შესავალი
ობიექტზე ორიენტირებული დაპროგრამების საწყისები

პროგრამისტების წინაშე მდგარი ამოცანები სულ უფრო და უფრო მრავალფეროვანი ხდება. ამასთან მაღალი ტემპით იზრდება დასამუშავებელი ინფორმაციის ზომა. თუ ცოტა ხნის წინათ ინფორმაციის განზომილების ჩვეულებრივი ერთეული იყო კილობაიტი და მეგაბაიტი, დღეს უკვე ლაპარაკია გიგაბაიტებსა და ტერაბაიტებზე. როგორც კი პროგრამისტები მოახდენენ მათ წინაშე მდგარი ამოცანის მეტნაკლებად მისაღებ გადაწყვეტას, მაშინვე წარმოიშვება ახალი, უფრო რთული ამოცანები. პროგრამისტები იგონებენ და ამუშავებენ ახალ მეთოდებს, ქმნიან ახალ ენებს. პროგრამირების ნახევარსაუკუნოვანი ისტორიის განმავლობაში წარმოიქმნა ასობით ენა, დამუშავებულია დაპროგრამების უამრავი მეთოდი და სტილი. ზოგიერთი მათგანი საყოველთაოდ მიღებული ხდება და გარკვეული პერიოდის განმავლობაში წარმოადგენენ **პროგრამირების პარადიგმას** (ძირითად მიდგომას). პროგრამირების პარადიგმა განმარტებით ლექსიკონში განმარტებულია, როგორც კონცეპტუალიზაციის მეთოდი, რომელიც განსაზღვრავს, როგორ ვაწარმოოთ გამოთვლები და როგორ იყოს ორგანიზებული და სტრუქტურირებული კომპიუტერის მიერ შესრულებული სამუშაოები.

პროგრამირების პარადიგმები

თავიდან, მანქანურ კოდებში დაწერილი უმარტივესი პროგრამაც კი, შეადგენდა ძნელად გასაგები, ასობით სტრიქონისგან შემდგარ ტექსტს. დაპროგრამების დაჩქარებისა და გამარტივების მიზნით შეიქმნა მაღალი დონის ალგორითმული ენები: Fortran, Algol და ასობით სხვა. ამ ენების გამოყენებით მანქანური ბრძანებების (კოდის) შექმნის რუტინული ოპერაციების განხორციელება დაეკისრა კომპილატორებს. მაღალი დონის ენებზე დაწერილი იგივე პროგრამები გახდა გაცილებით გასაგები და მოკლე. კომპიუტერული სისტემების მკვეთრმა განვითარებამ გამოიწვია უფრო რთული ამოცანების გადაწყვეტის აუცილებლობა, რამაც კვლავ მოახდინა პროგრამების ზომების მნიშვნელოვანი გაზრდა.

წარმოიქმნა იდეა: პროგრამა გაფორმდეს რამდენიმე, შეძლებისდაგვარად მარტივი პროცედურის ან ფუნქციის სახით, რომლებიც გადაწყვეტენ თავის კონკრეტულ ამოცანას. მცირე ზომის პროცედურის დაწერა, კომპილაცია და გამართვა გაცილებით უფრო ადვილად და სწრაფად შეიძლება. შემდეგ საჭირო იქნება, მხოლოდ ყველა პროცედურის საჭირო მიმდევრობით, ერთ პროგრამად გაერთიანება. გარდა ამისა, ერთხელ დაწერილი პროცედურები შეიძლება გამოყენებული იქნას სხვა პროგრამებში, როგორც „სამშენებლო აგურები“. **პროცედურული პროგრამირება** სწრაფად იქცა პარადიგმად. ყველა მაღალი დონის ენაში მოხდა პროცედურებისა და ფუნქციების დაწერის საშუალებების ჩართვა. სხვადასხვა საგნობრივი სფეროსათვის პროცედურებისა და ფუნქციების უამრავი ბიბლიოთეკა შეიქმნა.

პროგრამისტების წინაშე კვლავ დაისვა ახალი საკითხები: როგორ ჩამოყალიბდეს პროგრამის სტრუქტურა მისი პროცედურებად დასაყოფად, კოდის რა ნაწილი გამოიყოს ცალკეული პროცედურისათვის, როგორ გახდეს ამოცანის გადაწყვეტის ალგორითმი მარტივი და აღქმადი, როგორაა მოსახერხებელი პროცედურების ერთმანეთთან დაკავშირება. დაგროვილი მრავალწლიანი გამოცდილების მიხედვით, პროგრამისტებმა შეიმუშავეს რეკომენდაციები, რომლებსაც დაერქვა **სტრუქტურული პროგრამირება**. სტრუქტურული პროგრამირება იმდენად მოსახერხებელი და მნიშვნელოვანი გამოდგა, რომ იგი მალე პარადიგმა გახდა. შეიქმნა პროგრამირების ახალი ენები (მაგალითად, Pascal), რომლებზეც სტრუქტურული პროგრამების წერა გაცილებით მოსახერხებელია. უფრო მეტიც, არასტრუქტურირებული პროგრამების დაწერა ძალიან გაძნელდა. სტრუქტურული პროგრამირების ძირითადი მოთხოვნებია:

- მოდულურობა;
- კითხვადობა;
- პროგრამის შესრულების სტანდარტული მართვის სქემები;
- დოკუმენტირება;
- ტესტების შედგენა პროგრამის დამუშავების ადრეულ ეტაპზე.

პროგრამისტების წინაშე დამდგარი ამოცანების სირთულემ აქაც იჩინა თავი: დიდი პროგრამა შედგებოდა ასობით პროცედურისაგან და იგი კვლავ გახდა ძნელად გასააზრებელი

და გასამართი გახდა. აუცილებელი შეიქმნა პროგრამირების ახალი სტილის შემუშავება.

ამ დროისათვის გასაგები გახდა, რომ საწყისი მონაცემების სტრუქტურა დიდ გავლენას ახდენდა მათი დამუშავების ალგორითმებზე. ზოგიერთი საწყისი მონაცემი მოსახერხებელია წარმოდგენილი იქნეს მასივის სახით, ზოგიერთისათვის კი უფრო უპრიანია ხისებრი (იერარქიული) სტრუქტურა ან სტეკური ორგანიზება. ამიტომ წარმოიქმნა იდეა ერთ მოდულში გაერთიანდეს საწყისი მონაცემები და მისი დამუშავების ყველა პროცედურა. მოდულური პროგრამირების ეს იდეა სწრაფად გახდა პოპულარული და რაღაც დროის განმავლობაში იგი პარადიგმად იქცა.. პროგრამები იქმნებოდა ცალკეული მოდულებისაგან, რომლებიც ათობით სხვადასხვა პროცედურისა და ფუნქციისაგან შედგებოდა. ასეთი პროგრამების ეფექტურობა მით უფრო მაღალია, რაც უფრო ნაკლებადაა დამოკიდებული ცალკეული მოდულები ერთმანეთზე. მოდულების ავტონომიურობა საშუალებას იძლევა შეიქმნას მოდულების ბიბლიოთეკა, რომელიც შემდგომში შეიძლება გამოყენებულ იქნეს პროგრამის ასაგებ ბლოკებად.

მოდულების ერთმანეთის მიმართ მაქსიმალური ავტონომიურობის უზრუნველსაყოფად, საჭიროა მკაფიოდ გამოიყოს ის პროცედურები, რომლის გამოძახებაც შესაძლებელია სხვა მოდულებისგან - ანუ **ღია** (Public) პროცედურები, და დამხმარე პროცედურები, რომლებიც ამუშავებენ ამ მოდულში მოთავსებულ მონაცემებს - ანუ **დახურული** (Private - პრივატული) პროცედურები. პირველი ტიპის პროცედურები ჩა-

მოთვლილია მოდულის ცალკე ნაწილში - **ინტერფეისში** (interface), ხოლო მეორე ტიპის კი მონაწილეობს მხოლოდ მოდულის **რეალიზაციაში** (implementation). მოდულში აღწერილი ცვლადებიც იყოფა ღიად (ინტერფეისში მითითებული და სხვა მოდულებიდან წვდომადი) და დახურულად (მათზე წვდომა შესაძლებელია მხოლოდ იმავე მოდულში შემავალი პროცედურებისაგან). პროგრამირების სხვადასხვა ენაში ეს დაყოფა სხვადასხვანაირად ხდება. Turbo Pascal-ში მოდული სპეციალურად იყოფა ინტერფეისად და რეალიზაციად. ალგორითმულ ენა C-ში ინტერფეისი გატანილია „სათაო“ (header) ფაილში. ენა C++ -ში, „სათაო“ (header) ფაილში გატანის გარდა, ინტერფეისის აღწერისათვის შესაძლებელია აბსტრაქტული კლასების გამოყენება. Java-ში არსებობს ინტერფეისის აღწერის სპეციალური კონსტრუქცია, რომელსაც ასევე ეწოდება - interface, მაგრამ შესაძლებელია აბსტრაქტული კლასების შექმნაც.

ასე წარმოიშვა მონაცემებისა და მათი დამუშავების მეთოდების დაფარვის, **ინკაფსულაციის** (incapsulation) იდეა. ინკაფსულაციის მიზანი არაა სხვა მოდულს დაუმალოს საჭირო კომპონენტი. მისი ორი მთავარი მიზანია: პირველი - უზრუნველყოს მოდულის გამოყენების უსაფრთხოება, ინტერფეისში გამოიტანოს და გახადოს საყოველთაოდ ხელმისაწვდომი მხოლოდ ინფორმაციის დამუშავების ის მეთოდები, რომლებსაც არ შეუძლიათ დააზიანოს ან წაშალოს საწყისი მონაცემები; მეორე - შეამციროს სირთულე, გარე მოდულებისათვის რეალიზაციის წვრილმანი დეტალების დაფარვის გზით.

კვლავ წარმოიშვა ამოცანა, რა გზით მოხდეს პროგრამის დაყოფა მოდულებად. ამ შემთხვევაში უფრო მისაღები აღმოჩნდა პროგრამირების ძველი ამოცანების გადაწყვეტები - ბუნებრივი და ხელოვნური ობიექტების მოქმედებების მოდელირება (ადამიანების, ცხოველების, ტექნოლოგიური პროცესების მართვის სისტემების, პროგრამული მართვის ჩარხების და სხვა). მართლაც, ყოველი ობიექტი - ადამიანი, ავტომობილი, ტექნოლოგიური პროცესი, ხასიათდება გარკვეული პარამეტრებით. მაგალითად, გვარი, წლოვანება, წონა, სიმაღლე, მაქსიმალური სიჩქარე, ტვირთამწეობა და სხვა. ობიექტი შეიძლება ასრულებდეს სხვადსხვა მოქმედებებს: გადაადგილდებოდეს სივრცეში, იზრდებოდეს ან მცირდებოდეს, ჭამოს, სვას, შეიცვალოს თავისი პირვანდელი პარამეტრები. მოსახერხებელია ობიექტის მოდელირება მოხდეს მოდულის სახით. მისი პარამეტრები, მახასიათებლები იქნება საწყისი მონაცემები - მუდმივი ან ცვლადები, ხოლო მოქმედებები - პროცედურები.

მოსახერხებელი გამოდგა, აგრეთვე, საწინააღმდეგოც - პროგრამა ისე დაიყოს მოდულებად, რომ იგი გარდაიქმნას ურთიერთდაკავშირებულ ობიექტებად. ასე წარმოიშვა **ობიექტზე ორიენტირებული დაპროგრამება** (object-oriented programming - OOP), რომელიც პროგრამირების თანამედროვე პარადიგმას წარმოადგენს.

ობიექტზე ორიენტირებული დაპროგრამების იდეის წარმოშობას სხვა წანამძღვრებიც გააჩნია. ცნობილია, რომ ყოველი კომპიუტერული პროგრამა შედგება ორი ელემენტისაგან: კოდი (პროგრამის ტექსტის პროცედურული ნაწილი) და

მონაცემები. კონცეპტუალურად პროგრამა ორგანიზებული შეიძლება იყოს თავისი კოდის ან მონაცემების გარშემო. ანუ, ზოგიერთი პროგრამის ორგანიზება განისაზღვრება იმით, თუ „რა ხდება“, ხოლო სხვების „რაზე ხდება მოქმედებები“. არსებობს პროგრამების შედგენის ორი კონცეფცია. პირველ მიდგომას უწოდებენ **პროცესზე ორიენტირებულ მოდელს**. ასეთი მიდგომისას პროგრამა ესაა წრფივი ბიჯების მიმდევრობა (ე.ი. კოდი). პროცესზე ორიენტირებული მოდელი შეიძლება განვიხილოთ, როგორც მონაცემების დამმუშავებელი კოდი. ამ მოდელს იყენებენ პროცედურული ენები მაგალითად, C, Pascal და სხვა. თუმცა, ასეთი მიდგომა წარმოშობს სხვადასხვა პრობლემებს პროგრამის ზომისა და სირთულის ზრდასთან დაკავშირებით.

როგორც ზემოთ აღვნიშნეთ, სირთულის ზრდადი ხასიათის გადასალახად, დამუშავდა ობიექტზე ორიენტირებული მიდგომა. ობიექტზე ორიენტირებული პროგრამირება ახორციელებს პროგრამის ორგანიზებას მისი მონაცემების (ანუ ობიექტების) გარშემო. ობიექტზე ორიენტირებული პროგრამირება ესაა მონაცემები, რომლებიც მართავს კოდთან წვდომას.

ობიექტების სახით შეიძლება წარმოვადგინოთ სხვადასხვა ცნებები. მაგალითად, ფანჯარა დისპლეის ეკრანზე - ესაა ობიექტი, რომელსაც აქვს სიგანე width და სიმაღლე hight, ეკრანზე მდებარეობა, ჩვეულებრივ აღიწერება მარცხენა ზედა კუთხის (x,y) კოორდინატებით, შრიფტი, რომლითაც ფანჯარაში ტექსტი გამოდის მაგალითად, Times New Roman, ფონის ფერი color და სხვა პარამეტრები. ფანჯარა ეკრანზე

შეიძლება გადაადგილდებოდეს move() მეთოდით, გაეზარდოს ან შეუმცირდეს ზომები მეთოდით size(), გარკვეული რეაგირება მოახდინოს მაუსის დაჭერაზე და სხვა. ფაქტობრივად, ესაა სრულყოფილი ობიექტი. ღილაკები, ლიფტები და ფანჯრის სხვა ელემენტებიც ობიექტებს წარმოადგენენ, რომლებსაც, თავის მხრივ აქვთ თავისი ზომები, შრიფტები, შეუძლიათ გადაადგილება.

ობიექტზე ორიენტირებული დაპროგრამების იდეა ძალიან ნაყოფიერი გამოდგა და იგი აქტიურად განვითარდა. აღმოჩნდა, რომ უმჯობესია ამოცანა დაისვას მოქმედი ობიექტების ერთობლიობის სახით - ასე წარმოიშვა ობიექტზე ორიენტირებული ანალიზი (OOA, object-oriented Analysis). შესაძლებელი გახდა რთული სისტემების დაპროექტება ობიექტების სახით - წარმოიშვა ობიექტზე ორიენტირებული დაპროექტება (OOD, object-oriented design).

ობიექტზე ორიენტირებული დაპროგრამება უკვე 20 წელია ვითარდება. არსებობს რამდენიმე სკოლა, რომელიც თავისებურად აყალიბებს ძირითად პრინციპებს და ობიექტებთან მუშაობის თავისი პრინციპების ერთობლიობას გვთავაზობს.

ობიექტზე ორიენტირებული დაპროგრამების მნიშვნელოვან ელემენტს წარმოადგენს **აბსტრაქცია.** რაიმე ობიექტის, მაგალითად, ავტომობილის ქცევის აღწერისას, ჩვენ ვაგებთ მის მოდელს. როგორც წესი, მოდელი ობიექტს სრულად ვერ აღწერს, რეალური ობიექტები ძალიან რთულია. აირჩევა ობიექტის მხოლოდ ის თვისებები, რომლებიც მნიშვნელოვანია დასმული ამოცანის გადასაწყვეტად. მაგალითად, ტვირთის გადაზიდვის ამოცანის აღწერისათვის

მნიშვნელოვანია მანქანის ტვირთამწეობა, ხოლო საავტომობილო რბოლების აღწერისას ეს პარამეტრი მნიშვნელოვანი არაა. რბოლების მოდელირებისას უნდა აღიწეროს ავტომობილის მიერ სიჩქარის აკრეფის მეთოდი, რაც სხვადსხვა ტვირთის გადაზიდვისას არაა მნიშვნელოვანი.

ჩვენ უნდა მოვახდინოთ აბსტრაგირება ობიექტის ზოგიერთი კონკრეტული დეტალისაგან. ძალიან მნიშვნელოვანია აბსტრაქციის სწორი დონის შერჩევა. აბსტრაქციის ძალიან მაღალი დონე მოგვცემს ობიექტის ზოგად, ზერელე აღწერას, რაც არ მოგვცემს საშუალებას სწორად მოვახდინოთ ობიექტის ქცევის მოდელირება. აბსტრაქციის ძალიან დაბალი დონის შემთხვევაში მოდელი გამოვა ძალიან რთული, გადაიტვირთება დეტალებით და ამიტომ ის შეიძლება გამოუსადეგარი გახდეს.

აბსტრაქციის გამოყენების მძლავრ იარაღს წარმოადგენს იერარქიული კლასიფიკაციები. ის საშუალებას იძლევა გავამარტივოთ რთული სისტემების სემანტიკა (შინაარსი), მათი ფრაგმენტებად დაყოფის გზით, რომლებიც სამართავადაც უფრო ადვილი იქნება. მაგალითად, გარეგნულად ავტომობილი ერთ ობიექტად აღიქმება. მაგრამ თუ შიგ ჩავიხედავთ, ვნახავთ, რომ იგი შედგება რამდენიმე ქვესისტემისაგან: საჭის მართვის სისტემა, მუხრუჭები, აუდიო სისტემა, გამათბობელი და სხვა, თითოეული ეს ქვესისტემა აგებულია უფრო სპეციფიური კვანძებისაგან. მაგალითად, აუდიო სისტემა შედგება რადიომიმღებისაგან, კომპაქტ დისკებისა და/ან აუდიო კასეტების დამკვრელისაგან. ამ მაგალითის არსი

მდგომარეობს იმაში, რომ ავტომობილის რთული სისტემა (ან ნებისმიერი სხვა რთული სისტემა) შეიძლება აღვწეროთ იერარქიული აბსტრაქციის გზით.

რთული სისტემების იერარქიული აბსტრაქცია შეიძლება გამოვიყენოთ კომპიუტერული პროგრამების მიმართაც. პროცესზე ორიენტირებული პროგრამების მონაცემები აბსტრაქციის გზით შეიძლება გარდაიქმნას შემადგენელ ობიექტებად. ამასთან პროცესის მიმდევრობითი ბიჯები შეიძლება გარდაიქმნას შეტყობინებების კოლექციად, რომლებიც ამ ობიექტებს შორის გადაიცემა. ამგვარად, თითოეული ეს ობიექტი აღწერს თავის უნიკალურ მოქმედებას. ობიექტები შეიძლება ჩავთვალოთ კონკრეტულ ელემენტებად, რომლებიც პასუხობენ შეტყობინებებზე. შეტყობინებები კი მიუთითებენ, რაღაც ქმედების შესრულების აუცილებლობაზე. ფაქტიურად ესაა ობიექტზე ორიენტირებული პროგრამირების არსი.

ხშირად აგებენ დეტალიზაციის სხვადასხვა დონის რამდენიმე მოდელს. მაგალითად, ზოგადად ავტომობილის მოდელის აღწერისას არაა საჭირო ყველა იმ პარამეტრის გათვალისწინება, რომლებიც აუცილებელია სატვირთო ავტომობილის ან სპორტული რბოლების ავტომობილის უფრო კონკრეტული და ზუსტი მოდელის აგებისას. ამასთან, უფრო ზუსტი მოდელი, რომელიც აბსტრაქციის უფრო დაბალი დონით ხასიათდება, გამოიყენებს ნაკლებად ზუსტი მოდელის უკვე არსებულ მეთოდებს.

აქ შეიძლება დაისვას სხვადასხვა კითხვა: მსოფლიოში არსებობს ათასობით სხვადასხვა მარკისა და ტიპის ავტომობილი,

რა არის მათ შორის საერთო? ხომ არ ჯობია ყველა განსხვავებული ტიპის ავტომობილისათვის ავაგოთ განსხვავებული კლასი? როგორ მოვახდინოთ ყველა ამ კლასის ორგანიზება? ამ კითხვებზე ობიექტზე ორიენტირებული დაპროგრამება პასუხობს ასე: უნდა მოვახდინოთ კლასების იერარქიის ორგანიზება.

ობიექტზე ორიენტირებული დაპროგრამების სამი ძირითადი პრინციპი

ობიექტზე ორიენტირებული დაპროგრამების ყველა ენას აქვს მექანიზმები, რომლებიც აადვილებს ობიექტზე ორიენტირებული მოდელების აგებას. ამ მექანიზმებს წარმოადგენს **ინკაფსულაცია, მემკვიდრეობითობა** და **პოლიმორფიზმი**. განვიხილოთ ეს კონცეფციები.

ინკაფსულაცია

ინკაფსულაცია - ესაა მექანიზმი, რომელიც პროგრამის კოდს აკავშირებს იმ მონაცემებთან, რომლებთანაც ის მუშაობს და ორივე ამ კომპონენტს იცავს გარე ჩარევებისაგან და უნებართვო მიმართვებისგან. ინკაფსულაცია შეიძლება წარმოვიდგინოთ, როგორც დამცავი გარსი, რომელიც კოდსა და მონაცემებს იცავს სხვა, ამ გარსის გარეთ მყოფი კოდის მიერ თავისუფალი წვდომისაგან (მიმართვისაგან). გარსის შიგა კოდსა და მონაცემებზე წვდომა (მიმართვა) მკაცრად კონტროლდება წინასწარ განსაზღვრული ინტერფეისით. რეალურ სამყაროსთან ანალოგიის მიზნით, განვიხილოთ ავტომობილის სიჩქარის გადაცემათა კოლოფი. მასში

ინკაფსულირებულია სხვადასხვა ინფორმაცია ავტომობილის შესახებ, მაგალითად, აჩქარება, სავალი გზის ზედაპირის დახრილობა, სიჩქარის კოლოფის რეჟიმის გადამრთველის მდგომარეობა. მომხმარებელს (მძღოლს) ამ რთულ ინკაფსულაციაზე ზემოქმედება შეუძლია მხოლოდ ერთი გზით: სიჩქარეთა გადამრთველის სხვადასხვა პოზიციაში გადატანით. სიჩქარეთა კოლოფზე შეუძლებელია ვიმოქმედოთ მაგალითად, მოხვევის ნათურის გადამრთველით ან შუშების გამწმენდის ბერკეტით. ამრიგად, სიჩქარის გადაცემათა კოლოფზე მოქმედი ერთადერთი ინტერფეისია სიჩქარეთა გადამრთველი. გადაცემათა კოლოფის შიგნით მიმავალი პროცესები ზეგავლენას არ ახდენენ მის გარეთ მყოფ ობიექტებზე (მაგალითად, არ ანათებს ფარებს!?). ვინაიდან სიჩქარეთა გადართვის ფუნქცია ინკაფსულირებულია გადაცემათა კოლოფში, ავტომობილების ათობით სხვადასხვა გამომშვებმა ფირმამ კოლოფის რეალიზაცია შეიძლება მოახდინოს თავისი მეთოდით. მძღოლის კუთხით ყველა გადაცემათა კოლოფი ერთნაირად მუშაობს.

ანალოგიური მსჯელობა შეიძლება გამოვიყენოთ პროგრამირებაშიც. ინკაფსულირებული კოდის სიმძლავრე იმაში მდგომარეობს, რომ ყველამ იცის როგორ მიმართოს მას და როგორ გამოიყენოს იგი რეალიზაციის ნიუანსების ცოდნის გარეშე.

Java-ში ინკაფსულაციის განხორციელების საფუძველს წარმოადგენს კლასი. თითოეული მოდელის აღწერა ხდება ერთი ან რამდენიმე **კლასის** (classes) საშუალებით. კლასი შეიძლება ჩავთვალოთ პროექტად, მონახაზად, შაბლონად, რომლის

მიხედვითაც შემდგომში შეიქმნება კონკრეტული ობიექტები. კლასი შეიცავს ობიექტის დამახასიათებელ ცვლადებისა და კონსტანტების აღწერას. მათ უწოდებენ **კლასის ველებს** ან თვისებებს (class fields). ობიექტის მოქმედებების აღმწერ პროცედურებს უწოდებენ **კლასის მეთოდებს** (class methods). კლასის მეთოდები ან მოკლედ მეთოდები - ესაა პროგრამის კოდი, რომელიც ახდენს მონაცემების დამუშავებას. ე. ი. კლასის **წევრები** შეიძლება იყოს ველები (თვისებები, ცვლადები) და მეთოდები (პროცედურები).

სწორად დაწერილ Java პროგრამებში მეთოდები განსაზღვრავს კლასის წევრი-ცვლადების გამოყენების ხერხს. ანუ კლასის ინტერფეისი და ურთიერთობა განისაზღვრება მეთოდებით, რომლებიც ახდენს მოქმედებებს კლასის ეგზემპლარების მონაცემებზე.

კლასის შიგნით შეიძლება აღიწეროს **ჩადგმული კლასები** (nested classes) და **ჩადგმული ინტერფეისები**. ველები, მეთოდები, ჩადგმული კლასები წარმოადგენს **კლასის წევრებს** (class members). სხვადასხვა სკოლა (ალგორითმული ენები) სხვადასხვა ტერმინი იყენებს,. ჩვენ გამოვიყენებთ ალგორითმულ ენა Java-ში მიღებულ ტერმინებს.

მაგალითად, განვიხილოთ ავტომობილის აღწერის მონახაზი:

```
class Automobile {
    int maxVelocity;   // ველი, ავტომობილის მაქსიმალური სიჩქარე
    int speed;         // ველი, ავტომობილის მიმდინარე სიჩქარე
    int weight;        // ველი, ავტომობილის მასა
          //  .....    სხვა ველები
```

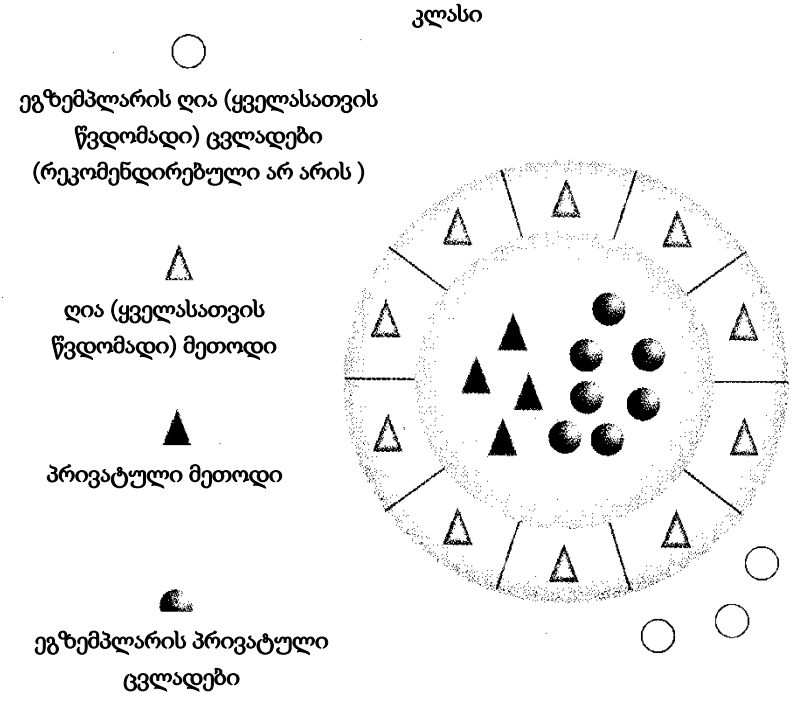
```
    void moveTo(int x, int y) {  // მეთოდი, ავტომობილის
                                 // გადაადგილების მოდელირება
                                 // x და y პარამეტრები არაა ველები
        int a = 1;   // ლოკალური ცვლადი, არაა ველი
            // .....  მეთოდის ტანი. აქ უნდა იყოს
            //          ავტომობილის (x,y) წერტილში
            //   გადაადგილების კანონის აღწერა
    }
        // სხვა მეთოდები
}
```

Pascal-ის მცოდნეებისათვის. Java-ში არ არსებობს ჩადგმული (შიგა) პროცედურები და ფუნქციები, მეთოდის ტანში არ შეიძლება სხვა მეთოდის აღწერა.

კლასის აღწერის შემდეგ შესაძლებელია ამ კლასის კონკრეტული ობიექტების, **ეგზემპლარების** (instances) შექმნა.

როგორც ზემოთ აღვნიშნეთ, კლასის დანიშნულებაა პროგრამის რთული სტრუქტურის ინკაფსულაცია, ამიტომ არსებობს მექანიზმი კლასის შიგა სტრუქტურის რეალიზაციის დასაფარად. კლასის შიგნით თითოეული მეთოდი ან ცვლადი შეიძლება მონიშნული იყოს, როგორც პრივატული ან ყველასათვის წვდომადი (ღია). კლასის წევრებთან ღია, ყველასათვის წვდომადი ინტერფეისი კლასის გარე მომხმარებლებს უფლებას აძლევს მიმართონ ამ წევრებს. კლასის პრივატული მეთოდები და ცვლადები წვდომადია მხოლოდ ამავე კლასის კოდისათვის. შესაბამისად, ნებისმიერი კოდი, რომელიც არაა იმავე კლასის წევრი ვერ მიიღებს უფლებას დაუკავშირდეს კლასის პრივატულ მეთოდს ან ცვლადს. ვინაიდან კლასის

პრივატულ წევრებთან მიმართვა შესაძლებელია მხოლოდ კლასის ღია მეთოდების საშუალებით, ამიტომ დარწმუნებული შეიძლება ვიყოთ, რომ შეუძლებელია პრივატულ წევრებზე უნებართვო მიმართვები. ცხადია, ღია ინტერფეისი გააზრებულად უნდა იყოს დაპროექტებული, ვინაიდან ზომაზე მეტად არ გახსნას კლასის მუშაობის შიგა ნიუანსები (სურ. 1.).



სურ.1. ინკაფსულაცია: ყველასათვის წვდომადი მეთოდები შეიძლება გამოყენებული იქნეს პრივატული მონაცემების დასაცავად

მემკვიდრეობითობა

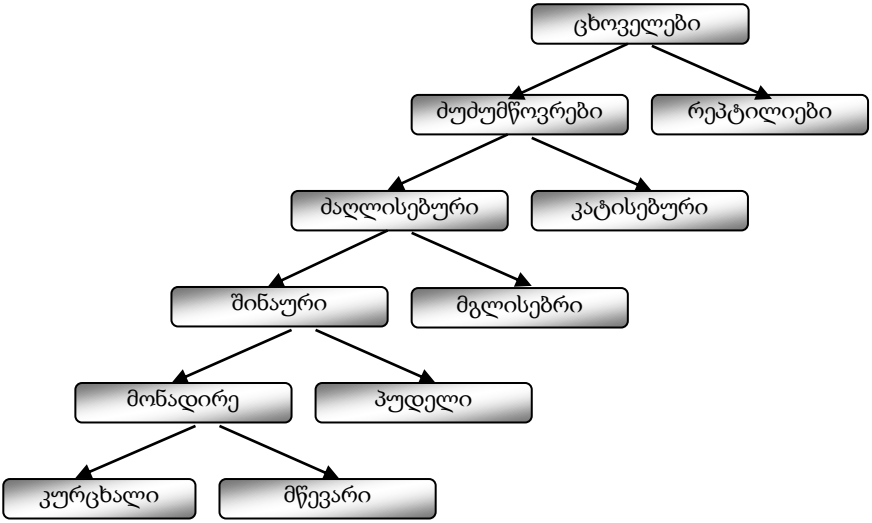
მემკვიდრეობითობა - ესაა პროცესი, რომლის საშუალებითაც ერთი ობიექტი იღებს მეორე ობიექტის თვისებებს. ეს განსაკუთრებით მნიშვნელოვანია იერარქიული კლასიფიკაციის კონცეფციის მხარდაჭერისათვის. იერარქიის გამოყენების გარეშე თითოეული ობიექტისათვის ცხადად უნდა აღიწეროს ყველა მისი თვისება. მემკვიდრეობითობის წყალობით ობიექტისათვის 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 ურთიერთ დაკავშირებული 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 წარმოადგენს ცხოველთა **კლასის** აღწერას.

თუ საჭიროა ცხოველთა უფრო კონკრეტული კლასის აღწერა, მაგალითად, ძუძუმწოვრების, საჭიროა უფრო კონკრეტული ატრიბუტების მითითება, მაგალითად,, კბილების ტიპი, სარძევე ჯირკვლები და სხვა. ამ აღწერას ვუწოდოთ ცხოველების კლასის ქვეკლასი. თავის მხრივ

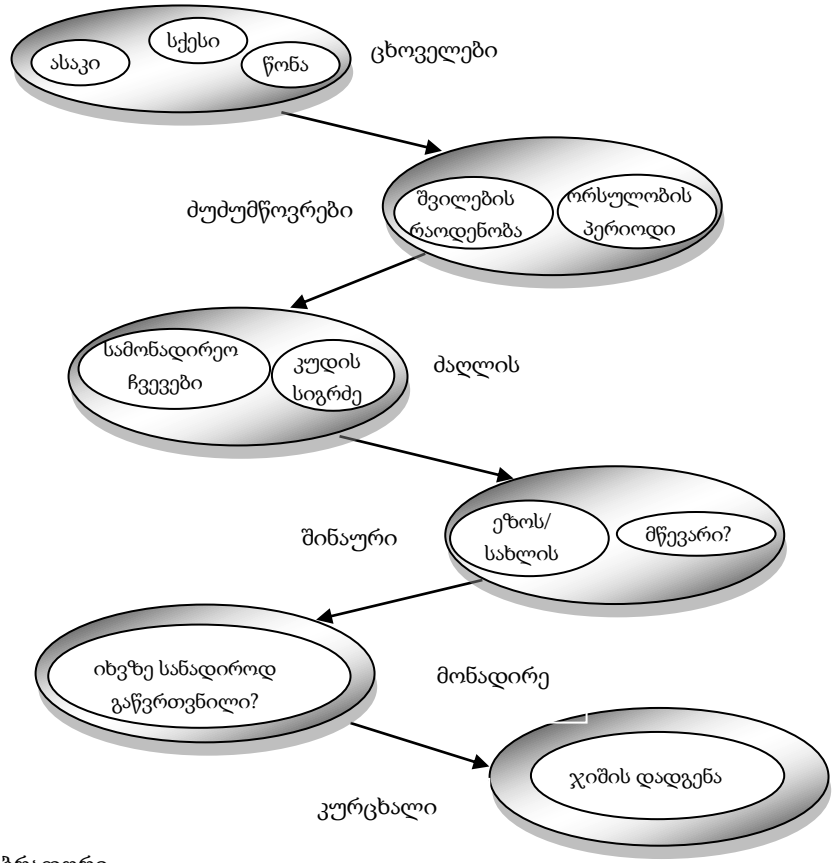
ძუძუმწოვრების კლასისათვის ცხოველების კლასი არის სუპერ კლასი (მშობელი კლასი).

ვინაიდან ძუძუმწოვრები ესაა უფრო დაზუსტებულად განმარტებული ცხოველები, მათ აქვთ მემკვიდრეობით გადმოცემული ცხოველების ყველა ატრიბუტი. კლასების იერარქიის ქვედა დონის ქვეკლასი მემკვიდრეობით იღებს ყოველი თავისი მშობელი კლასის ატრიბუტებს (სურ.2).



სურ. 2. კლასების იერარქიის სქემა

მემკვიდრეობითობა ინკაფსულაციასთანაცაა კავშირში. თუ მოცემულ კლასს ინკაფსულირებული აქვს გარკვეული ატრიბუტები, მის ყოველ ქვეკლასსაც ექნება ეს ატრიბუტები და პლუს დამატებითი ატრიბუტები, რომლებიც წარმოადგენენ მისი სპეციალიზაციის შემადგენელ ნაწილს (სურ.3).



ლაბრადორი

ასაკი	ორსულობის პერიოდი	მწევარი?
სქესი	სამონადირეო ჩვევები	იხვზე სანადიროდ გაწვრთნილი?
წონა	კუდის სიგრძე	ჯიშის დადგენა
შვილების რაოდენობა	ეზოს/სახლის	

სურ.3. კურცხალს მემკვიდრეობით სრულად გადმოეცა ყველა მშობელი კლასის ინკაფსულირებული თვისებები

პოლიმორფიზმი

პოლიმორფიზმი (ბერძნული სიტყვაა და ნიშნავს „მრავალ ფორმას“) - ესაა თვისება, რომელიც საშუალებას იძლევა ერთი და იგივე ინტერფეისი გამოყენებული 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 თითო-ეული სხვადასხვა ტიპის მონაცემებს ინახავს. არაობიექტ-ორიენტირებული ენისათვის საჭირო იქნებოდა სამი სხვა-დასხვა ქვეპროგრამის დაწერა, თითოეულ მათგანს უნდა ჰქონოდა განსხვავებული სახელი. Java-ში, პოლიმორფიზმის წყალობით შესაძლებელია სტეკის ქვეპროგრამები ისე შევქმნათ, რომ ყველას ერთი და იგივე 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 ერთი და იგივე ინტერფეისი გამოიყენება საერთო მოქმედებების კლასის

აღნიშვნისათვის. კონკრეტული მოქმედების (ანუ მეთოდის) არჩევა კი ხდება, თითოეული სიტუაციის მიხედვით და ამას აკეთებს კომპილატორი. პროგრამისტისათვის არაა აუცილებელი ამ არჩევანის გაკეთება ხელით. საკმარისია მხოლოდ იცოდეს საერთო ინტერფეისი და და გამოიყენოს იგი.

თუ ძაღლებზე ანალოგიას განვაგრძობთ, 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 მოქმედებს ერთი და იგივე შეგრძნება - ყნოსვა. განსხვავება ისაა, რა გამოსცემს ამ სუნს - ანუ მონაცემთა ტიპია განსხვავებული. ასეთი ზოგადი კონცეფცია შესაძლებელია გამოყენებული იქნეს Java-ში პროგრამის შიგა მეთოდების მიმართაც.

ინკაფსულაციის, პოლიმორფიზმისა და მემკვიდრეობითობის ერთობლივად სწორად გამოყენების შემთხვევაში ისინი ქმნიან პროგრამირების გარემოს, რომელიც უზრუნველყოფს უფრო მდგრადი და მასშტაბირებადი პროგრამების დამუშავებას, ვიდრე პროცესზე ორიენტირებული მოდელის გამოყენება. კლასების იერარქიის კარგად გააზრებული დაპროექტება, ერთი და იგივე კოდის მრავალჯერადი გამოყენების საფუძველია, რომლის დამუშავებაზე და ტესტირებაზე დიდი შრომისა და დროის დახარჯვაა საჭირო. ინკაფსულაცია საშუალებას იძლევა მივუბრუნდეთ ადრე შექმნილ რეალიზაციებს, მოვახდინოთ იქ მოდიფიკაციები, მაგრამ ისე

რომ არ დაირღვეს მასთან წვდომის ინტერფეისები. პოლიმორფიზმი საშუალებას იძლევა შეიქმნას, გასაგები, კითხვადი და საიმედო კოდი.

ამრიგად, **ობიექტზე ორიენტირებული პროგრამირება წარმოადგენს პროგრამირების მეთოდოლოგიას, რომელიც ემყარება პროგრამის წარმოდგენას ობიექტების ერთობლიობის სახით, სადაც თითოეული ობიექტი არის განსაზღვრული ტიპის რეალიზაცია, რომელიც იყენებს შეტყობინებების (გზავნილების) გადაგზავნის მექანიზმს და მემკვიდრეობის იერარქიულად ორგანიზებულ კლასებს.**

ობიექტზე ორიენტირებული პროგრამირების ძირითადი ელემენტია აბსტრაქცია. მონაცემები აბსტრაქციის საშუალებით გარდაიქმნება ობიექტებად, მონაცემების დამუშავების მიმდევრობა კი ხდება ამ ობიექტებს შორის გადაცემული შეტყობინებების ერთობლიობა. თითოეული ობიექტი უნიკალურია. ობიექტებთან ურთიერთობა ხდება როგორც კონკრეტულ არსებებთან, რომლებიც რეაგირებენ რაიმე ბრძანებების შესრულების შეტყობინებებზე.

ობიექტზე ორიენტირებული პროგრამირება ხასიათდება შემდეგი პრინციპებით (ალან კეის მიხედვით):

- ყველაფერი ობიექტის სახითაა წარმოდგენილი;
- გამოთვლები ხორციელდება ობიექტებს შორის ურთიერთქმედებით (მონაცემების გაცვლით), ამ დროს ერთი ობიექტი ითხოვს, რომ მეორე ობიექტმა შეასრულოს გარკვეული მოქმედებები; ობიექტები ურთიერთქმედებენ ერთმანეთში შეტყობინებების გადაცემით და მიღებით;

შეტყობინებები ესაა მოთხოვნა რაიმე მოქმედების შესასრულებლად, რომელსაც დამატებით შეიძლება ჰქონდეს არგუმენტები ამ მოქმედებების სამართავად;

- თითოეულ ობიექტს აქვს დამოუკიდებელი მეხსიერება, რომელიც სხვა ობიექტებისაგან შედგება;
- ყოველი ობიექტი არის ამ ტიპის ობიექტების საერთო თვისებების გამომხატავი კლასის წარმომადგენელი;
- კლასში მოცემულია ობიექტის ქცევა (ფუნქცია), ამიტომ კლასის ყველა ეგზემპლარს (ობიექტს) შეუძლია ერთი და იგივე მოქმედებების შესრულება;
- კლასები ორგანიზებულია საერთო ძირიანი ხისებრ იერარქიის სახით, რომელსაც **მემკვიდრეობის იერარქია** ჰქვია; კლასის ეგზემპლარის მეხსიერება და ქცევა ავტომატურად წვდომადია ყველა იმ კლასისათვის, რომელიც იერარქიულ ხეზე მასზე დაბლაა განლაგებული.

თავი I. ენის ლექსიკა, ბაზისური ტიპები და ოპერაციები

ახალი ენის შესასწავლად განვიხილოთ, რა ტიპის საწყისი მონაცემების დამუშავება შეიძლება ამ ენის საშუალებით, რა სახით შეიძლება მათი აღწერა და რა სტანდარტული საშუალებებია ჩადებული ენაში. ალგორითმულ ენა Java-ში მრავალი მონაცემთა ტიპი და კიდევ უფრო მეტი მათი გამოყენების წესი არსებობს. ამ წესების დაუცველობა იწვევს დაფარულ შეცდომებს, რომლების აღმოჩენაც საკმაოდ ძნელი და შრომატევადი სამუშაოა.

ტრადიციის მიხედვით, მრავალი სახელმძღვანელო იწყება უმარტივესი პროგრამით “Hello World!”. ლისტინგ 1-ზე ნაჩვენებია Java-ზე დაწერილი ეს პროგრამა

ლისტინგი 1. Java-ზე დაწერილი პირველი პროგრამა

```
class HelloWorld  {
    public static void main(String[] args)  {
        System.out.println("Hello World!");
    }
}
```

ამ მარტივ მაგალითზეც შესაძლებელია შევამჩნიოთ ენის მთელი რიგი თავისებურებები:

- ყოველი პროგრამა წარმოადგენს ერთ ან რამდენიმე **კლასს,** ჩვენს მაგალითში მხოლოდ ერთი კლასია (class).
- კლასის დასაწყისს აღნიშნავს დარეზერვებული სიტყვა class, რომლის შემდეგაც მოდის პროგრამისტის მიერ შერჩეული კლასის კონკრეტული სახელი, ჩვენს შემთხვე-

ვაში HelloWorld. ყველაფერი რაც კლასს ეკუთვნის ჩაიწერება ფიგურულ ფრჩხილებში და წარმოადგენს კლასის ტანს (class body).

- ყოველი მოქმედება (გამოთვლები) ხდება ინფორმაციის დამუშავების მეთოდების საშუალებით, მოკლედ ამბობენ მეთოდებით (method). Java-ში მიღებულია ეს ტერმინი, სხვა ენებში გამოყენებულია „ფუნქციის“ მაგივრად.
- 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, ფუნქცია, ჩვენს შემთხვევაში მეთოდი, შედეგად იძლევა (ხშირად ვამბობთ აბრუნებს - returns) ერთ მნიშვნელობას, რომლის ტიპი მეთოდის სახელის წინ მიეთითება. მეთოდმა შეიძლება არაფერიც არ დააბრუნოს, როდესაც იგი პროცედურის როლს ასრულებს, როგორც ეს ჩვენს შემთხვევაშია. ასეთ შემთხვევაში დაბრუნებული მნიშვნელობის ტიპის მაგივრად ჩაიწერება სიტყვა void.
- მეთოდის სახელის შემდეგ ფრჩხილებში ჩაიწერება არგუმენტების ან პარამეტრების ჩამონათვალი, რომლებიც ერთმანეთისაგან მძიმეებით გამოიყოფა. ყოველი არგუმენტისათვის მიეთითება 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 სახელად

ნებისმიერი სახელი შეიძლება იყოს, მაგალითში არჩეულია args.

- მეთოდის დაბრუნებული ტიპის წინ შეიძლება ჩაწერილი 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 აუცილებელია. მეთოდის სახელის შემდეგ ყოველთვის ვწერთ ფრჩხილებს, ამით ხაზს ვუსვამთ, რომ ეს მეთოდის სახელია და არა ჩვეულებრივი ცვლადის სახელი.
- მეთოდის მთელი შემცველობა წარმოადგენს მეთოდის **ტანს** (method body) და ჩაიწერება 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 ჩაიწერება ორმაგ ბრჭყალებში, რომელიც 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 არაა მნიშვნელოვანი პატარა ასოები იქნება, თუ დიდი, განსხვავება მხოლოდ ეკრანზე გამოჩნდება.

ალგორითმულ ენა Java-ში პროგრამისტის მიერ შერჩეული სახელები შეიძლება ჩაიწეროს მისი შეხედულებისამებრ, კლასისთვის შეიძლება დაგვერქმია helloworld ან helloWorld, მაგრამ java-პროგრამისტებს შორის მოქმედებს შეთანხმება, რომელსაც უწოდებენ „Code Conventions for the Java Programming Manguage“, რომელსაც შეიძლება გაეცნოთ java.sun.com/docs/codeconv/index.html მისამართზე. ამ შეთანხმების ზოგიერთი პუნქტი ასეთია:

- კლასის სახელები იწყება დიდი ასოებით; თუ სახელი შედგება რამდენიმე სიტყვისაგან, ყოველი შემდეგი სიტყვა უნდა იწყებოდეს დიდი ასოთი;
- ცვლადებისა და მეთოდების სახელები უნდა იწყებოდეს პატარა ასოთი; თუ სახელი შედგება რამდენიმე სიტყვისაგან, ყოველი შემდეგი სიტყვა უნდა იწყებოდეს დიდი ასოთი;
- კონსტანტების სახელები იწერება მხოლოდ დიდი ასოებით; თუ სახელი შედგება რამდენიმე სიტყვისაგან, ყოველ სიტყვას შორის უნდა ჩაიწეროს ხაზის გასმის სიმბოლო;

ეს წესები არაა აუცილებელი, მაგრამ მათი დაცვა მნიშვნელოვნად აადვილებს პროგრამის კოდის კითხვადობას და მას Java-ს სტილს აძლევს.

სტილს განსაზღვრავს არა მარტო სახელები, არამედ პროგრამის ტექსტის სტრიქონებად განლაგებაც. მაგალითად, ფიგურული ფრჩხილების განლაგება: გამხსნელი ფიგურული ფრჩხილი კლასის, მეთოდის სათაურის სტრიქონის ბოლოში ჩავწეროთ თუ ის მის შემდეგ სტრიქონზე გადავიტანოთ? ეს თითქოს უმნიშვნელო საკითხი ხშირად დავას იწვევს ხოლმე და სხვადასხვა რედაქტორი, როგორც წესი, საშუალებას იძლევა ფიგურული ფრჩხილების დასმის სასურველი სტილი ავირჩიოთ.

კომპილატორისათვის პროგრამირების სტილს მნიშვნელობა არა 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.

ლექსიკის საკითხები

პატარა Java-პროგრამის გაცნობის შემდეგ შესაძლებელია უფრო ფორმალურად აღვწეროთ ენის ძირითადი ელემენტები. Java პროგრამა წარმოადგენს ცარიელი სიმბოლოების,

იდენტიფიკატორების, კონსტანტების, კომენტარების, ოპერაციების, გამყოფებისა და საკვანძო სიტყვების კოლექციას.

ცარიელი სიმბოლო

Java თავისუფალი ფორმატის ენაა. ეს ნიშნავს, რომ პროგრამის დაწერისას არაა საჭირო რაიმე სპეციალური წესების დაცვა აბზაცების მიმართ. ერთადერთი აუცილებელი მოთხოვნაა, რომ ყოველ ლექსემას შორის, რომელიც არაა ოპერაციის სიმბოლოთი ან გამყოფით გამოყოფილი, ჩასმული. ერთი ცარიელი სიმბოლო მაინც იყოს Java-ში ცარიელ სიმბოლოდ ითვლება ჰარი (space), ტაბულაცია ან ახალი სტრიქონის სიმბოლო.

იდენტიფიკატორები

იდენტიფიკატორი გამოიყენება კლასების, მეთოდების და ცვლადების დასახელებისათვის. იდენტიფიკატორი შეიძლება იყოს დიდი და პატარა ასოების, ციფრების, ხაზგასმისა და დოლარის სიმბოლოს ნებისმიერი მიმდევრობა. იდენტიფიკატორი არ უნდა იწყებოდეს ციფრით, ვინაიდან კომპილატორს არ აერიოს რიცხვით კონსტანტებში. ისევ გავიმეოროთ, რომ ჯავა 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 „დავაკომენტიროთ“.

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 ვარსკვლავისაგან /** და მთავრდება */ -ით. შეიძლება დაიკავოს რამდენიმე სტრიქონი, იგი მუშავდება სპეცი-

ალური პროგრამა javadoc-ის მიერ. კომენტარების ეს ტიპი გამოიყენება თვით დოკუმენტირებისთვის.

გამყოფები

Java-ში რამდენიმე სიმბოლო შესაძლებელია გამყოფად იქნეს გამოყენებული. ცხრილი 1-ში ჩამოთვლილია გამყოფი სიმბოლოები.

ცხრილი 1. დასაშვები გამყოფი სიმბოლოები

სიმბოლო	დასახელება	დანიშნულება
()	მრგვალი ფრჩხილები	მეთოდების გამოძახებისა და აღწერის დროს გამოიყენება პარამეტრების სიის გადასაცემად. აგრეთვე გამოიყენება გამოსახულებებში პრიორიტეტის განსაზღვრისათვის, მმართველ ოპერატორებში გამოსახულების და ტიპების გარდაქმნის მისათითებლად.
{}	ფიგურული ფრჩხილები	გამოიყენება მასივების ინიციალიზაციისას მნიშვნელობების განსასაზღვრავად. მათ იყენებენ ასევე კოდების, კლასების, მეთოდების და ლოკალური განსაზღვრებების ბლოკების მისათითებლად.
[]	კვადრატული ფრჩხილები	გამოიყენება მასივების ტიპების გამოსაცხადებლად, ასევე მასივების ელემენტების შემოსატანად.
;	წერტილმძიმე	აღნიშნავს ოპერატორის დასრულებას
,	მძიმე	გამოიყენება ცვლადების გამოცხადების დროს ჩამონათვალში იდენტიფიკატორების სახელების გამოსაყოფად. ეს სიმბოლო გამყოფი ასევე შესაძლებელია გამოყენებულ იქნეს ოპერატორების ჯაჭვის შესაქმნელად FOR ოპერატორში
.	წერტილი	გამოიყენება პაკეტის სახელის გამოსაყოფად ქვეპაკეტებისა და კლასებისაგან, აგრეთვე

		ცვლადებისა და მეთოდების გამოსაყოფად მიმთითებელი ცვლადისაგან

საკვანძო სიტყვები

დღეისათვის Java-ში განსაზღვრულია 50 საკვანძო სიტყვა (ცხრილი 2.) ეს სიტყვები არ შეიძლება გამოყენებული იქნეს ცვლადების, კლასებისა და მეთოდების სახელებად.

ცხრილი 2. საკვანძო სიტყვები

abstract	continue	for	new	switch
assert	default	goto	packge	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	fimally	long	strictfp	volatile
const	float	native	super	while

const და goto საკვანძო სიტყვები ასევე დარეზერვირებულია, მაგრამ არ გამოიყენება.

საკვანძო სიტყვების გარდა Java-ში ასევე დარეზერვირებულია სიტყვები true, false და null. ისინი წარმოადგენენ მნიშვნელობებს და ისინიც არ გამოიყენება სახელებად.

ბაზისური ტიპები Java-ში

როგორც ყველა სხვა თანამედროვე ალგორითმული ენაში, Java-შიც არსებობს მონაცემთა სხვადასხვა ტიპი. ეს ტიპები

გამოიყენება ცვლადების აღწერისათვის, მასივების გამოცხადებისათვის და ა. შ.

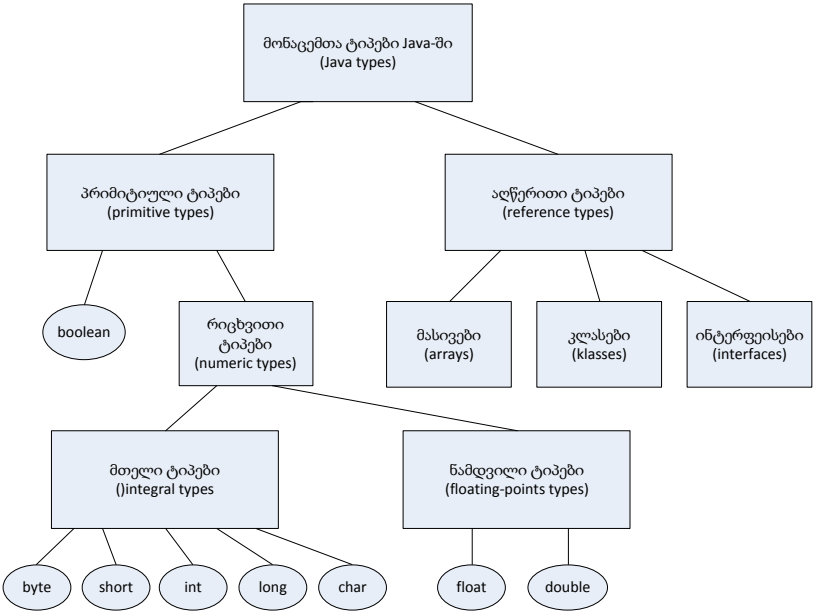
განსაკუთრებით უნდა აღვნიშნოთ, რომ Java არის მკაცრად ტიპიზებული ენა. ამ გარემოებით მიიღწევა Java პროგრამების განსაკუთრებული საიმედოობა და უსაფრთხოება. მკაცრი ტიპიზაცია განპირობებულია იმით, რომ:

- თითოეულ ცვლადს და თითოეულ გამოსახულებას აქვს გარკვეული ტიპი, რომელიც მკაცრადაა განსაზღვრული;
- ყოველი მინიჭებისას, როგორც ცხადი, ისე მეთოდების გამოძახების დროს პარამეტრების გადაცემისას, მოწმდება, შეესაბამება თუ არა ტიპები ერთმანეთს.

Java-ში არ არსებობს რაიმე ავტომატური საშუალებები კონფლიქტური ტიპების დაყვანის ან გარდაქმნის, როგორც ეს ბევრ სხვა ენაშია განხორციელებული. Java-ს კომპილატორი ამოწმებს ყველა გამოსახულებასა და პარამეტრს ტიპების შესაბამისობაზე. ნებისმიერი შეუსაბამობა ითვლება შეცდომად, რომელიც უნდა გასწორდეს კლასის კომპილაციის დასრულებამდე.

ხშირად სხვადასხვა ლიტერატურაში ბაზისურ ტიპებს უწოდებენ მარტივს, ელემენტარულს ან პრიმიტიულს. პრაქტიკულად ეს ტერმინები ამ შემთხვევაში სინონიმებია.

საწყისი მონაცემების ყველა ტიპი, რომლის აღწერაც შესაძლებელია Java-ში, იყოფა ორ ჯგუფად: პრიმიტიული (primitive types) და აღწერითი (reference types მიმთითებელი, მისამართი, მაკავშირებელი) ტიპები (სურ. 4).



სურ. 4. მონაცემთა ტიპები Java-ში

აღწერითი ტიპები იყოფა მასივებად (arrays), კლასებად (classes) და ინტერფეისებად (interfaces). ამ ტიპებს ჩვენ მომავალში განვიხილავთ დაწვრილებით.

პრიმიტიული სულ 8 ტიპია. ისინი შეიძლება დაიყოს **რიცხვით** (numeric), **სიმბოლურ** (character) და **ლოგიკურ** (boolean ბულის) ტიპებად.

რიცხვითი ტიპები თავის მხრივ იყოფა მთელ რიცხვა (integer) და ნამდვილ (floating-point) ტიპებად:

- **მთელ რიცხვა.** ამ ჯგუფში შედის byte, short, int, long ტიპები, რომლებიც წარმოადგენენ ნიშნიან მთელ რიცხვებს;

- **ნამდვილი (მცოცავ მძიმიანი) რიცხვები.** ამ ჯგუფში შედის float, double ტიპები, ისინი წარმოადგენენ ნიშნიან წილად რიცხვებს, რომლებიც ათობითი წერტილის შემდეგ თანრიგების გარკვეული სიზუსტით მოიცემა;

სიმბოლოების ტიპია char, რომელიც წარმოადგენს ენაში დაშვებულ ყველა სიმბოლოს, მაგალითად, ასოები, ციფრები, სასვენი ნიშნები, სპეციალური სიმბოლოები და სხვა. ხშირად ამ ტიპს მიაკუთვნებენ მთელ რიცხვა ტიპს და განიხილავენ როგორც უნიშნო მთელ რიცხვს.

ლოგიკური ანუ **ბულის** ტიპია boolean. ესაა სპეციალური ტიპი, რომელიც ჭეშმარიტი (true) ან ყალბი (false) მნიშვნელობის წარმოსადგენად გამოიყენება.

ეს ტიპები შეიძლება გამოყენებული იქნეს იმ სახით, როგორც არის განსაზღვრული ან გამოყენებული პროგრამისტის მიერ საკუთარი, ახალი ტიპების შესაქმნელად. ამრიგად, ამ ტიპების საფუძველზე შეიძლება შეიქმნას ყველა სხვა ტიპი.

ელემენტარული ტიპები აღწერენ რაიმე ერთ მნიშვნელობას და არა რთულ სტრუქტურებს ან ობიექტებს. Java-ში ამ 8 ტიპის გარდა სხვა ტიპები აკმაყოფილებს ობიექტზე ორიენტირებული ენის მოთხოვნებს ანუ რთული ტიპისაა (შეიძლება იყოს მასივი, კლასი, ინტერფეისი). ეს განსაკუთრებულობა გამოწვეულია მაქსიმალური ეფექტურობის მიღწევის სურვილით.

ელემენტარული ტიპები ისეა განსაზღვრული, რომ მათთვის ცხადად და მკაცრადაა შერჩეული დასაშვები მნიშვნელობების არე (დიაპაზონი). Java-ს სხვადასხვა

პლატფორმაზე გადატანადობის გამო, მას მოეთხოვება მონაცემთა ყველა ტიპი მკაცრად იყოს ერთნაირად წარმოდგენილი ყველა მანქანური არქიტექტურისათვის.

განვიხილოთ მონაცემთა თითოეული ტიპი.

მთელ რიცხვა ტიპები (მნიშვნელობები)

როგორც ზემოთ აღვნიშნეთ, Java-ში 4 მთელრიცხვა ტიპია: byte, short, int, long. ამ ენაში არაა განსაზღვრული მთელი უნიშნო რიცხვების ტიპი, როგორც ეს ბევრ სხვა ენაშია დაშვებული.

ცხრილ 3-ში ნაჩვენებია თითოეული ამ ტიპის მიერ დაკავებული მეხსიერების ზომა, დასაშვები მნიშვნელობების დიაპაზონი და სტანდარტული მნიშვნელობა გამოცხადების შემდეგ.

ცხრილი 3. მთელი ტიპის რიცხვების თანრიგების რაოდენობა და დიაპაზონი

ტიპი	ბიტების რაოდენობა	დასაშვები დიაპაზონი	სტანდ. მნიშ.
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

byte

ზომის მიხედვით ყველაზე პატარაა ტიპი byte. ეს ტიპი ხშირად გამოიყენება მონაცემთა ნაკადებთან სამუშაოდ, რომლებიც ქსელიდან ან რომელიმე ფაილიდან მიიღება/გაიცემა. ასევე მოსახერხებელია მათი გამოყენება ისეთ ორობით მონაცემებზე სამუშაოდ, რომლებიც არაა თავსებადი Java-ს არსებულ ტიპებთან.

ამ ტიპის ცვლადის გამოსაცხადებლად გამოიყენება საკვანძო სიტყვა byte. მაგალითად, შემდეგ სტრიქონში გამოცხადებულია ორი byte-ს ტიპის ცვლადი a და b:

byte a, b;

short

short ტიპი ყველაზე იშვიათად გამოიყენება. ამ ტიპის ცვლადების გამოცხადების მაგალითია:

short s;

short h;

int

int ტიპი ყველაზე ხშირად გამოყენებული ტიპია. იგი ხშირად იხმარება ციკლებისა და მასივების ინდექსების აღსანიშნავად. საყურადღებოა, რომ გამოსახულებების გამოთვლისას, სადაც მონაწილეობს byte და short ტიპის მნიშვნელობების მქონე ცვლადები ან კონსტანტები, მათი

ტიპი ამაღლდება int-მდე და ისე მოხდება არითმეტიკული ოპერაციები.

long

long ტიპის გამოყენება ხდება მაშინ, როდესაც int ტიპის დიაპაზონი არაა საკმარისი საჭირო მნიშვნელობის შესანახად. გამოიყენება დიდ მთელ რიცხვებთან სამუშაოდ.

მცოცავ მძიმიანი ტიპები

მცოცავმძიმიანი ტიპის რიცხვებს ასევე უწოდებენ **ნამდვილი ტიპის** რიცხვებს და ისინი გამოიყენება გამოსახულებების გამოთვლისას, როდესაც საჭიროა შედეგების მიღება ათობითი წერტილის შემდეგ გარკვეული სიზუსტით. Java-ში რეალიზებულია IEEE-754 სტანდარტის შესაბამისი ტიპები და მცოცავმძიმიანი ოპერაციები. მათი ზომები, დიაპაზონები, გამოცხადების შემდეგ სტანდარტული მნიშვნელობები და სიზუსტე ნაჩვენებია შემდეგ ცხრილში:

ცხრილი 4. ნამდვილი ტიპის რიცხვების თანრიგების რაოდენობა და დიაპაზონი

ტიპი	ზომა ბიტებში	მიახლოებითი დიაპაზონი	სტანდ. მნიშვ.	სიზუსტე
double	64 (8 B)	±4.9e-324 ÷ 1.8e+308	0.0D	17 ციფრი
float	32 (4 B)	±1.4e-045 ÷ 3.4e+038	0.0F	7-8 ციფრი

აქ აღსანიშნავია, რომ 0-ის შემდეგ უახლოესი პირველი დადებითი double ტიპის რიცხვია 4.9e-324, ხოლო დიაპაზონის ბოლო დადებითი რიცხვია 1.8e+308. ასევე 0-

მდე პირველი უახლოესი უარყოფითი რიცხვია -4.9e-324, ხოლო დიაპაზონის საწყისი რიცხვია -1.8e+308. ანალოგიურად განიხილება float ტიპის დიაპაზონი.

float

float ტიპი განსაზღვრავს **ერთმაგი სიზუსტის** მნიშვნელობებს, რომლებიც შესანახად იკავებენ 32 ბიტს. ამ ტიპის ცვლადები გამოიყენება, მაშინ როდესაც წილადი ნაწილის დიდი სიზუსტე არაა საჭირო. ამ ტიპის ცვლადის გამოცხადების მაგალითია:

float boxHigh, boxWidth;

double

ორმაგი სიზუსტე, რასაც მიანიშნებს დასახელების საკვანძო სიტყვა double (ორმაგი), ითხოვს 64 ბიტის გამოყენებას მნიშვნელობების შესანახად. ბევრ თანამედროვე პროცესორში ამ ტიპის რიცხვებზე მოქმედებები უფრო სწრაფად სრულდება, ვიდრე ერთმაგი სიზუსტისას. ყველა ტრანსცენდენტური მათემატიკური ფუნქციები, როგორიცაა sin(), cos(), sqrt() და სხვა, აბრუნებენ double ტიპის შედეგებს.

სიმბოლური ტიპი

Java-ში სიმბოლოების შესანახად გამოიყენება char ტიპი. უნდა აღვნიშნოთ, რომ ეს ტიპი ალგორითმული ენა C-შიც გამოიყენება, მაგრამ ისინი ექვივალენტური არ არის. C-ში char - ესაა 8 ბიტიანი მთელ რიცხვა ტიპი. Java-ში ეს ასე არაა, აქ სიმბოლოს წარმოდგენისათვის გამოიყენება Unicode.

Unicode სტანდარტი განსაზღვრავს სიმბოლოების საერთაშორისო ნაკრებს, რომელიც მოიცავს ყველა ცნობილი ენების სიმბოლოებს. იგი წარმოადგენს ათობით სხვადასხვა სიმბოლოების უნიფიცირებულ ერთობლიობას, როგორიცაა ქართული, ბერძნული ალფაბეტი, არაბული ალფაბეტი, კირილიცა, ივრითი, იაპონური იეროგლიფები და სხვა. ყველა ამ სიმბოლოს კოდის შესანახად საჭიროა 16 ბიტი. ამიტომ Javaში char ტიპის შესანახად გამოიყენება 16 ბიტი. ამ ტიპის მნიშვნელობების დასაშვები დიაპაზონია 0 ÷ 65536. char ტიპისთვის არ არსებობს უარყოფითი მნიშვნელობები. სიმბოლოების ნაკრების ცნობილ ASCII სტანდარტში მნიშვნელობები იცვლება 0 ÷ 127 დიაპაზონში, ხოლო გაფართოებულ 8 ბიტიან სტანდარტში ISO-Latin-1 მნიშვნელობები 0 ÷ 255 დიაპაზონშია. ვინაიდან Java განკუთვნილია პროგრამების შესაქმნელად მთელ მსოფლიოში, ამიტომ Unicode სტანდარტის გამოყენება ენაში გასაგებია. რა თქმა უნდა Unicode-ს გამოყენება არაა ეფექტური ლათინურენოვანი დამწერლობის მქონე ენებისათვის, რომლებისთვისაც სავსებით საკმარისია 8 ბიტიანი კოდირებები, მაგრამ სამაგიეროდ ეს აუცილებელია მსოფლიო მასშტაბით პროგრამების თავსებადობისა და გადატანადობისათვის.

აღსანიშნავია char ტიპის კიდევ ერთი თავისებურება. ვინაიდან გათვალისწინებულია Unicode სიმბოლოების შესანახად, იგი შეიძლება ჩავთვალოთ მთელ ტიპადაც, რომელზეც არითმეტიკული ოპერაციების ჩატარებაც შესაძლებელია. მაგალითად, შესაძლებელია სიმბოლური ცვლადის მნიშვნელობის გაზრდა ან შემცირება. განვიხილოთ პროგრამის ფრაგმენტი:

ლისტინგი 2. სიმბოლური ცვლადები იქცევიან მთელ რიცხვა ტიპებივით

```
public class CharDemo2 {
  public static void main(String[] args) {
    char chl;
    chl = 'X';
    System.out.println ("chl მნიშვნელობაა " + chl);
    chl++;   // მნიშვნელობის 1-ით გაზრდა
    System.out.println("chl ახალი მნიშვნელობაა " + chl);
  }
```

ამ პროგრამის შედეგად კონსოლზე გამოვა:

chl მნიშვნელობაა X
chl ახალი მნიშვნელობაა Y

პროგრამაში chl ცვლადს თავიდან ენიჭება მნიშვნელობა X. შემდეგ chl ცვლადის მნიშვნელობა იზრდება 1-ით. შედეგად chl ცვლადში ინახება სიმბოლო Y, ვინაიდან მისი კოდი ASCII (და Unicode-შიც) X-ს შემდეგია, ანუ Y-ის კოდი ერთით მეტია X-ის კოდზე.

ლოგიკური (ბულის) ტიპი

Java-ში გამოყენებულია ელემენტარული ტიპი, რომელიც boolean-ით აღინიშნება და მისი მნიშვნელობა შეიძლება იყოს ლოგიკური true (ჭეშმარიტი) ან false (მცდარი). ასეთი ტიპი ბრუნდება ყველა შედარების ოპერაციის შედეგად (მაგალითად, a > b). აუცილებელია boolean ტიპის იყოს ასევე პირობითი და ციკლის ოპერატორებში გამოყენებული პირობითი გამოსახულების შედეგი.

კონსტანტები (ლიტერალები)

კონსტანტები შეიძლება დავყოთ რიცხვით, რომელიც თავის მხრივ იყოფა მთელ რიცხვა და მცოცავმძიმიან (ნამდვილ) კონსტანტებად, სიმბოლურ, ლოგიკურ და სტრიქონულ კონსტანტებად. განვიხილოთ თითოეული მათგანი დეტალურად.

მთელ რიცხვა კონსტანტები

მთელი რიცხვები ტიპურ პროგრამაში ყველაზე უფრო ხშირად გამოყენებადი ტიპია. ნებისმიერი რიცხვითი მნიშვნელობა რიცხვითი კონსტანტაა. მისი მაგალითებია 0, 1, -17, 1024 და სხვა. ყველა ეს კონსტანტა წარმოადგენს ათობით რიცხვებს. რიცხვით კონსტანტებში გამოყენებული შეიძლება იყოს კიდევ სამი სახის წარმოდგენა - ოროობითი (რიცხვები 2-ის ფუძით), რვაობითი (რიცხვები 8-ის ფუძით) და თექვსმეტობითი (რიცხვები თექვსმეტის ფუძით).

Java-ში რვაობითი მნიშვნელობა აღინიშნება საწყისი 0-ით. ჩვეულებრივ, ათობითი რიცხვები არ უნდა იწყებოდეს 0-ით. გარეგნულად ჩვეულებრივი და დასაშვები მნიშვნელობა 09 იწვევს კომპილაციის შეცდომას, ვინაიდან რადგან რიცხვი იწყება 0-ით, კომპილატორი თვლის, რომ უნდა მოდიოდეს რვაობითი რიცხვი, ხოლო რვაობითში ციფრი 9 დაუშვებელია! რვაობითში ციფრები დასაშვებია მხოლოდ 0-7 დიაპაზონში. რვაობითში ჩაწერილი კორექტული კონსტანტებია: 017, -02003, 0777.

პროგრამისტები ხშირად იყენებენ რიცხვების თექვსმეტობით წარმოდგენას, რომლებიც ზუსტად შეესაბამება მანქანური სიტყვებს, რომელთა ზომებია 8, 16, 32 და 64 ბიტი. თექვსმეტობითი მნიშვნელობა აღინიშნება საწყისი 0-ით და სიმბოლო x-ით (0x ან 0X). თექვსმეტობითში სულ დასაშვებია 16 ციფრი, ეს ციფრებია 0,1,...9,A,B,C,D,E,F (ან a, b, c, d, e, f). თექვსმეტობითში ჩაწერილი კორექტული კონსტანტებია: 0x17, 0xff0b, 0X77FF, -0X12cd.

Java 7-ში (JDK 7) და მის შემდეგ ვერსიებში შესაძლებელია ორობითი ლიტერალის ჩაწერა (შექმნა). ორობით ლიტერალს წინ უნდა უძღოდეს სიმბოლოები 0b ან 0B. მაგალითად, 0b110010 წარმოადგენს 110010_2 ორობითი რიცხვის შესაბამის ლიტერალს. ორობითში ჩაწერილი კორექტული კონსტანტებია: 0b101001, -0b001011, 0B010101, -0B1100011.

მთელ რიცხვა კონსტანტები ქმნიან int ტიპის მნიშვნელობას, რომელიც java-ში 32 ბიტიანია. როგორც ზემოთ აღვნიშნეთ ეს ენა მკაცრად ტიპიზებულია, მაშინ შეიძლება დაისვას კითხვა, როგორ უნდა მიენიჭოს მთელრიცხვა კონსტანტა სხვა მთელ ტიპებს, მაგალითად, byte-ს, short-ს ან long-ს ისე, რომ არ მოხდეს ტიპების შეუსაბამობის შეცდომა. საბედნიეროდ, ასეთ სიტუაციებს კომპილატორი ადვილად უმკლავდება. როდესაც კონსტანტის მნიშვნელობა ენიჭება byte-ს ან short-ს თუ მნიშვნელობა იმყოფება მიმღების დასაშვებ დიაპაზონში შეცდომა არ წარმოიქმნება. გარდა ამისა long ტიპის ცვლადს ყოველთვის შეიძლება მიენიჭოს მთელრიცხვა კონსტანტა. მაგრამ თუ საჭიროა კომპილატორს ცხადად მივუთითოთ, რომ კონსტანტა long ტიპისაა,

კონსტანტას უკან უნდა მივუწეროთ l ან L ასო. გასათვალისწინებელია, რომ გრძელი მთელი კონსტანტების ჩაწერისას უნდა მოვერიდოთ l-ის გამოყენებას, ვინაიდან იგი ადვილად შეიძლება აგვერიოს ერთიანში (1 l). კორექტული მაგალითებია: 0x7fffffffffffffffL, 9223372036854775697L, ეს long ტიპის მაქსიმალური ზომის რიცხვებია.

აღსანიშნავია, რომ მთელ რიცხვა მნიშვნელობა შეიძლება მიენიჭოს char ტიპს, თუ რიცხვი იმყოფება ამ ტიპის დასაშვებ დიაპაზონში.

მცოცავმძიმიანი კონსტანტები

მცოცავ მძიმიანი კონსტანტები ჩაიწერება მხოლოდ ათობითი თვლის სისტემაში და ესაა რიცხვები წილადი ნაწილით. ისინი შეიძლება ჩაწერილი იყოს, როგორც სტანდარტულ - ფიქსირებულ ფორმატში, ისე სამეცნიერო - ექსპონენციალურ (მაჩვენებლიან) ფორმატში. ფიქსირებულ ფორმატში ჩაწერილი რიცხვი შედგება მთელი რიცხვისაგან, რომელსაც მოჰყვება ათობითი წერტილი და წილადი ნაწილი. მაგალითია, 2.0, 3.14159, -2.72 წარმოადგენენ სტანდარტულ ფორმატში დასაშვებ, კორექტულად ჩაწერილ რიცხვებს. სამეცნიერო ანუ ექსპონენციალური ფორმა იყენებს რიცხვის სტანდარტულ, მცოცავმძიმიან ფორმას, რომელსაც ემატება სუფიქსი, რომელიც მიუთითებს 10-ის რომელ ხარისხზე უნდა გამრავლდეს რიცხვი. ექსპონენციალური ფუნქციის მისათითებლად გამოიყენება სიმბოლო E ან e, რომლის შემდეგაც მოდის ათობითი მთელი რიცხვი (დადებითი ან უარყოფითი). მაგალითად, 6.022 E12, 314158 E-05, 2e+100.

სტანდარტულად ნამდვილი ტიპის კონსტანტები ინახება double ტიპის სახით. ნამდვილი ტიპის კონსტანტის ბოლოს შეიძლება ჩავწეროთ (მივუთითოთ) ასო F ან f, მაშინ კონსტანტა შენახული იქნება float ტიპის სახით. მაგალითად, 3.5f, -45.67F, 4.7e-5f. პრინციპში, შესაძლებელია ასო D ან d-ს მიწერაც, მაგალითად, -456.789d, 0.0123D, რაც მიუთითებს double ტიპს, მაგრამ იგი ზედმეტია, ვინაიდან ნამდვილი კონსტანტები ისედაც double ფორმატში ინახება.

სიმბოლური კონსტანტები

Java-ში სიმბოლოები წარმოდგენილია Unicode სტანდარტში. ესაა 16 ბიტიანი მნიშვნელობა, რომელიც შეიძლება გადაყვანილი იქნეს მთელ რიცხვებში, რომლებზედაც შესაძლებელია მთელ რიცხვა არითმეტიკული ოპერაციების, მაგალითად, შეკრებისა და გამოკლების ოპერაციების ჩატარება. სიმბოლური კონსტანტები ჩაიწერება ორი ერთმაგი ბრჭყალის (აპოსტროფების) შიგნით. სიმბოლოების ჩასაწერად გამოიყენება შემდეგი ფორმები:

- ASCII სტანდარტის ყველა გრაფიკული გამოსახულების მქონე სიმბოლო უშუალოდ შეიძლება ჩაიწეროს აპოსტროფებს შორის. მაგალითად, `a`, `z`, `@`, `&`;
- იმ სიმბოლოების ასაკრებად, რომლის უშუალოდ შეტანა შეუძლებელია, მაგალითად, მმართველი სიმბოლოებისათვის, უნდა გამოვიყენოთ სპეციალური მმართველი მიმდევრობა, მაგალითად, `\n` - ახალი სტრიქონის სიმბოლო (ASCII კოდი - 10); `\t` - ჰორიზონტალური ტაბულაციის სიმბოლო (ASCII კოდი - 9); `\f` - ახალ

ფურცელზე გადასვლის სიმბოლო (ASCII კოდი - 12); `\\`- შებრუნებული დახრილი ხაზი;`\"` - ორმაგი ბრჭყალები;`\`` აპოსტროფი; სიმბოლოების მმართველი მიმდევრობები ნაჩვენებია ცხრილ 5-ში:

ცხრილი 5. მმართველი სიმბოლოების მიმდევრობები

Escape მიმდევრობა	აღწერა
\ddd	სიმბოლო რვაობითში
\uxxxx	UNICODE სიმბოლო თექვსმეტობითში (xxxx)
\’	ერთმაგი ბრჭყალები
\”	ორმაგი ბრჭყალები
\\	სლეში(\)- შებრუნებული დახრილი ხაზი
\n	ახალ სტრიქონზე გადასვლა
\f	ახალ ფურცელზე გადასვლა
\t	ჰორიზონტალური ტაბულაციის სიმ-ბოლო (Tab)
\b	ერთი სიმბოლოთი დაბრუნება (BackSpace)
\r	სტრიქონის დასაწყისში დაბრუნება

- ნებისმიერი სიმბოლოს კოდი შეიძლება ჩავწეროთ სამ თანრიგა რვაობითში ჩაწერილი რიცხვის სახით, რომელ-საც წინ უწერია შებრუნებული დახრილი ხაზი და მო-თავსებულია აპოსტროფში. მაგალითად, `\123` არის ასო S-ს კოდი. არაა რეკომენდებული ჩაწერის ამ ფორმის გამოყენება ზემოთ ჩამოთვლილი გრაფიკული გამოსახუ-ლების მქონე სიმბოლოებისა და მმართავი სიმბოლოების

ჩასაწერად, ვინაიდან კომპილატორი რვაობით რიცხვს მაშინვე გადაიყვანს ზემოთ მითითებულ ფორმაში. უდიდესი კოდია `\377`, რომელიც შეესაბამება ათობით 255-ს;

- Unicode კოდირების სტანდარტში ნებისმიერი სიმბოლოს კოდი ჩაიწერება აპოსტროფებში დახრილი ხაზისა და ლათინური ასო U-ს შემდეგ ჩაწერილი ოთხი თექვსმეტობითი ციფრით. მაგალითად, `\u0053` არის ასო S-ს კოდი.

რა ფორმითაც არ უნდა იყოს ჩაწერილი სიმბოლოები, კომპილატორს ისინი გადაყავს Unicode-ში. ასევე Unicode-ში გადაყავს პროგრამის ტექსტი. Java-ს შემსრულებელი გარემო მუშაობს მხოლოდ Unicode კოდირებასთან.

ბულის კონსტანტები

ბულის კონსტანტების მნიშვნელობებია მხოლოდ ორი true და false. ეს მნიშვნელობები არ გარდაიქმნება არცერთ რიცხვით წარმოდგენაში. ეს მნიშვნელობები Java-ში შეიძლება მიენიჭოს მხოლოდ boolean-ად გამოცხადებულ ცვლადებს ან მონაწილეობა მიიღონ ბულის ოპერაციებიან გამოსახულებებში.

სტრიქონული კონსტანტები

სტრიქონული კონსტანტების ჩაწერა Java-ში ხდება სიმბოლოების მიმდევრობის ორმაგ ბრჭყალებში მითითებით. მაგალითად, “Hello World”, “two\nlines”, “\”This is in quotes\””.

მმართავი სიმბოლოები და რვაობითი/თექვსმეტობითი ჩაწერის ფორმები, რომლებიც განსაზღვრულია სიმბოლური კონსტანტებისათვის, ზუსტად ასევე მუშაობენ სტრიქონული კონსტანტების შიგნითაც. ცხადია, რომ მმართავი სიმბოლოები სტრიქონში იწერება აპოსტროფების გარეშე.

Java-ში სტრიქონები უნდა იწყებოდეს და მთავრდებოდეს ერთ სტრიქონში. ენაში არ არსებობს რაიმე სიმბოლო, რომელიც მიუთითებდა სტრიქონის გაგრძელებას.

Java 7-ის სიახლე რიცხვით ლიტერალებში

Java 7-ში (JDK 7) და მის შემდეგ ვერსიებში ნებისმიერ რიცხვით ლიტერალში (როგორც მთელ ისე მცოცავმძიმიან ფორმატში) ნებისმიერ ციფრებს შორის შესაძლებელია გამოყენებული იქნეს ქვედა ტირე (_). ქვედა ტირის გამოყენება საშუალებას იძლევა ერთმანეთისგან გამოვყოთ ციფრების ჯგუფები და ამდაგვარად გავაუმჯობესოთ პროგრამის კოდის კითხვადობა. ჩვეულებრივ, მრავალი ციფრისგან შემდგარი რიცხვის კითხვადობის ასამაღლებლად, ათასეულებს ერთმანეთისაგან აშორებენ ცარიელი სიმბოლოთი ან რაიმე სხვა გამყოფით, ამ მიზნით Java-ში შესაძლებელია გამყოფად ქვედა ტირეს (_) გამოყენება. მაგალითად, კორექტულად ჩაწერილი ლიტარალებია: 123_456_789L; 999_99_9999L; 3.14_15F; 0xFF_EC_DE_5E; 0xCAFE_BABE; 0x7fff_ffff_ffffL; 0b0010_0101; 0b11010010_01101001_10010100_10010010;

რიცხვით ლიტერალებში ქვედა ტირეს კორექტულად გამოსაყენებლად იგი უნდა ჩაიწეროს მხოლოდ ციფრებს შორის. არ შეიძლება მისი ჩასმა შემდეგ ადგილებში:

- რიცხვის თავში ან ბოლოში;
- მცოცავმძიმიან რიცხვებში ათობითი წერტილის წინ ან შემდეგ;
- F ან L სუფიქსების წინ;
- რიცხვის ფუძის პრეფიქსების სიმბოლოებს შორის ან მათ შემდეგ;
- იმ პოზიციებში, სადაც მოსალოდნელია სტრიქონულად ჩაწერილი რიცხვი.

შემდეგ მაგალითებში დემონსტრირებულია ქვედა ტირეს კორექტული და არაკორექტული გამოყენება:

```
3_.1415F;          // არასწორია; არ შეიძლება ქვედა ტირეს
                   // ჩასმა ათობითი წერტილის წინ
3._1415F;          // არასწორია; არ შეიძლება ქვედა ტირეს
                   //  ჩასმა ათობითი წერტილის შემდეგ
99_99_9999_L;      // არასწორია; არ შეიძლება ქვედა ტირეს
                   //ჩასმა L სუფიქსის წინ
_52;               // არასწორია;  ეს არის იდენტიფიკატორი, არ
                   არის ციფრული ლიტერალი
5_2;               // სწორია (ათობითი ლიტერალია)
52_;               // არასწორია; არ შეიძლება ქვედა ტირეს
                   //ჩასმა ლიტერალის ბოლოს
5_______2;         // სწორია (ათობითი ლიტერალია)
0_x52;             // არასწორია; არ შეიძლება ქვედა ტირეს ჩასმა
                   //0x პრეფიქსების სიმბოლოებს შორის
0x_52;             // არასწორია; არ შეიძლება ქვედა ტირეს ჩასმა
                   რიცხვის თავში
```

```
0x5_2;          // სწორია  (თექვსმეტობითი ლიტერალია)
0x52_;          // არასწორია; არ შეიძლება ქვედა ტირეს
                //ჩასმა  რიცხვის ბოლოში
```

ცვლადები

ცვლადი Java პროგრამაში მონაცემთა შენახვის ძირითადი კომპონენტია. ცვლადი განისაზღვრება იდენტიფიკატორის, ტიპის და საწყისი მნიშვნელობით, რომელიც პრინციპში არასავალდებულოა. გარდა ამისა, ყველა ცვლადს აქვს განსაზღვრის არე, რომელიც განაპირობებს მის ხილვადობას სხვა ობიექტების მიმართ და არსებობის დროს.

ცვლადების გამოცხადება

Java-ში ცვლადების გამოცხადება უნდა მოხდეს მის გამოყენებამდე. ცვლადების გამოცხადების ძირითადი ფორმა ასე შეიძლება წარმოვადგინოთ:

<ტიპი> <იდენტიფიკატორი> [=<მნიშვნელობა>] [,<იდენტიფიკატორი> [=მნიშვნელობა] ...];

<ტიპი> - აქ უნდა ჩაიწეროს კონკრეტული Java-ს ერთ-ერთი 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 მითით-

ებაც, ოღონდ მისი მნიშვნელობის ტიპი უნდა ემთხვეოდეს ცვლადის ტიპს ან დაიყვანებოდეს ამ ტიპზე.

ერთი და იგივე ტიპის რამდენიმე ცვლადის ერთდროულად გამოცხადებისათვის შეიძლება გამოვიყენოთ სია, სადაც წევრები ერთმანეთისგან გამოყოფილი იქნება მძიმით. მაგალითად,

```
int a, b, c;    // გამოცხადებულია სამი int ტიპის ცვლადი: a,b,c
int d = 3, e, f = 5;  // გამოცხადებულია 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
```

ამ მაგალითში გამოცხადებული იდენტიფიკატორები ძალიან მარტივია, თუმცა Java-ში დასაშვებია ნებისმიერი სწორად გაფორმებული იდენტიფიკატორი, ნებისმიერი გამოცხადებული ტიპით.

ტიპების გარდაქმნა და დაყვანა

პროგრამისტებს ხშირად უწევთ ერთი ტიპის ცვლადისათვის განსხვავებული ტიპის მნიშვნელობის მინიჭება. თუ ეს ორივე ტიპი **თავსებადია,** Java გარდაქმნას ავტომატურად აკეთებს. მაგალითად, ყოველთვის შეიძლება int ტიპის მნიშვნელობა მიენიჭოს long ტიპის ცვლადს. მაგრამ ყველა ტიპი არაა თავსებადი, ამიტომ **ავტომატური გარდაქმნაც** ასეთ შემთხვევაში შეუძლებელია. მაგალითად, არ არსებობს

არავითარი ავტომატური გარდაქმნის მეთოდი double-დან byte-ში. Java-ში **არათავსებად** ტიპებს შორის გარდაქმნა საბედნიეროდ მაინც შეიძლება. ამისათვის საჭიროა **დაყვანის** გამოყენება, რომელიც ახდენს არათავსებადი ტიპების **ცხად გარდაქმნას**.

ტიპების ავტომატური გარდაქმნა Java-ში

ცვლადზე სხვა ტიპის მნიშვნელობის მინიჭებისას ავტომატური გარდაქმნა გამოიყენება შემდეგი პირობების დაცვის შემთხვევაში:

- ორივე ტიპი თავსებადია;
- რასაც ენიჭება, იმ ტიპის სიგრძე მეტია მისანიჭებელ ტიპზე.

ამ ორი პირობის დაცვისას ხდება გარდაქმნა გაფართოებით (wedenning conversion). მაგალითად, int ტიპი ყოველთვის უფრო მეტია ვიდრე byte-ს ტიპის ყველა დასაშვები მნიშვნელობა, ამიტომ არაა საჭირო რაიმე ცხადი დაყვანის ოპერატორი.

გაფართოებითი გარდაქმნის თვალსაზრისით, მთელიდან მცოცავ მძიმიან რიცხვით ტიპებში გარდაქმნა თავსებადია. მაგრამ არ არსებობს ავტომატური გარდაქმნა რიცხვითი ტიპებიდან char და boolean ტიპებში. char და boolean ტიპები ასევე შეუთავსებადია ერთმანეთის მიმართაც.

Java ასრულებს ტიპების ავტომატურ გარდაქმნას მთელ რიცხვა კონსტანტის მინიჭებისას byte, short, long და char ტიპის ცვლადებზე.

არათავსებადი ტიპების დაყვანა

მართალია ავტომატური დაყვანა მოსახერხებელია, მაგრამ იგი ყველა შემთხვევაში არ მუშაობს. მაგალითად, როგორ უნდა მოხდეს გარდაქმნა int ტიპის მინიჭებისას byte ტიპზე? ასეთი ტიპის გარდაქმნას უწოდებენ **გარდაქმნას შემცირებით** (narrowing conversion), ვინაიდან საწყისი მნიშვნელობა ისე უნდა შემცირდეს, რომ იგი ჩაეტიოს მიზნობრივ ცვლადის ტიპში.

ორ არათავსებად ტიპს შორის გარდაქმნის განსახორციელებლად საჭიროა ტიპების **დაყვანა** (casting). **დაყვანა** - ესაა ტიპების ცხადი გარდაქმნა. დაყვანის ზოგად ფორმას ასეთი სახე აქვს:

(<მიზნობრივი ტიპი>) <მნიშვნელობა>

აქ <მიზნობრივი ტიპი> განსაზღვრავს იმ ტიპს, რომელშიც უნდა გარდაიქმნას <მნიშვნელობა>. კოდის შემდეგი ფრაგმენტი int ტიპს დაიყვანს byte ტიპზე, იგი შემცირდება (რედუცირდება) საწყისი რიცხვის byte ტიპის დიაპაზონის მოდულზე გაყოფის ნაშთამდე (ამ შემთხვევაში 128-ზე გაყოფის ნაშთი).

```
int a;
byte b;
// ...
b = (byte) a;
```

უფრო მარტივად რომ ვთქვათ, დაყვანის პროცესის დროს ხდება ზედმეტი უფროსი ბიტების მოცილება (გადაყრა). მაგალითად,

byte b = (byte) 300;

ცვლადი b მნიშვნელობად ღებულობს 44-ს. მართლაც, რიცხვი 300 ორობით წარმოდგენაში ტოლია 100101100, გადავარდება უფროსი ბიტი და მიიღება 00101100, ეს ორობითი რიცხვი კი ათობითში 44-ია.

მთელი ტიპის ცვლადზე მცოცავმძიმიანი მნიშვნელობის მინიჭებისას ხდება სხვა სახის დაყვანა - შეკვეცა (truncation, reduction). ამ დროს მთელი ნაწილი ენიჭება ცვლადს, ხოლო წილადი ნაწილი იკარგება. მაგალითად, თუ მთელი ტიპის ცვლადს ვანიჭებთ 1.23 წილად რიცხვს, მაშინ ცვლადის მნიშვნელობა იქნება 1-ის ტოლი, ხოლო 0.23 ჩამოეჭრება. რასაკვირველია, თუ მთელი ნაწილი ძალიან დიდია, იგი რედუცირდება მთელი ტიპის დიაპაზონის მოდულამდე.

ლისტინგ 3-ზე ნაჩვენები პროგრამა გვიჩვენებს ტიპების გარდაქმნის და დაყვანის რამდენიმე მაგალითს.

ლისტინგი 3. ტიპების გარდაქმნა და დაყვანა

```
public class Conversion {
  public static void main(String[] args) {
    byte b;
    int i = 257;
    double d = 323.142;
    System.out.println("\n int -ის გარდაქმნა byte-ში");
    b = (byte) i;
    System.out.println("i და b " + i + " " + b);
    System.out.println("\n double-ს გარდაქმნა int-ში");
    i = (int) d;
    System.out.println("d და i " + d +" " + i);
    System.out.println("\n double-ს გარდაქმნა byte-ში");
```

```
        b = (byte) d;
        System.out.println("d და b " + d + " " + b);
    }
}
```

ამ პროგრამის შედეგად გამოვა:

```
int -ის გარდაქმნა byte-ში
 i და b 257 1
double-ს გარდაქმნა int-ში
 d და i 323.142 323
double-ს გარდაქმნა byte-ში
d და b 323.142 67
```

გამოსახულებებში ტიპის ავტომატური ამაღლება

მინიჭების ოპერაციის გარდა გარკვეული გარდაქმნები შეიძლება მოხდეს გამოსახულებების გამოთვლისას. მაგალითად,

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

შუალედური გამოთვლის შედეგი a * b შეიძლება გავიდეს byte-ს ტიპის დასაშვები დიაპაზონის საზღვრებს გარეთ. ასეთი პრობლემების გადასაწყვეტად Java-ში გამოსახულებების გამოთვლისას ხდება ყოველი byte და short ოპერანდის **ავტომატური ამაღლება** int-მდე. ანუ შუალედური გამოსახულების გამოთვლა ხდება მთელრიცხვა მნიშვნელობებით და არა ბაიტებით. შუალედური გამოსახულება

იქნება 50*40, რომლის შედეგია 2000, რაც დასაშვებია, მიუხედავად იმისა, რომ a და b ცვლადები byte ტიპისაა.

მიუხედავად იმისა, რომ ავტომატური ამაღლება მოსახერხებელია, მას ზოგჯერ მივყავართ კომპილაციის დროს შეცდომებამდე. მაგალითად, გარეგნულად კორექტული კოდი იწვევს კომპილაციის შეცდომას:

```
byte b = 50;
b = b * 2;      // შეცდომა! int ტიპის მნიშვნელობა
                // ენიჭება byte ტიპის ცვლადს!
```

ასეთ შემთხვევაში გამოყენებული უნდა იქნეს ტიპების ცხადი დაყვანა:

```
byte b = 50;
b = (byte) (b * 2);
```

Java-ში მოქმედებს **ტიპის ამაღლების რამდენიმე წესი**:

1. byte, short და char მნიშვნელობები ამაღლდება int ტიპამდე;
2. თუ ერთი ოპერანდის ტიპია long, მთლიანი გამოსახულების ტიპი ამაღლდება long-მდე;
3. თუ ერთი ოპერანდი float-ის ტიპისაა, მთლიანი გამოსახულების ტიპი ამაღლდება float-მდე;
4. თუ ნებისმიერი ოპერანდი double-ის ტიპისაა, მთლიანი გამოსახულების ტიპი ამაღლდება double-მდე.

მასივები

მასივი ესაა ერთი ტიპის მონაცემების ერთობლიობა, რომლებზე მიმართვა ხდება ერთი, საერთო სახელის საშუალებით. Java-ში დაშვებულია, როგორც ერთგანზომილებიანი, ისე მრავალგანზომილებიანი ნებისმიერი ტიპის მასივების გამოცხადება. მასივის კონკრეტულ ელემენტზე მიმართვა ხდება მისი ინდექსის მიხედვით.

ერთგანზომილებიანი მასივები

ერთგანზომილებიანი მასივი წარმოადგენს ერთი ტიპის ცვლადების სიას. მასივის შესაქმნელად, ჯერ უნდა შეიქმნას საჭირო ტიპის მასივის ცვლადი. ერთგანზომილებიანი მასივის გამოცხადების ზოგადი ფორმაა:

<ტიპი> <მასივის სახელი>[];

აქ <ტიპი> წარმოადგენს მასივის შემადგენელი ელემენტების ტიპს. მაგალითად, შემდეგი ოპერატორით ხდება monthDays მასივის გამოცხადება, რომლის ელემენტები int ტიპისაა:

int monthDays[];

ამ გამოცხადებით ჯერ მასივი არ შექმნილა. monthDays მასივის ფაქტიური მნიშვნელობა ჯერჯერობით არის null, რომელიც წარმოადგენს მასივს მნიშვნელობის გარეშე. monthDays მასივის ცვლადის რეალური ფიზიკური მთელი რიცხვების მასივთან დასაკავშირებლად საჭიროა ოპერაციით new მეხსიერების გამოყოფა და მისი მისამართის მინიჭება ამ ცვლადზე. მომავალში ჩვენ ამ ოპერატორს უფრო დაწვრი-

ლებით განვიხილავთ, ახლა კი აღვნიშნავთ, რომ მისი ზოგადი ფორმა ასეთია:

<მასივის სახელი> = new <ტიპი>[<ზომა>] ;

<ტიპი> აღწერს დარეზერვირებული მონაცემების მეხსიერების ტიპს. <ზომა> მიუთითებს მასივში ელემენტების რაოდენობას, ხოლო <მასივის სახელი> მასივთან დაკავშირებული ცვლადია. ანუ, new ოპერაციით მეხსიერების გამოსაყოფად საჭიროა ელემენტების ტიპისა და რაოდენობის მითითება. მასივის ელემენტებისათვის new ოპერაციით გამოყოფილი მეხსიერების ინიციალიზაცია მოხდება ნულოვანი მნიშვნელობებით. ჩვენს მაგალითში ხდება 12 ელემენტიანი მასივისათვის მეხსიერების რეზერვირება და მისი დაკავშირება monthDays მასივის სახელთან.

monthDays = new int[12];

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. ამრიგად Java-ში ყველა მასივი დინამიკურად იქმნება.

მასივის გამოცხადებისა და შექმნის შემდეგ შესაძლებელია მის თითოეულ ელემენტს მივმართოთ ინდექსით, რომელიც კვადრატულ ფრჩხილებში იქნება მოთავსებული. მასივის ინდექსი იწყება 0-დან. მაგალითად, შემდეგი ოპერატორი monthDays მასივის მეორე ელემენტს ანიჭებს მნიშვნელობად 28-ს.

monthDays [1] = 28;

შემდეგი კოდის სტრიქონი ასახავს იმ ელემენტის მნიშვნელობას, რომელიც იმყოფება 3 ინდექსიან ელემენტად:

System.out.println(monthDays[3]);

მთელი პროცესის სადემონსტრაციოდ, ლისტინგ 4-ით ჩაწერილია პროგრამა, რომელიც ქმნის წლის ყოველი თვის დღეების რაოდენობის მასივს.

ლისტინგი 4.

```
public class Array {
    public static void main(String[] args) {
        int monthDays[];
        monthDays = new int[12];
        monthDays[0] = 31;
        monthDays[1] = 28;
        monthDays[2] = 31;
        monthDays[3] = 30;
        monthDays[4] = 31;
        monthDays[5] = 30;
        monthDays[6] = 31;
        monthDays[7] = 31;
        monthDays[8] = 30;
        monthDays[9] = 31;
        monthDays[10] = 30;
        monthDays[11] = 31;
        System.out.println ("აპრილში არის "+ monthDays[3] + "დღე");
    }
}
```

ამ პროგრამის შესრულებით კონსოლზე გამოვა აპრილში დღეების რაოდენობა.

მასივის ცვლადის გამოცხადება შეიძლება გავაერთიანოთ მისთვის მეხსიერების გამოყოფასთან:

```
int monthDays[] = new int[12];
```

ასევე შესაძლებელია მასივის ინიციალიზაცია მისი გამოცხადებისას. ეს პროცესი ჰგავს ცვლადის ინიციალიზაციის პროცესს. **მასივის ინიციალიზატორი** ესაა ერთმანეთისაგან მძიმით დაშორებული გამოსახულებების სია, რომელიც ფიგურულ ფრჩხილებშია მოთავსებული. მძიმით გამოყოფილია მასივის ელემენტების მნიშვნელობები. მასივი ავტომატურად ისეთი ზომის იქმნება, რომ მასში ჩაეტიოს ინიციალიზატორში მითითებული ყველა ელემენტი. ამ დროს new ოპერატორის გამოყენება საჭირო არაა. წინა მაგალითის პროგრამის ფრაგმენტი ასე შეიძლება ჩავწეროთ:

ლისტინგი 5.

```
public class AutoArray {
    public static void main(String[] args) {
        int monthDays[] = {31,28,31,30,31,30,31,31,30,31,30,31};
        System.out.println ("აპრილში არის "+ monthDays[3] + "დღე");
    }
}
```

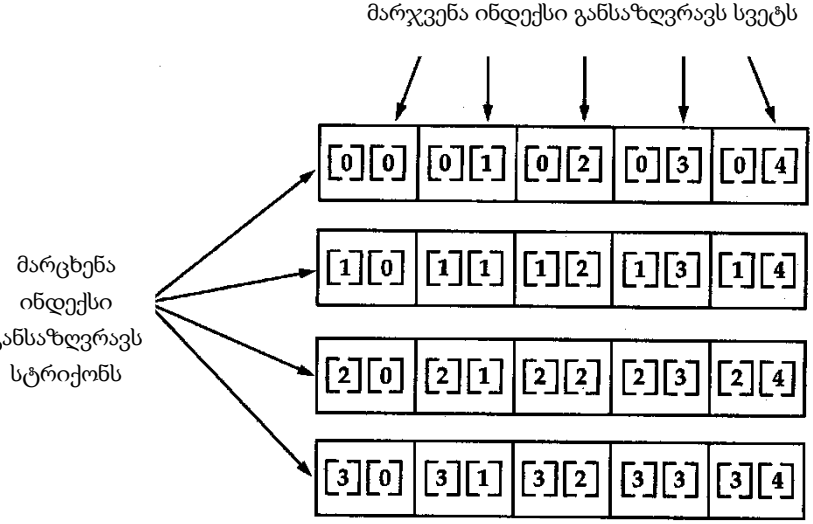
Java-ს შემსრულებელი გარემო გულდასმით ამოწმებს, ხომ არ მოხდა შენახვის ან მიმართვის მცდელობა ისეთ ინდექსზე, რომელიც გადის მასივის დასაშვები დიაპაზონის გარეთ. მაგალითად, ჩვენი მაგალითის შემთხვევაში შემსრულებელი გარემო ამოწმებს ინდექსი იმყოფება თუ არა 0 – 11 დიაპაზონში. ამ დიაპაზონის გარეთ მიმართვის მცდელობა შესრულების პროცესში იწვევს შეცდომას.

მრავალგანზომილებიანი მასივები

Java-ს მრავალგანზომილებიანი მასივი წარმოადგენს მასივების მასივს. ისინი მოქმედებენ ჩვეულებრივი მრავალგანზომილებიანი მასივის ანალოგიურად. თუმცა მათ გარკვეული თავისებურებაც ახასიათებთ. მრავალგანზომილებიანი მასივის თითოეული დამატებითი ინდექსის მისათითებლად იყენებენ კვადრატული ფრჩხილების ცალკე წყვილს. მაგალითად, ორგანზომილებიანი მასივი შეიძლება გამოვაცხადოთ ასე:

int twoD[][] = new int[4][5];

ეს ოპერატორი გამოჰყოფს მეხსიერებას 4x5 განზომილებიანი მასივისათვის. მასივის ლოგიკური ორგანიზება ნაჩვენებია სურ. 5-ზე



მოცემულია int twoD [] [] = new int [4] [5];

სურ. 5. ორგანზომილებიანი მასივის ლოგიკური წარმოდგენა

მრავალგანზომილებიანი მასივისათვის მეხსიერების გამოსაყოფად ჯერ საჭიროა მივუთითოთ მეხსიერება მხოლოდ პირველი (მარცხენა) განზომილებისათვის. ყოველი შემდეგი განზომილებისათვის მეხსიერება შეიძლება გამოვყოთ ცალკე. მაგალითად, შემდეგი კოდი არეზერვირებს მეხსიერებას twoD მასივის გამოცხადებისას პირველი განზომილებისათვის. მეორე განზომილებისათვის მეხსიერების გამოყოფა ხდება ხელით, ცალ-ცალკე.

```
int twoD [] [] = new int [4] [] ;
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

მართალია ამ შემთხვევაში მეორე განზომილებისათვის ცალკე მეხსიერების გამოყოფა არ იძლევა რაიმე უპირატესობას, მაგრამ სხვა სიტუაციაში ეს შეიძლება სასარგებლო გამოდგეს. მაგალითად, მეხსიერების ხელით გამოყოფისას არაა აუცილებელი თითოეული განზომილებისათვის ერთნაირი ზომა მივუთითოთ. ამრიგად, პროგრამისტი თვითონ მართავს მასივის თითოეული განზომილების სიგრძეს.

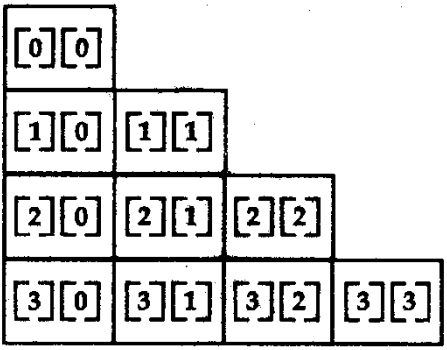
მაგალითად, შემდეგი პროგრამა ქმნის ორგანზომილებიან მასივს, მეორე განზომილების სხვადასხვა ზომებით.

ლისტინგი 6.

```
class TwoDAgain {
   public static void main(String args[]) {
      int twoD[][] = new int[4][];
      twoD[0] = new int[1];
```

```
      twoD[1] = new int[2];
      twoD[2] = new int[3];
      twoD[3] = new int[4];
     }
}
```

შექმნილ მასივს ექნება შემდეგი სახე (სურ 6):



სურ. 6. ორგანზომილებიანი მასივი, რომლის მეორე განზომილება სხვადასხვა ზომისაა

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.

ლისტინგი 7.

```
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 გამოცხადების ალტერნატიული ფორმა

მასივების გამოცხადებისათვის შეიძლება გამოყენებული იქნეს მეორე ფორმა:

<ტიპი> [] <ცვლადის სახელი>;

ამ ფორმაში კვადრატული ფრჩხილები მიეთითება ტიპის გამოცხადების შემდეგ და არა მასივის ცვლადის შემდეგ. მაგალითად, 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

ეს ალტერნატიული ფორმა ასევე მოსახერხებელია, როდესაც მასივი ცხადდება მეთოდის დაბრუნების ტიპად.

ოპერაციები

Java-ში შეიძლება გამოვყოთ ოპერაციების 4 ძირითადი ჯგუფი: არითმეტიკული ოპერაციები, ბიტური ოპერაციები, შედარების ოპერაციები და ლოგიკური ოპერაციები. არსებობს კიდევ სხვა დამატებითი ოპერაციებიც, რომლებიც განსაკუთრებულ სიტუაციაში გამოიყენება.

არითმეტიკული ოპერაციები

არითმეტიკული ოპერაციები ისევე გამოიყენება მათემატიკურ გამოსახულებებში, როგორც ამას ვაკეთებთ ალგებრაში. №6 ცხრილში ჩამოთვლილია არითმეტიკული ოპერაციები

ცხრილი 6. არითმეტიკული ოპერაციები

ოპერაცია	აღწერა
+	შეკრება
-	გამოკლება
*	გამრავლება
/	გაყოფა
%	მოდულით გაყოფა
++	ინკრემენტი
+=	ავტოასოციური შეკრება (შეკრება მინიჭებით)
-=	ავტოასოციური გამოკლება (გამოკლება მინიჭებით)
*=	ავტოასოციური გამრავლება (გამრავლება

	მინიჭებით)
/=	ავტოასოციური გაყოფა (გაყოფა მონიჭებით)
%=	ავტოასოციური მოდულით გაყოფა (გაყოფა მინიჭებით)
– –	დეკრიმენტი

არითმეტიკული ოპერაციების ოპერანდების ტიპი უნდა იყოს რიცხვითი. არითმეტიკული ოპერაციების ჩატარება არ შეიძლება boolean ტიპის ცვლადებზე, მაგრამ შესაძლებელია char ტიპთან, ვინაიდან Java-ში ეს ტიპი, ფაქტიურად int ტიპის ქვესიმრავლეა.

ძირითადი არითმეტიკული ოპერაციები

ყველა ძირითადი ოპერაცია: შეკრება, გამოკლება, გამრავლება და გაყოფა რიცხვით ტიპებთან მოქმედებს ისე, როგორც ალგებრაში. გამოკლების ოპერაციას, ასევე, აქვს უნარული (ერთ ოპერანდიანი) ფორმა, რომელიც ნიშანს უცვლის მის ერთადერთ ოპერანდს. უნდა გვახსოვდეს, რომ მთელრიცხვა ტიპზე გაყოფის ოპერაციის გამოყენებისას განაყოფს არ ექნება წილადი ნაწილი.

შემდეგი მარტივი პროგრამა დემონსტრირებას უკეთებს არითმეტიკულ ოპერაციებს (ლისტინგი 8). აქ დემონსტრირებულია განსხვავება მთელრიცხვა გაყოფასა და მცოცავ მძიმიან გაყოფას შორის.

ლისტინგი 8.

```
class BasicMath {
    public static void main(String args[]) {
        // მთელ რიცხვა არითმეტიკული ოპერაციები
     System.out.println("მთელ რიცხვა არითმეტიკა");
```

```
        int a = 1 + 1;
        int b =  a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.println("a =¬ " + a);
        System.out.println("b ¬= " + b);
        System.out.println("c ¬= " + c);
        System.out.println("d =¬"+ d);
        System.out.println("e ¬ =" + e);
           // double ტიპზე არითმეტიკული ოპერაციები
           System.out.println("\n მცოცავ მძიმიანი არითმეტიკა");
        double da = 1 + 1;
        double db = da * 3;
        double dc = db / 4;
        double dd = dc - da;
        double de = -dd;
        System.out.println("da = "+ da);
        System.out.println("db = "+ db);
        System.out.println("dc ¬= " + dc);
        System.out.println("dd ¬= " + dd);
        System.out.println("de =¬ " + de);
    }
}
```

ამ პროგრამის შესრულების შედეგად კონსოლზე გამოვა ასეთი შედეგი:

მთელრიცხვა არითმეტიკა
a = 2
b = 6
c = 1
d = -1
e = 1

მცოცავ მძიმიანი არითმეტიკა
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0. 5

როგორც მაგალითიდან ნახეთ მთელი რიცხვების გაყოფისას შედეგად ისევ მთელი რიცხვი მიიღება, ამიტომ მას ხშირად „მთელ გაყოფას“ უწოდებენ. მაგალითად, 5/2 შედეგად იძლევა 2-ს და არა 2.5, ხოლო 5/(-3) შედეგია -1. წილადი ნაწილი უბრალოდ ვარდება.

მათემატიკისათვის ეს უცნაური წესი ბუნებრივია პროგრამირებაში: თუ ორივე ოპერანდს ერთიდაიგივე ტიპი აქვთ, მაშინ შედეგსაც იგივე ტიპი აქვს. საკმარისია დაიწეროს 5/2.0 ან 5.0/2 ან 5.0/2.0 მაშინ მივიღებთ ნამდვილი რიცხვების გაყოფას და შედეგი იქნება 2.5.

მოდულით გაყოფის (ნაშთის) ოპერაცია

მოდულით გაყოფის ანუ ნაშთის გამოთვლის ოპერაცია % აბრუნებს გაყოფის ოპერაციის ნაშთს. ეს ოპერაცია შეიძლება გამოყენებული იქნეს როგორც მთელ ტიპებზე, ისე მცოცავ-მძიმიან რიცხვებზე. შემდეგ პროგრამაში დემონსტრირებულია % ოპერაციის გამოყენება

ლისტინგი 9.

```
class Modu1us {
    public static void main(String args[]) {
        int x =42;
```

```
        double y = 42.25;
        System.out.print1n("x mod 10 ="+x % 10);
        System.out.print1n("y mod 10 ="+y % 10);
    }
}
```

პროგრამის შესრულების შედეგია:

x mod 10 = 2
y mod 10 = 2.25

მოდულით გაყოფის ოპერაცია მთელი ტიპის რიცხვებისათვის ზოგადად ასე შეიძლება ჩავწეროთ:
a % b = a – (a / b) * b
მაგალითად, 5 % 2 შედეგად მოგვცემს 1 , ხოლო 5 % (-3) მოგვცემს 2 , ვინაიდან 5-(5/(-3))*(-3)=5-(-1)*(-3)=5-3=2 . ასევე (-5)%3 შედეგად იქნება -2 , ვინაიდან (-5)-((-5)/3)*3=-5-(-1*3)=-5+3=-2 .

ავტოასოციური ოპერაციები

Java-ში არსებობს სპეციალური ოპერაციები, რომლებიც წარმოადგენენ არითმეტიკული და მინიჭების ოპერაციის გაერთიანებას. პროგრამირებაში ხშირად გვხვდება ამდაგვარი ოპერატორები:

a = a + 4;

Java-ში ეს ოპერატორი შეიძლება ასე ჩაიწეროს

a += 4;

ოპერატორის ამ ვერსიაში გამოყენებულია **ავტოასოციური ოპერაცია** +=. ორივე ეს ოპერაცია ერთიდაიგივე მოქმედებას აკეთებს: 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 ორ ოპერანდიანი არითმეტიკული ოპერაციებისათვის. ამრიგად, ყველა ოპერატორი, რომელსაც აქვს ასეთი ფორმა

<ცვლადი> = <ცვლადი> <ოპერაცია> <გამოსახულება>;

შეიძლება ჩაიწეროს ასე

<ცვლადი> <ოპერაცია>= გამოსახულება;

ავტოასოციური ოპერაციები ორ უპირატესობას იძლევა:

1. იგი ამცირებს შესატანი კოდის მოცულობას, ვინაიდან წარმოადგენს გრძელი ფორმის ერთგვარ „შემოკლებულ“ ფორმას;
2. შემსრულებელ გარემოში მათი რეალიზაცია უფრო ეფექტურია, ვიდრე მისი ექვივალენტური გრძელი ფორმის.

ამ მიზეზების გამო პროფესიონალურად შედგენილ Java პროგრამებში ავტოასოციური ოპერაციები ძალიან ხშირად გვხვდება.

მაგალითი:

ლისტინგი 10.

```
class OpEquals {
    public static void main(String args[]){
     int a = 1;
     int b = 2;
     int c = 3;
     a += 5;
     b *= 4;
     c += a * b;
     c %= 6;
     System.out.println("a = ¬ " + a);
     System.out.println("b = ¬"+ b);
     System.out.println("c ¬=  " + c);
    }
}
```

პროგრამის შესრულების შედეგია:

a = 6

b = 8

c = 3

ინკრემენტი და დეკრემენტი

ოპერაცია ინკრემენტი აღინიშნება ასე: ++ , ხოლო დეკრემენტი აღინიშნება ასე: -- .

ინკრემენტის ოპერაცია ოპერანდის მნიშვნელობას ზრდის ერთით, ხოლო დეკრემენტი 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 პოსტფიქსური ფორმით, როდესაც ოპერაციის სიმბოლოები ჩაიწერება ოპერანდის შემდეგ, ისე პრეფიქსული ფორმით, ამ დროს ოპერაციის სიმბოლოები 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

იცვლება (იზრდება ან მცირდება იმისდა მიხედვით ++ თუ -- მითითებული). მაგალითად,

```
x = 25;
y = ++x;
```

ამ შემთხვევაში y-ის მნიშვნელობა გახდება 26, როგორც მოსალოდნელი 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 იქნება 26-ის ტოლი. შესაბამისად y = x++; სტრიქონის მნიშვნელობა ეკვივალენტური შემდეგი ორი სტრიქონის:

```
y = x;
x = x + 1;
```

შემდეგ პროგრამაში დემონსტრირებულია ინკრემენტის ოპერაციის გამოყენება

ლისტინგი 11. ++ ოპერაციის დემონსტრირება

```
class IncDec {
    public static void main(String args[]) {
     int a = 1;
     int b = 2;
     int c;
     int d;
     c = ++b;
     d = a++;
     c++;
     System.out.println("a = " + a);
     System.out.println("b = " + b);
     System.out.println("c = " + c);
     System.out.println("d = " + d);
    }
}
```

ამ პროგრამის შედეგად კონსოლზე გამოვა:

a = 2

b = 3

c = 4

d = 1

ოპერაციები ბიტებზე

Java-ში შესაძლებელია განხორციელდეს გარკვეული ოპერაციები long, int, short, char და byte მთელრიცხვა ტიპის ბიტებზე. ცხრილში 7. ჩამოთვლილია ეს ოპერაციები

ცხრილი 7. ბიტური ოპერაციები

ოპერაცია	აღწერა
~	ბიტური უნარული ოპერაცია NOT (უარყოფა)
&	ბიტური AND (და)
|	ბიტური OR (ან)

^	ბიტური გამორიცხვა OR
>>	მარჯვნივ დაძვრა
>>>	მარჯვნივ დაძვრა ნულებით შევსებით
<<	მარცხვნივ დაძვრა
&=	ავტოასოციური ბიტური AND
|=	ავტოასოციური ბიტური OR
^=	ავტოასოციური ბიტური OR გამორიცხვა
>>=	ავტოასოციური მარჯვნივ დაძვრა
>>>=	ავტოასოციური მარჯვნივ დაძვრა ნულებით შევსებით
<<=	ავტოასოციური მარცხნივ დაძვრა

ვინაიდან ბიტური ოპერაციები მანიპულირებენ მთელრიცხვა მნიშვნელობების ბიტებზე, ამიტომ მნიშვნელოვანია გავიგოთ, როგორ იცვლება რიცხვების მნიშვნელობები ამ მანიპულირების შედეგად. კერძოდ, მნიშვნელოვანია ვიცოდეთ, როგორ ინახავს Java-ს შემსრულებელი გარემო მთელრიცხვა მნიშვნელობებს და როგორ ხდება უარყოფითი რიცხვების წარმოდგენა.

ყველა მთელრიცხვა ტიპი მეხსიერებაში ჩაიწერება ორობით ფორმაში. მაგალითად, byte ტიპის ცვლადში რიცხვი 42 ორობით წარმოდგენაში ასე ჩაიწერება 00101010, სადაც თითოეული თანრიგი 2-ის ხარისხს წარმოადგენს, დაწყებული 2^0 უკიდურესი მარჯვენა ბიტიდან. შემდეგი ბიტი არის 2^1, შემდეგი 2^2, 2^3 და ა.შ.

ამრიგად,

$42=0*2^0+1*2^1+0*2^2+1*2^3+0*2^4+1*2^5+0*2^6+0*2^7=2^1+2^3+2^5=2+8+32$

ყველა მთელრიცხვა ტიპი (char-ის გამოკლებით) ნიშნიანი მთელი რიცხვებია. ანუ მათი მნიშვნელობები შეიძლება იყოს, როგორც დადებითი, ისე უარყოფითი რიცხვები. Java-ში რიცხვების წარმოსადგენად გამოყენებულია ორობითი რიცხვები დამატებით კოდში. დამატებით კოდში უარყოფითი რიცხვების წარმოსადგენად ჯერ ხდება რიცხვის ორობითი თანრიგების ინვერტირება (ანუ 0 მნიშვნელობის თანრიგები შეიცვლება 1-ით, ხოლო 1-ები 0-ით), ხოლო შემდეგ შედეგს ემატება 1. მაგალითად, -42 დამატებითი კოდის მისაღებად ჯერ 42-ის ორობითი 00101010 მნიშვნელობის ყოველი თანრიგი შეიცვლება თავისი შებრუნებულით ე.ი. მივიღებთ 11010101, შემდეგ 1 დამატებით მივიღებთ 11010110 ანუ -42-ს. პირიქით, უარყოფითი რიცხვის დეკოდირებისათვის (ორობითიდან ათობითი რიცხვის მისაღებად) ჯერ უნდა ინვერტირება გავუკეთოთ ყველა ბიტს, ხოლო შემდეგ შედეგს დავუმატოთ 1. მაგალითად, -42 ანუ 11010110 ინვერტირებით მივიღებთ 00101001 (ანუ 41), შემდეგ ვუმატებთ 1 და მივიღებთ 00101010 (ანუ 42).

გასათვალისწინებელია, რომ ვინაიდან Java-ში ყველა მთელ რიცხვს ნიშნიანი მნიშვნელობა აქვს, ბიტური ოპერაციების შემდეგ შეიძლება მოულოდნელი შედეგები მივიღოთ. მაგალითად, ყველაზე უფროსი ბიტის მნიშვნელობა თუ გახდება 1-ის ტოლი, მაშინ რიცხვი აღიქმება, როგორც უარყოფითი. უნდა გვახსოვდეს, რა გზითაც არ უნდა გახდეს უფროსი ბიტი 1-ის ტოლი, იგი იქნება უარყოფითი რიცხვი.

ბიტური ლოგიკური ოპერაციები

ბიტური ლოგიკური ოპერაციებია: &, |, ^, ~. თითოეული ამ ოპერაციის შედეგი ნაჩვენებია შემდეგ ცხრილში. ბიტური ოპერაციები ხორციელდება თითოეული ოპერანდის თითოეულ, ცალკეულ ბიტზე.

ცხრილი 8. ბიტური ლოგიკური ოპერაციები

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

ბიტური NOT (არა)

ამ ოპერაციას უწოდებენ ლოგიკურ „არა“-ს ან ინვერსიას, აღინიშნება „~“ სიმბოლოთი. ის არის უნარული (ერთ ოპერანდიანი) ოპერაცია და ახორციელებს თითოეული ბიტის ინვერტირებას (0-ს ცვლის 1-ით, 1-ს ცვლის 0-ით). მაგალითად, 42 რომელიც ოროობითში ასეთი ბიტების მიმდევრობით ჩაიწერება:

00101010

NOT ოპერაციის გამოყენების შემდეგ გარდაიქმნება ასე:

11010101

ბიტური AND (და)

ამ ოპერაციას უწოდებენ ბიტურ და-ს ან კონიუნქციას, აღინიშნება სიმბოლოთი „&“. ამ ოპერაციის შედეგად რომელიმე

თანრიგში მიღებული ბიტის მნიშვნელობა ტოლია 1-ის, თუ ოპერანდების შესაბამისი ბიტები ტოლია 1-ის. ყველა სხვა შემთხვევაში შედეგის ბიტი 0-ის ტოლია. მაგალითად,

```
   00101010      42
 & 00001111      15
- - - - - - - - - - - - - - - - - -
   00001010      10
```

ბიტური OR

ამ ოპერაციას უწოდებენ ბიტურ ან-ს ან დიზიუნქციას, აღინიშნება სიმბოლოთი „|“. ამ ოპერაციის შედეგად რომელიმე თანრიგში მიღებული ბიტის მნიშვნელობა ტოლია 1-ის, თუ ოპერანდების შესაბამისი ბიტების ერთ-ერთი მნიშვნელობა მაინც ტოლია 1-ის. მაგალითი:

```
   00101010      42
 | 00001111      15
- - - - - - - - - - - - - - - - - -
   00101111      47
```

ბიტური XOR

ამ ოპერაციას უწოდებენ გამომრიცხავ ან-ს (exclusive OR) ან 2-ის მოდულის შეკრებას, აღინიშნება სიმბოლოთი „^“. ამ ოპერაციის შედეგად რომელიმე თანრიგში მიღებული ბიტის მნიშვნელობა ტოლია 1-ის, თუ მხოლოდ ერთ-ერთი ოპერანდის შესაბამისი ბიტის მნიშვნელობა ტოლია 1-ის. ყველა სხვა შემთხვევაში 0-ის ტოლია. ანუ, თუ ოპერანდების შესაბამის თანრიგებში მნიშვნელობები განსხვავებულია,

შედეგის ბიტი ტოლია 1-ის, ხოლო თუ ერთნაირი მნიშვნელობები აქვთ, მაშინ შედეგია 0. მაგალითი:

```
   00101010       42
^  00001111       15
- - - - - - - - - - - - - - - - - -
   00100101       37
```

შემდეგ პროგრამაში დემონსტრირებულია ბიტური ლოგიკური ოპერაციები

ლისტინგი 12. ბიტური ოპერაციების დემონსტრირება

```
class BitLogic {
    public static void main(String args[]) {
     String binary [] = {
        "0000", "0001", "0010", "0011",
        "0100", "0101", "0110", "0111",
        "1000", "1001", "1010", "1011",
         "1100", "1101", "1110", "1111"
     };
     int a = 3; // 0011 ორობით წარმოდგენაში
     int b = 6; // 0110 ორობით წარმოდგენაში
     int c = a | b;
     int d = a & b;
     int e = a ^ b;
     int f = (~a & b) | (a & ~b);
     int g = ~a & 0x0f;
     System.out.println(" a = ¬"+ binary[a]);
     System.out.println(" b¬ = " + binary[b]);
     System.out.println(" a|b ¬ " + binary[c]);
     System.out.println(" a&b¬" + binary[d]);
     System.out.println(" a^b ¬ " + binary[e]);
     System.out.println("~a&b|a&~b = " + binary[f]);
```

```
        System.out.println(" ~a " + binary[g]);
    }
}
```

ამ მაგალითში სტრიქონული მასივი binary[] შეიცავს 0 ÷ 15 რიცხვების ორობით კითხვად წარმოდგენას. ამ მაგალითში მასივის ინდექსად ბიტური ოპერაციის შედეგის გამოყენებით შესაძლებელია ამ შედეგის ორობითი მნიშვნელობის დანახვა. მასივი ისეა აგებული, რომ n-ის ორობითი მნიშვნელობის სტრიქონული ფორმა ინახება მასივის binary[n] ელემენტად. ~a შედეგის binary მასივით გამოსატანად შედეგი 0x0F (00001111) რიცხვთან კონიუნქციით დაიყვანება 16-ზე ნაკლებ რიცხვამდე. ამ პროგრამის შედეგად კონსოლზე გამოვა:

a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
~a = 1100

მარცხნივ ძვრა

მარცხნივ ძვრის ოპერაცია << მნიშვნელობის ყველა ბიტს ძრავს მარცხნივ მითითებული პოზიციების რაოდენობით. მისი ზოგადი ფორმა ასეთია:

<მნიშვნელობა> << <რიცხვი>

<რიცხვი> აღნიშნავს იმ პოზიციების რაოდენობას, რამდენითაც უნდა მოხდეს <მნიშვნელობის> ბიტების მარცხნივ ძვრა. მარცხნივ ძვრისას უფროსი თანრიგი იძვრება დასაშვები დიაპაზონის გარეთ და ამიტომ იკარგება, ხოლო მარჯვნიდან ხდება 0-ებით შევსება. ეს ნიშნავს, რომ int ტიპის რიცხვზე მარცხნივ ძვრის ოპერაციის გამოყენებით ყველა ის ბიტი, რომელიც 31-ე პოზიციას გაცილდება, დაიკარგება. თუ ოპერანდის ტიპი long-ია, მაშინ 63-ე პოზიციის გაცილების შემთხვევაში დაიკარგება. გასათვალისწინებელია გამოსახულებებში ოპერანდების ტიპის ამაღლების ეფექტი. მაგალითად,:

ლისტინგი 13. byte-ის ტიპის ძვრა მარცხნივ

```
class ByteShift {
    public static void main(String args[]) {
     byte a = 64, b;
     int i;
     i=a << 2;
     b = (byte) (a << 2);
     System.out.println ("a-ს საწყისი მნიშვნელობაა: " + a);
     System.out.println ("i and b: " + i + " " + b);
    }
}
```

კონსოლზე გამოვა:

a-ს საწყისი მნიშვნელობაა: 64

i and b: 256 0

ვინაიდან ერთი თანრიგით მარცხნივ ძვრა ფაქტიურად აორმაგებს საწყის მნიშვნელობას, ამიტომ პროგრამისტები მას ხშირად, ეფექტურობის გამო, 2-ზე გამრავლების მაგივრად იყენებენ, მაგრამ ამ დროს სიფრთხილეა საჭირო, ვინაიდან

უფროს თანრიგში 1-ის მოხვედრით, რიცხვი უარყოფითი გახდება.

მარჯვნივ ძვრა

მარჯვნივ ძვრის ოპერაცია >> ყველა ბიტს ძრავს მარჯვნივ მითითებული პოზიციების რაოდენობით. ზოგადად ის ასე ჩაიწერება:

<მნიშვნელობა> >> <რიცხვი>

<რიცხვი> მიუთითებს იმ პოზიციების რაოდენობას რამდენითაც უნდა დაიძრას <მნიშვნელობის> თანრიგები. შემდეგ ფრაგმენტში 2 პოზიციით იძვრება რიცხვი 32, რის შედეგადაც a ცვლადის მნიშვნელობა გახდება 8:

```
int a = 35;
a = a >> 2;    // ახლა a-ს მნიშვნელობაა 8
```

განვიხილოთ ამ ოპერაციის შესრულება ორობით წარმოდგენაში

```
00100011    35
>> 2
00001000    8
```

ყოველი მარჯვნივ ძვრის დროს ხდება 2-ზე გაყოფა ნაშთის დაკარგვით. (წინა ოპერაციის ანალოგიურად მისი გამოყენება შესაძლებელია ეფექტური 2-ზე გაყოფის ოპერაციის განსახორციელებლად).

მარჯვნივ ძვრის ოპერაციისას უფროსი თანრიგები (მარცხენა პოზიციები) ძვრის შედეგად თავისუფლდება და ისინი შეივ-

სება უფროსი თანრიგის მნიშვნელობით. ამ ეფექტს უწოდებენ **ნიშნის თანრიგის გავრცელებას** და ამ შემთხვევაში ხდება უარყოფითი რიცხვის ნიშნის შენარჩუნება. მაგალითად, -8 >> 1 ტოლია -4, რაც ორობით წარმოდგენაში ასეა:

```
11111000    -8
>>1
11111100    -4
```

უცნაურია, მაგრამ -1 -ის მარჯვნივ ძვრა 1-ით ისევ -1 -ის ტოლია.

ზოგჯერ, მარჯვნივ ძვრისას არაა სასურველი ნიშნის თანრიგის დამატებითი თანრიგების გაჩენა. მაგალითად,:

ლისტინგი 14. ნიშნის დამატებითი თანრიგების მასკირება

```
class HexByte {
    static public void main(String args[]) {
     char hex [] = {
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
     };
     byte b = (byte) 0xf1;
     System.out.println("b = 0x" + hex[ (b >> 4) & 0x0f] +
                        hex[b & 0x0f]);
    }
}
```

ამ პროგრამის შედეგია:

b = 0xf1

მარჯვნივ ძვრა ნიშნის თანრიგის გაუთვალისწინებლად

ზოგჯერ არაა სასურველი ნიშნის თანრიგის გავრცელება მარჯვნივ ძვრის ოპერაციის დროს. ეს სიტუაცია ხშირად გვხვდება გრაფიკული ინფორმაციის ფიქსელებთან მუშაობის დროს. როგორც წესი, ამ დროს საჭიროა უფროსი თანრიგი შეივსოს 0-ით იმისდა მიუხედავად, რა ეწერა მასში მანამდე. ასეთ ოპერაციას უწოდებენ მარჯვნივ ძვრას ნიშნის გაუთვალისწინებლად. ამ ოპერაციის აღსანიშნავად გამოიყენება >>> სიმბოლოების მიმდევრობა და ასრულებს მარჯვნივ ძვრის ოპერაციას, ოღონდ უფროსი თანრიგის პოზიციაში ყოველთვის სვამს 0-ს. ეს ოპერაცია დემონსტრირებულია პროგრამის შემდეგ ფრაგმენტში:

```
int a = -1;
a = a >>> 24;
```

ამ მაგალითში a-ს მნიშვნელობის ორობით წარმოდგენაში ყველა 32 თანრიგში წერია 1. შემდეგ ხდება მათი 24 თანრიგით მარჯვნივ ძვრა ისე, რომ უფროს თანრიგებში ჩაიწერება 0. შედეგად a-ს მნიშვნელობა გახდება 255.

ოპერაციის შესრულების უკეთ გასაგებად, იგივე ოპერაცია ჩავწეროთ ორობითში:

```
11111111 11111111 11111111 11111111 //int ტიპის -1 ორობითში
>>>24
00000000 00000000 00000000 11111111 // int ტიპის 255 ორობითში
```

ხშირად >>> ოპერაცია იმდენად სასარგებლო არაა, ვინაიდან მას აზრი აქვს მხოლოდ 32 და 64 ბიტიანი მნიშვნელობებისთვის. უფრო ნაკლები ტიპების შემთხვევაში მათი გამო-

სახულებაში გამოყენებისას, როგორც ზემოთ აღვნიშნეთ, ხდება მათი ამაღლება 32 თანრიგიანამდე და ოპერაცია მაინც სრულდება 32 თანრიგის მიმართ და არა 8 ან 16 თანრიგის მიმართ.

ბიტური ავტოასოციური ოპერაციები

არითმეტიკული ოპერაციების მსგავსად, ყველა ბიტურ ოპერაციას აქვს ავტოასოციური ფორმაც, რომელიც აერთიანებს ძვრისა და მინიჭების ოპერაციებს. მაგალითად, შემდეგი ორი ოპერატორი, რომელიც ასრულებს 4 თანრიგით მარჯვნივ ძვრას ერთმანეთის ეკვივალენტურია:

```
a = a >> 4;
a >>=4;
```

ასევე ეკვივალენტურია შემდეგი ორ ოპერატორიც:

```
a = a | b;
a |= b;
```

შემდეგ პროგრამაში დემონსტრირებულია ავტოასოციური ოპერაციები:

ლისტინგი 15.

```
class OpBitEqua1s {
    public static void main(String args[]) {
     int a = 1;
     int b = 2;
     int c = 3;
     a |= 4;
     b >>= 1;
     c <<= 1;
     a ^= c;
```

```
        System.out.print1n("a = "+ a);
        System.out.print1n("b = " + b);
        System.out.print1n("c = " + c);
      }
}
```

ამ პროგრამის შედეგია:

a = 3

b = 1

c = 6

შედარების ოპერაციები

შედარების ოპერაციებით ხდება ოპერანდების მნიშვნელობები ტოლობისა და მათი ერთმანეთის მიმართ მეტობის ან ნაკლებობის განსაზღვრა. ეს ოპერაციები ჩამოთვლილია ცხრილში 9:

ცხრილი 9. შედარების ოპერაციები Java-ში

ოპერაცია	აღწერა
==	ტოლი
!=	არაა ტოლი
>	მეტი
<	ნაკლები
>=	მეტი ან ტოლი
<=	ნაკლები ან ტოლი

ამ ოპერაციების შესრულების შედეგი ყოველთვის boolean-ის ტიპისაა. შედარების ოპერაციებს ხშირად იყენებენ პირობით და ციკლის ოპერატორებში.

Java-ში შესაძლებელია ნებისმიერი ტიპის მნიშვნელობების შედარება == 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.

როგორც აღნიშნული იყო, შედარების ოპერაციის შედეგი boolean-ის ტიპისაა, ამიტომ დასაშვებია ასეთი ფრაგმენტი:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

მიაქციეთ ყურადღება, რომ ასეთი ჩანაწერი

```
int done;
// ......
if(!done) . . .          // დასაშვებია c/c++ ში, მაგრამ
if(done) . . .      // არაა დასაშვები Java-ში
```

Java-ში ეს ოპერატორები ასე იწერება:

```
if(done == 0) . . .   //  ესაა Java-ს სტილი
if(done != 0) . . .
```

ბულის ლოგიკური ოპერაციები

ბულის ლოგიკის ოპერაციები სრულდება მხოლოდ boolean-ის ტიპის ოპერანდებზე. ბულის ლოგიკის ოპერაციები ჩამოთვლილია ცხრილში 10.

ცხრილი 10. ბულის ლოგიკური ოპერაციები Java-ში

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)
==	ტოლი
!=	არაა ტოლი
?:	ტერნერული ოპერაცია if-then-else

ბულის ლოგიკის ოპერაციები &, | და ^ ბულის ტიპის მნიშვნელობებზე მოქმედებენ ისეთნაირედ, როგორც მთელრიცხვა მნიშვნელობების ბიტებზე. ბულის ოპერაცია ! ინვერტირებას უკეთებს ბულის მდგომარეობას: !true == false და !false == true. თითოეული ლოგიკური ოპერაციის შედეგი ნაჩვენებია ცხრილში 11.

ცხრილი 11. ლოგიკურები ოპერაციები

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

ქვემოთ ნაჩვენებია პროგრამის ფრაგმენტი, რომელიც პრაქტიკულად იმავე მოქმედებებს აკეთებს, რასაც ზემოთ მოტანილი პროგრამა BitLogic, ოღონდ იმ განსხვავებით, რომ აქ მოქმედებები ხდება boolean-ის ტიპის მნიშვნელობებზე და არა ბიტურ მნიშვნელობებზე.

ლისტინგი 16. ბულის ლოგიკური ოპერაციების დემონსტრირება

```
class BoolLogic {
    public static void main (String args [] ) {
     boolean a = true;
     boolean b = false;
     boolean c = a | b;
     boolean d = a & b;
     boolean e = a ^ b;
     boolean f = (!a&b) | (a& !b);
     boolean g = !a;
       System.out.println(" a = " + a);
       System.out.println(" b = " + b);
       System.out.println(" a | b  " + c);
       System.out.println(" a & b " + d);
       System.out.println(" a ^ b "+ e);
       System.out.println("!a&b | a&!b = " + f);
       System.out.println(" !a =" + g);
    }  }
```

მოკლე ლოგიკური ოპერაციები

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. მაგალითად,

```
if(denom != 0 && num / denom > 10)
```

მოკლე ფორმის (&&) გამოყენების წყალობით გამოირიცხება განსაკუთრებული სიტუაციის წარმოქმნის რისკი, როდესაც denom იქნება 0-ის ტოლი.

AND და OR ოპერაციების მოკლე ფორმა უფრო ხშირად გამოიყენება ბულის ლოგიკის დროს, ხოლო ერთსიმბოლოიანი ვერსიები ბიტური ოპერაციების დროს, თუმცა არსებობს გამონაკლისებიც. მაგალითად,:

```
if(e == 1 & e++ < 100) d = 100;
```

ამ შემთხვევაში & ერთმაგი სიმბოლო უზრუნველყოფს, რომ e ცვლადის მიმართ ინკრემენტის ოპერაცია აუცილებლად შესრულდება იმისდა მიუხედავად ტოლია თუ არა ის 1-ის.

მინიჭების ოპერაცია

ამ ოპერაციას მაგალითებში ჩვენ ხშირად ვიყენებდით, ახლა განვიხილოთ ის ფორმალურად. მინიჭების ოპერაციის აღსანიშნავად გამოიყენება სიმბოლო =. ამ ოპერაციას ასეთი ზოგადი ფორმა აქვს:

<ცვლადი> = <გამოსახულება>;

ამ ოპერატორში <ცვლადი>-ის ტიპი უნდა შეესაბამებოდეს <გამოსახულების> ტიპს.

მინიჭების ოპერაციით შესაძლებელია მინიჭების ოპერაციების ჯაჭვი განვახორციელოთ. მაგალითად,:

```
int x, y, z;
x = y = z = 100;      // x,y,z ცვლადების მნიშვნელობები გახდება 100
```

ეს ხდება იმიტომ, რომ = ოპერციის შედეგია მარჯვენა გამოსახულების მნიშვნელობა, ანუ z = 100 -ის მნიშვნელობაა 100, რომელიც შემდეგ მიენიჭება y-ს, ხოლო შემდეგ ანალოგიური მსჯელობით x-ს. ეს ფორმა საკმაოდ მოსახერხებელია რამდენიმე ცვლადისათვის ერთი და იგივე მნიშვნელობის მისანიჭებლად.

ტერნერული ოპერაცია

Java-ში რეალიზებულია სპეციალური სამ ოპერანდიანი ოპერაცია, რომელმაც გარკვეულწილად შეიძლება ჩაანაცვლოს ზოგჯერ If-then-else ოპერატორი. ესაა ოპერაცია „?:“. ამ ოპერაციის ზოგადი ფორმა ასე შეიძლება წარმოვადგინოთ:

<გამოსახულება1> ? <გამოსახულება2> : <გამოსახულება3>

აქ <გამოსახულება1> ნებისმიერი boolean-ის ტიპის შედეგის მომცემი გამოსახულებაა. თუ <გამოსახულება1>-ის მნიშვნელობა ტოლია true, მაშინ მოხდება <გამოსახულება2>-ის გამოთვლა, წინააღმდეგ შემთხვევაში გამოითვლება <გამოსახულება3>. „?:“ ოპერაციის შედეგი სწორედ გამოთვლილი გამოსახულების მნიშვნელობაა. <გამოსახულება2> და <გამოსახულება3> უნდა აბრუნებდნენ ერთიდაიგივე ტიპის მნიშვნელობებს და არ შეიძლება მისი ტიპი void იყოს. მაგალითი:

ratio = denom == 0 ? 0 : num /demon;

ამ მინიჭების ოპერატორის შესრულებისას, თავიდან მოწმდება კითხვის ნიშნამდე („?“) ჩაწერილი გამოსახულება. თუ denom-ის მნიშვნელობა 0-ის ტოლია (ანუ გამოსახულების მნიშვნელობაა true), მაშინ გამოითვლება „?“ ნიშანსა და “:“ ნიშანს შორის ჩაწერილი გამოსახულება და ამ გამოსახულების მნიშვნელობა იქნება მთლიანი „?:“ ოპერაციის მნიშვნელობა. თუ denom-ის მნიშვნელობა არაა 0-ის ტოლი (ანუ გამოსახულების მნიშვნელობაა false), მაშინ გამოითვლება „:“ სიმბოლოს შემდეგ ჩაწერილი გამოსახულება და მისი მნიშვნელობა აიღება მთლიანი „?:“

ოპერაციის მნიშვნელობად. შემდეგ, „?:“ ოპერაციის მნიშვნელობა მიენიჭება ratio-ს.

შემდეგ პროგრამაში ხდება ამ ოპერაციის დემონსტრირება:

ლისტინგი 17. ? : ოპერაციის დემონსტრირება

```
class Ternary {
    public static void main(String args[]) {
        int i, k;
        i = 10;
        k = i < 0 ?- i : i;         // i ცვლადის აბსოლუტური მნიშვნელობა
        System.out.print("აბსოლუტური მნიშვნელობა ");
        System.out.println(i +"-ის ტოლია " + k);
        i = -10;
        k = i < 0 ? -i : i;   // i ცვლადის აბსოლუტური მნიშვნელობა
        System.out.print ("აბსოლუტური მნიშვნელობა ");
        System.out.println (i + "-ის ტოლია " + k);
    }
}
```

ამ პროგრამის შედეგად კონსოლზე გამოვა:

აბსოლუტური მნიშვნელობა 10-ის ტოლია 10
აბსოლუტური მნიშვნელობა -10-ის ტოლია 10

ოპერაციების პრიორიტეტი

მე-12 ცხრილში ნაჩვენებია ოპერაციების პრიორიტეტები მაღალიდან დაბლისაკენ. პირველ სტრიქონში მითითებულია გამყოფები, რომლებიც, მართალია, არაა ოპერაციები, მაგრამ გამოსახულებებში ხშირად მონაწილეობენ და ოპერაციის მსგავსად მოქმედებენ. მრგვალი ფრჩხილები გამოიყენება ოპერაციების პრიორიტეტის შესაცვლელად,

ხოლო კვადრატული ფრჩხილები მასივების ინდექსირებისათვის.

მრგვალი ფრჩხილების გამოყენება უმატებს ოპერაციების პრიორიტეტს. ხშირად მათი გამოყენება აუცილებელია საჭირო შედეგის მისაღებად. მაგალითად,:

a >> b + 3

ცხრილი 12. ოპერაციათა პრიორიტეტები Java-ში

მაღალი პრიორიტეტი			
()	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
დაბალი პრიორიტეტი			

ამ გამოსახულებაში ჯერ 3 დაემატება b-ს, შემდეგ მიღებული მნიშვნელობის შესაბამისი თანრიგების რაოდენობით a-ს

მნიშვნელობა დაიძვრება მარჯვნივ. ჭარბი ფრჩხილების გამოყენებით იგივე გამოსახულება ასე ჩაიწერება:

```
a >> (b + 3)
```

თუ თავიდან საჭიროა a ცვლადის მნიშვნელობის დაძვრა b რაოდენობის პოზიციით და შემდეგ შედეგზე 3-ის დამატება, მაშინ საჭიროა ფრჩხილების ასეთნაირი გამოყენება:

```
(a >> b) + 3
```

ზოგჯერ ფრჩხილები გამოიყენება რთულად გასაგები გამოსახულების აღქმის გასაადვილებლად. მაგალითად,:

```
a | 4 + c >> b & 7
(a | (((4 + c) >> b) & 7))
```

ჭარბი ან არაჭარბი მრგვალი ფრჩხილების გამოყენება არ იწვევს პროგრამის შესრულების ეფექტურობის შემცირებას. ამიტომ, კითხვადობის ამაღლების მიზნით ფრჩხილების დამატება, პროგრამის შესრულებაზე უარყოფითად არ მოქმედებს.

თავი II. მართვის ოპერატორები

პროგრამირების ენებში მმართავი ოპერატორები გამოიყენება პროგრამის ბრძანებების მიმდევრობის შესრულების შესაცვლელად, გადასვლებისა და განშტოებების განსახორციელებლად. Java პროგრამაში მმართავი ოპერატორები შეიძლება დაიყოს შემდეგ კატეგორიებად: არჩევის ოპერატორები (if-else, switch), ციკლის ოპერატორები (while, do while, for, for-each) და გადასვლის ოპერატორები (continue, break, return). **არჩევის** ოპერატორები საშუალებას იძლევა გამოსახულების მნიშვნელობის ან ცვლადის მდგომარეობის მიხედვით პროგრამაში არჩეული იქნეს ბრძანებების შესრულების სხვადასხვა განშტოება. **ციკლის** ოპერატორები საშუალებას იძლევა განმეორდეს პროგრამაში ერთი ან რამდენიმე ოპერატორების შესრულება. **გადასვლის** ოპერატორებით შესაძლებელია პროგრამის არაწრფივად შესრულება. განვიხილოთ ყველა ეს ოპერატორები.

ოპერატორი if

if ოპერატორი - ესაა Java პროგრამის შტოების პირობითი შერჩევის ოპერატორი. ის გამოიყენება პროგრამის შესრულების სამართავად ორი სხვადასხვა მიმართულების შტოზე. ამ ოპერატორის ჩაწერის ზოგადი ფორმაა:

if(<პირობა>) <ოპერატორი1>;
[else <ოპერატორი2>;]

აქ თითოეული <ოპერატორი> ესაა ან რომელიმე ერთი ოპერატორი ან ერთ ბლოკში (ფიგურულ ფრჩხილებში) გაერთიანებული რამდენიმე ოპერატორი. <პირობა> ესაა ნებისმიერი გამოსახულება, რომელიც boolean-ის ტიპის შედეგს აბრუნებს. else აუცილებელი არაა.

if ოპერატორი ასე მუშაობს: 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. მაგალითად,:

```
int a=b=1;
/ / ...
if(a < b) a = 0 ;
else b = 0;
```

ამ შემთხვევაში თუ a-ს მნიშვნელობა ნაკლებია b-ს მნიშვნელობაზე, a ცვლადის მნიშვნელობა გახდება 0. წინააღმდეგ შემთხვევაში b-ს მნიშვნელობა იქნება 0-ის ტოლი. არცერთ შემთხვევაში არ მიიღებენ ორივე ცვლადი 0-ის ტოლ მნიშვნელობებს.

ხშირად if ოპერატორის მმართავ გამოსახულებაში გამოიყენება შედარების ოპერაციები, თუმცა ეს არაა აუცილებელი. შესაძლებელია if ოპერატორის მმართავ გამოსახულებაში გამოყენებული იქნეს boolean-ის ტიპის ცვლადიც:

```
boolean dataAvailable;
/ / ...
if (dataAvailable)
```

```
    ProcessData();
else
    waitForMoreData();
```

ყურადღება უნდა გავამახვილოთ, რომ if ან else საკვანძო სიტყვების შემდეგ შეიძლება მოდიოდეს, მხოლოდ ერთი ოპერატორი. თუ საჭიროა მეტი ოპერატორების მითითება, მაშინ ისინი უნდა ჩავსვათ ერთ ბლოკში:

```
int bytesAvailable;
/ / ...
if (bytesAvailable > 0) {
    ProcessData () ;
    bytesAvailable -= n;
} else
waitForMoreData();
```

ამ მაგალითში, ბლოკში მოთავსებული ორივე ოპერატორი შესრულდება თუ bytesAvailable-ს მნიშვნელობა მეტია 0-ზე.

ზოგიერთი პროგრამისტი ფიგურულ ფრჩხილებს იყენებს ერთი ოპერატორის დროსაც კი. ეს აადვილებს შემდგომში ოპერატორების დამატებას. ძალიან გავრცელებული შეცდომის ტიპია ბლოკის განსაზღვრის გამორჩენა, როდესაც ის საჭიროა. მაგალითად,:

```
int bytesAvailable;
/ / ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
```

```
} else
    waitForMoreData();
    bytesAvailable = n;
```

თუ გავითვალისწინებთ, რომ კოდში bytesAvailable = n; ოპერატორი შეწეულია, ალბათ იგულისხმება, რომ იგი უნდა სრულდებოდეს else გამოსახულების შიგნით. მაგრამ, Java პროგრამაში ცარიელ სიმბოლოებს მნიშვნელობა არ აქვს და კომპილატორი ვერაფრით ვერ „გაიგებს“ პროგრამისტის ჩანაფიქრს. ამ ფრაგმენტის კომპილაცია ჩაივლის ყოველგვარი შენიშვნის გარეშე, მაგრამ შესრულებისას პროგრამა არ მოიქცევა ისე როგორც იყო ჩაფიქრებული. შემდეგ ფრაგმენტში გასწორებულია წინა შეცდომა:

```
int bytesAvailable;
/ / ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
} else {
    waitForMoreData();
    bytesAvailable = n;
}
```

ჩალაგებული if ოპერატორები

პროგრამებში ხშირად გვხვდება ერთმანეთში ჩალაგებული if ოპერატორები. ამ შემთხვევაში უნდა გვახსოვდეს, რომ else ოპერატორი ყოველთვის უკავშირდება მის უახლოეს if ოპე-

რატორს, 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)-სთან
```

როგორც კომენტარებიდანაც ჩანს, ბოლო else ოპერატორი არაა 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.

მრავალრგოლიანი სტრუქტურა if - else-if

პროგრამირებაში გავრცელებულია კონსტრუქცია, რომელიც აიგება ჩალაგებული if ოპერატორების მიმდევრობისაგან, მას უწოდებენ მრავალრგოლიან if – else-if სტრუქტურას და ის ასე გამოიყურება:

```
if (<პირობა>)
    <ოპერატორი>;
else if (<პირობა>)
    <ოპერატორი>;
else if (<პირობა>)
```

```
    <ოპერატორი>;
else
    <ოპერატორი>;
```

if ოპერატორი სრულდება მიმდევრობით, ზემოდან ქვემოთ. როგორც კი if ოპერატორის მმართავი ერთერთი პირობა გახდება true, პროგრამა შეასრულებს ამ if ოპერატორთან დაკავშირებულ ოპერატორს და გამოტოვებს დანარჩენ მრავალრგოლიან სტრუქტურის ნაწილს. თუ არცერთი პირობა არ სრულდება (არაა true), პროგრამა შეასრულებს ბოლო, else ოპერატორს. ანუ, თუ ყველა სხვა შემოწმების პირობა იძლევა უარყოფით შედეგს, პროგრამა ასრულებს ბოლო else ოპერატორს. თუ ბოლო else ოპერატორი არაა მითითებული და ყველა წინა შემოწმების შედეგია false, მაშინ პროგრამა ამ კონსტრუქციიდან არც ერთ მოქმედებას არ შეასრულებს.

ქვემოთ განხილულია მაგალითი, სადაც მრავალრგოლიანი სტრუქტურა if – else if გამოყენებულია წლის იმ დროის გამოსათვლელად, რომელსაც მიეკუთვნება კონკრეტული თვე:

ლისტინგი 18. if – else if ოპერატორების გამოყენების დემონსტრირება

```
class IfE1se {
    public static void main(String args[]) {
        int month = 4;       // აპრილი
        String season;
        if (month == 12 || month == 1 || month == 2)
            season = "ზამთარი";
        else if (month == 3 || month == 4 || month == 5)
```

```
            season = "გაზაფხული";
        else if (month == 6 || month == 7 || month == 8)
            season = "ზაფხული";
        else if (month == 9 || month == 10 || month == 11)
            season = "შემოდგომა";
        else
            season = "არარსებული თვე";
        System.out.println("აპრილი არის " + season + ".");
    }
}
```

პროგრამა კონსოლზე გამოიტანს ასეთ შედეგს:

აპრილი არის გაზაფხული.

არჩევის ოპერატორი switch

ამ ოპერატორით ხდება პროგრამის შესრულების განშტოება. განშტოება ხდება მმართავი გამოსახულების გამოთვლის შედეგად მიღებული მნიშვნელობის მიხედვით. ხშირად მისი გამოყენება უფრო ეფექტურია ხოლმე, ვიდრე გრძელი if – else-if ოპერატორების მიმდევრობა. switch ოპერატორის ზოგადი ფორმა ასეთია:

```
switch (<გამოსახულება>) {
case <მნიშვნელობა1>:
// ოპერატორების მიმდევრობა
break;
case <მნიშვნელობა2>:
// ოპერატორების მიმდევრობა
break;
case <მნიშვნელობაN>:
```

```
// ოპერატორების მიმდევრობა
break;
defau1t:
// ოპერატორები
}
```

<გამოსახულება> უნდა იყოს byte, short, int ან char ტიპის. <მნიშვნელობა>-ში მითითებული ყოველი ტიპი უნდა იყოს <გამოსახულება>-ში მითითებული ტიპის თანადი (თავსებადი) ტიპი. Java 5-ში (JDK5-ში) switch ოპერატორის სამართავად შესაძლებელია enum ჩამოთვლადი ტიპების გამოყენება, ხოლო Java 7-ში (JDK7-ში) და უფრო ახალ ვერსიებში შესაძლებელია String ტიპის გამოყენებაც. case-ს თითოეული <მნიშვნელობა> უნდა იყოს უნიკალური კონსტანტა (ანუ მუდმივა და არა ცვლადი). ასევე არაა დაშვებული case-ის მნიშვნელობების დუბლირება.

switch ოპერატორი მუშაობს ასე: <გამოსახულება>-ის მნიშვნელობა დარდება case ოპერატორებში მითითებულ თითოეულ კონსტანტას. თუ მოხდება რომელიმესთან დამთხვევა, მაშინ შესრულება გრძელდება ამ case-ს შემდეგ ჩაწერილი ოპერატორიდან. თუ არც ერთი კონსტანტის მნიშვნელობა არ ემთხვევა <გამოსახულების> მნიშვნელობას, მაშინ პროგრამა ასრულებს default ოპერატორის შემდეგ ჩაწერილ ოპერატორებს. თუმცა, ეს ოპერატორი აუცილებელი არაა. ამ ოპერატორის არარსებობის შემთხვევაში და გამოსახულების არცერთ კონსტანტაზე დამთხვევის შემთხვევაში, პროგრამა switch კონსტრუქციაში არაფერს არ ასრულებს. პროგრამის შესრულება გაგრძელდება 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 ანხორციელებს switch ოპერატორიდან „გამოსვლას“. მაგალითი:

ლისტინგი 19. ოპერატორ switch -ის დემონსტრირება

```
class SampleSwitch {
    public static void main(String args[]) {
        int i = 3;
        switch (i) {
        case 0:
            System.out.println("i  ტოლია 0-ის");
            break;
        case 1:
            System.out.println("i ტოლია 1-ის");
            break;
        case 2:
            System.out.println("i ტოლია 2-ის");
            break;
        case 3:
            System.out.println("i ტოლია 3-ის");
            break;
        default:
        System.out.println("i მეტია 3-ზე");
        }
    }
}
```

პროგრამის შესრულების შედეგი:

i ტოლია 3-ის

break ოპერატორი აუცილებელი არაა, თუ მას არ დავწერთ, პროგრამა გააგრძელებს შესრულებას შემდეგი case ოპერატორიდან და ა.შ. ზოგჯერ საჭიროა რამდენიმე case ოპერატორის გამოყენება break ოპერატორით გამოყოფის გარეშე. მაგალითად,:

ლისტინგი 20.

```
class Switch {
    public static void main(String args[]) {
        int month = 4;
        String season;
        switch (month) {
        case 12:
        case 1:
        case 2:
            season = "ზამთარი";
            break;
        case 3:
        case 4:
        case 5:
            season = "გაზაფხული";
            break;
        case 6:
        case 7:
        case 8:
            season = "ზაფხული";
            break;
      case 9:
      case 10:
      case 11:
            season = "შემოდგომა";
            break;
        default:
            season = "არარსებული თვე";
```

```
        }
        System.out.println("აპრილის თვე არის " + season + ".");
    }
}
```

სტრიქონების გამოყენება switch ოპერატორში

Java 7-ში (JDK 7) და მის შემდეგ ვერსიებში, შესაძლებელია String ობიექტის გამოყენება switch ოპერატორში. შემდეგ მაგალითში StringSwitchDemo გამოითვლება თვის ნომერი String ტიპის ცვლადის month-ის გამოყენებით:

ლისტინგი 21.

```
class StringSwitchDemo {
    public static void main(String[] args) {
        String month = "აგვისტო";
        int monthNumber = 0;
        switch (month) {
          case "იანვარი":       monthNumber =  1; break;
          case "თებერვალი":monthNumber =  2; break;
          case "მარტი":         monthNumber =  3; break;
          case "აპრილი":     monthNumber =  4; break;
          case "მაისი":          monthNumber =  5; break;
          case "ივნისი":        monthNumber =  6; break;
          case "ივლისი":     monthNumber =  7; break;
          case "აგვისტო":    monthNumber =  8; break;
          case "სექტემბერი":monthNumber =  9; break;
          case "ოქტომბერი":monthNumber = 10; break;
          case "ნოემბერი":   monthNumber = 11; break;
          case "დეკემბერი":   monthNumber = 12; break;
          default:                 monthNumber =  0; break;
        }
```

```
        if (monthNumber == 0) {
            System.out.println("არასწორი თვე");
        } else {
            System.out.println(monthNumber);
        }
    }
}
```

პროგრამის შესრულების შედეგი:

8

ჩალაგებული switch ოპერატორი

switch ოპერატორი შეიძლება გამოყენებული იქნეს რომელიმე გარე ოპერატორის (მათ შორის switch ოპერატორის) შიგა ოპერატორების მიმდევრობაში. ასეთ ოპერატორს უწოდებენ ჩალაგებულ switch ოპერატორს. ვინაიდან switch ოპერატორი თავის ბლოკს ქმნის, რაიმე კონფლიქტი გარე და შიგა case კონსტანტებს შორის არ წარმოიშვება. მაგალითად,:

ლისტინგი 22.

```
switch(count) {
    case 1:
        switch(target) { // ჩადგმული switch ოპერატორი
            case 0:
                System.out.print1n("target ტოლია 0-ის");
                break;
            case 1: // კონფლიქტი გარე switch-თან არ გვაქვს
                System.out.print1n("target ტოლია 1-ის");
                break;
```

```
    }
  break;
  case 2:    // . . . .
```

ამ შემთხვევაში შიგა switch ოპერატორის case 1: ოპერატორსა და გარე switch ოპერატორის case 1: ოპერატორს შორის რაიმე კონფლიქტი არ არსებობს. პროგრამა ადარებს count ცვლადის მნიშვნელობას, მხოლოდ გარე დონეზე მყოფი case ოპერატორის შტოების კონსტანტებთან. თუ count ცვლადის მნიშვნელობა 1-ის ტოლია, პროგრამა ადარებს target ცვლადის მნიშვნელობას შიგა case შტოების კონსტანტებთან.

შედეგის სახით შეიძლება გამოვყოთ switch ოპერატორის სამი მნიშვნელოვანი თვისება:

- switch ოპერატორი if ოპერატორისაგან განსხვავდება იმით, რომ იგი ასრულებს მხოლოდ ტოლობაზე შემოწმებას, ხოლო if ოპერატორში შესაძლებელია ნებისმიერი ბულის ტიპის გამოსახულების გამოყენება. ანუ, switch ოპერატორი ეძებს მხოლოდ შესაბამისობას გამოსახულების მნიშვნელობასა და case-ის ერთ-ერთ კონსტანტას შორის;
- ერთ switch ოპერატორში არც ერთი case კონსტანტა ერთმანეთს არ უნდა ემთხვეოდეს;
- როგორც წესი, switch ოპერატორი უფრო ეფექტურია ვიდრე ჩალაგებული if ოპერატორები.

ციკლის ოპერატორები

Java-ში არსებობს შემდეგი ციკლის ოპერატორები: while, do - while და for. ამ ოპერატორებით იქმნება კონსტრუქციები,

რომლებსაც უწოდებენ ციკლებს. ციკლები მრავალჯერადად ასრულებენ ერთი და იგივე ბრძანებების მიმდევრობას მანამ, სანამ არ დაირღვევა ციკლის გაგრძელების პირობა.

while ციკლი

while ოპერატორი პროგრამებში ხშირად გამოიყენება. იგი იმეორებს ოპერატორს ან ოპერატორების ბლოკს მანამ, სანამ მისი მმართავი გამოსახულება ჭეშმარიტია. მას აქვს შემდეგი ფორმა:

```
while (<პირობა>) {
    // ციკლის ტანი
}
```

<პირობა> შეიძლება იყოს ნებისმიერი ბულის ტიპის გამოსახულება. ციკლის ტანი სრულდება სანამ პირობითი გამოსახულება ჭეშმარიტია. როდესაც პირობა გახდება მცდარი შესრულება გაგრძელდება ციკლის უშუალოდ მომდევნო ოპერატორისაგან. ფიგურული ფრჩხილები შეიძლება არ იქნეს გამოყენებული თუ ციკლის ტანი მხოლოდ ერთი ოპერატორისაგან შედგება. მაგალითი:

ლისტინგი 23. while ციკლის დემონსტრირება.

```
public class Aaaaaaaa {
    public static void main(String[] args) {
     int n = 10;
     while (n > 0) {
        System.out.println("ტაქტი " + n);
        n--;
     }
    }
```

```
}
```

პროგრამის შესრულების შედეგი:

ტაქტი 10
ტაქტი 9
ტაქტი 8
ტაქტი 7
ტაქტი 6
ტაქტი 5
ტაქტი 4
ტაქტი 3
ტაქტი 2
ტაქტი 1

ვინაიდან while ციკლი თავის პირობით გამოსახულებას გამოითვლის ციკლის დასაწყისში, ციკლის ტანი საერთოდ, ერთხელაც არ შესრულდება, თუ ციკლის პირობა თავიდანვე მცდარი იქნება. მაგალითად,:

```
int a = 10, b = 20;
while (a > b)
System.out.println("ეს სტრიქონი არ გამოჩნდება!");
```

while ციკლის ტანი ან Java-ს ნებისმიერი ციკლის ტანი შესაძლებელია იყოს ცარიელი. ეს განპირობებულია იმით, რომ ენის სინტაქსი დასაშვებად თვლის ნულოვან (ცარიელ) ოპერატორს, რომელიც მხოლოდ წერტილ-მძიმეს უნდა შეიცავდეს, მაგალითად,:

ლისტინგი 24. ციკლის ტანი შეიძლება ცარიელი იყოს

```
class NoBody {
    public static void main(String args[]) {
        int i, j;
        i = 100;
        j = 200;
        // i და j ცვლადების საშუალო მნიშვნელობა
        while(++i < --j) ;    // ციკლის ტანი ცარიელია
        System.out.println("საშუალო მნიშვნელობა ტოლია " + i);
    }
}
```

პროგრამის შესრულების შედეგი:

საშუალო მნიშვნელობა ტოლია 150

ეს პროცედურა იმუშავებს მხოლოდ იმ შემთხვევაში, როდესაც დასაწყისში i-ს მნიშვნელობა ნაკლებია j-ს მნიშვნელობაზე. ციკლის ორგანიზებისათვის ყველანაირი მოქმედება სრულდება პირობითი გამოსახულების შიგნით.

do-while ციკლი

როგორც წინა ოპერატორის განხილვისას ვნახეთ, თუ დასაწყისში პირობითი გამოსახულების მნიშვნელობა false-ს ტოლია, ციკლის ტანი არცერთხელ არ შესრულდება. ზოგჯერ საჭიროა, რომ ციკლის ტანი ერთხელ მაინც შესრულდეს, მაშინაც კი როდესაც ციკლის პირობა არ სრულდება. ანუ, ზოგჯერ ციკლის პირობის შემოწმება მოსახერხებელია მოხდეს ციკლის ტანის ბოლოს და არა ციკლის დასაწყისში. java-ში ასეთ მოთხოვნას ახორციელებს ოპერატორი do-while. მისი ზოგადი ფორმა ასეთია:

```
do {
    // ციკლის ტანი
} while (<პირობა>);
```

do-while ციკლის ყოველი განმეორების დროს პროგრამა ჯერ ასრულებს ციკლის ტანს, ხოლო შემდეგ სრულდება პირობითი გამოსახულება. თუ ეს პირობა ჭეშმარიტია, ციკლი მეორდება. წინააღმდეგ შემთხვევაში ციკლის შესრულება წყდება. როგორც ყველა Java ციკლში, <პირობის> შედეგი უნდა იყოს ბულის ტიპის. ქვემოთ ნაჩვენებია ტაქტების გამოტანის შეცვლილი პროგრამა, სადაც დემონსტრირებულია do-while ციკლის მუშაობა:

ლისტინგი 25.

```
class DoWhi1e {
    public static void main(String args[]) {
        int n = 10;
        do {
            System.out.println("ტაქტი" + n);
            n--;
        } while (n > 0);
    }
}
```

მართალია ეს პროგრამა ტექნიკური თვალსაზრისით სწორადაა ჩაწერილი, მაგრამ შესაძლებელია მისი უფრო ეფექტურად გადაკეთება:

```
do {
    System.out.print1n("ტაქტი" + n);
} whi1e(--n > 0);
```

პროგრამა ასე იმუშავებს: ჯერ შესრულდება --n დეკრემენტის ოპერაცია, ანუ n -ის მნიშვნელობა შემცირდება ერთით და დაბრუნდება ეს ახალი მნიშვნელობა. შემდეგ ეს მნიშვნელობა შედარდება 0-ს. თუ იგი 0-ზე მეტია, ციკლის შესრულება გაგრძელდება, წინააღმდეგ შემთხვევაში ციკლი მთავრდება.

for ციკლი

დაწყებული JDK5 -დან, Java-ში არსებობს for ციკლის ორი ფორმა. პირველი - ტრადიციული ფორმა, რომელიც თავიდანვე იქნა რეალიზებული ენაში. მეორე - ახალი ფორმა "for-each".

ტრადიცული for ციკლის ოპერატორის ზოგადი ფორმას ასეთი სახე აქვს:

```
for(<ინიციალიზაცია>; <პირობა>; <იტერაცია>) {
    // ციკლის ტანი
}
```

თუ ციკლში მხოლოდ ერთი ოპერატორის განმეორება ხდება, შესაძლებელია ფიგურული ფრჩხილების გამოტოვება.

for ციკლი მოქმედებს შემდეგნაირად: ციკლის პირველ გაშვებაზე პროგრამა ასრულებს ციკლის ინიციალიზაციის ნაწილს. ზოგადად ესაა გამოსახულება, რომელიც საწყის მნიშვნელობას ანიჭებს ციკლის მმართავ ცვლადს, რომელიც ციკლში მთვლელის როლს ასრულებს. ხაზგასასმელია, რომ ინიციალიზაციის გამოსახულების შესრულება ხდება მხოლოდ ერთხელ! შემდეგ პროგრამა გამოითვლის <პირობას>,

რომელიც ბულის ტიპის გამოსახულება უნდა იყოს. როგორც წესი, მმართავი ცვლადი დარდება მიზნობივ მნიშვნელობას. თუ ბულის ტიპის გამოსახულება ჭეშმარიტია, პროგრამა ასრულებს ციკლის ტანს, ხოლო თუ ყალბია, 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 მმართავი ცვლადის 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 ყალბი.

ზემოთ განხილული მაგალითი for ციკლით ასე ჩაიწერება:

ლისტინგი 26. for ციკლის გამოყენების დემონსტრირება

```
class ForTick {
    public static void main(String args[]) {
        int n;
        for(n=10; n>0; n--)
          System.out.println("ტაქტი " + n);
    }
}
```

მმართავი ცვლადის გამოცხადება for ციკლის შიგნით

ხშირად ცვლადი, რომელიც for ციკლის მართვას ანხორციელებს, მხოლოდ ამ ციკლისათვისაა საჭირო და სხვაგან არსად არ გამოიყენება. ასეთ შემთხვევაში შესაძლებელია ცვლადი გამოვაცხადოთ for ციკლის ოპერატორის ინი-

ციალიზაციის ნაწილში. მაგალითად, წინა პროგრამა ასე შეიძლება გადავწეროთ:

ლისტინგი 27. მმართავი ცვლადის გამოცხადება ციკლში

```
class ForTick {
    public static void main(String args[]) {
        for(int n=10; n>0; n--)
        System.out.println("ტაქტი" + n);
    }
}
```

ცვლადის ციკლის ტანის შიგნით გამოცხადების შემთხვევაში უნდა გვახსოვდეს, რომ ამ ცვლადის განსაზღვრის არე და არსებობის დრო მთლიანად ემთხვევა for ციკლის არესა და არსებობის დროს. ანუ ცვლადის განსაზღვრის არე შემოფარგლულია for ციკლით. for ციკლის გარეთ ცვლადი წყვეტს არსებობას. თუ მმართავი ცვლადი პროგრამის სხვა ადგილებშიცაა საჭირო, ის ციკლის შიგნით არ უნდა გამოცხადდეს.

განვიხილოთ პროგრამის მაგალითი, რომელიც ამოწმებს არის თუ არა რაიმე რიცხვი მარტივი:

ლისტინგი 28. მარტივი რიცხვის განსაზღვრა

```
class FindPrime {
    public static void main(String args[]) {
        int num;
        boolean isPrime = true;
        num  = 14;
        for(int i=2; i < num/i; i++) {
        if((num % i) == 0) {
            isPrime = false;
            break;
        }
```

```
        }
        if ( isPrime ) System.out.println ("მარტივია");
        else System.out.println("არაა მარტივი");
    }
}
```

მძიმის გამოყენება

ზოგჯერ for ციკლის ინიციალიზაციისა და იტერაციის ნაწილებში საჭიროა რამდენიმე ოპერატორის შესრულება. მაგალითი:

ლისტინგი 29.

```
class Sample {
    public static void main(String args[]) {
        int a, b;
        b = 4;
        for(a=1; a<b; a++) {
          System.out.println("a = " + a);
          System.out.println("b = " + b);
          b--;
        }
    }
}
```

როგორც ამ მაგალითიდან ჩანს, ციკლის მართვა ხდება ორი ცვლადის საშუალებით. სასურველია, რომ ორივე ცვლადის მითითება შესაძლებელი იყოს ციკლის ტანში და არ იყოს საჭირო b ცვლადის ხელით შემცირება. Java გვთავაზობს ასეთ შესაძლებლობას, დასაშვებია რამდენიმე ოპერატორის მითითება, როგორც ინიციალიზაციის, ისე იტერაციულ ნაწილში. ამ დროს ოპერატორები ერთმანეთისაგან უნდა

გამოვყოთ მძიმით. წინა მაგალითი უფრო ეფექტურად ასე შეიძლება გადავწეროთ:

ლისტინგი 30. მძიმის გამოყენება

```
class Comma {
    public static void main(String args[]) {
        int a, b;
        for(a=1, b=4; a<b; a++, b--) {
            System.out.println("a " + a);
            System.out.println ("b " + b);
        }
    }
}
```

პროგრამის შესრულების შედეგი:

```
a = 1
b = 4
a = 2
b = 3
```

for ციკლის სახეობები

ენაში დასაშვებია for ციკლის რამდენიმე სახეობა, რაც ზრდის მის მოქნილობას და გამოყენების შესაძლებლობას. ციკლის ამ ოპერატორის მოქნილობა გამოიხატება იმაში, რომ მისი სამი ნაწილი: ინიციალიზაცია, პირობის შემოწმება და იტერაცია არაა აუცილებელი ყოველთვის პირდაპირი დანიშნულებით იქნეს გამოყენებული.

ციკლის ოპერატორის ყველაზე ხშირ ვარიაციას წარმოადგენს პირობითი გამოსახულების გამოყენება. კერძოდ, არაა აუცილებელი მმართავი ცვლადის მნიშვნელობა შედარდეს

რაიმე მიზნობრივ მნიშვნელობას. for ციკლის შესრულების პირობა შეიძლება იყოს ნებისმიერი ბულის ტიპის გამოსახულება. მაგალითად,:

```
boolean done = false;
for(int i=1; !done; i++) {
    // .....
    if(interrupted()) done=  true;
}
```

ამ მაგალითში for ციკლის შესრულება გრძელდება მანამ, სანამ ცვლადი done-ს მნიშვნელობა არ გახდება true. ამ ციკლში ციკლის i მმართავი ცვლადის მნიშვნელობის შემოწმება არ ხდება.

შემდეგ მაგალითში ნაჩვენებია, რომ ციკლის ინიციალიზაციის ან იტერაციის ნაწილი ან ორივე ერთად შეიძლება სულ არ იყოს მითითებული:

ლისტინგი 31. for ციკლის ნაწილები შეიძლება ცარიელი იყოს

```
class ForVar {
    public static void main(String args[]) {
        int i;
        boolean done = false;
        i = 0;
        for(; !done; ) {
          System.out.println ("i ტოლია" + i);
          if(i == 10) done = true;
          i++;
        }
    }
}
```

ამ მაგალითში ინიციალიზაციის და იტერაციის ნაწილი გატანილია for ციკლის გარეთ. შესაბამისად ციკლის ოპერატორში ეს ნაწილები ცარიელია. მართალია ამ მაგალითში ეს სახეობა პრიმიტიულად გამოიყურება, მაგრამ ზოგიერთ შემთხვევაში ასეთი ხერხი ძალიან მომგებიანია.

განვიხილოთ კიდევ ერთი სახეობა, სადაც სამივე ნაწილი ცარიელია. ამ გზით შეიძლება შევქმნათ უსასრულო ციკლი (ციკლი, რომელიც არასოდეს არ მთავრდება). მაგალითად,

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

ეს ციკლი იმიტომაა უსასრულო, რომ პირობა, რომლის დარღვევის შემთხვევაში ციკლი წყდება, არ არსებობს. ასეთი ციკლები ზოგიერთ პროგრამაში საჭიროა.

for-each ციკლი

JDK5-ის შემდეგ ციკლის ამ ფორმის გამოყენება შესაძლებელია. თანამედროვე პროგრამირების ენებში სულ უფრო დიდ გამოყენებას პოულობს ციკლების “for-each” (თითოეულისათვის) კონცეფცია. ასეთი ციკლის დანიშნულებაა მკაცრი თანმიმდევრობით მოხდეს განმეორებადი მოქმედებები ობიექტების კოლექციის მიმართ, რომლებიც ვთქვათ, მასივადაა წარმოდგენილი. ზოგიერთ ენაში (მაგალითად, C#) ასეთი ციკლი წარმოდგენილია საკვანძო სიტყვით **foreach**. Java-ში ასეთი ციკლის გამოყენება მიღწეულია for ციკლის გაუმჯობესებით. ამ მიდგომის უპირატესობა ისაა, რომ არაა

საჭირო დამატებით სხვა საკვანძო სიტყვის შემოღება და ადრე არსებული პროგრამის კოდებში ცვლილებების შეტანა აუცილებელი არაა. “for-each” ციკლის ზოგადი ფორმა ასეთია:

for (<ტიპი> <იტერაციული ცვლადი> : <კოლექცია>)
<ოპერატორების ბლოკი>

<ტიპი> - ესაა ტიპი, ხოლო <იტერაციული ცვლადი> - იტერაციული ცვლადის სახელი, რომელიც მიმდევრობით მიიღებს მნიშვნელობებს კოლექციიდან, პირველი ელემენტიდან ბოლომდე. <კოლექცია> - ესაა ის კოლექცია რომლის მიხედვითაც ხორციელდება ციკლი. ჩვენ ჯერ-ჯეროზით განვიხილოთ მხოლოდ მასივის ტიპის კოლექცია. 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 ღებულობს კოლექციიდან მნიშვნელობებს, <ტიპი> უნდა ემთხვეოდეს (ან იყოს თავსებადი) კოლექციაში არსებული ელემენტების ტიპს. ანუ, ჩვენ შემთხვევაში, მასივის ელემენტებისათვის ციკლის შესრულებისას <ტიპი> უნდა იყოს თავსებადი მასივის ტიპთან.

“for-each” ციკლის შემოღების მიზეზებში გასარკვევად განვიხილოთ for ტიპის ციკლი, რომლის შესაცვლელადაც არის შემოღებული ეს სტილი. შემდეგ მაგალითში მასივის ელემენტების ჯამის გამოსაანგარიშებლად გამოიყენება ტრადიციული for ციკლი:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0;  i < 10;  i++) sum += nums[i];
```

ამ მაგალითში, nums მასივიდან ელემენტების ამოღება ხდება მასივის ინდექსაციით i ციკლის ცვლადის საშუალებით.

“for-each” სტილის ციკლი საშუალებას იძლევა მოვახდინოთ ამ პროცესის ავტომატიზაცია. კერძოდ, ასეთი ციკლის გამოყენებით აღარაა საჭირო ციკლის მთვლელის გამოყენება, რომლისთვისაც საწყისი და საბოლოო მნიშვნელობები უნდა მიგვეთითებინა. ასევე, მასივის ინდექსების ხელით მიწერაც აღარაა საჭირო. ამის მაგივრად, პროგრამა ავტომატურად შეასრულებს ციკლს მასივის ყველა ელემენტზე და მიმდევრობით ამოიღებს თითოეული ელემენტის მნიშვნელობას, პირველიდან ბოლო წევრამდე. მაგალითად, წინა პროგრამა “for-each” სტილის ციკლის გამოყენებით ასე შეიძლება ჩავწეროთ:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum =0;
for(int x: nums) sum += x;
```

ასეთი ციკლის დროს მარტივდება პროგრამის ჩაწერის სინტაქსი და ასევე გამოირიცხება მასივის საზღვრებს გარეთ გასვლის შეცდომის შესაძლებლობა. შემდეგ პროგრამაში ნაჩვენებია სრული მაგალითი:

ლისტინგი 32. for ციკლის გამოყენება foreach სტილში

```
class ForEach {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
        int sum = 0;
        for(int x : nums) {
          System.out.println ("შესაკრების მნიშვნელობა: " + x) ;
          sum += x;
        }
        System.out.println ("ჯამი ტოლია: " + sum);
    }
}
```

ამ პროგრამის შედეგი:

```
შესაკრების მნიშვნელობა: 1
შესაკრების მნიშვნელობა: 2
შესაკრების მნიშვნელობა: 3
შესაკრების მნიშვნელობა: 4
შესაკრების მნიშვნელობა: 5
შესაკრების მნიშვნელობა: 6
შესაკრების მნიშვნელობა: 7
შესაკრების მნიშვნელობა: 8
შესაკრების მნიშვნელობა: 9
შესაკრების მნიშვნელობა: 10
ჯამი ტოლია: 55
```

break ოპერატორით შესაძლებელია “for-each” ციკლის შეწყვეტა ყველა ელემენტის განხილვამდე. მაგალითად, შემდეგი მაგალითი აჯამებს მასივის მხოლოდ 5 წევრს:

ლისტინგი 33. break ოპერატორის გამოყენება foreach სტილის ციკლში

```
class ForEach2 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    for(int x : nums) {
       System.out.println("შესაკრების მნიშვნელობა: "+ x);
```

```
        sum += x;
        if(x == 5) break; // ციკლი წყდება თუ 5-ე ელემენტზე ვართ
    }
    System.out.println("პირველი 5 ელემნტის ჯამი ტოლია: "+ sum);
  }
}
```

ამ პროგრამის შედეგი:

```
შესაკრების მნიშვნელობა: 1
შესაკრების მნიშვნელობა: 2
შესაკრების მნიშვნელობა: 3
შესაკრების მნიშვნელობა: 4
შესაკრების მნიშვნელობა: 5
პირველი 5 ელემნტის ჯამი ტოლია: 15
```

საჭიროა ხაზი გავუსვათ “for-each” ციკლის ერთ მნიშვნელოვან გარემოებას. მისი იტერაციული ცვლადი წარმოადგენს ცვლადს „მხოლოდ წასაკითხად“, ვინაიდან იგი დაკავშირებულია მხოლოდ საწყის მასივთან. ამ ცვლადზე მნიშვნელობის მინიჭება არავითარ ცვლილებას არ ახდენს საწყის მასივში. ანუ, მასივის მნიშვნელობის შეცვლა არ შეიძლება იტერაციულ ცვლადზე მნიშვნელობის მინიჭებით. მაგალითად,:

ლისტინგი 34. foreach ციკლი წვდომადია მხოლოდ წასაკითხად

```
class NoChange {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        for(int x : nums) {
          System.out.print(x + " ");
          x = x * 10;   // ეს ცვლადი არავითარ გავლენას არ
                        //  ახდენს მასივზე
        }
```

```
        System.out.println();
        for(int x : nums)
          System.out.print(x +" ");
        System.out.println();
    }
}
```

პირველ ციკლში იტერაციული ცვლადი იზრდება 10-ჯერ, მაგრამ იგი საწყის nums მასივში არავითარ ცვლილებას არ იწვევს, როგორც ეს ჩანს მეორე ციკლის ოპერატორის შესრულების შედეგად.

ამ პროგრამის შედეგი:

1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

იტერაციები მრავალგანზომილებიან მასივებში

for ციკლის ახალი, გაუმჯობესებული ვერსია შესაძლებელია გამოყენებული იქნეს მრავალგანზომილებიანი მასივების მიმართაც. გასათვალისწინებელია, რომ Java-ში მრავალგანზომილებიანი მასივი ესაა მასივების მასივი. მაგალითად, ორგანზომილებიანი მასივი წარმოადგენს ერთგანზომილებიანი მასივის მასივს. ეს შეხსენება მნიშვნელოვანია იმიტომ, რომ მრავალგანზომილებიან მასივებში იტერაციის ჩატარების დროს, იტერაციის თითოეული შედეგი ცალკეული კონკრეტული ელემენტი კი არაა, არამედ მომდევნო მასივი. უფრო მეტიც, for ციკლის იტერაციული ცვლადის ტიპი თავსებადი უნდა იყოს მიღებული მასივის ტიპთან. მაგალითად, ორგანზომილებიანი მასივის შემთხვევაში, იტერაციული ცვლადი უნდა იყოს ერთგანზომილებიან მასივზე მიმთითე-

ბელი ტიპი. ზოგადად, N განზომილებიან მასივში იტერაციისათვის for-each ციკლის გამოყენებისას, მიღებული ობიექტები იქნება N-1 განზომილების მასივი. განვიხილოთ პროგრამა, სადაც ორდონიანი for ციკლი გამოიყენება ორგანზომილებიანი მასივის სტრიქონების მოსაწესრიგებლად:

ლისტინგი 35. foreach ციკლის გამოყენება ორგანზომილებიან ციკლში

```
class ForEach3 {
    public static void main (String args []) {
        int sum = 0;
        int nums[] [] = new int[3] [5];
    // nums მასივის ელემენტების ინიციალიზაცია
        for(int i = 0; i < 3; i++)
        for (int j = 0; j < 5; j++)
         nums [i] [j] = (i +1) * (j + 1) ;
        // for ციკლის foreach სტილში გამოყენება
        // მნიშვნელობების გამოტანისა და აჯამვისათვის
        for(int x[] : nums) {
        for(int y: x) {
         System.out.println ("მნიშვნელობა ტოლია: "+ y);
         sum += y;
         }
        }
        System.out.println("ჯამი: "+ sum);
    }
}
```

ამ მაგალითში განსაკუთრებით ყურადღება გასამახვილებელია სტრიქონზე:

```
for(int x[] : nums) {
```

x ცვლადის ასეთი გამოცხადება განპირობებულია იმით, რომ იგი წარმოადგენს ერთგანზომილებიან მთელრიცხვა მასივზე

კავშირს. მასივად იმიტომაა გამოცხადებული, რომ for ციკლის ყოველი იტერაციის შემდეგ შედეგი არის nums მასივში შემდგომი მასივი, დაწყებული nums [0]-დან. შიგა ციკლი ასრულებს იტერაციას თითოეულ ამ მასივში და კონსოლზე გამოაქვს თითოეული ელემენტის მნიშვნელობა.

გაუმჯობესებული for ციკლის გამოყენება

for-each ციკლის ოპერატორს შეუძლია მასივის ელემენტების მხოლოდ მიმდევრობითი გავლა, პირველიდან ბოლო ელემენტამდე, მაგრამ ეს არ ნიშნავს, რომ მისი გამოყენება შეზღუდულია. ბევრი ალგორითმი სწორედ ასეთ მექანიზმს მოითხოვს. ერთ-ერთი ყველაზე ხშირად გამოყენებული ალგორითმია ძებნის ალგორითმი. მაგალითად, შემდეგი პროგრამა მოუწესრიგებელ მასივში ეძებს კონკრეტულ მნიშვნელობას. ძებნა წყდება საჭირო მნიშვნელობის მოძებნისთანავე.

ლისტინგი 36. foreach სტილის for ციკლით მასივში ძებნა

```
class Search {
    public static void main(String args[]) {
        int nums[] = { 6, 8, 3, 7, 5, 6,1,4 };
        int val = 5;
        boolean found = false;
        for(int x : nums) {
          if (x == val) {
            found = true;
            break;
          }
        }
        if(found)
          System.out.println("მნიშვნელობა მოიძებნა!");
```

```
    }
}
```

ამ შემთხვევაში for-each სტილის ციკლის გამოყენება გამართლებულია, ვინაიდან მოუწესრიგებელ მასივში ძებნა გულისხმობს ყველა ელემენტის მიმდევრობით გადასინჯვას. მასივი რომ მოწესრიგებული იყოს, შესაძლებელი იყო ბინარული ძებნის ალგორითმის გამოყენება, რომლის რეალიზაცია სხვა სტილის ციკლს მოითხოვს. for-each სტილის ციკლის გამოყენება ასევე გამართლებული და მოსახერხებელია საშუალო მნიშვნელობის გამოთვლის, მაქსიმალური ან მინიმალური წევრის მოძებნის, დუბლიკატების მოძებნის და სხვა მრავალ ალგორითმში.

ჩალაგებული ციკლები

სხვა ენების ანალოგიურად, Java-ში დასაშვებია ერთმანეთში ჩალაგებული ციკლების ორგანიზება. მაგალითად,

ლისტინგი 37. ჩალაგებული ციკლები

```
class Nested {
    public static void main (String args [] ){
     int i, j;
     for(i=0; i<10; i++) {
        for (j=i; j<10; j++)
            System.out.print(".");
        System.out.println();
     }
    }
}
```

ამ პროგრამის შედეგი:

```
..........
.........
```

.....
.....
.....
.....
....
...
..
.

გადასვლის ოპერატორები

java-ში განსაზღვრულია გადასვლის სამი ოპერატორი: break, continue და return. ისინი მართვას გადასცემენ პროგრამის სხვა ნაწილს.

ოპერატორი break

ოპერატორი break გამოიყენება სამ ვარიანტში:

1. ოპერატორ switch-ში ასრულებს ოპერატორების მიმდევრობას;
2. ანხორციელებს ციკლიდან გამოსვლას;
3. ის შეიძლება გამოყენებული იქნეს უპირობო გადასვლის (goto) „ცივილიზებულ“ სახედ.

განვიხილოთ ბოლო ორი ვარიანტის გამოყენება.

break ოპერატორი ციკლიდან გამოსასვლელად

break ოპერატორის გამოყენებით შესაძლებელია ციკლის მომენტალური დამთავრება, ამ დროს გამოიტოვება ციკლის

პირობა და ყველა სხვა ოპერატორი ციკლის ტანში. როდესაც პროგრამის შესრულება მივა ციკლის ტანში break ოპერატორთან, იგი წყვეტს ციკლის შესრულებას და მართვა გადაეცემა ციკლის შემდეგ ოპერატორს. მაგალითი:

ლისტინგი 38. break ოპერატორის გამოყენება ციკლიდან გამოსასვლელად

```
class BreakLoop {
    public static void main(String args[]) {
        for(int i=0; i<100; i++) {
        if(i == 10) break; // ციკლიდან გამოსვლა თუ i ტოლია 10
        System.out.println("i: "+ i);
     }
        System.out.println("ციკლი დამთავრებულია.");
    }
}
```

პროგრამის შედეგია:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
ციკლი დამთავრებულია.
```

მმართავი ცვლადის საზღვრების მიხედვით, ციკლი უნდა შესრულებულიყო 0-დან 99-მდე, break ოპერატორი იწვევს ციკლიდან ადრეულ გამოსვლას, როდესაც i ცვლადის მნიშვნელობა 10-ის ტოლი გახდება.

break ოპერატორის ჩალაგებულ ციკლში გამოყენებისას, ის ანხორციელებს გამოსვლას მისი შემცველი უახლოესი ციკლიდან. მაგალითი:

ლისტინგი 39. break ოპერატორი ჩალაგებულ ციკლებში.

```
class BreakLoop3 {
    public static void main(String args[]) {
        for(int i=0; i<3; i++) {
            System.out.print("ბიჯი" + i + ": ");
            for (int j=0; j<100; j++) {
              if(j == 10) break; // გამოსვლა, თუ j ტოლია 10-ის
          System.out.print(j +" ");
        }
        System.out.println();
     }
     System.out.println ("ციკლი დასრულდა.");
    }
}
```

პროგრამის შედეგი:

```
ბიჯი 0: 0 1 2 3 4 5 6 7 8 9
ბიჯი 1: 0 1 2 3 4 5 6 7 8 9
ბიჯი 2: 0 1 2 3 4 5 6 7 8 9
ციკლი დასრულდა.
```

break ოპერატორის გამოყენება უპირობო გადასვლის ოპერატორად

Java-ში არაა რეალიზებული უპირობო გადასვლის ოპერატორი goto, ვინაიდან მას შეუძლია პროგრამის განშტოება ნებისმიერად და არასტრუქტურულად. მაგრამ ზოგიერთ შემთხვევაში ეს ოპერატორი სასარგებლოა, მაგალითად,

ჩალაგებული ციკლიდან რამდენიმე დონით ერთდროული ამოსვლისათვის. ასეთი შემთხვევისათვის Java-ში განსაზღვრულია break-ის გაფართოებული ფორმა. ამ ფორმით შესაძლებელია ერთი ან რამდენიმე ბლოკიდან გამოსვლა. არაა აუცილებელი ეს ბლოკები მარტო ციკლის ან switch ოპერატორის ნაწილი იყოს. შესაძლებელია ნებისმიერი ბლოკისათვისაც, უფრო მეტიც, შესაძლებელია ზუსტად მიეთითოს ოპერატორი, საიდანაც მოხდება პროგრამის შესრულების გაგრძელება, ვინაიდან ეს ფორმა მუშაობს ჭდეებთან. ჭდიანი break ოპერატორის ზოგადი ფორმა ასეთია:

break <ჭდე> ;

<ჭდე> - ესაა რომელიმე ბლოკის საიდენტიფიკაციო ჭდე. ამ ფორმის break ოპერატორის შესრულებისას მართვა გაგრძელდება მითითებული ჭდით მონიშნული ბლოკის შემდეგ. ჭდიანი break ოპერატორი შეიძლება გამოყენებული იქნეს ჩალაგებული ბლოკებიდან გამოსასვლელად. მაგრამ ის არ შეიძლება გამოყენებული იქნეს ისეთ ბლოკზე გადასასვლელად, რომელიც ამ break ოპერატორს არ შეიცავს.

ბლოკის ჭდით მოსანიშნად, იგი უნდა მივუთითოთ ბლოკის დასაწყისში. ჭდე Java-ში დაშვებული ნებისმიერი იდენტიფიკატორია, რომლის შემდეგაც მოდის ორწერტილი. მაგალითად, შემდეგი პროგრამა შეიცავს სამ ჩალაგებულ ციკლს. თითოეული მათგანი დაჭდევებულია. break ოპერატორი იწვევს second ჭდით მონიშნულ ბლოკის ბოლოზე გადასვლას და გამოტოვებს ორ println() ოპერატორს.

ლისტინგი 40. goto ოპერატორის მაგივრად გამოყენება.

```
class Break {
    public static void main(String args[]) {
        boolean t = true;
          first: {
        second: {
            third: {
            System.out.println(" break-ს წინ.");
            if(t) break second; // second ბლოკიდან გამოსვლა
             System.out.println("ეს ოპერატორი არ სრულდება");
             }
        System.out.println(" ეს ოპერატორი არ სრულდება");
            }
        System.out.println ("ეს ოპერატორი second-ის შემდეგია.");
        }
    }
}
```

პროგრამის შედეგი:

break-ს წინ.

ეს ოპერატორი second-ის შემდეგია.

ჭდიანი break ოპერატორის გამოყენების ერთ-ერთი ყველაზე გავრცელებული შემთხვევაა ჩალაგებული ციკლებიდან გამოსვლა. მაგალითად, ამ პროგრამაში გარე ციკლი სრულდება მხოლოდ ერთხელ:

ლისტინგი 41. break-ით ჩალაგებული ციკლებიდან გამოსვლა

```
class BreakLoop4 {
    public static void main(String args[]) {
        outer: for(int i=0; i<3; i++) {
          System.out.print("ბიჯი" + i + ": ");
          for(int j=0; j<100; j++) {
            if(j == 10) break outer; // ორივე ციკლიდან გამოსვლა
```

```
            System.out.print(j +" ");
          }
          System.out.println ("ეს სტრიქონი არ სრულდება");
        }
    System.out.println ("ციკლები დასრულდა.");
    }
}
```

პროგრამის შედეგი:

ბიჯი 0: 0 1 2 3 4 5 6 7 8 9 ციკლები დასრულდა.

ოპერატორი continue

ზოგჯერ საჭიროა კონკრეტულ იტერაციაში ციკლის გაგრძელება მოხდეს, ტანში დანარჩენი ოპერატორების შესრულების გარეშე. ფაქტობრივად, ესაა ციკლის ტანიდან მის ბოლოში გადასვლა. ამ მოქმედების შესასრულებლად გამოიყენება ოპერატორი continue. while და do-while ციკლებში ოპერატორი continue იწვევს მართვის გადაცემას უშუალოდ ციკლის შესრულების პირობაზე. for ციკლში მართვა გადაეცემა ციკლის თავში იტერაციულ ნაწილს, ხოლო შემდეგ პირობით ნაწილს. სამივე ამ ციკლში ნებისმიერი შუალედური ოპერაცია გამოიტოვება. ქვემოთ ნაჩვენებ მაგალითში continue გამოიყენება ერთ სტრიქონში ორი რიცხვის გამოსატანად.

ლისტინგი 42. continue-ს დემონსტრირება.

```
class Continue {
    public static void main(String args[]) {
        for(int i=0; i<10; i++) {
          System.out.print(i +" ");
          if (i%2 == 0) continue;
```

```
            System.out.println("");
        }
    }
}
```

ამ კოდში % გამოიყენება i ცვლადის ლუწობის შესამოწმებლად. პროგრამა კონსოლზე გამოიტანს:

0 1
2 3
4 5
6 7
8 9

ოპერატორმა continue-მ break-ის მსგავსად შეიძლება გამოიყენოს მისი შემცველი ციკლის ჭდე. ქვემოთ ნაჩვენებია პროგრამა, რომელსაც გამოაქვს 0-და 9-მდე რიცხვების გამრავლების სამკუთხა ცხრილი:

ლისტინგი 43. continue ჭდით.

```
class ContinueLabel{
    public static void main(String args[]) {
        outer: for (int i=0; i<10; i++) {
            for (int j=0; j<10; j++) {
             if(j > i) {
                System.out.println();
                continue outer;
             }
             System.out.print(" "+ (i * j));
            }
        }
        System.out.println();
    }
}
```

ამ მაგალითში ოპერატორი continue წყვეტს j-ს მნიშვნელობის თვლის ციკლს და აგრძელებს i-ს დათვლის ციკლს შემდეგი იტერაციიდან. კონსოლზე გამოვა:

0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81

ოპერატორი return

ოპერატორი return გამოიყენება მეთოდიდან ცხადი დაბრუნებისათვის. ანუ იგი უბრუნებს მართვას იმ ობიექტს, რომელმაც ეს მეთოდი გამოიძახა. ამ ოპერატორის უფრო დეტალური აღწერა მოხდება მეთოდის განხილვისას.

ამრიგად, ოპერატორი return მყისიერად ახდენს იმ მეთოდის შესრულების წყვეტას, სადაც ის იმყოფება. შემდეგ მაგალითში return ოპერატორი ახდენს მართვის გადაცემას Java-ს შემსრულებელ გარემოზე, ვინაიდან სწორედ ის იძახებს main() მეთოდს:

ლისტინგი 44. return მეთოდის დემონსტრირება

```
class Return {
```

```
    public static void main(String args[]) {
        boolean t = true;
        System.out. println ("დაბრუნებამდე.");
        if(t) return; // გამომძახებელ ობიექტზე დაბრუნება
        System.out. println("ეს არ სრულდება.");
    }
}
```

ამ პროგრამას გამოაქვს:

დაბრუნებამდე.

ილუსტრირებულ პროგრამაში if(t) ოპერატორის გამოყენება აუცილებელია. მის გარეშე Java-ს კომპილატორი გამოიტანდა შეცდომას “Unreachable code” (მიუღწევადი კოდი), ვინაიდან ის მიხვდებოდა, რომ ბოლო println() ოპერატორი არასდროს არ შესრულდებოდა. ამ შეცდომისგან თავის დასაღწევად კომპილატორი „მოვატყუეთ“ if ოპერატორის ჩამატებით.

თავი III. კლასები

ითვლება, რომ არისტოტელე იყო პირველი ადამიანი, რომელმაც **ტიპის** ცნების ღრმა კვლევები განახორციელა. იგი საუბრობდა თევზებისა და ფრინველების კლასებზე. ის კონცეფცია, რომ ყველა ობიექტი უნიკალურია და ამავე დროს ეკუთვნის (წარმოადგენს) მსგავსი მახასიათებლებისა და ქცევების მქონე ობიექტების კლასს, ფუნდამენტურია ობიექტზე ორიენტირებულ მიდგომაში. ობიექტზე ორიენტირებული დაპროგრამების ენა Java-ში შემოღებულია ფუნდამენტალური საკვანძო სიტყვა class, რომლითაც შეიძლება პროგრამაში აღვწეროთ ახალი ტიპი. განვიხილოთ „ბანკის მოლარეს“ კლასიკური ამოცანა. არსებობს მოლარეების, კლიენტების, ანგარიშების, გადახდების და ფულადი ერთეულების ჯგუფების მრავალი „ობიექტი“. ობიექტები, რომლებიც ერთმანეთის იდენტურები არიან, მაგრამ განსხვავდებიან მხოლოდ მდგომარეობით, შეიძლება დავაჯგუფოთ **ობიექტების კლასებად**. აქედან გაჩნდა საკვანძო სიტყვა **class (კლასი)**. მონაცემების აბსტრაქტული ტიპების შექმნა არის ობიექტზე ორიენტირებული პროგრამირების ფუნდამენტარი ცნება. მონაცემთა აბსტრაქტული ტიპები ისევე მოქმედებენ, როგორც პრიმიტიული ტიპები: შესაძლებელია შეიქმნას ტიპების შესაბამისი ცვლადები (ობიექტზე ორიენტირებული პროგრამირების ტერმინებში მათ უწოდებენ ობიექტებს ან ეგზემპლარებს) და მათზე მოხდეს მანიპულირებები (ამას უწოდებენ შეტყობინების, გზავნილის ან მოთხოვნის გაგზავნას); ხდება მოთხოვნის გაგზავნა და შემდეგ ობიექტი

ღებულობს გადაწყვეტილებას, როგორ დაამუშაოს იგი. თითოეული კლასის წარმომადგენელი (წევრი) ერთმანეთის მსგავსია, ჩვენ მაგალითში: ყოველ ანგარიშს აქვს ბალანსი, ყოველ მოლარეს ჰქვია სახელი. ამიტომ ყველა მოლარე, ანგარიში, თანხა და ა.შ. პროგრამაში შეიძლება წარმოვად-გინოთ უნიკალური არსის სახით. სწორედ ესაა ობიქტის არსი, თითოეული ობიექტი ეკუთვნის გარკვეულ კლასს, რომელიც განსაზღვრავს მის მახასიათებლებსა და ქცევას.

ობიექტზე ორიენტირებულ ყველა ენაში, მონაცემების ახალი ტიპის შექმნისას, იყენებენ საკვანძო სიტყვას class (კლასი), ამიტომ როდესაც პროგრამაში ვნახავთ სიტყვა class, ჩავთვალოთ, რომ ლაპარაკია ტიპზე და პირიქით.

ამრიგად, ვინაიდან კლასი განსაზღვრავს იდენტური თვისებების (მონაცემების ელემენტების) და ქცევის (ფუნქციონალობის) ობიექტების ერთობლიობას, ამიტომ კლასი ფაქტობრივად არის მონაცემთა ტიპი. მაგალითად, მცოცავმძიმიანი რიცხვს აქვს გარკვეული თვისებები, ქცევის თავისებურებები. განსხვავება იმაშია, რომ პროგრამისტი ამოცანის რაიმე ასპექტის წარმოსადგენად, კომპიუტერის მეხსიერებაში მონაცემის შენახვის უკვე არსებული ტიპის გამოყენების მაგივრად ქმნის ახალ კლასს. ასეთნაირად შესაძლებელია პროგრამისტის მოთხოვნების შეესაბამისად პროგრამირების ენის გაფართოება მონაცემთა ახალი ტიპების დამატების გზით.

ახალი კლასის შექმნის შემდეგ შესაძლებელია ნებისმიერი რაოდენობის ამ კლასის ობიექტის შექმნა და შემდგომ მათზე მანიპულირება (მათთან მუშაობა).

კლასი Java-ში ცენტრალური კომპონენტია. ვინაიდან კლასი აღწერს ობიექტის ფორმასა და არსს, ამიტომ იგი არის ის ლოგიკური კონსტრუქცია, რომელზედაც აგებულია მთელი ენა. ნებისმიერი გამოთვლითი პროცესი (კონცეფცია), რომელიც საჭიროა Java პროგრამაში განხორციელდეს, უნდა მოთავსდეს კლასის შიგნით.

ჩვენ მაგალითებში აქამდეც ვიყენებდით კლასებს, მაგრამ მის პრიმიტიულ ფორმებს. კლასებში მოთავსებული იყო მხოლოდ main() მეთოდი, რომლებსაც ენის სინტაქსის გასაცნობი პროგრამების შესაქმნელად ვიყენებდით.

ამრიგად, კლასი ესაა ობიექტის შაბლონი, ხოლო ობიექტი ესაა კლასის ეგზემპლარი. ვინაიდან ობიექტი კლასის ეგზემპლარია, ამიტომ ამ ორ ტერმინს (ობიექტი და ეგზემპლარი) ჩვენ სინონიმებად ვთვლით.

კლასის ზოგადი ფორმა

კლასის გამოცხადებისას უნდა აღვწეროთ მისი კონკრეტული ფორმა და არსი. ეს ხდება კლასში მონაცემების აღწერით და იმ კოდის ჩაწერით, რომელიც მოქმედებს მონაცემებზე. მარტივი კლასები შესაძლებელია შეიცავდეს მხოლოდ კოდს ან მხოლოდ მონაცემებს, მაგრამ რეალურ პროგრამებში გამოყენებული კლასების უმრავლესობა შეიცავს ორივე კომპონენტს.

აქამდე გამოყენებული კლასები წარმოადგენდნენ სრული ფორმის შეზღუდულ მაგალითებს. კლასის გამოცხადების გამარტივებულ ზოგად ფორმას ასეთი სახე აქვს:

```
class <კლასის სახელი> {
    <ტიპი> <კლასის ცვლადი1>;
    <ტიპი> <კლასის ცვლადი2>;
    //  . . .
    <ტიპი> <კლასის ცვლადიN>;
    <ტიპი> <მეთოდის სახელი1>(<პარამეტრების სია>){
        // მეთოდის ტანი
    };
    <ტიპი> <მეთოდის სახელი2>(<პარამეტრების სია>){
        // მეთოდის ტანი
    };
    // . . .
    <ტიპი> <მეთოდის სახელი N>(<პარამეტრების სია>){
        // მეთოდის ტანი
    };
}
```

კლასის შიგნით გამოცხადებულ მონაცემებს (ანუ ცვლადებს) **კლასის ან ეგზემპლარის ცვლადებს** უწოდებენ. კოდი მოთავსებულია მეთოდის შიგნით. კლასის შიგნით გამოცხადებულ მეთოდებსა და ცვლადებს ერთად უწოდებენ **კლასის წევრებს**. კლასების უმრავლესობაში ეგზემპლარის ცვლადებზე მოქმედებები და მიმართვები ხდება ამ კლასში აღწერილი მეთოდების საშუალებით. საზოგადოდ, მეთოდები განსაზღვრავენ კლასის მონაცემების გამოყენების ხერხებს.

კლასის შიგნით გამოცხადებულ ცვლადებს იმიტომ უწოდებენ ეგზემპლარის ცვლადებს, რომ კლასის ყოველი ეგზემპლარი (ანუ კლასის ყოველი ობიექტი) შეიცავს ამ

ცვლადების საკუთარ ასლს (კოპიოს). ამრიგად, ერთი ობიექტის მონაცემები განცალკევებული და განსხვავებულია მეორე ობიექტის მონაცემებისაგან.

ყველა მეთოდს ისეთივე ზოგადი ფორმა აქვს, როგორც main() მეთოდს, რომელსაც ჩვენ აქამდე ვიყენებდით, თუმცა მათ უმრავლესობას არ ექნება მითითებული static ან public მოდიფიკატორი. საყურადღებოა, რომ კლასის ზოგად ფორმაში main() მეთოდი არაა აღწერილი. Java-ს კლასები შეიძლება არც შეიცავდეს ამ მეთოდს. ამ მეთოდის მითითება აუცილებელია მხოლოდ იმ შემთხვევაში, როდესაც ეს კლასი საჭიროა გამოცხადდეს პროგრამის საწყის წერტილად.

მარტივი კლასი

განვიხილოთ მარტივი კლასი Box (პარალელეპიპედი), რომელიც აღწერს ეგზემპლარის სამ ცვლადს: width (სიგანე), height (სიმაღლე) და depth (სიღრმე).

```
class Box {
    double width;
    double height;
    double depth;
}
```

როგორც აღვნიშნეთ, კლასი განსაზღვრავს მონაცემთა ახალ ტიპს. ამ შემთხვევაში მონაცემთა ახალ ტიპს ქვია Box. ეს სახელი გამოყენებული იქნება Box ტიპის ობიექტების გამოსაცხადებლად. უნდა გვახსოვდეს, რომ class გამოცხადება ქმნის მხოლოდ შაბლონს და არა ნამდვილ

ობიექტს. ამიტომ ზემოთ ჩაწერილი კოდი არ ქმნის Box-ის ტიპის ობიექტს.

Box ტიპის ობიექტის შესაქმნელად, უნდა გამოვიყენოთ ამდაგვარი ოპერატორი:

```
Box mybox = new Box();     // Box ტიპის mybox ობიექტის
                           // შექმნა
```

ამ ოპერატორის შესრულების შემდეგ mybox გახდება Box კლასის ეგზემპლარი. ანუ, ის შეიძენს ფიზიკურ არსს.

გავიმეოროთ, რომ: კლასის ყოველი ეგზემპლარის შექმნით ჩვენ ვქმნით ობიექტს, რომელსაც გააჩნია თავისი, საკუთარი ეგზემპლარის ცვლადი(ები), რომლებიც კლასშია განსაზღვრული. ამრიგად, Box-ის ყოველი ობიექტი შეიცავს width, height და depth ცვლადების საკუთარ კოპიოებს. ამ ცვლადებზე მიმართვისათვის საჭიროა ოპერაცია წერტილის (.) გამოყენება. ეს ოპერაცია ობიექტის სახელს აკავშირებს ეგზემპლარის ცვლადის სახელთან. მაგალითად, mybox ობიექტის width ცვლადს მნიშვნელობად 100 რომ მივანიჭოთ, საჭიროა შემდეგი ოპერატორის გამოყენება:

```
mybox.width = 100;
```

ეს ოპერატორი კომპილატორს მიუთითებს, რომ width ცვლადის კოპიოს, რომელიც mybox ობიექტის შიგნით ინახება, უნდა მივანიჭოთ მნიშვნელობად 100. საზოგადოდ, ოპერაცია წერტილი გამოიყენება, როგორც ეგზემპლარის ცვლადზე მიმართვისათვის, ისე ობიექტის შიგა მეთოდებზე მისამართავადაც. ქვემოთ ნაჩვენებია Box კლასის გამოყენების სრული პროგრამა.

ლისტინგი 45.

```
class Box {
    double width;
    double height;
    double depth;
}
// ამ კლასში იქმნება Box ტიპის ობიექტი
class BoxDemo {
    public static void main(String args[]) {
        Box mybox = new Box();
        double vol;
        // mybox-ის ცვლადების ინიციალიზაცია
        mybox.width = 10;
        mybox.height = 20;
        mybox.depth = 15;
    // მოცულობის გამოთვლა
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
    }
}
```

ამ პროგრამის ფაილს სახელად უნდა მივანიჭოთ BoxDemo.java, ვინაიდან main() მეთოდი განსაზღვრულია BoxDemo კლასში და არა Box კლასში. საერთოდ შესაძლებელია ყოველი კლასი ცალკე ფაილში ჩავდოთ, რომელსაც კლასის სახელს დავარქმევთ.

ამ პროგრამის შესრულების შედეგად კონსოლზე გამოვა:

Volume is 3000.0

თუ Box ტიპის ორი ობიექტი გვექნება, თითოეულს width, height და depth ცვლადების საკუთარი კოპიოები ექნებათ. მნიშვნელოვანია გავიაზროთ, რომ ერთი ობიექტის ცვლადების ეგზემპლარების ცვლილება არ იწვევს სხვა

ობიექტების ცვლადების ეგზემპლარების ცვლილებას. მაგალითი:

ლისტინგი 46.

```
class Box {
    double width;
    double height;
    double depth;
}
class BoxDemo2 {
    public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
     double vol;
    // mybox1 ცვლადების ინიციალიზაცია
     mybox1.width = 10;
     mybox1.height = 20;
     mybox1.depth = 15;
    //* mybox2 ცვლადების ინიციალიზაცია
     mybox2.width = 3;
     mybox2.height = 6;
     mybox2.depth = 9;
    // მოცულობის გამოთვლა 1
     vol = mybox1.width * mybox1.height * mybox1.depth;
     System.out.println("Volume is "+ vol);
    // მოცულობის გამოთვლა 2
     vol = mybox2.width * mybox2.height * mybox2.depth;
     System.out.println ("Volume is " + vol);
    }
}
```

ამ პროგრამის შედეგი კონოლზე:

Volume is 3000.0

Volume is 162.0

როგორც მაგალითიდან ჩანს mybox1 და mybox2 ობიექტების ცვლადები ერთმანთისაგან სრულიად იზოლირებულია.

ობიექტების გამოცხადება

როგორც აღვნიშნეთ, კლასის შექმნით ჩვენ ვქმნით ახალ ტიპს, რომელიც შემდეგ შეიძლება გამოყენებული იქნეს ამ ტიპის ობიექტების შესაქმნელად. რომელიმე კლასის ობიექტის შექმნა ორ-საფეხურიანი ოპერაციაა. ჯერ უნდა გამოცხადდეს კლასის ტიპის ცვლადი. ეს ცვლადი ობიექტს არ განსაზღვრავს და არ ქმნის, იგი მხოლოდ ცვლადია, რომელსაც შესაძლებელია წვდომა ჰქონდეს ობიექტზე. შემდეგ, უნდა შეიქმნას ნამდვილი ობიექტის ფიზიკური ასლი, კოპიო და მიენიჭოს ამ ცვლადს. ეს სრულდება new ოპერაციით. ეს ოპერაცია დინამიურად, ანუ პროგრამის შესრულების პროცესში გამოუყოფს მეხსიერებას ობიექტს და აბრუნებს მასთან კავშირს (შეიძლება ითქვას, რომ ესაა ობიექტისათვის გამოყოფილი მეხსიერების მისამართი). ამრიგად, Java-ში კლასის ყველა ობიექტი იქმნება დინამიურად.

წინა მაგალითში ობიექტის შესაქმნელად გამოყენებულია ასეთი სტრიქონი:

```
Box mybox = new Box();
```

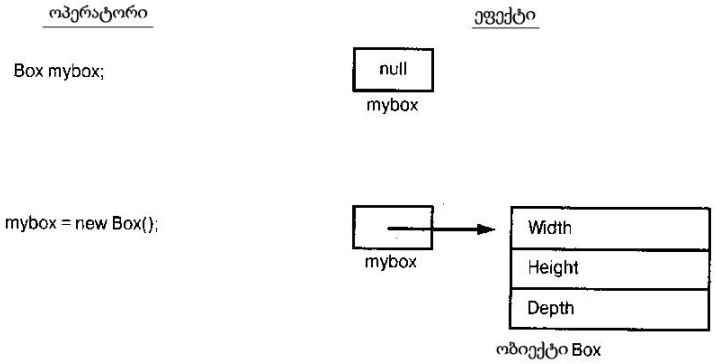
ამ ოპერატორით ხდება აღწერილი ბიჯების შესრულება. თითოეული ბიჯი უფრო ნათელი რომ გახდეს, ეს ოპერატორი ასე შეიძლება ჩავწეროთ:

```
Box mybox;          // ობიექტზე მიმართვის გამოცხადება
```

```
mybox = new Box();  // Box-ის ობიექტებისათვის მეხსიერების გამოყოფა
```

პირველ სტრიქონში ხდება mybox-ის გამოცხადება Box ტიპის ობიექტზე მიმართვისათვის (წვდომისათვის). ამ სტრიქონის შესრულების შემდეგ mybox ცვლადის მნიშვნელობა იქნება null, ანუ იგი ჯერ რეალურ ობიექტზე არ მიუთითებს. ამ ეტაპზე mybox-ის გამოყენება გამოიწვევს კომპილაციის შეცდომას. შემდეგი სტრიქონი რეალური ობიექტისათვის გამოყოფს მეხსიერებას და mybox ცვლადს მიანიჭებს ამ ობიექტზე მითითებას. ამ სტრიქონის შესრულების შემდეგ mybox ცვლადი შეიძლება გამოყენებული იქნეს, როგორც Box-ის ობიექტი. თუმცა სინამდვილეში mybox-ში წერია მხოლოდ რეალური Box ობიექტის მეხსიერების მისამართი.

ამ ორი სტრიქონის შესრულების ეფექტი ნაჩვენებია შემდეგ სურათზე



სურათი 7. Box ტიპის ობიექტის გამოცხადება

ახლა უფრო დაწვრილებით განვიხილოთ ოპერაცია new. როგორც ზემოთ აღვნიშნეთ, ოპერაცია new დინამიურად

უყოფს მეხსიერებას ობიექტს. ამ ოპერაციის ზოგადი ფორმა ასეთია:

<კლასის ცვლადი> = new <კლასის სახელი>();

<კლასის ცვლადი> ესაა კლასის შესაბამისი ტიპის ცვლადი; <კლასის სახელი> იმ კლასის სახელია, რომლის ობიექტიც იქმნება ანუ რომლის კონკრეტიზაციაც ხდება. კლასის სახელი, რომლის შემდეგაც მრგვალი ფრჩხილები მოდის, მიუთითებს ამ კლასის **კონსტრუქტორს**. კონსტრუქტორი განსაზღვრავს იმ მოქმედებებს, რომლებიც საჭიროა ჩატარდეს კლასის ობიექტის შესაქმნელად. კონსტრუქტორი ყველა კლასისათვის მნიშვნელოვანი ნაწილია. რეალურ პროგრამებში გამოყენებული კლასების უმრავლესობა ცხადად აღწერს თავის კონსტრუქტორს თვით ამ კლასის აღწერის შიგნით. თუ კონსტრუქტორი კლასის აღწერაში ცხადად არაა მითითებული, Java ავტომატურად უზრუნველყოფს კლასს სტანდარტული კონსტრუქტორით. ეს ხდება Box-ის ობიექტის შექმნისას.

ახლა შეიძლება გავცეთ პასუხი კითხვას, რატომ არ იქმნება new ოპერაციით ელემენტარული ცვლადები, მაგალითად, მთელი რიცხვები, სიმბოლოები ან ნამდვილი რიცხვები? ეს იმითაა გამოწვეული, რომ ელემენტარული (პრიმიტიული) ტიპები Java-ში რეალიზებული არაა ობიექტების სახით. ეს გაკეთებულია ეფექტურობის ამაღლების მიზნით. როგორც შემდგომში ვნახავთ, ობიექტებს აქვთ სხვადასხვა თვისებები და ატრიბუტები, რომლის დამუშავებაც სხვანაირად ხდება ვიდრე ელემენტარული ტიპების. ეს დამუშავება კი მეტ კომპიუტერულ რესურსებს (პროცესორის დროს, მეხსიერე-

ბას) მოითხოვს. ამ დამატებითი რესურსებისა და დამუშავების გარეშე ელემენტარულ ტიპებთან მუშაობა უფრო ეფექტურად ხდება.

კიდევ ერთხელ განვიხილოთ განსხვავება ობიექტსა და კლასს შორის. კლასი ქმნის მონაცემთა ახალ ტიპს, რომელიც შეიძლება გამოყენებული იქნეს ობიექტის შესაქმნელად. ანუ კლასი ქმნის ლოგიკურ კარკასს, სადაც განსაზღვრულია მის წევრებს შორის ურთიერთკავშირი. კლასის ობიექტის გამოცხადების დროს იქმნება კლასის ეგზემპლარი. ამრიგად, კლასი არის ლოგიკური კონსტრუქცია, ხოლო ობიექტი- ფიზიკური არსი (ე.ი. ობიექტს უკავია რეალური მეხსიერება).

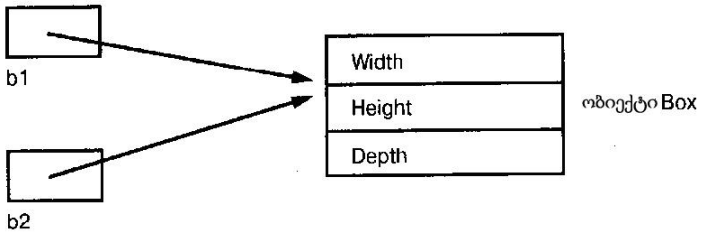
ობიექტზე მიმთითებელ ცვლადზე მინიჭება

საინტერესოა განვიხილოთ ობიექტზე მიმთითებელ ცვლადებზე მინიჭების ოპერაციის შესრულება. მაგალითად, რა მოქმედება შესრულდება ამ ფრაგმენტის შედეგად?

```
Box b1 = new Box();
Box b2 = b1;
```

შეიძლება ვიფიქროთ, რომ b2 ცვლადს ენიჭება იმ ობიექტის კოპიოზე მითითება, რომელსაც უკავშირდება b1 ცვლადი. ანუ შეიძლება გვეგონოს, რომ b1 და b2 ცვლადები მიმართავენ სხვადასხვა დამოუკიდებელ ობიექტს. მაგრამ ეს ასე არაა. კოდის ამ ფრაგმენტის შესრულების შედეგად ორივე b1 და b2 ცვლადს ერთიდაიგივე ობიექტზე ექნებათ მიმართვა. b1 ცვლადის მნიშვნელობის b2 ცვლადზე მინიჭების შედეგად არ ხდება ახალი ობიექტისათვის

მეხსიერების გამოყოფა ან არსებული ობიექტის რაიმე ნაწილის კოპირება. მინიჭების ამ ოპერაციის შედეგად, b2 ცვლადი მიმართავს იმავე ობიექტს, რომელსაც b1 ცვლადი. ამრიგად, ნებისმიერი ცვლილება, რომელიც ჩატარდება ობიექტში b2 ცვლადის საშუალებით, ცვლილებას გამოიწვევს ობიექტში, რომელზეც მიუთითებს b1 ცვლადი, ვინაიდან ისინი ერთიდაიგივე ობიექტებია. ეს სიტუაცია აღწერილია შემდეგ სურათზე



სურათი 8. ობიექტზე მიმთითებელი ტიპის ცვლადის გამოყენება

მართალია b1 და b2 ცვლადები მიმართავენ ერთი და იგივე ობიექტს, სხვა მხრივ ისინი ერთმანეთთან დაკავშირებულები არ არიან. მაგალითად, შემდეგი მინიჭების ოპერაცია b1 ცვლადის კავშირს გაწყვეტს საწყის ობიექტთან, ხოლო თვით ობიექტზე ან b2 ცვლადის მნიშვნელობაზე, არავითარ ზეგავლენას არ მოახდენს:

```
Box b1 = new Box();
Box b2 = b1;
// . . .
b1 = null;
```

ამ მაგალითში b1-ის მნიშვნელობა გახდება null, ხოლო b2 ცვლადი ისევ მიუთითებს საწყის ობიექტს.

ამრიგად, დავიმახსოვროთ!!! ერთ ობიექტზე მიმთითებელ ცვლადზე მეორე ობიექტზე მიმთითებელი ცვლადის მინიჭება არ იწვევს ობიექტის კოპიოს შექმნას, ენიჭება მხოლოდ თვით მითითების ასლი.

მეთოდები კლასებში

როგორც აღვნიშნეთ, ჩვეულებრივ კლასები შედგება ორი ელემენტისაგან: ეგზემპლარის ცვლადებისა და მეთოდებისაგან.

მეთოდის გამოცხადების ზოგადი ფორმა ასეთია:

```
<ტიპი> <მეთოდის სახელი>(<პარამეტრების სია>){
    // მეთოდის ტანი
}
```

<ტიპი> მიუთითებს მეთოდის მიერ დაბრუნებული მონაცემის ტიპს. იგი ნებისმიერი დასაშვები ტიპი შეიძლება იყოს, მათ შორის პროგრამისტის მიერ შექმნილი კლასის ტიპიც. თუ მეთოდი მნიშვნელობას არ აბრუნებს, ტიპი მაინც უნდა მიეთითოს და მისი მნიშვნელობა უნდა იყოს void.

<მეთოდის სახელი> წარმოადგენს მეთოდის სახელს. სახელი შეიძლება იყოს ნებისმიერი დასაშვები იდენტიფიკატორი, გარდა იმ სახელებისა, რომლებიც უკვე გამოყენებულია კლასის სხვა ელემენტებისაგან მიმდინარე სახელთა განსაზღვრის არეში.

<პარამეტრების სია> წარმოადგენს წყვილების მიმდევრობას, რომლებიც ერთმანეთისაგან გამოყოფილია მძიმით (,). წყვილი ესაა ტიპი და იდენტიფიკატორი. შინაარსობრივად, პარამეტრები ესაა ცვლადები, რომლებიც მნიშვნელობად ღებულობენ იმ არგუმენტების მნიშვნელობებს, რომლებიც გადაეცემა მეთოდს მისი გამოძახებისას. თუ მეთოდს პარამეტრები არ აქვს, ფრჩხილებს შიგნით პარამეტრების სია იქნება ცარიელი.

მეთოდები, რომლებსაც დაბრუნებული მნიშვნელობის ტიპად არ უწერიათ void, გამომძახებელ პროცედურას მნიშვნელობას უბრუნებენ return ოპერატორის საშუალებით:

return <მნიშვნელობა>;

რეალურ პროგრამებში ისეთი კლასი, რომელიც მხოლოდ ცვლადებს შეიცავს, იშვიათია. უმრავლეს შემთხვევაში, ეგზემპლარის ცვლადებზე რომელიმე კლასიდან მიმართ-ვისათვის, გამოყენებული უნდა იქნეს მეთოდი. ფაქტობრივად მეთოდები განსაზღვრავს კლასების ინტერფეისს. ეს კი პროგრამისტს საშუალებას აძლევს კლასის შიგა მონაცემთა სტრუქტურა დაფაროს და მათთან მიმართვა მოახდინოს მეთოდების საშუალებით. გარდა მონაცემებთან მიმართვის მეთოდებისა, კლასებში შეიძლება აღვწეროთ შიგა გამოყენების მეთოდებიც.

მაგალითისათვის დავუმატოთ მეთოდი Box კლასს. წინა პროგრამებს თუ განვიხილავთ, შეიძლება დავასკვნათ, რომ უმჯობესი იქნება, თუ მოცულობის გამოთვლის პროცედურა თვით კლასში იქნება მოთავსებული. მართლაც, ვინაიდან

პარალელეპიპედის მოცულობა მის ზომებზეა დამოკიდებული, უფრო ლოგიკურია მისი გამოთვლა Box კლასში მოხდეს. ამიტომ Box კლასს დავუმატოთ მოცულობის გამოთვლის მეთოდი:

ლისტინგი 47. პროგრამაში Box კლასი შეიცავს მეთოდს

```
class Box {
    double width;
    double height;
    double depth;
// მოცულობის გამოთვლა და ბეჭდვა
    void volume () {
     System.out.println("Volume is ");
     System.out.println(width * height * depth);
    }
}
class BoxDemo3 {
    public static void main(String args[]) {
     Box mybox1 = new Box();
     Box mybox2 = new Box();
// mybox1 ეგზემპლარის ცვლადების ინიციალიზაცია
     mybox1.width = 10;
     mybox1.height = 20;
     mybox1.depth = 15;
// mybox2 ეგზემპლარის ცვლადების ინიციალიზაცია
     mybox2.width = 3;
     mybox2.height = 6;
     mybox2.depth = 9;
// მოცულობების გამოთვლის მეთოდის გამოძახება
     mybox1.volume();
     mybox2.volume();
    }
}
```

ამ პროგრამის შედეგი ემთხვევა წინა პროგრამის შედეგს:

Volume is 3000.0
Volume is 162.0

უფრო დაწვრილებით განვიხილოთ სტრიქონები:

```
mybox1.volume();
mybox2.volume();
```

პირველ სტრიქონში ხდება mybox1 ობიექტიდან volume() მეთოდზე მიმართვა. ამისათვის გამოყენებული იქნა ობიექტის სახელი, რომლის შემდეგ მოდის ოპერაცია წერტილის სიმბოლო. ამრიგად, mybox1.volume() მიმართვა კონსოლზე გამოიტანს mybox1 ობიექტის მოცულობის მნიშვნელობას, ხოლო mybox2.volume() მიმართვა mybox2 ობიექტის მოცულობას. volume() მეთოდის თითოეული გამოძახების შედეგად გამოვა გამომძახებელი ობიექტის მოცულობა.

მეთოდის გამოძახების მექანიზმი ასეთია: mybox1.volume() მეთოდის გამოძახების შემდეგ Java-ს შემსრულებელი გარემო მართვას გადასცემს volume() მეთოდის შიგა კოდს (მეთოდის შიგა ოპერატორებს). მეთოდში შესრულდება ყველა საჭირო ოპერატორი და მართვა უბრუნდება გამომძახებელ პროგრამას, რომლის შესრულება გაგრძელდება, გამომძახებელი ოპერატორის შემდეგი ოპერატორიდან. ზოგადად შეიძლება ითქვას, რომ მეთოდი ესაა ქვეპროგრამების რეალიზაციის საშუალება.

ყურადღება გავამახვილოთ volume() მეთოდში ერთ მნიშვნელოვან ნიუანსზე: width, height და depth ეგზემპლარის ცვლადებზე მიმართვა ხდება უშუალოდ, მის წინ მითითებული არაა ობიექტის სახელი ან წერტილი. როდესაც მეთოდი იყენებს ეგზემპლარის ცვლადს, რომელიც იმავე კლასშია განსაზღვრული, იგი მას მიმართავს უშუალოდ, ობიექტზე ცხადი მითითების გარეშე და წერტილის ოპერაციის გარეშე. თუ დავფიქრდებით, მართლაც, მეთოდის გამოძახება ყოველთვის ხდება მისი კლასის რომელიმე ობიექტიდან. როდესაც გამოძახება მოხდება ობიექტი ცნობილია. ამიტომ მეთოდის შიგნით ობიექტის ხელმეორედ მითითება სრულიად ზედმეტია. ეს ნიშნავს, რომ width, height და depth მიმართავენ ცვლადების კოპიოებს, რომლებიც ინახება იმ ობიექტში, რომელმაც volume() მეთოდი გამოიძახა.

შეიძლება დასკვნის სახით ვთქვათ: როდესაც ეგზემპლარის ცვლადზე მიმართვა ხდება კოდიდან, რომელიც ამ ცვლადის განმსაზღვრელი (გამომცხადებელი) კლასი არაა, მაშინ ეს მიმართვა უნდა მოხდეს ობიექტიდან ოპერაცია წერტილის გამოყენებით. ხოლო, როდესაც მიმართვა ხდება კოდიდან, რომელიც იმავე კლასის ნაწილია, სადაც ეგზემპლარის აღწერა მოხდა, ცვლადზე მიმართვა შეიძლება მოხდეს უშუალოდ. ეს წესები გამოიყენება მეთოდების მიმართაც.

მნიშვნელობის დაბრუნება

მართალია, volume() მეთოდს მოცულობის გამოთვლის გამოთვლა გადააქვს იმ კლასის შიგნით, რომელსაც ეს

მეთოდი მიეკუთვნება, მაგრამ ასეთი გამოთვლა მაინც არ ითვლება საუკეთესო ვარიანტად. მაგალითად, შესაძლებელია პროგრამის სხვა ნაწილში საჭირო იყოს მოცულობის გამოთვლა კონსოლზე ბეჭდვის გარეშე. volume() მეთოდის უფრო რაციონალური რეალიზაციაა მოცულობის მნიშვნელობის გამოთვლა და ამ მნიშვნელობის დაბრუნება გამომძახებელი ობიექტისათვის. შემდეგი პროგრამა წარმოადგენს გაუმჯობესებულ ვერსიას და ასრულებს სწორედ ამ ამოცანას:

ლისტინგი 48. volume() მეთოდი აბრუნებს მოცულობის მნიშვნელობას.

```
class Box {
    double width;
    double height;
    double depth;
// მოცულობის გამოთვლა და დაბრუნება
    double volume() {
        return width * height * depth;
    }
}
class BoxDemo4 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
//  mybox1 და mybox2 ცვლადების ინიციალიზაცია
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
```

```
// პირველი პარალელეპიპდის მოცულობა
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
// მეორე პარალელეპიპდის მოცულობა
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

როგორც ხედავთ, პროგრამაში volume() მეთოდის გამოძახება ხდება მინიჭების ოპერატორის მარჯვენა მხარეში. ამ ოპერატორის მარცხენა ნაწილი წარმოადგენს ცვლადს vol, რომელიც მიიღებს მეთოდის მიერ დაბრუნებულ მნიშვნილობას. ამრიგად, ასეთი ოპერატორის შესრულების შედეგად:

```
vol = mybox1.volume();
```

mybox1.volume() მეთოდის მნიშვნელობა იქნება 3000.0 და ეს მნიშვნელობა მიენიჭება vol ცვლადს.

დაბრუნებული მნიშვნელობებთან მუშაობის დროს გასათვალისწინებელია ორი გარემოება:

- მეთოდის მიერ დაბრუნებული მნიშვნელობა, თავსებადი უნდა იყოს მეთოდის აღწერის დროს მითეთებულ დაბრუნების ტიპთან. მაგალითად, თუ რომელიმე მეთოდის დაბრუნების ტიპია boolean, არ შეიძლება რიცხვითი ტიპის მნიშვნელობის დაბრუნება;
- ცვლდი, რომელსაც ენიჭება მეთოდის მიერ დაბრუნებული მნიშვნელობა (ჩვენ მაგალითში vol), ასევე

თავსებადი უნდა იყოს მეთოდში მითითებულ დაბრუნების ტიპთან.

კიდევ ერთი ნიუანსი: წინა პროგრამა შეიძლება კიდევ უფრო ეფექტურად ჩაიწეროს, ვინაიდან ცვლადი vol პრინციპში შეიძლება სულაც არ გამოვიყენოთ. volume() მეთოდზე მიმართვა პირდაპირ შეიძლება გამოვიყენოთ println() ოპერატორში ასეთნაირად:

System.out.println("მოცულობა ტოლია " + mybox1.volume());

ამ შემთხვევაში, ბეჭდვის ოპერატორის შესრულებისას mybox1.volume() მეთოდი ავტომატურად გამოიძახება და მის მიერ დაბრუნებული მნიშვნელობა გადაეცემა println() მეთოდს.

პარამეტრებიანი მეთოდის დამატება

ხშირ შემთხვევაში მეთოდებს სჭირდებათ პარამეტრების გადაცემა. პარამეტრების გადაცემა მეთოდს უფრო ზოგადს ხდის. ანუ პარამეტრებიანი მეთოდები შეიძლება მუშაობდეს სხვადასხვა მონაცემებთან და/ან გამოყენებული იქნეს სხვადასხვა სიტუაციებში. საილუსტრაციოდ განვიხილოთ მარტივი მაგალითი, მეთოდი აბრუნებს 10-ის კვადრატს:

```
int square (){
    return 10 * 10;
}
```

მართალია ეს მეთოდი აბრუნებს 100, მაგრამ მისი გამოყენება ძალიან შეზღუდულია. თუ ამ მეთოდს ისე შევცვლით, რომ მან მიიღოს პარამეტრი, მაშინ ის უფრო გამოსადეგი იქნება:

```
int square (int i){
    return i * i;
}
```

ახლა მეთოდი square() დააბრუნებს პარამეტრად გადაცემული ნებისმიერი მთელი რიცხვის კვადრატს.

მაგალითად,

```
int x, y;
x = square (5) ;      // x = 25
x = square(9);        // x = 81
y = 2;
x = square(y);        // x = 4
```

square() მეთოდის პირველი მიმართვისას მნიშვნელობა 5 გადაეცემა i პარამეტრს. მეორე მიმართვისას i მნიშვნელობად ღებულობს 9-ს. მესამე გამოძახებისას მეთოდს გადაეცემა y ცვლადის მნიშვნელობა, რომელიც 2-ის ტოლია.

მნიშვნელოვანია ერთმანეთისაგან განვასხვავოთ ტერმინები პარამეტრი და არგუმენტი. **პარამეტრი** ესაა მეთოდის მიერ განსაზღვრული ცვლადი, რომელიც მნიშვნელობას ღებულობს მეთოდის გამოძახებისას. მაგალითად, square() მეთოდში პარამეტრი არის i. არგუმენტი ესაა მნიშვნელობა, რომელიც გადაეცემა მეთოდს მისი გამოძახებისას. მაგალითად, square(100) მეთოდი არგუმენტად გადასცემს 100. მეთოდის შიგნით i პარამეტრი მიიღებს მის მნიშვნელობას.

პარამეტრებიანი მეთოდის გამოყენებით შესაძლებელია Box კლასის გაუმჯობესება. წინა მაგალითებში თითოეული

პარალელეპიპედის ზომებს მნიშვნელობებს ცალცალკე ვანიჭებდით:

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

მართალია ეს კოდი სწორად მუშაობს, მაგრამ იგი მოუხერხებელია ორი მიზეზის გამო. პირველი - იგი დიდია და ამიტომ შეიძლება შეცდომის წყარო იყოს, მაგალითად, შეიძლება გამოგვრჩეს რომელიმე ზომის ინიციალიზაცია. მეორე - კარგად დაწერილ Java პროგრამაში ეგზემპლარის ცვლადებზე მიმართვა უნდა ხდებოდეს მხოლოდ ამ კლასში განსაზღვრული მეთოდების საშუალებით. შემდგომში მეთოდის მოქმედებები შეიძლება შეიცვალოს, მაგრამ არ შეიძლება ეგზემპლარის ღია ცვლადის მოდიფიკატორის ტიპის შეცვლა. ამიტომ პარალელეპიპედის ზომების საწყისი მნიშვნელობების მინიჭების უფრო რაციონალური მიდგომაა ისეთი მეთოდის შექმნა, რომელიც ზომებს მიიღებს თავისი პარამეტრების სახით და ეგზემპლარის ყოველ ცვლადს შესაბამისად მიანიჭებს მნიშვნელობას. ეს აზრი განხორციელებულია შემდეგ პროგრამაში:

ლისტინგი 49. პროგრამაში გამოყენებულია პარამეტრებიანი მეთოდი

```
class Box {
    double width;
    double height;
    double depth;
// მოცულობის გამოთვლა და დაბრუნება
    double volume() {
```

```
        return width * height * depth;
    }
// პარალელეპიპედის ზომების ინიციალიზაცია
    void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}
class BoxDemo5 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vo1;
// Box-ის ეგზემპლარების ინიციალიზაცია
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);
// მოცულობების გამოთვლა
        vo1 = mybox1.volume();
        System.out.println("Volume is " + vo1);
        vo1 = mybox2.volume();
        System.out.println("Volume is " + vo1);
    }
}
```

როგორც ხედავთ, setDim() მეთოდი გამოყენებულია ზომების დასაყენებლად. მაგალითად, ასეთი ოპერატორის შესრულებისას:

```
mybox1.setDim(10, 20, 15);
```

მნიშვნელობა 10 მიენიჭება w პარამეტრს, 20 მიენიჭება h-ს, 15 – d-ს. შემდეგ setDim() მეთოდის შიგნით w, h და d-ს მნიშვნელობები მიენიჭებათ შესაბამისად width, height და depth ცვლადებს.

კლასის წევრების ინიციალიზაცია

ზოგჯერ, Java-ში ცვლადის გამოყენებამდე მისი ინიციალიზაციის გარანტია ირღვევა. მეთოდში ლოკალურად გამოცხადებული ცვლადის შემთხვევაში ეს გარანტია გამოხატულია კომპილატორის შეცდომის შესახებ შეტყობინების სახით. მაგალითად, ამ ფრაგმენტის გამოყენების მცდელობისას

```
void f() {
    int i;
    i++; // შეცდომა - ცვლადი i არაა ინიციალიზებული
}
```

კომპილატორი გამოიმუშავებს შეცდომის შეტყობინებას, რომელიც მიუთითებს, რომ ცვლადი i არაა ინიციალიზებული. შეიძლებოდა, რომ კომპილატორს მოეხდინა ასეთი ცვლადების სტანდარტული მნიშვნელობით ინიციალიზაცია, მაგრამ ასეთი სიტუაცია პროგრამისტის შეცდომას უფრო ჰგავს და ასეთი მიდგომა, ამ შეცდომის გამოაშკარავებას ხელს შეუშლიდა. კომპილატორის გამოცემული შეცდომის შეტყობინება პროგრამისტს აიძულებს მოახდინოს ცვლადის ინიციალიზაცია, ანუ თავიდან აიცილოს შეცდომა პროგრამაში.

თუ კლასის ველი (კლასის ცვლადი) პრიმიტიული ტიპია, მასთან მიმართვის ხერხიც ცოტა სხვაგვარია. კლასის პრიმიტიული ტიპის ველს გარანტირებულად მიენიჭება სტანდარტული მნიშვნელობა. ეს ფაქტი ნაჩვენებია შემდეგ პროგრამაში:

ლისტინგი 50. სტანდარტულად მინიჭებული მნიშვნელობები ბეჭდვა

```
class InitialValues {
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    void printInitialValues() {
        System.out.println("მონაცემთა ტიპი     საწყისი
მნიშვნელობა");
        System.out.println("boolean                    "+t);
        System.out.println("char     ["+c+"]");
        System.out.println ("byte     "+b);
        System.out.println ("short    "+s);
        System.out.println ("int      "+i);
        System.out.println ("long     "+l);
        System.out.println ("float    "+f);
        System.out.println ("double "+d);
    }
    public static void main(String[] args) {
        InitialValues iv = new InitialValues();
        iv.printInitialValues();
    }
}
```

კონსოლზე გამოვა:

```
მონაცემთა ტიპი     საწყისი მნიშვნელობა
boolean             false
char                [ ]
byte                0
```

short	*0*
int	*0*
long	*0*
float	*0.0*
double	*0.0*

მიაქციეთ ყურადღება, რომ თუ ცვლადს მნიშვნელობა ცხადად არ მიენიჭება, ის ავტომატურად ინიციალიზდება. სიმბოლურ ცვლადს მიენიჭება 0, რომელიც ცარიელი სიმბოლოს სახით აისახება. ამ შემთხვევაში არა არის საშიშროება, რომ არაინიციალიზებული ცვლადი იქნეს გამოყენებული.

კონსტრუქტორები

კლასის თითოეული ეგზემპლარის შექმნისას, კლასის ყოველი ცვლადის ინიციალიზაცია საკმაოდ შრომატევადი პროცესი შეიძლება იყოს. setDim() მეთოდის დამატება მართალია, გარკვეულწილად აადვილებს ამ პროცესს, მაგრამ უფრო ადვილი და მოსახერხებელი იქნებოდა, თუ ცვლადების დაყენების მოქმედებები ობიექტის შექმნისას შესრულდებოდა, რადგან ინიციალიზაციის საჭიროება ხშირია. Java ობიექტებს საშუალებას აძლევს მოახდინონ საკუთარი ცვლადების ინიციალიზაცია მათი შექმნისას. ეს ავტომატური ინიციალიზაცია ხორციელდება კონსტრუქტორის საშუალებით.

კონსტრუქტორი ობიექტის ინიციალიზაციას ახდენს მისი შექმნისთანავე. მისი სახელი ემთხვევა იმ კლასის სახელს,

რომელშიც იგი იმყოფება, ხოლო სინტაქსი მეთოდის ანალოგიური აქვს. როგორც კი მოხდება კონსტრუქტორის გამოცხადება, იგი ავტომატურად გამოიძახება უშუალოდ ობიექტის შექმნის შემდეგ, new ოპერაციის დასრულებამდე. კონსტრუქტორები ცოტა უჩვეულოდ გამოიყურებიან, რადგან მათ არ გააჩნიათ დაბრუნების ტიპი და არც void-ი. ეს იმითაა გამოწვეული, რომ კლასის კონსტრუქტორის არაცხადად განსაზღვრული ტიპი არის თვით კლასის ტიპი. სწორედ, კონსტრუქტორის კოდი ახდენს ობიექტის შიგა მდგომარეობის ინიციალიზაციას ისე, რომ შექმნილი ეგზემპლარი მთლიანად ინიციალიზებული და გამოყენებისათვის ვარგისი იყოს.

Box კლასის მაგალითში, კონსტრუქტორის დამატებით, პარალელეპიპედის ზომების თავიდანვე ინიციალიზდება. ჯერ მარტივი კონსტრუქტორი განვიხილოთ, რომელიც ერთნაირ მნიშვნელობებს აყენებს ყველა პარალელეპიპედი-სათვის:

ლისტინგი 51. Box კლასი იყენებს კონსტრუქტორს ინიციალიზაციისათვის

```
class Box {
    double width;
    double height;
    double depth;
// Box კლასის კონსტრუქტორს
    Box () {
        System.out.println("Box-ის ობიექტის კონსტრუირება");
        width = 10;
        height = 10;
        depth = 10;
```

```
    }
// მოცულობის გამოთვლა და დაბრუნება
    double volume() {
        return width * height * depth;
    }
}
class BoxDemo6 {
    public static void main(String args[]) {
// Box-ის ობიექტების გამოცხადება და ინიციალიზაცია
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
// მოცულობების გამოთვლა
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        vol= mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

ამ პროგრამის შესრულების შედეგი კონსოლზე ასეთია:

Box-ის ობიექტის კონსტრუირება
Box-ის ობიექტის კონსტრუირება
Volume is 1000.0
Volume is 1000.0

როგორც პროგრამიდან ჩანს, mybox1 და mybox2 ინიციალიზებული იქნა Box() კონსტრუქტორის მიერ მისი შექმნისას. ვინაიდან, კონსტრუქტორი ყველა პარალელეპიპედს ერთნაირ ზომას ანიჭებს 10x10x10, mybox1-ს და mybox2-ს მოცულობები ერთნაირი ექნებათ. Box() კონსტრუქტორში println() ოპერატორს მხოლოდ ილუსტრაციის ფუნქცია აქვს და საერთოდ,

კონსტრუქტორებს, როგორც წესი, არავითარი ინფორმაცია არ გამოაქვთ, ისინი ობიექტის ინიციალიზაციას ასრულებენ.

კიდევ ერთხელ განვიხილოთ ოპერაცია new. ობიექტისათვის მეხსიერების გამოყოფისათვის გამოიყენება შემდეგი ზოგადი ფორმა:

<კლასის_ცვლადი> = new <კლასის_სახელი>();

ახლა ჩვენთვის უკვე ცხადია, რატომ არის საჭირო კლასის სახელის შემდეგ მრგვალი ფრჩხილები. სინამდვილეში, ეს ოპერატორი აკეთებს კონსტრუქტორის გამოძახებას:

Box mybox1 = new Box();

თუ კლასის კონსტრუქტორი ცხადად არაა გამოცხადებული (აღწერილი), Java კლასისათვის თვითონ ქმნის კონსტრუქტორს, რომელიც გამოყენებული იქნება ნაგულისხმევი წესით (სტანდარტულად). სწორედ, ამიტომ ჩვენს მიერ განხილული მაგალითის წინა ვერსიებში ეს სტრიქონი მუშაობდა სტანდარტულად, ვინაიდან Box კლასში კონსტრუქტური ცხადად გამოცხადებული არ იყო. სტანდარტულად გამოძახებული კონსტრუქტორი ახდენს ეგზემპლარის ყველა ცვლადს ინიციალიზაციასნულოვანი მნიშვნელობებით. ხშირად, სტანდარტული კონსტრუქტორი საკმარისია მარტივი კლასებისათვის. თუ მოხდა კონსტრუქტორის გამოცხადება (აღწერა), მაშინ სტანდარტული კონსტრუქტორის ავტომატური გამოძახება აღარ მოხდება.

პარამეტრებიანი კონსტრუქტორი

წინა მაგალითში, მართალია Box() კონსტრუქტორი ობიექტის ინიციალიზაციას ახდენდა, მაგრამ იგი დიდად გამოსადეგი არ იყო, ვინაიდან ყველა პარალელეპიპედის ზომები ერთნაირი იყო, ამიტომ საჭიროა ისეთი კონსტრუქტორი, რომელიც ნებისმიერ ზომებს დააყენებს. ამ საკითხის უმარტივესი გადაწყვეტა არის კონსტრუქტორზე პარამეტრების დაატება:

ლისტინგი 52. Box კლასი იყენებს პარამეტრიან კონსტრუქტორს

```
class Box {
    double width;
    double height;
    double depth;
//  Box კლასის კონსტრუქტორი
    Box(double w, double h, double d) {
     width = w;
     height = h;
     depth = d;
    }
// მოცულობის გამოთვლა და დაბრუნება
    double volume() {
     return width * height * depth;
    }
}
class BoxDemo7 {
    public static void main(String args[]) {
// Box-ის ობიექტების გამოცხადება და ინიციალიზაცია
     Box mybox1 = new Box(10, 20, 15);
     Box mybox2 = new Box(3, 6, 9);
     double vol;
// მოცულობების გამოთვლა
```

```
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);
    vol = mybox2.volume();
    System.out.println ("Volume is " + vol);
   }
}
```

ამ პროგრამის შედეგად კონსოლზე გამოვა:

Volume is 3000.0
Volume is 162.0

როგორც პროგრამიდან ჩანს, თითოეული ობიექტის ინიციალიზაცია ხდება კონსტრუქტორის პარამეტრებში მითითებული მნიშვნელობებით. მაგალითად,

```
Box mybox1 = new Box(10, 20, 15);
```

ამ სტრიქონით მნიშვნელობები 10, 20 და 15 new ოპერაციით ობიექტის შექმნის დროს, გადაეცემა Box() კონსტრუქტორს. ამრიგად, width, height და depth ცვლადების კოპიები მიიღებენ მნიშვნელობად 10, 20 და 15.

this -ის გამოყენება

ხანდახან საჭიროა მეთოდმა მიმართოს მის გამომძახებელ ობიექტს. ეს შესაძლებელი რომ იყოს, Java-ში შემოღებულია საკვანძო სიტყვა this. ის შეიძლება გამოყენებული იქნეს ნებისმიერი მეთოდის შიგნით მიმდინარე ობიექტის ელემენტებზე მიმართვისათვის. ანუ, this-ი ყოველთვის არის იმ ობიექტზე მითითება, რომლისთვისაც მოხდა მეთოდის გამოძახება. this საკვანძო სიტყვა შეგვიძლია გამოვიყენოთ

ყველგან, სადაც დასაშვებია მიმდინარე კლასის ტიპის ობიექტზე მიმართვა.

განვიხილოთ Box() კონსტრუქტორის ახალი მაგალითი:

```
// this-ის გამოყენება ზედმეტია
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this. depth = d;
}
```

კოსტრუქტორის ეს ვერსია წინა ვერსიის ანალოგიურად მოქმედებს. ამ შემთხვევაში this-ის გამოყენება ზედმეტია, მაგრამ სრულიად კორექტულია. Box() მეთოდის შიგნით this ყოველთვის მიუთითებს გამომძახებელ ობიექტს.

ეგზემპლარის ცვლადის გადაფარვა

Java-ში დაუშვებელია ერთ განსაზღვრის არეში ან მის ქვეარეში ორი ისეთი ცვლადის გამოცხადება, რომლებსაც ერთიდაიგივე სახელი აქვთ. არადა, შეიძლება არსებობდეს ლოკალური ცვლადები, მათ შორის მეთოდების ფორმალური პარამეტრები, რომლის სახელებიც ემთხვევა კლასის ეგზემპლარის ცვლადის სახელებს. ასეთ შემთხვევაში ამბობენ, რომ ლოკალური ცვლადი გადაფარავს (ფარავს) ეგზემპლარის ცვლადს, ხოლო თვით ამ შემთხვევას ცვლადების გადაფარვას უწოდებენ.

სწორედ, ამიტომ Box კლასის შიგნით width, height და depth სახელები არ იქნა გამოყენებული Box() კონსტრუქტორის

პარამეტრების სახელებად. წინააღმდეგ შემთხვევაში width ცვლადი მიმართავდა ფორმალურ პარამეტრს, რომელიც გადაფარავს ეგზემპლარის ცვლადს. მართალია, ხშირად უფრო ადვილია სხვადასხვა სახელების ხმარება, მაგრამ არსებობს ასეთი სიტუაციიდან სხვა გამოსავალიც. ვინაიდან, this მიმართავს უშუალოდ ობიექტს, ეს შეიძლება გამოვიყენოთ სახელების განსაზღვრის არეში წარმოქმნილი ასეთი კონფლიქტების გადასაწყვეტად. მაგალითად, ქვემოთ ნაჩვენებია Box() კონსტრუქტორის კიდევ ერთი ვერსია, სადაც width, height და depth სახელები გამოყენებულია პარამეტრების სახელებად. ამ შემთხვევაში, this გამოყენებულია ეგზემპლარის ცვლადებზე მიმართვისათვის, რომლებსაც იგივე სახელები ქვიათ:

```
// სახელების განსაზღვრის არის კონფლიქტის მოგვარება
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
this.depth  depth;
}
```

გასათვალისწინებელია, რომ ზოგჯერ this-ის ამგვარმა გამოყენებამ შეიძლება გაურკვევლობა გამოიწვიოს, ამიტომ ზოგიერთი პროგრამისტი ცდილობს არ გამოიყენოს ერთნაირი სახელები ლოკალური ცვლადებისა და პარამეტრებისათვის, რათა არ მოხდეს გადაფარვა. ზოგიერთი პროგრამისტი, კი პირიქით თვლის, რომ პროგრამის აღქმის გასაადვილებლად უმჯობესია გამოყენებული იქნეს ერთიდაიგივე სახელები, ხოლო

გადაფარვის დასაძლევად გამოყენებული იქნეს this. ამ ორი მიდგომიდან არჩევანი გემოვნების საკითხია.

„ნაგვის“ შეგროვება

ვინაიდან ობიექტებისათვის მეხსიერების გამოყოფა new ოპერაციის შედეგად დინამიკურად ხდება, შეიძლება დაისვას საკითხი: როგორ უქმდება ობიექტები და როგორ თავისუფლდება გამოყოფილი მეხსიერება მისი შემდგომი განაწილებისათვის. ზოგიერთ ენაში (მაგალითად, C++) დინამიურად გამოყოფილი მეხსიერება პროგრამისტმა თვითონვე უნდა გაანთავისუფლოს შესაბამისი ოპერაციით (მაგალითად, delete). Java-ში სხვა მიდგომაა გამოყენებული. მეხსიერების განთავისუფლება ავტომატურად ხდება. ამ ამოცანის შესასრულებლად გამოყენებულ ტექნოლოგიას ეწოდება **ნაგვის შეგროვება**. პროცესი ასე ხორციელდება: თუ ობიექტზე არავითარი მიმართვა არ არსებობს, პროგრამა ასკვნის, რომ ეს ობიექტი საჭირო აღარაა და მის მიერ დაკავებული მეხსიერება შეიძლება განთავისუფლდეს. Java-ში ობიექტების ცხადად განადგურება საჭირო არაა. პროგრამის შესრულების პერიოდში ნაგვის შეგროვება იშვიათად ხდება (თუ საერთოდ მოხდა). იმის გამო, რომ ერთი ან რამდენიმე ობიექტი საჭირო აღარა არის, ეს პროცესი არ ჩატარდება. Java-ს შემსრულებელი გარემოს სხვადასხვა რეალიზაციაში ამ საკითხზე სხვადასხვა მიდგომა აქვს. მთავარია, რომ პროგრამისტი ამ პროცესისგან თავისუფალია და მას, არ აინტერესებს სისტემა როდის მოახდენს ნაგვის შეგროვების პროცესის ინიცირებას.

ზოგჯერ, ობიექტის განადგურების დროს გარკვეული მოქმედებები უნდა შესრულდეს. მაგალითად, თუ ობიექტი იყენებს რაიმე რესურსს (მაგალითად, რომელიმე ფაილის დესკრიპტორს, შრიფტს), შეიძლება საჭირო იყოს იმის გარანტია, რომ ობიექტის განადგურებამდე მოხდეს რესურსების განთავისუფლება. ასეთი სიტუაციებისათვის Java-ში გათვალისწინებულია მექანიზმი, რომელსაც ფინალიზაცია ეწოდება. ფინალიზაციის გამოყენებით, შეიძლება განისაზღვროს კონკრეტული მოქმედებები, რომლის შესრულება უნდა მოხდეს ნაგვის შეგროვების ოპერაციის ჩატარებამდე.

კლასში ფინალიზაციის საშუალების დასამატებლად საჭიროა გამოვაცხადოთ finalize() მეთოდი. Java-ს შემსრულებელი გარემო, ამ მეთოდს გამოიძახებს ობიექტის განადგურების წინ. ამ მეთოდის ტანში უნდა მივუთითოთ ყველა ის მოქმედება რისი ჩატარებაც ობიექტის განადგურებამდე არის საჭირო.

ამ მეთოდის ზოგადი ფორმა ასეთია:

```
protected void finalize( ) {

// აქ უნდა განთავსდეს ფინალიზაციის კოდი

}
```

ამ აღწერაში საკვანძო სიტყვა protected ესაა სპეციფიკატორი (აღმწერი), რომელიც კრძალავს კლასის გარე კოდისაგან finalize() მეთოდზე მიმართვას.

უნდა გვახსოვდეს, რომ მეთოდი finalize() გამოიძახება უშუალოდ ობიექტის განადგურების წინ, ამიტომ უცნობია როდის მოხდება (და მოხდება თუ არა საერთოდ) ამ მეთოდის შესრულება. სწორედ, ამიტომ პროგრამამ სხვა გზებით უნდა მოახდინოს სისტემური რესურსების განთავისუფლების უზრუნველყოფა. პროგრამის ნორმელური მუშაობა არ უნდა იყოს დამოკიდებული finalize() მეთოდზე.

კლასი Stack

მართალია, Box კლასის განხილვა მოსახერხებელია კლასის ძირითადი ელემენტების საილუსტრაციოდ, მაგრამ მას დიდი პრაქტიკული ღირებულება არა აქვს. განვიხილოთ ცოტა უფრო რთული მაგალითი. როდესაც ობიექტზე ორიენტირებულ მიდგომას განვიხილავდით, მაშინ აღვნიშნეთ, რომ მონაცემებისა და იმ კოდის ინკაფსულაცია ერთერთი ძირითადი პრინციპია, რომელიც ამ მონაცემებით მანიპულირებს. შემდეგ ვთქვით, რომ ინკაფსულაციის მიღწევის მექანიზმი Java-ში ესაა - კლასი. კლასის შექმნით პროგრამისტი ქმნის მონაცემთა ახალ ტიპს, სადაც აღწერილია, როგორც მონაცემების არსი (შინაარსი), ანუ რომლებსაც ჩაუტარდებათ მანიპულირება, ასევე ამ მანიპულირების ჩამტარებელი პროცედურები. პროცედურები კლასში ფორმდება მეთოდების სახით, რომლებიც განსაზღვრავენ კლასის ცვლადებთან სრულ და მართვად ინტერფეისს. ამრიგად, კლასი შეიძლება გამოყენებული იქნეს მისი მეთოდების საშუალებით ისე, რომ არ ვიზრუნოთ (არ დავწვრილმანდეთ) მისი

რეალიზაციის ნიუანსებზე ან კლასის შიგნით მონაცემების მართვის მიმდინარე ხერხებზე. გარკვეულწილად კლასი მსგავსია „მონაცემების მანქანის“. მანქანის სამართავად არაა აუცილებელი მის შიგნით მიმავალი პროცესების ცოდნა (მაგალითად, შიგაწვის ძრავის ან ტრანსმისიის რეალიზაციის დეტალები). ფაქტობრივად, რეალიზაციის დეტალები დაფარულია, შიგა წვრილმანები საჭიროების მიხედვით შეიძლება შევცვალოთ. ე.ი. თუ კლასის გამოყენება ხდება მხოლოდ მისი მეთოდების საშუალებით, შიგა დეტალები შეიძლება იცვლებოდეს და ეს ცვლილება რაიმე გვერდით მოვლენებს კლასის გარეთ არ გამოიწვევს.

ამ მსჯელობის საილუსტრაციოდ, ინკაფსულაციის ერთერთი ტიპური მაგალითი სტეკი განვიხილოთ. **სტეკი** მონაცემებს ინახავს პრინციპით: „პირველი შემოსული ბოლოს გადის“. ანუ სტეკი მსგავსია ერთმანეთზე დალაგებული თეფშების სვეტის. თეფში, რომელიც პირველი დაიდო, მისი აღება მოხდება ბოლოს. სტეკის სამართავად ორი ოპერაციაა საჭირო, რომლებსაც ტრადიციულად უწოდებენ სტეკში „ჩაგდებას“ (push) და სტეკიდან „ამოგდებას“ (pop). სტეკის თავში ელემენტის მოსათავსებლად იყენებენ push ოპერაციას, ხოლო სტეკიდან ელემენტის მისაღებად pop ოპერაციას. სტეკის სრული მექანიზმის ინკაფსულაცია ძნელი არაა. ქვემოთ ნაჩვენებია Stack კლასის კოდი, რომელიც მთელრიცხვა სტეკის რეალიზაციას აკეთებს:

ლისტინგი 53. კლასში აღწერილია მთელრიცხვა სტეკი, რომელსაც 10 ელემენტის შენახვა შეუძლია

```
class Stack {
```

```
    int stck[] = new int[10];
    int tos;
    // სტეკის წვეროს ინიციალიზაცია
    Stack () {
        tos =- 1;
    }
// ელემენტის სტეკში ჩაგდება
    void push(int item) {
        if (tos == 9)
        System.out.println("სტეკი სავსეა ");
        else
        stck[++tos] = item;
    }

// სტეკიდან ელემენტის ამოგდება
    int pop () {
        if(tos<0) {
        System.out.println("სტეკი ცარიელია ");
        return 0;
        }
        else
        return stck[tos--];
    }
}
```

Stack კლასში აღწერილია მონაცემთა ორი ელემენტი და სამი მეთოდი. მთელრიცხვა სტეკის ელემენტები ინახება stck მასივში. ამ მასივის ინდექსირებისათვის გამოყენებულია ცვლადი tos, რომლის მნიშვნელობა ყოველთვის მიუთითებს სტეკის წვეროს (ანუ თეფშების მაგალითში ყველაზე ზედა თეფშის ინდექსს). Stack() კონსტრუქტორი tos-ის ინიციალიზაციას აკეთებს მნიშვნელობით -1, რაც მიუთითებს, რომ სტეკი ცარიელია. სტეკთან მიმართვა ხდება push() და pop() მეთოდებით, ამიტომ, ის ფაქტი, რომ მონაცემები მასივში

ინახება თუ სხვა უფრო რთულ მონაცემთა სტრუქტურაში (მაგალითად, ცალმხრივ მიმართულ სიაში), სტეკთან მუშაობისათვის მნიშვნელობა არ აქვს. სტეკთან მუშაობის ინტერფეისი მაინც push() და pop() მეთოდებით ხორციელდება.

შემდეგ მაგალითში ნაჩვენებ კლასში TestStack დემონსტრირებულია Stack კლასის გამოყენება. იქმნება ორი სტეკი, გარკვეული მნიშვნელობები ჩაიდება სტეკში და მერე ამოიღება იქედან.

ლისტინგი 54.

```
class TestStack {
    public static void main(String args[]) {
        Stack  mystack1 = new Stack();
        Stack mystack2 = new Stack();
// რიცხვების სტეკში ჩაყრა
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);
// სტეკიდან რიცხვების ამოღება
        System.out.println("სტეკიდან mystack1: ");
        for(int i=0; i<10; i++)
         System.out.println(mystack1.pop());
        System.out.println("სტეკიდან mystack2: ");
        for(int i=0; i<10; i++)
         System.out.print1n(mystack2.pop());
    }
}
```

ამ პროგრამის შესრულების შედეგად კონსოლზე გამოვა: -

სტეკიდან mystack1:

9

8

7
6
5
4
3
2
1
0
სტეკიდან mystack2: 19
18
17
16
15
14
13
12
11
10

როგორც ხედავთ, სტეკების შემცველობები (მნიშვნელობები) განსხვავებულია. უნდა აღვნიშნოთ, რომ ამ სახით რეალიზებულ სტეკში, stck მასივზე მიმართვა შესაძლებელია კლასის გარე კოდიდანაც, ამიტომ იგი დაუცველია გარე დაზიანებებისა და ბოროტად გამოყენებისაგან. შემდგომში ჩვენ გავარჩევთ, როგორ შეიძლება ამ სიტუაციის გამოსწორება.

მეთოდების გადატვირთვა

Java-ში დაშვებულია ერთი კლასის შიგნით რამდენიმე მეთოდის აღწერა ერთი და იგივე სახელით, ოღონდ მათი პარამეტრების გამოცხადება განსხვავებული უნდა იყოს. ამ შემთხვევაში მეთოდებს უწოდებენ გადატვირთულს, ხოლო პროცესს - მეთოდების გადატვირთვას. მეთოდების გადატვირთვა ობიექტზე ორიენტირებული პროგრამირების ერთ-ერთი ძირითადი პრინციპის - პოლიმორფიზმის უზრუნველყოფის გამოხატულებაა.

შეიძლება კითხვა დაისვას: თუ რამდენიმე მეთოდს ერთი და იგივე სახელი აქვთ, მაშინ ამ სახელის მეთოდის გამოძახებისას Java-ს შემსრულებელი გარემო რომელ მეთოდს გამოიძახებს? ან რა მექანიზმია პროგრამისტის ხელში, რომ იმ მეთოდის გამოძახება მოახდინოს, რომელიც მას სჭირდება? ამ საკითხის გადასაწყვეტად Java-ში ასეთი მექანიზმია რეალიზებული:

გადატვირთული მეთოდის გამოძახებისას, საჭირო მეთოდის ვერსიის გამოსაძახებლად Java იყენებს მეთოდის არგუმენტების რაოდენობას და/ან ტიპს. შესაძლოა გადატვირთული მეთოდების მიერ დაბრუნებული მნიშვნელობის ტიპები სხვადასხვა იყოს, მაგრამ თვით დაბრუნებული ტიპის განსხვავებულობა არაა საკმარისი მეთოდის სხვადასხვა ვერსიის გასარჩევად. როდესაც Java-ს ხვდება გადატვირთული მეთოდის გამოძახება, იგი ასრულებს იმ ვერსიას, რომლის პარამეტრებიც შეესაბამება

გამოძახებული მეთოდის არგუმენტებს. საილუსტრაციოდ განვიხილოთ მარტივი პროგრამა:

ლისტინგი 55. მეთოდების გადატვირთვის დემონსტრირება

```
class OverloadDemo {
    void test () {
        System.out.println("პარამეტრები არაა");
    }
//  გადატვირთვის შემოწმება ერთი მთელრიცხვა
//  პარამეტრის არსებობაზე
    void test(int a) {
        System.out.println("a: " + a);
    }
//  გადატვირთვის შემოწმება ორ მთელრიცხვა
//  პარამეტრის არსებობაზე
    void test(int a, int b) {
        System.out.println ("a და b: " + a + " " + b);
    }
// double ტიპის პარამეტრის არსებობის შემოწმება
    double test(double a) {
        System.out.println ("double a: " + a);
        return a*a;
    }
}
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;
// test () მეთოდის ყველა ვრსიის გამოძახება
        ob. test ();
        ob.test(10);
        ob.test(10,20);
        result = ob.test(123.25);
        System.out.println("ob.test(123.25)-ის შედეგი: " + result);
    }
```

}
ამ პროგრამის შედგი კონსოლზე:

პარამეტრები არაა
a: 10
a და b: 10 20
double a: 123.25
ob.test(123.25)-ის შედეგი: 15190.5625

test() მეთოდი გადაიტვირთება ოთხჯერ. პირველ ვერისიას პარამეტრები არ აქვს; მეორე ერთ მთელრიცხვა პარამეტრს ღებულობს; მესამე - ორ მთელრიცხვა პარამეტრს; მეოთხე - ერთ double ტიპის პარამეტრს. ის, რომ მეოთხე ვერსია შედეგსაც აბრუნებს, არავითარი მნიშვნელობა არ აქვს გადატვირთვისათვის, ვინაიდან, დაბრუნებული ტიპი არანაირად არ მოქმედებს გადატვირთული მეთოდის ვერსიის არჩევაზე.

გადატვირთული მეთოდის გამოძახების დროს, Java ეძებს შესაბამისობას მეთოდის გამოძახებისას მითითებულ არგუმენტებსა და მეთოდის პარამეტრებს შორის. ეს შესაბამისობა აუცილებელი არ არის სრული იყოს. ზოგჯერ გადატვირთვის ნებართვისათვის შეიძლება გამოყენებული იქნეს ტიპების ავტომატური დაყვანის მექანიზმი. მაგალითად,

ლისტინგი 56. ტიპების ავტომატური გარდაქმნა გადატვირთვის დროს

```
class OverloadDemo {
    void test () {
     System.out.println ("პარამეტრები არაა") ;
```

```
    }
//  გადატვირთვის შემოწმება ორ მთელრიცხვა
//  პარამეტრის არსებობაზე
    void test(int a, int b) {
     System.out. println ("a და b: " + a + " " + b);
    }
//  გადატვირთვის შემოწმება double ტიპის
//  პარამეტრის არსებობაზე
    void test(double a) {
     System.out. println ("შიგა გარდაქმნა test(double) a: "+ a);
    }
}
class Overload {
    public static void main(String args[]) {
     OverloadDemo ob = new OverloadDemo();
     int i = 88;
     ob. test ();
     ob.test(10, 20);
     ob.test(i);      // ეს ოპერატორი იძახებს test(double)
     ob.test(123.2);          // ეს ოპერატორი იძახებს test(double)
    }
}
```

პროგრამის შედეგი კონსოლზე:

პარამეტრები არაა
a და b: 10 20
შიგა გარდაქმნა test(double) a: 88
შიგა გარდაქმნა test(double) a: 123.2

როგორც პროგრამიდან ჩანს, ამ ვერსიაში test(int) გადატვირთვა განსაზღვრული არაა. ამიტომ, თუ მოხდება test() მეთოდის გამოძახება მთელრიცხვა არგუმენტით

Overload კლასში მისი შესაბამისი მეთოდი არ დახვდება. Java-ს შეუძლია int ტიპი ავტომატურად გარდაქმნას double ტიპად და ამ ფაქტის გამო, შესაძლებელია გადაწყდეს რომელი მეთოდი იქნეს გამოძახებული. ამიტომ, როდესაც ვერ მოხდება test(int) მეთოდის ვერიის მოძებნა, Java აამაღლებს i-ს ტიპს double-მდე და შემდეგ გამოიძახებს test(double) მეთოდს. ცხადია test(int) მეთოდი განსაზღვრული, რომ ყოფილიყო მოხდებოდა მისი გამოძახება. Java ტიპების ავტომატურად გარდაქმნას იყენებს მხოლოდ იმ შემთხვევაში, თუ არ მოხდა სრული შესაბამისობა.

მეთოდების გადატვირთვა უზრუნველყოფს პოლიმორფიზმს და იგი არის Java-ს ერთ-ერთი კონცეფციის „ერთი ინტერფეისი, რამდენიმე მეთოდი“ რეალიზაციის ხერხი (საშუალება). იმ ენებში, სადაც მეთოდების გადატვირთვის მექანიზმი არაა რეალიზებული, თითოეულ მეთოდს უნდა ჰქონდეს უნიკალური (განსხვავებული) სახელი. არადა, ხშირად ერთი და იგივე პროცედურის განხორციელება საჭიროა ხოლმე სხვადასხვა ტიპის მონაცემზე. მაგალითად, განვიხილოთ აბსოლუტური მნიშვნელობის გამოთვლის ფუნქცია. ენებში, სადაც გადატვირთვა დაშვებული არ არის, ამ ფუნქციის რამდენიმე ვერსია არსებობს, რომელთა სახელებიც ერთმანეთისაგან უმნიშვნელოდ, მაგრამ მაინც განსხვავდება. მაგალითად, C-ში abs()აბრუნებს მთელი ტიპის int მნიშვნელობას; labs() აბრუნებს long ტიპის მნიშვნელობას; fabs() აბრუნებს მცოცავმძიმიან მნიშვნელობას. ვინაიდან ეს ენა მეთოდების გადატვირთვას არ უშვებს, ყოველ ფუნქციას უნდა ჰქონდეს თავისი სახელი, მიუხედავად იმისა, რომ სამივე ფუნქცია პრაქტიკულად

ერთნაირ მოქმედებებს ატარებს. ასეთი ენებისათვის პროგრამისტი იძულებულია სამივე ფუნქციის სახელი დაიმახსოვროს. Java-ში ასეთი სიტუაცია არ წარმოიქმნება, ვინაიდან აბსოლუტური მნიშვნელობის ყველა მეთოდს ერთი და იგივე სახელი შეიძლება ჰქონდეს. მართლაც, Java-ს კლასების სტანდარტული ბიბლიოთეკა შეიცავს ასეთ მეთოდს, რომელსაც ჰქვია abs(). ამ მეთოდის გადატვირთვები ყველანაირი რიცხვითი ტიპის დასამუშავებლად განსაზღვრულია Math კლასში. Java საჭირო ვერსიის გამოძახებას ახდენს არგუმენტის ტიპის მიხედვით.

გადატვირთვა იმითაა ღირებული, რომ მსგავს მეთოდებს შეიძლება მივმართოთ ერთი და იგივე, საერთო სახელით. ამრიგად, სახელი abs შესასრულებელ საერთო მოქმედებას განსაზღვრავს. მოცემულ სიტუაციაში კონკრეტული ვერსიის არჩევა კომპილატორის საქმეა. პროგრამისტმა უნდა იცოდეს მხოლოდ საერთო შესასრულებელი მოქმედება. პოლიმორ-ფიზმი საშუალებას იძლევა რამდენიმე სახელი ერთზე დავიყვანოთ. მართალია, ნაჩვენები მაგალითი მარტივია, მაგრამ თუ ამ აზრს განვაზოგადებთ, დავასკვნით, რომ გადატვირთვამ შეიძლება გააადვილოს ბევრი რთული ამოცანის გადაწყვეტა.

მეთოდების გადატვირთვისას, თითოეულმა ვერსიამ შეიძლება შეასრულოს ნებისმიერი საჭირო მოქმედებები. არავითარი წესი არ არსებობს, რომლის მიხედვითაც გადატვირთული მეთოდები ერთმანეთთან რამენაირად იყვნენ დაკავშირებული. თუმცა, სტილისტიკური

თვალსაზრისით, მეთოდების გადატვირთვა გარკვეულ კავშირს გულისხმობს. მართალია, ერთი და იგივე სახელი შეიძლება გამოყენებული იქნეს ერთმანეთთან დაუკავშირებელი მეთოდების გადატვირთვისათვის, მაგრამ ასე მოქცევა არ ღირს. მაგალითად, სახელი sqr შეიძლება გამოყენებული იქნეს მთელი რიცხვის კვადრატის მნიშვნელობის დასაბრუნებლადაც და მცოცავმძიმიანი რიცხვის კვადრატული ფესვის მნიშვნელობის დასაბრუნებლადაც. მაგრამ ეს ორი ოპერაცია პრინციპულად განსხვავდება ერთმანეთისაგან. მეთოდების გადატვირთვის ასეთი გამოყენება ეწინააღმდეგება მის საწყის დანიშნულებას. კერძოდ, მხოლოდ მჭიდროდ დაკავშირებული მეთოდების გადატვირთვა უნდა მოხდეს.

კონსტრუქტორების გადატვირთვა

ჩვეულებრივი მეთოდების მსგავსად შესაძლებელია კონსტრუქტორების გადატვირთვაც. ბევრი რეალური კლასისათვის გადატვირთული კონსტრუქტორები უფრო ნორმაა, ვიდრე გამონაკლისი. ეს გამონათქვამი გასაგები, რომ იყოს მივუბრუნდეთ Box კლასს. ამ კლასის ბოლო ვერსიაში Box() კონსტრუქტორს სამი პარამეტრის გადაცემა ჭირდება. ეს ნიშნავს, რომ Box-ის ობიექტების ყველა გამოცხადებამ, კონსტრუქტორს უნდა გადასცეს სამი არგუმენტი. ასეთი ოპერატორი კი დაუშვებელია:

```
Box ob = new Box();
```

ვინაიდან, Box() კონსტრუქტორი სამ არგუმენტს მოითხოვს, მისი უარგუმენტო გამოძახება გამოიწვევს შეცდომას. ეს

სიტუაცია შემდეგ კითხვებს წარმოშობს: როგორ მოვიქცეთ, როდესაც უბრალოდ პარალელეპიპედის შექმნა გვინდა და გვერდების საწყის ზომას არავითარი მნიშვნელობა არ აქვს (ან არ ვიცით ეს მნიშვნელობები)? თუ კუბის შექმნა გვინდა არ უნდა შეგვეძლოს მხოლოდ ერთი მნიშვნელობის მითითება, რომელსაც მერე ყველა გვერდისთვის გამოიყენებდა? Box კლასის ძველი აღწერით ამის საშუალებები არ არსებობდა.

ასეთი ამოცანების გადაწყვეტა გადატვირთვის მექანიზმის გამოყენებით საკმაოდ ადვილია. საკმარისია მოვახდინოთ Box კლასის კონსტრუქტორის გადატვირთვა, რომელიც გაითვალისწინებს ყველა ჩამოთვლილ სიტუაციას. ქვემოთ ნაჩვენებია პროგრამა, რომელიც შეიცავს Box კლასის გაუმჯობესებულ ვერსიას:

ლისტინგი 57. Box კლასში ინიციალიზაციისათვის აღწერილია სამი კონსტრუქტორი

```
class Box {
    double width;
    double height;
    double depth;
// კონსტრუქტორი სამივე განზომილებისათვის
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
// უარგუმენტო კონსტრუქტორი
    Box () {
        width = -1;
        height = -1;
```

```
        depth = -1;
    }
// კონსტრუქტორი კუბის შესაქმნელად
    Box(double len) {
        width = height = depth = len;
    }
// მოცულობის გამოთვლა და დაბრუნება
    double volume() {
        return width * height * depth;
    }
}
class OverloadCons {
    public static void main(String args[]) {
// პარალელეპიპედის შექმნა სხვადასხვა კონსტრუქტორებით
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        double vol;
// მოცულობების გამოთვლა
        vol = mybox1.volume () ;
        System.out.println("Volume mybox1 is " + vol);
        vol = mybox2.volume();
        System.out.println("Volume mybox2 is " + vol);
        vol = mycube.volume();
        System. out. println ("Volume mycube is " + vol) ;
    }
}
```

კონსოლზე გამოვა:

Volume mybox1 is 3000.0

Volume mybox2 is -1.0

Volume mycube is 343.0

როგორც პროგრამიდან ჩანს, ოპერაცია new -ის შესრულებისას, შესაბამისი გადატვირთული

კონსტრუქტორის გამოძახება პარამეტრების მიხედვით ხდება.

კონსტრუქტორიდან კონსტრუქტორის გამოძახება

როდესაც კლასისათვის რამდენიმე კონსტრუქტორს ვწერთ, კოდის გამეორების (დუბლირების) თავიდან ასაცილებლად, სასურველია ერთი კონსტრუქტორიდან მოვახდინოთ მეორე კონსტრუქტორის გამოძახება. ასეთი ოპერაციის განხორციელება შესაძლებელია this-ის გამოყენებით.

ჩვეულებრივ, this-ის გამოყენებისას იგულისხმება „ეს ობიექტი“ ან „მიმდინარე ობიექტი“, თვით სიტყვა წარმოადგენს მიმდინარე ობიექტზე მითითებას. კონსტრუქტორში ამ სიტყვას სხვა შინაარსი აქვს: მისი არგუმენტების სიით გამოყენებისას ხდება იმ კონსტრუქტორის გამოძახება, რომელიც მოცემული სიის ელემენტების შესაბამისია. ამგვარად, შესაძლებელი ხდება სხვა კონსტრუქტორის პირდაპირი გამოძახება. მაგალითი:

ლისტინგი 58. Box კლასში ინიციალიზაციისათვის აღწერილია სამი კონსტრუქტორი

```
class Box {
    double width;
    double height;
    double depth;
// კონსტრუქტორი სამივე განზომილებისათვის
    Box(double w, double h, double d) {
        width = w;
        height = h;
```

```
        depth = d;
    }
// უარგუმენტო კონსტრუქტორი
    Box () {
        this (-1,-1,-1);
    }
// კონსტრუქტორი კუბის შესაქმნელად
    Box(double len) {
        this(len, len, len);
    }
// მოცულობის გამოთვლა და დაბრუნება
    double volume() {
        return width * height * depth;
    }
}
class OverloadCons {
    public static void main(String args[]) {
// პარალელეპიპედის შექმნა სხვადასხვა კონსტრუქტორებით
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        double vol;
// მოცულობების გამოთვლა
        vol = mybox1.volume () ;
        System.out.println("Volume mybox1 is " + vol);
        vol = mybox2.volume();
        System.out.println("Volume mybox2 is " + vol);
        vol = mycube.volume();
        System.out.println ("Volume mycube is " + vol) ;
    }
}
```

კონსოლზე გამოვა:

Volume mybox1 is 3000.0

Volume mybox2 is -1.0

Volume mycube is 343.0

ამ პროგრამაში უარგუმენტო და კუბის კონსტრუქტორის დასაწერად გამოყენებულია სამ არგუმენტიანი კონსტრუქტორის გამოძახება, რომელსაც პირველ შემთხვევაში სამივე არგუმენტად -1 გადაეცემა, ხოლო მეორე შემთხვევაში სამივე არგუმენტად len ერთი და იგივე ცვლადის მნიშვნელობა გადაეცემა.

this-ის ამ სახით გამოყენებას აქვს გარკვეული შეზღუდვები:

1. თუ კონსტრუქტორი this-ითაა გამოძახებული, მას მეორე კონსტრუქტორის გამოძახება ეკრძალება;

2. კონსტრუქტორიდან this-ით მეორე კონსტრუქტორის გამოძახება უნდა იყოს პირველი შესრულებადი ოპერატორი.

ობიექტების გამოყენება პარამეტრებად

აქამდე, ჩვენ მეთოდის პარამეტრებად, მხოლოდ მარტივ ტიპებს ვიყენებდით. თუმცა მეთოდებზე ობიექტების გადაცემა სრულიად დასაშვებია და საკმაოდ გავრცელებულია. მაგალითად,

ლისტინგი 59. მეთოდებს შეიძლება გადაეცეთ ობიექტი

```
class Test {
    int a, b;
    Test (int i, int j) {
     a = i;
     b = j;
    }
```

```
// აბრუნებს true-ს, თუ o ტოლია გამომძახებელი ობიექტის
    boolean equals(Test o) {
     if(o.a == a && o.b ==b) return true;
     else return false;
    }
}
class PassOb {
    public static void main(String args[]) {
     Test ob1 = new Test(100, 22);
     Test ob2 = new Test(100, 22);
     Test ob3 = new Test(1, 1);
     System.out.println("ob1== ob2:" + ob1.equals(ob2));
     System.out.println("ob1 == ob3: " + ob1.equals(ob3));
    }
}
```

კონსოლზე გამოვა:

ob1 == ob2: true

ob1 == ob3: false

როგორც ხედავთ, Test კლასის შიგნით equals() მეთოდი ამოწმებს ორი ობიექტის ტოლობას და აბრუნებს შემოწმების შედეგს. ანუ იგი ადარებს გამომძახებელ ობიექტს იმ ობიქტთან, რომელიც მას გადაეცა პარამეტრად. თუ ისინი ერთნაირ მნიშვნელობებს შეიცავენ, მეთოდი მნიშვნელობად აბრუნებს True-ს. წინააღმდეგ შემთხვევაში იგი აბრუნებს მნიშვნელობას false. მიაქციეთ ყურადღება, რომ equals() მეთოდში o პარამეტრს ტიპად მითითებული აქვს Test-ი. მართალია, ეს ტიპი პროგრამაში ჩვენს მიერ არის შექმნილი, მაგრამ იგი ზუსტად ისევე გამოიყენება, როგორც Java-ში არსებული (ჩაშენებული) ტიპები.

ობიექტები პარამეტრებად ძალიან ხშირად გამოიყენება კონსტრუქტორებში. ხშირად საჭიროა ობიექტის ისე შექმნა, რომ თავიდან ის არ განსხვავდებოდეს რომელიმე არსებული ობიექტისაგან. ამისათვის საჭიროა კონსტრუქტორს პარამეტრად მიეწოდოს შესაბამისი კლასის ობიექტი. მაგალითად, Box კლსის შემდეგი ვერსია საშუალებას იძლევა მოხდეს ერთი ობიექტის ინიციალიზაცია მეორე კლასის მიხედვით:

ლისტინგი 60. Box კლასში ერთი ობიექტის ინიციალიზაცია ხდება მეორეთი

```
class Box {
    double width;
    double height;
    double depth;
// კონსტრუქტორი სამივე განზომილებისათვის
    Box(Box ob){
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
// უარგუმენტო კონსტრუქტორი
    Box () {
        width = -1;
        height = -1;
        depth = -1;
    }
// კონსტრუქტორი კუბის შესაქმნელად
```

```
Box(double len) {
        width = height = depth = len;
    }
// მოცულობის გამოთვლა და დაბრუნება
    double volume() {
        return width * height * depth;
    }
}
class OverloadCons {
    public static void main(String args[]) {
// პარალელეპიპედის შექმნა სხვადასხვა კონსტრუქტორებით
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        Box myclone = new Box(mybox1);
        double vol;
// მოცულობების გამოთვლა
        vol = mybox1.volume () ;
        System.out.println("Volume mybox1 is " + vol);
        vol = mybox2.volume();
        System.out.println("Volume mybox2 is " + vol);
        vol = mycube.volume();
        System.out.println ("Volume mycube is " + vol) ;
        vol = myclone.volume();
        System.out.println ("Volume myclone is " + vol);
    }
}
```

როგორც ხედავთ, საკუთარი კლასის შექმნისას, ობიექტების შექმნა მოსახერხებელი და ეფექტური რომ იყოს, საჭიროა სხვადასხვა ფორმის კონსტრუქტორების დამუშავება.

არგუმენტების გადაცემის ორი მექანიზმი

ზოგადად, ორი მექანიზმი არსებობს, რომლის მიხედვითაც კომპიუტერულ ენას შეუძლია არგუმენტების გადაცემა ქვეპროგრამაზე. პირველი მექანიზმია - **გამოძახება მნიშვნელობის** მიხედვით. ამ მიდგომის დროს, არგუმენტის მნიშვნელობა კოპირდება ქვეპროგრამის ფორმალურ პარამეტრში. შესაბამისად, ქვეპროგრამის პარამეტრებზე განხორციელებული ცვლილებები, არ იწვევს რაიმე ცვლილებას არგუმენტებზე. არგუმენტების გადაცემის მეორე გზაა - **გამოძახება მითითების** მიხედვით. ამ მიდგომის დროს, პარამეტრს გადაეცემა არგუმენტის მისამართი, ანუ მასზე მითითება (და არა მისი მნიშვნელობა). ქვეპროგრამის შიგნით ეს მითითება გამოიყენება იმ რეალურ არგუმენტზე მიმართვისათვის, რომელიც მითითებულია გამოძახებაში. ანუ, ქვეპროგრამაში პარამეტრზე განხორციელებული ცვლილებები. ცვლილებებს გამოიწვევს გამოძახების დროს მითითებულ არგუმენტშიც. Java-ში ორივე სახის მექანიზმია გამოყენებული, გადასაცემი მონაცემების მიხედვით.

Java-ში ბაზისური (პრიმიტიული, ელემენტარული) ტიპები გადაიცემა მნიშვნელობის მიხედვით. ამგვარად, პარამეტრის მნიშვნელობის ნებისმიერი ცვლილება, არავითარ ზეგავლენას არ ახდენს ამ მეთოდის გარეთ, თუნდაც იმ არგუმენტის მნიშვნელობაზე, რომელიც გადაეცა პარამეტრს.

განვიხილოთ შემდეგი პროგრამა:

ლისტინგი 61. ელემენტარული ტიპები გადაიცემა მნიშვნელობით

```
class Test {
    void meth(int i, int j) {
     i *= 2;
     j /= 2;
    }
}
class CallByValue {
    public static void main(String args[]){
     Test ob = new Test();
     int a = 15, b = 20;
     System.out.println("a და b გამოძახებამდე: " + a + " " + b);
     ob.meth(a, b);
     System.out.println("a და b გამოძახების შემდეგ: " + a + " " + b);
    }
}
```

კონსოლზე გამოვა:

a და b გამოძახებამდე: 15 20
a და b გამოძახების შემდეგ: 15 20

როგორც ხედავთ, meth() მეთოდის შიგნით შესრულებული ოპერაციები გავლენას არ ახდენენ a და b ცვლადებზე, რომლებიც გამოიყენება გამოძახების დროს. მათი მნიშვნელობები არ გახდა 30 და 10.

მეთოდისთვის რაიმე ობიექტის გადაცემისას სიტუაცია იცვლება, ვინაიდან ობიექტები გადაიცემა მითითების გადაცემით. რაიმე კლასის შესაბამისი ტიპის ცვლადის გამოცხადებისას უნდა გვახსოვდეს, რომ მხოლოდ ობიექტზე მითითება იქმნება. ამგვარად, მეთოდისთვის ამ მითითების გადაცემისას, მისი მიმღები პარამეტრი მიმართავს იმავე ობიექტს, რომელზეც მიუთითებს

არგუმენტი. ფაქტობრივად, ეს ნიშნავს, რომ მეთოდებს ობიექტები გადაეცემა მათზე მითითებით. ობიექტებზე მეთოდის შიგნით განხორციელებული ცვლილებები, გავლენას ახდენენ ობიქტზე, რომელიც არგუმენტად იყო გამოყენებული. მაგალითი:

ლისტინგი 62. ობიექტები გადაიცემა მითითებით

```
class Test {
    int a, b;
    Test (int i, int j) {
     a = i;
     b = j;
    }
// ობიექტის გადაცემა
    void meth(Test o) {
     o.a = 2;
     o. b /= 2;
    }
}
class CallByRef { public static void main(String args[]) {
     Test ob = new Test(15, 20);
     System.out.println("ob.a и ob.b გამოძახებამდე: " + ob.a +" " + ob.b) ;
     ob.meth(ob) ;
     System.out.println("ob.a и ob.b გამოძახების შემდეგ:"ob.a +" "+ob.b);
    }
}
```

ამ პროგრამის შედეგად კონსოლზე გამოვა:

ob.a და ob.b გამოძახებამდე: 15 20
ob.a და ob.b გამოძახების შემდეგ: 30 10

როგორც ხედავთ, ამ შემთხვევაში meth() მეთოდის შიგნით მოქმედებები ზეგავლენას ახდენენ ობიექტზე, რომელიც არგუმენტად იქნა გამოყენებული.

უნდა აღვნიშნოთ, რომ როდესაც მეთოდს გადაეცემა ობიექტზე მითითება (ანუ ობიექტის მისამართი), თვით მითითება (მისამართი) გადაიცემა მნიშვნელობის მიხედვით. მაგრამ, ვინაიდან გადაცემული მნიშვნელობა მიუთითებს ობიექტს, ამიტომ მისი კოპირებული მნიშვნელობაც იმავე ობიექტზე მიმართვა იქნება.

ობიექტების დაბრუნება

მეთოდმა შეიძლება ნებისმიერი ტიპის მონაცემი დააბრუნოს, მათ შორის ჩვენს მიერ შექმნილი კლასის ტიპიც. მაგალითად, შემდეგ პროგრამაში incrByTen() მეთოდი აბრუნებს ობიექტს, რომელშიც a ცვლადის მნიშვნელობა 10-ით მეტია, ვიდრე მისი მნიშვნელობაა გამომძახებელ ობიექტში:

ლისტინგი 63. ობიექტის დაბრუნება

```
class Test {
    int a;
    Test(int i) {
     a = i;
    }
    Test incrByTen() {
     Test temp = new Test(a+10);
     return temp;
    }
}
class RetOb {
    public static void main(String args[]) {
     Test ob1 = new Test(2);
     Test ob2;
     ob2 = ob1.incrByTen();
```

```
        System.out.println("ob1.a: "+ ob1.a);
        System.out.println("ob2.a: "+ ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a მნიშვნელობის მეორედ  გაზრდის შემდეგ: "
+ ob2.a);
    }
}
```

პროგრამის შესრულების შედეგად კონსოლზე გამოვა:

ob1.a: 2

ob2.a: 12

ob2.a მნიშვნელობის მეორედ გაზრდის შემდეგ: 22

როგორც პროგრამიდან ჩანს, incrByTen() მეთოდის ყოველი გამოძახებისას იქმნება ახალი ობიექტი და გამომძახებელ პროცედურას უბრუნებს მასზე მითითებას.

ეს პროგრამა კიდევ ერთი მნიშვნელოვანი მომენტის ილუსტრირებას აკეთებს, ვინაიდან ყველა ობიექტი დინამიკურად იქმნება new ოპერციის შედეგად, პროგრამისტს არ უნდა აღელვებდეს გავა თუ არა ობიექტი განსაზღვრის არეს გარეთ, ვინაიდან მეთოდის მუშაობა, რომელშიც ობიექტი შეიქმნა, წყდება. ობიექტი იარსებებს მანამ, სანამ, პროგრამაში სადმე მაინც იარსებებს ამ ობიექტზე მითითება. თუ არავითარი მითითება აღარ იარსებებს, მაშინ შემდგომი ნაგვის გასუფთავებისას ობიექტი განადგურდება.

რეკურსია

Java-ს აქვს რეკურსიის მხარდაჭერა. რეკურსია, ესაა რაიმეს განმარტება თავისსავე ტერმინებში. Java პროგრამირების კუთხით რეკურსია ესაა ატრიბუტი, რომელიც მეთოდს საშუალებას აძლევს გამოიძახოს თავისი თავი. ასეთ მეთოდს უწოდებენ რეკურსიულს.

რეკურსიის კლასიკური მაგალითია ფაქტორიალის გამოთვლა. როგორც იცით, N რიცხვის ფაქტორიალი ესაა 1-დან N-მდე ყველა მთელი რიცხვის ნამრავლი. მაგალითად, 3! ტოლია 1 * 2 * 3 = 6. რეკურსიული მეთოდის გამოყენებით ფაქტორიალი ასე შეიძლება გამოვთვალოთ:

ლისტინგი 64. ფაქტორიალის რეკურსიული გამოთვლა

```
class Factorial {
    // ესაა რეკურსიული მეთოდი
    int fact(int n) {
     int result;
     if(n==1) return 1;
     result = fact(n-1) * n;
     return result;
    }
}
class Recursion {
    public static void main(String args[]) {
     Factorial f = new Factorial();
     System.out.println(" 3-ის ფაქტორიალია" + f.fact(3));
     System.out.println(" 4-ის ფაქტორიალია" + f.fact(4));
     System.out.println(" 5-ის ფაქტორიალია" + f.fact(5));
    }
}
```

ამ პროგრამის შედეგი::

3-ის ფაქტორიალია 6
4-ის ფაქტორიალია 24
5-ის ფაქტორიალია 120

fact() მეთოდის გამოძახებისას, როდესაც მისი არგუმენტია 1, მეთოდი აბრუნებს 1-ს. წინააღმდეგ შემთხვევაში იგი აბრუნებს fact(n-1) * n ნამრავლს. ამ გამოსახულების გამოსათვლელად, პროგრამა იძახებს fact() მეთოდს არგუმენტით 2. ეს იწვევს მეთოდის მესამედ გამოძახებას არგუმენტით 1. შემდეგ, ეს გამოძახება მნიშვნელობად აბრუნებს 1-ს, რომელიც მრავლდება 2-ზე (ანუ, მეთოდის მეორე გამოძახებისას n-ის მნიშვნელობა). ეს შედეგი (ანუ 2) უბრუნდება fact() მეთოდის პირველ გამოძახებას და იგი მრავლდება 3-ზე (n-ის საწყისი მნიშვნელობა). შედეგად ვღებულობთ 6-ს. fact() მეთოდში შეიძლებოდა ჩაგვესვა printnln() ოპერატორი, რომელიც ასახავდა ყოველი გამოძახების დონეს და შუალედურ შედეგს.

როდესაც მეთოდი თავის თავს გამოიძახებს, ახალ ლოკალურ ცვლადებს და პარამეტრებს სტეკში გამოეყოფათ ადგილი და მეთოდის კოდი შესრულდება ამ ახალი საწყისი მნიშვნელობებით. რეკურსიული გამოძახების ყოველი დაბრუნების შემდეგ, ძველი ლოკალური ცვლადები და პარამეტრები ამოვარდება სტეკიდან და შესრულება გრძელდება მეთოდის შიგნით გამოძახების მომენტიდან.

რეკურსიული მეთოდების გამოყენება რესურსების მხრივ ეფექტური არაა. მისი გამოყენების ძირითადი უპირატესობაა

ის, რომ ზოგიერთი ალგორითმი ამ გზით უფრო გასაგებად და მარტივად ჩაიწერება, ვიდრე მათი იტერაციული ანალოგები.

რეკურსიული მეთოდების გამოყენების დროს პროგრამისტმა უნდა იზრუნოს, რომ პროგრამაში სადმე იყოს გამოყენებული if ოპერატორი, რომელიც ანხორციელებს რეკურსიული მეთოდიდან ისეთ დაბრუნებას, რომ რეკურსიული გამოძახება საჭირო არ იყოს. წინააღმდეგ შემთხვევაში, თუ მოხდა მისი გამოძახება ის ვეღარასოდეს მოახდენს დაბრუნდებას. ამიტომ, პროგრამის დამუშავებისას ჯობია ხშირად გამოვიყენოთ printnln() ოპერატორი, ვინაიდან შესაძლებელი იყოს გამოთვლითი პროცესის თვალყურისდევნება და შეცდომის შემთხვევაში შესაძლებელი იყოს პროცესის შეწყვეტა.

განვიხილოთ რეკურსიის კიდევ ერთი მაგალითი. რეკურსიულ მეთოდს printArray() გამოაქვს პირველი i ელემენტი values მასივიდან.

ლისტინგი 65. რეკურსიის მაგალითი

```
class RecTest {
    int values[];
    RecTest (int i) {
        values = new int[i];
    }
// მასივის ელემენტების რეკურსიულად ასახვა
    void printArray (int i) {
        if(i == 0) return;
        else printArray(i-1);
        System.out.println("[" + (i-1) + "] " + values[i-1]);
    }
```

```
}
class Recursion2 {
    public static void main(String args[]){
        RecTest ob = new RecTest(10);
        int i;
        for(i=0; i<10; i++) ob.values[i] = i;
        ob.printArray(10);
    }
}
```

კონსოლზე გამოვა:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

კლასის სტატიკური ელემენტები

ზოგიერთ შემთხვევაში სასურველია, განისაზღვროს კლასის ისეთი წევრი, რომლის გამოყენება შესაძლებელია დამოუკიდებლად, კლასის ობიექტის შექმნის გარეშე. ჩვეულებრივ, კლასის წევრზე მიმართვა უნდა ხდებოდეს მხოლოდ ამ კლასის ობიექტიდან (მაგალითად, ob.values). მაგრამ შესაძლებელია კლასის ისეთი წევრის შექმნა, რომელიც გამოყენებული იქნება დამოუკიდებლად,

კონკრეტულ ობიექტზე (ეგზემპლარზე) მითითების გარეშე. ასეთი წევრის შესაქმნელად, მისი გამოცხადების დასაწყისში უნდა მივუთითოთ static მოდიფიკატორი. როდესაც კლასის რომელიმე წევრი გამოცხადებულია როგორც static (სტატიკური), მაშინ მასზე მიმართვა შესაძლებელია ამ კლასის ობიექტის შექმნამდეც და რომელიმე კონკრეტულ ობიექტზე მითითების გარეშეც. სტატიკურად შეიძლება გამოცხადდეს მეთოდები, ცვლადები და ცალკეული ბლოკებიც კი. სტატიკური წევრის ყველაზე უფრო გავრცელებული მაგალითია მეთოდი main(). ეს მეთოდი static-ად არის გამოცხადებული, ვინაიდან იგი უნდა არსებობდეს (უნდა იყოს გამოცხადებული) ნებისმიერი ობიექტის შექმნამდე.

ეგზემპლარების ცვლადები, რომლებიც გამოცხადებულია როგორც static, ფაქტიურად წარმოადგენენ გლობალურ ცვლადებს. ასეთი ელემენტების შემცველი კლასის შესაბამისი ობიექტის შექმნისას არ იქმნება static ელემენტის ცალკეული ასლი (კოპიო). ამის მაგივრად კლასის ყველა ეგზემპლარი ერთიდაიგივე სტატიკურ ცვლადს იყენებს ერთობლივად. ანუ, ასეთი ცვლადი ყველა ეგზემ-პლარისათვის საერთოა.

static-ად გამოცხადებული მეთოდები უნდა აკმაყოფილებ-დნენ გარკვეულ მოთხოვნებს:

- მათ შეუძლიათ მხოლოდ სხვა სტატიკური მეთოდების გამოძახება;

- მათ შეუძლიათ მიმართონ მხოლოდ სტატიკურ ცვლადებს;

- მათ არა აქვთ უფლება მიმართონ this და super ტიპის წევრებს (super საკვანძო სიტყვა დაკავშირებულია მემკვიდრეობითობასთან და მომავალში განვიხილავთ).

თუ static ტიპის ცვლადების ინიციალიზაციისათვის საჭიროა გამოთვლების ჩატარება, შესაძლებელია გამოვაცხადოთ სტატიკური ბლოკი, რომელიც შესრულდება მხოლოდ ერთხელ კლასის შემსრულებელ გარემოში, პირველად ჩატვირთვისას. შემდეგ მაგალითში განხილულია კლასი, რომელიც შეიცავს სტატიკურ მეთოდს, რამდენიმე სტატიკურ ცვლადს და ინიციალიზაციის სტატიკურ ბლოკს:

ლისტინგი 66. სტატიკური მეთოდის, ცვლადების და ბლოკის დემონსტრირება

```
class UseStatic {
    static int a = 3;
    static int b;
    static void meth (int x) {
        System.out.println("x ¬ " + x);
        System.out.println("a ¬"+ a);
        System.out.println("b ¬"+ b);
    }
    static {
        System.out.println("სტატიკური ბლოკის ინიციალიზაცია ");
        b =  a * 4;
    }
    public static void main(String args[]) {
        meth(42);
    }
}
```

UseStatic კლასის ჩატვირთვისთანავე პროგრამა ასრულებს ყველა static ოპერატორს. თავიდან a-ს მნიშვნელობა გახდება 3, შემდეგ სრულდება static ბლოკი, რომელსაც კონსოლზე გამოაქვს შეტყობინება და შემდეგ b-ს ანიჭებს a*4. შემდეგ ხდება main() მეთოდის გამოძახება, რომელიც მიმართავს meth() მეთოდს და x პარამეტრს მნიშვნელობად გადასცემს 42. სამი println() ოპერატორი მიმართავს ორ a, b სტატიკურ ელემენტს და x ლოკალურ ცვლადს. ამ პროგრამის შედეგი კონსოლზე ასე გამოვა:

სტატიკური ბლოკის ინიციალიზაცია

x = 42
a = 3
b = 12

იმ კლასის ფარგლებს გარეთ, სადაც აღწერილია სტატიკური ცვლადები და მეთოდები, მათი გამოყენება შეიძლება რაიმე ობიექტისგან დამოუკიდებლად. ამისათვის საკმარისია მივუთითოთ მათი კლასის სახელი, რომლის შემდეგაც უნდა მოდიოდეს ოპერაცია „ . “ . მაგალითად, თუ static მეთოდის გამოძახება უნდა მოხდეს მისი კლასის გარედან, საჭიროა ასეთი ზოგადი ფორმის გამოყენება:

<კლასის სახელი>.<მეთოდის სახელი>()

აქ <კლასის სახელი> ესაა იმ კლასის სახელი, სადაც გამოცხდებულია static მეთოდი. როგორც ხედავთ ეს ფორმატი ობიექტზე მიმთითებელი სახელის საშუალებით ჩვეულებრივი, არასტატიკური მეთოდების გამოძახების ანალოგიურია. სტატიკურ ცვლადზე მიმართვისათვის,

ანალოგიურად, საჭიროა კლასის სახელის შემდეგ მივუთითოთ წერტილის ოპერაცია და შემდეგ სტატიკური ცვლადის სახელი. ამ სახითაა რეალიზებული Java-ში მეთოდებისა და ცვლადების მართვადი გლობალური ვერსიები. მაგალითი:

ლისტინგი 67.

```
class StaticDemo {
    static int a = 42;
    static int b = 99;
    static void callme() {
        System.out.println("a = " + a);
    }
}
class StaticByName {
    public static void main(String args[]) {
        StaticDemo. callme() ;
        System.out.println("b = " + StaticDemo.b);
    }
}
```

ამ პროგრამის შედეგად კონსოლზე გამოვა:

a = 42

b = 99

final მოდიფიკატორი

ცვლადი შეიძლება გამოცხადდეს, როგორც final (საბოლოო, ფინალური). ეს საშუალებას იძლევა აიკრძალოს ცვლადის მნიშვნელობის ცვლილება. ეს ნიშნავს, რომ final ტიპის ცვლადი ინიციალიზებული უნდა იყოს მისი გამოცხადებისას. მაგალითად,

```
final int FILENEW = 1;
final int FILEOPEN = 2;
final int FILESAVE = 3;
final int FILESAVEAS = 4;
final int FILEQUIT = 5;
```

ამის შემდეგ, პროგრამის დანარჩენ ნაწილში, FILEOPEN და სხვა ანალოგიური ცვლადებით შეიძლება ვისარგებლოთ, როგორც კონსტანტებით, მათი ცვლილების შესაძლებლობის რისკის გარეშე.

Java-ში მიღებულია final ტიპის ცვლადების იდენტიფიკატორები ჩავწეროთ დიდი ასოებით. ცვლადებს, რომლებიც გამოცხადებულია final ტიპად, ყოველი ეგზემპლარისათვის არ გამოეყოფათ ცალკე მეხსიერება. ანუ, ფაქტობრივად, ისინი წარმოადგენენ კონსტანტებს.

საკვანძო სიტყვა,final შეიძლება გამოყენებული იქნეს მეთოდების მიმართაც. ამ საკითხს მომავალში განვიხილავთ.

მასივების ხელმეორე განხილვა

მასივები ჩვენ განვიხილეთ მანამ, სანამ კლასებს გავეცნობოდით. ახლა, როდესაც კლასების შესახებ გარკვეული ცოდნა დაგვიგროვდა, მასივების შესახებ შეიძლება მნიშვნელოვანი დასკვნა გავაკეთოთ: მასივები რეალიზებულია ობიექტების სახით. კერძოდ, მასივის ზომა ანუ მასივში ელემენტების რაოდენობა შენახულია მისი ეგზემპლარის ცვლადში - length. ასეთი ცვლადი ყველა მასივს თავისი აქვს და მასში ყოველთვის წერია მასივის

მიმდინარე სიგრძე. ქვემოთ ნაჩვენებია პროგრამა, სადაც დემონსტრირებულია ეს თვისება:

ლისტინგი 68. მასივის ზომის შემნახველი წევრი

```
class Length {
    public static void main(String args[]) {
     int a1[] = new int[10];
     int a2[] = {3, 5, 7, 1, 8, 99, 44, 10};
     int a3[] = {4, 3, 2, 1};
     System.out.println("a1 სიგრძეა " + a1.length);
     System.out.println("a2 სიგრძეა " + a2.length);
     System.out.println("a3 სიგრძეა " + a3.length);
    }
}
```

კონსოლზე გამოვა:

a1 სიგრძეა 10
a2 სიგრძეა 8
a3 სიგრძეა 4

პროგრამას გამოაქვს თითოეული მასივის ზომა. უნდა გახსოვდეთ, რომ length ცვლადის მნიშვნელობა არაა დაკავშირებული რეალურად გამოყენებული ელემენტების რაოდენობასთან. იგი ასახავს თუ რამდენ ელემენტს შეიძლება შეიცავდეს მასივი.

length ცვლადი მრავალ სიტუაციაში შეიძლება იქნეს გამოყენებული. მაგალითად, ქვემოთ ნაჩვენებია Stack კლასის გაუმჯობესებული ვერსია. ადრე განხილულ მაგალითებში ყოველთვის იქმნებოდა 10 ელემენტიანი სტეკი. შემდეგ ვერსიაში შესაძლებელია ნებისმიერი ზომის სტეკის შექმნა. ცვლადი stck.length გამოიყენება სტეკის გადავსების თავიდან ასაცილებლად.

ლისტინგი 69. გაუმჯობესებული Stack კლასი

```
class Stack {
    private int stck[];
    private int tos;
// სტეკის შექმნა და ინიციალიზაცია
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }
// ელემენტის სტეკში ჩაგდება
    void push(int item) {
        if(tos==stck.length-1)
        System.out.println("სტეკი სავსეა");
        else
         stck[++tos] = item;
    }
// ელემენტის სტეკიდან ამოგდება
 int pop () {
     if(tos < 0) {
        System.out.println ("სტეკი ცარიელია");
        return 0;
    }
    else
    return stck[tos--];
} }
class TestStack2 {
    public static void main(String args[]) {
        Stack mystackl = new Stack(5);
        Stack mystack2 = new Stack(8);
// რიცხვების სტეკში „ჩაგდება“
        for(int i=0; i<5; i++) mystackl.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);
// სტეკიდან რიცხვების ამოგდება
        System.out.println("სტეკი mystackl:");
        for(int i=0; i<5; i++)
```

```
            System.out.println(mystackl.pop());
        System.out.println("სტეკი mystack2:");
        for(int i =0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

მიაქციეთ ყურადღება, რომ პროგრამა ორ სტეკს ქმნის: ერთის სიღრმეა 5 ელემენტი, მეორესი - 8 ელემენტი. ვინაიდან მასივი შეიცავს ინფორმაციას თავისი სიგრძის შესახებ, ნებისმიერი ზომის სტეკის შექმნა გაადვილებულია.

ჩადგმული და შიგა კლასები

Java საშუალებას იძლევა განისაზღვროს კლასი სხვა კლასის შიგნით. ასეთ კლასებს უწოდებენ ჩადგმულ კლასებს. ჩადგმული კლასის განსაზღვრის არე შემოიფარგლება გარე კლასის განსაზღვრის არით. ამრიგად, თუ B კლასი განსაზღვრულია A კლასის შიგნით, მაშინ B კლასი არ შეიძლება არსებობდეს A კლასისგან დამოუკიდებლად. ჩადგმულ კლასს უფლება აქვს მიმართოს იმ კლასის წევრებს, რომელიც მას მოიცავს, მათ შორის პრივატულებსაც. მაგრამ, გარე კლასს არა აქვს უფლება მიმართოს ჩადგმული კლასის წევრებს. ჩადგმული კლასი, რომელიც გამოცხადებულია მისი მომცველი კლასის განსაზღვრის არეში, წარმოადგენს მის წევრს. ასევე შესაძლებელია ჩადგმული კლასების გამოცხადება ბლოკში და იგი იქნება ლოკალური ამ ბლოკისათვის.

ჩადგმული კლასების ორი ტიპი არსებობს: სტატიკური და არასტატიკური. სტატიკური ჩადგმული კლასი ესაა, ისეთი

კლასი, სადაც გამოყენებულია static მოდიფიკატორი. ვინაიდან იგი სტატიკურია, თავის გარე კლასს მან უნდა მიმართოს ობიექტის საშუალებით. ანუ არ შეიძლება უშუალოდ მიმართოს გარე კლასის წევრებს. ამ შეზღუდვის გამო სტატიკური ჩადგმული კლასები იშვიათად გამოიყენება.

უფრო მნიშვნელოვანია ჩადგმული კლასების მეორე ტიპი - შიგა კლასები. შიგა კლასი ესაა, არასტატიკური ჩადგმული კლასი. მას წვდომა აქვს თავისი გარე კლასის ყველა მეთოდსა და ცვლადთან. მას შეუძლია უშუალოდ მიმართოს მათ ისევე, როგორც ამას აკეთებენ გარე კლასის დანარჩენი არასტატიკური წევრები.

შემდეგ პროგრამაში ილუსტრირებულია შიგა კლასის განსაზღვრა და გამოყენება. კლასი Outer შეიცავს ეგზემპლარის ერთ ცვლადს outer_x, ეგზემპლარის ერთ მეთოდს test() და განსაზღვრავს ერთ შიგა კლასს Inner.

ლისტინგი 70. შიგა კლასის დემონსტრირება

```
class Outer {
    int outer_x = 100;
    void test () {
        Inner inner = new Inner();
        inner.display();
    }
// ესაა შიგა კლასი
    class Inner {
        void display() {
        System.out.println("შედეგი: outer_x = " + outer_x);
        }
    }
```

```
}
class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

კონსოლზე გამოვა:

შედეგი: outer_x = 100

ამ პროგრამაში შიგა კლასი Inner განსაზღვრულია Outer კლასის განსაზღვრის არეში. ამიტომ Inner კლასის ნებისმიერ წევრს შეუძლია მიმართოს outer_x ცვლადს. ეგზემპლარის მეთოდი display() განსაზღვრულია Inner კლასის შიგნით. ამ მეთოდს outer_x ცვლადის მნიშვნელობა გამოაქვს კონსოლზე. InnerClassDemo კლასის ეგზემპლარის main() მეთოდი ქმნის Outer კლასის ეგზემპლარს და გამოიძახებს მის მეთოდს test(). ეს მეთოდი ქმნის Inner კლასის ეგზემპლარს და გამოიძახებს display() მეთოდს.

მნიშვნელოვანია გავიგოთ, რომ Inner კლასის ეგზემპლარი შეიძლება შეიქმნას მხოლოდ Outer კლასის განსაზღვრის არეს შიგნით. Java-ს კომპილატორი იძლევა შეცდომას, როცა Outer კლასის გარე ნებისმიერი კოდი ცდილობს Inner კლასის ინიციალიზაციას. (ზოგადად, შიგა კლასის ეგზემპლარი შეიძლება შეიქმნას მისი შემცველი დიაპაზონიდან.) თუმცა Inner კლასის ეგზემპლარი შეიძლება შეიქმნას Outer კლასის გარედან, შიგა კლასის სახელის დაზუსტება ხდება გარე კლასის სახელის მიხედვით - მაგალითად, Outer.Inner.

როგორც აღვნიშნეთ, შიგა კლასს შეუძლია წვდომა მისი გარე კლასის ყველა წევრთან, მაგრამ არა პირიქით. შიგა კლასის წევრები ცნობილია მხოლოდ შიგა კლასის განსაზღვრის არეში და არ შეიძლება გამოყენებული იქნან გარე კლასიდან. მაგალითი:

ლისტინგი 71. ამ პროგრამის კომპილაცია არ მოხდება

```
class Outer {
    int outer_x = 100;
    void test () {
     Inner inner = new Inner();
     inner.display();
}
// შიგა კლასი
    class Inner {
     int y = 10; // y  ლოკალური ცვლადი Inner კლასში
     void display () {
        System.out.println("შედეგი: outer_ x = " + outer_x);
     }
    }
    void showy () {
     System.out.println(y); // შეცდომა!! აქ ცვლადი y უცნობია
    }
}
class InnerClassDemo {
    public static void main(String args[]) {
     Outer outer = new Outer();
     outer.test();
    }
}
```

ამ მაგალითში y ცვლადი გამოცხადებულია Inner კლასის ეგზემპლარის ცვლადად. ამიტომ იგი კლასის საზღვრებს

გარეთ ცნობილი არაა და მისი გამოყენება showy() მეთოდს არ შეუძლია.

მართალია, ჩვენ ყურადღება დავუთმეთ შიგა კლასებს, რომლებიც წევრის სახით განსაზღვრულია გარე კლასის შიგნით, შიგა კლასი შეიძლება განვსაზღვროთ ნებისმიერი ბლოკის განსაზღვრის არეშიც. მაგალითად, ჩადგმული კლასი შეიძლება განისაზღვროს მეთოდით განსაზღვრულ შიგა ბლოკში ან for ციკლის ტანის შიგნით, როგორც ეს ნაჩვენებია მაგალითში:

ლისტინგი 72. for ციკლის შიგნით კლასის გამოცხადება

```
class Outer {
    int outer_x = 100;
    void test () {
        for(int i=0; i<10; i++) {
         class Inner {
              void display () {
               System.out.println ("შედეგი: outer_x ¬="+ outer_x) ;
              }
         }
         Inner inner = new Inner();
         inner.display();
        }
    }
}
class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

კონსოლზე გამოვა:

შედეგი: outer_x = 100
შედეგი: outer_x = 100
შედეგი: outer_x = 100
შედეგი: outer_x =100
შედეგი: outer_x = 100
შედეგი: outer_x = 100
შედეგი: outer_x = 100
შედეგი: outer_x = 100
შედეგი: outer_x = 100
შედეგი: outer_x = 100

მართალია ჩადგმული კლასები ყველა სიტუაციაში არაა გამოსადეგი, მაგრამ ისინი მოსახერხებელია განსაკუთრებული სიტუაციების დამუშავებისათვის, რასაც მომავალში განვიხილავთ.

String კლასის აღწერა

მართალია, String კლასი დაწვრილებით იქნება აღწერილი მომავალში, მაგრამ ეს კლასი იმდენად ხშირად გამოყენებადია, რომ მაინც ღირს განვიხილოთ ამ კლასის ზოგიერთი მეთოდი.

გავიხსენოთ, რომ ნებისმიერი შექმნილი სტრიქონი სინამდვილეში წარმოადგენს String ტიპის კლასის ობიექტს. სტრიქონული კონსტანტებიც კი String კლასის ობიექტებია. მაგალითად,

```
System.out.println("ესეც String ტიპის ობიექტია");
```

სტრიქონი "ესეც String ტიპის ობიექტია" String ტიპის კონსტანტაა.

String ტიპის ობიექტები არიან უცვლელი ანუ String ტიპის ობიექტის შექმნის შემდეგ, მისი შემცველობა არ შეიძლება შეიცვალოს. შეიძლება მოგვეჩვენოს, რომ ეს მნიშვნელოვანი შეზღუდვაა, მაგრამ ეს ასე არაა ორი მიზეზის გამო:

- თუ საჭიროა სტრიქონის მნიშვნელობის შეცვლა, ყოველთვის შეიძლება შევქმნათ ახალი სტრიქონი, რომელშიც გათვალისწინებული იქნება ყველა საჭირო ცვლილება;
- Java-ში არსებობს String კლასის ანალოგიური კლასი SringBuffer, რომელშიც დასაშვებია სტრიქონების ცვლილება, რაც საშუალებას იძლევა სტრიქონებზე ჩავატაროთ ყველა ჩვეულებრივი (სტანდარტული) მანიპულაციები. ამ კლასის აღწერასაც მომავალში ვნახავთ.

სტრიქონის შექმნის რამდენიმე გზა არსებობს. ყველაზე მარტივია ასეთი ოპერატორის გამოყენება:

String myString = “test string”;

როგორც კი შეიქმნება String ობიექტი, ის შეიძლება გამოვიყენოთ ყველგან, სადაც სტრიქონული მნიშვნელობის გამოყენება დასაშვებია. მაგალითად, შემდეგ ოპერატორს კონსოლზე გამოაქვს myString სტრიქონული ცვლადის მნიშვნელობა:

System.out.println(myString);

String ტიპის ობიექტებისათვის Java-ში განსაზღვრულია ერთი ოპერაცია „+“. იგი ანხორციელებს ორი სტრიქონის გაერთიანებას (კონკატენაციას):

String myString = "მე" + " მომწონს" + " Java.";

ამ ოპერატორის შედეგად myString ცვლადის მნიშვნელობა გახდება "მე მომწონს Java". შემდეგ პროგრამაში ასახულია აღწერილი კონცეფციები:

ლისტინგი 73. სტრიქონების გამოყენების დემონსტრირება

```
class StringDemo {
    public static void main(String args[]) {
     String strObl = "პირველი სტრიქონი";
     String strOb2 = "მეორე სტრიქონი";
     String strOb3 = strObl + " და " + strOb2;
     System.out.println(strObl);
     System.out.println(strOb2);
     System.out.println(strOb3);
    }
}
```

კონსოლზე გამოვა:

პირველი სტრიქონი
მეორე სტრიქონი
პირველი სტრიქონი და მეორე სტრიქონი

String კლასს აქვს რამდენიმე ათეული მეთოდი, მათგან განვიხილოთ ისინი, რომლებიც ხშირად გამოიყენება. equals() მეთოდის საშუალებით შეიძლება შევამოწმოთ ორი სტრიქონის ტოლობა. length () მეთოდი აბრუნებს სტრიქონის სიგრძეს. charAt() მეთოდის გამოძახებით შეიძლება

მივიღოთ მითითებული ინდექსის შესაბამისი სიმბოლო. ქვემოთ ნაჩვენებია ამ მეთოდების ზოგადი ფორმები:

boolean equals(String <ობიექტი>)
int length()
char charAt(int <ინდექსი>)

შემდეგ პროგრამაში დემონსტრირებულია ამ მეთოდების გამოყენების მაგაილითი:

ლისტინგი 74. String კლასის ზოგიერთი მეთოდის დემონსტრირება

```
class StringDemo2 {
   public static void main(String args[]){
     String strObl = "პირველი სტრიქონი";
     String strOb2 = "მეორე სტრიქონი";
     String strOb3 = strObl;
     System.out.println("strObl-ის სიგრძე: " +
     strObl.length());
     System.out.println("strObl-ის სიმბოლო რომლის ინდექსია 3: " +
strObl.charAt(3));
     if(strObl.equals(strOb2))
         System.out.println("strObl == strOb2");
     else
         System.out.println("strOb1 != strOb2");
     if(strObl.equals(strOb3))
         System.out.println("strOb1 == strOb3");
     else
         System.out.println("strObl != strOb3");
   }
}
```

კონსოლზე გამოვა:

strOb1-ის სიგრძე: 16

strOb1-ის სიმბოლო რომლის ინდექსია 3: 3
strOb1 != strOb2
strOb1 == strOb3

ვინაიდან შეიძლება არსებობდეს ნებისმიერი ტიპის მასივი, ამიტომ სტრიქონული ტიპის მასივებიც დასაშვებია. მაგალითად,

ლისტინგი 75. String ტიპის მასივების დემონსტრირება

```
class StringDemo3 {
    public static void main(String args[]) {
     String str[] = { "ერთი", "ორი", "სამი" };
     for(int i=0; i<str.length; i++)
        System.out.println("str[" + i + "]: " +str [i]);
    }
}
```

კონსოლზე გამოვა:

str [0]: ერთი
str [1]: ორი
str [2]: სამი

ბრძანებათა სტრიქონიდან არგუმენტების გამოყენება

ზოგჯერ პროგრამის გაშვებისას საჭიროა მას გადაეცეს გარკვეული ინფორმაცია. ამისათვის გამოიყენება main() მეთოდის ბრძანებათა სტრიქონის არგუმენტები. ბრძანებათა სტრიქონის არგუმენტი ესაა ინფორმაცია, რომელიც პროგრამის გაშვებისას მიეთითება უშუალოდ ამ პროგრამის შემცველი ფაილის სახელის მითითების შემდეგ. ბრძანებათა სტრიქონის არგუმენტებზე მიმართვა ძნელი არაა, ისინი

სტრიქონის სახით ინახება main() მეთოდისთვის გადაცემულ String მასივში. ბრძანებათა სტრიქონის პირველი არგუმენტი ინახება args[0] მასივის ელემენტად, მეორე args[1] და ა.შ. მაგალითად, შემდეგი პროგრამა ასახავს ბრძანებათა სტრიქონის ყველა არგუმენტს, რომლებითაც ის გამოიძახება:

ლისტინგი 76. ბრძანებათა სტრიქონის ყველა არგუმენტის ასახვა

```
class CommandLine {
    public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
        System.out.println("args[" + i + "]: " +args[i]);
    }
}
```

ეს პროგრამა უნდა შევასრულოთ ბრძანებათა სტრიქონში, მაგალითისათვის, ასეთი სტრიქონის შეტანით:

java CommandLine this is a test 100 1

კონსოლზე ასეთი შედეგი გამოვა:

args [0]: this
args [1]: is
args [2]: a
args [3]: test
args [4]: 100
args [5]: 1

Varargs: ცვლადი რაოდენობის არგუმენტები

JDK 5-ში დაემატა ახალი ფუნქციონალური შესაძლებლობის მეთოდი, რომელიც აადვილებს ცვლადი რაოდენობის

არგუმენტების გადაცემას. ამ ფუნქციონალურ შესაძლებლობას უწოდებენ varargs (varieble-length arguments ტერმინების შემოკლებით).

სიტუაციები, როდესაც საჭიროა სხვადასხვა რაოდენობის არგუმენტი, არც ისე იშვიათია. მაგალითად, მეთოდს, რომელიც ხსნის ინტერნეტთან მიერთებას, შეიძლება გადაეცეს მომხმარებლის სახელი, პაროლი, ფაილის სახელი, პროტოკოლი და სხვა ამგვარი, მაგრამ თუ რომელიმე მონაცემი გამოტოვებულია, მის მაგივრად სტანდარტული მნიშვნელობა უნდა მიიღოს. ასეთ სიტუაციაში უმჯობესი იქნებოდა მეთოდისთვის გადაგვეცა მხოლოდ ის არგუმენტები, რომლებისთვისაც სტანდარტული მნიშვნელობების მინიჭება შეუძლებელი იქნებოდა.

J2SE 5 ვერსიის გამოსვლამდე ცვლადი რაოდენობის არგუმენტების დამუშავება შეიძლებოდა ორი ხერხით, მაგრამ არცერთი მათგანი მოსახერხებელი არ იყო. თუ არგუმენტების მაქსიმალური რაოდენობა ცნობილი იყო და დიდი არ იყო, შეიძლებოდა მეთოდის გადატვირთული ვერსიების შექმნა. მეთოდის გამოძახების ყოველ ვარიანტზე მისი შესაბამისი გადატვირთული მეთოდი უნდა შექმნილიყო. მართალია, ზოგ შემთხვევაში ასეთი ხერხი მისაღებია, მაგრამ ის მხოლოდ შეზღუდული ამოცანების კლასისთვისაა მოსახერხებელი.

როდესაც არგუმენტების მაქსიმალური რაოდენობა დიდია ან უცნობია, გამოიყენებოდა მეორე ხერხი. არგუმენტები უნდა მოგვეთავსებია მასივში და შემდეგ ეს მასივი უნდა გადაგვეცა მეთოდისათვის. მაგალითი:

ლისტინგი 77.მასივის გამოყენება ცვლადი რაოდენობის პარამეტრების გადასაცემად (ძველი ხერხი)

```
class PassArray {
    static void vaTest(int v[]) {
     System.out.print("არგუმენტების რაოდენობა: " +
                v.length +" მნიშვნელობა: ");
     for(int x : v)
        System.out.print(x +" ");
     System.out.println();
    }
    public static void main(String args[]){
// ყურადღება მიაქციეთ არგუმენტების შესანახი მასივის შექმნის ხერხს
     int n1[] = {10};
     int n2[] = { 1, 2, 3 };
     int n3 [] = { };
     vaTest(n1); // 1 არგუმენტი
     vaTest(n2); // 3 არგუმენტი
     vaTest(n3); // უარგუმენტო
    }
}
```

კონსოლზე გამოვა:

არგუმენტების რაოდენობა: 1 მნიშვნელობა: 10
არგუმენტების რაოდენობა: 3 მნიშვნელობა: 1 2 3
არგუმენტების რაოდენობა: 0 მნიშვნელობა:

პროგრამაში vaTest() მეთოდს არგუმენტები გადაეცემა v მასივის საშუალებით. არგუმენტების გადაცემის ეს ძველი მეთოდი საშუალებას აძლევს vaTest() მეთოდს მიიღოს ნებისმიერი რაოდენობის არგუმენტები. მაგრამ ის მოითხოვს, რომ მასივში ელემენტები ხელით იქნეს ჩადებული. vaTest() მეთოდის ყოველი გამოძახებისას მასივის შექმნა

შრომატევადი სამუშაოა და შეიძლება შეცდომების წყაროც იყოს. უფრო ადვილი და ეფექტური ხერხია varargs მეთოდების გამოყენება.

ცვლადი რაოდენობის არგუმენტების მისათითებლად გამოიყენება სამი წერტილი (...). მაგალითად, როგორ უნდა ჩაიწეროს vaTest() მეთოდი ცვლადი რაოდენობის არგუმენტების მისათითებლად:

static void vaTest (int ... v) {

ეს სინტაქსური კონსტრუქცია კომპილატორს მიუთითებს, რომ vaTest() მეთოდი შეიძლება გამოიძახოს 0 ან მეტი რაოდენობის არგუმენტი. v არაცხადად განისაზღვრება, როგორც int[] ტიპის მასივი. ამრიგად, vaTest() მეთოდის შიგნით v-ზე მიმართვა ხორციელდება ჩვეულებრივი მასივის სინტაქსის სახით. წინა მაგალითი vararg მეთოდის გამოყენებით ასე შეიძლება ჩავწეროთ:

ლისტინგი 78. ცვლადი რაოდენობის არგუმენტების გამოყენების დემონსტრირება

```
class VarArgs {
// vaTest() იყენებს ცვლადი რაოდენობის არგუმენტებს
    static void vaTest (int ... v) {
     System.out.println("არგუმენტების რაოდენობა: " +
                v.length +" მნიშვნელობა: ");
     for(int x : v)
        System.out.print(x +" ");
     System.out.println();
    }
    public static void main(String args[]){
// ყურადღება მიაქციეთ vaTest()-ის გამოძახების სახეს
     vaTest(10);   // 1 არგუმენტი
```

```
        vaTest(1, 2, 3);          // 3 არგუმენტი
        vaTest();       // უარგუმენტო
    }
}
```

ამ პროგრამის შედეგი ემთხვევა წინა პროგრამის შედეგს.

ჩამოვთვალოთ ამ პროგრამის მნიშვნელოვანი თავისებურებები:

1. vaTest() მეთოდის შიგნით v ცვლადი მოქმედებს როგორც მასივი. ეს გამოწვეულია იმით, რომ v წარმოადგენს მასივს. სინტაქსური კონსტრუქცია ... კომპილატორს მიანიშნებს, რომ მეთოდი გამოიყენებს ცვლადი რაოდენობის არგუმენტებს და ეს არგუმენტები შენახული იქნება მასივში, რომელზეც მითითებას აკეთებს v ცვლადი;

2. main() მეთოდში vaTest() მეთოდის გამოძახება ხდება სხვადასხვა რაოდენობის არგუმენტით, მათ შორის უარგუმენტოდაც. არგუმენტები ავტომატურად თავსდება მასივში და გადაეცემა v ცვლადს. არგუმენტების არ არსებობის შემთხვევაში მასივის სიგრძეა 0.

ცვლადი სიგრძის პარამეტრთან ერთად მასივი შეიძლება შეიცავდეს „ნორმალურ“ პარამეტრებსაც. მაგრამ ცვლადი სიგრძის პარამეტრი უნდა იყოს მეთოდში გამოცხადებული ბოლო პარამეტრი. მაგალითად, დასაშვებია მეთოდის შემდეგნაირი გამოცხადება:

```
int doIt(int a, int b, double c, int ... va1s) {
```

ამ შემთხვევაში პირველი სამი არგუმენტი, რომლებიც მითითებულია doIt() მეთოდზე მიმართვაში, შეესაბამება პირველ სამ პარამეტრს. ყველა დანარჩენი არგუმენტი ითვლება, რომ მიეკუთვნება vals პარამეტრს.

გვახსოვდეს, რომ vararg პარამეტრი უნდა იყოს უკანასკნელი. მაგალითად, შემდეგი გამოძახება ჩაწერილია არასწორად:

```
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // შეცდომა!
```

ამ მაგალითში, მცდელობა არის გამოცხადდეს ჩვეულებრივი პარამეტრი vararg ტიპის პარამეტრის შემდეგ, რაც დაუშვებელია.
არსებობს კიდევ ერთი შეზღუდვა: მეთოდი შეიძლება შეიცავდეს მხოლოდ ერთ vararg ტიპის პარამეტრს. მაგალითად, ასეთი გამოცხადებაც არასწორია:

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // შეცდომა!
```

განვიხილოთ vaTest() მეთოდის შეცვლილი ვერსია, რომელიც იღებს ჩვეულებრივ და ცვლადი სიგრძის არგუმენტებს:

ლისტინგი 79. ცვლადი სიგრძის და ჩვეულებრივი არგუმენტების ერთობლივი გამოყენება

```
public class VarArgs2 {
    // ამ მაგალითში msg ჩვეულებრივი პარამეტრია,
    // ხოლო v კი vararg ტიპისაა
    static void vaTest(String msg, int ... v) {
        System.out.print(msg + v. length +" მნიშვნელობა: ");
        for (int x : v)
            System.out.print (x + " ");
        System.out.println();
    }
    public static void main(String[] args) {
```

```
        vaTest ("ერთი პარამეტრია vararg: ", 10);
        vaTest("სამი პარამეტრია vararg: ", 1, 2, 3);
        vaTest ("უპარამეტრო vararg: ");
    }
}
```

ამ პროგრამის შედეგია:

ერთი პარამეტრია vararg: 1 მნიშვნელობა: 10
სამი პარამეტრია vararg: 3 მნიშვნელობა: 1 2 3
უპარამეტრო vararg: 0 მნიშვნელობა:

vararg მეთოდების გადატვირთვა

მეთოდი, რომელიც ცვლადი სიგრძის არგუმენტს ღებულობს, შეიძლება გადაიტვირთოს. მაგალითად,

ლისტინგი 80. vararg პარამეტრები და გადატვირთვა

```
class VarArgs3 {
    static void vaTest(int ... v) {
     System.out.print("vaTest(int ...): "+
     "არგუმენტების რაოდენობა: " +
      v.length +" მნიშვნელობა: ");
     for (int x : v)
        System.out.print(x + " ");
     System.out.println();
    }
    static void vaTest(boolean ... v) {
     System.out.print("vaTest(boolean ...) "+
        "არგუმენტების რაოდენობა: " +
        v.length +" მნიშვნელობა: ");
     for(boolean x : v)
        System.out.print(x +" ");
     System.out.println();
    }
```

```
    static void vaTest (String msg, int ... v) {
     System.out.print("vaTest(String, int ...): "+
                msg + v.length +" მნიშვნელობა: ");
     for (int x : v)
        System.out.print(x + " ");
     System.out.println();
    }
    public static void main(String args[]) {
     vaTest(1, 2, 3);
     vaTest("შემოწმება: ", 10, 20);
     vaTest(true, false, false);
    }
}
```

ამ პროგრამის შედეგია:

vaTest(int ...): არგუმენტების რაოდენობა: 3 მნიშვნელობა: 1 2 3

vaTest(String, int ...): შემოწმება: 2 მნიშვნელობა: 1020

vaTest(boolean ...) არგუმენტების რაოდენობა: 3 მნიშვნელობა: true false false

ეს პროგრამა vararg მეთოდის ორივე შესაძლო გადატვირთვის ხერხის დემონსტრირებას აკეთებს.

1. მისი vararg პარამეტრის ტიპი შეიძლება იყოს განსხვავებული. სწორედ ამასა აქვს ადგილი vaRest (int ...) და vaTest (boo1ean .. .) ვარიანტებში. უნდა გვახსოვდეს, რომ ... კონსტრუქცია კომპილატორს აიძულებს დაამუშაოს პარამეტრი, როგორც მითითებული ტიპის მასივი. ამიტომ, მეთოდების მასივის სხვადასხვა ტიპების გადატვირთვის გამოყენების მსგავსად, შესაძლებელია vararg მეთოდების გადატვირთვა, ცვლადი სიგრძის არგუმენტის სხვადასხვა ტიპის მიხედვით. ამ შემთხვე-

ვაში საჭირო ვარიანტის მეთოდის შესაჩევად Java იყენებს ტიპებს შორის განსხვავებას.

2. vararg მეთოდის გადატვირთვისათვის გამოიყენება ჩვეულებრივი პარამეტრი. ეს ხერხი vaTest (String, int ...) მეთოდისათვის იქნა გამოყენებული. ამ შემთხვევაში საჭირო მეთოდის განსაზღვრისათვის Java იყენებს არგუმენტების რაოდენობასაც და ტიპსაც.

სარჩევი