერნეტ-მისამართი აქვს, რაც „სხვა“ პარამეტრშია მითითებული.
byte getAddress ()	აბრუნებს ბაიტების მასივს, რომელიც IP-მისამართს წარმოადგენს.
String getHostAddress ()	აბრუნებს სტრიქონს, რომელიც ჰოსტის მისამართს წარმოადგენს. ეს უკანასკნელი InetAddress კლასის ობიექტთან ასოცირდება.
String getHostName ()	აბრუნებს სტრიქონს, რომელიც ჰოსტის სახელს წარმოადგენს. ეს უკანასკნელი InetAddress კლასის ობიექტთან ასოცირდება.
boolean isMulticastAddress ()	აბრუნებს true (ჭეშმარიტ) მნიშვნელობას, თუ მისამართი ჯგუფურია, წინააღმდეგ შემთხვევაში აბრუნებს false (მცდარ) მნიშვნელობას.
String toString ()	აბრუნებს სტრიქონს, რომელიც მოიცავს ჰოსტის სახელს და IP-მისამართს.

ზოგადად, თქვენს ლოკალურ კომპიუტერს განსაზღვრული სახელი შეუძლია ავტომატურად შეუსაბამოს მის IP-მისამართს. სხვა სახელების შემთხვევაში, მას ასევე შეუძლია DNS სერვერებს

მიმართოს, საიდანაც IP-მისამართების შესახებ მიიღებს ინფორმაციას. თუ სერვერი განსაზღვრულ მისამართთან დაკავშირებით ინფორმაციას არ ფლობს, მას შემდეგ დისტანციურ საიტს შეუძლია მიმართოს და მას მოთხოვოს საჭირო ინფორმაცია. ეს პროცესი შესაძლოა მთავარ სერვერამდე გაგრძელდეს, რაც დროში მნიშვნელოვნად გაიჭიმება. ამიტომ, თქვენი კოდის სტრუქტურა ისე უნდა ააწყოთ, რომ IP-მისამართების შესახებ ინფორმაცია ლოკალურად იქნეს ქეშირებული და ყოველ ჯერზე ხელახალი ძიება საჭირო აღარ გახდეს.

6.1.3. TCP/IP სოკეტები

სოკეტები (ქსელური გასართები) ეს ლოგიკური ცნებაა, რომელიც იმ გასართებს შეესაბამება, რომლებთანაც ქსელური კომპიუტერებია დაკავშირებული და რომელთა საშუალებით მონაცემთა ორმხრივი ნაკადური გადაცემა წარმოებს კომპიუტერებს შორის.

სოკეტი პორტის ნომრით და IP - მისამართით განისაზღვრება. ამასთან, IP - მისამართი კომპიუტერის იდენტიფიცირებისთვის გამოიყენება, ხოლო პორტის ნომერი - პროცესის იდენტიფიცირებისთვის. კლიენტი ცდილობს დაუკავშირდეს სერვერს, ახდენს რა სოკეტური შეერთების ინიციალებას. კლიენტის მიერ სერვერზე გაგზავნილი პირველი შეტყობინება კლიენტის სოკეტს შეიცავს. თავის მხრივ, სერვერიც ქმნის სოკეტს, რომელიც შემდგომში კლიენტთან დასაკავშირებლად გამოიყენება და უგზავნის მას კლიენტს. ამის შემდეგ მყარდება კომუნიკაცია.

კლიენტის მიერ სერვერთან სოკეტური დაკავშირება Socket კლასის ობიექტის შექმნის საშუალებით ხდება. ამასთან, ეთითება სერვერის IP - მისამართი და პორტის ნომერი. თუ დომენის სიმბოლური სახელია მითითებული, მაშინა Java IP - მისამართად გარდაქმნის მას DNS-სერვერის საშუალებით.

TCP/IP სოკეტები ინტერნეტში ნაკადების საფუძველზე ჰოსტების ორმხრივი მუდმივი შეერთებების სარეალიზაციოდ გამოიყენება. სოკეტი ასევე შეგვიძლია გამოვიყენოთ Java-ს შეტანა-გამოტანის სისტემის სხვა პროგრამებთან დასაკავშირებლად, რომლებიც შესაძლოა განთავსებული იყოს როგორც ლოკალურ კომპიუტერზე, ასევე ნებისმიერ სხვა კომპიუტერზე ინტერნეტში. თუმცა, როგორც წესი, აპლეტები სოკეტურ კავშირს მხოლოდ იმ ჰოსტთან ახდენენ, საიდანაც მათი ჩატვირთვა ხდება.

Java-ში TCP სოკეტების ორი სახე არსებობს - სერვერებისთვის და კლიენტებისთვის. ServerSocket კლასი „მსმენელს“ წარმოადგენს, რომელიც სანამ რაიმეს მოიმოქმედებდეს, კლიენტების ჩართვას ელოდება. სხვა სიტყვებით რომ ვთქვათ, ServerSocket კლასი სერვერებისთვის არის განკუთვნილი. Socket კლასი კი - კლიენტებისთვის. ეს უკანასკნელი (Socket კლასი)

ისეა შემუშავებული, რომ დაუკავშირდეს სერვერულ სოკეტებს და გაცვლა პროტოკოლის მიხედვით განახორციელოს.

44-ე ცხრილში წარმოდგენილია Socket კლასის ორი კონსტრუქტორი, რომლებიც კლიენტის სოკეტების შესაქმნელად გამოიყენება.

ცხრილი 44. Socket კლასის კონსტრუქტორები

კონსტრუქტორი	აღწერა
Socet(String ჰოსტის სახელი, int პორტი) throws UnknownHostException, IOException	ქმნის სოკეტს, რომელიც ჩართულია ინიცია-ლებულ ჰოსტსა და პორტთან.
Socet(InetAddress, IP-მისამართი, int პორტი) throws IOException	იყენებს რა ადრე არსებულ InetAddress კლა-სის ობიექტს და პორტს, ქმნის სოკეტს.

Socket კლასი ეგზემპლიარის რამდენიმე მეთოდს განსაზღვრავს. ისინი 45-ე ცხრილშია წარმოდგენილი.

ცხრილი 45. Socket კლასის ეგზემპლიარის მეთოდები

მეთოდი	აღწერა
InetAddress getInetAddress()	აბრუნებს InetAddress კლასის ობიექტს, რომელიც Socket კლასის ობიექტთან ასოცირდება. თუ სოკეტი არ არის ჩართული, null მნიშვნელობას აბრუნებს.
int getPort()	აბრუნებს დისტანციურ პორტს, რომელთანაც Socket კლასის ობი-ექტია ჩართული. თუ სოკეტი არ არის ჩართული, 0 მნიშვნელობას აბრუნებს.
int getLocalPort()	აბრუნებს ლოკალურ პორტს, რომელთანაც Socket კლასის ობიექ-ტია დაკავშირებული. წინააღმდეგ შემთხვევაში აბრუნებს -1-ის ტოლ მნიშვნელობას.

Socket კლასთან ასოცირებულ შემავალ და გამავალ ნაკადებზე წვდომის მისაღებად getInputStream() და getOutputStream() მეთოდები გამოიყენება. ეს ნაკადები მონაცემთა მისაღებად და გადასაცემად ისევე გამოიყენება, როგორც ჩვენ მიერ ადრე განხილული ნაკადები. ეს მეთოდები 46-ე ცხრილშია წარმოდგენილი.

ცხრილი 46. Socket კლასთან დაკავშირებული შეტანა-გამოტანის ნაკადებზე წვდომის მეთოდები

მეთოდი	აღწერა
InputStream get InputStream() throws IOException	აბრუნებს InetAddress კლასის ობიექტს, რომელიც გამომძახებელ სოკეტთან ასოცირდება.
OutputStream get OutputStream() throws IOException	აბრუნებს OutputStream კლასის ობიექტს, რომელიც გამომძახებელ სოკეტთან ასოცირდება.

აქვე გამოიყენება connect() მეთოდი, რომელიც უფლებას გვაძლევს ახალი ჩართვა (დაკავშირება) განვახორციელოთ. isConnected() მეთოდი true მნიშვნელობას აბრუნებს, თუ სოკეტი სერვერთან ჩართულია. isBound() მეთოდი true მნიშვნელობას აბრუნებს, თუ სოკეტი მისამართთან დაკავშირებულია. isClosed() მეთოდი true მნიშვნელობას აბრუნებს, როდესაც სოკეტი დაკეტილია. close() მეთოდი სოკეტის დასაკეტად გამოიძახება. სოკეტის დაკეტვა მასთან დაკავშირებული შეტანა-გამოტანის ნაკადების დაკეტვასაც იწვევს.

მაგალითი 2. შევადგინოთ Socket კლასის გამოყენების სადემონსტრაციო პროგრამა. კერძოდ, ის ხსნის „whois" (პორტი 43) პორტთან კავშირს InterNIC სერვერზე, სოკეტს უგზავნის ბრძანების ხაზის არგუმენტებს და შემდგომ კონსოლზე გამოაქვს დაბრუნებული მონაცემები. InterNIC სერვერი ცდილობს არგუმენტი აღიქვას როგორც ინტერნეტის დარეგისტრირებული დომენის სახელი, შემდეგ ის აბრუნებს IP-მისამართს და ამ საიტის საკონტაქტო ინფორმაციას.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.net.*;
import java.io.*;
public class Sockets {
	public static void main(String args[])throws Exception{
		int c;
		//სოკეტური დაკავშირება internic.net-თან პორტი 43.
		Socket s=new Socket("whois.internic.net", 43);
		//შემავალი და გამომავალი ნაკადების მიღება
		InputStream in=s.getInputStream();
		OutputStream out=s.getOutputStream();
		//მოთხოვნის სტრიქონის შექმნა
		String str=(args.length==0 ? "MHProfessional.com" : args[0]) +"\n";
		//ბიატებში გადაყვანა
		byte buf[]=str.getBytes();
		//მოთხოვნის გაგზავნა
		out.write(buf);
		//კითხვა და პასუხის გამოტანა
		while ((c = in.read()) != -1){
			System.out.print((char) c);
		}
		s.close();}}
```

შედეგი:

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: MHPROFESSIONAL.COM
   Registrar: CSC CORPORATE DOMAINS, INC.
   Sponsoring Registrar IANA ID: 299
   Whois Server: whois.corporatedomains.com
   Referral URL: http://www.cscglobal.com/global/web/csc/digital-brand-services
   Name Server: PDNS85.ULTRADNS.BIZ
   Name Server: PDNS85.ULTRADNS.COM
   Name Server: PDNS85.ULTRADNS.NET
   Name Server: PDNS85.ULTRADNS.ORG
   Status: clientTransferProhibited
http://www.icann.org/epp#clientTransferProhibited
   Updated Date: 06-jun-2015
   Creation Date: 09-jun-2006
   Expiration Date: 09-jun-2016 …
```

ნახ. 290

ახლა კი განვიხილოთ, თუ როგორ მუშაობს ზემოთ წარმოდგენილი პროგრამა. თავდაპირველად, Socket კლასის ობიექტი იქმნება, რომელიც იძლევა ჰოსტის სახელს "whois.internic.net" და პორტის ნომერს (43). internic.net - ეს არის InterNIC ვებ-საიტი, ის ამუშავებს whois მოთხოვნებს. 43-ე პორტი სწორედ ამას ემსახურება. შემდეგ შემავალი და გამომავალი ნაკადები სოკეტში იხსნება. შემდგომ იქმნება სტრიქონი, რომელიც იმ ვებ-საიტის სახელს მოიცავს, რომლის შესახებაც გვსურს ინფორმაციის მიღება. თუ საიტი მითითებული არ იქნა, გამოიყენება „MHProfessional.com“. სტრიქონი გარდაიქმნება ბაიტის ტიპის მასივად და გადაეგზავნება სოკეტს. ამის შემდეგ, პასუხი სოკეტიდან იკითხება და შედეგი ეკრანზე გამოიტანება. პროგრამის დასასრულს სოკეტი იხურება, რაც შეტანა-გამოტანის ნაკადების დახურვას განაპირობებს.

ამ პროგრამაში სოკეტი ხელით იქნა დახურული, close() მეთოდის გამოძახებით. სოკეტის ავტომატურად დახურვისთვის შეგვიძლია რესურსებიანი try ბლოკი გამოვიყენოთ.

ამ ცვლილების გათვალისწინებით მე-2 მაგალითის რეალიზების შესაბამის პროგრამა შემდეგ სახეს მიიღებს:

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.net.*;
import java.io.*;
public class Sockets {
	public static void main(String args[])throws Exception{
		int c;
		//სოკეტური დაკავშირება internic.net-თან პორტი 43.
		//ამ სოკეტს რესურსებიანი try ბლოკი მართავს
		try(Socket s=new Socket("whois.internic.net", 43)){
		//შემავალი და გამომავალი ნაკადების მიღება
		InputStream in=s.getInputStream();
		OutputStream out=s.getOutputStream();
		//მოთხოვნის სტრიქონის შექმნა
		String str=(args.length==0 ? "MHProfessional.com" : args[0]) +"\n";
		//ბიატებში გადაყვანა
		byte buf[]=str.getBytes();
		//მოთხოვნის გაგზავნა
		out.write(buf);
		//კითხვა და პასუხის გამოტანა
		while ((c = in.read()) != -1){
			System.out.print((char) c);}}
		//ახლა სოკეტი დახურულია
		}}
```

ნახ.291

პროგრამის ამ ვერსიაში try ბლოკის ბოლოს სოკეტი ავტომატურად იხურება.

6.1.4. URL კლასი

ცნობილია, რომ დღეს ინტერნეტში მსოფლიო „ობობის ქსელი“ **WWW** (World Wide Web) ბატონობს. ამიტომ, ისეთ (ძველ) პროტოკოლებთან, როგორებიცაა whois, finger ან FTP - ინტერნეტი მათთან აღარ ასოცირდება.

URL (Uniform Resource Locator) **რესურსების უნიფიცირებული ლოკატორი** ვებ-ში სამისამართო ინფორმაციის უნიკალური იდენტიფიკაციის საკმაოდ მკაფიო ფორმას უზრუნველყოფს. Java-ს კლასების ბიბლიოთეკაში URL კლასი ინფორმაციაზე წვდომის მარტივ, შეთანხმებულ პროგრამულ ინტერფეისს წარმოგვიდგენს მთელს ქსელში URL-ის დახმარებით.

ინტერნეტის ქსელზე წვდომისათვის ბრაუზერში URL მისამართი ეთითება. ის ორი ნაწილისგან შედგება: პროტოკოლის პრეფიქსისგან (http, https, ftp და ა.შ.) და URI (Universal Resource Identifier). URI მოიცავს ინტერნეტ-მისამართს, პორტის არასავალდებულო ნომერს და გზას კატალოგისკენ, რომელიც საჭირო ფაილს შეიცავს. მაგალითად:http://www.bsu.by.

URI არ შეიძლება მოიცავდეს ისეთ სპეციალურ სიმბოლოებს, როგორებიცაა „ჰარი“, ტაბულაცია. მათი წარმოდგენა თექვსმეტობითი კოდით არის შესაძლებელი. მაგალითად: %20 – „ჰარს“ აღნიშნავს. URI კლასი რესურსის უნივერსალური იდენტიფიკატორის (Uniform Resource

Identifier) ინკაფსულირებას ახდენს. ის URL-ს ძალიან გავს. სინამდვილეში, URL წარმოადგენს URI-ის ქვესიმრავლეს. URI რესურსების იდენტიფიცირების სტანდარტულ საშუალებას გვთავაზობს. URL რესურსზე წვდომას აღწერს. ყველა URL ერთი და იგივე საბაზო ფორმატს ერთობლივად იყენებს. თუმცა გარკვეული ვარიაციები აქაც დასაშვებია.

აღნიშნულის საილუსტრაციოდ ორი მაგალითი მოვიყვანოთ. http://www.MHProfessional.com/ და http://www. MHProfessional.com:80/index.html. URL სპეციფიკაცია ოთხ კომპონენტზეა დაფუძნებული. პირველი მათგანი გამოყენებული პროტოკოლია, რომელიც ლოკატორის დანარჩენი ნაწილისგან ორწერტილით (:) გამოიყოფა. გავრცელებულ პროტოკოლებს წარმოადგენს: HTTP, FTP, gopher და file. თუმცა დღეს თითქმის ყველაფერი HTTP პროტოკოლით სრულდება. ფაქტიურად, ბრაუზერების უმეტესობა კორექტულად მუშაობს, მაშინაც კი, თუ თქვენ URL სპეციფიკაციიდან "http://" ფრაგმენტს გამოტოვებთ. მეორე კომპონენტი ჰოსტის სახელი ან IP-მისამართია, რომელსაც ჰოსტი იყენებს. ის მარცხნიდან ორმაგი სლეშით (//) გამოიყოფა, ხოლო მარჯვნიდან - ერთი სლეშით (/) ან, რაც სავალდებულო არ არის, ორწერტილით (:). მესამე კომპონენტი პორტის ნომერია; ის არასავალდებულო პარამეტრია, რომელიც ჰოსტის სახელისგან მარცხნიდან ორწერტილით (:) გამოიყოფა, ხოლო მარჯვნიდან - სლეშით (/). (თუ პორტი №80 HTTP პროტოკოლის სტანდარტულ (მიუთითებლობის შემთხვევაში) პორტად მიიჩნევა, მაშინ „:80"-ის მითითება ზედმეტია). მეოთხე კომპონენტი ფაილისკენ მიმავალი რეალური გზაა. HTTP სერვერების უმრავლესობა ფაილის სახელს ამატებს index.html-ს ან index.htm-ს, რომლებიც უშუალოდ რომელიმე კატალოგზე მიუთითებენ. ამგვარად, http://www.MHProfessional.com/ იგივეა, რაც მისამართი: http://www.MHProfessional.com/index.htm.

URL კლასს რამდენიმე კონსტრუქტორი გააჩნია და ყოველ მათგანს MalformedURLException ტიპის გამონაკლისის გადაცემა შეუძლია. ამ კონსტრუქტორებიდან ერთ-ერთი ყველაზე გავრცელებული (გამოყენებადი) ფორმა URL-ს სტრიქონის სახით განსაზღვრავს. ქვემოთ წარმოდგენილი კონსტრუქტორის ამ ფორმის ჩანაწერი იმის იდენტურია, რასაც თქვენ ბრაუზერში ხედავთ.

URL (String სპეციფიკატორი URL) throws MalformedURLException

კონსტრუქტორის შემდეგი ორი ფორმა საშუალებას გაძლევთ URL ნაწილებად, კომპონენტებად გაყოთ.

URL (String პროტოკოლის სახელი, String ჰოსტის სახელი, int პორტი, String გზა) throws MalformedURLException

URL (String პროტოკოლის სახელი, String ჰოსტის სახელი, String გზა) throws MalformedURLException

მეორე, ასევე ხშირად გამოყენებადი კონსტრუქტორი უფლებას გვაძლევს არსებული URL კონტექსტის სახით მივუთითოთ და შემდეგ ამ კონტექსტიდან ახალი URL შევქმნათ.

URL(URL ობიექტი URL, String სპეციფიკატორი URL) throws MalformedURLException

მაგალითი 3. შევადგინოთ პროგრამა, რომელიც HerbSchildt.com სტატიების გვერდის URL-ს ქმნის და შემდეგ ნახულობს მის თვისებებს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.net.*;
public class URLDemo {
	public static void main(String args[])throws MalformedURLException{
		URL hp=new URL("http://www.HerbSchildt.com/Articles");
		System.out.println("პროტოკოლი:" + hp.getProtocol());
		System.out.println("პორტი:" + hp.getPort());
		System.out.println("ჰოსტი:" + hp.getHost());
		System.out.println("ფაილი:" + hp.getFile());
		System.out.println("სრულად:" + hp.toExternalForm());}}
```

ნახ. 292

შედეგი:

```
პროტოკოლი:http
პორტი:-1
ჰოსტი:www.HerbSchildt.com
ფაილი:/Articles
სრულად:http://www.HerbSchildt.com/Articles
```

ნახ. 293

მიაქციეთ ყურადღება პორტს: -1. ეს ნიშნავს, რომ პორტი ცხადი სახით დაყენებული არ არის. URL ობიექტის გადაცემით, მასთან ასოცირებული მონაცემების მიღება შეგვიძლია. იმისათვის, რომ URL-ით წვდომა რეალურ ბიტებზე ან ინფორმაციაზე გქონდეთ, მისგან URLConnection კლასის ობიექტი შექმენით. ამ მიზნით openConnection() მეთოდი ისე გამოიყენეთ, როგორც ეს ქვემოთ არის წარმოდგენილი:

urlc=url. openConnection;

openConnection() მეთოდს ჩაწერის შემდეგი ზოგადი ფორმა გააჩნია:

URLConnection OpenConnection() throws IOException

ის URLConnection კლასის ობიექტს აბრუნებს, რომელიც URL კლასის ობიექტთან ასოცირდება. მას IOException ტიპის გამონაკლისის გადაცემა შეუძლია.

6.1.5. URLConnection კლასი

URLConnection კლასი საერთო დანიშნულების კლასია, რომელიც შორს მყოფი რესურსის ატრიბუტებზე წვდომისთვის გამოიყენება. სერვერზე კავშირის ერთხელ დამყარების შემდეგ URLConnection კლასი შეგიძლიათ შორს მყოფი ობიექტის თვისებების სანახავად გამოიყენოთ, სანამ მის ლოკალურ ტრანსპორტირებას მოახდენდეთ. ეს ატრიბუტები HTTP პროტოკოლის სპეციფიკაშია წარმოდგენილი და აზრი მხოლოდ იმ URL ობიექტებისთვის აქვს, რომელთა ატრიბუტები HTTP პროტოკოლს იყენებენ.

URLConnection კლასი რამდენიმე მეთოდს განსაზღვრავს. ზოგიერთი მათგანი 47-ე ცხრილშია წარმოდგენილი.

ცხრილი 47. URLConnection კლასის მეთოდები

მეთოდი	აღწერა
int getContentLength()	აბრუნებს ზომას ბაიტებში. თუ ზომა არ არის წვდომადი, აბრუნებს -1-ს.
long getContentLengthLong()	აბრუნებს ზომას ბაიტებში. თუ ზომა არ არის წვდომადი, აბრუნებს -1-ს (დამატებულია JDK7-ში).
String getContentType()	აბრუნებს რესურსში მოძებნილ ტიპს. ეს არის content-type ველის სათაურის მნიშვნელობა. თუ ტიპი არ არის წვდომადი, აბრუნებს null მნიშვნელობას.
long getDate()	აბრუნებს პასუხის დროსა და თარიღს წარმოდგენილს მილიწამებში, რომელიც 1970 წლის პირველი იანვრიდან არის გასული.
long getExpiration()	აბრუნებს რესურსის დაძველების დროსა და თარიღს მილიწამებში, რომელიც 1970 წლის პირველი იანვრიდან არის გასული. თუ თარიღი მიუწვდომელია, აბრუნებს ნულს.
String getHeaderField (int ინდექსი)	აბრუნებს სათაურის ველის მნიშვნელობას „ინდექსი“-ს მიხედვით. (ინდექსების ნუმერაცია 0-დან იწყება). თუ ინდექსის მნიშვნელობა ველების რაოდენობას აღემატება, აბრუნებს null მნიშვნელობას.
String getHeaderField (int ველის სახელი)	აბრუნებს სათაურის ველის მნიშვნელობას, რომლის სახელი მითითებულია „ველის სახელ“-ში. თუ მითითებული ველი ვერ მოიძებნა, აბრუნებს null მნიშვნელობას.

String getHeaderFieldKey (int ინდექსი)	აბრუნებს სათაურის ველის გასაღებს „ინდექსის“ მიხედვით. (ინდექსების ნუმერაცია 0-დან იწყება). თუ ინდექსის მნიშვნელობა ველების რაოდენობას აღემატება, აბრუნებს null მნიშვნელობას.
Map<String,List<String>> getHeaderFields	აბრუნებს ყველა სათაურის ველისა და მათი მნიშვნელობების შემცველ რუკას.
long getLastModified()	აბრუნებს რესურსის ბოლო მოდიფიკაციის დროსა და თარიღს მილიწამებში, რომელიც 1970 წლის პირველი იანვრიდან არის გასული. თუ ეს ინფორმაცია მიუწვდომელია, აბრუნებს ნულს.
InputStream getInputStream() throws IOException	აბრუნებს InputStream კლასის ობიექტს, დაკავშირებულს რესურსთან. აღნიშნული ნაკადი რესურსის შიგთავსის მისაღებად შეიძლება იქნეს გამოყენებული.

ყურადღება მიაქციეთ იმ ფაქტს, რომ URLConnection კლასი რამდენიმე მეთოდს განსაზღვრავს, რომლებიც „სათაური“-ს ინფორმაციას მართავს. სათაური გასაღებებისა და მნიშვნელობების წყვილისგან შედგება, რაც სტრიქონების სახით არის წარმოდგენილი. getHeaderField() მეთოდის გამოყენებით თქვენ შეგიძლიათ სათაურის გასაღებთან ასოცირებული მნიშვნელობა მიიღოთ.

მაგალითი 4. შევადგინოთ პროგრამა, URLConnection კლასის ობიექტის შესაქმენლად URL კლასის ობიექტის openConnection() მეთოდს იყენებს. შემდეგ კი ადგილი აქვს დოკუმენტის შიგთავსისა და თვისებების შემოწმებას.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.net.*;
import java.io.*;
import java.util.Date;
public class UCDemo {
	public static void main(String args[])throws Exception{
		int c;
		URL hp=new URL("http://www.internic.net");
		URLConnection hpCon=hp.openConnection();
		//თარიღის მიღება
		long d=hpCon.getDate();
		if(d==0)
			System.out.println("თარიღის შესახებ ინფორმაცია არ არის.");
		else
			System.out.println("თარიღი: " + new Date(d));
		//ტიპის მიღება
		System.out.println("ტიპი: " + hpCon.getContentType());
		//დამველების თარიღის მიღება
		d=hpCon.getExpiration();
		if(d==0)
		System.out.println("მოქმედების ვადის  შესახებ ინფორმაცია არ არის.");
		else
		System.out.println("დამველდება: " + new Date(d));
		//ბოლო მოდიფიკაციის თარიღის მიღება
		d=hpCon.getLastModified();
		if(d==0)
		System.out.println("ბოლო მოდიფიკაციის  შესახებ ინფორმაცია არ არის.");
		else
			System.out.println("ბოლო მოდიფიკაციის თარიღი: " + new Date(d));
		//სიგრძის მიღება
		long len=hpCon.getContentLength();
		if(len==-1)
			System.out.println("სიგრძის შესახებ ნფორმაცია მიუწვდომელია.");
		else
			System.out.println("სიგრძე: " + len);
		if(len!=0){
			System.out.println("==== შიგთავსი====: ");
		InputStream input=hpCon.getInputStream();
		while(((c=input.read())!=-1)){
			System.out.print((char)c);
		}
		input.close();}
	else{
		System.out.println("შიგთავსი  მიუწვდომელია.");}}}
```

ნახ. 294

შედეგი:

```
თარიღი: Sun Nov 08 20:30:31 GET 2015
ტიპი: text/html; charset=UTF-8
დაძველდება: Sun Nov 08 21:30:31 GET 2015
ბოლო მოდიფიკაციის თარიღი: Sat Oct 17 02:22:48 GET 2015
სიგრძე: 8544
==== შიგთავსი====:
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>InterNIC | The Internet's Network Information Center</title>
…
```

ნახ. 295

ზემოთ წარმოდგენილი პროგრამა HTTP კავშირს www.internic.net სერვერთან ამყარებს მე-80 პორტით. შემდეგ ის რამდენიმე სათაურის მნიშვნელობას და შიგთავსს წარმოგვიდგენს.

6.1.6. HttpURLConnection კლასი

Java გვთავაზობს URLConnection კლასის ქვეკლასს, რომელიც HTTP კავშირის მხარდაჭერას უზრუნველყოფს. ეს HttpURLConnection კლასია. აღნიშნული კლასის ობიექტი აქაც URL კლასის ობიექტის openConnection() მეთოდით მიიღება, მაგრამ შედეგი HttpURLConnection კლასს უკავშირდება. მიიღებთ რა წარმოდგენილი კლასის ობიექტზე წვდომას, თქვენ მისი ნებისმიერი მეთოდის გამოძახებას შეძლებთ. ეს მეთოდები კი URLConnection კლასისგან მემკვიდრეობით არის მიღებული. თქვენ ასევე, შეგიძლიათ HttpURLConnection კლასში განსაზღვრული ნებისმიერი მეთოდი გამოიყენოთ, რომლებიც 48-ე ცხრილშია წარმოდგენილი.

ცხრილი 48. HttpURLConnection კლასის მეთოდები

მეთოდი	აღწერა
static boolean getFollowRedirects()	თუ გადამისამართება ავტომატურად ხორციელდება, აბრუნებს true მნიშვნელობას, წინააღმდეგ შემთხვევაში - false-ს.
String getRequesMethod()	აბრუნებს მოთხოვნის შესრულების მეთოდის სტრიქონულ წარმოდგენას. მიუთითებლობის შემთხვევაში GET მეთოდი გამოიყენება. სხვა მეთოდებიც წვდომადია, მაგ. POST

int getResponseCode() **throws IOException**	აბრუნებს HTTP პასუხის კოდს. თუ პასუხის კოდის მიღება შეუძლებელია, აბრუნებს -1-ს. კავშირის გაწყვეტისას IOException გამონაკლისი გადაიცემა.
String getResponseMessage() **throws IOException**	აბრუნებს პასუხის შეტყობინებას, რაც პასუხის კოდთან ასოცირდება. თუ შეტყობინება მიუწვდომელია, აბრუნებს null მნიშვნელობას.
static void setFollowRedirects(boolean როგორ)	თუ პარამეტრი „როგორ“ შეიცავს true მნიშვნელობას, ე.ი. გადამისამართება ავტომატურად ხდება. თუ ის false მნიშვნელობას შეიცავს, ე.ი. არ ხდება. მიუთითებლობის შემთხვევაში აღნიშნული პროცესი ავტომატურად ხორციელდება.
void setRequesMethod(String როგორ) **throws ProtocolException**	აყენებს მეთოდს, რომლითაც HTPP მოთხოვნები „როგორ“ პარამეტრის შესაბამისად სრულდება. მიუთითებლობის შემთხვევაში GET მეთოდი გამოიყენება. სხვა მეთოდებიც წვდომადია, მაგ. POST. თუ „როგორ“ პარამეტრში არასწორი მნიშვნელობაა მითითებული, ProtocolException გამონაკლისი გადაიცემა.

მაგალითი 5. შევადგინოთ პროგრამა, HttpURLConnection კლასის გამოყენებით. დავამყაროთ კავშირი www.google.com საიტთან. წარმოვადგინოთ მოთხოვნის მეთოდი, პასუხის კოდი, პასუხის შეტყობინება და პასუხის სათაურში გასაღებები და მნიშვნელობები.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.net.*;
import java.io.*;
import java.util.*;
public class HttpURLDemo {
	public static void main(String args[])throws Exception{
		URL hp=new URL("http://www.google.com");
			HttpURLConnection hpCon=(HttpURLConnection)
	hp.openConnection();
		//მოთხოვნის მეთოდის წარმოდგენა
System.out.println("მოთხოვნის მეთოდი: "  + hpCon.getRequestMethod());
		//პასუხის კოდის  წარმოდგენა
	System.out.println("პასუხის კოდი: "  + hpCon.getResponseCode());
```

```
            //პასუხის შეტყობინების  წარმოდგენა
System.out.println("პასუხის შეტყობინება: "   + hpCon.getResponseMessage());
            //სათაურის ველების სიის და მისი გასაღებების ნაკრების მიღება
            Map<String,List<String>> hdrMap=hpCon.getHeaderFields();
            Set<String> hdrField=hdrMap.keySet();
            System.out.println("\nაქედან სათაური იწყება:");
            //სათაურის ყველა გასაღებისა და მნიშვნელობის ჩვენება
      for(String k : hdrField){
System.out.println("გასაღები: " + k + "მნიშვნელობა: " + hdrMap.get(k));}}}
```

ნახ. 296

შედეგი:

```
მოთხოვნის მეთოდი: GET
პასუხის კოდი: 200
პასუხის შეტყობინება: OK

აქედან სათაური იწყება:
გასაღები: Transfer-Encodingმნიშვნელობა: [chunked]
გასაღები: nullმნიშვნელობა: [HTTP/1.1 200 OK]
გასაღები: Serverმნიშვნელობა: [gws]
გასაღები: P3Pმნიშვნელობა: [CP="This is not a P3P policy! See
http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657 for more info."]
გასაღები: Dateმნიშვნელობა: [Sun, 08 Nov 2015 17:29:20 GMT]
გასაღები: Accept-Rangesმნიშვნელობა: [none]
გასაღები: X-Frame-Optionsმნიშვნელობა: [SAMEORIGIN]
გასაღები: Cache-Controlმნიშვნელობა: [private, max-age=0]
გასაღები: Varyმნიშვნელობა: [Accept-Encoding]
გასაღები: Set-Cookieმნიშვნელობა: [NID=73=jsdKbvOu7wZdoWvHiGh-1h4x3DEaC6JliXImA5J49-
53PaTQZzNfEuOYxUAP6h-7RQrgJOxVGru5UcdD5pddFYcL2M4e2KzysITjVDK2jVMFN8T-
SQ5vOOOziBuD_voxa7JxDqOlwSf2CWkPFfAePtJl_FqKwIc; expires=Mon, 09-May-2016 17:29:20 GMT;
path=/; domain=.google.ge; HttpOnly,
PREF=ID=1111111111111111:FF=0:TM=1447003760:LM=1447003760:V=1:S=Kpzju114UfqgFVGw;
expires=Thu, 31-Dec-2015 16:02:17 GMT; path=/; domain=.google.ge]
გასაღები: Expiresმნიშვნელობა: [-1]
გასაღები: X-XSS-Protectionმნიშვნელობა: [1; mode=block]
გასაღები: Content-Typeმნიშვნელობა: [text/html; charset=UTF-8]
```

ნახ. 297

6.1.7. TCP/IP სერვერული სოკეტები

როგორც უკვე აღვნიშნეთ, Java ენაში სოკეტების არაერთი კლასია, რომლებსაც სერვერული პროგრამების შესაქმნელად მივმართავთ. ServerSocket კლასი იმ სერვერების შესაქმნელად გამოიყენება, რომლებიც როგორც ლოკალური, ისე არალოკალური კლიენტების პროგრამების მოსმენას ახდენენ. ისინი, თავის მხრივ, კავშირის დამყარებას ღია პორტებით ცდილობენ. ServerSocket კლასი ჩვეულებრივი Socket კლასებისგან მნიშვნელოვნად განსხვავდება. როდესაც თქვენ ServerSocket კლასის ობიექტს ქმნით, ის სისტემაში თავს არეგისტრირებს, როგორც კლიენტებთან დაკავშირებით დაინტერესებული ობიექტი. აღნიშნული კლასის კონსტრუქტორები წარმოგვიდგენენ პორტის ნომერს, რომლითაც კავშირის მიღება გვსურს და ასევე, (არასავალდებულო) რიგის სიგრძეს მოცემული პორტისთვის. რიგის სიგრძე სისტემას ატყობინებს, თუ რამდენი კლიენტური შეერთება შეიძლება შეკავდეს, სანამ ახალი შეერთებების მცდელობის უარყოფას დაიწყებდეს ის. სტანდარტულად, მიუთითებლობის შემთხვევაში, მისი მნიშვნელობა 50-ის ტოლია. განსაზღვრულ კონკრეტულ პირობებში კონსტრუქტორებს IOException გამონაკლისის გადაცემა შეუძლიათ. ServerSocket კლასის კონსტრუქტორები 49-ე ცხრილშია წარმოდგენილი.

ცხრილი 49 . ServerSocket კლასის კონსტრუქტორები

კონსტრუქტორი	აღწერა
ServerSocket (int პორტი) throws IOException	მითითებულ პორტზე ქმნის სერვერულ სოკეტს 50-ის ტოლი რიგის სიგრძით.
ServerSocket (int პორტი, int მაქსიმალური რიგი) throws IOException	მითითებულ პორტზე ქმნის სერვერულ სოკეტს, რომლის რიგის მაქსიმალური სიგრძე მითითებულია პარამეტრში „მაქსიმალური სიგრძე“.
ServerSocket (int პორტი, int მაქსიმალური რიგი, InetAddress ლოკალური მისამართი) throws IOException	მითითებულ პორტზე ქმნის სერვერულ სოკეტს, რომლის რიგის მაქსიმალური სიგრძე მითითებულია პარამეტრში „მაქსიმალური სიგრძე“. ჯგუფურ ჰოსტზე „ლოკალური მისამართი“ მიუთითებს იმ IP-მისამართზე, რომელზეც სოკეტია მიერთებული.

6.1.8. დეიტაგრამები

ქსელური ურთიერთობა TCP/IP სტილში ძალზე ხშირად გამოიყენება, რადგან ის მონაცემთა პაკეტების სერიალიზებულ და საიმედო ნაკადებს უზრუნველყოფს, თუმცა ეს მარტივი არ არის. TCP პროტოკოლი ისედაც გადატვირთულ ქსელებში ნაკედების მართვის არაერთ რთულ ალგორითმს მოიცავს და არც პაკეტების დაკარგვა არის გამორიცხული. ეს კი, გარკვეულ წილად, მონაცემთა ტრანსპორტირების არაეფექტურ საშუალებას წარმოადგენს. ალტერანტივის სახით, ამ შემთხვევაში, დეიტაგრამები შეგვიძლია გამოვიყენოთ.

დეიტაგრამები (datagramms) ინფორმაციის პორციაა, რომელიც კომპიუტერებს შორის გადაიცემა. როგორც კი დეიტაგარამა საჭირო მიმართულებით გაიშვება, არ არსებობს გარანტია იმისა, რომ ის მიზნამდე მიაღწევს ან ვინმე აღმოჩნდება ადგილზე და მას დაიჭერს. ანალოგიურად, დეიტაგრამის მიღების დროს, ასევე არ არსებობს გარანტია იმისა, რომ ის გზაში არ დაზიანდება ან რომ მისი გამგზავნი ჯერ კიდევ ელის პასუხს.

დეიტაგრამის რეალიზებისთვის Java ორ კლასს იყენებს: DatagramPacket (მონაცემთა კონტეინერი) და DatagramSocket (მექანიზმი, რომელიც DatagramPacket კლასის მომსახურებისთვის გამოიყენება). დაწვრილებით განვიხილოთ ისინი.

DatagramSocket კლასი ოთხ კონსტრუქტორს განსაზღვრავს:

DatagramSocket() throws SocketException

DatagramSocket(int პორტი) throws SocketException

DatagramSocket(int პორტი, InetAddress ip-მისამართი) throws SocketException

DatagramSocket(SocketAddress მისამართი) throws SocketException

პირველი კონსტრუქტორი DatagramSocket კლასის ობიექტს ქმნის, რომელიც ლოკალური კომპიუტერის ნებისმიერ დაუკავებელ პორტთან არის დაკავშირებული. მეორე - DatagramSocket კლასის ობიექტს, რომელიც „პორტი“ პარამეტრით მითითებულ პორტს უკავშირდება. მესამე ქმნის DatagramSocket კლასის ობიექტს, რომელიც მითითებული პორტით და InetAddress კლასის ობიექტით არის დაკავშირებული. მეოთხე კონსტრუქტორი DatagramSocket კლასის ობიექტს ქმნის, რომელიც მოცემული SocketAddress მისამართით არის დაკავშირებული.

SocketAddress კლასი აბსტრაქტული კლასია, რომელიც კონკრეტული InetSocketAddress კლასით არის რეალიზებული. ეს უკანასკნელი კი IP-მისამართის ინკაფსულაციას ახდენს პორტის ნომრით. სოკეტის შექმნისას შეცდომის წარმოქმნის შემთხვევაში ყველა კონსტრუქტორს SocketException გამონაკლისის გადაცემა შეუძლია.

DatagramSocket კლასი მრავალ მეთოდს განსაზღვრავს. მათგან ორი ყველაზე მნიშვნელოვანი მეთოდია: send() და receive() მეთოდები, რომლებიც ქვემოთ არის წარმოდგენილი:

void send(DatagramPacket პაკეტი) throws IOException

void receive(DatagramPacket პაკეტი) throws IOException

send() მეთოდი პორტს „პაკეტი“ პარამეტრში მითითებულ პაკეტს უგზავნის. receive() მეთოდი პაკეტის მიღებას იმ პორტიდან ელოდება, რომელიც „პაკეტი“ პარამეტრშია მითითებული და აბრუნებს შედეგს.

DatagramSocket კლასი close() მეთოდსაც განსაზღვრავს, რომელიც სოკეტს კეტავს. აღნიშნული კლასი AutoCloseable ინტერფეისის რეალიზებას ახდენს და შესაბამისად, რესურსებიანი try ბლოკის მართვაც შეუძლია. DatagramSocket კლასის დანარჩენი მეთოდები 50-ე ცხრილშია წარმოდგენილი.

ცხრილი 50. DatagramSocket კლასის მეთოდები

მეთოდი	აღწერა
InetAddress getInetAddress()	თუ სოკეტი მიერთებულია, აბრუნებს მისამართს. წინააღმდეგ შემთხვევაში - null მნიშვნელობას.
int getLocalPort()	აბრუნებს ლოკალური პორტის ნომერს.
int getPort()	აბრუნებს პორტის ნომერს, რომელზე სოკეტია მიერთებული. თუ სოკეტი არცერთ პორტთან არ არის დაკავშირებული, აბრუნებს -1-ს.
boolean isBound()	თუ სოკეტი მისამართზე მიბმულია, აბრუნებს true მნიშვნელობას. წინააღმდეგ შემთხვევაში - false მნიშვნელობას.
boolean isConnected()	თუ სოკეტი სერვერთან ჩართულია აბრუნებს true მნიშვნელობას. წინააღმდეგ შემთხვევაში - false მნიშვნელობას.
void setSotimeout(int მილიწამი) throws SocketException	აყენებს მოლოდინის პერიოდს მილიწამებში, რომელიც „მილიწამი“ პარამეტრით გადაიცემა.

DatagramPacket კლასი მრავალ კონსტრუქტორს განსაზღვრავს. მათგან ორი კონსტრუქტორი ქვემოთ არის წარმოდგენილი:

DatagramPacket(byte მონაცემები [], int ზომა)

DatagramPacket(byte მონაცემები [], int ზომა, InetAddress ipAddress, int პორტი)

პირველი კონსტრუქტორი ბუფერს განსაზღვარავს, რომელმაც მონაცემები და პაკეტის ზომა უნდა მიიღოს. ის DatagramSocket კლასით მონაცემების მისაღებად გამოიყენება. მეორე კონსტრუქტორი მიზნობრივ მისამართს და პორტს განსაზღვრავს, რომელსაც DatagramSocket კლასი იმისთვის გამოიყენებს, რომ განსაზღვროს თუ სად გაიგზავნება პაკეტის მონაცემები.

DatagramPacket კლასი რამდენიმე მეთოდს განსაზღვრავს, რომლებიც წვდომის საშუალებას იძლევიან როგორც პაკეტის პორტის მისამართსა და ნომერზე, ასევე, საბაზო მონაცემებსა და მათ სიგრძეზე. ზოგად შემთხვევაში, get() მეთოდი მისაღებ პაკეტებში გამოიყენება, ხოლო set() მეთოდი - გასაგზავნ პაკატებში.

DatagramPacket კლასის მეთოდები 51-ე ცხრილშია წარმოდგენილი.

ცხრილი 51. DatagramPacket კლასის მეთოდები

მეთოდი	აღწერა
InetAddress getAddress()	აბრუნებს წყაროს მისამართს (მიმღები დეიტაგრამებისთვის) ან დანიშნულების ადგილს (გამგზავნი დეიტაგრამებისთვის).
byte[] getData()	აბრუნებს დეიტაგრამაში არსებულ მონაცემთა ბაიტების მასივს. (ძირითადად დეიტაგრამიდან მონაცემების ამოსაღებად გამოიყენება მისი მიღების შემდეგ).
int getLength()	აბრუნებს ბაიტების მასივში არსებული კორექტული მონაცემების სიგრძეს (აღნიშნული მასივი getData() მეთოდის მიერ ბრუნდება). ის შესაძლებელია ბაიტების მასივის სრულ სიგრძეს არ ემთხვეოდეს.
int getOffset()	აბრუნებს მონაცემთა საწყისს ინდექსს.
int getPort()	აბრუნებს პორტის ნომერს.
void setAddress(InetAddress ip-მისამართი)	აყენებს მისამართს, რომელზეც პაკეტი იგზავნება. მისამართი „ip-მისამართი“ პარამეტრით ეთითება.
void seData(byte[] მონაცემები, int ინდექსი, int ზომა)	აყანებს „მონაცემები“ პარამეტრის შესაბამის მონაცემებს და სიგრძეს „ზომა“ პარამეტრის შესაბამისად.
void setLength(int ზომა)	აყენებს პაკეტის სიგრძეს „ზომა“ პარამეტრით.
ovid setPort(int პორტი)	აყენებს პორტს „პორტი“ პარამეტრით.

მაგალითი 6. შევადგინოთ კლიენტ-სერვერის მარტივი ურთიერთობის ამსახველი პროგრამა, სადაც შეტყობინებები სერვერის ფანჯარაში შეიყვანება, ქსელით გადაიცემა კლიენტის მხარეს და იქვე გამოიძახება.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.net.*;
public class WriteServer {
public static int serverPort=998;
public static int clientPort=999;
public static int buffer_size=1024;
public static DatagramSocket ds;
public static byte buffer[]=new byte[buffer_size];
public static void TheServer() throws Exception{
       int pos=0;
       while(true){
              int c=System.in.read();
              switch(c){
              case -1:
                     System.out.println("სერვერმა დაასრულა მუშაობა");
                     ds.close();
                     return;
              case '\r':
                     break;
              case '\n':
                     ds.send(new DatagramPacket(buffer,pos,
InetAddress.getLocalHost(),clientPort));
                     pos=0;
                     break;
                     default:
                            buffer[pos++]=(byte)c;}}}
public static void TheClient() throws Exception{
       while(true){
              DatagramPacket p=new DatagramPacket(buffer, buffer.length);
              ds.receive(p);
              System.out.println(new String(p.getData(), 0, p.getLength()));}}
```

```
public static void main(String args[])throws Exception{
       if(args.length==1){
              ds=new DatagramSocket(serverPort);
              TheServer();}
              else{
                     ds=new DatagramSocket(clientPort);
                     TheClient();}}}
```

ნახ. 298

პროგრამის ეს ნიმუში DatagramSocket კლასის კონსტრუქტორით არის შეზღუდული. ამიტომ ლოკალური კომპიუტერის პორტებზე მის გასაშვებად ერთ ფანჯარაში გამოიყენეთ ბრძანება:

```
java WriteServer
```

ეს იქნება კლენტი. ხოლო მეორე ფანჯარაში გაუშვით შემდეგი ბრძანება:

```
java WriteServer  1
```

ეს იქნება სერვერი. რასაც სერვერის ფანჯარაში შეიყვანთ, კლიენტის ფანჯარაში გაიგზავნება.

მაგალითი 7. შევადგინოთ კლიენტისთვის სტრიქონის გადაცემისა და მის მიერ სტრიქონის მიღების ამსახველი პროგრამა.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.io.*;
import java.net.*;
public class MyServerSocket {
	public static void main(String[] args)throws IOException {
		Socket s=null;
	try { // კლიენტისთვის სტრიქონის გაგზავნა
		//ობიექტის შექმნა და პორტის ნომრის დანიშვნა
		ServerSocket server = new ServerSocket(8040);
		s = server.accept();//შეერთების მოლოდინი
		PrintStream ps =new PrintStream(s.getOutputStream());
		// "გამარჯობათ!" სტრიქონის ბუფერში მოთავსება
		ps.println("გამარჯობათ");
	// ბუფერის შიგთავსის კლიენტისთვის გაგზავნა და ბუფერის გასუფთავება
		ps.flush();
		ps.close();
```

```
} catch (IOException e) {
		System.err.println("შეცდომა: " + e);
		} finally {
		if (s != null)
		s.close(); // კავშირის გაწყვეტა
		}}}
```

ნახ. 299

```
package network;
import java.io.*;
import java.net.*;
public class MyClientSocket {
	public static void main(String[] args)throws IOException {
		Socket socket = null;
		try { // კლიენტის მიერ სტრიქონის მიღება
		socket = new Socket("სერვერის სახელი", 8030);
		/* აქ "სერვერის სახელი" კომპიუტერია,
		რომელზეც სერვერ-სოკეტი დგას*/
		BufferedReader br =
		new BufferedReader(
		new InputStreamReader(
		socket.getInputStream()));
		String msg = br.readLine();
		System.out.println(msg);
		socket.close();
		} catch (IOException e) {
		System.err.println("შეცდომა: " + e);}}}
```

ნახ. 300

კითხვები თვითშემოწმებისთვის:

1. *რას წარმოადგენს სოკეტი? TCP სოკეტების რა სახეებია თქვენთვის ცნობილი?*
2. *რა მახასიათებლებით განისაზღვრება სოკეტი?*
3. *რას წარმოადგენს პორტი?*
4. *განმარტეთ ინტერნეტ პროტოკოლის ცნება.*
5. *რა დანიშნულება აქვს გადაცემების მართვის პროტოკოლს?*
6. *დაახასიათეთ სამომხმარებლო დეიტაგრამების პროტოკოლი.*
7. *რომელი პროტოკოლები გამოიყენება TCP/IP პროტოკოლების სტეკში?*
8. *რას წარმოადგენს მისამართი ინტერნეტში?*
9. *რას წარმოადგენს რესურსების უნიფიცირებული ლოკატორი?*
10. *რას წარმოადგენს დეიტაგრამა? რისთვის გამოიყენება ის?*

დავალება:

1. შეადგინეთ პროგრამა, რომელიც კონსოლზე გამოიტანს თქვენი ლოკალური კომპიუტერისა და თქვენთვის სასურველი სამი ინტერნეტ-საიტის სახელებსა და მისამართებს.
2. შეადგინეთ პროგრამა, რომელიც ხსნის „whois" (პორტი 43) პორტთან კავშირს InterNIC სერვერზე, სოკეტს უგზავნის ბრძანების ხაზის არგუმენტებს და შემდგომ კონსოლზე გამოაქვს დაბრუნებული მონაცემები.
3. შეადგინეთ პროგრამა, რომელიც თქვენთვის სასურველი სტატიების გვერდის URL-ს ქმნის და ნახულობს მის თვისებებს.
4. შეადგინეთ პროგრამა, რომელიც URLConnection კლასის ობიექტის შესაქმენლად URL კლასის ობიექტის openConnection() მეთოდს იყენებს. შემდეგ კი ადგილი აქვს თქვენთვის საინტერესო დოკუმენტის შიგთავსისა და თვისებების შემოწმებას.
5. შეადგინეთ პროგრამა, რომელიც კავშირს ამყარებს www.gtu.ge საიტთან. წარმოადგინეთ მოთხოვნის მეთოდი, პასუხის კოდი, პასუხის შეტყობინება და პასუხის სათაურში გასაღებები და მნიშვნელობები.
6. შეადგინეთ კლიენტ-სერვერის მარტივი ურთიერთობის ამსახველი პროგრამა.
7. შეადგინეთ კლიენტისთვის სტრიქონის გადაცემისა და მის მიერ სტრიქონის მიღების ამსახველი პროგრამა.

აღნიშნულ თავთან დაკავშირებული ვიდეო-მასალა შეგიძლიათ იხილოთ ბმულზე:

https://www.youtube.com/watch?v=BsJiO6ergt0

6.2. აპლეტებზე მუშაობა

6.2.1. HTML APPLET დესკრიპტორი

აპლეტის ხელით გასაშვებად Oracle-ის კომპანია APPLET დესკრიპტორს გვთავაზობს მაშინ, როდესაც JNLP პროტოკლის გამოყენებას ადგილი არ აქვს. ვებ-ბრაუზერები უფლებას გვაძლევს რამდენიმე აპლეტი ერთ ფანჯარაში, ერთ გვერდზე შევასრულოთ. აქამდე ჩვენ APPLET დესკრიპტორის მხოლოდ გამარტივებულ ფორმას ვიყენებდით. ახლა დადგა მისი ყურადღებით განხილვის დრო.

APPLET დესკრიპტორის სრული ფორმის სინტაქსი ქვემოთ არის წარმოდგენილი, სადაც კვადრატულ ფრჩხილებში მითითებული ელემენტები სავალდებულო არ არის.

```
<APPLET
[CODEBASE=URL_ კოდის ბაზები]
CODE=აპლეტის ფაილი
[ALT=ალტერნატიული ტექსტი]
[NAME=აპლეტის ეგზემპლიარის სახელი]
WIDTH=პიქსელები, HEIGHT=პიქსელები
[ALIGN=გასწორება]
[VSPACE=პიქსელები] [HSPACE=პიქსელები]
>
[<PARAM NAME=ატრიბუტის სახელი VALUE=ატრიბუტის მნიშვნელობა>]
[<PARAM NAME=ატრიბუტის სახელი2 VALUE=ატრიბუტის მნიშვნელობა>]
...
[HTML კოდი, რომელიც Java-ს არარსებობის შემთხვევაში გამოისახება]
</APPLET>
```

ახლა ყოველი მათგანი ცალ-ცალკე განვიხილოთ.

- **CODEBASE** არასავალდებულო ატრიბუტია, რომელიც აპლეტის კოდის საბაზო URL-ს იძლევა. ამ შემთხვევაში აპლეტი კატალოგია, სადაც ადგილი ექნება კლასის შესრულებადი ფაილის ძებნას (მითითებულს დესკრიპტორში CODE). თუ ეს ატრიბუტი ცხადი სახით არ არის წარმოდგენილი, მაშინ CODEBASE ატრიბუტის როლში HTML დოკუმენტის URL კატალოგი გამოიყენება. სავალდებულო არ არის, რომ CODEBASE ატრიბუტი იმავე ჰოსტზე იყოს, საიდანაც HTML დოკუმენტია წაკითხული.
- **CODE** სავალდებულო ატრიბუტია. ის იმ ფაილის სახელს იძლევა, რომელიც თქვენი აპლეტის .class კომპილირებულ ფაილს შეიცავს. აღნიშნული ფაილის სახელი ან აპლეტის საბაზო URL

კოდიდან გადაიცემა, რომელიც კატალოგს წარმოადგენს (მასში HTML ფაილია მოთავსებული), ან კატალოგიდან, რომელიც CODEBASE ატრიბუტშია მითითებული.

- **ALT** არასავალდებულო ატრიბუტია, რომელიც მოკლე ტექსტური შეტყობინების მისათითებლად გამოიყენება იმ შემთხვევაში, თუ ბრაუზერი APPLET დესკრიპტორს ამოიცნობს, მაგრამ მოცემულ მომენტში მას Java-ს აპლეტის შესრულება არ შეუძლია. ეს განსხვავდება - ალტერნატიული HTML კოდისგან, რომელსაც ბრაუზერებში ვიყენებთ.
- **NAME** არასავალდებულო ატრიბუტია, რომელიც აპლეტის ეგზემპლიარის სახელის მისათითებლად გამოიყენება. აპლეტებზე სახელების დარქმევა ისე უნდა მოხდეს, რომ იმავე გვერდზე არსებულმა აპლეტებმა შეძლონ მათი მოძებნა სახელების მიხედვით და მათთან ურთიერთობა. აპლეტის სახელის მიხედვით მისაღებად getApplet() მეთოდი გამოიყენება, რომელიც AppletContext ინტერფეისშია განსაზღვრული.
- **WIDTH და HEIGHT** სავალდებულო ატრიბუტებია, რომლებიც აპლეტის წარმოდგენის არეს ზომას გვაძლევენ პიქსელებში.
- **ALIGN** არასავალდებულო ატრიბუტია, რომელიც აპლეტის ადგილმდებარეობის მიხედვით გასწორების პარამეტრს იძლევა. აღნიშნულ ატრიბუტს შემდეგი მნიშვნელობები გააჩნია: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE და ABSBOTTOM.
- **VSPACE** და **HSPACE** არასავალდებულო ატრიბუტებია. VSPACE ატრიბუტი პიქსელებში განსაზღვრავს აპლეტის ზემოთ და ქვემოთ არსებულ სივრცეს, ხოლო HSPACE ატრიბუტი - სივრცეს პიქსელებში აპლეტის გვერდებზე.
- **PARAM NAME** და **VALUE** არასავალდებულო ატრიბუტებია. PARAM დესტრუქტორი საშუალებას იძლევა მივუთითოთ აპლეტის სპეციფიური არგუმენტები. აღნიშნულ ატრიბუტებზე წვდომას აპლეტები getParameter() მეთოდის საშუალებით იღებენ.

გარდა ზემოთ წარმოდგენილი ატრიბუტებისა, გამოიყენება ARCHIVE ატრიბუტი, რომელიც საშუალებას გვაძლევს გადავცეთ ერთი ან რამდენიმე დაარქივებული ფაილი და OBJECT ატრიბუტი, რომელიც აპლეტის შენახული ვერსიის მითითებას ახდენს. ზოგად შემთხვევაში, APPLET დესკრიპტორმა უნდა ჩართოს ან მხოლოდ CODE ატრიბუტი ან მხოლოდ OBJECT ატრიბუტი, მაგრამ არა ორივე ერთად.

6.2.2. აპლეტებისთვის პარამეტრების გადაცემა

HTML APPLET დესტრუქტორი საშუალებას გვაძლევს აპლეტს პარამეტრები გადავცეთ. პარამეტრების მისაღებად getParameter() მეთოდი გამოიყენება. ის მითითებული პარამეტრის მნიშვნელობას სტრიქონული ფორმით აბრუნებს. ამგვარად, რიცხვითი და ბულის ტიპის მნიშვნელობებისთვის მოგვიწევს მათი სტრიქონულ ფორმატში გადაყვანა.

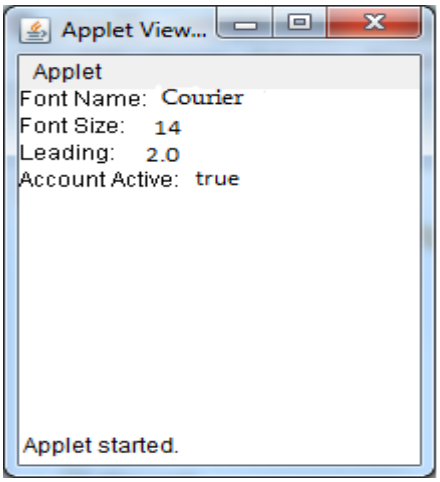
მაგალითი 8. შევადგინოთ პარამეტრების აპლეტში გადაცემის ამსახველი პროგრამა.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.awt.*;
import java.applet.*;
/*
 <applet code="ParamDemo" width=300 height=80>
 <param name=fontName value=Courier>
  <param name=fontSize value=14>
  < <param name=leading value=2>
   <param name=accountEnabled value=true>
   </applet>
 */
public  class ParamDemo extends Applet {
       String fontName;
       int fontSize;
       float leading;
       boolean active;
       //წარმოსადგენი სტრიქონის ინიციალება
       public void start(){
              String param;
              fontName=getParameter("fontName");
              if(fontName==null)
                     fontName="Not Found";
              //fontName= "ვერ მოიძებნა";
              param=getParameter("fontSize");
try{
                     if(param != null)
                            fontSize=Integer.parseInt(param);
                     else
                            fontSize=0;
              }catch(NumberFormatException e){
                     fontSize=-1;}
              param=getParameter("leading");
              try{
                     if(param!=null)
                            leading=Float.valueOf(param).floatValue();
                     else
                            leading=0;
              }catch(NumberFormatException e){
                     leading=-1;}
              param=getParameter("accountEnabled");
              if(param!=null)
                     active=Boolean.valueOf(param).booleanValue();}
                     //პარამეტრების გამოტანა
              public void paint(Graphics g){
                     g.drawString("Font Name: " + fontName, 0, 10);
                     g.drawString("Font Size: " + fontSize, 0, 26);
                  g.drawString("Leading: " + leading, 0, 42);
                  g.drawString("Account Active: " + active, 0, 58);}}
```

ნახ. 301

შედეგი:



ნახ. 302

როგორც ზემოთ წარმოდგენილი კოდიდან ჩანს, მნიშვნელოვანია getParameter() მეთოდიდან დასაბრუნებელი მნიშვნელობების შემოწმება. თუ პარამეტრი მიუწვდომელია, მეთოდი null მნიშვნელობას აბრუნებს. ასევე, აუცილებელია მონაცემთა რიცხვით ტიპებში გარდასახვა try ოპერატორში, რომელიც NumberFormatException ტიპის გამონაკლისს იჭერს.

6.2.3. getDocumentBase() და getCodeBase() მეთოდები

ხშირია შემთხვევები, როდესაც ისეთი აპლეტის შექმნა გვიწევს, რომელმაც მედია-ინფორმაცია და ტექსტი უნდა ჩატვირთოს.

Java აპლეტებს უფლებას აძლევს მონაცემები არა მხოლოდ იმ კატალოგიდან ჩატვირთოს, რომელიც HTML ფაილს შეიცავს და გაუშვას აპლეტი (დოკუმენტის ბაზა) შესრულებაზე, არამედ იმ კატალოგიდანაც, საიდანაც აპლეტის კლასი (კოდის ბაზა) იტვირთება. ეს კატალოგები getDocumentBase() და getCodeBase() მეთოდებიდან ბრუნდებიან, როგორც URL კლასის ობიექტები. ისინი შეიძლება დაკავშირებული იყოს სტრიქონთან - ფაილის სახელთან, რომლის ჩატვირტვაც გსურთ. სხვა ფაილის რეალურად ჩასატვირთად showDocument() მეთოდი გამოიყენება, რომელიც AppletContext ინტერფეისშია განსაზღვრული.

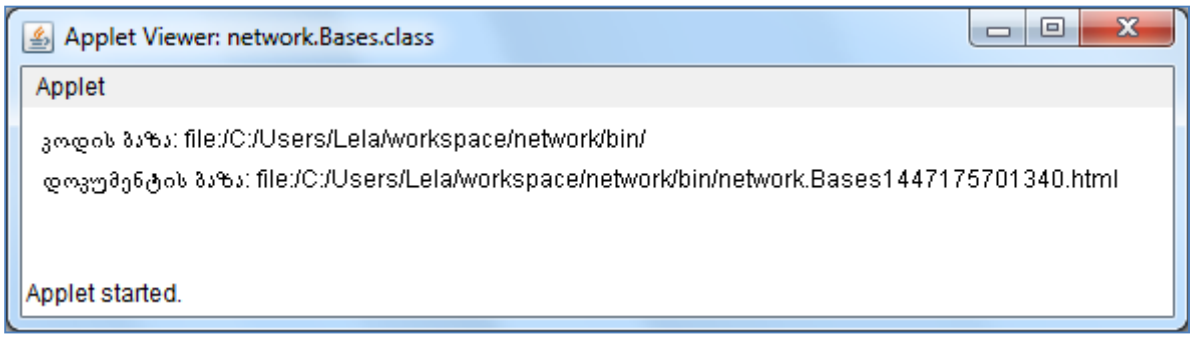
მაგალითი 9. შევადგინოთ პროგრამა, რომელიც getDocumentBase() და getCodeBase() მეთოდების გამოყენებით დოკუმენტისა და კოდის ბაზებს წარმოგვიდგენს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
 <applet code="Bases" width=300, height=50>
 </applet>
 */
public class Bases extends Applet {
	public void paint(Graphics g){
		String msg;
		URL url=getCodeBase(); //კოდის ბაზის მიღება
		msg="კოდის ბაზა: " + url.toString();
		g.drawString(msg, 10, 20);
		url=getDocumentBase(); //დოკუმენტის ბაზის მიღება
		msg="დოკუმენტის ბაზა: " + url.toString();
		g.drawString(msg, 10, 40);}}
```

ნახ. 303

შედეგი:



ნახ. 304

6.2.4. AppletContext ინტერფეისი და showDocument() მეთოდი

Java-ს გამოყენების ერთ-ერთ მნიშვნელოვან ფაქტორს აქტიური გამოსახულებებისა და ანიმაციის წარმოდგენა შეადგენს, რაც თავის მხრივ, გრაფიკული საშუალებების ნავიგაციის შესაძლებლობას იძლევა ვებ-ში. იმისათვის, რომ თქვენმა აპლეტმა მართვა სხვა URL-ს გადასცეს, showDocument() მეთოდის გამოყენებაა საჭირო. ეს უკანასკნელი AppletContext ინტერფეისშია

განსაზღვრული. აღნიშნული ინტერფეისი კი საშუალებას გვაძლევს ინფორმაცია აპლეტის შესრულების გარემოდან მივიღოთ.

AppletContext ინტერფეისში განსაზღვრული მეთოდები მე-52 ცხრილშია წარმოდგენილი. მიმდინარე შესრულებადი აპლეტის კონტექსტი getAppletContext() მეთოდის გამოძახებით მიიღება, რომელიც Applet კლასშია განსაზღვრული.

ცხრილი 52. AppletContext ინტერფეისში განსაზღვრული მეთოდები

მეთოდი	აღწერა
Applet getApplet (String აპლეტის სახელი)	იმ შემთხვევაში აბრუნებს აპლეტს, რომელიც „აპლეტის სახელი“-თ არის გადაცემული მეთოდში, თუ ის მიმდინარე აპლეტის კონტექსტშია. წინააღმდეგ შემთხვევაში აბრუნებს null მნიშვნელობას.
Enumeration<Applet> getApplets()	აბრუნებს ყველა იმ აპლეტის შემცველ ჩამოთვლად სიას, რომლებიც მიმდინარე აპლეტის კონტექსტს შეადგენენ.
AudioClip getAudioClip(URL url)	აბრუნებს AudioClip ინტერფეისის ობიექტს, რომელიც url -ით მითითებული აუდიოკლიპის ინკაფსულირებას ახდენს.
Image getImage(URL url)	აბრუნებს Image ინტერფეისის ობიექტს, რომელიც url -ით მითითებული გრაფიკული გამოსახულების ინკაფსულირებას ახდენს.
InputStream getStream (String გასაღები)	აბრუნებს „გასაღებთან“ დაკავშირებულ ნაკადს. გასაღებების ნაკადთან მიმაგრება setStream() მეთოდის საშუალებით ხდება. თუ გასაღებთან დაკავშირებული ნაკადი არ არსებობს, აბრუნებს null მნიშვნელობას.
Iterator <String> getStreamKeys()	აბრუნებს იტერატორს გასაღებებისთვის, რომელიც გამომძახებელ ობიექტთან ასოცირდება.
void setStream(String გასაღები, InputStream ნაკადი) throws IOException	„ნაკადი“ პარამეტრით გადაცემულ ნაკადს აკავშირებს „გასაღები“ პარამეტრით გადაცემულ პარამეტრთან. გამომძახებელი ობიექტიდან გასაღები იშლება, თუ ნაკადი null მნიშვნელობას შეიცავს.
void showDocument(URL url)	წარმოგვიდგენს URL მისამართზე url პარამეტრით გადაცემულ დოკუმენტს.

void showDocument(URL url, String სად)	წარმოგვიდგენს URL მისამართზე url პარამეტრით გადაცემულ დოკუმენტს. დოკუმენტის ადგილმდებარეობა „სად“ პარამეტრით არის მითითებული.
void showStatus(String სტრიქონი)	სტატუსის ზოლში (სტრიქონში) წარმოადგენს პარამეტრ „სტრიქონი“-ს შიგთავსს.

showDocument() მეთოდი მნიშვნელობის დაბრუნებას არ ახდენს და არც გამონაკლისს გადასცემს საჭირო შემთხვევაში. ამიტომ, ამ მეთოდის გამოყენებისას სიფრთხილე გმართებთ. ხელმისაწვდომი ორი showDocument() მეთოდია. showDocument(URL) მეთოდი წარმოადგენს დოკუმენტს, რომელიც მითითებულ URL-ზეა, ხოლო showDocument(URL, String) - დოკუმენტს, რომელიც ბრაუზერის ფანჯარაში მითითებულ ადგილზეა. „სად“ პარამეტრის კორქტულ არგუმენტებს წარმოადგენს: “_self” (წარმოდგენა მიმდინარე ფრეიმში), “_parent” (წარმოდგენა მშობელ ფრეიმში), “_top” (წარმოდგენა უმაღლესი დონის ფრეიმში), “_blank” (წარმოდგენა ბრაუზერის ახალ ფანჯარაში). ასევე, შესაძლებელია სახელის მიცემაც, რაც საშუალებას მოგცემთ დოკუმენტი მისი სახელით წარმოადგინოთ ბრაუზერის ახალ ფანჯარაში.

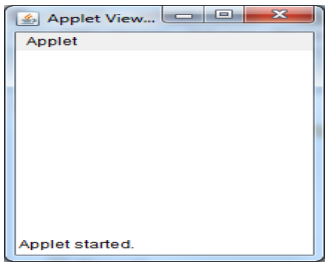
მაგალითი 10. შევადგინოთ პროგრამა, რომელიც AppletContext ინტერფეისისა და showDocument() მეთოდის გამოყენების დემონსტრირებას მოახდენს. შესრულებისას ის მიმდინარე აპლეტის კონტექსტს მიიღებს და მას Test.html ფაილისთვის მართვის გადასაცემად გამოიყენებს. ფაილი და აპლეტი ერთსა და იმავე კატალოგში უნდა იყოს მოთავსებული. Test.html ფაილი შესაძლოა ნებისმიერ დასაშვებ ჰიპერტექსტს შეიცავდეს.

პროგრამის კომპიუტერული რეალიზაცია:

```
package network;
import java.net.*;
import java.awt.*;
import java.applet.*;
/*
<applet code="ACDemo" width=300, height=50>
</applet>
*/
public class ACDemo extends Applet {
	public void start(){
		AppletContext ac=getAppletContext();
		URL url=getCodeBase(); //მიმდინარე აპლეტის url-ის მიღება
		try{
			ac.showDocument(new URL(url + "Tets.html"));
		}catch(MalformedURLException e){
			showStatus("URL ვერ მოიძებნა");}}}
```

ნახ. 305

შედეგი:



ნახ. 306

ორიოდე სიტყვით წარმოავდგინოთ AudioClip ინტერფეისიც. ის სამ მეთოდს განსაზღვრავს. ესენია: play() - კლიპის შესრულებაზე გაშვება, stop() - გაშვებული კლიპის შეჩერება და loop() - უწყვეტი ციკლური შესრულება. აუდიოკლიპის ჩატვირთვა getAudioClip() მეთოდით ხდება. შემდეგ კი კლიპის შესრულებაზე გაშვებას ზემოთ წარმოდგენილი მეთოდებიდან ერთ-ერთით ვახდენთ.

კითხვები თვითშემოწმებისთვის:

1. *რას წარმოადგენს HTML APPLET დესკრიპტორი?*
2. *APPLET დესკრიპტორის რომელი სავალდებულო ატრიბუტებია თქვენთვის ცნობილი?*
3. *რა დანიშნულება აქვს OBJECT ატრიბუტს?*
4. *რა მიზნით გამოიყენება getParameter() მეთოდი?*
5. *რა დანიშნულება აქვს getCodeBase() მეთოდს?*
6. *რისთვის გამოიყენება showDocument() მეთოდი?*
7. *რატომ არის სიფრთხილის გამოჩენა საჭირო showDocument() მეთოდის გამოყენების დროს?*

დავალება:

1. შექმენით აპლეტი, რომელიც თქვენთვის სასურველი დოკუმენტისა და კოდის ბაზებს წარმოგიდგენთ.
2. შეადგინეთ პროგრამა, რომელიც აპლეტს გადასცემს შემდეგ პარამეტრებს: შრიფტის სახელს, შრიფტის ზომას, შრიფტის სტილს და მოახდენს მითითებული პარამეტრების აპლეტის ფორმაზე გამოტანას.

აღნიშნულ თავთან დაკავშირებული ვიდეო-მასალა შეგიძლიათ იხილოთ ბმულზე:

https://www.youtube.com/watch?v=RUQT7X_5X7k

6.3. სერვლეტებთან მუშაობა

სერვლეტები (servlet) მცირე ზომის პროგრამებს ეწოდება, რომლებიც სერვერის მხარეს სრულდება ვებ-ში. აპლეტები დინამიურად აფართოებენ ვებ-ბრაუზერის ფუნქციონალურ შესაძლებლობებს, ხოლო სერვლეტები - ვებ-სერვერის შესაძლებლობებს. სერვლეტების თემა საკმაოდ ფართოა. ამიტომ, წინამდებარე სახელმძღვანელოში ჩვენ განვიხილავთ სერვლეტებთან დაკავშირებულ ძირითად კონცეფციებს, ინტერფეისებს, კლასებს და შესაბამისი მაგალითების ანალიზს მოვახდენთ.

6.3.1. სერვლეტები და მათი სიცოცხლის ციკლი

სერვლეტების უპირატესობაში რომ უკეთ გავერკვეთ, საჭიროა ზოგადი წარმოდგენა ვიქონიოთ იმის შესახებ, თუ შეთანხმებულად როგორ მუშაობენ ვებ-ბრაუზერები და სერვლეტები, რათა მომხმარებელს საჭირო ინფორმაცია მიეწოდოს. განვიხილოთ სტატიკური ვებ-გვერდის მოთხოვნა. მომხმარებელს ბრაუზერის ფანჯარაში URL-მისამართი (Uniform Resource Locator - ინფორმაციული რესურსის უნიფიცირებული მიმთითებელი) შეაქვს. ბრაუზერი HTTP მოთხოვნას ქმნის შესაბამისი ვებ-სერვერისთვის. ეს უკანასკნელი (ვებ-სერვერი) მოთხოვნასა და კონკრეტულ ფაილს შორის შესაბამისობას ადგენს. ეს ფაილი ბრაუზერს HTTP პასუხის სახით უბრუნდება. პასუხში HTTP სათაური ფაილის შიგთავსის ტიპს მიუთითებს. ამ მიზნით MIME (Multipurpose Internet Mail Exstensions - ელექტრონული ფოსტის მრავალამოცანიანი გაფართოება) სტანდარტების ნაკრები გამოიყენება. მაგალითად, ASCII ფორმატში წარმოდგენილ ჩვეულებრივ ტექსტს გააჩნია ტიპი MIME text/plain; საწყის HTML (Hypertext Markup Language - ჰიპერტექსტის აღნიშვნის ენა) კოდს კი ტიპი - MIME text/html.

ახლა დინამიური სიტუაცია განვიხილოთ. წარმოვიდგინოთ, მაღაზია, რომელიც ინტერაქტიურ რეჟიმში მუშაობს და საკუთარი ბიზნეს-საქმიანობის ინფორმაციის შეანხვის მიზნით მონაცემთა ბაზას იყენებს. ეს ბაზა შესაძლოა შეიცავდეს შემდეგ ელემენტებს: გაყიდვების რეგისტრაციას, საქონლის ნაირსახეობას, ანგარიშებს და ა.შ. მაღაზიის ხელმძღვანელობამ გადაწყვიტა, ეს ინფორმაცია მომხმარებლისთვის ხელმისაწვდომი გახადოს ვებ-გვერდის საშუალებით. ამ ვებ-გვერდების შიგთავსი დინამიურად უნდა იქმნებოდეს, რათა მონაცემთა ბაზაში ბოლო ინფორმაცია (სიახლეები) აისახოს.

არსებობის ადრეულ ეტაპებზე ვებ-სერვერს გვერდის დინამურად ფორმირება შეეძლო. ამ თვალსაზრისით, ის კლიენტის ყოველი მოთხოვნის დასამუშავებლად ცალკეულ პროცესს ქმნიდა. საჭირო ინფორმაციის მისაღებად, პროცესს შეეძლო ერთ ან რამდენიმე მონაცემთა ბაზასთან დაკავშირება. სერვერთან კავშირი CGI (Common Gateway Interface - საერთო „შლუზის“ ინტერფეისი) ინტერფეისით ხორციელდებოდა. ეს ინტერფეისი ცალკეულ პროცესს უფლებას

აძლევდა HTTP მოთხოვნიდან აეღო მონაცემები და ისინი HTTP პასუხში ჩაეწერა. CGI პროგრამების დასაწერად სხვადასხვა დაპროგრამების ენები გამოიყენებოდა. ამ ენებში შედიოდა: C, C++ და Perl. მაგრამ CGI ინტერფეისს სერიოზული პრობლემები ახლდა თან, რაც მის წარმადობას უკავშირდებოდა. ეს ინტერფეისი საკმაოდ ძვირი ჯდებოდა კომპიუტერის პროცესორისა და მეხსიერების გამოყენების თვალსაზრისით, რადგან კლიენტის ყოველი მოთხოვნისთვის ცალკეული პროცესის შექმნა იყო საჭირო. ამას გარდა, CGI პროგრამების მუშაობა კონკრეტულ პლატფორმაზე იყო დამოკიდებული. სწორედ ამიტომ იქნა სხვა ტექნოლოგიები შემოთავაზებული, რომელთა რიცხვს სერვლეტებიც მიეკუთვნება.

CGI ინტერფეისთან შედარებით სერვლეტები გარკვეული უპირატესობებით ხასიათდება. უპირველეს ყოვლისა, სერვლეტების წარმადობა გაცილებით მაღალია. სერვლეტები ვებ-სერვერის შიდა სამისამართო სივრცეში სრულდება. კლიენტის ყოველი მოთხოვნის დასამუშავებლად ცალკეული პროცესის შექმნა სავალდებულო არ არის. მეორეს მხრივ, სერვლეტების მუშაობა პლატფორმისგან დამოუკიდებელია, რადგან ყველა მათგანი Java-ენაზე იწერება. და, მესამე - Java-ს უსაფრთხოების დისპეტჩერი შეზღუდვების მთელი რიგი სერიის რეალიზებას ახდენს კომპიუტერ-სერვერზე რესურსების დაცვის მიზნით. სერვლეტებისთვის ხელმისაწვდომია Java-ს კლასების ბიბლიოთეკების ყველა ფუნქციონალური შესაძლებლობა. სერვლეტს შეუძლია აპლეტებთან, მონაცემთა ბაზებთან და სხვა პროგრამულ უზრუნველყოფასთან მუშაობა სოკეტებისა და RMI მექანიზმების საშუალებით.

სერვლეტის სიცოცხლის ციკლს სამი მეთოდი განსაზღვრავს. ესენია: init(), service() და destroy(). ყოველი სერვლეტი მათ რეალიზებას ახდენს და სერვერის მიერ განსაზღვრულ დროში ხდება მათი გამოძახება. განვიხილოთ, მარტივი სამომხმარებლო სცენარი, რომელიც საშუალებას მოგვცემს, გავიგოთ, თუ როგორ ხდება ამ მეთოდების გამოძახება.

დავუშვათ, მომხმარებელმა ბრაუზერის ფანჯარაში შეიყვანა URL-მისამართი. აღნიშნული მისამართის საფუძველზე, ბრაუზერი HTTP მოთხოვნას ქმნის, რომელიც შესაბამის სერვერს ეგზავნება.

HTTP მოთხოვნას ვებ-სერვერი იღებს. სერვერი მოთხოვნასა და კონკრეტულ სერვლეტს შორის შესაბამისობას ნახულობს. სერვლეტი სერვერის სამისამართო სივრცეში დინამიურად იტვირთება.

სერვერი იძახებს სერვლეტის init() მეთოდს. ეს მეთოდი მხოლოდ მაშინ გამოიძახება, როდესაც სერვლეტი კომპიუტერის მეხსიერებაში პირველად იტვირთება. სერვლეტს შეიძლება ინიციალიზაციის პარამეტრები გადავცეთ, ამიტომ მას საკუთარი თავის კონფიგურირება დამოუკიდებლად შეუძლია.

სერვერი იძახებს სერვლეტის service() მეთოდს. ეს მეთოდი HTTP მოთხოვნის დასამუშავებლად გამოიძახება. სერვლეტს შეუძლია HTTP მოთხოვნაში არსებული მონაცემების წაკითხვა. მას ასევე, შეუძლია HTTP პასუხის კლიენტისთვის ფორმულირება.

სერვლეტი სამისამართო სივრცეში რჩება და წვდომადია კლიენტებისგან მიღებული სხვა ნებისმიერი HTTP მოთხოვნის დასამუშვებლად. service() მეთოდი ყოველი HTTP მოთხოვნისთვის გამოიძახება.

და ბოლოს, სერვერმა შესაძლოა მიიღოს სერვლეტის მეხსიერებიდან წაშლის გადაწყვეტილება. ამ მიზნით ყოველი სერვერი სხვადასხვა ალგორითმს იყენებს. რესურსების გასათავისუფლებლად, ისეთის, როგორიცაა: სერვლეტისთვის გამოყოფილი ფაილების ინდექსები, სერვერი destroy() მეთოდს იძახებს. საჭირო მონაცემები მუდმივ მატარებლებზე (დისკებზე) შეიძლება იქნეს შენახული. სერვლეტისა და მისი ობიექტებისთვის გამოყოფილი მეხსიერება კი „ნაგვის“ შეგროვების პროცესში შესაძლოა გასუფთავდეს.

6.3.2. სერვლეტების შემუშავების შესაძლებლობები

სერვლეტების შესაქმნელად თქვენ კონტეინერთან ან სერვლეტების სერვერთან წვდომა დაგჭირდებათ. მათ შორის ყველაზე პოპულარულია სერვლეტების სერვერი Glassfish და სერვლეტების კონტეინერი Tomcat. Oracle ორგანიზაციის მიერ Glassfish სერვერი SDK Java EE კომპლექტთან ერთად მოიცემა. Tomcat კონტეინერი კი რეალიზაციის გახსნილი საწყისი კოდის მქონე პროდუქტია, რომელსაც Apache Software Foundation ორგანიზაცია გვთავაზობს. ის NetBeans ინტეგრირებულ გარემოშიც შეიძლება გამოყენებულ იქნეს. როგორც Glassfish სერვერი, ასევე Tomcat კონტეინერი სხვა ინტეგრირებულ გარემოშიც ჰპოვებს გამოყენებას, მაგალითად, Eclipse-ში. მიმდინარე სახელმძღვანელოში ჩვენ Tomcat კონტეინერს გამოვიყენებთ. საშუალებები, რომლითაც სერვლეტები მუშავდება კონკრეტულ ინტეგრირებულ გარემოზეა დამოკიდებული. Tomcat კონტეინერს ჩვენ იმიტომ მივმართეთ, რომ ამ შემთხვევაში ბრძანების ხაზის მხოლოდ ინსტრუმენტული შესაძლებლობების გამოყენება დაგვჭირდება და არ იქმნება იმის საჭიროება, რომ სერვლეტებთან სამუშაოდ სპეციალური ინტეგრირებული გარემო ჩავტვირთოთ და დავაინსტალიროთ.

Tomcat კონტეინერის სისტემის შემადგენლობაში შედის კლასების ბიბლიოთეკები, დოკუმენტაცია და შესრულების გარემო სერვლეტების შესაქმნელად და შესამოწმებლად. Tomcat კონტეინერის სისტემის ჩასატვირთად თქვენ შემდეგი მისამართი შეგიძლიათ გამოიყენოთ: **tomcat.apache.org**. აღნიშნული კონტეინერის ვერსიებს, როგორც 32, ისე 64-თანრიგიანი Windows ოპერაციული სისტემის მხარდაჭერა აქვთ. თქვენ თქვენი სისტემის შესაბამისი ვერსია უნდა

აირჩიოთ. ჩვენ შემთხვევაში გამოყენებულია Tomcat 7.0.4 კონტეინერის 64-ბიტიანი ვერსია და ქვემოთ წარმოდგენილი მისი ადგილმდებარეობის ამსახველი სრული გზა:

C:\apache-tomcat-7.0.4-windows-64x\ apache-tomcat-7.0.4

თუ თქვენ Tomcat კონტეინერს სხვა ადგილას ჩატვირთავთ (ან კონტეინერის სხვა ვერსიას გამოიყენებთ), მაშინ პროგრამებში გარკვეული ცვლილებების შეტანა მოგიწევთ.

Tomcat კონტეინერის დაყენების შემდეგ, მისი გაშვება **startup.bat** ფაილით ხდება, რომელიც **bin** კატალოგშია განთავსებული. ეს უკანასკნელი **apache-tomcat-7.0.4** კატალოგში მდებარეობს.

Tomcat კონტეინერის შესაჩერებლად **shutdown.bat** ფაილის გაშვება მოგიწევთ, რომელიც ასევე **bin** კატალოგშია მოთავსებული.

სერვლეტების შესაქმნელად საჭირო კლასები და ინტერფეისები **servlet-api.jar** ფაილშია თავმოყრილი, რომელიც ქვემოთ წარმოდგენილ კატალოგში მდებარეობს:

C:\apache-tomcat-7.0.4-windows-64x\ apache-tomcat-7.0.4\lib

servlet-api.jar ფაილი რომ ხელმისაწვდომი გახდეს, CLASSPATH გზა ქვემოთ წარმოდგენილი ცვლილების ხარჯზე განაახლეთ:

C:\Programm Files\ Apache Software Foundation\Tomcat 5.5\common\lib\ servlet-api.jar

როგორც კი სერვლეტის კომპილაციას დაასრულებთ, Tomcat კონტეინერმა ის უნდა მოძებნოს. ამისთვის აუცილებელია მისი მოთავსება **Tomcat** კატალოგის **webapp** ქვეკატალოგში და მისი სახელის **web.xml** ფაილში შეტანა. ქვემოთ წარმოდგენილია თქვენ მიერ შესასრულებელი პროცედურა.

თავდაპირველად სერვლეტის კლასის ფაილის კოპირება შემდეგ კატალოგში მოახდინეთ:

C:\apache-tomcat-7.0.4-windows-64x\ apache-tomcat-7.0.4\webapp\examples\WEB-INF\classes

შემდეგ დაამატეთ სერვლეტის სახელი და ასახეთ ის **web.xml** ფაილში. ბოლოს ჩართეთ შემდეგ კატალოგში:

C:\apache-tomcat-7.0.4-windows-64x\ apache-tomcat-7.0.4\webapp\examples\WEB-INF

მაგალითად, თუ დავუშვებთ, რომ პირველ პროგრამას HelloServlet ერქმევა, სერვლეტის აღმწერ ნაწილში თქვენ მოგიწევთ შემდეგი სტრიქონების დამატება:

```
<servlet>
 <servlet-name>HelloServlet</servlet-name>
 <servlet-class>HelloServlet</servlet-class>
</servlet>
```

ქვემოთ წარმოგიდგენთ იმ სტრიქონებს, რომლის ჩამატებაც სერვლეტების ცვლილებების ნაწილში დაგჭირდებათ:

```
<servlet-mapping>
```

```
<servlet-name>HelloServlet</servlet-name>
<url-pattern>/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

აღნიშნული მოქმედებების შესრულება ყველა სერვლეტის შემთხვევაში დაგჭირდებათ.

6.3.3. მარტივი სერვლეტის შექმნა და შემოწმება

იმისათვის, რომ შექმნათ და შეამოწმოთ მარტივი სერვლეტი, აუცილებელია ქვემოთ წარმოდგენილი ძირითადი მოქმედებების შესრულება:

1. შექმენით სერვლეტის საწყისი კოდი და მოახდინეთ მისი კომპილაცია. შემდეგ სერვლეტის კლასების ფაილი გადააკოპირეთ შესაბამის კატალოგში, ხოლო web.xml ფაილში ჩაამატეთ სერვლეტის სახელი და ზემოთ წარმოდგენილი ცვლილებები.
2. შესრულებაზე გაუშვით Tomcat კონტეინერი.
3. შესრულებაზე გაუშვით ვებ-ბრაუზერი და მოითხოვეთ სერვლეტი.

ზემოთ წარმოდგენილი ყველა მოქმედება ახლა დაწვრილებით განვიხილოთ.

თავდაპირველად შექმენით შემდეგი პროგრამული კოდის შემცველი HelloServlet.java ფაილი.

```
package myServlets;
import java.io.*;
import javax.servlet.*;
public class HelloServlet extends GenericServlet {
      public void service(ServletRequest request, ServletResponse
response ){
            throws ServletException, IOException{
                  response.setContentType("text/html");
                  PrintWriter pw=response.getWriter();
                  pw.println("<B>Hello World!");
                  pw.close();}}}
```

ნახ. 307

განვიხილოთ ზემოთ წარმოდგენილი პროგრამა. თავდაპირველად ყურადღება მიაქციეთ იმ ფაქტს, რომ ის javax.servlet პაკეტის იმპორტირებას ახდენს. ეს პაკეტი სერვლეტების შესაქმნელად საჭირო კლასებსა და ინტერფეისებს შეიცავს. პროგრამა ასევე, განსაზღვრავს HelloServlet კლასს, როგორც GenericServlet კლასის ქვეკლასს. ეს უკანასკნელი იმ ფუნქციონალურ შესაძლებლობებს ფლობს, რომლებიც სერვლეტის შექმნას აიოლებს. ის გთავაზობთ init() და destroy() მეთოდების ვერსიებს, რომლებიც ცვლილებების გარეშე შეგიძლიათ გამოიყენოთ. თქვენ მხოლოდ service() მეთოდზე უნდა „იზრუნოთ“. აღნიშნული მეთოდის ხელახალი განსაზღვრა HelloServlet კლასში ხდება. ეს მეთოდი კლიენტის მოთხოვნებს ამუშავებს. მასში პირველ არგუმენტს ServletRequest ინტერფეისის ობიექტი წარმოადგენს, რომლის წყალობით სერვლეტი კლიენტის მოთხოვნაში წარმოდგენილ მონაცემებს კითხულობს. მეორე არგუმენტს ServletResponse ინტერფეისის

ობიექტი წარმოადგენს. მისი საშუალებით სერვლეტს კლიენტისთვის პასუხის ფორმირება შეუძლია. setContentType() მეთოდის გამოძახებისას MIME ტიპი განისაზღვრება HTTP პასუხისთვის. ამ პროგრამაში MIME ტიპს text/html წარმოადგენს. ის მიუთითებს, რომ ბრაუზერმა ის HTML-ის საწყის კოდად უნდა განიხილოს.

getWriter() მეთოდი PrintWriter კლასის ობიექტს იღებს. ყველაფერი, რაც ნაკადს გადაეცემა, კლიენტს მიეწოდება როგორც, HTTP პასუხის ნაწილი. println() მეთოდი HTML-ის მარტივი საწყისი კოდის ჩასაწერად გამოიყენება, როგორც HTTP პასუხი.

მოახდინეთ ამ საწყისი კოდის კომპილაცია. HelloServlet.class ფაილი მოათავსეთ Tomcat კონტეინერის შესაბამის კატალოგში და HelloServlet კლასი დაამატეთ web.xml ფაილში (როგორც ზემოთ აღვწერეთ).

Tomcat კონტეინერი გაუშვით შესრულებაზე. მისი ფუნქციონირება სერვლეტის შესრულებას წინ უნდა უსწრებდეს.

ჩატვირთეთ ბრაუზერი და შეიყვანეთ შემდეგი URL მისამართი:

http://localhost:8080/examples/servlets/servlet/HelloServlet

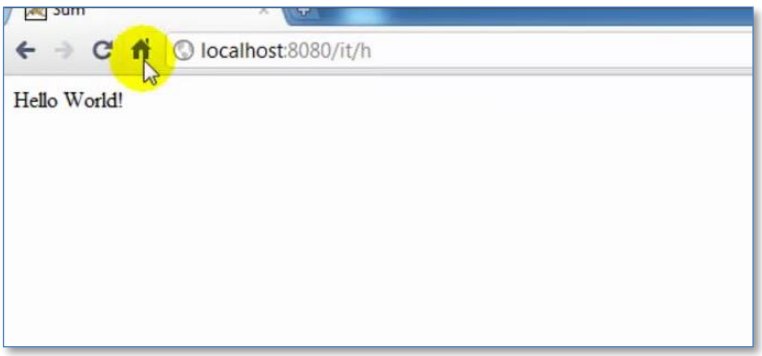
URL მისამართის მითითების ალტერნატიული გზაც არსებობს:

http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet

ეს URL მისამართი დასაშვებია, რადგან IP-მისამართი 127.0.01 განსაზღვრულია, როგორც ლოკალური კომპიუტერის მისამართი.

ბრაუზერის ფანჯარაში თქვენ სერვლეტის გამომავალ მონაცემებს დაინახავთ. მასში მოთავსებული იქნება სტრიქონი: Hello World!

შედეგი:



ნახ. 308

წარმოდგენილი პროგრამის სრული ვიდეო-მასალა შეგიძლიათ იხილოთ ბმულზე:

https://www.youtube.com/watch?v=Y04z4M5hfYw

6.3.4. ინტერფეისები Servlet Api, Servlet და პაკეტი javax.servlet

კლასები და ინტერფეისები, რომლებიც სერვლეტებთან სამუშაოდ გამოიყენება, javax.servlet და javax.servlet.http პაკეტებშია განთავსებული. ისინი Servlet API ინტერფეისს ქმნიან. ეს პაკეტების Java-ს სტანდარტულ პაკეტებს არ განეკუთვნება და შესაბამისად, არ არიან ჩართული Java SE კომპლექტში. მათ Tomcat კონტეინერი და Java EE კომპლექტი შეიცავენ.

Servlet API ინტერფეისი დამუშვებისა და სრულყოფის სტადიაშია. სერვლეტის მიმდინარე სპეციფიკაციას 3.0 ვერსია წარმოადგენს და სწორედ მას ვიყენებთ წინამდებარე სახელმძღანელოში.

javax.servlet პაკეტი მრავალ ინტერფეისსა და კლასს შეიცავს, რომლებიც იმ ინფრასტრუქტურას აყალიბებენ, რომლის ჩარჩოებშიც ფუნქციონირებენ სერვლეტები.

53-ე ცხრილში javax.servlet პაკეტის მნიშვნელოვანი ინტერფეისებია წარმოდგენილი. მათ შორის ყველაზე მთავარი Servlet ინტერფეისია. ყველა სერვლეტმა აღნიშნული ინტერფეისის რეალიზება უნდა მოახდინოს და გააფართოვოს მისი რეალიზებადი კლასი. ასევე, მნიშვნელოვან ინტერფეისებს ServletRequest და ServletResponce წარმოადგენს.

ცხრილი 53. javax.servlet პაკეტის მნიშვნელოვანი ინტერფეისები

ინტერფეისი	აღწერა
Servlet	სერვლეტისთვის აცხადების მეთოდების სიცოცხლის ციკლს.
ServletConfig	სერვლეტებს უფლებას აძლევს ინიციალიზაციის პარამეტრები მიიღონ.
ServletContext	სერვლეტებს უფლებას აძლევს დაარეგისტრირონ მოვლენები და მიმართონ მათი გარემოს შესახებ ინფორმაციას.
ServletRequest	გამოიყენება კლიენტის მოთხოვნიდან მონაცემების წასაკითხად.
ServletResponse	გამოიყენება კლიენტის პასუხში მონაცემების ჩასაწერად.

54-ე ცხრილში javax.servlet პაკეტის მნიშვნელოვანი კლასებია წარმოდგენილი.

ცხრილი 54. javax.servlet პაკეტის მნიშვნელოვანი კლასები

კლასი	აღწერა
GenericServlet	ახდენს Servlet და ServletConfig ინტერფეისების რეალიზებას.
ServletInputStream	იძლევა შეტანის ნაკადს კლიენტის მოთხოვნების წასაკითხად.

ServletOutputStream	იძლევა გამოტანის ნაკადს კლიენტისთვის პასუხების ჩასაწერად.
ServletException	მიუთითებს სერვლეტის შეცდომაზე.
UnavailableException	მიუთითებს, რომ სერვლეტი მიუწვდომელია.

Servlet ინტერფეისში გამოცხადებულია init(), service() და destroy() მეთოდები, რომლებიც სერვერის მიერ მისი სიცოცხლის ციკლის პერიოდში გამოიძახება. აღნიშნული მეთოდების გარდა Servlet ინტერფეისი სხვა მეთოდებსაც მოიცავს, რომლებიც 55-ე ცხრილშია წარმოდგენილი.

ცხრილი 55. Servlet ინტერფეისის მიერ განსაზღვრული მეთოდები

მეთოდი	აღწერა
void destroy()	გამოიძახება სერვლეტის ჩატვირთვის დროს.
ServletConfig getServletConfig()	აბრუნებს ServletConfig ინტერფეისის ობიექტს, რომელიც ინიციალიზაციის ნებისმიერ პარამეტრს შეიცავს.
String getServletInfo()	აბრუნებს სერვლეტის აღმწერ სტრიქონს.
void init(ServletConfig sc) **throws ServletException**	გამოიძახება სერვლეტის ინიციალიზაციის დროს. ინიციალიზაციის პარამეტრებს სერვლეტი „sc" პარამეტრიდან იღებს. თუ სერვლეტის ინიციალება შეუძლებელია, ServletException გამონაკლისი გადაიცემა.
void service(ServletRequest მოთხოვნა, **ServletResponse შედეგი) throws ServletException, IOException**	გამოიძახება კლიენტის მოთხოვნის დასამუშავებლად. კლიენტის მოთხოვნის წაკითხვა შესაძლებელია „მოთხოვნა" პარამეტრიდან. კლიენტისთვის პასუხი შეიძლება „შედეგი" პარამეტრში ჩაიწეროს. სერვლეტის ან შეტანა-გამოტანის შეცდომის შემთხვევაში გადაიცემა ServletException ან IOException გამონაკლისი.

მეთოდები init(), service() და destroy() სერვლეტის სიცოცხლის ციკლს განსაზღვრავენ. მათ სერვერი იძახებს. getServletConfig() მეთოდს ინიციალიზაციის პარამეტრების მისაღებად სერვლეტი იძახებს. getServletInfo() მეთოდი სერვლეტის დამამუშავებლის მიერ ხელახლა განისაზღვრება, რათა სასარგებლო (საჭირო) ინფორმაციის შემცველი სტრიქონი წარმოადგინოს

(მაგალითად, ავტორის გვარი, ვერსიის ნომერი, გამოშვების თარიღი, საავტორო უფლებები და სხვა). აღნიშნულ მეთოდს სერვერი იძახებს.

6.3.5. ინტერფეისები ServletConfig და ServletContext

ServletConfig ინტერფეისი სერვლეტს უფლებას აძლევს ჩატვირთვის დროს კონფიგურაციის პარამეტრები მიიღოს. აღნიშნული ინტერფეისის მეთოდები 56-ე ცხრილშია წარმოდგენილი.

ცხრილი 56. ServletConfig ინტერფეისის მიერ განსაზღვრული მეთოდები

მეთოდი	აღწერა
ServletContext getServletContext()	მოცემული სერვლეტისთვის აბრუნებს კონტექსტს.
String getInitParameter (String პარამეტრი)	აბრუნებს ინიციალიზაციის პარამეტრის მნიშვნელობას
Enumeration getInitParameterNames()	აბრუნებს ინიციალიზაციის პარამეტრების სახელების ჩამოთვლად სიას.
String getServletName()	აბრუნებს გამომძახებელი სერვლეტის სახელს.

ServletContext ინტერფეისი სერვლეტებს უფლებას აძლევს მათი გარემოს შესახებ მიიღონ ინფორმაცია. აღნიშნულ ინტერფეისში განსაზღვრული მეთოდები 57-ე ცხრილშია წარმოდგენილი.

ცხრილი 57. ServletContext ინტერფეისის მიერ განსაზღვრული მეთოდები

მეთოდი	აღწერა
Object getAttribute(String ატრიბუტი)	აბრუნებს სერვერის ატრიბუტის მნიშვნელობას, რომელიც „ატრიბუტი“ პარამეტრშია მითითებული.
String getMimeType(String ფაილი)	აბრუნებს MIME ფაილის ტიპს.
String getRealPath(String ვირტუალური გზა)	აბრუნებს ვირტუალური გზის შესაბამის რეალურ გზას.
String getServerInfo()	აბრუნებს ინფორმაციას სერვერის შესახებ.
void log(String s)	სერვერის ჟურნალში წერს სტრიქონს, მითითებულს შესაბამის პარამეტრში.

void log(String s, Throwable e)	სერვერის ჟურნალში წერს სტრიქონს და სტეკის ტრასირებას გამონაკლისისთვის.
void setAttribute(String ატრიბუტი, object მნიშვნელობა)	მოცემულ ატრიბუტს ანიჭებს მითითებულ მნიშვნელობას.

6.3.6. ინტერფეისები ServletRequest და ServletResponce

ServletRequest ინტერფეისი სერვლეტებს საშუალებას აძლევს მიიღონ ინფორმაცია კლიენტის მოთხოვნის შესახებ. 58-ე ცხრილში ამ ინტერფეისში განსაზღვრული მეთოდებია წარმოდგენილი.

ცხრილი 58 . ServletRequest ინტერფეისის მიერ განსაზღვრული მეთოდები

მეთოდი	აღწერა
Object getAttribute(String ატრიბუტი)	აბრუნებს მითითებული ატრიბუტის მნიშვნელობას.
String getCharacterEncoding()	აბრუნებს მოთხოვნაში არსებული სიმბოლოების კოდირების სქემას.
int getContentLenght()	აბრუნებს მოთხოვნის ზომას. თუ ზომის განსაზღვრა შეუძლებელია, აბრუნებს -1-ის ტოლ მნიშვნელობას.
int getContentType()	აბრუნებს მოთხოვნის ტიპს. თუ ტიპის განსაზღვრა შეუძლებელია, აბრუნებს null მნიშვნელობას.
ServletInputStream getInputStream() **throws IOException**	აბრუნებს ServletInputStream კლასის ობიექტს, რომელიც მოთხოვნაში ორობითი მონაცემების წასაკითხად შეიძლება იქნეს გამოყენებული. თუ ამ მოთხოვნისთვის getReader() მეთოდი უკვე გამოძახებული იყო, გადაიცემა IllegalStateException გამონაკლისი.
String getParameter(String პარამეტრის სახელი)	აბრუნებს მითითებული პარამეტრის მნიშვნელობას.

Enumeration<String>getParameterNames()	მოცემული მოთხოვნისთვის აბრუნებს პარამეტრების სახელების ჩამოთვლად სიას.
String [] getParameterValues(String სახელი)	აბრუნებს „სახელი“ პარამეტრთან დაკავშირებული მნიშვნელობების მასივს.
String getProtocol()	აბრუნებს პროტოკოლის აღწერას.
BufferedReader getReader() **throws IOException**	აბრუნებს ბუფერიზებულ „მკითხველს“, რომელიც მოთხოვნიდან ტექსტის წასაკითხად შეიძლება იქნეს გამოყენებული. თუ ამ მოთხოვნისთვის getInputStream() მეთოდი უკვე გამოძახებული იყო, გადაიცემა IllegalStateException გამონაკლისი.
String getRemoteAddr()	აბრუნებს კლიენტის IP-მისამართის სტრიქონულ ეკვივალენტს.
String getRemoteHost()	აბრუნებს კლიენტის ჰოსტის სახელის სტრიქონულ ეკვივალენტს.
String getScheme()	აბრუნებს URL მისამართის გადაცემის სქემას, რომელიც მოთხოვნისთვის გამოიყენება (მაგალითად, “http”, “ftp”).
String getServerName()	აბრუნებს სერვერის სახელს.
int getServerPort()	აბრუნებს პორტის ნომერს.

ServletResponce ინტერფეისი სერვლეტს საშუალებას აძლევს კლიენტისთვის პასუხის ფორმირება განახორციელოს. აღნიშნულ ინტერფეისში განსაზღვრული მეთოდები 59-ე ცხრილშია წარმოდგენილი.

ცხრილი 59.ServletResponce ინტერფეისის მიერ განსაზღვრული მეთოდები

მეთოდი	აღწერა
String getCharacterEncoding()	აბრუნებს პასუხში არსებული სიმბოლოების კოდირების სქემას.
ServletOutputStream getOutputStream() **throws IOException**	აბრუნებს ServletOutputStream კლასის ობიექტს, რომელიც პასუხში ორობითი მონაცემების ჩასაწერად შეიძლება იქნეს გამოყენებული. თუ ამ მოთხოვნისთვის

	getWriter() მეთოდი უკვე გამოძახებული იყო, გადაიცემა IllegalStateException გამონაკლისი.
PrintWriter getWriter() **throws IOException**	აბრუნებს PrintWriter კლასის ობიექტს, რომელიც პასუხში სიმბოლური მონაცემების ჩასაწერად შეიძლება იქნეს გამოყენებული. თუ ამ მოთხოვნისთვის getOutputStream() მეთოდი უკვე გამოძახებული იყო, გადაიცემა IllegalStateException გამონაკლისი.
void setContentLenght(int ზომა)	იძლევა კონტენტის ზომას პასუხითვის.
void setContentTpe(String ტიპი)	იძლევა კონტენტის ტიპს პასუხითვის.

6.3.7. კლასები: GenericServlet, ServletInputStream, ServletOutputStream და ServletException

GenericServlet კლასი სერვლეტის სასიცოცხლო ციკლის ძირითადი მეთოდების რეალიზაციისთვის გამოიყენება. ის ახდენს Servlet და ServletConfig ინტერფეისების რეალიზებას. ამას გარდა, ხელმისაწვდომია სერვლეტის ჟურნალში სტრიქონის დამატების მეთოდი, რომლის სიგნატურები ქვემოთ არის წარმოდგენილი:

void log (String s)

void log (String s, Throwable e)

აქ s ის სტრიქონია, რომელიც აუცილებლად უნდა ჩაემატოს სერვლეტის ჟურნალში, ხოლო e - გადასაცემი გამონაკლისია.

ServletInputStream კლასი InputStream კლასის შესაძლებლობებს აფართოებს. მის რეალიზებას სერვლეტის კონტეინერი ახდენს. ის შემავალ ნაკადს გვაწვდის, რომელსაც სერვლეტის დამამუშავებელი კლიენტის მოთხოვნიდან მონაცემების წასაკითხად იყენებს. ის სტანდარტულ კონსტრუქტორს განსაზღვრავს. ამას გარდა, არსებობს მეთოდი, რომელიც ნაკადიდან ბაიტების კითხვის მიზნით გამოიყენება. მის სიგნატურას ქვემოთ წარმოდგენილი სახე აქვს:

int readLine(byte [] ბუფერი, int წანაცვლება, int ზომა) throws IOException

აქ ბუფერი მასივია, რომელშიც ბაიტების გარკვეული რაოდენობა (ზომა) ინახება. მეთოდი წაკითხული ბაიტების ფაქტიურ რაოდენობას აბრუნებს, ხოლო ნაკადის დასრულების პირობის შესრულების შემთხვევაში, მის მიერ დაბრუნებული მნიშვნელობა -1-ის ტოლია.

ServletOutputStream კლასი OutputStream კლასის შესაძლებლობებს აფართოებს. მის რეალიზებას სერვლეტის კონტეინერი ახდენს. ის გამომავალ ნაკადს გვაწვდის, რომელსაც სერვლეტის დამამუშავებელი კლიენტის პასუხში მონაცემების ჩასაწერად იყენებს. ის ასევე, განსაზღვრავს სტანდარტულ კონსტრუქტორს და მეთოდებს: print() და println(), რომლებსაც მონაცემები ნაკადში გამოაქვთ.

javax.servlet პაკეტი ორ გამონაკლისს განსაზღვრავს. პირველი ServletException გამონაკლისია, რომელიც სერვლეტის შეცდომაზე მეტყველებს, ხოლო მეორე UnavailableException გამონაკლისი-კლასი, რომელსაც ServletException კლასი აფართოებს. აღნიშნული გამონაკლისი იმაზე მიუთითებს, რომ სერვლეტი მიუწვდომელია.

6.3.8. სერვლეტის პარამეტრების წაკითხვა

ServletRequest ინტერფეისი მოიცავს მეთოდებს, რომლებიც საშუალებას იძლევა კლიენტის მოთხოვნიდან წაკითხულ იქნეს პარამეტრების სახელები და მნიშვნელობები. ახლა ჩვენ შევუდგებით იმ სერვლეტის შექმნას, რომელიც ამ მეთოდების გამოყენების დემონსტრირებას ახდენს.

მაგალითი 11. შევქმნათ ორი ფაილი. პირველ ფაილში, რომლის სახელია PostParameters.html განსაზღვრულია ვებ-გვერდი, ხოლო მეორე ფაილში სახელით PostParametersServlet.java - თავად სერვლეტი. ქვემოთ წარმოდგენილია PostParameters.html ფაილის საწყისი კოდი. ის გასაზღვრავს ცხრილს, რომელიც ორი ჭდისა და ორი ტექსტური ველისგან შედგება. ერთ-ერთ ჭდეს თანამშრომელი (Employee) წარმოადგენს, ხოლო მეორეს თანამშრომლის ტელეფონის ნომერი (Phone). გვაქვს ასევე, თანხმობის ღილაკი. ყურადღება მიაქციეთ იმ ფაქტს, რომ ფორმის (<form>) დესკრიპტორის action პარამეტრი URL მისამართს განსაზღვრავს. ამ მისამართის იდენტიფიცირებას ახდენს სერვლეტი, რომელიც HTTP POST მოთხოვნას დაამუშავებს.

PostParameters.html ფაილის საწყისი კოდი:

```
<html>
<body>
<center>
<form name="Form1"
method="post"
action="http://localhost : 8080/examples/servlets/servlet/PostParametersServlet">
<table>
<tr>
<td><B>Employee</td>
<td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
<td><B>Phone</td>
<td><input type=textbox name="p" size="25" value=""></td>
</tr>
</table>
<input type=submit value="Submit">
</body>
</html>
```

ნახ. 309

ახლა წარმოვადგინოთ PostParametersServlet.java ფაილის საწყისი კოდი. service() მეთოდი კლიენტების მოთხოვნების დამუშავების მიზნით ხელახლა განისაზღვრება. getParameterNames() მეთოდი პარამეტრების სახელების ჩამონათვალს აბრუნებს. მათი დამუშავება ციკლში ხდება. შესაბამისად, კლიენტისთვის გამოიტანება პარამეტრის სახელი და მისი მნიშვნელობა. ამ უკანასკნელის (პარამეტრის მნიშვნელობის) მიღება getParameter() მეთოდით ხდება.

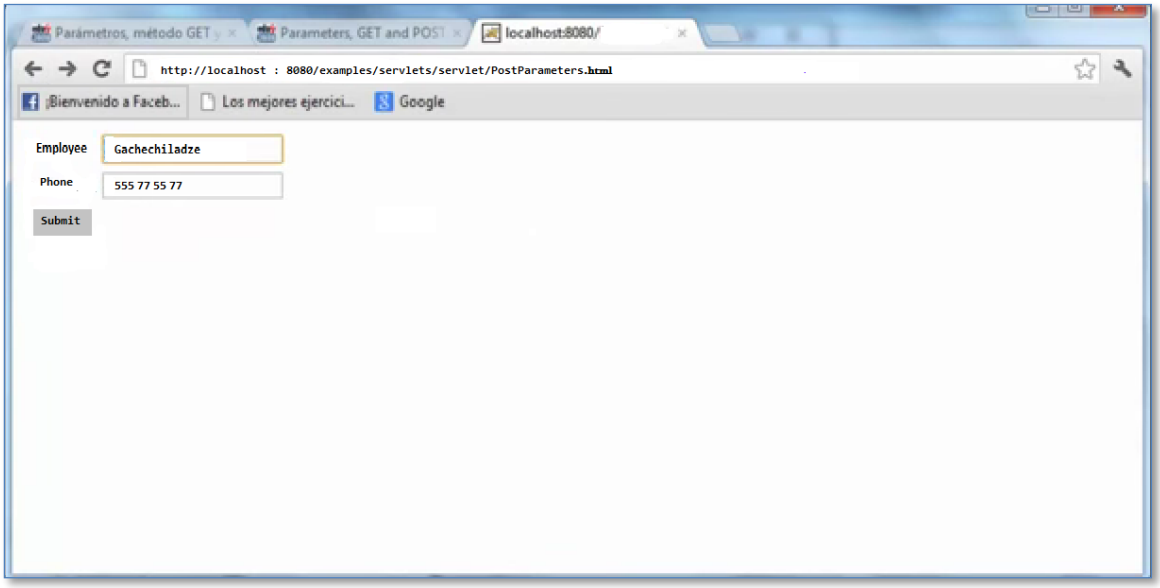
PostParametersServlet.java ფაილის საწყისი კოდი:

```
package net;
import java.io.*;
import java.util.*;
import javax.servlet.*;
public class PostParametersServlet extends GenericServlet {
	public void servaice(ServletRequest request, ServletResposce
response)
	throws ServletException, IOException{
		//ვიღებთ PrintWriter კლასს
		PrintWriter pw=response.getWriter();
		//ვიღებთ პარამეტრების სახელების ჩამონათვლას
		Enumeration e=request.getParameterNames();
```

```
//წარმოვადგენთ პარამეტრების სახელებს და მათ მნიშვნელობებს
            while(e.hasMoreElements()){
                  String pname=(String)(e.nextElement());
                  pw.print(pname + "=");
                  String pvalue=request.getParameter(pname);
                  pw.println(pvalue);}
                pw.close(); }}
```

ნახ. 309

შედეგი:



ნახ. 310

დავალება

მოახდინეთ სერვლეტის კომპილაცია. შემდეგ გადააკოპირეთ ის საჭირო კატალოგში და განაახლეთ **web.xml** *ფაილი (როგორც ზემოთ აღვნიშნეთ). პროგრამის შესამოწმებლად შეახრულეთ შემდეგი მოქმედებები:*

1. *შესრულებაზე გაუშვით Tomcat კონტეინერი.*
2. *ბრაუზერის ფანჯარაში წარმოადგინეთ ვებ-გვერდი*
3. *ტექსტურ ველებში შეიყვანეთ თანამშრომლის გვარი და ტელეფონის ნომერი.*
4. *გააგზავნეთ ვებ-გვერდი.*

ამის შემდეგ ბრაუზერის ფანჯარაში გამოიტანება პასუხი, რომელიც დინამიურად შექმნა სერვლეტმა.

(წარმოდგენილი პროგრამის მსგავსი ვიდეო-მასალა შეგიძლიათ იხილოთ გმულზე: https://www.youtube.com/watch?v=ugRMw8Wrg3w)

6.3.9. javax.servlet.http პაკეტი

HTTP პროტოკოლთან მუშაობის დროს ინტერფეისებსა და კლასებს თქვენ javax.servlet.http პაკეტიდან გამოიყენებთ. მისი ფუნქციონალური შესაძლებლობები სერვლეტების შექმნას გაცილებით ამარტივებს. ეს სერვლეტები კი HTTP მოთხოვნებთნ და პასუხებთან მუშაობს. მე-60 ცხრილში წამოდგენილია javax.servlet.http პაკეტის ძირითდი ინტერფეისები.

ცხრილი 60. javax.servlet.http პაკეტის ძირითდი ინტერფეისები

ინტერფეისი	აღწერა
HttpServletRequest	სერვლეტებს საშუალებას აძლევს HTTP მოთხოვნიდან წაიკითხოს მონაცემები.
HttpServletResponse	სერვლეტებს საშუალებას აძლევს HTTP პასუხში ჩაწეროს მონაცემები.
HttpSession	საშუალებას აძლევს წაიკითხოს და ჩაწეროს სეანსების მონაცემები.
HttpSessionBindingListener	ინფორმაციას აწვდის ობიექტს იმის შესახებ, არის თუ არ ის სეანსთან დაკავშირებული.

61-ე ცხრილში წამოდგენილია javax.servlet.http პაკეტის ძირითდი კლასები. მათ შორის ყველაზე მნიშვნელოვანს HttpServlet კლასი წარმოადგენს. როგორც წესი, სერვლეტებზე მუშაობის პროცესში პროგრამისტები აფართოებენ აღნიშნულ კლასს HTTP მოთხოვნების დამუშავების მიზნით.

ცხრილი 61. javax.servlet.http პაკეტის ძირითდი კლასები

კლასი	აღწერა
Cookie	საშუალებას იძლევა შეინახოს ინფორმაცია კლიენტის კომპიუტერის მდგომარეობის შესახებ.
HttpServlet	გვთავაზობს HTTP მოთხოვნებისა და დამუშავების მეთოდებს.
HttpSessionEvent	ახდენს სეანსის ცვლილების შესაბამისი მოვლენის ინკაფსულირებას.

HttpSessionBindingEvent	უჩვენებს მსმენელი დაკავშირებული არის თუ არა სეანსის მნიშვნელობასთან ან ახდენს იმის დემონსტრირებას, რომ სეანსის ატრიბუტი შეიცვალა.

6.3.10. HttpServletReqsuest და HttpServletResponse ინტერფეისები

HttpServletReqsuest ინტერფეისის რეალიზებას სერვლეტის კონტეინერი ახორციელებს. ის სერვლეტს უფლებას აძლევს მიიღოს ინფორმაცია კლიენტის მოთხოვნის შესახებ. 62-ე ცხრილში წარმოდგენილია აღნიშნული ინტერფეისის მნიშვნელოვანი მეთოდები.

ცხრილი 62. HttpServletReqsuest ინტერფეისის მნიშვნელოვანი მეთოდები

მეთოდი	აღწერა
String getAuthType()	აბრუნებს აუტენტიფიკაციის სქემას.
Cookie [] getCookies()	აბრუნებს მასივს, რომელიც მიმდინარე მოთხოვნაში არსებულ Cookie ფაილს შეიცავს.
long getDateHeader(String ველი)	აბრუნებს თარიღის სათაურის ველის მნიშვნელობას.
String getDateHeader (String ველი)	აბრუნებს სათაურის ველის მნიშვნელობას.
Enumeration<String> getHeaderNames()	აბრუნებს სათაურების სახელებისგან შემდგარ ჩამოთვლად სიას.
int getIntHeader(String ველი)	აბრუნებს სათაურის ველის მთელრიცხვა (int) ექვივალენტს.
String getMethod()	მოთხოვნისთვის აბრუნებს HTTP მეთოდს.
String getPathInfo()	აბრუნებს ნებისმიერ ინფორმაციას მარშრუტზე, რომელიც სერვლეტის მარშრუტის შემდეგ და URL მისამართში მოთხოვნის სტრიქონის წინაა განსაზღვრული.
String getPathTranslated()	აბრუნებს ნებისმიერ ინფორმაციას მარშრუტზე, რომელიც სერვლეტის მარშრუტის შემდეგ და URL მისამართში მოთხოვნის

	სტრიქონის წინაა განსაზღვრული მისი რეალურ მარშრუტზე გადაყვანის შემდეგ.
String getQueryString()	აბრუნებს URL მისამართზე ნებისმიერ მოთხოვნას.
String getRemoteUser()	აბრუნებს მიმდინარე მოთხოვნის მომხმარებლის სახელს.
String getRequestedSessionId()	აბრუნებს სეანსის იდენტიფიკატორს.
String getRequestURI()	აბრუნებს URL მისამართს.
StringBuffer getRequestURL()	აბრუნებს URL მისამართს.
String getServletPath()	აბრუნებს URL მისამართის ნაწილს, რომლის იდენტიფიცირებასაც ახდენს სერვლეტი.
HttpSession getSession()	აბრუნებს მიმდინარე მოთხოვნის სეანსს. თუ სეანსი არ არსებობს, ის ჯერ იქმნება და შემდეგ ხდება მისი დაბრუნება.
HttpSession getSession (boolean ახალი)	თუ „ახალი“ პარამეტრის მნიშვნელობაა treu და სეანსი არ არსებობს, მეთოდი ქმნის და აბრუნებს სეანსს მიმდინარე მოთხოვნისთვის. წინააღმდეგ შემთხვევაში, ის არსებულ სეანსს აბრუნებს.
boolean isRequested SessionIdFromCookie()	აბრუნებს treu მნიშვნელობას, თუ სეანსის იდენტიფიკატორს შეიცავს Cookie ფაილი. წინააღმდეგ შემთხვევაში აბრუნებს false მნიშვნელობას.
boolean isRequested SessionIdFromURL()	აბრუნებს treu მნიშვნელობას, თუ სეანსის იდენტიფიკატორს შეიცავს URL მისამართი. წინააღმდეგ შემთხვევაში აბრუნებს false მნიშვნელობას.

HttpServletResponse ინტერფეისი სერვლეტს უფლებას აძლევს კლიენტისთვის HTTP პასუხის ფორმულირება განახორციელოს. ის რამდენიმე კონსტანტას განსაზღვრავს. ისინი სხვადასხვა მდგომარეობების კოდებს შეესაბამებიან, რომლებიც HTTP პასუხს შეიძლება მივანიჭოთ. მაგალითად, მნიშვნელობა SC_OK გვიჩვენებს, რომ HTTP პასუხმა მიზნამდე მიაღწია; მნიშვნელობა SC_NOT_FOUND გვიჩვენებს, რომ მოთხოვნილი რესურსი მიუწვდომელია.

63-ე ცხრილში წარმოდგენილია აღნიშნული ინტერფეისის მნიშვნელოვანი მეთოდები.

ცხრილი 63. HttpServletResponse ინტერფეისის მნიშვნელოვანი მეთოდები

მეთოდი	აღწერა
void addCookie (Cookie cookie)	HTTP პასუხში ამატებს cookie-ს.
boolean conatainsHeader(String ველი)	აბრუნებს true მნიშვნელობას, თუ HTTP პასუხის სათაური მითითებულ ველს შეიცავს.
String encodeURL (String url)	განსაზღვრავს იყოს თუ არა სეანსის იდენტიფიკატორი კოდირებული URL მისამართში. დადებითი პასუხის შემთხვევაში აბრუნებს URL მისამართის სახეშეცვლილ ვერსიას. წინააღმდეგ შემთხვევაში აბრუნებს URL მისამართს. სერვერის მიერ შექმნილი ყველა URL მისამართი ამ მეთოდით უნდა დამუშავდეს.
String encodeRedirectURL (String url)	განსაზღვრავს იყოს თუ არა სეანსის იდენტიფიკატორი კოდირებული URL მისამართში. დადებითი პასუხის შემთხვევაში აბრუნებს URL მისამართის სახეშეცვლილ ვერსიას. წინააღმდეგ შემთხვევაში აბრუნებს URL მისამართს. ყველა URL მისამართი, რომელიც sendRedirect() მეთოდს გადაეცემა, ამ მეთოდით (encodeRedirectURL()) უნდა დამუშავდეს.
void sendError(int c) throws IOException	უგზავნის კლიენტს შეცდომის მოცემულ კოდს.
void sendError(int c, String s) throws IOException	უგზავნის კლიენტს შეცდომის მოცემულ კოდს და შეტყობინების სტრიქონს.
void sendRedirect(String url) throws IOException	ახდენს კლიენტის გადამისამართებას მითითებულ URL მისამართზე.
setDateHeader(String ველი, long მილიწამები)	სათაურში ამატებს ველს თარიღის მნიშვნელობით, რომელიც მილიწამებში იზომება. მილიწამების ათვლა 1970 წლის პირველი იანვრიდან წარმოებს.
void setHeader(String ველი, String მნიშვნელობა)	სათაურში ამატებს ველს მნიშვნელობით, რომელიც პარამეტრში „მნიშვნელობა“ არის მითითებული.

void setIntHeader(String ველი, int მნიშვნელობა)	სათაურში ამატებს ველს მნიშვნელობით, რომელიც პარამეტრში „მნიშვნელობა“ არის მითითებული.
void setStatus(int კოდი)	მოცემული პასუხის მდგომარეობას ანიჭებს კოდის მნიშვნელობას.

6.3.11. HttpSession და HttpSessionBindingListener ინტერფეისები

HttpSession ინტერფეისი სერვლეტს HTTP სეანსის მდგომარეობასთან დაკავშირებული ინფეორმაციის ჩაწერის და წაკითხვის საშუალებას აძლევს. აღნიშნული ინტერფეისის მეთოდები 64-ე ცხრილშია წარმოდგენილი. იმ შემთხვევაში, თუ სეანსი მოქმედი აღარ არის, ყველა მეთოდი IllegalStateException გამონაკლისს გადასცემს.

ცხრილი 64. HttpSession ინტერფეისის ძირითადი მეთოდები

მეთოდი	აღწერა
Object getAttribute(String ატრიბუტი)	აბრუნებს სახელთან დაკავშირებულ მნიშვნელობას (პარამეტრი „ატრიბუტი“). თუ მითითებული პარამეტრი ვერ მოიძებნა, აბრუნებს null მნიშვნელობას.
Enumeration<String> getAttributeNames()	აბრუნებს სეანსსთან დაკავშირებული სახელების ატრიბუტების ჩამოთვლად სიას.
long getCreationTime()	აბრუნებს სეანსის შექმნის მომენტიდან გასულ დროს. დროის ათვლა წარმოებს მილიწამებში 1970 წლის პირველი იანვრის შუაღამიდან.
String getId()	აბრუნებს სეანსის იდენტიფიკატორს.
long getAccessedTime()	აბრუნებს დროს იმ მომენტიდან, რომელიც კლიენტის ბოლო მოთხოვნიდან გავიდა მოცემულ სეანსსზე. დროის ათვლა წარმოებს მილიწამებში 1970 წლის პირველი იანვრის შუაღამიდან.
void invalidate()	აუქმებს ბოლო სეანსს.
boolean isNew()	აბრუნებს trues მნიშვნელობას, თუ სერვერმა შექმნა სეანსი, რომლისთვისაც კლიენტს ჯერ არ მიუმართავს.

void removeAttribute **(String ატრიბუტი)**	სეანსიდან შლის მითითებულ ატრიბუტს.
void setAttribute(String ატრიბუტი, Object მნიშვნელობა)	მოცემულ მნიშვნელობას აკავშირებს სახელის მოცემულ ატრიბუტთან.

HttpSessionBindingListener ინტერფეისის რეალიზება იმ ობიექტების მიერ ხდება, რომელთათვისაც აუცილებელია ინფორმაციის ფლობა იმის შესახებ, არიან თუ არა ისინი (ობიექტები) დაკავშირებული HTTP სეანსსთან. ქვემოთ წარმოდგენილია ორი მეთოდი, რომლებიც მაშინ გამოიძახება, როდესაც ობიექტი დაკავშირებულია (ან არ არის დაკავშირებული) სეანსსთან.

void valueBound(HttpSessionBindingEvent e)

void valueUnBound(HttpSessionBindingEvent e)

აქ e - მოვლენის შესაბამისი ობიექტია, რომელიც კავშირს აღწერს.

6.3.12. Cookie კლასი

Cookie კლასი cookie ფაილის ინკაფსულირებას ახდენს. cookie ფაილი - ეს მონაცემების შემცველი სტრიქონებია, რომლებიც კლიენტის კომპიუტერზე ინახება და შეიცავენ ინფორმაციას მდგომარეობის შესახებ. ეს ფაილები მომხმარებელთა აქტივობის თვალყურის სადევნად არის სასარგებლო. მაგალითად, დავუშვათ, მომხმარებელი სტუმრობს ინტერაქტიურ მაღაზიას. cookie ფაილს შეუძლია შეინახოს მომხმარებლის სახელი, მისამართი და სხვა ინფორმაცია. მაღაზიაში ყოველი სტუმრობისას კი მომხმარებელს აღარ დაჭირდება ამ მონაცემების ხელახლა შეყვანა.

მომხმარებლის კომპიუტერზე cookie ფაილის შენახვას სერვლეტი addCookie() მეთოდისა და HttpServletResponse ინტერფეისის საშუალებით ახდენს. მონაცემები ამ ფაილიდან შემდეგ HTTP პასუხის სათაურში თავსდება და ბრაუზერს ეგზავნება.

cookie ფაილის სახელები და მნიშვნელობები მომხმარებლის კომპიუტერზე ინახება. ყოველი cookie ფაილისთვის შენახული ინფორმაციის ნაწილი შემდეგ ცნობებს მოიცავს:

- cookie ფაილის სახელი.
- cookie ფაილის მნიშვნელობა.
- cookie ფაილის მოქმედების დროის შეწყვეტა.
- დომენი და გზა cookie ფაილისკენ.

cookie ფაილის მოქმედების დროის შეწყვეტა განსაზღვრავს, თუ როდის წაიშლება მომხმარებლის კომპიუტერიდან მოცემული cookie ფაილი. თუ ეს დრო ცხადი სახით დანიშნული არ

არის, cookie ფაილი ბრაუზერის მიმდინარე სეანსის დასრულებისთანავე წაიშლება. წინააღმდეგ შემთხვევაში ის მომხმარებლის კომპიუტერზე ფაილში ინახება.

დომენი და გზა cookie ფაილისკენ განსაზღვრავს, თუ როდის განთავსდება ის HTTP პასუხის სათაურში. თუ მომხმარებელს შეაქვს URL მისამართი, რომლის დომენი და გზა ამ მნიშვნელობებს ემთხვევა, მაშინ cookie ფაილი ვებ-სერვერს დაენიშნება, წინააღმდეგ შემთხვევაში - არა.

Cookie კლასს ერთადერთი კონსტრუქტორი გააჩნია, რომლის სიგნატურას შემდეგი სახე აქვს:

Cookie(String სახელი, String მნიშვნელობა)

აქ cookie ფაილის სახელი და მნიშვნელობა პარამეტრების სახით გადაეცემა კონსტრუქტორს.

Cookie კლასის მეთოდები 65-ე ცხრილშია წარმოდგენილი.

ცხრილი 65. Cookie კლასის მეთოდები

მეთოდი	აღწერა
Object clone()	აბრუნებს ობიექტის ასლს.
String getComment()	აბრუნებს კომენტარს.
String getDomain()	აბრუნებს დომენს.
int getMaxAge()	აბრუნებს მაქსიმალურ ასაკს (მილიწამებში)
String getName()	აბრუნებს სახელს.
String getPath()	აბრუნებს გზას.
booelan getSecure()	აბრუნებს true მნიშვნელობას უსაფრთხო cookie ფაილისთვის. წინააღმდეგ შემთხვევაში აბრუნებს false მნიშვნელობას.
String getValue()	აბრუნებს მნიშვნელობას.
int getVersion()	აბრუნებს ვერსიას.
booelan isHttponly()	აბრუნებს true მნიშვნელობას, თუ cookie ფაილი Httponly ატრიბუტს შეიცავს.
void setComment(String c)	ანიჭებს მითითებულ კომენტარს.
void setDomain(String d)	ანიჭებს მითითებულ დომენს.
void setComment(String void setHttponly(boolean მხოლოდ Http)	თუ „მხოლოდ Http“ პარამეტრი შეიცავს true მნიშვნელობას, მაშინ Httponly ატრიბუტი ემატება cookie ფაილს. თუ

	„მხოლოდ Http“ პარამეტრი შეიცავს false მნიშვნელობას, მაშინ Httponly ატრიბუტი იშლება.
void setMaxAge(int წამები)	აყენებს cookie ფაილის მაქსიმალურ ასაკს (წამებში). ეს მნიშვნელობა წარმოადგენს წამების რაოდენობას, რომლის გასვლის შემდეგ cookie ფაილი წაიშლება.
void setPath(String p)	ანიჭებს მითითებულ გზას.
void setSecure (Boolean დაცულია)	ანიჭებს უსაფრთხოების მითითებულ ალამს.
void setValue(String v)	ანიჭებს მითითებულ მნიშვნელობას.
void setVersion(int v)	ანიჭებს მითითებულ ვერსიას.

6.3.13. HttpServlet, HttpSessionEvent და HttpSessionBingingEvent კლასები

HttpServlet კლასი აფართოებს GenericServlet კლასის შესაძლებლობებს. ზოგადად, ეს კლასი იმ სერვლეტებთან სამუშაოდ გამოიყენება, რომლებიც HTTP მოთხოვნებს იღებენ და ამუშავებენ. აღნიშნული კლასის მეთოდები 66-ე ცხრილშია წარმოდგენილი.

ცხრილი 66. HttpServlet კლასის მეთოდები

მეთოდი	აღწერა
void doDelete (HttpServletRequest მოთხოვნა, HttpServletResponse შედეგი) throws IOException, ServletException	ამუშავებს HTTP მოთხოვნას DELETE.
void doGet (HttpServletRequest მოთხოვნა, HttpServletResponse შედეგი) throws IOException, ServletException	ამუშავებს HTTP მოთხოვნას GET.
void doOptions (HttpServletRequest მოთხოვნა, HttpServletResponse შედეგი) throws IOException, ServletException	ამუშავებს HTTP მოთხოვნას OPTIONS.

void doPost(HttpServletRequest მოთხოვნა, HttpServletResponse შედეგი) throws IOException, ServletException	ამუშავებს HTTP მოთხოვნას POST.
void doPut(HttpServletRequest მოთხოვნა, HttpServletResponse შედეგი) throws IOException, ServletException	ამუშავებს HTTP მოთხოვნას PUT.
void doTrace(HttpServletRequest მოთხოვნა, HttpServletResponse შედეგი) throws IOException, ServletException	ამუშავებს HTTP მოთხოვნას TRACE.
long getLastModified(HttpServletRequest მოთხოვნა)	აბრუნებს დროს მილიწამებში, როდესაც მოთხოვნილი რესურსი ბოლოჯერ იქნა შეცვლილი. დროის ათვლა წარმოებს 1970 წლის პირველი იანვრის შუაღამიდან.
void service(HttpServletRequest მოთხოვნა, HttpServletResponse შედეგი) throws IOException, ServletException	გამოიძახება სერვერის მიერ მაშინ, როდესაც მოცემული სერვლეტისთვის HTTP მოთხოვნა შემოდის. პარამეტრები ითვალისწინებს HTTP მოთხოვნასა და პასუხთან შესაბამისად წვდომას.

HttpSessionEvent კლასი სეანსების მოვლენების ინკაფსულირებას ახდენს. ის EvenObject კლასს აფართოებს და იქმნება სეანსებში წარმოებული ცვლილებების დროს. აღნიშნულ კლასში განსაზღვრულია შემდეგი კონსტრუქტორი:

HttpSessionEvent(HttpSessionEvent სეანსი)

აქ „სეანსი“ - მოვლენის მაუწყებელია. HttpSessionEvent კლასი მხოლოდ ერთ მეთოდს განსაზღვრავს, რომლის სიგნატურას შემდეგი სახე აქვს:

HttpSessionEvent getSession()

ის აბრუნებს სეანსს, რომელშიც მოვლენა მოხდა.

HttpSessionBingingEvent კლასი აფართოებს HttpSessionEvent კლასის შესაძლებლობებს. ის იმ შემთხვევაში იქმნება, როდესაც მსმენელი დაკავშირებულია (ან არ არის დაკავშირებული) HttpSession ინტერფეისის ობიექტის მნიშვნელობასთან. აღნიშნული კლასის კონსტრუქტორებს შემდეგი სახე აქვს:

HttpSessionBingingEvent(HttpSession სეანსი, String სახელი)

HttpSessionBingingEvent(HttpSession სეანსი, String სახელი, Object მნიშვნელობა)

აქ „სეანსი“ - მოვლენის მაუწყებელია, „სახელი“ - დაკავშირებული ან დაუკავშირებელი ობიექტის სახელია. მას მითითებული „მნიშვნელობა“ ნებისმიერ შემთხვევაში გადაეცემა. getName() მეთოდი დაკავშირებულ ან დაუკავშირებელ სახელს იღებს. მის კონსტრუქტორს შემდეგი სახე აქვს:

String getName()

getSession() მეთოდი კი იღებს სეანსს, რომელთანაც დაკავშირებულია (ან არ არის დაკავშირებული) მსმენელი. მის სიგნატურას შემდეგი სახე აქვს:

HttpSession getSession()

getValue() მეთოდი იღებს დაკავშირებული ან დაუკავშირებელი პარამეტრის მნიშვნელობას.

6.3.14. HTTP მოთხოვნებისა და პასუხების დამუშავება

HttpServlet კლასი სპეციალურ მეთოდებს გვთავაზობს, რომლებიც სხვადასხვა ტიპის HTTP მოთხოვნების დამუშავებას ახორციელებს. სერვლეტებზე მომუშავე პროგრამისტები, როგორც წესი, ამ მეთოდებიდან ერთ-ერთის ხელახლა განსაზღვრას ახდენენ. აღნიშნულ მეთოდებს მიკუთვნება: doDelete(), doGet(), doHead(), doOptions(), doPost(), doPut() და doTrace(). ყველაზე ხშირად ფორმების დამუშავებისას GET და POST მოთხოვნები გამოიყენება.

მაგალითი 12. განვიხილოთ სერვლეტი, რომელიც HTTP GET მოთხოვნას ამუშავებს. სერვლეტი მაშინ გამოიძახება, როდესაც ვებ-გვერდზე ფორმის შევსების შესახებ თანხმობა მიიღება. მაგალითი ორ ფაილს მოიცავს: ვებ-გვერდი განსაზღვრულია ColorGet.html ფაილში, ხოლო სერვლეტი - ColorGetServlet.java ფაილში. ColorGet.html ფაილი განსაზღვრავს ფორმას, რომელიც არჩევის ელემენტს და გაგზავნის ღილაკს შეიცავს. ყურადღება მიაქციეთ იმ ფაქტს, რომ <form> დესკრიპტორის action პარამეტრი URL მისამართს განსაზღვრავს. ეს მისამართი ახდენს სერვლეტის იდენტიფიცირებას HTTP GET მოთხოვნის დამუშავების მიზნით.

ColorGet.html ფაილის საწყისი კოდი:

```
<html>
<body>
<center>
<form name="Form1"
action="http://localhost : 8080/examples/servlets//servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
```

```
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

ნახ. 311

ColorGetServlet.java ფაილის საწყის კოდში doGet() მეთოდი ხელახლა განისაზღვრება ნებისმიერი HTTP GET მოთხოვნების დამუშავების მიზნით, რომლებიც მოცემულ სერვლეტს ეგზავნება. მომხმარებლის მიერ გაკეთებული არჩევანის მისაღებად ის getParamater() მეთოდს და HttpServletRequest ინტერფეისს იყენებს. ამის შემდეგ ადგილი აქვს პასუხის ფორმირებას.

ColorGetServlet.java ფაილის საწყისი კოდი:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ColorGetServlet extends HttpServlet {
public void doGet( HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException{
      String color=request.getParameter("color");
      response.setContentType("text/thml");
      PrintWriter pw=response.getWriter();
      pw.println("<B>The selected color is: ");
      pw.println(color);
      pw.close(); }}
```

ნახ. 312

მოახდინეთ სერვლეტის კომპილირება. შემდეგ გადააკოპირეთ ის შესაბამის კატალოგში და განაახლეთ web.xml ფაილი. მაგალითის შესამოწმებლად განახორციელეთ შემდეგი მოქმედებები:

1. *ჩატვირთეთ Tomcat კონტეინერი.*
2. *ბრაუზერის ფანჯარაში წარმოადგინეთ ვებ-გვერდი.*
3. *აირჩიეთ ფერი.*
4. *გააგზავნეთ ფორმა ვებ-გვერდზე.*

ამის შემდეგ ბრაუზერი წარმოგიდგენთ პასუხს, რომელიც დინამიურად შექმნა სერვლეტმა.

HTTP GET მოთხოვნის პარამეტრები ჩართულია როგორც URL მისამართის ნაწილი, რაც ვებ-სერვერს ეგზავნება. დავუშვათ, მომხმარებელმა აირჩია წითელი ფერი და ფორმა გააგზავნა. ბრაუზერიდან სერვერზე გადაგზავნილ URL მისამართს შემდეგი სახე ექნება:

http://localhost : 8080/examples/servlets//servlet/ColorGetServlet?color=Red

სიმბოლოებს, რომლებიც კითხვის ნიშნის მარჯვნივაა, **მოთხოვნის სტრიქონი** ეწოდება.

მაგალითი 13. განვიხილოთ სერვლეტი, რომელიც HTTP POST მოთხოვნას ამუშავებს. სერვლეტი მაშინ გამოიძახება, როდესაც ვებ-გვერდზე ფორმის შევსების შესახებ თანხმობა მიიღება. მაგალითი ორ ფაილს მოიცავს: ვებ-გვერდი განსაზღვრულია ColorPost.html ფაილში, ხოლო სერვლეტი - ColorPostServlet.java ფაილში. ColorPost.html ფაილი ColorGett.html ფაილის იდენტურია, იმ განსხვავებით, რომ <form> დესკრიპტორის method პარამეტრი ცხადად განსაზღვრავს, რომ საჭიროა POST მეთოდის გამოყენება. ამავე დესკრიპტორის action პარამეტრში სხვა სერვლეტია მითითებული.

ColorPost.html ფაილის საწყისი კოდი:

```
<html>
<body>
<center>
<form name="Form1"
method ="post"
action="http://localhost : 8080/examples/servlets//servlet/ColorPostServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type=submit value="Submit">
</form>
</body>
</html>
```

ნახ. 313

ColorPostServlet.java ფაილში doPost() მეთოდი ხელახლა განისაზღვრება ნებისმიერი HTTP POST მოთხოვნების დამუშავების მიზნით, რომლებიც მოცემულ სერვლეტს ეგზავნება. მომხმარებლის მიერ გაკეთებული არჩევანის მისაღებად ის getParamater() მეთოდს და HttpServletRequest ინტერფეისს იყენებს. ამის შემდეგ ადგილი აქვს პასუხის ფორმირებას.

ColorPostServlet.java ფაილის საწყისი კოდი:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ColorPostServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse
response)
throws ServletException, IOException{
       String color=request.getParameter("color");
       response.setContentType("text/thml");
       PrintWriter pw=response.getWriter();
       pw.println("<B>The selected color is: ");
       pw.println(color);
       pw.close(); }}
```

ნახ. 314

მოახდინეთ სერვლეტის კომპილირება. მის შესამოწმებლად იგივე მოქმედებები შეასრულეთ, რაც წინა (მე-12 მაგალითის) შემთხვევაში.

HTTP POST მოთხოვნის პარამეტრები URL მისამართში, რაც ვებ-სერვერს ეგზავნება, ჩართული არ არის. წარმოდგენილ მაგალითში ბრაუზერი სერვერს შემდეგი სახის მისამართს უგზავნის:

http://localhost : 8080/examples/servlets//servlet/ColorGetServlet

პარამეტრების სახელები და მნიშვნელობები აქ HTTP მოთხოვნის ტანში იგზავნება.

GET და POST მოთხოვნების დამუშავების მაგალითების შესაბამისი ვიდეო-მასალა იხილეთ ბმულზე: **https: //www.youtube.com/watch?v=IyC3ej2fqE0**

6.3.15. cookie ფაილების გამოყენება

მაგალითი 14. შევიმუშავოთ სერვლეტი, რომელიც საშუალებას მოგვცემს cookie ფაილების გამოყენების დემონსტრირება მოვახდინოთ. სერვლეტი ფორმის ვებ-გვერდზე გაგზავნის დროს გამოიძახება. ჩვენი მაგალითი სამ ფაილს მოიცავს:

AddCookie.html ფაილს, რომელიც მომხმარებელს უფლებას აძლევს cookie ფაილისთვის მნიშვნელობა განსაზღვროს MyCookie სახელით. AddCookiesServlet.java ფაილს, რომელიც AddCookie.html ფაილის გაგზავნას ამუშავებს და GetCookiesServlet.java ფაილს, რომელიც cookie ფაილის მნიშვნელობას წარმოგვიდგენს.

AddCookie.html ფაილის შესაბამისი გვერდი მნიშვნელობის შესატანად ტექსტურ ველს შეიცავს. ამავე გვერდზე მოთავსებულია გაგზავნის ღილაკი. ამ ღილაკზე დაჭერით, ტექსტურ ველში არსებული მნიშვნელობა AddCookiesServlet-ს გაეგზავნება HTTP POST მოთხოვნის საშუალებით.

AddCookie.html ფაილის საწყისი კოდი:

```
<html>
<body>
<center>
<form name="Form1"
method ="post"
action="http://localhost : 8080/examples/servlets//servlet/AddCookieServlet">
<B>Enter a value for MyCookie</B>

<input type=textbox name="data" size=25 value=" ">
<input type=submit value="Submit">
</form>
</body>
</html>
```

ნახ. 315

AddCookiesServlet.java ფაილი პარამეტრის მნიშვნელობას data სახელით იღებს. შემდეგ ის Cookie კლასის MyCookie ობიექტს ქმნის, რომელიც data პარამეტრის მნიშვნელობას შეიცავს. ამის შემდეგ, HTTP პასუხის სათაურს cookie ფაილი addCookie() მეთოდის საშუალებით ემატება. ბოლოს ბრაუზერი საპასუხო შეტყობინებას იღებს.

AddCookiesServlet.java ფაილის საწყისი კოდი:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class AddCookieServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException{
       //HTTP მოთხოვნიდან პარამეტრის მიღება
       String data=request.getParameter("data");
```

```
      //cookie ფაილის შექმნა
       Cookie cookie=new Cookie("MyCookie", data);
       //cookie ფაილის HTTP პასუხში დამატება
       response.addCookie(cookie);
       //გამოტანის ბრაუზერში ჩაწერა
       response.setContentType("text/html");
       PrintWriter pw=response.getWriter();
       pw.println("<B>MyCookie has been set to ");
       pw.println(data);
       pw.close(); }}
```

ნახ. 316

GetCookiesServlet.java ფაილი getCookie() მეთოდს იძახებს ნებისმიერი cookie ფაილის წასაკითხად, რომელიც HTTP GET მოთხოვნაშია ჩართული. აღნიშნული ინფორმაციის მისაღებად getName() და getValue() მეთოდები გამოიძახება.

GetCookiesServlet.java ფაილის საწყისი კოდი:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class GetCookieServlet extends HttpServlet {
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException{
       //cookie ფაილების HTTP მოთხოვნის სათაურიდან მიღება
       String data=request.getParameter("data");
       //cookie ფაილის შექმნა
       Cookie[] cookies=request.getCookies();
       //ყველა cookie ფაილის წარმოდგენა
       response.setContentType("text/html");
       PrintWriter pw=response.getWriter();
       pw.println("<B>");
       for(int i=0; i<cookies.length; i++){
              String name=cookies[i].getName();
              String value=cookies[i].getValue();
       pw.println("name=" + name + "; value= " + value);}
       pw.close(); }}
```

ნახ. 317

მოახდინეთ ზემოთ წარმოდგენილი სერვლეტების კომპილირება. გადააკოპირეთ ისინი საჭირო კატალოგში და განაახლეთ web.xml ფაილი. მოცემული მაგალითის შესამოწმებლად შეასრულეთ შემდეგი მოქმედებები:

დავალება:

1. *ჩატვირთეთ Tomcat კონტეინერი.*
2. *ბრაუზერის ფანჯარაში წარმოადგინეთ Addcookie.html ფაილი.*
3. *MyCookie ობიექტისთვის შეიყვანეთ მნიშვნელობა.*
4. *გააგზავნეთ ფორმა ვებ-გვერდზე.*

   ამის შემდეგ ბრაუზერი წარმოგიდგენთ საპასუხო შეტყობინებას.

   ბრაუზერის მისამართის ველში შემდეგი URL მისამართი ჩაწერეთ:

   http://localhost : 8080/examples/servlets//servlet/GetCookiesServlet

ბრაუზერის ფანჯარაში cookie ფაილების სახელები და მნიშვნელობები გამოჩნდება. წარმოდგენილ მაგალითში Cookie კლასის getMaxAge() მეთოდი არ გამოიყენება cookie ფაილების მოქმედების ვადის გასვლის ცხადი სახით მისათითებლად. შესაბამისად, cookie ფაილების მოქმედების ვადა ბრაუზერის სეანსის დასრულებისთანავე მთავრდება. getMaxAge() მეთოდის გამოყენების შემთხვევაში cookie ფაილი კლიენტის კომპიუტერის მყარ დისკზე შეინახება.

6.3.16. სეანსების განხილვა

ცნობილია, რომ ყოველი მოთხოვნა დამოუკიდებელია და ის წინა მოთხოვნაზე, შესაბამისად არ არის დამოკიდებული. მაგრამ, ზოგჯერ საჭიროა მდგომარეობის შესახებ ინფორმაციის ფლობა, რაც ბრაუზერისა და სერვერის ურთიერთობის შემდეგ შეგვიძლია გავაანალიზოთ. ამ მექანიზმს სეანსები გვთავაზობენ. სეანსი შეგვიძლია HttpServletRequest ინტერფეისის getSession() მეთოდის საშუალებით შევქმნათ. აღნიშნული მეთოდიდან HttpSession ინტერფეისის ობიექტი ბრუნდება. ამ ობიექტს სახელებსა და ობიექტებს შორის კავშირის ამსახველი ნაკრების შენახვა შეუძლია. ამ კავშირების მართვა შემდეგი მეთოდებით ხდება: setAttribute(), getAttribute(), getAttributeNames() და removeAttribute(). ყველა ეს მეთოდი HttpSession ინტერფეისის მეთოდს წარმოადგენს. სეანსის მდგომარეობა ყველა სერვლეტის მიერ ერთობლივად გამოიყენება, რომლებიც განსაზღვრულ კლიენტთან არიან დაკავშირებული.

მაგალითი 15. განვიხილოთ სერვლეტი, რომელიც სეანსის მდგომარეობის შესახებ ინფორმაციის გამოყენების დემონსტრირებას ახდენს. getSession() მეთოდი იღებს მიმდინარე სეანსს. სეანსის არარსებობის შემთხვევაში ახალი სეანსი იქმნება. getAttribute() მეთოდი date სახელთან დაკავშირებული ობიექტის მისაღებად გამოიძახება. ამ ობიექტს Date კლასის ობიექტი წარმოადგენს, რომელიც მოცემულ გვერდზე ბოლო წვდომის თარიღისა და დროის ინკაფსულირებას ახდენს (ცხადია, ასეთი კავშირი არ იქნება, თუ გვერდზე წვდომა პირველად ხდება). შემდეგ Date კლასის ობიექტი იქმნება, რომელიც მიმდინარე თარიღისა და დროის ინკაფსულირებას ახდენს. setAttribute() მეთოდი date სახელის ობიექტთან დაკავშირების მიზნით გამოიძახება.

პროგრამის კომპიუტერული რეალიზაცია:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class DateServlet extends HttpServlet {
	public void doGet(HttpServletRequest request, HttpServletResponse
response)
			throws ServletException, IOException{
		//HttpSession  ობიექტის მიღება
		HttpSession hs=request.getSession(true);
		//PrintWriter კლასის მიღება
		response.setContentType("text/html");
		PrintWriter pw=response.getWriter();
		pw.println("<B>");
		//ბოლო წვდომის თარიღისა და დროის წარმოდგენა
		Date date=(Date)hs.getAttribute("date");
		if(date != null){
			pw.print("Last access: " + date + "<br>");}
		//მიმდინარე თარღისა და დროის წარმოდგენა
		date=new Date();
		hs.setAttribute("date", date);
		pw.println("Current date: " + date);
	}
}
```

ნახ. 318

როდესაც აღნიშნულ სერვლეტს პირველად მოითხოვთ, ბრაუზერი მიმდინარე თარიღისა და დროის შემცველი ინფორმაციის ერთ სტრიქონს წარმოგიდგენთ. მომდევნო გამოძახებაზე კი ორი სტრიქონის აისახება. პირველ სტრიქონში მითითებული იქნება სერვლეტზე ბოლო წვდომის თარიღი და დრო, ხოლო მეორე სტრიქონში - მიმდინარე თარიღი და დრო.

კითხვები თვითშემოწმებისთვის:

1. GGI ინტერფეისთან შედარებით რა უპირატესობებით ხასიათდებიან სერვლეტები?
2. განმარტეთ სერვლეტის სიცოცხლის ციკლის განმსაზღვრელი მეთოდები.
3. სერვლეტების შემუშავების რა შესაძლებლობებია თქვენთვის ცნობილი?
4. რას წარმოადგენს cookie ფაილი?
5. რა ცნობებს მოიცავს cookie ფაილისთვის შენახული ინფორმაციის ნაწილი?

დავალება:

1. შექმენით მარტივი სერვლეტი, მოახდინეთ მისი კომპილაცია და შეამოწმეთ სერვლეტის ქმედითუნარიანობა.
2. შექმენით მიმდინარე თავის მე-12 მაგალითში განხილული სერვლეტი და შეამოწმეთ ის.
3. შექმენით მიმდინარე თავის მე-13 მაგალითში განხილული სერვლეტი და შეამოწმეთ ის.
4. შექმენით მიმდინარე თავის მე-14 მაგალითში განხილული სერვლეტები და შეამოწმეთ ისინი.
5. შექმენით მიმდინარე თავის მე-15 მაგალითში განხილული სერვლეტი და შეამოწმეთ ის.

6.4. Java-ს სერვერული ფურცლების დამუშავება

6.4.10. JSP ტექნოლოგია და ძირითადი კონსტრუქციები

JSP (JavaServer Pages) ტექნოლოგია ვებ-პროგრამისტებს საშუალებას აძლევს შექმნან როგორც სტატიკური, ისე დინამიკური კომპონენეტების შემცველი სერვერული ფურცლები. JSP ფურცელი ორი ტიპის ტექსტს მოიცავს: სტატიკურ საწყის მონაცემებს, რომლებიც ერთ-ერთ ტექსტურ ფორმატში: HTML, SVG, WML, ან XML შეიძლება იყოს გაფორმებული და JSP-ელემენტებს, რომლებიც დინამიური შიგთავსის კონსტრუირებას ახორციელებენ. ამას გარდა, შესაძლოა გამოვიყენოთ ბიბლიოთეკები: JSP-ტეგების და EL (Expression Language), იმ მიზნით, რომ Java-კოდის დანერგვა განვახორციელოთ JSP ფურცლების სტატიკურ შიგთავსში.

JSP-ფურცლების კოდის ტრანსლირება სერვლეტის Java-კოდში JSP-ფურცლების Jasper კომპილატორის საშუალებით ხდება, ხოლო შემდეგ მის კომპილირებას ბაიტ-კოდში (JVM) Java-ვირტუალური მანქანა ასრულებს. სერვლეტების კონტეინერები, რომელთაც JSP-ფურცლების შესრულება შეუძლიათ, დაწერილია პლატფორმისგან დამოუკიდებელ Java ენაზე. JSP-ფურცლები სერვერზე იტვირთება და მათ მართვას Java-ს სპეციალური server paket პაკეტი ახდენს, რომელსაც Java EE Web Application-ს უწოდებენ. როგორც წესი, ფურცლები .war და .ear ფაილურ არქივებში თავსდება. JSP პლატფორმისგან დამოუკიდებელი, გადატანითი და მარტივად გავრცელებადი ტექნოლოგიაა ვებ-უზრუნველყოფის დასამუშავებლად. ის საშუალებას გვაძლევს ფურცლის დინამიური ნაწილი გამოვყოთ სტატიკური HTML-ისგან. დინამიური ნაწილი სპეციალურ ტეგებში **"<% %>"** თავსდება. ასე, მაგალითად:

```
თქვენი ჰოსტის სახელი:  <%= request.getRemoteHost() %>
```

JSP-ფურცლებს .jsp გაფართოება აქვთ და იქ თავსდებიან, სადაც ჩვეულებრივი ვებ-გვერდები. ასეთი გვერდების სტრუქტურა ხუთი კონსტრუქციისგან შეიძლება შედგებოდეს: HTML, კომენტარი, სკრიპტული ელემენტები, დირექტივები და მოქმედებები. კომპილაციის დროს JSP-ფურცელი სტატიკური შიგთავსის მქონე სერვლეტად გარდაიქმნება, რომელიც გამოტანის ნაკადში მიემართება. ეს უკანასკნელი service() მეთოდთან არის დაკავშირებული. ამიტომ, პირველი მოთხოვნის შემთხვევაში, ამ პროცესმა შეიძლება მცირე დაყოვნება გამოიწვიოს. დოკუმენტში ან პროგრამაში არსებული კომენტარები პროგრამის შენელების მიზეზს არ წარმოადგენს, რადგან ტრანსლატორიც და შემსრულებელიც მის იგნორირებას ახდენენ. სკრიპტული ელემენტების საშუალებით კოდი Java ენაზე მიეთითება, რომელიც საბოლოოდ სერვლეტის ნაწილად იქცევა. დირექტივების საშუალებით სერვლეტის მთელი სტრუქტურის მართვაა შესაძლებელი. მოქმედებები კი არსებული გამოსაყენებელი კომპონენტების მიწოდებას ემსახურება. სკრიპტებთან მუშაობის გასამარტივებლად წინასწარ განსაზღვრული ცვლადები

გამოიყენება, ისეთები, როგორიცაა: request, response, pageContext, session, out, application, config, page, exception.

ახლა ცალ-ცალკე განვიხილოთ JSP-ფურცლების სტრუქტურის თითოეული კონსტრუქცია:

კომენტარები პროგრამის საწყისი ტექსტის განმარტების მიზნით გამოიყენება. JSP-ფურცლებში კომენტარები ორ ჯგუფად შეიძლება დავყოთ:

- JSP საწყისი კოდის კომენტარები.
- HTML -კომენტარები.

JSP საწყისი კოდის კომენტარები სიმბოლოთა სპეციალური მიმდევრობით აღინიშნება: <%-- კომენტარის დასაწყისში და --%> კომენტარის ბოლოს. კომენტარების ეს სახე JSP-გვერდის კომპილაციის ეტაპზე იშლება. JSP საწყისი კოდის კომენტარის ნიმუში ქვემოთ არის წარმოდგენილი:

```
<%--
    წარმოგვიდგენს ნაკეთობათა კატალოგს
    და მყიდველის აქტუალურ კალათს.
--%>
```

ნახ. 319

HTML კომენტარები HTML ენის წესების შესაბამისად ფორმდება. კომენტარების ეს სახე JSP-კომპილატორის მიერ განიხილება, როგორც სტატიკური ტექსტი და გამომავალ HTML-დოკუმენტში თავსდება. HTML-კომენტარებში მოთავსებული JSP-გამოსახულებები სრულდება. HTML-კომენტარის ნიმუში წარმოდგენილია 320-ე ნახაზზე:

```
<!-- გვერდის შექმნის თარიღი: <%= new java.util.Date()    %> -- >
```

ნახ. 320

სკრიპტული ელემენტები. JSP სპეციფიკაცია სკრიპტული ელემენტების სამ ტიპს განასხვავებს:

- გამოცხადებები <%! ერთი ან რამდენიმე დეკლარაცია %>
- გამოსახულებები <%= ერთი გამოსახულება %>
- სკრიპლეტები <% სკრიპლეტი%>

გამოცხადებები ძირითადად, ცვლადების, მეთოდების შიდა კლასების და კლასის დონეზე არსებული Java-ს დანარჩენი მოქმედი კონსტრუქციების განსაზღვრისთვის გამოიყენება. გამოსახულებებში Java-ს ნებისმიერი მოქმედი გამოსახულება იდება. სკრიპლეტების საშუალებით კი JSP-ფურცლებში Java კოდის მუშა ნაწილები თავსდება.

JSP გამოცხადებები საშუალებას გვაძლევს აღვწეროთ ცვლადები, მეთოდები, შიდა კლასები და ა.შ. გამოცხადებები პროგრამაში გამოყენებული Java-ს კონსტრუქციების განსაზღვრისთვის გამოიყენება. რადაგნ გამოცხადება მონაცემთა გამოტანას არ ითვალისწინებს, ამიტომ ისინი, როგორც წესი, JSP-გამოსახულებებსა და სკრიპლეტებთან ერთად გამოიყენება. 321-ე სურათზე წარმოდგენილ ფრაგმენტში ნაჩვენებია მოთხოვნების რაოდენობა მოცემულ გვერდზე სერვერის ჩატვირთვის მომენტიდან (ან ბოლო ცვლილების მომენტიდან და სერვლეტის გადატვირთვიდან). ყურადღება მიაქციეთ იმ ფაქტს, რომ ნიმუშში ჩვენ ვიყენებთ როგორც გამოცხადებას, ასევე გამოსახულებას და კონსტრუქციაში გამოცხადების შემდეგ წერტილ-მძიმე (;) დგას.

```
<%! private int accessCount = 0; %>

გვერდზე მიმართვების რაოდენობა სერვერის ჩატვირთვის მომენტიდან: <%= ++accessCount %>
```

ნახ. 321

JSP გამოსახულებები იმ მიზნით გამოიყენება, რომ Java-ს მნიშვნელობები უშუალოდ გამოტანის ნაკადში მოთავსდეს. ეს გამოსახულებები გამოითვლება, ხდება მათი კონვერტირება სტრიქონში და ვებ-გვერდზე ჩადგმა. ეს გამოთვლები შესრულების პროცესში მიმდინარეობს (ანუ გვერდის მოთხოვნის დროს) და მიტომ არსებობს სრული წვდომა თავად მოთხოვნის შესახებ ინფორმაციაზე. გამოსახულებებში შეიძლება მუდმივების, ცვლადების გამოყენება და სხვადასხვა მეთოდის გამოძახება. მიუხედავად სირთულისა, ყველა გამოსახულება გამოითვლება და მიიღება შედეგი. JSP-ფურცლები JSP Writer-ს ეყრდნობა, რაც იმ ფაქტს გულისხმობს, რომ ის გამოსახულების ნებისმიერ შედეგს იღებს, გარდაქმნის მას String (ტექსტურ) ტიპში და შეაქვს ბუფერულ მეხსიერებაში.

მაგალითად, ნახ.322-ე წარმოდგენილი კოდი მოცემულ ვებ-გვერდზე მოთხოვნის თარიღისა და დროის გამოტანას ემსახურება.

```
მიმდინარე დრო: <%= new java.util.Date()  %>

თქვენი ჰოსტის სახელი:  <%= request.getRemoteHost ()  %>
```

ნახ. 322

აუცილებელია ყურადღება მივაქციოთ სამ წესს:

- JSP გამოსახულებები უნდა შეიცავდნენ Java-ს გამოსახულებებს;
- ყოველი JSP გამოსახულება უნდა შეიცავდეს Java-ს მხოლოდ ერთ გამოსახულებას;
- JSP გამოსახულებები არ უნდა დასრულდეს წერტილ-მძიმით (;), Java-ს გამოცხადებებისგან განსხვავებით.

JSP სკრიპლეტები საშუალებას იძლევა ნებისმიერი კოდი სერვლეტის მეთოდში მოვათავსოთ, რომელიც გვერდის დამუშავების დროს შეიქმნება. ამასთან, შესაძლებელია Java-ს კონსტრუქციების უმეტესობა იქნეს გამოყენებული. სკრიპლეტებს ასევე გააჩნიათ წვდომა იმ წინასწარ განსაზღვრულ ცვლადებზე, რაც გამოსახულებებს. ამიტომ, მაგალითად, მნიშვნელობის გვერდზე გამოსატანად აუცილებელია წინასწარ განსაზღვრული out ცვლადის გამოყენება.

```
<%
String queryData = request.getQueryString();
out.println("მოთხოვნის დამატებითი მონაცემები: " + queryData);
%>
```

ნახ. 323

სკრიპლეტში კოდი იმ სახით თავსდება, როგორც იყო ჩაწერილი. სკრიპლეტამდე და მის შემდეგ მთელი სტატიკური HTML (შაბლონის ტექსტი) კონვერტირებას განიცდის print ოპერატორის საშუალებით.

მაგალითად, ნახ. 324-ზე წარმოდგენილი JSP ფრაგმენტი შეიცავს შაბლონის შერეულ ტექსტს და სკრიპლეტს:

```
<% if (Math.random() < 0.5) { %>
       <B>ბედნიერ</B> დღეს გისურვებთ!
<% } else { %>
      <B>ლამაზ</B> დღეს გისურვებთ!
<% } %>
```

ნახ. 324

სკრიპლეტის გარდაქმნის შემდეგ კოდი ნახ.325-ზე წარმოდგენილ სახეს მიიღებს:

```
if (Math.random() < 0.5) {
       out.println ("<B>ბედნიერ</B> დღეს გისურვებთ!");
} else {
      out.println ("<B>ლამაზ</B> დღეს გისურვებთ!");
 }
```

ნახ. 325

ეს ნიშნავს, რომ სავალდებულო არ არის, რომ სკრიპლეტები Java-ს დასრულებულ ფრაგმენტებს მოიცავდნენ და რომ, ღიად დატოვებული ბლოკები სკრიპლეტის გარეთ სტატიკურ HTML-ზე გავლენას იქონიებენ.

JSP დირექტივები. JSP-ფურცელს შეუძლია შესაბამის კონტეინერს გაუგზავნოს შეტყობინება იმ მოქმედებების მითითებით, რომელთა შესრულება აუცილებელია. ამ შეტყობინებებს დირექტივები ეწოდება. ყველა დირექტივა <%@ სიმბოლოებით იწყება, მას მოყვება დირექტივის დასახელება და ერთი ან რამდენიმე ატრიბუტი მნიშვნელობებით. დირექტივები %> სიმბოლოებით სრულდება. დირექტივების საშუალებით კონტეინერი აგზავნის განაცხადს განსაზღვრული მომსახურების შესრულების თაობაზე, რომელიც გენერირებულ დოკუმენტში არ ცხადდება. დირექტივების ჩაწერის ფორმა ნახ.326-ზეა წარმოდგენილი:

```
<%@ დირექტივა ატრიბუტი ="მნიშვნელობა" %>
```

ნახ. 326

არსებობს დირექტივების ჩაწერის გაერთიანებული ფორმაც, როგორც ეს ნახ.327-ზეა წარმოდგენილი:

```
<%@ დირექტივა ატრიბუტი1 ="მნიშვნელობა1"
              ატრიბუტი2 ="მნიშვნელობა2"
              . . .
              ატრიბუტიN ="მნიშვნელობაN"  %>
```

ნახ. 327

არსებობს დირექტივების სამი ძირითადი ტიპი: **page**, რომელიც თქვენ უფლებას გაძლევთ შეასრულოთ ისეთი ოპერაციები, როგორიცაა კლასების იმპორტირება, სერვლეტის სუპერ კლასის შეცვლა და ა.შ.; **include,** რომელიც საშუალებას გაძლევთ სერვლეტის კლასში ჩასვათ ფაილი სერვლეტში JSP ფაილის ტრანსლაციის დროს; და **taglib**, რომელიც უფლებას გაძლევთ გააფართოვოთ ტეგების სიმრავლე საკუთარით, რომელთაც JSP კონტეინერი აღიქვამს.

JSP page დირექტივა. როგორ შეიძლება სახელწოდებით მივხვდეთ, რომ მოცემული დირექტივა ატრიბუტებს აწვდის JSP ფურცელს? დირექტივის მიერ განსაზღვრული ატრიბუტები ინერგება მოცემულ JSP ფურცელში და ყველა მის ჩადგმულ სტატიკურ ელემენტში, დამოუკიდებლად იმისა, include დირექტივის საშუალებით იყვნენ ისინი ჩადგმული, თუ jsp:include მოქმედებით. page დირექტივის ჩაწერის ფორმა შემდეგია:

```
<%@ page ატრიბუტი="მნიშვნელობა" %>
```

ნახ. 328

ნიმუშად შემდეგი ჩანაწერი მოვიყვანოთ:

```
<%@ page import="java.util.*, com.myclasses.*" buffer="15kb" %>
```

ნახ. 329

ეს დირექტივა აცხადებს, რომ JSP ფურცელი კლასების იმპორტირებას ორი პაკეტიდან java.util და com.myclasses ახდენს. შემდეგ ამოწმებს ბუფერული მეხსიერების ზომას, რომელიც გამოყენებულ უნდა იქნეს მოცემული JSP გვერდის დასამუშავებლად.

განვიხილოთ **page** დირექტივის ატრიბუტები:

- **import="პაკეტი.class1, პაკეტი.class2, ..., პაკეტი.classN".** ეს ატრიბუტი უფლებას გაძლევთ მიუთითოთ პაკეტები, რომლებიც იმპორტირებული უნდა იქნეს. ეს ერთადერთი ატრიბუტია, რომელიც ერთ დირექტივაში რამდენჯერმე შეიძლება გამოიყენოთ. სიაში აუცილებელია Java-ს ყველა იმ კლასის ჩართვა, რომელთა გამოყენებაც გსურთ და რომლებიც იმპორტირებული კლასების საწყისი ნაკრების ნაწილს არ წარმოადგენენ. საწყისი ნაკრები შეიცავს: java.lang.*, javax.servlet.*, javax.servlet.jsp.* და javax.servlet.http.*-ს.
  import ატრიბუტის გამოყენების მაგალითი ნახ.330-ზეა წარმოდგენილი.

```
<%@ page import="java.util.Date, javax.text.SimpleDateFormat,com.myclasses.*"
%>
```

ნახ. 330

- **language="java".** მოცემული ატრიბუტი განკუთვნილია გამოყენებული დაპროგრამების ენის მისათითებლად. სტანდარტულად (მიუთითებლობის შემთხვევაში), „java“ მიიღება. ამ ატრიბუტის გამოყენება სავალდებულო არ არის, მაგრამ შესაძლოა პრობლემა მაშინ წარმოიქმნას, თუ JSP კონტეინერის მომწოდებელი სხვა ენას იყენებს (მაგალითად, JavaScript). აღნიშნული ატრიბუტის ჩაწერის ფორმა შემდეგია:

```
<%@ page language="java" %>
```

ნახ. 331

- **extends="პაკეტი. class"**. ეს ატრიბუტი გენერირებადი სერვლეტისთვის იძლევა სუპერკლასს (მშობელ კლასს). როგორც წესი, სერვლეტი საწყისი კლასის გაფართოებით წარმოიქმნება. გამოცდილ პროგრამისტებს ამ ატრიბუტის საშუალებით საკუთარი სუპერკლასების შექმნა შეუძლიათ. აღნიშნული ატრიბუტი გამოყენების ერთ-ერთი ნიმუში შემდეგი სახით შეიძლება წარმოვადგინოთ:

```
<%@ page extends="myPackage.HttpExample" %>
```

ნახ. 332

- **session="true|false".** წარმოდგენილმა ატრიბუტმა შეიძლება მიიღოს true ან false მნიშვნელობა, რომლებიც განსაზღვრავენ იღებს თუ არა მონაწილეობას JSP ფურცელი HTTP-ს ტრანსლაციაში. true („ჭეშმარიტი“ მიუთითებლობის შემთხვევაში) მნიშვნელობა იმაზე მეტყველებს, რომ წინასწარ განსაზღვრული session (ტიპი HttpSession) ცვლადი არსებულ სესიასთან უნდა იყოს დაკავშირებული (ცხადია, თუ ასეთი არსებობს), წინააღმდეგ შემთხვევაში ახალი სესია

იქმნება, რომელზეც კავშირი ხორციელდება. false („მცდარი“) მნიშვნელობა განსაზღვრავს, რომ სესიები არ იქნება გამოყენებული და session ცვლადზე მიმართვის ყოველი მცდელობა შეცდომას გამოიწვევს JSP გვერდის სერვლეტში ტრანსლაციის დროს. აღნიშნული ატრიბუტის გამოყენების ერთ-ერთი ნიმუში შემდეგი სახით შეიძლება წარმოვადგინოთ:

```
<%@ page session="false" %>
```

ნახ. 333

- **buffer="ზომა kb|none".** წარმოდგენილი ატრიბუტი ბუფერული მეხსიერების მოცულობას იძლევა, რაც JspWriter ობიექტისთვის აუცილებელია. მასზე მიმართვას წინასწარ განსაზღვრული out ცვლადი ახორციელებს. მისი სტანდარტული მნიშვნელობა სერვერის პარამეტრებზეა დამოკიდებული, მაგრამ ის 8 კბაიტს არ უნდა აღემატებოდეს. მნიშვნელობა მიეთითება ან „ზომა kb" ფორმით ან „none" ფორმით. თუ ბუფერული მეხსიერების მნიშვნელობას მიუთითებთ, როგორც „none", მაშინ სერვერი არაფერს შეინახავს ბუფერში და out ცვლადამდე ჩაწერილ შედეგს პირდაპირ გადასცემს PrintWriter ობიექტს. თუ ბუფერის ზომას კონკრეტული მნიშვნელობით აიღებთ, მაშინ JspWriter ობიექტი ამ მეხსიერებაში შეინახავს მონაცემებს, რის შედეგადაც წარმადობა მნიშვნელოვნად გაიზრდება. PrintWriter ობიექტისგან განსხვავებით, JspWriter ობიექტს შეუძლია IOException გამონაკლისის გადაცემა. ბუფერული მეხსიერების საწყისი მნიშვნელობა 8 კბაიტის ტოლია. აღნიშნული ატრიბუტის ჩაწერის ერთ-ერთი ნიმუში შემდეგი სახით შეიძლება წარმოვადგინოთ:

```
<%@ page buffer="12kb" %>
```

ნახ. 334

- **autoflush="true|false".** მოცემული ატრიბუტი true ან false მნიშვნელობებს იღებს. true („ჭეშმარიტი", მუთითებლობის შემთხვევაში) მნიშვნელობა ადგენს, რომ ბუფერული მეხსიერების გადავსების შემთხვევაში, ის ავტომატურად გასუფთავდება. false („მცდარი") მნიშვნელობა, რაც ძალზე იშვიათად გამოიყენება, ადგენს, რომ ბუფერის გადავსებამ გამონაკლისი სიტუაცია (IOExceptions) უნდა გამოიწვიოს. როგორც წესი, buffer და autoflush ატრიბუტები ერთ დირექტივაში ერთად ეთითება. ატრიბუტის buffer="none" მნიშვნელობის დაყენებისას და autoflush ატრიბუტისთვის false მნიშვნელობის დაყენება დაუშვებელია. აღნიშნული ატრიბუტის ჩაწერის ერთ-ერთი ნიმუში შემდეგი სახით შეიძლება წარმოვადგინოთ:

```
<%@ page buffer="16kb" autoflush="true" %>
```

ნახ. 335

- **isThreadSafe="true|false"**. ეს ატრიბუტი true ან false მნიშვნელობებს იღებს. true („ჭეშმარიტი“, მუთითებლობის შემთხვევაში) მნიშვნელობა სერვლეტის შესრულების ნორმალურ რეჟიმს აყენებს მაშინ, როდესაც მრავალი მოთხოვნა ერთდროულად მუშავდება სერვლეტის ერთი ეგზემპლიარის გამოყენებით, გამომდინარე იქიდან, რომ ავტორმა ამ ეგზემპლიარის ცვლადებზე წვდომა სინქრონიზებული გახადა. false („მცდარი“) მნიშვნელობა მიუთითებს, რომ სერვლეტმა SingleThreadModel (ერთნაკადიანი მოდელი) უნდა მიიღოს, რომლის დროსაც მიმდევრობითი და ერთდროული მოთხოვნები სერვლეტის ცალკეული ეგზემპლიარის მიერ დამუშავდება. სხვა სიტყვებით რომ ვთქვათ, true მნიშვნელობა ქმნის იმ გარემოებას, როდესაც კონტეინერი სერვლეტს რამდენიმე მოთხოვნას ერთდროულად უგზავნის, მაშინ, როდესაც false მნიშვნელობის შემთხვევაში, ცალკეული მოთხოვნის გაგზავნას აქვს ადგილი. აღნიშნული ატრიბუტის გამოყენების ერთ-ერთ ნიმუშს შემდეგი სახე აქვს:

```
<%@ page isThreadSafe="false" %>
```

ნახ. 336

- **info="ინფორმაცია".** ეს ატრიბუტი სტრიქონს იძლევა, რომელიც Servlet.getServletInfo() მეთოდის გამოყენების შემთხვევაში მიიღება. ჩვეულებრივ, ეს მეთოდი სერვლეტის შესახებ ინფორმაციას (მაგალითად, ინფორმაცია ავტორის, ვერსიის, საავტორო უფლებების შესახებ) აბრუნებს. აღნიშნული ატრიბუტის ჩაწერის ერთ-ერთ ნიმუშს შემდეგი სახე აქვს:

```
<%@ page info="ავტორი: ლელა გაჩეჩილაძე; ვერსია:1.0" %>
```

ნახ. 337

- **errorPage="url".** აღნიშნული ატრიბუტი JSP გვერდს წარმოგვიდგენს, რომელიც გარკვეული Throwables მოვლენის (რაც მოცემულ გვერზე არ მუშავდება) წარმოქმნის შემთხვევაში გამოიძახება. თუ JSP გვერდზე იქმნება გამონაკლისი სიტუაცია და JSP გვერდს ამ გამონაკლისის გადაწყვეტის კოდი არ გააჩნია, მაშინ კონტეინერი მართვას ავტომატურად იმ URL-ს გადასცემს, რომელიც თქვენ errorPage ატრიბუტის მნიშვნელობის სახით მიუთითეთ.
აღნიშნული ატრიბუტის ჩაწერის ერთ-ერთ ნიმუშს შემდეგი სახე აქვს:

```
<%@ page errorPage="/ myweb/errors/myerror.jsp" %>
```

ნახ. 338

- **isErrorPgae="true|false".** ეს ატრიბუტი true ან false მნიშვნელობებს იღებს. იგი მიუთითებს, შეიძლება თუ არა შეცდომების დასამუშავებლად მიმდინარე გვერდის გამოყენება სხვა JSP გვერდებზე. მიუთითებლობის შემთხვევაში false („მცდარი") მნიშვნელობა აიღება. აღნიშნული ატრიბუტის გამოყენების ერთ-ერთ ფორმას შემდეგი სახე აქვს:

```
<%@ page isErrorPage="true" %>
```

ნახ. 339

- **contentType="MIME-ტიპი".** წარმოდგენილი ატრიბუტი გამოტანის მიზნით MIME ტიპს იძლევა, ხოლო სურვილის შემთხვევაში, HTML პასუხში შესაძლებელია ნიშნების კოდირებაც მიეთითოს. მიუთითებლობის შემთხვევაში MIME-ს მნიშვნელობად text/html გამოიყენება. მეტი თვალსაჩინოებისთვის შემდეგი მაგალითი შეგვიძლია წარმოვადგინოთ:

```
<%@ page contentType="text/plain" %>
```

ნახ. 340

იგივე შედეგს სკრიპლეტის გამოყენებითაც მივიღებთ:

```
<% response<% response ("text/plain"); %>
```

ნახ. 341

JSP include დირექტივა. აღნიშნული დირექტივა უფლებას გაძლევთ JSP ფურცლის ტრანსლირების პროცესში სერვლეტში ფაილები ჩართოთ. დირექტივის გამოყენების ნიმუში ნახ. 342-ზეა წარმოდგენილი:

```
<%@ include file="შესაბამისი url" %>
```

ნახ. 342

მოცემული URL ჩვეულებრივ, JSP ფურცლის შესაბამისად განიხილება, რომელზეც მიმართვა ხდება, მაგრამ ისევე, როგორც სხვა ნებისმიერი URL-ის გამოყენების შემთხვევაში, თქვენ შეგიძლიათ სისტემას გადასცეთ თქვენთვის საინტერესო რესურსი Web სერვერის შიდა კატალოგის შესაბამისად. ამისთვის საკმარისია URL-ის დასაწყისში <</>> სიმბოლოს დამატება. ჩართული ფაილის შიგთავსი მუშავდება, როგორც ჩვეულებრივი JSP ტექსტი და ამიტომ ის შეიძლება

მოიცავდეს ისეთ ელემენტებს, როგორიცაა სტატიკური HTML, სკრიპტების ელემენტები, დირექტივები და მოქმედებები. მაგალითად, მრავალი საიტი ნავიგაციის პანელს იყენებს ყოველ გვერდზე. HTML ფრეიმების გამოყენების პრობლემების გამო, ხშირად ეს ამოცანა ფურცლის ზემოთ ან მარცხენა ნახევარში მცირე ზომის ცხრილის განთავსებით წყდება, ხოლო მისი შესაბამისი HTML კოდი საიტის ყოველი გვერდისთვის მრავალჯერ მეორდება. ამ შემთხვევაში, include დირექტივის გამოყენება დასმული ამოცანის გადაწყვეტის ყველაზე მარტივი გზაა, რაც პროგრამისტებს საშუალებას აძლევს თავი აარიდონ HTML კოდის ყოველ ცალკეულ ფაილში კოპირებას. ეს კი შემდეგი სახით ხორციელდება:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<html>
<head>
<title>სატესტო გვერდი</title>
<head>
<body>
<%@ include file="/ navbar.html" %>
<!-- ამ გვერდის სპეციფიური ფრაგმენტი... -->
<body>
<html>
```

ნახ. 343

გაითვალისწინეთ, რადგან include დირექტივა ფაილების ჩართვას გვერდის ტრანსლაციის დროს ახორციელებს, ამიტომ ნავიგაციის პანელში ცვლილებების შეტანის შემდეგ, თქვენ განმეორებითი ტრანსლაცია დაგჭირდებათ ყველა იმ JSP ფურცლის, რომლებიც მის გამოყენებას ახდენენ; რაც, მოცემულ შემთხვევაში კარგი კომპრომისია, რადგან, როგორც წესი, ნავიგაციის პანელი საკმაოდ იშვიათად იცვლება და მისი ჩართვის პროცესი თავის ეფექტურობას არ კარგავს. თუ ჩართული ფაილები ხშირად იცვლება, მაშინ თქვენ jsp:include მოქმედება შეგიძლიათ გამოიყენოთ. ის ფაილის ჩართვას JSP-ზე მიმართვის პროცესში ახდენს.

JSP taglib დირექტივა. როგორც ცნობილია, JSP-ფურცლებში ელემენტები ნიშნების (პირობითი ნიშნების, ჭდეების) საშუალებით ჩაიწერება. ამ ნიშნების სიმრავლე შესაძლებელია ე.წ. ნიშნების ბიბლიოთეკის საშუალებით გავაფართოვოთ. ასევე, ნიშნების გაფართოებულ სიმრავლეს შეიძლება დავუმატოთ მოქმედებები, რის შედეგადაც, ადგილი აქვს თავად JSP ენის ფართოებას. მოქმედებები შეგვიძლია დავყოთ სტანდარტულ და საკუთარ მოქმედებებად.

taglib დირექტივის ჩაწერის განზოგადებულ ფორმას შემდეგი სახე აქვს:

```
<%@ taglib uri="URI ნიშნების ბიბლიოთეკისკენ" prefix="ნიშნის პრეფიქსი" %>
```

ნახ. 344

ნიშნების ბიბლიოთეკის იდენტიფიცირება აუცილებელია URI მისამართის (რესურსის უნიკალური იდენტიფიკატორის) საშუალებით. URI შეიძლება იყოს როგორც აბსოლუტური, ისე ფარდობითი. ის განსაზღვრავს ნიშნების ბიბლიოთეკის ადგილმდებარეობას (TLD), რომელიც აღნიშნული ბიბლიოთეკის საკუთარ ნიშნებს განსაზღვრავს. დირექტივის ჩაწერის სინტაქსი შემდეგია:

```
<%@ taglib uri="http://www.moywebserver.ru/naydiznaki.tld" prefix="iskat"%>
```

ნახ. 345

ყოველი დირექტივისთვის აუცილებელია განსხვავებული პრეფიქსების დართვა, რომლებიც ბიბლიოთეკის შიგთავსს განსაზღვრავენ ფურცელზე. პრეფიქსის სახით შეიძლება გამოიყენოთ ნებისმიერი ტექსტი, სიტყვა. მაშინ, როდესაც taglib დირექტივის გამოყენება JSP-ფურცელზე ნებისმიერ ადგილას შეიძლება, ყველა საკუთარი ნიშანი, რომლებსაც ეს დირექტივები მიმართავს, მათ შემდეგ გამოიყენება.

მოქმედებები. JSP მოქმედებები XML სინტაქსის მქონე კონსტრუქციებს იყენებენ. თქვენ შეგიძლიათ დინამიურად მოახდინოთ ფაილების ჩართვა, გამოიყენოთ Java Beans კომპონენტები, გადაამისამართოთ მომხმარებელი სხვა გვერდზე ან მოახდინოთ HTML-ის გენერირება Java plugin-ისთვის. აღნიშნულ მოქმედებებს დეტალურად გავეცნობით. გახსოვდეთ, რომ, ისევე როგორც, მთელ XML-ში, აქაც ელემენტებისა და ატრიბუტების სახელები დამოკიდებულია რეგისტრზე. მოქმედებები შეგვიძლია ორ ჯგუფად დავყოთ: სტანდარტული და შექმნილი (საკუთარი, რასაც თავად პროგრამისტი ქმნის). დასაშვებია შემდეგი სტანდარტული მოქმედებების გამოყენება:

- ***jsp:declaration*** - გამოცხადება, რომელიც ანალოგიურია ტეგის <%! … %>;
- ***jsp:scriptlet*** - სკრიპლეტი, რომელიც ანალოგიურია ტეგის <% … %>;
- ***jsp:expression*** - გამოსახულება, რომელიც ანალოგიურია ტეგის <%= … %>;
- ***jsp:text*** - ტექსტის გამოტანა;
- ***jsp:useBean*** - JavaBean-ის ახალი ეგზემპლიარის ძებნა ან შექმნა;
- ***jsp:setProperty*** - JavaBean-ის თვისებების დაყენება;
- ***jsp:getProperty*** - JavaBean-ის თვისებების გამოტანის ნაკადში ჩასმა;
- ***jsp:include*** - ფაილის ჩართვა გვერდის მოთხოვნის მომენტში;

- ***jsp:forward*** - სხვა გვერდზე მოთხოვნის გადამისამართება
- ***jsp:param*** - მოთხოვნის ობიექტში ამატებს პარამეტრებს, მაგალითად, ელემენტებში forward, include, plugin.;
- ***jsp:plugin*** - ახდენს კოდის გენერირებას (გამოყენებული ბრაუზერის ტიპზე დამოკიდებულებით), რომელიც ქმნის OBJECT ან EMBED ტეგს Java plugin-ისთვის;
- ***jsp:params*** - აჯგუფებს პარამეტრებს jsp:plugin ტეგის შიგნით;
- ***jsp:fallback*** - უთითებს შიგთავსს, რომელსაც კლიენტის ბრაუზერი გამოიყენებს იმ შემთხვევაში, თუ ჩართული მოდული არ გაეშვება. გამოიყენება plugin ელემენტის შიგნით.

კითხვები თვითშემოწმებისთვის:

1. რას წარმოადგენს JSP ტექნოლოგია?
2. რა დანიშნულება აქვს Jasper კომპილატორს?
3. რა კონსტრუქციებისგან შეიძლება შედგებოდეს JSP-ფურცლები?
4. რა განსხვავებაა JSP საწყისი კოდის კომენტარებსა და HTML -კომენტარებს შორის?
5. სკრიპტული ელემენტების რა ტიპებია თქვენთვის ცნობილი?
6. რა დანიშნულება აქვთ JSP სკრიპლეტებს?
7. რა დანიშნულება აქვთ JSP დირექტივებს?
8. რისთვის გამოიყენება JSP page დირექტივა?
9. დაახასიათეთ page დირექტივის import ატრიბუტი.
10. დაახასიათეთ page დირექტივის session ატრიბუტი.
11. დაახასიათეთ page დირექტივის buffer ატრიბუტი.
12. დაახასიათეთ page დირექტივის autoflush ატრიბუტი.
13. რა განსხვავებაა page დირექტივის errorPage და isErrorPgae ატრიბუტებს შორის?
14. რა დანიშნულება აქვს JSP include დირექტივას?
15. რისთვის გამოიყენება JSP taglib დირექტივა?

6.4.11. java-ს სერვერული ფურცლების კავშირი java ბინებთან (სპეციალურ კლასებთან)

Java Beans ეს პროგრამული უზრუნველყოფის ან პროგრამის კომპონენტია, რომლის მრავალჯერადი გამოყენება არაერთ სხვადასხვა გარემოშია შესაძლებელი. ფუნქციონალური შესაძლებლობების თვალსაზრისით, Java Beans კომპონენტს შეზღუდვები არ გააჩნია. მას შეუძლია როგორც მარტივი ფუნქციის შესრულება, მაგალითად, საქონლის მარაგის ღირებულების მიღება, ასევე რთული ფუნქციების შესრულებაც, მაგალითად, კომპანიის აქციების მდგომარეობის წინასწარი ანალიზი. Java Beans კომპონენტი ხილვადი მხოლოდ ბოლო მომხმარებლისთვისაა. ასეთი კომპონენტის ერთ-ერთ მაგალითს მომხმარებლის გრაფიკული ინტერფეისის ღილაკი წარმოადგენს. Java Beans კომპონენტი მომხმარებლისთვისაც შეიძლება იყოს უხილავი. და ბოლოს, Java Beans კომპონენტის დანიშნულებას შეიძლება მომხმარებლის სამუშაო „სადგურზე“ ავტომატურ რეჟიმში მუშაობა წარმოადგენდეს. Java Beans კომპონენტები განსაზღვრავს არქიტექტურას, რომელიც სამშენებლო ბლოკებს შორის ურთიერთობას აწესრიგებს. ამ კომპონენტების მნიშვნელობა უდავოდ დიდია, რადგან ისინი საშუალებას გვაძლევენ რთული სისტემები პროგრამული კომპონენტების საფუძველზე შევქმნათ.

Java Beans კომპონენტების ტექნოლოგია მთელი რიგი უპირატესობებით ხასიათდება, კერძოდ:

- Java Beans კომპონენტი Java-ს პარადიგმების ყველა უპირატესობას ფლობს, რომელთა მთავარი იდეა შემდეგში მდგომარეობს: „ერთხელ დაწერილი ყველგან მუშაობს“.
- შესაძლებელია Java Beans კომპონენტების თვისებების, მოვლენებისა და მეთოდების მართვა.
- Java Beans კომპონენტის კონფიგურაციის პარამეტრები შეიძლება ინფორმაციის მუდმივ მატარებელზე იქნეს შენახული და საჭიროების შემთხვევაში აღვადგინოთ.
- Java Beans კომპონენტებს შეუძლიათ შეტყობინებები მიიღონ სხვა ობიექტების მოვლენებთან დაკავშირებით და ასევე, მოგვაწოდონ ინფორმაცია სხვა ობიექტებში მიმდინარე მოვლენებზე.

Java Beans კომპონენტებს საფუძვლად თვითდიაგნოსტირების (introspection) თვისება უდევთ. თვითდიაგნოსტირება - ეს Java Beans კომპონენტის ანალიზის პროცესია, რომლის დროსაც მისი მახასიათებლები განისაზღვრება. თვითდიაგნოსტირების გარეშე Java Beans კომპონენტების ტექნოლოგია არ იმუშავებს.

არსებობს ორი საშუალება, რომელთა გამოყენებით Java Beans კომპონენტის შემქმნელს შეუძლია გვაჩვენოს, თუ მისი რომელი თვისებები, მოვლენები და მეთოდები იქნება წვდომადი. ერთ შემთხვევაში დასახელებებთან დაკავშირებული მარტივი შეთანხმებები გამოიყენება. ისინი თვითდიაგნოსტირების მექნიზმებს საშუალებას აძლევენ Java Beans კომპონენტის შესახებ ლოგიკურად გამოიტანონ ინფორმაცია. მეორე შემთხვევაში დამატებითი კლასი გამოიყენება,

რომელიც BeansInfo ინტერფეისს აფართოვებს. ეს უკანასკნელი ინფორმაციას ცხადი სახით გვაწვდის.

Java Beans კომპონენტის თვისება (property) მისი მდგომარეობის შემოკლებულ ვარიანტს წარმოადგენს. თვისებებზე მინიჭებული მნიშვნელობები კომპონენტის ქცევას და გამოჩენას განსაზღვრავენ. თვისების აწყობა ე.წ. ჩაწერის მეთოდის (setter method) საშუალებით ხდება, ხოლო მისი მიღება - წაკითვხის მეთოდით (getter method). თვისებები ორი სახისაა: მარტივი და ინდექსირებული. მარტივ თვისებას ერთი მნიშვნელობა აქვს, ხოლო ინდექსირებული თვისება რამდენიმე მნიშვნელობისგან შედგება.

მუდმივობა (persistence) Java Beans კომპონენტის უნარია შეინარჩუნოს მიმდინარე მდგომარეობა მუდმივ დამამახსოვრებელ მოწყობილობაზე და საჭიროების შემთხვევაში მოხდეს მისი ამოღება. Java Beans კომპონენტების მუდმივობის უზრუნველსაყოფად სერიალიზაციის საშუალებები გამოიყენება, რომლებსაც Java-ს კლასების ბიბლიოთეკები გვთავაზობენ.

ახლა კი განვიხილოთ ის მოქმედებები და მათი დანიშნულებები, რომლებიც უშუალოდ Java Beans კომპონენტებს უკავშირდება.

jsp:useBean მოქმედება უფლებას გვაძლევს ჩავტვირთოთ Java-ს სპეციალური შესაძლებლობა Java Beans, JSP ფურცელზე მისი შემდგომი გამოყენების მიზნით. ეს საშუალებას გვაძლევს მრავალჯერ გამოვიყენოთ Java-ს კლასები და ამავდროულად, უარი არ ვთქვათ იმ უპირატესობებზე, რომელსაც JSP სერვლეტები გვთავაზობენ. გამოყენებული Bean-ის მითითების მარტივ სინტაქსს შემდეგი სახე აქვს:

```
<jsp:useBean id="სახელი" class="პაკეტი.class" />
```

ნახ. 346

როგორც წესი, ეს ნიშნავს „კლასის ობიექტის ახალი ეგზემპლიარის შექმნას და მის კავშირს ცვლადთან, რომლის სახელიც id საშუალებით გადაიცა. თუმცა, შესაძლებელია scope ატრიბუტის გადაცემაც (მას შეუძლია მიიღოს შემდეგი შესაძლო მნიშვნელობები: page|request|session|application. page მნიშვნელობას ის გვერდისთვის იღებს, request მნიშვნელობას - მოთხოვნებისთვის, session მნიშვნელობას - სესიების ან დიალოგებისთვის, application მნიშვნელობას - პროგრამისთვის), რომელიც Beans-ის ასოცირებას არამხოლოდ მიმდინარე გვერდთან ახდენს. ასეთ შემთხვევაში, სასარგებლოა არსებულ Beans-ზე მიმართვების მიღება, ხოლო jsp:useBean მოქმედება ახალი ობიექტის ეგზემპლიარს მხოლოდ იმ შემთხვევაში ქმნის, თუ არცერთი ობიექტი არ არსებობს იგივე id და scope მნიშვნელობებით. ახლა, როდესაც თქვენ უკვე გაქვთ Bean, მისი თვისებები შეგიძლიათ jsp:setProperty მოქმედებით შეცვალოთ. ამ მიზნით

სკრიპლეტსაც შეგიძლიათ მიმართოთ და ცხადი სახით გამოიძახოთ ობიექტის მეთოდი ცვლადის იმ სახელით, რომელიც მანამდე id ატრიბუტით გადაეცით. როდესაც ვამბობთ, რომ „ამ Beans-ს აქვს x ტიპის თვისება სახელით foo", სინამდვილეში ვგულისხმობთ, რომ „ამ კლასს აქვს getFoo მეთოდი, რომელიც x ტიპის მონაცემებს აბრუნებს და setFoo მეთოდი, რომელსაც პარამეტრის როლში x გადაეცემა". არსებული თვისებების მნიშვნელობები ასევე შეგიძლიათ მიიღოთ JSP გამოსახულებების ან სკრიპლეტების საშუალებით შესაბამისი getxxx მეთოდის გამოძახებით ან (რაც ძალზე ხშირია) გამოიყენოთ მოქმედება: jsp:getProperty.

Bean-თვის განსაზღვრული კლასი სერვერის კლასების კატალოგში უნდა იყოს მოთავსებული და არა კლასებისთვის დარეზერვებულ ნაწილში, რომლებიც რედაქტირების შემდეგ ავტომატურად გადაიტვირთებიან ხოლმე. მაგალითად, Java Web Server-თვის ყველა გამოყენებული კლასი classes კატალოგში უნდა იყოს განთავსებული ან lib კატალოგის .jar ფაილში და არა servlet კატალოგში. ქვემოთ წარმოდგენილია მარტივი მაგალითი, რომელიც ახდენს Bean-ის ჩატვირთვას და იღებს მარტივ სტრიქონულ პარამეტრს.

BeanTest.jsp-ს პროგრამული რეალიზაცია:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title> JavaBeans მრავალჯერადი გამოყენება JSP-ში</title>
</head>
<body>
<h1> JavaBeans მრავალჯერადი გამოყენება JSP-ში</h1>
<jsp:useBean id="test" class="hall.SimpleBean" />
<jsp:setProperty name="test" property="message" value="Hi,WWW"/>
<p>
შეტყობინება:
<jsp:getProperty name="test" property="message" />
</p>
</body>
</html>
```

ნახ. 347

SimpleBean.java-ს პროგრამული რეალიზაცია:

```
package hall;
public class SimpleBean {
private String message = "შეტყობინების ტექსტი არ არის";
public String getMessage() {
return(message); }
public void setMessage(String message) {
this.message = message; } }
```

ნახ. 348

Bean-ის გამოყენების უმარტივეს გზას შემდეგი კონსტრუქცია წარმოადგენს:

```
<jsp:useBean id="სახელი" class=" პაკეტი.class" />
```

ნახ. 349

Bean-ის მოდიფიკაციისა და მისი თვისებების (პარამეტრების) მისაღებად jsp:setProperty და jsp:getProperty მოქმედებები გამოიყენება. არსებობს კიდევ ორი სხვა შესაძლებლობა. პირველ რიგში, თქვენ გაქვთ საშუალება კონტეინერის შემდეგი ფორმატი გამოიყენოთ:

```
<jsp:useBean ...>
ტანი
</jsp: useBean>
```

ნახ.350

Beans შეიძლება ერთობლივად გამოიყენოთ, ამიტომ ყოველი jsp:useBean გამოსახულება არ იწვევს ახალი beans ეგზემპლიარის შექმნას. მეორეს მხრივ, id და class-ის გარდა კიდევ სამი ატრიბუტი არსებობს, რომლებიც შეგიძლიათ გამოიყენოთ. ეს ატრიბუტებია: scope, type და beanName. მოკლედ დავახასიათოთ ისინი:

- **id** - იმ ცვლადის სახელს იძლევა, რომელიც bean-ს მიმართავს. თუ მოიძებნება იმავე id და scope მნიშვნელობების მქონე bean, მაშინ ახალი ეგზემპლიარის შექმნის ნაცვლად ადრე შექმნილი ობიექტი გამოიყენება;
- **class** - იძლევა bean პაკეტის სრულ სახელს;
- **scope** - წარმოგვიდგენს იმ არეს, რომელშიც bean წვდომადი უნდა იყოს. მან შეიძლება ოთხი შესაძლო მნიშვნელობა მიიღოს. ესენია: **page**, **request**, **session** და **application.** მიუთითებლობის შემთხვევაში, სტანდარტულად page მნიშვნელობა აიღება, რაც ნიშნავს, რომ bean წვდომადია მხოლოდ მიმდინარე გვერდზე (თავსდება მიმდინარე გვერდის PageContext-ში). request

მნიშვნელობა გვიჩვენებს, რომ bean წვდომადია კლიენტის მხოლოდ მიმდინარე მოთხოვნისთვის (თავსდება ობიექტში ServletRequest). session მნიშვნელობა ნიშნავს, რომ ობიექტი წვდომადია ყველა გვერდისთვის მიმდინარე HttpSession-ის სიცოცხლის მთელი ციკლის მანძილზე. და, ბოლოს application მნიშვნელობა გვიჩვენებს, რომ ის წვდომადია ყველა გვერდისთვის, რომლებიც იმავე ServletContext-ს იყენებენ. ამ ატრიბუტის გამოყენების აუცილებლობის მიზეზი იმაში მდგომარეობს, რომ jsp:useBean მოქმედება ობიექტის ახალი ეგზემპლიარის შექმნას იმ შემთხვევაში იწვევს, თუ იმავე id და scope მნიშვნელობების მქონე ობიექტი აღარ არსებობს. წინააღმდეგ შემთხვევაში, არსებული ობიექტი გამოიყენება და ყველა jsp:setParameter ელემენტი იგნორირებული იქნება.

- **type** - მიუთითებს იმ ცვლადის ტიპზე, რომელიც ობიექტს მიმართავს. ის უნდა ემთხვეოდეს კლასის, სუპერ კლასის ან ინტერფეისის სახელს. ცვლადის სახელი id ატრიბუტით გადაიცემა.
- **beanName** – bean-ს აძლევს სახელს, რომელსაც instantiate() მეთოდი გამოიყენებს. შესაძლებელია type და beanName ატრიბუტების გამოყენება და class ატრიბუტის გამოტოვება.

jsp:setProperty მოქმედება. თქვენ შეგიძლიათ jsp:setProperty მოქმედება გამოიყენოთ იმისთვის, რომ ადრე აღწერილი Java ბინების (beans) თვისებებს მიანიჭოთ მნიშვნელობები. ამის განხორციელება ორი გზით არის შესაძლებელი. პირველ რიგში, თქვენ შეგიძლიათ გამოიყენოთ jsp:setProperty მოქმედება, ოღონდ jsp:useBean ელემენტის გარეთ ისე, როგორც ეს ნახ.351-ზეა წარმოდგენილი.

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName"
property="someProperty" ... />
```

ნახ. 351

ამ შემთხვევაში jsp:setProperty მოქმედება ნებისმიერ ვარიანტში შესრულდება, მიუხედავად იმისა, არსებული Java ბინი მოიძებნა თუ არა, ან შეიქმნა თუ არა მისი ახალი ეგზემპლიარი. მეორე ვარიანტი კი შემდეგში მდგომარეობს. თქვენ შეგიძლიათ jsp:setProperty მოქმედება განათავსოთ jsp:useBean ელემენტის ტანში, როგორც ეს ნახ.352-ზეა წარმოდგენილი:

```
<jsp:useBean id="myName" . . . >

...

<jsp:setProperty name="myName"

                 property=" myName" . . . />

</jsp:useBean>
```

ნახ. 352

ამასთან, jsp:setProperty მოქმედება მხოლოდ იმ შემთხვევაში სრულდება, თუ ობიექტის ახალი ეგზემპლიარია შექმნილი, და არა მაშინ, როდესაც არსებული ეგზეპლარი მოიძებნა.

jsp:setProperty მოქმედება შემდეგი ოთხი ატრიბუტის გამოყენების საშუალებას იძლევა:

- **name** - ეს აუცილებელი ატრიბუტი იმ Java ბინის მოსაცემად გამოიყენება, რომლის თვისებები შემდგომ დაყენდება. jsp:useBean ელემენტი წინ უნდა უსწრებდეს jsp:setProperty ელემენტის გამოყენებას.
- **property** - ეს აუცილებელი ატრიბუტი ჩვენთვის სასურველ (საჭირო) თვისებებს აყენებს. თუმცა არსებობს კერძო შემთხვევა: <<*>> მნიშვნელობა ნიშნავს, რომ მოთხოვნის ყველა პარამეტრი, რომელთა სახელები Java ბინის თვისებების სახელებს ემთხვევა, თვისებების დამყენებელ მეთოდს გადაეცემა.
- **value** - ეს არასავალდებულო ატრიბუტია, რომელიც თვისების მნიშვნელობას აყენებს. სტრი-ქონული მონაცემები ამ შემთხვევაში, სტანდარტული valueof() მეთოდის გამოყენებით ავტომატურად გარდაიქმნება სხვადასხვა ტიპის მნიშვნელობებად, კერძოდ, რიცხვით, boolean, Boolean, byte, Byte char და Character მნიშვნელობებად. მაგალითად, „true" მნიშვნელობა boolean ან Boolean თვისებებისთვის კონვერტირებას Boolean.valueOf() მეთოდის გამოყენებით განიცდის, ხოლო <<42>> მნიშვნელობა int და Integer თვისებებისთვის Integer.valueOf() მეთოდის საშუალებით იქნება კონვერტირებული. თქვენ შეგიძლიათ value და param ატრიბუტების ერთდროულად გამოყენებაც, თუმცა შეგიძლიათ საერთოდ არ გამოიყენოთ ისინი.
- **param** - ეს არასავალდებულო ატრიბუტი მოთხოვნის პარამეტრს აყენებს, რომელიც თვისების მისაღებად გამოიყენება. თუ მიმდინარე მოთხოვნა აღნიშნულ პარამეტრს არ შეიცვას, არანაირი მოქმედება არ სრულდება: სისტემა null მნიშვნელობას არ გადასცემს თვისებების დასაყენებლად არსებულ მეთოდს. ასეთ დროს, Java ბინში დასაშვებია სტანდარტული

თვისებების გამოყენება, რომელთა ხელახლა განსაზღვრას იმ შემთხვევაში ვახორციელებთ, თუ ამას მოთხოვნის პარამეტრები საჭიროებს და მოითხოვს. მაგალითად, 353-ე სურათზე წარმოდგენილი ფრაგმენტი ნიშნავს, რომ „საჭიროა დავაყენოთ numberOfItems თვისება, მოთხოვნის numItems პარამეტრის მნიშვნელობის შესაბამისად. თუ აღნიშნულ პარამეტრს მოთხოვნა არ შეიცავს, არანაირი მოქმედება არ სრულდება“.

```
<jsp:setProperty name="orderBean"
property="numberOfItems"
param="numItems" />
```

ნახ. 353

ნახ.354-ზე წარმოდგენილია Java ბინის გამოყენების კიდევ ერთი მაგალითი, სადაც ადგილი აქვს მარტივი რიცხვების ცხრილის შექმნას. თუ მოთხოვნის მონაცემებში არსებობს numDigits პარამეტრი, მაშინ ის bean numDigits თვისებას გადაეცემა. ანალოგიურია, numPrimes-ის შემთხვევაშიც.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title> JavaBeans-ის მრავალჯერადი გამოყენება JSP-ში</title>
</head>
<body>
<h1> JavaBeans-ის მრავალჯერადი გამოყენება JSP-ში</h1>
```

```
<jsp:useBean id="primeTable" class="hall.NumberedPrimes" />
<jsp:setProperty name="primeTable" property="numDigits" />
<jsp:setProperty name="primeTable" property="numPrimes" />
<p>
<jsp:getProperty name="primeTable" property="numDigits" />
<jsp:getProperty name="primeTable" property="numberedList"/>
 <p>
</body>
</html>
```

ნახ. 354

jsp:getProperty მოქმედება. აღნიშნული ელემენტი განსაზღვრავს Java ბინის თვისების მნიშვნელობას, ახდენს მის კონვერტირებას სტრიქონში და აგზავნის გამოტანის ნაკადში. მოქმედების შესასრულებლად საჭიროა ორი ატრიბუტის გამოყენება: Java ბინის სახელის, რაც jsp:useBean მოქმედებაში წინასწარ მოიცემა და თვისების სახელი, რომლის მნიშვნელობა აუცილებლად უნდა განისაზღვროს.

jsp:getProperty მოქმედების გამოყენების ნიმუში ნახ.355-ზეა წარმოდგენილი:

```
<jsp:useBean id="itemBean" ... />
...
<UL>
<LI>საგნების რაოდენობა:
<jsp:getProperty name="itemBean" property="numItems" />
<LI> ერთეულის ფასი:
<jsp:getProperty name="itemBean" property=" unitCost" />
</UL>
```

ნახ. 355

jsp:include მოქმედება. ეს მოქმედება უფლებას გვაძლევს ფაილების შიგთავსი გენერირებად ფურცელში ჩავსვათ. აღნიშნული მოქმედების ჩაწერის სინტაქსი შემდეგია:

```
<jsp:include page="შესაბამისი URL" flush="true" />
```

ნახ. 356

include დირექტივისგან განსახვავებით, რომელიც ფაილს JSP-ფუცლის ტრანსლაციის ეტაპზე სვამს, ეს მოქმედება ფაილის ჩასმას გვერდის მოთხოვნის პროცესში ახდენს. ეს ეფექტურობის გარკვეულ დანაკარგს იწვევს, მაგრამ თავად მოქმედება საკმაოდ მოქნილია.

ნახ. 357-ზე წარმოდგენილია ოთხი ფაილის შიგთავსის JSP-ფურცელზე ჩასმის მაგალითი.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>სიახლეები</title>
</head>
<body>
<h1>სიახლეები</h1>
<p>ოთხი ყველაზე პოპულარული სტატიის ფრაგმენტები:</p>
<ol>
<li><jsp:include page="news/Item1.html" flush="true"/></li>
<li><jsp:include page="news/Item2.html" flush="true"/></li>
<li><jsp:include page="news/Item3.html" flush="true"/></li>
<li><jsp:include page="news/Item4.html" flush="true"/></li>
</ol>
</body>
</html>
```

ნახ. 357

jsp:include მოქმედებისგან განსხვავებით, აქ აქტუალური გვერდის დამუშავება სრულდება. ის მხოლოდ ერთ page ატრიბუტს გამოიყენებს, რომელსაც შესაბამისი URL უნდა შეიცავდეს და რომლის საფუძველზეც request ობიექტი წესრიგშია. სხვა გვერდისათვის გადასაცემი მოთხოვნის პარამეტრებს jsp:param მოქმედების საშუალებით სხვა პარამეტრებიც შეგიძლიათ დაუმატოთ. page ატრიბუტის მნიშვნელობას შეიძლება წარმოადგენდეს როგორც სტატიკური მნიშვნელობა, ასევე მოთხოვნის პროცესში გამოთვლილი მნიშვნელობა, რაც ნახ. 358-ზეა ნაჩვენები:

```
<jsp:forward page="/utils/errorReporter.jsp" />

<jsp:forward page="="<%= რაიმე გამოსახულება Java-ზე %>" />
```

ნახ. 358

jsp:forward მოქმედების საშუალებით ასევე შესაძლებელია მართვის სხვა გვერდისთვის გადაცემა, მაგრამ მხოლოდ იმ პირობით, რომ აღნიშნული მოქმედების გამოძახებამდე გამომავალ ბუფერულ მეხსიერებაში არაფერი იყო ჩაწერილი. წინააღმდე შემთხვევაში ადგილი ექნება IllegalStateException გამონაკლისს.

jsp:param და jsp:params მოქმედებები. jsp:param მოქმედება გვაწვდის დასახელე-ბა/მნიშვნელობა ტიპის ინფორმაციას. მისი გამოყენება, ძირითადად, ჩვენთვის უკვე ნაცნობ jsp:include და jsp:forward მოქმედებებთან ერთად ხდება. ასევე შესაძლებელია jsp:plugin მოქმდებასთან ერთად გამოყენებაც. სხვა შემთხვევებში მის გამოყენებას არ აქვს მნიშვნელობა. jsp:param მოქმედების jsp:include და jsp:forward მოქმედებებთან ერთად გამოყენება ახალ გვერდებს საწყის request ობიექტს გადასცემს, რომელიც ახალი პარამეტრების ხარჯზე

ფართოვდება. თუ თქვენ არსებულ პარამეტრებს ახალ მნიშვნელობებს მისცემთ, მაშინ სწორედ ამ ახალ მნიშვნელობებს ექნებათ უპირატესობა. jsp:params მოქმედებით ერთდროულად რამდენიმე პარამეტრის გადაცემაა შესაძლებელი.

jsp:plugin მოქმედება. აღნიშნული მოქმედება უფლებას გაძლევთ OBJECT ან EMBED ელემენტის (ეს გამოყენებული ბრაუზერის ტიპზეა დამოკიდებული) ჩასმა განახორციელოთ, რაც აუცილებელია იმ აპლეტების გასაშვებად, რომლებიც plugin Java-ს იყენებენ. სხვა სიტყვებით რომ ვთქვათ, აღნიშნული მოქმედება HTML-ის გენერირებას ემსახურება, რათა Java API-ის ჩადგმა JSP-ფურცელში მოხდეს. ამგვარად, შეძლებთ URL-ის ჩადგმას იმისთვის, რომ JavaSoft-დან Java API-ის მოდულები გადმოწეროთ. აღნიშნული მოქმედების ჩაწერის ფორმას შემდეგი სახე აქვს:

```
<jsp: plugin
type="bean|applet"
code="კლასის ფაილი"
codebase="ობიექტი CodeBase"
align="ადგილმდებარეობა"
archive="არქივების სია"
height="სიმაღლე"
hspace="ჰორიზონტალური არე"
jreversion="ვერსია"
name="კომპონენტის სახელი"
vspace="ვერტიკალური არე"
width="სიგანე"
nspluginurl="url"
iepluginurl="url">
<jsp:params>
<jsp:param name="დასახელება1" value="მნიშვნელობა1" />
<jsp:param name="დასახელება2" value="მნიშვნელობა2" />
...
<jsp:param name="დასახელებაN" value="მნიშვნელობაN" />
</jsp:params>
<jsp:fallback> </jsp:fallback>
</jsp:plugin>
```

ნახ. 359

ზემოთ აღნიშნული მოქმედების გამოყენება ახლა აპლეტის კოდში ვნახოთ:

```
<jsp:plugin type="applet" code="Blink.class" width=300
height=100>
<jsp:params>
<jsp:param name=lbl value="ეს მართლაც გემრიელია!" />
<jsp:param name=speed value="4" />
<jsp:params>
<jsp:fallback>თქვენი ბრაუზერი გაურკვეველი მიზეზების გამო ვერ უშვებს
ამ აპლეტს </fallback>
</jsp:plugin>
```

ნახ. 360

კითხვები თვითშემოწმებისთვის:

1. მოკლედ დაახასიათეთ Java Beans კომპონენტი.
2. რა უპირატესობებით ხასიათდება Java Beans კომპონენტების ტექნოლოგია?
3. რას წარმოადგენს Java Beans კომპონენტის თვისება (property)?
4. რაში გამოიხატება Java Beans კომპონენტის მუდმივობა?
5. რას წარმოადგენს jsp:useBean მოქმედება?
6. რა დანიშნულება აქვს scope ატრიბუტს?
7. რა დანიშნულება აქვს jsp:setProperty მოქმედებას?
8. მოკლედ დაახასიათეთ jsp:getProperty მოქმედება.
9. რა განსხვავებაა include დირექტივისა და jsp:include მოქმედებას შორის?
10. მოკლედ დაახასიათეთ jsp:plugin მოქმედება.

დავალება:

1. შეადგინეთ Bean-ის ჩატვირთვის მარტივი მაგალითი, სადაც ადგილი აქვს მარტივი სტრიქონული პარამეტრის მიღებას.
2. შეადგინეთ Java ბინის გამოყენების მაგალითი, სადაც ადგილი ექნება წილადრიცხვა მონაცემების ცხრილის შექმნას.
3. შეადგინეთ ნებისმიერი სამი ფაილის შიგთავსის JSP-ფურცელზე ჩასმის პროგრამა.
4. შეადგინეთ პროგრამული კოდი, რომელიც Java API-ის ჩადგმას განახორციელებს JSP-ფურცელში.

JSP-ფურცლებთან დაკავშირებული სრული ვიდეო მასალა (179 ვიდეო-ფაილი) შეგიძლიათ იხილოთ ბმულზე:

https://www.youtube.com/watch?v=eKxKMa_AGaw&index=2&list=PLa3RgLJMYWO-UhrjkUHC-MHqBULpCTMI5

7. უნიფიცირებული პროგრამირების ენა (UML)

7.1. ვიზუალური მოდელირება

უკანასკნელ პერიოდში დიდი ყურადღება ეთმობა რთული პროგრამული უზრუნველყოფის შექმნას.

ამ სისტემების სირთულემ განაპირობა მოდელირების „კარგი“ მეთოდების გამოყენება. მოდელირების ენა უნდა შეიცავდეს:

- მოდელის ელემენტებს (ძირითადი კონცეფციები და მათი სემანტიკა);
- ნოტაციებს (მოდელირების ელემენტების ვიზუალიზაცია);
- გამოყენების პრინციპებს (წესები).

ვიზუალური მოდელების აგება გვაძლევს ურთულეს პროექტებთან და სისტემებთან მუშაობის საშუალებას, ვიზუალიზაცია კი - დამკვეთის და შემსრულებლის ურთიერთობის შემსუბუქებას და პროექტებზე დინამიკაში მუშაობის უზრუნველყოფას.

90-ანი წლების დასაწყისში არსებობდა 50-ზე მეტი ობიექტზე ორიენტირებული მოდელირების ენა. საკმაოდ რთული იყო ერთ-ერთის არჩევა, საერთო კრიტერიუმები არ არსებობდა. ამ დროისთვის უკვე არსებობდა: Booch’93, OMT-2 (Object Modelling Technique), Fusion, OOSE (Object-Oriented Software Engineering). დადგა მათი სტანდარტიზაციისა და უნიფიკაციის აუცილებლობა.

1994 წელს გრედი ბუჩმა და ჯიმ რამბომ UML -ზე (Rational Software Corporation -კომპანია) Booch'93 и OMT მეთოდოლოგიების უნიფიცირება მოახდინეს.

UML-ის შემადგენელი ნაწილებია:

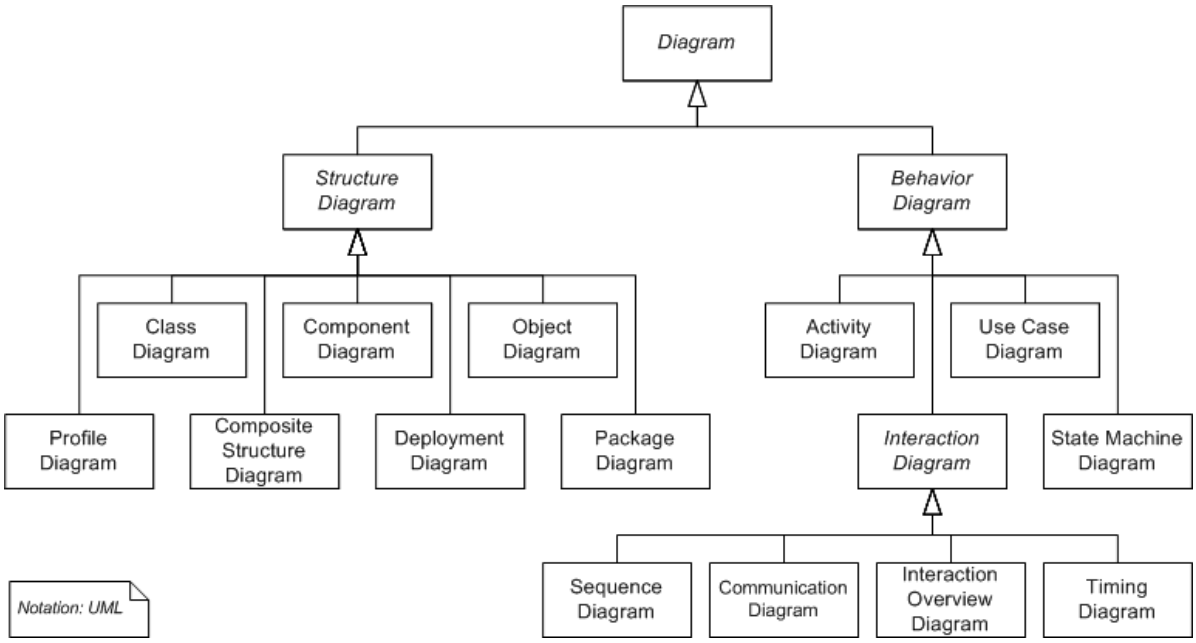
1. UML Summary,
2. UML Notation Guide,
3. UML Semantics,
4. UML OCL (Object Constraint Language Specification),
5. UML Objectory (UML Extension for Objectory Process for Software Engineering),
6. UML Business (UML Extension for Business Modeling),
7. UML Metamodel_Diagram

7.1.1. დიაგრამებთან მუშაობა

UML-ში გამოყენებულია დიაგრამების შემდეგი სახეობები:

Structure Diagrams:
• Class diagram
• Component diagram
• Composite structure diagram
  • Collaboration (UML2.0)
• Deployment diagram
• Object diagram
• Package diagram
• Profile diagram (UML2.2)
Behavior Diagrams:
• Activity diagram
• State Machine diagram
•
• Use case diagram
• **Interaction Diagrams:**
  • Communication diagram (UML2.0) / Collaboration (UML1.x)
  • Interaction overview diagram (UML2.0)
  • Sequence diagram
  • Timing diagram (UML2.0)
	სტრუქტურული დიაგრამები:
	• კლასების დიაგრამა
	• კომპონენტების დიაგრამა
	• კომპოზიტური/შედგენილი სტრუქტურის დიაგრამა
	• კოოპერაციის დიაგრამა (UML2.0)
	• განშლადი დიაგრამები
	• ობიექტების დიაგრამა
	• პაკეტების დიაგრამა
	• პროფილების დიაგრამა (UML2.2)
	ქცევის დიაგრამები:
	• საქმიანობის დიაგრამა
	• ავტომატის (საბოლოო მდგომარეობების) დიაგრამა
	• გამოყენების ვარიანტების დიაგრამა
	• **ურთიერთკავშირების დიაგრამები:**
	  • კომუნიკაციების დიაგრამა (UML2.0) / კოოპერაციის დიაგრამა (UML1.x)
	  • ურთიერთკავშირების მიმოხილვის დიაგ-რამა (UML2.0)
	  • თანამიმდევრობების დიაგრამა
	  • სინქრონიზაციის დიაგრამა (UML2.0)

UML 2.3 დიაგრამების სტრუქტურა მოცემულია ნახ. 361-ზე

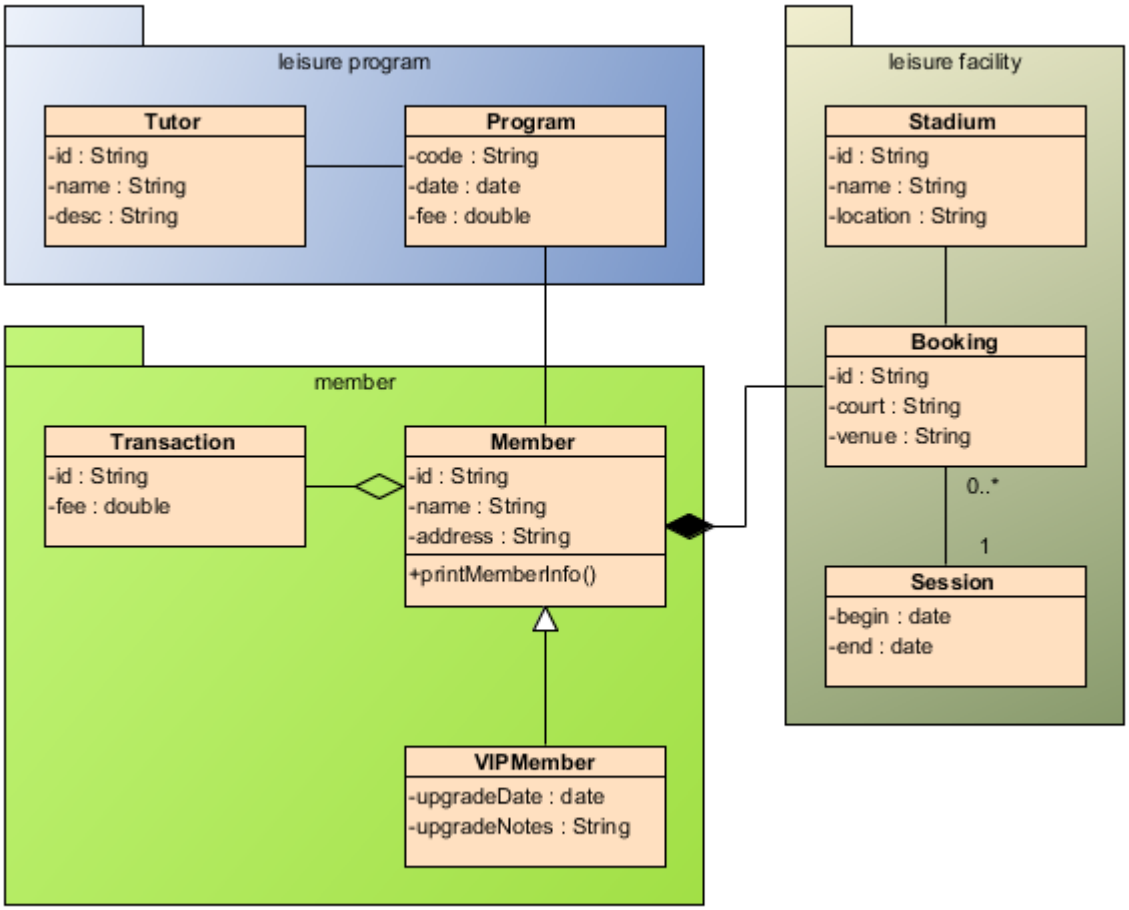


ნახ. 361

კლასების დიაგრამა

დიაგრამა არის სტატიკური და აღწერს სისტემის სტრუქტურას, მის კლასებს და ატრიბუტებს, მეთოდებს და დამოკიდებულებებს. არსებობს დიაგრამების აგების რამდენიმე ხერხი:

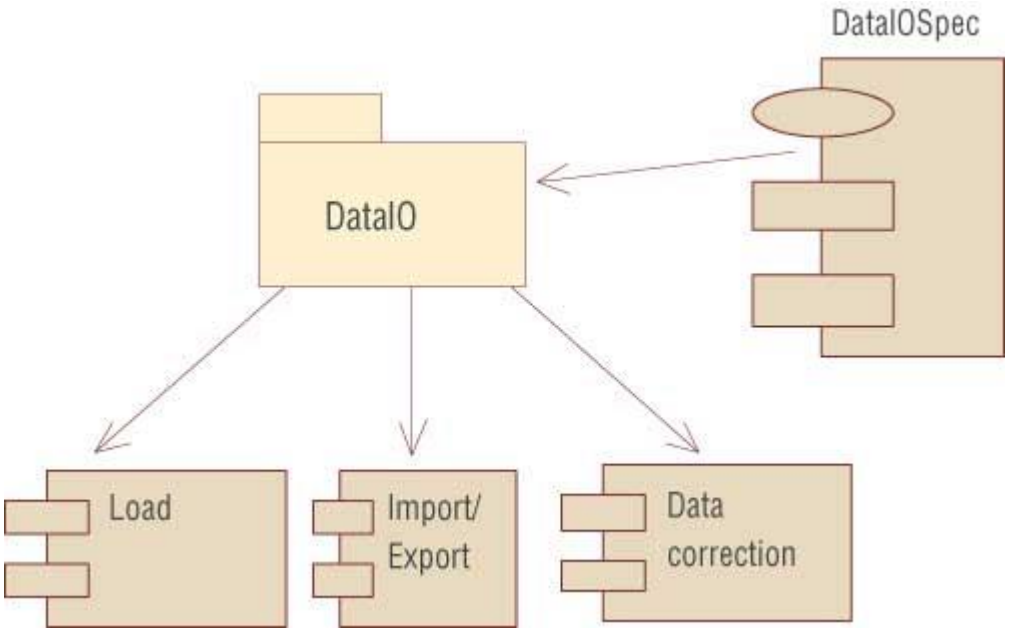
- კონცეპტუალური - დიაგრამა აღწერს ობიექტის მოდელს და შეიცავს მხოლოდ ობიექტის კლასებს;
- სპეციფიკური - დიაგრამა გამოიყენება საინფორმაციო სისტემების დაგეგმვისათვის;
- სარეალიზაციო - დიაგრამა შეიცავს პროგრამულ კოდებში აღწერილ მოდელებს.



ნახ. 362

კომპონენტების დიაგრამა

ეს სტატიკური დიაგრამა გვიჩვენებს პროგრამული სტრუქტურის დაყოფას სტრუქტურულ კომპონენტებად და ასახავს კომპონენტთა შორის კავშირებს. ფიზიკურ კომპონენტებს წარმოადგენენ ფაილები, მოდულები, პაკეტები და ა.შ.



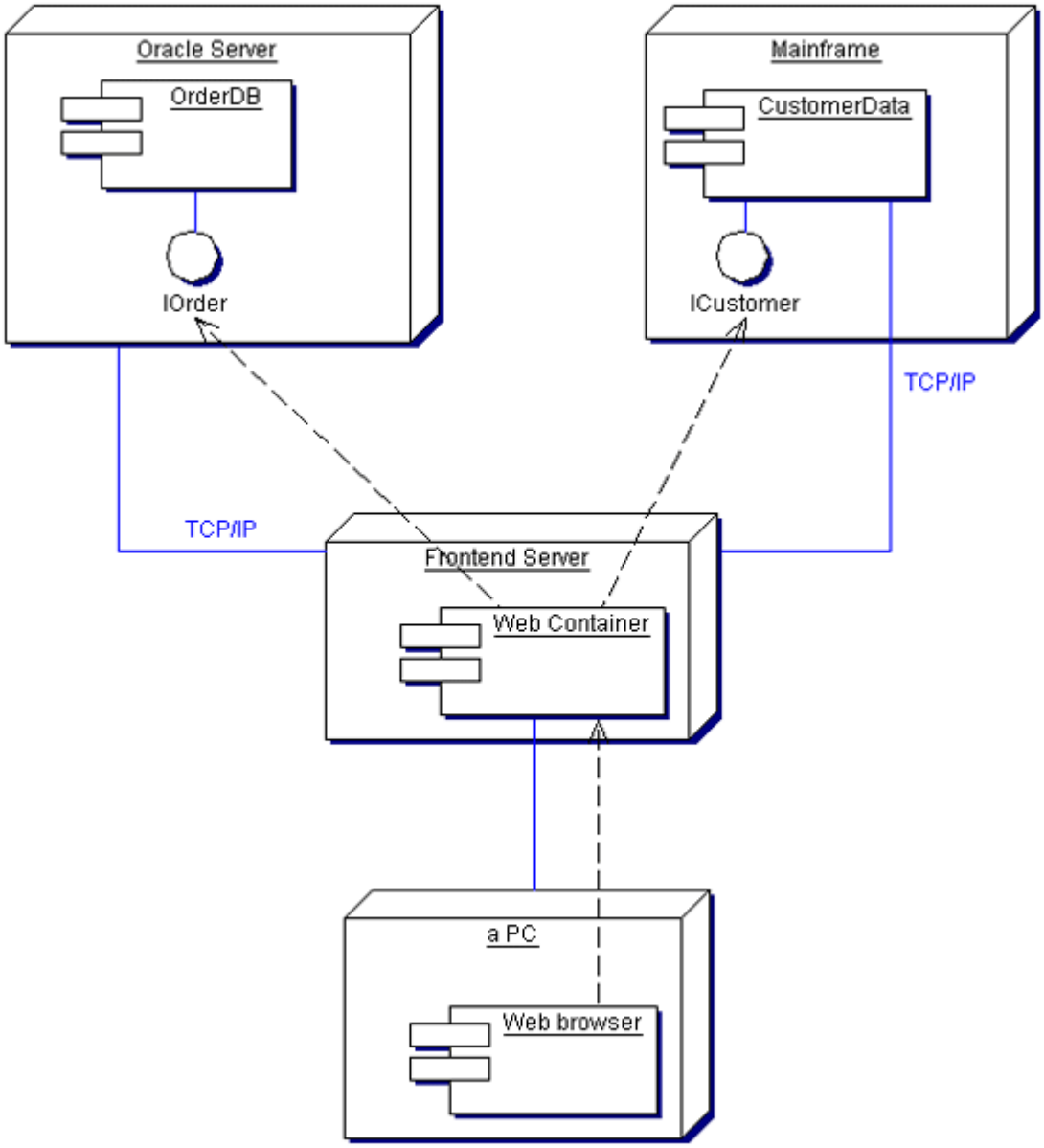
ნახ. 363

კომპოზიტური/შედგენილი სტრუქტურის დიაგრამა

ეს სტატიკური დიაგრამა გვიჩვენებს კლასების შიდა სტრუქტურას და მათი ელემენტების ურთიერთქმედებას. კოოპერაციის დიაგრამები წარმოადგენს კომპოზიტური დიაგრამების სახეობას, სადაც ნაჩვენებია კლასების როლი და ურთიერთკავშირი კოოპერაციის ფარგლებში. კოოპერაციები გამოიყენება დაგეგმვის შაბლონების შექმნის დროს.

განშლადი დიაგრამა

გამოიყენება მომუშავე კვანძების და არტეფაქტების მოდელირებისათვის (აპარატურის კავშირების დადგენა)



ნახ. 364

ობიექტების დიაგრამა

გვიჩვენებს მოდელის სრულ ან ნაწილობრივ სურათს დროის გარკვეულ მონაკვეთში. დიაგრამაზე გამოისახება კლასების ობიექტების მიმდინარე მნიშვნელობები.

პაკეტების დიაგრამა

აქ ნაჩვენებია სხვადასხვა სტრუქტურების თემატიკურ ჯგუფებში გაერთიანება.

საქმიანობის დიაგრამა

ამ დიაგრამაში ნაჩვენებია გარკვეული საქმიანობის შემადგენელ ნაწილებად დაშლა. საქმიანობის ქვეშ იგულისხმება ბიზნეს-, ტექნოლოგიური ან გამოთვლითი პროცესების თანა-მიმდევრული ან პარალელური აღსრულება.

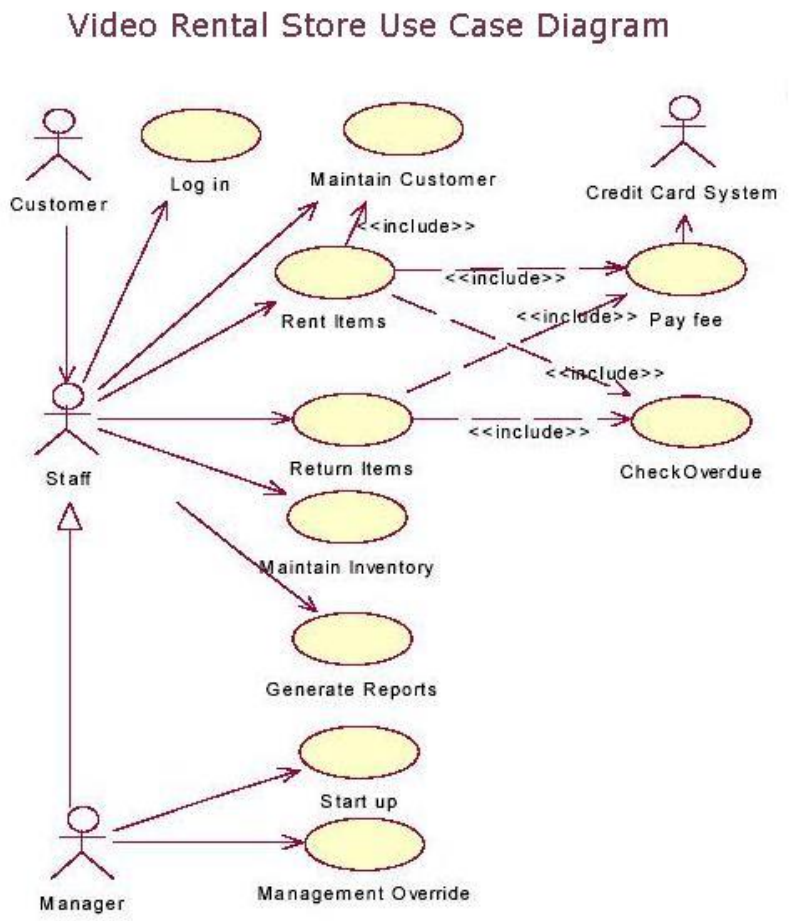
ავტომატის დიაგრამა

ამ დიაგრამაზე ასახულია დასრულებული ავტომატი მარტივი კომპონენტებით და კავშირებით. ავტომატი დაკავშირებულია საწყის ელემენტთან (კლასთან, კოოპერაციასთან ან მეთოდთან) და გამოიყენება თავისი ეგზემპლიარების ქცევის დასადგენად.

დიაგრამ-ავტომატების ანალოგია დრაკონ-სქემა.

გამოყენების ვარიანტების დიაგრამა

ეს დიაგრამა გვიჩვენებს „მსახიობებსა“ და ვარიანტებს შორის ურთიერთობებს. ძირითადი მიზანია დამკვეთმა და პროგრამისტმა ერთად იმსჯელონ სისტემის ფუნ-ქციონირებაზე და პარალელურად დინამიკაში შეიტანონ ცვლილებები უკვე დაპროექტებულ სისტემაში.

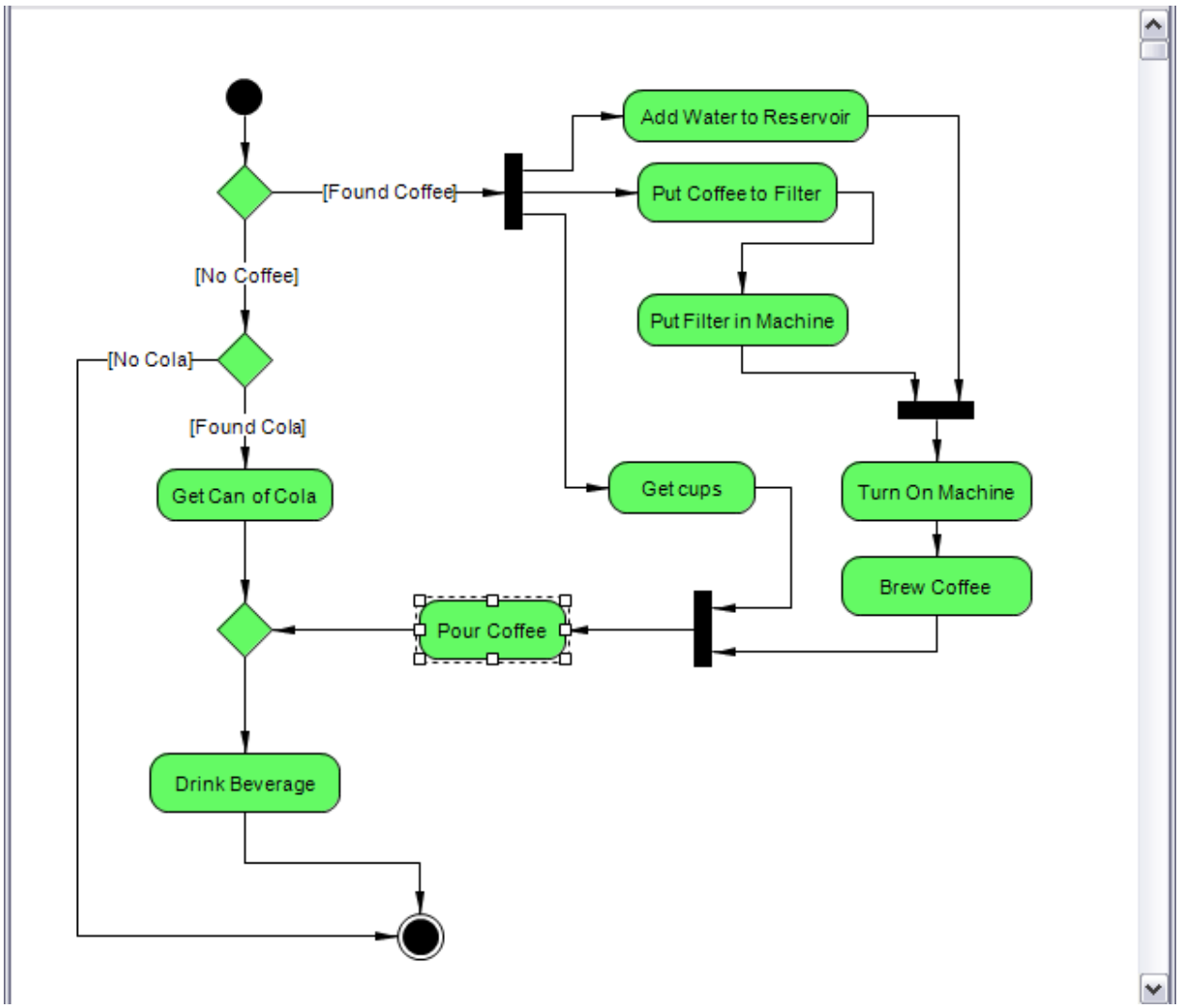


ნახ. 365

კომუნიკაციისა და თანამიმდევრობების დიაგრამა

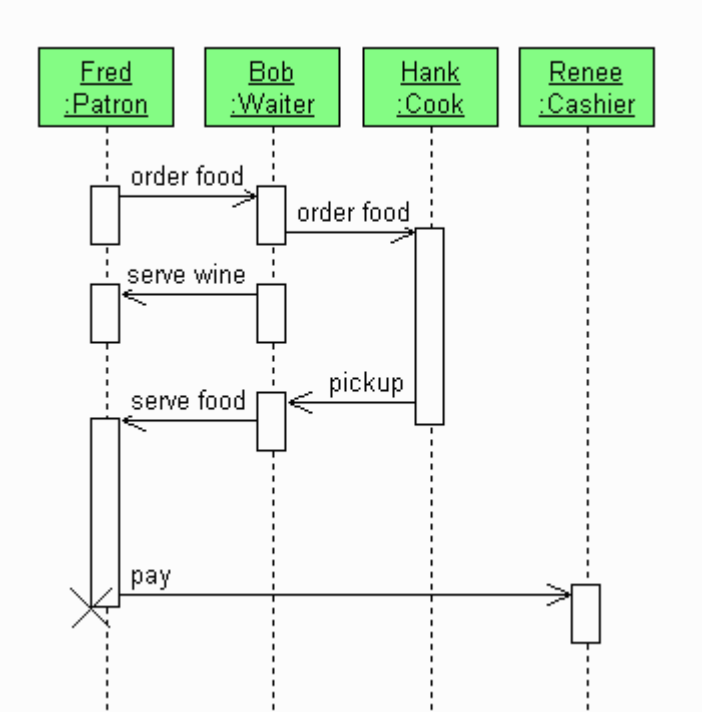
ეს დიაგრამები არა სტატიკური, არამედ ტრანზიტიულებია და ასახავენ ურთიერთქმედებებს.

კომუნიკაციის დიაგრამაზე ნაჩვენებია კომპოზიტური სტრუქტურების ურთიერთობა. აქ თვალნათლივ ჩანს ელემენტებს შორის კავშირი, ხოლო დროის ფაქტორი არ არის გათვალისწინებული.



ნახ. 366

თანამიმდევრობების დიაგრამაზე ნაჩვენებია ობიექტების ურთიერთობა დროის მიხედვით. მაგალითად, შეტყობინებების თანამიმდევრობები.



ნახ. 367

ურთიერთკავშირების მიმოხილვის დიაგრამა

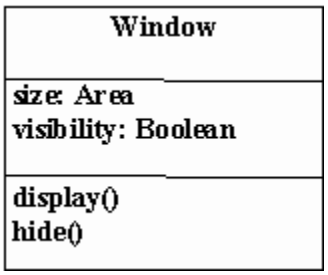
ეს დიაგრამები შეიცავენ თანამიმდევრობების და თანამშრომლობის დიაგრამებს და ასახავენ ობიექტების ურთიერთქმედებას მოდელირებად სისტემაში.

სინქრონიზაციის დიაგრამა

გვიჩვენებს სისტემის მდგომარეობის ცვლას რეალურ დროში.

7.2. ობიექტები და კლასები

ობიექტების ერთობლიობას, რომლებსაც საერთო სტრუქტურა, კავშირი და მოქმედება აქვს ეწოდება **კლასი**. დიაგრამებზე კლასი აისახება მართკუთხედით ჰორიზონტალურად დაყოფილ 3 ნაწილად:



ნახ. 368

ზედა სექციაში იწერება კლასის სახელი და სხვა საერთო თვისება (კლასის ტიპი). შუა სექციაში იწერება ატრიბუტების სია, ხოლო ქვედა სექციაში კი - ოპერაციების თანამიმდევრობა. ატრიბუტი აღწერს კლასის მონაცემებს, ოპერაციები კი - კლასის ობიექტების მოქმედებებს. სექცია შეიძლება იყოს ცარიელი.

კლასების გაერთიანება შეიძლება პაკეტებში. კლასების სახელი უნდა იყოს უნიკალური. თუ საჭიროა სხვა პაკეტში შემავალი კლასის გამოძახება, პაკეტის სახელი აუცილებლად უნდა იყოს მოხსენებული.

<პაკეტის სახელი>::<კლასის სახელი>

აქ აუცილებლად უნდა დაცული იყოს იერარქია:

<პაკეტი1>::<პაკეტი2>...<პაკეტი N::<კლასის სახელი>

კლასის სახელის სექციაში შეიძლება იყოს:

- კლასის ტიპის აღწერა;
- კლასის სახელი;
- დამატებითი თვისებები.

სხვა სექციების შევსება კლასის აღწერის დროს შეიძლება გამოვტოვოთ. თუ კი ზუსტად ვიცით, რას უნდა აკეთებდეს კლასი, მაშინ შეგვიძლია შევავსოთ ატრიბუტიც და ოპერაციებიც.

ატრიბუტი - კლასის ელემენტია, რომელსაც გააჩნია მარტივი ან რთული ტიპი:

CArray<CString *, CPoint *>

ტიპის აღწერილობა დამოკიდებულია დაპროგრამების ენაზე. ატრიბუტის მაგალითი:

<ხედვის ნიშანი><სახელი>::<ტიპი>=<მნიშვნელობა> {თვისება}

- ხედვის ნიშანს გააჩნია კლასის ხედვის C++ სემანტიკა.
- ღია ატრიბუტი (public) ნიშნავს, იმას, რომ ნებისმიერ ობიექტზე მიმართვისას, ვიყენებთ მის ატრიბუტსაც.
- დაცული ატრიბუტი (protected #) ნიშნავს იმას, რომ მას მიმართავენ მხოლოდ მისი კლასის მეთოდები და მემკვიდრეები.
- დახურული ატრიბუტი (private) (-) ნიშნავს, რომ ის მისაწვდომია მხოლოდ კლასის მეთოდისთვის.
- ხედვის არე ინიშნება სიტყვებით „public“, “private”, “protected”, ან საერთოდ არ არის მითითებული (იგულისხმება public).
- სახელი - ატრიბუტის სახელი.
- ტიპი - დაპროგრამების ენაზე დამოკიდებული ატრიბუტის ტიპი.
- თვისებები - დამატებითი ინფორმაცია (აუცილებელი არ არის). იწერება { }.

{Author = Smith}

თუ თვისებებში მოვნიშნავთ {frozen}, ატრიბუტის შეცვლა შეუძლებელია. ატრიბუტში მოინიშნება რაოდენობრიობაც:

coords[3]: integer

მოქმედებას, რომელსაც ახორციელებს კლასისი წარმომადგენელი, ეწოდება **ოპერაცია.** ოპერაციას გააჩნია სახელი და არგუმენტების სია. ოპერაცია წარმოადგენს ტექსტურ სტრიქონს:

<ხედვის ნიშანი><სახელი>(პარამეტრების სია):<გამოსახულების შედეგის ტიპი>{თვისებები}

კლასის ყველა ოპერაცია იყოფა ორ ჯგუფად: კლასის ოპერაციები და წარმომადგენლის ოპერაციები. კლასის ოპერაციები ხორციელდება მხოლოდ მთლიან კლასზე და არა მის შემადგენელ ნაწილებზე, ამიტომ კლასის ოპერაციებს არა აქვთ წვდომა ობიექტების ატრიბუტებზე. მაგალითად, კლასის ოპერაციაა - ახალი ობიექტის შექმნა. კლასის ოპერაციები მოინიშნებიან ხაზგასმით.

createObject(void): PObject

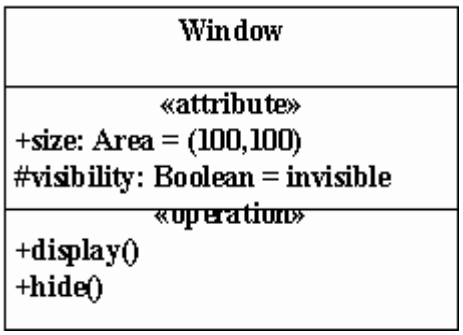
ოპერაცია, რომელიც არ ცვლის სისტემის მდგომარეობას მოინიშნება სიტყვა {query}. ატრიბუტების სიის ელემენტების დაჯგუფება ხდება სხვადასხვა თვისებების მიხედვით, ამიტომ ელემენტების ჯგუფის წინ იწერება **„თვისება“**. მაგალითად:

```
"constructors"
CRect(CPoint left_up, CPoint right_down)
CRect(left:Integer, top:Integer, right:Integer, bottom:Integer)
"Data access"
GetLeftUp(): CPoint
SetLeftUp(CPoint lup)
GetRightDown():CPoint
SetRightDown (CPoint rdn)
...
```

ერთ სიასი შეიძლება რამოდენიმე თვისება იყოს წარმოდგენილი:

```
"Get-Data functions"
GetLeftUp(): CPoint
SetLeftUp(CPoint lup)
"Set-Data functions"
GetRightDown():CPoint
SetRightDown (CPoint rdn)
...
```

კლასის აღწერაში თითოეულ სექციას შეიძლება ჰქონდეს სახელი:



ნახ. 369

ობიექტი

ობიექტზე ორიენტირებული დაპროგრამების ერთ-ერთი მნიშვნელოვანი ცნება არის **ობიექტის** (object) ცნება. ობიექტი წარმოადგენს კლასის ეგზემპლიარს და მას გააჩნია

მდგომარეობები, განსაზღვრული ქცევები და სტრუქტურა. საერთო სტრუქტურის ობიექტები ერთიანდებიან კლასში.

ობიექტები ასრულებენ განსაზღვრულ როლებს. როლი კი განსაზღვრავს კლასის და ეგზემპლიარის ურთიერთობას. ითვლება, რომ ყველა ობიექტი ჰგავს ერთმანეთს ქცევით და მდგომარეობით.

დიაგრამაზე ობიექტი წარმოადგენს ორსექციან მართკუთხედს. ზედა სექცია შეიცავს ობიექტის და კლასის სახელს, რომელიც გახაზულია უწყვეტი ხაზით:

<ობიექტის სახელი>:<კლასის სახელი>

კლასის სახელი შეიძლება შეიცავდეს სრულ გზას. პაკეტების სახელები იწერება კლასის სახელის წინ:

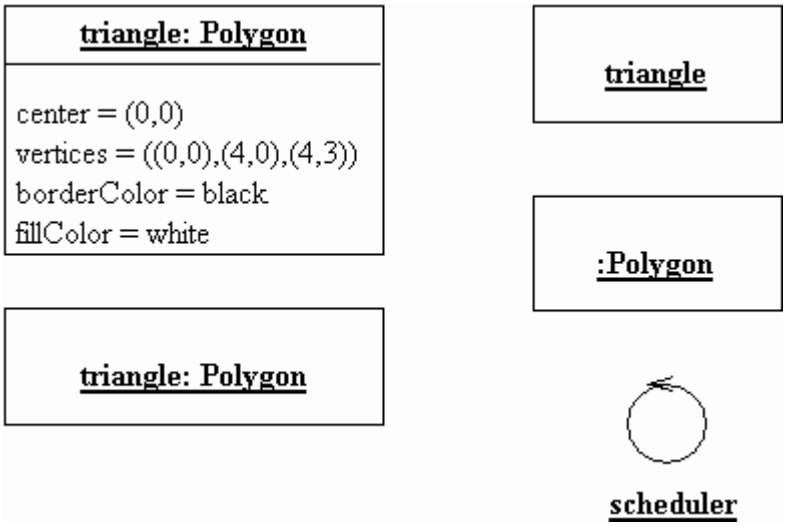
display_window : WindowingSystem : : GraphicWindows : : Window

ობიექტს შეიძლება არც ჰქონდეს სახელი. ამ შემთხვევაში პირველ სექციაში იწერება მხოლოდ კლასის სახელი და „:“. კლასის სახელიც შეიძლება არ დაიწეროს.

მეორე სექცია შეიცავს ატრიბუტებს, ტიპებს და მნიშვნელობებს:

<ატრიბუტის სახელი>:<ტიპი>=<მნიშვნელობა>

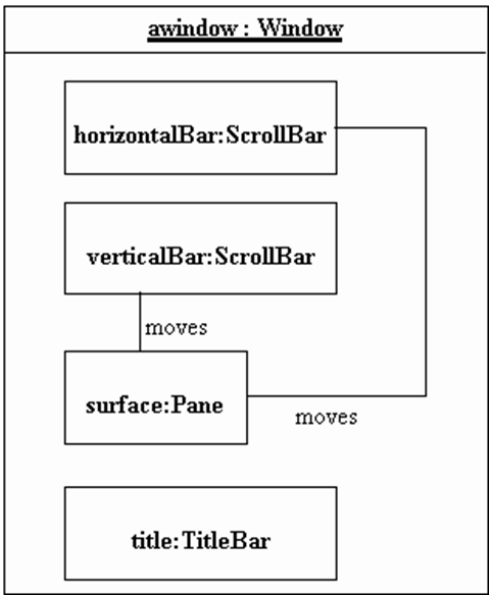
ატრიბუტის ტიპის აღწერა არ არის სავალდებულო. შესაძლებელია ობიექტის მდგომარეობის აღწერაც, ამისათვის გამოიყენება „[ ]“. პროცესის მსვლელობისას შესაძლებელია მდგომარეობების კორექტირება და შეცვლა.



ნახ. 370

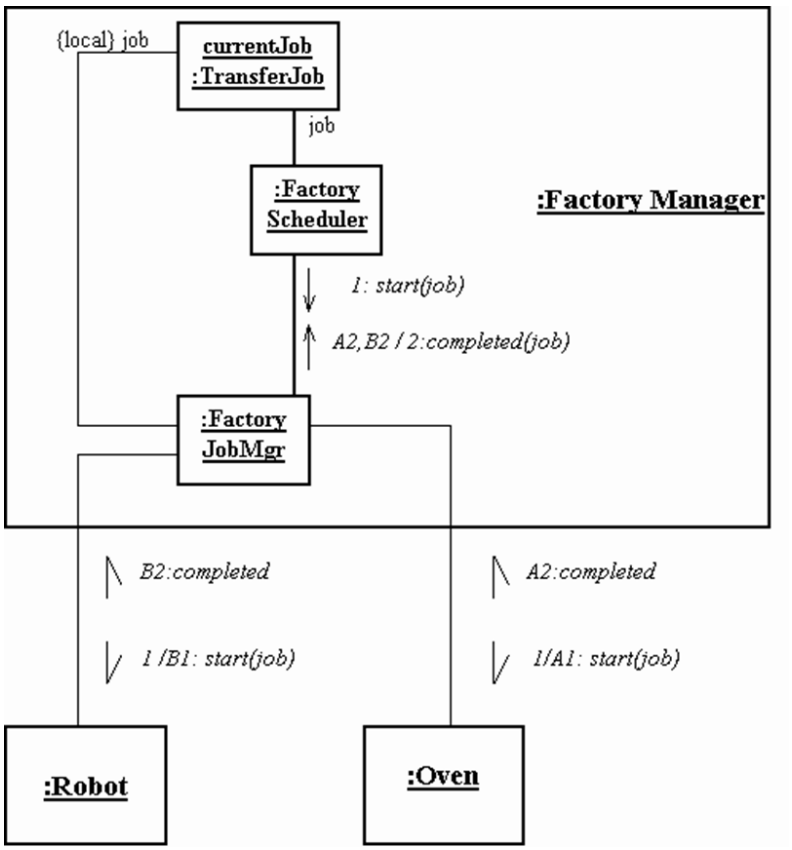
ობიექტი შეიძლება იყოს **შედგენილი (compose object).** ასეთი ობიექტი შედგენილი კლასის ნაწილია, ანუ კლასის, რომელიც შედის კომპოზიციაში. შედგენილი ობიექტის აღწერა

წააგავს მარტივი ობიექტის აღწერას, მხოლოდ ქვედა სექციაში ატრიბუტების ნაცვლად აღიწერება შედგენილი ობიექტის ნაწილები.



ნახ. 371

აქტიურ ობიექტს შეუძლია მოქმედების ინიცირება. **პასიური** ობიექტი შეიცავს მონაცემებს და არა მოქმედებებს, მაგრამ შეუძლია შეტყობინებების გაგზავნა მიღებული მოთხოვნის მიხედვით. აქტიურ ობიექტს გააჩნია მართვის ნაკადი. დიაგრამაზე ის მოინიშნება სქელი უწყვეტი ხაზით:



ნახ. 372

UML-ში არსებობს კლასის რამოდენიმე სახესხვაობა: ინტერფეისი, შაბლონი, უტილიტა და ა.შ.

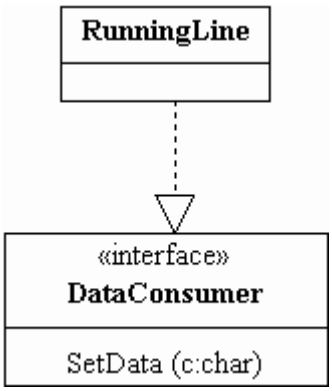
7.2.1. ინტერფეისები

ინტერფეისი (interface) - კლასი, რომელიც ავრცელებს ოპერაციებს, მაგრამ არ შეიცავს ამ ოპერაციების ველებს და რეალიზაციებს. ინტერფეისის კლასი თვითონ განსაზღვრავს ამ ოპერაციების შინაარსს.

შაბლონი (Template) - იგივე პარამეტრებიანი კლასი წარმოადგენს კლასების ერთობლიობას, რომლებიც განსხვავდებიან ერთმანეთისგან ზოგიერთი ფორმალური პარამეტრით. **უტილიტა (utility)** - ეს არის გლობალური ცვლადების და პროცედურების კლასი.

UML-ში **ინტერფეისი** წარმოადგენს ფუნქციათა ჯგუფის აღწერილობას, რომელიც საჭიროა სხვა კლასებთან დასაკავშირებლად. ამ ფუნქციების ლოგიკა არ განისაზღვრება, აქ მხოლოდ არაფორმალურ ენაზე ხდება ფუნქციების აღწერა.

კლასი ახდენს ინტერფეისის რეალიზაციას იმ შემთხვევაში, თუ ის შეიცავს ინტერფეისის ყველა მეთოდს და ოპერაციას. დიაგრამაზე UML-ინტერფეისი აისახება ორნაირად: გაშლილად და შეკვეცილად. გაშლილ მდგომარეობაში ინტერფეისი აისახება ატრიბუტის გარეშე, როგორც კლასი სტერეოტიპით „interface“:



ნახ. 373

ნახ.373-ზე მოცემულია კლასი RunningLine, რომელიც აღწერს DataConsumer ინტერფეისს. მათ შორის კავშირს ეწოდება დეტალიზაცია და დიაგრამაზე წყვეტილი ხაზითაა ნაჩვენები. ასე რომ RunningLine კლასი გვიჩვენებს მეთოდს, რომელიც ასრულებს DataConsumer-ით გამოგზავნილ SetData ოპერაციას.

ობიექტზე ორიენტირებულ დაპროგრამების ენებში ინტერფეისების გამოყენება პოლიმორფიზმის უზრუნველყოფის ერთ-ერთი ვარიანტია. ერთი და იგივე ინტერფეისის ქვეშ მომუშავე კლასი ერთნაირად იქცევა, რაც განზოგადოებული ალგორითმების შედგენის საშუალებას იძლევა. მეთოდი იყენებს ინტერფეისში აღწერილ კოდებს და არ იცის კლასის შემადგენლობის შესახებ. ამიტომ ინტერფეისების გამოყენება ყოფს პროგრამულ სივრცეს დამოუკიდებელ მოდულებად, რაც ამცირებს შეცდომების არსებობას.

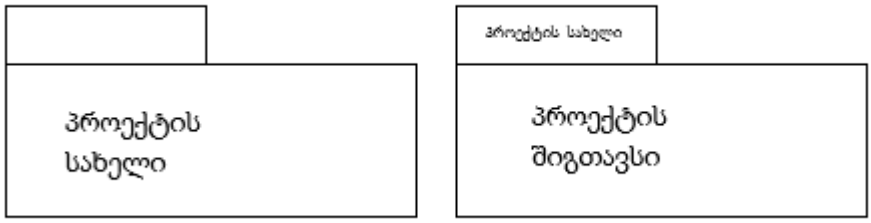
ინტერფეისი შეიძლება შეიცავდეს რამდენიმე ინტერფეის-ჩანართებს. და ის ინფორ-მაცია, რომელიც ჩაწერილია ინტერფეის-წინაპარში, ავტომატურად გამოიყენება ინტერფეის-მემკვიდრეშიც. აქ არის ერთი პრობლემა - მრავლობითი ინტერფეისის დროს ერთნაირი მეთოდების გამოყენება შეუძლებელია. ამისათვის მიმართავენ შემდეგ მეთოდებს:

- **აკრძალვა.** ერთ კლასში აიკრძალება ერთნაირი სიგნატურების მქონე რამდენიმე ინტერფეისის გამოყენება;
- ერთნაირი მეთოდების რეალიზაცია ხდება ცალ-ცალკე, და გამოძახება ხდება სახელის მიხედვით;
- **ერთგვაროვანი მეთოდების ზოგადი რეალიზაცია**. თუ რამდენიმე ერთნაირი სიგნატურის მეთოდის რეალიზება ხდება, ისინი ერთიანდებიან ინტერფეის-მემკვიდრეში და ამით კლას-რეალიზატორში ექმნებათ ერთნაირი რეალიზაცია.

7.3. UML-პაკეტები

პაკეტი წარმოადგენს მოდელის ელემენტების ორგანიზების ძირითად მეთოდს. თითოეული ელემენტი მიეკუთვნება მხოლოდ ერთ პაკეტს. პაკეტები კი შეიძლება იყოს შედგენილი და შეიცავდეს ქვე-პაკეტებს. ამით იქმნება პაკეტების იერარქია.

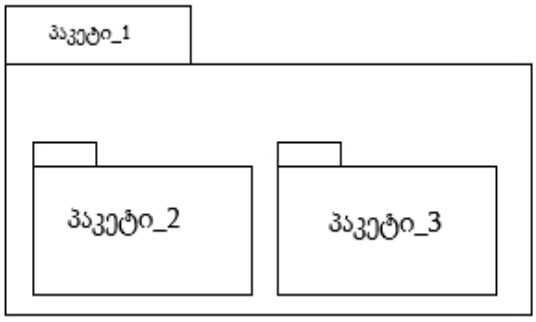
პაკეტების ვიზუალიზაციისთვის შემუშავებულია სპეციალური სიმბოლიკა. პაკეტი დიაგრამებში აისახება დიდი მართკუთხედის ზემოთ პატარა მართკუთხედით (საქაღალდის პიქტოგრამის ასოციაცია). დიდი მართკუთხედის შიგნით იწერება ინფორმაცია პაკეტზე ან პაკეტის სახელი, რომელიც პროექტის ფარგლებში უნდა იყოს უნიკალური. თუ ორივეს ჩაწერაა აუცილებელი, მაშინ სახელი იწერება პატარა მართკუთხედში:



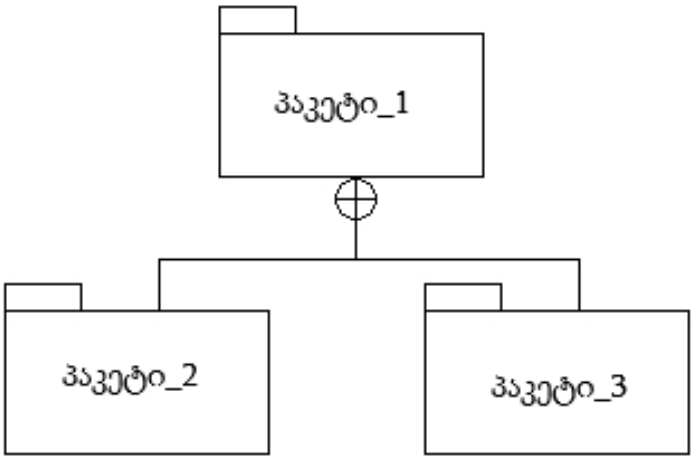
ნახ. 5674

პაკეტის სახელის წინ შეიძლება იყოს ჩაწერილი გასაღები სიტყვა, რომლების UML-ში წინასწარ განსაზღვრული არის და ეწოდება **სტერეოტიპი**. მაგალითად, **facade, framework, stub и topLevel.**

მნიშვნელოვანია, დიაგრამებზე აისახოს პაკეტებსშორისი კავშირი და მათში შემავალი ელემენტები. პაკეტებს შორის კავშირი იხაზება სხვადასხვა სახის ხაზებით.
თუ პაკეტი არის შედგენილი, მაშინ კავშირი აისახება დიაგრამაზე ხაზების გარეშე:



ნახ. 375 ან



ნახ. 376

ნიშანი ⊗ გვიჩვენებს, რომ 2 და 3 პაკეტები შედიან პაკეტ1-ის შემადგენლობაში.

7.4. კლას-დიაგრამებთან მუშაობა

7.4.1. კლასთაშორისი კავშირები

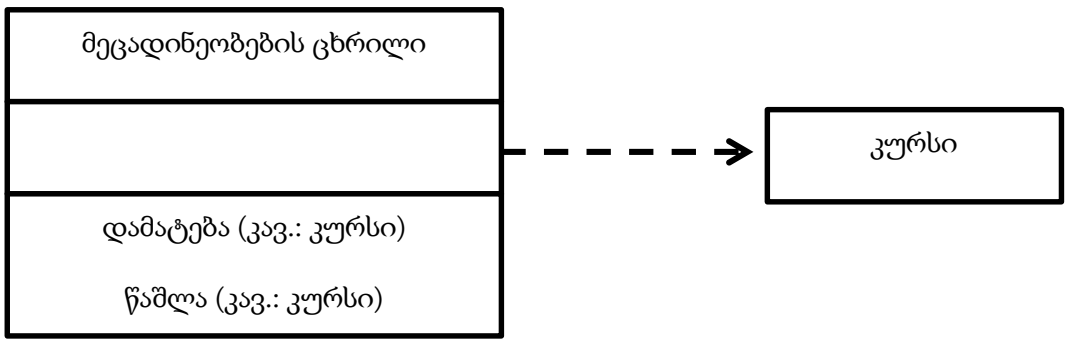
კლას-დიაგრამებში არსებობს 3 კატეგორიის კავშირი:

- დამოკიდებულობა (dependency),
- განზოგადება (generalization),
- ასოციაცია (Association).

მონაცემთა ბაზის დაპროექტებისათვის ყველაზე მნიშვნელოვანია მეორე და მესამე კავშირის კატეგორია.

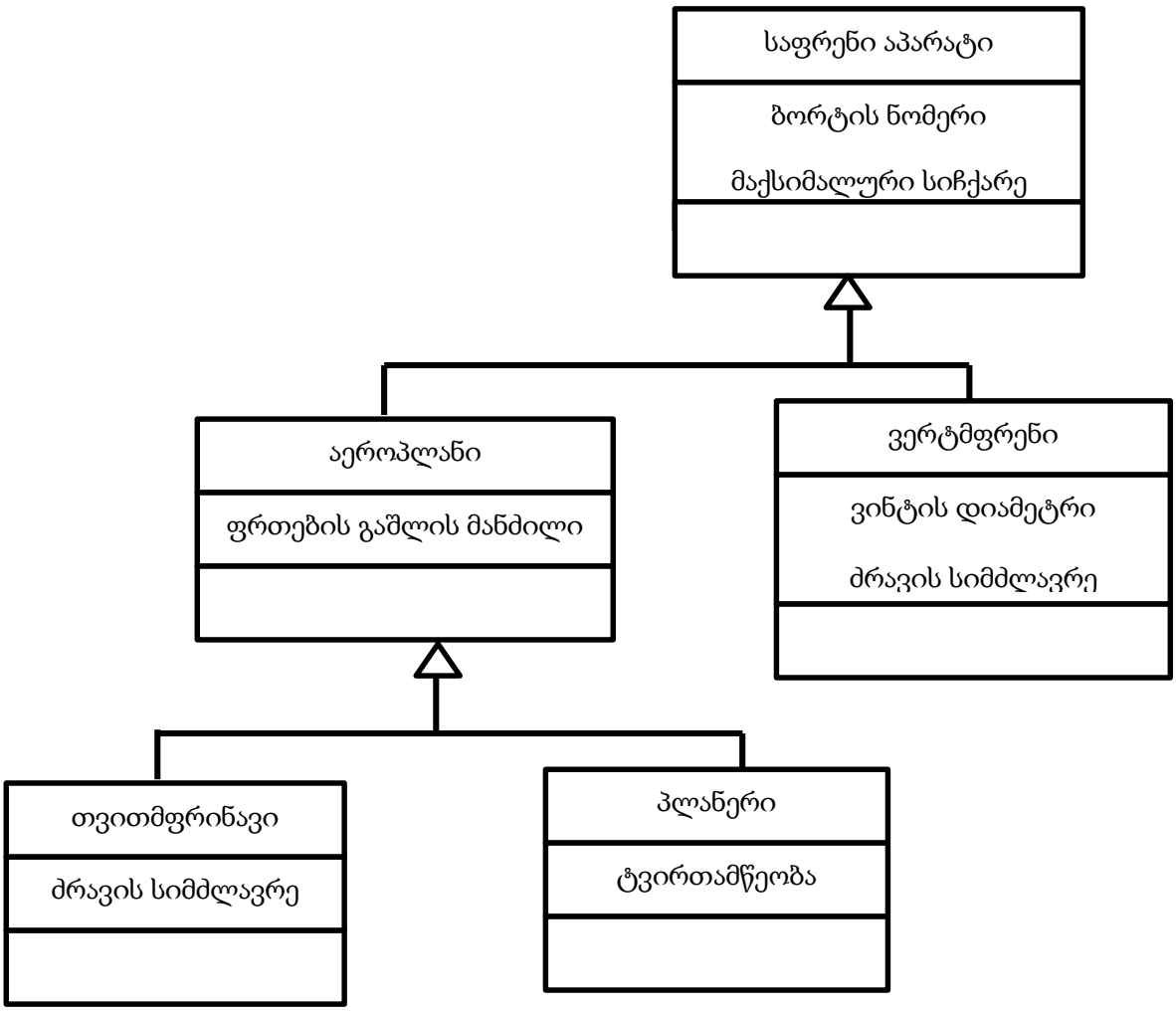
დამოკიდებულება წარმოადგენს კავშირს კლასებს შორის ან ობიექტებს შორის. დამოკიდებულება გვიჩვენებს, რომ ერთის ცვლილება ზეგავლენას მოახდენს მეორეზე, ანუ იმაზე, რომელიც იყენებს პირველს. დამოკიდებულება სტრუქტურული კავშირი არ არის. ის ჩნდება ლოკალური ან გლობალური ცვლადის ცვლილების გამო, ან მეთოდის შეცვლის გამო.

ხშირად დამოკიდებულებებს იყენებენ კლას_დიაგრამებში იმისათვის, რომ გარკვეულ კლასში გამოჩნდეს ის ფაქტი, რომ ოპერაციის პარამეტრები შეიძლება იყონ სხვა კლასის ობიექტები. ამიტომ, თუ მეორე კლასის ინტერფეისი იცვლება, ეს ცვლილება ახდენს ზეგავლენას პირველი კლასის ობიექტებზე. ქვემოთ მოცემულია დიაგრამა, რომელზეც ასახულია კავშირი-დამოკიდებულება:



ნახ. 377

დამოკიდებულება ნაჩვენებია წყვეტილი ხაზით, ისარი მიმართულია იმ კლასზე, რომლის მიმართ არსებობს დამოკიდებულება. კავშირი-დამოკიდებულება არსებითია ობიექტზე დამოკიდებული სისტემებისათვის.

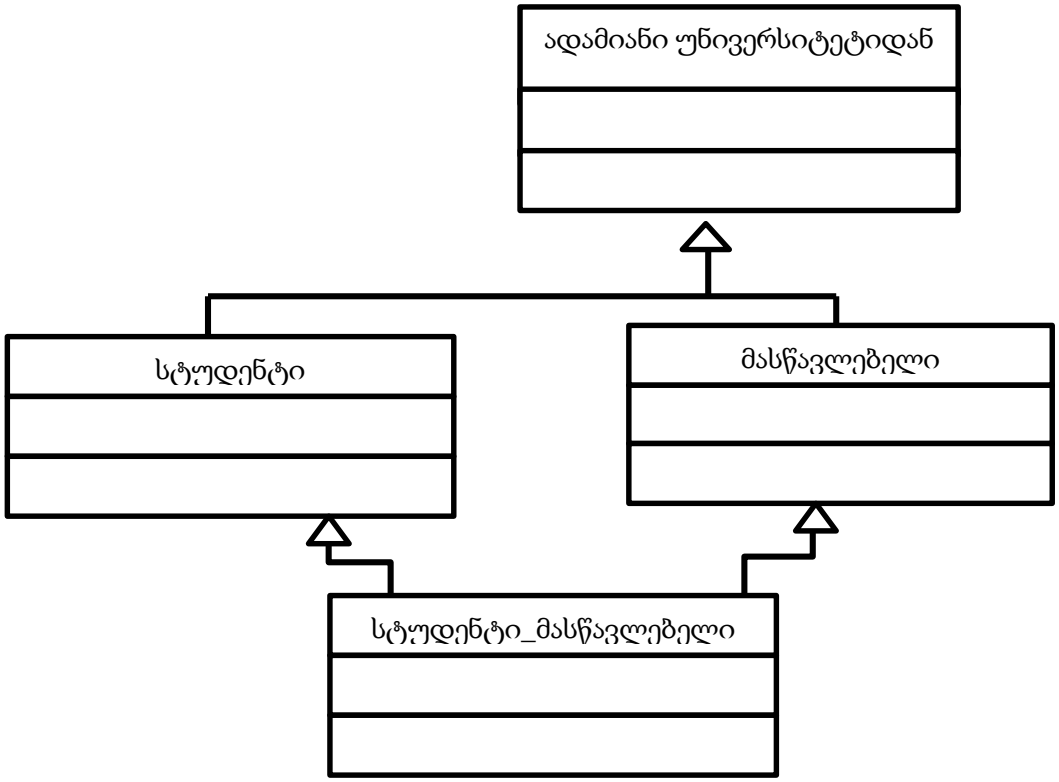


ნახ. 378

კავშირი-განზოგადოება წარმოადგენს სუპერკლასის, ანუ წინაპრის და ქვეკლასის, ანუ მემკვიდრეს შორის კავშირს. კლასი-მემკვიდრე წარმოადგენს კლას-წინაპრის ერთ-ერთ ნაწილს. ის მემკვიდრეობით ღებულობს კლას-წინაპრის ატრიბუტებს, მაგრამ ამასთან ერთად - ახალ ატრიბუტებს და ოპერაციებს.

კლასი-მემკვიდრის ობიექტების გამოყენება შესაძლებელია ყველგან, სადაც გამოიყენება კლასი-წინაპრის ობიექტები. ეს **პოლიმორფიზმის** თვისებაა, კლას-მემკვიდრის ობიექტების ჩართვა ხდება კლას-წინაპრის ობიექტების ერთობლიობაში.

გრაფიკულად განზოგადება გამოისახება უწყვეტი ხაზით და დიდი თეთრი ისრით, რომელიც მიმართულია სუპერკლასისკენ. მაგალითისთვის განვიხილოთ საფრენი აპარატების კლასიფიკაცია. აქ განხილულია ერთეული მემკვიდრეობის იერარქია: თითოეულ ქვეკლასს აქვს მხოლოდ ერთი სუპერკლასი.



ნახ. 379

7.4.2. მემკვიდრეობითობა

მარტივი მემკვიდრეობითობა საკმარისია კავშირ-განზოგადოებისათვის. მაგრამ UML-ში დაშვებულია მრავლობითი მემკვიდრეობითობაც, როდესაც ერთი ქვეკლასის განსაზღვრა ხდება რამდენიმე სუპერკლასის საფუძველზე. განვიხილოთ ნახ.379-ზე მოცემული დიაგრამა:

ამ დიაგრამაზე **სტუდენტი** და **მასწავლებელი** კლასები წარმოქმნილია **ადამიანი უნივერსიტეტიდან** სუპერკლასიდან. კლასს **სტუდენტი** მიეკუთვნება **ადამიანი უნივერსიტეტიდან** კლასის ობიექტებს, რომლებიც აღწერენ სტუდენტებს; ხოლო კლასს **მასწავლებელი** - მიეკუთვნებიან **ადამიანი უნივერსიტეტიდან** კლასის ობიექტები, რომლებიც აღწერენ მასწავლებლებს. არსებობს სიტუაციები, როდესაც სტუდენტები მუშაობენ მასწავლებლებად, მაშინ ერთ ობიექტს **ადამიანი უნივერსიტეტიდან** შეესაბამება ორი ობიექტი კლასებიდან **სტუდენტი** და **მასწავლებელი**. **სტუდენტი** კლასის ობიექტებში შეიძლება იყოს მასწავლებელიც, და ზოგიერთი მასწავლებელი შეიძლება იყოს სტუდენტი. მაშინ კლასი **სტუდენტი მასწავლებელი** განისაზღვრება მრავლობითი მემკვიდრეობითობით **სტუდენტი** და **მასწავლებელი** სუპერკლასებიდან. ამ ახალი კლასის ობიექტს გააჩნია ყველა ის თვისება, რომლებიც ახასიათებს **სტუდენტი** და **მასწავლებელი** კლასებს. მრავლობითი მემკვიდრეობა არც ისე

ხშირად გამოიყენება პრაქტიკაში და ქმნის რამდენიმე პრობლებას. ერთ-ერთ მათგანს წარმოადგენს ატრიბუტებისა და ოპერაციების დასახელება ქვეკლასებში.

მაგალითად, ვთქვათ, ქვეკლასში **სტუდენტი** და **მასწავლებელი** აღწერილია ატრიბუტი **ოთახის ნომერი.** დიდი ალბათობაა, რომ **სტუდენტი** კლასის ობიექტებისათვის გაწერილი იქნება საერთო საცხოვრებლის ოთახის ნომრები, ხოლო **მასწავლებელი** კლასისათვის - სამსახურის კაბინეტების ნომრები. **სტუდენტ-მასწავლებელი** კლასისათვის ორივე ნომერი არსებითია - სტუდენტს შეიძლება ჰქონდეს ოთახი საერთო საცხოვრებელში, და როგორც მასწავლებელს, იმავე სტუდენტს შეიძლება ჰქონდეს კაბინეტიც.

გამოსავალი შემდეგშია:

- აიკრძალოს **სტუდენტი მასწავლებელი** კლასის შექმნა სუპერკლასებში, სანამ არ გამოიცვლება დასახელება **ოთახის ნომერი;**
- მემკვიდრეობა გადაეცეს მხოლოდ ერთი სუპერკლასიდან, ასე რომ **ოთახის ნომერი** ატრიბუტი ყოველთვის ნიშნავდეს კაბინეტების ნომერს.
- ორივე ატრიბუტს შეეცვალოს სახელი, მაგალითად, **სტუდენტის ოთახის ნომერი** და **მასწავლებლის ოთახის ნომერი.**

არცერთი გადაწყვეტილება არ არის დამაკმაყოფილებელი, რადგან ქმნის არეულობას მონაცემთა ბაზის ატრიბუტებში და ართულებს პროცესს.

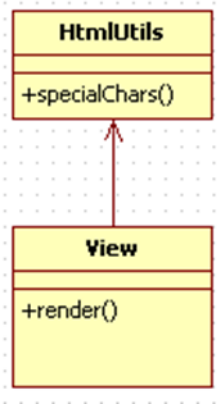
7.4.3. ასოციაციები

Associarions - კლასთაშორისი კავშირი, რომელიც აღწერს ამ კლასების ობიექტებს შორის კავშირს. ასოციაციას გააჩნია ნავიგაცია, რომელიც აჩვენებს კლასებს შორის დამოკიდებულებებს. ნებისმიერი პირი, რომელიც შედის ამ კავშირში ასრულებს გარკვეულ როლს. მაგალითად:

- 0..1 ნული ან ერთი
- 1 მხოლოდ ერთი
- 0..* ნული ან ბევრი
- 1..* ერთი ან ბევრი
- n მხოლოდ n (n>1)
- 0..n ნული n-თან
- 1..n ერთი n-თან

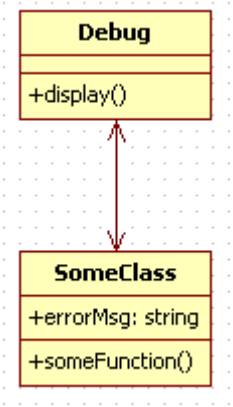
მიმართულებითი ასოციაცია (Message/ Directed Association) - გამოიყენება იმ შემთხვევაში, როდესაც ერთი კლასის მიმართვა სხვა კლასისადმი ხდება ინსტანცირების მეშვეობით.

ინსტანცირება - კლასის ეგზემპლიარის შექმნაა. **კლასის ეგზემპლიარში აღწერილია** და დამახსოვრებულია კლასის ობიექტი. კლასი აღწერს ობიექტის თვისებებს და მეთოდებს. ეგზემპლიარები კი გამოიყენება რეალური სამყაროს კონკრეტული პირების წარსადგენად. ყოველივე ეს გრაფიკულად აისახება შემდეგი სახით:



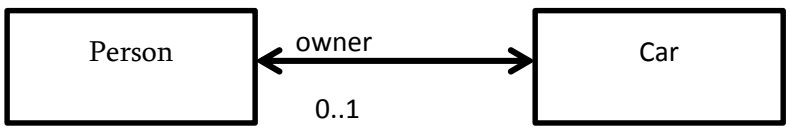
ნახ. 380

იგივე მიმართულებრივი ასოციაცია შეიძლება იყოს ორმხრივი:



ნახ. 381

ორმხრივი ასოციაცია წარმოადგენს თვისებებს, რომლებიც დაკავშირებულნი არიან ურთიერთსაპირისპირო მიმართულებით.



ნახ. 382

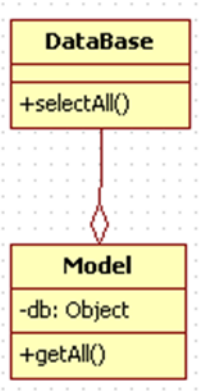
Owner: Person[1], კლასი - Person, თვისება: cars: Car[*].

მათ შორის უკუკავშირი გულისხმობს საწყის წერტილში დაბრუნებას. მაგალითად, თუ დავიწყებთ კონკრეტული მოდელით MGMidget, ვიპოვით მის პატრონს, შემდეგ კი ვათვალიერებთ მის მფლობელობაში მყოფ მანქანების სიას და ამ სიაში აუცილებლად შეგვხვდება MGMidget, რითაც დავიწყეთ ძიება.

მთავარია, რომ ასოციაციის ერთი მხარე მართავდეს ყველა ურთიერთობას. ამისათვის ბოლო წევრმა - Person უნდა წარუდგინოს უფროსს წევრს ყველა თავისი მონაცემი. ასეთი ნავიგაცია ძალიან მნიშვნელოვანია კონცეპტუალურ მოდელებში.

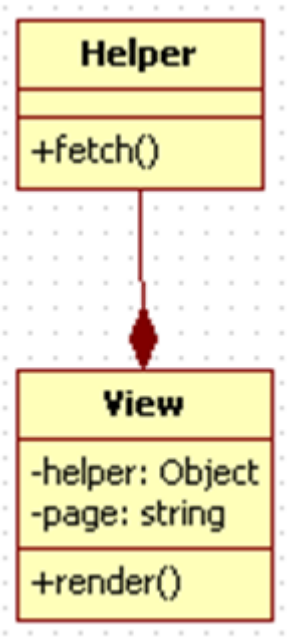
აგრეგაცია (has a) ასოციაციის ნაწილი. ის გამოიყენება შემთხვევებში, როდესაც ერთი კლასი წარმოადგენს კონტეინერს დანარჩენი კლასებისთვის. ჩვენს შემთხვევაში Model-ის კლასის ობიექტი წარმოადგენს კონტეინერს. და თუ მას წავშლით, ის არანაირ ზეგავლენას არ მოახდენს DataBase კლასზე.

გრაფიკულად აგრეგაცია აისახება ცარიელი რომბით კლასების ხაზზე (ნახ.383):



ნახ. 383

კომპოზიცია წარმოადგენს ასოციაციის ძალზე მკაცრ შემთხვევას. აგრაგაციისგან განსხვავებით, კომპოზიციას გააჩნია კონტეინერში შემავალი კლასის ეგზემპლიარის ხისტი დამოკიდებულება სხვა კლასების მიმართ. თუ კონტეინერი წაიშლება, ყველაფერი განადგურდება. გრაფიკულად კომპოზიცია აისახება აგრეგაციის მსგავსად, ოღონდ რომბი არის გაფერადებული:



ნახ. 384

აგრეგაციის და კომპოზიციის განსხვავება მდგომარეობს იმაში, რომ კომპოზიცია არის ერთადერთი მთელი ობიექტის ნაწილი, ხოლო აგრეგაცია კი - რამდენიმე ობიექტის ნაწილი.

7.5. ვიზუალური დიაგრამით პროექტის შექმნა

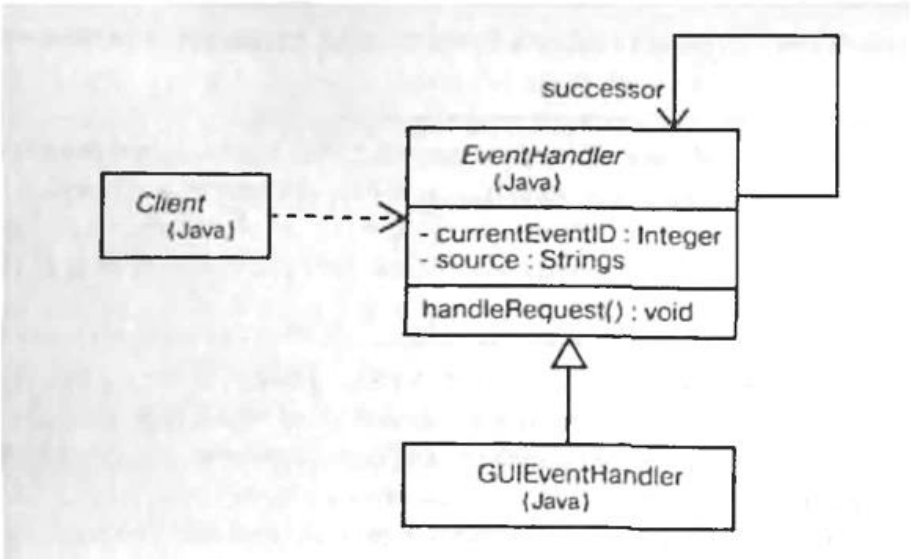
7.5.1. პროგრამული კოდის გენერაცია

მოდელირების ძირითადი მიზანია - მოკლე ვადებში დაეხმაროს პერსონალს პროგრამული უზრუნველყოფის შექმნაში. ამიტომ ძალზე მნიშვნელოვანია სინქრონიზირებული მოდელების შექმნა. ხანდახან UML-ს იყენებენ ისეთი მოდელის შესაქმნელად, რომელიც არც ერთ პროგრამულ კოდში არ გარდაისახება. მაგალითად, თუ ხდება მოქმედებების დიაგრამებით ბიზნეს-პროცესის შექმნა. ამ შემთხვევაში მოქმედებებს ასრულებენ ადამიანები და არა კომპიუტერები. მაგრამ უფრო ხშირად ხდება UML-დიაგრამებით შექმნილი მოდელების გარდაქმნა პროგრამულ კოდში Java, C++, Ada, Object Pascal და სხვა დაპროგრამების ენების მეშვეობით.

პირდაპირი დაპროექტება (Forward engineering) წარმოადგენს მოდელის პროგრამულ კოდში გარდაქმნას სპეციალური დაპროგრამების ენის მეშვეობით. ამ შემთხვევაში ინფორმაციის ნაწილი იკარგება, რადგან UML-მოდელები სემანტიკურად უფრო ზუსტად აღწერენ პროცესს. მაგალითად სისტემის კოოპერაციული ან მოქმედებითი თვისებები იკარგება პროგრამული კოდის გამოყენების შემთხვევაში, UML-მოდელებში კი ეს თვისებები თვალნათლივ ჩანს. კლას-დიაგრამების პირდაპირი დაპროგრამება ხდება შემდეგნაირად:

- მოდელის ინდენტიფიკაცია ერთ-ერთ დაპროგრამების ენაზე;
- რამდენიმე ოპერაციის ამოგდება მოდელიდან, რადგან დაპროგრამების ენაზე ეს ოპერაციები ვერ განხორციელდება. (მაგალითად, UML-ში შეიძლება მრავლობითი მემკვიდრეობის გამოყენება, დაპროგრამების ენაში კი - არა);
- უნდა მოხდეს დაპროგრამების ენის სპეციფიკაციების აღწერა. სჯობს ეს გაკეთდეს ინდივიდუალური კლასების დონეზე. შეიძლება პაკეტებში და კოოპერაციებშიც განვმარტოთ ეს სპეციფიკაციები;
- აუცილებლად უნდა იყოს გამოყენებული ინსტრუმენტები. ნახაზზე მოცემულია ობიექტის რეალიზაციის მარტივი კლას-დიაგრამა, რომელიც წარმოადგენს მოვალეობების ჯაჭვს. აქ ნაჩვენებია სამი კლასი: **Client** (კლიენტი), **EventHandler** (მოვლენების დამმუშავებელი), **GUIE EventHandler** (მოვლენების დამმუშავებელი პროგრამულ ენაზე). პირველი ორი - აბსტრაქტულია, ხოლო მესამე - კონკრეტული. კლასი **EventHandler** შეიცავს ჩვეულებრივ ოპერაციას handleRequest (დავამუშავოთ კითხვა).

მოცემულ მაგალითში დამატებულია ორი დამალული ატრიბუტი.



ნახ. 385

პირდაპირი დაპროექტება ხდება სპეციალური ინსტრუმენტის გამოყენებით. მაგალითად, **EventHandler** კლასისთვის პროგრამული კოდია:

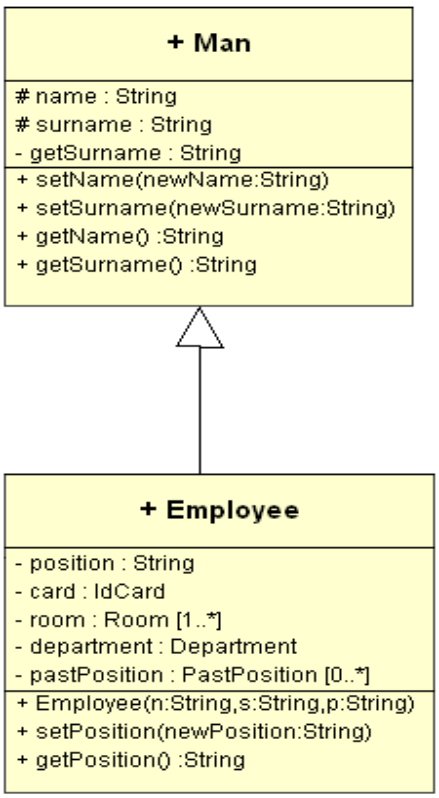
```
public abstract class EventHandler {
EventHandler successor;
private Integer currentEventID;
private String source;
EventHandler () {}
public void handleRequest () {}
}
```

Reverse engineering – რომელიმე დაპროგრამების ენაზე დაწერილი კოდის გარდაქმნა მოდელში. აქ დამპროექტებელი იღებს უამრავ დამატებით ინფორმაციას, რომელიც იმყოფება დეტალიზაციის უფრო დაბალ დონეებზე, ვიდრე საჭიროა მოდელის ასაწყობად. პირდაპირი დაპროექტების დროს ხდება ინფორმაციის დაკარგვა, ასე, რომ მოდელის აღდგენა სრულად ვერ ხერხდება.

საპირისპირო დაპროექტება ხოციელდება შემდეგნაირად:

- მოდელის ინდენტიფიკაცია ერთ-ერთ დაპროგრამების ენაზე;
- საპირისპირო დაპროექტების გამოყენება;
- მიღებული მოდელის გამოყენებით კლას-დიაგრამის შექმნა.

მაგალითი. ავაგოთ UML-კლას-დიაგრამა (Class Model), შემდეგ კი გადავიყვანოთ ობიექტ-ორიენტირებულ კოდში. ავაგოთ ერთ-ერთი საწარმოს კადრების განყოფილების მოდელი. დაპროგრამების ენა - Java.



ნახ. 386

კლასი «**Man**»(ადამიანი) — აბსტრაქტულია, ხოლო «**Employee**»(თანამშრომელი) უფრო სპეციალიზირებული, მას გააჩნია «Man» კლასის თვისებები და მეთოდები. ამ დიაგრამის პროგრამული კოდია:

```
public class Man{
protected String name;
   protected String surname;
   public void setName(String newName){
      name = newName;
   }
   public String getName(){
      return name;
```

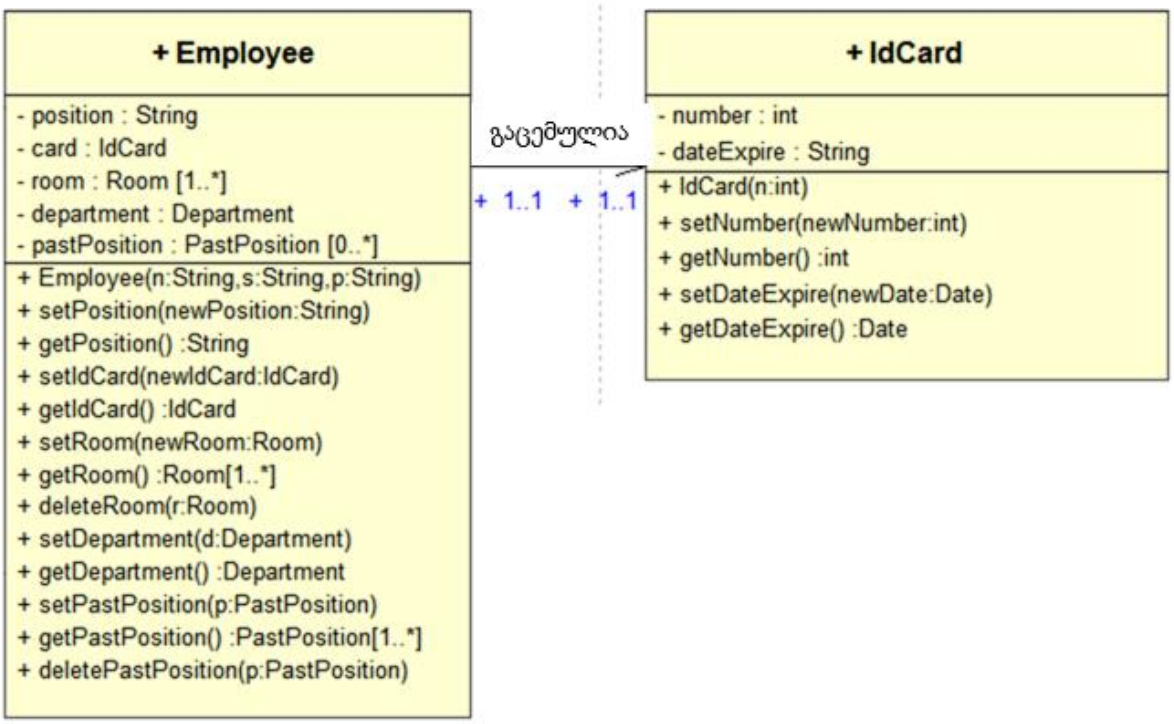
```
        }
   public void setSurname(String newSurname){
      name = newSurname;
   }
   public String getSurname(){
      return surname;
   }
}
// Man კლასის მემკვიდრე
public class Employee extends Man{
   private String position;
    public Employee(String n, String s, String p){
      name = n;
      surname = s;
      position = p;
   }
   public void setPosition(String newProfession){
      position = newProfession;
   }
   public String getPosition(){
      return position;
   }

}
```

ასოციაცია

ასოციაცია მიგვითითებს ობიექტსა და ეგზემპლიარს შორის ურთიერთობებზე

ორმხრივი ასოციაცია

მოდელს დავუმატოთ კლასი «**IdCard**», რომელიც წარმოადგენს თანამშრომლის საინდენტიფიკაციო ბარათს (საშვს). თითოეულ თანამშრომელს შეესაბამება მხოლოდ ერთი საინდენტიფიკაციო ბარათი, კავშირი 1 – 1.



ნახ. 387

კოდი:

```
public class Employee extends Man{
   private String position;
   private IdCard iCard;
   public Employee(String n, String s, String p){
      name = n;
      surname = s;
      position = p;
   }
   public void setPosition(String newPosition){
      position = newPosition;
   }
   public String getPosition(){
      return position;
   }
   public void setIdCard(IdCard c){
      iCard = c;
   }
   public IdCard getIdCard(){
      return iCard;
   }
```

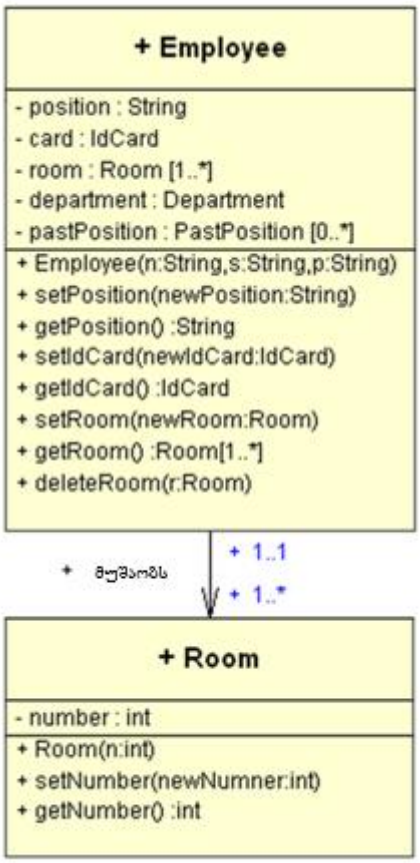
```
}
public class IdCard{
	private Date dateExpire;
	private int number;
	public IdCard(int n){
		number = n;
	}
	public void setNumber(int newNumber){
		number = newNumber;
	}
	public int getNumber(){
		return number;
	}
	public void setDateExpire(Date newDateExpire){
		dateExpire = newDateExpire;
	}
	public Date getDateExpire(){
		return dateExpire;
	}
}
```

ციკლში ვაკავშირებთ ყველა ობიექტს:

```
IdCard card = new IdCard(123);

card.setDateExpire(new SimpleDateFormat("yyyy-MM-dd").parse("2015-12-31"));

sysEngineer.setIdCard(card);

System.out.println(sysEngineer.getName() +" working in office "+ sysEngineer.getPosition());

System.out.println("The certificate is valid until " + new SimpleDateFormat("yyyy-MM-dd").format(sysEngineer.getIdCard().getDateExpire()) );
```

Employee კლასს გააჩნია ველი **card** ტიპით **IdCard**. კლასს აქვს მეთოდები (**setIdCard**) და (**getIdCard**). Employee ობიექტის მეშვეობით შეგვიძლია მასთან დაკავშირებული IdCard ტიპის ობიექტზე ინფორმაციის მოძიება. ამიტომ ისარი მიმართულია Employee-დან IdCard -ზე.

N-დონიანი ასოციაცია



ნახ. 388

ორგანიზაციებში თანამშრობლებზე უნდა იყოს ოთახები გადანაწილებული. დავუმატოთ ახალი კლასი Room. თითო თანამშრომელს (Employee) შეიძლება შეესაბამებოდეს რამდენიმე ოთახი. კავშირი ერთი-მრავალი. ნავიგაცია Employee-დან Room-ზე.

ამ დიაგრამის კოდი:

```
public  class Room{
    private int number;
    public Room(int n){
       number = n;
    }
    public void setNumber(int newNumber){
       number = newNumber;
    }
    public int getNumber(){
       return number;  }}
```

დავუმატოთ **Employee** კლასს ველი და მეთოდი **Room:**

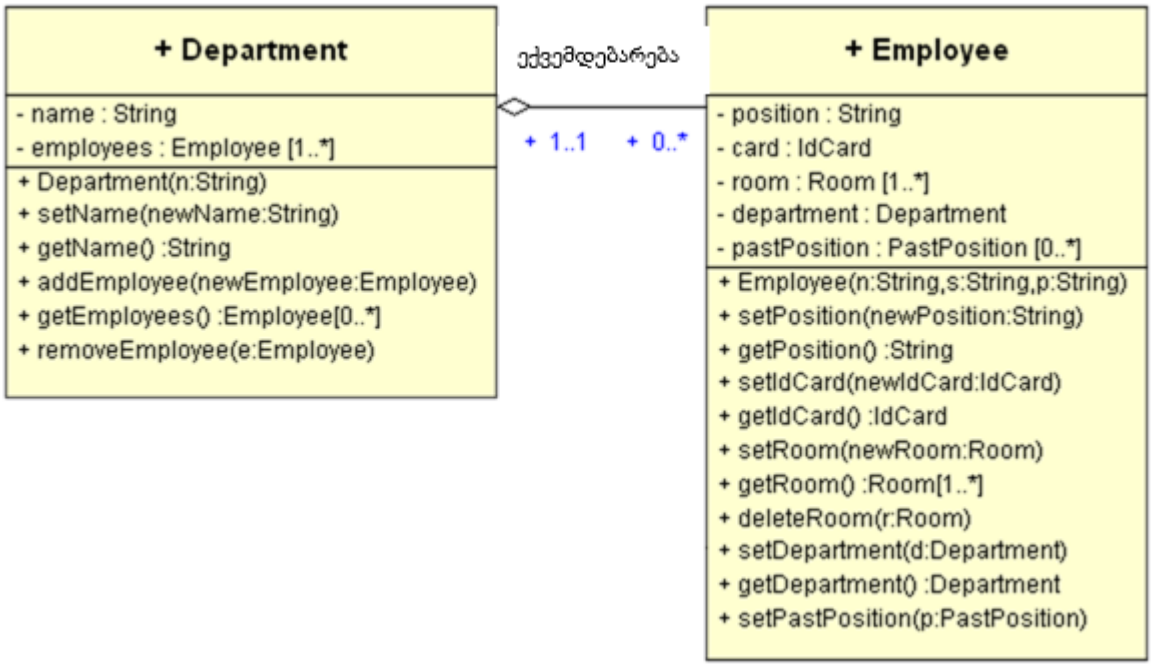
```
...
private Set room = new HashSet();
...
public void setRoom(Room newRoom){
   room.add(newRoom);
}
public Set getRoom(){
   return room;
}
public void deleteRoom(Room r){
   room.remove(r);
}
...
```

გამოყენების მაგალითი:

```
public static void main(String[] args){

   Employee sysEngineer = new Employee("John", "Connor", "Manager");
   IdCard card = new IdCard(123);
   card.setDateExpire(new SimpleDateFormat("yyyy-MM-dd").parse("2015-12-31"));
          sysEngineer.setIdCard(card);
          Room room101 = new Room(101);
   Room room321 = new Room(321);
   sysEngineer.setRoom(room101);
   sysEngineer.setRoom(room321);
   System.out.println(sysEngineer.getName() +" Position "+ sysEngineer.getPosition());
   System.out.println("The validity of the permit:" + sysEngineer.getIdCard().getDateExpire());
   System.out.println("Could be the following rooms“);
   Iterator iter = sysEngineer.getRoom().iterator();
   while(iter.hasNext()){
      System.out.println( ((Room) iter.next()).getNumber());}}
```

აგრეგაცია

შემოვიყვანოთ მოდელში კლასი **Department**(განყოფილება) —ჩვენს საწარმოს გააჩნია სტრუქტურირება განყოფილებების მიხედვით. თითოეულ განყოფილებაში შეუძლია ერთ და მეტ თანამშრომელს მუშაობა. შეიძლება ითქვას, განყოფილება შეიცავს ერთს ან მეტს თანამშრომელს. მაგრამ საწარმოში შეიძლება მუშაობდეს თანამშრომელი, რომელიც არც ერთ განყოფილებას არ ექვემდებარება, მაგალითად, დირექტორი.



ნახ. 389

აგრეგაციის პროგრამული კოდი:

```
public class Department{
   private String name;
   private Set employees = new HashSet();
   public Department(String n){
      name = n;
   }
   public void setName(String newName){
      name = newName;
   }
   public String getName(){
      return name;
```

```
  }
  public void addEmployee(Employee newEmployee){
     employees.add(newEmployee);
     // თანამშრომელი და განყოფილება
     newEmployee.setDepartment(this);
  }
  public Set getEmployees(){
     return employees;
  }
  public void removeEmployee(Employee e){
     employees.remove(e);
  }
}
```

კლასი კონსტრუქტორის და მეთოდის გარდა შეიცავს განყოფილებაში თანამშრომლის შესახებ ინფორმაციის შეტანის, წაშლის და მთელი განყოფილების სიის მიღების მეთოდებს. დიაგრამაზე ნავიგაცია არ არის ნაჩვენები, ამიტომაც ის წარმოადგენს ორმიმართულებიან დიაგრამას - «Department» ტიპის ობიექტიდან ვღებულობთ ინფორმაციას თანამშრომელზე, და «Employee» ტიპის ობიექტიდან ვგებულობთ რომელ განყოფილებაში მუშაობს ის. დავუმატოთ «Employee» ობიექტს მეთოდები განყოფილების დასახელების მისაღებად.

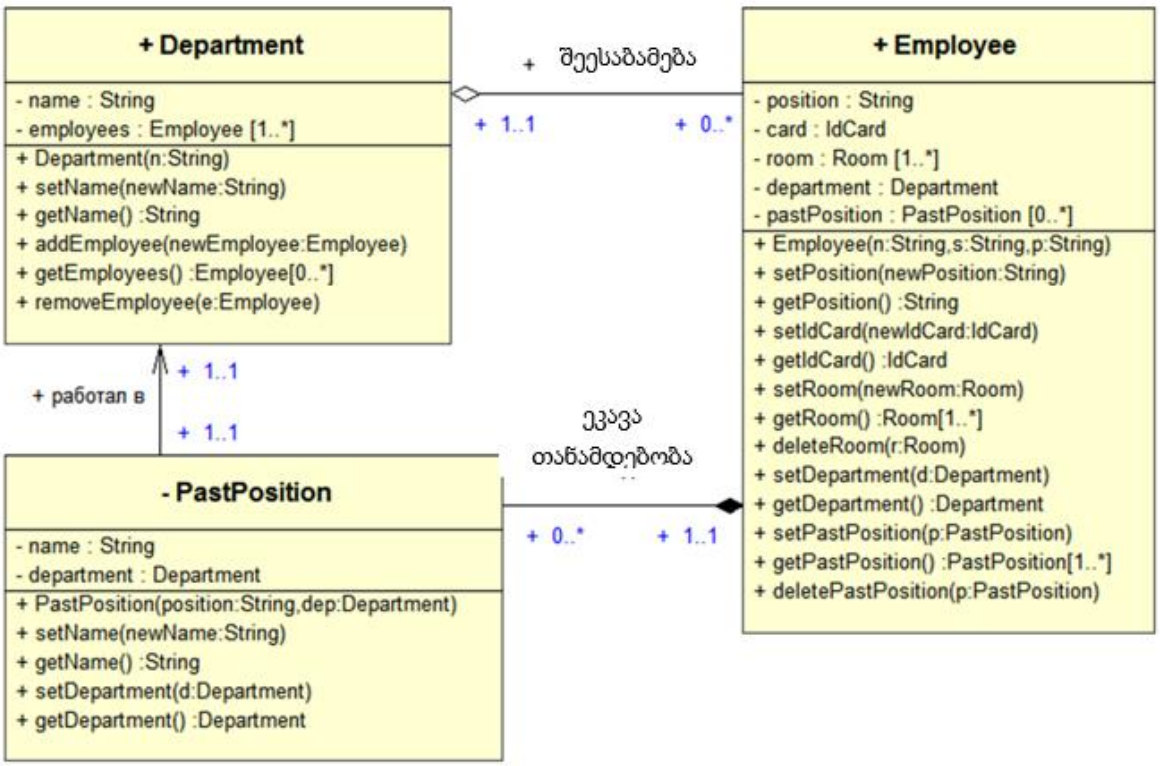
```
...
private Department department;
...
public void setDepartment(Department d){
   department = d;
}
public Department getDepartment(){
   return department;
}
```

გამოყენება:

```
Department programmersDepartment = new Department("Programmers ");

programmersDepartment.addEmployee(sysEngineer);

System.out.println("ეკუთვნის განყოფილებას "+sysEngineer
```

კომპოზიცია

დავუშვათ, რომ ჩვენი სისტემის ერთ-ერთი მოთხოვნაა - თანამშრომლის ადრე დაკავებული თანამდებობის შესახებ ინფორმაციის დამახსოვრება. ამისათვის შემოვიღოთ ახალი კლასი «pastPosition». ამ კლასის თვისებებია: სახელი (name) და «department», რომელიც დააკავშირებს მას «Department» კლასთან. ძველი თანამდებობები წარმოადგენენ თანამშრომლის მონაცემების ნაწილს, ამიტომ ისინი უნდა იყოს «Employee» ობიექტის ნაწილები. თუ წავშლით მას, ავტომატურად უნდა წაიშალოს «pastPosition» ობიექტებიც.



ნახ. 390

```
private class PastPosition{
   private String name;
    private Department department;
    public PastPosition(String position, Department dep){
       name = position;
       department = dep;
    }
    public void setName(String newName){
       name = newName;
    }
    public String getName(){
       return name;
    }
    public void setDepartment(Department d){
       department = d;
    }
    public Department getDepartment(){
       return department;
    }
}
```

Employee კლასში დავამატოთ ადრინდელი თანამდებობის თვისებები და მეთოდები:

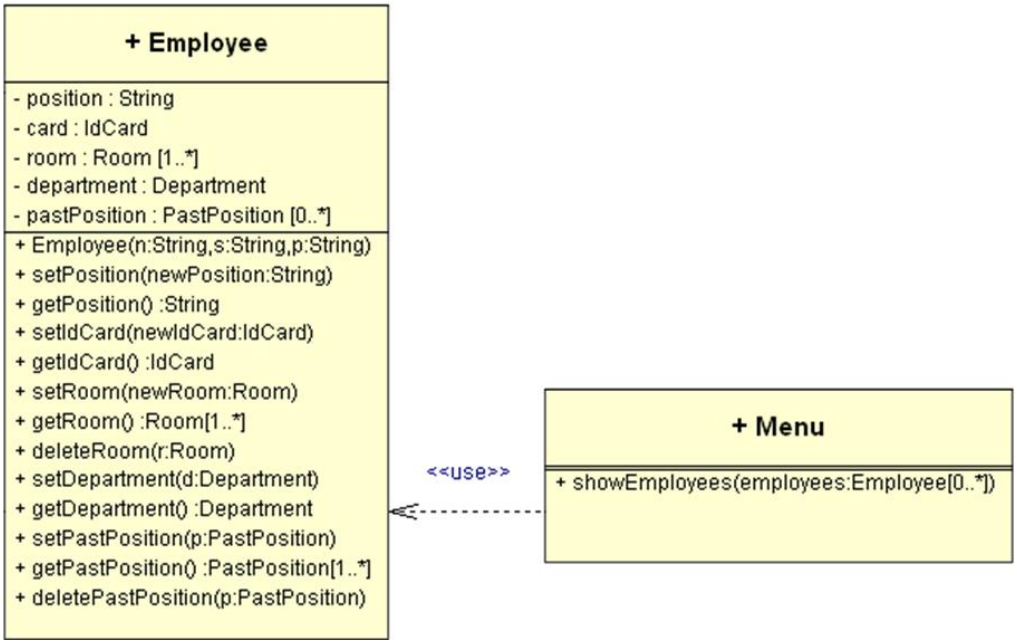
```
...
private Set pastPosition = new HashSet();
...
public void setPastPosition(PastPosition p){
    pastPosition.add(p);
}
public Set getPastPosition(){
    return pastPosition;
}
public void deletePastPosition(PastPosition p){
    pastPosition.remove(p);}
```

გამოყენება:

```
// ვცვლით თანამდებობას
sysEngineer.setPosition("Watchman");
// ადრე დაკავებული თანამდებობები:
System.out.println("In the past he held the position:“);
Iterator iter = sysEngineer.getPastPosition().iterator();
while(iter.hasNext()){
        System.out.println( ((PastPosition) iter.next()).getName());
}
```

დამოკიდებულება

მომხმარებელთან დიალოგის შესაქმნელად შემოვიღოთ კლასი «**Menu**».დავუმატოთ მეთოდი «showEmployees», (თანამშრომელთა სია და მათი თანამდებობები). ამ მეთოდის პარამეტრია «Employee». კლასის ობიექტების მასივია. თუ შევცვლით «Employee» კლასს, ავტომატურად შეიცვლება «Menu» კლასიც.



ნახ. 391

კოდი:

```
public  class Menu{
   private static int i=0;
   public static void showEmployees(Employee[] employees){
```

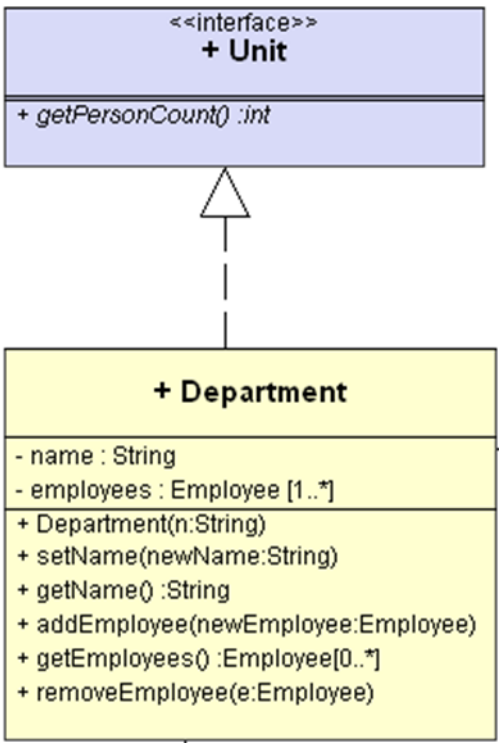
```
    System.out.println("list of employees:");
    for (i=0; i<employees.length; i++){
       if(employees[i] instanceof Employee){
          System.out.println(employees[i].getName() +" - " + employees[i].getPosition());
                              }
                    }
  }
}
```

გამოყენება:

```
// ერთი თანამშრომლის დამატება

Employee director = new Employee("Nemsadze", "Ninua", "Mkheidze");

Menu menu = new Menu();

Employee employees[] = new Employee[10];

employees[0]= sysEngineer;

employees[1] = director;

Menu.showEmployees(employees);
```

რეალიზაცია

შევქმნათ «Unit» ინტერფეისი. წარმოვიდგინოთ, რომ ჩვენ საწარმოს გააჩნია არა მხოლოდ განყოფილებები, არამედ საამქროები და ფილიალები. ინტერფეისი წარმოადგენს აბსტრაქტულ ერთეულს. თითოეულ ერთეულში მუშაობს თანამშრომელთა გარკვეული რაოდენობა.



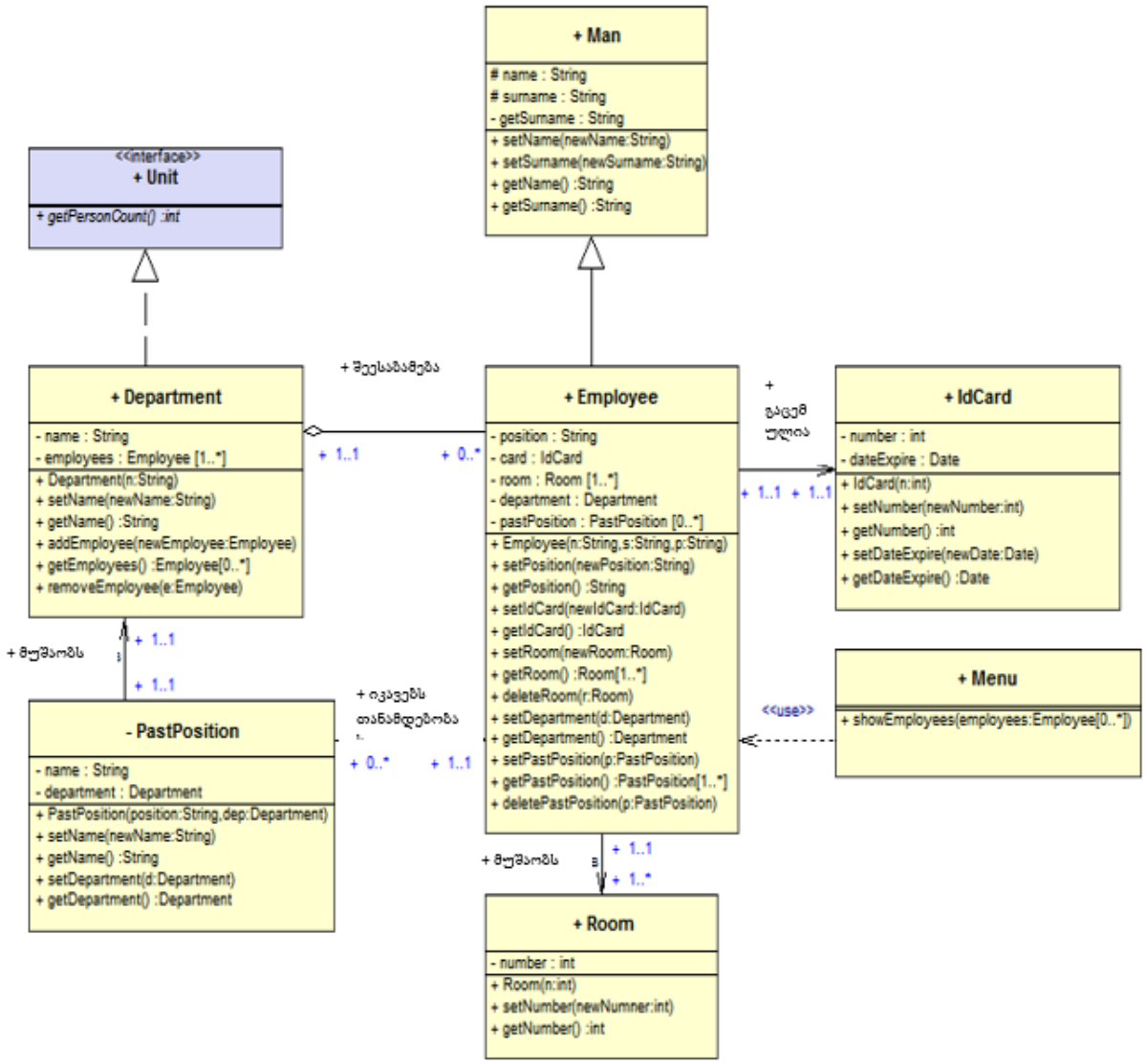
ნახ. 392

```
public interface Unit{
    int getPersonCount();
}
//«Department»კლასის რეალიზაცია:
public class Department implements Unit{
   ...
   public int getPersonCount(){
      return getEmployees().size();
}
```

გამოყენება:

```
System.out.println("განყოფილებაში "+sysEngineer.getDepartment().getName()+" Working "
+sysEngineer.getDepartment().getPersonCount()+" Employee.");
```

მოდელირების შედეგად მივიღებთ შემდეგ დიაგრამას:



ნახ. 393

კითხვები თვითშემოწმებისათვის

1. *UML-ენა. განმარტება, დიაგრამების შემადგენლობა*
2. *UML –ის ძირითადი კონსტრუქციები*
3. *რა არის UML-დიაგრამა?*
4. *UML-დიაგრამების სია.*
5. *რა არის ურთიერთობები UML-ში?*

7.6. მონაცემთა ბაზები და UML

მონაცემთა ბაზიდან ცხრილების გამოტანა

მონაცემთა მოდელი გამოიყენება მონაცემთა ბაზების აგების დროს. ეფექტური მოდელი უნდა იყოს საკმაოდ მარტივი, იმისათვის, რომ მომხმარებელმა შეძლოს მონაცემთა სტრუქტურებთან მუშაობა. ამავდროულად ეს მოდელი უნდა იყოს ყველა კუთხიდან აღწერილი, იმისათვის, რომ დამპროექტებელმა შეძლოს ფიზიკური სტრუქტურების შექმნა.

განვიხილოთ **Rational Application Developer 6.0 –ში (RAD)** მონაცემთა მოდელების შექმნა და მონაცემთა ბაზების დამუშავება კლას-დიაგრამების მეშვეობით. თქვენ შეძლებთ:

- მონაცემთა ბაზების შექმნას;
- სქემების აგებას (მხოლოდ კლას-დიაგრამებში)
- ცხრილებთან მუშაობას;
- საწყისი და გარე გასაღებების შექმნას;
- პროცედურების გამოყენებას;
- მომხმარებლის მიერ განსაზღვრულ ფუნქციებთან მუშაობას.

მონაცემთა ბაზების შექმნა.

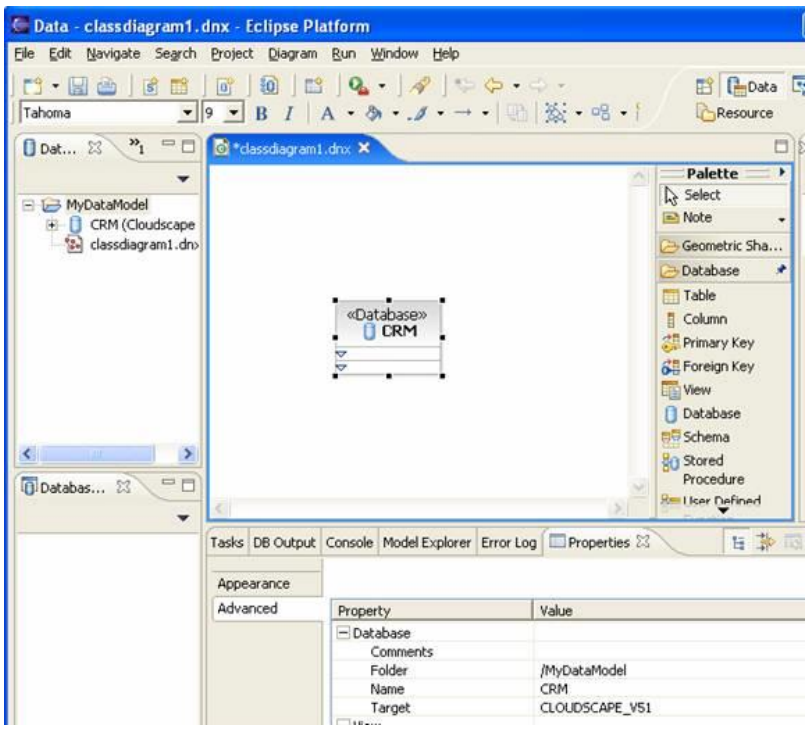
უპირველეს ყოვლისა საჭიროა მონაცემთა მოდელის მარტივი პროექტის შექმნა (MyDataModel) და მონაცემთა პერსპექტივის გახსნა. როცა მონაცემთა განსაზღვრის ფანჯარა ღიაა:

1. მონაცემთა განსაზღვრის ფანჯრიდან აირჩიეთ (**Select**) პროექტების საქაღალდე **(Project Folder).**
2. კონტექსტური მენიუდან (მაუსის მარჯვენა ღილაკი) აირჩიეთ ახალი კლას-დიაგრამა **(New Class Diagram)**
3. დაარქვით დიაგრამას სახელი, მაგალითად, **classdiagram1** და დააჭირეთ **Finish.**
4. ამით იქმნება და იხსნება ახალი კლას-დიაგრამა.
5. აირჩიეთ Database (მონაცემთა ბაზის შექმნის ინსტრუმენტი) Data palette მენიუდან, როგორც ნახაზზეა ნაჩვენები:

ნახ. 394

6. დააჭირეთ დიაგრამას;
7. დაარქვით მონაცემთა ბაზა **(Database name),** მაგალითად, **CMR**. აირჩიეთ **Database vendor type** და დააჭირეთ **Finish**.

ჩვენ შევქმენით მონაცამთა ბაზა **CMR**. გავხსნათ პროექტის საქაღალდე მონაცემთა განსაზღვრის ფანჯარაში, იქ გამოჩნდება ახალი მონაცემთა ბაზა, რომელიც ასახული იქნება როგორც კლას-დიაგრამის UML კომპონენტი.



ნახ. 395

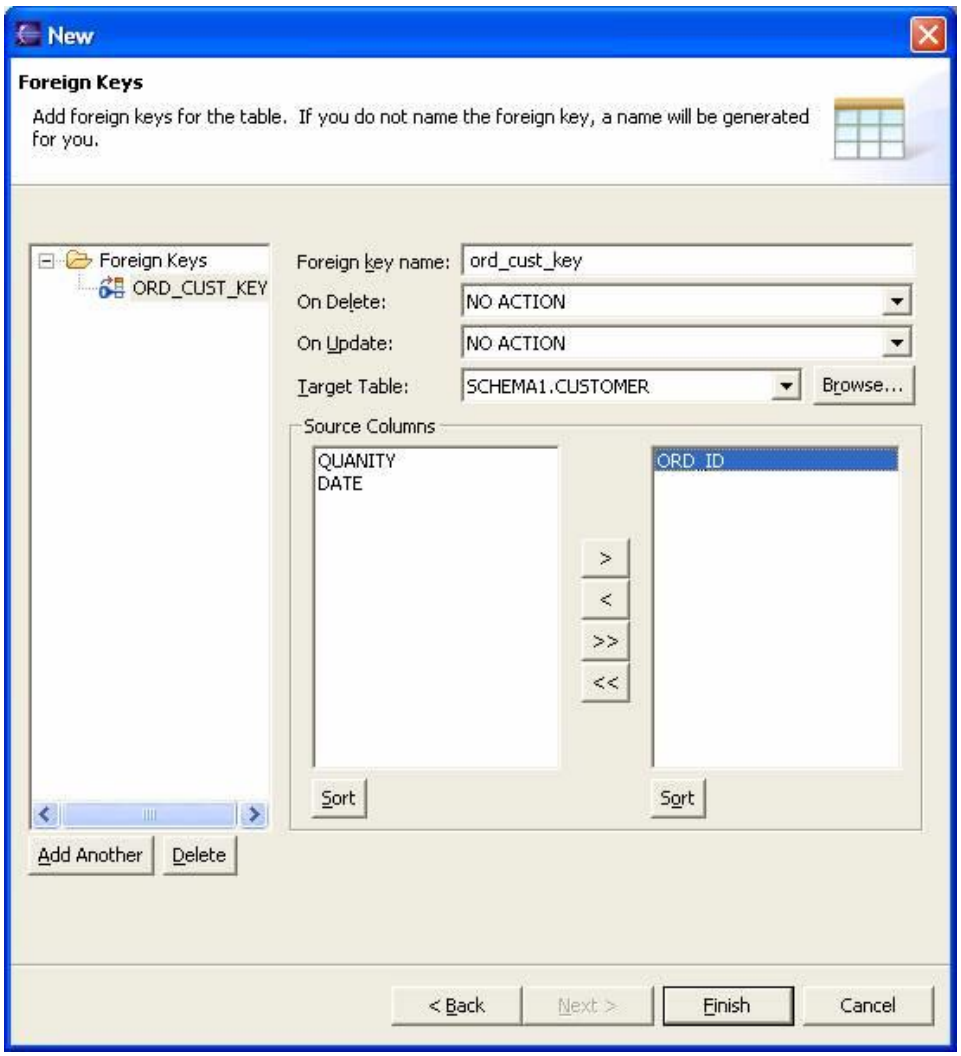
ცხრილების შექმნა.

შევქმნათ 2 ცხრილი: გამყიდველი **(Customer**) და გზავნილები (**Order)** მათ შორის.

1. აიღეთ ინსტრუმენტი **Table;**
2. დააჭირეთ დიაგრამას;
3. დაარქვით სქემას სახელი **schema1** და შემდეგ დააჭირეთ **Next.**
4. შეიყვანეთ ცხრილის სახელი **Customer** , ისევ **Next.**
5. **შეიყვანეთ შემდეგი სვეტები: cust_id** (ველის კოდი), **name** (სახელი), **address** (მისამართი), **phone** (ტელეფონი).
6. **Finish.**

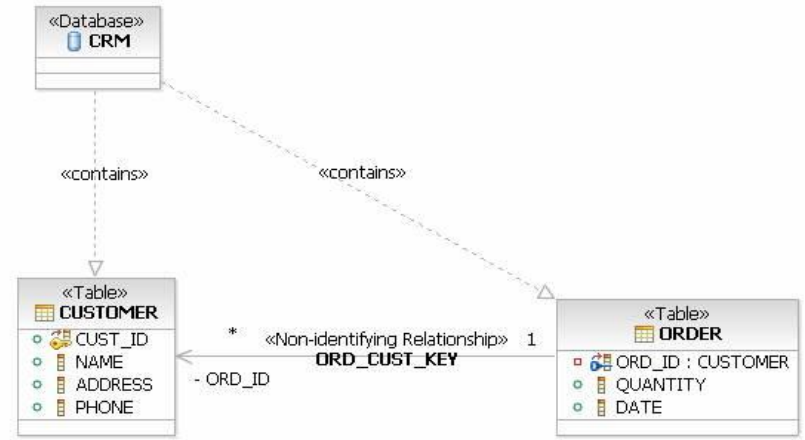
   ამგვარად, შეიქმნა სქემა **schema1** და ცხრილი **Customer.** ეს ცხრილი დიაგრამაზე წარმოადგენს UML-კლასს. შევქმნათ მეორე ცხრილი - **Orders.**

1. აიღეთ ინსტრუმენტი **Table;**
2. დააჭირეთ დიაგრამას;
3. შეიყვანეთ ცხრილის სახელი **Orders** , ისევ **Next.**
4. **შეიყვანეთ შმდეგი სვეტები: ord_id** (ველის კოდი), **quantity** (რაოდენობა), **date** (თარიღი), **Next.**
5. არ შეავსოთ პირველადი გასაღები და დააჭირეთ **Next.**
6. გარე გასაღების გვერდზე დაუმატეთ ახალი გარე გასაღები **ord_cust_key.** აირჩიეთ **(Select) Customer** პირველად ცხრილად**(Target Table),** ხოლო **ord_id -** პირველად სვეტად.



ნახ. 396

Order ცხრილი დიაგრამაზე მიიღებს UML-კლასის ფორმას. ქვემოთ მოყვანილია ცხრილებს შორის ურთიერთობები გარე გასაღების მეშვეობით:



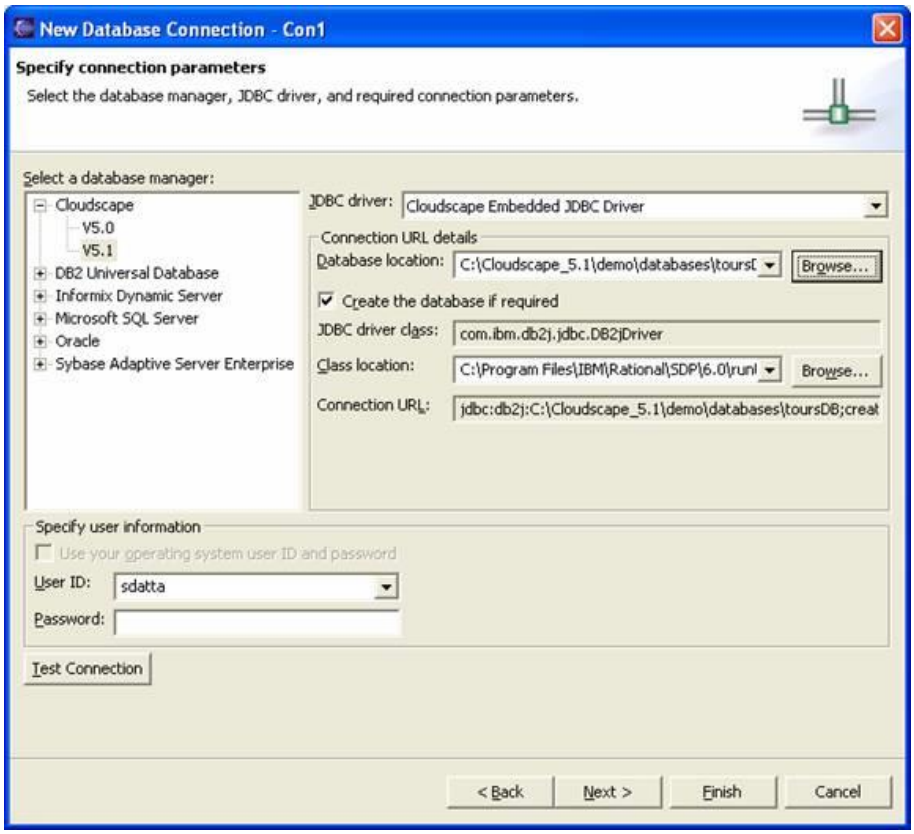
ნახ. 397

შესაძლებელია პირველადი და გარე გასაღებების შექმნა სხვა ხერხითაც. აირჩიეთ ინსტრუმენტების პანელზე პირველადი გასაღები (**Primary Key**) და დააჭირეთ ცხრილს. ველისაგან პირველადი გასაღების მისაღებად, დააჭირეთ დიაგრამაზე სვეტზე. პროგრამა გახსნის და შემდეგ გვიჩვენებს ცხრილში ველის გვერდით პირველად გასაღებს. ასევე იქმნება გარე გასაღები, ოღონდ უნდა აირჩიოთ **Foreign Key**.

ხედების, პროცედურების და ფუნქციების შექმნისათვის ინსტრუმენტების პანელიდან აირჩიეთ შესაბამისი ინსტრუმენტი და დააჭირეთ დიაგრამას. შემდეგ აირჩიეთ სქემა და ისევ დააჭირეთ დიაგრამას.

მონაცემთა მოდელების ვიზუალიზაცია.

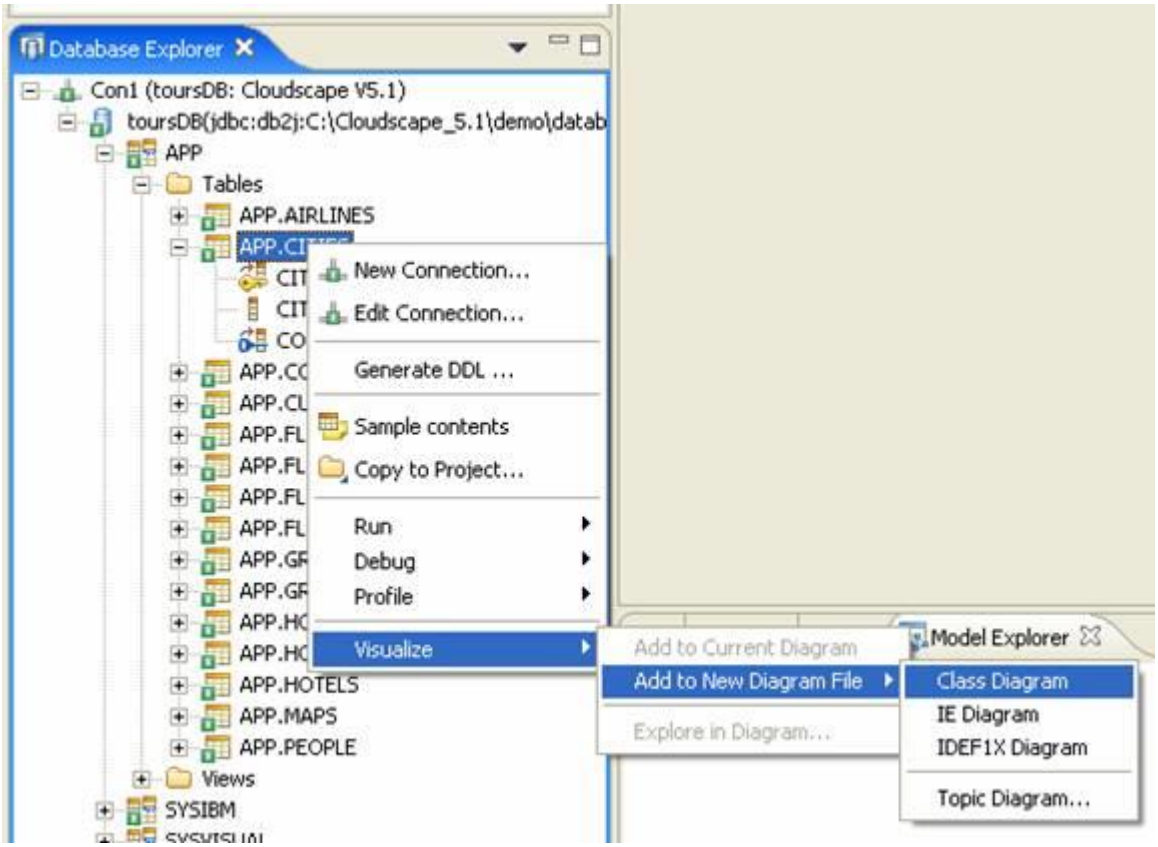
შესაძლებელია ლოკალური და დაშორებული მონაცემთა ბაზების ელემენტების ვიზუალიზაცია. ამისათვის უნდა იყოს დაყენებული JBDC-კავშირი სერვერთან. ნახაზზე მოყვანილია ასეთი კავშირი:



ნახ. 398

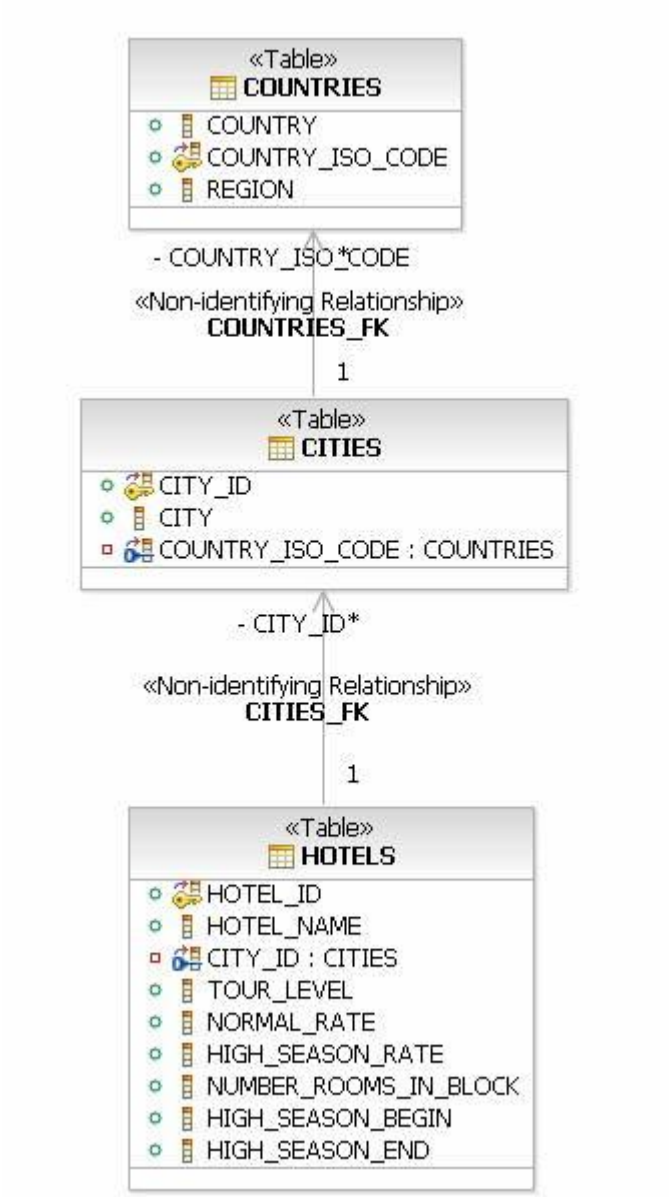
1. **Data server view** -ს დააჭირეთ მაუსის მარჯვენა ღილაკს და აირჩიეთ **New connection** (ახალი კავშირი).
2. შემდეგ აირჩიეთ **Choose a database manager and JDBC driver** და **Next.**
3. ზემოთ მოყვანილი ნახატის მიხედვით შეავსეთ ფორმა.
4. **JDBC driver** -ისთვის აირჩიეთ **Cloudscape Embedded Driver.**
5. დააჭირეთ **Browse** ღილაკს და აირჩიეთ მონაცემთა ბაზის ადგილმდებარეობა **(Database Location).** ჩვენი მაგალითისათვის: **C:\Cloudscape_5.1\demo\databases\toursDB.**
6. აირჩიეთ კლასის მდებარეობის დრაივერი: **C:\Cloudscape_5.1\lib\db2j.jar.**
7. **Finish**

მონაცემთა ბაზა **toursDB** შეივსება კავშირების საქაღალდის მეშვეობით. გახსენით სქემა, აირჩიეთ ცხრილი, რომელსაც სჭირდება ვიზუალიზაცია, დააჭირეთ მაუსის მარჯვენა ღილაკს, **Visualize -> Add to New Diagram File -> Class diagram**. ასეთნაირად შეიქმნება ცხრილის ასლი ახალი პროექტის კლას-დიაგრამაში:



ნახ. 399

იმისათვის, რომ ვნახოთ ყველა დაკავშირებული ცხრილი, ავირჩიოთ ძირითადი ცხრილი, შემდეგ კონტექსტური მენიუდან **Filters -> Show Related Elements**. ყველა ცხრილი, რომელიც გარე გასაღებით არის დაკავშირებული ამ ძირითად ცხრილთან და ყველა გარე გასაღები, აისახება დიაგრამაზე:



ნახ. 400

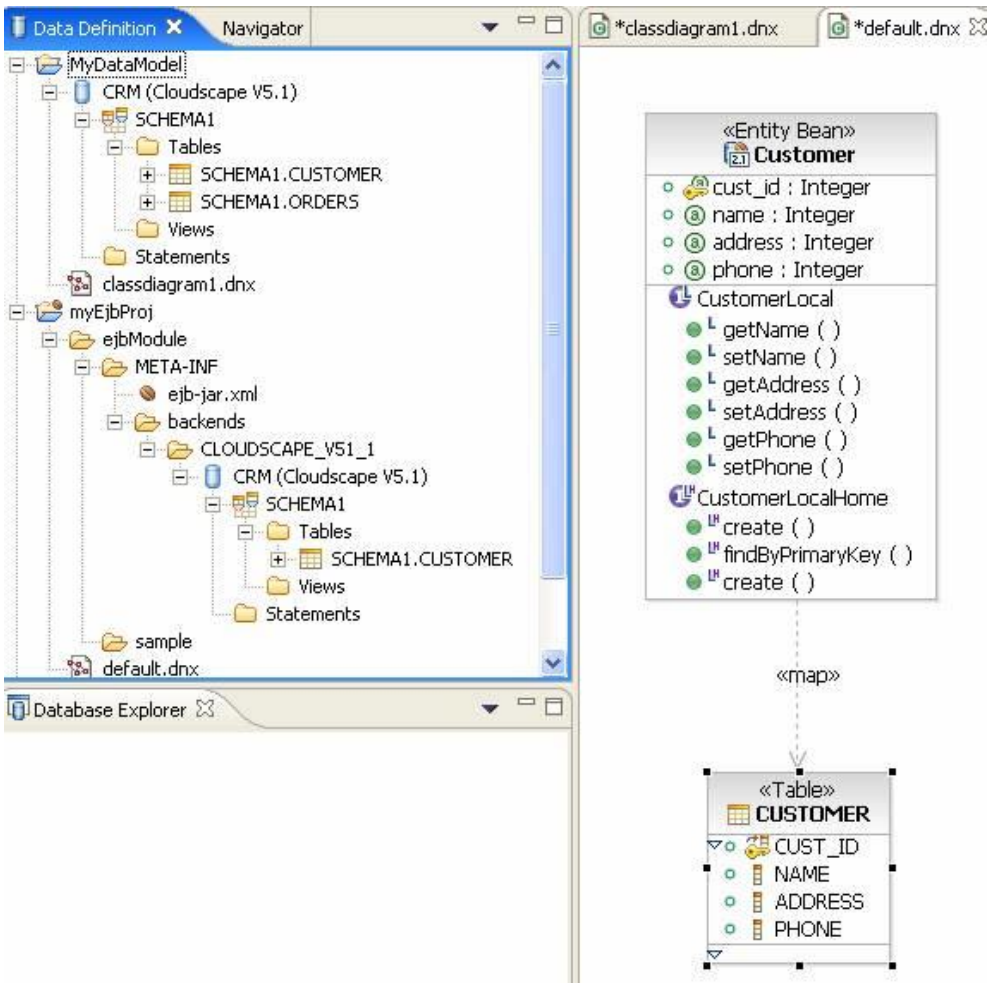
ელემენტის თვისებების დასათვალიერებლად, მაუსის მარჯვენა ღილაკით გამოიძახეთ კონტექსტური მენიუ და აირჩიეთ **Show Properties View. თვისებების შესაცვლელად გახსენით ელემენტი რედაქტორით და იქ მოახდინეთ განახლება.**

მონაცემთა მოდელის ელემენტების კორექტირება შესაძლებელია დიაგრამაზე. მაგალითად, ახალი სვეტის დამატება და ა.შ. მაგრამ ხშირად ამისათვის საჭიროა რედაქტორში შესვლა. ცხრილზე მაუსის ორჯერ დაჭერით გაიხსნება რედაქტორი. ყველა ცვლილება აისახება დიაგრამაზე. დიაგრამის ელემენტის შესაცვლელად რედაქტორი უნდა დაიხუროს.

ცხრილის გარდაქმნა EJB -ში (Enterprise JavaBeans™ (EJB™) .

როდესაც ცხრილი გარდაიქმნება **EJB -ში ,** იქმნება პროექტი, და შემდეგ მიმდინარე ცხრილი კოპირდება გარდასაქმნელ დირექტორიაში. ამისათვის შეასრულეთ შემდეგი მოქმედებები:

1. აირჩიეთ (Select) ცხრილი დიაგრამაზე;
2. კონტექსტური მენიუდან აირჩიეთ (**Create EJB from Table**).
3. შექმენით ახალი **EJB** პროექტი (**EJB project**), მაგალითად **myEjbproj.**
4. შექმენით ახალი სერვერი **(target server);**
5. **Finish**

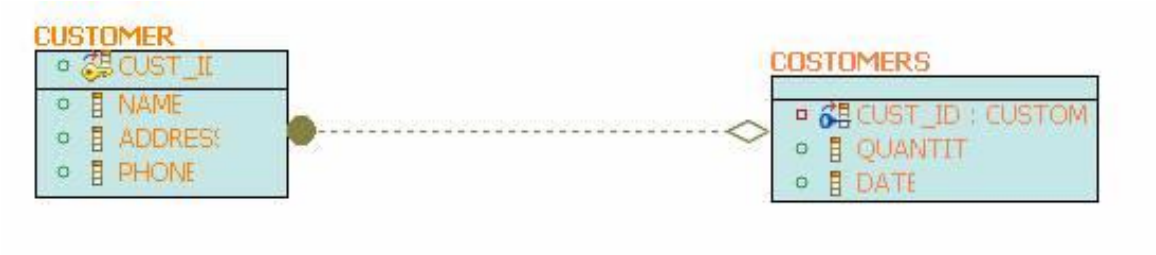


ნახ. 401

IDEf1X- და IE-დიაგრამები

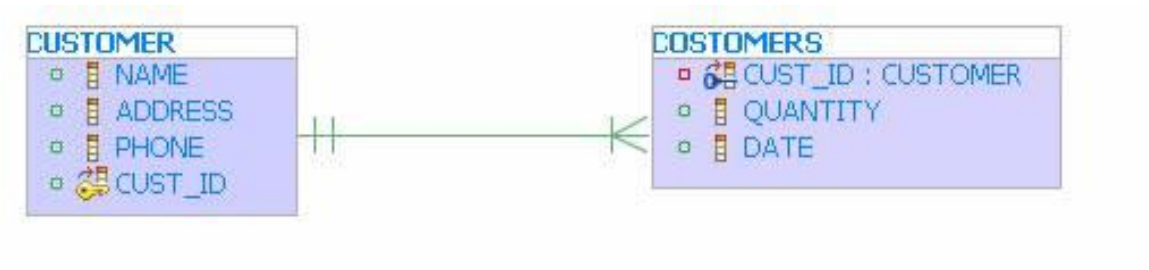
აქამდე ფორმები იყო ნაჩვენები მხოლოდ UML-ნოტაციაში, მაგრამ შესაძლებელია ამ ფორმების შექმნა **IDEf1X- და IE** - ნოტაციებშიც. დიაგრამების შექმნის დროს ხელმისაწვდომია 3 ოპერაცია: დიაგრამების კლასი, **IE**- და **IDEf1X**- დიაგრამები:

IE-დიაგრამა:

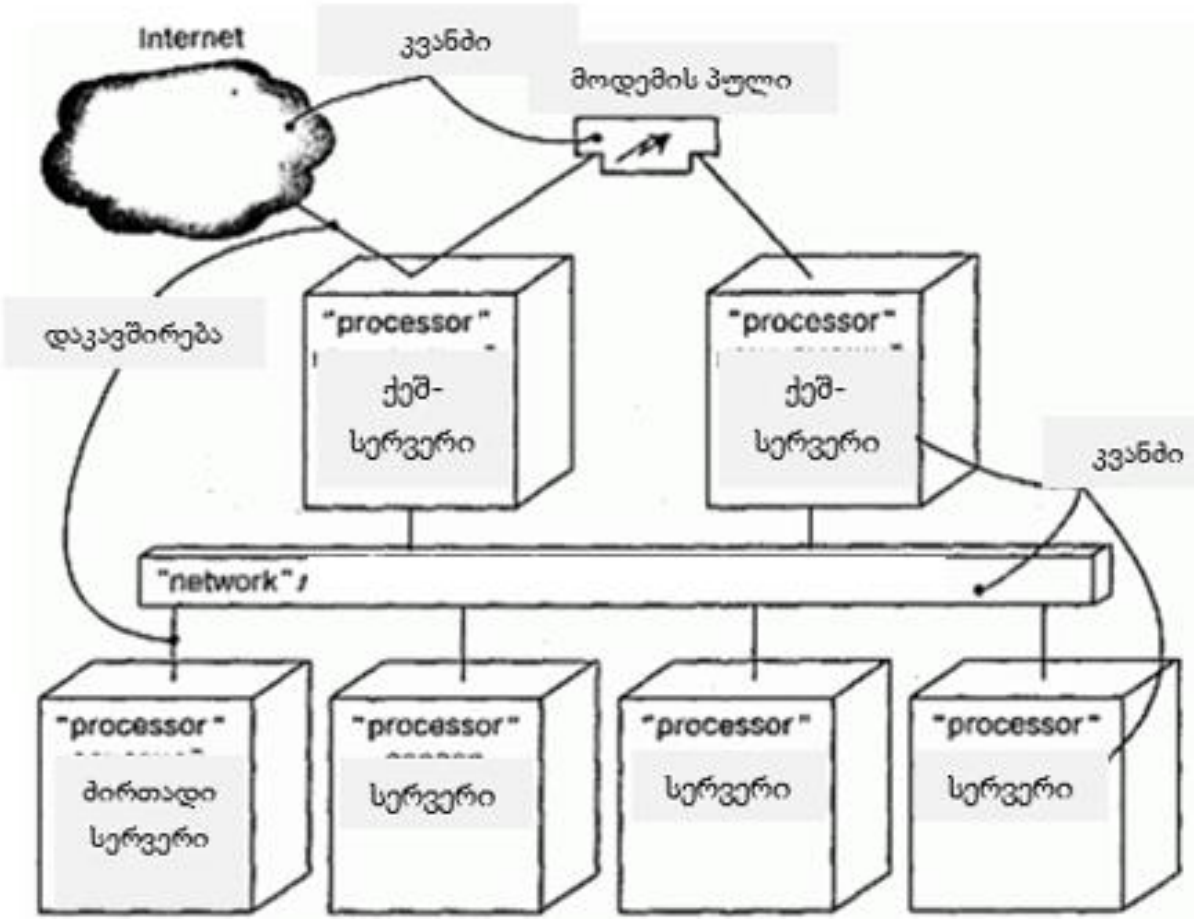


ნახ. 402

IDEf1X-დიაგრამა:



ნახ. 403



ნახ. 409

7.7. ინტერნეტის ქსელის ვიზუალური აგება

ინტერნეტ-ქსელის ვიზუალური აგებისათვის გამოიყენება განშლადი დიაგრამები, რომლებიც ასახავენ კვანძების კონფიგურაციას, რომლებშიც ხდება ინფორმაციის დამუშავება და ასევე კვანძების კომპონენტების ადგილმდებარეობას.

განშლადი დიაგრამები გამოიყენება პირდაპირი და უკუ-დაპროგრამების შემთხვევაში. UML-ში განშლადი დიაგრამები ასახავენ ფიზიკური კვანძების სტატიკურ ასპექტებს და სხვა დეტალებს, რომლებიც ეხება სისტემის კონსტრუირებას.

განშლადი დიაგრამა

განშლადი დიაგრამა განსხვავდება სხვა დიაგრამებისგან სპეციფიური შინაარსით. ამ დიაგრამის გამოყენება სამ შემთხვევაში ხდება:

- **ჩაშენებული (embedded)** სისტემების მოდელირების დროს. ისინი მართავენ ძრავებს, დისპლეებს და თვითონ კი იმართებიან გარე მოწყობილობებით, მაგალითად, ტემპერატურის ან მოძრაობის პირველადი გარდამქმნელით.
- **კლიენტ-სერვერის (client/server)** ტიპის სისტემების მოდელირებისათვის. აქ დიდი ყურადღება ეთმობა მოვალეობის განაწილებას მომხმარებელთა ინტერფეისისა და სერვერზე შენახულ მონაცემთა შორის. დაგეგმვა მდგომარეობს ამ ორ კომპონენტს შორის ურთიერთობების დამყარაბაში და ქსელის აგებაში.
- **სრულად განაწილებილი სისტემების (fully distributed)** მოდელირებისათვის. ეს სისტემები შეიცავენ ფართოდ განაწილებულ სხვადასხვა დონის სერვერებს. ასეთი სისტემების დაპროექტება ითხოვს ტოპოლოგიის უწყვეტ შეცვლას. ამ ტოპოლოგიის ვიზუალიზაციას ახორციელებს განშლადი დიაგრამა.

ჩაშენებული სისტემის მოდელირება შეიძლება დავყოთ შემდეგ ეტაპებად:

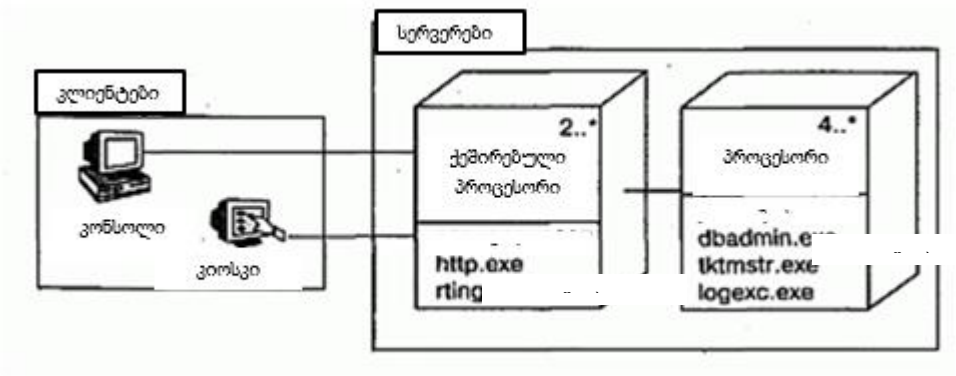
1. სისტემის უნიკალური კვანძებისა და მოწყობილობების იდენტიფიცირება.
2. არასტანდარტული მოწყობილობებისათვის შეიმუშავეთ პირობითი აღნიშვნები. ამის განხორციელება შესაძლებელია UML-გაფართოების მექანიზმით - სპეციფიკური სტერეოტიპებისათვის პიქტოგრამების არჩევით. აუცილებელია განვასხვავოთ პროცესორები, რომლებზეც პროგრამები ინახება იმ მოწყობილობებისაგან, რომლებზეც აბსტრაქციის ამ ეტაპზე არ იმყოფება არც ერთი პროგრამული კომპონენტი.
3. დიაგრამაზე აჩვენეთ ურთიერთობა პროცესორებს შორის.
4. გაუკეთეთ „ინტელექტუალური“ მოწყობილობების მუშაობის დეტალური აღწერა.

კლიენტ-სერვერის სისტემის მოდელირების დროს დიდი ყურედღება უნდა დაეთმოს კომპონენტების ფიზიკურ განლაგებას და მათ შორის ბალანსის მიღწევას. მაგალითად, უმეტესი სისტემების არქიტექტურა - სამდონიანია - მომხმარებლის ინტერფეისი, ბიზნეს-ლოგიკა და მონაცემთა ბაზა. ეს სამივე დაშორებულები არიან ერთმანეთისაგან. სად მოთავსდება ინტერფეისი და მონაცამთა ბაზა - ადვილად გადასაწყვეტი საკითხია, ხოლო ბიზნეს-ლიგიკის კომპონენტების ადგილმდებარეობა წარმოადგენს რთულ საკითხს.

კლიენტ-სერვერის სისტემის მოდელირება მიმდინარეობს შემდეგნაირად:

1. კლიენტისა და სერვერის პროცესორების კვანძებისა და მოწყობილობების იდენ-ტიფიცირება.
2. გამოარჩიეთ მოწყობილობები, რომლებიც სისტემაზე ზეგავლენას ახდენენ. მაგალითად, საკრედიტო ბარათების წამკითხავი მოწყობილობების მოდელი-რებისავის მიზანშეწონილია ასახვის მოწყობილობები არ იყოს სტანდარტული მონი-ტორების მსგავსი.
3. სტერეოტიპების გამოყენებით დაამუშავეთ ვიზუალური გამოსახულებები პროცე-სორებისათვის და სხვა საჭირო მოწყობილობებისათვის.
4. განშლად დიაგრამაზე მოახდინეთ ტოპოლოგიის კვანძების მოდელირება.

ნახაზზე მოყვანილია კლასიკური კლიენტ-სერვერის სისტემის ტოპოლოგია. ნაჩვენებია, პაკეტების გამოყენება (კლიენტი და სერვერი). პაკეტი კლიენტი შეიცავს ორ კვანძს: კონსოლი და კიოსკი. მათ საკუთარი სტერეოტიპები გააჩნია და ამიტომაც ვიზუალურად განსხვავდებიან. სერვერ-პაკეტი შეიცავს ორ კვანძს: ქეიშირებულ სერვერს და სერვერს. თითოეულისათვის გაწერილია კვანძში შემავალი კომპონენტები. თითოეული სერვერისათვის მითითებულია ჯერადობა, ანუ ეგზემპლიარების რაოდენობა გაშლილ კოფიგურაციაში. ამ ნახაზზე ჩანს, რომ ქეშირებული სერვერების რაოდენობა 2-ზე მეტი უნდა იყოს.



ნახ. 410

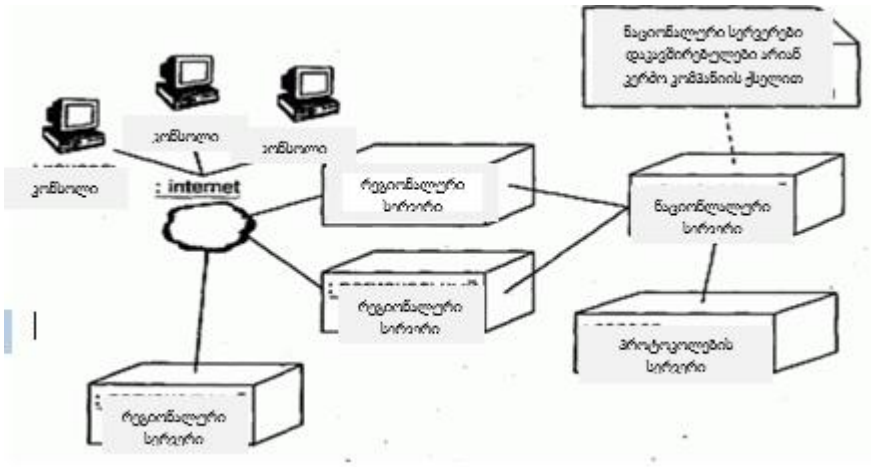
სრულად განაწილებული სისტემები შეიძლება იყონ სრულიად განსხვავებულები - მარტივი ორპროცესორიანიდან - მრავალ გეოგრაფიულ კვანძებად გაშლილ სისტემამდე. ეს ბოლო სისტემები - სტატიკურია. კვანძები ემატება და იშლება ქსელური ტრაფიკის მიხედვით; იქმნება ახალი, სწრაფი არხები ძველები კი დემონტაჟს ექვემდებარებიან და ა.შ. იცვლება არა მხოლოდ სისტემის ტოპოლოგია, არამედ პროგრამული კომპონენტების განაწილებაც.

სრულად განაწილებული სისტემის ვიზუალიზაცია ეხმარება ადმინისტრატორს გამოთვლითი მოწყობილობების აღრიცხვაში. ამისათვის იყენებენ UML -ის განშლად დიაგრამებს.

სრულად განაწილებული სისტემის მოდელირებისათვის საჭიროა:

1. კლიენტ-სერვერის სისტემის პროცესორების და კვანძებისა იდენტიფიცირება.
2. საკმაოდ ღრმა დეტალიზაცია.
3. ყურადღება მიაქციეთ კვანძების ლოგიკურ დაჯგუფებას.
4. განშლადი დიაგრამების მეშვეობით მოახდინეთ მოწყობილობების და პროცესორების მოდელირება. ყველგან, სადაც შესაძლებელია, გამოიყენეთ ინსტრუმენტები ქსელური ტოპოლოგიის საჩვენებლად.
5. თუ საჭიროა დინამიკის ჩვენება - გამოიყენეთ პრეცედენტების დიაგრამა და ურთიერთობების დიაგრამა.

ნახ.406-ზე ნაჩვენებია სრულად განაწილებული სისტემის ტოპოლოგია. ეს დიაგრამა წარმოადგენს ობიექტების დიაგრამასაც, რადგანაც შეიცავს მხოლოდ ეგზემლარებს. ნაჩვენებია სამი კონსოლი (კონსოლის სტერეოტიპის ანონიმური ეგზემპლიარები), რომლებიც დაკავშირებულნი არიან INTERNET-თან. და მეორე მხრიდან, არსებობს 3 რეგიონალური სერვერის ეგზემლარი, რომლებიც უკავშირდებიან ნაციონალურ სერვერებს. ნაციონალური სერვერებიც უკავშირდებიან ერთმანეთს, მაგრამ დიაგრამაზე ეს ასახული არ არის.



ნახ. 411

დავალება

1. *შექმენით სამი UML-დიაგრამა ვიდეო-გაქირავების მარკეტის მაგალითზე. დიაგრამები:*
- *კლასის*
- *პრეცედენტების*
- *საკუთარი სურვილით*

2. *შექმენით სამი UML-დიაგრამა ბანკომატის მაგალითზე. დიაგრამები:*
- *აქტივობის*
- *კლას-დიაგრამა*
- *საკუთარი სურვილით*

3. *შექმენით სამი UML-დიაგრამა ბანკის მუშაობის მაგალითზე. დიაგრამები:*
- *პრეცედენტების*
- *კლას-დიაგრამა*
- *საკუთარი სურვილით*

4. *შექმენით სამი UML-დიაგრამა ბიბლიოთეკის მაგალითზე. დიაგრამები:*
- *კომპონენტების*
- *განშლადი*
- *საკუთარი სურვილით*

5. *შექმენით სამი UML-დიაგრამა ტელესისტემის მუშაობის (ტელევიზორი, არხები) მაგალითზე. დიაგრამები:*
- *თანამიმდევრობების*
- *კოოპერაციის*
- *საკუთარი სურვილით*

6. *შექმენით სამი UML-დიაგრამა ბილეთების დაჯავშნის მაგალითზე. დიაგრამები:*
- *კომპონენტების*
- *კლას-დიაგრამა*
- *საკუთარი სურვილით*

7. *შექმენით სამი UML-დიაგრამა სარეცხი მანქანის მაგალითზე. დიაგრამები:*
- *პრეცედენტების*
- *კლას-დიაგრამა*
- *საკუთარი სურვილით*

8. *შექმენით სამი UML-დიაგრამა კომპიუტერზე მუშაობის მაგალითზე. დიაგრამები:*
- *აქტივობის*
- *პრეცედენტების*
- *საკუთარი სურვილით*

9. *შექმენით სამი UML-დიაგრამა მაღაზიაში წასვლის მაგალითზე. დიაგრამები:*
- *პრეცედენტების*
- *აქტიურობის*
- *საკუთარი სურვილით*

10. *შექმენით სამი UML-დიაგრამა სამშენებლო კომპანიის მუშაობის მაგალითზე. დიაგრამები:*

- მდგომარეობის
- კლას-დიაგრამა
- საკუთარი სურვილით

11. შექმენით სამი UML-დიაგრამა საავადმყოფის მუშაობის მაგალითზე. დიაგრამები:
- აქტივობის
- კლას-დიაგრამა
- საკუთარი სურვილით

12. შექმენით სამი UML-დიაგრამა სპორტული შეჯიბრებების ორგანიზების მაგალითზე. დიაგრამები:
- თანამიმდევრობის
- კოოპერაციის
- საკუთარი სურვილით

13. შექმენით სამი UML-დიაგრამა ავტომანქანის არჩევის მაგალითზე. დიაგრამები:
- პრეცედენტების
- მდგომარეობის
- საკუთარი სურვილით

14. შექმენით სამი UML-დიაგრამა უმაღლეს დაწესებულებაში სწავლის მაგალითზე. დიაგრამები:
- კომპონენტების
- კლას-დიაგრამა
- საკუთარი სურვილით

15. შექმენით სამი UML-დიაგრამა სამუშაოს არჩევის მაგალითზე. დიაგრამები:
- მდგომარეობის
- პრეცედენტების
- საკუთარი სურვილით

16. შექმენით სამი UML-დიაგრამა სასტუმროს მუშაობის მაგალითზე. დიაგრამები:
- აქტივობის
- კლას-დიაგრამა
- საკუთარი სურვილით

17. შექმენით სამი UML-დიაგრამა აეროპორტის მუშაობის მაგალითზე. დიაგრამები:
- თანამიმდევრობის
- აქტივობის
- საკუთარი სურვილით

18. შექმენით სამი UML-დიაგრამა სილამაზის სალონის მუშაობის მაგალითზე. დიაგრამები:
- პრეცედენტები
- კლას-დიაგრამა
- საკუთარი სურვილით

19. შექმენით სამი UML-დიაგრამა რესტორანის მუშაობის მაგალითზე. დიაგრამები:
- აქტივობის

- *კლას-დიაგრამა*
- *საკუთარი სურვილით*

20. შექმენით სამი UML-დიაგრამა კომპიუტერის არქიტექტურის მაგალითზე. დიაგრამები:
- *პრეცედენტის*
- *კლას-დიაგრამა*
- *საკუთარი სურვილით*

8. პროგრამული დიზაინის ნიმუშების შედგენა

8.1. პროგრამული დიზაინის ნიმუშის არსი და დანიშნულება

პროგრამული უზრუნველყოფის შემუშავებისას ხშირად გვხვდება ერთიდაიგივე პრობლემები და ამოცანები. ხშირად ეს არც იმაზეა დამოკიდებული. რომელ ენაზე იწერება მოცემული პროგრამა. ასეთი პრობლემებისათვის არსებობს გადაწყვეტის საშუალებები, ე.წ. პროგრამული დიზაინის ნიმუშები. პროგრამირების შაბლონი - ეს არის ენისგან დამოუკიდებელი სტრატეგია პრობლემის გადასაწყვეტად ობიექტურ-ორიენტირებული დაპროგრამების მეთოდების გამოყენებით.

ჩვეულებრივ შაბლონი არ წარმოადგენს დასრულებულ ნიმუშს, რომელიც პირდაპირ უნდა გარდაიქმნას პროგრამულ კოდში. ეს არის მხოლოდ ამოცანის გადაწყვეტის გზა, რომელიც შესაძლებელია გამოვიყენოთ სხვადასხვა სიტუაციაში.

პროგრამული დიზაინის ნიმუშების არცოდნა არ ნიშნავს ცუდ პროგრამისტობას. ხშირად პროგრამისტები თვითონ პოულობენ საპროექტო გადაწყვეტას, სწორეს ეს არის შაბლონი, დიზაინის ნიმუში.

.

რატომ უნდა გამოვიყენოთ პროგრამული დიზაინის ნიმუში?

- ყოველი ნიმუში აღწერს პრობლემების მთელ კლასს;
- ყოველ ნიმუშს გააჩნია სახელი;
- ამარტივებს პროგრამული უზრუნველყოფის დეველოპერებს შორის ურთიერთო-ბას;
- სწორად ფორმულირებული ნიმუშის გამოყენება შესაძლებელია მრავალჯერ;
- პროექტირების ნიმუშები არ არის დამოკიდებული დაპროგრამების ენაზე.

ამრიგად დიზაინის ნიმუში - ეს არის დაპროგრამების სტანდარტული პრობლემის სტანდარტული გადაწყვეტა.

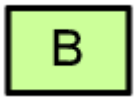
ნიმუში შედგება ოთხი ძირითადი ელემენტისაგან:

1. სახელი - დიზაინის ნიმუშებზე სახელების მინიჭება საშუალებას გვაძლევს პროექტირება მოხდეს აბსტრაქციის უფრო მაღალ დონეზე. ნიმუშების ლექსიკონი ამარტივებს ურთიერთობას გადასაცემი ინფორმაციის რაოდენობის შემცირების ხარჯზე, ასევე საშუალებას იძლევა მარტივად წარმოვიდგინოთ დიზაინის სისტემა.

2. ამოცანა - აღწერა, თუ როდის არის მიზანშეწონილი შაბლონის გამოყენება. ახდენს ამოცანის და მისი კონტექსტის ფორმულირებას. აღწერს დაპროექტების კონკრეტულ პრობლემას, მაგალითად ალგორითმების წარმოდგენის საშულებას ობიექტების სახით. ზოგჯერ აღინიშნება, სტრუქტურის როგორი ობიექტები ქმნის მოუქნელ დიზაინს. ასევე შესაძლებელია ჩამოთვლილი იყო პირობები, რომელთა შესრულებისას მიზანშეწონილია შაბლონის გამოყენება. უნდა ვიცოდეთ რომელი შაბლონი, როგორ და რა პირობებში უნდა გამოვიყენოთ.
3. გადაწყვეტა - დიზაინის ელემენტების და მათ შორის დამოკიდებულების აღწერა. მხედველობაში არ მიიღება კონკრეტული დიზაინი ან რეალიზაცია, რადგანაც ეს ნიმუშია, რომელიც გამოყენებულ უნდა იქნეს სხვადასხვა სიტუაციებში. უბრალოდ მოცემულია პროქტირების ამოცანების აბსტრაქტული აღწერა და ასევე, როგორ შეიძლება ეს ამოცანა გადაწყდეს;
4. შედეგები: ეს არის ნიმუშის და სხვადასხვა კომპრომისების გამოყენების შედეგი. საპროქტო გადაწყვეტისას შედეგებზე ხშირად არც არის საუბარი, თუმცა მის შესახებ ინფორმაცია საჭიროა და აუცილებელი, იმისათვის, რომ სხვადასხვა ვარიანტებს შორის ამორჩევის საშუალება გვქონდეს და შევაფასოთ მოცემული ნიმუშის დადებითი და უარყოფითი მხარეები. ამ შემთხვევაში ასევე მნიშვნელოვანია რეალიზაციისათვის დაპროგრამების ენის შერჩევა.

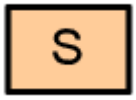
8.1.1. ნიმუშების კლასიფიკაცია

პროექტირების ნიმუშები იყოფა რამდენიმე ჯგუფად იმისდამიხედვით, რომელი ჯგუფის ამოცანას წყვეტს ის. გამოყოფენ შემოქმედებით (Creational), სტრუქტურულ (Structual) და ქცევითი (Behavioral) დიზაინის ნიმუშებს.



ქცევითი (Behavioral)

შემოქმედებითი (Creational)

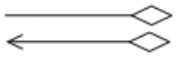
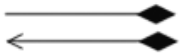


სტრუქტურული (Structual)

ობიექტურ-ორიენტირებული ნიმუშები გვიჩვენებენ დამოკიდებულებას და კავშირს კლასებს და ობიექტებს შორის.

8.1.1.2. პირობითი აღნიშვნები

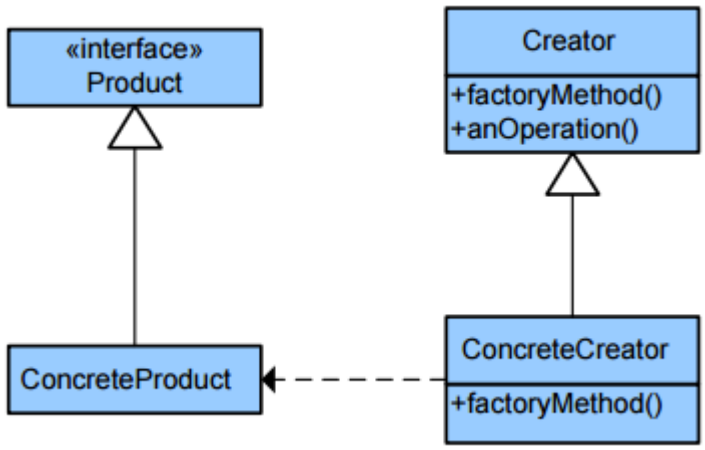
დამოკიდებულება კლასებს შორის

-  — აგრეგაცია **(aggregation)** — აღწერს კავშირს «ნაწილი»–«მთელი», რომელშიც „ნაწილი“ შესაძლებელია განხორციელდეს ცალკე „მთელის“-გან. რომბი იხაზება „მთელის“ მხარეს.
-  — კომპოზიცია **(composition)** — აგრეგაციის ქვესახე, რომელშიც „ნაწილი“ არ შეიძლება განხორციელდეს „მთელის“ გარეშე.
-  — დამოკიდებულება **(dependency)** — ერთის (დამოუკიდებელი) ცვლილებამ შესაძლებელია გავლენა მოახდინოს მეორის (დამოკიდებული) მდგომარეობაზე ან ქცევაზე. ისრის მხარე მიუთითებს დამოუკიდებელ კლასს.
-  — განზოგადება **(generalization)** — ინტერფეისის რეალიზაციის ან მემკვიდრეობითობის დამოკიდებულება. ისრის მხარეს მდებარეობს სუპერკლასი ან ინტერფეისი.

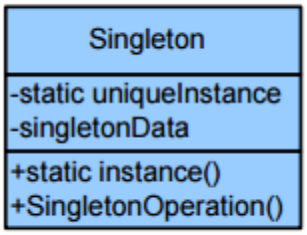
8.2. ამოცანის პროექტირება შემოქმედებითი დიზაინის (Creational Design) ნიმუშების, ნიმუშების მეშვეობით

შემოქმედებითი ნიმუში საშუალებას იძლევა ობიექტის წარმოდგენის, კომპოზიციის და შექმნის საშულებებისგან დამოუკიდებელი გახადოს სისტემა. ნიმუშის მაგალითებია:

8.2.1. აბსტრაქტული ფაბრიკა (Abstract Factory, Factory)

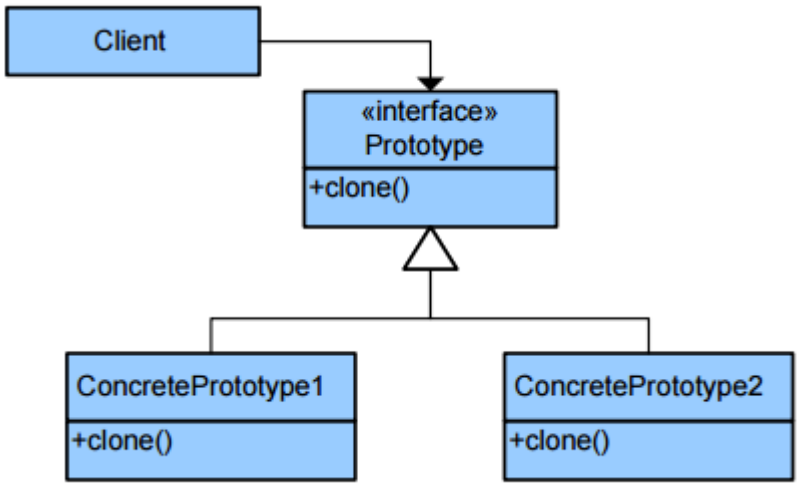
პრობლემა	ურთიერთდაკავშირებული ან ურთიერთდამოკიდებული ობეიქტების ოჯახის შექმნა
გადაწყვეტა	აბსტრაქტული კლასის შექმნა, რომელშიც გამოცხადებულია ინტერფეისი კონკრეტული კლასების შესაქმნელად. 
მაგალითი	რომელი კლასი უნდა იყოს პასუხისმგებელი ობიექტ-ადაპტერის შექმნაზე „ადაპტერ“ ნიმუშის გამოყენებით. თუ ასეთი ობიექტები იქმნება, როგორც რომელიმე ობიექტის დაქვემდებარებული, ის დაარღვევს გამიჯვნის ვალდებულებების პრინციპს.
უპირატესობა	ახდენს კონკრეტული კლასის იზოლირებას. რამდენადაც „აბსტრაქტული ფაბრიკა“ ახდენს კლასის შექმნაზე და შექმნის პორცესზე პასუხისმგებლობის ინკაფსულირებას, ის ახდენს კლიენტის იზოლირებას კლასის დეტალების რეალიზაციისაგან. გამარტივებულია „აბსტრაქტული ფაბრიკის“ ცვლილება, რამდენადაც ის გამოიყენება დანართში მხოლოდ ერთხელ ინსტანცირებისას.
ნაკლოვანებები	ინტერფეისი „აბსტრაქტული ფაბრიკა“ აფიქსირებს ობიექტების ნაკრებს, რომლებიც შესაძლებელია შეიქმნას. „აბსტრაქტული ფაბრიკის“ გაფართოება ახალი ობიექტების დამზადებისათვის ხშირად გართულებულია

8.2.2. Singleton

პრობლემა	რომელ სპეციალურ კლასს შეუძლია შექმნას „აბსტრატქული ფაბრიკა“ და როგორ მივიღოთ მასთან წვდომა? ამისათვის აუცილებელია სპეციალური კლასის მხოლოდ ერთი ეგზემპლიარი, სხვადასხვა ობიექტებმა უნდა მიმართონ ამ ეგზემპლიარს ერთდადერთი წვდომის წერტილით.
გადაწყვეტა	შვქმნათ კლასი და განვსაზღვროთ კლასის სტატისტიკური მეთოდი, რომელიც აბრუნებს ერთადერთ ობიექტს. 
რეკომენდაციები	გონივრულია შევქმნათ სპეციალური კლასის სტატიკური ეგზემპლიარი, რამდენადაც ეგზემპლიარის მეთოდების გამოყენებისას შესაძლებელია მემკვიდრეობითობის მექანიზმის გამოყენება და ქვეკლასების შექმნა. სტატიკური მეთოდები დაპროგრამების ენებში არა პოლიმორფულია და არ უშვებს წარმოებულ კლასებში გადაფარვას. ეგზემპლიარის შექმნის საფუძველზე გადაწყვეტა წარმოადგენს უფრო მოქნილ გზას, რამდენადაც შედეგად შესაძლებელია მოთხოვნილ იქნეს კლასის არა ერთადერთი ეგზემპლიარი, არამედ რამდენიმე.

8.2.3. პროტოტიპი (Prototype)

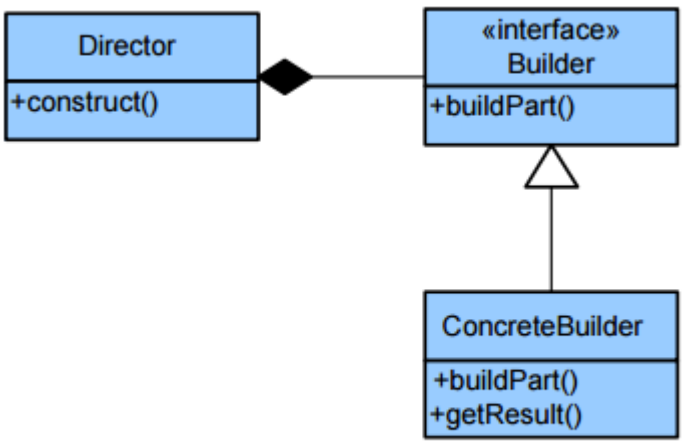
პრობლემა	სისტემა არ უნდა ჩამოეკიდოს იმის გამო, როგორ იქმნება, იწყობა და წარმოსდგება ობიექტები.
გადაწყვეტა	შვქმნათ ახალი ობიექტები ნიმუშის პროტოტიპის გამოყენებით. „პროტოტიპი“ აცხადებს ინტერფეისს საკუთარი თავის კლონირებისათვის. „კლიენტი“ ქმნის ახალ ობიექტს, მიმართავს „პროტოტიპს“ - მოთხოვნით: მიიღოს „პროტოტიპის“ კლონი.



8.2.4. კლასის ეგზემპლიარის შემქმნელი (Creator)

პრობლემა	„ვინ“ უნდა იყოს პასუხისმგებელი კლასის „ეგზემპლიარის“ შექმნაზე
გადაწყვეტა	დავუნიშნოთ კლასს ვალდებულება შექმნას სხვა A კლასის ობიექტები
რეკომენდაცია	ლოგიკურია გამოვიყენოთ ნიმუში, თუ კლასი B შეიცავს, აქტიურად გამოიყენებს და ა.შ, A კლასის ობიექტებს.
მაგალითი	აუცილებელია განვსაზღვროთ, რომელი ობიექტი აგებს პასუხს „საქონელი გაყიდავა“ ეგზემპლიარის შექმნაზე. ლოგიკურია, რომ ეს არის ობიექტი „გაყიდვა“, რამდენადაც ის შეიცავს „საქონელი-გაყიდვა“ რამდენიმე ობიექტს.
უპირატესობა	ამ ნიმუშის გამოყენებით არ იზრდება თანხვედრა, რამდენადაც შექმნილი კლასი, როგორც წესი, ხილულია მხოლოდ კლასი - შემქმნელისათვის.
ნაკლოვანება	თუ პროცედურა ობეიქტის შექმნა საკმარისად რთულია (მაგალითად სრულდება გარე პირობების საფუძველზე), ლოგიკურია „აბსტრაქტული ფაბრიკა“ ნიმუშის გამოყენება.

8.2.5. მშენებელი (Builder)

პრობლემა	რთული ობიექტის კონსტრუირების პროცესის განცალკევება მის წარდგენასთან, ისე, რომ ერთი და იგივე ობიექტის კონსტრუირებისას შესაძლებელი იყოს განსხვავებული წარდგენა. რთული ობიექტის შექმნის ალგორითმი არ უნდა იყოს დამოკიდებული იმაზე, თუ რა ნაწილებისგან შედგება ობიექტი და როგორია მათი თანხვედრა.
გადაწყვეტა	„კლიენტი“ ქმნის ობიექტს - გამანაწილებელს „დირექტორი“ და კონფიგურირებას ახდენს ობიექტით - „მშენებელი“. „დირექტორი“ „მშენებელს“ გადასცემს ინფორმაციას, რომ უნდა აშენდეს „პროდუქტის“ მორიგი ნაწილი. „მშენებელი“ ამუშავებს „დირექტორის“ მოთხოვნას და ამატებს ახალ ნაწილს „პროდუქტს“, ამის შემდეგ „კლიენტი“ იღებს „პროდუქტს“ „მშენებლისგან“ 
უპირატესობა	ობიექტი „მშენებელი“ წარუდგენს ობიექტს „დირექტორი“ აბსტრაქტულ ინტერფეისს „პროდუქტის“ კონსტრუირებისათვის, რომლის უკანაც შეიძლება დაიმალოს როგორც პროდუქტის შიდა სტრუქტურა და წარდგენა, ასევე „პროდუქტის“ აწყობის პროცესი. „პროდუქტის“ შიდა წარდგენის ცვლილებისათვის საკმარისია განისაზღვროს „მშენებლის“ ახალი სახე. მოცემული ნიმუში ახდენს ობიექტის შექმნის და წარდგენის კოდის იზოლაციას

8.2.6. ფაბრიკული მეთოდი Factory Method და ვირტუალური კონსტრუქტორი (Virtual Constructor)

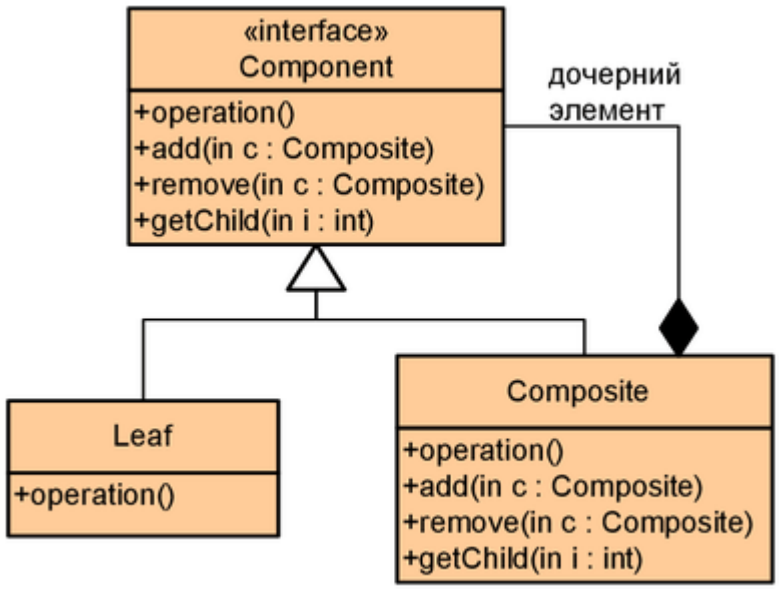
პრობლემა	ობიექტის შექმნის ინტერფეისის განსაზღვრა, მაგრამ ქვეკლასებისთვის, როგორი კლასის ინსტანცირება მოხდეს, ამ გადაწყვეტილების მიღების უფლების დატოვება
გადაწყვეტა	აბსტრაქტული კლასი „შემქმნელი“ აცხადებს ფაბრიკულ მეთოდს, რომელიც აბრუნებს „პროდუქტი“ ტიპის ობიექტს (აბსტრაქტული კლასი, ობიექტების განმსაზღვრელი ინტერფეისი). „შემქნელს“ ასევე შეუძლია განსაზღვროს გულისხმობის პრინციპით (Default) ფაბრიკული მეთოდის რეალიზაცია, რომელიც აბრუნებს „კონკრეტულ პროდუქტ“-ს. „შემქმნელი“ ეყრდნობა საკუთარ ქვეკლასებს ფაბრიკული მეთოდის განსაზღვრისას, რომელიც აბრუნებს „კონკრეტულ პროდუქტ“-ს.
უპირატესობა	დამპროექტებელს ათავისუფლებს აუცილებლობისაგან, რომ კოდში ჩააშენოს
ნაკლოვანებები	იქმნება ქვეკლასების დამატებითი დონეები

8.3. ამოცანის პროექტირება სტრუქტურული დიზაინის (Structural Design) ნიმუშების მეშვეობით

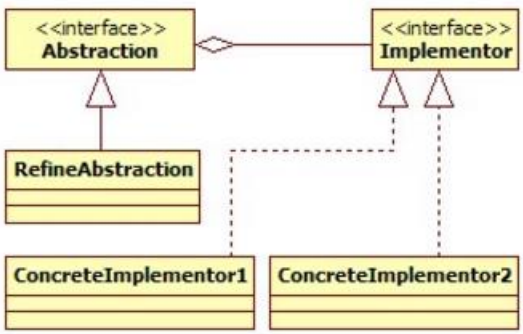
სტრუქტურულ ნიმუშებში განიხილება საკითხები, თუ როგორ წარმოიქმნება კლასებისა და ობიექტებისაგან უფრო მსხვილი სტრუქტურები. სტრუქტურული ნიმუშები კლასის დონედ იყენებენ მემკვიდრეობითობას ინტერფეისისა და რეალიზაციების კომპოზიციის შედგენით.

ინტერფეისების კომპოზიციის ან სტრუქტურული ნიმუშების რეალიზაციის ნაცვლად ობიექტის დონე კრებს ობიექტებს ახალი ფუნქციების მისაღებად. დამატებითი მოქნილობა ამ შემთხვევაში დაკავშირებულია შესრულების დროს ობიექტების კომპოზიციის ცვლილების შესაძლებლობასთან, რაც დაუშვებელია კლასის სტატიკური კომპოზიციისთვის.

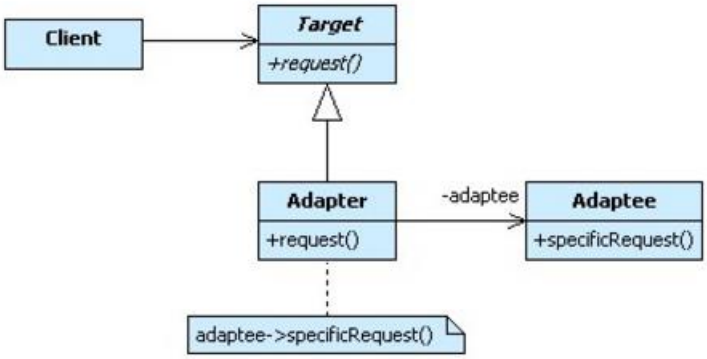
8.3.1. Composite კომბინირებული, შედგენილი

გამოყენება	გააერთიანოს ობიექტები ხისებრ სტრუქტურაში იერარქიის წარმოსადგენად. უზრუნველყოს შექმნილი სტრუქტურის ობიექტებთან თანაბარი წვდომა.
გადაწყვეტა	

8.3.2. ხიდი (Bridge)

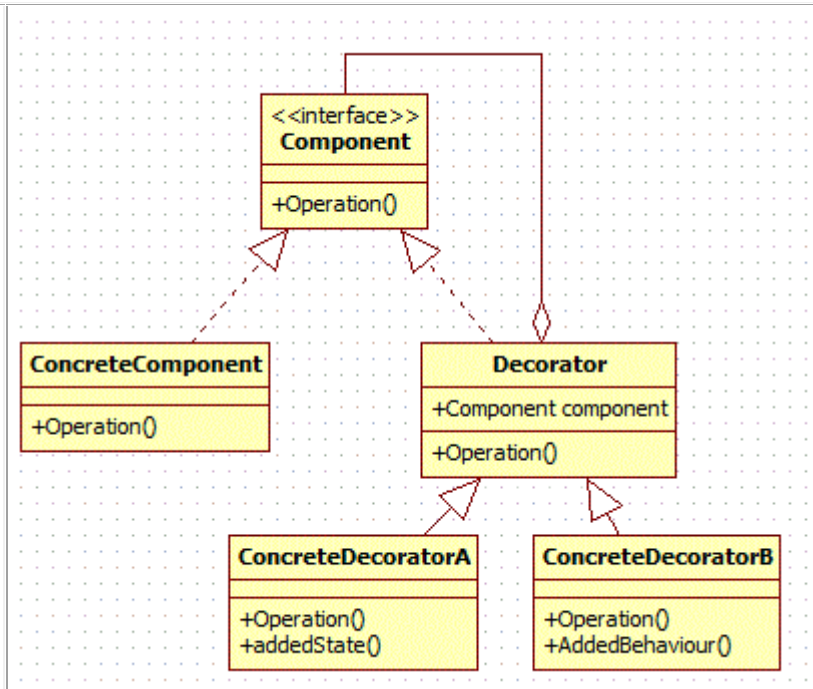
პრობლემა	საჭიროა აბსტრაქციის გამოყოფა რეალიზაციისაგან ისე, რომ ორივეში ცვლილებების შეტანა შესაძლებელი იყოს დამოუკიდებლად. მემკვიდრეობითობის გამოყენებისას რეალიზაცია მჭიდროს არის დაკავშირებული აბსტრაქციასთან, რაც ართულებს მოდიფიკაციის დამოუკიდებლად განხორციელებას.
გადაწყვეტა	აბსტრაქცია და რეალიზაცია განვათავსოთ კლასების სხვადასხვა იერარქიაში
Пример	"აბსტრაქცია“ (*Abstraction)* განსაზღვრავს „აბსტრაქციის“ ინტერეფისს და ინახავს მიმართვას „რეალიზაცია“(*Implementor)* ობიექტზე. „რეალიზაცია“ განსაზღვრავს ინტერფეისს რეალიზაციის კლასებისათვის, მაგრამ არ არის ვალდებული „აბსტრაცია“ კლასის ინტერფეისს შეესაბამებოდეს ზუსტად, ორივე ინტერფეისი შესაძლებელია იყოს განსხვავებული ერთმანეთისაგან. ჩვეულებრივ „რეალიზაცია“ კლასის ინტერფეისი წარმოადგენს მარტივ ოპერაციებს, ხოლო „აბსტრაქცია“ კლასი განსაზღვრავს უფრო მაღალი დონის ოპერაციებს, რომელთაც საფუძვლად სწორედ „რეალიზაცია“ კლასი მიერ განსაზღვრული მარტივი ოპერაციები უდევთ. „კონკრეტული რეალიზაცია“ (*Concrete Implementation)* შეიცავს სწორედ „რეალიზაცია“ კლასის კონკრეტულ რეალიზაციას. ობიექტი „აბსტრაქცია“ მომხმარებლის მოთხოვნებს გადასცემს „რეალიზაცია“ კლასის ობიექტს. 
უპირატესობა	აბსტრაქციის (Abstraction) და რეალიზაციის (Implementor) კონფიგურირება შესაძლებელია შესრულების დროს. ამასთან ერთად მათი გაყოფა საშუალებას გვაძლევს „რეალიზაცია“ კლასი შევცვალოთ, ისე რომ არ დაგვჭირდეს „აბსტრაქცია“ კლასის ხელმეორედ კომპილაცია.

8.3.3. ადაპტერი (Adapter)

პორბლემა	აუცილებელია უზრუნველყოფილ ქნეს არათავსებადი ინტერფეისების ურთიერთქმედება ან უნდა შეიქმნას ერთიანი მდგრადი ინტერფეისი სხვადასხვა ინტერფეისის მქონე რამდენიმე კომპონენტისათვის
გადაწყვეტა	შევასრულოთ კომპონენტის საწყისი ინტერფეისის კონვერტირება შუალედური ობიექტის - ადაპტერის გამოყენებით, სხვა სახის ინტერფეისად. ამისათვის დავამატოთ სპეციალური ობიექტი საერთო ინტერფეისით და დავამყაროთ კავშირი მისი საშუალებით ადაპტერთან.
მაგალითი	 . დიაგრამის მიხედვით, კლასს, რომლის ინტერფეისის ადაპტირებაც უნდა მოხდეს გარკვეულ სახემდე, ჰქვია Adaptee. ამოცანის შესასრულებლად ნიმუში Adapter-ს შემოაქვს კლასების შემდეგი იერარქია; • ვირტუალური საბაზო კლასი Target. აქ ხდება საჭირო სახის სამომხმარებლო ინტერფეისის გამოცხადება. მომხმარებლისთვის ხელმისაწვდომია მხოლოდ ეს ინტერფეისი. • წარმოებული კლასი Adapter, რომელიც ახდენს ინტერფეის Target-ის რეალიზაციას. ამ კლასში ასევე არის მიმმართველი ან მიმთითებელი Adaptee ეგზემპლიარზე. ნიმუში Adapter იყენება ამ მიმმართველს მომხმარებლის მოთხოვნების Adaptee-თან გადასამისამართებლად. რამდენადაც Adaptee და Target ინტერფეისები არათავსებადია, ეს გამოძახებები ჩვეულებრივ მოითხოვს შესაბამის გარდაქმნებს

8.3.4. დეკორატორი (Decorator)

პრობლემა	ცალკეულ ობიექტზე და არა მთელ კლასზე, დამატებითი ვალდებულებების დაკისრება. ობიექტს, რომელიც ასრულებს ძირითად ფუნქციებს, საჭიროა დაემატოს დამატებითი ფუნქციები, რომელიც შეიძლება შესრულდეს ობიექტის ძირითადი ფუნქციების შესრულებამდე, შესრულების შემდეგ ან სულაც მის ნაცვლად. .
გადაწყვეტა	დეკორატორი ითვალისწინებს ობიექტის ფუნქციების გაფართოებას ქვეკლასების განსაზღვრის გარეშე.
რეალიზაცია	იქმნება აბსტრაქტული კლასი, რომელიც წარმოადგენს, როგორც საწყის კლასს, ასევე ახალ კლასში დამატებულ ფუნქციებს. დეკორატორში ახალი ფუნქციის გამოძახება ხდება მოთხოვნილი თანმიმდევრობით. სურვილის შემთხვევაში შესაძლებელია საწყისი კლასის გამოყენება ფუნციონალური შესაძლებლობების გაფართოების გარეშე. დეკორატორის რეალიზაციისათვის საჭიროა: საბაზო კლასიდან მემკვიდრე კლასი-დეკორატორის შექმნა, საბაზო კლასში დავამატოთ მიმთითებელი დეკორატორზე, შვასრულოთ მითითება ობიექტზე, რომელსაც უნდა დაემატოს ახალი ფუნქციები, დეკორატორის კონსტრუქტორში. დეკორატორიდან მეთოდები მივმართოთ ობიექტზე, წინასწარ განვსაზღვროთ დეკორატორში მეთოდები, რომელთა ალგორითმები აუცილებელია შეიცვალოს.



8.3.5. არქიტექტურული სისტემური ნიმუშები

რეპოზიტორები

აღწერა	მონაცემები ინახება ცენტრალურ მონაცემთა ბაზაში, რომელიც ხელმისაწვდომია ყველა ქვესისტემისთვის. რეპოზიტორები წარმოადგენენ პასიურ ელემენტებს, ხოლო მათი მართვა ევალება ქვესისტემას.
რეკომენდაციები	ლოგიკურია მისი გამოყენება, თუ სისტემა ამუშავებს მონაცემთა დიდ მოცულობას.
უპირატესობები	დიდი მოცულობის მონაცემების ერთობლივი გამოყენება ეფექტური ხდება, რამდენადაც არ არის საჭირო მონაცემები გადაეცეს ერთი ქვესისტემიდან მეორეს. ქვესისტემამ არ არის აუცილებელი იცოდეს, როგორ იყენებს სხვა ქვესისტემა მონაცემებს. მცირდება დაკავშირების ხარისხი.

	სისტემებში რეპოზიტორებით სარეზერვო კოპირება, უსაფრთხოების უზრუნველყოფა, მონაცემებთან წვდომის მართვა და აღდგენა ცენტრა-ლიზებულია, რამდენადაც ისინი შედიან რეპოზიტორიების მართვის სისტემაში.
ნაკლოვანებები	ყველა ქვესისტემა უნდა იყოს შეთანხმებული რეპოზიტორების (მონაცემთა მოდელების) სტრუქტურასთან. სხვადასხვა ქვესისტემას წარედგინება განსხვავებული მოთხოვნები მონაცემების უსაფრთხოებასთან, აღდგენასთან და რეზერვირებასთან დაკავშირებით, ხოლო რეპოზიტორების ნიმუშთან ყველა ქვესისტემასთან გამოიყენება ერთიდაიგივე პოლიტიკა.

კლიენტ-სერვერი

აღწერა	მონაცემები და სისტემაში მიმდინარე პროცესები განაწილებულია რამდენიმე პროცესორს შორის. ნიმუშს გააჩნია სამი ძირითადი კომპონენტი: ავტონომიური სერვისების ნაკრები (სხვა ქვესისტემებს წარუდგენს სერვი-სებს), კლიენტ-ქვესისტემების ნაკრები (სერვერების მიერ წარმოდგენილ სერვისების გამოსაძახებლად) და ქსელი (კლიენტების სერვისთან წვდომის უზრუნველსაყოფად). კლიენტებმა უნდა იცოდნენ სერვერების და სერვისების სახელები, ამასთან ერთად, სერვერებმა არ უნდა იცოდნენ კლიენტების სახელები და მათი რაოდენობა. კლიენტებს გააჩნიათ სერვერების მიერ წარმოდგენილ სერვისებთან წვდომა პროცედურების დისტანციური გამოძახების საშუალებით.
რეკომენდაციები	მოცემული მიდგომის გამოყენება შესაძლებელაი რეპოზიტორებზე დაფუძნებული სისტემების რეალიზაციის დროს. როცა რეპოზიტორი წარმოადგენს სისტემის სერვერს, ხოლო ქვესისტემები, რომლებსაც გააჩნიათ წვდომა რეპოზიტორებთან, წარმოადგენს კლიენტებს.
უპირატესობები	მოცემული ნიმუში ახდენს განაწილებული არქიტექტურის ფორმირებას, მისი ეფექტური გამოყენება შესაძლებელია ქსელურ სისტემებში, სადაც ადვილად იქნება შესაძლებელი ახალი სერვერის დამატება და მისი

	სისტემასთან ინტეგრირება ან სერვერის განახლება, ისე, რომ ზეგავლენა არ მოახდინოს სისტემის სხვა ნაწილებზე.
ნაკლოვანებები	მუშაობის პროცესში სერვერები და კლიენტები ერთმანეთს უცვლიან მონაცემებს, მაგრამ დიდი მოცულობის ინფორმაციის შემთხვევაში სერვერებსა და კლიენტებს შორის ქსელის გამტარუნარიანობასთან დაკავშირებით შეიძლება წარმოიქმნას პრობლემები.

8.4. ამოცანის პროექტირება ქცევითი დიზაინის (BehavioralDesign) ნიმუშების მეშვეობით

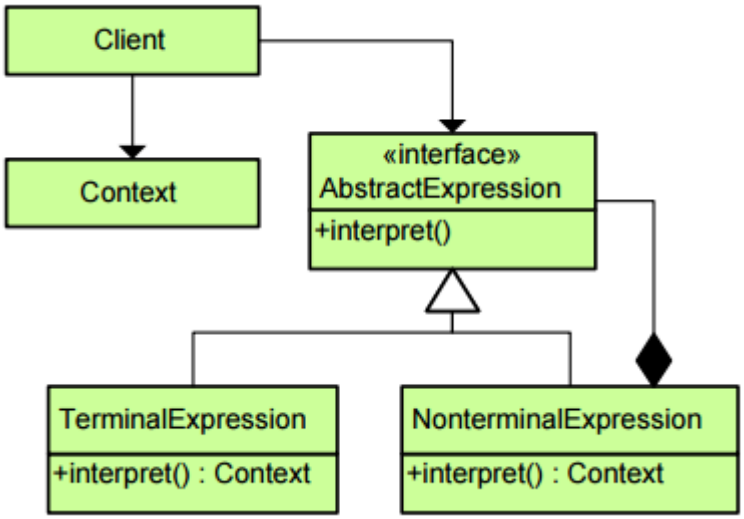
ქცევის ნიმუშები დაკავშირებულია ობიექტებს შორის ვალდებულებების განაწილებასთან. ქცევის ნიმუშები ახასიათებს მართვის ისეთ რთულ ნაკადებს, რომელთა დაკვირვებაც პროგრამის მუშაობის პერიოდში ძალიან რთულია. ყურადღება გამახვილებულია არა მართვის ნაკადზე, არამედ ობეიქტებს შორის კავშირებზე.

ქცევის ნიმუშებში კლასის დონედ გამოიყენება მემკვიდრეობითობა - იმისათვის, რომ განაწილდეს ქცევა სხვადასხვა კლასებს შორის.

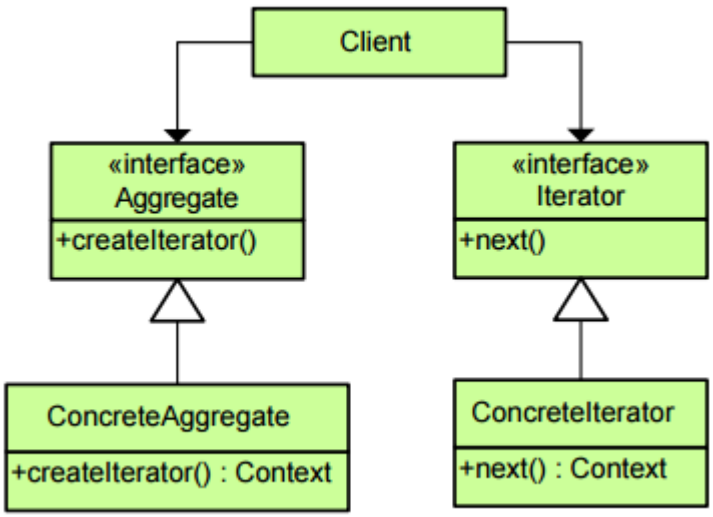
ქცევის ნიმუშებში ობიექტების დონედ გამოიყენება კომპოზიცია. ზოგიერთი მათგანი აღწერს, კოოპერაციის გამოყენებით როგორ წყვეტს ამოცანას თანაბარი ობიექტების სიმრავლე. აქ მთავარია, როგორ ღებულობენ ინფორმაციას ობიექტები ერთმანეთის არსებობის შესახებ.

8.4.1. ინტერპრეტატორი (Interpreter)

პრობლემა	გვაქვს ყველაზე გავრცელებული, ცვლილებებს დაქვემდებარებული ამოცანა
გადაწყვეტა	ინტერპრეტატორის შექმნა, რომელიც გადაწყვეტს დასმულ ამოცანას.

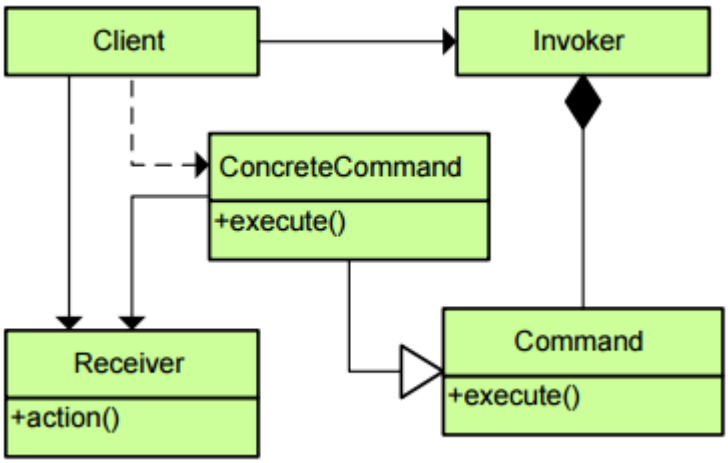
	
მაგალითი	ნიმუშის მიხედვით სტრიქონის მოძებნის ამოცანის გადაწყვეტა შესაძლებელია უშუალოდ ინტერპრეტატორის შექმნით, რომელიც განსაზღვრავს ენის გრამატიკას. „კლიენტი“ ქმნის წინადადებას აბსტრაქტული სინტაქსური ხის სახით, რომლის კვანძებში მდებარეობს კლასების ობიექტები „არატერმინალური გამოსახულება“ და „ტერმინალური გამოსახულება“(რეკურსიული), ამის შემდეგ „კლიენტი“ ახდენს კონტექსტის ინიციალიზაციას და იძახებს ოპერაციას „დაშლა“ (კონტექსტი). „არატერმინალური გამოსახულების“ ტიპის ყოველ კვანძში განისაზღვრება „დაშლა“ ოპერაცია ყოველი ქვეგამოსახულებისთვის. „ტერმინალური გამოსახულების“ ოპერაცია „დაშლა“ განსაზღვრავს რეკურსიის ბაზას. „აბსტრაქტული გამოსახულება“ განსაზღვრავს „დაშლა“ აბსტრაქტულ ოპერაციას, რომელიც საერთოა ყველა კვანძისათვის აბსტრაქტულ სინტაქსურ ხეზე. „კონტექსტი“ შეიცავს ინტერპრეტატორთან მიმართებაში გლობალურ ინფორმაციას.
უპირატესობა	ადვილია გრამატიკის გაფართოება და შეცვლა, კლასის რეალიზაცია, აბსტრაქტული სინტაქსური ხის კვანძების აღწერა მსგავსია. მარტივად იცვლება გამოსახულების გამოთვლის საშუალება.
ნაკლოვანებები	გრამატიკის მხარდაჭრა წესების დიდი რაოდენობის გამო ძალიან გართულებულია.

8.4.2. იტერატორი (Iterator) ან კურსორი (Cursor)

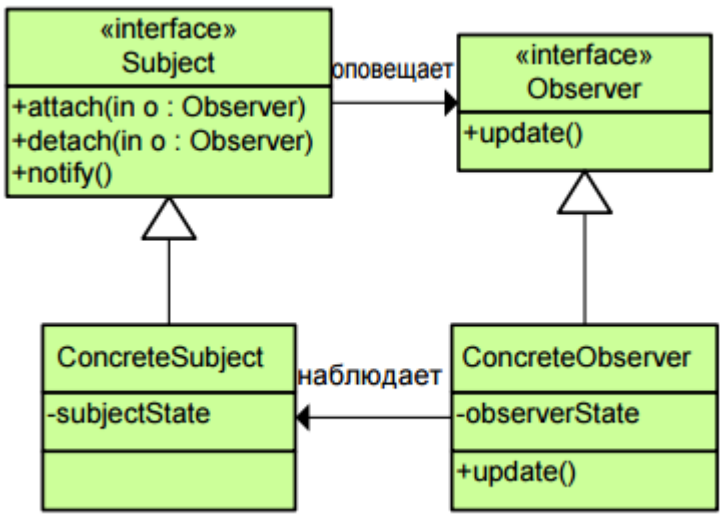
პრობლემა	შედგენილი ობიექტი, მაგ, სიამ, უნდა უზრუნველყოს საკუთარ ელემენტებზე წვდომა ისე, რომ არ გახსნას მათი შიდა სტრუქტურა, ამასთან ერთად, სიის გადალაგება ხდება სხვადასხვა გზით ამოცანის შესაბამისად.
გადაწყვეტა	იქმნება კლასი „იტერატორი“, რომელიც განსაზღვრავს ინტერფეისს ელემენტებთან წვდომისა და გადარჩევისათვის, „კონკრეტული იტერატორი“ ახდენს „იტერატორი“ კლასის ინტერფეისის რეალიზაციას და ახორციელებს მონიტორინგს „აგრეგატის“ მიმდინარე პოზიციაზე. „კონკრეტული აგრეგატი“ ახორციელებს იტერატორის შექმნის ინტერფეისის რეალიზაციას და აბრუნებს „კონკრეტული იტერტორი“-ს კლასის ეგზემპლიარს. „კონკრეტული იტერატორი“ ასრულებს მიმდინარე ობიექტის მონიტორინგს და შეუძლია გადარჩევის მომდევნო ობიექტის გამოთვლა. 
უპირატესობა	გააჩნია აგრეგატის გადარჩევის სხვადასხვა საშუალებების მხარდაჭერა, შესაძლებელია ერთდროულად რამდენიმე გადარჩევის აქტიურობა.

8.4.3. ბრძანება ((Command), მოქმედება (Action) ან ტრანზაქცია

პრობლემა	აუცილებელია ობიექტს გაეგზავნოს მოთხოვნა, ისე, რომ არ იქნება გარკვეული, რომელი ოპერაციის შესრულება არის მოთხოვილი და ვინ არის მიმღები.

გადაწყვეტა	მოთხოვნის, როგორც ობიექტის ინკაფსულირება. „კლიენტი“ ქმნის ობეიქტს „კონკრეტული ბრძანება“, რომელიც იძახებს მიმღების ოპერაცის მოთხოვნის შესასრულებლად. „ინიციატორი“ აგზავნის მოთხოვნას, შეასრულოს ბრძანება „შესრულდეს()“. „ბრძანება“ აცხადებს ოპერაციის შესასრულებლად ინტერფეისს, „კონკრეტულ იბრძანება“ განსაზღვრავს კავშირს „მიმღებ“ და „მოქმედება()“ ოპერაციის ობიექტებს შორის, ამასთან ერთად ახდენს „შესრულდეს ()“ ოპერაციის რეალიზაციას „მიმღების“ ობიექტის შესაბამისი ოპერაციის გამოძახებით. „კლიენტი“ ქმნის „კონკრეტული ბრძანების“ კლასის ეგზემპლიარს და ადგენს მის მიმღებს, „ინიციატორი“ მიმართავს ბრძანებას მოთხოვნის შესრულებისათვის, „მიმღებს“ (ნებისმიერი კლასი) გააჩნია მოთხოვნის შესასრულებლად საჭირო ოპერაციების შესრულების საშუალებებზე ინფორმაცია. 
უპირატესობა	ნიმუში „ბრძანება“ არღვევს კავშირს ოპერაციის მაინიცირებელ ობიექტსა და იმ და ობიექტებს შორის, რომელსაც გააჩნია ინფორმაცია, როგორ უნდა შესრულდეს, ამასთან ერთად იქმნება ობიექტი „ბრძანება“, რომლის გაფართოება და მანიპულირება შესაძებელია.

8.4.4. დამკვირვებელი (Observer), გამოსცეს- გამოიწეროს (Publish - Subscribe) ან ღონისძიების მოდლების დელეგირება Delegation Event Model

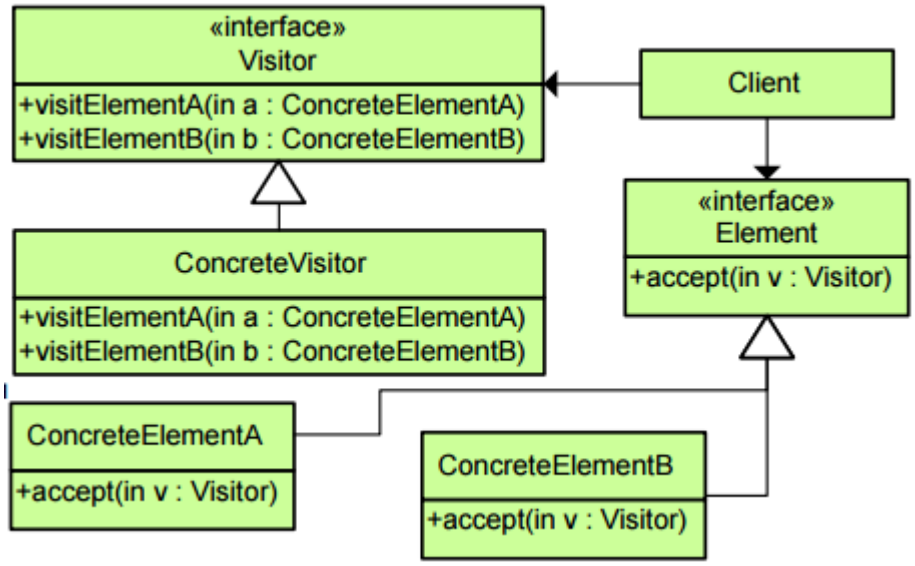
პრობლემა	ერთმა ობიექტმა („გამომწერი“) უნდა იცოდეს სხვა ობიექტის ზოგიერთი ღონისძიების ან მდგომარეობის ცვლილებების შესახებ. ამასთან ერთად აუცილებელია უზრუნველყოფილი იყოს ობიექტთან „გამომწერი“ დაკავშირების დაბალი დონე.
გადაწყვეტა	უნდა განისაზღვროს „გამოწერის“ ინტერფეისი. ობიექტები - გამომწერები ახდენენ ამ ინტერფეისის რეალიზაციას და დინამიურად რეგისტრირდებიან გარკვეული ღონისძიებების შესახებ ინფორმაციის მისაღებად. ამის შესახებ, პირობითი ღონისძიების რეალიზაციისას, გაფრთხილებულია ყველა ობიექტი - გამომწერები. 

8.4.5. არ ესაუბროთ უცხოს (Don't talk to strangers)

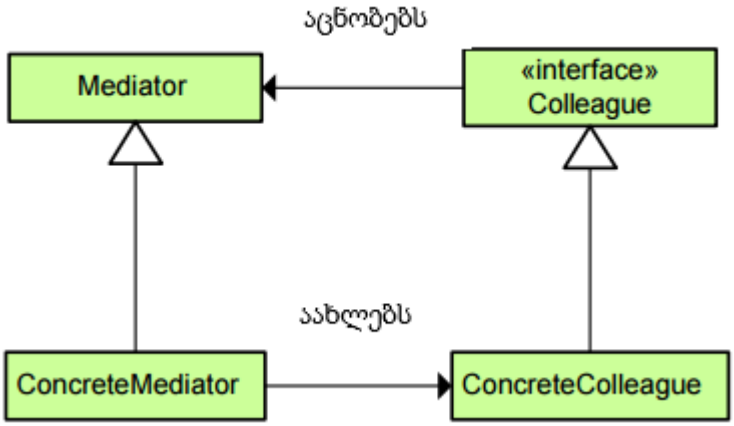
პრობლემა	კლიენტის ობიექტის კავშირის უზრუნველყოფა ირიბი ობიექტებისადმი (ანუ ისეთი ობიექტებისადმი, რომლებიც ცნობილია სხვა ობიექტებისთვის, ხოლო უცნობია თვითონ კლიენტისთვის.
გადაწყვეტა	აუცილებელია თავიდან ავირიდოთ საპროექტო გადაწყვეტები, რომელიც ითვალისწინებს შეტყობინების გადაცემას დისტანციურ ირიბ (უცნობ) ობიექტებთან. ამ პრობლემის გადაწყვეტა შეიძლება იყოს ნიმუშის „მდგრადი ცვლილებებისადმი“ კერძო შემთხვევის გამოყენება (3.1.9).
უპირატესობა	უზრუნველყოფს სისტემის მდგრადობას ობეიქტის სტრუქტურის ცვლილების შემთხვევაში.

8.4.6. ვიზიტორი (Visitor)

პრობლემა	ზოგიერთი სტრუქტურის ყოველ ობიექტზე სრულდება ოპერაცია. განი-საზღვროს ახალი ოპერაცია, ისე, რომ არ შეიცვალოს ობიექტების კლასები.
გადაწყვეტა	კლიენტმა, რომელიც იყენებს მოცემულ ნიმუშს, უნდა შექმნას კლასის ობეიქტი „კონკრეტული ვიზიტორი“, ხოლო შემდეგ ეწვიოს სტრუქტურის ყოველ ელემენტს. „ვიზიტორი“ აცხადებს ოპერაციას „ეწვიეთ“ ყოველი „კონკრეტული ელემენტის“ ყოველი კლასისთვის. „კონკრეტული ელემენტი“ ახდენს ყველა ოპერაციის რეალიზებას, რომელიც გამოცხადე-ბულია კლასში „ვიზიტორი“. ყოველი ოპერაცია ახდენს ალგორითმის ფრაგმენტის რეალიზაციას სტრუქტურაში განსაზღვრული კლასისთვის. კლასი „კონკრეტული ვიზიტორი“ წარმოადგენს კონტექსტს ყოველი ალგორითმისათვის და ინახავს მის ლოკალურ მდგომარეობას. „ელემენტი“ განსაზღვრავს ოპერაციას „მიღება“, რომელიც ღებულობს „ვიზიტორს“ როგორც არგუმენტს. „ობიექტის სტრუქტურა“-ს შეუძლია გადაუგზავნოს თავისი არგუმენტები და წარუდგინოს ვიზიტორს მაღალი დონის ინტერ-ფეისი თავისი ელემენტების მისაღებად.

	
რეკომენდაციები	ლოგიკურია მისი გამოყენება, თუ სტრუქტურაში არსებობს ობიექტები მრავალი კლასის სხვადასხვა ინტერფეისებით და აუცილებელია მათზე ისეთი ოპერაციების შესრულება, რომელიც დამოკიდებულია კონკრეტულ კლასზე ან თუ ობიექტის სტრუქტურის დამყენებელი იშვიათად იცვლება, მაგრამ ახალი ოპერაციების დამატება ამ სტრუქტურაზე ხშირად ხდება.
უპირატესობა	მარტივდება ახალი ოპერაციების დამატება, აერთიანებს „ვიზიტორ“ კლას-ში ნათესაურ ოპერაციებს.
ნაკლოვანებები	რთულია ახალი კლასის დამატება, რამდენადაც საჭირო ხდება „ვიზიტორ“ კლასში ახალი აბსტრაქტული ოპერაციის გამოცხადება.

8.4.7. მედიატორი (Mediator)

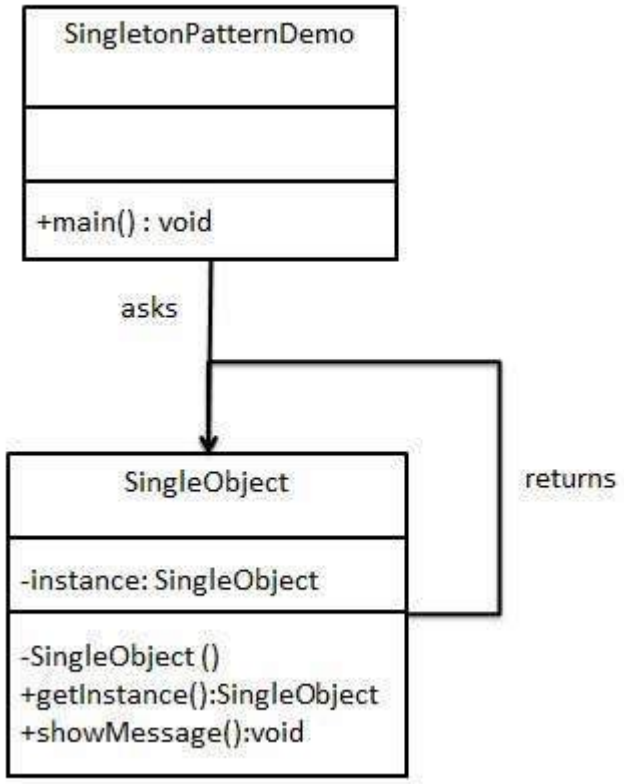
პრობლემა	ობიექტების სიმრავლის ურთიერთქმედების უზრუნველყოფა, ისე, რომ ამ შემთხვევაში მოხდეს სუსტი კავშირის ფორმირება და ობიექტები განთავისუფლებული იქნენ ერთმანეთთან მიმართვის აუცილებლობისაგან.
გადაწყვეტა	შეიქმნას ობიექტი, რომელიც ობიექტების სიმრავლის ურთიერთქმედების საშუალების ინკაფსულირებას მოახდენს 
მაგალითი	„მედიატორი“ განსაზღვრავს ინტერფეისს „კოლეგების“ ობიექტებთან ინფორმაციის გაცვლისათვის. „კონკრეტული მედიატორი“ არეგულირებს „კოლეგა“ ობიექტების მოქმედების კოორდინაციას. „კოლეგის“ ყოველმა კლასმა იცის ობიექტის „მედიატორის“ შესახებ, ყველა „კოლეგა“ ცვლის ინფორმაციას მხოლოდ მედიატორთან. მისი არარსებობის შემთხვევაში საჭირო გახდებოდა ინფორმაციის პირდაპირი გაცვლა. „კოლეგები“ აგზავნიან მოთხოვნას მედიატორთან და ღებულობენ მათგან მოთხოვნებს. „მედიატორი“ ახდენს კორპორატიული ქცევის რეალიზებას, ყოველი მოთხოვნის ერთ ან რამდენიმე „კოლეგასთან“ გადაგზავნით.
უპირატესობა	უზრუნველყოფს კავშირს „კოლეგებს“ შორის. მართვა ხდება ცენტრალიზებული.

8.5. პროექტის პროგრამული დიზაინის შექმნა

იხ. დანართი

შემოქმედებითი დიაზინის ნიმუში - Singleton Pattern

შევქმნათ კლასი SingleObject. კლასის წევრები მოცემულია დიაგრამაზე (ნახ.412).



ნახ. 412

კლასის შექმნის პროგრამულ კოდს აქვს შემდეგი სახე:

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
```

```
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

მივიღოთ მხოლოდ ერთი ობიექტი singleton კლასიდან

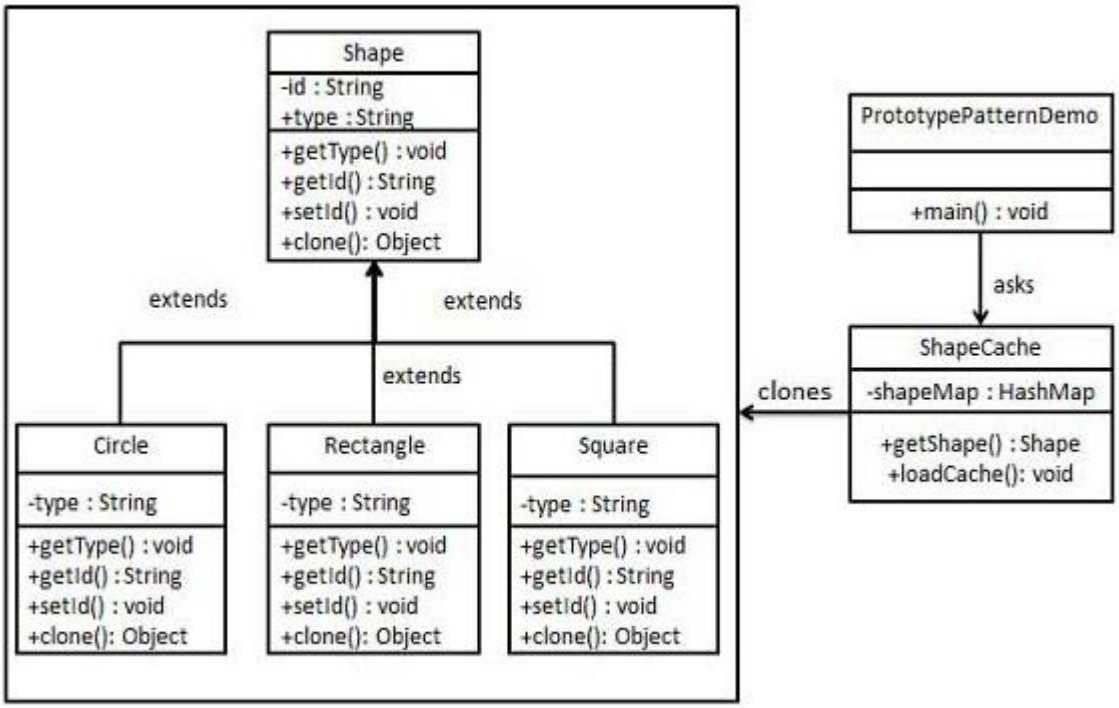
SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

შესრულების შედეგი :

```
Hello World!
```

შევქმნათ აბსტრაქტული კლასი *Shape* და კონკრეტული კლასი, რომელიც წარმოადგენს shape კლასის გაფართოებას (ნახ. 413) .



ნახ. 413

შექვმნათ აბსტრაქტული კლასი, Cloneable ინტერფესის რეალიზაციისათვის.

Shape.java

```
public abstract class Shape implements Cloneable {

   private String id;
   protected String type;

   abstract void draw();

   public String getType(){
      return type;
   }

   public String getId() {
      return id;
   }

   public void setId(String id) {
```

```
      this.id = id;
   }

   public Object clone() {
      Object clone = null;

      try {
         clone = super.clone();

      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
      }

      return clone;
   }
}
```

შევქმნათ კონკრეტული კლასები:

Rectangle.java

```
public class Rectangle extends Shape {

   public Rectangle(){
     type = "Rectangle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square extends Shape {

   public Square(){
     type = "Square";
   }

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle extends Shape {

   public Circle(){
     type = "Circle";
   }

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

შევქმნათ კლასი, იმისათვის, რომ მივიღოთ მონაცემთა ბაზიდან კონკრეტული კლასები და შევინახოთ *Hashtable*-ში.

ShapeCache.java

```
import java.util.Hashtable;


public class ShapeCache {


   private static Hashtable<String, Shape> shapeMap  = new Hashtable<String,
Shape>();


   public static Shape getShape(String shapeId) {
      Shape cachedShape = shapeMap.get(shapeId);
      return (Shape) cachedShape.clone();
   }


   // for each shape run database query and create shape
   // shapeMap.put(shapeKey, shape);
   // for example, we are adding three shapes


   public static void loadCache() {
      Circle circle = new Circle();
      circle.setId("1");
      shapeMap.put(circle.getId(),circle);
```

```
      Square square = new Square();
      square.setId("2");
      shapeMap.put(square.getId(),square);

      Rectangle rectangle = new Rectangle();
      rectangle.setId("3");
      shapeMap.put(rectangle.getId(), rectangle);
   }
}
```

PrototypePatternDemo იყენებს *ShapeCache* კლასს, რომ მიიღოს *Hashtable*-ში შენახული ფორმების (ფიგურების) კლონები.

PrototypePatternDemo.java

```
public class PrototypePatternDemo {
   public static void main(String[] args) {
      ShapeCache.loadCache();

      Shape clonedShape = (Shape) ShapeCache.getShape("1");
      System.out.println("Shape : " + clonedShape.getType());

      Shape clonedShape2 = (Shape) ShapeCache.getShape("2");
      System.out.println("Shape : " + clonedShape2.getType());

      Shape clonedShape3 = (Shape) ShapeCache.getShape("3");
      System.out.println("Shape : " + clonedShape3.getType());
   }
}
```

პროგრამის შედეგი:

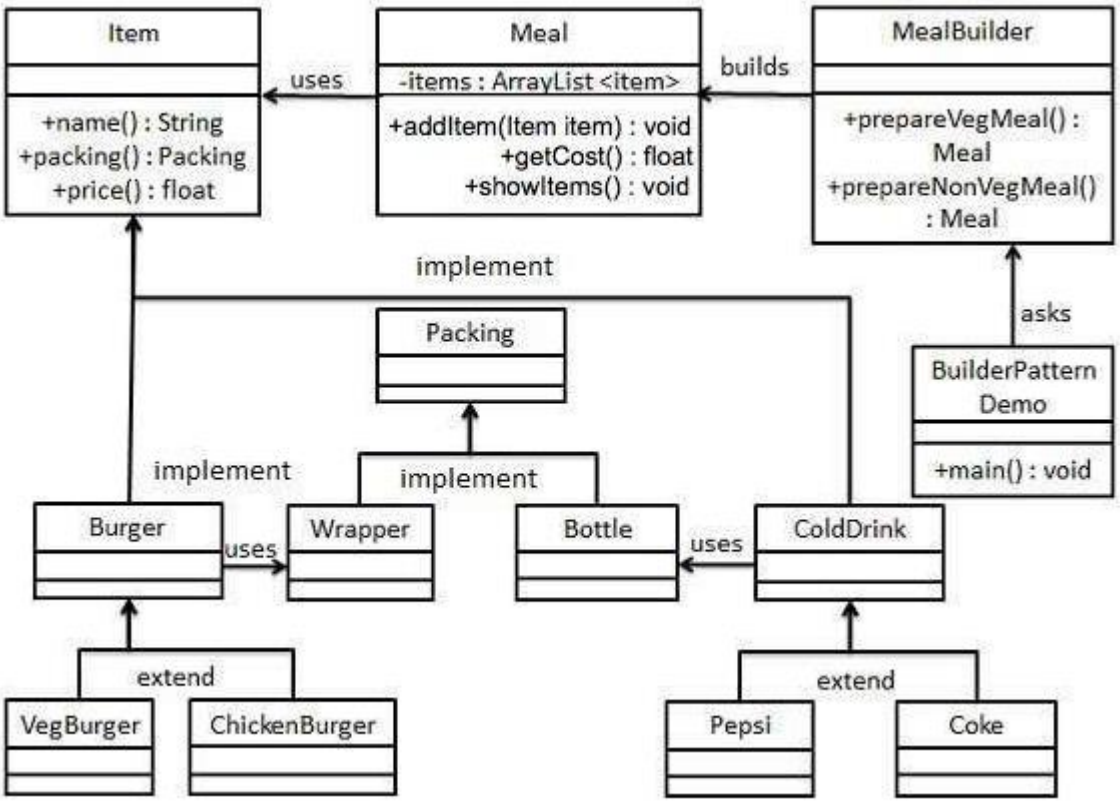
```
Shape : Circle
Shape : Square
Shape : Rectangle
```

შემოქმედებითი დიაზინის ნიმუში Builder pattern

განვიხილოთ ბიზნეს-ამოცანა - სწრაფი კვების რესტორანი, ტიპიური მენიუთი: ჰამბურგერი და ცივი სასმელი. ბურგერი შესაძლებელია იყოს Veg Burger ან Chicken Burger, რომლებიც იქნება შეფუთული. ცივი სასმელი შეიძლება იყოს კოკა-კოლა ან პეპსი ბოთლში.

ჩვენ უნდა შევქმნათ საგნების ინტერფეისი, რომლებიც წარმოადგენენ კვების პროდუქტებს და ცივ სასმელებს და ასევე შესაფუთ მასალებს.

ნახ. 409-ზე მოცემულია სტრუქტურა, რომლის მიხედვითაც უნდა მოხდეს კლასების და ობიექტების შექმნა



ნახ. 413

შევქმნათ ინტერფეისი Item, რომელიც მოიცავს სურსათის და შეფუთვის ელემენტებს

Item.java

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

Packing.java

```
public interface Packing {
   public String pack();
}
```

შევქმნათ კონკრეტული კლასები, რომლებიც უზრუნველყოფს შეფუთვის ინტერფეისს.

Wrapper.java

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

შევქმნათ აბსტრაქტული კლასები, რომლებიც უზრუნველყოფს გულისხმობის პრინციპით ფუნქციონირების ინტერფეისს.

Burger.java

```
public abstract class Burger implements Item {

   @Override
   public Packing packing() {
      return new Wrapper();
   }

   @Override
   public abstract float price();
}
```

ColdDrink.java

```
public abstract class ColdDrink implements Item {

        @Override
        public Packing packing() {
       return new Bottle();
        }

        @Override
        public abstract float price();
}
```

შევქმნათ Burger და ColdDrink კლასების მემკვიდრე კონკრეტული კლასები

VegBurger.java

```
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
```

```
    public String name() {
        return "Veg Burger";
    }
}
```

ChickenBurger.java

```
public class ChickenBurger extends Burger {

    @Override
    public float price() {
        return 50.5f;
    }

    @Override
    public String name() {
        return "Chicken Burger";
    }
}
```

Coke.java

```
public class Coke extends ColdDrink {

    @Override
    public float price() {
        return 30.0f;
    }

    @Override
    public String name() {
        return "Coke";
    }
}
```

Pepsi.java

```
public class Pepsi extends ColdDrink {
```

```
   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

შევქმნათ კლასი Meal, რომელიც მოიცავს ზემოთ აღწერილ ობიექტს.

Meal.java

```
import java.util.ArrayList;
import java.util.List;

public class Meal {
   private List<Item> items = new ArrayList<Item>();

   public void addItem(Item item){
      items.add(item);
   }

   public float getCost(){
      float cost = 0.0f;

      for (Item item : items) {
         cost += item.price();
      }
      return cost;
   }

   public void showItems(){

      for (Item item : items) {
```

```
            System.out.print("Item : " + item.name());
            System.out.print(", Packing : " + item.packing().pack());
            System.out.println(", Price : " + item.price());
         }
      }
   }
```

შევქმნათ კლასი MealBuilder, კლასი - მშენებელი, რომელიც პასუხს აგებს Meal კლასის ობიექტების შექმნაზე.

MealBuilder.java

```
public class MealBuilder {

   public Meal prepareVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new VegBurger());
      meal.addItem(new Coke());
      return meal;
   }

   public Meal prepareNonVegMeal (){
      Meal meal = new Meal();
      meal.addItem(new ChickenBurger());
      meal.addItem(new Pepsi());
      return meal;
   }
}
```

BuiderPatternDemo იყენებს MealBuider-ს builder pattern-ის სადემონსტრაციოდ.

BuilderPatternDemo.java

```
public class BuilderPatternDemo {
   public static void main(String[] args) {

      MealBuilder mealBuilder = new MealBuilder();
```

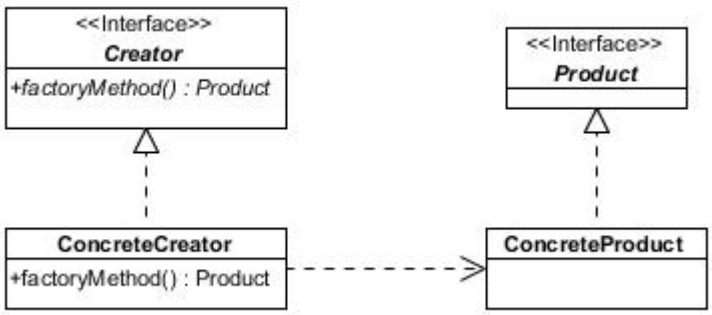
```
        Meal vegMeal = mealBuilder.prepareVegMeal();
        System.out.println("Veg Meal");
        vegMeal.showItems();
        System.out.println("Total Cost: " + vegMeal.getCost());

        Meal nonVegMeal = mealBuilder.prepareNonVegMeal();
        System.out.println("\n\nNon-Veg Meal");
        nonVegMeal.showItems();
        System.out.println("Total Cost: " + nonVegMeal.getCost());
    }
}
```

შესრულების შედეგი:

```
Veg Meal
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0


Non-Veg Meal
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

შევქმნათ ინტერფეისი, რომელიც უზრუნველყოფს Tank ტიპის ობიექტის შექმნას

```
1  public interface TankCreator {
2
3      Tank getTank(String name);
4  }
```



ნახ. 414

დიაგრამაზე, `TankCreator` შესაბამება ინტერეფისს `Creator`, ხოლო მეთოდი `getTank()` - `factoryMethod()`-ს. შევქმნათ კლასი `DidgoriTankCreator`, რომლის რეალიზაციას ახდენს ინტერფეისი `TankCreator`. წინასწარ განვსაზღვროთ მეთოდი `getTank()`, ისე, რომ მან დააბრუნოს `Tank` ტიპის ობიექტის კონკრეტული ეგზემპლიარი, გადაცემული არგუმენტისგან დამოუკიდებლად.

```
public class DidgoriTankCreator implements TankCreator {

    @Override
    public Tank getTank(String name) {
        if ("T-34".equals(name)) {
            return new T34_Tank();
        } else if ("T-35".equals(name)) {
            return new T35_Tank();
        } else if ("T-38".equals(name)) {
            return new T38_Tank();
        }

        return null;
    }
}
```

კლასი DidgoriTankCreator შეესაბამება კლასს `ConcreteCreator`.

შევქმნათ ინტერფეისი `Tank`. ის შეიცავს მეთოდს `getDescription()`, რომელიც აბრუნებს ტანკის აღწერას.

```
1  public interface Tank {
2
3      String getDescription();
4  }
```

Tank ინტერფეისი შეესაბამება Product ინტერფეისს.

შევქმნათ კლასი, რომელიც აღწერს "T-34" ტანკს. წინასწარ განვსაზღვროთ მეთოდი getDescription() ისე, რომ მან დააბრუნოს ინფორმაცია ამ ტანკზე:

```
public class T34_Tank implements Tank {
    private static final String NAME = "T-34";
    private static final String COUNTRY = "Georgia";

    public String getDescription() {
        return NAME + " " + COUNTRY;
    }
}
```

ანალოგიურად, შევქმნათ კლასი, რომელიც აღწერს ტანკებს "T-35" და "T-38":

```
public class T35_Tank implements Tank {
    private static final String NAME = "T-35";
    private static final String COUNTRY = "Georgia";

    public String getDescription() {
        return NAME + " " + COUNTRY;
    }
}
```

```
public class T38_Tank implements Tank {
    private static final String NAME = "T-38";
    private static final String COUNTRY = "Georgia";

    public String getDescription() {
        return NAME + " " + COUNTRY;
    }
```

კლასები `T34_Tank`, `T35_Tank` და `T38_Tank` არის კონკრეტული პორდუქტები, რომლებიც შეესაბამებიან კლასს `ConcreteProduct`.

სადემონსტრაციო მაგალითისათვის შევქმნათ კლასი `FactoryMethodDemo`. შევქმნათ `TankCreator` ტიპის ობიექტი ფაბრიკული მეთოდის `getTank()` მეშვეობით. მივიღოთ ობიექტის ეგზემპლიარები სხვადასხვა ტანკების შესახებ და გამოვიტანოთ ინფორმაცია მათზე:

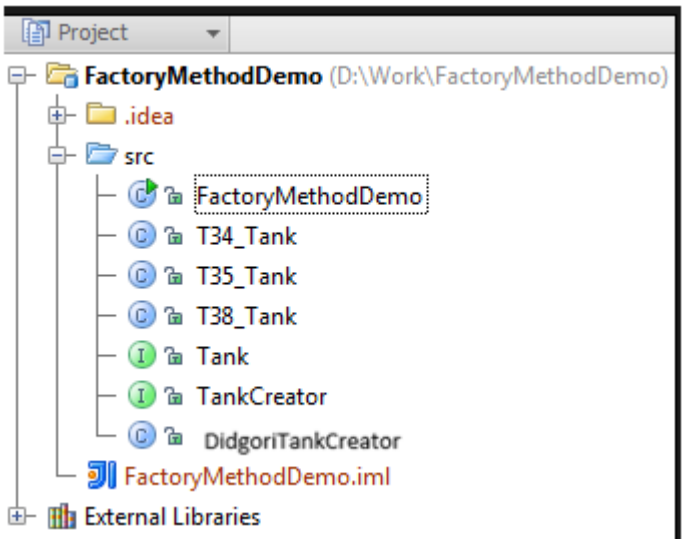
```
public class FactoryMethodDemo
{

    public static void main(String[] args) {
        TankCreator tankCreator = new DidgoriTankCreator();

        Tank tank = tankCreator.getTank("T-34");
        System.out.println(tank.getDescription());

        tank = tankCreator.getTank("T-35");
        System.out.println(tank.getDescription());

        tank = tankCreator.getTank("T-38");
        System.out.println(tank.getDescription());
    }
}
```

გავუშვათ შესრულებაზე პროგრამა `main()`:

```
C:\jdk1.6.0_30\bin\java
T-34 USSR
T-35 USSR
T-38 USSR
```

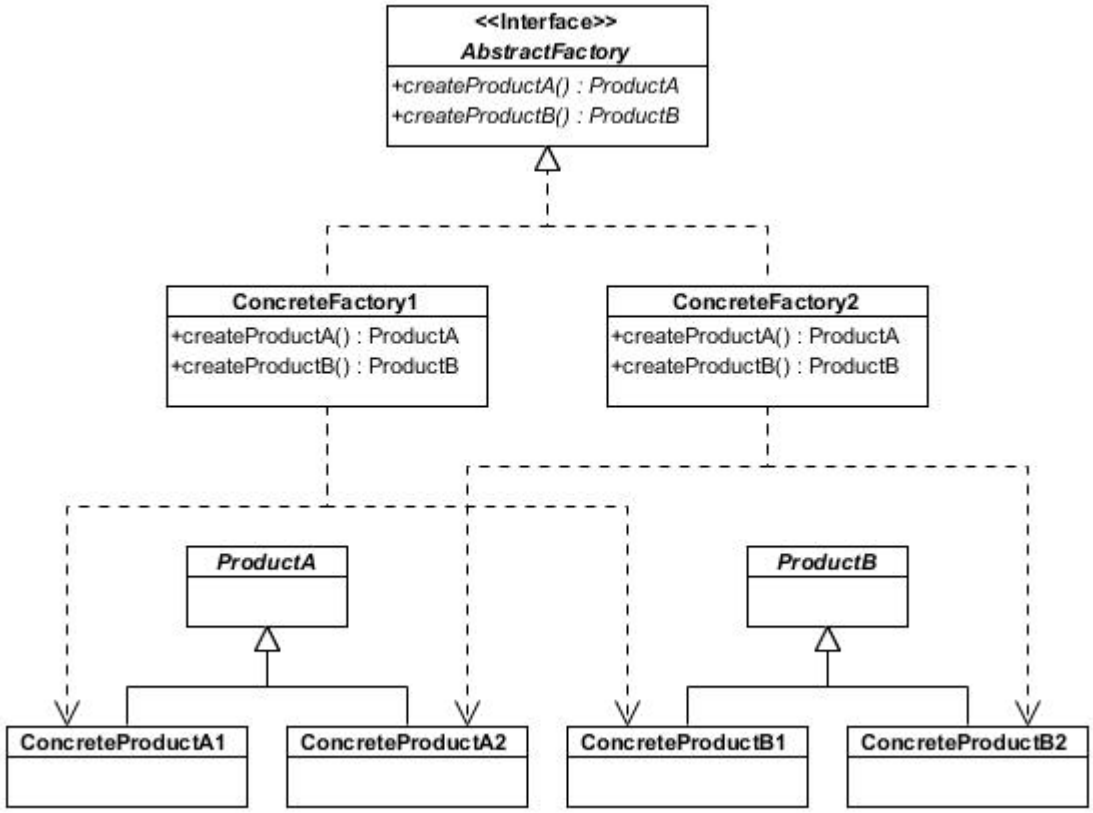
პროგრამის მუშაობის პროცესში ჩვენ შევქმენით `Tank` ტიპის სხვადასხვა ობიექტები კონსტრუქტორების არაცხადი გამოძახებით `main()`-ის მეთოდებით.

პროექტის სტრუქტურა:



ნახ. 415

შემოქმედებითი დიზაინის ნიმუში Abstract Factory



ნახ. 416

წარმოვიდგინოთ, რომ ჩვენ ვქმნით თამაშს საბრძოლო მოქმედებების წარმართვისათვის. ჩვენ გვეყოლება ორი ერთმანეთის მოწინააღმდეგე სახელმწიფო - საქართველო და რუსეთი. თითოეულ მხარეს ჰყავს მხოლოდ ორი ტიპის ტექნიკა - ტანკი და თვითმფრინავი. შევქმნათ აუცილებელი კლასები `Abstract Factory`-ის ნიმუშის გამოყენებით და სამხედრო ტექნიკის ფაბრიკის ინტერფეისი. ის შეიცავს ტანკისა და თვითმფრინავის შექმნის მეთოდებს:

```
public interface EngineryFactory {

    public Tank createTank();

    public Aircraft createAircraft();
}
```

შექმნილი ინტერფეისი შეესაბამება AbstractFactory- ს ინტერფეისს დიაგრამაზე. შევქმნათ აბსტრაქტული კლასი `Tank`. მასში აღწერილია ტანკის თვისებები, ამ თვისებების მიღება/ცვლილების შესაბამისი მეთოდები და მეთოდი `getDescription()`, რომლის საშუალებით შესაძლებელია ტანკის აღწერა:

```
public abstract class Tank {
    private int speed;
    private int power;

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    public int getPower() {
        return power;
    }

    public void setPower(int power) {
        this.power = power;
    }

    public abstract String getDescription();
}
```

`Tank` კლასი ეს არის `ProductA` კლასის ანალოგი. ანალოგიურად, შვქმნით `Aircraft` კლასს, რომელიც აღწერს აბსტრაქტულ თვითმფრინავს:

```
public abstract class Aircraft {
    int speed;
    int power;
    int altitude;

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    public int getPower() {
        return power;
    }

    public void setPower(int power) {
        this.power = power;
    }

    public int getAltitude() {
        return altitude;
    }

    public void setAltitude(int altitude) {
        this.altitude = altitude;
    }

    public abstract String getDescription();
}
```

`Aircraft` კლასი შეესაბამება კლასს `ProductB`. გადავიდეთ კონკრეტული ტანკების და თვითმფრინავების მწარმოებელი კონკრეტული ფაბრიკის შექმნაზე. კლასი `GEOEngineryFactory` ახდენს `EngineryFactory` ინტერფეისის რეალიზაციას და საშუალებას იძლევა შექმნას მხოლოდ შემდეგი სამხედრო ტექნიკა: ტანკი "T-34" და თვითმფრინავი „Geo-2”

```
public class GEOEngineryFactory implements EngineryFactory {

    @Override
    public Tank createTank() {
        return new T34_Tank();
    }

    @Override
    public Aircraft createAircraft() {
        return new Il2_Aircraft();
    }
}
```

კლასი GEOEngineryFactory შეესაბამება კლასს ConcreteFactory1. შევქმნათ კლასი, როელიც აღწერს ტანკს "T-34", განვსაზღვროთ მეთოდი getDescription() ისე, რომ მან დააბრუნოს ინფორმაცია ტექნიკის შესახებ:

```
public class T34_Tank extends Tank {
    private static final String NAME = "T-34";
    private static final String COUNTRY = "Georgia";

    public String getDescription() {
        return NAME + " " + COUNTRY;
    }
}
```

კლასი `T34_Tank` ეს არის `GEOEngineryFactory` ფაბრიკის კონკრეტული პროდუქტი. ის შეესაბამება `ConcreteProductA1` კლასს.

შევქმნათ კლასი, რომელიც აღწერს თვითმფრინავს „Geo-2”.

```
public class Geo2_Aircraft extends Aircraft {
    private static final String NAME = "Geo-2";
    private static final String COUNTRY = Georgia";

    public String getDescription() {
        return NAME + " " + COUNTRY;
    }
}
```

Geo2_Aircraft კლასი შეესაბამება `ConcreteProductB1` კლასს.

ანალოგიურად შევქმნათ ფაბრიკა რუსეთის სამხედრო ტექნიკის - ტანკი „"E-25" და თვითმფრინავი “IL-2”-ის წარმოებისათვის.

```
public class RuEngineryFactory implements EngineryFactory {

    @Override
    public Tank createTank() {
        return new E25_Tank();
    }

    @Override
    public Aircraft createAircraft() {
        return new MesserschmittBf110_Aircraft();
    }
}
```

კლასი RuEngineryFactory შეესაბამება კლასს `ConcreteFactory2`. კლასი `E25_Tank` ტანკი შეესაბამება კლასს `ConcreteProductA2`:

```
public class E25_Tank extends Tank {
    private static final String NAME = "E-25";
    private static final String COUNTRY = "Russia";

    public String getDescription() {
        return NAME + " " + COUNTRY;}}
```

კლასი IL2Bf110_Aircraft შეესაბამება კლასს ConcreteProductB2:

```
public class IL2Bf110_Aircraft extends Aircraft {
    private static final String NAME = "IL-2";
    private static final String COUNTRY = "Russia";

    public String getDescription() {
        return NAME + " " + COUNTRY;
    }
}
```

დიაგრამის ყველა კლასი უკვე შექმნილია. მაგალითის დემონსტრირებისათვის შევქმნათ კლასი AbstractFactoryDemo. მასში შემოვიტანოთ მეთოდი showEnginery(), რომელიც ფაბრიკას მიიღებს, როგორც არგუმენტს და გამოიტანს ინფორმაციას მის სამხედრო ტექნიკაზე და გადავცეთ მას GEOEngineryFactory и RuEngineryFactory ფაბრიკების ეგზემპლიარები:

```
public class AbstractFactoryDemo {
    public static void main(String[] args) {
        EngineryFactory engineryFactory = new GEOEngineryFactory();

        System.out.println("Georgia enginery:");
        showEnginery(engineryFactory);

        System.out.println();

        System.out.println("Russia enginery:");
        engineryFactory = new RuEngineryFactory();
        showEnginery(engineryFactory);
    }

    public static void showEnginery(EngineryFactory engineryFactory) {
        Tank tank = engineryFactory.createTank();
        System.out.println(tank.getDescription());

        Aircraft aircraft = engineryFactory.createAircraft();
```

```
        System.out.println(aircraft.getDescription());
    }
}
```

გავუშვათ შესრულებაზე მეთოდი `main()`:

შესრულების შედეგი იქნება:

```
Georgia enginery:
T-34 Georgia
Geo-2 Georgia
Russia engonery:
```

პროგრამის მუშაობის შედეგად ჩვენ მივიღეთ იმ სამხედრო ტექნიკის აღწერა, რომელსაც ფლობს თითოეული სახელმწიფო. საჭიროების შემთხვევაში ჩვენ შეგვიძლია მარტივად დავამატოთ ახალი ფაბრიკები და ასევე შევცვალოთ ერთი მეორეთი.

ქცევითი დიაზინის ნიმუში Command

დავუშვათ, ჩვენ ვქმნით კლასს Switch, რომელიც ჩართავს/გამორთავს ნათურას

```
public class Switch {
    private Light light;
    public Switcher(Light light){
        this.light = light;
    }
    public void flipUp(){
        light.turnOn();
    }
    public void flipDown(){
        light.turnOff();
    }
}
```

როგორც მოყვანილი მაგალითიდან ჩანს, გვაქვს ჩვეულებრივი კლასი, რომელიც უზრუნველყოფს ერთი ობიექტის ჩართვა/გამორთვას (ობიექტი light). ანალოგიური მოქმედების ტელევიზორისთვის გამოსაყენებლად მოგვიწევს პროგრამის გადაკეთება.

```
public class Switcher {
```

```
    private TV tv;
    public Switcher(TV tv){
        this.tv = tv;
    }
    public void tvOn(){
        tv.on();
    }
    public void tvOff(){
        tv.off();
    }
}
```

ობიექტზე ორიენტირებული დაპროგრამების ერთ-ერთი ძირითადი პრინციპი მოითხოვს, რომ კლასი იყოს ღია გაფართოებისთვის და დახურული ცვლილებისთვის. ამ შემთხვევაში, ჩვენ სწორედ ეს პრინციპი დავარღვიეთ. ყოველ ჯერზე, როცა შეიცვლება ხელსაწყო, რომლის გამორთვა ან ჩართვა გვჭირდება, ჩვენ უკვე გამზადებულ კლასში უნდა მოვახდინოთ ცვლილებები.

ქცევითი დიზაინის ნიმუშ Command-ს თუ გამოვიყენებთ, ჩვენი კლასი დაცული იქნება ცვლილებებისაგან. ამ შაბლონში არსებობს საბაზო აბსტრაქტული კლასი (ინტერფეისი), რომლის რეალიზაციასაც ახდენენ კონკრეტული ბრძანებები.

ინტერფეისს ამ შემთხვევაში ექნება სახე:

```
public interface Command {
    public void execute();
}
```

ხოლო კლას-ბრძანებების ჩაწერა შესაძლებელიაშ სახით:

```
public class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;}
```

```
    public void execute() {
        light.turnOff();
    }
}
```

```
public class TvOnCommand implements Command {
    private TV tv;

    public TvOnCommand(TV tv) {
        this.tv = tv;
    }

    public void execute() {
        tv.on();
    }
}
```

ამ შემთხვევაში ჩვენ გვაქვს კლასები, რომელბიც უზრუნველყოფენ როგორც ტელევიზორის, ისე ნათურის ჩართვა/გამორთვას. Switch კლასის ახალი ვარიანტი უკვე შემდეგი სახით გამოიყურება:

```
public class Switch {
    private Command onCommand;
    private Command offCommand;

    public Switch(Command onCommand, Command offCommand) {
          this.onCommand = onCommand;
          this.offCommand = offCommand;
    }

    public void on() {
          onCommand.execute();
    }
```

```
    public void off() {
          offCommand.execute();
    }
}
```

როგორც ვხედავთ, კონსტრუქტორში ხდება იმ ბრძანებების გადაცემა, რომლებიც ასრულებენ ყველა მოქმედებას. Switch მიმართავს მხოლოდ მათ, როცა უნდა გამორთოს/ჩართოს ხელსაწყო. ამ კლასში არ არის არანაირი ინფორმაცია ხელსაწყოების შესახებ. იმისათვის, რომ ის ამუშავდეს ახალი მოწყობილობისთვის - უნდა შევქმნათ ახალი ბრძანება და გადავცეთ ის Switch-ს. ჩვენი კოდი უკვე არ არღვევს Opend/Closed Principle-ს.

ქცევითი დიაზინის ნიმუში - დამკვირვებელი

“დამკვირვებელი” ნიმუშის რეალიზაცია გამოიყენება სისტემაში ობიექტების მდგომარეობის დაკვირვებისათვის. თუ ობიექტის მდგომარეობა იცვლება სასიცოცხლო ციკლის პროცესში, „დამკვირვებელი“ სისტემის სხვა ნაწილებს აცნობებს ამ შემთხვევის შესახებ.

„დამკვირვებელს“ უნდა გააჩნდეს ღია მეთოდები, რომელთა საშუალებითაც მოახდენს ობიექტის ცვლილების შესახებ ინფორმაციის გადაცემას. ამ მეთოდს ხშირად უწოდებენ notify-ს. რამდენადაც დამკვირვებელი შესაძლებელია იყოს რამდენიმე, მუშაობის გასამარტივებლად შესაძლებელია გამოვიყენოთ დამკვირვებლების კოლექცია - collection of observers.

```
public interface Observer {
        void objectCreated(Object obj);
        void objectModified(Object obj);
    }
    class EmptyObserver implements Observer {
        public void objectCreated(Object obj) { }
        public void objectModified(Object obj) { }
    }
    class Observers<T extends Observer> extends ArrayList<T> {
        public void notifyObjectCreated(Object obj) {
            for (Iterator<T> iter = (Iterator<T>) iterator(); iter.hasNext();)
                iter.next().objectCreated(obj);
        }
        public void notifyObjectModified(Object obj) {
            for (Iterator<T> iter = (Iterator<T>) iterator(); iter.hasNext();)
```

```
                iter.next().objectModified(obj);
        }
    }
```

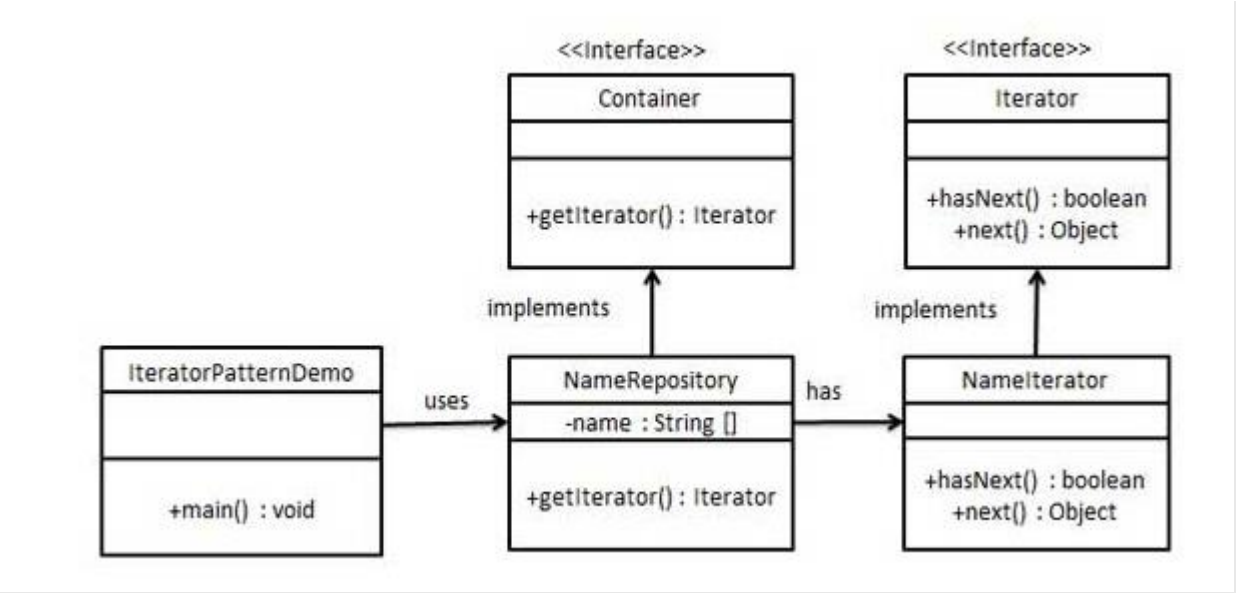
კლასი EmptyObserver სასარგებლოა იმ შემთხვევაში, როცა დამკვირვებელს გააჩნია საკმარისად დიდი რაოდენობის notify მეთოდები. ამ შემთხვევაში, ანონიმური კლასების გამოყენებით შესაძლებელია შეიქმნას ჩვენთვის აუცილებელი „ვიწრო სპეციალიზირებული“ დამკვირვებლები.

```
Observers observers = new Observers();
  observers.add(new EmptyObserver() {
    public void objectCreated(Object obj) { /* realization */ }
  });
```

კლასი Observers განთავსდება სუბიექტში, რომელზეც დაკვირვება ხდება. კოდის ნებისმიერ ადგილზე, სადაც კლას-სუბიექტთან ჩვენთვის საინტერესო მოქმედებები სრულდება, ვამატებთ დამკვირვებელთა კოლექციიდან შესაბამისი notify მეთოდის გამოძახებას.

```
public class Subject {
        Observers observers = new Observers();

        private Object field;

        public void setField(Object o) {
            field = o;
            observers.notifyObjectModified(this);
        }
    }
```

იტერატორი



ნახ. 417

ინტერფეისის შექმნა

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}
```

Container.java

```
public interface Container {
   public Iterator getIterator();
}
```

შევქმნათ კონკრეტული კლასი *Container* ინტერფეისის რეალიზაციისათვის. ეს კლასი თავის მხრივ მოიცავს კლასს NameIterator, რომელიც უზრუნველყოფს *Iterator ინტერფეისის რეალიზაციას.*

NameRepository.java

```
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @Override
   public Iterator getIterator() {
```

```
      return new NameIterator();
   }

   private class NameIterator implements Iterator {

      int index;

      @Override
      public boolean hasNext() {

         if(index < names.length){
            return true;
         }
         return false;
      }

      @Override
      public Object next() {

         if(this.hasNext()){
            return names[index++];
         }
         return null;
      }
   }
}
```

გამოვიყენოთ NameRepository იტერატორის მისაღებად და სახელების გამოსატანად.

IteratorPatternDemo.java

```
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
```

```
            System.out.println("Name : " + name);
        }
    }
}
```

შედეგი:

```
Name : Robert
Name : John
Name : Julie
Name : Lora
```

შემაჯამებელი დავალებები :

თითოეული დავალებისათვის შექმენით UML დიაგრამა და შესაბამისი პროგრამული კოდი

ნიმუში „სტრატეგია“.

პროექტი „პრინტერები“. პროექტში რეალიზებულ უნდა იქნეს პრინტერების სხვადასხვა მოდელები, რომლებიც ასრულებენ სხვადასხვა ტიპის ბეჭდვას.

ნიმუში „ფაბრიკული მეთოდი“

პროექტი „სმარტფონების ფაბრიკა“. პროექტში უნდა იყოს რეალიზებული სმარტფონების შექმნა სხვადასხვა მახასიათებლებით.

ნიმუში „აბსტრაქტული ფაბრიკა“.

პროექტი „ავტომობილების ქარხანა“. პროექტში შესაძლებელი უნდა იყოს სხვადასხვა ტიპის ავტომობილების შექმნა სხვადასხვა ქარხნებში.

ნიმუში „ბრძანება“.

პროექტი „კალკულატორის კლავიატურა“. ციფრულ და არითმეტიკულ ღილაკებს უნდა გააჩნდეთ ფიქსირებული ფუნქციები, ხოლო დანარჩენი ღილაკების ფუნქციების ცვლილება შესაძლებელი უნდა იყოს.

პრაქტიკული ამოცანები, სავარჯიშოები და დავალებები

გამოყენებული ლიტერატურა:

1. H. Schildt “Java: The Complete Reference”. Seventh Edition. NewYork, 2007.
2. www.youtube.com/watch?v
3. http://www.tutorialspoint.com/java/index.htm
4. http://www.javatpoint.com/java-tutorial
5. https://www.thenewboston.com/videos.php?cat=31
6. ზ. ბოსიკაშვილი, დ. კაპანაძე, ნ. არაბული „ვიზუალური დაპროგრამება (Java)”. თბილისი 2006.
7. ლ. გაჩეჩილაძე, ნ. ჯიბლაძე, ნ. კურკუმული „ალგორითმიზაციის საფუძვლები“. სტუ, 2004.
8. თ. მშვიდობაძე. „ალგორითმები და მონაცემთა სტრუქტურები I“. 2006
9. Cormen,TH. Introduction to Algorithms 2nd-ed. New York, 2001.
10. Wirth, N., Algorithms & Data Structures. Prentice-Hall, 1986.
11. Hiebeler, R.C., Structural Analysis, Sixth Edition, 2006.
12. http://algolist.manual.ru/
13. https://lobjana.wordpress.com/
14. https://www.google.ge/search?q
15. З.Э.Фримен, К.Сьерра, Б.Бейтс Паттерны проектирования. СПб.: Питер, 2011.
16. Классическая книга “банды четырех”. Э. Гамма, Р. Хелм, Р. Джонсон, Д. Влиссидес. Приемы объектноориентированного проектирования. Паттерны проектирования. (2015).
17. Д. Кнут "Исскуство программирования", 3-е изд., 2001
18. Т. Кормен, Ч. Лейзерсон, Р. Ривест "Алгоритмы: построение и анализ", М., МЦНМО,
19. Дж. Макконнелл "Основы современных алгоритмов", 2-е дополненное издание
20. Х. Дейтел, П. Дейтел "Как программировать на С"
21. ბ.მეფარიშვილი, გ.ჯანელიძე Web დანართების პროგრამული რეალიზაცია MySQL-ის გამოყენებით, თბილისი, 2015
22. http://www.uml.org/
23. https://sourcemaking.com/design_patterns/structural_patterns
24. http://www.tutorialspoint.com/java/java_networking.htm
25. https://www.java.com
26. http://www.tutorialspoint.com/mysql/

Abstract factory pattern

```
/GuiFactory example

//Abstract Product
interface Button {
	void paint();
}
//Abstract Product
interface Label {
	void paint();
}

//Abstract Factory
interface GUIFactory {
	Button createButton();
	Label createLabel();
}

//Concrete Factory
class WinFactory implements GUIFactory {
	public Button createButton() {
		return new WinButton();
	}

	public Label createLabel() {
		return new WinLabel();
	}
}

//Concrete Factory
class OSXFactory implements GUIFactory {
	public Button createButton() {
		return new OSXButton();
	}

	public Label createLabel() {
		return new OSXLabel();
	}

}

//Concrete Product
class OSXButton implements Button {
```

```
        public void paint() {
                System.out.println("I'm an OSXButton");
        }
}

//Concrete Product
class WinButton implements Button {
        public void paint() {
                System.out.println("I'm a WinButton");
        }
}

//Concrete Product
class OSXLabel implements Label {
        public void paint() {
                System.out.println("I'm an OSXLabel");
        }
}

//Concrete Product
class WinLabel implements Label {
        public void paint() {
                System.out.println("I'm a WinLabel");
        }
}

//Client application is not aware about the how the product is created. Its
only responsible to give a name of
//concrete factory
class Application {
        public Application(GUIFactory factory) {
                Button button = factory.createButton();
                Label label = factory.createLabel();
                button.paint();
                label.paint();
        }
}

public class ApplicationRunner {
        public static void main(String[] args) {
                new Application(createOsSpecificFactory());
        }

        public static GUIFactory createOsSpecificFactory() {
                String osname = System.getProperty("os.name").toLowerCase();
                if(osname != null && osname.contains("windows"))
```

```
                return new WinFactory();
            else
                return new OSXFactory();
        }
}
```

Builder pattern

```
public class StreetMap {
        private final Point origin;
        private final Point destination;

        private final Color waterColor;
        private final Color landColor;
        private final Color highTrafficColor;
        private final Color mediumTrafficColor;
        private final Color lowTrafficColor;

        public static class Builder {
                // Required parameters
                private final Point origin;
                private final Point destination;

                // Optional parameters - initialize with default values
                private Color waterColor         = Color.BLUE;
                private Color landColor          = new Color(30, 30, 30);
                private Color highTrafficColor   = Color.RED;
                private Color mediumTrafficColor = Color.YELLOW;
                private Color lowTrafficColor    = Color.GREEN;

                public Builder(Point origin, Point destination) {
                        this.origin      = origin;
                        this.destination = destination;
                }

                public Builder waterColor(Color color) {
                        waterColor = color;
                        return this;
                }

                public Builder landColor(Color color) {
                        landColor = color;
                        return this;
                }
```

```
                public Builder highTrafficColor(Color color) {
                        highTrafficColor = color;
                        return this;
                }

                public Builder mediumTrafficColor(Color color) {
                        mediumTrafficColor = color;
                        return this;
                }

                public Builder lowTrafficColor(Color color) {
                        lowTrafficColor = color;
                        return this;
                }

                public StreetMap build() {
                        return new StreetMap(this);
                }

        }

        private StreetMap(Builder builder) {
                // Required parameters
                origin      = builder.origin;
                destination = builder.destination;

                // Optional parameters
                waterColor         = builder.waterColor;
                landColor          = builder.landColor;
                highTrafficColor   = builder.highTrafficColor;
                mediumTrafficColor = builder.mediumTrafficColor;
                lowTrafficColor    = builder.lowTrafficColor;
        }

        public static void main(String args[]) {
                StreetMap map = new StreetMap.Builder(new Point(50, 50), new
Point(100,

        100)).landColor(Color.GRAY).waterColor(Color.BLUE.brighter())
                                .build();
        }
}
```

Factory method pattern

```
public abstract class MazeGame {
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        this.addRoom(room1);
        this.addRoom(room2);
    }

    abstract protected Room makeRoom();
}
```

```
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}

public class OrdinaryMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}

MazeGame ordinaryGame = new OrdinaryMazeGame();
MazeGame magicGame = new MagicMazeGame();
```

Prototype pattern

```
// Prototype pattern

        public abstract class Prototype implements Cloneable {
                public abstract Prototype clone();
```

```
        }

        public class ConcretePrototype1 extends Prototype {
                @Override
                public Prototype clone() {
                        return super.clone();
                }
        }

        public class ConcretePrototype2 extends Prototype {
                @Override
                public Prototype clone() {
                        return super.clone();
                }
        }
```

Singleton pattern

```
public class SingletonDemo {
    private static volatile SingletonDemo instance;
    private SingletonDemo() { }

    public static SingletonDemo getInstance() {
        if (instance == null ) {
            synchronized (SingletonDemo.class) {
                if (instance == null) {
                    instance = new SingletonDemo();
                }
            }
        }

        return instance;
    }
}
```

зб

```
public class SingletonDemo {
    private static volatile SingletonDemo instance = null;
    private SingletonDemo() { }

    public static synchronized SingletonDemo getInstance() {
        if (instance == null) {
            instance = new SingletonDemo();
        }

        return instance;
    }
}
```

Bridge pattern

```
/** "Implementor" */
interface DrawingAPI {
    public void drawCircle(double x, double y, double radius);
}

/** "ConcreteImplementor"  1/2 */
class DrawingAPI1 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
    }
}

/** "ConcreteImplementor" 2/2 */
class DrawingAPI2 implements DrawingAPI {
    public void drawCircle(double x, double y, double radius) {
        System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
    }
}

/** "Abstraction" */
abstract class Shape {
    protected DrawingAPI drawingAPI;

    protected Shape(DrawingAPI drawingAPI){
        this.drawingAPI = drawingAPI;
    }
```

```
    public abstract void draw();                                    // low-level
    public abstract void resizeByPercentage(double pct);      // high-level
}

/** "Refined Abstraction" */
class CircleShape extends Shape {
    private double x, y, radius;
    public CircleShape(double x, double y, double radius, DrawingAPI
drawingAPI) {
        super(drawingAPI);
        this.x = x;  this.y = y;  this.radius = radius;
    }

    // low-level i.e. Implementation specific
    public void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }
    // high-level i.e. Abstraction specific
    public void resizeByPercentage(double pct) {
        radius *= (1.0 + pct/100.0);
    }
}

/** "Client" */
class BridgePattern {
    public static void main(String[] args) {
        Shape[] shapes = new Shape[] {
            new CircleShape(1, 2, 3, new DrawingAPI1()),
            new CircleShape(5, 7, 11, new DrawingAPI2())
        };

        for (Shape shape : shapes) {
            shape.resizeByPercentage(2.5);
            shape.draw();
        }
    }
}
```

Composite pattern

```
/** "Component" */
interface Graphic {

    //Prints the graphic.
    public void print();
}

/** "Composite" */
import java.util.List;
import java.util.ArrayList;
class CompositeGraphic implements Graphic {

    //Collection of child graphics.
    private List<Graphic> childGraphics = new ArrayList<Graphic>();

    //Prints the graphic.
    public void print() {
        for (Graphic graphic : childGraphics) {
            graphic.print();
        }
    }

    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        childGraphics.add(graphic);
    }

    //Removes the graphic from the composition.
    public void remove(Graphic graphic) {
        childGraphics.remove(graphic);
    }
}

/** "Leaf" */
class Ellipse implements Graphic {

    //Prints the graphic.
    public void print() {
```

```
        System.out.println("Ellipse");
    }
}

/** Client */
public class Program {

    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialize three composite graphics
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes the graphics
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);

        graphic2.add(ellipse4);

        graphic.add(graphic1);
        graphic.add(graphic2);

        //Prints the complete graphic (four times the string "Ellipse").
        graphic.print();
    }
}
```

Decorator pattern

```
// The Window interface class
public interface Window {
    public void draw(); // Draws the Window
    public String getDescription(); // Returns a description of the Window
}

// Extension of a simple Window without any scrollbars
class SimpleWindow implements Window {
    public void draw() {
        // Draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
```

Command pattern

```
import java.util.List;
import java.util.ArrayList;

/** The Command interface */
public interface Command {
   void execute();
}

/** The Invoker class */
public class Switch {
   private List<Command> history = new ArrayList<Command>();

   public void storeAndExecute(Command cmd) {
      this.history.add(cmd); // optional
      cmd.execute();
```

```
    }
}

/** The Receiver class */
public class Light {

    public void turnOn() {
        System.out.println("The light is on");
    }

    public void turnOff() {
        System.out.println("The light is off");
    }
}

/** The Command for turning on the light - ConcreteCommand #1 */
public class FlipUpCommand implements Command {
    private Light theLight;

    public FlipUpCommand(Light light) {
        this.theLight = light;
    }

    @Override    // Command
    public void execute() {
        theLight.turnOn();
    }
}

/** The Command for turning off the light - ConcreteCommand #2 */
public class FlipDownCommand implements Command {
    private Light theLight;

    public FlipDownCommand(Light light) {
        this.theLight = light;
    }

    @Override    // Command
    public void execute() {
        theLight.turnOff();
    }
}

/* The test class or client */
public class PressSwitch {
    public static void main(String[] args){
```

```
        // Check number of arguments
        if (args.length != 1) {
            System.err.println("Argument \"ON\" or \"OFF\" is required.");
            System.exit(-1);
        }

        Light lamp = new Light();
        Command switchUp = new FlipUpCommand(lamp);
        Command switchDown = new FlipDownCommand(lamp);

        Switch mySwitch = new Switch();

        switch(args[0]) {
            case "ON":
                mySwitch.storeAndExecute(switchUp);
                break;
            case "OFF":
                mySwitch.storeAndExecute(switchDown);
                break;
            default:
                System.err.println("Argument \"ON\" or \"OFF\" is required.");
                System.exit(-1);
        }
    }
}
```

Strategy pattern

```
import java.util.ArrayList;
import java.util.List;

public class StrategyPatternWiki {

        public static void main(String[] args) {
                Customer a = new Customer(new NormalStrategy());

                // Normal billing
                a.add(1.0, 1);

                // Start Happy Hour
                a.setStrategy(new HappyHourStrategy());
                a.add(1.0, 2);

                // New Customer
```

```
                Customer b = new Customer(new HappyHourStrategy());
                b.add(0.8, 1);
                // The Customer pays
                a.printBill();

                // End Happy Hour
                b.setStrategy(new NormalStrategy());
                b.add(1.3, 2);
                b.add(2.5, 1);
                b.printBill();

        }
}

class Customer {

        private List<Double> drinks;
        private BillingStrategy strategy;

        public Customer(BillingStrategy strategy) {
                this.drinks = new ArrayList<Double>();
                this.strategy = strategy;
        }

        public void add(double price, int quantity) {
                drinks.add(strategy.getActPrice(price * quantity));
        }

        // Payment of bill
        public void printBill() {
                double sum = 0;
                for (Double i : drinks) {
                        sum += i;
                }
                System.out.println("Total due: " + sum);
                drinks.clear();
        }

        // Set Strategy
        public void setStrategy(BillingStrategy strategy) {
                this.strategy = strategy;
        }

}

interface BillingStrategy {
```

```
        public double getActPrice(double rawPrice);
}

// Normal billing strategy (unchanged price)
class NormalStrategy implements BillingStrategy {

        @Override
        public double getActPrice(double rawPrice) {
                return rawPrice;
        }

}

// Strategy for Happy hour (50% discount)
class HappyHourStrategy implements BillingStrategy {

        @Override
        public double getActPrice(double rawPrice) {
                return rawPrice*0.5;
        }

}
```