მემკვიდრეობითობა

მემკვიდრეობითობა ობიექტზე ორიენტირებული პროგრამირების ერთ-ერთი ფუნდამენტური ცნებაა. იგი საშუალებას იძლევა შეიქმნას იერარქიულად დაკავშირებული კლასები. მემკვიდრეობითობის გამოყენებით შეიძლება აიგოს ზოგადი კლასი, რომელიც განსაზღვრავს მსგავსი ელემენტების საერთო თვისებებს. ამ კლასიდან, შემდეგ შეიძლება შეიქმნას მემკვიდრე კლასები, რომლებიც უფრო დაკონკრეტებული, სპეციალიზებული იქნებიან და თითოეული მათგანი დაამატებს რაიმე უნიკალურ თვისებას ან მეთოდს. Java ტერმინოლოგიის მიხედვით მშობელ კლასს უწოდებენ **სუპერ კლასს**, ხოლო მემკვიდრე (შვილობილ) კლასებს უწოდებენ **ქვეკლასებს.** შესაბამისად, ქვეკლასი ესაა სუპერკლასის სპეციალიზებული ვერსია. მას მემკვიდრეობით გადაეცემა სუპერკლასში განსაზღვრული ყველა ეგზემპლარის ცვლადი, ყველა მეთოდი და თვითონ უმატებს საკუთარ, უნიკალურ ელემენტებს.

მემკვიდრეობითობის საფუძვლები

მემკვიდრე კლასის შესაქმნელად საკმარისია ამ კლასის აღწერისას extends საკვანძო სიტყვის გამოყენებით მივუთითოთ მშობელი კლასის სახელი. საილუსტრაციოდ განვიხილოთ მარტივი მაგალითი. შემდეგი პროგრამა ქმნის A სუპერ კლასს და B ქვეკლასს.

ლისტინგი 1. მემკვიდრეობითობის მარტივი მაგალითი, სუპერკლასის გამოცხადება:

```
class A {
   int i, j;
   void showij() {
      System.out.println("i და j: " + i + " " + j);
   }
}
// A-ს ქვეკლასის გამოცხადება
class B extends A {
   int k;

   void showk() {
      System.out.println("k: " + k);
   }
   void sum() {
      System.out.println("i+j+k: " + (i + j + k));
   }
}
class Simplelnheritance {
   public static void main(String args[]) {
      A superOb = new A();
      B subOb = new B();
      // სუპერკლასი შესაძლებელია დამოუკიდებლად იქნას
      // გამოყენებული
      superOb.i = 10;
      superOb.j = 20;
      System.out.println("superOb მნიშვნელობა: ");
      superOb.showij();
      System.out.println();
      /* ქვეკლასს შეუძლია მიმართოს თავისი სუპერკლასის
ყველა წვდომად ელემენტს */
      subOb.i = 7;
      subOb.j = 8;
      subOb.k = 9;
      System.out.println("subOb მნიშვნელობა: ");
      subOb.showij();
      subOb.showk();
```

```
        System.out.println();
        System.out.println("i, j და k ჯამი subOb-ში:");
        subOb.sum();
    }
}
```

ამ პროგრამის შედეგი:

superOb მნიშვნელობა:
i და j: 10 20
subOb მნიშვნელობა:
i და j: 7 8
k: 9
i, j და k ჯამი subOb-ში:
i+j+k: 24

როგორც ხედავთ B ქვეკლასი შეიცავს თავისი A სუპერკლასის ყველა ელემენტს. ამიტომ subOb-ს შეუძლია მიმართოს i და j ცვლადებს და გამოიძახოს showij() მეთოდი. გარდა ამისა, sum() მეთოდის შიგნით შესაძლებელია უშუალოდ მივმართოთ i და j ცვლადებს, ისევე, როგორც ეს ცვლადები B კლასში აღწერილი წევრები რომ ყოფილიყო.

მიუხედავად იმისა, რომ B კლასის სუპერკლასია A, ისინი სრულიად დამოუკიდებელი კლასებია. ის რომ, კლასი რომელიმე სხვა კლასის სუპერკლასია არ ნიშნავს, რომ მისი დამოუკიდებლად გამოყენება არ შეიძლება. უფრო მეტიც, ქვეკლასი შეიძლება იყოს სხვა რომელიმე ქვეკლასის სუპერკლასი.

ზოგადად, ისეთი კლასის გამოცხადების ზოგადი ფორმა, რომელიც რომელიმე სუპერკლასის მემკვიდრეა, ასე შეიძლება წარმოვადგინოთ:

class <კლასის სახელი> extends <სუპერკლასის სახელი> {
    // კლასის ტანი
}

ქვეკლასის გამოცხადებისას შეიძლება მივუთითოთ მხოლოდ ერთი სუპერკლასი. Java კრძალავს ერთი კლასისთვის რამდენიმე სუპერკლასის არსებობას. როგორც ზემოთ აღვნიშნეთ, შესაძლებელია მემკვიდრეობითობის იერარქიის შექმნა, სადაც ქვეკლასი შეიძლება იყოს სხვა კლასის სუპერკლასი. მაგრამ არცერთი კლასი არ შეიძლება იყოს თავისივე სუპერკლასი.

მემკვიდრეობითობა და წევრებზე წვდომა

მართალია ქვეკლასი შეიცავს თავისი სუპერკლასის ყველა ელემენტს, მაგრამ მას არ შეუძლია სუპერკლასის იმ წევრებზე მიმართვა, რომლებიც გამოცხადებულია private ატრიბუტით. მაგალითად, განვიხილოთ კლასების მარტივი იერარქია:

/* კლასების იერარქიაში პრივატული კლასები რჩები თავისი კლასების პრივატულად.
ამ პროგრამაში დაშვებულია შეცდომა და მისი კომპილაცია არ შეიძლება */

ლისტინგი 2. სუპერკლასის გამოცხადება:

```
class A {
   int i;
   private int j;
```

```
   void setij (int x, int y) {
      i = x;     // ნაგულისხმევი წვდომადი
      j = y;     // პრივატული A-ში
   }
}
// A კლასის j ცვლადი ამ კლასში არაა წვდომადი
class B extends A {
   int total;
   void sum() {
      total = i + j;
// შეცდომა! j ამ კლასში არაა წვდომადი
}
class Access {
   public static void main(String args[]) {
      B subOb = new B();
      subOb.setij  (10, 12);
      subOb.sum() ;
      System.out.println("ჯამი ტოლია " + subOb.total);
   }
}
```

ამ პროგრამის კომპილაცია შეუძლებელია, ვინაიდან B კლასის შიგნით sum() მეთოდიდან j ცვლადზე მიმართვა არ შეიძლება. რადგან j ცვლადი გამოცხადებულია როგორც private, იგი წვდომადია მხოლოდ თავისი კლასის სხვა წევრებისათვის. ქვეკლასს მასთან წვდომა არ აქვს.

განვიხილოთ უფრო რეალური მაგალითი. გავიხსენოთ Box კლასის აღწერის ბოლო ვერსია და მას დავუმატოთ მეოთხე კომპონენტი, რომელსაც დავარქვათ weight (წონა). ამრიგად, ახალი კლასი შეიცავს პარალელეპიპედის სიგანეს, სიმაღლეს, სიგრძეს და წონას.

ლისტინგი 3. ამ პროგრამაში მემკვიდრეობითობა გამოიყენება კლასის გაფართოებისათვის:

```
class Box {
   double width;
   double height;
   double depth;
   // ობიექტის კლონის კონსტრუირება
   Box(Box ob) { // ობიექტის გადაცემა კონსტრუქტორზე
      width = ob.width;
      height = ob.height;
      depth = ob.depth;
   }
   // ყველა განზომილების გამომყენებელი კონსტრუქტორი
   Box(double w, double h, double d) {
      width = w;
      height = h;
      depth = d;
   }
   // უარგუმენტო კონსტრუქტორი
   Box() {
      width = -1;
      height = -1;
      depth = -1;
   }
   // კუბის შესაქმნელი კონსტრუქტორი
   Box(double len) {
      width = height = depth = len;
   }
   // მოცულობის გამოთვლა და დაბრუნება
   double volume() {
      return width * height * depth;
   }
}
// Box კლასის გაფართოება წონის ჩამატებით
class BoxWeight extends Box {
   double weight; // პარალელეპიპედის წონა
   // BoxWeight კონსტრუქტორი
```

```
   BoxWeight(double w, double h, double d, double m) {
      width = w;
      height = h;
      depth = d;
      weight = m;
   }
}
class DemoBoxWeight {
   public static void main(String args[]) {
      BoxWeight mybox1 = new BoxWeight(10, 20, 15,
34.3);
      BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
      double vol;
      vol = mybox1.volume();
      System.out.println("mybox1 მოცულობაა " + vol);
      System.out.println("mybox1 წონაა " +
mybox1.weight);
      System.out.println();
      vol = mybox2.volume();
      System.out.println("mybox2 მოცულობაა " + vol);
      System.out.println("mybox2 წონაა " +
mybox2.weight);
   }
}
```

ამ პროგრამის შედეგი:

mybox1 მოცულობაა 3000.0
mybox1 წონაა 34.3
mybox2 მოცულობაა 24.0
mybox2 წონაა 0.076

BoxWeight კლასს მემკვიდრეობით გადმოეცა Box კლასის ყველა თვისება და დაემატა კომპონენტი weight. BoxWeight კლასში არაა საჭირო Box კლასის ყველა თვისების თავიდან გამოცხადება. იგი უბრალოდ გარკვეული მიზნებით აფართოებს Box კლასს.

მემკვიდრეობითობის ძირითადი უპირატესობა ისაა, რომ როგორც კი ობიექტების ერთობლიობის ზოგადი ატრიბუტების განმსაზღვრელი სუპერკლასი აღიწერება, იგი შეიძლება გამოყენებული იქნას ნებისმიერი რაოდენობის უფრო სპეციალიზებული კლასების შესაქმნელად. თითოეულ ქვეკლასს შეუძლია ზუსტად განსაზღვროს თავისი სპეციფიკა. მაგალითად, შემდეგ კლასს მემკვიდრეობით გადმოეცემა Box კლასის თვისებები და თვითონაც ამატებს ახალ ატრიბუტს - ფერს:

ლისტინგი 4. ეს კოდი აფართოებს Box კლასს, ამატებს ფერის ატრიბუტს:

```
class ColorBox extends Box {
   int color; // პარალელოპიპედის ფერი
   ColorBox(double w, double h, double d, int c) {
      width = w;
      height = h;
      depth = d;
      color = c;
   }
}
```

ამრიგად, როდესაც ობიექტის ზოგადი ასპექტების განმსაზღვრელი კლასი შეიქმნება, იგი შეიძლება გამოვიყენოთ, როგორც მშობელი კლასი მემკვიდრეობითობის გადასაცემად. სწორედ ეს წარმოადგენს მემკვიდრეობითობის არსს.

სუპერკლასის ცვლადის დაკავშირება ქვეკლასის ობიექტთან

სუპერკლასის მიმთითებელ ცვლადს შეიძლება მიენიჭოს მისი ნებისმიერი ქვეკლასის მისამართი (ქვეკლასზე მითითება). ანუ, სუპერკლასის მიმთითებელი ცვლადი შეიძლება დაუკავშირდეს მისი შვილობილი კლასის

ნებისმიერ ობიექტს. მემკვიდრეობითობის ეს ასპექტი ძალიან სასარგებლოა სხვადასხვა სიტუაციებში. განვიხილოთ შემდეგი მაგალითი:

ლისტინგი 5.

```
class RefDemo {
 public static void main(String args[]) {
   BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
   Box plainbox = new Box();
   double vol;
   vol = weightbox.volume();
   System.out.println("weightbox წონაა " + vol);
   System.out.println("weightbox წონაა " +
   weightbox.weight);
   System.out.println();
   // BoxWeight ტიპის მიმთითებელს ენიჭება Box ტიპის
   // ობიექტზე მითითება
   plainbox = weightbox;
   vol = plainbox.volume(); // OK, volume() მეთოდი
                            // აღწერილია Box-ში
   System.out.println("plainbox მოცულობაა " + vol);
   /* შემდეგ ოპერატორში შეცდომაა, plainbox არ აღწერს
weight-ს */
   // System.out.println("plainbox წონაა " +
plainbox.weight);
 }
}
```

ამ მაგალითში weightbox ესაა მითითება BoxWeight ობიექტზე, ხოლო plainbox ესაა Box ობიექტზე მითითება. ვინაიდან BoxWeight წარმოადგენს Box კლასის ქვეკლასს, ამიტომ plainbox მითითებას შეიძლება მიენიჭოს weightbox ობიექტზე მითითება.

მნიშვნელოვანია გავიაზროთ, რომ ობიექტებზე მიმართვა განისაზღვრება მიმთითებელი ცვლადის ტიპით და არა იმ ობიექტის ტიპით, რომელზედაც ის მიუთითებს. ანუ, სუპერკლასზე მიმთითებელ ცვლადს თუ მივანიჭებთ ამ კლასის ქვეკლასის ობიექტზე კავშირს, მაშინ მიმართვა შესაძლებელია მითითებული ობიექტის მხოლოდ იმ წევრებზე, რომლებიც აღწერილია სუპერკლასში. სწორედ ამიტომ ობიექტ plainbox-ს არ აქვს წვდომის უფლება weight ცვლადზე, იმ შემთხვევაშიც კი, როდესაც ის მიუთითებს BoxWeightის ობიექტს. თუ დავფიქრდებით, მართლაც, სუპერკლასისათვის უცნობია თუ რას დაუმატებს ქვეკლასი მას. ამიტომაა კომენტარის სახით გაფორმებული წინა პროგრამის ფრაგმენტის კოდში ბოლო სტრიქონი. Box-ის ობიექტზე მითითებას უფლება არ აქვს მიმართოს weight ცვლადს (ველს), ვინაიდან იგი Box კლასში აღწერილი არაა.

ზემოთ თქმული შეიძლება ცოტა გაურკვეველი მოგეჩვენოთ, მაგრამ მას აქვს პრაქტიკული გამოყენება, რომელთაგან ორ მათგანს შემდგომ პარაგრაფებში განვიხილავთ.

საკვანძო სიტყვა super-ის გამოყენება

წინა მაგალითებში Box კლასის ბაზაზე შექმნილი კლასები შეიძლება უფრო ეფექტურად და საიმედოდ დაიწეროს. მაგალითად, BoxWeight კონსტრუქტორში ცხადი სახით ხდება Box კლასის width, height და depth ველების ინიციალიზაცია. ეს იწვევს სუპერკლასის კოდის დუბლირებას, რაც ძალიან არაეფექტურია და თან გულისხმობს, რომ ქვეკლასს უნდა ჰქონდეს ამ წევრებთან წვდომა. არადა ხში-

რად უნდა ხდებოდეს ისეთი სუპერკლასის შექმნა, რომ მხოლოდ თვითონ ამ კლასში უნდა იყოს შესაძლებელი რეალიზაციის დეტალებზე მიმართვები (მაგალითად პრივატულ ცვლადებზე მიმართვა). ამ შემთხვევაში ქვეკლასი ვერ შეძლებს დამოუკიდებლად უშუალოდ მიმართოს ან ინიციალიზაცია ჩაუტაროს ასეთ ცვლადებს. ვინაიდან ინკაფსულაცია ობიექტზე ორიენტირებული პროგრამირების ერთ-ერთი მთავარი არგუმენტია, ამიტომ Java გვერდს ვერ აუვლის ამ პრობლემის გადაწყვეტას. ყველა იმ შემთხვევაში, როდესაც ქვეკლასს ესაჭიროება მის უშუალო სუპერკლასზე მიმართვა, ეს უნდა განხორციელდეს საკვანძო სიტყვა super-ის საშუალებით.

super-ს აქვს ორი ზოგადი ფორმა. პირველი გამოიყენება სუპერკლასის კონსტრუქტორის გამოსაძახებელად, ხოლო მეორე სუპერკლასის წევრზე მიმართვისათვის, რომელიც გადაფარულია ქვეკლასის წევრის მიერ. განვიხილოთ ეს ფორმები.

სუპერკლასის კონსტრუქტორის გამოძახება

ქვეკლასს შეუძლია ასეთი ფორმით გამოიძახოს სუპერკლასში განსაზღვრული კონსტრუქტორი:

super (<არგუმენტების სია>);

<არგუმენტების სია> ესაა სუპერკლასში აღწერილი კონსტრუქტორისათვის საჭირო ყველა არგუმენტი. ოპერატორი super() ყოველთვის უნდა იყოს ქვეკლასის კონსტრუქტორში პირველი შესრულებადი ოპერატორი.

ოპერატორი super()-ის გამოყენების საილუსტრაციოდ განვიხილოთ BoxWeight კლასის შემდეგი გაუმჯობესებული ვერსია:

ლისტინგი 6. კლასი BoxWeight იყენებს საკვანძო სიტყვა super-ს Box კლასის თავისი ცვლადების ინიციალიზაციისათვის:

```
class BoxWeight extends Box {
   double weight;   // პარალელეპიპედის წონა
   // width, height და depth ინიციალიზაცია super()-ით
   BoxWeight(double w, double h, double d, double m) {
      // სუპერკლასის კონსტრუქტორის გამოძახება
      super(w, h, d);
      weight = m;
   }
}
```

ამ მაგალითში BoxWeight() კონსტრუქტორი იძახებს super()-ს არგუმენტებით w, h და d. ეს იწვევს Box() კონსტრუქტორის გამოძახებას, რომელიც ინიციალიზაციას უკეთებს width, height და depth-ს გადაცემული პარამეტრების მიხედვით. BoxWeight კლასი დამოუკიდებლად არ ახდენს ამ მნიშვნელობების ინიციალიზაციას. მან მხოლოდ თავის უნიკალური weight ცვლადის ინიციალიზაცია უნდა მოახდინოს. შედეგად, საჭიროების შემთხვევაში ეს ცვლადები შესაძლებელია დარჩეს პრივატულად Box კლასში.

ამ მაგალითში მეთოდი super() გამოძახებული იქნა სამი არგუმენტით. ვინაიდან კონსტრუქტორები შეიძლება გადატვირთული იყოს, ამიტომ super()-ით შესაძლებელია სუპერკლასში აღწერილი ნებისმიერი კონსტრუქტორის ფორმის გამოძახება. პროგრამა შეასრულებს იმ

კონსტრუქტორს, რომელსაც შეესაბამება მითითებული არგუმენტები. მაგალითის სახით განვიხილოთ BoxWeight-ის სრული ვერსია, რომელიც გვთავაზობს სხვადასხვა სახის კონსტრუქტორებს. თითოეულ შემთხვევაში კონსტრუქტორი super() გამოიძახება შესაბამისი არგუმენტებით. მიაქციეთ ყურადღება, რომ Box კლასის შიგნით width, height და depth წევრები გამოცხადებულია როგორც პრივატული.

ლისტინგი 7. BoxWeight კლასის სრული რეალიზაცია:

```
class Box {
   private double width;
   private double height;
   private double depth;
   // ობიექტის კლონის კონსტრუირება
   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;
   }
   Box() {
      width = -1;
      height = -1;
      depth = -1;
   }
   Box(double len) {
      width = height = depth = len;
   }
   double volume() {
      return width * height * depth;
   }
}
class BoxWeight extends Box {
```

```
   double weight;
   BoxWeight(BoxWeight ob) {
      super(ob);
      weight = ob.weight;
   }
   BoxWeight(double w, double h, double d, double m) {
      super(w, h, d);
      weight = m;
   }
   BoxWeight() {
      super();
      weight = 1;
   }
   BoxWeight(double len, double m) {
      super(len);
      weight = m;
   }
}
class DemoSuper {
   public static void main(String args[]) {
      BoxWeight mybox1 = new BoxWeight(10, 20, 15,
      34.3);
      BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
      BoxWeight mybox3 = new BoxWeight();
      // სტანდარტულად
      BoxWeight mycube = new BoxWeight(3, 2);
      BoxWeight myclone = new BoxWeight(mybox1);
      double vol;
      vol = mybox1.volume();
      System.out.println("mybox1 მოცულობაა " + vol);
      System.out.println("mybox1 წონაა " +
      mybox1.weight);
      System.out.println();
      vol = mybox2.volume();
      System.out.println("mybox2 მოცულობაა " + vol);
      System.out.println("mybox2 წონაა " +
      mybox2.weight);
      System.out.println();
      vol = mybox3.volume();
```

```
        System.out.println("mybox3 მოცულობაა " + vol);
        System.out.println();
        vol = myclone.volume();
        System.out.println("myclone მოცლობაა " + vol);
        System.out.println("myclone წონაა " +
        myclone.weight);
        System.out.println();
        vol = mycube.volume();
        System.out.println("mycube მოცულობაა " + vol);
        System.out.println("mycube წონაა " +
        mycube.weight);
        System.out.println();
    }
}
```

პროგრამის შესრულების შედეგი:

mybox1 მოცულობაა 3000.0
mybox1 წონაა 34.3
mybox2 მოცულობაა 24.0
mybox2 წონაა 0.076
mybox3 მოცულობაა -1.0
mybox3 წონაა -1.0
myclone მოცულობაა 3000.0
myclone წონაა 34.3
mycube მოცულობაა 27.0
mycube წონაა 2.0

ყურადღება მიაქციეთ BoxWeight კლასის შემდეგ კონსტრუქტორს:

ლისტინგი 8. ობიექტის კლონის კონსტრუირება:

```
BoxWeight(BoxWeight ob) {
        super(ob);
        weight = ob.weight;
```

```
}
```

კონსტრუქტორი super() გადასცემს BoxWeight ტიპის ობიექტს და არა Box ტიპისას. მიუხედავად ამისა მაინც ხდება Box(Box ob) კონსტრუქტორის გამოძახება. როგორც ზემოთ აღვნიშნეთ, სუპერკლასის ცვლადი შეიძლება გამოყენებული იქნას ნებისმიერი ამ კლასის მემკვიდრე კლასის ობიექტზე მითითებისათვის. ამრიგად, BoxWeight ტიპის ობიექტი შეიძლება გადაეცეს Box კლასის კონსტრუქტორს. რა თქმა უნდა Box კლასისათვის ცნობილი იქნება მხოლოდ საკუთარი წევრები.

თავი მოვუყაროთ კონსტრუქტორ super()-ის გამოყენების ძირითად კონცეფციებს: როდესაც ქვეკლასი იძახებს კონსტრუქტორ super()-ს, იგი იძახებს მისი უშუალო სუპერკლასის კონსტრუქტორს. super()-ი ყოველთვის მიუთითებს იმ სუპერკლასს, რომელიც იერარქიულად გამომძახებელი კლასის უშუალოდ ზემოთ იმყოფება. ეს გამონათქვამი სწორია მრავალდონიანი იერარქიის შემთხვევაშიც. გარდა ამისა, ოპერატორი super()-ი ყოველთვის უნდა იყოს პირველი შესრულებადი ოპერატორი ქვეკლასის კონსტრუქტორში.

super-ის მეორე გამოყენება

საკვანძო სიტყვა super-ის გამოყენების მეორე ფორმა მოქმედებს საკვანძო სიტყვა this-ის მსგავსად, ოღონდ super ყოველთვის მიუთითებს ქვეკლასის შესაბამის სუპერკლასს. მისი გამოყენების ზოგადი ფორმა ასე ჩაიწერება:

super.<წევრი>

აქ <წევრი> შეიძლება იყოს ეგზემპლარის რომელიმე მეთოდი ან ცვლადი.

super-ის გამოყენების მეორე ფორმა განსაკუთრებით მოსახერხებელია ისეთ სიტუაციებში, როდესაც ქვეკლასის წევრების სახელი გადაფარავს იმავე სახელის მქონე სუპერკლასის წევრებს. განვიხილოთ კლასების მარტივი იერარქია:

ლისტინგი 9. super-ის გამოყენება დაფარულ წევრზე წვდომისათვის:

```
class A {
   int i;
}
// ქვეკლასის შექმნა A კლასის გაფართოებით
class B extends A {
   int i; // ეს i ცვლადი ფარავს A კლასის i ცვლადს
   B(int a, int b) {
      super.i = a; // A კლასის i
      i = b; // B კლასის i
   }
   void show() {
      System.out.println("i სუპერკლასში: " + super.i);
      System.out.println("i ქვეკლასში: " + i);
   }
}
class UseSuper {
   public static void main(String args[]) {
      B subOb = new B(1, 2);
      subOb.show();
   }
}
```

ამ პრორამის შედეგი:

i სუპერკლასში: 1

i ქვეკლასში: 2

მართალია B კლასში ეგზემპლარის ცვლადი i ფარავს A კლასის i ცვლადს, მაგრამ super-ის გამოყენება საშუალებას იძლევა მივმართოთ სუპერკლასში განსაზღვრულ i ცვლადს. ასევე შესაძლებელია super-ის გამოყენება ქვეკლასის მიერ გადაფარული მეთოდების გამოსაძახებელად.

მრავალდონიანი იერარქიის შექმნა

აქამდე ჩვენ განვიხილავდით კლასების მარტივ იერარქიას, რომლებიც შედგებოდა მხოლოდ სუპერკლასისა და ქვეკლასისაგან. შესაძლებელია აიგოს ისეთი იერარქიები, რომლებიც შეიცავენ მემკვიდრეობის ნებისმიერი რაოდენობის დონეებს. როგორც უკვე აღვნიშნეთ, სავსებით შესაძლებელია ქვეკლასის გამოყენება სხვა ქვეკლასის სუპერკლასად. მაგალითად, კლასი C შეიძლება იყოს B კლასის ქვეკლასი, რომელიც, თავის მხრივ, წარმოადგენს A კლასის ქვეკლასს. ასეთ სიტუაციებში ყოველ ქვეკლასს მემკვიდრეობით გადაეცემა ყველა მისი სუპერკლასის ყველა თვისება. ჩვენს მაგალითში, C კლასს მემკვიდრეობით გადაეცემა B და A კლასების ყველა თვისება. მრავალდონიანი იერარქიის მაგალითად განვიხილოთ შემდეგი პროგრამა. ამ პროგრამაში BoxWeight გამოყენებულია სუპერკლასად და იქმნება ქვეკლასი Shipment. კლასი Shipment-ს მემკვიდრეობით გადაეცემა BoxWeight და Box კლასების ყველა თვისება და ემატება ახალი ველი cost, რომელიც აღნიშნავს ასეთი პაკეტის მიწოდების ღირებულებას.

ლისტინგი 10. BoxWeight კლასის გაფართოება მიტანის ღირებულების გათვალისწინებით:

```
class Box {
   private double width;
   private double height;
   private double depth;
   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;
   }
   Box () {
      width = -1;
      height = -1;
      depth = -1;
   }
   Box(double len) {
      width = height = depth = len;
   }
   double volume() {
      return width * height * depth;
   }
}
class BoxWeight extends Box {
   double weight; // პარალელეპიპედის წონა
   BoxWeight(BoxWeight ob) {
      super(ob);
      weight = ob.weight;
   }
   BoxWeight(double w, double h, double d, double m){
      super(w, h, d);
      weight = m;
   }
   BoxWeight() {
```

```
      super();
      weight = 1;
   }
   BoxWeight(double len, double m) {
      super(len);
      weight = m;
   }
}
class Shipment extends BoxWeight (
   double cost;
   Shipment(Shipment ob) {
      super (ob) ;
      cost = ob.cost;
   }
   Shipment(double w, double h, double d, double m,
   double c) {
      super(w, h, d, m);
      cost = c;
   }
   Shipment() {
      super();
      cost = 1;
   }
   Shipment(double len, double m, double c) {
      super (len, m);
      cost = c;
   }
}
class DemoShipment {
   public static void main(String args[]) {
      Shipment shipment1 = new Shipment(10, 20, 15, 10,
      3.41);
      Shipment shipment2 = new Shipment(2, 3, 4, 0.76,
      1.28);
      double vol;
      vol = shipment1.volume();
      System.out.println(" shipment1 მოცულობაა " +
      vol);
      System.out.println("shipment1 წონაა "+
      shipment1.weight);
```

```
        System.out.println("მიტანის ფასი: $" +
        shipment1.cost);
        System.out.println();
        vol = shipment2.volume();
        System.out.println("shipment2 მოცულობაა "+ vol) ;
        System.out.println("shipment2 წონაა "+
        shipment2.weight);
        System.out.println("მიტანის ფასი: $"+
        shipment2.cost);
    }
}
```

პროგრამის შედეგი:

shipment1 მოცულობაა 3000.0
shipment1 წონაა 10.0
მიტანის ფასი: $3.41
shipment2 მოცულობაა 24.0
shipment2 წონაა 0.76
მიტანის ფასი: $1.28

მემკვიდრეობითობის გამო Shipment კლასს შეუძლია გამოიყენოს ადრე აღწერილი Box და BoxWeight კლასები, იგი ამატებს მხოლოდ იმ დამატებით ინფორმაციას, რომელიც საჭიროა საკუთრივ მისი სპეციალიზებული გამოყენებისათვის. სწორედ ესაა მემკვიდრეობითობის ერთ-ერთი მთავარი ღირსება. იგი საშუალებას იძლევა ხელმეორედ იქნას გამოყენებული კოდი.

ბოლოს ნაჩვენები მაგალითი აკეთებს მნიშვნელოვანი ასპექტის დემონსტრირებას: კონსტრუქტორი super() ყოველთვის მიმართავს იერარქიაში უახლოესი სუპერკლასის კონსტრუქტორს. BoxWeight კლასის კონსტრუქტორი super() იძახებს Box კლასის კონსტრუქტორს. თუ კლასების

იერარქიაში სუპერკლასის კონსტრუქტორი მოითხოვს მისთვის პარამეტრების გადაცემას, მაშინ ყველა ქვეკლასმა ეს პარამეტრები უნდა გადასცეს „ესტაფეტით“. ეს გამონათქვამი ჭეშმარიტია იმის და მიუხედავად ჭირდება თუ არა ქვეკლასს საკუთარი პარამეტრები.

კონსტრუქტორების გამოძახების მიმდევრობა

რა მიმდევრობით ხდება იერარქიული კლასების შემთხვევაში კონსტრუქტორების გამოძახება? მაგალითად, ვთქვათ A სუპერკლასია, ხოლო B ქვეკლასი. რომელი კონსტრუქტორი გამოიძახება უფრო ადრე A თუ B? კლასების იერარქიაში კონსტრუქტორები გამოიძახება მემკვიდრეობითობის მიმდევრობის მიხედვით, დაწყებული სუპერკლასიდან, დამთავრებული ქვეკლასით. უფრო მეტიც, ვინაიდან super() უნდა იყოს პირველი შესრულებადი ოპერატორი ქვეკლასის კონსტრუქტორში, ეს მიმდევრობა რჩება უცვლელად იმისდა მიუხედავად გამოიყენება თუ არა super()-ის ეს ფორმა. თუ კონსტრუქტორში super()-ი არ გამოიყენება, მაშინ პროგრამა გამოიყენებს თითოეული სუპერკლასის კონსტრუქტორს, რომლებიც მოცემულია ან ნაგულისხმევი წესით (სტანდარტულად), ან რომელიც არ შეიცავს პარამეტრებს. შემდეგ მაგალითში განხილულია კონსტრუქტორების შესრულების მიმდევრობა:

ლისტინგი 11. კონსტრუქტორების გამოძახების მიმდევრობის დემონსტრირება:

```
class A {
```

```
   A() {
      System.out.println(" A კონსტრუქტორის შიგნით ");
   }
}

class B extends A {
   B() {
      System.out.println(" B კონსტრუქტორის შიგნით");
   }
}

class C extends B {
   C() {
      System.out.println(" C კონსტრუქტორის შიგნით");
   }
}

class CallingCons {
   public static void main(String args[]) {
      C c = new C();
   }
}
```

პროგრამის შედეგი:

A კონსტრუქტორის შიგნით
B კონსტრუქტორის შიგნით
C კონსტრუქტორის შიგნით

როგორც ხედავთ, კონსტრუქტორები გამოიძახება მემკვიდრეობის მიხედვით.

თუ დავფიქრდებით, მართლაც ცხადია, რომ კონსტრუქტორების გამოძახებას მემკვიდრეობის მიმდევრობის მიხედვით მართლაც აქვს აზრი. ვინაიდან სუპერკლასმა არაფერი არ იცის მისი ქვეკლასების შესახებ, ნებისმიერი ინიციალიზაცია, რომელიც მან უნდა შეასრულოს, სრულიად დამოუ-

კიდებელია და შესაძლოა აუცილებელიც იყოს ქვეკლასის მიერ განხორციელებული ყველა ინიციალიზაციისათვის. ამიტომ ის პირველი უნდა შესრულდეს.

მეთოდების ხელახალი განსაზღვრა (გადაფარვა)

კლასების იერარქიაში თუ ქვეკლასში მეთოდის სახელი და ტიპის სიგნატურა ემთხვევა სუპერკლასის მეთოდის ატრიბუტებს, ამბობენ რომ ქვეკლასის მეთოდი ხელახლა განსაზღვრავს (გადაფარავს) სუპერკლასის მეთოდს. როდესაც ქვეკლასიდან გამოიძახება გადაფარული მეთოდი, იგი ყოველთვის მიმართავს მეთოდის ქვეკლასში განსაზღვრულ ვერსიას. სუპერკლასში განსაზღვრული მეთოდი იქნება დაფარული. განვიხილოთ მაგალითი:

ლისტინგი 12. მეთოდის ხელახალი განსაზღვრა (აღწერა)

```
class A {
   int i, j;
   A(int a, int b) {
      i = a;
      j = b;
   }
   //  i და j მნიშვნელობების გამოტანა
   void show () {
      System.out.println("i და j: " + i + " " + j);
   }
}
class B extends A {
   int k;
   B(int a, int b, int c) {
      super (a, b);
      k = c;
   }
   /*k-ს გამოტანა, ეს მეთოდი ხელმეორედ აღწერს A
```

```
   კლასის show() მეთოდს*/
   void show () {
      System.out.println("k: "+ k);
   }
}
class Override {
   public static void main(String args[]) {
      B subOb = new B(1, 2, 3);
      // გამოიძახება B კლასის show() მეთოდი
      subOb.show();
   }
}
```

პროგრამის შედეგი:

k: 3

როდესაც პროგრამა B ტიპის ობიექტისათვის გამოიძახებს show() მეთოდს, იგი გამოიყენებს B კლასის შიგნით აღწერილი მეთოდის ვერსიას. ანუ, B კლასის შიგნით აღწერილი show() მეთოდის ვერსია გადაფარავს A კლასის შიგნით აღწერილ show() მეთოდს.

თუ საჭიროა გადაფარულ მეთოდზე წვდომა, რომელიც სუპერკლასში იმყოფება, ეს შეიძლება განვახორციელოთ საკვანძო სიტყვა super-ის გამოყენებით. მაგალითად, B კლასის შემდეგ ვერსიაში სუპერკლასში გამოცხადებული show() მეთოდის გამოძახება ხდება ქვეკლასის ვერსიის შიგნიდან. ეს საშუალებას იძლევა კონსოლზე გამოვიტანოთ ეგზემპლარის ყველა ცვლადი.

ლისტინგი 13.

```
class B extends A {
```

```
   int k;
   B(int a, int b, int c) {
      super (a, b);
      k = c;
   }
   void show () {
      // გამოიძახება A კლასის show() მეთოდი
      super.show();
      System.out.println("k: " + k);
   }
}
```

წინა პროგრამაში A კლასის ამ ვერსიის ჩანაცვლება კონსოლზე გამოიტანს:

i და j:1 2
k: 3

ამ ვერსიაში super.show() გამოიძახებს სუპერკლასში განსაზღვრულ show() მეთოდის ვერსიას.

მეთოდების გადაფარვა ხდება მხოლოდ მაშინ, როდესაც ორი მეთოდის სახელი და ტიპების სიგნატურები იდენტურია. წინააღმდეგ შემთხვევაში ორი მეთოდი ითვლება უბრალოდ გადატვირთულად. მაგალითად განვიხილოთ წინა მაგალითის შეცვლილი ვერსია:

ლისტინგი 14. განსხვავებული სიგნატურის მქონე მეთოდები არის გადატვირთული და არა გადაფარული

```
class A {
   int i, j;
   A(int a, int b){
      i = a;
      j = b;
   }
   void show () {
```

```
      System.out.println("i და: " + i + " " + j);
   }
}
class B extends A {
   int k;
   B(int a, int b, int c) {
      super (a, b);
      k = c;
   }
// show() მეთოდის გადატვირთვა
   void show(String msg) {
      System.out.println(msg + k);
   }
}
class Override {
   public static void main(String args[]) {
      B subOb = new B(1, 2, 3);
      //B კლასის show()მეთოდის გამოძახება
      subOb.show("k: ") ;
      // A კლასის show() მეთოდის გამოძახება
      subOb.show () ;
   }
}
```

პროგრამის შედეგი:

k: 3

i და j: 1 2

დინამიკური დისპეტჩერიზაცია

წინა მაგალითებში განხილული მეთოდების გადაფარვის მექანიზმი მხოლოდ სახელთა სივრცესთან მოხერხებულად სამუშაოდ არაა შექმნილი. ის რომ მხოლოდ ამ მიზნით ყოფილიყო რეალიზებული, მას მხოლოდ თეორიული ინტერესი ექნებოდა და მისი პრაქტიკული გამოყენებაც შეზღუდული იქნებოდა. მეთოდების ხელახლა გამოცხადება

საფუძვლად უდევს Java-ს ერთ-ერთ ყველაზე მძლავრ კონცეფციას - მეთოდების დინამიკურ დისპეტჩერიზაციას. მეთოდების **დინამიკურ დისპეტჩერიზაცია** წარმოადგენს მექანიზმს, რომლის საშუალებითაც მეთოდებზე მიმართვის ნებართვა წყდება (განისაზღვრება) უშუალოდ შესრულების დროს და არა კომპილაციის დროს.

მეთოდების დინამიკურ დისპეტჩერიზაცია მნიშვნელოვანია იმითაც, რომ მისი საშუალებით ხორციელდება შესრულების დროს პოლიმორფიზმი.

ამ კონცეფციის განხილვამდე ხელმეორედ ჩამოვაყალიბოთ ერთი ძირითადი პრინციპი: სუპერკლასზე მიმმართველი ცვლადი შეიძლება უკავშირდებოდეს (მიმართავდეს) ქვეკლასის ობიექტს. Java სისტემა ამ ფაქტს იყენებს შესრულების პროცესში მეთოდებზე მიმართვის ნებართვის გადასაწყვეტად. კლასებს შორის მემკვიდრეობითი იერარქიის არსებობისას, როგორც ვნახეთ შესაძლებელია ერთნაირი სიგნატურის მქონე რამდენიმე მეთოდი არსებობდეს. ასეთი მეთოდები რა თქმა უნდა არ შეიძლება ერთ კლასში არსებობდეს, მაგრამ მემკვიდრეობით იერარქიაში შემავალ რამდენიმე კლასში შესაძლებელია რამდენიმე ასეთი მეთოდი იქნას აღწერილი, ანუ მეთოდების ხელახალი გამოცხადება იქნას რეალიზებული. როგორც უკვე აღვნიშნეთ, სუპერკლასის ტიპის ცვლადს შეიძლება დაუკავშირდეს მისი ნებისმიერი ქვეკლასის ტიპის ობიექტი. გადასაწყვეტია პრობლემა: ასეთ შემთხვევაში როგორ მოვახერხოთ, რომ შესაძლებელი იყოს გადაფარული მეთოდის ნებისმიერი საჭირო ვერსიის გამოძახება?

ეს პრობლემა ასე წყდება: როდესაც გადაფარულ მეთოდზე მიმართვა ხდება სუპერკლასის ტიპის ცვლადიდან, Java ამ მეთოდის საჭირო ვერსიას ირჩევს იმისდა მიხედვით, თუ რომელი ობიექტია დაკავშირებული ამ მომენტში სუპერკლასის ტიპის ცვლადთან. ამრიგად მეთოდის არჩევანი წყდება შესრულების პროცესში. სხვადასხვა ტიპის ობიექტებთან დაკავშირებისას პროგრამა დაუკავშირდება გადაფარული მეთოდების სხვადასხვა ვერსიებს. ანუ, გადაფარული მეთოდის შესასრულებელი ვერსიის არჩევა ხდება ობიექტის ტიპის მიხედვით და არა მიმთითებელი ტიპის მიხედვით. ამრიგად, თუ სუპერკლასი შეიცავს ისეთ მეთოდს, რომელიც ხელახლა განსაზღვრულია (გადაფარულია) ამ კლასის რომელიმე ქვეკლასში, მაშინ სუპერკლასის ტიპის მიმთითებელ ცვლადთან სხვადასხვა ტიპის ობიექტების დაკავშირებისას პროგრამა შეასრულებს მეთოდის სხვადასხვა ვერსიას.

შემდეგ მაგალითში ილუსტრირებულია მეთოდების დინამიკური დისპეტჩერიზაცია:

ლისტინგი 15. მეთოდების დინამიკური დისპეტჩერიზაცია

```
class A {
   void callme() {
      System.out.println("A კლასის შიდა მეთოდი
      callme()");
   }
}
class B extends A {
   // callme() მეთოდის გადაფარვა
   void callme() {
      System.out.println("B კლასის შიდა მეთოდი
      callme()");
```

```
    }
}
class C extends A {
    // callme() მეთოდის გადაფარვა
    void callme() {
        System.out.println("C კლასის შიდა მეთოდი
        callme()");
    }
}
class Dispatch {
    public static void main(String args[]) {
        A a = new A(); // A ტიპის ობიექტი
        B b = new B(); // B ტიპის ობიექტი
        C c = new C(); // C ტიპის ობიექტი
        A r; // A ტიპის მიმთითებელი
        r = a; /* r უკავშირდება A-ს ობიექტს
        A-ში აღწერილი callme-ის გამოძახება*/
        r.callme();
        r = b; /* r უკავშირდება B-ს ობიექტს
        B-ში აღწერილი callme-ის გამოძახება*/
        r.callme();
        r = c; /* r უკავშირდება C-ს ობიექტს
        C-ში აღწერილი callme-ის გამოძახება*/
        r.callme();
    }
}
```

ამ პროგრამის შედეგი:

A კლასის შიდა მეთოდი callme()
B კლასის შიდა მეთოდი callme()
C კლასის შიდა მეთოდი callme()

ამ პროგრამაში აღწერილია ერთი სუპერკლასი A და მისი ორი ქვეკლასი B და C. ქვეკლასები B და C ხელახლა აღწერენ (გადაფარავენ) A კლასში აღწერილ callme() მეთოდს. main() მეთოდის შიგნით შექმნილია A, B და C ტიპის ობიექტები.

ასევე გამოცხადებულია A მიმთითებელი ტიპის ცვლადი r. პროგრამაში r ცვლადს რიგრიგობით მიენიჭება თითოეული კლასის ტიპის ობიექტი და ამ ცვლადიდან ხდება callme() მეთოდის გამოძახება. როგორც პროგრამის შედეგიდან ხედავთ, callme() მეთოდის შესასრულებელი ვერსია განისაზღვრება შესრულების დროს r ცვლადთან დაკავშირებული ობიექტის ტიპის მიხედვით. შესასრულებელი მეთოდის არჩევანი მიმთითებელი ცვლადის ტიპის მიხედვით რომ ხდებოდეს, მაშინ შედეგში გვექნებოდა სამჯერ მიმართვა A კლასის callme() მეთოდზე.

როგორც აღვნიშნეთ, მეთოდების გადაფარვით Java ანხორციელებს პოლიმორფიზმს შესრულების დროს. ობიექტზე ორიენტირებული დაპროგრამებისას პოლიმორფიზმის დიდი მნიშვნელობა განპირობებულია შემდეგი მიზეზით:

ზოგად (საერთო) კლასში შეიძლება აღვწეროთ მეთოდები, რომლებიც საერთო იქნება ყველა მის ქვეკლასში, ამავე დროს ქვეკლასებს უფლება ექნებათ განახორციელონ ზოგიერთი ან ყველა ამ მეთოდის კონკრეტული რეალიზაცია. ამ გზით ხდება Java-ში პოლიმორფიზმის მთავარი ასპექტის „ერთი ინტერფეისი, მრავალი მეთოდი“-ს რეალიზაცია.

პოლიმორფიზმის პრინციპის წარმატებით გამოყენების ძირითადი პირობაა იმის გაცნობიერება, რომ სუპერკლასები და ქვეკლასები წარმოადგენენ იერარქიას სპეციალიზაციის ხარისხის გაზრდით. მისი სწორი გამოყენების დროს, სუპერკლასში წარმოდგენილია ყველა ის ელემენტი, რომელთა უშუალო გამოყენებაც შეუძლია ქვეკლასებს.

სუპერკლასში ასევე განსაზღვრულია ის მეთოდები, რომლის რეალიზაცია ქვეკლასმა დამოუკიდებლად უნდა განახორციელოს. ეს ქვეკლასს საშუალებას აძლევს აღწეროს საკუთარი მეთოდები და ამავე დროს შეინარჩუნოს ერთგვაროვანი ინტერფეისი. ამგვარად, მემკვიდრეობითობისა და მეთოდების გადაფარვის გაერთიანებით, სუპერკლასს შეუძლია აღწეროს მეთოდების საერთო (ზოგადი) ფორმა, რომელიც გამოყენებული იქნება ყველა მისი ქვეკლასის მიერ.

დინამიკური ანუ შესრულების პროცესში რეალიზებული პოლიმორფიზმი ერთ-ერთი ყველაზე მძლავრი მექანიზმია ობიექტზე ორიენტირებულ არქიტექტურაში, რომელიც უზრუნველყოფს პროგრამული კოდის ხელმეორედ გამოყენებასა და საიმედოობას.

მეთოდების გადაფარვის გამოყენება

განვიხილოთ მეთოდების გადაფარვის უფრო რეალური მაგალითი. შემდეგი პროგრამა ქმნის სუპერკლასს Figure, რომელიც ინახავს ორგანზომილებიანი ობიექტის ზომებს. იგი ასევე აღწერს area() მეთოდს, რომელიც ობიექტის ფართობს გამოითვლის. პროგრამაში იქმნება ორი Figure-დან ნაწარმოები კლასი Rectangle და Triangle. თითოეული ეს ქვეკლასი ხელახლა აცხადებს ანუ გადაფარავს area() მეთოდს, რათა მან გამოითვალოს და დააბრუნოს შესაბამისად ოთხკუთხედისა და სამკუთხედის ფართობი.

ლისტინგი 16. შესრულების დროს პოლიმორფიზმის გამოყენება

```
class Figure {
   double dim1;
   double dim2;
   Figure(double a, double b) {
      dim1 = a;
      dim2 = b;
   }
   double area() {
      System.out.println("ფიგურის ფართობი
      განუსაზღვრელია");
      return 0;
   }
}
class Rectangle extends Figure {
   Rectangle(double a, double b) {
      super(a, b);
   }
   /* ოთხკუთხედისათვის ფართობის გამოთვლის area
   მეთოდის გადაფარვა*/
   double area() {
      System.out.println("ოთხკუთხედის არე");
      return dim1 * dim2;
   }
}
class Triangle extends Figure {
   Triangle(double a, double b) {
      super(a, b);
   }
   /* მეთოდ area()-ს გადაფარვა მართკუთხა
   სამკუთხედისთვის*/
   double area() {
      System.out.println("სამკუთხედის არე");
      return dim1 * dim2 / 2;
   }
}
class FindAreas {
   public static void main(String args[]) {
      Figure f = new Figure(10, 10);
      Rectangle r = new Rectangle(9, 5);
      Triangle t = new Triangle(10, 8);
```

```
        Figure figref;
        Figref = r;
        System.out.println("ფართობი ტოლია" +
        figref.area());
        Figref = t;
        System.out.println("ფართობი ტოლია" +
        figref.area());
        Figref = f;
        System.out.println("ფართობი ტოლია" +
        figref.area());
    }
}
```

პროგრამის შესრულების შედეგი:

სამკუთხედის არე
ფართობი ტოლია 45
სამკუთხედის არე
ფართობი ტოლია 40
ფიგურის ფართობი განუსაზღვრელია
ფართობი ტოლია 0

შესრულების დროს მემკვიდრეობითობისა და პოლიმორფიზმის ორმაგი ბუნება საშუალებას იძლევა განისაზღვროს ერთიანი ინტერფეისი, რომელსაც იყენებს სხვადასხვა მაგრამ მსგავსი ობიექტის ტიპები. ამ შემთხვევაში თუ ობიექტი ნაწარმოებია Figure სუპერკლასისაგან, მისი ფართობი შეიძლება გამოვითვალოთ area() მეთოდის გამოძახებით. ამ ოპერაციის შესრულებისას ინტერფეისი ერთნაირია იმისდა მიუხედავად რა ტიპისაა კონკრეტული ფიგურა.

აბსტრაქტული კლასები

ზოგჯერ საჭიროა განისაზღვროს სუპერკლასი, სადაც აღწერილია გარკვეული აბსტრაქციის სტრუქტურა, სადაც ყველა მეთოდის რეალიზაცია არაა განსაზღვრული. ანუ, ზოგჯერ საჭიროა შეიქმნას სუპერკლასი, რომელშიც აღწერილია მხოლოდ ზოგადი (საერთო) ფორმა, რომელსაც ერთობლივად გამოიყენებს ყველა მისი ქვეკლასი, რომლებიც თავის მხრივ დაუმატებენ კონკრეტულ დეტალებს. ასეთი კლასი განსაზღვრავს მეთოდების არსს (ძირითად შინაარსს), რომლების რეალიზაცია შემდგომში ქვეკლასებმა უნდა განახორციელონ. მაგალითად, ასეთი სიტუაცია შეიძლება წარმოიშვას, როდესაც სუპერკლასს არ შეუძლია მეთოდის სრულყოფილი რეალიზაცია მოახდინოს. სწორედ ასეთ სიტუაციას ჰქონდა ადგილი წინა მაგალითის Figure კლასში. ამ კლასში area() მეთოდის აღწერა უბრალოდ შაბლონია. იგი არ გამოითვლის და არ აბრუნებს რაიმე ტიპის ფიგურის ფართობს.

ხშირად კლასების საკუთარი ბიბლიოთეკის შექმნის პროცესში სუპერკლასში შემავალი მეთოდი სრულყოფილად არაა განსაზღვრული. ეს პრობლემა ორი გზით შეიძლება გადაიჭრას. პირველი გზა, როგორც ეს წინა მაგალითში იქნა ნაჩვენები, ესაა უბრალოდ, გამაფრთხილებელი შეტყობინების დაბეჭდვა. მართალია ეს გზა ზოგჯერ სასარგებლოა, განსაკუთრებით, როდესაც პროგრამის გამართვა ხდება, მაგრამ ჩვეულებრივ არაა საკმარისი. შეიძლება არსებობდეს მეთოდები, რომლებიც უნდა გადაიფაროს ქვეკლასებში, რათა მათ უფრო კონკრეტული

შინაარსი ჰქონდეთ. განვიხილოთ Triangle კლასი. იგი ყოველგვარ აზრს მოკლებულია თუ ამ კლასში არაა განსაზღვრული area() სამკუთხედის ფართობის გამოთვლის მეთოდი. ასეთ შემთხვევაში საჭიროა არსებობდეს ხერხი რათა დავრწმუნდეთ, რომ ქვეკლასში ნამდვილად ხდება ყველა საჭირო მეთოდის ხელახლა გამოცხადება (გადაფარვა). Java-ში ამ მიზნით გამოიყენება **აბსტრაქტული მეთოდი**.

იმის მოთხოვნა, რომ ქვეკლასმა აუცილებლად მოახდინოს გარკვეული მეთოდების ხელახლა გამოცხადება შესაძლებელია ასეთ კლასებზე abstract მოდიფიკატორის მითითებით. ზოგჯერ ამბობენ, რომ ასეთი მეთოდები განეკუთვნება ქვეკლასის კომპეტენციას, ვინაიდან სუპერკლასში მათი რეალიზაცია განსაზღვრული არაა. ამგვარად, ქვეკლასი ვალდებულია განახორციელოს ასეთი მეთოდების რეალიზაცია, ვინაიდან მას არ შეუძლია სუპერკლასში აღწერილი მეთოდის ვერსიის გამოყენება. აბსტრაქტული მეთოდის გამოცხადებისათვის გამოიყენება შემდეგი ზოგადი ფორმა:

abstarct <ტიპი> <მეთოდის სახელი>(<პარამეტრების სია>);

როგორც ხედავთ ამ ფორმაში მეთოდის ტანის აღწერა არ გვაქვს.

ნებისმიერი კლასი, რომელიც შეიცავს ერთ ან მეტ აბსტრაქტულ მეთოდს, ასევე უნდა გამოცხადდეს აბსტრაქტულ კლასად. ამისათვის საჭიროა საკვანძო სიტყვა abstract მოვათავსოთ საკვანძო სიტყვა class-ის წინ. აბსტრაქტული კლასი არ შეიძლება წარმოშობდეს რაიმე ობიექტებს. ანუ,

აბსტრაქტული კლასი არ შეიძლება კონკრეტიზებული იქნას new ოპერაციის საშუალებით. აბსტრაქტული კლასებით წარმოქმნილი ობიექტები გამოუსადეგარი იქნებოდნენ ვინაიდან ასეთ კლასში მისი ელემენტები არაა სრულყოფილად განსაზღვრული. არაა დასაშვები აბსტრაქტული კონსტრუქტორებისა და აბსტრაქტული სტატიკური ელემენტების გამოცხადებაც. აბსტრაქტული კლასის ნებისმიერმა ქვეკლასმა ან უნდა მოახდინოს მისი სუპერკლასის ყველა აბსტრაქტული მეთოდის სრულყოფილი რეალიზაცია, ან თვითონაც უნდა გამოცხადდეს როგორც აბსტრაქტული.

ქვემოთ ნაჩვენებია მარტივი კლასის მაგალითი, რომელიც შეიცავს აბსტრაქტულ მეთოდს, და ისეთი კლასი რომელიც ახდენს აბსტრაქტული მეთოდების რეალიზაციას

ლისტინგი 17. აბსტრაქციის გამოყენების მაგალითი

```
abstract class A {
   abstract void callme();
   /*აბსტრაქტული კლასი შეიძლება შეიცავდეს კონკრეტულ
მეთოდსაც*/
   void callmetoo() {
      System.out.println("ეს კონკრეტული მეთოდია");
   }
}
class B extends A {
   void callme() {
      System.out.println("B კლასის callme მეთოდის
      რეალიზაცია");
   }
}
class AbstractDemo {
   public static void main(String args[]) {
      B b = new B();
```

```
    b.callme();
    b.callmetoo();
  }
}
```

მიაქციეთ ყურადღება, რომ ამ პროგრამაში A კლასი არ შეიცავს რაიმე ობიექტების გამოცხადებას. როგორც ნათქვამი იყო, აბსტრაქტული კლასის რეალიზაცია (კონკრეტიზაცია) შეუძლებელია. კიდევ ერთი ნიუანსი: A კლასში აღწერილია კონკრეტული მეთოდი callmetoo(). ეს სავსებით დასაშვებია. აბსტრაქტული კლასები შეიძლება შეიცავდეს ნებისმიერი საჭირო რაოდენობის კონკრეტულ რეალიზაციებს.

მართალია აბსტრაქტული კლასები არ შეიძლება გამოყენებული იქნას ობიექტების შესაქმნელად, მაგრამ ისინი შეიძლება გამოვიყენოთ ობიექტებზე მიმართვისათვის (წვდომისათვის), ვინაიდან Java-ში შესრულების პროცესში პოლიმორფიზმი რეალიზებულია სუპერკლასის მიმთითებლის საშუალებით. ამიტომ საჭიროა არსებობდეს აბსტრაქტულ კლასზე მიმართვის შესაძლებლობა, რომელიც შეიძლება გამოყენებული იქნას ქვეკლასის ობიექტზე მისათითებლად. ამ თვისების გამოყენება ნაჩვენებია შემდეგ მაგალითში.

აბსტრაქტული კლასის გამოყენებით, შეიძლება სრულ ვყოთ ადრე შექმნილი Figure კლასი. ვინაიდან ფართობის გამოთვლა ნებისმიერი ორგანზომილებიანი ფიგურისათვის პრაქტიკულად შეუძლებელია, ამიტომ პროგრამის ამ ვერსიის Figure კლასში გამოცხადებულია area() მეთოდი როგორც abstract. ეს ნიშნავს, რომ Figure კლასიდან ნაწარმოები (შვილობილი) ყველა კლასი ვალდებულია ხელახლა განსაზღვროს area() მეთოდი:

ლისტინგი 18. აბსტრაქტული კლასებისა და მეთოდების გამოყენება

```
abstract class Figure {
   double dim1;
   double dim2;
   Figure(double a, double b) {
      dim1 = a;
      dim2 =b;
   }
   // აბსტრაქტული მეთოდი area
   abstract double area();
}
class Rectangle extends Figure {
   Rectangle(double a, double b) {
      super(a, b);
   }
   // area-ს ხელახალი აღწერა მართკუთხედისათვის
   double area() {
      System.out.println("მართკუთხედის არეში");
      return dim1 * dim2;
   }
}
class Triangle extends Figure {
   Triangle(double a, double b) {
      super(a, b);
   }
   // area-ს ხელახალი აღწერა სამკუთხედისათვის
   double area() {
      System.out.println("სამკუთხედის არეში");
      return dim1 * dim2 / 2;
   }
}
class AbstractAreas {
   public static void main(String args[]) {
      Figure f = new Figure(10, 10); //დაუშვებელია
      Rectangle r = new Rectangle(9, 5);
      Triangle t = new Triangle(10, 8);
      Figure figref; /* ეს ოპერატორი შესაძლებელია,
      ობიექტი არ იქმნება*/
```

```
        figref = r;
        System.out.println("ფართობია " +
        figref.area());
        figref = t;
        System.out.println("ფართობია " +
        figref.area());
    }
}
```

როგორც პროგრამის კომენტარიდანაც ჩანს, Figure კლასის ობიექტების შექმნა აღარ შეიძლება, ვინაიდან ეს კლასი აბსტრაქტულია. მისმა ყველა ქვეკლასმა უნდა მოახდინოს area() აბსტრაქტული მეთოდის ხელახალი გამოცხადება. ამაში დარწმუნდებით თუ შეეცდებით შექმნათ ქვეკლასი, რომელიც ხელმეორედ არ აღწერს ამ მეთოდს. ასეთ შემთხვევაში მოხდება კომპილაციის შეცდომა.

მართალია Figure ტიპის ობიექტის შექმნა შეუძლებელია, მაგრამ შესაძლებელია Figure ტიპის მიმთითებელი ტიპის ცვლადის შექმნა. ცვლადი figref გამოცხადებულია, როგორც Figure-ზე კავშირის ტიპი ანუ, ეს ცვლადი შეიძლება გამოყენებული იქნას Figure-დან ნაწარმოები (შვილობილი) კლასის ნებისმიერ ობიექტზე წვდომისათვის.

final მოდიფიკატორი და მემკვიდრეობითობა

არსებობს final მოდიფიკატორის გამოყენების სამი ხერხი. პირველი ჩვენ ადრე განვიხილეთ და გამოიყენებოდა კონსტანტის დასახელების ეკვივალენტად. დანარჩენი ორი გამოიყენება მემკვიდრეობითობისას. განვიხილოთ ეს ხერხები.

final მოდიფიკატორის გამოყენება ხელახალი გამოცხადების (გადაფარვის) აკრძალვისათვის

მართალია მეთოდების ხელახალი გამოცხადება Java-ს ერთ-ერთი ყველაზე მძლავრი საშუალებაა, მაგრამ ზოგჯერ სასურველია ეს მექანიზმი აიკრძალოს. მეთოდის გადაფარვა, რომ აიკრძალოს საჭიროა მისი აღწერის დასაწყისში მივუთითოთ საკვანძო სიტყვა final. მეთოდები, რომლებიც გამოცხადებულია final მოდიფიკატორით არ შეიძლება გადაიფაროს. პროგრამის შემდეგი ფრაგმენტში დემონსტრირებულია final-ის ასეთი გამოყენება:

ლისტინგი 19.

```
class A {
   final void meth() {
      System.out.println("ეს ფინალური მეთოდია");
   }
}
class B extends A {
void meth() { /* შეცდომა! ამ მეთოდის გადაფარვა არ
შეიძლება*/
      System.out.println("დაუშვებელია!");
   }
}
```

ვინაიდან მეთოდი meth()გამოცხადებულია, როგორც final, იგი არ შეუძლება გადაფარული იქნას B კლასში. გადაფარვის მცდელობა იწვევს კომპილაციის შეცდომას.

ზოგჯერ მეთოდები, რომლებიც გამოცხადებულია final მოდიფიკატორით, ხელს უწყობს პროგრამის მწარმოებლობის გაზრდას. კომპილატორს შეუძლია ამ მეთოდის

გამოძახება უშუალოდ ჩასვას სტრიქონში, ვინაიდან მან „იცის“, რომ მეთოდის გადაფარვა ქვეკლასში აღარ მოხდება. ხშირად final ტიპის პატარა ზომის მეთოდის გამოძახებისას კომპილატორმა ამ მეთოდის ბაიტ-კოდი შეიძლება ჩაწეროს გამომძახებელი მეთოდის კომპილირებულ კოდში, რითაც შემცირდება მეთოდის გამოძახებასთან დაკავშირებული დამატებითი სისტემური რესურსები. final მეთოდების გამომძახებელ კოდში ჩასმა ეს მხოლოდ პოტენციური შესაძლებლობაა. ჩვეულებრივ Java მეთოდების გამოძახებას ახდენს დინამიკურად, შესრულების დროს. ასეთ მიდგომას უძახიან **მოგვიანებით დაკავშირებას**. ვინაიდან final მეთოდები არ შეიძლება გადაფარული იქნას, ასეთ მეთოდზე მიმართვა შეიძლება გადაწყვეტილი იქნას კომპილაციის დროს. ასეთ მიდგომას უწოდებენ **ადრეულ დაკავშირებას**.

final მოდიფიკატორით მემკვიდრეობითობის აკრძალვა

ზოგჯერ მოითხოვება აიკრძალოს კლასის შემდგომი მემკვიდრეობითობა. ამისათვის კლასის გამოცხადების თავში საჭიროა final მოდიფიკატორის მითითება. კლასის final-ად გამოცხადება მის ყველა მეთოდს, არა ცხადად, final მეთოდად განსაზღვრავს. როგორც ხვდებით, კლასის ერთდროულად abstract და final გამოცხადება არ შეიძლება, ვინაიდან აბსტრაქტული კლასი პრინციპში წარმოადგენს დაუმთავრებელ კლასს და მისი მემკვიდრე კლასები უზრუნველყოფენ მეთოდების სრულ რეალიზაციას.

ქვემოთ ნაჩვენებია final ტიპის კლასის მაგალითი:

```
final class A {
   / / ...
 }
// შემდეგი კლასი დაუშვებელია
class B extends A {
      /* შეცდომა!! A კლასს არ შეიძლება ქვეკლასები
      ჰქონდეს*/
      // ...
}
```

როგორც კომენტარებიდან ხედავთ B კლასს არ შეიძლება A კლასი სუპერკლასად მიეთითოს, ვინაიდან A კლასი final კლასია.

Object კლასი

Java-ში განსაზღვრულია ერთი სპეციალური კლასი Object. ყველა დანარჩენი კლასი წარმოადგენს ამ კლასის ქვეკლასს. ანუ, Object არის ყველა სხვა კლასის სუპერკლასი. ეს ნიშნავს, რომ Object ტიპის მიმთითებელი ცვლადი შეიძლება დაუკავშირდეს ნებისმიერი სხვა კლასის ტიპის ობიექტს. გარდა ამისა, ვინაიდან მასივები რეალიზებულია კლასის სახით, ამიტომ Object ტიპის ცვლადი შეიძლება დაუკავ-შირდეს ნებისმიერ მასივს.

1 ცხრილში ჩამოთვლილია Object კლასში განსაზღვრული ის მეთოდები, რომლებიც წვდომადია ნებისმიერი კლასიდან.

ცხრილი 1.

მეთოდი	დანიშნულება
Object clone ()	ქმნის ობიექტის კლონს

boolean equals(Object object)	განსაზღვრავს ობიექტების ეკვივალენტობას
void finalize ()	გამოიძახება გამოუყენებელი ობიექტის განადგურებისას
Class getClass()	შესრულების დროს აბრუნებს ობიექტის კლასს
int hashCode ()	აბრუნებს გამომძახებელი მეთოდის ჰეშ-კოდს
String toString()	აბრუნებს ობიექტის აღმწერ სტრიქონს

ამ მეთოდებიდან getClass() გამოცხადებულია final-ად, ხოლო დანარჩენი მეთოდები შეიძლება გადაიფაროს.

მეთოდი equals() ადარებს ერთმანეთს ორი ობიექტის შემცველობას, თუ ობიექტები ეკვივალენტურია აბრუნებს true-ს, წინააღმდეგ შემთხვევაში false-ს.

მეთოდი toString() აბრუნებს სტრიქონს, რომელიც შეიცავს ობიექტის აღწერას. ეს მეთოდი ავტომატურად გამოიძახება ობიექტის println() მეთოდის გამოძახებისას. ბევრი კლასი ამ მეთოდს ხელახლა აცხადებს, რათა გამოტანილ ინფორმაციაში გათვალისწინებული იქნას კონკრეტული ტიპის ობიექტების თავისებურებები.

პაკეტები

პაკეტები წარმოადგენს Java-ს ერთ-ერთ ყველაზე ნოვატორულ კონცეფციას. პაკეტი - ესაა კლასების კონტეინერი, რომელიც გამოიყენება კლასების სახელების განსაზღვრის არეს (სახელ სივრცის) იზოლირებისათვის. მაგალითად, პაკეტი საშუალებას იძლევა შეიქმნას კლასი სახელით List, რომელიც შეიძლება შენახული იქნას ცალკე პაკეტში, და საჭირო აღარ იქნება ზრუნვა რომელიმე სხვა პაკეტში იგივე List სახელით შენახულ კლასის სახელებს შორის კონფლიქტზე. პაკეტები ინახება იერარქიული სტრუქტურის სახით და მათი ჩართვა რომელიმე კლასის აღწერაში ხდება ცხადი სახით.

წინა თავებში კლასების ყველა მაგალითში ჩვენ ვიყენებდით სახელებს სახელ სივრცის ერთი არედან. ეს ნიშნავდა, რომ სახელების კონფლიქტის თავიდან ასაცილებლად, ყოველი კლასის სახელი უნდა ყოფილიყო უნიკალური. გარკვეული პერიოდის შემდეგ, თუ არ იქნებოდა სახელ სივრცის მართვის გარკვეული მექანიზმი, შეიძლება წარმოქმნილიყო სიტუაცია, როდესაც მოსახერხებელი აღმწერი სახელების შერჩევა რთული გახდებოდა. გარდა ამისა, საჭირო გახდა ისეთი ხერხი, რომ კლასისათვის შერჩეული სახელი იყოს საკმარისად უნიკალური და კონფლიქტში არ მოდიოდეს სხვა პროგრამისტების მიერ შერჩეულ სახელებთან. ამ მიზნით Java-ში განხორციელებულია მექანიზმი, სახელ სივრცის ფრაგმენტებად დასაყოფად, მათი უკეთ მართვის მიზნით. ამ მექანიზმს წარმოადგენს პაკეტი. პაკეტი ერთ-

დროულად გამოიყენება, როგორც სახელების მინიჭებისა და ხილვადობის მართვის მექანიზმი.

პაკეტის შიგნით შეიძლება განისაზღვროს კლასები, რომლებიც არ იქნება წვდომადი ამ პაკეტის გარედან. ასევე შესაძლებელია განისაზღვროს კლასის წევრები, რომლებიც ხილვადია მხოლოდ იმავე პაკეტის წევრებისათვის. ასეთი მექანიზმი კლასებს საშუალებას აძლევს ფლობდნენ სრულ ინფორმაციას ერთმანეთის მიმართ, მაგრამ არ მიაწოდონ ეს ცნობები (ინფორმაცია) დანარჩენ სამყაროს (პაკეტებს).

პაკეტის აღწერა

პაკეტის შექმნა მარტივია: Java-ს საწყის ფაილში პირველ ოპერატორად უნდა ჩაისვას package ბრძანება. ყველა კლასი, რომელიც აღწერილი იქნება ამ ფაილში მიეკუთვნება მითითებულ პაკეტს. ოპერატორი package განსაზღვრავს სახელ სივრცეს, რომელშიც ინახება კლასები. თუ package ოპერატორი გამოტოვებულია, კლასის სახელები განთავსდება ნაგულისხმევ (სტანდარტულ) უსახელო პაკეტში. სწორედ ამიტომ აქამდე ჩვენ არ ვზრუნავდით პაკეტის აღწერაზე. ნაგულისხმევი (სტანდარტული) პაკეტების გამოყენება სავსებით საკმარისია ისეთი პატარა პროგრამებისათვის, რომლებსაც ჩვენ ვიყენებთ შესწავლის პროცესში, მაგრამ რეალური ამოცანების შემთხვევაში იგი საკმარისი არაა. ხშირ შემთხვევაში საჭიროა პაკეტის აღწერა.

package ოპერატორს აქვს ასეთი ზოგადი ფორმა:

package <პაკეტის სახელი>;

<პაკეტის სახელი> მიუთითებს პაკეტის დასახელებას. მაგალითად:

```
package MyPackage;
```

პაკეტების შესანახად Java სისტემა იყენებს ფაილური სისტემის კატალოგებს (ფოლდერებს). მაგალითად, ნებისმიერი კლასის .class ფაილები, რომლებიც გამოცხადებულია MyPackage პაკეტის შემადგენლობაში, უნდა ინახებოდეს MyPackage კატალოგში. ყურადღება უნდა გავამახვილოთ იმ ფაქტზე, რომ სიმბოლოების რეგისტრს მნიშვნელობა აქვს, ხოლო კატალოგის სახელი ზუსტად უნდა ემთხვეოდეს პაკეტის სახელს.

ერთი და იგივე package ოპერატორი შეიძლება გამოიყენებოდეს რამდენიმე ფაილში. ეს ოპერატორი უბრალოდ მიუთითებს პაკეტს, რომელსაც მიეკუთვნება ის კლასები, რომლებიც აღწერილია მიმდინარე ფაილში. ის არ კრძალავს იმას, რომ სხვა კლასები სხვა ფაილებში იყოს იმავე პაკეტის წევრი. რეალურ პროგრამებში გამოყენებულ პაკეტებში, როგორც წესი განაწილებულია მრავალი ფაილი.

Java-ში ნებადართულია პაკეტების იერარქიის შექმნა. ამისათვის გამოიყენება სიმბოლო წერტილი. მრავალდონიანი პაკეტის ოპერატორს ასეთი სახე აქვს:

package <პაკეტი1>[.<პაკეტი2>[.<პაკეტი3>]...];

პაკეტების იერარქია აისახება Java-ს დაპროექტების გარემოს ფაილურ სისტემაში. მაგალითად, Windows გარემოში პაკეტი, რომელიც ასეა გამოცხადებული:

```
package java.awt.image;
```

უნდა ინახებოდეს კატალოგში java\awt\image. საჭიროა გულდასმით მოხდეს პაკეტების დასახელებების არჩევა. არ შეიძლება პაკეტის სახელის შეცვლა იმ კატალოგის სახელის შეცვლის გარეშე, სადაც განთავსებულია კლასები.

პაკეტების ძიება და გარემოს პარამეტრი CLASSPATH

როგორც აღვნიშნეთ პაკეტების აისახება კატალოგებზე, ამიტომ ისმება კითხვა: Java-ს შემსრულებელმა გარემომ სად უნდა ეძებოს შექმნილი პაკეტები? ამ კითხვაზე პასუხი შემდეგი ნაწილებისგან შედგება:

- სტანდარტულად Java-ს შემსრულებელი გარემოსათვის ათვლის წერტილად ითვლება მიმდინარე, მუშა კატალოგი. შესაბამისად, თუ პაკეტი იმყოფება მიმდინარე კატალოგის ქვეკატალოგში, იგი მოძებნილი იქნება;

- კატალოგამდე გზა ან გზები შესაძლებელია მიეთითოს გარემოს პარამეტრზე CLASSPATCH-ზე მნიშვნელობის მინიჭებით;

- java და javac (შემსრულებელი და კომპილატორი) გამოძახებისას შესაძლებელია -classpatch პარამეტრის გამოყენება, რომელიც მიუთითებს საჭირო კლასებამდე გზებს.

მაგალითად, განვიხილოთ ასეთი პაკეტი:

```
package MyPack;
```

პროგრამამ MyPack პაკეტი რომ მიაგნოს (მოძებნოს), უნდა შესრულდეს ერთ-ერთი პირობა. ან პროგრამა უნდა

შესრულდეს იმ კატალოგიდან, რომელიც იმყოფება MyPack კატალოგის უშუალოდ ზემოთ, ან გარემოს პარამეტრი CLASSPATCH უნდა შეიცავდეს გზას MyPack კატალოგამდე, ან java ბრძანებით პროგრამის შესრულებისას -classpatch პარამეტრი უნდა უთითებდეს გზას MyPack კატალოგამდე.

ბოლო ორი ხერხის გამოყენებისას კლასამდე გზა არ უნდა შეიცავდეს თვით MyPack პაკეტს. იგი უბრალოდ უნდა უთითებდეს გზას ამ კატალოგამდე. მაგალითად, Windows გარემოში, თუ MyPack კატალოგამდე გზას ასეთი სახე აქვს:

c:\MyPrograms\Java\MyPack

მაშინ კლასის გზა MyPack-მდე იქნება ასეთი:

c:\MyPrograms\Java

განვიხილოთ პაკეტების გამოყენების მოკლე მაგალითი:

ლისტინგი 20. მარტივი პაკეტი

```
package Mypack;

class Balance {
   String name;
   double bal;
   Balance(String n, double b) {
      name = n;
      bal = b;
   }

   void show() {
      if (bal < 0)
           System.out.print("-->");
      System.out.println(name + ". $" + bal);
   }
}
```

```
class AccountBalance {
   public static void main(String args[]) {
      Balance current[] = new Balance[3];
      current[0] = new Balance("K. J. Fielding",
      123.23);
      current[1] = new Balance("Will Tell", 157.02);
      current[2] = new Balance("Tom Jackson",
      12.33);
      for (int i = 0; i < 3; i++)
         current[i].show();
   }
}
```

ამ ფაილს დავარქვათ AccountBalance.java და მოვათავსოთ იგი MyPack კატალოგში.

მოვახდინოთ ფაილის კომპილაცია. დავრწმუნდეთ, რომ კომპილირებული .class ფაილები ასევე მოთავსებულია MyPack კატალოგში. შევასრულოთ AccountBalance კლასი შემდეგი ბრძანებით:

java MyPack.AccountBalance

ამ ბრძანების შესრულებისას, მიმდინარე კატალოგი უნდა იყოს MyPack კატალოგის უშუალოდ ზედა კატალოგი, ან პარამეტრი CLASSPATCH უნდა შეიცავდეს გზას MyPack კატალოგამდე.

წვდომის დაცვა

წინა პარაგრაფებში ჩვენ ვნახეთ, რომ კლასის პრივატულ წევრზე წვდომა დასაშვებია მხოლოდ ამავე კლასის სხვა წევრებისათვის. პაკეტების გამოყენება წვდომის მართვის მექანიზმს კიდევ ერთ განზომილებას უმატებს. ჩვენ ახლა განვიხილავთ Java-ში რეალიზებულ მექანიზმებს, რომლებიც

უზრუნველყოფენ დაცვის მრავალ დონეს, მართავენ მეთოდებისა და ცვლადების ხილვადობას კლასების, ქვეკლასების და პაკეტების შიგნით.

კლასები და პაკეტები ერთდროულად გამოიყენება, სახელების სივრცის და ცვლადების/მეთოდების განსაზღვრის არეს, როგორც ინკაფსულაციისათვის ისე შესანახად. პაკეტები ასრულებენ კონტეინერების როლს კლასებისა და სხვა შემადგენელი პაკეტებისათვის. კლასები წარმოადგენს მონაცემებისა პროგრამული კოდის კონტეინერს. კლასი Java-ში აბსტრაქციის მინიმალური ერთეულია. კლასებისა და პაკეტების ურთიერთ მიმართების გათვალისწინებით Java-ში არსებობს კლასის წევრებს შორის ხილვადობის ოთხი კატეგორია:

- ქვეკლასები ერთ პაკეტში;
- კლასები ერთ პაკეტში, რომლებიც ქვეკლასები არ არიან;
- ქვეკლასები სხვადასხვა პაკეტებში;
- კლასები, რომლებიც არ იმყოფებიან ერთ პაკეტში და არც არიან ქვეკლასები.

წვდომის სამი მოდიფიკატორი private, public და protected უზრუნველყოფენ დაცვის სხვადასხვა დონეს ამ კატეგორიებისათვის. ცხრილში 2 ნაჩვენებია მათ შორის დამოკიდებულებები:

ცხრილი 2. კლასის წევრებზე წვდომა

	private	მოდიფიკატორის გარეშე	privatepr otected	protected	public
იმავე კლასში	კი	კი	კი	კი	კი
ქვეკლასები იმავე პაკეტში	არა	კი	კი	კი	კი
არა ქვეკლასები იმავე პაკეტში	არა	კი	არა	კი	კი
ქვეკლასები სხვადასხვა პაკეტებში	არა	არა	კი	კი	კი
დამოუკიდებელი კლასები სხვადასხვა პაკეტებში	არა	არა	არა	არა	კი

წვდომის დაცვის მაგალითი

შემდეგ მაგალითში დემონსტრირებულია წვდომის მმართველი მოდიფიკატორების ყველა ვარიანტის კომბინაცია.

პირველ პაკეტის საწყის ფაილში აღწერილია სამი კლასი Protection, Derived და SamePackage. პირველი კლასში აღწერილია ოთხი int ტიპის ცვლადი, თითოეულ მათგანს მითითებული აქვს განსხვავებული წვდომის დაცვის მოდიფიკატორი. ცვლადი n აღწერილია დაცვის სტანდარტული (ნაგულისხმევი) დონით, n_pri აღწერილია, როგორც - private, n_pro როგორც - protected, ხოლო n_pub როგორც - public.

ამ მაგალითში ყველა სხვა კლასი შეეცდება მიმართოს ამ კლასის ეგზემპლარის ცვლადებს. ის სტრიქონები, რომელთა კომპილაცია დაუშვებელია წვდომის წესების დარღვევის გამო, ჩასმულია კომენტარებში. ყოველი ამ სტრიქონის წინ ჩაწერილია კომენტარი, სადაც პროგრამის ის წერტილებია მითითებული, საიდანაც შესაძლებელია დაცვის ამ დონისთვის მიმართვა.

მეორე Derived კლასი წარმოადგენს Protection კლასის ქვეკლასს და ორივე ეს კლასი იმყოფება p1 პაკეტში. Protection კლასი Derived კლასს უფლებას აძლევს მიმართოს მის ყველა ცვლადს გარდა n_pri ცვლადისა, რომელიც გამოცხადებულია private მოდიფიკატორით. მესამე კლასი SamePackage არ წარმოადგენს Protection კლასის ქვეკლასს, მაგრამ იგი იმყოფება იგივე პაკეტში. ამიტომ მას წვდომის უფლება აქვს ყველა ცვლადთან, გარდა n_pri ცვლადისა.

Protection.java ფაილი შეიცავს შემდეგ კოდს:

ლისტინგი 21.

```
package p1;

public class Protection {
   int n = 1;
   private int n_pri = 2;
   protected int n_pro = 3;
   public int n_pub = 4;
   public Protection() {
      System.out.println("ბაზისური კლასის
      კონსტრუქტორი");
      System.out.println("n = " + n);
      System.out.println("n_pri = " + n_pri);
      System.out.println("n_pro = " + n_pro);
```

```
      System.out.println("n_pub =" + n_pub);
   }
}
```

ფაილი Derived.java შეიცავს შემდეგ კოდს:

ლისტინგი 22.

```
package p1;

class Derived extends Protection {
   Derived() {
      System.out.println("ქვეკლასის კონსტრუქტორი");
      System.out.println("n = " + n);
      // წვდომადია მხოლოდ კლასისათვის
      // System.out.println{"n_pri = "4 + n_pri);
      System.out.println("n_pro = " + n_pro);
      System.out.println("n_pub = " + n_pub);
   }
}
```

ფაილი SamePackage.java შეიცავს შემდეგ კოდს:

ლისტინგი 23.

```
package p1;

class SamePackage {
   SamePackage() {
      Protection p = new Protection();
      System.out.println("ამავე პაკეტის
      კონსტრუქტორი");
      System.out.println("n = " + p.n);
      // class only წვდომადია მხოლოდ კლასისათვის
      // System.out.println("n_pri = " + p.n_pri);
      System.out.println("n_pro " + p. n_pro);
      System.out.println("n_pub " + n_pub);
   }
}
```

ქვემოთ ნაჩვენებია p2 პაკეტში მოთავსებული საწყისი კოდები. ამ პაკეტში აღწერილია ორი კლასი, რომლებშიც ასახუ-

ლია დარჩენილი ორი წვდომის მართვის სიტუაცია. პირველი კლასია Protection2 - რომელიც არის p1.Protection კლასის ქვეკლასი. მას აქვს თავისი სუპერკლასის (p1.Protection) ყველა ცვლადთან წვდომის უფლება, გარდა n_pri-სა (ვინაიდან იგი პრივატულია) და n-ისა, რომელიც გამოცხადებულია სტანდარტული (ნაგულისხმევი) წვდომის უფლებით. გავიხსენოთ, რომ სტანდარტული წვდომის რეჟიმი ნებას რთავს მიმართოს იმავე კლასიდან ან იმავე პაკეტიდან და არა სხვა პაკეტის ქვეკლასებიდან.

კლასს OtherPackage-ს უფლება აქვს მიმართოს, მხოლოდ ერთ ცვლადს n_pub, ვინაიდან იგი გამოცხადებულია, როგორც public.

ფაილი Protection2.java შეიცავს შემდეგ კოდს:

ლისტინგი 24.

```
package p2;
class Protection2 extends p1.Protection {
   Protection2() {
      System.out.println("სხვა პაკეტიდან ნაანდერძევი
      კონსტრუქტორი");
//       წვდომადია მხოლოდ ამ კლასისთვის ან პაკეტისთვის
//       System.out.println ("n = " + n);
//       წვდომადია მხოლოდ ამ კლასისთვის
//       System.out.println("n_pri = " + n_pri);
      System.out.println("n_pro = "+ n_pro);
      System.out.println("n_pub " + n_pub);
   }
}
```

ფაილი OtherPackage.java შეიცავს შემდეგ კოდს:

ლისტინგი 25.

```
package p2;

class OtherPackage {
   OtherPackage() {
      p1.Protection p = new p1.Protection();
      System.out.println("სხვა პაკეტის
      კონსტრუქტორი");
      /* წვდომადია მხოლოდ ამ კლასისთვის ან
      პაკეტისთვის*/
      // System.out.println("n ="+ p.n);
      // წვდომადია მხოლოდ ამ კლასისთვის
      // System.out.println("n_pri = " + p.n_pri);
      /* წვდომადია მხოლოდ ამ კლასისთვის,
      ქვეკლასისთვის ან პაკეტისთვის*/
      // System.out.println ("n_pro = " + p.n_pro);
      System.out.println("n_pub = " + p. n_pub);
   }
}
```

ამ ორი პაკეტის მუშაობის შესამოწმებლად შეიძლება გამოვიყენოთ შემდეგი ორი ტესტური ფაილი. p1 პაკეტის ტესტურ ფაილს ასეთი სახე აქვს:

ლისტინგი 26. სადემონსტრაციო პაკეტი p1

```
package p1;

// p1 პაკეტის სხვადასხვა კლასების კონკრეტიზაცია
public class Demo {
   public static void main(String args[]) {
      Protection ob1 = new Protection();
      Derived ob2 = new Derived();
      SamePackage ob3 = new SamePackage();
   }
}
```

p2 პაკეტის ტესტურ ფაილს ასეთი სახე აქვს:

ლისტინგი 27. სადემონსტრაციო პაკეტი p2.

```
package p2;
```

```
// p2 პაკეტის სხვადასხვა კლასების კონკრეტიზაცია
public class Demo {
   public static void main(String args[]) {
     protection2 ob1 = new Protection2();
     OtherPackage ob2 = new OtherPackage();
   }
}
```

პაკეტების იმპორტირება

თუ გავიხსენებთ, რომ პაკეტები ახორციელებენ სხვადასხვა კლასების ერთმანეთისგან იზოლაციის ეფექტურ მექანიზმს, ცხადი გახდება, რატომ ინახავს Java თავის ჩაშენებულ კლასებს (კლასები რომლებიც სისტემას გამზადებული მოჰყვება) პაკეტებში. Java-ს არცერთი ძირითადი კლასი არ ინახება სტანდარტულად გამოყენებულ უსახელო პაკეტში. ყველა სტანდარტული კლასი ინახება რომელიმე დასახელების მქონე პაკეტში. ვინაიდან პაკეტების შიგნით კლასები ცალსახად უნდა იყოს განსაზღვრული კლასის ან მათი პაკეტების სახელებით, ამიტომ სრული სახელები შეიძლება ძალიან გრძელი გამოვიდეს. სრული სახელი, როგორც გახსოვთ, შედგება პაკეტების გზისა და თვით კლასის სახელისაგან, რომლებიც ერთმანეთისაგან წერტილითაა გამოყოფილი. სახელები უფრო ლაკონურად, რომ ჩაიწეროს Java-ში შემოტანილია ოპერატორი import. კლასის იმპორტირების შემდეგ მას შეძლება მივმართოთ უშუალოდ, მხოლოდ მისი სახელის საშუალებით. ოპერატორი import გამოიყენება მხოლოდ პროგრამისტების კომფორტულობისათვის და ტექნიკური თვალსაზრისით აუცილებელი არაა. როდესაც გამოყენებით პროგრამაში საჭიროა რამდენიმე ათე-

ულ კლასზე მიმართვა, ოპერატორი import მნიშვნელოვნად ამცირებს შესატანი კოდის მოცულობას.

Java პროგრამის საწყის კოდში import ოპერატორები უნდა მოდიოდეს უშუალოდ package ოპერატორის შემდეგ (თუ ასეთი არსებობს). იგი წინ უნდა უსწრებდეს კლასების აღწერას. ოპერატორ import-ს ასეთი ზოგადი ფორმა აქვს:

import <პაკეტი1>[.<პაკეტი2>...].(<კლასის სახელი>|*);

ამ ფორმაში <პაკეტი1> არის ზედა დონის პაკეტის სახელი, <პაკეტი2> არის ქვეპაკეტის სახელი. ისინი ერთმანეთისაგან გამოიყოფა წერტილით (.). პაკეტების ჩალაგების სიღრმე პრაქტიკულად არაფრით არ შემოიფარგლება, გარდა ფაილური სისტემის შეზღუდვისა. <კლასის სახელი> შეიძლება მოცემული იყოს ცხადად ან სიმბოლო (*) ვარსკვლავის საშუალებით, რომელიც კომპილატორს მიუთითებს, რომ საჭიროა მთელი პაკეტის იმპორტირება. პროგრამის შემდეგ ფრაგმენტში დემონსტრირებულია ორივე ფორმა:

```
import java.util.Date;
import java.io.*;
```

აღსანიშნავია, რომ ვარსკვლავიანი ფორმის გამოყენებისას შეიძლება მნიშვნელოვნად გაიზარდოს კომპილაციის დრო, განსაკუთრებით როდესაც ხდება რამდენიმე დიდი პაკეტის იმპორტირება. ამიტომ, გირჩევთ ცხადად მიუთითოთ მხოლოდ იმ კლასის სახელები, რომელთა გამოყენებასაც აპირებთ და არ მოახდინოთ მთელი პაკეტის სრული იმპორტირება. თუმცა ვარსკვლავიანი ფორმის გამოყენება არანაირ ზეგავლენას არ ახდენს არც პროგრამის შესრულების

პროცესის დროის გაზრდაზე და არც შემსრულებელი პროგრამის ზომაზე.

ყველა ის სტანდარტული კლასი, რომელიც Java სისტემას მოჰყვება ინახება java კლასში. ენის ძირითადი ფუნქციები ინახება java.lang პაკეტში. ჩვეულებრივ, ყოველი კლასი ან პაკეტი, რომლის გამოყენებაც გვინდა, უნდა იქნას იმპორტირებული. ვინაიდან Java სისტემა უსარგებლოა java.lang პაკეტში განსაზღვრული ბევრი ფუნქციის გარეშე, ამიტომ კომპილატორი ყველა პროგრამისათვის ახდენს ამ პაკეტის არაცხად იმპორტირებას. ეს ეკვივალენტურია, რომ ყოველ პროგრამაში ეწეროს ასეთი სტრიქონი:

```
import java.lang.*;
```

ორი განსხვავებული პაკეტიდან ვარსკვლავიანი ფორმით ერთი და იგივე სახელის მქონე კლასების იმპორტირებისას, კომპილატორი არავითარ რეაგირებას არ მოახდენს, თუ არ მოხდება რომელიმე ამ კლასის გამოყენების მცდელობა. ამ შემთხვევაში წარმოიქმნება კომპილაციის შეცდომა და საჭირო გახდება კლასის სახელის ცხადი ჩაწერა პაკეტის სახელის მითითებით.

კლასის სრულად განსაზღვრული სახელი პაკეტების იერარქიის მითითებით შეიძლება გამოვიყენოთ ყველგან, სადაც დასაშვებია კლასის სახელის გამოყენება. მაგალითად, შემდეგ ფრაგმენტში გამოყენებულია იმპორტირების ოპერატორი:

```
import java.util.*;

class MyDate extends Date {
```

```
}
```

იგივე მაგალითი იმპორტირების ოპერატორის გარეშე ასე ჩაიწერება:

```
class MyDate extends java.util.Date {
}
```

ამ ვერსიაში ობიექტი Date სრულადაა განსაზღვრული.

როგორც ცხრილი 2-დან ჩანს პაკეტის იმპორტირებისას, იმ კოდის კლასებს, რომლებიც ამ იმპორტირებას ახდენენ და არ არიან პაკეტის ქვეკლასები, უფლება აქვთ მიმართონ მხოლოდ პაკეტის იმ წევრებს, რომლებიც გამოცხადებულია როგორც public. მაგალითად, თუ საჭიროა, ადრე განხილული MyPack პაკეტის Balance კლასი გახდეს დამოუკიდებლად წვდომადი MyPack პაკეტის გარეთ, იგი გამოცხადებული უნდა იქნას როგორც public და უნდა მოთავსდეს ცალკე ფაილში:

ლისტინგი 28.

```
/* კლასი Balance, მისი კონსტრუქტორი და მეთოდი show() წარმოადგენენ ყველასათვის წვდომადს. ეს ნიშნავს, რომ ამ კოდის გარეთ ისინი შეიძლება გამოყენებული იქნას პაკეტის არაქვეკლასი კოდიდანაც */
public class Balance {
   String name;
   double bal;
   public Balance(String n, double b) {
      name = n;
      bal = b;
   }
   public void show() {
      if (bal<0)
         System.out.print("-->");
```

```
      System.out.println(name + ": $" + bal);
   }
}
```

როგორც ხედავთ, Balance კლასი გამოცხადებულია, როგორც public. მისი კონსტრუქტორი და show() მეთოდი ასევე public-ია. ეს ნიშნავს, რომ ისინი წვდომადია MyPack პაკეტის გარე ნებისმიერი კოდისთვის. მაგალითად, TestBalance კლასი ახდენს MyPack პაკეტის იმპორტირებას, ამიტომ შეუძლია გამოიყენოს Balance კლასი:

ლისტინგი 29.

```
import MyPack.*;

class TestBalance {
   public static void main(String args[]) {
      /* ვინაიდან კლასი Balance გამოცხადებულია
      public, ის შეიძლება გამოყენებული იქნას, მოხდეს
      მისი კონსტრუქტორის გამოძახება */
      Balance test = new Balance(HJ. J. Jaspers H ,
      99.88);
      test.show(); /* ასევე შეგვიძლია გამოვიძახოთ
      show() მეთოდი*/
   }
}
```

ექსპერიმენტის მიზნით წაშალეთ მოდიფიკატორი public კლასიდან Balance, შემდეგ შეეცადეთ შეასრულოთ TestBalance კლსის კომპილაცია. ეს გამოიწვევს კომპილატორის შეცდომას.

ინტერფეისები

საკვანძო სიტყვა interface-ის გამოყენებით შესაძლებელია მოვახდინოთ კლასის ინტერფეისის აბსტრაგირება მისი რეა-

ლიზაციისაგან. ანუ ამ საკვანძო სიტყვის მითითებით შეიძლება განვსაზღვროთ ის მოქმედებები, რომლებიც უნდა შეასრულოს კლასმა და არა ის, თუ კონკრეტულად როგორ უნდა მოახდინოს მან ეს მოქმედებები. სინტაქსურად ინტერფეისები კლასების ანალოგიურად გამოიყურება, მაგრამ არ შეიცავს ეგზემპლარების ცვლადებს, ხოლო მათი მეთოდების გამოცხადება არ შეიცავს მეთოდების ტანს. პრაქტიკულად ეს ნიშნავს, რომ შესაძლებელია გამოცხადდეს ინტერფეისები, რომლებიც არ ახდენენ რეალიზაციის მიმართ რაიმე დაშვებას. როგორც კი ინტერფეისი გამოცხადდება, მისი რეალიზაცია შესაძლებელია მოახდინოს ნებისმიერი რაოდენობის კლასმა. გარდა ამისა, ერთ კლასს შეუძლია ნებისმიერი რაოდენობის ინტერფეისის რეალიზაცია.

ინტერფეისის რეალიზაციისათვის, კლასმა უნდა მოახდინოს ყველა იმ მეთოდის რეალიზაცია, რომელიც აღწერილია ინტერფეისში. თუმცა, თითოეულმა კლასმა შესაძლებელია განსხვავებულად მოახდინოს ინტერფეისის რეალიზება. საკვანძო სიტყვა interface საშუალებას იძლევა განხორციელდეს პოლიმორფიზმის ძირითადი იდეა: „ერთი ინტერფეისი, რამდენიმე მეთოდი“ .

ინტერფეისების საშუალებით შესაძლებელია შესრულების დროს მეთოდების დინამიკურად დანიშვნა. ჩვეულებრივ, ერთი კლასიდან მეორე კლასის მეთოდის გამოძახება რომ განხორციელდეს, კომპილაციის დროს ორივე ეს კლასი უნდა არსებობდეს, რათა კომპილატორმა შეძლოს მეთოდების სიგნატურების შემოწმება.

ინტერფეისის აღწერა

ინტერფეისის აღწერა კლასის აღწერის მსგავსია. მისი ზოგადი ფორმა ასეთია:

<წვდომა> interface <სახელი> {

<დაბრუნების ტიპი> <მეთოდის სახელი1>(<პარამეტრების სია>);

<დაბრუნების ტიპი> <მეთოდის სახელი2>(<პარამეტრების სია>);

// .....

<დაბრუნების ტიპი> <მეთოდის სახელიN>(<პარამეტრების სია>);

<ტიპი> <ფინალური ცვლადის სახელი1> = <მნიშვნელობა>;

<ტიპი> <ფინალური ცვლადის სახელი2> = <მნიშვნელობა>;

// .....

<ტიპი> <ფინალური ცვლადის სახელიM> = <მნიშვნელობა>;

}

თუ აღწერაში არაა მითითებული <წვდომის> რაიმე მოდიფიკატორი, იგულისხმება სტანდარტული წვდომა და ინტერფეისი წვდომადია მხოლოდ იმავე პაკეტის წევრებისათვის, რომელშიც ისაა გამოცხადებული. თუ ინტერფეისი გამოცხადებულია როგორც public, ის შეიძლება გამოყენებული იქნას ნებისმიერი სხვა კოდის მიერ. ამ შემთხვევაში ინტერფეისი ფაილში უნდა იყოს ერთადერთი ყველასათვის წვდომადი ინტერფეისი, ხოლო მისი სახელი უნდა ემთხვეოდეს ფაილის სახელს.

<სახელი> ესაა ინტერფეისის სახელი. იგი შეიძლება იყოს ნებისმიერი დასაშვები იდენტიფიკატორი. განსაკუთრებული ყურადღება მიაქციეთ, რომ მეთოდების გამოცხადებას არ აქვს ტანი. მათი აღწერა მთავრდება პარამეტრების სიით, რომლის ბოლოში ზის წერტილმძიმე. ფაქტიურად ისინი არიან აბსტრაქტული მეთოდები. ინტერფეისში აღწერილ არც ერთ მეთოდს არ შეიძლება ჰქონდეს რაიმე რეალიზაცია. ყოველი კლასი, რომელიც მიუთითებს ინტერფეისს ვალდებულია მოახდინოს ინტერფეისში მითითებული ყველა მეთოდის რეალიზაცია.

თუ ინტერფეისში გამოცხადებულია ცვლადები, იგულისხმება (სტანდარტულად), რომ ისინი აღწერილია, როგორც final და static, ანუ კლასი რომელიც რეალიზაციას ახდენს, მათ მნიშვნელობას ვერ შეცვლის. ყველა მეთოდი და ცვლადი არაცხადად გამოცხადებულია როგორც public.

ქვემოთ ნაჩვენებია ინტერფეისის გამოცხადება. მასში აღწერილია მარტივი ინტერფეისი, რომელიც შეიცავს ერთ მეთოდს callback()-ს, რომელიც იღებს ერთ მთელრიცხვა პარამეტრს:

```
interface Callback {
   void callback (int param) ;
}
```

ინტერფეისების რეალიზაცია

ინტერფეისის განსაზღვრის შემდეგ შესაძლებელია ის რეალიზებული იქნას ერთი ან მეტი კლასის მიერ. ინტერფეისის რეალიზებისათვის საჭიროა კლასის აღწერაში მიეთითოს

implements კონსტრუქცია, ხოლო შემდეგ შეიქმნას ყველა ის მეთოდი, რომელიც აღწერილია ინტერფეისში. კლასის ზოგადი ფორმა, რომელსაც მითითებული აქვს implements ასეთი სახე აქვს:

ლისტინგი 30.

```
class Client implements Callback {
   // რეალიზდება Callback ინტერფეისი
   public void callback (int p) {
       System.out.println("მეთოდი callback,
       გამოძახებულია " +"პარამეტრით " + p);
   }
}
```

ყურადღება მიაქციეთ, რომ callback() მეთოდი გამოცხადებულია public წვდომის მოდიფიკატორით.

ინტერფეისის მეთოდის რეალიზაციისას იგი ყოველთვის გამოცხადებული უნდა იყოს როგორც public.

დასაშვებია და ხშირად გამოიყენება, როდესაც კლასი რომელიც ინტერფეისის რეალიზაციას ახდენს, დამატებით ასევე აცხადებს საკუთარ წევრებს. მაგალითად:

ლისტინგი 31.

```
class Client implements Callback {
   // რეალიზდება Callback ინტერფეისი
   public void callback(int p) {
      System.out.println("მეთოდი callback,
      გამოძახებულია " + "პარამეტრით " + p);
   }
   void nonIfaceMeth() {
```

```
        System.out.println("კლსებს, რომლებიც
        ინტერფეისის რეალიზებას ახდენენ, შეიძლება სხვა
        წევრებიც ჰქონდეთ");
    }
}
```

რეალიზაციებზე მიმართვა ინტერფეისული მიმთითებელით

ობიექტზე მიმთითებელი ცვლადი შეიძლება გამოვაცხადოთ არა მარტო, როგორც რომელიმე კლასის ტიპი, არამედ როგორც ინტერფეისული ტიპიც. ასეთი მიმთითებელით (ცვლადით) შესაძლებელია დავუკავშირდეთ ნებისმიერი კლასის ნებისმიერ ეგზემპლარს, ოღონდ ეს კლასი უნდა ახდენდეს გამოცხადებული ინტერფეისის რეალიზაციას. მიმთითებლის შედეგად გამოძახებული მეთოდის საჭირო ვერსიის არჩევა მოხდება ინტერფეისის იმ კონკრეტული რეალიზაციის მიხედვით, რომელსაც უკავშირდება ცვლადი. შესასრულებელი მეთოდის არჩევა ხდება დინამიკურად, შესრულების დროს, ამიტომ შესაძლებელია კლასები შეიქმნას უფრო მოგვიანებით, ვიდრე ის კოდი, რომელიც გამოიძახებს კლასის მეთოდებს. კოდის დისპეტჩერიზაცია შეიძლება მოხდეს ინტერფეისის საშუალებით ისე, რომ არ იყოს აუცილებელი რაიმე ცნობები "გამომძახებელი" შესახებ. ეს პროცესი ანალოგიურია სუპერკლასის ტიპის მიმთითებლის გამოყენებისა ქვეკლასის ტიპის ობიექტზე, რომელიც ჩვენ ადრე განვიხილეთ.

ვინაიდან Java სისტემაში შესრულების დროს მეთოდების დინამიკური ძებნა დაკავშირებულია მნიშვნელოვან დამატებით რესურსებთან, ვიდრე ჩვეულებრივი მეთოდის

გამოძახება, ამიტომ იმ პროგრამაში სადაც მნიშვნელოვანია მწარმოებლობა, ინტერფეისების გამოყენება უნდა მოხდეს მხოლოდ მაშინ, როცა ეს აუცილებელია.

შემდეგ მაგალითში callback() მეთოდი გამოიძახება ინტერფეისული ტიპის ცვლადისაგან :

ლისტინგი 32.

```
class Testlface {
   public static void main(String args[]) {
      Callback c = new Client();
      c.callback(42);
   }
}
```

ამ პროგრამის შედეგად გამოვა:

მეთოდი callback, გამოძახებულია პარამეტრით 42

ყურადღება მიაქციეთ, რომ მართალია ცვლადი c გამოცხადებულია როგორც Callback ინტერფეისის ტიპი, მას მიენიჭა Client კლასის ტიპის ეგზემპლარი. მართალია ცვლადი c შეიძლება გამოყენებული იქნას callback მეთოდის გამოსაძახებელად, მას არა აქვს მიმართვის უფლება Client კლასის სხვა წევრებზე. ინტერფეისის ტიპის მიმთითებელ ცვლადისათვის ცნობილია, მხოლოდ ის მეთოდები, რომლებიც აღწერილია ინტერფეისში. ამრიგად, c ცვლადი არ შეიძლება გამოყენებული იქნას nonIfaceMeth() მეთოდის გამოსაძახებელად, ვინაიდან იგი გამოცხადებულია Client კლასის მიერ და არა Callback ინტერფეისის მიერ.

მართალია განხილული მაგალითი ფორმალურად აჩვენებს, როგორ ახდენს ინტერფეისის ტიპის ცვლადი რეალიზებულ

ობიექტთან დაკავშირებას, მაგრამ არაა დემონსტრირებული ასეთი დაკავშირების პოლიმორფული შესაძლებლობები. ასეთი გამოყენების დემონსტრირებისათვის ჯერ შევქმნათ Callback ინტერფეისის მეორე რეალიზაცია:

ლისტინგი 33. Callback ინტერფეისის ახალი რეალიზაცია:

```
class AnotherClient implements Callback {
   // Callback ინტერფეისის რეალიზაცია
   public void callback(int p) {
      System.out.println("callback-ის ახალი ვერსია");
      System.out.println("p კვადრატში ტოლია " +
      (p * p));
   }
}
```

ახლა ვნახოთ შემდეგი კლასის მუშაობის შედეგი:

ლისტინგი 34.

```
class TestIface2 {
   public static void main(String args[]) {
      Callback c = new Client();
      AnotherClient ob = new AnotherClient();
      c.callback(42);
      c = ob; /* ახლა დაკავშირებული AnotherClient
      ობიექტთან*/
      c.callback(42);
   }
}
```

ამ პროგრამის შესრულების შედეგი:

მეთოდი callback, გამოძახებულია პარამეტრით 42
callback-ის ახალი ვერსია
p კვადრატში ტოლია 1764

როგორც ხედავთ, callback()-ის გამოძახებული ვერსია განისაზღვრება იმ ობიექტის ტიპით, რომელსაც უკავშირდება c

ცვლადი შესრულების დროს. ნაჩვენები მაგალითი ძალიან მარტივია, ამიტომ შემდეგში ჩვენ განვიხილავთ უფრო რეალისტურ მაგალითს.

ნაწილობრივი რეალიზაცია

თუ კლასი მოიცავს ინტერფეისს, მაგრამ არ ახდენს ინტერფეისში გამოცხადებული ყველა მეთოდის სრულ რეალიზაციას, იგი უნდა იქნას გამოცხადებული abstract (აბსტრაქტული) მოდიფიკატორით. მაგალითი:

ლისტინგი 35.

```
abstract class Incomplete implements Callback {
   int a, b;
   void show() {
      System.out.println(a + " " + b);
   }
   // ...
}
```

ამ მაგალითში Incomplete კლასი არ ახდენს callback() მეთოდის რეალიზაციას, ამიტომ ის უნდა გამოცხადდეს აბსტრაქტულად. ნებისმიერი კლასი რომელიც იყენებს Incomplete კლასს სუპერკლასად, უნდა ახდენდეს callback() მეთოდის რეალიზაციას ან ისიც უნდა იყოს გამოცხადებული როგორც abstract.

ინტერფეისების გამოყენება

ინტერფეისების შესაძლებლობების უკეთ გაცნობისათვის განვიხილოთ უფრო რეალისტური მაგალითი. წინა მასალაში ჩვენ განვიხილეთ Stack კლასი, რომელიც ქმნიდა

ფიქსირებული სიგრძის მარტივ სტეკს. არსებობს სტეკის რეალიზების სხვა ხერხებიც. მაგალითად, სტეკი შეიძლება იყოს „ზრდადი“ ზომის, ან ინახებოდეს მასივში, დაკავშირებულ სიაში, ბინარულ ხეში და ა.შ.

სტეკის რეალიზაციის მიუხედავად მისი ინტერფეისი უცვლელი რჩება. ანუ, მეთოდები push() და pop() რეალიზაციის ნიუანსებისაგან დამოუკიდებლად განსაზღვრავენ სტეკთან ინტერფეისს. ვინაიდან სტეკთან ინტერფეისი განცალკევებულია მისი რეალიზაციისაგან, შესაძლებელია ცალკე განისაზღვროს სტეკთან მუშაობის ინტერფეისი, ხოლო რეალიზაციაში განხორციელდეს სპეციფიკური თავისებურებების. განვიხილოთ ორი მაგალითი.

ჯერ შევქმნათ ინტერფეისი `IntStack`, რომელიც განსაზღვრავს მთელრიცხვა სტეკს და იგი მოვათავსოთ ფაილში InStack.java. ეს ინტერფეისი გამოიყენოს ყველა რეალიზაციამ.

ლისტინგი 36. მთელრიცხვა სტეკის ინტერფეისის განსაზღვრა:

```
interface IntStack {
   void push(int item); // ელემენტის შენახვა
   int pop(); // ელემენტის ამოღება
}
```

შემდეგი პროგრამა ქმნის FixedStack კლასს, რომელიც რეალიზებას უკეთებს მთელრიცხვა, ფიქსირებული სიგრძის სტეკს:

ლისტინგი 37. IntStack რეალიზაცია, რომელიც შესანახ არედ იყენებს ფიქსირებული ზომის მეხსიერებას:

```
class FixedStack implements IntStack {
   private int stck[];
   private int tos;
   // სტეკის რეზერვირება და ინიციალიზაცია
   FixedStack(int size) {
      stck = new int[size];
      tos = -1;
   }
   // სტეკში ელემენტის ჩაგდება
   public void push(int item) {
      if (tos == stck.length - 1)
         System.out.println("სტეკი სავსეა!");
      else
         stck[++tos] = item;
   }
   // სტეკიდან ელემენტის ამოღება
   public int pop() {
      if (tos < 0) {
         System.out.println("სტეკი ცარიელია!");
         return 0;
      } else
         return stck[tos--];
   }
}
class IFTest {
   public static void main(String args[]) {
      FixedStack mystack1 = new FixedStack(5);
      FixedStack mystack2 = new FixedStack(8);
      // რიცხვების სტეკში ჩაყრა
      for (int i = 0; i < 5; i++)
         mystack1.push(i);
      for (int i = 0; i < 8; i++)
         mystack2.push(i);
      // სტეკიდან რიცხვების ამოღება
      System.out.println("mystack1- ში: ");
      for (int i = 0; i < 5; i++)
         System.out.println(mystack1.pop());
      System.out.println("mystack2-ში: ");
      for (int i = 0; i < 8; i++)
         System.out.println(mystack2.pop());
```

```
    }
}
```

ახლა შევქმნათ IntStack-ის კიდევ ერთი რეალიზაცია, რომელიც იგივე ინტერფეისს იყენებს, მაგრა ქმნის დინამიკურ სტეკს. ამ რეალიზაციაში ყოველი სტეკი იქმნება საწყისი სიგრძით. ამ საწყისი სიგრძის გადაჭარბების შემთხვევაში სტეკის სიგრძე იზრდება. ყოველთვის, როდესაც საჭიროა სტეკის სიგრძის გაზრდა, მისი სიგრძე ორმაგდება.

ლისტინგი 38. ''ზრდადი'' სტეკის ორგანიზება:

```
class DynStack implements IntStack {
  private int stck[];
  private int tos;
  // სტეკის რეზერვირება და ინიციალიზაცია
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }
  // სტეკში ელემენტის ჩაგდება
  public void push(int item) {
    /* თუ სტეკი სავსეა, ორმაგი სიგრძის სტეკის
    რეზერვირება*/
    if (tos == stck.length - 1) {
      // ზომის გაორმაგება
      int temp[] = new int[stck.length * 2];
      for (int i = 0; i < stck.length; i++)
        temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
    } else
      stck[++tos] = item;
  }
  // სტეკიდან ელემენტის ამოღება
  public int pop() {
    if (tos < 0) {
      System.out.println("სტეკი ცარიელია!");
```

```
        return 0;
      } else
        return stck[tos--];
    }
}
class IFTest2 {
  public static void main(String args[]) {
    DynStack mystack1 = new DynStack(5);
    DynStack mystack2 = new DynStack(8);
    // ეს ციკლები იწვევს სტეკების ზომის ზრდას
    for (int i = 0; i < 12; i++)
      mystack1.push(i);
    for (int i = 0; i < 20; i++)
      mystack2.push(i);
    System.out.println("mystack1-ში:");
    for (int i = 0; i < 12; i++)
      System.out.println(mystack1.pop());
    System.out.println("mystack2-ში:");
    for (int i = 0; i < 20; i++)
      System.out.println(mystack2.pop());
  }
}
```

შემდეგი კლასი იყენებს ორივე რეალიზაციას FixedStack და DynStack. ეს ხდება ინტერფეისზე მიმართვით. ანუ, push() და pop() მეთოდებზე მიმართვის ნებართვა წყდება შესრულების დროს და არა კომპილაციის დროს.

ლისტინგი 39. ინტერფეისული ცვლადის შექმნა და სტეკებთან ამ ცვლადებით მიმართვა

```
class IFTest3 {
  public static void main(String args[]) {
    IntStack mystack; // ინტერფეისული ცვლადის
    გამოცხადება
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);
    mystack = ds; // დინამიკურ სტეკთან დაკავშირება
    // სტეკში რიცხვების ჩაყრა
```

```
      for (int i = 0; i < 12; i++)
        mystack.push(i);
      mystack = fs; // ფიქსირებულ სტეკთან დაკავშირება
      for (int i = 0; i < 8; i++)
        mystack.push(i);
      mystack = ds;
      System.out.println("დინამიკური სტეკის
      მნიშვნელობა:");
      for (int i = 0; i < 12; i++)
        System.out.println(mystack.pop());
      mystack = fs;
      System.out.println("ფიქსირებული სტეკის
      მნიშვნელობა:");
      for (int i = 0; i < 8; i++)
        System.out.println(mystack.pop());
    }
}
```

ამ პროგრამაში mystack არის ინტერფეის IntStack-ის ტიპის ცვლადი. ამრიგად, როდესაც იგი უკავშირდება ds ცვლადს, პროგრამა იყენებს push() და pop() მეთოდების იმ ვერსიას, რომლებიც განსაზღვრულია DynStack რეალიზაციაში. როდესაც იგი უკავშირდება fs-ს, მაშინ პროგრამა იყენებს push() და pop() მეთოდების ვერსიას, რომელიც FixedStack-შია რეალიზებული. ინტერფეისის რამდენიმე რეალიზაციასთან დაკავშირება ინტერფეისული ტიპის ცვლადის საშუალებით ესაა პოლიმორფიზმის მხარდაჭერის მძლავრი შესაძლებლობა.

ცვლადები ინტერფეისებში

ინტერფეისები შეიძლება გამოყენებული იქნას რამდენიმე კლასში ერთობლივად გამოყენებული კონსტანტების იმპორტირებისათვის. ამისათვის საჭიროა ეს კონსტანტები, საჭირო

ინიციალიზაციით გამოცხადდეს ინტერფეისში, ხოლო შემდეგ საჭირო კლასებმა მოახდინონ ამ ინტერფეისის იმპლემენტირება. თუ ინტერფეისი არ შეიცავს მეთოდებს, ნებისმიერი კლასი, რომელიც ასეთ ინტერფეისს მიიერთებს, სინამდვილეში არაფრის რეალიზებას არ ახდენს. განვიხილოთ ეს ტექნოლოგია შემდეგ მაგალითში:

ლისტინგი 40.

```
import java.util.Random;

interface SharedConstants {
   int NO = 0;
   int YES = 1;
   int MAYBE = 2;
   int LATER = 3;
   int SOON = 4;
   int NEVER = 5;
}
class Question implements SharedConstants {
   Random rand = new Random();
   int ask() {
      int prob = (int) (100 * rand.nextDouble());
      if (prob < 30)
         return NO; // 30%
      else if (prob < 60)
         return YES; // 30%
      else if (prob < 75)
         return LATER; // 15%
      else if (prob < 98)
         return SOON; // 13%
      else
         return NEVER; // 2%
   }
}
class AskMe implements SharedConstants {
   static void answer(int result) {
      switch (result) {
      case NO:
```

```
        System.out.println("არა");
        break;
      case YES:
        System.out.println("კი");
        break;
      case MAYBE:
        System.out.println("შესაძლებელია");
        break;
      case LATER:
        System.out.println("მოგვიანებით");
        break;
      case SOON:
        System.out.println("მალე");
        break;
      case NEVER:
        System.out.println("არასოდეს");
        break;
      }
    }
    public static void main(String args[]) {
      Question q = new Question();
      answer(q.ask());
      answer(q.ask());
      answer(q.ask());
      answer(q.ask());
    }
}
```

ყურადღება მიაქციეთ, რომ ამ პროგრამაში გამოყენებული იქნა Java-ს ერთ-ერთი სტანდარტული კლასი Random. ეს კლასი ახდენს ფსევდოშემთხვევითი რიცხვების გამომუშავებას (გენერირებას). იგი შეიცავს რამდენიმე მეთოდს, რომლებიც საშუალებას იძლევა მივიღოთ შემთხვევითი რიცხვები პროგრამისათვის საჭირო ფორმაში. ამ მაგალითში გამოყენებულია nextDouble() მეთოდი, რომელიც აბრუნებს შემთხვევით რიცხვებს 0.0-დან 1.0-მდე დიაპაზონში.

ამ მაგალითში ორი კლასი Question და AskMe რეალიზებას უკეთებენ SharedConstants ინტერფეისს. ამ ინტერფეისში განსაზღვრულია კონსტანტები NO (არა), YES (კი), MAYBE (შესაძლებელია), SOON (მალე), LATER (მოგვიანებით) და NEVER (არასოდეს). თითოეული კლასის შიგნით კოდი მიმართავს ამ კონსტანტებს ისე, თითქოს ისინი თითოეული ამ კლასის შიგნით ყოფილიყოს აღწერილი ან თითქოს მემკვიდრეობით გადმოეცა. ქვემოთ ნაჩვენებია ამ პროგრამის შესრულების შედეგი. ყურადღება მიაქციეთ, რომ ყოველი გაშვების შედეგად სხვადასხვა შედეგი გამოვა:

მოგვიანებით
მალე
არა
კი

ინტერფეისების მემკვიდრეობითობა

კლასების მსგავსად, საკვანძო სიტყვა extends-ის მითითებით შესაძლებელია ინტერფეისებს შორის მემკვიდრეობითობა დამყარდეს. როდესაც კლასი ახდენს ისეთი ინტერფეისის რეალიზაციას, რომელიც სხვა ინტერფეისის მემკვიდრეა, მან უნდა მოახდინოს ყველა იმ მეთოდის რეალიზაცია, რომლებიც გამოცხადებულია ინტერფეისების მემკვიდრეობით ჯაჭვში. მაგალითი:

ლისტინგი 41. ერთი ინტერფეისი შეიძლება მეორის მემკვიდრე იყოს:

```
interface A {
```

```
   void meth1();
   void meth2();
}
/* ახლა B შეიცავს meth1() და meth2() და ემატება
meth3()*/
interface B extends A {
   void meth3();
}

/* ამ კლასმა A და B ინტერფეისების ყველა მეთოდის
რეალიზაცია უნდა მოახდინოს*/
class MyClass implements B {
   public void meth1() {
      System.out.println("meth1()-ის რეალიზაცია");
   }
   public void meth2() {
      System.out.println("meth2()-ის რეალიზაცია ");
   }
   public void meth3() {
      System.out.println("meth3()-ის რეალიზაცია  ");
   }
}
class IFExtend {
   public static void main(String arg[]) {
      MyClass ob = new MyClass();
      ob.meth1();
      ob.meth2();
      ob.meth3();
   }
}
```

ექსპერიმენტის სახით თუ შევეცდებით meth1() მეთოდის რეალიზაციის წაშლას, ეს გამოიწვევს სინტაქსურ შეცდომას. როგორც ვთქვით, ნებისმიერი კლასი, რომელიც ინტერფეისის რეალიზაციას ახდენს, ამ ინტერფეისში გამოცხადებული ყველა მეთოდის რეალიზაცია უნდა მოახდინოს, მათ შორის მემკვიდრეობით გადმოცემული მეთოდებისაც.

პაკეტები და ინტერფეისები წარმოადგენენ Java-ს დაპროგრამების გარემოს უმნიშვნელოვანეს შემადგენელ მექანიზმს. Java-ზე დაწერილი ყველა რეალური პროგრამა ინახება პაკეტებში. როგორც წესი, ბევრი მათგანი ახდენს რომელიმე ინტერფეისის იმპლემენტირებას (რეალიზაციას).

განსაკუთრებული სიტუაციების დამუშავება

ალგორითმულ ენა Java-ში ჩადებულია განსაკუთრებული სიტუაციების (გამონაკლისების) დამუშავების მექანიზმი. **განსაკუთრებული სიტუაცია (გამონაკლისი)** ესაა არანორმალური მდგომარეობა, რომელიც წარმოიშვება პროგრამული კოდის შესრულების მიმდინარეობისას. ანუ, გამონაკლისი ესაა შესრულების დროს წარმოქმნილი შეცდომა. იმ ალგორითმულ ენებში, რომლებიც არ უზრუნველყოფენ განსაკუთრებული სიტუაციების დამუშავებას, შეცდომები უნდა შემოწმდეს და დამუშავდეს „ხელით“. ჩვეულებრივ ეს ხდება შეცდომის კოდების ანალიზით ან რაიმე მსგავსი მეთოდით. ეს გზა საკმაოდ შრომატევადი და პრობლემატურია. განსაკუთრებული სიტუაციების დამუშავება საშუალებას იძლევა თავი ავარიდოთ ასეთ პრობლემებს და გარდა ამისა, შესრულების დროს წარმოქმნილი შეცდომების დამუშავება გადავიტანოთ ობიექტ-ორიენტირებულ სამყაროში.

Java-ში განსაკუთრებული სიტუაცია ესაა ობიექტი, რომელიც აღწერს განსაკუთრებულ (როგორც წესი შეცდომიან) სიტუაციას, რომელიც წარმოიქმნება პროგრამული კოდის რაღაც ნაწილში. როდესაც ასეთი განსაკუთრებული სიტუ-

აცია წარმოიქმნება (აღიძვრება), შეიქმნება იმ სიტუაციის ამსახველი ობიექტი, რომელიც შეიქმნა შეცდომის გამომწვევ მეთოდში. ამ მეთოდმა შესაძლებელია ეს შეცდომა თვითონ დაამუშაოს ან გაატაროს იგი, ორივე შემთხვევაში, რომელიღაც წერტილში ხდება განსაკუთრებული სიტუაციის აღმოჩენა (გამოჭერა) და დამუშავება. განსაკუთრებული სიტუაცია შეიძლება აღიძრას ჯავას შემსრულებელი გარემოს მიერ ან ხელოვნურად იქნას გენერირებული პროგრამული კოდის მიერ. გამონაკლისები, რომელსაც Java წარმოშობს, დაკავშირებულია ფუნდამენტურ შეცდომებთან, რომლებიც არღვევენ ენის წესებს ან სისტემის შეზღუდვებს. ხელოვნურად აღძრული გამონაკლისები ჩვეულებრივ გამოიყენება იმისათვის, რომ მცდარი სიტუაციები ეცნობოს მეთოდის გამომძახებელს.

განსაკუთრებული სიტუაციების დამუშავება Java-ში იმართება ხუთი საკვანძო სიტყვის დახმარებით: try, catch, throw, throws და finally. მოკლედ რომ ჩამოვაყალიბოთ ისინი მუშაობენ ასე: ოპერატორები, რომელთა თვალყურის დევნება გვინდა მოხდა თუ არა აქ განსაკუთრებული სიტუაცია, ჩაისმება ბლოკში. თუ ბლოკში აღიძრა გამონაკლისი იგი გამოიტყორცნება (გამოვარდება). catch ოპერატორის გამოყენებით შესაძლებელია აღძრული გამონაკლისის აღმოჩენა („გამოჭერა“) და შემდეგ მისი დამუშავება რაიმე გააზრებული ხერხით. სისტემის მიერ წარმოქმნილი გამონაკლისები ავტომატურად გამოიტყორცნება Java-ს შემსრულებელი გარემოს მიერ. გამონაკლისის ხელოვნურად წარმოქმნისათვის გამოიყენება საკვანძო სიტყვა throw. მეთოდის შიგნით წარმოქმნილი ყოველი გამონაკლისი მის

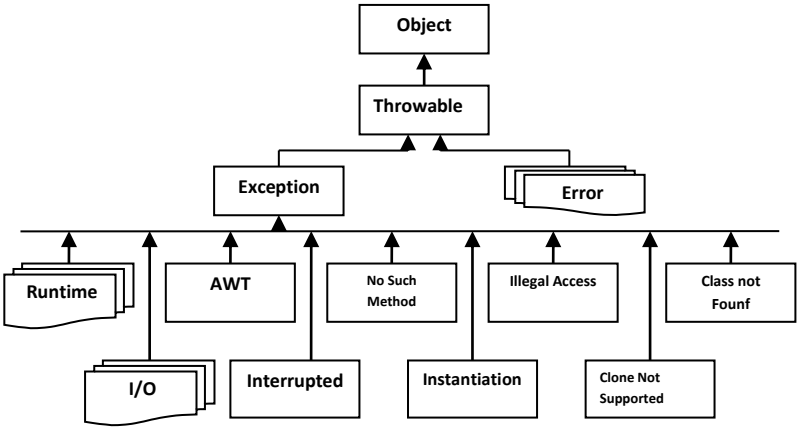
ინტერფეისში სპეციფირებული უნდა იქნას საკვანძო სიტყვით throws. ნებისმიერი კოდი, რომელიც აუცილებლად უნდა შესრულდეს try ბლოკის დამთავრების შემდეგ, უნდა მოთავსდეს finally ბლოკში. ქვემოთ ნაჩვენებია განსაკუთრებული სიტუაციის დამუშავების ბლოკის ზოგადი ფორმა:

```
try {
    // შეცდომებზე დაკვირვების ბლოკი
}
catch (<გამონაკლისის ტიპი 1> exOb) {
    // ExceptionType1 ტიპის გამონაკლისის დამუშავება
}
catch (<გამონაკლისის ტიპი 2>  exOb) {
    // ExceptionType2 ტიპის გამონაკლისის დამუშავება
}
/ / ...
finally {
    // try ბლოკის შემდეგ შესასრულებელი ბლოკი
}
```

<გამონაკლისის ტიპი> ესაა წარმოქმნილი განსაკუთრებული სიტუაციის ტიპი.

გამონაკლისი ტიპები

გამონაკლისების ყველა შესაძლო ტიპი წარმოადგენს Trowable სისტემაში ჩაშენებული (ჩაშენებულია ჯავას კლასების სტანდარტულ ბიბლიოთეკაში) კლასის ქვეკლასს. ანუ, Trowable კლასი განლაგებულია გამონაკლისების კლასების იერარქიის თავში.



სურ. 1 გამონაკლისების კლასების იერარქია

უშუალოდ Throwable კლასის ქვემოთ იმყოფება ორი ქვეკლასი, რომლებიც გამონაკლისებს ყოფენ ორ შტოდ. ერთი შტოს თავშია Exception კლასი. ეს კლასი გამოიყენება იმ განსაკუთრებული სიტუაციებისათვის, რომლებიც მომხმარებლის პროგრამამ უნდა გამოიჭიროს და დაამუშაოს. ამ კლასიდან შესაძლებელია ისეთი მემკვიდრე ქვეკლასების შექმნა, რომლებიც ჩვენს მიერ განსაზღვრული გამონაკლისების ტიპების შესაბამისი იქნება. Exception კლასს აქვს მნიშვნელოვანი ქვეკლასი რომლის სახელია RuntimeException. ამ ტიპის გამონაკლისები ავტომატურად განისაზღვრება იმ პროგრამებისათვის, რომლებსაც ჩვენ ვწერთ და შეიცავს ისეთ სიტუაციებს, როგორიცაა 0-ზე გაყოფა, მასივის ინდექსის საზღვრებს გარეთ გასვლა და სხვა.

მეორე შტო იწყება კლასიდან Error, რომელიც განსაზღვრავს გამონაკლისებს, რომელთა გამოძახება მოსალოდნელი არაა პროგრამის ნორმალური შესრულების დროს. Error-ის ტიპის გამონაკლისები Java-ს შემსრულებელი გარემოსაგან გამოიყენება იმ შეცდომების აღწერისათვის, რომლებიც თვით გარემოში წარმოიშვება. ამ ტიპის შეცდომის მაგალითია სტეკის გადავსება. Error-ის ტიპის შეცდომებს ჩვენ არ განვიხილავთ, ვინაიდან ისინი ჩვეულებრივ წარმოიქმნება კატასტროფული მტყუნების შედეგად და მათი დამუშავება ჩვენ პროგრამას არ შეუძლია.

დაუმუშავებელი გამონაკლისები

განვიხილოთ შემთხვევა, როდესაც ჩვენი პროგრამა არ ახდენს წარმოქმნილი განსაკუთრებული სიტუაციის დამუშავებას. განვიხილოთ პატარა პროგრამა, სადაც ხდება 0-ზე გაყოფის შეცდომის წარმოქმნა:

ლისტინგი 42.

```
class Exc0 {
   public static void main(String args[]) {
      int d = 0;
      int a = 42 / d;
   }
}
```

როდესაც Java სისტემა აღმოაჩენს შესრულების პროცესში 0-ზე გაყოფის მცდელობას, იგი შექმნის გამონაკლისის ახალ ობიექტს და შემდეგ აღძრავს გამონაკლისს. ეს იწვევს Exc0-ის შესრულებას, ვინაიდან როგორც კი გამონაკლისი აღიძვრება მას გამოიჭერს გამონაკლისების დამმუშავებელი, ხოლო მან ამ გამონაკლისს მაშინვე რაღაც უნდა უყოს. ამ

მაგალითში გამონაკლისების ჩვენი დამმუშავებელი არ გამოგვიყენებია, ამიტომ გამონაკლისი გამოიჭირება სტან-დარტული (ნაგულისხმევი) გამონაკლისების დამმუშავებლისაგან, რომლითაც აღჭურვილია Java-ს შემსრულებელი გარემო. ნებისმიერი გამონაკლისი, რომლის გამოჭერა თქვენი პროგრამით არ ხდება, საბოლოო ჯამში ამ სტანდარტული დამმუშავებლის მიერ გამოიჭირება. სტანდარტული დამმუშავებელი გამოიტანს სტრიქონს, რომელიც გამონაკლისს აღწერს, ბეჭდავს გამონაკლისის წარმოქმნის წერტილიდან დაწყებული სტეკის ტრასირებას და წყვეტს პროგრამას.

ქვემოთ ნაჩვენებია Exc0 პროგრამის კოდის მიერ გენერირებული გამონაკლისი მაგალითი:

java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)

მიაქციეთ ყურადღება, რომ კლასის სახელი Exc0, მეთოდის სახელი main, ფაილის სახელი Exc0.java სტრიქონის ნომერი 4 შესულია სტეკის ტრასირებაში. ასევე ყურადღება მისაქცევია, რომ აღძრული გამონაკლისი Exception კლასის ქვეკლასია და მისი სახელია ArithmeticException, რომელიც უფრო ზუსტად აღწერს აღძრული გამონაკლისის ტიპს.

სტეკის ტრასირება ყოველთვის აჩვენებს იმ მეთოდების გამოძახების მიმდევრობას, რამაც გამოიწვია შეცდომა. მაგალითად:

ლისტინგი 43.

```
class Exc1 {
   static void subroutine() {
      int d = 0;
      int a = 10 / d;
   }
   public static void main(String args[]) {
      Exc1.subroutine();
   }
}
```

გამონაკლისის სტანდარტული დამმუშავებლის სტეკის ტრასირების შედეგი შეიცავს გამოძახებების მთელ მიმდევრობას:

java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)

სტეკის ტრასირება საკმაოდ მოსახერხებელია პროგრამის გამართვისათვის, ვინაიდან აჩვენებს შეცდომის გამომწვევი გამოძახებების მთელ მიმდევრობას.

try და catch-ის გამოყენება

მართალია გამონაკლისების სტანდარტული დამმუშავებელი გამართვისათვის მოსახერხებელია, მაგრამ ჩვენ შეიძლება მოვახდინოთ შეცდომის ჩვენებური დამუშავება. ეს ორ მნიშვნელოვან უპირატესობას იძლევა: პირველი - საშუალება გვეძლევა გამოვასწოროთ შეცდომა; მეორე - აღარ მოხდება პროგრამის შესრულების ავტომატური წყვეტა. მართლაც ჩვენს მიერ დაწერილი პროგრამის მომხმარებელი ძალიან უკმაყოფილო იქნება თუ ყოველი შეცდომის წარმოქმნის შედეგად პროგრამა გაჩერდება და დაიწყებს შეცდომების ტრასირების ბეჭდვას.

შეცდომების სტანდარტული დამუშავების თავიდან ასაცილებლად საჭიროა დასაკვირვებელი პროგრამის კოდი მოვათავსოთ try ბლოკის შიგნით. უშუალოდ try ბლოკის შემდეგ უნდა ჩავრთოთ catch ბლოკი, სადაც მოხდება გამოსაჭერი გამონაკლისების ტიპების სპეციფირება. საილუსტრაციოდ განვიხილოთ მაგალითი:

ლისტინგი 44.

```
class Exc2 {
   public static void main(String args[]) {
      int d, a;
      try { // კოდის მონიტორინგის ბლოკი
         d = 0;
         a = 42 / d;
         System.out.println("ეს სტრიქონი არ
         დაიბეჭდება");
       } catch (ArithmeticException e) {/* 0-ზე
       გაყოფის გამოჭერა */
         System.out.println("0-ზე გაყოფა");
      }
      System.out.println("catch ოპერატორის შემდეგ");
   }
}
```

პროგრამის შედეგი:

```
0-ზე გაყოფა
catch ოპერატორის შემდეგ
```

ყურადღება მიაქციეთ, რომ try-ს შიგნით println() გამოძახება არასდროს არ მოხდება. როგორც კი გამონაკლისი წარმოიქმნება, try ბლოკიდან მართვა გადაეცემა catch ბლოკს. ანუ სტრიქონი "ეს სტრიქონი არ დაიბეჭდება" კონსოლზე არ

გამოვა. try ბლოკის შესრულების შემდეგ, მართვა გადაეცემა try/catch კონსტრუქციის შემდეგ ოპერატორს.

try და catch ოპერატორები ქმნიან ერთიან კონსტრუქციას. catch-ის მოქმედების არე არ ვრცელდება იმ ოპერატორებზე, რომლებიც განლაგებულია try ოპერატორის წინ. catch ოპერატორს არ შეუძლია გამოიჭიროს გამონაკლისები, რომლებიც აღძრულია სხვა try ოპერატორის მიერ. ოპერატორები, რომლებიც დაცულია try ბლოკის მიერ, მოთავსებული უნდა იყვნენ ფიგურული ფრჩხილების შიგნით. არ შეიძლება try-ის გამოყენება ცალკე ერთ ოპერატორზე.

სწორად აგებული catch ოპერატორების მიზანია განსაკუთრებული სიტუაციების გადაწყვეტა და მუშაობის ისე გაგრძელება, თითქოს არავითარი შეცდომა არ მომხდარიყოს. მაგალითად, შემდეგ პროგრამაში for ციკლის თითოეულ იტერაციაზე ვღებულობთ ორ შემთხვევით რიცხვს. ერთი რიცხვი იყოფა მეორეზე, შედეგი გამოიყენება 12345-ის გასაყოფად. საბოლოო შედეგი თავსდება a-ში. თუ რომელიმე გაყოფის ოპერაცია გამოიწვევს 0-ზე გაყოფის შეცდომას - ეს შეცდომა გამოიჭირება და a-ს მნიშვნელობა გახდება 0, ხოლო პროგრამის შესრულება გაგრძელდება:

ლისტინგი 45. გამონაკლისის დამუშავება და მუშაობის გაგრძელება:

```
import java.util.Random;
class HandleError {
   public static void main(String args[]) {
      int a=0, b=0, c=0;
      Random r = new Random();
```

```
      for(int i=0; i<32000; i++) {
         try {
            b = r.nextInt();
            c = r.nextInt();
            a = 12345 / (b/c);
         } catch (ArithmeticException e) {
         System.out.println("0-ზე გაყოფა.");
         a =0; // მიენიჭოს 0 და გაგრძელდეს გამოთვლა
      }
      System.out.println("a: " + a);
   }
}
```

განსაკუთრებული სიტუაციის აღწერის ასახვა

Throwable ხელახლა განსაზღვრავს toString() მეთოდს (ეს მეთოდი როგორც გახსოვთ აღწერილია Object კლასში) ისე, რომ იგი აბრუნებს გამონაკლისის აღმწერ სტრიქონს. ამ აღწერის კონსოლზე გამოსატანად შესაძლებელია println() მეთოდის გამოყენება, რომელსაც არგუმენტად გადავცემთ გამონაკლისს. მაგალითად, წინა მაგალითის catch ბლოკი გადავწეროთ ასე:

ლისტინგი 46.

```
catch (ArithmeticException e) {
   System. out. println ("გამონაკლისი: " + e) ;
   a = 0; // მიანიჭებს 0-ს და განაგრძობს მუშაობას
}
```

თუ ამ ფრაგმენტით ჩავანაცვლებთ და პროგრამას გავუშვებთ 0-ზე გაყოფის ყოველი მცდელობა კონსოლზე გამოიტანს შემდეგ შეტყობინებას:

გამონაკლისი: java.lang.ArithmeticException: / by zero

მართალია ამ კონტექსტში ამას დიდი მნიშვნელობა არა აქვს, მაგრამ ზოგ შემთხვევაში განსაკუთრებული სიტუაციის აღწერის ასახვა სასარგებლოა, განსაკუთრებით პროგრამის გამართვის დროს.

მრავალჯერადი catch ოპერატორები

ზოგჯერ პროგრამის ერთმა ფრაგმენტმა შეიძლება გამოიწვიოს რამდენიმე განსხვავებული ტიპის გამონაკლისი. ამ სიტუაციის დასაძლევად შესაძლებელია რამდენიმე catch ოპერატორის გამოყენება, თითოეული თავისი ტიპის გამონაკლისის გამოსაჭერად. როდესაც გამონაკლისი აღიძვრება, თითოეული catch ოპერატორი მოწმდება რიგ-რიგობით და შესრულდება პირველი მათგანი, რომლის ტიპიც შეესაბამება გამონაკლისის ტიპს. მას შემდეგ რაც შესრულდება ერთერთი catch ოპერატორი, ყველა დანარჩენი გამოიტოვება და პროგრამის შესრულება გრძელდება try/catch კონსტრუქციის შემდეგი ოპერატორიდან. შემდეგ მაგალითში ხდება ორი სხვადასხვა ტიპის გამონაკლისის გამოჭერა:

ლისტინგი 47. მრავალჯერადი catch ოპერატორების დემონსტრირება:

```
class MultiCatch {
   public static void main(String args[]) {
      try {
         String s = ""; // "abc"
         int a = s.length();
         System.out.println("a = " + a);
         int b = 42 / a;
         int c[] = { 1 };
         c[42] = 99;
      } catch (ArithmeticException e) {
```

```
            System.out.println("0-ზე გაყოფა: " + e);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("მასივის ინდექსის შეცდომა:
            " + e);
        }
        System.out.println("try/catch ბლოკის შემდეგ");
    }
}
```

ეს პროგრამა გამოიწვევს 0-ზე გაყოფის გამონაკლისს, თუ სტრიქონული ტიპის ცვლადს s-ს ნულოვანი სიგრძის სტრიქონი მიენიჭება (მაგ., s="";), ვინაიდან ამ დროს a-ს მნიშვნელობა იქნება 0. თუ სტრიქონი ცარიელი არაა (ერთი ან მეტი სიმბოლოსგან შედგება), მაშინ a 0-ზე მეტი იქნება და ეს გამონაკლისი აღარ მოხდება. ამ შემთხვევაში მოხდება მეორე გამონაკლისი ArrayIndexOutOfBoundsException, ვინაიდან მთელრიცხვა c მასივის სიგრძეა 1, მაგრამ პროგრამაში მცდელობაა 42-ე ელემენტს (c[42]) მიენიჭოს 99.

თუ პროგრამას გავუშვებთ s-ის ორი სხვადასხვა მნიშვნელობით (პირველი როგორც პროგრამაში, მეორე კომენტირებულით თუ შევცვლით) კონსოლზე პირველ და მეორე შემთხვევაში ასეთი მნიშვნელობები გამოვა:

a = 0
0-ზე გაყოფა: java.lang.ArithmeticException: / by zero
try/catch ბლოკის შემდეგ

a = 3
მასივის ინდექსის შეცდომა:
java.lang.ArrayIndexOutOfBoundsException:42
try/catch ბლოკის შემდეგ

როდესაც მრავალჯერადი catch ოპერატორი გამოიყენება, მნიშვნელოვანია გვახსოვდეს, რომ გამონაკლისის ქვეკლასები წინ უსწრებდეს მის ყველა სუპერკლასს. ეს იმიტომ, რომ catch ოპერატორი, რომელიც იყენებს სუპერკლასს, გამოიჭერს ამ კლასისა და მისი ყველა ქვეკლასის ყველა გამონაკლისს. ანუ, გამონაკლისი ქვეკლასით არასოდეს არ დამუშავდება, თუ მის გამოჭერას ჩასვამთ სუპერკლასის შემდეგ. უფრო მეტიც, Java-ში კოდი, რომელიც ვერასოდეს ვერ შესრულდება, ითვლება შეცდომად. მაგალითად:

ლისტინგი 48. ამ პროგრამაში შეცდომაა !!! ქვეკლასი უნდა იჯდეს მისი სუპერკლასის წინ მრავალჯერადი catch ოპერატორების შემთხვევაში. წინააღმდეგ შემთხვევაში მიიღება მიღწევადი კოდი, რაც გამოიწვევს კომპილაციის შეცდომას.

```
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch (Exception e) {
      System.out.println("ზოგადი გამოჭერა
      Exception.");
    }
    /* ეს catch არასდროს არ გამოიჭირება, ვინაიდან
    ArithmeticException წარმოადგენს Exception-ის
    ქვეკლასს. */
    catch (ArithmeticException e) { // შეცდომა !!!
      System.out.println("ეს არასოდეს არ
      შესრულდება.");
    }
  }
}
```

თუ ამ პროგრამის კომპილირებას შევეცდებით, მივიღებთ შეცდომას, რომელიც მიუთითებს, რომ მეორე catch ოპერატორი არასოდეს არ გამოიძახება, ვინაიდან გამონაკლისი უკვე გამოჭერილია. ვინაიდან ArithmeticException წარმოადგენს Exception-ის ქვეკლასს, პირველი catch ოპერატორი დაამუშავებს Exception-ზე დაფუძნებულ ყველა შეცდომას, მათ შორის ArithmeticException-ს. ამ პრობლემის გასასწორებლად საჭიროა ადგილები შევუცვალოთ catch ოპერატორებს.

ჩალაგებული try ოპერატორები

try ოპერატორები შეიძლება იყოს ერთიმეორეში ჩალაგებული. ყოველთვის, როდესაც მართვა მოხვდება try ბლოკში, ამ განსაკუთრებული სიტუაციის კონტექსტი ჩავარდება სტეკში. თუ ჩადგმულ try ოპერატორს წარმოქმნილი განსაკუთრებული სიტუაციის დამმუშავებელი არ აქვს, ხდება გარე catch ბლოკის დამმუშავებლის შემოწმება შესაბამისობაზე. ეს გრძელდება მანამ, სანამ არ იქნება მოძებნილი საჭირო catch ოპერატორი ან სანამ არ გადაისინჯება ჩალაგებული try ბლოკების ყველა დონე. თუ შესაბამისი catch ბლოკი ვერ მოიძებნა, მაშინ გამონაკლისს დაამუშავებს Java სისტემის შემსრულებელი გარემო. მაგალითი:

ლისტინგი 49. ჩალაგებული try ოპერატორები

```
class NestTry {
  public static void main(String args[]) {
    try {
      int a = 0; // 1; 2; ...
      int b = 42 / a;
      System.out.println("a " + a);
```

```
        try { // ჩადგმული try ბლოკი
          if (a == 1)
            a = a / (a - a); // 0-ზე გაყოფა
          if (a == 2) {
            int c[] = { 1 };
            c[42] = 99; /* მოხდება მასივის საზღვრებს
            გარეთ გასვლა */
          }
        } catch (ArrayIndexOutOfBoundsException e){
          System.out.println("იდექსი საზღვრებს
          გარეთაა: " + e);
        }
      } catch (ArithmeticException e) {
        System.out.println("0-ზე გაყოფა: " + e);
      }
    }
}
```

როგორც პროგრამიდან ხედავთ, ერთი try ბლოკი ჩადგმულია მეორე ბლოკში. პროგრამა მუშაობს შემდეგნაირად. თუ შესრულებაზე გავუშვებთ პროგრამას, როდესაც a=0, გარე try ბლოკის შედეგად გენერირებული იქნება 0-ზე გაყოფის გამონაკლისი. პროგრამის გაშვება, როდესაც a=1 გამოიწვევს 0-ზე გაყოფის გამონაკლისს შიდა try ბლოკში. ვინაიდან შიდა (ჩადგმული) ბლოკი არ ახდენს ამ გამონაკლისის დამუშავებას, იგი გადაეცემა გარე try ბლოკს, რომელიც მას დაამუშავებს. თუ პროგრამა გაეშვება, როდესაც a=2, მაშინ აღიგზნება(აღიძვრება,წარმოშობა) მასივის ინდექსის საზღვრებს გარეთ გასვლის გამონაკლისი შიდა try ბლოკში.

ოპერატორი throw

აქამდე ჩვენ განვიხილავდით მხოლოდ იმ გამონაკლისებს, რომელსაც იწვევდა Java სისტემა. შესაძლებელია გამონაკლისი აღიგზნას ჩვენი პროგრამისაგან ცხადი სახით throw ოპერატორის საშუალებით. მისი ზოგადი ფორმა ასეთია:

throw <Throwable ეგზემპლარი>;

<Throwable ეგზემპლარი> უნდა იყოს ან Throwable კლასის ან მისი ქვეკლასის ტიპის ობიექტი. პრიმიტიული ტიპები, როგორიცაა int ან char, ასევე Throwable კლასისგან განსხვავებული, მაგალითად, String, Object არ შეიძლება გამოყენებული იქნას გამონაკლისებისათვის.

არსებობს Throwable-ს ობიექტის მიღების ორი გზა: new ოპერატორით ობიექტის შექმნით და catch ოპერატორში პარამეტრის მითითებით.

შემსრულებელი ნაკადი ჩერდება უშუალოდ throw ოპერატორის შემდეგ, მის შემდეგ ჩაწერილი ოპერატორები აღარ სრულდება. განისაზღვრება უახლოესი დახურული try ბლოკი, რომელსაც გამონაკლისის ტიპის შესაბამისი catch ოპერატორი აქვს. თუ შესაბამისობა მოიძებნა, მართვა გადაეცემა ამ ოპერატორს. თუ ვერა, მაშინ მოწმდება გარე try ბლოკი და ა.შ. თუ შესაბამისი catch ბლოკი ვერ მოიძებნა, მაშინ სტანდარტული დამმუშავებელი წყვეტს პროგრამის შესრულებას და ბეჭდავს სტეკის ტრასირებას.

ქვემოთ ნაჩვენებ მაგალითში იქმნება და აღიგზნება გამონაკლისი. დამმუშავებელი, რომელიც მას გამოიჭერს

ხელახლა აღაგზნებს გამონაკლისს გარე დამმუშავებლისათვის:

ლისტინგი 50. throw-ს დემონსტრირება:

```
class ThrowDemo {
   static void demoproc() {
      try {
         throw new NullPointerException("demo");
      } catch (NullPointerException e) {
       System.out.println("გამოიჭირება demoproc-ის
       შიგნით");
         throw e; // ხელახლა აღიგზნება
      }
   }
   public static void main(String args[]) {
      try {
         demoproc();
      } catch (NullPointerException e) {
         System.out.println("ხელმეორე გამოჭერა: " +
         e);
      }
   }
}
```

ეს პროგრამა ორჯერ იღებს შანსს ერთიდაიგივე გამონაკლისის დასამუშავებლად. პირველად main() მეთოდი აყენებს გამონაკლისის კონტექსტს, შემდეგ იძახებს demoproc(). ეს უკანასკნელი მეთოდი აყენებს სხვა კონტექსტს გამონაკლისის დასამუშავებლად და მაშინვე აღაგზნებს (წარმოშობს) NullPointerException ტიპის გამონაკლისს, რომელიც გამოიჭირება შემდეგ სტრიქონში. შემდეგ გამონაკლისი კვლავ აღიგზნება. შედეგად კონსოლზე გამოვა:

გამოიჭირება demoproc-ის შიგნით
ხელმეორე გამოჭერა: java.lang.NullPointerException: demo

ამ პროგრამაში დემონსტრირებულია, როგორ შეიძლება Java-ში შეიქმნას სტანდარტული გამონაკლისის ობიექტი. ყურადღება გაამახვილეთ სტრიქონზე:

throw new NullPointerException ("demo");

აქ new ოპერაცია გამოიყენება NullPointerException-ის ეგზემპლარის კონსტრუირებისთვის. Java-ს ბევრ სტანდარტულ გამონაკლისს აქვს მინიმუმ ორი კონსტრუქტორი: ერთი პარამეტრების გარეშე და ერთი სტრიქონული პარამეტრით. როდესაც მეორე ფორმა გამოიყენება, არგუმენტი მიუთითებს სტრიქონს, რომელიც გამონაკლისს აღწერს. ეს სტრიქონი აისახება, როდესაც ობიექტი გამოიყენება print()ან println()-ის არგუმენტად. იგი შეიძლება ასევე მივიღოთ getMessage() მეთოდის გამოძახებითაც, რომელიც განსაზღვრულია Throwable კლასში.

ოპერატორი throws

თუ მეთოდი აღაგზნებს ისეთ გამონაკლისს, რომლის დამუშავება თვითონ არ შეუძლია, მაშინ მან თავისი მოქმედება ისე უნდა აღწეროს, რომ ამ გამონაკლისის დამუშავებაზე იზრუნოს მეთოდის გამომძახებელმა კოდმა. ეს კეთდება მეთოდის გამოცხადებაში throws კონსტრუქციის დამატებით. throws კონსტრუქცია ახდენს იმ გამონაკლისი ტიპების ჩამონათვალს, რომლების წარმოშობაც მეთოდს შეუძლია. ეს საჭიროა ყველა გამონაკლისისათვის, გარდა Error, RuntimeException კლასებისა და ამ კლასების მემკვიდრე კლასებისათვის. ყველა დანარჩენი კლასი,

რომლის აღძვრა შეუძლია მეთოდს, გამოცხადებული უნდა იყოს throws კონსტრუქციაში. თუ ამას არ გავაკეთებთ, კომპილაციის დროს მოხდება შეცდომა.

throws ოპერატორიანი მეთოდის გამოცხადების ზოგადი ფორმა ასეთია:

<ტიპი> <მეთოდის სახელი> (<პარამეტრების სია>) throws <გამონაკლისების სია> {

// მეთოდის ტანი

}

<გამონაკლისების სია> ესაა მძიმით (,) გამოყოფილი გამონაკლისების სია, რომლებიც შეიძლება აღძრას მეთოდმა.

ქვემოთ განხილულია არასწორი პროგრამის მაგალითი, რომელიც ცდილობს აღძრას ისეთი გამონაკლისი, რომელსაც თვითონ არ გამოიჭერს და არ დაამუშავებს. ვინაიდან პროგრამაში throws ოპერატორი მითითებული არაა, ამიტომ ასეთი პროგრამის კომპილირებისას მივიღებთ შეცდომას:

ლისტინგი 51. ამ პროგრამაში შეცდომაა და არ კომპილირდება:

```
class ThrowsDemo {
   static void throwOne() {
      System.out.println("throwOne-ის შიგნით");
      throw new IllegalAccessException("Demo");
   }
   public static void main(String args[]) {
      throwOne();
   }
}
```

ამ პროგრამის კომპილირებისათვის საჭიროა შევიტანოთ ორი ცვლილება. პირველი - უნდა გამოვაცხადოთ, რომ throwOne() მეთოდი აღძრავს IllegalAccessException გამონაკლისს; მეორე - main() მეთოდში უნდა ჩაისვას try/catch ბლოკი, რომელიც გამოიჭერს ამ გამონაკლისს.

გასწორებული მაგალითი ასეთ სახეს მიიღებს:

ლისტინგი 52. ახლა კოდი კორექტულია:

```
class ThrowsDemo {
   static void throwOne() throws
                          IllegalAccessException {
      System.out.println("throwOne-ის შიგნით");
      throw new IllegalAccessException("Demo");
   }
   public static void main(String args[]) {
      try {
         throwOne();
      } catch (IllegalAccessException e) {
         System.out.println("გამოიჭირება" + e);
      }
   }
}
```

ამ პროგრამის გაშვების შედეგად მივიღებთ:

throwOne-ის შიგნით
გამოიჭირება java.lang.IllegalAccessException: Demo

ოპერატორი finally

როდესაც მეთოდში გამონაკლისი აღძრულია, მეთოდის შესრულების ნორმალური მიმდევრობა ირღვევა. იმისდა მიხედვით, თუ როგორაა პროგრამა კოდირებული, შესაძლებელია მოხდეს მართვის ნაადრევი დაბრუნებაც. ზოგ

მეთოდში ეს შეიძლება გახდეს სერიოზული შეცდომების მიზეზი. მაგალითად, თუ მეთოდის დაწყებისას ხსნის ფაილს, ხოლო გამოსვლისას მას ხურავს, ალბათ პროგრამისტს არ ენდომება, რომ ფაილის დახურვის პროგრამის კოდი იქნას გამოტოვებული გამონაკლისის დამუშავების მექანიზმის გამოყენების გამო. საკვანძო სიტყვა finally-ს დანიშნულებაა ამ სიტუაციის გამოსწორება.

finally ქმნის კოდის ბლოკს, როემლიც სრულდება try/catch ბლოკის შემდეგ, მაგრამ იმ ოპერატორამდე, რომელიც მოსდევს try/catch ბლოკს. finally ბლოკი სრულდება იმისდა მიუხედავად, აღიძრა თუ არა გამონაკლისი. თუ გამონაკლისი აღიძრა, finally ბლოკი სრულდება მაშინაც კი, როდესაც არცერთი catch ოპერატორი მას არ შეესაბამება. ნებისმიერ შემთხვევაში, როდესაც მეთოდი ახდენს (ან შეეცდება) try/catch ბლოკისგან მართვის დაბრუნებას გამომძახებელ კოდისათვის, ვინაიდან არ მოხდა გამონაკლისის დამუშავება ან როდესაც return ოპერატორის შედეგად ცხადად ხდება ეს დაბრუნება - მეთოდისგან მართვის დაბრუნების წინ სრულდება finally ბლოკი. ასეთი მექანიზმი მოსახერხებელია ფაილების დესკრიპტორების დასახურად ან რაიმე სხვა რესურსების გათავისუფლები-სათვის, რომლებიც დაკავებული იქნა მეთოდის დასაწყისში და განთავისუფლებული უნდა იქნას მეთოდის დასრულების შემდეგ. finally ოპერატორი აუცილებელი არაა. თუმცა თითოეული try ოპერატორი მოითხოვს (საჭიროებს) თუნდაც ერთ catch ან finally ოპერატორს. ქვემოთ ნაჩვენებია მაგალითი, სადაც სამი სხვადასხვა მეთოდი მართვას

აბრუნებს სხვადასხვანაირად, მაგრამ არცერთი მათგანი არ ახდენს finally ბლოკის გამოტოვებას:

ლისტინგი 53.

```
class FinallyDemo {
   // გამონაკლისი აღიძვრება მეთოდიდან
   static void procA() {
      try {
         System.out.println("procA-ის შიგნით");
         throw new RuntimeException("Demo");
      } finally {
         System.out.println("ბლოკი finally procA");
      }
   }
   // მართვის დაბრუნება try ბლოკში
   static void procB() {
      try {
         System.out.println("procB-ის შიგნით");
         return;
      } finally {
         System.out.println("ბლოკი finally procB");
      }
   }
   // try ბლოკის ნორმალური შესრულება
   static void procC() {
      try {
         System.out.println("procC-ის შიგნით");
      } finally {
         System.out.println("ბლოკი finally procC");
      }
   }
   public static void main(String args[]) {
      try {
         procA();
      } catch (Exception e) {
         System.out.println("გამონაკლისი
         გამოჭერილია");
      }
```

```
        procB();
        procC();
    }
}
```

ამ მაგალითში procA() ახდენს წყვეტას try ბლოკში გამონაკლისი აღძვრით. finally ბლოკი მაინც სრულდება. procB() მეთოდში მართვის დაბრუნება ხორციელდება try ბლოკში return ოპერატორის საშუალებით. finally ბლოკი სრულდება procB()-დან დაბრუნების წინ. procC() ბლოკში try სრულდება ნორმალურად, შეცდომების გარეშე. მაგრამ finally ბლოკი მაინც სრულდება.

ამ პროგრამის შესრულების შედეგი:

procA-ს შიგნით
ბლოკი finally procA
გამონაკლისი გამოჭერილია
procB-ს შიგნით
ბლოკი finally procB
procC-ს შიგნით
ბლოკი finally procC

გახსოვდეთ! თუ ბლოკი finally ასოცირდება try ბლოკთან, მაშინ finally შესრულდება try-ის დამთავრების შემდეგ.

Java-ს ჩაშენებული განსაკუთრებული სიტუაციები

java.lang სტანდარტულ პაკეტში განსაზღვრულია გამონაკლისების რამდენიმე კლასი. ზოგიერთი მათგანი გამოყენებული იქნა წინა მაგალითებში. უფრო ხშირად გამოყენებადი გამონაკლისები არიან RuntimeException სტანდარტული

კლასის ქვეკლასი ტიპის გამონაკლისები. როგორც ზემოთ აღვნიშნეთ, ამ გამონაკლისების მეთოდის throws სიაში შეტანა აუცილებელი არაა.

ამგვარ კლასებს უწოდებენ **არაკონტროლირებად** (არაშემოწმებად) გამონაკლისებს, ვინაიდან კომპილატორი აღარ ამოწმებს მეთოდის მიერ ასეთი გამონაკლისების აღძვრის ან დამუშავების ფაქტს. java.lang პაკეტში აღწერილი არაშემოწმებადი გამონაკლისები ჩამოთვლილია 3 ცხრილში. 4 ცხრილში ჩამოთვლილია java.lang პაკეტში აღწერილი ის გამონაკლისები, რომლებიც ჩართული უნდა იქნას მეთოდის throws სიაში. მეთოდებმა ეს გამონაკლისები შეიძლება აღძრას, მაგრამ თვითონ არ დაამუშაოს. ასეთ გამონაკლისებს უწოდებენ **კონტროლირებად** (შემოწმებად) გამონაკლისებს. Java-ში ასევე განსაზღვრულია რამდენიმე სხვა ტიპის გამონაკლისებიც, რომლებიც დაკავშირებულია კლასების ბიბლიოთეკასთან.

ცხრილი 3. RunTimeException-ის არაკონტროლირებადი (არაშემოწმებად) ქვეკლასები java.lang პაკეტში

გამონაკლისი	აღწერა
ArithmeticException	არითმეტიკული შეცდომა, როგორიცაა 0-ზე გაყოფა
ArrayIndexOutOfBoundsException	ინდექსის მასივის საზღვრებს გარეთ გასვლა
ArrayStoreException	მასივის ელემენტზე შეუსაბამო

	ტიპის ობიექტის მინიჭება
ClassCastException	არასწორი დაყვანა
EnumConstantNotPresentException	ჩამონათვალის განუსაზღვრელი მნიშვნელობის გამოყენების მცდელობა
IllegalArgumentException	მეთოდის გამოძახებისას გამოყენებულია არასწორი არგუმენტი
IllegalMonitorStateException	მონიტორინგის არასწორი ოპერაცია
IllegalStateException	გარსი ან პროგრამა არაა კორექტულ მდგომარეობაში
IllegalThreadStateException	მოთხოვნილი ოპერაცია არ შეესაბამება ნაკადის მიმდინარე მდგომარეობას
IndexOutOfBoundsException	ინდექსის რომელიღაც ტიპი გავიდა საზღვრებს გარეთ
NegativeArraySizeException	იქმნება უარყოფითი ზომის მასივი
NullPointerException	ნულოვანი მითითების არასწორი გამოყენება
NumberFormatException	სტრიქონის რიცხვით ფორმატში არასწორი გადაყვანა

SecurityException	უსაფრთხოების დარღვევის მცდელობა
StringIndexOutOfBounds	ინდექსის სტრიქონის ფარგლებს გარეთ გამოყენების მცდელობა
TypeNotPresentException	ტიპი ვერ მოიძებნა (J2SE 5)
UnsupportedOperationException	აღმოჩენილია დაუშვებელი ოპერაცია

ცხრილი 4. java.lang პაკეტის კონტროლირებადი (შემოწმებადი) ქვეკლასები

გამონაკლისი	აღწერა
ClassNotFoundException	კლასი ვერ მოიძებნა
CloneNotSupportedException	ისეთი ობიექტის კლონირების მცდელობა, რომელიც Cloneable ინტერფეისს არ უზრუნველყოფს
IllegalAccessException	კლასზე წვდომა არაა ნებადართული
InstantiationException	აბსტრაქტული კლასის ან ინტერფეისის ობიექტის შექმნის მცდელობა

InterruptedException	ერთი ნაკადი შეწყვეტილია მეორეთი
NoSuchFieldException	მოთხოვნილი ველი (ცვლადი) არ არსებობს
NoSuchMethodException	მოთხოვნილი მეთოდი არ არსებობს

გამონაკლისების საკუთარი ქვეკლასების შექმნა

მართალია Java-ს სტანდარტული გამონაკლისები ახდენენ ხშირად გამოყენებადი გამონაკლისების (შეცდომების) უმრავლესობის დამუშავებას, მაგრამ შეიძლება საჭირო გახდეს გამონაკლისების საკუთარი ტიპების შექმნა რომელი-ღაც სიტუაციის დასამუშავებლად, რომელიც სპეციფიკური იქნება გამოყენებითი პროგრამისათვის. ამის გაკეთება საკმაოდ მარტივად შეიძლება: აღვწეროთ Exception კლასის ქვეკლასი, რომელიც რა თქმა უნდა Throwable კლასის მემკვიდრე კლასი იქნება. არაა აუცილებელი ეს კლასები რაიმეს რეალიზაციას ახდენდნენ, მთავარია მათი არსებობა იმ ტიპების სისტემაში, რომლებსაც შეუძლიათ მისი, როგორც გამონაკლისის გამოყენება.

Exception კლასი არ აღწერს არავითარ საკუთარ მეთოდს. მას მემკვიდრეობით გადმოეცემა Throwable-ში აღწერილი მეთოდები. ამრიგად, ყველა გამონაკლისისათვის, მათ შორის თქვენს მიერ განსაზღვრული (შექმნილი) გამონაკლისები-სათვის, წვდომადია (შეუძლიათ გამოიყენონ) Throwable-ში

აღწერილი მეთოდები. ეს მეთოდები ჩამოთვლილია 5 ცხრილში:

ცხრილი 5. Throwable-ში განსაზღვრული მეთოდები

მეთოდი	აღწერა
Throwable fillInStackTrace()	აბრუნებს Throwable ობიექტს, რომელიც შეიცავს სტეკის სრულ ტრასირებას. ეს ობიექტი ხელმეორედ შეიძლება აღიგზნას
Throwable getCause()	აბრუნებს გამონაკლისს, რომელიც მიმდინარე გამონაკლისს იწვევს. თუ ასეთი არ არსებობს აბრუნებს null-ს
String getLocalizedMessage()	აბრუნებს გამონაკლისის ლოკალიზებულ აღწერას
String getMessage()	აბრუნებს გამონაკლისის აღწერას
StackTraceElement[] getStackTrace()	აბრუნებს მასივს, რომელიც შეიცავს სტეკის ტრასირებას და შედგება StackTraceElement ტიპის ელემენტებისაგან
Throwable initCause (Throwable გამონაკლისი)	<გამონაკლისს> აღიქვამს, როგორც გამონაკლისის გამომწვევს. აბრუნებს გამონაკლისზე მითითებას

void printStackTrace()	ასახავს სტეკის ტრასირებას
void printStackTrace (PrintStream ნაკადი)	აგზავნის სტეკის ტრასირებას მოცემულ ნაკადში
void printStackTrace (PrintWriter ნაკადი)	აგზავნის სტეკის ტრასირებას მოცემულ ნაკადში
void setStackTrace (StackTraceElement ელემენტები[])	აგზავნის სტეკის ტრასირებას ელემენტებში <ელემენტები[]>
String toString()	აბრუნებს String-ის ტიპის ობიექტს, რომელიც შეიცავს გამონაკლისის აღწერას. ეს მეთოდი გამოიძახება println()-დან თუ ხდება Throwable ობიექტის გამოტანა

ასევე შესაძლებელია მოვახდინოთ ჩამოთვლილი მეთოდებიდან ერთი ან რამდენიმე მათგანის გადაფარვა გამონაკლისის საკუთარ კლასში.

Exception-ში განსაზღვრულია 4 კონსტრუქტორი. ორი მათგანი დაემატა JDK 1.4-ში და მათ შემდგომში განვილილავთ, დანარჩენი ორი კი ასე გამოიყურება:

Exception ()
Exception(String tsg)
პირველი ფორმა ქმნის გამონაკლისს, რომელსაც აღწერა არ აქვს. მეორე - საშუალებას იძლევა გამონაკლისის სპეციფირება მოვახდინოთ.

მართალია ასეთი აღწერა გამონაკლისის შექმნისას ხშირად გამოსადეგია, ზოგჯერ უმჯობესია მოხდეს toString()-ის გადაფარვა.

შემდეგ მაგალითში ხდება Exception-ის შვილობილი ახალი გამონაკლისის გამოცხადება, რომელიც შემდგომში გამოიყენება მეთოდში მცდარი სიტუაციის სიგნალიზაციისათვის. იგი ხელახლა აცხადებს (გადაფარავს) toString() მეთოდს, რომელიც უფრო მოხერხებულად ახდენს გამონაკლისის ასახვას კონსოლზე:

ლისტინგი 54. ამ პროგრამაში მომხმარებლის მიერ იქმნება გამონაკლისის ახალი ტიპი

```
class MyException extends Exception {
   private int detail;

   MyException(int a) {
      detail = a;
   }

   public String toString() {
      return "MyException [" + detail + "]";
   }
}

class ExceptionDemo {
   static void compute(int a) throws MyException {
      System.out.println ("გამოიძახება compute (" + a
      + ") ");
      If (a>10)
         throw new MyException(a);
      System.out.println("ნორმალური დამთავრება");
   }

   public static void main(String args[]) {
      try {
```

```
            compute(1);
            compute(20);
        } catch (MyException e) {
            System.out.println("გამოჭერილია " + e);
        }
    }
}
```

განსაკუთრებული სიტუაციების დამუშავება წარმოადგენს მძლავრ მექანიზმს რთული პროგრამების მართვისათვის, რომლებსაც შესრულებისას მრავალი დინამიკური მახასიათებლები გააჩნიათ. მნიშვნელოვანია, რომ try, throws და catch მივუდგეთ, როგორც შეცდომებისა და არაორდინალური სიტუაციების დამუშავების საშუალებას ჩვენს პროგრამაში. სხვა ენებისაგან გასხვავებით, სადაც პროგრამაში შეფერხების (მტყუნების) დამუშავებისა და იდენტიფიკაციისათვის გამოიყენება შეცდომების ცხრილები ან შეცდომების კოდები, Java-ში გამოიყენება გამონაკლისები. ანუ, როდესაც მეთოდში ხდება შეფერხება, ის აღძრავს გამონაკლისს.

ერთი გაფრთხილება: Java-ს გამონაკლისების მართვის ოპერატორების გამოყენება არალოკალური განშტოებისათვის რეკომენდებული არაა. ამ მიზნით ამ ოპერატორების გამოყენება გაართულებს პროგრამის ტექსტს და შესაბამისად პროგრამის გამართვის პროცესს.

სტრიქონებთან მუშაობა

დაპროგრამების სხვა ენების მსგავსად Java სტრიქონი ესაა სიმბოლოების მიმდევრობა, მაგრამ ბევრი სხვა ენისაგან განსხვავებით, სადაც სტრიქონები რეალიზებულია სიმბო-

ლოების მასივის სახით, Java-ში სტრიქონები რეალიზებულია String-ის ტიპის ობიექტებად.

სტრიქონების ჩაშენებული ობიექტების სახით რეალიზაცია Java-ს საშუალებას აძლევს უზრუნველყოს მრავალი სხვადასხვა საშუალება სტრიქონების დასამუშავებლად. მაგალითად, Java გვთავაზობს მეთოდებს ორი სტრიქონის შესადარებლად, ქვესტრიქონების ძებნას, ორი სტრიქონის გაერთიანებას, სტრიქონში სიმბოლოების რეგისტრის ცვლილებას და სხვა მრავალს. ასევე შესაძლებელია String ობიექტების შექმნა სხვადასხვანაირად, რაც საშუალებას იძლევა, როდესაც საჭიროა, ადვილად შეიქმნას სტრიქონები.

შეიძლება უცნაური ფაქტია, მაგრამ როდესაც სტრიქონის ტიპის ობიექტი იქმნება, ამ სტრიქონის შეცვლა აღარ შეიძლება. ანუ, როდესაც String ობიექტი შექმნილია, მასში შემავალი სიმბოლოების შეცვლა აღარ შეიძლება. შეიძლება ეს მკაცრ შეზღუდვად მოგვეჩვენოს, მაგრამ პრაქტიკულად ეს არც ისეთი მნიშვნელოვანია. ყოველთვის, როდესაც გვჭირდება არსებული სტრიქონის ვერსიაში ცვლილების შეტანა, ხდება ახალი String ობიექტის შექმნა, რომელშიც განხორციელებულია ყველა საჭირო ცვლილება (მოდიფიკაცია). საწყისი - ორიგინალური სტრიქონი კი რჩება უცვლელი. ასეთი მიდგომა იმიტომაა გამოყენებული, რომ ფიქსირებული, უცვლელი სტრიქონის რეალიზება უფრო ეფექტურად შეიძლება, ვიდრე ცვალებადი სტრიქონის. იმ შემთხვევებისათვის, როდესაც საჭიროა ცვალებადი, მოდიფიცირებადი სტრიქონები, Java გვათავაზობს ორ არჩევანს: StringBuffer და StringBuilder კლასებს. ორივე შეიცავს ისეთ

სტრიქონებს, რომლებიც შეიძლება შექმნის შემდეგ შეიცვალოს.

კლასები String, StringBuffer და StringBuilder მოთავსებულია java.lang პაკეტში, ამიტომ ისინი ყველასათვის ავტომატურადაა წვდომადი. ეს კლასები გამოცხადებულია final მოდიფიკატორით, ამიტომ არც ერთი მათგანისგან ქვეკლასის მიღება არ შეიძლება. ეს კი საშუალებას იძლევა განხორციელდეს გარკვეული ოპტიმიზაცია, ვინაიდან მოხდეს სტრიქონებთან მუშაობის წარმადობის გაუმჯობესება. სამივე კლასი ახორციელებს CharSequence ინტერფეისის რეალიზაციას.

როდესაც ვამბობთ, რომ String ტიპის ობიექტები უცვლელია, ეს ნიშნავს, რომ String-ის ეგზემპლარის შეცვლა მისი შექმნის შემდეგ არ შეიძლება. თუმცა ცვლადი, რომელიც String ობიექტზე მიმთითებელადაა გამოცხადებული, ნებისმიერ მომენტში შეიძლება სხვა String ობიექტს დაუკავშირდეს.

სტრიქონების კონსტრუქტორები

String კლასს აქვს რამდენიმე კონსტრუქტორი. String ტიპის ცარიელი ობიექტის შესაქმნელად ხდება სტანდარტული კონსტრუქტორის გამოძახება. მაგალითად:

```
String s = new String();
```

ხშირად საჭიროა სტრიქონის შექმნა კონკრეტული საწყისი მნიშვნელობით. ამისათვის String კლასს რამდენიმე კონსტრუქტორი აქვს. სიმბოლოების მასივისაგან String კლასის შესაქმნელად ასეთი კონსტრუქტორი გამოიყენება:

String(char chars[])

მაგალითი:

char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);

ამ კონსტრუქტორით s სტრიქონის ინიციალიზაცია მოხდება "abc" სტრიქონით.

სტრიქონის კონსტრუირებისას შესაძლებელია მივუთითოთ სიმბოლოების მასივის დიაპაზონი ასეთი კონსტრუქტორით:

String(char chars[ ], int startIndex, int numChars)

აქ startIndex მიუთითებს დიაპაზონის დასაწყისს, ხოლო numChars იმ სიმბოლოების რაოდენობაა, რომლებიც გვინდა სტრიქონში შევიდეს. მაგალითი:

char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);

შეიქმნება სტრიქონი "cde".

შესაძლებელია შევქმნათ String ობიექტი, რომელიც იგივე სიმბოლოების მიმდევრობას შეიცავს რასაც მეორე String ობიექტი:

(String strObj)

აქ strObj სტრიქონული String ტიპის ობიექტია. მაგალითი:

ლისტინგი 55.

```
class MakeString {
  public static void main(String args[]) {
```

```
      char c [] = {'J', 'a', 'v', 'a'};
      String s1 = new String(c);
      String s2 = new String(s1);
      System.out.println(s1);
      System.out.println(s2);
   }
}
```

ამ პროგრამის შედეგი:

Java
Java

იმისდა მიუხედავად, რომ Java-ში char ტიპი 16 ბიტიანია, Internet-ში სტრიქონებისათვის ტიპიური ფორმატად გამოყენებულია 8 ბიტიანია ბაიტის მასივები და აგებულია ASCII სიმბოლოებისაგან. ვინაიდან 8 ბიტიანი ASCII სტრიქონები ხშირად გამოიყენება, String კლასი გვთავაზობს კონსტრუქტორებს, რომლებიც ქმნის სტრიქონებს byte ტიპის მასივისგანაც:

String(byte asciiChars[ ])
String(byte asciiChars[ ], int startIndex, int numChars)

აქ asciiChars ბაიტების მასივია. მეორე ფორმა საშუალებას იძლევა მივუთითოთ ქვედიაპაზონი. მაგალითი:

ლისტინგი 56.

```
class SubStringCons {
   public static void main(String args[]) {
      byte ascii[] = {65, 66, 67, 68, 69, 70 };
      String s1 = new String(ascii);
      System.out.println(s1);
      String s2 = new String(ascii, 2, 3);
      System.out.println(s2);
   }
}
```

შედეგი:

ABCDEF
CDE

აღვნიშნოთ, რომ როდესაც ხდება String ობიექტის შექმნა სიმბოლოების მასივისაგან, მასივის შემცველობა ყოველთვის კოპირდება. თუ შემდეგ მოვახდენთ სიმბოლოების მასივის მოდიფიცირებას (ცვლილებას), შექმნილ String ობიექტში არავითარი ცვლილება არ მოხდება.

String ობიექტის კონსტრუირება შესაძლებელია StringBuffer კლასის ობიექტისაგან კონსტრუქტორით:

String(StringBuffer strBufObj)

J2SE 5-ში დამატებულია ორი ახალი კონსტრუქტორი.

პირველი მათგანი ახორციელებს Unicode სიმბოლოების გაფართოებული ნაკრების მხარდაჭერას და ასე გამოიყურება:

String(int codePoints[], int startIndex, int numChars)

აქ codePoints ესაა Unicode სიმბოლოების შემცველი მასივი. სტრიქონი შეიქმნება startIndex-დან დაწყებული და იქნება numChars სიგრძის.

მეორე ახალი კონსტრუქტორი დაკავშირებულია ახალ StringBuilder კლასთან და ასე გამოიყურება:

String(StringBuilder strBuildObj)

შედეგად StringBuilder ტიპის strBuildObj ობიექტისაგან შეიქმნება String ობიექტი.

სტრიქონის სიგრძე

სტრიქონის სიგრძე ესაა მისი შემადგენელი სიმბოლოების რაოდენობა. მისი მნიშვნელობის მისაღებად უნდა გამოვიძახოთ მეთოდი length():

int length()

შემდეგი მაგალითი დაბეჭდავს 3-ს:

```
char chars [] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

სპეციალური სტრიქონული ოპერაციები

ვინაიდან სტრიქონები პროგრამირებაში ხშირად გამოიყენება, ამიტომ ენის სინტაქსში დაამატეს ზოგიერთი ოპერაციის სპეციალური მხარდაჭერა. ამ ოპერაციებით მიიღწევა String-ის ეგზემპლარების ავტომატური მიღება სტრიქონული ლიტერალებისაგან; + ოპერაციის საშუალებით ხდება რამდენიმე String ობიექტის კონკატენაცია (შეერთება, გადაბმა); სხვა ტიპის მონაცემების გადაყვანა სტრიქონულ წარმოდგენაში. არსებობს ამ ფუნქციების განხორციელების ცხადი მეთოდები, მაგრამ Java მათ ავტომატურადაც ასრულებს პროგრამისტების ხელშეწყობისა და მეტი სიცხადისათვის.

სტრიქონული ლიტერალები

წინა მაგალითებში ნაჩვენები იყო როგორ შეიძლება შეიქმნას String ობიექტები ცხადად სიმბოლოების მასივისაგან new ოპერაციით. შესაძლებელია ეს უფრო ადვილად მოვახდინოთ სტრიქონული ლიტერალების საშუალებით. ყოველი სტრიქონული ლიტერალისათვის პროგრამაში ავტომატურად იქმნება String ობიექტი. ამრიგად, შესაძლებელია სტრიქონული ლიტერალი გამოვიყენოთ String ობიექტის ინიციალიზაციისათვის. მაგალითად:

```
char chars [] = { 'a', 'b', 'c' };
String s1 = new String(chars);
String s2 = "abc"; /* გამოიყენება სტრიქონული
       ლიტერალი */
```

ვინაიდან String ობიექტის კონსტრუირება ხდება ყოველი ლიტერალისათვის, ამიტომ ლიტერალი შეგვიძლია გამოვიყენოთ ყველგან, სადაც დასაშვებია String ობიექტის გამოყენება. მაგალითად, შესაძლებელია მეთოდების გამოძახება უშუალოდ ორმაგი ბრჭყალებში ჩასმული სტრიქონის შემდეგ:

```
System.out.println("abc".length());
```

სტრიქონების კონკატენაცია

საზოგადოდ, Java არ იძლევა საშუალებას გამოვიყენოთ ოპერაციები String ობიექტებზე. ამ წესიდან ერთ გამონაკლისს წარმოადგენს ოპერაცია +, რომელიც აერთებს ორ სტრიქონს და შედეგად ქმნის ახალ String ობიექტს. შესაძლებელია რამდენიმე + ოპერაციის განხორციელებაც:

```
String age = "9";
String s = "იგი " + age + " წლისაა.";
System.out.println(s);
```

შედეგად დაიბეჭდება *„იგი 9 წლისაა“*.

პრაქტიკული რჩევა: როდესაც პროგრამაში შესატანია ერთი გრძელი სტრიქონი, უმჯობესია იგი დაიყოს რამდენიმე მოკლე სტრიქონად და გამოიყენოთ + ოპერაცია მათ შორის. მაგალითი:

ლისტინგი 57.

```
class ConCat {
   public static void main(String args[]) {
      String longStr = "აქ შეიძლება იყოს "
        + "ძალიან გრძელი სტრიქონი, რომელიც ჯობია "
        + "გადატანილი იქნას; კონკატენაციას ეს შეუძლია";
      System.out.println(longStr);
   }
}
```

კონკატენაცია სხვადასხვა ტიპის მონაცემებთან

შესაძლებელია სტრიქონები შევაერთოთ სხვა ტიპის მონაცემებთან. მაგალითად:

```
int age = 9;
String s = "იგი" + age + " წლისაა.";
System.out.println(s);
```

ამ შემთხვევაში age-ს ტიპია int და არა String, მაგრამ შედეგი მაინც იგივე მიიღება რაც წინა მაგალითში. ასე იმიტომ ხდება, რომ int ტიპის მნიშვნელობა ავტომატურად გარდაიქმნება სტრიქონულ წარმოდგენაში String ობიექტად.

ამის შემდეგ ჩვეულებრივ ხდება სტრიქონების კონკატენაცია. კომპილატორი ოპერანდებს გარდაქმნის მათ სტრიქონულ ეკვივალენტში. ყურადღება უნდა გავამახვილოთ კონკატენაციის გამოსახულებაში სხვადასხვა ტიპის ოპერანდების შერევაზე, წინააღმდეგ შემთხვევაში შეიძლება მივიღოთ მოულოდნელი შედეგები. მაგალითად:

String s = "ოთხი: " + 2 + 2;
System.out.println(s);
კონსოლზე გამოვა:

ოთხი: 22

ეს იმიტომ გამოვიდა, რომ ოპერაციების პრიორიტეტის გამო ჯერ მოხდება კონკატენაციის ოპერაცია სტრიქონ "ოთხი: "-სა და რიცხვ 2-ს შორის. შედეგი შემდეგ გაერთიანდება 2-ის სტრიქონულ ეკვივალენტთან. თავიდან შეკრების ოპერაცია რომ ჩატარდეს საჭიროა ფრჩხილების ასეთნაირი გამოყენება:

String s =" ოთხი: " + (2 + 2);

ახლა გამოვა:

ოთხი: 4

სტრიქონების გარდაქმნა და toString()

როდესაც კონკატენაციისას Java გარდაქმნის მონაცემებს სტრიქონულ ფორმაში, იგი ამას ახორციელებს valueOf() გადატვირთული მეთოდის ერთერთი ვერსიის გამოძახების საშუალებით, რომელიც String კლასშია განსაზღვრული. valueOf() მეთოდი გადატვირთულია ყველა ბაზისური

(პრიმიტიული) ტიპის მონაცემისათვის და Object ტიპისათვის. ბაზისური ტიპებისათვის valueOf() მეთოდი აბრუნებს სტრიქონს, რომელიც ადამიანისათვის კითხვადია მისთვის ჩვეულ ფორმატში. ობიექტებისათვის valueOf() მეთოდი იძახებს ამ ობიექტის toString() მეთოდს. valueOf() მეთოდს ჩვენ მომავალში კიდევ უფრო დეტალურად განვიხილავთ, ახლა გავარჩიოთ toString() მეთოდი.

ყოველი კლასი ახდენს toString() მეთოდის რეალიზაციას, ვინაიდან ეს მეთოდი Object კლასშია განსაზღვრული. მაგრამ მისი სტანდარტული ვერსია ხშირად გამოუსადეგარია. ამიტომ ჩვენს მიერ შედგენილი კლასებისათვის შესაძლებელია მოვახდინოთ მისი გადაფარვა (ხელახალი აღწერა) ისე, როგორც ამას ჩვენ ვთვლით საჭიროდ. toString() მეთოდს ასეთი ფორმა აქვს:

String toString()

მისი რეალიზაციისათვის, უნდა დავაბრუნოთ String ობიექტი, რომელიც ადამიანისათვის ადვილად კითხვადი იქნება და ადეკვატურად ასახავს კლასის ობიექტს.

კლასებისათვის toString()-ის გადაფარვით, კლასები სრულად ინტეგრირდება Java-ს პროგრამულ გარემოსთან. მაგალითად, ისინი უკვე შეიძლება გამოყენებული იქნან print(), println() მეთოდებში და სხვა სტრიქონულ გამოსახულებებში კონკატენაციისას. შემდეგ პროგრამაში დემონსტრირებულია toString()-ის გადაფარვა:

ლისტინგი 58. toString()-ის ხელახალი გამოცხადება Box კლასისათვის

```
class Box {
   double width;
   double height;
   double depth;
   Box(double w, double h, double d) {
      width = w;
      height = h;
      depth = d;
   }
   public String toString() {
      return "ზომებია" + width + " x " +
      depth + " x " + height + ".";
 }
}
class toStringDemo {
   public static void main(String args[])
      Box b = new Box(10, 12, 14);
      String s = "Box b: " + b; / / Box ობიექტის
      კონკატენაცია
      System.out.println(b); /* Box-ის გარდაქმნა
                                სტრიქონად */
      System.out.println(s);
   }
}
```

ამ პროგრამის შედეგია:

ზომებია 10.0 x 14.0 x 12.0
Box b: ზომებია 10.0 x 14.0 x 12.0

სიმბოლოების ამოღება

String კლასი გვთავაზობს String ობიექტიდან სიმბოლოების ამორჩევის მრავალ მეთოდს. სტრიქონის შემადგენელი სიმბოლოები არ შეიძლება ინდექსირებით ამოვირჩიოთ, როგორც ამას ვახდენდით მასივის შემთხვევაში, მაგრამ String კლასის ბევრი მეთოდი იყენებს ინდექსს ან

წანაცვლებას. მასივების მსგავსად სტრიქონების დამმუშავებელ მეთოდებში ინდექსირება იწყება ნულიდან.

მეთოდი charAt()

სტრიქონიდან ერთადერთი სიმბოლოს ამოსარჩევად, მას შეიძლება მივმართოთ charAt() მეთოდით. მას ასეთი ფორმა აქვს:

char charAt(int where)

აქ where საჭირო სიმბოლოს ინდექსია. Where-ს მნიშვნელობა არ უნდა იყოს უარყოფითი და უნდა უჩვენებდეს სიმბოლოს პოზიციას. იგი აბრუნებს მითითებულ პოზიციაში მყოფ სიმბოლოს. მაგალითად:

char ch;
ch = "abc".charAt (1);
ch ცვლადს მიენიჭება სიმბოლო b.

მეთოდი getChars()

თუ საჭიროა ერთზე მეტი სიმბოლოს ერთდროულად ამოღება შეგვიძლია გამოვიყენოთ getChars() მეთოდი. მისი ზოგადი ფორმაა:

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

აქ sourceStart მიუთითებს ქვესტრიქონის დასაწყისის ინდექსს, sourceEnd - იმ სიმბოლოს ინდექსია, რომელიც საჭირო ქვესტრიქონის შემდეგ მოდის. ამოიღება

ქვესტრიქონი, რომელიც შეიცავს სიმბოლოებს დაწყებული sourceStart პოზიციიდან დამთავრებული sourceEnd-1 -მდე. მასივი, რომელიც ამ სიმბოლოებს მიიღებს, მითითებულია target პარამეტრით. target მასივში ის ინდექსი, საიდანაც მოხდება ქვესტრიქონის ჩაწერა, მიეთითება targetStart-ში. პროგრამისტმა უნდა იზრუნოს, რომ target მასივის სიგრძე ისეთი იყოს, რომ ქვესტრიქონმა არ გადაავსოს. მაგალითი:

ლისტინგი 59.

```
class getCharsDemo {
   public static void main(String args[]) {
      String s = "ესაა getChars მეთოდის
                                          დემონსტრირება";
      int start = 5;
      int end = 8;
      char buf[] = new char[end - start];
      s.getChars(start, end, buf, 0);
      System.out.println(buf);
   }
}
```

პროგრამის შედეგი:

get

მეთოდი getBytes ( )

არსებობს getChars-ის ალტერნატივა, რომელიც სიმბოლოებს ინახავს ბაიტების მასივში. ეს მეთოდია getBytes ( ). მისი მარტივი ფორმა ასეთია:

byte [] getBytes ()

არსებობს ამ მეთოდის სხვა ფორმებიც. ეს მეთოდი ძირითადად გამოიყენება, როდესაც ხდება String-ის ისეთ გარემოსთვის ექსპორტირება, რომელსაც არა აქვს 16 ბიტიანი Unicode-ს მხარდაჭერა. მაგალითად, ინტერნეტის უმრავლესი პროტოკოლები და ASCII ცოდნის 8 ბიტიანი სიმბოლოები.

მეთოდი toCharArray ( )

თუ საჭიროა String ობიექტის ყველა სიმბოლოს გადაყვანა სიმბოლოების მასივში, ამის გაკეთების უმარტივესი გზაა toCharArray ( ) მეთოდის გამოძახება. მისი ზოგადი ფორმაა:

char [] toCharArray()

ეს ფუნქცია დამატებითაა წარმოდგენილი, ვინაიდან იგივეს გაკეთება შესაძლებელია getChars () მეთოდის საშუალებით.

სტრიქონების შედარება

String კლასში ჩართულია რამდენიმე მეთოდი, სტრიქონების ან მისი ქვესტრიქონების შესადარებლად. განვიხილოთ ისინი.

მეთოდები equals () და equalsIgnoreCase ()

ორი სტრიქონის ეკვივალენტურობის შესადარებლად გამოიყენება equals() მეთოდი. მისი ზოგადი ფორმაა:

boolean equals(Object str)

აქ str ესაა String ობიექტი, რომელიც შედარდება გამომძახებელ String ობიექტს. მეთოდი აბრუნებს true-ს, თუ სტრიქონები შეიცავს ერთიდაიგივე სიმბოლოების მიმდევრობას. წინააღმდეგ შემთხვევაში შედეგი იქნება false. შედარება დამოკიდებულია და ითვალისწინებს სიმბოლოების რეგისტრს.

თუ გვინდა მოვახდინოთ ისეთი შედარება, რომ სიმბოლოების რეგისტრი არ იქნას გათვალისწინებული, უნდა გამოვიძახოთ equalsIgnoreCase() მეთოდი. როდესაც ეს მეთოდი ორ სტრიქონს ადარებს, იგი განიხილავს A – Z დიაპაზონს, როგორც ეკვივალენტურს (ანუ იგივეს) a – z დიაპაზონისა. მისი ზოგადი ფორმაა:

boolean equalsIgnoreCase(Object str)

აქაც str ესაა String ობიექტი, რომელიც შედარდება გამომძახებელ String ობიექტს. მეთოდი აბრუნებს true-ს, თუ სტრიქონები შეიცავს ერთიდაიგივე სიმბოლოების მიმდევრობას. წინააღმდეგ შემთხვევაში შედეგი იქნება false. მაგალითი:

ლისტინგი 60. equals() და equalsIgnoreCase() მეთოდების დემონსტრირება

```
class equalsDemo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = " Good¬bye ";
    String s4 = "HELLO";
    System.out.println(s1 + " ეკვივალენტურია " +
                       s2 + " -> " +
```

```
        s1.equals(s2));
        System.out.println(s1 + " ეკვივალენტურია " +
                                    s3 + " -> " +
        s1.equals(s3));
        System.out.println(s1 + " ეკვივალენტურია " +
                                    s4 + " -> " +
        s1.equals (s4));
        System.out.println(s1+"რეგისტრის იგნორირებით "
                    + "ეკვივალენტურია " + s4 + " ->
        " + s1.equalsIgnoreCase(s4));
    }
}
```

პროგრამის შესრულების შედეგი:

Hello ეკვივალენტურია Hello -> true
Hello ეკვივალენტურია Goodbye -> false
Hello ეკვივალენტურია HELLO -> false
Hello რეგისტრის იგნორირებით ეკვივალენტურია HELLO -> true

მეთოდი regionMatches()

მეთოდი regionMatches() ადარებს სტრიქონის მითითებულ ნაწილს რომელიმე (შეიძლება იგივე) სტრიქონის სხვა ნაწილთან. არსებობს მისი გადატვირთული ვერსია, რომელიც შედარებისას ყურადღებას არ აქცევს სიმბოლოების რეგისტრს. ამ მეთოდების ზოგადი ფორმაა:

boolean regionMatches(int startIndex, String str2,
int str2StartIndex, int numChars)

boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)

ორივე ვერსიაში startIndex მიუთითებს გამომძახებელი String ობიექტის დიაპაზონის დასაწყისის ინდექსს. შესადარებელი სტრიქონი მეთოდს გადაეცემა str2 სტრიქონით. იმ სიმბოლოს ინდექსი str2 სტრიქონში, რომლიდან დაწყებული უნდა მოხდეს სიმბოლოების შედარება მითითებულია str2StartIndex ცვლადით, ხოლო შესადარებელი ქვესტრიქონის სიგრძე - numChars-ით. მეორე ვერსიაში, თუ ignoreCase ტოლია true-სი, ხდება სიმბოლოების რეგისტრის იგნორირება. წინააღმდეგ შემთხვევაში რეგისტრი გაითვალისწინება.

მეთოდები startsWith () და endsWith()

String კლასში განსაზღვრულია ორი მეთოდი, რომლებსაც regionMatches() მეთოდთან შედარებით, მეტნაკლებად სპეციალიზებული ფორმა აქვთ. startsWith() მეთოდი განსაზღვრავს, იწყება თუ არა მოცემული String ობიექტი მითითებული სტრიქონით. შესაბამისად, endsWith() მეთოდი განსაზღვრავს, მთავრდება თუ არა String ობიექტი მითითებული ფრაგმენტით. ამ მეთოდებს აქვთ შემდეგი ზოგადი სახე:

boolen startsWith(String str)
boolen endsWith(String str)

str ესაა სტრიქონის ფრაგმენტი, რომლის არსებობა, შესაბამისად მოცემული სტრიქონის თავში ან ბოლოში, უნდა შემოწმდეს. თუ დამთხვევას აქვს ადგილი მეთოდი აბრუნებს true-ს, წინააღმდეგ შემთხვევაში false-ს. მაგალითი:

"Foobar".endsWith("bar")

და

"Foobar".startsWith("Foo")

აბრუნებენ true-ს.

equals () და == ოპერაციების შედარება

საჭიროა გვესმოდეს განსხვავება equals() მეთოდსა და == ოპერაციას შორის. ეს ორი სხვადასხვა მოქმედებაა. როგორც ზემოთ აღვწერეთ მეთოდი equals() ადარებს სიმბოლოებს String ობიექტის შიგნით. ოპერაცია == ადარებს მითითებას ორ ობიექტზე და განსაზღვრავს, უთითებენ თუ არა ისინი ერთიდაიგივე ეგზემპლარზე. შემდეგ მაგალითში ნაჩვენებია, რომ ორი სხვადასხვა String ობიექტი შეიძლება შეიცავდეს ერთნაირ სიმბოლოებს, მაგრამ შედარებისას ამ ობიექტებზე მითითება არ იყოს ეკვივალენტური:

ლისტინგი 61. equals() vs ==

```
class EqualsNotEqualTo {
   public static void main(String args[]) {
      String s1 = "Hello";
      String s2 = new String(s1);
      System.out.println(s1 + " equals" + s2 + " ->"
                                 + s1.equals(s2));
      System.out.println(s1 + " == " + s2 + " -> "
                                 + (s1 == s2));
   }
}
```

s1 ცვლადი დაკავშირებულია String კლასის ეგზემპლართან, რომელიც შექმნილია "Hello" ლიტერალის მინიჭებით.

ობიექტი, რომელთანაც დაკავშირებულია s2, შექმნილია s1 ცვლადის ინიციალიზატორად გამოყენებით. ამგვარად, ორივე String ობიექტის შემცველობა იდენტურია, მაგრამ ისინი ერთმანეთისაგან განსხვავებული ობიექტებია. ეს კი ნიშნავს, რომ s1 და s2 ცვლადები არ მიუთითებენ ერთიდაიგივე ობიექტზე და ამიტომ ერთმანეთის ტოლი არ არიან (== ოპერაციით შედარებისას). წინა პროგრამის შედეგი ამას ამტკიცებს:

Hello equals Hello -> true
Hello == Hello -> false

მეთოდი compareTo ()

ხშირად საკმარისი არაა უბრალოდ ვიცოდეთ, რომ სტრიქონები იდენტურია. პროგრამა, რომელიც სორტირებას ახდენს, საჭიროა ვიცოდეთ რომელი სტრიქონია ნაკლები, ტოლი ან მეტი მეორე სტრიქონზე. სტრიქონი ნაკლებია მეორე სტრიქონზე, თუ იგი ლექსიკოგრაფიულ მიმდევრობაში უფრო წინაა განლაგებული. სტრიქონი მეტია მეორე სტრიქონზე, თუ იგი ლექსიკოგრაფიულ მიმდევრობაში უფრო შემდეგაა. compareTo() მეთოდის საშუალებით ხდება სტრიქონების ამდაგვარი შედარება და მას ასეთი ზოგადი ფორმა აქვს:

int compareTo(String str)

str ესაა String ობიექტი, რომელიც დარდება გამომძახებელ String ობიექტს. მეთოდის მიერ დაბრუნებული შედეგი ასე განისაზღვრება:

ნულზე ნაკლები	გამომძახებელი სტრიქონი str-ზე ნაკლებია
ნულზე მეტი	გამომძახებელი სტრიქონი str-ზე მეტია
ნული	ორი სტრიქონი ეკვივალენტურია

ქვემოთ ნაჩვენებია პროგრამა, რომელიც მასივის სტრიქონების სორტირებას (მოწესრიგებას, დალაგებას) აკეთებს:

ლისტინგი 62. String ობიექტის „ბუშტუკისებური“ სორტირება

```
class SortString {
   static String arr[] = { "Now", "is", "the",
      "time", "for", "all", "good","men", "to",
      "come", "to", "the", "aid", "of", "their",
      "country" };
   public static void main(String args[]) {
      for (int j = 0; j < arr.length; j++) {
         for (int i = j + 1; i < arr.length; i++) {
            if (arr[i].compareTo(arr[j]) < 0) {
               String t = arr[j];
               arr[j] = arr[i];
               arr[i] = t;
            }
         }
         System.out.println(arr[j]);
      }
   }
}
```

ამ პროგრამის შედეგია:

Now
aid
all
come
country
for

good
is
men
of
the
the
their
time
to
to

როგორც ხედავთ, compareTo() მეთოდი ყურადღებას აქცევს დიდ და პატარა ასოებს. სიტყვა Now დგას ყველაზე წინ, ვინაიდან ის იწყება დიდი ასოთი, რაც იმას ნიშნავს, რომ მისი მნიშვნელობა ASCII კოდში ყველაზე პატარაა.

თუ გვინდა, რომ სტრიქონების შედარებისას რეგისტრის იგნორირება მოვახდინოთ, უნდა გამოვიყენოთ compareToIgnoreCase() მეთოდი:

int compareToIgnoreCase(String str)

ეს მეთოდი იგივე შედეგს აბრუნებს, რასაც compareTo() მეთოდი, ოღონდ სიმბოლოების რეგისტრი იგნორირდება. თუ წინა მაგალითში ამ მეთოდს გამოვიყენებთ New აღარ იქნება სიაში პირველი სიტყვა.

სტრიქონების ძებნა

String კლასი გვთავაზობს ორ მეთოდს, რომლებსაც შეუძლიათ განახორციელონ სტრიქონში გარკვეული სიმბოლოს ან ქვესტრიქონის ძებნა.

indexOf() - ეძებს პირველად შემხვედრ სიმბოლოს ან ქვესტრიქონის;

lastIndexOf() - ეძებს სულ ბოლო შემხვედრ სიმბოლოს ან ქვესტრიქონს.

ეს ორი მეთოდი სხვადასხვანაირადაა გადატვირთულია. ყველა შემთხვევაში ისინი აბრუნებენ სტრიქონში იმ პოზიციას (ინდექსს), სადაც სიმბოლო ან ქვესტრიქონი იქნა მოძებნილი ან -1 წარუმატებლობის შემთხვევაში.

სიმბოლოს პირველად შეხვედრის მოსაძებნად გამოიყენება მეთოდი:

int indexOf(char ch)
int lastIndexOf (char ch)
ch ესაა სიმბოლო, რომელიც უნდა მოიძებნოს.

ქვესტრიქონის პირველი ან ბოლო შეხვედრის მოსაძებნად გამოიყენება მეთოდები:

int indexOf(String str)
int lastlndexOf(String str)
str აქ მიუთითებს საძიებო ქვესტრიქონს.

შესაძლებელია მივუთითოთ ძიების დასაწყისის პოზიცია თუ გამოვიყენებთ მეთოდების შემდეგ ფორმებს:

int indexOf(char ch, int startIndex)
int lastIndexOf(char ch, int startIndex)
int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)

startIndex აქ მიუთითებს საწყის პოზიციას. indexOf() მეთოდისათვის ძებნა იწყება startIndex-დან და გრძელდება სტრიქონის ბოლომდე, ხოლო lastIndexOf()-თვის იწყება startIndex-დან და გრძელდება 0-მდე.

შემდეგი მაგალითი აჩვენებს, როგორ გამოვიყენოთ სხვადასხვა ინდექსიანი მეთოდები String–ის შიგნით ძებნისათვის:

ლისტინგი 63. indexOf() და lastIndexOf() მეთოდები

```
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men "
        + "to come to the aid of their country.";
    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf (the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                     s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) " +
                       s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) " +
                     s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) " +
```

```
                        s.indexOf("the", 10));
    System.out.println("lastIndexOf (the, 60) " +
                        s.lastIndexOf("the", 60));
  }
}
```

პროგრამის შესრულების შედეგი:

Now is the time for all good men to come to the aid of their country.
indexOf (t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55

სტრიქონების მოდიფიცირება

ვინაიდან String ტიპის ობიექტები უცვლელია, ყოველთვის, როცა საჭიროა მათი მოდიფიცირება, ან უნდა მოვახდინოთ მათი შემცველობის კოპირება StringBuffer-ში ან StringBuilder-ში, ანდა გამოვიყენოთ String კლასის რომელიმე მეთოდი, რომელიც ახდენს სტრიქონის ახალი კოპიოს კონსტრუირებას, რომელშიც საჭირო მოდიფიცირებები ჩატარებული იქნება.

მეთოდი substring ()

substring() მეთოდით შესაძლებელია ქვესტრიქონის ამოღება. მას აქვს ორი ფორმა. პირველი:

String substring(int startIndex)

startIndex მიუთითებს ინდექსს, საიდანაც დაიწყება ქვესტრიქონი. ეს ფორმა აბრუნებს ქვესტრიქონის კოპიოს, რომელიც იწყება startIndex პოზიციიდან და გრძელდება გამომძახებელი სტრიქონის ბოლომდე.

substring()-ის მეორე ფორმა საშუალებას იძლევა მივუთითოთ ქვესტრიქონის, როგორც საწყისი, ისე საბოლოო ინდექსი:

String substring(int startIndex, int endIndex)

startIndex მიუთითებს ინდექსს, საიდანაც იწყება ქვესტრიქონი, ხოლო endIndex - ქვესტრიქონის ბოლო წერტილს. დაბრუნებული სტრიქონი შეიცავს ყველა სიმბოლოს, დაწყებული პირველი პოზიციიდან ბოლომდე, ბოლო სიმბოლოს გამოკლებით.

შემდეგ პროგრამაში substring() გამოიყენება სტრიქონში ერთი ქვესტრიქონის ყველა ეგზემპლარის მეორით შესაცვლელად:

ლისტინგი 64. ქვესტრიქონების შეცვლა

```
class StringReplace {
  public static void main(String args[]) {
    String org = "This is a test. This is, too.";
    String search = "is";
```

```
      String sub = "was";
      String result = "";
      int i;
      do { // ყველა თანხვედრილი ქვესტრიქონის შეცვლა
         System.out.println(org);
         i = org.indexOf(search);
         if (i != -1) {
            result = org.substring(0, i);
            result = result + sub;
            result = result + org.substring(i +
                                 search.length());
            org = result;
         }
      } while (i != -1);
   }
}
```

ამ პროგრამის შედეგი:

This is a test. This is, too.

Thwas is a test. This is, too.

Thwas was a test. This is, too.

Thwas was a test. Thwas is, too.

Thwas was a test. Thwas was, too.

მეთოდი concat ()

ორი სტრიქონის შესაერთებლად გამოიყენება concat() მეთოდი:

String concat(String str)

ეს მეთოდი ქმნის ახალ ობიექტს, რომელიც შეიცავს გამომძახებელ სტრიქონის ბოლოზე მიერთებულ str სტრიქონს. ეს მეთოდი ასრულებს იმავე ფუნქციას რასაც + ოპერაცია. მაგალითად:

String s1 = "one";
String s2 = s1.concat("two");
ამ კოდის შედეგი იგივეა რაც:

String s1 = "one";
String s2 = s1 + "two";

მეთოდი replace ()

ამ მეთოდს აქვს ორი ფორმა. პირველი, საწყის ტექსტში ერთ სიმბოლოს ყველგან ცვლის (ჩაანაცვლებს) მეორე სიმბოლოთი. ეს ფორმაა:

String replace(char original, char replacement)

Original მიუთითებს სიმბოლოს, რომელიც უნდა შეიცვალოს replacement სიმბოლოთი. მეთოდი აბრუნებს მიღებულ სტრიქონს. მაგალითად:

String s = "Hello".rep1ace('l', 'w');

s სტრიქონში მოთავსდება "Hewwo".

replace() მეთოდის მეორე ფორმა სიმბოლოების ერთ მიმდევრობას ჩაანაცვლებს მეორეთი:

String replace(CharSequence original, CharSequence replacement)

ეს ფორმა დაემატა J2SE 5-ში.

მეთოდი trim()

მეთოდი trim() აბრუნებს გამომძახებელი სტრიქონის კოპიოს, რომელსაც ჩამოშორებული აქვს ყველა საწყისი და საბოლოო ცარიელი სიმბოლო (space, " "). მას ასეთი ფორმა აქვს:

String trim ()

მაგალითი:

String s = " Hello World ".trim();

s-ს მიენიჭება "Hello World".

მონაცემების გარდაქმნა valueOf() მეთოდით

valueOf() მეთოდი მონაცემებს შიდა წარმოდგენიდან გარდაქმნის მომხმარებლისთვის უფრო კითხვად ფორმაში. ეს მეთოდი სტატიკურია და გადატვირთულია String კლასში ისეთნაირად, რომ Java-ში არსებული ყველა ჩაშენებული ტიპი სწორად იქნას გარდაქმნილი სტრიქონად. valueOf() მეთოდი ასევე გადატვირთულია Object კლასისათვის, ამიტომ ჩვენს მიერ შექმნილი ნებისმიერი კლასის ობიექტი შეიძლება გამოყენებული იქნას მის არგუმენტად. ქვემოთ ნაჩვენებია ზოგიერთი მისი ფორმა:

static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[])

როგორც ადრე აღვნიშნეთ, valueOf() გამოიძახება, როდესაც საჭიროა რომელიმე სხვა ტიპის სტრიქონული წარმოდგენა, მაგალითად, კონკატენაციის დროს. შესაძლებელია ამ მეთოდის უშუალოდ გამოძახება მონაცემთა ნებისმიერი ტიპისათვის და მიღებული იქნება String ტიპის ადეკვატური წარმოდგენა. ყველა მარტივი (პრიმიტიული) ტიპი გარდაიქმნება ზოგად წარმოდგენაში. ყოველი ობიექტი, რომელსაც გადავცემთ valueOf() მეთოდს, დააბრუნებს ამ ობიექტის toString() მეთოდის შესრულების შედეგს. ფაქტიურად შესაძლებელია toString() მეთოდის გამოძახება და შედეგად მივიღებთ იგივეს.

მასივების უმრავლესობისათვის valueOf() აბრუნებს დაშიფრულ სტრიქონს, რომელიც აღნიშნავს, რომ მასივი არის რაღაც გარკვეული ტიპის. ოღონდ, char მასივებისათვის იქმნება String ობიექტი, რომელიც შედგება char მასივის ყველა სიმბოლოსაგან. მის გადატვირთულ ვერსიას ასეთი ფორმა აქვს:

static String valueOf(char chars[], int startIndex, int numChars)

chars ესაა მასივი, რომელიც შეიცავს სიმბოლოებს, startIndex მასივის საწყისი პოზიციაა, საიდანაც იწყება ქვესტრიქონი, ხოლო numChars მიუთითებს ქვესტრიქონის სიგრძეს.

სტრიქონში სიმბოლოების რეგისტრის ცვლილება

მეთოდი toLowerCase() სტრიქონის ყველა სიმბოლოს გარდაქმნის ზედა რეგისტრიდან ქვედა რეგისტრში. მეთოდი toUpperCase() სტრიქონის ყველა სიმბოლოს ქვედა

რეგისტრიდან გარდაქმნის ზედაში. არაასოითი სიმბოლოები, როგორიცაა ათობითი ციფრები, რჩება უცვლელი. ამ მეთოდების ზოგადი ფორმაა:

String toLowerCase( )

String toUpperCase( )

ორივე მეთოდი აბრუნებს String ობიექტს, რომელიც შეიცავს გამომძახებელი სტრიქონის ეკვივალენტს შესაბამისად ზედა ან ქვედა რეგისტრში. მაგალითი:

ლისტინგი 65. toUpperCase() და toLowerCase() მეთოდები

```
class ChangeCase {
   public static void main(String args[]) {
      String s = "This is Test.";
      System.out.println("საწყისი სტრიქონი: " + s);
      String upper = s.toUpperCase();
      String lower = s.toLowerCase();
      System.out.println("ზედა რეგისტრი: " + upper);
      System.out.println("ქვედა რეგისტრი: " + lower);
   }
}
```

საწყისი სტრიქონი: This is Test.
ზედა რეგისტრი: THIS IS TEST
ქვედა რეგისტრი: this is test

String-ის დამატებითი მეთოდები

ზემოთ ჩამოთვლილი მეთოდების გარდა String კლასი შეიცავს სხვა მეთოდებსაც. ისინი ჩამოთვლილია ცხრილ 6-ში. ამ მეთოდების დიდი ნაწილი დაემატა J2SE 5.

ცხრილი 6. String კლასის დამატებითი მეთოდები

მეთოდი	აღწერა
int codePointAt (int i)	აბრუნებს i პოზიციაში მყოფი Unicode სიმბოლოს კოდის წერტილს (J2SE 5)
int codePointBefore (int i)	აბრუნებს i პოზიციის წინა პოზოციაზე მყოფი Unicode სიმბოლოს კოდის წერტილს (J2SE 5)
int codePointCount (int start, int end)	აბრუნებს გამომძახებელი სტრიქონის კოდის წერტილების რაოდენობას start-დან end-1 მდე (J2SE 5)
boolean contains(CharSequence str)	აბრუნებს true, თუ გამომძახებელი ობიექტი str სტრიქონს შეიცავს. წინააღმდეგ შემთხვევაში აბრუნებს false-ს (J2SE 5)
boolean contentEquals(CharSequence str)	აბრუნებს true, თუ გამომძახებელი ობიექტი შეიცავს იმავე სტრიქონს, რასაც str. წინააღმდეგ შემთხვევაში აბრუნებს false-ს (J2SE 5)
boolean contentEquals(StringBuffer str)	აბრუნებს true, თუ გამომძახებელი ობიექტი შეიცავს იმავე სტრიქონს, რასაც str. წინააღმდეგ შემთხვევაში აბრუნებს false-ს (J2SE 5)
static String format (String fmtstr,	აბრუნებს სტრიქონს, რომელიც დაფორმატებულია fmtstr-ის

Object ... args)	მიხედვით (J2SE 5)
static String format (Locale loc, String fmtstr, Object ... args)	აბრუნებს სტრიქონს, რომელიც დაფორმატებულია fmtstr-ის მიხედვით (J2SE 5)
boolean matches(String regExp)	აბრუნებს true, თუ გამომძახებელი სტრიქონი შეესაბამება რეგულარულ გამოსახულებას, რომელიც გადმოცემულია regExp-ში. წინააღმდეგ შემთხვევაში აბრუნებს false-ს (J2SE 5)
int offsetByCodePoints(int start, int num)	აბრუნებს გამომძახებელი სტრიქონის ინდექსს, რომელიც იმყოფება num კოდის წერტილით დაშორებული იმ საწყისი ინდექსიდან, რომელიც მითითებულია start-ში (J2SE 5)
String replaceFirst(String regExp, String newStr)	აბრუნებს სტრიქონს, რომელშიც regExp რეგულარული გამოსახულების შესაბამისი პირველი ქვესტრიქონი, იცვლება newStr-ით
String replaceAll(String regExp, String newStr)	აბრუნებს სტრიქონს, რომელშიც regExp რეგულარული გამოსახულების შესაბამისი ყველა ქვესტრიქონი იცვლება newStr-ით
String[] split(String regExp)	გამომძახებელ სტრიქონს ყოფს ნაწილებად და შედეგად აბრუნებს

	დაყოფილი ნაწილების მასივს. თითოეული ნაწილი შემოფარგლულია რეგულარული regExp გამოსახულებით. ნაწილების რაოდენობა მითითებულია
String[] split(String regExp, int max)	გამომძახებელ სტრიქონს ყოფს ნაწილებად და შედეგად აბრუნებს დაყოფილი ნაწილების მასივს. თითოეული ნაწილი შემოფარგლულია რეგულარული regExp გამოსახულებით. ნაწილების რაოდენობა მითითებულია max-ში. თუ max უარყოფითია, ე.ი. საწყისი სტრიქონად სრულად იშლება. წინააღმდეგ შემთხვევაში, თუ max არაუარყოფითია, დაბრუნებული მასივის ბოლო ელემენტი შეიცავს გამომძახებელი სტრიქონის ნაშთს. თუ max ტოლია 0-ის, სტრიქონი სრულად იშლება
CharSequence subSequence(int startIndex, int stopIndex)	აბრუნებს გამომძახებელი სტრიქონის ქვესტრიქონს, დაწყებული startIndex-დან და დამთავრებული stopIndex-მდე. ეს მეთოდი მოითხოვს CharSequence ინტერფეისს, რომელსაც ახლა რეალიზებას უკეთებს String კლასი

კლასი StringBuffer

StringBuffer კლასი არის String-ის მსგავსი და სტრიქონების დამუშავების ბევრ მეთოდს გვთავაზობს. როგორც აღვნიშნეთ String-ი მუდმივი სიგრძისაა და წარმოადგენს სიმბოლოების უცვლელ მიმდევრობას. მისგან განსხვავებით StringBuffer წარმოადგენს გაფართოებად და სიმბოლოების მიმდევრობის ცვლილების შესაძლებლობის მქონე (შემძლე) კლასს. StringBuffer საშუალებას იძლევა ჩაემატოს სიმბოლოები და ქვესტრიქონები შუაში ან დაემატოს ისინი ბოლოში. StringBuffer-ი ავტომატურად იზრდება და ხშირად ამ ზრდადობის მისაღწევად სიმბოლოებისთვის გამოყოფილი აქვს უფრო მეტი ადგილი, ვიდრე ფაქტობრივადაა საჭირო მოცემულ მომენტში. Java-ში ინტენსიურად გამოიყენება ორივე კლასი, მაგრამ ბევრი პროგრამისტი მხოლოდ String-თან მუშაობს.

StringBuffer-ის კონსტრუქტორები

StringBuffer-სათვის განსაზღვრულია 4 კონსტრუქტორი:

StringBuffer ()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)

სტანდარტული კონსტრუქტორი (უპარამეტრო კონსტრუქტორი) არეზერვირებს ადგილს 16 სიმბოლოსათვის. კონსტრუქტორის მეორე ვერსია ღებულობს არგუმენტს, რომელიც ცხადად აყენებს ბუფერის ზომას. მესამე ვერსია

ღებულობს String ტიპის არგუმენტს, რომელიც აყენებს StringBuffer ობიექტის საწყის მნიშვნელობას და არეზერვირებს ადგილს დამატებითი 16 სიმბოლოსათვის. ასეთი დარეზერვირებები ხდება იმიტომ, რომ მეხსიერების განაწილების ოპერაცია ძვირადღირებული ოპერაციაა დროითი დანახარჯების მხრივ. გარდა ამისა, მეხსიერების ხელახალმა გამოყოფამ შეიძლება მეხსიერების ფრაგმენტაცია გამოიწვიოს. რამდენიმე დამატებითი სიმბოლოსთვის მეხსიერების გამოყოფით StringBuffer ამცირებს საჭირო ხელახალი განაწილებების რაოდენობას. მეოთხე კონსტრუქტორი ქმნის ობიექტს, რომელიც შედგება chars პარამეტრში შემავალი სიმბოლოების მიმდევრობისაგან.

Length() და capacity() მეთოდები

StringBuffer-ის მიმდინარე სიგრძე შეიძლება მივიღოთ length() მეთოდით, ხოლო გამოყოფილი მეხსიერების მიმდინარე ზომა capacity() მეთოდით. მათ აქვთ შემდეგი ზოგადი ფორმა:

int length()
int capacity ()
მაგალითი:

ლისტინგი 66. StringBuffer-ის length() და capacity() მეთოდების შედარება.

```
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer " + sb);
    System.out.println("length " + sb.length());
```

```
        System.out.println("capacity " +
                                        sb.capacity());
    }
}
```

პროგრამის შედეგია:

buffer = Hello
length = 5
capacity = 21

ვინაიდან sb შექმნისას ინიციალიზებულია Hello სტრიქონით, მისი სიგრძეა 5. გამოყოფილი მეხსიერების (capacity) ზომაა 21, ვინაიდან 16 დამატებითი სიმბოლოს ადგილი ემატება ავტომატურად.

მეთოდი ensureCapacity()

როდესაც გვინდა წინასწარ გამოვყოთ ადგილი გარკვეული რაოდენობის სიმბოლოებისათვის, მას შემდეგ რაც StringBuffer ობიექტი უკვე შექმნილია, შეგვიძლია ვისარგებლოთ ensureCapacity() მეთოდით. ეს მეთოდი აყენებს ბუფერის სიგრძეს. ასეთი გზა მოსახერხებელია, როდესაც წინასწარ ვიცით, რომ StringBuffer ობიექტს უნდა დაემატოს დიდი რაოდენობის მოკლე სტრიქონები. ensureCapacity() მეთოდის ზოგადი ფომაა:

void ensureCapacity(int capacity)

capacity მიუთითებს ბუფერის ზომას.

მეთოდი setLength ()

StringBuffer-ის ობიექტის შიგნით ბუფერის სიგრძის მნიშვნელობის დასაყენებლად გამოიყენება მეთოდი setLength(). ამ მეთოდის ზოგადი ფორმაა:

void setLength(int len)

len მიუთითებს ბუფერის სიგრძეს და მისი მნიშვნელობა არ უნდა იყოს უარყოფითი.

როდესაც ბუფერის სიგრძეს ვზრდით, არსებული ბუფერის ბოლოს ემატება ე.წ. ნულოვანი სიმბოლოები. თუ გამოვიძახებთ setLength() მეთოდს ისეთი მნიშვნელობით, რომელიც ნაკლებია იმ მიმდინარე მნიშვნელობაზე, რომელსაც აბრუნებს length() მეთოდი, მაშინ სიმბოლოები, რომლებიც ახლად დაყენებული სიგრძის გარეთ იმყოფებიან, დაიკარგება.

მეთოდები charAt() და setCharAt()

StringBuffer-იდან შესაძლებელია ცალკეული სიმბოლოს მნიშვნელობის ამოღება charAt() მეთოდის საშუალებით. StringBuffer-ში სიმბოლოს მნიშვნელობის მინიჭება შესაძლებელია setCharAt() მეთოდით. ამ მეთოდების ზოგადი ფორმა ასეთია:

char charAt(int where)
void setCharAt(int where, char ch)

charAt() მეთოდში პარამეტრი where მიუთითებს ინდექსს, რომელიც უნდა იქნას ამოღებული. setCharAt() მეთოდში

პარამეტრი where მიუთითებს იმ სიმბოლოს ინდექსს, სადაც უნდა მოხდეს მნიშვნელობის მინიჭება, ხოლო ch ესაა მისანიჭებელი მნიშვნელობა. ორივე მეთოდისათვის where უნდა იყოს არაუარყოფითი და არ უნდა იმყოფებოდეს ბუფერის საზღვარს გარეთ. მაგალითი:

ლისტინგი 67. charAt() და setCharAt() მეთოდები

```
class setCharAtDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("Hello");
      System.out.println("ბუფერი თავიდან =" + sb);
      System.out.println("charAt(1)-მდე = " +
                                        sb.charAt(1));
      sb.setCharAt(1, 'i');
      sb.setLength(2);
      System.out.println("ბუფერი  შემდეგ = " + sb);
      System.out.println("charAt(1)-ს შემდეგ = " +
                                        sb.charAt(1));
   }
}
```

ამ პროგრამის შედეგი:

ბუფერი თავიდან = Hello
charAt(1)-მდე = e
ბუფერი შემდეგ = Hi
charAt(1)-ს შემდეგ = i

მეთოდი getChars()

StringBuffer-დან ქვესტრიქონის მასივში კოპირებისათვის, გამოიყენება getChars() მეთოდი. მისი ზოგადი ფორმაა:

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

sourceStart მიუთითებს ქვესტრიქონის დასაწყისის ინდექსს, ხოლო sourceEnd იმ სიმბლოს ინდექსია, რომელიც საჭირო ქვესტრიქონის შემდეგ დგას. ეს ნიშნავს, რომ ქვესტრიქონი შეიცავს სიმბოლოებს sourceStart-დან sourceEnd-1 -მდე. მასივი, რომელიც სიმბოლოებს იღებს, მოიცემა target-ით. target-ის შიგნით ის ინდექსი, სადაც კოპირდება ქვესტრიქონი, მოცემულია targetStart პარამეტრით. პროგრამისტმა უნდა იზრუნოს იმაზე, რომ მასივი target-ი იყოს საკმარისი სიგრძის, რათა მასში ჩაეტიოს ქვესტრიქონის სიმბოლოები.

მეთოდი append()

მეთოდი append() აერთიანებს ნებისმიერი სხვა ტიპის მონაცემის სტრიქონულ წარმოდგენას გამომძახებელი StringBuffer ობიექტის ბოლოსთან. მას აქვს რამდენიმე გადატვირთული მეთოდი, მაგალითად:

StringBuffer append(String str)

StringBuffer append(int num)

StringBuffer append (Object obj)

ამ მეთოდებში თითოეული პარამეტრისათვის ხდება String.valueOf()-ის გამოძახება, ვინაიდან მიღებული იქნას მისი სტრიქონული წარმოდგენა. შედეგი ემატება StringBuffer-ის მიმდინარე ობიექტს. თვით ბუფერი ბრუნდება append-ის ყოველი ვერსიის შედეგად. ეს საშუალებას იძლევა გავაერთიანოთ რამდენიმე

მიმდევრობითი გამოძახება ერთად, როგორც ეს შემდეგ მაგალითშია ნაჩვენები:

ლისტინგი 68. append()-ის დემონსტრირება

```
class appendDemo {
   public static void main(String args[]) {
      String s;
      int a = 42;
      StringBuffer sb = new StringBuffer(40);
      s = sb.append("a = ").append(a).append(";")
                                          .toString();
      System.out.println(s);
   }
}
```

ამ პროგრამის შედეგი:

a = 42;

მეთოდი append() უფრო ხშირად გამოიძახება, როდესაც ხდება + ოპერაციის გამოყენება String ობიექტებზე. ამ დროს Java ავტომატურად ცვლის String ეგზემპლარების მოდიფიკაციებს StringBuffer-ის ეგზემპლარების შესაბამისი ოპერაციებით. ამრიგად, კონკატენაცია იწვევს StringBuffer ობიექტის append() მეთოდის გამოძახებას. კონკატენაციის ოპერაციის შესრულების შემდეგ, კომპილატორი სვამს toString()-ის გამოძახებას, ვინაიდან მოახდინოს StringBuffer ობიექტის უკან, String ობიექტად გარდაქმნა. შეიძლება ეს რთულად მოგვეჩვენოს და დაისვას ლეგიტიმური კითხვა: ხომ არ ჯობდა ყოფილიყო ერთი კლასი, რომელსაც ექნებოდა StringBuffer-ის შესაძლებლობები? ამ კითხვაზე პასუხია - არა, წარმადობის გამო. ამ თემაზე ჩვენ ზემოთაც ვიმსჯელეთ. არსებობს ოპტიმიზაციის მრავალი ხერხი, რომელსაც Java-ს შემსრულებელი გარემო იყენებს, ოღონდ

იმის გათვალისწინებით, რომ String ობიექტი უცვლელია. უნდა აღინიშნოს, რომ საბედნიეროდ Java ფარავს თითქმის ყველა იმ სირთულეს, რაც String და StringBuffer ტიპებს შორის გარდაქმნებთანაა დაკავშირებული.

მეთოდი insert()

insert() მეთოდი ერთ სტრიქონს სვამს მეორეში. იგი გადატვირთულია, რადგან მისმა პარამეტრმა მიიღოს ყველა შესაძლო მარტივი ტიპი და String, Object და CharSequence ობიექტები. append()-ის მსგავსად იგი მიმართავს String.valueOf()-ს სტრიქონული წარმოდგენის მისაღებად. შემდეგ ეს სტრიქონი ჩაისმება გამომძახებელ StringBuffer ობიექტში. ამ მეთოდის ფორმებია:

StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)

index აქ მიუთითებს გამომძახებელი StringBuffer ობიექტის იმ პოზიციის ინდექსს, სადაც ჩაისმება სტრიქონი. მაგალითი:

ლისტინგი 69. insert()-ის დემონსტრირება

```
class insertDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("I like
                                          Java!");
      System.out.println(sb);
   }
}
```

ამ პროგრამის შედეგი:

I like Java!

მეთოდი reverse()

StringBuffer ობიექტში სიმბოლოების მიმდევრობის მისი შებრუნებულით შეცვლა შესაძლებელია reverse() მეთოდის საშუალებით:

StringBuffer reverse()

ეს მეთოდი აბრუნებს ობიექტს, რომელიც შეიცავს მისი გამომძახებელი ობიექტის სიმბოლოების მიმდევრობის შებრუნებულ ვარიანტს. მაგალითი:

ლისტინგი 70. reverse() მეთოდი StringBuffer ობიექტისათვის

```
class ReverseDemo {
   public static void main(String args[]) {
      StringBuffer s = new StringBuffer("abcdef");
      System.out.println(s);
      s.reverse();
      System.out.println(s);
   }
}
```

შედეგი:

abcdef
fedcba

მეთოდები delete() და deleteCharAt()

StringBuffer-იდან სიმბოლოების ამოსაგდებად (წასაშლელად) შეიძლება გამოვიყენოთ მეთოდები delete() და deleteCharAt():

StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)

მეთოდი delete() წაშლის სიმბოლოების მიმდევრობას გამომძახებელი ობიექტიდან. startIndex მიუთითებს პირველი სიმბოლოს ინდექსს, რომლიც უნდა წაიშალოს, ხოლო endIndex ბოლო წასაშლელი სიმბოლოს შემდეგია. ამრიგად წასაშლელი ქვესტრიქონი იწყება startIndex-დან და მთავრდება endIndex-1 -ით. ბრუნდება შედეგად მიღებული StringBuffer ობიექტი.

მეთოდი deleteCharAt() წაშლის სიმბოლოს, რომელიც იმყოფება loc პოზიციაში. ბრუნდება შედეგად მიღებული StringBuffer ობიექტი. მაგალითი:

ლისტინგი 71. delete() და deleteCharAt() მეთოდების დემონსტრირება

```
class deleteDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("This is a
                                          test.");
      sb.delete(4, 7);
      System.out.println("delete-ს შემდეგ : " + sb);
      sb.deleteCharAt(0);
      System.out.println("deleteCharAt-ის შემდეგ : "
                                          + sb);
   }
}
```

პროგრამის შედეგი:

delete-ს შემდეგ: This a test.
deleteCharAt-ს შემდეგ: his a test.

მეთოდი replace()

StringBuffer-ის შიგნით სიმბოლობის ერთი მიმდევრობა შეიძლება შევცვალოთ მეორეთი. ამისათვის გამოიყენება მეთოდი replace(). ამ მეთოდს ასეთი სიგნატურა აქვს:

StringBuffer replace(int startIndex, int endIndex, String str)

შესაცვლელი ქვესტრიქონი მოიცემა startIndex და endIndex ინდექსებით. ამრიგად შეიცვლება ქვესტრიქონი startIndex პოზიციაში მყოფი სიმბოლოდან endIndex-1 სიმბოლომდე. შემცვლელი სტრიქონი მითითებულია str-ით. ბრუნდება შედეგად მიღებული StringBuffer ობიექტი. მაგალითი:

ლისტინგი 72. replace()-ის დემონსტრირება

```
class replaceDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("This is a
                                          test.");
      sb.replace(5, 7, "was");
      System.out.println("შეცვლის შემდეგ: " + sb);
   }
}
```

პროგრამის შედეგი:

შეცვლის შემდეგ: This was a test.

მეთოდი substring()

substring() მეთოდის გამოძახებით შეგვიძლია StringBuffer-ის ნაწილი მივიღოთ. ამ მეთოდს ასეთი ფორმები აქვს:

String substring(int startIndex)
String substring(int startIndex, int endIndex)

პირველი ფორმა აბრუნებს ქვესტრიქონს, რომელიც იწყება startIndex-დან და გრძელდება გამომძახებელი StringBuffer-ის ობიექტის ბოლომდე. მეორე ფორმა აბრუნებს ქვემიმდევრობას startIndex-დან endIndex-1 -მდე. ეს მეთოდები ისევე მუშაობენ, როგორც ზემოთ აღწერილი მათი ანალოგები String ობიექტებში.

აღწერილი მეთოდების გარდა StringBuffer კლასი ფლობს კიდევ ბევრ სხვა მეთოდს. ზოგიერთი მათგანი ჩამოთვლილია 7 ცხრილში. აქვე მითითებულია ის მეთოდები, რომლებიც j2SE 5-ში დაემატა.

ცხრილი 7.

მეთოდი	აღწერა
StringBuffer appendCodePoint (int ch)	ამატებს Unicode-ს კოდის წერტილს გამომძახებელი ობიექტის ბოლოში. (J2SE 5)
int codePointAt (int i)	აბრუნებს i პოზიციაში მყოფი Unicode სიმბოლოს კოდის წერტილს (J2SE 5)
int codePointBefore (int i)	აბრუნებს i პოზიციის წინა

	პოზიციაზე მყოფი Unicode სიმბოლოს კოდის წერტილს (J2SE 5)
int codePointCount (int start, int end)	აბრუნებს გამომძახებელი სტრიქონის კოდის წერტილების რაოდენობას start-დან end-1 მდე (J2SE 5)
int indexOf (String str)	გამომძახებელ StringBuffer ობიექტში ხორციელდება ძებნა str-ის პირველ შეხვედრამდე. თუ მოიძებნა აბრუნებს დამთხვევის პოზიციის ინდექსს, წინააღმდეგ შემთხვევაში -1-ს
int indexOf (String str, int startIndex)	გამომძახებელ StringBuffer ობიექტში ხორციელდება ძებნა str-ის პირველ შეხვედრამდე დაწყებული startIndex პოზიციიდან. თუ მოიძებნა აბრუნებს დამთხვევის პოზიციის ინდექსს, წინააღმდეგ შემთხვევაში -1-ს
int lastIndexOf (String str)	გამომძახებელ StringBuffer ობიექტში ხორციელდება ძებნა str-ის ბოლო შეხვედრამდე. თუ მოიძებნა აბრუნებს დამთხვევის პოზიციის ინდექსს, წინააღმდეგ შემთხვევაში -1-ს

int lastIndexOf (String str, int startIndex)	გამომძახებელ StringBuffer ობიექტში ხორციელდება ძებნა str-ის ბოლო შეხვედრამდე დაწყებული startIndex-დან. თუ მოიძებნა აბრუნებს დამთხვევის პოზიციის ინდექსს, წინააღმდეგ შემთხვევაში -1-ს
int offsetByCodePoints (int start, int num)	აბრუნებს გამომძახებელი სტრიქონში იმ სიმბოლოს ინდექსს, რომელიც იმყოფება num კოდის წერტილებით უკან საწყის start ინდექსთან შედარებით (J2SE 5)
CharSequence subSequence (int startIndex, int stopIndex)	აბრუნებს გამომძახებელის ქვესტრიქონს, დაწყებული startIndex-დან დამთავრებული stopIndex-მდე. ეს მეთოდი ითხოვს CharSequence ინტერფეისს, რომლის რეალიზაციასაც StringBuffer ახდენს.
void trimToSize ()	გამომძახებელი ობიექტის ბუფერს ამცირებს იმდენად, რომ იგი შეესაბამებოდეს მიმდინარე შემცველობას (J2SE 5)

subSequence() მეთოდის გარდა, რომელიც CharSequence ინტერფეისის მეთოდის რეალიზაციას წარმოადგენს, ამ ცხრილის სხვა მეთოდები StringBuffer-ს საშუალებას აძლევს მოახდინოს String-ის ძებნა ბუფერში. მაგალითი:

ლისტინგი 73.

```
class IndexOfDemo {
   public static void main(String args[]) {
      StringBuffer sb = new StringBuffer("one two one");
      int i;
      i = sb.indexOf("one");
      System.out.println("პირველი შეხვედრის ინდექსი:
         "+i);
      i = sb.lastIndexOf("one");
      System. out .println ("ბოლო შეხვედრის ინდექსი:" +
         i);
   }
}
```

შედეგი:

პირველი შეხვედრის ინდექსი: 0
ბოლო შეხვედრის ინდექსი: 8

StringBuilder კლასი

სტრიქონების დამუშავების არსებულ მდიდარ საშუალებებს J2SE 5-ში დაემატა სტრიქონებთან მუშაობის ახალი კლასი StringBuilder. იგი StringBuffer-ის ანალოგიურია, ოღონდ აქვს ერთი მნიშვნელოვანი განსხვავება: იგი არაა სინქრონიზებული, რაც ნიშნავს, რომ იგი არაა უსაფრთხო ნაკადების მიმართ. StringBuilder კლასის გამოყენების უპირატესობაა მისი მწარმოებლობა. მაგრამ მრავალ

ნაკადიანი პროგრამების დამუშავებისას უნდა მოხდეს StringBuffer კლასის გამოყენება და არა StringBuilder-სა.

გარსი კლასები

როგორც ადრე ავღნიშნეთ java.lang პაკეტი შეიცავს კლასებსა და ინტერფეისებს, რომლებიც ფუნდამენტალურებია Java-ზე დაწერილი ყველა პროგრამისათვის. ეს პაკეტი ყველაზე უფრო ფართედ გამოყენებადია და იგი ავტომატურად იმპორტირდება ყველა პროგრამაში.

java.lang პაკეტი შეიცავს შემდეგ კლასებს:

Boolean	InheritableThreadLocal
Byte	ProcessBuilder
Character	Runtimepermission
Class	SecurityManager
ClassLoader	Short
Compiler	StackTraceElement
Double	StrictMath
Enum	String
Float	StringBuffer
Integer	StringBuilder
Long	System
Math	Thread
Number	ThreadGroup
Object	ThreadLocal
Package	Throwable
Process	Void
Runtime	

java.lang პაკეტში განსაზღვრულია შემდეგი ინტერფეისები:

Appendable	Iterable
CharSequence	Readable
Cloneable	Runnable
Comparable	

ამ პაკეტში ჩართული ზოგიერთი კლასი შეიცავს მოძველებულ მეთოდებს, რომელთა უმრავლესობა ეკუთვნოდა ჯერ კიდევ Java 1.0-ს. ეს ძველი მეთოდები ჯერ-ჯერობით ისევ შენარჩუნებულია მაგრამ არაა რეკომენდებული მათი გამოყენება ახალ პროგრამებში. ძველ მეთოდებს ჩვენ არ განვიხილავთ.

როგორც სასწავლო კურსის თავში აღვნიშნეთ, Java იყენებს პრიმიტიულ ტიპებს, როგორიცაა int, char და სხვას, წარმადობის მოსაზრებით. მონაცემთა ეს ტიპები არ წარმოადგენენ ობიექტური იერარქიის ნაწილს. ისინი მეთოდებში გადაიცემა მნიშვნელობის მიხედვით და არ შეიძლება მათი მიმთითებლით გადაცემა. ასევე არაა შესაძლებელი ორი მეთოდი მიუთითებდეს მაგალითად int-ის ერთიდაიგივე ეგზემპლარს. ხშირად აუცილებელი ხდება პრიმიტიული ტიპის ობიექტური წარმოდგენის შესაძლებლობა. მაგალითად, არსებობს კლასი-კოლექციები, რომლებიც მხოლოდ ობიექტებთან მუშაობენ. პრიმიტიული ტიპი რომ რომელიმე ასეთ კლასში მოვათავსოთ (შევინახოთ), საჭიროა პრიმიტიული ტიპი შევფუთოთ კლასის გარსით. ამ აუცილებლობის დასაკმაყოფილებლად, Java გვთავაზობს კლასებს, რომლებიც შეესაბამებიან თითოეულ პრიმიტიულ ტიპს. შინაარსობრივად, ეს კლასები

ახდენენ პრიმიტიული ტიპების ინკაფსულირებას კლასებში. ანუ, პრიმიტიულ ტიპებს ათავსებენ კლასების გარსებში, ამიტომ მათ ხშირად უწოდებენ ტიპების გარსებს. განვიხილოთ ტიპების გარსები ანუ გარსი კლასები უფრო დეტალურად.

აბსტრაქტული კლასი Number

აბსტრაქტული კლასი Number ესაა სუპერკლასი, რომელიც რეალიზებულია byte, short, int, long, float და double მარტივი რიცხვითი ტიპების გარსი კლასებით. Number-ს აქვს აბსტრაქტული მეთოდები, რომლებიც ობიექტისგან აბრუნებენ მნიშვნელობას ყველა რიცხვით ფორმატში. მაგალითად doubleValue() აბრუნებს double ტიპის მნიშვნელობას, floatValue() აბრუნებს float ტიპს და ა.შ. ეს მეთოდებია:

byte byteValue()
double doubleValue()
float floatValue()
int intValue ()
long longValue()
short shortValue()

ამ მეთოდების მიერ დაბრუნებული მნიშვნელობები შეიძლება იყოს დამრგვალებული.

Number-ს აქვს 6 კონკრეტული ქვეკლასი, რომლებიც შეიცავენ თითოეული რიცხვითი ტიპის ცხად მნიშვნელობას. ესენია: Double, Float, Byte, Short, Integer და Long.

Double და Float

Double და Float ესაა გარსები double და float მცოცავმძიმიანი ტიპების მნიშვნელობებისათვის. ქვემოთ ნაჩვენებია Float კლასის კონსტრუქტორები:

Float(double num)
Float(float num)
Float(String str) throws NumberFormatException

Float ობიექტები კონსტრუირებული უნდა იქნას float ან double ტიპის მნიშვნელობებისაგან. მათი კონსტრუირება ასევე შესაძლებელია მცოცავმძიმიანი რიცხვების სტრიქონული წარმოდგენისაგან.

Double-ის კონსტრუქტორებს ასეთი სახე აქვს:

Double(double num)
Double(String str) throws NumberFormatException

Double ობიექტები შეიძლება შეიქმნას double ტიპის ან მცოცავმძიმიანი რიცხვების სტრიქონული წარმოდგენის მნიშვნელობებისაგან.

Float და Double კლასებში განსაზღვრულია შემდეგი კონსტანტები:

MAX_EXPONENT	მაქსიმალური ექსპონენტა (java SE 6)
MAX_VALUE	მაქსიმალური დადებითი მნიშვნელობა
MIN_EXPONENT	მინიმალური ექსპონენტა (java SE 6)
MIN_NORMAL	მინ. დადებ. ნორმალური მნიშვნ.(java SE 6)
MIN_VALUE	მინიმალური დადებითი მნიშვნელობა
NaN	არაა რიცხვითი მნიშვნელობა

POSITIVE_INFINITY	დადებითი უსასრულობა
NEGATIVE_INFINITY	უარყოფითი უსასრულობა
SIZE	გარსში მოთავსებულის ზომა ბიტებში
TYPE	Class ობიექტი float და double-სთვის

ცხრილში 8. აღწერილია Float კლასის მეთოდები

მეთოდი	აღწერა
byte byteValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც byte-ს
static int compare (float num1, float num2)	ადარებს num1 და num2-ის მნიშვნელობებს. აბრუნებს 0-ს, თუ მნიშვნელობები ტოლია; აბრუნებს უარყოფით მნიშვნელობას, თუ num1<num2; დადებითს, თუ num1>num2
int compareTo (Float f)	გამომძახებელი ობიექტის რიცხვით მნიშვნელობას ადარებს f-ს. აბრუნებს: 0-ს, თუ მნიშვნელობები ტოლია; უარყოფით მნიშვნელობას, თუ გამომძახებლის მნიშვნელობა ნაკლებია; დადებით მნიშვნელობას, თუ გამომძახებელის მნიშვნელობა მეტია
double doubleValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც double-ს
boolean equals(Object FloatObj)	აბრუნებს true-ს, თუ გამომძახებელი Float ობიექტი ეკვივალენტურია

	FloatObj-ის. წინააღმდეგ შემთხვევაში ბრუნდება false
static int floatToIntBits(float num)	აბრუნებს num-ის შესაბამის, IEEE-სთან თავსებად ერთმაგი სიზუსტის ბიტურ შაბლონს
static int floatToRawIntBits(float num)	აბრუნებს num-ის შესაბამის, IEEE-სთან თავსებად ერთმაგი სიზუსტის ბიტურ შაბლონს. NaN-სგან დაცულია
float floatValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც float-ს
int hashCode ()	აბრუნებს გამომძახებელი ობიექტის ჰეშ კოდს
static float intBitsToFloat(int num)	აბრუნებს num-ის შესაბამის float-ეკვივალენტს, რომლის ერთმაგი სიზუსტის ბიტურ შაბლონი IEEE-სთან თავსებადია
int intValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც int-ს
boolean isInfinite()	აბრუნებს true-ს, თუ გამომძახებელი ობიექტის მნიშვნელობაა უსასრულობა. წინააღმდეგ შემთხვევაში false
static boolean isInfinite (float num)	აბრუნებს true-ს, თუ num განსაზღვრავს უსასრულობას. წინააღმდეგ შემთხვევაში false

boolean isNaN ()	აბრუნებს true-ს, თუ გამომძახებელი ობიექტის მნიშვნელობა არაა რიცხვითი. წინააღმდეგ შემთხვევაში false
static boolean isNaN (float num)	აბრუნებს true-ს, თუ num არაა რიცხვითი მნიშვნელობა. წინააღმდეგ შემთხვევაში false
long longValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც long-ს
static float parseFloat(String str) throws NumberFormatException	აბრუნებს str სტრიქონში ჩაწერილი ათობითი რიცხვის float ეკვივალენტს.
short shortValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც short-ს
static String toHexString(float num)	აბრუნებს სტრიქონს, რომელიც წარმოადგენს num-ის მნიშვნელობას თექვსმეტობითში (J2SE 5)
String toString ()	აბრუნებს გამომძახებელი ობიექტის სტრიქონულ ეკვივალენტს
static String toString (float num)	აბრუნებს num-ის სტრიქონულ ეკვივალენტს
static Float valueOf (float num)	აბრუნებს Float ობიექტს, რომელიც შეიცავს num-ის მნიშვნელობას (J2SE 5)
static Float valueOf(String str) throws	აბრუნებს Float ობიექტს, რომელიც შეიცავს str-ში მითითებულ

NumberFormatException	მნიშვნელობას

ცხრილში 9. აღწერილია Double კლასის მეთოდები

მეთოდი	აღწერა
byte byteValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც byte-ს
static int compare (double num1, double num2)	ადარებს num1 და num2-ის მნიშვნელობებს. აბრუნებს 0-ს, თუ მნიშვნელობები ტოლია; აბრუნებს უარყოფით მნიშვნელობას, თუ num1<num2; დადებითს, თუ num1>num2
int compareTo (Double d)	გამომძახებელი ობიექტის რიცხვით მნიშვნელობას ადარებს d-ს. აბრუნებს: 0-ს, თუ მნიშვნელობები ტოლია; უარყოფით მნიშვნელობას, თუ გამომძახებლის მნიშვნელობა ნაკლებია; დადებით მნიშვნელობას, თუ გამომძახებელის მნიშვნელობა მეტია
static long doubleToLongBits(double num)	აბრუნებს num-ის შესაბამის, ორმაგი სიზუსტის IEEE-სთან თავსებად ბიტურ შაბლონს
static long doubleToRawLongBits (double num)	აბრუნებს num-ის შესაბამის, ორმაგი სიზუსტის IEEE-სთან თავსებად ბიტურ შაბლონს. NaN

	მნიშვნელობისგან დაცულია
double doubleValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც double-ს
boolean equals(Object DoubleObj)	აბრუნებს true-ს, თუ გამომძახებელი Double ობიექტი ეკვივალენტურია DoubleObj-ის. წინააღმდეგ შემთხვევაში ბრუნდება false
float floatValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც float-ს
int hashCode ()	აბრუნებს გამომძახებელი ობიექტის ჰეშ კოდს
int intValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც int-ს
boolean isInfinite()	აბრუნებს true-ს, თუ გამომძახებელი ობიექტის მნიშვნელობაა უსასრულობა. წინააღმდეგ შემთხვევაში false
static boolean isInfinite(double num)	აბრუნებს true-ს, თუ num განსაზღვრავს უსასრულობას. წინააღმდეგ შემთხვევაში false
boolean isNaN ()	აბრუნებს true-ს, თუ გამომძახებელი ობიექტის მნიშვნელობა არაა რიცხვითი. წინააღმდეგ შემთხვევაში false
static boolean isNaN (double num)	აბრუნებს true-ს, თუ num არაა რიცხვითი მნიშვნელობა.

	წინააღმდეგ შემთხვევაში false
static double longBitsToDouble(long num)	აბრუნებს num-ის შესაბამის, double ეკვივალენტს, რომელიც IEEE-სთან თავსებადი ბიტური შაბლონია
long longValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც long-ს
static double parseDouble(String str) throws NumberFormatException	აბრუნებს str-ში ჩაწერილი რიცხვის 10-ბით double ეკვივალენტს
short shortValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც short-ს
static String toHexString(double num)	აბრუნებს num-ში ჩაწერილი რიცხვის 16-მნიშვნელობას სტრიქონულ ფორმატში
String toString ()	აბრუნებს გამომძახებელი ობიექტის სტრიქონულ ეკვივალენტს
static String toString(double num)	აბრუნებს num-ში ჩაწერილი რიცხვის სტრიქონულ ეკვივალენტს
static Double valueOf(double num)	აბრუნებს num-ით გადაცემული მნიშვნელობის შემცველ Double ობიექტს
staticDouble valueOf(String str) throws NumberFormatException	აბრუნებს str-ით გადაცემული მნიშვნელობის შემცველ Double ობიექტს

შემდეგ მაგალითში იქმნება ორი Double ობიექტი, ერთი double მნიშვნელობის გამოყენებით, ხოლო მეორე იმ სტრიქონის გამოყენებით, რომლის ინტერპრეტირებაც შესაძლებელია, როგორც double ტიპის რიცხვი:

ლისტინგი 74.

```
class Doubledemo {
   public static void main(String args[]) {
      Double d1 = new Double(3.14159);
      Double d2 = new Double("314159E-5");
      System.out.println(d1 + " = " + d2 + " ->" +
                                          d1.equals(d2));
   }
}
```

ამ პროგრამის შედეგია:

3.14159 = 3.14159 -> true

როგორც ამ პროგრამის შედეგიდან ჩანს, ორივე კონსტრუქტორი ქმნის იდენტურ Double ეგზემპლარებს, რასაც ამტკიცებს equals() მეთოდი, რომლიც აბრუნებს true-ს.

მეთოდები isInfinite() და isNaN()

Double და Float კლასები გვთავაზობენ მეთოდებს isInfinite() და isNaN(), რომლებიც გვეხმარება double და float ტიპის ორ სპეციალურ მნიშვნელობასთან მუშაობაში. ეს მეთოდები ახდენენ ორი უნიკალური მნიშვნელობის ტოლობაზე შემოწმებას. უსასრულობა და NaN (არა რიცხვითი მნიშვნელობა) მნიშვნელობები წარმოადგენენ მცოცავ მძიმიან სპეციფიკაციებს, რომლებიც განსაზღვრულია IEEE სტანდარტში. isInfinite() აბრუნებს true-ს, თუ შესამოწმებელი

რიცხვი უსასრულოდ დიდია ან უსასრულოდ მცირეა. isNaN() მეთოდი აბრუნებს true-ს, თუ შესამოწმებელი მნიშვნელობა რიცხვითი არაა. მაგალითი:

ლისტინგი 75. isInfinite() და isNaN() მეთოდები:

```
class InfNaN {
   public static void main(String args[]) {
      Double d1 = new Double(1 / 0.);
      Double d2 = new Double(0 / 0.);
      System.out.println(d1 + ": " + d1.isInfinite()
                                   + ", " + d1.isNaN());
      System.out.println(d2 + ": " + d2.isInfinite()
                                   + "," + d2.isNaN());
   }

}
```

პროგრამის შედეგია:

Infinity: true, false
NaN: false, true

კლასები Byte, Short, Integer და Long

კლასები Byte, Short, Integer და Long არის გარსი კლასები შესაბამისად byte, short, int და long მარტივი ტიპებისთვის. მათი კონსტრუქტორებია:

Byte(byte num)
Byte(String str) throws NumberFormatException
Short(short num)
Short(String str) throws NumberFormatException
Integer(int num)

Integer(String str) throws NumberFormatException
Long (long num)
Long(String str) throws NumberFormatException

როგორც ხედავთ ეს რიცხვები შესაძლებელია აიგოს, როგორც რიცხვითი მნიშვნელობისგან, ისე სტრიქონული გამოსახულებისაგან, ოღონდ დაცული უნდა იყოს რიცხვის ჩაწერის დასაშვები ფორმები.

ამ კლასებში განსაზღვრული მეთოდები ნაჩვენებია 16.3-დან 16.6-მდე ცხრილებში. ისინი შიცავენ მეთოდებს მთელი რიცხვების სტრიქონულად გარდასაქმნელად და, პირიქით, სტრიქონიდან მთელად გარდასაქმნელად. ამ მეთოდების ვარიაციებში შესაძლებელია მივუთითოთ radix - რიცხვის წარმოდგენის ფუძე. უფრო ხშირად გამოიყენება 2-ის ფუძე ორობითისათვის, 8 - რვაობითისათვის, 10 - ათობითისათვის და 16 - თექვსმეტობითი რიცხვებისათვის.

ამ კლასებში განსაზღვრულია შემდეგი კონსტანტები:

MIN_VALUE	მინიმალური მნიშვნელობა
MAX_VALUE	მაქსიმალური მნიშვნელობა
SIZE	გარსში მოთავსებული მნიშვნელობის ზომა
TYPE	byte, short, int ან long-თვის Class ობიექტი

ცხრილი 10. Byte კლასის მეთოდები

მეთოდი	აღწერა
byte byteValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც byte-ს

int compareTo (Byte b)	გამომძახებელი ობიექტის რიცხვით მნიშვნელობას ადარებს b-ს. აბრუნებს: 0-ს, თუ მნიშვნელობები ტოლია; უარყოფით მნიშვნელობას, თუ გამომძახებლის მნიშვნელობა ნაკლებია; დადებით მნიშვნელობას, თუ გამომძახებელის მნიშვნელობა მეტია
static Byte decode (String str) throws NumberFormatException	აბრუნებს Byte ობიექტს, რომელიც შეიცავს str-ში მითითებულ მნიშვნელობას
double doubleValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც double-ს
boolean equals(Object ByteObj)	აბრუნებს true-ს, თუ გამომძახებელი Byte ობიექტი ეკვივალენტურია ByteObj-ის. წინააღმდეგ შემთხვევაში ბრუნდება false
float floatValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც float-ს
int hashCode ()	აბრუნებს გამომძახებელი ობიექტის ჰეშ კოდს
int intValue ()	აბრუნებს გამომძახებელი

	ობიექტის მნიშვნელობას, როგორც int-ს
long longValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც long-ს
static byte parseByte(String str) throws NumberFormatException	აბრუნებს str-ში ჩაწერილი რიცხვის 10-ბით byte ეკვივალენტს
static byte parseByte (String str, int radix) throws NumberFormatException	აბრუნებს str-ში radix ფუძით ჩაწერილი რიცხვის byte ეკვივალენტს
short shortValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც short-ს
static String toHexString(byte num)	აბრუნებს num-ში ჩაწერილი რიცხვის 16-ით მნიშვნელობას სტრიქონულ ფორმატში
String toString ()	აბრუნებს გამომძახებელი ობიექტის სტრიქონულ ეკვივალენტს
static String toString(byte num)	აბრუნებს num-ში ჩაწერილი რიცხვის სტრიქონულ ეკვივალენტს
static Byte valueOf(byte num)	აბრუნებს num-ით გადაცემული მნიშვნელობის შემცველ Byte ობიექტს

static Byte valueOf (String str) throwsNumberFormatException	აბრუნებს str-ით გადაცემული მნიშვნელობის შემცველ Byte ობიექტს
static Byte valueOf (String str, int radix) throws NumberFormatException	აბრუნებს str-ით radix ფუძით გადაცემული მნიშვნელობის შემცველ Byte ობიექტს

ცხრილი 11. Short კლასის მეთოდები

მეთოდი	აღწერა
byte byteValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც byte-ს
int compareTo (Short b)	გამომძახებელი ობიექტის რიცხვით მნიშვნელობას ადარებს b-ს. აბრუნებს: 0-ს, თუ მნიშვნელობები ტოლია; უარყოფით მნიშვნელობას, თუ გამომძახებლის მნიშვნელობა ნაკლებია; დადებით მნიშვნელობას, თუ გამომძახებელის მნიშვნელობა მეტია
static Short decode (String str) throwsNumberFormatException	აბრუნებს Short ობიექტს, რომელიც შეიცავს str-ში მითითებულ მნიშვნელობას
double doubleValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც

	double-ს
boolean equals(Object ShortObj)	აბრუნებს true-ს, თუ გამომძახებელი Short ობიექტი ეკვივალენტურია ShortObj-ის. წინააღმდეგ შემთხვევაში ბრუნდება false
float floatValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც float-ს
int hashCode ()	აბრუნებს გამომძახებელი ობიექტის ჰეშ კოდს
int intValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც int-ს
long longValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც long-ს
static short parseShort(String str) throws NumberFormatException	აბრუნებს str-ში ჩაწერილი რიცხვის 10-ბით ეკვივალენტს
static short parseShort(String str, int radix) throws NumberFormatException	აბრუნებს str-ში radix ფუძით ჩაწერილი რიცხვის short ეკვივალენტს
static short reverseBytes (short num)	ადგილებს უცვლის num-ის უფროს და უმცროს ბაიტებს და შედეგს აბრუნებს

short shortValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც short-ს
String toString ()	აბრუნებს გამომძახებელი ობიექტის სტრიქონულ ეკვივალენტს
static String toString(short num)	აბრუნებს num-ში ჩაწერილი რიცხვის სტრიქონულ ეკვივალენტს
static Short valueOf(short num)	აბრუნებს num-ით გადაცემული მნიშვნელობის შემცველ Short ობიექტს
static Short valueOf (String str) throwsNumberFormatException	აბრუნებს str-ით გადაცემული მნიშვნელობის შემცველ Short ობიექტს
static Short valueOf (String str, int radix) throws NumberFormatException	აბრუნებს str-ით radix ფუძით გადაცემული მნიშვნელობის შემცველ Short ობიექტს

ცხრილი 12. Integer კლასის მეთოდები

მეთოდი	აღწერა
static int bitCount(int num)	აბრუნებს ბიტების რაოდენობას num-ში
byte byteValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც byte-ს

int compareTo (Integer i)	გამომძახებელი ობიექტის რიცხვით მნიშვნელობას ადარებს i-ს. აბრუნებს: 0-ს, თუ მნიშვნელობები ტოლია; უარყოფით მნიშვნელობას, თუ გამომძახებლის მნიშვნელობა ნაკლებია; დადებით მნიშვნელობას, თუ გამომძახებელის მნიშვნელობა მეტია
static Integer decode (String str) throwsNumberFormatException	აბრუნებს Integer ობიექტს, რომელიც შეიცავს str-ში მითითებულ მნიშვნელობას
double doubleValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც double-ს
boolean equals(Object IntegerObj)	აბრუნებს true-ს, თუ გამომძახებელი Integer ობიექტი ეკვივალენტურია IntegerObj-ის. წინააღმდეგ შემთხვევაში ბრუნდება false
float floatValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც float-ს
static Integer getlnteger (String propertyName)	აბრუნებს მნიშვნელობას, რომელიც ასოცირდება propertyName-ში მითითებულ გარემოს თვისებასთან.

	წარუმატებლობის შემთხვევაში ბრუნდება null
static Integer getlnteger(String propertyName, int default)	აბრუნებს მნიშვნელობას, რომელიც ასოცირდება propertyName-ში მითითებულ გარემოს თვისებასთან. წარუმატებლობის შემთხვევაში ბრუნდება default
static Integer getlnteger(String propertyName, Integer default)	აბრუნებს მნიშვნელობას, რომელიც ასოცირდება propertyName-ში მითითებულ გარემოს თვისებასთან. წარუმატებლობის შემთხვევაში ბრუნდება default
int hashCode ()	აბრუნებს გამომძახებელი ობიექტის ჰეშ კოდს
static int highestOneBit(int num)	განსაზღვრავს num-ის უმაღლეს ბიტს. აბრუნებს მნიშვნელობას, რომელშიც დაყენებულია მხოლოდ ეს ბიტი. თუ არცერთი ბიტი არაა დაყენებული, მაშინ ბრუნდება 0
int intValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც int-ს
long longValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც

	long-ს
static int lowestOneBit (int num)	განსაზღვრავს num-ის უმცროს ბიტს. აბრუნებს მნიშვნელობას, რომელშიც დაყენებულია მხოლოდ ეს ბიტი. თუ არცერთი ბიტი არაა დაყენებული 1-ში, მაშინ ბრუნდება 0
static int numberOfLeadingZeros(int num)	აბრუნებს 0-ში დაყენებული უფროსი ბიტების რაოდენობას, რომლებიც წინ უსწრებს num-ში პირველ დაყენებულ უფროს ბიტს. თუ num=0, ბრუნდება 32
static int numberOfTrailingZeros (int num)	აბრუნებს 0-ში დაყენებული უმცროსი ბიტების რაოდენობას, რომლებიც წინ უსწრებს num-ში პირველ დაყენებულ უმცროს ბიტს. თუ num=0, ბრუნდება 32
static int parseInt(String str) throws NumberFormatException	აბრუნებს str-ში ჩაწერილი რიცხვის 10-ბით ეკვივალენტს
static int parseInt(String str, int radix) throws NumberFormatException	აბრუნებს str-ში radix ფუძით ჩაწერილი რიცხვის int ეკვივალენტს
static int reverse(int num)	ცვლის ბიტების მიმდევრობას num-ში საწინააღმდეგოთი და აბრუნებს მიღებულ შედეგს
static int reverseBytes (int num)	ადგილებს უცვლის num-ის

	უფროს და უმცროს ბაიტებს და შედეგს აბრუნებს
static int rotateLeft(int num, int n)	num-ს ძრავს მარცხნივ n პოზიციით და აბრუნებს შედეგს
static int rotateRight(int num, int n)	num-ს ძრავს მარჯვნივ n პოზიციით და აბრუნებს შედეგს
static int signum (int num)	აბრუნებს -1, თუ num უარყოფითია; 0, თუ ნულია, +1, თუ დადებითია
short shortValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც short-ს
static String toBinaryString(int num)	აბრუნებს სტრიქონს, რომელიც num-ის ორობითი ეკვივალენტია
static String toHexString (int num)	აბრუნებს სტრიქონს, რომელიც num-ის თექვსმეტობითი ეკვივალენტია
static String toOctalString(int num)	აბრუნებს სტრიქონს, რომელიც num-ის რვაობითი ეკვივალენტია
String toString ()	აბრუნებს გამომძახებელი ობიექტის სტრიქონულ ეკვივალენტს
static String toString(int num)	აბრუნებს num-ში ჩაწერილი რიცხვის სტრიქონულ ეკვივალენტს

static Integer valueOf(int num)	აბრუნებს num-ით გადაცემული მნიშვნელობის შემცველ Integer ობიექტს
static Integer valueOf (String str) throws NumberFormatException	აბრუნებს str-ით გადაცემული მნიშვნელობის შემცველ Integer ობიექტს
static Integer valueOf (String str, int radix) throws NumberFormatException	აბრუნებს str-ით radix ფუძით გადაცემული მნიშვნელობის შემცველ Integer ობიექტს

ცხრილი 13. Long კლასის მეთოდები

მეთოდი	აღწერა
static int bitCount(long num)	აბრუნებს ბიტების რაოდენობას num-ში
byte byteValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც byte-ს
int compareTo (Long l)	გამომძახებელი ობიექტის რიცხვით მნიშვნელობას ადარებს l-ს. აბრუნებს: 0-ს, თუ მნიშვნელობები ტოლია; უარყოფით მნიშვნელობას, თუ გამომძახებლის მნიშვნელობა ნაკლებია; დადებით მნიშვნელობას, თუ გამომძახებელის მნიშვნელობა მეტია

static Long decode (String str) throwsNumberFormatException	აბრუნებს Long ობიექტს, რომელიც შეიცავს str-ში მითითებულ მნიშვნელობას
double doubleValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც double-ს
boolean equals(Object LongObj)	აბრუნებს true-ს, თუ გამომძახებელი Long ობიექტი ეკვივალენტურია LongObj-ის. წინააღმდეგ შემთხვევაში ბრუნდება false
float floatValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც float-ს
static Long getLong (String propertyName)	აბრუნებს მნიშვნელობას, რომელიც ასოცირდება propertyName-ში მითითებულ გარემოს თვისებასთან. წარუმატებლობის შემთხვევაში ბრუნდება null
static Long getLong(String propertyName, Long default)	აბრუნებს მნიშვნელობას, რომელიც ასოცირდება propertyName-ში მითითებულ გარემოს თვისებასთან. წარუმატებლობის შემთხვევაში ბრუნდება default
static Integer getlnteger(String	აბრუნებს მნიშვნელობას,

propertyName, Long default)	რომელიც ასოცირდება propertyName-ში მითითებულ გარემოს თვისებასთან. წარუმატებლობის შემთხვევაში ბრუნდება default
int hashCode ()	აბრუნებს გამომძახებელი ობიექტის ჰეშ კოდს
static int highestOneBit(long num)	განსაზღვრავს num-ის უმაღლეს ბიტს. აბრუნებს მნიშვნელობას, რომელშიც დაყენებულია მხოლოდ ეს ბიტი. თუ არცერთი ბიტი არაა დაყენებული, მაშინ ბრუნდება 0
int intValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც int-ს
long longValue()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც long-ს
static int lowestOneBit (long num)	განსაზღვრავს num-ის უმცროს ბიტს. აბრუნებს მნიშვნელობას, რომელშიც დაყენებულია მხოლოდ ეს ბიტი. თუ არცერთი ბიტი არაა დაყენებული 1-ში, მაშინ ბრუნდება 0
static int numberOfLeadingZeros(long	აბრუნებს 0-ში დაყენებული უფროსი ბიტების რაოდენობას,

num)	რომლებიც წინ უსწრებს num-ში პირველ დაყენებულ უფროს ბიტს. თუ num=0, ბრუნდება 32
static int numberOfTrailingZeros (long num)	აბრუნებს 0-ში დაყენებული უმცროსი ბიტების რაოდენობას, რომლებიც წინ უსწრებს num-ში პირველ დაყენებულ უმცროს ბიტს. თუ num=0, ბრუნდება 32
static long parseLong(String str) throws NumberFormatException	აბრუნებს str-ში ჩაწერილი რიცხვის 10-ბით ეკვივალენტს
static int parseInt(String str, int radix) throws NumberFormatException	აბრუნებს str-ში radix ფუძით ჩაწერილი რიცხვის int ეკვივალენტს
static long reverse(long num)	ცვლის ბიტების მიმდევრობას num-ში საწინააღმდეგოთი და აბრუნებს მიღებულ შედეგს
static long reverseBytes (long num)	ადგილებს უცვლის num-ის უფროს და უმცროს ბაიტებს და შედეგს აბრუნებს
static long rotateLeft(long num, int n)	num-ს ძრავს მარცხნივ n პოზიციით და აბრუნებს შედეგს
static long rotateRight(long num, int n)	num-ს ძრავს მარჯვნივ n პოზიციით და აბრუნებს შედეგს
static int signum (long num)	აბრუნებს -1, თუ num უარყოფითია; 0, თუ ნულია, +1,

	თუ დადებითია
short shortValue ()	აბრუნებს გამომძახებელი ობიექტის მნიშვნელობას, როგორც short-ს
static String toBinaryString(long num)	აბრუნებს სტრიქონს, რომელიც num-ის ორობითი ეკვივალენტია
static String toHexString (long num)	აბრუნებს სტრიქონს, რომელიც num-ის თექვსმეტობითი ეკვივალენტია
static String toOctalString(long num)	აბრუნებს სტრიქონს, რომელიც num-ის რვაობითი ეკვივალენტია
String toString ()	აბრუნებს გამომძახებელი ობიექტის სტრიქონულ ეკვივალენტს
static String toString(long num)	აბრუნებს num-ში ჩაწერილი რიცხვის სტრიქონულ ეკვივალენტს
static Long valueOf(int num)	აბრუნებს num-ით გადაცემული მნიშვნელობის შემცველ Long ობიექტს
static Long valueOf (String str) throws NumberFormatException	აბრუნებს str-ით გადაცემული მნიშვნელობის შემცველ Long ობიექტს
static Long valueOf (String str, int radix) throws NumberFormatException	აბრუნებს str-ით radix ფუძით გადაცემული მნიშვნელობის

	შემცველ Long ობიექტს

რიცხვების სტრიქონებად გარდაქმნა და პირიქით

პროგრამირებაში ერთ-ერთი ხშირად შესასრულებელი რუტინული ოპერაციაა რიცხვის სტრიქონული წარმოდგენიდან შიდა ორობით წარმოდგენაში გარდაქმნა. Java-ში რეალიზებულია ამ ოპერაციის ჩატარების მარტივი ხერხი. კლასები Byte, Short, Integer და Long შეიცავენ შესაბამის მეთოდებს parseByte(), parseShort(), parseInt() და parseLong(). ეს მეთოდები აბრუნებენ რიცხვითი სტრიქონის byte, short, int ან long ეკვივალენტებს გამომძახებელი ობიექტისგან. ანალოგიური მეთოდები აქვს Float და Double კლასებს.

შემდეგ პროგრამაში დემონსტრირებულია parseInt() მეთოდის გამოყენება. იგი აჯამებს სტრიქონში ჩაწერილ რიცხვებს:

ლისტინგი 76. parseInt() მეთოდით სტრიქონის რიცხვების აჯამვა

```
import java.io.*;

class ParseDemo {
  public static void main(String args[]) throws
                                        IOException {
    /* BufferedReader-ის შექმნა System.in-ის
                                    გამოყენებით */
    BufferedReader br = new BufferedReader(new
                  InputStreamReader(System.in));
    String str;
```

```
      int i;
      int sum = 0;
      System.out.println("შეიტანეთ რიცხვი, 0
                                  გამოსასვლელაად");
      do {
         str = br.readLine();
         try {
            i = Integer.parseInt(str);
         } catch (NumberFormatException e) {
            System.out.println("არასწორი ფორმატი");
            i = 0;
         }
         sum += i;
         System.out.println("მიმდინარე ჯამი: "+sum);
      } while (i != 0);
   }
}
```

შემდეგ მაგალითში ხდება სტრიქონული მასივის ელემენტების double ტიპში გადაყვანა, თუ ეს შესაძლებელია და აჯამვა:

ლისტინგი 77.

```
public class StringToNumber {
 public static void main(String[] args) {
  String str[] = {
      "12.3", "-23.5", "567E-3", ".5",
      "5,6", "2/0.", "45&","0X23k",
      "5.6", "067", "45", "0X23c" };
  double x = 0.;
  double sum = 0.;
  for (int i = 0; i < str.length; i++) {
   try {
      x = Double.parseDouble(str[i]);
   } catch (NumberFormatException e) {
      System.out.println("***Error    რიცხვის
                  არასწორი ფორმატი!!!: " + str[i]);
      x = 0;
   }
   sum += x;
```

```
    System.out.println(i + "." + "დასამატებელი
რიცხვი:" + x+ "  მიმდინარე sum = " + sum);
   }
  }
}
```

პროგრამის შესრულების შედეგია:

```
0.dasamatebeli ricxvi:12.3  mimdinare sum = 12.3
1.dasamatebeli ricxvi:-23.5  mimdinare sum = -11.2
2.dasamatebeli ricxvi:0.567  mimdinare sum = -10.633
3.dasamatebeli ricxvi:0.5  mimdinare sum = -10.133
***Error   ricxvis araswori formati!!!: 5,6
4.dasamatebeli ricxvi:0.0  mimdinare sum = -10.133
***Error   ricxvis araswori formati!!!: 2/0.
5.dasamatebeli ricxvi:0.0  mimdinare sum = -10.133
***Error   ricxvis araswori formati!!!: 45&
6.dasamatebeli ricxvi:0.0  mimdinare sum = -10.133
***Error   ricxvis araswori formati!!!: 0X23k
7.dasamatebeli ricxvi:0.0  mimdinare sum = -10.133
8.dasamatebeli ricxvi:5.6  mimdinare sum = -
4.5329999999999995
9.dasamatebeli ricxvi:67.0  mimdinare sum = 62.467
10.dasamatebeli ricxvi:45.0  mimdinare sum = 107.467
***Error   ricxvis araswori formati!!!: 0X23c
11.dasamatebeli ricxvi:0.0  mimdinare sum = 107.467
```

რიცხვის ათობით სტრიქონში გარდასაქმნელად, გამოიყენება toString() მეთოდი, რომელიც განსაზღვრულია კლასებში Byte, Short, Integer ან Long. კლასები Integer და Long ასევე გვთავაზობენ toBinaryString(), toHexString() და toOctalString() მეთოდებს, რომლებიც მნიშვნელობას გარდაქმნიან შესაბამისად ბინარულ (ორობით), თექვსმეტობით და რვაობით სტრიქონებში:

ლისტინგი 78. მთელის გარდაქმნა ორობით, თექვსმეტობით და რვაობით ფორმატში

```
class StringConversions {
  public static void main(String args[]) {
    int num = 19648;
```

```
        System.out.println(num + " ორობით ფორმაში: "
                        + Integer.toBinaryString(num));
        System.out.println(num + " რვაობით ფორმაში: "
                        + Integer.toOctalString(num));
        System.out.println(num + " თექვსმეტობით
             ფორმაში: " + Integer.toHexString(num));
    }
}
```

ამ პროგრამის შედეგია:

19648 ორობით ფორმაში: 100110011000000
19648 რვაობით ფორმაში: 46300
19648 თექვსმეტობით ფორმაში: 4cc0

კლასი Character

Character ესაა char ტიპისათვის მარტივი გარსი. მისი კონსტრუქტორია:

Character(char ch)

ch განსაზღვრავს სიმბოლოს, რომელიც უნდა ჩაისვას Character ობიექტის მიერ შექმნილ გარსში. Character ობიექტში შემავალი char მნიშვნელობის მისაღებად უნდა გამოვიძახოთ charValue() მეთოდი:

char charValue()

ეს მეთოდი აბრუნებს სიმბოლოს.

Character კლასში განსაზღვრულია რამდენიმე კონსტანტა:

MAX_RADIX მაქსიმალური ფუძე
MIN_RADIX მინიმალური ფუძე

MAX_VALUE	მაქსიმალური მნიშვნელობა
MIN_VALUE	მინიმალური მნიშვნელობა
TYPE	Class ობიექტი char-სთვის

Character კლასი შეიცავს რამდენიმე სტატიკურ მეთოდს, რომლებიც კატეგორიებად ყოფს სიმბოლოებს და უცვლის მათ რეგისტრს. ისინი აღწერილია ცხრილ 14-ში.

ცხრილი 14. Character კლასის მეთოდები

მეთოდი	აღწერა
static boolean isDefined(char ch)	აბრუნებს true, თუ ch განსაზღვრულია Unicode-ში. წინააღმდეგ შემთხვევაში false
static boolean isDigit (char ch)	აბრუნებს true, თუ ch ათობითი რიცხვია. წინააღმდეგ შემთხვევაში false
static Boolean isIdentifierIgnorable (char ch)	აბრუნებს true, თუ ch იდენთიფიკატორში იგნორირებული უნდა იქნას. წინააღმდეგ შემთხვევაში false
static boolean isISOControl (char ch)	აბრუნებს true, თუ ch ISO-ს მმართველი სიმბოლოა. წინააღმდეგ შემთხვევაში false
static Boolean isJavaIdentifierPart (char ch)	აბრუნებს true, თუ ch შეიძლება იყოს Java-ს იდენთიფიკატორის ნაწილი. წინააღმდეგ შემთხვევაში false

static Boolean isJavaIdentifierStart (char ch)	აბრუნებს true, თუ ch შეიძლება იყოს Java-ს იდენთიფიკატორის პირველი ასო. წინააღმდეგ შემთხვევაში false
static boolean isLetter (char ch)	აბრუნებს true, თუ ch ასოა. წინააღმდეგ შემთხვევაში false
static Boolean isLetterOrDigi t (char ch)	აბრუნებს true, თუ ch ასო ან ციფრია. წინააღმდეგ შემთხვევაში false
static boolean isLowerCase(char ch)	აბრუნებს true, თუ ch ქვედა რეგისტრის ასოა. წინააღმდეგ შემთხვევაში false
static boolean isMirrored(char ch)	აბრუნებს true, თუ ch სარკული Unicode სიმბოლოა (სარკული ნიშნავს ტექსტისთვის დარეზერვირებულ ერთერთ მარჯვნიდან მარცხნივ ჩასაწერ სიმბოლოს). წინააღმდეგ შემთხვევაში false
static boolean isSpaceChar (char ch)	აბრუნებს true, თუ ch სიცარიელის სიმბოლოა. წინააღმდეგ შემთხვევაში false
static boolean isTitleCase(char ch)	აბრუნებს true, თუ ch სატიტულო სიმბოლოა. წინააღმდეგ შემთხვევაში false
static Boolean isUnicodeIdentifierPart	აბრუნებს true, თუ ch დასაშვებია Unicode იდენტიფიკატორის

(char ch)	ნაწილად (პირველი სიმბოლოს გარდა). წინააღმდეგ შემთხვევაში false
static Boo1ean isUnicodeIdentifierStart(char ch)	აბრუნებს true, თუ ch დასაშვებია Unicode იდენტიფიკატორის პირველ სიმბოლოდ. წინააღმდეგ შემთხვევაში false
static boolean isUpperCase(char ch)	აბრუნებს true, თუ ch ზედა რეგისტრის სიმბოლოა. წინააღმდეგ შემთხვევაში false
static boolean isWhitespace (char ch)	აბრუნებს true, თუ ch ერთერთი ცარიელი სიმბოლოა. წინააღმდეგ შემთხვევაში false
static char toLowerCase(char ch)	აბრუნებს ch-ს ქვედა რეგისტრის ეკვივალენტს
static char toTitleCase (char ch)	აბრუნებს ch-ს სატიტულო რეგისტრის ეკვივალენტს
static char toUpperCase(char ch)	აბრუნებს ch-ს ზედა რეგისტრის ეკვივალენტს

Character კლასში განსაზღვრულია ორი მეთოდი forDigits() და digit(), რომლებიც ახორციელებენ გარდაქმნებს მთელ მნიშვნელობებსა და ციფრებს შორის:

static char forDigit(int num, int radix)
static int digit(char digit, int radix)

forDigit() აბრუნებს ათობით ციფრს, რომელიც ასოცირდება num-ის მნიშვნელობასთან. გარდაქმნის ფუძე მოიცემა radix პარამეტრით. digit() აბრუნებს მთელს, რომელიც ასოცირდება მოცემულ სიმბოლოსთან, მითითებული ფუძით.

```
public class MyClass {
   public static void main(String[] args) {
      for (int i = 0; i <= 15; i++)
         System.out.print(Character.forDigit(i, 16)
                                              + " ");
   }
}
```

ამ პროგრამის შედეგია:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

```
public class MyClass {
   public static void main(String[] args) {
      // The character:
      System.out.println('\u0042');
      // The numeric value:
      System.out.println(Character.digit('\u0042',
16));
   }
}
```

ამ პროგრამის შედეგია:

```
B
11
```

Character კლასში ასევე განსაზღვრულია მეთოდი compareTo(), რომელსაც ასეთი ფორმა აქვს:

int compareTo(Character c)

იგი აბრუნებს 0-ს, თუ გამომძახებელი ობიექტი და c ეკვივალენტურია; უარყოფით რიცხვს, თუ გამომძახებელი

ობიექტი შეიცავს ნაკლებ მნიშვნელობას; წინააღმდეგ შემთხვევაში დადებით რიცხვას.

Character კლასში განსაზღვრულია მეთოდი getDirectionality(), რომელიც შეიძლება გამოყენებული იქნას სიმბოლოს მიმართულების გასარკვევად. დამატებულია რამდენიმე კონსტანტა სიმბოლოს ჩაწერის მიმართულების მისათითებლად.

ამ კლასში ასევე გადაფარულია equals() და hashCode() მეთოდები.

Character-ის დამატებები Unicode-ს კოდური წერტილების მხარდაჭერისათვის....

კლასი Boolean

Boolean ესაა boolean მარტივი ტიპის გარშემო შექმნილი გარსი კლასი, რომელიც ძირითადად გამოიყენება, როდესაც საჭიროა boolean-ის მნიშვნელობის გადაცემა მიმთითებლით. ეს კლასი შეიცავს TRUE და FALSE კონსტანტებს, რომლებიც განსაზღვრავენ Boolean კლასის ობიექტებს. Boolean-ში ასევე განსაზღვრულია TYPE ველი, რომელიც არის Class ობიექტი boolean-სათვის. ამ კლასს აქვს შემდეგი კონსტრუქტორები:

Boolean(boolean boolValue)
Boolean(String boolString)

პირველ ვერსიაში boolValue უნდა იყოს true ან false. კონსტრუქტორის მეორე ვერსიაში თუ boolString შეიცავს

"true" (ზედა ან ქვედა რეგისტრებში), მაშინ Boolean-ის ახალი ობიექტი იქნება true. წინააღმდეგ შემთხვევაში false. ცხრილში 15 ჩამოთვლილია ამ კლასში განსაზღვრული მეთოდები.

ცხრილი 15. Boolean კლასის მეთოდები

მეთოდი	აღწერა
boolean booleanValue()	აბრუნებს boolean ეკვივალენტს
int compareTo(Boolean b)	აბრუნებს 0-ს, თუ გამომძახებელი ობიექტი და b შეიცავენ ერთნაირ მნიშვნელობებს; აბრუნებს დადებით მნიშვნელობას, თუ გამომძახებელი არის true, ხოლო b არის false; წინააღმდეგ შემთხვევაში აბ რუნებს უარყოფით რიცხვს
boolean equals(Object boolObj)	აბრუნებს true, თუ გამომძახებელი ობიექტი boolObj-ის ეკვივალენტურია. წინააღმდეგ შემთხვევაში აბრუნებს false-ს
static boolean getBoolean (String propertyName)	აბრუნებს true, თუ propertyName-ში მითითებული სისტემური თვისება წარმოადგენს true-ს. წინააღმდეგ შემთხვევაში აბრუნებს false-ს
int hashCode ()	აბრუნებს გამომძახებელი ობიექტის ჰეშ-კოდს

static boolean parseBoolean(String str)	აბრუნებს true, თუ str სტრიქონი შეიცავს "true"-ს, წინააღმდეგ შემთხვევაში აბრუნებს false-ს
String toString()	აბრუნებს გამომძახებელი ობიექტის სტრიქონულ ექვივალენტს
static String toString(boolean boolVal)	აბრუნებს boolVal-ის სტრიქონულ ექვივალენტს
static Boolean valueOf (boolean boolVal)	აბრუნებს boolVal-ის Boolean ობიექტს
static boolean valueOf (String boolString)	აბრუნებს true-ს, თუ boolString შეცავს "true"-ს (ზედა ან ქვედა რეგისტრში), წინააღმდეგ შემთხვევაში აბრუნებს false-ს

ავტომატური შეფუთვა (autoboxing - ავტობოქსინგი)

ალგორითმულ ენა ჯავაში JDK 5 ვერსიის შემდეგ შევიდა ახალი ფუნქციური სიახლეები რომლებიც საშუალებას იძლევა ავტომატურად მოხდეს ბაზისური ტიპების შეფუთვა (autoboxing) ან განფუთვა (auto-unboxing). ეს ცვლილება განხორციელებული იქნა 3 მიზეზის გამო: პირველი, ავტომატური შეფუთვა/განფუთვა მნიშვნელოვნად აადვილებს და უფრო რაციონალურს ხდის საწყის კოდს, როდესაც საჭიროა ბაზისური ტიპების ობიექტურ ფორმაში გადაყვანა; მეორე, ავტომატური შეფუთვა/განფუთვა ხელს უწყობს JDK 5 ვერსიაში შემოტანილ სიახლის - ტიპების

მორგების (generics) მარტივად და მოხერხებულად გამოყენებას. ამიტომ ავტომატური შეფუთვა/განფუთვის ცოდნა მნიშვნელოვნად შეუწყობს ხელს ამ სიახლის მექანიზმის შესწავლას; მესამე, ავტომატური შეფუთვა/განფუთვა ნათელს გახდის ბაზისურ ტიპებსა და მათ ობიექტურ წარმოდგენებს შორის ურთიერთკაშირს.

როგორც ავღნიშნეთ, ბაზისური ცვლადის მნიშვნელობის ობიექტში ინკაფსულაციას ვუწოდებთ შეფუთვას (boxing). JDK 5 ვერსიამდე ყოველი შეფუთვა ხორციელდებოდა პროგრამისტის მიერ ხელით - საჭირო მნიშვნელობის გარსი კლასის ეგზემპლარის შექმნით. შემდეგ მაგალითში მთელი რიცხვი 100 შეიფუთება Integer-ის ტიპის ობიექტად.

Integer iOb = new Integer(100);

ამ მაგალითში Integer-ის ტიპის ობიექტი იქმნება ხელით ცხადი სახით და მასზე კავშირი მიენიჭება iOb ცვლადს.

გარსი კლასიდან მნიშვნელობის ამოღების პროცესს ვუწოდოთ განფუთვა (unboxing). ანალოგიურად, JDK 5 ვერსიამდე განფუთვა ხორციელდებოდა პროგრამისტის მიერ ხელით. მაგალითად

int i = iOb.intValue();

ამ მაგალითში intValue() მეთოდის საშუალებით ხდება iOb ობიექტიდან int ტიპის მნიშვნელობის ამოღება.

Java-ს საწყისი ვერსიებიდან დაწყებული, შეფუთვისა და განფუთვის განსახორციელებლად, სრულდებოდა ზემოთ განხილული მაგალითების მსგავსი პროცედურები.

მართალია ასეთი ხერხი თანამედროვე ვერსიებშიც მუშაობს, მაგრამ იგი რუტინული სამუშაოა (დამღლელია) და წარმოადგენს შეცდომების წყაროს, ვინაიდან პროგრამისტისაგან მოითხოვს ცხადად მიუთითოს შესაფუთი ან განსაფუთი მონაცემების შესაბამისი მეთოდები და ტიპები. სწორედ ეს პროცესია მოდერნიზებული ავტომატური შეფუთვა/განფუთვის პროცედურებით.

ავტოშეფუთვა (autoboxing) ესაა მარტივი (ბაზისური) ტიპის მონაცემების, მაგალითად int ან double ავტომატური ინკაფსულაცია შესაბამის გარს-კლასში. ამ დროს არაა აუცილებელი საჭირო ტიპის ობიექტის ცხადად შექმნა. ავტოგანფუთვა (auto-unboxing) - ესაა ობიექტიდან მნიშვნელობის ავტომატურად ამოღების პროცესი. ისეთი მეთოდების გამოძახება, როგორიცაა intValue() ან doubleValue() საჭირო აღარაა.

ავტოშეფუთვის არსებობის გამო აღარაა საჭირო ხელით შექიმნას ობიექტი მარტივი ტიპის მნიშვნელობის ინკაფსულაციისათვის. მხოლოდ საკმარისია ეს მნიშვნელობა მიენიჭოს გარსი კლასის ტიპის ობიექტზე მიმთითებელ ცვლადს. Java ავტომატურად შექმნის ამ ობიექტს. მაგალითი

Integer iOb = 100; // ავტომატურად შეიფუთება int ტიპის მნიშვნელობა

როგორც ხედავთ new ბრძანებით ცხადად ობიექტის შექმნა არ ხდება.

ასევე პირიქით, ობიექტის ავტოგანფუთვისთვის ამ ობიექტზე მიმთითებელის მნიშვნელობა უნდა მიენიჭოს შესაბამის ბაზისურ ცვლადს. მაგალითი

```
int i = iOb;  // ავტოგანფუთვა
```

პროგრამის შემდეგ ლისტინგში X.X თავმოყრილია ზემოთ განხილული ფრაგმენტები და დემონსტრირებულია ავტოშეფუთვა/განფუთვის მექანიზმი

```
public class AutoBoxing {

    public static void main(String[] args) {
        Integer iOb = 100;
        int i = iOb;
        System.out.println(i + " " + iOb);

    }
}
```

მინიჭების მარტივი შემთხვევების გარდა, ავტოშეფუთვა/ავტოგანფუთვა შესაძლებელია ავტომატურად განხორციელდეს, როდესაც არგუმენტი გადაეცემა მეთოდს ან მეთოდის შედეგად ბრუნდება მნიშვნელობა. მაგალითი განხილულია ლისტინგ 99.99

```
public class AutoBoxing {

    static  int  m(Integer v) {
        return v;       // auto-unbox to int
    }
```

```
    public static void main(String args[]) {
// int მნიშვნელობა გადაეცემა მეთოდს m()
// დაბრუნებული მნიშვნელობა ენიჭება Integer ტიპის ობიექტს.
// არგუმენტი 100 ავტომატურად შეიუთება Integer ტიპის
// ობიექტად.
// დაბრუნებული მნიშვნელობაც ავტომატურად შეიფუთება Integer
// ტიპად.

        Integer iOb = m(100);
        System.out.println(iOb);
    }
}
```

პროგრამის შესრულების შედეგი:

100

ავტოშეფუთვა/ავტოგანფუთვის მექანიზმი მუშაობს გამოსახულებებშიც. მაგალითი

```
public class AutoBoxing {
    public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);

    ++iOb;
    System.out.println("After ++iOb: " + iOb);

    iOb2 = iOb + (iOb / 3);
    System.out.println("iOb2 after expression: " + iOb2);
```

```
    i = iOb + (iOb / 3);
    System.out.println("i after expression: " + i);
    }
}
```

პროგრამის შესრულების შედეგი:

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

განსაკუთრებული ყურადღება მიაქციეთ ოპერატორს ++iOb; ეს ოპერატორი სრულდება ასე: iOb ობიექტიდან ამოიღება მნიშვნელობა (განიფუთება), მნიშვნელობა გაიზრდება ერთით და შედეგი კვლავ შეიფუთება.

ავტოშეფუთვა/ავტოგანფუთვა საშუალებას იძლევა გამოსახულებაში შეერიოს სხვადასხვა რიცხვითი ობიექტების ტიპები. მაგალითი

ლისტინგი 99.99

```
public class AutoBoxing {
    public static void main(String args[]) {

        Integer iOb = 100;
        Double dOb = 98.6;

        dOb = dOb + iOb;
        System.out.println("dOb after expression: " + dOb);
    }
}
```

პროგრამის შესრულების შედეგი:

```
iOb after expression: 198.6
```

ავტოშეფუთვა/ავტოგანფუთვის საშუალებით მთელრიცხვა ობიექტი შეიძლება გამოყენებული იქნას switch ოპერატორის სამართავად. მაგალითი

```
public class AutoBoxing {
    public static void main(String args[]) {

        Integer iOb = 2;
        switch (iOb) {
        case 1:
            System.out.println("one");
            break;
        case 2:
            System.out.println("two");
            break;
        default:
            System.out.println("error");
        }
    }
}
```

როდესაც switch ოპერატორში გამოითვლება გამოსახულება, განიფუთება iOb ობიექტი და მისგან ამოიღება int ტიპის მნიშვნელობა.

განხილული მაგალითებიდან ჩანს, რომ ავტოშეფუთვა/ავტოგანფუთვის მექანიზმის არსებობა რიცხვითი ობიექტების გამოყენებას ხდის უფრო მარტივს და

ინტუიციურად გასაგებს. თუმცა, აღსანიშნავია, რომ ბაზისური ტიპებისგან მთლიანად განთავისუფლება და მხოლოდ რიცხვითი ობიექტების გამოყენება გამართლებული არ არის. შემდეგ მაგალითში ნაჩვენებია ავტოშეფუთვა/ავტოგანფუთვის მექანიზმის ცუდი გამოყენება

```
public class AutoBoxing {
    public static void main(String args[]) {
        Double a, b, c;
        a = 10.0;
        b = 4.0;
        c = Math.sqrt(a*a + b*b);
        System.out.println("Hypotenuse is " + c);
    }
}
```

მართალია პროგრამის კოდი ტექნიკურად კორექტულია, მაგრამ ძალიან ცუდად ხდება ავტოშეფუთვა/ავტოგანფუთვის მექანიზმის გამოყენება. გაცილებით ეფექტური იქნებოდა ასეთ შემთხვევაში მარტივი ტიპის ცვლადების გამოყენება, ვინაიდან ყოველი ავტოშეფუთვა/ავტოგანფუთვა იწვევს დამატებით რესურსების გამოყენებას.

კლასი Void.....

აბსტრაქტული კლასი Process .....

კლასი Runtime ....

კლასი ProcessBuilder...

კლასი System

System კლასი შეიცავს სტატიკურ მეთოდებსა და ცვლადებს. Java-ს შემსრულებელი გარემოს სტანდარტული შეტანა/გამოტანა, შეცდომების გამოტანა ინახება ცვლადებში in, out და err. System კლასის მეთოდები ჩამოთვლილია 16 ცხრილში. ბევრი მათგანი იწვევს SecurityException განსაკუთრებული სიტუაციის აღგზნებას, თუ ოპერაცია უსაფრთხოების მენეჯერისგან დაშვებული არაა.

ცხრილი 16. System კლასის მეთოდები

მეთოდი	აღწერა
static void arraycopy (Object source, int sourceStart, Obj ect target, int targetStart, int size)	აკოპირებს მასივს. source - დასაკოპირებელი მასივია, sourceStart - კოპირების დასაწყისის ინდექსი მასივში, კოპიოს მიმღები მასივია target, targetStart - მასში კოპირების დაწყების ინდექსი, size - დასაკოპირებელი ელემენტების რაოდენობაა
static long currentTimeMillis()	აბრუნებს 1970 წლის 1 იანვრის შუაღამიდან მიმდინარე მომენტამდე გასულ დროს

	მილიწამებში
static void gc ()	იწყებს „ნაგვის შემკრების“ მუშაობას
static int identityHashCode (Object obj)	აბრუნებს obj-ის საიდენტიფიკაციო ჰეშ-კოდს
static long nanoTime ()	მიიღებს სისტემის შეძლებისდაგვარად ზუსტ ტაიმერს და აბრუნებს მის მნიშვნელობას ნანოწამებში, რომელიც გასულია რომელიღაც საწყისი მომენტიდან. ტაიმერის სიზუსტე უცნობია
static void runFinalization ()	ახდენს finalize() მეთოდების გამოძახებას იმ ობიექტებისათვის, რომლებიც აღარ გამოიყენება, მაგრამ ჯერ უტილიზებული (განადგურებული) არაა

currentTimeMills() მეთოდის გამოყენება

System კლასის ერთ-ერთი საინტერესო გამოყენებაა currentTimeMills() მეთოდით პროგრამის ცალკეული ნაწილების შესრულების დროის გამოთვლა. ეს მეთოდი აბრუნებს 1970 წლის 1 იანვრის შუაღამიდან მიმდინარე მომენტამდე გასულ დროს მილიწამებში. პროგრამის რომე-ლიმე ნაწილის ქრონომეტრაჟისათვის, უნდა გამოვიძახოთ ეს მეთოდი უშუალოდ ამ ნაწილის დაწყების წინ და შევინახოთ მიღებული მნიშვნელობა. პროგრამის ნაწილის

დამთავრებისთანავე უნდა გამოვიძახოთ იგივე მეთოდი კიდევ ერთხელ და გამოვითვალოთ მათი სხვაობა. ეს პროცესი დემონსტრირებულია შემდეგ პროგრამაში:

ლისტინგი 79. პროგრამის შესრულების დროის გამოთვლა

```
class Elapsed {
   public static void main(String args[]) {
      long start, end;
      System.out.println("გასული დრო 0-დან "
                     + "1 000 000-მდე ციკლისათვის");
      start = System.currentTimeMillis(); // საწყისი
                                              დრო
      for (int i = 0; i < 1000000; i++)
         ;
      end = System.currentTimeMillis();//საბოლოო
                                              დრო
      System.out.println("შესრულების დრო: "+(end));
   }
}
```

შესაძლოა ამ პროგრამის შედეგი სხვადასხვა კომპიუტერზე სხვადასხვა იყოს:

გასული დრო 0-დან 1 000 000-მდე ციკლისათვის
შესრულების დრო: 10

თუ კონკრეტული სისტემის ტაიმერის სიზუსტე ნანოწამების დონისაა, შესაძლებელია წინა პროგრამაში currentTimeMills() მეთოდი შევცვალოთ nanoTime() მეთოდით. ქვემოთ ნაჩვენებია ამ ცვლილების საკვანძო ფრაგმენტი:

```
start = System.nanoTime(); // საწყისი დროის მიღება
for(int i=0; i < 1000000; i++) ;
end = System.nanoTime();  // საბოლოო დროის მიღება
```

arraycopy() მეთოდი

მეთოდი arraycopy() შეიძლება გამოყენებული იქნას ნებისმიერი ტიპის მასივის სწრაფი კოპირებისათვის ერთი ადგილიდან მეორეში. ეს გაცილებით სწრაფია, ვიდრე Java-ზე დაწერილი ეკვივალენტური ციკლი. ქვემოთ ნაჩვენებია ორი მასივის მაგალითი. თავიდან ხდება a-ს კოპირება b-ში. შემდეგ a მასივის ყველა ელემენტი დაიძვრება ქვემოთ ერთი ელემენტით. შემდეგ b-ს ელემენტები დაიძვრება ერთი პოზიციით ზემოთ:

ლისტინგი 80. arraycopy()-ს გამოყენება

```
class ACDemo {
   static byte a[] = { 65, 66, 67, 68, 69, 70, 71,
                                    72, 73, 74 };
   static byte b[] = { 77, 77, 77, 77, 77, 77, 77,
                                    77, 77, 77 };
   public static void main(String args[]) {
      System.out.println("a " + new String(a));
      System.out.println("b " + new String(b));
      System.arraycopy(a, 0, b, 0, a.length);
      System.out.println("a= " + new String(a));
      System.out.println("b= " + new String(b));
      System.arraycopy(a, 0, a, 1, a.length - 1);
      System.arraycopy(b, 1, b, 0, b.length - 1);
      System.out.println("a= " + new String(a));
      System.out.println("b= " + new String(b));
   }
}
```

როგორც შედეგიდან ჩანს, ერთიდაიგივე წყარო და მიმღები შეიძლება კოპირდებოდეს ორივე მიმართულებით:

a = ABCDEFGHIJ
b = MMMMMMMMMM

a = ABCDEFGHIJ

b = ABCDEFGHIJ

a = AABCDEFGHI

b = BCDEFGHIJJ

კლასი Math

Math კლასი შეიცავს ყველა იმ მცოცავმძიმიან ფუნქციებს, რომლებსაც იყენებენ გეომეტრიაში და ტრიგონომეტრიაში, ასევე ზოგადი დანიშნულების ზოგიერთ მეთოდს. ამ კლასში განსაზღვრულია ორი კონსტანტა E (დაახლოებით 2.72) და PI (დაახლობით 3.14).

ტრანსცენდენტული ფუნქციები

ცხრილ 17-ში ჩამოთვლილია მეთოდები, რომლებიც პარამეტრად იღებენ double ტიპს, რომელიც კუთხეს აღწერს რადიანებში და შედეგად აბრუნებს შესაბამის ტრანსცენდენტული ფუნქციის მნიშვნელობას.

ცხრილი 17. Math კლასის ტრანსცენდენტული ფუნქციები

მეთოდი	აღწერა
static double sin (double arg)	აბრუნებს arg კუთხის სინუსს. arg მოცემულია რადიანებში
static double cos (double arg)	აბრუნებს arg კუთხის კოსინუსს. arg მოცემულია რადიანებში
static double tan (double arg)	აბრუნებს arg კუთხის ტანგენსს.

	arg მოცემულია რადიანებში

ცხრილი 18. Math კლასის შებრუნებული ტრანსცენდენტული ფუნქციები

მეთოდი	აღწერა
static double asin (double arg)	აბრუნებს კუთხეს, რომლის სინუსი ტოლია arg-ის
static double acos (double arg)	აბრუნებს კუთხეს, რომლის კოსინუსი ტოლია arg-ის
static double atan (double arg)	აბრუნებს კუთხეს, რომლის ტანგენსი ტოლია arg-ის
static double atan2 (double x, double y)	აბრუნებს კუთხეს, რომლის ტანგენსი ტოლია x / y

ცხრილი 19. Math კლასის ჰიპერბოლური ფუნქციები

მეთოდი	აღწერა
static double sinh (double arg)	აბრუნებს arg კუთხის ჰიპერბოლურ სინუსს. arg მოცემულია რადიანებში
static double cosh (double arg)	აბრუნებს arg კუთხის ჰიპერბოლურ კოსინუსს. arg მოცემულია რადიანებში
static double tanh (double arg)	აბრუნებს arg კუთხის ჰიპერბოლურ ტანგენსს. arg

	მოცემულია რადიანებში

ექსპონენციალური ფუნქციები

ცხრილი 20. Math კლასის ექსპონენციალური ფუნქციები

მეთოდი	აღწერა
static double cbrt (double arg)	აბრუნებს arg-დან კუბურ ფესვს
static double exp (double arg)	აბრუნებს arg-ის ექსპონენტას
static double expml (double arg)	აბრუნებს arg-1 -ის ექსპონენტას
static double log (double arg)	აბრუნებს arg-ის ნატურალურ ლოგარითმს
static double log10 (double arg)	აბრუნებს arg-ის ლოგარითმს 10-ის ფუძით
static double loglp (double arg)	აბრუნებს arg+1 -ის ნატურალურ ლოგარითმს
static double pow(double y, double x)	აბრუნებს Y^X
static double scalb (double arg, int factor)	აბრუნებს arg x 2^{factor} (Java SE 6)
static float scalb(float arg, int factor)	აბრუნებს arg x 2^{factor} (Java SE 6)
static double sqrt (double arg)	აბრუნებს კვადრატულ ფესვს arg-დან

დამრგვალების ფუნქციები

Math კლასში განსაზღვრულია ზოგიერთი მეთოდი, რომლებიც გამოყენებულია სხვადასხვა სახის დამრგვალების ოპერაციებისათვის. ისინი ჩამოთვლილია ცხრილში 16.20. ყურადღება მისაქცევია ცხრილის ბოლოში მითითებული ulp() ორი მეთოდი. ამ კონტექსტში ulp ნიშნავს "units in the last place" (ერთეულები ბოლო ადგილზე). ეს მიუთითებს ერთეულების რაოდენობას მნიშვნელობასა და უახლოეს დიდ მნიშვნელობას შორის. იგი გამოიყენება შედეგის სიზუსტისათვის.

ცხრილი 21. Math კლასის დამრგვალების ფუნქციები

მეთოდი	აღწერა
static int abs (int arg)	აბრუნებს arg-ის აბსოლუტურ მნიშვნელობას
static long abs (long arg)	აბრუნებს arg-ის აბსოლუტურ მნიშვნელობას
static float abs (float arg)	აბრუნებს arg-ის აბსოლუტურ მნიშვნელობას
static double abs (double arg)	აბრუნებს arg-ის აბსოლუტურ მნიშვნელობას
static double ceil (double arg)	აბრუნებს უმცირეს მთელ რიცხვს, რომელიც მეტია arg-ზე
static double floor(double arg)	აბრუნებს უდიდეს მთელ რიცხვს, რომელიც ნაკლებია arg-ზე

static int max (int x, int y)	აბრუნებს ორ x და y რიცხვებს შორის უდიდესს
static long max (long x, long y)	აბრუნებს ორ x და y რიცხვებს შორის უდიდესს
static float max (float x, float y)	აბრუნებს ორ x და y რიცხვებს შორის უდიდესს
static double max (double x, double y)	აბრუნებს ორ x და y რიცხვებს შორის უდიდესს
static int min (int x, int y)	აბრუნებს ორ x და y რიცხვებს შორის უმცირესს
static long min (long x, long y)	აბრუნებს ორ x და y რიცხვებს შორის უმცირესს
static float min (float x, float y)	აბრუნებს ორ x და y რიცხვებს შორის უმცირესს
static double min (double x, double y)	აბრუნებს ორ x და y რიცხვებს შორის უმცირესს
static double nextAfter(double arg, double toward)	დაწყებული arg -ის მნიშვნელობიდან, აბრუნებს toward-ის მიმართულებით შემდეგ მნიშვნელობას. თუ arg == toward, მაშინ აბრუნებს toward-ს. (Java SE 6)
static float nextAfter (float arg, double toward)	დაწყებული arg -ის მნიშვნელობიდან, აბრუნებს toward-ის მიმართულებით შემდეგ მნიშვნელობას. თუ arg == toward,

	მაშინ აბრუნებს toward-ს. (Java SE 6)
static double nextUp (double arg)	აბრუნებს arg-დან დადებითი მიმართულებით შემდეგ მნიშვნელობას. (Java SE 6)
static float nextUp (float arg)	აბრუნებს arg-დან დადებითი მიმართულებით შემდეგ მნიშვნელობას. (Java SE 6)
static double rint (double arg)	აბრუნებს arg-ის უახლოეს მთელს
static int round (float arg)	აბრუნებს დამრგვალებულ arg-ს, დამრგვალებულს ზემოთ უახლოეს int მთელამდე
static long round (double arg)	აბრუნებს დამრგვალებულ arg-ს, დამრგვალებულს ზემოთ უახლოეს long მთელამდე
static float ulp (float arg)	აბრუნებს arg-ის ulp-ს
static double ulp (double arg)	აბრუნებს arg-ის ulp-ს

Math კლასის სხვა მეთოდები

ცხრილი 22. Math კლასის სხვადასხვა ფუნქციები

მეთოდი	აღწერა
static double copySign(double arg, double signarg)	აბრუნებს arg-ს იმავე ნიშნით, როგორიცა აქვს signarg-ს (Java SE 6)
static float copySign (float arg,	აბრუნებს arg-ს იმავე ნიშნით,

float signarg)	როგორიცა აქვს signarg-ს (Java SE 6)
static int getExponent (double arg)	აბრუნებს 2-ის ფუძით ექსპონენტას, რომელიც გამოყენებულია arg-ის ოროობითი წარმოდგენისათვის (Java SE 6)
static int getExponent (float arg)	აბრუნებს 2-ის ფუძით ექსპონენტას, რომელიც გამოყენებულია arg-ის ოროობითი წარმოდგენისათვის (Java SE 6)
static double IEEEremainder(double dividend, double divisor)	აბრუნებს dividend/divisor გაყოფის ნაშთს (Java SE 6)
static hypot(double sidel, double side2)	აბრუნებს მართკუთხა სამკუთხედის ჰიპოტენუზას ორი კათეტის მიხედვით
static double random ()	აბრუნებს ფსევდოშემთხვევით რიცხვს 0-დან 1-მდე
static float signum (double arg)	განსაზღვრავს მნიშვნელობის ნიშანს. აბრუნებს 0, თუ arg ტოლია 0-ის; 1, თუ arg მეტია 0-ზე; -1, თუ arg ნაკლებია 0-ზე
static float signum (float arg)	განსაზღვრავს მნიშვნელობის ნიშანს. აბრუნებს 0, თუ arg ტოლია 0-ის; 1, თუ arg მეტია 0-ზე; -1, თუ arg ნაკლებია 0-ზე
static double toDegrees(double angle)	რადიანებს გარდაქმნის

	გრადუსებში.
static double toRadians(double angle)	გრადუსებს გარდაქმნის რადიანებში

კლასი StrictMath

StrictMath კლასი Math კლასის პარალელურად განსაზღვრავს მათემატიკური მეთოდების სრულ ნაკრებს. განსხვავება ისაა, რომ StrictMath ვერსია იძლევა სიზუსტის გარანტიას Java-ს ყველა რეალიზაციისათვის, მაშინ, როდესაც Math კლასს აქვს უფრო მეტი თავისუფლება ეფექტურობის მიღწევის მიზნით.

კოლექციები

განვხილოთ java.util პაკეტი. იგი შეიცავს კლასებისა და ინტერფეისების დიდ ასორტიმენტს, რომლებიც აფართოებენ ფუნქციონალობის დიაპაზონს. მაგალითად, java.util შეიცავს კლასებს, რომლებიც გამოიმუშავებენ ფსევდო შემთხვევით რიცხვებს,მართავენ დროსა და თარიღს, მანიპულირებას ახდენენ ბიტებზე, მართავენ ობიექტთა ჯგუფებს. ეს პაკეტი ზომით ერთ-ერთი ყველაზე დიდი პაკეტია. ჩვენ განვიხილავთ მისი შემადგენელი კლასებისა და ინტერფეისების მხოლოდ ნაწილს, რომლებიც ეძღვნება კოლექციებს. კოლექციის კარკასი ესაა ინტერფეისებისა და კლასების რთული იერარქია, რომლებიც უზრუნველყოფენ ობიექტთა ჯგუფების მართვის მოქნილ ტექნოლოგიას. პროგრამისტები ხშირად იყენებენ კოლექციებს.

Java-ს კოლექციების კარკასი ახდენს ობიექტთა ჯგუფების მართვის ხერხების სტანდარტიზებას. კოლექციები თავიდან არ შედიოდა Java-ს შემადგენლობაში და იგი დამატებული იქნა მის მეორე ვერსიაში. თავიდან, ობიექტთა ჯგუფების მანიპულირებისა და შენახვისათვის Java გვთავაზობდა სპეციალურ კლასებს Dictionari, Vector, Stack და Properties. მართალია ეს კლასები საკმაოდ მოსახერხებელი იყო, მაგრამ მათ აკლდა ცენტრალიზებული, უნივერსალური იდეა და არ იყო გათვალისწინებული მათი მომავალი გაფართოება და ადაპტაცია. კოლექციები გახდა ამ და კიდევ სხვა პრობლემების გადაწყვეტის საშუალება.

კოლექციების კარკასი დამუშავებული იქნა შემდეგი მიზნების მისაღწევად:

- ის უნდა იყოს მაღალი წარმადობის. ძირითადი კოლექციები, როგორიცაა დინამიური მასივები, სიები, ხეები და ჰეშ ცხრილები მართლაც გამოირჩევა მაღალი წარმადობით და იშვიათად თუ დასჭირდება რომელიმე პროგრამისტს ასეთი მონაცემთა მართვის მექანიზმების ხელით კოდირება;

- სხვადასხვა ტიპის კოლექციებს, სისტემა საშუალებას უნდა აძლევდეს იმუშაონ ერთი მანერით და ურთიერთკავშირით;

- კოლექციების გაფართოება და/ან ადაპტაცია უნდა იყოს მარტივი;

- კოლექციები აგებული უნდა იქნან ერთიანი ინტერფეისების ნაკრების საშუალებით.

პროგრამისტის ხელშეწყობისათვის, ზოგიერთი ინტერფეისის სტანდარტული რეალიზაცია მზადაა გამოსაყენებლად, ზოგიერთი ნაწილობრივაა რეალიზებული. ისინი მნიშვნელოვნად ამარტივებენ საკუთარი კოლექციების შექმნასაც, თუ ეს საჭირო გახდა.

კოლექციების კარკასის მეორე მნიშვნელოვანი ნაწილია ალგორითმები. ისინი ოპერირებენ კოლექციებზე და განსაზღვრულია სტატიკური მეთოდების სახით. ამგვარად, მათი გამოყენება ყველა კოლექციას შეუძლია. კოლექციის თითოეული კლასისათვის აღარაა საჭირო ამ ალგორიმების საკუთარი ვერსიის რეალიზაცია მოახდინოს. ალგორითმები წარმოადგენენ კოლექციებზე მანიპულირების სტანდარტულ ხერხებს.

კოლექციების სისტემასთან მჭიდროდაა ასოცირებული ინტერფეისი Iterator. იტერატორი წარმოადგენს კოლექციის ელემენტებზე სათითაოდ მიმართვის ერთიან, სტანდარტული საშუალებას. ანუ, იტერატორი ესაა კოლექციის შემადგენლობის ჩამოთვლის ხერხი. ვინაიდან თითოეული კოლექცია ახდენს Iterator ინტერფეისის რეალიზაციას, კოლექციის ყოველი კლასის ელემენტებზე წვდომა შეიძლება მოხდეს ამ ინტერფეისში ჩამოთვლილი მეთოდებით.

აღსანიშნავია, რომ მართალია კოლექციების დამატებამ გამოიწვია ზოგიერთი ორიგინალური გამოყენებითი კლასის არქიტექტურის ცვლილება, მაგრამ არცერთი მათგანი არ აკრძალულა. კოლექციები უბრალოდ გვთავაზობს ზოგიერთი რამის უკეთესად გადაწყვეტას.

ბოლო დროს კოლექციებმა კვლავ განიცადეს მნიშვნელოვანი ცვლილებები, რომლებმაც გააფართოვეს მათი გამოყენების არე და აამაღლეს მისი სიმძლავრე. მოკლედ განვიხილოთ ორი სფრო, რომლებსაც შეეხოთ ეს ცვლილებები.

პრიმიტიული ტიპების ავტომატური შეფუთვის საშუალებები

კოლექციებში პრიმიტიული ტიპების შენახვას ამარტივებს ავტომატური შეფუთვა. როგორც მალე ვნახავთ, კოლექციებში შესაძლებელია, მხოლოდ ობიექტებზე მიმთითებლების შენახვა და არა პრიმიტიული ტიპებისა. ადრე, თუ გვინდოდა კოლექციაში პრიმიტიული ტიპის შენახვა მაგალითად int ტიპის, ვალდებული ვიყავით იგი ცხადად, ხელით, შეგვეფუთა Integer გარსი კლასის ტიპად. როდესაც კოლექციიდან მნიშვნელობის ამოღება გვინდოდა („განფუთვა“), ასევე ცხადად, ხელით უნდა ამოგვეღო შენახული გარსი კლასის შესაბამისი ობიექტიდან კორექტული პრიმიტიული ტიპი. ავტომატური შეფუთვა/განფუთვის დროს, Java ახლა ამას ავტმატურად აკეთებს, როდესაც საჭიროა პრიმიტიული ტიპების კოლექციაში შენახვა ან ამოღება. აღარაა საჭირო ამ ოპერაციების ხელით ჩატარება.

for-each ციკლის ტიპი

კოლექციების ყველა კლასი ისეა მოდიფიცირებული, რომ ისინი ანხორციელებენ Iterable ინტერფეისის რეალიზაციას. ეს კი ნიშნავს, რომ შესაძლებელია ციკლის საშუალებით გავიაროთ კოლექციის ყოველ ელემენტზე for-each ციკლის

საშუალებით. ადრე კოლექციის ელემენტების გასავლელად საჭირო იყო იტერატორის გამოყენება და ციკლის პროგრამული კონსტრუირება. მართალია იტერატორები გარკვეული მიზნებისათვის ისევ გამოიყენება, მაგრამ იტერატორების გამოყენებაზე აგებული ციკლები ახლა უმრავლეს შემთხვევაში შეიძლება შეიცვალოს for-each ციკლებით.

კოლექციების ინტერფეისები

კოლექციების კარკასი შეცავს რამდენიმე ინტერფეისს. კოლექციების ინტერფეისების განხილვა აუცილებელია, ვინაიდან ისინი განსაზღვრავენ კოლექციების კლასების ფუნდამენტურ თვისებებს. ცალკეული კონკრეტული კლასები ანხორციელებენ სტანდარტული ინტერფეისების სხვადასხვა რეალიზაციას. ჩვენ განვიხილავთ 17.1 ცხრილში ჩამოთვლილ ზოგიერთ ინტერფეისს

ცხრილი 17.1 კოლექციების ინტერფეისები

ინტერფეისი	აღწერა
Collection	უზრუნველყოფს ობიექტთა ჯგუფებთან მუშაობას. ესაა კოლექციების იერარქიის სათავე
Deque	აფართოებს Queue-ს ორმხრივი რიგების დასამუშავებლად. (Java SE 6)
List	აფართოებს Collection-ს მიმდევრობების (ობიექტთა სიების) სამართავად

NavigableSet	აფართოებს SortedSet-ს უახლოესი შესაბამისობის მიხედვით ელემენტების ამოსაღებად. (Java SE 6)
Queue	აფართოებს Collection-ს სიების სპეციალური ტიპების სამართავად, სადაც ელემეტები მხოლოდ თავიდან იმართება
Set	აფართოებს Collection-ს სიმრავლეების (ერთობლიობების, ნაკრებების) სამართავად, რომლებიც უნდა ინახავდნენ უნიკალურ ელემენტებს
SortedSet	აფართოებს Set-ს სორტირებული სიმრავლეების სამართავად

ინტერფეისი Collection

ინტერფეისი Collection წამოადგენს ფუნდამენტს, რომელზედაც აგებულია კოლექციის მთელი კარკასი, ვინაიდან იგი რეალიზებული უნდა იქნას კოლექციების ყველა კლასის მიერ. Collection არის ზოგადი ინტერფეისი და ასეთი სახე აქვს:

interface Collection<E>

E - მიუთითებს იმ ობიექტების ტიპს, რომლებიც უნდა შედიოდეს კოლექციაში. Collection არის Iterable ინტერფეისის მემკვიდრე. ეს ნიშნავს, რომ ყოველ კოლექციაში შესაძლებელია for-each ციკლით ელემენტების გავლა.

Collection განსაზღრავს ძირითად მეთოდებს, რომლებიც ექნებათ ყველა კოლექციას. ეს მეთოდები ჩამოთვლილია 17.2 ცხრილში

ცხრილი 17.2 Collection-ში განსაზღვრული მეთოდები

მეთოდი	აღწერა
boolean add(E obj)	ამატებს obj-ს გამომძახებელ კოლექციას. აბრუნებს true-ს, თუ მოხდა obj-ის დამატება
boolean addAll(Collection<?> extends E	ამატებს c-ს ყველა ელემენტს გამომძახებელ კოლექციას. აბრუნებს true-ს, თუ ოპერაცია შესრულდა (ანუ ყველა ელემენტი დაემატა). წინააღმდეგ შემთხვევაში აბრუნებს false-ს
void clear ()	გამომძახებელი კოლექციიდან შლის ყველა ელემენტს
boolean contains(Object obj)	აბრუნებს true-ს, თუ obj წარმოადგენს გამომძახებელი კოლექციის ელემენტს. წინააღმდეგ შემთხვევაში აბრუნებს false-ს
boolean containsAll(Collection<?> c)	აბრუნებს true-ს, თუ გამომძახებელი კოლექცია შეიცავს c-ს ყველა ელემენტს. წინააღმდეგ შემთხვევაში აბრუნებს false-ს

boolean equals(Object obj)	აბრუნებს true-ს, თუ გამომძახებელი კოლექცია და obj ეკვივალენტურებია. წინააღმდეგ შემთხვევაში აბრუნებს false-ს
int hashCode ()	აბრუნებს გამომძახებელი კოლექციის ჰეშ-კოდს
boolean isEmpty ()	აბრუნებს true-ს, თუ გამომძახებელი კოლექცია ცარიელია. წინააღმდეგ შემთხვევაში აბრუნებს false-ს
Iterator<E> iterator()	აბრუნებს გამომძახებელი კოლექციის იტერატორს
boolean remove(Object obj)	გამომძახებელი კოლექციიდან ამოაგდებს obj-ის ერთ ეგზემპლარს. აბრუნებს true-ს , თუ ელემენტი ამოვარდა (წაიშალა), წინააღმდეგ შემთხვევაში false
boolean removeAll(Collection<?> c)	გამომძახებელი კოლექციიდან წაშლის ყველა იმ ელემენტს, რომლებიც c-ში შედის. აბრუნებს true-ს, თუ შედეგად კოლექცია იცვლება (ანუ, ელემენტები წაიშალა). წინააღმდეგ შემთხვევაში false
boolean retainAll(Collection<?> c)	გამომძახებელი კოლექციიდან წაშლის ყველა ელემენტს, გარდა იმ ელემენტებისა, რომლებიც c-ში

	შედის. აბრუნებს true-ს, თუ შედეგად კოლექცია იცვლება (ანუ, ელემენტები წაიშალა). წინააღმდეგ შემთხვევაში false
int size ()	აბრუნებს კოლექციაში შემავალი ელემენტების რაოდენობას
Object [] toArray ()	აბრუნებს მასივს, რომელიც შეიცავს გამომძახებელი კოლექციის ყველა ელემენტს. მასივის ელემენტები წარმოადგენენ კოლექციის ელემენტების კოპიოს
<T> T [] toArray (T array[])	აბუნებს მასივს, რომელიც შეიცავს გამომძახებელი კოლექციის ელემენტებს. მასივის ელემენტები წარმოადგენენ კოლექციის ელემენტების კოპიოს. თუ array მასივის ზომა ტოლია ელემენტების რაოდენობისა, იგი დაბრუნდება. თუ array მასივის ზომა ნაკლებია ელემენტების რაოდენობაზე, მაშინ შეიქმნება და დაბრუნდება საჭირო სიგრძის ახალი მასივი. თუ array მასივის ზომა ტიამე ელემენტების რაოდენობაზე, მაშინ კოლექციის შემდეგი ელემენტები გახდება null-ის ტოლი. თუ კოლექციის რომელიმე ელემენტის ტიპი არაა array-ის ქვეტიპი, წარმოიქმნება

	ArrayStoreException გასაკუთრებული სიტუაცია

ვინაიდან ყველა კოლექცია ახდენს Collection-ის რეალიზაციას, ამიტომ მისი მეთოდების ცოდნა აუცილებელია კოლექციების კარკასის ცხადად წარმოდგენისათვის (აღქმისათვის). ზოგიერთმა ამ მეთოდმა შეიძლება აღძრას UnsupportedOperationException სიტუაცია. ეს ხდება, როდესაც შეუძლებელია კოლექციის მოდიფიცირება. ClassCastException წარმოიქმნება, როდესაც ობიექტები არაა ურთიერთთავსებადი. NullPointerException წარმოიქმნება, როდესაც მცდელობაა ჩაისვას null მნიშვნელობა ისეთ კოლექციაში, რომელშიც null ელემენტები არაა დაშვებული. IllegalArgumentException წარმოიქმნება, როდესაც მცდელობაა ჩაისვას ახალი ელემენტი ფიქსირებული სიგრძის შევსებულ კოლექციაში.

კოლექციაში ობიექტების დამატება ხდება add() მეთოდით. ავღნიშნოთ, რომ add() იღებს E ტიპის ელემენტს, რაც ნიშნავს, რომ კოლექციაში ჩასამატებელი ობიექტები უნდა იყოს თავსებადი კოლექციის მონაცემთა ტიპებისადმი. ერთი კოლექციის მთლიანი შემადგენლობის (შემცველობის) დამატება მეორე კოლექციაში შესაძლებელია addAll() მეთოდის გამოძახებით.

ობიექტის ამოგდება შესაძლებელია remove() მეთოდით. ობიექტთა ჯგუფის ამოსაგდებად გამოიყენება removeAll() მეთოდი. შესაძლებელია მითითებულის გარდა ყველა

ობიექტის ამოგდება retainAll() მეთოდით. კოლექციის მთლიანად გასასუფთავებლად შესაძლებელია clear() მეთოდის გამოძახება.

contains() მეთოდით შესაძლებელია განვსაზღვროთ შეიცავს თუ არა კოლექცია გარკვეული ტიპის ობიექტს. თუ გაინტერესებს ერთი კოლექცია შეიცავს თუ არა მეორე კოლექციის ყველა ელემენტს უნდა გამოვიძახოთ containsAll() მეთოდი. კოლექცია ცარიელია თუა რა ამის შემოწმება ხდება მეთოდით isEmpty(). კოლექციის შემადგენელი ელემენტების მიმდინარე რაოდენობას აბრუნებს მეთოდი size().

toArray() მეთოდები აბრუნებენ მასივს, რომელიც შეიცავს კოლექციაში შემავალ ელემენტებს. პირველი მათგანი აბრუნებს Object-ების მასივს. მეორე - იმ ტიპის ელემენტების მასივს, რა ტიპის მასივიცაა მითითებული პარამეტრში. ჩვეულებრივ მეორე მეთოდი უფრო უმჯობესია, ვინაიდან იგი აბრუნებს საჭირო ტიპის ელემენტების მასივს. ეს მეთოდები იფრო მნიშვნელოვანია, ვიდრე ერთი შეხედვით შეიძლება მოგვეჩვენოს. ხშირად უფრო მომგებიანია კოლექციაში შემავალი ელემენტების დამუშავება მასივის სახით. ვინაიდან გვაქვს კოლექციიდან მასივად გარდაქმნის მარტივი ხერხი, ამიტომ შეგვიძლია ვისარგებლოთ ორივე წარმოდგენის უპირატესობებით.

ორი კოლექცია შეიძლება შევადაროთ ეკვივალენტურობაზე equals() მეთოდით. „ეკვივალენტურობის“ ზუსტი შინაარსი შეიძლება იცვლებოდეს კოლექციების მიხედვით. მაგალითად, შესაძლებელია equals() მეთოდის ისეთი

რეალიზაცია, რომ იგი ადარებდეს კოლექციაში შენახული ელემენტების მნიშვნელობებს. ალტერნატივის სახით equals() შეიძლება ადარებდეს ამ ელემენტებზე მიმთითებლებს.

დანარჩენი ინტერფეისების აღწერის ნახვა შესაძლებელია ამ სახელმძღვანელოს ბოლოში მითითებულ წყაროებში.

კოლექციებზე წვდომა იტერატორის საშუალებით (გვ. 479)

...

კონტეინერული კლასები

java.util პაკეტში განსაზღვრულია Collection-ის ერთი მემკვიდრე ინტერფეისი Enumeration და შემდეგი მემკვიდრე კლასები: Dictionary, Hashtable, Properties, Stack, Vector.

განვიხილოთ ეს ინტერფეისი და ზოგიერთი კლასი.

ინტერფეისი Enumeration

Enumeration ინტერფეისში განსაზღვრულია მეთოდები, რომლებითაც შესაძლებელია ჩამოვთვალოთ (სათითაოდ მივიღოთ) ობიექტების კოლექციის ელემენტები. მართალია Enumeration მოძველებულია, მაგრამ იგი არაა აკრძალული და გამოიყენება ზოგიერთ კონტეინერულ კლასებში (მაგ.Vector, Properties). იგი ასევე ხშირადაა გამოყენებული მრავალ გამოყენებით პროგრამაში, ამიტომ JDK 5-ში მოხდა მისი ხელახალი დაპროექტება და განზოგადება. მას აქვს ასეთი გამოცხადება:

interface Enumeration<E>

E განსაზღვრავს იმ ელემენტების ტიპს, რომელთა ჩამოთვლაც მოხდება.

Enumeration-ში განსაზღვრულია ორი მეთოდი:

boolean hasMoreElements()
E nextElement()

hasMoreElements() მეთოდის რეალიზაციამ უნდა დააბრუნოს true მანამ, სანამ დარჩენილია ამოსაღები ელემენტები. აბრუნებს false-ს, როდესაც ყველა ელემენტი უკვე ჩამოთვლილია. nextElement() აბრუნებს ჩამონათვალის შემდეგ ელემენტს. ანუ, nextElement()-ის ყოველი გამოძახება აბრუნებს ჩამონათვალის შემდეგ ელემენტს. ჩამონათვალში გავლის დამთავრების შემდეგ, ამ მეთოდის გამოძახება იწვევს NoSuchE1ementException გამონაკლისის გენერირებას.

კლასი Vector

Vector კლასი ანხორციელებს დინამიურ მასივს. მას აქვს მრავალი მემკვიდრეობით გადმოცემული მეთოდი, რომლებიც არ წარმოადგენენ კოლექციის კარკასის ნაწილს. JDK 5-ში მოხდა მისი ხელახალი დაპროექტება და იგი ახდენს Iterable ინტერფეისი რეალიზაციას. ეს ნიშნავს, რომ იგი სრულად თავსებადია კოლექციებთან და შეუძლია თავისი შემცველობა გასცეს for-each ციკლის საშუალებითაც.

კლასი Vector-ის გამოცხდება ხდება ასე:

class Vector<E>

E განსაზღვრავს შესანახი ელემენტების ტიპს. მისი კონსტრუქტორებია:

Vector ()
Vector (int size)
Vector{int size, int incr)
Vector(Collection<? extends E> c)

პირველი ფორმა ქმნის სტანდარტულ ვექტორს, რომლის საწყისი ზომაა 10. მეორე ფორმა ქმნის ვექტორს, რომლის საწყისი მოცულობა size-ის ტოლია. მესამე ფორმა ქმნის ვექტორს, რომლის საწყისი ზომაა size, ხოლო ინკრემენტი მითითებულია incr ცვლადით. იგი მიუთითებს ელემენტების რაოდენობას, რომლებიც გამოიყოფა მაშინ, როდესაც საჭიროა ვექტორის ზომის გაზრდა. მეოთხე ფორმა ქმნის ვექტორს, რომელიც შეიცავს c კოლექციის ელემენტებს.

ყველა ვექტორი იწყება რაღაც საწყისი მოცულობიდან. როდესაც ეს მოცულობა შევსებულია, ობიექტის ვექტორში შენახვის შემდგომი მცდელობისას, ვექტორში ავტომატურად მოხდება გამოყოფილი მოცულობის გაზრდა. ეს გაზრდა მნიშვნელოვანია, ვინაიდან მეხსიერების განაწილება დროის მიხედვით ძვირადღირებული ოპერაციაა. დამატებითი მეხსიერების საერთო მოცულობა, რომელიც გამოიყოფა ყოველი განაწილების დროს, მოიცემა ინკრემენტის მნიშვნელობით, რომელიც ვექტორის გამოცხდების დროს მიეთითება. თუ ინკრემენტის მნიშვნელობა მითითებული არაა, მაშინ ვექტორის ზომა ორმაგდება მეხსიერების განაწილების ყოველ ციკლზე.

ვექტორში განსაზღვრულია შემდეგი დაცული წევრი-მონაცემები:

```
int capacityIncrement;
int elementCount;
Object[] elementData;
```

ინკრემენტის მნიშვნელობა შენახულია capacityIncrement ველში. ვექტორში მიმდინარე მომენტში არსებული ელემენტების რაოდენობა ინახება elementCount ველში. მასივი, რომელიც ინახავს თვით ვექტორს, ესაა elementData.

ცხრილში 17.19 ჩამოთვლილია ზოგიერთი მემკვიდრეობით გადმოცემული მეთოდი:

ცხრილი 17.19 Vector კლასის მეთოდები

მეთოდი	აღწერა
void addElement(E element)	element-ში მითითებული ობიექტი ემატება ვექტორს
int capacity()	აბრუნებს ვექტორის მოცულობას
Object clone ()	აბრუნებს გამომძახებელი ობიექტის დუბლიკატს (კლონს)
boolean contains(Object element)	აბრუნებს true-ს, თუ element შედის ვექტორში. წინააღმდეგ შემთხვევაში false
void copyInto (Object array[])	გამომძახებელ ვექტორის ელემენ-ტები კოპირდება array მასივში

E elementAt (int index)	აბრუნებს index პოზიციაში განლაგებულ ელემენტს
Enиmeration<E> elements()	აბრუნებს ვექტორის ელემენტების ჩამონათვალს
void ensureCapacity(int size)	ვექტორის მინიმალური მოცულობა გახდება size-ის ტოლი
E firstElement()	აბრუნებს ვექტორის პირველ ელემენტს
int indexOf(Object element)	აბრუნებს element-ის პირველი დამთხვევის ინდექსს. თუ ვექტორში ობიექტი ვერ მოინახა აბრუნებს -1
int indexOf(Object element, int start)	აბრუნებს element-ის start-ის შემდეგ პირველი დამთხვევის ინდექსს. თუ ვექტორში ობიექტი ვერ მოინახა აბრუნებს -1
void insertElementAt(E element, int index)	index პოზიციაში ვექტორს ემატება element-ი
boolean isEmpty ()	აბრუნებს true, თუ ვექტორი ცარიელია და false - წინააღმდეგ შემთხვევაში
E lastElement ()	აბრუნებს ვექტორის ბოლო ელემენტს
int lastIndexOf(Object element)	აბრუნებს element-ის ბოლო დამთხვევის ინდექსს. თუ ვექტორში ობიექტი ვერ მოინახა

	აბრუნებს -1
int lastIndexOf(Object element, int start)	აბრუნებს start-ის წინ element-ის ბოლო დამთხვევის ინდექსს. თუ ვექტორში ობიექტი ვერ მოინახა აბრუნებს -1
void removeAllElements ()	ასუფთავებს ვექტორს. ამ მეთოდის შესრულების შემდეგ ვექტორის ზომა 0-ის ტლია
boolean removeElement(Object element)	ამოაგდებს ვექტორიდან element-ს.თუ ვექტორში ამ ელემენტის რამდენიმე ეგზემპლარია, წაიშლება მხოლოდ პირველი მათგანი.აბრუნებს true-ს თუ ელემენტი წაიშალა. წინააღმდეგ შემთხვევაში false
void removeElementAt(int index)	index პოზოციიდან წაშლის ელემენტს
void setElementAt(E element, int index)	index პოზოციაში ჩაწერს element-ს
void setSize (int size)	ვექტორში ელემენტების რაოდენობას აყენებს size-ის ტოლად. თუ ახალი ზომა ნაკლებია ძველზე, ელემენტები იკარგება. თუ ახალი მეტია ძველზე, მაშინ ემატება null ელემეტები

int size ()	აბრუნებს ვექტორში მოთავსებული ელემენტების რაოდენობას
String toString ()	აბრუნებს ვექტორის სტრიქონულ ეკვივალენტს
void trimToSize ()	ვექტორის მოცულობას აყენებს მოცემულ მომენტში ვექტორის ელემენტების რაოდენობს ტოლად

შემდეგ პროგრამაში ვექტორი გამოიყენება სხვადასხვა ტიპის რიცხვითი ობიექტების შესანახად. პროგრამაში ასევე ნაჩვენებია Enumeration ინტერფეისთან მუშაობა:

```
// Vector-თან სხვადასხვა ოპერაციების დემონსტრირება
import java.util.*;
class VectorDemo {
    public static void main(String args[]) {
// საწყისი ზომა 3, ინკრემენტი 2
        Vector<Integer> v = new Vector<Integer> (3, 2);
        System.out.println ("საწყისი ზომა: " + v. size () ) ;
        System.out.println("საწყისი ზომა: " + v.capacity());
        v.addElement(l);
        v.addElement(2);
        v.addElement(3);
        v.addElement(4);
        System.out.println("მოცულობა ოთხი დამატების შემდეგ: " +
                                v.capacity());
```

```
        v.addElement(5);
        System.out.println("მიმდინარე მოცულობა:" + v.capacity());
        v. addElement (6);
        v.addElement(7);
        System.out.println("მიმდინარე მოცულობა: " + v.capacity());
        v.addElement'(9) ;
        v.addElement(l0);
        System.out.println("მიმდინარე მოცულობა: " + v.capacity());
        v.addElement(ll);
        v.addElement(12);
        System.out.println("პირველი ელემენტი: "+
                                    v.firstElement());
        System.out.println("ბოლო ელემენტი: " + v.lastElement () );
        if(v.contains(3)) System.out.println("ვექტორში შედის 3.");
        // ვექტორის ელემენტების ჩამონათვალი
        Enumeration vEnum = v.elements();
        System.out.println("\n ვექტორის ელემენტები:");
        while(vEnum.hasMoreElements())
            System.out.print (vEnum.nextElement() + " ");
        System.out.println();
    }
}
```

ამ პროგრამის შედეგი:

```
საწყისი ზომა: 0
საწყისი მოცულობა: 3
მოცულობა ოთხი დამატების  სემდეგ 5
მიმდინარე მოცულობა: 5
მიმდინარე მოცულობა: 7
```

მიმდინარე მოცულობა: 9
პირველი ელემენტი: 1
ბოლო ელემენტი: 12
ვექტორი შეიცავს 3.
ვექტორის ელემენტები:
1 2 3 4 5 6 7 9 10 11 12

ვექტორში ელემენტები შეიძლება გავიაროთ for-each ციკლით, როგორც ეს ნაჩვენებია წინა კოდის შემდეგი ვერსიაში:

```
System.out.println("\n ვექტორის ელემენტები:");
for (int i : v)
    System.out.print(i + " ");
System.out.println();
```

ვინაიდან Enumeration-ის გამოყენება ახალ კოდში არაა რეკომენდებული, ამიტომ ყველა ელემენტის გასავლელად ჩვეულებრივ გამოყენებული უნდა იქნას for-each ციკლი ან იტერატორი.

კლასი Stack

კლასი Stack წარმოადგენს Vector კლასის ქვეკლასს და რეალიზაციას უკეთებს სტანდარტულ სტეკს, რომელიც მუშაობს პრინციპით - „ბოლო შემოსული - პირველი გამოვა (LIFO)“. Stack-ში განსაზღვრულია მხოლოდ სტანდარტული კონსტრუქტორი, რომელიც ცარიელ სტეკს ქმნის. JDK 5-ის გამოსვლის შემდეგ Stack კლასი განახლდა განზოგადებული

სინტაქსის შესაბამისად და ახლა მისი გამოცხადება ასე ხდება:

class Stack<E>

E მიუთითებს ელემენტების ტიპს, რომლებიც სტეკში შეინახება.

Stack კლასში შედის ყველა ის მეთოდი, რომელიც განსაზღვრულია Vector კლასში და ამატებს ზოგიერთ თავისს (ცხრილი 17.20)

ცხრილი 17.20 Vector კლასის მეთოდები

მეთოდი	აღწერა
boolean empty()	აბრუნებს true, თუ სტეკი ცარიელია; false, თუ ელემენტებს შეიცავს
E peek ()	აბრუნებს სტეკის თავში მდგომ ელემენტს, მაგრამ მას არ ამოაგდებს
E pop ()	აბრუნებს სტეკის თავში მდგომ ელემენტს და ამოაგდებს მას სტეკიდან
E push (E element)	element-ს ჩააგდებს სტეკში. ასევე ეს ელემენტი ბრუნდება მეთოდის მნიშვნელობად
int search(Object element)	ეძებს element-ს სტეკში. თუ იგი მოიძებნა, აბრუნებს სტეკის

	თავიდან ამ ელემენტამდე ძვრას. წინააღმდეგ შემთხვევაში აბრუნებს -1

ელემენტის სტეკის თავში მოსათავსებლად გამოიძახება push() მეთოდი. სტეკის თავში მდგომი ელემენტის ამოსაღებად და დასაბრუნებლად უნდა მოხდეს pop() მეთოდის გამოძახება. თუ ცარიელი სტეკის მიმართ გამოვიყენებთ pop() მეთოდს, წარმოიქმნება EmptyStackException განსაკუთრებული სიტუაცია.

შემდეგ მაგალითში იქმნება სტეკი, მასში მოთავსდება რამდენიმე Integer ობიექტი და შემდეგ მოხდება მათი ამოღება:

```
// კლას Stack-ის დემონსტრირება
import java.util.*;
class StackDemo {
    static void showpush(Stack<Integer> st, int a) {
        st.push(a);
        System.out.println("push(" + a + ")");
        System.out.println("stack: " + st);
    }
    static void showpop(Stack<Integer> st) {
        System.out.print("pop > ");
        Integer a = st.pop();
        System.out.println{a);
        System.out.println("stack: " + st);
```

```
    }
    public static void main(String args[]) {
        Stack<Integer> st = new Stack<Integer>();
        System.out.println("stack: " + st);
        showpush(st, 42);
        showpush(st, 66);
        showpush(st, 99);
        showpop(st);
        showpop (st);
        showpop(st);
        try {
            showpop(st);
        } catch (EmptyStackException e) {
            System.out.println("stack empty");
    }
}
```

ქვემოთ ნაჩვენებია პროგრამის შესრულების შედეგი. ყურადღება მიაქციეთ განსაკუთრებული სიტუაციის დამმუშავებელ ფრაგმენტს, რომელიც წარმატებით წყვეტს ცარიელ სტეკზე მიმართვის გამო წარმოქმნილ გამონაკლისს:

```
stack: []
push(42)
stack: [42]
push(66)
stack: [42, 66]
push (99)
stack: [42, 66, 99]
pop > 99
```

stack: [42, 66]
pop > 66
stack: [42]
pop > 42
stack: []
pop > stack empty

კლასი Dictionary

Dictionary (ლექსიკონი, სიტყვარი) კლასი წარმოადგენს აბსტრაქტულ კლასს და ანხორციელებს მონაცემების წყვილების შენახვის მექანიზმს პრინციპით „გასაღები - მნიშვნელობა“. გასაღებისა და მნიშვნელობის გადაცემით შესაძლებელია მნიშვნელობის შენახვა Dictionary კლასში. ერთხელ შენახული მნიშვნელობა შეიძლება ამოვიღოთ მისი გასაღებით. ანუ Dictionary კლასი შეიძლება ჩავთვალოთ „გასაღები - მნიშვნელობა“ წყვილების სიად. JDK 5-ის გამოსვლის შემდეგ ეს კლასიც გახდა განზოგადებული. იგი უნდა გამოვაცხადოთ ასე:

class Dictionary<K, V >

K მიუთითებს გასაღების ტიპს, ხოლო V - მნიშვნელობის ტიპს. Dictionary კლასში განსაზღვრული აბსტრაქტული მეთოდები ჩამოთვლილია ცხრილ 17.21-ში.

ცხრილი 17.21 Dictionary კლასის აბსტრაქტული მეთოდები

მეთოდი	აღწერა

Enumeration<V> elements ()	აბრუნებს ლექსიკონში შენახულ მნიშვნელობების ჩამონათვალს
V get (Object key)	აბრუნებს key-სთან ასოცირებულ ობიექტს. თუ key ლექსიკონში არ მოიძებნა - აბრუნებს null
boolean isEmpty ()	აბრუნებს true, თუ ლექსიკონი ცარიელია, false - თუ იგი ერთ ელემეტს მაინც შეიცავს
Enumeration<K> keys()	აბრუნებს ლექსიკონში მოთავსე-ბული ელემენტების გასაღებების ჩამონათვალს
V put (K key, V value)	ლექსიკონში მოათავსებს გასაღებსა და მნიშვნელობას. აბრუნებს null, თუ key ლექსიკონში არ არსებობს, წინააღმდეგ შემთხვევაში აბრუნებს key-სთან წინათ ასოცირებული ელემენტის მნიშვნელობას
V remove(Object key)	ლექსიკონიდან ამოშლის key-სა და მასთან ასოცირებულ ელემენტს. თუ key ლექსიკონში არ არსებობს აბრუნებს null
int size ()	აბრუნებს ლექსიკონში ელემენტების რაოდენობას

კლასი Hashtable

Hashtable კლასი შედის java.util პაკეტში და იგი შეიძლება ჩავთვალოთ Dictionary აბსტრაქტული კლასის რეალიზაციად. JDK-ს ახალ ვერსიებში ეს კლასიც განახლდა და ახლა იგი ინტეგრირებულია კოლექციების კარკასთან. Hashtable კლასი ჰეშ-ცხრილში ინახავს წყვილებს „გასაღები - მნიშვნელობა“. თუმცა, აღსანიშნავია, რომ არც გასაღები და არც მნიშვნელობა არ შეიძლება null-ის ტოლი იყოს.

Hashtable-ის გამოყენებისას უნდა მივუთითოთ ობიექტი, რომელიც გამოიყენება გასაღებად და მნიშვნელობა, რომელიც გვინდა ასოცირდებოდეს გასაღებთან. ამის შემდეგ, მოხდება გასაღების ჰეშირება და შედეგად მიღებული ჰეშ-კოდი გამოიყენება იმ ცხრილის ინდექსად, რომელშიც უნდა მოხდეს მნიშვნელობის შენახვა.

JDK 5-ში Hashtable გახდა განზოგადებული და მისი გამოცხადება ასე ხდება:

class Hashtable<K, V>

K მიუთითებს გასაღების ტიპს, ხოლო V - მნიშვნელობის ტიპს.

ჰეშ-ცხრილი ინახავს მხოლოდ ისეთ ობიექტებს, რომლებიც გადაფარავენ Object კლასში აღწერილ hashCode() და equals() მეთოდებს. hashCode() მეთოდმა უნდა გამოითვალოს და დააბრუნოს ობიექტის ჰეშ-კოდი. equals() მეთოდმა უნდა შეადაროს ორი ობიექტი. Java-ს თითქმის ყველა ჩაშენებული კლასი ანხორციელებს hashCode() მეთოდის რეალიზაციას.

Hashtable კლასის კონსტრუქტორებია:

Hashtable ()
Hashtable(int size)
Hashtable(int size, float fillRatio)

პირველი ვარიანტი სტანდარტული კონსტრუქტორია. მეორე ვერსია ქმნის ჰეშ-ცხრილს, რომლის საწყისი ზომა მითითებულია size პარამეტრით. სტანდარტულად ზომა 11-ის ტოლია. მესამე ვერსია ქმნის ჰეშ-ცხრილს, რომლის საწყისი ზომა size-ის ტოლია და შევსების კოეფიციენტი მითითებულია fillRatio პარამეტრით. ეს კოეფიციენტი 0.0-დან 1.0-მდე დიაპაზონშია და განსაზღვრავს, რამდენად შევსებული უნდა იყოს ცხრილი, რომ მისი გაფართოება მოხდეს. უფრო ზუსტად, როდესაც ელემენტების რაოდენობა აჭარბებს მოცულობისა და შევსების კოეფიციენტის ნამრავლს, ჰეშ-ცხრილი ფართოვდება. თუ შევსების კოეფიციენტს არ მივუთითებთ, სტანდარტულად გამოიყენება 0.75.

ცხრილში 17.22 ჩამოთვლილია Hashtable კლასის ძირითადი მეთოდები. ზოგიერთი მათგანი null მნიშვნელობის გამოყენებისას წარმოქმნის NullPointerException გამონაკლისს.

ცხრილი 17.22 Hashtable კლასის მეთოდები

მეთოდი	აღწერა
void clear ()	ასუფთავებს ჰეშ-ცხრილს
Object clone ()	აბრუნებს გამომძახებელი ობიექტის დუპლიკატს

boolean contains(Object value)	აბრუნებს true, თუ ჰეშ-ცხრილში არსებობს value-ს ეკვივალენტური რომელიმე მნიშვნელობა. აბრუნებს false, თუ მნიშვნელობა ვერ მოიძებნა
boolean containsKey(Obj ect key)	აბრუნებს true, თუ ჰეშ-ცხრილში არსებობს key-ს ეკვივალენტური რომელიმე მნიშვნელობა. აბრუნებს false, თუ მნიშვნელობა ვერ მოიძებნა
boolean containsValue(Object value)	აბრუნებს true, თუ ჰეშ-ცხრილში არსებობს value-ს ეკვივალენტური რომელიმე მნიშვნელობა. აბრუნებს false, თუ მნიშვნელობა ვერ მოიძებნა
Enumeration<V> elements()	აბრუნებს ჰეშ-ცხრილში მოთავსებული მნიშვნელობების ჩამონათვალს
V get (Object key)	აბრუნებს key-სთან ასოცირებული ობიექტს. თუ ჰეშ-ცხრილში key ვერ მოიძებნა აბრუნებს null
boolean isEmpty ()	აბრუნებს true, თუ ჰეშ-ცხრილი ცარიელია. აბრუნებს false, თუ ცხრილში ერთი მაინც წევრია
Enumeration<K> keys()	აბრუნებს ჰეშ-ცხრილში შემავალი გასაღებების ჩამონათვალს

V put (K key, V value)	ჰეშ-ცხრილში ათავსებს გასაღებსა და მნიშვნელობას. აბრუნებს null, თუ key ამ მომენტისათვის ცხრილში ვერ მოიძებნა. წინააღმდეგ შემთხვევაში აბრუნებს ამ გასაღებთან ასოცირებულ წინა მნიშვნელობას
void rehash ()	ზრდის ჰეშ-ცხრილს და თავიდან უკეთებს ჰეშირებას მის ყველა გასაღებს
V remove (Object key)	ჰეშ-ცხრილიდან ამოაგდებს key-ს. აბრუნებს key-სთან ასოცირებულ მნიშვნელობას. თუ ჰეშ-ცხრილში key ვერ მოიძებნა აბრუნებს null
int size ()	აბრუნებს ჰეშ-ცხრილის ელემენტების რაოდენობას
String toString ()	აბრუნებს ჰეშ-ცხრილის სტრიქონულ ეკვივალენტს

განვიხილოთ Hashtable კლასის გამოყენებით საბანკო ანგარიშების მართვის პროგრამა, რომელშიც ხდება მომხმარებლების სახელებისა და მიმდინარე ბალანსების შენახვა:

```
// Hashtable კლასის გამოყენების დემონსტრირება

import java.util.*;
class HTDemo {
    public static void main(String args[]) {
```

```
        Hashtable<String,Double> balance =
                new Hashtable<String, Double>();
        Enumeration<String> names;
        String str;
        double bal;
        balance.put("John doe", 3434.34);
        balance.put("Tom Smith", 123.22);
        balance.put("Jane Baker", 1378.00);
        balance.put("Tod Hall", 99.22);
        balance.put("Ralph Smith", 19.08);
// ჰეშ-ცხრილში არსებული ყველა ანგარიშის ჩვენება
        names = balance.keys();
        whi1e(names.hasMoreElements()) {
            str = names.nextElement();
            System.out.println(str + ": " + balance.get(str));
        }
        System.out.println();
// 1,000 დამატება John doe-ს ანგარიშზე
        bal = balance.get ("John doe");
        balance.put("John doe", bal+1000);
        System.out.println("John doe-ს ახალი ბალანსი: " +
                        balance.get("John doe"));
    }
}
```

ამ პროგრამის შედეგი:

```
Tod Hall: 99.22
Ralph Smith: 19.08
John doe: 3434.34
```

Jane Baker: 1378.0
Tom Smith: 123.22
John doe-ს ახალი ბალანსი: 4434.34

ამრიგად, კოლექციის კარკასი პროგრამისტს თავაზობს მძლავრ გულდასმით დაპროექტებული გადაწყვეტილებების კრებულს (ერთობლიობას) პრაქტიკაში ხშირად გამოყენებადი ამოცანების გადასაწყვეტად. მას შემდეგ რაც კოლექციების კარკასი გახდა განზოგადებული, იგი შეიძლება გამოყენებული იქნას ტიპების უსაფრთხოების სრული დაცვით, რაც ხელს უწყობს მის შემდგომ განვითარებას. არსანიშნავია, რომ კოლექციების გამოყენება მხოლოდ დიდი ამოცანების გადასაწყვეტად არაა ეფექტური. ისინი ასევე ეფექტურია მცირე ამოცანების გადაწყვეტის დროსაც. ზოგადად, მათი გამოყენებით ხშირად დიდი მოგების მიღებაა შესაძლებელი. და მათი გამოყენების სფერო მხოლოდ პროგრამისტის ფანტაზიითაა შემოსაზღვრული.

AWT პაკეტი

აქამდე ჩვენს მიერ განხილული ყველა პროგრამის შესრულება დაკავშირებული იყო ტექსტურ ტერმინალთან, ამიტომ ასეთ პროგრამებს **კონსოლურ** პროგრამებს უწოდებენ. კონსოლურ პროგრამებში მომხმარებელთან ინტერაქტიული კავშირი შეზღუდულია. მომხმარებლისთვის გამიზნული ყოველი თანამედროვე გამოყენებითი პროგრამა აღიქვამს კლავიატურისა და მაუსიდან მომავალ სიგნალებს, ფლობს მოხერხებულ, გასაგებ და (სასურველია) ლამაზ სამომხმარებლო გრაფიკულ ინტერფეისს (GUI - Graphical user

interface). გრაფიკულ გარემოცვაში მომუშავე ყოველმა პროგრამამ უნდა შექმნას მინიმუმ ერთი ფანჯარა, სადაც წარიმართება მისი მუშაობა. გრაფიკული ფანჯარა უნდა დარეგისტრირდეს ოპერაციული სისტემის გრაფიკულ გარემოში, რათა მან შეძლოს ურთიერთქმედება ოპერაციულ სისტემასა და სხვა ფანჯრებთან: გადაფარვა, გადაადგილება, ზომების შეცვლა, დახურვა და სხვა.

არსებობს სხვადასხვა გრაფიკული სისტემები: MS Windows, X Window System, Macintosh. თითოეულ მათგანში განსხვავებულია ფანჯრებისა და მისი კომპონენტების - მენიუების, შეტანის ველების, ღილაკების, სიების, რბიების აგების წესები. ეს წესები, როგორც წესი რთულია და ჩახლართული. ფანჯრებისა და კომპონენტების აგების გასაადვილებლად შექმნილია სხვადასხვა კლასების ბიბლიოთეკები: MFC, Motif, OpenLook, Qt, Tk, Xview, OpenWindows და სხვა. Java ტექნოლოგიაში საქმე კიდევ უფრო გართულებულია, ვინაიდან Java-ზე დაწერილმა პროგრამებმა უნდა იმუშაოს ნებისმიერ ან თუნდაც უმრავლეს გრაფიკულ გარემოში. საჭიროა კლასების ბიბლიოთეკა, რომელიც არ იქნება დამოკიდებული კონკრეტულ გრაფიკულ სისტემაზე.

ვინაიდან Java-ზე დაწერილი გამოყენებითი პროგრამა შესაძლებელია მუშაობდეს სხვადასხვა პლატფორმაზე, მომხმარებლის გრაფიკული ინტერფეისი უნდა იყოს ან ერთნაირი ყველა პლატფორმისათვის, ან პირიქით, პროგრამას უნდა ჰქონდეს მოცემული პლატფორმის (ოპერაციული სისტემის) დამახასიათებელი გარეგნული იერსახე.

გარკვეული მიზეზების გამო, GUI-ს რეალიზაციისათვის საჭირო ძირითადი ბიბილიოთეკისათვის არჩეული იქნა მეორე გზა. ეს არჩევანი კიდევ ერთხელ ხაზს უსვამს ამ ენის მოქნილობას, ვინაიდან სხვადასხვა პლატფორმის მომხმარებლებს შეუძლიათ იმუშაონ ერთიდაიგივე პროგრამასთან მისთვის ჩვეულ გრაფიკულ გარემოში და არ შეიცვალონ თავისი ჩვევები. განხორციელებული რეალიზაცია უზრუნველყოფს მაღალ მწარმოებლობას, ვინაიდან დაფუძნებულია ოპერაციული სისტემის შესაძლებლობებზე, ამიტომ იგი არის კომპაქტური, მარტივი და შესაბამისად საიმედო კოდის მქონე.

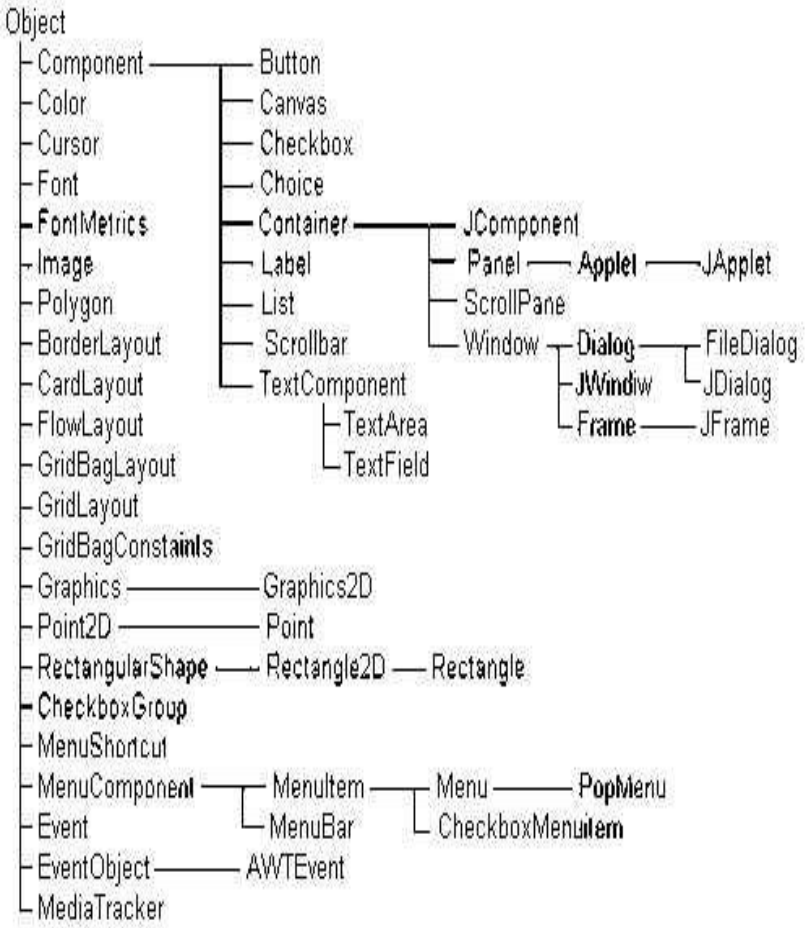
ალგორითმულ ენა Java-შიც GUI-ს განსახორციელებლად არსებობს სპეციალური ტექნოლოგია, რომელსაც ეწოდება AWT (Abstract Window Toolkit). ამ ტექნოლოგიასთან დაკავშირებული კლასებისა და ინტერფეისების ბაზისური ბიბილიოთეკა მოთავსებულია java.awt პაკეტში. ენის განვითარებასთან ერთად ამ პაკეტმა განიცადა ყველაზე უფრო ბევრი ცვლილება.

კლასების ბიბილიოთეკის სახელში AWT (Abstract Window Toolkit) სიტყვა აბსტრაქტი მიუთითებს იმას, რომ ყველა სტანდარტული კომპონენტი არაა დამოუკიდებელი, ისინი მუშაობენ კონკრეტული ოპერაციული სისტემის შესაბამის ელემენტთან შეწყვილებულად.

პაკეტი შეიცავს მრავალრიცხოვან კლასებსა და მეთოდებს, რომლებითაც შესაძლებელია ფანჯრების შექმნა და მათი მართვა. AWT–ს კლასები მოთავსებულია java.awt პაკეტში. ეს

პაკეტი ორგანიზებულია იერარქიულად, ამიტომ გასაგებად და გამოსაყენებლად ადვილია.

AWT განსაზღვრავს ფანჯრებს, რომლებსაც კლასთა იერარქიის მიხედვით, ყოველი დონე ამატებს ფუნქციონალურ შესაძლებლობებსა და სპეციფიკას. შემდეგ ნახატზე მოცემულია ამ იერარქიის ზოგიერთი ძირითადი კლასი.



გრაფიკული კომპონენტები და კონტეინერები

Component

მომხმარებლის გრაფიკული ინტერფეისის ძირითადი ელემენტია ვიზუალური კომპონენტი (component). გრაფიკულ ინტერფეისში ეს ცნება განსაზღვრავს ცალკეულ, სრულყოფილად განსაზღვრულ ელემენტს, რომელიც შესაძლებელია გამოყენებული იქნას სხვა ელემენტებისაგან დამოუკიდებლად. მაგალითად, ესაა შეტანის ველი, ღილაკი, მენიუს სტრიქონი, რბია, რადიოღილაკი. თვით პროგრამის ფანჯარაც კომპონენტს წარმოადგენს. კომპონენტები შეიძლება იყოს უხილავიც, მაგალითად პანელი, რომელიც აერთიანებს კომპონენტებს, ასევე წარმოადგენს კომპონენტს.

AWT კლასების ბიბილიოთეკის ყველა ვიზუალური კომპონენტის ბაზისური (მშობელი) კლასია აბსტრაქტული კლასი Component და იგი აღწერს მათ ძირითად თვისებებს. AWT-ს ვიზუალურ კომპონენტს აქვს მართკუთხა ფორმა, შესაძლებელია მისი ასახვა ეკრანზე და შეუძლია მომხმარებელთან ურთიერთქმედება. განვიხილოთ ამ კლასის ძირითადი თვისებები.

მდებარეობა

კომპონენტის მდებარეობა აღიწერება ორი მთელი რიცხვით (int ტიპის) x და y. Java-ში (როგორც ბევრ სხვა ალგორითმულ ენაში) x ღერძი ტრადიციულად გადის ჰორიზონტალურად და მიმართულია მარჯვნივ, ხოლო y ღერძი - ვერტიკალურად, მაგრამ მიმართულია ქვევით.

კომპონენტის მდებარეობის აღწერისათვის შემოღებულია სპეციალური კლასი Point (წერტილი). ამ კლასში აღწერილია ორი public int ველი x და y, ასევე რამდენიმე კონსტრუქტორი და დამხმარე მეთოდი. Point კლასი გამოიყენება AWT-ს მრავალ ტიპში, სადაც საჭიროა წერტილის სიბრტყეზე მოცემა. კომპონენტებისათვის ეს წერტილი მიუთითებს მარცხენა ზედა კუთხის მდებარეობას. კომპონენტის მდებარეობის დაყენება შესაძლებელია მეთოდით setLocation(), რომელიც არგუმენტად ღებულობს ორ მთელი რიცხვს ან Point კლასის ეგზემპლარს. მიმდინარე მდებარეობის გაგება შესაძლებელია მეთოდით getLocation(), რომელიც აბრუნებს Point-ს, ან მეთოდებით getX() და getY().

ზომა

როგორც ზემოთ აღვნიშნეთ, AWT კომპონენტი მართკუთხედის ფორმისაა, ამიტომ მისი ზომაც აღიწერება ორი მთელრიცხვა პარამეტრით width (სიგანე) და height (სიმაღლე). ზომის აღწერისათვის არსებობს სპეციალური კლასი Dimension (ზომა), რომელშიც განსაზღვრულია ორი public int ტიპის ველი width და height, ასევე რამდენიმე დამხმარე მეთოდი.

კომპონენტის ზომის დაყენება შესაძლებელია მეთოდით setSize(), რომელმაც არგუმენტათ შეიძლება მიიღოს მთელი რიცხვების წყვილი ან Dimension. მიმდინარე ზომების გაგება შესაძლებელია მეთოდით getSize(), რომელიც აბრუნებს Dimension-ს ან მეთოდებით getWidth() და getHeight().

კომპონენტის მდებარეობა და ზომა ერთობლივად მიუთითებს მის საზღვრებს. კომპონენტის მიერ დაკავებული არე შეიძლება აღიწეროს ან მთელი რიცხვების ოთხეულით ან Point და Dimension კლასების ეგზემპლარებით ან სპეციალური კლასით Rectangle (მართკუთხედი).

ობიექტის საზღვრების მითითება შესაძლებელია მეთოდით setBounds(), რომელმაც შესაძლებელია მიიღოს ოთხი რიცხვი ან მართკუთხედი. მიმდინარე მნიშვნელობის გაგება შესაძლებელია მეთოდით getBounds(), რომელიც აბრუნებს Rectangle-ს.

ხილვადობა

არსებული კომპონენტი შესაძლებელია იყოს, როგორც ხილვადი მომხმარებლისათვის, ასევე უხილავი. ეს თვისება აღიწერება ბულის ტიპის პარამეტრით visible. მმართველი მეთოდებია setVisible() და isVisible().

წვდომა

შესაძლებელია კომპონენტი ეკრანზე აისახებოდეს, ხილვადიც იყოს, მაგრამ მომხმარებელთან ურთიერთობა აკრძალული იყოს. აკრძალვის შედეგად კომპონენტი ვერ შეძლებს კლავიატურის ან მაუსის მიერ წარმოქმნილ ხდომილებების აღქმას და დამუშვებას. ასეთ კომპონენტს ეწოდება disabled. თუ კომპონენტი აქტიურია მას უწოდებენ enabled. ამ თვისების ცვლილებისათვის გამოიყენება მეთოდები setEnabled(), რომელიც ღებულობს ბულის ტიპის პარამეტრს

ან აბრუნებს მდგომარეობის ბინარულ მნიშვნელობას isEnabled().

ფერები

ცხადია, თანამედროვე გრაფიკული ინტერფეისის ასაგებად აუცილებელია ფერებთან მუშაობა. კომპონენტს ფერის აღწერისათვის გააჩნია ორი თვისება - foreground და background ფერები. პირველი თვისება მიუთითებს რა ფერით გამოვიდეს წარწერები, გაივლოს ხაზები და ა.შ. მეორე თვისება მიუთითებს ფონის ფერს, რომლითაც იფერება კომპონენტის მიერ დაკავებული მთელი არე, მანამ სანამ გამოიხატება კომპონენტის გარეგნობა. AWT-ში ფერის მისათითებლად გამოიყენება სპეციალური კლასი Color. ფერი მოიცემა 3 მთელრიცხვა მახასიათებლით, რომელიც შეესაბამება RGB მოდელს - წითელი, მწვანე, ცისფერი. თითოეული მათგანის მნიშვნელობა შეიძლება იყოს 0÷255 დიაპაზონში. შედეგად (0,0,0) შეესაბამება შავს, ხოლო (255,255,255) - თეთრს. სხვა ფერები მიიღება ამ სამეულის სხვადასხვა მნიშვნელობების კომბინაციებით. foreground თვისებასთან სამუშაოდ გამოიყენება მეთოდები setForeground() და getForeground(). ხოლო background თვისებასთან setBackground() და getBackground().

შრიფტი

ვინაიდან კომპონენტის გამოსახულება შეიძლება შეიცავდეს წარწერებს, ამიტომ საჭიროა არსებობდეს თვისება შრიფტის აღსაწერად. AWT-ში აღწერისათვის არსებობს კლასი Font,

რომელი შეიცავს სამ პარამეტრს - შრიფტის სახელს, ზომას და სტილს. შრიფტებთან მუშაობისათვის არსებობს მეთოდები setFont() და getFont().

Container

არსებობს კიდევ ერთი მნიშვნელოვანი სხვა სახის თვისება. პრაქტიკულად ყოველთვის სამომხმარებლო ინტერფეისი შედგება ერთზე მეტი კომპონენტისაგან. დიდ პროგრამებში ისინი საკმაოდ ბევრია. მათთან მუშაობის ორგანიზებისათვის კომპონენტები ერთიანდებიან რომელიმე კონტეინერში. AWT-ში არსებობს კლასი, რომელსაც ასევე ჰქვია - Container. აღსანიშნავია, რომ კომპონენტი შეიძლება ერთდროულად იმყოფებოდეს მხოლოდ ერთ კონტეინერში. თუ კომპონენტს სხვა კონტეინერში გავაერთიანებთ იგი წინა კონტეინერიდან ამოვარდება. აბზაცის თავში ნახსენებ თვისებას ეწოდება parent და მისი გამოისობით კომპონენტმა ყოველთვის „იცის“ რომელ კონტეინერში იმყოფება.

Container კლასი წარმოადგენს Component-ის შვილობილ კლასს, ამიტომ მას გადაეცემა გრაფიკული ელემენტის ყველა თვისება.

ამ კლასის ძირითადი დანიშნულებაა სხვადასხვა კომპონენტების დაჯგუფება. ამ მიზნით Container კლასში გამოცხადებულია მთელი რიგი მეთოდები. დამატებისათვის გამოიყენება მეთოდი add(), ამოგდებისათვის მეთოდი remove() და removeAll(). დამატებული ელემენტები ინახება მოწესრიგებულ სიაში, ამიტომ წასაშლელად (ამოსაგდებად) შესაძლებე-

ლია მივუთითოთ კავშირი კომპონენტზე, რომლის ამოგდებაც გვინდა, ან მისი რიგითი ნომერი კონტეინერში. კლასში ასევე აღწერილია კონტეინერში მყოფი კომპონენტების მიღების მეთოდები, ყველა მათგანი საკმაოდ ცხადია, ამიტომ მათ მოკლე აღწერით ჩამოვთვლით:

getComponent(int n) - აბრუნებს n რიგითი ნომრით მითითებულ კომპონენტს;

getComponents() - აბრუნებს ყველა კომპონენტს მასივის სახით;

getComponentCount() - აბრუნებს კომპონენტების რაოდენობას;

getComponentAt(int x, int y) ან (Point p) - კომპონენტს, რომელიც შეიცავს მითითებულ წერტილს;

findComponentAt(int x, int y) (Point p) - აბრუნებს ხილვად კომპონენტს, რომელიც შეიცავს მითითებულ წერტილს.

როგორც ზემოთ აღვნიშნეთ, კომპონენტის მდებარეობა (location) მოიცემა ზედა მარცხენა კუთხის კოორდინატებით. მნიშვნელოვანია, რომ ეს მნიშვნელობები აითვლება კონტეინერის ზედა მარცხენა კუთხიდან, რომელიც ამგვარად წარმოადგენს კოორდინატთა სისტემის ცენტრს კონტეინერში შემავალი ყოველი კომპონენტისათვის. თუ საჭიროა კომპონენტის ეკრანზე განლაგება მისი კონტეინერის გათვალისწინების გარეშე, მაშინ შეიძლება ვისარგებლოთ მეთოდით getLocationOnScreen().

მემკვიდრეობითობის გამო კონტეინერებს აქვთ თვისება size(). ეს ზომა მოიცემა ჩადებული კომპონენტის ზომისა და მდებარეობის მიუხედავად, ამიტომ კომპონენტები შესაძლებელია განლაგებული იყვნენ თავისი კონტეინერის მიმართ ნაწილობრივ ან მთლიანად გარეთ.

ვინაიდან კონტეინერი წარმოადგენს Component-კლასის მემკვიდრეს, იგი, თვითონაც არის კომპონენტი, ამიტომ იგი შეიძლება დაემატოს სხვა, ზემოთ მდგომ კონტეინერს. ამავე დროს კომპონენტი შეიძლება იმყოფებოდეს მხოლოდ ერთ კონტენერში. ეს ნიშნავს, რომ მომხმარებლის რთული ინტერფეისის ყველა ელემენტი ერთიანდება ხისებურ იერარქიულ სტრუქტურაში. კლასების ასეთი ორგანიზება არამარტო აადვილებს მათზე ოპერაციებს, არამედ იძლევა მთლიანი AWT-ს მუშაობის ძირითად თვისებებს.

კლასი Panel

Panel კლასი Container კლასის კონკრეტული ქვეკლასია. ის რაიმე ახალ მეთოდებს არ ამატებს. იგი წარმოადგენს Container კლასის რეალიზაციას. Panel კლასის ობიექტი – ესაა ფანჯარა, რომელსაც არ აქვს: სათაურის არე, მენიუს სტრიქონი და გარე ბორდიური. Panel-ობიექტს შეიძლება დაუმატოთ სხვა კომპონენტები add() მეთოდის საშუალებით. მას შემდეგ რაც ეს ელემენტები დაემატება, შეგვიძლია მათი პოზიციონირება, ზომების შეცვლა setLocation(), setSize() და setBounds() მეთოდების საშუალებით, რომლებიც განსაზღვრულია Component კლასში.

კლასი Window

თანამედროვე ოპერაციული სისტემების გრაფიკულ ინტერფეისთან მუშაობის გამოცდილებიდან გამომდინარე, ჩვენ მიჩვეული ვართ, რომ ყოველი გამოყენებითი პროგრამა ფლობს ერთ ან რამდენიმე ფანჯარას. კლასი Window წარმოადგენს Java-ში წარმოქმნილი ყველა ფანჯრის ბაზისურ კლასს. იგი ასევე წარმოადგენს ოპერაციული სისტემის იმ ფანჯარასთან ინტერფეისს, რომელიც ემსახურება ყველა პროგრამის ფანჯარას.

Window კლასის ეგზემპლარებს არ გაჩნიათ არც ჩარჩო, არც დახურვისა და მინიმიზაციის ღილაკები. ანუ ბაზური ფანჯარა არ წარმოადგენს დამოუკიდებელ ერთეულს, ის უნდა დაკავშირდეს სხვა ფანჯრებთან. როგორც წესი, გამოიყენება Window კლასის მემკვიდრე კლასები Frame და Dialog.

კლასი Frame

Frame კლასის დანიშნულებაა შექმნას პროგრამის სრულ-ყოფილი ფუნქციონირების ფანჯარა, რომელსაც ექნება ჩარჩო, სათაურის ზოლი, დახურვის, მინიმიზაციისა და მაქსიმაზაციის ღილაკები. ფრეიმი, როგორც წესი, პროგრამის მთავარი ფანჯარაა და იგი თავიდან იქმნება უხილავი, რათა მოხდეს მისი ყველა პარამეტრის მორგება (ინიციალიზაცია, დაყენება), დაემატოს ყველა ჩადგმული კონტეინერი და კომპონენტი, მხოლოდ ამის მერე მოხდეს მისი ასახვა

(გამოჩენა) ეკრანზე დასრულებული სახით. ამ კლასს აქვს ორი კონსტრუქტორი:

Frame()

Frame(String <სათაური>)

პირველი როგორც ხედავთ ცარიელი, ხოლო მეორე ტექსტური პარამეტრის სახით ღებულობს ფრეიმის დასახელებას. როგორც კონსტრუქტორიდან ჩანს, ფანჯრის საწყისი ზომის წინასწარ დაყენება არ შეიძლება, ამიტომ ფანჯრის შექმნის შემდეგ უნდა შევცვალოთ ფანჯრის ზომა ჩვენთვის საჭირო მნიშვნელობებით. ასევე შესაძლებელია Frame-ს მშობელ კლასებში განხილული სხვადასხვა მეთოდების გამოყენება. მაგალითი

```
import java.awt.*;

public class Frame1 {
   public static void main(String[] args) {
      Frame f = new Frame("Test frame");
      f.setSize(400, 300);
      f.setVisible(true);
   }
}
```

მიაქციეთ ყურადღება, რომ შექმნილი ფანჯარა არ იხურება სათაურის ზოლის მარჯვენა ზედა კუთხეში დახურვის ღილაკით. როგორც ვხედავთ, შესაძლებელია ფრეიმ-ფანჯრის შექმნა, უბრალოდ, Frame-ობიექტის შექმნით, მაგრამ ასეთი ფანჯრის შემთხვევაში არ შეგვიძლია მივიღოთ

და დავამუშაოთ ხდომილებები, რომლებიც წარმოიშვება ფანჯრის შიგნით ან გამოვიტანოთ ინფორმაცია ფანჯარაში. ამიტომ ძირითადად უნდა შევქმნათ Frame კლასის ქვეკლასი და ამ ქვეკლასში მოვახდენოთ Frame კლასში განსაზღვრული ხდომილებების დამმუშავებელი მეთოდების ხელახალი გამოცხადება (გადაფარვა). ეს გზა შემდგომში იქნება განხილული.

თუ Frame კლასი პროგრამის ძირითადი ფანჯრის შესაქმნელად გამოიყენება, Dialog კლასის ეგზემპლარები საშუალებას გვაძლევს გავხსნათ დამატებითი ფანჯრები მომხმარებელთან ურთიერთქმედებისათვის. ეს შესაძლოა საჭირო გახდეს კრიტიკული შეტყობინებების გამოსატანად, პარამეტრების შესატანად და ა.შ. დიალოგის ფანჯარას აქვს სტანდარტული გაფორმება - დასახელების ზოლი, ჩარჩო. დასახელების ზოლის მარჯვენა კუთხეში იმყოფება მხოლოდ დახურვის ღილაკი. ვინაიდან Dialog არ წარმოადგენს დამოუკიდებელ ფანჯარას, კონსტრუქტორში საჭიროა გადაეცეს მიმთითებელი მშობლიურ ფრეიმზე ან სხვა დიალოგის ფანჯარაზე. ასევე შესაძლებელია გადაეცეს სათაურის ტექსტი. ფრეიმის მსგავსად დიალოგის ფანჯარაც დასაწყისში იქმნება უხილავი.

მართვის ელემენტები

მართვის ელემენტები(controls) – ესაა კომპონენტები, რომლებიც მომხმარებელს საშუალებას აძლევს ურთიერთობა დაამყაროს პროგრამასთან. მაგალითად ღილაკები, რედაქტირებადი ველი, სიები, მენიუები, რბიები და სხვა. ფანჯრის

შემცველობასა და მის გამოსახულებას განსაზღვრავს გამოყენებული მართვის ელემენტების კომბინაცია და მათი განლაგება. ფანჯარაში შესაძლებელია კომპონენტები განვალაგოთ ხელით, მაგრამ ეს საკმაოდ შრომატევადი სამუშაოა. ამოტომ ფანჯრისათვის შემოღებულია კომპონირების მენეჯერი, რომლის დანიშნულებაა კომპონენტების ავტომატური განლაგება და პოზიციონირება ფანჯარაში. AWT შეიცავს შემდეგი ტიპის მართვის ელემენტებს:

- ტექსტური ჭდე(Labels);
- ღილაკები(Buttons)
- ტექსტური ველი(Text fields)
- ტექსტური არე(Text areas)
- მონიშვნის “დროშები”(Check boxes)
- სიები ელემენტის არჩევით(Choice lists)
- სიები(Lists)
- “რბიები”(Scroll bars)

ფანჯარას მართვის ელემენტი რომ დაემატოს, საჭიროა ჯერ შეიქმნას სასურველი ელემენტის ეგზემპლიარი და შემდეგ ის დაემატოს ფანჯარას add() მეთოდის გამოყენებით, რომელიც განსაზღვრულია Container კლასში. add() მეთოდს აქვს რამოდენიმე ფორმა, ჩვენ გამოვიყენებთ მის ასეთ ფორმას:

Component add(Component compObj)

სადაც compObj – იმ მართვის ელემენტის ეგზემპლარია, რომელიც გვინდა დავამატოთ. მართვის ელემენტის დამატებისთანავე იგი ავტომატურად გამოჩნდება ეკრანზე ყოველთვის, როცა ეკრანზე აისახება მისი მშობელი ფანჯარა.

ფანჯრიდან მართვის ელემენტის წასაშლელად უნდა გამოვიძახოთ მეთოდი remove(), რომელიც განსაზღვრულია Container კლასში. მისი ზოგადი ფორმაა:

void remove(Component compObj)

სადაც compObj – იმ მართვის ელემენტთან კავშირია, რომლის წაშლაც გვინდა.

removeAll() მეთოდის გამოძახებით შესაძლებელია ფანჯარასთან დაკავშირებული ყველა მართვის ელემენტის წაშლა.

ტექსტური ჭდეების გარდა, რომლებიც ითვლებიან პასიურად, ყველა მართვის ელემენტს შეუძლია ხდომილების გენერირება, როცა მას მიმართავს მომხმარებელი. მაგალითად თუ მომხმარებელი აჭერს ღილაკს , პროგრამას გაეგზავნება შეტყობინება ხდომილებაზე, რომელიც წარმოშვა ამ ღილაკმა. ზოგადად ჩვენი პროგრამა რეალიზაციას უკეთებს ხდომილების შესაბამის ინტერფეისს და შემდეგ ყოველი ელემენტისათვის ქმნის ხდომილების გამომცნობ (მომსმენ) და დამმუშავებელ ბლოკს. როგორც კი ელემენტს მომსმენი ბლოკი შეექმნება, ხდომილებები ავტომატურად მიეწოდება მას. თვითოეულ მართვის ელემენტს აქვს თავისი ინტერფეისი.

ტექსტური ჭდეები

ყველაზე მარტივი მართვის ელემენტია ჭდე(Label). ტექსტური ჭდე – ესაა Label კლასის ობიექტი, რომელიც შეიცავს სტრიქონს, რომელსაც იგი გამოსახავს ფანჯარაში. ჭდე პასიური ელემენტია და მომხმარებელთან დიალოგი არ შეუძლია დაამყაროს. მისი კონსტრუქტორებია:

Label()

Label(String str)

Label(String str, int how)

პირველი ფორმა წარმოშობს ცარიელ ჭდეს, მეორე str სტრიქონიან ჭდეს, რომელიც მარცხნივაა მიჯრილი. მესამე ფორმა წარმოშობს ჭდეს, რომელიც ინიციალიზირებულია str სტრიქონით და გასწორებულია how პარამეტრში მითითებული მნიშვნელობის მიხედვით. how პარამეტრის მნიშვნელობა უნდა იყოს ერთერთი ამ სამი კონსტანტისაგან: Label.LEFT, Label.RIGTH, Label.CENTER.

ტექსტი ჭდეში შეიძლება დავაყენოთ ან შევცვალოთ setText() მეთოდით. getText() მეთოდით შესაძლებელია ჭდის მიმდინარე მნიშვნელობის წაკითხვა. ამ მეთოდების ფორმატია:

void setText(String str)

String getText()

setAlignment() მეთოდის გამოძახებით, შესაძლებელია სტრიქონის გასწორება ჭდის არეს ფარგლებში. მიმდინარე

გასწორების მისაღებად გამოიყენება getAlignment() მეთოდი. ამ მეთოდების ფორმატია:

void setAlignment(int how)

int getAlignment()

სადაც how ზემოთ აღწერილი გასწორების კონსტანტებია.

ღილაკები (Button)

ყველაზე ხშირად გამოყენებადი მართვის ელემენტი ღილაკი – ესაა კომპონენტი, რომელიც შეიცავს ტექსტურ ჭდეს და იწვევს ხდომილებას, როცა მასზე აწვებიან. იგი წარმოადგენს Button კლასის ობიექტს. ამ კლასში განსაზღვრულია ორი კონსტრუქტორი:

Button()

Button(String str)

პირველი ვერსია ქმნის ცარიელ ღილაკს, მეორე – ღილაკი ტექსტური ჭდით.

ღილაკის შექმნის შემდეგ შესაძლებელია ჭდის დაყენება (satLabel()) ან ჭდის მიმდინარე მნიშვნელობის წაკითხვა (getLabel()). ამ მეთოდების ფორმატებია:

void setLabel(String str)

String getlabel()

ხდომილების დამუშავება

როგორც მოგეხსენებათ, გრაფიკული ინტერფეისის მქონე პროგრამაში ზოგჯერ ღილაკზე დაჭერა რაიმე მოქმედების ინიცირებას არ იწვევს, ხოლო ზოგჯერ ხდება რაღაც მოქმედებების წამოწყება. განვიხილოთ ხდომილებების დამუშვების მოდელი. ცხადია, მომხმარებელთან ურთიერთქმედებისათვის არაა საკმარისი ეკრანზე მხოლოდ კომპონენტების განლაგება. AWT (ან Swing) კომპონენტების ნაკრების ხდომილებათა დამუშვების მოდელში თითოეული კომპონენტი ასოცირებულია ერთ ან რამდენიმე დამკვირებელთან. როდესაც კომპონენტზე წარმოიქმნება რაიმე ხდომილება, ეს ეცნობება ყველა დამკვირებელს. დამკვირვებელი ფაქტიურად ესაა ობიექტი, რომელიც „დაინტერესებულია“ ხდომილებით.

AWT (ან Swing) კომპონენტების დამკვირვებლებს უწოდებენ ხდომილების მსმენელებს (Listener). მათ უნდა განახორციელონ ცარიელი ინტერფეისის java.util.EventListener-ის რეალიზაცია და თითქმის ყველა შემთხვევაში მსმენელი ქვეკლასის ინტერფეისი, რომელსაც ერთი მეთოდი მაინც უნდა ჰქონდეს. თითოეული ხდომილება წარმოადგენს java.util.EventObject კლასის ქვეკლასს.ძალიან მწშვნელოვანია ცხადად იქნას განსაზღვრული კომპონენტის რომელ ხდომილებებთანაა ასოცირებული კონკრეტული მსმენელები.

მაგალითად განვიხილოთ კერძო შემთხვევა - როდესაც ხდება ღილაკ Button-ის (ან Swing-ში JButton) არჩევა. ამ მოქმედებაზე პასუხისმგებელია ხდომილება, რომლის გენერირებაც ხდება მომხმარებლის მიერ ღილაკ Button-ის (ან Swing-ში JButton) არჩევისას. თუ რომელიმე ობიექტი დაინტერესებულია ამ ხდომილებით იგი უნდა

დარეგისტრირდეს ღილაკ Button-თან. ActionEvent ხდომილებისათვის რეგისტრაცია ხდება ActionListener-ის ფორმით. მათ შორის სახელები შეთანხმებულია ასეთნაირად - ყოველი ABCEvent ფორმის ხდომილებისათვის ასოცირებულ მსმენელად უნდა იქნას ABCListener მსმენელი, სადაც ABC-ს მაგივრად მითითებული იქნება კონკრეტული ხდომილების ტიპი.

რეგისტრაცია ხდება addActionListener მეთოდის გამოძახებისას. ყოველ დარეგისტრირებულ რეალიზატორს ActionEvent ხდომილების გენერაციის შემთხვევაში ეცნობება ეს ფაქტი. ერთ ხდომილებას შეიძლება ჰქონდეს რამდენიმე დამკვირვებელი. მთელი ეს პროცესი ასეთი მიმდევრობით შეიძლება წარმოვადგინოთ:

-მოცემული ხდომილებისათვის განისაზღვროს კლასი, რომელიც რეალიზაციას უკეთებს მასთან დაკავშირებულ ინტერფეისს. კლასის აღწერაში დამატებული უნდა იქნას „implements ABCListener“ ან შეიქმნას ახალი კლასი, რომელიც ამ ინტერფეისის რეალიზაციას ახორციელებს;

-კლასის აღწერაში მხოლოდ „implements ABCListener“-ის დამატება საკმარისი არაა. საჭიროა ABCListener ინტერფეისის ყველა მეთოდის რეალიზაციის განხორციელება. ზოგ მსმენელს (მაგალითად ActionListener) აქვს მხოლოდ ერთი მეთოდი, ხოლო სხვებს (მაგალითად, WindowListener, იგი გამოიყენება ფანჯრის გადაადგილებისას ან დახურვისას) რამდენიმე;

-ინტერფეისის რეალიზაციის განსაზღვრის შემდეგ უნდა შეიქმნას რეალიზაციის ეგზემპლარი და მოხდეს მისი ასოცირება კომპონენტთან. მხოლოდ ამის მერე შეეძლება

მსმენელს (დამკვირვებელს) მიიღოს შეტყობინება რაიმე ხდომილების წარმოქმნის შესახებ.

შემდეგ პროგრამაში დემონსტრირებულია ღილაკზე ზემოქმედების - „ღილაკის არჩევის“ შესაბამისი ხდომილების დამუშავება:

```
package src;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.WindowConstants;
import javax.swing.SwingUtilities;

public class NewJFrame extendsjavax.swing.JFrame {
   private JButton jButton1;

   public static voidmain(String[] args) {
     SwingUtilities.invokeLater(newRunnable() {
        public void run() {
           NewJFrame inst = newNewJFrame();
           inst.setLocationRelativeTo(null);
           inst.setVisible(true);
        }
     });
   }

   public NewJFrame() {
     super();
     initGUI();
   }

   private void initGUI() {
     try {
```

```
setDefaultCloseOperation(WindowConstants.DISPOSE_O
N_CLOSE);
        getContentPane().setLayout(null);
        {
           jButton1 = new JButton();
           getContentPane().add(jButton1);
           jButton1.setText("jButton1");
           jButton1.setBounds(36, 22, 59, 23);

jButton1.addActionListener(newActionListener() {
              public
voidactionPerformed(ActionEvent evt) {

System.out.println("jButton1.actionPerformed,
event="+evt);
                //TODO add your code for
jButton1.actionPerformed
                jButton1.setText(Hiii!);
              }
           });
        }
        pack();
        setSize(400, 300);
     } catch (Exception e) {
         //add your error handling code here
        e.printStackTrace();
     }
   }

}
```

თითოეული კომპონენტის მიერ წარმოქმნილი შესაძლო ხდომილებების ჩამონათვალი, მათი დასახელებები და სხვა დეტალები მოძიებული უნდა იქნას API-ს აღწერის დოკუმენტაციაში კიდევ ერთხელ დავაკვირდეთ

add/removeABCListener მეთოდების წყვილს, სადაც ABC მიუთითებს ხდომილების ტიპს. თითოეული add/remove მეთოდების წყვილისათვის წარმოიქმნება კონკრეტული ხდომილებები. ხდომილების წარმოქმნის შეტყობინების გზავნილის აკრძალვისათვის შეგვიძლია გამოვიყენოთ მეთოდი removeABCListener.

სარჩევი