Kako naredimo ...

... v programskem jeziku Java

Pripravila: Alenka Kavčič

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
april 2006

KAZALO

1. Uvod

Gradivo je namenjeno študentom prvih letnikov na Fakulteti za računalništvo in informatiko, ki pri predmetih Programiranje 2 in Osnove programiranja 2 spoznavajo naprednejše koncepte programskega jezika Java. Zajema izbrane teme iz programiranja v jeziku Java in skozi primere programov, ki so tudi podrobneje razloženi, prikazuje pristope k reševanju različnih problemov.

To gradivo ni učbenik za programski jezik Java in ni namenjeno samostojnemu učenju jezika, temveč je le dopolnilo drugim učbenikom oziroma zapiskom s predavanj. Tako predpostavlja poznavanje in razumevanje osnovnih konceptov programskih jezikov (kot so spremenljivke in njihova uporaba, zanke, odločitveni stavki, itd.) in objektno usmerjenega programiranja (kot so objekti, članske spremenljivke, metode, konstruktorji, dedovanje, vmesniki, abstraktni razredi, itd.) ter se osredotoči na naprednejše koncepte jezika Java.

Priprava platforme za delo

Zelo priporočljivo je, da v tem gradivu obdelane primere tudi sami preizkusite. To pomeni, da program napišete v računalniku, ga izvedete in preverite njegovo delovanje. Po potrebi kodo programa ustrezno spremenite in preverite, ali se delovanje spremenjenega programa ujema z vašimi predpostavkami.

Izbira delovnega okolja je lahko poljubna. Jezik Java je namreč neodvisen od operacijskega sistema, saj se izvaja znotraj javanskega navideznega stroja (*Java Virtual Machine* ali JVM). Nekaj manjših razlik med različnimi operacijskimi sistemi najdemo le v podrobnostih, kot je na primer zapis datotečne poti, vendar tudi te lahko obvladamo programsko. Ker pa na vajah pri predmetih Programiranje 2 in Osnove programiranja 2 uporabljamo Unix/Linux okolje, se bomo tudi v tem gradivu osredotočili nanj.

Za razvoj javanskih programov priporočamo uporabo razvojnega kompleta JDK (Java2 SE JDK), ki vključuje tudi ustrezen prevajalnik, in preprostega urejevalnika besedil. V tem gradivu bomo uporabljali Java 2 Platform Standard Edition 5.0, čeprav večina primerov deluje tudi v starejših različicah (kjer uporabljamo novosti Jave 5.0, je to posebej poudarjeno).

Java 2 Standard Edition version 5.0 JDK

Za razvoj javanskih programov torej potrebujemo razvojni komplet Java 2 Standard Edition version 5.0 JDK (JDK je kratica za *Java Development Kit*), ki je tudi najprimernejše orodje za učenje Jave. Med drugim vključuje različna orodja JDK (kot je na primer prevajalnik), izvajalno okolje (*Java Runtime Environment* ali JRE) ter knjižnice. Java 2 SE 5.0 JDK lahko brezplačno prenesete s spletnega naslova http://java.sun.com/j2se/.

Dokumentacija

Obsežna dokumentacija JDK zajema podrobneje opisane razrede in njihovo uporabo. Dokumentacijo J2SE Development Kit Documentation 5.0 lahko pregledujemo preko spleta (http://java.sun.com/j2se/1.5.0/docs) ali pa jo v obliki stisnjene ZIP datoteke prenesemo s spletnih strani (http://java.sun.com/j2se/1.5.0/download.jsp) ter jo razširimo v poljubno mapo v našem računalniku.

Nastavitev poti do orodij

Orodja JDK se namestijo v podmapo `bin` tiste mape, v katero smo namestili JDK. Za enostaven dostop do teh orodij moramo nastaviti pot do omenjene mape. To naredimo tako, da spremenimo sistemsko spremenljivko PATH in ji dodamo pot do mape `bin`. Način spreminjanja spremenljivke PATH je odvisen od operacijskega sistema.

Druga sistemska spremenljivka, ki jo nastavimo po potrebi, se imenuje CLASSPATH. Uporablja jo izvajalno okolje JRE, spremenljivka pa določa pot do prevedenih razredov (`.class` datotek). Za naše primere zadostuje, da spremenljivke CLASSPATH nimamo nastavljene (spremenljivka sploh ne obstaja v sistemskem okolju). Če spremenljivka obstaja in je napačno nastavljena, imate lahko težave pri izvajanju programov. Zato vam priporočamo, da v pot CLASSPATH, če le-ta obstaja, dodate tudi pot do tekočega direktorija (`.`).

Razvojna orodja

Čeprav je za učenje jezika Java in razvoj programov iz tega gradiva povsem primeren razvojni komplet JDK, nam pri razvoju obsežnejših projektov ta ne zadostuje več. V tem primeru lahko uporabimo različne razvojne pripomočke (brezplačne ali plačljive), ki ponavadi vključujejo urejevalnik, prevajalnik, izvajalno okolje z možnostjo razhroščevanja ter obsežno dokumentacijo, ter tako omogočajo hitrejši in enostavnejši razvoj programske kode. Primer takih razvojnih orodij, ki jih lahko dobimo na spletu, so *Eclipse* (http://www.eclipse.org), *NetBeans* (http://www.netbeans.org) ali Borlandov *JBuilder* (http://www.borland.com/jbuilder).

Primeri programov

Izvorne kode vseh primerov programov, kjer je označeno tudi ime programa, so priložene temu gradivu. Dobite jih na spletu pod dodatnimi gradivi na domači strani predmeta Osnove programiranja 2 (http://lgm.fri.uni-lj.si/op2/).

Ker nihče ni popoln, tudi to gradivo ni odporno na napake. Zato vas prosim, da najdene napake sporočite na naslov alenka.kavcic@fri.uni-lj.si, da jih bomo lahko popravili. Prav tako so dobrodošli tudi vsi predlogi za nove teme, nove primere, podrobnejše razlage opisanih primerov in podobno.

2. Pisanje, prevajanje in izvajanje programov

Najprej bomo za ogrevanje ponovili, kako napišemo enostaven program v programskem jeziku Java in kaj vse moramo narediti, preden si lahko ogledamo rezultate (delovanje) našega programa.

Celoten postopek razdelimo na tri korake:
- pisanje izvorne kode programa,
- prevajanje programa ter
- izvajanje programa.

Pisanje izvorne kode programa

Vsak program začnemo s pisanjem izvorne kode, ki jo lahko napišemo v poljubnem urejevalniku besedila. Pomembno je le, da nam urejevalnik besedila omogoča shranjevanje vsebine kot navadno besedilo (*plain text*). Ponavadi je preprost urejevalnik že sestavni del operacijskega sistema. Primera takih preprostih urejevalnikov sta *Beležnica* (*Notepad*) v Windows okolju ali *vi* (ali izboljšana različica *vim*) v Linux/Unix okolju. Seveda je uporaba določenega urejevalnika stvar okusa in navad vsakega posameznika. Pomembno je le, da je urejevalnik enostaven za uporabo in da ga dovolj dobro poznate.

Naj kot prvi primer napišemo enostaven program, ki na zaslon izpiše besedo `Pozdravljeni!` in skoči v novo vrstico. Odpremo urejevalnik besedil in vanj vpišemo naslednje vrstice:

```
public class Primer
{

  public static void main(String[] args)
  {
    System.out.println("Pozdravljeni!\n");
  }

}
```

Program `Primer.java`

Vsebino shranimo kot navadno besedilo v datoteko z imenom `Primer.java` (pri tem pazimo, da je ime datoteke res tako, kot smo ga podali tu). Imena datotek, ki vsebujejo izvorno kodo Java, morajo vedno imeti podaljšek `.java`, samo ime datoteke pa mora biti enako imenu razreda, v katerem je definirana metoda `main`. Pri tem moramo paziti, da se ime datoteke popolnoma ujema z imenom razreda (velika začetnica), saj Java loči velike in male črke.

Prevajanje programa

Program prevedemo v ukazni lupini (odpremo terminalsko okno), kjer za prevajanje uporabimo javanski prevajalnik, ki ga pokličemo z ukazom `javac`, kateremu sledi ime datoteke z izvorno kodo.

Naš program, katerega izvorna koda se nahaja v datoteki `Primer.java`, prevedemo z ukazom:

```
javac Primer.java
```

Rezultat prevajanja (če seveda med prevajanjem ni prišlo do napake zaradi napačno napisane kode) je javanska vmesna koda (*Java bytecode*), ki se zapiše v datoteko, katere ime je enako imenu datoteke z izvorno kodo, podaljšek pa je `.class`. V našem primeru se ob prevajanju ustvari datoteka `Primer.class`.

Izvajanje programa

Tudi izvajanje programa poteka v ukazni lupini. Vmesno kodo, ki smo jo ustvarili v prejšnjem koraku, lahko izvajamo v posebnem, izvajalnem okolju. Zaženemo ga z ukazom `java`, kateremu sledi ime prevedene datoteke brez podaljška. V našem primeru bi program pognali z ukazom:

```
java Primer
```

In kaj naredi naš program? Najprej izpiše:

```
Pozdravljeni!
```

in nato skoči v novo vrstico ter konča.

3. Zbirke podatkovnih struktur

Ogrodje zbirk (*collection framework*) zajema celotno strukturo razredov (skupaj z vmesniki in abstraktnimi razredi), ki določajo delo z zbirkami podatkovnih struktur. Java nudi v ta namen temeljni vmesnik `Collection`, ki povzema značilnosti vseh zbirk. Iz njega sta izpeljana seznam `List` in množica `Set`. Slednja je lahko tudi urejena, kar določa vmesnik `SortedSet`.

Vse zbirke omogočajo določene temeljne operacije, kot so:
- dodajanje elementa v zbirko (`add()`),
- odstranjevanje elementa iz zbirke (`remove()`),
- preverjanje, ali zbirka vsebuje določen element (`contains()`),
- preverjanje, ali je zbirka prazna (`isEmpty()`),
- preverjanje velikosti zbirke (`size()`) ali
- pretvorbo zbirke v polje elementov (`toArray()`).

Seznami

Seznami so zbirke elementov v znanem zaporedju, torej ima vsak element točno določeno mesto v seznamu. V tem so seznami pomensko podobni poljem, saj vsakemu elementu pripada natančno določen položaj (indeks). Vendar pa seznami nimajo vnaprej določene dolžine in se njihova dolžina lahko dinamično spreminja (polja tega ne omogočajo). Seznami omogočajo hranjenje podvojenih elementov.

Sezname lahko izvedemo s prilagodljivim poljem `ArrayList` ali pa s povezanim seznamom `LinkedList`.

Poglejmo si uporabo seznama `ArrayList` na primeru. Seznam bomo uporabili za shranjevanje naključnega števila celih števil, prikazali uporabo operacij za dodajanje elementa v seznam in za odstranjevanja elementa iz seznama ter za izpis vseh elementov seznama.

Ogrodje zbirk (vsi razredi in vmesniki) je opisano v paketu `java.util`, zato moramo na začetki programa napovedati uporabo tega paketa:

```
import java.util.*;
```

Seznam deklariramo kot objekt razreda `ArrayList`, kateremu v oklepajih <> podamo tip elementov, ki jih hrani (slednje, tako imenovana generična koda, je novost Jave 5.0). Tip mora biti določen z razredom, saj seznam `ArrayList` lahko hrani le objekte. V našem primeru, ko gre za cela števila (tip `int`) smo torej podali razred `Integer`:

```
ArrayList<Integer> sez;
```

Klic konstruktorja ustvari prazen seznam z začetno kapaciteto desetih elementov:

```
sez = new ArrayList<Integer>();
```

Nov element dodamo v seznam s pomočjo metode `add()`, ki doda element na konec seznama. Objekt, ki ga dodajamo, podamo kot argument metode. Zato moramo pred tem iz primitivnega tipa `int` ustvariti objekt `Integer` z isto vrednostjo. Če želimo dodati vrednost `5`, to naredimo v dveh korakih:

```
Integer i = new Integer(5);
sez.add(i);
```

ki pa ju lahko tudi združimo:

```
sez.add(new Integer(5));
```

Če metodi `add()` dodamo še en argument, to je pozicijo novega elementa, dosežemo vrivanje elementa v seznam na točno določeno mesto. Število `5` dodamo na prvo mesto v seznamu z naslednjim stavkom:

```
sez.add(0, new Integer(5));
```

Pri dodajanju novih elementov v seznam pa lahko uporabimo še eno novost, ki jo prinaša Java 5.0, to je samodejno pretvarjanje primitivnih tipov (*boxing/unboxing*). To pomeni, da programerju ni potrebno skrbeti za pretvorbo primitivnega tipa `int` v ovijalni tip `Integer`, temveč za ustrezno pretvorbo poskrbi kar sam prevajalnik (seveda moramo prevajati s prevajalnikom različice 1.5 ali 5.0). Tako lahko oba stavka za dodajanje elementa zapišemo krajše kot:

```
sez.add(5);
sez.add(0, 5);
```

Element na določeni poziciji odstranimo iz seznama z metodo `remove()`, kateri kot argument podamo pozicijo elementa v seznamu. Prvi element seznama torej odstranimo s stavkom:

```
sez.remove(0);
```

Seveda je pred tem smiselno preveriti, ali je v seznamu sploh kakšen element. Metoda `isEmpty()` vrne `true`, če je seznam prazen:

```
if(!sez.isEmpty())
  sez.remove(0);
```

Kadar želimo izbrisati iz seznama vse elemente, lahko uporabimo metodo `clear()`, ki počisti seznam:

```
sez.clear();
```

Za izpis seznama bomo napisali svojo metodo, ki se bo sprehodila preko vseh elementov seznama in izpisovala njihove vrednosti. Naj se metoda imenuje `izpisiSeznam`, kot argument pa prejme referenco na seznam, katerega elemente želimo izpisati. V našem primeru gre za seznam `ArrayList` objektov `Integer`:

```
public static void izpisiSeznam(ArrayList<Integer> s)
```

Telo metode sestavlja `for` zanka, ki gre preko vseh indeksov v seznamu (od `0` do `size()`) ter za vsakega od indeksov pokliče metodo `get()`, ki vrne element s podanim indeksom. Vrednost tako dobljenega elementa pretvorimo v niz in jo izpišemo:

```
for(int i=0; i<s.size(); i++)
  System.out.print(s.get(i).toString() + "  ");
```

Seveda se lahko tudi v tej zanki opremo na novosti Jave 5.0 in jo malo poenostavimo. Kot prvo izkoristimo samodejno ovijanje primitivnih tipov in metodo `printf`, ki omogoča lažje formatiranje izpisa:

```
for(int i=0; i<s.size(); i++)
  System.out.printf(" %d ", s.get(i));
```

Naslednja uporabna novost Jave 5.0 pa je izboljšana zanka `for`, ki omogoča enostavnejši (in tudi bolj pregleden) prehod preko vseh elementov zbirke. Zapišemo jo tudi z rezervirano besedo `for`, kateri v oklepajih sledi tip elementa, znak `:` in ime zbirke. V našem primeru bi nova `for` zanka izgledala takole:

```
for(int i: s)
  System.out.printf(" %d ", i);
```

Zapisano sintakso bi lahko prebrali kot “za vsak element `i`, ki je tipa `int`, v zbirki `s`”. Pri tem moramo opozoriti, da na tak način lahko elemente zbirke le pregledujemo, ne moremo pa jih spreminjati.

Še korak naprej k poenostavitvi izpisa elementov seznama po lahko naredimo, če uporabimo preobloženo metodo `toString()` razreda `ArrayList` (ta jo podeduje od razreda `AbstractCollection`). Metoda vrne niz vseh elementov seznama po vrsti, ločenih z vejico, v oglatih oklepajih. Taka predstavitev seznama je za nas povsem primerna, zato jo bomo uporabili pri izpisu:

```
System.out.println("Seznam vsebuje naslednje elemente: " + s);
```

Razred `ArrayList` ima na zalogi še cel kup drugih metod, ki jih tukaj nismo omenili, lahko pa si jih pogledate v JDK dokumentaciji.

Sestavimo sedaj program, ki bo prikazoval delo s seznamom. V programu bomo najprej s pomočjo generatorja naključnih števil izbrali število med `0` in `MAX` (ki je konstanta) ter napolnili seznam po vrsti s števili od `0` do izbranega (naključnega) števila.

```
int meja = (int) (Math.random()*MAX);
for(int i=0; i<=meja; i++)
  sez.add(i);
```

Potem uporabimo različne operacije nad seznamom: odstranimo prvi element seznama (če le-ta ni prazen), dodamo število `715` na začetek seznama ter na koncu seznam še izpraznimo (odstranimo vse elemente). Po vsaki operaciji tudi izpišemo vsebino seznama in trenutno število elementov v seznamu (metoda `izpisiSeznam()`).

Datoteka `Seznam.java` z izvorno kodo programa, kjer smo upoštevali že vse omenjene poenostavitve kode:

```
import java.util.*;

public class Seznam
{
  private static final int MAX = 15;

  public static void main(String[] args)
  {
    ArrayList<Integer> sez = new ArrayList<Integer>();
    int meja = (int) (Math.random()*MAX);
    for(int i=0; i<=meja; i++)
      sez.add(i);
    izpisiSeznam(sez);
    if(!sez.isEmpty())
      sez.remove(0);
    izpisiSeznam(sez);
    sez.add(0, 715);
    izpisiSeznam(sez);
    sez.clear();
    izpisiSeznam(sez);
  }

  public static void izpisiSeznam(ArrayList<Integer> s)
  {
    System.out.println("Seznam vsebuje naslednje elemente: " + s);
    System.out.println("Število elementov v seznamu: " + s.size());
    System.out.println();
  }
}
```

Program `Seznam.java`

Za primerjavo napišimo še podobno kodo, ki je združljiva s starejšimi različicami Jave (v kateri ni novosti Jave 5.0). Najdete jo v datoteki `Seznam1.java`.

```
// Združljivo z Java 1.4
import java.util.*;

public class Seznam1
{
  private static final int MAX = 15;

  public static void main(String[] args)
  {
    ArrayList sez = new ArrayList();
    int meja = (int) (Math.random()*MAX);
    for(int i=0; i<=meja; i++)
      sez.add(new Integer(i));
    izpisiSeznam(sez);
    if(!sez.isEmpty())
      sez.remove(0);
    izpisiSeznam(sez);
    sez.add(0, new Integer(715));
    izpisiSeznam(sez);
    sez.clear();
    izpisiSeznam(sez);
  }

  public static void izpisiSeznam(ArrayList s)
  {
    System.out.println("Seznam vsebuje naslednje elemente: " + s);
    System.out.println("Število elementov v seznamu: " + s.size());
    System.out.println();
  }
}
```

Program Seznam1.java

Množice

Množica je primer zbirke, ki ne more vsebovati podvojenih elementov. Opisuje jo vmesnik `Set`, izvedemo pa jo lahko z zgoščeno tabelo (razred `HashSet`) ali z uravnoteženim drevesom (razred `TreeSet`). Prva izvedba je primernejša za neurejeno množico za splošno rabo, zato jo bomo uporabili tudi v našem primeru.

Iz nizov, ki so podani kot argumenti programa, bomo sestavili množico besed. Ker se podvojene besede ne shranjujejo v množico, bomo na koncu z izpisom elementov množice dobili seznam vseh različnih besed.

Množico deklariramo kot objekt razreda `HashSet`, kateremu v oklepajih <> podamo tip elementov, ki jih hrani (generična koda iz Jave 5.0). Tip mora biti določen z razredom, saj množica `HashSet` lahko hrani le objekte. Naši objekti bodo tipa `String`, saj bomo hranili besede (podane argumente):

```
HashSet<String> m = new HashSet<String>();
```

Za vstavljanje elementov v množico uporabimo metodo `add()`. Ta metoda doda podani element k elementom množice, če elementa še ni v množici. V primeru uspešnega

dodajanja metoda vrne vrednost `true`. V zanki se sprehodimo preko polja nizov `args` ter v množico dodajamo vsak argument posebej:

```
for(int i=0; i<args.length; i++)
  if(!m.add(args[i]))
    System.out.println("Podvojena: " + args[i]);
```

V primeru, da element ni bil uspešno dodan v množico (kar pomeni, da gre za podvojen element), smo ta podvojeni element (besedo) tudi izpisali.

Na koncu smo izpisali vse elemente tako zgrajene množice. Za izpis množice smo se tudi tukaj oprli na preobloženo metodo `toString()` razreda `HashSet` (tudi ta jo podeduje od razreda `AbstractCollection`), ki poskrbi za primeren izpis elementov množice. Metoda `size()` tudi tukaj vrne število elementov v množici:

```
System.out.println("Skupaj je bilo " + m.size() + " različnih
besed: " + m);
```

Datoteka `Mnozica.java` hrani izvorno kodo tega primera:

```
import java.util.*;

public class Mnozica
{
  public static void main(String[] args)
  {
    HashSet<String> m = new HashSet<String>();
    for(int i=0; i<args.length; i++)
      if(!m.add(args[i]))
        System.out.println("Ponovna pojavitev besede <" + args[i] + ">");
    System.out.println("Skupaj je bilo " + m.size() + " različnih besed: " + m);
  }
}
```

Program `Mnozica.java`

Pri izvajanju programa ne pozabite podati argumentov (poljubne besede), kajti brez njih bo množica vedno prazna. Primer klica programa:

```
java Mnozica prva druga prva tretja beseda druga
```

Tudi pri tem primeru si za primerjavo poglejmo še kodo, ki je združljiva s starejšimi različicami Jave (brez novosti Jave 5.0). Najdete jo v datoteki `Mnozica1.java`.

```
// Združljivo z Java 1.4
import java.util.*;

public class Mnozica1
{
  public static void main(String[] args)
  {
```

```
    HashSet m = new HashSet();
    for(int i=0; i<args.length; i++)
      if(!m.add(args[i]))
        System.out.println("Ponovna pojavitev besede <" + args[i] + ">");
    System.out.println("Skupaj je bilo " + m.size() + " različnih besed: " + m);
  }
}
```

Program Mnozica1.java

4. Grafika

Platforma JFC (*Java Foundation Classes*) vključuje tudi dva paketa knjižnic za gradnjo grafičnih uporabniških vmesnikov (*Graphical User Interface* ali GUI), to sta AWT (*Abstract Windowing Toolkit*) in Swing. Prvi je starejši in je hkrati osnova grafičnih gradnikov. Drugi pa je novejši in nudi grafične elemente, katerih videz je neodvisen od okolja operacijskega sistema. Zato bomo pri gradnji grafičnih vmesnikov raje uporabljali Swing komponente.

Osnovni gradniki Swing so izpeljani iz gradnikov AWT, na primer razred `Frame` iz AWT je osnova razreda `JFrame` iz Swing (imena razredov Swing se začnejo s črko `J`).

Programi z GUI so dogodkovno vodeni (*event driven*), saj se odzivajo na akcije uporabnika (kot je na primer klik na gumb). Za razliko od postopkovnega programiranja (*procedural programming*) pri dogodkovno vodenih programih pišemo poleg kode za izgradnjo grafičnega vmesnika le kodo, ki pomeni odziv na različne dogodke. V Javi uporabljamo za to poslušalce (*listeners*), ki si jih bomo ogledali enem naslednjih razdelkov.

Preprosto okno

Začnimo z najpreprostejšim programom, ki na zaslonu prikaže grafično okno. Program je zelo enostaven in nima (še) nobene praktične uporabnosti.

Osnova javanskega programa z grafičnim uporabniškim vmesnikom je samostojno okno, objekt razreda `JFrame`, ki ima vlogo vsebnika, v katerega lahko postavimo druge gradnike.

Razred `JFrame` se nahaja v paketu `javax.swing`, zato najprej napovemo uporabo tega paketa (kar vseh razredov iz paketa):

```
import javax.swing.*;
```

Kot vsi javanski programi mora tudi ta program definirati nek osnovni razred, v našem primeru razred `Okno`, v katerem moramo definirati metodo `main`, saj se program začne izvajati v tej metodi.

V `main` metodi najprej ustvarimo nov objekt razreda `JFrame`:

```
JFrame okno = new JFrame();
```

nato pa ta objekt prikažemo na zaslonu (naredimo viden) s klicem metode `setVisible(true)`.

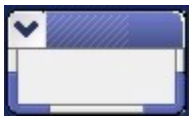
Program `Okno.java` v prvi različici izgleda takole:

```
import javax.swing.*;

public class Okno
{
  public static void main(String[] args)
  {
    JFrame okno = new JFrame();
    okno.setVisible(true);
  }
}
```

Program Okno.java

Ko program `Okno` poženemo, se zgoraj levo na zaslonu prikaže okno, podobno tistemu na sliki 4.1. Ker oknu nismo določili velikosti, se izriše najmanjše možno okno.



Slika 4.1: Najpreprostejše okno (`Okno.java`).

Po izrisu okna je program stopil v zanko, v kateri čaka na dogodek. Oknu lahko spremenimo velikost, ga minimiziramo, maksimiziramo ali zapremo.

Gumb za zapiranje okna sicer deluje (zapre okno), a ob zapiranju okna ne konča našega programa, ker za to nismo napisali ustrezne kode. Zato moramo po zaprtju okna še prekiniti delovanje programa (v ukazni lupini pritisnemo Ctrl+C).

Problem rešimo enostavno, če dodamo naslednjo vrstico kode:

```
okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Metoda `setDefaultCloseOperation()` določi, kot pove že njeno ime, privzeto operacijo, ki se izvede ob zaprtju okna. Kot argument ji podamo celo število, ki določa to operacijo. Najlažje je, če uporabimo kar že definirane konstante, kot je na primer `EXIT_ON_CLOSE` iz razreda `JFrame`, ki določa, da se ob zaprtju okna program konča z uporabo metode `exit` razreda `System`.

Okno lahko tudi poimenujemo (določimo naslov okna), če konstruktorju kot argument podamo niz z imenom:

```
JFrame okno = new JFrame("Moje okno");
```

Že ustvarjenemu oknu pa lahko določimo (ali spremenimo) ime s pomočjo metode `setTitle(ime)`.

Oknu spremenimo oziroma določimo velikost s klicem metode `setSize()`, ki ji podamo širino in višino okna v pikah:

```
okno.setSize(200, 100);
```

Program `Okno1.java` v nekoliko dopolnjeni različici je naslednji:

```
import javax.swing.*;

public class Okno1
{
  public static void main(String[] args)
  {
    JFrame okno = new JFrame("Moje okno");
    okno.setSize(200, 100);
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.setVisible(true);
  }
}
```

Program Okno1.java

Rezultat prikazuje slika 4.2.

Slika 4.2: Nekoliko popravljeno okno (`Okno1.java`).

Grafične komponente

Program z grafičnim vmesnikom lahko vključuje vrsto elementov oz. gradnikov (*components*), kot so okno (*window*), gumb (*button*), stikalo (*checkbox* in *radio button*), vnosno polje (*text field*), seznam (*choice* in *list*), oznaka (*label*), menujska vrstica (*menu bar*), orodna vrstica (*tool bar*) in drugi. Zanimiv gradnik je vsebnik (razred `Container`), ki lahko vsebuje druge gradnike.

Vsebniki so lahko osnovni ali vmesni. Osnovni vsebniki, kot sta na primer `JFrame` ali `JApplet`, določajo samostojna okna in tako predstavljajo osnovni gradnik, zato jih ne moremo vključevati v druge vsebnike. Vsak grafični program mora vsebovati vsaj en osnovni vsebnik. Vmesni vsebniki, kot je na primer `JPanel`, pa ponuja površino, na

katero polagamo gradnike, ter s tem omogočajo združevanje gradnikov v skupine tako, da jih lahko uporabljamo kot en sam gradnik.

Ostali gradniki, kot so `JButton`, `JTextField` ali `JLabel`, ne morejo vsebovati drugih komponent in morajo biti vsebovani v katerem od vsebnikov.

Hierarhijo razredov Swing si lahko ogledate v dokumentaciji JDK.

Okno s komponentami

Dodajmo v naše okno še en nov gradnik, recimo, da je to nov gumb. Gumb ustvarimo s klicem konstruktorja, s katerim hkrati določimo tudi vsebino napisa na gumbu:

```
JButton gumb = new JButton("Klikni me!");
```

Potem moramo gumb tudi dodati vsebini okna. To naredimo s pomočjo metode `add(gumb)`, kateri kot argument podamo gumb, ki ga dodajamo. Vendar pa gradnikov (pri Swing vsebnikih) ne moremo dodati neposredno v osnovni vsebnik, temveč jih lahko dodamo le na delovno površino vsebnika. Vsak vsebnik namreč ponuja delovno površino (*content pane*), na katero potem razvrščamo ostale gradnike. Tako moramo najprej pridobiti referenco na delovno površino vsebnika, kar naredimo s klicem metode `getContentPane()`, nato pa z metodo `add` dodamo novi gumb:

```
okno.getContentPane().add(gumb);
```

Program `Okno2.java` sedaj izgleda takole:

```
import javax.swing.*;

public class Okno2
{
  public static void main(String[] args)
  {
    JFrame okno = new JFrame("Moje okno");
    JButton gumb = new JButton("Klikni me!");
    okno.getContentPane().add(gumb);
    okno.setSize(200, 100);
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.setVisible(true);
  }
}
```

Program Okno2.java

Dodani gumb pri tem zasede celotno površino okna, kot to prikazuje slika 4.3.

Slika 4.3: Okno z gumbom (`Okno2.java`).

Z dodajanjem novih gradnikov na ta način postaja koda vedno bolj nepregledna, zato je bolje, da uvedemo nov razred, ki je izpeljan iz razreda `JFrame`, gradnike postavimo za atribute tega razreda, v konstruktorju pa poskrbimo, da se gradniki dodajo vsebini okna na ustrezno mesto.

V primeru našega zadnjega programa bi definirali nov razred `Okvir`, ki je izpeljan iz razreda `JFrame`. Atributi razreda so poleg gumba tudi tri spremenljivke, ki določajo velikost okna (širino in višino) in njegov naslov. V konstruktorju poskrbimo, da se nastavi naslov okna na ustrezen niz, določi njegova velikost in privzeta operacija ob zaprtju okna ter da se doda gumb k obstoječi vsebini okna:

```
public class Okvir extends JFrame
{
  private final int SIRINA = 200;
  private final int VISINA = 150;
  private String naslov = "Moje okno";
  private JButton gumb = new JButton("Klikni me!");

  public Okvir()
  {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().add(gumb);
  }
}
```

Pri klicu metod (kot je na primer `setTitle()`) nismo rabili reference na objekt `JFrame`, saj metode, gre za podedovane metode tega razreda, kličemo znotraj konstruktorja. Enakovredno bi lahko uporabili tudi določilo `this` (na primer `this.setTitle()`).

V `main` metodi nato ustvarimo objekt razreda `Okvir` in ga prikažemo na zaslonu (razred `Okvir` je izpeljan iz razreda `JFrame`, zato tudi deduje metodo `setVisible`):

```
Okvir okno = new Okvir();
okno.setVisible(true);
```

V celoti je izvorna koda programa `Okno3.java` naslednja:

```
import javax.swing.*;

public class Okno3
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  private final int SIRINA = 200;
  private final int VISINA = 100;
  private String naslov = "Moje okno";
  private JButton gumb = new JButton("Klikni me!");

  public Okvir()
  {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    getContentPane().add(gumb);
  }
}
```

Program Okno3.java

Program, ki je v datoteki `Okno3.java`, se pri izvajanju v ničemer ne razlikuje od našega prejšnjega programa `Okno2.java`. Preizkusite ga sami!

Pa dodajmo v obstoječe okno še nekaj gradnikov. Kot primer preprostega grafičnega vmesnika v Javi bi lahko napisali program, kjer osnovno okno zajema dve oznaki, vnosno polje za vpis besedila ter dva gumba. Prvi gumb zbriše vsebino vnosnega polja, drugi gumb pa v drugo oznako izpiše vsebino vnosnega polja.

Program zgradimo postopoma. Najprej dodamo vse potrebne gradnike (oznaki, gumba in vnosno polje), ki jih zapišemo kot atribute razreda:

```
private JLabel oznaka1 = new JLabel("Vpiši besedilo: ");
private JLabel oznaka2 = new JLabel(" ..... ");
private JTextField tekst =  new JTextField(5);
private JButton gumb1 = new JButton("Zbriši");
private JButton gumb2 = new JButton("Potrdi");
```

Pri kreiranju oznak smo konstruktorju kot argument podali besedilo oznake, konstruktor vnosnega polja pa kot argument prejme privzeto velikost vnosnega polja.

Zaradi enostavnosti dodajmo še en vmesni vsebnik, ki bo združeval vse navedene gradnike v skupino. Zato ustvarimo nov objekt razreda `JPanel`, ki bo igral vlogo tega vmesnega vsebnika:

```
private JPanel ozadje = new JPanel();
```

V konstruktorju razreda `Okvir` najprej poskrbimo za to, da vse oznake, gumbe in vnosno polje združimo (dodamo) v vmesnem vsebniku `ozadje`:

```
ozadje.add(oznaka1);
ozadje.add(tekst);
ozadje.add(oznaka2);
ozadje.add(gumb1);
ozadje.add(gumb2);
```

Pri tem je vrstni red dodajanja elementov pomemben, saj se elementi dodajajo po vrsti od leve proti desni, vrstico za vrstico. Zakaj je tako, bomo videli v naslednjem razdelku.

Na koncu (še vedno znotraj konstruktorja razreda `Okvir`) pa dodamo vmesni vsebnik na delovno površino okna:

```
getContentPane().add(ozadje);
```

Celoten program je v datoteki `Okno4.java`, rezultat pa prikazuje slika 4.4.

Slika 4.4: Okno z oznakama, vnosnim poljem in gumboma (`Okno4.java`).

```
import javax.swing.*;

public class Okno4
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  private final int SIRINA = 200;
  private final int VISINA = 100;
  private String naslov = "Moje okno";

  private JLabel oznaka1 = new JLabel("Vpiši besedilo: ");
  private JLabel oznaka2 = new JLabel(" ..... ");
  private JTextField tekst =  new JTextField(5);
  private JButton gumb1 = new JButton("Zbriši");
```

```
  private JButton gumb2 = new JButton("Potrdi");
  private JPanel ozadje = new JPanel();

  public Okvir()
  {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ozadje.add(oznaka1);
    ozadje.add(tekst);
    ozadje.add(oznaka2);
    ozadje.add(gumb1);
    ozadje.add(gumb2);
    getContentPane().add(ozadje);
  }
}
```

Program Okno4.java

Delovna površina vsebnika, referenco nanjo dobimo s klicem metode `getContentPane()`, je objekt razreda `Container`. Torej bi lahko delovno površino okna (objekt `vsebina`) opisali takole:

```
Container vsebina = getContentPane();
```

Stavek `getContentPane().add(ozadje);` pa bi lahko zapisali z zaporedjem stavkov:

```
Container vsebina = getContentPane();
vsebina.add(ozadje);
```

Ker je razred `Container` opisan v paketu `java.awt`, moramo v tem primeru napovedati tudi uporabo paketa `java.awt` s stavkom:

```
import java.awt.*;
```

Včasih želimo referenco na delovno površino okna shraniti kot atribut razreda. Tako spremenjen program je v datoteki `Okno5.java`.

```
import javax.swing.*;
import java.awt.*;

public class Okno5
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  private final int SIRINA = 200;
  private final int VISINA = 100;
  private String naslov = "Moje okno";
```

```
  private JLabel oznaka1 = new JLabel("Vpiši besedilo: ");
  private JLabel oznaka2 = new JLabel(" ..... ");
  private JTextField tekst =  new JTextField(5);
  private JButton gumb1 = new JButton("Zbriši");
  private JButton gumb2 = new JButton("Potrdi");
  private JPanel ozadje = new JPanel();
  private Container vsebina = null;

  public Okvir()
  {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ozadje.add(oznaka1);
    ozadje.add(tekst);
    ozadje.add(oznaka2);
    ozadje.add(gumb1);
    ozadje.add(gumb2);
    vsebina = getContentPane();
    vsebina.add(ozadje);
  }
}
```

Program Okno5.java

Razporejevalniki

V prejšnjem primeru so se gradniki na delovni površini prikazali po vrsti od leve proti desni, kot smo jih dodajali na površino. Taka razporeditev seveda ni vedno primerna, zato lahko uporabimo ustrezen razporejevalnik (*layout manager*), ki omogoča za naš primer najboljšo in najenostavnejšo razporeditev gradnikov.

Z uporabo razporejevalnikov dosežemo, da gradniki ohranjajo svojo obliko ne glede na velikost posameznih gradnikov. Tako se ohranjajo medsebojna razmerja med gradniki, tudi če spremenimo velikost vsebnika. Videz grafičnega vmesnika torej v celoti določa izbrani način razvrščanja in vrstni red dodajanja gradnikov.

AWT ima pet razredov, ki jih lahko uporabimo za razporejevalnike, poleg tega pa še vmesnik `LayoutManager`, ki omogoča izdelavo lastnega razporejevalnika. Najpogosteje se uporabljajo razporejevalniki `BorderLayout`, `FlowLayout` ter `GridLayout`, katere si bomo pogledali na primerih v nadaljevanju. Razporejevalnik `CardLayout` je uporaben v posebnih primerih (za izdelavo večličnih obrazcev), saj vsebnik razdeli na več strani, od katerih je naenkrat vidna le ena. Razporejevalnik `GridBagLayout` pa je sicer zelo zmogljiv, a nekoliko bolj zapleten za uporabo, saj vsebnik razdeli na mrežo različno velikih celic.

Seveda lahko gradnike postavljamo tudi brez razporejevalnika (*null layout*), če podamo njihov položaj glede na koordinate zaslona. Na način ni preveč priporočljiv, saj nastane velik problem pri spreminjanju velikosti gradnikov (*resize*).

Ker so razporejevalniki opisani v paketu `java.awt`, ob uporabi razporejevalnikov ne smemo pozabiti napovedati uporabe tega paketa.

V nadaljevanju si najprej poglejmo tri primere razvrščanja gumbov v oknu, nato pa se bomo vrnili k našemu primeru (`Okno5.java`) in s pomočjo primernega razporejevalnika bolje razvrstili gradnike v oknu.

Robno razvrščanje (`BorderLayout`)

`BorderLayout` je privzet razporejevalnik za okna. Gradnike lahko dodajamo na štiri robove vsebnika, preostalo površino pa zapolni peti gradnik. Ustvarimo ga s klicem konstruktorja `new BorderLayout()`, vsebniku pa določimo razporejevalnik s klicem metode `setLayout()`:

```
BorderLayout raz = new BorderLayout();
vsebnik.setLayout(raz);
```

Posamezne gradnike dodajamo z metodo `add(gradnik, poz)`, kjer je `poz` položaj gradnika v vsebniku: zgoraj, spodaj, levo, desno ali v sredini. Za položaj lahko uporabimo tudi konstante razreda `BorderLayout`, ki so poimenovane po straneh neba (`NOTRH`, `SOUTH`, `WEST`, `EAST` in `CENTER`, po vrsti).

Velikost gradnikov se spremeni tako, da le-ti zavzamejo ves razpoložljiv prostor.

Primer (program `Robno.java`) prikazuje slika 4.5.

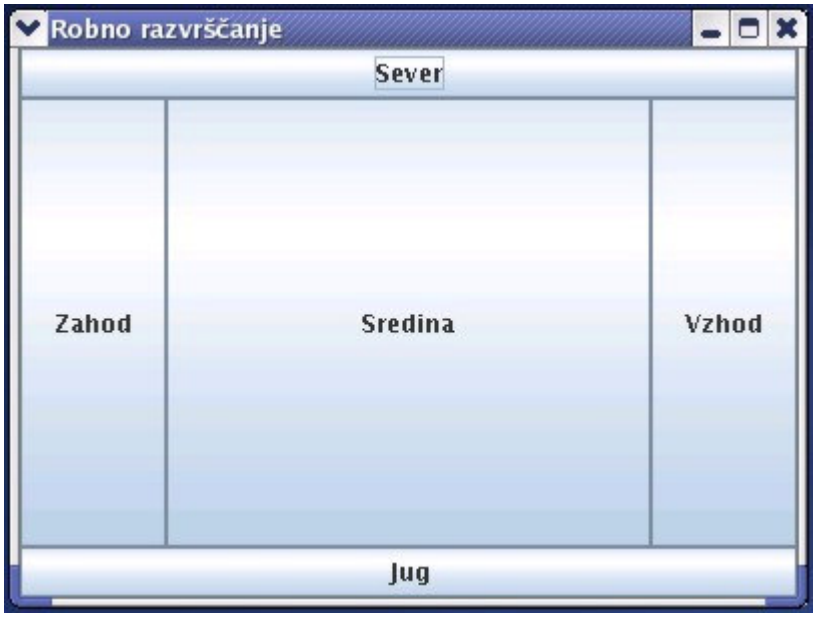
```
import javax.swing.*;
import java.awt.*;

public class Robno
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  public Okvir()
  {
    Container vsebina;
    setTitle("Robno razvrščanje");
    setSize(400, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    vsebina = getContentPane();
    vsebina.setLayout(new BorderLayout());
    vsebina.add(new JButton("Sever"), BorderLayout.NORTH);
    vsebina.add(new JButton("Jug"), BorderLayout.SOUTH);
    vsebina.add(new JButton("Vzhod"), BorderLayout.EAST);
    vsebina.add(new JButton("Zahod"), BorderLayout.WEST);
```

```
    vsebina.add(new JButton("Sredina"), BorderLayout.CENTER);
  }
}
```

Program `Robno.java`



Slika 4.5: Robno razvrščanje v oknu (`Robno.java`).

Tekoče razvrščanje (`FlowLayout`)

`FlowLayout` je privzet razporejevalnik za vsebnike razreda `Panel`. Gradnike dodajamo po vrsti od leve proti desni, vrstico za vrstico. Pri tem gradniki ohranijo svojo velikost (v našem primeru se velikost gumbov prilagodi velikosti napisa na gumbu). Razporejevalnik `FlowLayout` ustvarimo s klicem ustreznega konstruktorja ter ga določimo za razporejevalnik vsebnika:

```
FlowLayout raz = new FlowLayout();
vsebnik.setLayout(raz);
```

Oba stavka lahko tudi združimo:

```
vsebnik.setLayout(new FlowLayout());
```

Program `Tekoce.java` izriše okno z gumbi, kot prikazuje slika 4.6.

Slika 4.6: Tekoče razvrščanje v oknu (`Tekoce.java`).

```
import javax.swing.*;
import java.awt.*;

public class Tekoce
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  public Okvir()
  {
    Container vsebina;
    setTitle("Tekoče razvrščanje");
    setSize(400, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    vsebina = getContentPane();
    vsebina.setLayout(new FlowLayout());
    vsebina.add(new JButton("Prvi gumb"));
    vsebina.add(new JButton("Drugi gumb"));
    vsebina.add(new JButton("Tretji gumb"));
    vsebina.add(new JButton("Četrti gumb"));
    vsebina.add(new JButton("Peti gumb"));
  }
}
```

Program Tekoce.java

Razvrščanje v mrežo (`GridLayout`)

`GridLayout` razdeli vsebnik v mrežo enakih celic, v katere po vrsti razvrsti gradnike. Tako so vsi gradniki enako veliki, ne glede na njihovo vsebino. Ustvarimo ga s klicem konstruktorja, kateremu podamo velikost mreže (število vrstic in število stolpcev):

```
vsebnik.setLayout(new GridLayout(2,3));
```

Primer razvrščanja v mrežo prikazuje slika 4.7 (program `Mreza.java`).

Slika 4.7: Razvrščanje v mrežo enako velikih celic (`Mreza.java`).

```
import javax.swing.*;
import java.awt.*;

public class Mreza
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  public Okvir()
  {
    Container vsebina;
    setTitle("Razvrščanje v mrežo");
    setSize(400, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    vsebina = getContentPane();
    vsebina.setLayout(new GridLayout(2,3));
    vsebina.add(new JButton("Prvi gumb"));
    vsebina.add(new JButton("Drugi gumb"));
    vsebina.add(new JButton("Tretji gumb"));
    vsebina.add(new JButton("Četrti gumb"));
    vsebina.add(new JButton("Peti gumb"));
  }
}
```

Program Mreza.java

Preizkusite opisane primere razvrščanja ter opazujte njihovo obnašanje pri spreminjanju velikosti okna (z vlečenjem miške okno povečamo ali pomanjšamo). Vsi gradniki okna se primerno razporedijo po razpoložljivem prostoru, skladno s pravili, ki jih določa razporejevalnik.

Naš program

Če se ponovno vrnemo k našemu programu `Okno5.java`, ugotovimo, da smo v njem uporabljali razporejevalnik, čeprav ga nismo eksplicitno ustvarili. Osnovne gradnike (oznaki, vnosno polje in gumba) smo namreč združili v vsebniku razreda `JPanel`, za katerega velja privzet razporejevalnik `FlowLayout`. Zato se je vseh pet osnovnih gradnikov razporedilo po vrsti od leve proti desni, tako kot smo jih dodajali v vsebnik. Sam vsebnik smo nato postavili na delovno površino okna, za katerega je privzet razporejevalnik `BorderLayout`. Ker pri dodajanju nismo navedli, na katero stran želimo postaviti vsebnik, se je upoštevala privzeta vrednost, to je dodajanje v sredino. Sedaj tudi lažje razumemo, zakaj smo dobili takšen izgled in obnašanje okna (slika 4.4).

Še večjo prilagodljivost pa lahko dosežemo, če posamezne gradnike najprej združimo v vmesne vsebnike ter na delovno površino postavimo te vsebnike. Najbolj smiselno je združevati gradnike, ki na nek način (na primer pomensko) pašejo skupaj. V našem primeru bi lahko združili prvo oznako in vnosno polje v enem vsebniku ter oba gumba v drugem vsebniku. Tretji vsebnik bi tako vseboval le drugo oznako. Zato moramo ustvariti tri vsebnike, ki so lahko definirani kot atributi razreda:

```
private JPanel vnos = new JPanel();
private JPanel izpis = new JPanel();
private JPanel gumbi = new JPanel();
```

Vsebnik vnos združuje oznako in vnosno polje, oboje postavljeno po vrsti eno za drugim:

```
vnos.setLayout(new FlowLayout());
vnos.add(oznaka1);
vnos.add(tekst);
```

Prva vrstica (določitev razporejevalnika `FlowLayout`) ni potrebna, saj je `FlowLayout` tako ali tako privzet razporejevalnik za vsebnike razreda `JPanel`.

Drugemu vsebniku, `izpis`, dodamo le drugo oznako (tudi tu uporabimo kar privzet razporejevalnik `FlowLayout`):

```
izpis.add(oznaka2);
```

Čeprav v vsebnik `izpis` postavimo le eno oznako, smo nov vsebnik uporabili zato, da oznaka `oznaka2` ohrani svojo privzeto velikost in da je postavljena na sredino (sredinska poravnava je privzeta poravnava pri tekočem razvrščanju).

Oba gumba združimo v vsebniku `gumbi`, za katerega tudi uporabimo privzet razporejevalnik `FlowLayout`:

```
gumbi.add(gumb1);
gumbi.add(gumb2);
```

Vse tri vsebnike potem postavimo na ustrezna mesta na delovno površino. Tu smo izbrali razporejevalnik `GridLayout`, ki ima tri vrstice, en stolpec ter 20 pik vodoravnega razmaka med stolpci in 5 pik navpičnega razmaka med vrsticami:

```
vsebina.setLayout(new GridLayout(3,1,20,5));
vsebina.add(vnos);
vsebina.add(izpis);
vsebina.add(gumbi);
```

Pri tem smo tudi malo povečali okno, ki je sedaj višine 150 pik, da imamo dovolj prostora za vse gradnike. Celoten program je v datoteki `Okno6.java`, njegov rezultat pa je prikazan na sliki 4.8.

Slika 4.8: Okno z uporabo razporejevalnikov (`Okno6.java`).

```
import javax.swing.*;
import java.awt.*;
```

```
public class Okno6
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  private final int SIRINA = 200;
  private final int VISINA = 150;
  private String naslov = "Moje okno";
  private JLabel oznaka1 = new JLabel("Vpiši besedilo: ");
  private JLabel oznaka2 = new JLabel(" ..... ");
  private JTextField tekst =  new JTextField(5);
  private JButton gumb1 = new JButton("Zbriši");
  private JButton gumb2 = new JButton("Potrdi");
  private JPanel vnos = new JPanel();
  private JPanel izpis = new JPanel();
  private JPanel gumbi = new JPanel();
  private Container vsebina = null;

  public Okvir()
  {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    vnos.add(oznaka1);
    vnos.add(tekst);
    izpis.add(oznaka2);
    gumbi.add(gumb1);
    gumbi.add(gumb2);
    vsebina = getContentPane();
    vsebina.setLayout(new GridLayout(3,1,20,5));
    vsebina.add(vnos);
    vsebina.add(izpis);
    vsebina.add(gumbi);
  }
}
```

Program Okno6.java

Program sicer odpre okno in v njem izriše vse gradnike, med njimi tudi dva gumba, a nima nobene funkcionalnosti. Če kliknemo na gumb, se ne zgodi nič. V naslednjem razdelku bomo pogledali, kako programu dodamo tudi funkcionalnost.

Dogodki in poslušalci

Programi z grafičnim vmesnikom so dogodkovno vodeni, kar pomeni, da se odzivajo na akcije uporabnika. Preko mehanizma dogodkov (*events*) lahko izvajamo nadzor množice dejavnih elementov, ki se lahko kadarkoli odzovejo na dejanje uporabnika.

Izvajanje programa z grafičnim vmesnikom je odvisno od povzročenih dogodkov. Dogodek je sporočilo v obliki predmeta `AWTEvent`, ki ga javanski sistem posreduje metodi `processEvent()` gradnika, kadar se v izvajanju programa zgodi kaj takega,

kar bi lahko zanimalo ta gradnik (kot je na primer sprememba stanja, interakcija z uporabnikom in podobno).

Seveda nas ponavadi izmed množice sproženih dogodkov zanima le majhen del. Zato ima AWT vzpostavljen dogodkovni model, s katerim gradnik o dogodku obvesti poslušalce (*listeners*). Tako lahko vsakemu gradniku prijavimo ali odjavimo poslušalca (metodi `addXListener()` in `removeXListener()`), ta pa mora nositi izvedbo predpisanega vmesnika `XListener` (`X` označuje poljubnega poslušalca).

Paket `java.awt.event` ponuja več različnih vmesnikov, vsak je namenjen določenemu tipu dogodkov. Med drugimi naj omenimo vmesnik `ActionListener`, ki opazuje akcijske dogodke, ter vmesnik `WindowListener` za dogodke v zvezi z okni. Zanimivi so tudi `MouseListener` in `MouseMotionListener`, ki opazujeta mišje dogodke, ter `KeyListener` za dogodke v zvezi s tipkovnico. Ostale vmesnike in opis metod posameznih vmesnikov si lahko ogledate v JDK dokumentaciji.

V splošnem se rokovanja z dogodki lotimo tako, da za posamezno vrsto dogodkov napišemo ustreznega poslušalca (izpolnimo zahtevan vmesnik) in ga prijavimo gradniku. V primeru dogodka izvajalno okolje pokliče ustrezno metodo na vmesniku poslušalca, kar izkoristimo za pisanje odzivov. Pri tem so o dogodku obveščeni vsi prijavljeni poslušalci, istega poslušalca pa lahko prijavimo na različne vire dogodkov.

Poslušalec kot notranji razred

Pa si oglejmo obravnavo dogodkov na našem primeru. Programu `Okno6.java` dodajmo tudi odziv na dogodke, ki se zgodijo ob aktivaciji (na primer ob kliku) katerega od gumbov. Zato moramo najprej definirati ustreznega poslušalca, ki se odziva na akcije gumbov. Tako definiramo nov razred `GumbPoslusalec`, ki implementira vmesnik `ActionListener`, ter v njem metodo `actionPerformed()`, ki je tudi edina metoda, ki jo zahteva ta vmesnik:

```
private class GumbPoslusalec implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    ...
  }
}
```

Razred definiramo kot notranji privatni razred, saj ni potrebe, da bi ga uporabljali še kjerkoli drugje. Odločitev za notranji razred nam bo tudi poenostavila napisano kodo, kar bomo spoznali v nadaljevanju na koncu tega razdelka.

Pri tem ne smemo pozabiti na napoved uporabe paketa `java.awt.event` na začetku programske kode, saj so v njem definirani z dogodki povezani vmesniki in razredi.

Potem obema gumboma dodamo nov izvod poslušalca (poslušalca prijavimo gumbu) na naslednji način:

```
gumb1.addActionListener(new GumbPoslusalec());
gumb2.addActionListener(new GumbPoslusalec());
```

Oba stavka postavimo v konstruktor razreda `Okvir`.

Sedaj moramo le še definirati metodo `actionPerformed()`, ki se pokliče ob dogodku. V njej torej določimo odziv našega programa na ta dogodek. Ker smo istega poslušalca prijavili obema gumboma, moramo najprej ugotoviti, kateri gumb je bil izvor dogodka. Sam dogodek `e` dobimo kot argument metode, s pomočjo metode `getSource()` pa dobimo referenco na objekt, na katerem se je dogodek dejansko zgodil. Potem lahko enostavno preverimo, kateri od obeh gumbov se ujema z izvorom dogodka, in ustrezno ukrepamo:

```
public void actionPerformed(ActionEvent e)
{
  Object izvor = e.getSource();
  if (izvor == gumb1)
    tekst.setText("");
  if (izvor == gumb2)
    oznaka2.setText(tekst.getText());
}
```

Če je izvor dogodka `gumb1`, potem s klicem metode `setText("")` nastavimo vsebino vnosnega polja na prazen niz (kar pomeni, da jo zbrišemo). Če pa je izvor dogodka `gumb2`, vsebino vnosnega polja, ki jo dobimo s klicem metode `getText()`, zapišemo v oznako `oznaka2` (slednje dosežemo z uporabo metode `setText()`).

Celoten program je v datoteki `Okno7.java`. Preverite njegovo delovanje.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Okno7
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  private final int SIRINA = 200;
  private final int VISINA = 150;
  private String naslov = "Moje okno";
  private JLabel oznaka1 = new JLabel("Vpiši besedilo: ");
```

```
  private JLabel oznaka2 = new JLabel(" ..... ");
  private JTextField tekst =  new JTextField(5);
  private JButton gumb1 = new JButton("Zbriši");
  private JButton gumb2 = new JButton("Potrdi");
  private JPanel vnos = new JPanel();
  private JPanel izpis = new JPanel();
  private JPanel gumbi = new JPanel();
  private Container vsebina = null;

  public Okvir()
  {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    vnos.add(oznaka1);
    vnos.add(tekst);
    izpis.add(oznaka2);
    gumb1.addActionListener(new GumbPoslusalec());
    gumb2.addActionListener(new GumbPoslusalec());
    gumbi.add(gumb1);
    gumbi.add(gumb2);
    vsebina = getContentPane();
    vsebina.setLayout(new GridLayout(3,1,20,5));
    vsebina.add(vnos);
    vsebina.add(izpis);
    vsebina.add(gumbi);
  }

  private class GumbPoslusalec implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      Object izvor = e.getSource();
      if (izvor == gumb1)
        tekst.setText("");
      if (izvor == gumb2)
        oznaka2.setText(tekst.getText());
    }
  }
}
```

Program Okno7.java

Poslušalec ni njuno nov razred

Obstaja pa tudi krajši in zato nekoliko enostavnejši način obravnave dogodkov. Tako ni potrebno definirati novega razreda za poslušalca dogodkov, ampak lahko vmesnik `ActionListener` implementira kar naš razred `Okvir`:

```
public class Okvir extends JFrame implements ActionListener
```

Ker je razred `Okvir` tudi poslušalec dogodkov, moramo v njem definirati metodo `actionPerformed()`, ki jo zahteva ta vmesnik. Metoda je enaka kot v prejšnjem primeru.

Tako definiranega poslušalca dodamo gumbu s stavkom:

```
gumb1.addActionListener(this);
```

kjer se določilo `this` nanaša na ta isti razred, torej razred `Okvir`, ki je hkrati tudi poslušalec.

Koda tega programa je v datoteki `Okno8.java`. V delovanju obeh programov ni razlik.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Okno8
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame implements ActionListener
{
  private final int SIRINA = 200;
  private final int VISINA = 150;
  private String naslov = "Moje okno";
  private JLabel oznaka1 = new JLabel("Vpiši besedilo: ");
  private JLabel oznaka2 = new JLabel(" ..... ");
  private JTextField tekst =  new JTextField(5);
  private JButton gumb1 = new JButton("Zbriši");
  private JButton gumb2 = new JButton("Potrdi");
  private JPanel vnos = new JPanel();
  private JPanel izpis = new JPanel();
  private JPanel gumbi = new JPanel();
  private Container vsebina = null;

  public Okvir()
  {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    vnos.add(oznaka1);
    vnos.add(tekst);
    izpis.add(oznaka2);
    gumb1.addActionListener(this);
    gumb2.addActionListener(this);
    gumbi.add(gumb1);
    gumbi.add(gumb2);
    vsebina = getContentPane();
    vsebina.setLayout(new GridLayout(3,1,20,5));
    vsebina.add(vnos);
    vsebina.add(izpis);
    vsebina.add(gumbi);
  }

  public void actionPerformed(ActionEvent e)
  {
    Object izvor = e.getSource();
    if (izvor == gumb1)
      tekst.setText("");
    if (izvor == gumb2)
      oznaka2.setText(tekst.getText());
  }
}
```

Program Okno8.java

Poslušalec kot anonimni razred

Kadar želimo napisati preprost odziv na en sam gradnik, lahko to naredimo tudi v obliki anonimnega notranjega razreda, ki ga definiramo kar v argumentu metode za prijavo poslušalca. Če bi v našem primeru želeli narediti odziv le na gumb `gumb2`, bi to lahko zapisali takole:

```
gumb2.addActionListener(new ActionListener() {
       public void actionPerformed(ActionEvent e) {
         oznaka2.setText(tekst.getText());
       }
});
```

Kadar pa želimo dogodke obravnavati bolj splošno in metode uporabiti za različne gradnike, vedno napišemo samostojni razred. Ta je sicer lahko tudi notranji razred.

`XAdapter` namesto `XListener`

Poglejmo si še en primer. Program `Okno7.java` dopolnimo še s poslušalcem za dogodke, povezane z okni, ter ga uporabimo za odziv na dogodek zapiranja okna.

Naj se nov razred, ki sprejema okenske dogodke, imenuje `OknoPoslusalec`. Tako namesto metode:

```
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

dodamo oknu nov izvod poslušalca, ki smo ga napisali, da bdi nad okenskimi dogodki:

```
addWindowListener(new OknoPoslusalec());
```

Vmesnik `WindowListener`, ki naj bi ga izdelal naš razred `OknoPoslušalec`, vsebuje naslednje metode: `windowActivated()`, `windowClosed()`, `windowClosing()`, `windowDeactivated()`, `windowDeiconified()`, `windowIconified()` in `windowOpened()`. Ta podatek smo dobili z vpogledom v dokumentacijo JDK.

Od vseh omenjenih metod vmesnika pa nas zanima le metoda `windowClosed()`, ki se pokliče ob zapiranju okna.

Če bi razred `OknoPoslusalec` implementiral vmesnik `WindowListener`, bi moral implementirati tudi vseh sedem metod tega vmesnika (vsak razred, ki implementira vmesnik, mora implementirati tudi vse metode tega vmesnika). Čeprav nas zanima ena sama metoda, to je metoda `windowClosed()`, bi morali definirati še preostalih šest metod, ki bi pač ostale prazne.

Na srečo pa obstaja tudi druga, krajša pot. Namesto da razred implementira vmesnik `WindowListener`, naj razred raje razširja abstraktni razred `WindowAdapter`. To omogoča, da nam ni potrebno definirati vseh metod vmesnika (ker so le-te že definirane v abstraktnem razredu kot prazne metode), temveč lahko preobložimo le tisto metodo, ki nas zanima (torej samo metodo `windowClosed()`):

```
public class OknoPoslusalec extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}
```

S pomožnimi razredi `XAdapter`, ki nosijo prazne izvedbe vseh metod vmesnika poslušalca `XListener`, si pomagamo v primeru, ko nas zanima le nekaj metod iz vmesnika poslušalca. Poslušalca napišemo tako, da razširimo ustrezen adapter in s svojo izvedbo prekrijemo le tiste metode, ki nas zanimajo.

Opisana koda je v datoteki `Okno9.java`. Preverite, da se njeno izvajanje v ničemer ne razlikuje od programa `Okno7.java`.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Okno9
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  private final int SIRINA = 200;
  private final int VISINA = 150;
  private String naslov = "Moje okno";
  private JLabel oznaka1 = new JLabel("Vpiši besedilo: ");
  private JLabel oznaka2 = new JLabel(" ..... ");
  private JTextField tekst =  new JTextField(5);
  private JButton gumb1 = new JButton("Zbriši");
  private JButton gumb2 = new JButton("Potrdi");
  private JPanel vnos = new JPanel();
  private JPanel izpis = new JPanel();
  private JPanel gumbi = new JPanel();
  private Container vsebina = null;

  public Okvir()
  {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    addWindowListener(new OknoPoslusalec());
    vnos.add(oznaka1);
    vnos.add(tekst);
```

```
    izpis.add(oznaka2);
    gumb1.addActionListener(new GumbPoslusalec());
    gumb2.addActionListener(new GumbPoslusalec());
    gumbi.add(gumb1);
    gumbi.add(gumb2);
    vsebina = getContentPane();
    vsebina.setLayout(new GridLayout(3,1,20,5));
    vsebina.add(vnos);
    vsebina.add(izpis);
    vsebina.add(gumbi);
  }

  private class GumbPoslusalec implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      Object izvor = e.getSource();
      if (izvor == gumb1)
        tekst.setText("");
      if (izvor == gumb2)
        oznaka2.setText(tekst.getText());
    }
  }
}

class OknoPoslusalec extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}
```

Program Okno9.java

Seveda bi tudi v tem primeru lahko uporabili kar anonimni notranji razred, saj gre za enostaven odziv okna na dogodek. Razred lahko definiramo kar v argumentu metode za prijavo poslušalca:

```
addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
         System.exit(0);
       }
});
```

Prenos reference na objekt

Verjetno ste opazili, da smo v zadnjem primeru (`Okno9.java`) uporabili kot poslušalce objekte dveh različnih razredov: `GumbPoslusalec` in `OknoPoslusalec`. Medtem ko je slednji navaden javni razred, pa je razred `GumbPoslusalec` definiran kot notranji razred razreda `Okvir`. Zakaj? No, za tako rešitev smo imeli zelo tehten razlog. Metoda `actionPerformed` tega razreda se namreč sklicuje na različne objekte (`gumb1`, `gumb2`, `oznaka2`, `tekst`), ki so deklarirani kot privatne objektne spremenljivke razreda `Okvir`. Da so te spremenljivke vidne (v dosegu) tudi v metodi `actionPerformed`, mora biti ta metoda definirana znotraj razreda `Okvir` (kot metoda razreda `Okvir` ali kot metoda razreda `GumbPoslusalec`, ki je notranji razred razreda

`Okvir`). Obe rešitvi smo prikazali v prejšnjih primerih (`Okno8.java` in `Okno7.java`, po vrsti).

Ali lahko sestavimo tudi rešitev, kjer je metoda `actionPerformed` definirana v razredu `GumbPoslusalec`, ki pa ni definiran kot notranji razred? Seveda lahko.

Razred `GumbPoslusalec` lahko najavimo tudi kot javni razred, njegova edina metoda pa ostane nespremenjena:

```
public class GumbPoslusalec implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    Object izvor = e.getSource();
    if (izvor == gumb1)
      tekst.setText("");
    if (izvor == gumb2)
      oznaka2.setText(tekst.getText());
  }
}
```

Vendar pa tu nastane manjši problem, ki ga moramo rešiti. Ker se v metodi `actionPerformed` sklicujemo na spremenljivke `gumb1`, `gumb2`, `tekst` in `oznaka2`, morajo biti te spremenljivke tudi deklarirane v tem razredu, saj jih drugače ne moremo uporabljati. Deklariramo jih lahko kot privatne atribute razreda:

```
private JButton gumb1, gumb2;
private JTextField tekst;
private JLabel oznaka2;
```

Seveda pa želimo, da se te spremenljivke sklicujejo na tiste (v naši kodi istoimenske) objekte, ki smo jih že ustvarili znotraj razreda `Okvir` (kjer so deklarirane kot privatni atributi razreda). Zato ob kreiranju novega objekta `GumbPoslušalec` (ki ga ustvarimo v razredu `Okvir`), prenesemo tudi reference na te štiri spremenljivke v objekt razreda `GumbPoslušalec`. Za to najenostavneje poskrbimo kar v konstruktorju razreda, saj se ta zagotovo pokliče ob vsakem kreiranju novega izvoda tega razreda. Konstruktorju ob klicu kot argumente podamo reference na vse štiri objekte, ki se na ta način prenesejo v novo ustvarjen objekt:

```
public GumbPoslusalec(JButton g1, JButton g2,
                      JTextField t, JLabel o)
{
  this.gumb1 = g1;
  this.gumb2 = g2;
  this.tekst = t;
  this.oznaka2 = o;
}
```

Seveda moramo sedaj pri klicu samega konstruktorja podati tudi vse štiri argumente:

```
new GumbPoslusalec(gumb1, gumb2, tekst, oznaka2);
```

Celoten program je v datoteki `Okno10.java`. Tudi delovanje tega programa se v ničemer ne razlikuje od programa `Okno7.java`. Preverite sami!

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Okno10
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  private final int SIRINA = 200;
  private final int VISINA = 150;
  private String naslov = "Moje okno";

  private JLabel oznaka1 = new JLabel("Vpiši besedilo: ");
  private JLabel oznaka2 = new JLabel(" ..... ");
  private JTextField tekst =  new JTextField(5);
  private JButton gumb1 = new JButton("Zbriši");
  private JButton gumb2 = new JButton("Potrdi");
  private JPanel vnos = new JPanel();
  private JPanel izpis = new JPanel();
  private JPanel gumbi = new JPanel();
  private Container vsebina = null;

  public Okvir()
  {
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    addWindowListener(new OknoPoslusalec());

    vnos.add(oznaka1);
    vnos.add(tekst);

    izpis.add(oznaka2);

    gumb1.addActionListener(new GumbPoslusalec(gumb1, gumb2, tekst, oznaka2));
    gumb2.addActionListener(new GumbPoslusalec(gumb1, gumb2, tekst, oznaka2));
    gumbi.add(gumb1);
    gumbi.add(gumb2);

    vsebina = getContentPane();
    vsebina.setLayout(new GridLayout(3,1,20,5));
    vsebina.add(vnos);
    vsebina.add(izpis);
    vsebina.add(gumbi);
  }
}

class GumbPoslusalec implements ActionListener
{
  private JButton gumb1, gumb2;
  private JTextField tekst;
  private JLabel oznaka;
```

```
  public GumbPoslusalec(JButton g1, JButton g2, JTextField t, JLabel o)
  {
    this.gumb1 = g1;
    this.gumb2 = g2;
    this.tekst = t;
    this.oznaka = o;
  }

  public void actionPerformed(ActionEvent e)
  {
    Object izvor = e.getSource();
    if (izvor == gumb1)
      tekst.setText("");
    if (izvor == gumb2)
      oznaka.setText(tekst.getText());
  }
}

class OknoPoslusalec extends WindowAdapter
{
  public void windowClosing(WindowEvent e)
  {
    System.exit(0);
  }
}
```

Program Okno10.java

Lastnosti sistema

Za predstavitev dejanske implementacije AWT uporabljamo razred `Toolkit`. Preko objekta tega razreda lahko tako pridobimo informacije o dejanskem sistemu, ki je v uporabi. Metoda `getDefaultToolkit()` vrne referenco na objekt razreda `Toolkit`. Metoda `getScreenSize()` vrne velikost zaslona v pikah v obliki objekta razreda `Dimension`. Tako bi na primer dejansko velikost zaslona ugotovili z naslednjimi stavki:

```
Toolkit tk = Toolkit.getDefaultToolkit();
Dimension zaslon = tk.getScreenSize();
System.out.printf("Velikost zaslona je %d x %d\n",
                  zaslon.width, zaslon.height);
```

Potem lahko velikost okna v programu prilagodimo dejanski velikosti zaslona:

```
JFrame okno = new JFrame("Četrtinsko okno");
okno.setSize(zaslon.width/2, zaslon.height/2);
```

Risanje

Za risanje in slikanje lahko uporabimo tudi javanske metode. Programi lahko rišejo neposredno na osnovni vsebnik (torej na objekte razreda `JApplet` ali `JFrame` ter

njihove izpeljanke) ali pa na vmesni vsebnik `JPanel`. Ponavadi pri risanju uporabljamo objekt razreda `JPanel` kot risalno površino.

Vsak javanski gradnik ima svoj grafični kontekst (*graphic context*), ki ga predstavlja objekt razreda `java.awt.Graphics`. Uporabljamo ga za risanje na ta gradnik, saj vsebuje različne metode za risanje. Objekta `Graphics` ne moremo ustvariti neposredno, temveč ga dobimo od gradnika (kot je na primer `JPanel`) s pomočjo metode `getGraphics()`, ki vrne referenco na grafični kontekst tega gradnika.

Grafični gradniki (tako v AWT kot v Swing) se izrišejo avtomatično, kadar se za to izkaže potreba (kadar se spremeni vsebina gradnika, njegova lega, velikost ali vidnost). Za njihov izris poskrbi metoda `paint()` razreda `java.awt.Component`, ki jo samodejno pokliče izvajalni sistem. Ob tem se metodi avtomatično posreduje referenca na objekt razreda `Graphics` za risanje na ta gradnik.

Ponoven izris gradnika (oziroma obnovitev vsebine gradnika) pa lahko zahtevamo tudi programsko s klicem metode `repaint()`, ki potem pokliče metodo `paint()`. V programu ponavadi ne kličemo metode `paint()` neposredno, temveč vedno preko metode `repaint()`.

Pri Swing gradnikih metoda `paint()` pokliče tri ločene metode v naslednjem vrstnem redu: `paintComponent()` za izris gradnika, `paintBorder()` za izris obrobe in `paintChildren()` za izris njegovih otrok (podgradnikov). Kadar se torej ponovno izriše vsebnik, se ponovno izrišejo tudi vsi gradniki, ki jih vsebuje.

Kodo za risanje postavimo v programu znotraj preobložene metode za izris gradnika. Za izris uporabljajo metodo `paint()` vsi gradniki, ki neposredno izhajajo iz razreda `java.awt.Component` (torej vsi AWT gradniki ter `JApplet` in `JFrame`). Razred `javax.swing.JComponent` in njegovi nasledniki pa uporabljajo za svoj izris metodo `paintComponent()`. Zato pri AWT gradnikih preobložimo metodo `paint()`, pri Swing gradnikih pa metodo `paintComponent()`.

Razredi, izpeljani iz `java.awt.Container`, ki preobložijo metodo `paint()`, morajo v njej na začetku poklicati `super.paint()`, da zagotovijo pravilen izris otrok. Podobno velja za razrede, izpeljane iz razreda `JComponent` (vključno z razredom `JPanel`); ti morajo tipično znotraj svoje preobložene metode `paintComponent()` najprej poklicati metodo `super.paintComponent()`.

Za komponente v Swingu je privzeto dvojno pomnjenje (*double buffering*), kar omogoča bolj gladko izrisovanje. Izris se najprej naredi v predpomnilnik (*offscreen buffer*) in šele ko je končan, se v celoti prikaže na zaslonu.

Risanje z grafičnimi primitivi

Ključni element pri programsko podprtem izrisovanju grafičnih primitivov (kot so črte, liki, besedilo) in pri upodabljanju bitnih slik na grafični površini je objekt razreda `Graphics`, ki predstavlja grafični kontekst gradnika. Ustrezen grafični kontekst je avtomatično podan kot argument metode `paint()` oziroma metode `paintComponent()`.

Za risalno površino programskega risanja ponavadi izberemo gradnik `JPanel`. Tako napišemo svoj razred, ki razširja razred `JPanel`, in preobložimo metodo `paintComponent()`, v kateri pokličemo ustrezne metode za izris želenih grafičnih primitivov:

```
public class Risalnica extends JPanel
{
  public void paintComponent(Graphics g)
  {
    // tukaj kličemo metode za izris
  }
}
```

Grafični kontekst uporablja enostaven koordinatni sistem, kjer je vsaka pika predstavljena z dvema koordinatama: *x* in *y*. Izhodišče koordinatnega sistema je v levem zgornjem kotu gradnika. Koordinate *x* naraščajo od leve proti desni, koordinate *y* pa od zgoraj navzdol. Koordinatni sistem lahko tudi prestavimo v katerokoli drugo točko z metodo `translate(dx, dy)`, kjer vrednosti *dx* in *dy* označujeta spremembo koordinat izhodišča.

Razred `Graphics` nudi več metod za risanje grafičnih primitivov. Tako lahko črto narišemo z uporabo metode `drawLine()`, kateri kot argumente podamo koordinati (*x* in *y*) začetne točke in koordinati končne točke.

Metoda `drawRect()` izriše pravokotnik, argumenti metode pa so *x* in *y* koordinati zgornjega levega oglišča, dolžina pravokotnika ter višina pravokotnika.

Elipso izrišemo z metodo `drawOval()`, kateri kot argumente podamo *x* in *y* koordinati zgornjega levega oglišča, dolžina pravokotnika ter višina pravokotnika, elipsa pa se izriše znotraj tako določenega pravokotnika. Za izris kroga določimo isto velikost dolžine in širine (izris elipse v kvadratu).

Vse metode `drawX()` imajo, kjer je to smiselno, tudi sorodno metodo `fillX()`, ki izriše polnjen lik. Tako na primer `fillRect()` izriše polnjen pravokotnik, `fillOval()` pa polnjeno elipso.

Ostale metode za risanje si oglejte v JDK dokumentaciji.

Za primer si poglejmo še program, ki izriše graf funkcije sinus (sin *x*). Program naj se imenuje `Sinus.java` in ga najdete med primeri tega poglavja. Velikost grafa (velikost enote) se prilagaja trenutni velikosti okna, tako da graf vedno zavzame celo površino okna.

Program mora prikazati enostavno okno, v katerega postavimo risalno površino, na njej pa se izriše graf. Za risalno površino lahko uporabimo kar razred `Sinus`, če je le-ta izpeljan iz razreda `JPanel`. V `main()` metodi razreda `Sinus` najprej ustvarimo novo okno, ki je objekt razreda `JFrame`:

```
JFrame okno = new JFrame("Graf sin(x)");
```

Pri tem smo uporabili konstruktor, ki nastavi tudi naslov okna. Potem ustvarimo risalno površino (objekt razreda `Sinus`) in jo postavimo na delovno površino okna:

```
Sinus graf = new Sinus();
okno.getContentPane().add(graf);
```

Velikost okna prilagodimo velikosti risalne površine (metoda `getSize()` vrne velikost gradnika v obliki objekta `Dimension`, katerega atribut `width` določa širino, `height` pa višino gradnika):

```
okno.setSize(graf.getSize().width, graf.getSize().height);
```

Na koncu oknu določimo privzeto operacijo ob zapiranju okna ter ga prikažemo na zaslonu:

```
okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
okno.setVisible(true);
```

Razred `Sinus` ima tudi nekaj privatnih atributov, ki določajo privzeto širino in višino gradnika (velikost gradnika se nastavi ob klicu konstruktorja), ter rob, ki določa razpoložljivo risalno površino. Sedaj moramo le še preobložiti metodo `paintComponent()`, v kateri bomo poskrbeli, da se graf dejansko izriše. Zaenkrat smo sestavili naslednjo kodo:

```
public class Sinus extends JPanel
{
  private final int SIRINA = 800;
  private final int VISINA = 400;
  private final int ROB = 10;

  public static void main(String[] args)
  {
    JFrame okno = new JFrame("Graf sin(x)");
    Sinus graf = new Sinus();
    okno.getContentPane().add(graf);
    okno.setSize(graf.getSize().width, graf.getSize().height);
```

```
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.setVisible(true);
  }

  public Sinus()
  {
    setSize(SIRINA, VISINA);
  }

  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    // poskrbimo za izris grafa
  }
}
```

V metodi `paintComponent()` najprej pokličemo istoimensko metodo razreda `JPanel`, torej `super.paintComponent()`, ki med drugim poskrbi za pravilen izris ozadja (tudi izbris obstoječe vsebine) in naredi gradnik neprozoren. Nato na prazno risalno površino izrišemo koordinatni osi (klic metode `narisiOsi(g)`) in pokličemo še metodo za izris vrednosti funkcije `narisiSinus(g)`. Pred klicem zadnjih dveh metod smo prestavili izhodišče koordinatnega sistema na sredino risalne površine, kar nam poenostavi risanje grafa. V celoti metoda `paintComponent()` izgleda takole:

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Dimension velikost = this.getSize();
  g.setClip(ROB,ROB,velikost.width-2*ROB,velikost.height-2*ROB);
  g.translate(velikost.width/2, velikost.height/2);
  narisiOsi(g);
  narisiSinus(g);
}
```

Z metodo `setClip()` smo določili območje gradnika, kjer je risanje dovoljeno (*clip region*). Grafa namreč ne rišemo preko celotne risalne površine, ampak okoli risalne plošče pustimo primeren rob, katerega velikost določa konstanta `ROB`.

In kako izrišemo koordinatni osi? Ne pozabimo, da smo izhodišče koordinatnega sistema že prestavili na sredino. Ker je velikost enote odvisna od velikosti risalne površine, najprej ugotovimo velikost tega gradnika (metoda `getSize()`). V odvisnosti od širine risalne površine določimo tudi velikost izrisanih črtic, ki označujejo enote (1 in -1 na *y* osi ter mnogokratnik PI/4 na *x* osi), in velikost koraka, to je razmika med izrisanimi črticami. Osi *x* in *y* izrišemo preko celotnega območja risanja tako, da se sekata v središču koordinatnega sistema. Izris koordinatnih osi ni zapleten, potrebno je le temeljito premisliti, si skicirati na papir in morda tudi nekajkrat poizkusiti in po potrebi ustrezno popraviti kodo.

```
private void narisiOsi(Graphics g)
{
  Dimension velikost = this.getSize();
  int hMeja = velikost.width/2;
  int vMeja = velikost.height/2;
  int crtica = velikost.width/200;
  g.setColor(Color.black);
  g.drawLine(-hMeja, 0, hMeja, 0);
  int korak = hMeja/6;
  for(int i = korak; i <= hMeja; i += korak) {
    g.drawLine(i, crtica, i, -crtica);
    g.drawLine(-i, crtica, -i, -crtica);
  }
  g.drawLine(0, vMeja, 0, -vMeja);
  g.drawLine(-crtica, -2*vMeja/3, crtica, -2*vMeja/3);
  g.drawLine(-crtica, 2*vMeja/3, crtica, 2*vMeja/3);
}
```

Z metodo `setColor()` smo nastavili barvo risanja na črno barvo, torej se koordinatni sistem izriše s črnimi črtami.

Podobno se lotimo tudi izrisa same funkcije. Tukaj moramo najprej določiti razmerje med vrednostjo *x* in radiani (širina risalne površine naj bo enaka vrednosti 3 PI). Na *y* osi postavimo vrednost 1 na 2/3 izrisane koordinatne osi. Oboje smo pravzaprav določili že v prejšnji metodi ob izrisu koordinatnega sistema, ko smo izrisali oznake enot (črtice). Izrisa grafa se lotimo tako, da za vsako vrednost *x* izračunamo vrednost *y* = sin *x* (seveda z ustrezno pretvorbo enot) in dobljeno točko izrišemo kot majhen rdeč krogec.

```
private void narisiSinus(Graphics g)
{
  Dimension velikost = this.getSize();
  int hMeja = velikost.width/2;
  int vMeja = velikost.height/2;
  double razmerjeX = 3*Math.PI/(2*hMeja);
  double razmerjeY = 2*vMeja/3;
  int korak = hMeja/6;
  g.setColor(Color.red);
  for(int x = -hMeja; x <= hMeja; x++) {
    int y = (int) (razmerjeY*Math.sin(x*razmerjeX));
    y = -y;
    g.fillOval(x-1, y-1, 3, 3);
  }
}
```

Bodite pozorni na stavek:

```
y = -y;
```

Ker koordinata *y* v koordinatnem sistemu gradnika narašča od zgoraj navzdol, moramo izračunano vrednost *y* najprej obrniti (preslikati preko osi *x*), da jo pravilno izrišemo.

Celoten program `Sinus.java` je naslednji:

```
import java.awt.*;
import javax.swing.*;

public class Sinus extends JPanel
{
  private final int SIRINA = 800;
  private final int VISINA = 400;
  private final int ROB = 10;

  public static void main(String[] args)
  {
    JFrame okno = new JFrame("Graf sin(x)");
    Sinus graf = new Sinus();
    okno.getContentPane().add(graf);
    okno.setSize(graf.getSize().width, graf.getSize().height);
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.setVisible(true);
  }

  public Sinus()
  {
    setSize(SIRINA, VISINA);
  }

  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    Dimension velikost = this.getSize();
    g.setClip(ROB, ROB, velikost.width-2*ROB, velikost.height-2*ROB);
    g.translate(velikost.width/2, velikost.height/2);
    narisiOsi(g);
    narisiSinus(g);
  }

  private void narisiOsi(Graphics g)
  {
    Dimension velikost = this.getSize();
    int hMeja = velikost.width/2;
    int vMeja = velikost.height/2;
    int crtica = velikost.width/200;
    g.setColor(Color.black);
    g.drawLine(-hMeja, 0, hMeja, 0);
    int korak = hMeja/6;
    for(int i = korak; i <= hMeja; i += korak) {
      g.drawLine(i, crtica, i, -crtica);
      g.drawLine(-i, crtica, -i, -crtica);
    }
    g.drawLine(0, vMeja, 0, -vMeja);
    g.drawLine(-crtica, -2*vMeja/3, crtica, -2*vMeja/3);
    g.drawLine(-crtica, 2*vMeja/3, crtica, 2*vMeja/3);
  }

  private void narisiSinus(Graphics g)
  {
    Dimension velikost = this.getSize();
    int hMeja = velikost.width/2;
    int vMeja = velikost.height/2;
    // razmerje med x in radiani (1/6 hMeja je Pi/4)
    double razmerjeX = 3*Math.PI/(2*hMeja);
    // razmerje za y os (2/3 vMeja je 1)
    double razmerjeY = 2*vMeja/3;
    int korak = hMeja/6;
    g.setColor(Color.red);
```

```
    for(int x = -hMeja; x <= hMeja; x++) {
      int y = (int) (razmerjeY*Math.sin(x*razmerjeX));
      // obrni y koordinato (kartezicni sistem)
      y = -y;
      g.fillOval(x-1, y-1, 3, 3);
    }
  }
}
```

Program Sinus.java

Bitne slike

Bitne slike (kot so slike v datotekah *gif* ali *jpg*) so predstavljene z objekti razreda `java.awt.Image`.

Če imamo bitno sliko zapisano v datoteki (v formatu GIF, JPEG ali PNG), jo lahko preberemo z metodo `getImage()` in jo naložimo v pomnilnik. Omenjena metoda vrne objekt `Image`, ki je referenca na bitno sliko v pomnilniku.

Pred tem pa moramo s klicem metode `getDefaultToolkit()` dobiti tudi ustrezen izvod razreda, ki izvede vmesnik `Toolkit`, saj metoda `getImage()` pripada temu razredu:

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image slika = tk.getImage("slika.jpg");
```

Bitno sliko potem prikažemo s pomočjo metode `drawImage()` razreda `Graphics`.

Kot primer si poglejmo program `Slika.java`, ki iz tekočega direktorija na disku iz datoteke *slika.jpg* prebere sliko in jo prikaže na zaslonu (na risalni površini v oknu).

Okno in risalno površino določimo podobno kot v prejšnjem primeru (za opis glejte primer `Sinus.java` v razdelku *Risanje z grafičnimi primitivi*). Risalno površino predstavlja objekt razreda `Slika`, ki ga izpeljemo iz razreda `JPanel`. Preobložena metoda `paintComponent()` poskrbi za izris bitne slike, če le-ta obstaja:

```
public void paintComponent(Graphics g)
{
  if (bitna != null)
    g.drawImage(bitna, ROB, ROB, this);
}
```

Slika je shranjena v spremenljivki `bitna`, ki je privatni atribut razreda.

Za nalaganje slike poskrbimo že v konstruktorju razreda `Slika`. Konstruktor lahko zapišemo kot:

```
public Slika()
{
  bitna = Toolkit.getDefaultToolkit().getImage("slika.jpg");
  this.sirina = bitna.getWidth(this) + 2*ROB;
  this.visina = bitna.getHeight(this) + 2*ROB;
}
```

Na koncu (zadnja dva stavka) smo poskrbeli, da se velikost risalne površine prilega velikosti prebrane slike (z upoštevanjem nekaj roba okoli slike). Metodi `getWidth()` in `getHeight()` namreč vrneta širino in višino slike oziroma `-1`, če velikost slike še ni poznana. V praksi lahko ugotovimo, da se obe metodi izvršita veliko prej, kot se naloži cela slika v pomnilnik, zato vedno vrneta vrednost `-1`, ker velikost slike še ni poznana. Metoda `getImage()` namreč vrne le referenco na sliko, ki dobi podatke iz podane datoteke, ne poskrbi pa v celoti za prenos vseh teh podatkov iz datoteke in za pripravo slike na izris na gradniku. Zato moramo pred določitvijo velikosti risalne površine počakati, da se iz datoteke preberejo vsi podatki o sliki oziroma dokler ne dobimo pozitivnih vrednosti za širino in višino slike. Zato dodamo zanko, katera se v prazno izvaja toliko časa, dokler podatki o velikosti slike niso znani:

```
while((bitna.getWidth(this)<0)||(bitna.getHeight(this)<0))
  ;
```

Zanka se torej zaključi, ko obe metodi `getWidth()` in `getHeight()` vrneta pozitivni vrednosti.

Program `Slika.java` prikazuje, kako lahko enostavno izrišemo bitno sliko. Pri tem se nismo obremenjevali s tem, ali datoteka s sliko sploh obstaja in ali jo program lahko prebere. V primeru napak pri branju slike bi naš program obtičal v neskončni zanki (saj velikost neobstoječe slike ni nikoli znana) in se ne bi nikoli končal. Problem lahko enostavno rešimo tako, da v zanko vpeljemo števec `i` in izvajanje zanke prekinemo, ko števec `i` preseže neko vnaprej določeno veliko vrednost.

Datoteka `Slika.java` pri primerih tega poglavja vsebuje celotno kodo:

```
import java.awt.*;
import javax.swing.*;

public class Slika extends JPanel
{
  private final int ROB = 10;
  private final int NASLOV = 34;
  private final int OKVIR = 8;
  private int sirina = 800;
  private int visina = 400;
  private Image bitna = null;

  public static void main(String[] args)
  {
    JFrame okno = new JFrame("Prikaz bitne slike");
    Slika s = new Slika();
    okno.getContentPane().add(s);
```

```
    okno.setSize(s.sirina + s.OKVIR, s.visina + s.NASLOV);
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.setVisible(true);
  }

  public Slika()
  {
    int i = 0;
    bitna = Toolkit.getDefaultToolkit().getImage("slika.jpg");
    while(((bitna.getWidth(this)<0)||(bitna.getHeight(this)<0))&&(++i<10000000))
      ;
    this.sirina = bitna.getWidth(this) + 2*ROB;
    this.visina = bitna.getHeight(this) + 2*ROB;
  }

  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    if (bitna != null)
      g.drawImage(bitna, ROB, ROB, this);
  }
}
```

Program Slika.java

Risanje na bitno sliko v ozadju

Poglejmo si še en primer, kako lahko bitno sliko uporabimo pri risanju na gradnik. V našem naslednjem primeru bomo v okno postavili risalno površino in gumb. Na risalni površini se bojo ob vsakem miškinem kliku okoli točke klika narisali krogi. S klikom na gumb pa bomo izbrisali celo risalno površino.

Program bomo napisali tako, da se risanje ne bo izvajalo neposredno na risalno površino, temveč bo potekalo na sliko risalne površine v pomnilniku in šele po zaključenem izrisu bomo z metodo `paint` poskrbeli za prikaz te slike risalne površine na zaslonu. Tak postopek imenujemo dvojno pomnjenje (*double buffering*).

Swing ima vgrajeno podporo za dvojno pomnjenje in vsi gradniki Swing privzeto uporabljajo to tehniko.

Za začetek napišemo razred `Krogi`, ki vsebuje `main` metodo. V njej ustvarimo `okno`, objekt razreda, izpeljanega iz `JFrame`, ter ga prikažemo na zaslonu:

```
public class Krogi
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}
```

Razred `Okvir`, ki je izpeljan iz razreda `JFrame`, ima med drugim dva atributa. Eden je gumb `JButton`, drugi pa določa risalno površino, to je objekt razreda `Risalnica`, izpeljanega iz razreda `JPanel`. V konstruktorju razreda `Okvir` poskrbimo za primerno postavitev gradnikov in druge, sedaj nam že domače podrobnosti:

```
class Okvir extends JFrame
{
  private Container vsebina = null;
  private Risalnica platno = new Risalnica();
  private JButton brisi = new JButton("Briši");

  public Okvir()
  {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ...
    vsebina = this.getContentPane();
    vsebina.add(brisi, BorderLayout.SOUTH);
    vsebina.add(platno, BorderLayout.CENTER);
  }
}
```

Risalna površina se mora tudi odzivati na miškine klike. Zato ji moramo dodati ustreznega poslušalca, to je `MouseListener`, ki poskrbi za primeren odziv. Če uporabimo vmesnik `MouseListener`, moramo preobložiti vse štiri njegove metode, čeprav nas zanima le ena metoda, to je `mouseClicked()`. Lahko bi sicer namesto vmesnika uporabili razred `MouseAdapter`, a ker je razred `Okvir` že izpeljan iz razreda `JFrame`, ne more istočasno razširjati tudi razreda `MouseAdapter`. Ena rešitev je, da definiramo povsem nov razred za poslušalca. A ker gre le za enostaven odziv na dogodek, je krajša in bolj enostavna uporaba notranjega anonimnega razreda:

```
platno.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent e)
  {
    platno.narisiKrog(e.getX(), e.getY());
    platno.repaint();
  }
});
```

Z metodama `getX()` in `getY()` smo dobili koordinati (x,y) tiste točke v gradniku, v kateri se je zgodil miškin klik. Obe vrednosti posredujemo metodi `narisiKrog()`.

Gumbu smo določili oranžno barvo ozadja (klic metode `setBackground()`) in črno barvo ospredja oziroma barvo napisa (metoda `setForeground()`). Ker se gumb tudi odziva na klik, mu moramo dodati poslušalca za ta dogodek (`ActionListener`). Tudi tu smo se odločili za anonimen notranji razred, saj gre le za enostaven odziv na dogodek:

```
brisi.setBackground(Color.orange);
brisi.setForeground(Color.black);
```

```
brisi.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent d) {
    platno.brisiRisalnico();
    platno.repaint();
  }
});
```

V razredu `Risalnica` definiramo obe metodi (za risanje krogov in za brisanje risalne površine). V razredu imamo privatno spremenljivko `drugaSlika`, ki predstavlja sliko risalne površine v pomnilniku:

```
private BufferedImage drugaSlika = null;
```

Preobložimo metodo `paintComponent()`, v kateri pa nimamo nobene kode za risanje, temveč le prikažemo sliko risalne površine (če ta obstaja). Seveda pred tem poskrbimo za pravilen izris in čiščenje ozadja:

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  if (drugaSlika != null)
    g.drawImage(drugaSlika, 0, 0, this);
}
```

Tudi metoda `brisiRisalnico()` je enostavna. Spremenljivko `drugaSlika` postavi na `null` in s tem se obstoječa slika izbriše:

```
public void brisiRisalnico()
{
  drugaSlika = null;
}
```

Vso kodo v zvezi z risanjem tako postavimo v metodo `narisiKrog()`, ki poskrbi za izris krogov. Če slika `drugaSlika` ne obstaja, se najprej kreira nova slika enake velikosti kot je risalna površina. To dosežemo s klicem konstruktorja `BufferedImage`, ki mu kot argumente podamo velikost slike in njen tip. Slednji je kar ena od konstant tega razreda; v našem primeru smo izbrali 8-bitni RGBA barvni model. Potem s klicem metode `getGraphics()` dobimo referenco na objekt `Graphics`, to je grafični kontekst te slike, s pomočjo katerega lahko rišemo na sliko. Naključno izberemo barvo risanja izmed različnih barv v tabeli `barve` ter v zanki izrišemo pet krogov, vsakega z malo večjim polmerov, središče vseh pa je v točki (*x*, *y*); *x* in *y* sta metodi podana kot argumenta:

```
public void narisiKrog(int x, int y)
{
  if (drugaSlika == null)
    drugaSlika =
      new BufferedImage(getSize().width, getSize().height,
                        BufferedImage.TYPE_4BYTE_ABGR);
```

```
    Graphics g = drugaSlika.getGraphics();
    g.setColor(barve[(int) (Math.random()*barve.length)]);
    for(int r=5; r<26; r+=5)
      g.drawOval(x-r, y-r, 2*r, 2*r);
  }
```

Datoteka `Krogi.java` z izvorno kodo programa:

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.image.*;

public class Krogi
{
  public static void main(String[] args)
  {
    Okvir okno = new Okvir();
    okno.setVisible(true);
  }
}

class Okvir extends JFrame
{
  private final int SIRINA = 600;
  private final int VISINA = 400;
  private String naslov = "Risanje krogov z miško";
  private Container vsebina = null;
  private Risalnica platno = new Risalnica();
  private JButton brisi = new JButton("Briši");

  public Okvir()
  {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle(naslov);
    setSize(SIRINA, VISINA);
    setResizable(false);
    brisi.setBackground(Color.orange);
    brisi.setForeground(Color.black);
    brisi.addActionListener( new ActionListener() {
      public void actionPerformed(ActionEvent d) {
        platno.brisiRisalnico();
        platno.repaint();
      }
    });
    platno.addMouseListener(new MouseAdapter() {
      public void mouseClicked(MouseEvent e)
      {
        platno.narisiKrog(e.getX(), e.getY());
        platno.repaint();
      }
    });
    vsebina = this.getContentPane();
    vsebina.add(brisi, BorderLayout.SOUTH);
    vsebina.add(platno, BorderLayout.CENTER);
  }
}

class Risalnica extends JPanel
{
  private Color[] barve =
    {Color.yellow, Color.orange, Color.red, Color.black,
     Color.blue, Color.green, Color.cyan, Color.magenta};
  private BufferedImage drugaSlika = null;
```

```
  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    if (drugaSlika != null)
      g.drawImage(drugaSlika, 0, 0, this);
  }

  public void narisiKrog(int x, int y)
  {
    if (drugaSlika == null)
      drugaSlika =
        new BufferedImage(getSize().width, getSize().height,
                          BufferedImage.TYPE_4BYTE_ABGR);
    Graphics g = drugaSlika.getGraphics();
    g.setColor(barve[(int) (Math.random()*barve.length)]);
    for(int r=5; r<26; r+=5)
      g.drawOval(x-r, y-r, 2*r, 2*r);
  }

  public void brisiRisalnico()
  {
    drugaSlika = null;
  }
}
```

Program Krogi.java

Apleti

Apleti so javanski programčki, ki se ne izvajajo samostojno, saj je njihov namen razširitev zmogljivosti njihovega izvajalnega okolja. Aplet je torej javanska koda, ki se izvaja v pregledovalniku *Applet Viewer* ali znotraj javansko osveščenega brskalnika.

Telo apleta mora biti razred, ki je izpeljan iz že pripravljenega razreda `Applet` (iz paketa `java.applet`) ali pa iz njega izpeljanega razreda `JApplet`, kadar gre za Swing komponento (paket `javax.swing`). Oba razreda ponujata metode za vzpostavitev, zagon, zaustavitev in uničenje apleta (metode `init()`, `start()`, `stop()` in `destroy()`, po vrsti).

Za razliko od samostojnih javanskih programov, ki morajo imeti vstopno metodo `main()`, kjer se začne izvajati koda, imajo apleti metodo `init()`, v kateri se začne izvajati koda. Zato ponavadi v apletu preobložimo `init` metodo, ostale štiri metode pa le po potrebi.

Najpreprostejši aplet, ki izpiše niz “Pozdravljeni!”, bi torej lahko izgledal takole:

```
import javax.swing.*;

public class MojAplet extends JApplet
{
  public void init()
  {
```

```
    this.getContentPane().add(new JLabel("  Pozdravljeni!  "));
  }
}
```

Niz “Pozdravljeni!” smo zapisali v oznako (gre za grafični program!), ki smo jo postavili na delovno površino apleta.

Za pregled apletov moramo napisati tudi html datoteko, kjer med oznakama `<APPLET>` in `</APPLET>` določimo datoteko s prevedeno javansko kodo in (opcijsko) velikost okna, v katerem se bo izvajala. V našem primeru gre za javansko vmesno kodo iz datoteke `MojAplet.class`, velikost pa je 300 pik po širini in 50 pik po višini:

```
<APPLET CODE="MojAplet.class" WIDTH=300 HEIGHT=50></APPLET>
```

Obe datoteki, `MojAplet.java` in `MojApletTest.html`, najdete med primeri tega poglavja.

```
import javax.swing.*;

public class MojAplet extends JApplet
{
  public void init()
  {
    this.getContentPane().add(new JLabel("  Pozdravljeni!  "));
  }
}
```

Program MojAplet.java

```
<HTML>
<HEAD><TITLE>Moj aplet</TITLE></HEAD>
<BODY>
  <APPLET CODE="MojAplet.class" WIDTH=300 HEIGHT=50></APPLET>
</BODY>
</HTML>
```

Datoteka MojApletTest.html

Delovanje primera lahko preizkusimo tako, da v ukaznem oknu pokličemo pregledovalnik apletov, ki mu kot argument podamo html datoteko:

```
appletviewer MojApletTest.html
```

Druga možnost pa je, da html datoteko `MojApletTest.html` odpremo neposredno v spletnem brskalniku.

Za konec pa si poglejmo še primer apleta, ki podan znesek preračuna iz tolarjev v evre. Ker smo vse prvine tega programa (gradnike oznaka, vnosno polje, gumb, vsebnike,

razporejevalnike ter odziv gumba na akcijo) spoznali in opisali že na začetku tega poglavja, jih tu ne bomo ponovno razlagali, temveč zapišimo le kodo tega apleta:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Menjava extends JApplet
{
  final double tecaj = 239.64;
  JLabel znesek = new JLabel("Vpiši znesek v SIT: ");
  JLabel rez = new JLabel("");
  JTextField vnos = new JTextField("0", 6);
  JButton gumb = new JButton("Pretvori v EUR");

  public void init()
  {
    JPanel izracun = new JPanel();
    izracun.add(znesek);
    izracun.add(vnos);
    JPanel gumbi = new JPanel();
    gumbi.add(gumb);
    JPanel izpis = new JPanel();
    izpis.add(rez);
    Container vsebina = getContentPane();
    vsebina.add(izracun, BorderLayout.NORTH);
    vsebina.add(gumbi, BorderLayout.CENTER);
    vsebina.add(izpis, BorderLayout.SOUTH);
    gumb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent d) {
          rez.setText(vnos.getText() + "  SIT  =  " +
Math.round(100*Double.parseDouble(vnos.getText())/tecaj)/100.0 + "  EUR");
        }
    });
  }
}
```

Program Menjava.java

Pripadajočo html datoteko pa lahko zapišemo takole:

```
<HTML>
<HEAD><TITLE>Primer apleta</TITLE></HEAD>
<BODY>
  <H2>Pretvorba SIT v EUR</H2><BR>
  <APPLET CODE="Menjava.class" WIDTH=300 HEIGHT=100></APPLET>
</BODY>
</HTML>
```

Datoteka menjava.html

Omenjeno kodo najdete v datotekah `Menjava.java` in `menjava.html`.

Apleti in aplikacije

Kako izbrati, ali naj program napišemo, da bo deloval kot aplet ali kot samostojen program (aplikacija)? In zakaj ne bi bil naš program kar oboje hkrati?

Seveda lahko zapišemo kodo tudi tako, da naš program po potrebi deluje kot samostojen program (aplikacija) ali pa znotraj spletnega brskalnika (aplet)[1]. Izhajamo iz kode apleta:

```
public class Aplet extends JApplet
{
  public void init()
  {
    // telo apleta
  }
}
```

Če želimo ta aplet poganjati kot samostojen program, mu moramo za začetek dodati metodo `main`, saj se vsak program začenja v tej metodi (podobno kot se vsak aplet začenja v metodi `init` in zažene z metodo `start`). Torej imamo:

```
public class Aplet extends JApplet
{
  public void init()
  {
    // telo apleta
  }

  public void main(String[] args)
  {
    // poskrbimo za ustrezen zagon apleta
  }
}
```

V metodi `main` moramo torej poskrbeti za ustvarjanje primerno velikega okna, kateremu na delovno površino postavimo objekt razreda `Aplet`. Ne smemo pozabiti na klica obeh metod, s katerima poskrbimo za pravilen zagon apleta, torej `init` in `start`, ter seveda tudi na prikaz samega okna.

Pa pojdimo lepo po vrsti. Najprej ustvarimo objekt razreda `Applet`:

```
Aplet aplet = new Aplet();
```

Ustvarimo tudi novo okno, ki mu določimo ime, velikost in poskrbimo za ustrezno reakcijo ob zapiranju okna:

[1] Pri tem seveda nismo upoštevali tistih omejitev, ki jih imajo apleti zaradi varnostnih razlogov, kot tudi ne posebnih zmožnosti, ki so specifične le za aplete.

```
JFrame okno = new JFrame("Okno");
okno.setSize(400,200);
okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Nato postavimo objekt `aplet` v središče delovne površine okna (spomnimo se, da je privzet razporejevalnik za okna `BorderLayout`):

```
okno.getContentPane().add(aplet, BorderLayout.CENTER);
```

Na koncu moramo poskrbeti še za pravilen zagon apleta:

```
aplet.init();
aplet.start();
```

ter za prikaz okna na zaslonu:

```
okno.setVisible(true);
```

Metodo `main` bi torej lahko zapisali takole:

```
public static void main(String[] args)
{
  Aplet aplet = new Aplet();
  JFrame okno = new JFrame("Okno");
  okno.setSize(400,200);
  okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  okno.getContentPane().add(aplet, BorderLayout.CENTER);
  aplet.init();
  aplet.start();
  okno.setVisible(true);
}
```

Program `ApletProgram.java` prikazuje tak dvoličen program, ki ga lahko izvajamo samostojno ali pa znotraj spletnega brskalnika. Program ne naredi nič posebnega, prikaže le eno oznako z vnaprej določenim besedilom.

```
import javax.swing.*;
import java.awt.*;

public class ApletProgram extends JApplet
{
  public void init()
  {
    this.getContentPane().add(new JLabel("   Program ali aplet?   Oboje! "));
  }

  public static void main(String[] args)
  {
    ApletProgram aplet = new ApletProgram();
    JFrame okno = new JFrame("Program: ApletProgram.class");
    okno.setSize(400,200);
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.getContentPane().add(aplet, BorderLayout.CENTER);
```

```
    aplet.init();
    aplet.start();
    okno.setVisible(true);
  }
}
```

ApletProgram.java

Seveda ne smemo pozabiti na pripadajočo html kodo (`ApletProgramTest.html`), ki jo uporabimo v primeru prikazovanja znotraj spletnega brskalnika ali programa *appletviewer*.

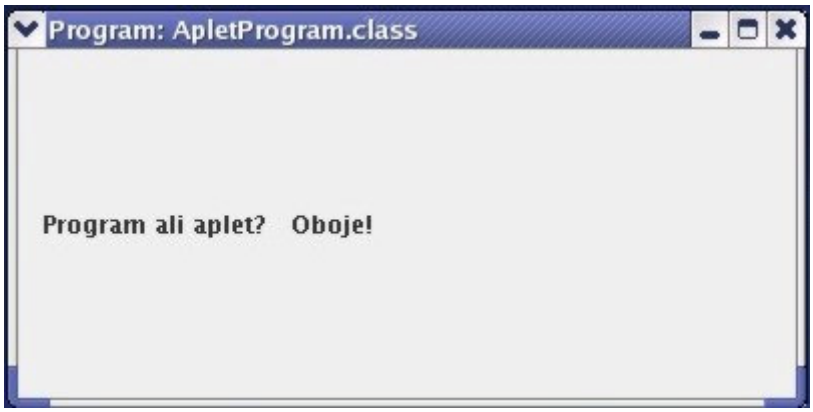
```
<HTML>
<HEAD><TITLE>Aplet ali program? Aplet.</TITLE></HEAD>
<BODY>
  <APPLET CODE="ApletProgram.class" WIDTH=400 HEIGHT=200></APPLET>
</BODY>
</HTML>
```

ApletProgramTest.html

Program prevedemo, nato pa ga lahko izvajamo na dva načina. Z ukazom

```
java ApletProgram
```

poženemo samostojen program, katerega rezultat je okno, prikazano na sliki 4.9.

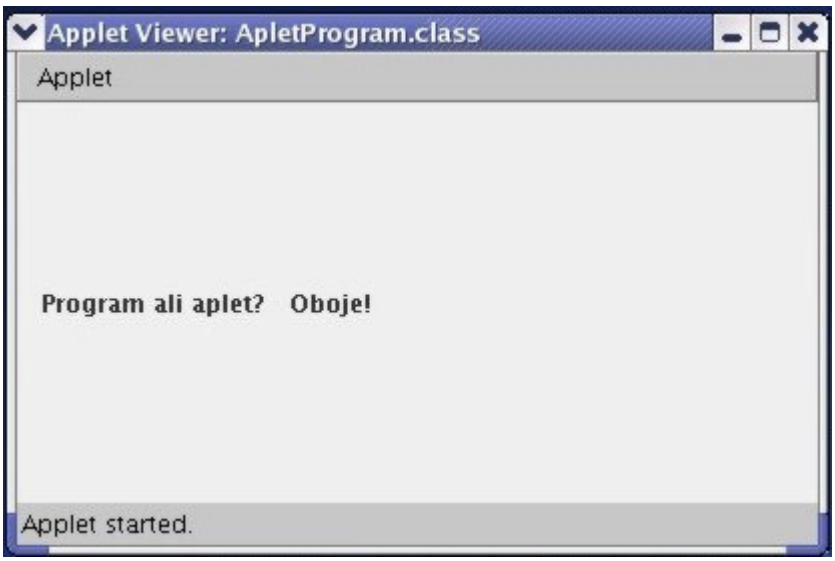


Slika 4.9: Program poženemo kot aplikacijo (ukaz `java ApletProgram`).

Podoben rezultat dobimo tudi z ukazom

```
appletviewer ApletProgramTest.html
```

pri čemer se naš program požene kot aplet znotraj html dokumenta (slednje je podobno, kot če html dokument odpremo v spletnem brskalniku). Rezultat je okno, ki ga prikazuje slika 4.10.



Slika 4.10: Program poženemo kot aplet (ukaz `appletviewer ApletProgram.html`).

V obeh primerih (sliki 4.9 in 4.10) se torej odpre okno, v katerem se prikaže aplet, znotraj njega pa se v oznaki izpiše neko besedilo.

In kako lahko ugotovimo, ali se program izvaja samostojno ali kot aplet? Če pozorno pogledamo obe sliki (4.9 in 4.10), opazimo, da se izgled samega okna nekoliko razlikuje. Poleg tega pa smo v primeru samostojnega programa (v metodi `main`) naslov okna določili sami (izbrali smo naslov "Program: ApletProgram.class"), medtem ko na naslov okna pri uporabi ukaza *appletviewer* nimamo vpliva (spletni brskalnik pa apleta sploh ne prikaže v samostojnem oknu, temveč ga vključi neposredno v vsebino strani).

5. Datoteke in tokovi

Tokovi (*streams*) so objektna predstavitev vhodno/izhodnih pogovorov in omogočajo branje oz. zapisovanje podatkov. Prednost uporabe tokov je v tem, da različne zunanje enote (kot so zaslon, datotečni sistem, tiskalnik, omrežna vtičnica, idr.) prikažejo kot razred, ki omogoča njihovo preprosto obravnavo pri izmenjavi podatkov. Tako lahko tokove uporabljamo tudi za dostop do datotek.

Izraz tok se nanaša na zaporedni dostop do podatkov, kjer vsebino preberemo kot zaporedje bajtov. Razrede za delo s tokovi ponuja paket `java.io`.

Tokove lahko v grobem razdelimo na znakovne in binarne tokove. Vsaka od skupin pa se deli tudi na vhodne in izhodne tokove. Iz tega izhajajo tudi štirje temeljni razredi, ki jih v Javi uporabljamo pri delu s tokovi. Za binarne tokove (tok bajtov) sta temeljna razreda `InputStream`, ki predstavlja vhodne tokove za branje, in `OutputStream`, ki zajema izhodne tokove za pisanje. Pri znakovnih tokovih pa paket `java.io` nudi temeljna razreda `Reader` (za branje) in `Writer` (za pisanje). Ponavadi v praksi uporabljamo razrede, ki so izpeljani iz teh temeljnih razredov.

Nov tok ustvarimo s klicem konstruktorja. Tok lahko zapremo eksplicitno s klicem metode `close()` ali pa počakamo, da se zapre implicitno, ko ga pospravi smetar (ko se nanj ne sklicuje nihče več). Ker pa so tokovi pogosto vezani na pomembna sistemska sredstva, je zelo priporočljivo, da jih vedno zapiramo z metodo `close()`, takoj ko je to mogoče.

Preden pa se lotimo konkretnih primerov dela z datotekami in tokovi, si poglejmo še izjeme in njihovo obravnavo, saj se bomo pri delu z datotekami zagotovo srečali z njimi.

Izjeme

Javanska koda, če je napisana pravilno, je lahko zelo zanesljiva in trdoživa. K temu pripomore tudi mehanizem izjem, ki je vgrajen v jezik in izvajalno okolje. Tako lahko v programu predvidimo tudi neobičajne okoliščine delovanja in nanje ustrezno reagiramo.

Kaj so izjeme

Med procesom izvajanja programa se lahko pripetijo različni neobičajni dogodki. Obravnava problematičnih okoliščin, ki jih imenujemo tudi izjemni dogodki ali izjeme (*exceptions*), je pri Javi vgrajena v sam jezik.

Izjeme so torej nepričakovani dogodki ali napake, ki se pojavijo med izvajanjem programa, zaradi katerih se prekine normalen potek programskih ukazov, saj nadaljnje

izvajanje ni mogoče. Primeri izjem so deljenje z nič, branje iz neobstoječe datoteke, nedostopno omrežje, napačen URL naslov, dostop do elementa z indeksom izven meja tabele, ...

Vsaka metoda lahko poleg običajne vrnjene vrednosti vrne tudi objekt, ki opisuje izjemni dogodek. Temu rečemo, da metoda sproži izjemo (*throws an exception*). Izvajanje metode se zaključi v trenutku, ko sproži izjemo. Takrat se ustvari nov objekt, ki opisuje to izjemo, in ta potuje nazaj po klicnem skladu (*method call stack*), dokler ne najde primernega prestreznika izjem (*exception handler*).

Vrste izjem

Posebno obnašanje izjem in njihovo potovanje po klicnem skladu omogoča razred `Throwable`, ki je skupni predhodnik vseh izjemnih dogodkov. Iz njega sta izpeljani dve veji, razred `Error` in razred `Exception`. Prvi je namenjen opisu hudih napak, zaradi katerih program ne more več nadaljevati (na primer izčrpanje ključnega vira, kot je pomnilnik). V tem primeru ne moremo narediti veliko, zato ponavadi takih napak sploh ne obravnavamo. Drugi pa opisuje izjeme, katerih obravnavo ponavadi vključimo v pravilno napisan program.

Razred `RuntimeException` je neposredno izpeljan iz razreda `Exception` in opisuje izjeme, ki so posledica napake v programu (na primer prekoračitev obsega polja ali neprimerna pretvorba tipov). Izjemam te vrste bi se s previdnejšim kodiranjem lahko izognili, zato jih ponavadi v kodi niti ne razglašamo niti ne prestrezamo.

Vsi ostali potomci razreda `Exception` pa opisujejo izjeme, ki jih v programu težko zaznamo in preprečimo (na primer nedostopno omrežje ali nedovoljen dostop do datoteke). Zato zanje poskrbimo s programsko obravnavo (razglašanjem ali prestrezanjem) teh izjem.

Proženje izjem

Kadar med delovanjem metode lahko pride do izjeme, tako metodo posebej označimo, da metoda lahko sproži izjemo.

Če želimo razglasiti, da med delovanjem metode lahko pride do izjeme, zapišemo v glavi metode tudi tip možne izjeme. Pri tem uporabimo rezervirano besedo `throws`:

```
public int pisiDat(String imeDat) throws IOException
```

Metoda `pisiDat` zapiše podatke v datoteko `imeDat`. Pri tem lahko pride do izjeme `IOException`, ki jo metoda sproži. Tako prevajalnik ve, da metoda sicer vrača

vrednost `int`, a se ob napaki lahko tudi predčasno konča in sproži izjemo tipa `IOException`. Ista metoda lahko razglasi tudi več izjem:

```
public int pisiDat(String imeDat)
  throws FileNotFoundException, IOException
```

Ko v metodi zaznamo neobičajno stanje ali dogodek, ki onemogoči pravilno delovanje metode, znotraj metode sprožimo izjemo:

```
throw new IOException("Datoteka ne obstaja");
```

Ustvarili smo nov objekt `IOException` z opisom izjeme (objekt mora biti tipa `Throwable`). S stavkom `throw` prekinemo izvajanje metode ter posredujemo izjemo (novo ustvarjeni objekt) po klicnem skladu do ustreznega prestreznika izjem.

Kjer kličemo metode, ki razglašajo izjeme, imamo tri možnosti za obravnavo izjem:
- izjemo lahko spustimo nazaj po klicnem skladu,
- izjemo prestrežemo in jo ustrezno obdelamo ali pa
- izjemo prestrežemo in sprožimo drugo izjemo (novo izjemo bomo morali obravnavati kje drugje).

Kadar se odločimo za možnost, da izjemo spustimo nazaj po klicnem skladu, moramo v glavi metode le razglasiti njen tip (na primer `throws IOException`). S tem povemo, da naša metoda ne bo obravnavala izjem, ampak jih bo le posredovala naprej po klicnem skladu.

Prestrezanje in obravnavo izjem pa opisuje naslednji razdelek.

Obravnava izjem

Obravnava izjem (*exception handling*) pomeni prestrezanje sproženih izjem in ustrezno reakcijo nanje.

Lasten prestreznik izjem zapišemo s stavčnim blokom `try`/`catch`/`finally`, ki ga simbolično lahko zapišemo takole:

```
try {
  // stavki, ki lahko sprožijo izjemo
}
catch (XException e) {
  // obravnava izjeme XException
}
catch (YException e) {
  // obravnava izjeme YException
}
finally {
```

```
  // stavki za varen zaključek dela
}
```

Kodo pri kateri lahko pride do izjeme, postavimo v `try` blok.

Vsako posamezno izjemo obravnava lasten prestreznik `catch`. Pri tem lahko izjeme tudi združujemo in jih več obravnavamo v istem prestrezniku. Ker so vse izjeme izpeljane iz razreda `Exception`, bi univerzalni prestreznik za poljubne izjeme lahko zapisali takole:

```
catch (Exception e) {
  // obravnava vseh izjem
}
```

Ena od možnih obravnav izjem je izpis podrobnega sporočila izjeme (metoda `getMessage()`) skupaj z izpisom klicnega sklada (metoda `printStackTrace()`):

```
catch (Exception e) {
  System.out.println(e.getMessage());
  e.printStackTrace();
}
```

Stavki znotraj bloka `finally` se izvedejo v vsakem primeru, če je do izjeme prišlo ali tudi če do nje ni prišlo. Izvedejo se tudi, če pride do izjeme v bloku `catch`. Blok `finally` je opcijski, zato ni nujno, da ga podamo.

Tak način obravnave izjem, kjer pišemo ločeno kodo za njihovo prestrezanje, prispeva k jasnosti kode in posredno tudi k manj napakam v programih.

Tok bajtov

Za prenašanje binarne vsebine (slike, zvok, zaporedna predstavitev objektov) uporabljamo razreda `InputStream` (za branje) in `OutputStream` (za pisanje) ter njune potomce, saj z njimi obdelujemo posamezne bajte.

Razred `FileInputStream` je namenjen branju binarnih podatkov (bajtov) iz datotek v datotečnem sistemu. Podobno lahko za pisanje binarnih podatkov v datoteke uporabimo razred `FileOutputStream`. Oba razreda imata konstruktor, ki kot argument prejme niz znakov, ki določa ime datoteke. Nov tok za branje lahko torej odpremo s klicem konstruktorja:

```
InputStream vhodniTok = new FileInputStream(ime);
```

Za ustvarjanje toka za pisanje spet kličemo ustrezen konstruktor:

```
OutputStream izhodniTok = new FileOutputStream(drugoIme);
```

Oba razreda pa imata tudi konstruktor, ki kot argument prejme objekt `File`, ki predstavlja datoteko. Z uporabo tega konstruktorja lahko nov tok za branje iz datoteke odpremo v dveh korakih. Najprej ustvarimo nov objekt, ki predstavlja našo datoteko, nato pa s pomočjo tega objekta ustvarimo še vhodni tok:

```
File datoteka = new File(ime);
InputStream vhodniTok = new FileInputStream(datoteka);
```

Za branje enega bajta iz datoteke uporabimo metodo `read()`, ki vrne prebrani bajt oziroma vrednost `-1`, če smo že prišli do konca datoteke. Branje datoteke tako izvajamo v zanki, dokler je prebrana vrednost različna od `-1`:

```
int bajt;
while ((bajt = vhodniTok.read()) != -1)
  ...
```

Za zapisovanje enega bajta v datoteko pa uporabimo metodo `write()`:

```
izhodniTok.write(bajt);
```

Po zaključku dela z datotekami moramo datoteke tudi zapreti, za kar uporabimo metodo `close()`:

```
vhodniTok.close();
izhodniTok.close();
```

Seveda pri delu s tokovi ne smemo pozabiti napovedati uporabe paketa z ustreznimi razredi:

```
import java.io.*;
```

Pri delu z datotekami lahko pride tudi do nepredvidenih dogodkov, povezanih predvsem z V/I operacijami (torej do vhodno/izhodnih izjem). V JDK dokumentaciji so pri opisih posameznih metod navedene tudi vrste izjem, ki jih posamezna metoda razglaša (to pomeni, da ta metoda lahko sproži ali posreduje to vrsto izjeme). Tako na primer pri klicu konstruktorjev `FileInputStream(ime)` ali `FileOutputStream(ime)` lahko pride do izjeme `FileNotFoundException` (ki je posebna vrsta izjeme `IOException`), pri branju iz datoteke ali pisanju v datoteko pa se lahko sproži izjema `IOException`. Slednja je možna tudi pri zapiranju datoteke.

Seveda moramo v dobro napisanem programu poskrbeti tudi za ustrezno obravnavo možnih izjem.

Vse navedeno lahko povzamemo na enostavnem primeru, kjer vsebino poljubne datoteke prepišemo v drugo datoteko (imeni obeh datotek podamo kot argumenta programa). Ker gre za poljubno binarno datoteko, bo prepisovanje potekalo po bajtih.

Pri obravnavi izjem (možne so različne izjeme, a so vse vrste `IOException`) smo se zaradi enostavnosti kode odločili, da v programu izjem ne bomo obravnavali, ampak jih bomo posredovali v izvajalno okolje. Zato v `main` metodi razglasimo izjemo `IOException`.

Izvorna koda programa je v datoteki `Prepisi.java`.

```
import java.io.*;

public class Prepisi
{
  public static void main(String[] args) throws IOException
  {
    if(args.length < 2) {
      System.out.println("Uporaba: java Prepisi izvorDat ponorDat");
      System.exit(1);
    }
    InputStream vhTok = new FileInputStream(args[0]);
    OutputStream izhTok = new FileOutputStream(args[1]);
    int bajt;
    while ((bajt = vhTok.read()) != -1)
      izhTok.write(bajt);
    vhTok.close();
    izhTok.close();
  }
}
```

Program `Prepisi.java`

Če bi želeli v našem programu obravnavati tudi izjeme, do katerih lahko pride pri odpiranju datotek ali pri branju oziroma pisanju, bi morali ustrezne kritične dele postaviti v blok `try` in ustrezne izjeme ujeti v bloku `catch`. V blok `finally` bi postavili zapiranje obeh datotek, saj naj bi program kljub izjemi poskrbel za sprostitev virov. Še vedno pa mora `main` metoda razglasiti izjemo, saj v primeru, ko do nje pride pri zapiranju datotek (v bloku `finally`), te ne ulovimo, temveč jo le posredujemo naprej po klicnem skladu.

Tako dopolnjen program `Prepisi1.java` ima malo več vrstic, a koda še vedno ostaja dovolj pregledna.

```
import java.io.*;

public class Prepisi1
{
  public static void main(String[] args) throws IOException
  {
    if(args.length < 2) {
      System.out.println("Uporaba: java Prepisi izvorDat ponorDat");
```

```
      System.exit(1);
    }
    InputStream vhTok = null;
    OutputStream izhTok = null;
    try {
      vhTok = new FileInputStream(args[0]);
      izhTok = new FileOutputStream(args[1]);
      int bajt;
      while ((bajt = vhTok.read()) != -1)
        izhTok.write(bajt);
    }
    catch(FileNotFoundException e) {
      System.out.println("Napaka pri odpiranju datoteke: " + e.getMessage());
      System.exit(1);
    }
    catch(IOException e) {
      System.out.println("Napaka: " + e.getMessage());
      System.exit(1);
    }
    finally {
      vhTok.close();
      izhTok.close();
    }
  }
}
```

Program Prepisi1.java

Tok znakov

Razreda `Reader` (za branje) in `Writer` (za pisanje) ter njune potomce uporabljamo za prenašanje besedila, saj z njimi lahko prenašamo posamezne znake. Neposredno iz razreda `Reader` je izpeljan razred `InputStreamReader`, ki je neke vrste most med bitnim tokom in znakovnim tokom (bere bajte in jih dekodira v znake skladno s kodno tabelo). Iz njega pa je izpeljan razred `FileReader`, ki ga lahko uporabimo za branje znakovnih datotek. Podobno tudi za pisanje znakovnih datotek lahko uporabimo razred `FileWriter`, ki je preko razreda `OutputStreamWriter` izpeljan iz razreda `Writer`.

Za branje znakovnih datotek tako ponavadi uporabljamo razred `FileReader`, razred `FileWriter` pa za pisanje znakovnih datotek. Ustrezen tok odpremo s klicem konstruktorja, ki mu podamo ime datoteke:

```
FileReader vhodniTok = new FileReader(ime);
FileWriter izhodniTok = new FileWriter(ime);
```

Spremenimo naš prejšnji program tako, da bo prepisoval datoteko po znakih. Zato odpremo primeren vhodni in izhodni tok, medtem ko sta preobloženi metodi za branje in pisanje podobni. Kodo spremenjenega programa najdete v datoteki `PrepisiZnake.java`.

```
import java.io.*;

public class PrepisiZnake
```

```
{
  public static void main(String[] args) throws IOException
  {
    if(args.length < 2) {
      System.out.println("Uporaba: java Prepisi izvorDat ponorDat");
      System.exit(1);
    }
    FileReader vhTok = new FileReader(args[0]);
    FileWriter izhTok = new FileWriter(args[1]);
    int znak;
    while ((znak = vhTok.read()) != -1)
      izhTok.write(znak);
    vhTok.close();
    izhTok.close();
  }
}
```

Program PrepisiZnake.java

Ovijanje tokov

V prejšnjih razdelkih smo spoznali, kako beremo iz datotečnega toka temeljne informacije, to je bajte oziroma znake. Velikokrat pa bi želeli iz datoteke prebrati druge tipe podatkov, kot so na primer cela ali realna števila. V takih primerih uporabimo druge razrede, ki jih nudi paket `java.io`, in jih po potrebi povežemo z datotečnimi tokovi. Takemu sestavljanju tokov pravimo tudi ovijanje tokov.

Tokove ovijamo tako, da konstruktorju novega toka podamo tok, ki ga želimo oviti. Tako na primer tok `FileInputStream` bere binarne podatke iz datoteke. Če želimo iz datoteke prebirati realna števila, bomo obstoječi datotečni tok ovili z razredom `DataInputStream`, ki zna prebrane bajte združiti v primitivni tip `double` (realna števila):

```
FileInputStream tok = new FileInputStream(ime);
DataInputStream podatki = new DataInputStream(tok);
```

Oba stavka lahko združimo v naslednji zapis:

```
DataInputStream podatki =
    new DataInputStream(new FileInputStream(ime));
```

In kako najlažje poiščemo ustrezen razred za ovijanje? Poiščemo pač tisti razred, ki nam nudi ustrezne metode za branje želenih podatkov. V našem primeru je to metoda `readDouble()`, ki jo najdemo v razredu `DataInputStream`:

```
double stevilo = podatki.readDouble();
```

Seveda lahko tokove po potrebi poljubno ovijamo. Tako bi lahko pri branju iz datoteke uporabili še medpomnenje (*buffering*), kar omogoča razred `BufferedInputStream`:

```
DataInputStream podatki = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream(ime)));
```

Vedno pa velja, da je zadnji tok v verigi ovijanja tisti, katerega metode nas zanimajo.

Pa si poglejmo ovijanje tokov še na enem primeru. Spremenimo naš prejšnji program tako, da bo datoteko prepisoval po vrsticah. Iščemo torej razreda, katerih metode omogočajo branje cele vrstice naenkrat in tudi zapis tako prebrane vrstice. Datotečni tok, ki bere znake, lahko ovijemo z razredom `BufferedReader`, ki omogoča branje preko medpomnilnika. Metodo `readLine()` za branje cele vrstice znakov najdemo med metodami tega razreda. Bodite pozorni, da metoda vrne vsebino vrstice kot niz znakov (brez znaka za novo vrstico) oziroma vrednost `null`, ko doseže konec toka.

```
FileReader tok = new FileReader(ime);
BufferedReader podatki = new BufferedReader(tok);
String niz = podatki.readLine();
```

Podoben razred poiščemo tudi za pisanje:

```
FileWriter tok = new FileWriter(ime);
BufferedWriter podatki = new BufferedWriter(tok);
```

Ta razred nudi tudi metodi `write()` za zapis niza znakov in `newLine()` za zapis znaka za novo vrstico.

```
podatki.writeLine(niz);
podatki.newLine();
```

Koda tega primera je v datoteki `PrepisiVrstice.java`.

```
import java.io.*;

public class PrepisiVrstice
{
  public static void main(String[] args) throws IOException
  {
    if(args.length < 2) {
      System.out.println("Uporaba: java Prepisi izvorDat ponorDat");
      System.exit(1);
    }
    BufferedReader vhTok = new BufferedReader(new FileReader(args[0]));
    BufferedWriter izhTok = new BufferedWriter(new FileWriter(args[1]));
    String vrstica;
    while ((vrstica = vhTok.readLine()) != null) {
      izhTok.write(vrstica);
      izhTok.newLine();
    }
    vhTok.close();
    izhTok.close();
  }
}
```

Program `PrepisiVrstice.java`

Datoteke z naključnim dostopom

Pri naključnem dostopu do datotek lahko v datoteko pišemo ali pa iz nje beremo vsebino. Naključni dostop v Javi omogoča razred `RandomAccessFile`, ki implementira vmesnika `DataInput` (vmesnik omogoča branje bajtov iz binarnega toka in njihovo sestavljanje v katerikoli javanski primitivni tip) in `DataOutput` (pretvori javanski primitivni tip v zaporedje bajtov in jih zapiše v binarni tok).

Način dostopa do datoteke podamo kot argument konstruktorja. Tako lahko datoteko `dat.dat` odpremo za branje (`r`) ali pa za branje in pisanje (`rw`) z naslednjima klicema konstruktorja:

```
RandomAccessFile b = new RandomAccessFile("dat.dat", "r");
RandomAccessFile bp = new RandomAccessFile("dat.dat", "rw");
```

Če odpiramo datoteko za branje in pisanje in če ta datoteka še ne obstaja, se ustvari nova datoteka.

V tako odprti datoteki se lahko s pomočjo metode `seek()` postavimo na poljuben položaj v datoteki, trenutni položaj datotečnega kazalca pa vrne metoda `getFilePointer()`. Z metodo `length()`, ki vrne dolžino datoteke, lahko preverimo, ali je v datoteki že kaj zapisano.

Branje podatkov lahko poteka preko metode `read()`, ki prebere en bajt, ali pa uporabimo metode, ki sestavijo prebrane bajte v podatke primitivnih tipov, kot so `readDouble()`, `readInt()`, `readChar()`, ali v celo vrstico (metoda `readLine()`).

Podobno lahko za pisanje podatkov uporabimo metodo `write()`, ki zapiše en bajt, ali pa višjenivojske metode, kot so `writeDouble()`, `writeInt()`, `writeChar()` ali `writeChars()`.

Podrobnejši opis metod najdete v JDK dokumentaciji.

Za primer dela z datoteko z naključnim dostopom si poglejmo naslednji program. Program odpre datoteko, ki je podana kot argument, za hkratno branje in pisanje (če ime ni podano, vzame privzeto ime `datoteka.dat`). Najprej preveri dolžino datoteke in izpiše opozorilo o prepisu datoteke, če datoteka ni prazna. Potem v datoteko zapiše naključno število celih števil od 0 naprej, nato pa naključno izbere neko pozicijo v datoteki in se postavi nanjo. Prebere število, ki je na izbrani poziciji, in ga izpiše skupaj s trenutno pozicijo datotečnega kazalca (pred branjem in po branju). Na koncu se postavi spet na začetek datoteke in po vrsti bere in izpisuje na zaslon vsa števila iz datoteke.

Za branje števil iz datoteke uporabljamo metodo `readInt()`, ki prebere predznačeno 32-bitno celo število. Metoda torej prebere naslednje štiri bajte iz datoteke in jih tolmači

kot tip `int`. V primeru, ko je datoteke konec, preden uspe prebrati vse štiri bajte, metoda sproži izjemo `EOFException`, ki je vrsta `IOException`. Zato moramo branje iz datoteke postaviti v `try`/`catch` blok, v katerem potem ujamemo izjemo `EOFException`.

V datoteki zapisana cela števila so zapisana po bajtih, kjer vsako število zavzame štiri bajte. To moramo upoštevati pri nastavljanju datotečnega kazalca, saj se le-ta za vsako naslednje število poveča za vrednost štiri. Prvo število je zapisano na lokaciji `0`. Tako v primeru, da želimo prebrati `n`-to število po vrsti, nastavimo vrednost datotečnega kazalca na `(n-1)*4`:

```
datoteka.seek((n-1)*4);
```

Bodite pozorni na delovanje programa pri datotekah, ki že imajo zapisano neko vsebino. Pri pisanju števil v datoteko se namreč datoteka (delno) prepiše. Pri tem lahko v njej poleg novo vpisanih števil ostanejo tudi stari podatki. Vsi ti podatki se pri izpisu datoteke interpretirajo kot cela števila, zato lahko v tem primeru dobimo zelo nenavaden in nepričakovan izpis vsebine datoteke.

V celoti je program zapisan v datoteki `NakljucenDostop.java`.

```
import java.io.*;

public class NakljucenDostop
{
  public static void main(String[] args) throws IOException
  {
    String ime = "datoteka.dat";
    if(args.length > 0)
      ime = args[0];
    RandomAccessFile datoteka = new RandomAccessFile(ime,"rw");
    long dolzina = datoteka.length();
    if (dolzina > 0)
      System.out.println("Datoteka " + ime + " bo prepisana.");
    int max = (int) (Math.random()*100) + 1;
    for(int i=0; i<=max; i++)
      datoteka.writeInt(i);
    System.out.println("V datoteko smo vpisali števila od 0 do " + max);
    int lokacija = (int) (Math.random()*max) + 1;
    datoteka.seek(lokacija*4);
    System.out.println("Datotečni kazalec je na lokaciji " +
datoteka.getFilePointer());
    int stevilo = datoteka.readInt();
    System.out.println("Preberemo število " + stevilo);
    System.out.println("Datotečni kazalec je na lokaciji " +
datoteka.getFilePointer());
    datoteka.seek(0);
    try {
      while( true ) {
        stevilo = datoteka.readInt();
        System.out.print(stevilo + "   ");
      }
    }
    catch(EOFException e) {
      // obravnavamo izjemo "konec datoteke"
      // in s tem prekinemo while zanko
```

```
    }
    finally {
      datoteka.close();
    }
  }
}
```

Program NakljucenDostop.java

Pisanje in branje objektov

Java omogoča tudi zapisovanje in branje objektov (v prejšnjih razdelkih smo brali/pisali le primitivne podatkovne tipe). Temu sta namenjena tokova `ObjectOutputStream` in `ObjectInputStream`, ki vsak objekt spremenita v zaporedje bajtov, tako da ga lahko zapišemo v poljuben tok in ga kasneje znamo tudi prebrati. Ta postopek pretvorbe objekta v bajte imenujemo serializacija objektov (*object serialization*).

Nov tok za zapisovanje objektov ustvarimo s klicem konstruktorja, kateremu podamo izhodni tok, v katerega želimo zapisovati:

```
OutputStream tok = new FileOutputStream(ime);
ObjectOutputStream objTok = new ObjectOutputStream(tok);
```

Metoda `writeObject()` poskrbi za zapis objekta, ki ji ga podamo kot argument:

```
objTok.writeObject(new Date());
```

Ker metoda `writeObject()` objekt najprej pretvori v zaporedje bajtov, morajo biti ti objekti take vrste, da omogočajo serializacijo (vsi objekti niso taki). To pomeni, da morajo ti objekti izdelati vmesnik `Serializable` (vmesnik nima nobenih atributov ali metod ter služi le za označitev primernosti za serializacijo). Če temu ni tako, metoda sproži izjemo `NotSerializableException`. Izjema `InvalidClassException` pa se sproži v primeru napake razreda. Obe izjemi sta potomca bolj splošne izjeme `IOException`.

Na koncu zapisovanja, preden tok zapremo, moramo poklicati še metodo `flush()`, ki poskrbi za splakovanje toka (vsi bajti iz medpomnilnika se zapišejo v tok). Pisanje torej zaključimo z:

```
objTok.flush();
objTok.close();
```

Za branje in obnovo objekta iz toka poskrbi metoda `readObject()` razreda `ObjectInputStream`. Seveda moramo najprej tok za branje objektov odpreti s klicem konstruktorja, ki mu podamo vhodni tok, iz katerega želimo brati:

```
InputStream tok = new FileInputStream(ime);
ObjectInputStream objTok = new ObjectInputStream(tok);

Date d = (Date) objTok.readObject();
```

Ker metoda `readObject()` bere poljubne objekte, vrača tip `Object`. Zato moramo prebranemu objektu nastaviti ustrezen tip s prirejanjem.

Za demonstracijo zapisovanja in branja objektov v oziroma iz datoteke napišimo lasten razred, katerega izvode bomo zapisali v datoteko in jih nato tudi prebrali iz datoteke.

Naj naš nov razred opisuje naše prijatelje. Razred `Prijatelj` je povsem preprost, vsebuje nekaj atributov, dva konstruktorja in metodo za pretvorbo v niz (ta zajema vrednosti vseh atributov):

```
public class Prijatelj
{
  private String priimek;
  private String ime;
  private int starost;
  private String telefon;

  public Prijatelj()
  {
    this("", "", 0, "");
  }

  public Prijatelj(String i, String p, int s, String t)
  {
    this.priimek = p;
    this.ime = i;
    this.starost = s;
    this.telefon = t;
  }

  public String toString()
  {
    return "Prijatelj: " + this.ime + " " + this.priimek +
           ", " + this.starost + " let, tel.: " + this.telefon;
  }
}
```

Razredu lahko dodamo še metodi za branje in zapis objekta. Za zapis objekta v datoteko bomo napisali metodo `pisiDat`, ki prejme želeni izhodni tok kot argument:

```
public void pisiDat(FileOutputStream tok) throws IOException
{
  ObjectOutputStream objIzhTok = new ObjectOutputStream(tok);
  objIzhTok.writeObject(this);
  objIzhTok.flush();
}
```

Ker pri zapisovanju lahko pride do proženja izjem, mora metoda razglasiti izjemo. V telesu metode najprej odpremo tok za zapis objekta, poskrbimo za zapis tega objekta (določilo `this` se nanaša na objekt, ki ga zapisujemo) ter splaknemo tok.

Zapisani objekt lahko preberemo z metodo `beriDat`, ki prejme kot argument vhodni tok. Tudi tu najprej ustvarimo tok za branje objektov in iz njega preberemo en objekt:

```
public void beriDat(FileInputStream tok)
    throws IOException, ClassNotFoundException
{
  ObjectInputStream objVhTok = new ObjectInputStream(tok);
  Prijatelj stari = (Prijatelj)objVhTok.readObject();
  this.priimek = stari.priimek;
  this.ime = stari.ime;
  this.starost = stari.starost;
  this.telefon = stari.telefon;
}
```

Tudi v metodi `beriDat()` vse morebitne izjeme le posredujemo naprej.

Prebranemu objektu priredimo tip `Prijatelj` ter nastavimo vrednosti atributov objekta (nanj se nanaša določilo `this`) na vrednosti atributov prebranega objekta.

Da lahko objekte pišemo oziroma beremo, jih moramo serializirati, zato mora naš razred implementirati vmesnik `Serializable`:

```
public class Prijatelj implements Serializable
```

In kako lahko uporabljamo obe metodi razreda? Recimo, da imamo podatke o svojih prijateljih shranjene v polju prijatelji:

```
Prijatelj[] prijatelji = new Prijatelj[MAX];
```

Potem lahko vse podatke o prijateljih zapišemo v datoteko v zanki, ki gre preko vseh elementov polja in vsakega zapiše v datoteko. Seveda moramo najprej odpreti ustrezno datoteko za zapis podatkov in jo na koncu tudi zapreti. Vse stavke postavimo tudi v `try`/`catch` blok, da prestrežemo sprožene izjeme:

```
try {
  FileOutputStream izhTok = new FileOutputStream(imeDat);
  for(int i=0; i<MAX; i++)
    prijatelji[i].pisiDat(izhTok);
  izhTok.close();
}
catch(IOException e) {
  System.out.println("Napaka: " + e.getMessage());
}
```

Pri branju postopamo podobno, le da tokrat odpremo datoteko za branje podatkov:

```
try {
  FileInputStream vhTok = new FileInputStream(imeDat);
  i = 0;
  while(i<MAX) {
    prijatelji[i] = new Prijatelj();
    prijatelji[i++].beriDat(vhTok);
  }
  vhTok.close();
}
catch(IOException e) {
  System.out.println("Napaka: " + e.getMessage());
}
```

Poglejmo si še vse skupaj na primeru, v katerem ustvarimo polje petih objektov razreda `Prijatelj` in v datoteko zapišemo vseh pet objektov. Nato iz datoteke preberemo vse zapisane objekte in jih izpišemo na zaslon.

Datoteka `Objekti.java` z izvorno kodo programa je naslednja:

```
import java.io.*;

public class Objekti
{
  static final int MAX = 5;

  public static void main(String[] args)
  {
    String imeDatoteke;
    if(args.length < 1)
      imeDatoteke = "Objekti.dat";
    else
      imeDatoteke = args[0];
    zapisiPodatke(imeDatoteke);
    preberiPodatke(imeDatoteke);
  }

  private static void zapisiPodatke(String imeDat)
  {
    Prijatelj[] prijatelji = new Prijatelj[MAX];
    prijatelji[0] = new Prijatelj("Janez", "Novak", 25, "123456");
    prijatelji[1] = new Prijatelj("Micka", "Kovačeva", 21, "123456");
    prijatelji[2] = new Prijatelj("Franci", "Nabalanci", 25, "121221");
    prijatelji[3] = new Prijatelj("Nana", "Študent", 19, "654321");
    prijatelji[4] = new Prijatelj("Miki", "Žmauc", 35, "566556");
    try {
      FileOutputStream izhTok = new FileOutputStream(imeDat);
      for(int i=0; i<MAX; i++)
        prijatelji[i].pisiDat(izhTok);
      izhTok.close();
    }
    catch(IOException e) {
      System.out.println("Napaka: " + e.getMessage());
    }
  }

  private static void preberiPodatke(String imeDat)
  {
```

```
    try {
      FileInputStream vhTok = new FileInputStream(imeDat);
      System.out.println("Prebrali smo naslednje prijatelje: ");
      try {
        while(true) {
          Prijatelj p = new Prijatelj();
          p.beriDat(vhTok);
          System.out.println(p.toString());
        }
      }
      catch(IOException e) {
      }
      vhTok.close();
    }
    catch(FileNotFoundException e) {
      System.out.println("Napaka: ni datoteke " + imeDat);
    }
    catch(IOException e) {
      System.out.println("Napaka: " + e.getMessage());
    }
    catch(ClassNotFoundException e) {
      System.out.println("Napaka: ni razreda " + e.getMessage());
    }
  }
}

class Prijatelj implements Serializable
{
  private String priimek;
  private String ime;
  private int starost;
  private String telefon;

  public Prijatelj()
  {
    this("", "", 0, "");
  }

  public Prijatelj(String i, String p, int s, String t)
  {
    this.priimek = p;
    this.ime = i;
    this.starost = s;
    this.telefon = t;
  }

  public void pisiDat(FileOutputStream izhodniTok)
      throws IOException
  {
    ObjectOutputStream objIzhTok = new ObjectOutputStream(izhodniTok);
    objIzhTok.writeObject(this);
    objIzhTok.flush();
  }

  public void beriDat(FileInputStream vhodniTok)
      throws IOException, ClassNotFoundException
  {
    ObjectInputStream objVhTok = new ObjectInputStream(vhodniTok);
    Prijatelj stari = (Prijatelj)objVhTok.readObject();
    this.priimek = stari.priimek;
    this.ime = stari.ime;
    this.starost = stari.starost;
    this.telefon = stari.telefon;
  }
```

```
  public String toString()
  {
    return "Prijatelj: " + this.ime + " " + this.priimek + ", " + this.starost +
" let, tel.: " + this.telefon;
  }
}
```

Program Objekti.java

6. Delo z mrežo

Javanski razredi iz paketa `java.net` ponujajo visokonivojski vmesnik za dostop do TCP/IP omrežij. Z uporabo že pripravljenih razredov lahko enostavno napišemo program za delo z vtičnicami (*sockets*), izdelamo porazdeljen program tipa odjemalec/strežnik (*client/server*), delamo z URL (*Universal Resource Locator*) naslovi, vzpostavimo omrežno povezavo različnih protokolov ali obravnavamo prejeto vsebino. Omogočajo nam tako priklapljanje na obstoječe omrežne storitve kot vzpostavitev lastnega strežnika.

Različne omrežne vire lahko dosegamo preko različnih protokolov. V svetovnem spletu ponavadi uporabljamo `http` (*HyperText Transfer Protocol*) protokol, ki uporablja URL naslove za lociranje podatkov. Vzpostavljene povezave z omrežnimi viri so podobne tokovom, ki smo jih obravnavali v poglavju *Datoteke in tokovi*.

Razred `URL`

Vire v internetu v Javi opisujejo izvodi razreda `URL`. Objekte `URL` ustvarimo s klicem konstruktorjev, katerim lahko podamo različne argumente. Primera:

```
URL vir1 = new URL("http://lgm.fri.uni-lj.si/op2/index.html");
URL vir2 = new URL("http", "lgm.fri.uni-lj.si/op2/", "index.html");
```

Prvi konstruktor prejme en sam argument, to je niz znakov, ki opisuje cel URL naslov. Drugi konstruktor pa prejme tri ločene nize, ki skupaj sestavljajo URL naslov, to so protokol, gostitelj (*host*) in ime datoteke.

Če je URL napačno sestavljen, konstruktor sproži izjemo `MalformedURLException`, ki jo moramo ujeti ali pa posredovati naprej.

Ko objekt `URL` enkrat ustvarimo, ga ne moremo več spreminjati. Seveda pa ga vedno lahko zavržemo in ustvarimo novega.

Posamezne elemente URL naslova lahko dobimo preko različnih metod tega razreda. Navedimo le nekatere od njih (vse se nanašajo na URL objekt):

- `getFile()` vrne ime datoteke,
- `getHost()` vrne ime gostitelja,
- `getPath()` vrne tisti del URL naslova, ki določa pot,
- `getPort()` vrne številko vrat (*port number*),
- `getProtocol()` vrne ime protokola,
- `getRef()` vrne sidro (*anchor* ali *reference*).

Seveda vse metode niso primerne za vse vrste URL naslovov (na primer, URL naslov morda sploh ne vsebuje sidra).

Ko imamo vzpostavljeno povezavo z virom (ko smo uspešno ustvarili objekt `URL`), lahko s klicem metode `openStream()` odpremo vhodni tok tipa `InputStream`, ki ga uporabimo za branje vira. Potem postopamo podobno kot pri branju datotek.

Branje datoteke, podane z URL

Za primer si poglejmo program, ki na zaslon izpiše vsebino datoteke, ki je podana z URL naslovom. Datoteko bomo brali (in izpisovali) po vrsticah, zato bomo okoli vhodnega toka `InputStream` ovili najprej `InputStreamReader`, ki omogoča branje po znakih, okoli njega pa še `BufferedReader`, s katerim lahko naenkrat preberemo celo vrstico:

```
URL vir = new URL(naslov);
InputStream tok = vir.openStream();
InputStreamReader vhod = new InputStreamReader(tok);
BufferedReader branje = new BufferedReader(vhod);
```

Za branje vhoda uporabimo kar metodo `readLine()`, ki naenkrat prebere celo vrstico:

```
String vrstica;
while ((vrstica = branje.readLine()) != null)
  System.out.println(vrstica);
```

V `while` zanki vsako prebrano vrstico sproti izpišemo. Na koncu vzpostavljen tok tudi zapremo:

```
branje.close();
```

Vzpostavljanje povezave z virom in vhodnega toka ter branje iz vira postavimo v `try`/`catch` blok, saj pri tem lahko pride tudi do izjeme, ki jo ujamemo in obravnavamo.

Na začetku programa moramo tudi napovedati uporabo obeh paketov (za delo s tokovi in z mrežo):

```
import java.io.*;
import java.net.*;
```

Datoteka `PreberiURL.java` z izvorno kodo programa:

```
import java.io.*;
import java.net.*;

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

```
    if(args.length < 1) {
      System.out.println("Uporaba: java PreberiURL URL");
      System.exit(1);
    }
    try {
      URL vir = new URL(args[0]);
      BufferedReader tok =
          new BufferedReader(new InputStreamReader(vir.openStream()));
      String vrstica;
      while ((vrstica = tok.readLine()) != null)
        System.out.println(vrstica);
      tok.close();
    }
    catch(IOException e) {
      System.out.println("Napaka: " + e.getMessage());
      System.exit(1);
    }
  }
}
```

Program PreberiURL.java

Pri zagonu programa moramo kot argument podati veljaven URL naslov in na zaslon se izpiše vsebina te lokacije. Primer klica programa:

```
java PreberiURL http://lgm.fri.uni-lj.si/op2/index.html
```

Namesto da vsebino na podanem URL naslovu izpisujemo na zaslon, bi jo lahko enostavno zapisali kar v datoteko na lokalnem disku. Naslednji primer prikazuje program, ki na lokalni disk v tekoči direktorij shrani poljubno datoteko (na primer tekstovno datoteko ali sliko). Datoteka je podana z URL naslovom.

Izhajamo iz prejšnjega primera, kjer smo vzpostavili povezavo z virom in prebrali njegovo vsebino. Ker želimo brati poljubno datoteko (tekstovno ali binarno), bomo okoli vhodnega toka ovili le razred `BufferedInputStream`, ki omogoča branje preko medpomnilnika:

```
URL vir = new URL(naslov);
BufferedInputStream vhod =
    new BufferedInputStream(vir.openStream());
```

Seveda moramo ustvariti še en izhodni tok, ki bo prebrane podatke zapisal v datoteko na disku. `BufferedOutputStream` je primeren ovoj okoli datotečnega izhodnega toka:

```
BufferedOutputStream izhod =
  new BufferedOutputStream(new FileOutputStream(ime));
```

Spremenljivka `ime` določa ime datoteke na disku. Najbolj smiselno je, da se to ime ujema z imenom datoteke spletnega vira. Slednje nam vrne metoda `getFile()`:

```
String ime = izvor.getFile();
```

Ker pa to ime lahko vključuje tudi znak / kot ločilo v poti, bomo za ime datoteke upoštevali le tiste znake, ki sledijo zadnji pojavitvi znaka / v nizu (če se znak '/' v nizu ne pojavi, je rezultat spodnjega stavka kar nespremenjen niz):

```
ime = ime.substring(ime.lastIndexOf('/')+1);
```

Ker gre tokrat za branje in pisanje binarnih datotek, uporabimo metodi `read()` in `write()` za branje oziroma zapis bajtov:

```
int bajt;
while ((bajt = vhod.read()) != -1)
  izhod.write(bajt);
```

Na koncu oba toka tudi zapremo:

```
vhod.close();
izhod.close();
```

Datoteka `Shrani.java` z izvorno kodo programa:

```
import java.io.*;
import java.net.*;

public class Shrani
{
  public static void main(String[] args) throws IOException
  {
    if(args.length < 1) {
      System.out.println("Uporaba: java Shrani URL");
      System.exit(1);
    }
    BufferedInputStream vhod = null;
    BufferedOutputStream izhod = null;
    try {
      URL izvor = new URL(args[0]);
      String ime = izvor.getFile();
      ime = ime.substring(ime.lastIndexOf('/')+1);
      vhod = new BufferedInputStream(izvor.openStream());
      izhod = new BufferedOutputStream(new FileOutputStream(ime));
      int bajt;
      while ((bajt = vhod.read()) != -1)
        izhod.write(bajt);
    }
    catch(IOException e) {
      System.out.println("Napaka: " + e.getMessage());
      System.exit(1);
    }
    finally {
      vhod.close();
      izhod.close();
    }
  }
}
```

Program Shrani.java

Prikaz slike, podane z URL

Za konec pa si oglejmo še primer, kako v oknu prikažemo sliko, ki je podana z URL naslovom. Pri izdelavi programa se naslonimo na primer `Slika.java` iz razdelka *Bitne slike* v poglavju *Grafika*. Primer je zelo podoben, le da smo tam datoteko s sliko prebrali z lokalnega diska, ne iz spletnega vira.

Za nalaganje slike tudi tokrat poskrbimo že v konstruktorju razreda (razred smo poimenovali `Prikazi`). Za branje bitne slike uporabimo metodo `getImage()`, ki pa ji kot argument podamo objekt `URL`, ki predstavlja URL naslov slike za prikaz. Zato najprej ustvarimo `URL` objekt, preko katerega se povežemo na spletni vir:

```
URL url = new URL(naslov);
Image slika = Toolkit.getDefaultToolkit().getImage(url);
```

Metoda vrne objekt `Image`, ki je referenca na bitno sliko v pomnilniku. Pri ustvarjanju objekta `URL` lahko pride do izjeme, če niz `naslov` ne predstavlja poznanega protokola. Zato ta stavek postavimo v `try` blok, izjemo pa prestrežemo in obravnavamo v `catch` bloku.

Vse povedano sestavimo v kodo, ki je zapisana v datoteki `Prikazi.java`.

```
import java.awt.*;
import javax.swing.*;
import java.net.*;

class Prikazi extends JPanel
{
  public final int SIRINA = 800;
  public final int VISINA = 600;
  public final int ROB = 10;
  public Image slika = null;
  private URL url = null;

  public static void main(String[] args)
  {
    if(args.length < 1) {
      System.out.println("Uporaba: java Prikazi URL_slike");
      System.exit(1);
    }
    JFrame okno = new JFrame("Prikaz bitne slike " + args[0]);
    Prikazi s = new Prikazi(args[0]);
    okno.getContentPane().add(s);
    okno.setSize(s.SIRINA, s.VISINA);
    okno.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    okno.setVisible(true);
  }

  public Prikazi(String naslov)
  {
    try {
      url = new URL(naslov);
      slika = Toolkit.getDefaultToolkit().getImage(url);
    }
    catch(MalformedURLException e){
      System.out.println("Napaka: " + e.getMessage());
```

```
    }
  }

  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    if (slika != null)
      g.drawImage(slika, ROB, ROB, this);
  }
}
```

Program Prikazi.java

Ob klicu programa `Prikazi` moramo podati tudi URL datoteke z bitno sliko. Primer klica programa je:

```
java Prikazi http://lgm.fri.uni-lj.si/op2/vaje/slika.jpg
```

Če podana slika obstaja in če jo program uspe prebrati, jo prikaže v oknu na zaslonu.

7. Niti

Java ima vključeno tudi podporo za niti. Nit (*thread*) je tok izvajanja programskih stavkov, ki se odvija znotraj programa hkrati z ostalimi nitmi. Niti so sicer podobne procesom, a se od njih razlikujejo predvsem po načinu delitve virov (kot je na primer pomnilnik). Procesi so ponavadi medsebojno neodvisni in imajo ločene naslovne prostore, medtem ko niti delijo isti pomnilnik in druge vire.

Večnitnost je model, ki omogoča več nitim, da sobivajo znotraj enega procesa, si delijo njegove vire, a se (lahko) izvajajo neodvisno. Ta mehanizem omogoča hitro izmenjavo posameznih opravil na procesorju, kar daje vtis hkratnega izvajanja opravil.

Vsaka nit ima svojo prioriteto, ki določa, kolikšen delež celotnega procesorskega časa bo dodeljen posamezni niti. Javanski izvajalni sistem vedno uporablja več niti: najmanj eno nit za programski proces in eno nit za smetarja (*garbage collector*), ki je nit z najnižjo prioriteto.

Za delo z nitmi v Javi uporabljamo razred `Thread` oziroma vmesnik `Runnable`. Oba sta v paketu `java.lang`, torej sta na voljo vsem razredom brez posebne napovedi.

Razred `Thread`

Eden od načinov izdelave večnitnega programa je, da izdelamo razred, ki je izpeljan iz razreda `Thread`. Razred lahko zapišemo takole:

```
public class Nit extends Thread
{
  Nit()
  {
    // konstruktor
  }

  public void run()
  {
    // telo niti
  }
}
```

Metoda `run` je tista, ki določa delovanje niti. Podedujemo jo iz razreda `Thread`, a ker želimo sami določiti obnašanje niti, to metodo ponavadi preobložimo. Novo nit (nov izvod našega razreda `Nit`) ustvarimo s klicem konstruktorja in jo poženemo s klicem metode `start`. S tem nit preide v delujoče stanje; izvajati se začne njena metoda `run`:

```
Nit nitka = new Nit();
nitka.start();
```

Pa si poglejmo izdelavo niti natančneje na primeru. Kot primer vzemimo program, v katerem izdelamo pet izvodov našega razreda (pet niti) in jih poženemo hkrati. Vsak izvod razreda naj v zanki izpisuje svoje ime, ki mu ga podamo kot argument konstruktorju. Tako bomo ob izvajanju programa lahko opazovali, kako se izvajajo posamezne niti.

Kot smo že rekli, naš razred izpeljemo iz razreda `Thread`, ki omogoča večnitnost:

```
public class Niti extends Thread
```

V razredu napišemo tudi konstruktor, ki sicer le pokliče konstruktor razreda `Thread` in hkrati določi tudi ime niti. Vsaka nit ima namreč svoje ime, ki ga vrne metoda `getName()`. Če imena niti ne podamo eksplicitno, dobi nit privzeto ime "Thread-*x*", kjer je *x* zaporedna številka ustvarjene niti.

```
public class Niti extends Thread
{
  Niti(String ime)
  {
    super(ime);
  }

  public void run()
  {
    // tu zapišemo, kaj naj naša nit dela
    ...
  }
}
```

Eno nit lahko potem ustvarimo s klicem konstruktorja, ki mu kot argument podamo ime te niti:

```
Niti nit = new Niti("Nova Nitka");
```

Nit nato poženemo s klicem metode `start`, ki prične izvajanje te niti (metodo `start` smo podedovali iz razreda `Thread`):

```
nit.start();
```

Metoda poskrbi za vzpostavitev nove niti in pokliče metodo `run` te niti. Ko se metoda `run` konča, nit umre. Ko umrejo vse niti, se zaključi tudi izvajanje programa.

Metoda `run`, ki jo vsebuje razred `Thread`, določa izvajanje niti. V izpeljanem razredu preobložimo to metodo in s tem določimo, kaj naj nit dela. Metodo `run` bi lahko zapisali takole:

```
public void run()
{
  for(int i=0; i<20; i++) {
    System.out.print(getName());
  }
  System.out.println("Konec niti: " + getName());
}
```

V zanki, ki se dvajsetkrat ponovi, izpisujemo ime niti ter na koncu izpišemo tudi, da se je nit končala.

Pet izvodov tega razreda naredimo kar v `main` metodi, ki jo lahko vključimo v razred `Niti`. V njej ustvarimo tabelo petih niti (ime vsake niti je kar zaporedna številka te niti) ter vse ustvarjene niti tudi poženemo.

```
public static void main(String[] args)
{
  final int MAX = 5;
  Niti[] niti = new Niti[MAX];
  for(int i=0; i<MAX; i++)
    niti[i] = new Niti(" " + i + " ");
  System.out.println("Zagon vseh niti ...");
  for(int i=0; i<MAX; i++)
    niti[i].start();
}
```

Da se posamezne niti ne izvajajo prehitro, bomo po vsakem izpisu imena niti v metodi `run` dodali še začasno zaustavitev niti. S klicem metode `sleep(n)` lahko nit spravimo k počitku za `n` milisekund (začasno zaustavimo izvajanje te niti). Dolžina zaustavitve, ki jo določa atribut `pocakaj`, naj bo naključna (med 0 in 5 sekundami) in jo nastavimo že v konstruktorju.

Ob klicu metode `sleep` moramo ujeti izjemo `InterruptedException`, ki jo metoda sproži v primeru, ko jo prekine druga nit. Ker je tak dogodek pričakovan, bomo izjemo le prestregli, ukrepali pa ne bomo.

Izvorna koda celega programa je zajeta v datoteki `Niti.java`.

```
public class Niti extends Thread
{
  private int pocakaj;

  public Niti(String ime)
  {
    super(ime);
    pocakaj = (int) (Math.random()*5000);
    System.out.println("Ime niti: " + getName() + ", čas počivanja: " + pocakaj);
  }

  public void run()
  {
```

```
    for(int i=0; i<20; i++) {
      System.out.print(getName());
      try {
        sleep(pocakaj);
      }
      catch (InterruptedException e) {
      }
    }
    System.out.println("Konec niti: " + getName());
  }

  public static void main(String[] args)
  {
    final int MAX = 5;

    Niti[] niti = new Niti[MAX];
    for(int i=0; i<MAX; i++)
      niti[i] = new Niti(" " + i + " ");
    System.out.println("Zagon vseh niti ...");
    for(int i=0; i<MAX; i++)
      niti[i].start();
  }
}
```

Program Niti.java

Ob izvajanju programa vidimo, da je rezultat odvisen tako od posameznih časov počivanja niti kot od zaporedja izvajanja niti (slednje najlažje preverimo, če nastavimo čas počivanja vseh niti na isto vrednost). Ker pa se niti izvajajo neodvisno, lahko dobimo ob ponovnem zagonu programa povsem drugačno zaporedje izpisov.

Vmesnik `Runnable`

Drug način izdelave niti pa je, da razred implementira vmesnik `Runnable`. Vmesnik ponuja le metodo `run`, ki jo izdelamo kot telo niti, to metodo pa mora implementirati tudi naš razred, ki ta vmesnik izdela. Ta način je uporaben predvsem v primeru, ko je naš razred že izpeljan iz nekega drugega razreda in ga zato ne moremo izpeljati še iz razreda `Thread` (Java ne pozna večkratnega dedovanja).

```
public class Nit implements Runnable
{

  public void run()
  {
    // tu zapišemo, kaj naj nit dela
    ...
  }

}
```

Za delovanje same niti pa v našem razredu še vedno potrebujemo objekt razreda `Thread`. Ponavadi je to kar privatni atribut razreda, lahko pa ga podamo kot spremenljivko v metodi `main`:

```
public static void main (String[] args)
{
  Thread nitka = new Thread(new Nit());
  nitka.start();
}
```

Ob ustvarjanju nove niti smo konstruktorju razreda `Thread` podali en argument, to je ime objekta, katerega `run` metodo naj ta nit uporabi za svoje telo. Seveda moramo zatem nit tudi zagnati.

Če objekt razreda `Thread` v razred vključimo kot privatni atribut, ga lahko deklariramo na naslednji način:

```
private Thread nit;
```

Razredu potem dodamo tudi metodo `pozeni`, ki poskrbi za ustvarjanje in zagon niti, če ta še ne obstaja:

```
public void pozeni()
{
  if(nit == null) {
    nit = new Thread(this, ime);
    nit.start();
  }
}
```

Če nit še ne obstaja (spremenljivka `nit` je enaka `null`), ustvarimo novo nit in ji določimo, čigavo metodo `run` naj uporabi kot svoje telo (`this`, torej tega razreda) in podamo ime niti (`ime`). Potem ustvarjeno nit tudi zaženemo.

Naš prejšnji program bi torej na drugačen način lahko zapisali tudi takole (program `Niti1.java`):

```
public class Niti1 implements Runnable
{
  private Thread nit;
  private String ime;
  private int pocakaj;

  public Niti1(String ime)
  {
    this.ime = ime;
    this.pocakaj = (int) (Math.random()*5000);
    System.out.println("Ime niti: " + ime + ", čas počivanja: " + pocakaj);
  }

  public void pozeni()
  {
    if(nit == null) {
      nit = new Thread(this, ime);
      nit.start();
```

```
    }
  }

  public void run()
  {
    for(int i=0; i<20; i++) {
      System.out.print(nit.getName());
      try {
        nit.sleep(pocakaj);
      }
      catch (InterruptedException e) {
      }
    }
    System.out.println("Konec niti: " + nit.getName());
  }

  public static void main(String[] args)
  {
    final int MAX = 5;

    Niti1[] niti = new Niti1[MAX];
    for(int i=0; i<MAX; i++)
      niti[i] = new Niti1(" " + i + " ");
    System.out.println("Zagon vseh niti ...");
    for(int i=0; i<MAX; i++)
      niti[i].pozeni();
  }
}
```

Program Niti1.java

Delovanje obeh programov je identično.

Stanja niti, prioriteta

Življenje niti sestavljajo štiri stanja, nekatera med njimi se lahko tudi ponavljajo, med stanji pa prehajamo s pomočjo metod. Nit se tako vedno nahaja v enem izmed naslednjih stanj:

- ustvarjena nova nit (*new thread*),
- aktivno stanje (*runnable*),
- zaustavljeno izvajanje (*not runnable*, *blocked*),
- končano izvajanje, nit umre (*dead*).

Prehode med posameznimi stanji omogočajo različne metode:

- `start()`: prične izvajanje niti in jo tako spravi v aktivno stanje (kliče se njena metoda `run`),
- `sleep()`: začasno zaustavi izvajanje niti,
- `wait()`: blokira izvajanje niti, dokler ni izpolnjen določen pogoj,
- `notify()`: nadaljuje izvajanje niti, ko je izpolnjen določen pogoj,
- `yield()`: prepusti procesorski čas drugim nitim.

Vsaka nit ima tudi svojo prioriteto, ki pomeni njeno razvrstitev glede na prednostni dostop do virov operacijskega sistema (kot je na primer procesorski čas). Če niti

prioritete ne podamo posebej, prevzame prioriteto svojega starša. Prioriteta je predstavljena s celim številom med 1 (najmanjša, konstanta `Thread.MIN_PRIORITY`) in 10 (največja, konstanta `Thread.MAX_PRIORITY`), privzeta pa je 5 (normalna prioriteta, konstanta `Thread.NORM_PRIORITY`).

Prioriteto niti lahko izvemo s klicem metode `getPriority()`, nastavimo oziroma spremenimo pa jo z metodo `setPriority()`.

Uporabo prioritete niti si poglejmo še na primeru. Program `Niti.java` spremenimo tako, da bo vsaka od niti začasno zaustavljena za enak, vnaprej določen čas (pol sekunde). Vsaki niti nastavimo tudi prioriteto in sicer na naključno vrednost med 1 in najvišjo prioriteto. Sedaj je izvajanje posameznih niti odvisno od njihovih prioritet. Nit z najvišjo prioriteto se bo končala najhitreje, nit z najnižjo prioriteto pa zadnja.

Program najdete v datoteki `Niti2.java`.

```
public class Niti2 extends Thread
{
  private int pocakaj;

  public Niti2(String ime)
  {
    super(ime);
    pocakaj = 500;
    this.setPriority((int) (Math.random()*Thread.MAX_PRIORITY) + 1);
    System.out.println("Ime niti: " + this.getName() +
                       ", s prioriteto: " + this.getPriority());
  }

  public void run()
  {
    for(int i=0; i<20; i++) {
      System.out.print(getName());
      try {
        sleep(pocakaj);
      }
      catch (InterruptedException e) {
      }
    }
    System.out.println("Konec niti: " + getName());
  }

  public static void main(String[] args)
  {
    final int MAX = 5;

    Niti2[] niti = new Niti2[MAX];
    for(int i=0; i<MAX; i++)
      niti[i] = new Niti2(" " + i + " ");
    System.out.println("Zagon vseh niti ...");
    for(int i=0; i<MAX; i++)
      niti[i].start();
  }
}
```

Program Niti2.java

Uporaba niti

In zakaj sploh potrebujemo večnitno izvajanje programa? Eden glavnih razlogov za večnitnost je ustvarjanje odzivnih uporabniških vmesnikov. Poglejmo si torej tak primer programa, ki demonstrira praktično uporabo niti.

Napišimo program, ki deluje kot števec (preprosta štoparica). Naj bo to grafični program, ki v oknu prikazuje trenutno vrednost števca, zraven pa nam trije gumbi omogočajo upravljanje s števcem (zagon, ustavitev, ponastavitev).

Izdelava grafičnega vmesnika nam ne bi smela delati težav, saj vse prvine poznamo že iz poglavja *Grafika*. V okno postavimo tri gumbe za upravljanje števca in oznako za izpis vrednosti števca, dodamo poslušalce za te gumbe in ustrezno reakcijo na dogodek. Kodo bi za začetek lahko zapisali takole (podrobnosti in razlago te kode si poglejte v poglavju *Grafika*):

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Stevec extends JFrame implements ActionListener
{
  private String naslov = "Števec";
  private Container vsebina = null;
  private JPanel kontrole = new JPanel();
  private JPanel plosca = new JPanel();
  private JButton start = new JButton("Start");
  private JButton stop = new JButton("Stop");
  private JButton reset = new JButton("Reset");
  private JLabel izpis = new JLabel("...");

  public Stevec()
  {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle(naslov);
    setSize(300,100);
    start.addActionListener(this);
    stop.addActionListener(this);
    reset.addActionListener(this);
    kontrole.add(start);
    kontrole.add(stop);
    kontrole.add(reset);
    plosca.add(izpis);
    vsebina = getContentPane();
    vsebina.add(kontrole, BorderLayout.NORTH);
    vsebina.add(plosca, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource() == start) {
      // poženi števec
```

```
    }
    if(e.getSource() == stop) {
      // zaustavi števec
    }
    if(e.getSource() == reset) {
      // postavi števec na 0
    }
  }

  public static void main(String[] args)
  {
    Stevec stevec = new Stevec();
    stevec.setVisible(true);
  }
}
```

Seveda v tej kodi manjka najpomembnejša stvar: naš števec in upravljanje z njim. Za števec uporabimo kar celoštevilsko spremenljivko `st`, ki je atribut razreda, njena začetna vrednost pa je nič. Potrebujemo tudi eno logično spremenljivko `pocakaj`, ki določa, ali števec trenutno teče ali stoji.

```
private int st = 0;
private boolean pocakaj = false;
```

Potem lahko v metodi `actionPerformed` definiramo naslednje odzive na posamezne gumbe. Ob kliku gumba `stop` števec ustavimo tako, da postavimo spremenljivko `pocakaj` na `true`:

```
if(e.getSource() == stop) {
  pocakaj = true;
}
```

Števec ponastavimo tako, da mu določimo vrednost 0 in njegovo vrednost prikažemo v oznaki:

```
if(e.getSource() == reset) {
  st = 0;
  izpis.setText(Integer.toString(st));
}
```

Zagon števca pa vključuje nastavitev spremenljivke `pocakaj` in klic metode, v kateri poteka štetje:

```
if(e.getSource() == start) {
  pocakaj = false;
  delaj();
}
```

Metoda `delaj` bi bila lahko naslednja:

```
public void delaj()
{
  while (true) {
    if(!pocakaj)
      izpis.setText(Integer.toString(st++));
  }
}
```

Naredimo neskončno zanko, v kateri povečujemo vrednost števca in jo izpisujemo v oznaki, če seveda števec ni zaustavljen (vrednost `pocakaj` mora biti `false`).

Ideja sicer zveni dobro, a je neuporabna. Problem nastane v sami metodi `delaj`, kjer smo ustvarili zanko, se nikoli ne izteče (neskončno zanko). Dokler teče ta zanka, je procesor zaseden in ne more obdelovati drugih akcij, kot je na primer obnavljanje okna ali odziv ob kliku na gumb. Dokler se zanka ne konča, se ne konča niti metoda `delaj`, pa tudi metoda `actionPerformed`, kjer smo poklicali metodo `delaj`, se zato ne konča. Rezultat je, da se program ne odziva več na druge dogodke (zamrzne). Po drugi strani pa moramo povečevanje števca postaviti v zanko, ki se izvaja, dokler teče program, saj je to tisto, kar naj bi program delal.

Kako lahko rešimo ta problem, da ohranimo povečevanje števca v neskončni zanki in hkrati zagotovimo odzivnost programa? Tu nam pomagajo niti. Če postavimo povečevanje števca v novo nit, obnavljanje vsebine okna in odzivi na klikanje gumbov pa ostanejo v osnovni niti, se obe niti lahko izvajata istočasno (delita si procesorski čas).

Za števec moramo torej ustvariti novo nit, neskončno zanko pa postaviti za telo te nove niti. Naš razred `Stevec` je že izpeljan iz razreda `JFrame`, zato ne more biti izpeljan tudi iz razreda `Thread`. Razred mora torej izdelati vmesnik `Runnable`:

```
public class Stevec extends JFrame
    implements ActionListener, Runnable
```

Tako moramo v razredu napisati tudi metodo `run`, ki prevzame delo te niti:

```
public void run()
{
  while (true) {
    if(!pocakaj)
      izpis.setText(Integer.toString(st++));
  }
}
```

Za delovanje niti pa potrebujemo še objekt razreda `Thread`, ki naj bo kar privatni atribut razreda:

```
private Thread stejNit = null;
```

Novo nit kreiramo ob prvem kliku na gumb `start`. Če nit še ne obstaja (`stejNit==null`), jo ustvarimo s klicem konstruktorja, ki mu kot argument podamo objekt, katerega metodo `run` bo uporabljala ta nit (v našem primeru je to referenca `this`, saj uporabimo `run` metodo tega razreda):

```
stejNit = new Thread(this);
```

Seveda moramo potem nit tudi zagnati:

```
stejNit.start();
```

Ob kliku na gumb `start` se torej izvede naslednja koda:

```
if(e.getSource() == start) {
  pocakaj = false;
  if(stejNit == null) {
    stejNit = new Thread(this);
    stejNit.start();
  }
}
```

Povečevanje vrednosti števca v metodi `run` je zelo hitro, saj v zanki ne delamo skoraj nič drugega. Štoparica bi delovala bolje, če bi se števec povečeval na primer vsako stotinko ali vsako desetinko sekunde, saj je hitrejše izpisovanje števca nesmiselno. Zato v metodi `run` dodamo znotraj zanke tudi klic metode `sleep(100)`, ki začasno zaustavi nit za 100 milisekund:

```
try {
  stejNit.sleep(100);
}
catch(InterruptedException e) {
}
```

Tako se števec poveča in izpiše le enkrat na desetinko sekunde, pa že tem izpisom le težko sledimo.

Izvorna koda programa, ki smo ga tako sestavili, je v datoteki `Stevec.java`.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Stevec extends JFrame implements ActionListener, Runnable
{
  private String naslov = "Števec";
  private Container vsebina = null;
  private JPanel kontrole = new JPanel();
  private JPanel plosca = new JPanel();
  private JButton start = new JButton("Start");
```

```
  private JButton stop = new JButton("Stop");
  private JButton reset = new JButton("Reset");
  private JLabel izpis = new JLabel("...");
  private int st = 0;
  private boolean pocakaj = false;
  private Thread stejNit = null;

  public Stevec()
  {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle(naslov);
    setSize(300,100);
    start.addActionListener(this);
    stop.addActionListener(this);
    reset.addActionListener(this);
    kontrole.add(start);
    kontrole.add(stop);
    kontrole.add(reset);
    plosca.add(izpis);
    vsebina = getContentPane();
    vsebina.add(kontrole, BorderLayout.NORTH);
    vsebina.add(plosca, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource() == start) {
      pocakaj = false;
      if(stejNit == null) {
        stejNit = new Thread(this);
        stejNit.start();
      }
    }
    if(e.getSource() == stop) {
      pocakaj = true;
    }
    if(e.getSource() == reset) {
      st = 0;
      izpis.setText(Integer.toString(st));
    }
  }

  public void run()
  {
    while (true) {
      try {
        stejNit.sleep(100);
      }
      catch(InterruptedException e) {
      }
      if(!pocakaj)
        izpis.setText(Integer.toString(st++));
    }
  }

  public static void main(String[] args)
  {
    Stevec stevec = new Stevec();
    stevec.setVisible(true);
  }
}
```

Program Stevec.java

Sinhronizacija niti

Pri večnitnih programih lahko nastopijo težave, kadar želi več niti hkrati dostopati do istih virov (na primer spreminjati iste podatke). Te težave, ki izvirajo iz nedoločljivosti izvajanja niti, lahko rešujemo s časovno uskladitvijo ali sinhronizacijo posameznih niti.

V dosedanjih primerih smo uporabljali le neodvisne, neusklajene niti (*asynchronous threads*), kjer je vsaka nit vsebovala lastne kopije podatkov, ki jih je potrebovala pri izvajanju. Pa si poglejmo še primer, ko več niti dostopa do zunanjih (skupnih) podatkov.

Recimo, da imamo trgovino, ki prodaja en sam izdelek (na primer koncertne vstopnice) in seveda vodi evidenco o trenutnem številu izdelkov na zalogi. Razred `Trgovina` naj ima en sam atribut, to je `zaloga`, ki določa število izdelkov, ki so na zalogi v trgovini:

```
private int zaloga = 0;
```

V konstruktorju nastavimo vrednost zaloge na neko naključno vrednost (število vstopnic, ki so v prodaji):

```
public Trgovina()
{
  zaloga = (int) (Math.random()*500) + 10;
}
```

Napišimo še metodo, ki vrača trenutno vrednost zaloge v trgovini:

```
public int zaloga()
{
  return zaloga;
}
```

Nakup `n` izdelkov izvedemo s pomočjo metode `nakup`, v kateri najprej preverimo, ali je na zalogi dovolj izdelkov ter ustrezno zmanjšamo zalogo (opravimo nakup). Pri tem metoda vrne število kupljenih izdelkov oziroma `-1`, če nakupa nismo mogli opraviti zaradi premajhne zaloge.

```
public int nakup(int n)
{
  if(zaloga >= n) {
    zaloga -= n;
    return n;
  }
  return -1;
}
```

Pred nakupom moramo preveriti, ali je na zalogi toliko izdelkov, kot bi jih želeli kupiti. Metoda `prosto` vrne resnično, če je na zalogi najmanj `n` izdelkov:

```
public boolean prosto(int n)
{
  if(zaloga >= n)
    return true;
  else
    return false;
}
```

V trgovini lahko kupuje več kupcev (tudi sočasno), zato za vsakega kupca izdelamo svojo nit. Razred `Kupec` je tako izpeljan iz razreda `Thread`, saj vsak kupec nakupuje neodvisno od ostalih v svoji niti:

```
public class Kupec extends Thread
```

Razred ima en sam atribut, to je referenca na trgovino, v kateri kupuje:

```
private Trgovina trgovina;
```

Vrednost atributa `trgovina` nastavimo v konstruktorju. Dobimo jo preko argumentov konstruktorja skupaj z imenom kupca (imenom niti):

```
public Kupec(String i, Trgovina t)
{
  super(i);
  trgovina = t;
}
```

Srce razreda je metoda `run`, v kateri izvajamo nakupovanje, dokler ima trgovina še kakšen izdelek na zalogi:

```
public void run()
{
  while (trgovina.zaloga() > 0) {
    // nakupuj
  }
  System.out.println(getName() + " je zaključil nakupovanje");
}
```

Samo nakupovanje pa bi lahko izgledalo takole. Najprej se odločimo, koliko izdelkov bomo kupili (število izdelkov `stevilo` izberemo naključno med 1 in 10):

```
int stevilo = (int) (Math.random()*10) + 1;
```

Nato preverimo, ali je v trgovini dovolj izdelkov na zalogi za naš nakup:

```
if(trgovina.prosto(stevilo)) {
  // opravi nakup
}
else
  System.out.println("Ni dovolj zaloge.");
```

Samo transakcijo ob nakupu simuliramo tako, da nit za nekaj časa zaustavimo (interval izberemo naključno):

```
try {
  sleep((int) (Math.random()*500+50));
}
catch(InterruptedException e) {
}
```

Potem opravimo nakup in pri tem preverimo, ali se je uspešno izvedel:

```
if(trgovina.nakup(stevilo) == stevilo)
  System.out.println("Uspešen nakup.");
else
  System.out.println("Napaka pri nakupu.");
```

V metodi `main` najprej ustvarimo novo trgovino, nato pa še nekaj kupcev (njihovo število podamo kot argument programa). Nato izpišemo trenutno zalogo in poženemo kupce v nakupovanje.

```
public static void main(String[] args)
{
  Trgovina trg = new Trgovina();
  Kupec[] kupci = new Kupec[Integer.parseInt(args[0])];
  for(int i=0; i<kupci.length; i++)
    kupci[i] = new Kupec("Kupec " + Integer.toString(i+1), trg);
  System.out.println("Zaloge je " + trg.zaloga());
  for(int i=0; i<kupci.length; i++)
    kupci[i].start();
}
```

Program je v datoteki `Trgovina.java`, poskusite ga izvajati z različnim številom kupcev in opazujte njegovo delovanje (število kupcev morate programu podati kot argument).

```
public class Trgovina
{
  private int zaloga = 0;

  public Trgovina()
  {
    zaloga = (int) (Math.random()*500) + 10;
  }

  public int zaloga()
  {
    return zaloga;
  }

  public int nakup(int n)
  {
    if(zaloga >= n) {
      zaloga -= n;
      return n;
```

```
    }
    return -1;
  }

  public boolean prosto(int n)
  {
    if(zaloga >= n)
      return true;
    else
      return false;
  }

  public static void main(String[] args)
  {
    Trgovina trg = new Trgovina();
    Kupec[] kupci = new Kupec[Integer.parseInt(args[0])];
    for(int i=0; i<kupci.length; i++)
      kupci[i] = new Kupec("Kupec " + Integer.toString(i+1), trg);
    System.out.println("Zaloge je " + trg.zaloga() + ", trgovina je odprta ...");
    for(int i=0; i<kupci.length; i++)
      kupci[i].start();
  }
}

class Kupec extends Thread
{
  private Trgovina trgovina;

  public Kupec(String i, Trgovina t)
  {
    super(i);
    trgovina = t;
  }

  public void run()
  {
    while (trgovina.zaloga() > 0) {
      int stevilo = (int) (Math.random()*10) + 1;
      if(trgovina.prosto(stevilo)) {
        try {
          sleep((int) (Math.random()*500+50));
        }
        catch(InterruptedException e) {
        }
        if(trgovina.nakup(stevilo) == stevilo)
          System.out.println("Uspešen nakup - " + getName() + " -- število: "
                             + stevilo + " -- zaloga: " + trgovina.zaloga());
        else
          System.out.println("---> Napaka pri nakupu ...");
      }
      else
        System.out.println("Ni dovolj zaloge - " + getName() + " -- število: "
                           + stevilo + " -- zaloga: " + trgovina.zaloga());
    }
    System.out.println(getName() + " je zaključil nakupovanje");
  }
}
```

Program Trgovina.java

Glede na to, da pred vsakim nakupom preverimo, ali je na zalogi dovolj izdelkov, bi si mislili, da je preverjanje uspešno izvedenega nakupa povsem odveč, saj je nakup neuspešen le takrat, ko na zalogi ni dovolj izdelkov. A ob zagonu programa vidimo, da ni tako. Kljub temu, da pred nakupom preverjamo stanje zaloge in opravimo nakup le v

primeru zadostne zaloge, nam program občasno izpiše tudi napako pri nakupu (torej je ob klicu metode nakup stanje zalog manjše od želenega nakupa).

Pri enem samem kupcu program lepo deluje. Težave pa lahko nastanejo že pri dveh kupcih. Zakaj?

Ta primer prikazuje glavno težavo pri uporabi niti - nikoli namreč ne vemo, kdaj se nit izvaja oziroma kdaj bo njeno izvajanje prekinjeno. Tako se lahko zgodi, da en kupec (ena nit) preveri zalogo za želen nakup, preden pa mu uspe nakup do konca izvesti, nakupovanje prevzame drug kupec (izvajanje niti se prekine in procesorski čas se nameni drugi niti). Ta drugi kupec pokupi vso zalogo izdelkov ter preda nakupovanje spet prvemu kupcu. Prvi kupec, ki se te prekinitve pri nakupovanju sploh ne zaveda, nadaljuje z nakupovanjem tam, kjer je prej ostal. Zalogo je že preveril, torej izvede sam nakup. Ker pa se je medtem zaloga brez njegove vednosti spremenila, je nakup neuspešen.

Kadar lahko različne niti dostopajo do istih podatkov, lahko pride do nepredvidljivih rezultatov, zato moramo te podatke na nek način zaščititi pred deljenim dostopom. Poskrbeti moramo, da več niti ne dostopa istočasno do istega vira vsaj v kritičnih delih. Takim navzkrižjem se lahko izognemo tako, da vir zaklenemo, ko ga uporablja ena nit. Dokler je zaklenjen, ga druge niti ne morejo uporabiti. Ko nit vira ne potrebuje več, ga ponovno odklene in s tem omogoči drugim nitim, da ga uporabijo.

Na ta način damo deljene objekte v izključno rabo le eni niti naenkrat. To pomeni, da ima le ena nit dostop do objekta, vse ostale pa čakajo, da ta nit zaključi svojo nalogo. Šele potem lahko druga nit dobi dostop do objekta. Ta proces imenujemo sinhronizacija niti (*thread synchronization*).

Java ima že vgrajen mehanizem, ki preprečuje sočasen dostop do podatkov objekta. Ker ponavadi do njih dostopamo preko metod, lahko metodo označimo kot sinhronizirano (*synchronized*). Samo ena nit naenkrat lahko kliče sinhronizirano metodo določenega objekta (čeprav lahko ta nit kliče hkrati več sinhroniziranih metod tega objekta).

Vsak objekt privzeto vsebuje eno ključavnico, ki jo imenujemo tudi semafor (*monitor*), ki je avtomatično del objekta. Ob klicu katerekoli sinhronizirane metode se objekt zaklene in nobena druga sinhronizirana metoda tega objekta ne more biti poklicana (iz druge niti), dokler se prva ne zaključi in ponovno odklene objekt. Eno samo ključavnico si torej delijo vse sinhronizirane metode posameznega objekta. Vsaka metoda, ki dostopa do kritičnih virov, mora biti sinhronizirana.

V Javi sinhronizacijo napovemo z rezervirano besedo `synchronized`. Najpogostejši način sinhronizacije je sinhronizacija cele metode:

```
public synchronized int imeMetode()
```

Sinhroniziramo pa lahko tudi le kritični del kode znotraj metode, ki jo imenujemo kritični odsek (*critical section*). Tu uporabimo blok `synchronized`, kjer v oklepaju podamo objekt, katerega ključavnico uporabimo za sinhronizacijo kode v bloku:

```
synchronized(objekt) {
  ...
}
```

Če se vrnemo na naš zadnji primer, moramo najprej ugotoviti, kje so težave pri usklajevanju dostopa. Problem predstavlja sočasen dostop do atributa `zaloga`, torej bomo morali uporabiti ključavnico objekta `trgovina`.

Kritični del kode je sama akcija nakupa:

```
if(trgovina.prosto(stevilo)) {
  try {
    sleep((int) (Math.random()*500+50));
  }
  catch(InterruptedException e) {
  }
  if(trgovina.nakup(stevilo) == stevilo)
    System.out.println("Uspešen nakup");
  else
    System.out.println("Napaka pri nakupu ...");
}
else
  System.out.println("Ni dovolj zaloge");
```

Zato ta del kode postavimo v blok `synchronized`, za ključavnico pa uporabimo objekt `trgovina`:

```
synchronized(trgovina) {
  // kritični del kode
}
```

Katere pa so metode znotraj objekta trgovina, ki morajo biti sinhronizirane? To sta dve metodi, prva za izvedbo nakupa in druga za preverjanje razpoložljive zaloge:

```
public synchronized int nakup(int n)
public synchronized boolean prosto(int n)
```

Ustrezno popravljen program (datoteka `TrgovinaS.java`) sedaj deluje pravilno, saj smo preprečili več nitim sočasen dostop do atributa `zaloga`. Njegovo delovanje preverite z različnim številom kupcev.

```
public class TrgovinaS
{
  private int zaloga = 0;
```

```
  public TrgovinaS()
  {
    zaloga = (int) (Math.random()*500) + 10;
  }

  public int zaloga()
  {
    return zaloga;
  }

  public synchronized int nakup(int n)
  {
    if(zaloga >= n) {
      zaloga -= n;
      return n;
    }
    return -1;
  }

  public synchronized boolean prosto(int n)
  {
    if(zaloga >= n)
      return true;
    else
      return false;
  }

  public static void main(String[] args)
  {
    TrgovinaS trg = new TrgovinaS();
    Kupec[] kupci = new Kupec[Integer.parseInt(args[0])];
    for(int i=0; i<kupci.length; i++)
      kupci[i] = new Kupec("Kupec " + Integer.toString(i+1), trg);
    System.out.println("Zaloge je " + trg.zaloga() + ", trgovina je odprta ...");
    for(int i=0; i<kupci.length; i++)
      kupci[i].start();
  }
}

class Kupec extends Thread
{
  private TrgovinaS trgovina;

  public Kupec(String i, TrgovinaS t)
  {
    super(i);
    trgovina = t;
  }

  public void run()
  {
    while (trgovina.zaloga() > 0) {
      int stevilo = (int) (Math.random()*10) + 1;
      synchronized(trgovina) {
        if(trgovina.prosto(stevilo)) {
          try {
            sleep((int) (Math.random()*500+50));
          }
          catch(InterruptedException e) {
          }
          if(trgovina.nakup(stevilo) == stevilo)
            System.out.println("Uspešen nakup - " + getName() + " -- število: "
                               + stevilo + " -- zaloga: " + trgovina.zaloga());
          else
            System.out.println("---> Napaka pri nakupu ...");
        }
        else
          System.out.println("Ni dovolj zaloge - " + getName() + " -- število: "
                             + stevilo + " -- zaloga: " + trgovina.zaloga());
```

```
      }
    }
    System.out.println(getName() + " je zaključil nakupovanje");
  }
}
```

Program TrgovinaS.java

Proizvajalec in potrošnik

Včasih pa moramo delo dveh niti tudi časovno uskladiti. Tak primer sta dve niti, od katerih ena pripravlja podatke (proizvajalec ali *producer*), druga pa te podatke uporablja (potrošnik ali *consumer*). Pri tem ni dovolj, da obe niti ne dostopata istočasno do podatkov, temveč moramo njuno delo uskladiti tako, da druga nit ne dobi podatka, ki še ni pripravljen, hkrati pa tudi ne sme dvakrat prebrati istega podatka.

Oglejmo si preprost primer. Proizvajalec naj v naključnih časovnih intervalih generira števila med 1 in 10 ter jih shranjuje v odložišču. Potrošnik pa naj iz odložišča (ki je isti objekt, kot ga uporablja proizvajalec za odlaganje števil) jemlje števila, takoj ko so ta na voljo. Svoje delo zaključi, ko dobi zadnje število (to je 10).

Napišimo najprej razred, ki predstavlja odložišče. Razred ima en privatni atribut, to je trenutna vsebina odložišča:

```
public class Odlozisce
{
  private int vsebina;

}
```

V razredu imamo tudi dve metodi, s pomočjo katerih beremo oz. spreminjamo vrednost atributa. Metoda `postavi` nastavi vrednost atributa `vsebina` na podano vrednost (postavi število v odložišče):

```
public void postavi(int v)
{
  vsebina = v;
  System.out.println("Postavljeno: " + v);
}
```

Metoda `vzemi` pa vrne vrednost atributa `vsebina` (branje vsebine odložišča); ko je vsebina prebrana, se njena vrednost nastavi na 0:

```
public int vzemi()
{
  int v = vsebina;
  vsebina = 0;
  System.out.println("Vzeto: " + v);
  return v;
}
```

Proizvajalca bomo opisali z razredom `Proizvajalec`, ki ga izpeljemo iz razreda `Thread`, saj želimo svojo nit za generiranje števil.

```
public class Proizvajalec extends Thread
```

Metodo, ki določa delovanje te niti, lahko zapišemo takole: v zanki postavljamo na odložišče števila od 1 do 10 po vrsti, pred vsako ponovitvijo zanke pa počakamo nekaj časa (interval je izbran naključno med 0 in 100 milisekundami).

```
public void run()
{
  for (int i = 1; i <= 10; i++) {
    polica.postavi(i);
    try {
      sleep((int)(Math.random()*100));
    }
    catch (InterruptedException e) {
    }
  }
}
```

Objekt, ki predstavlja odložišče, smo poimenovali `polica`, objektna spremenljivka je atribut razreda.

```
private Odlozisce polica;
```

Ker naj bi bilo odložišče skupno proizvajalcu in potrošniku, saj preko njega izmenjujeta podatke (števila), moramo referenco na ta objekt podati ob klicu konstruktorja proizvajalca.

```
public Proizvajalec(Odlozisce o)
{
  this.polica = o;
}
```

Razred `Potrosnik` opisuje potrošnika, ki iz odložišča jemlje števila. Tudi potrošnik deluje v svoji niti:

```
public class Potrosnik extends Thread
```

Njegovo delovanje določa metoda `run`. V zanki jemlje števila iz odložišča, dokler ne pride do zadnjega števila, to je števila z vrednostjo 10. Takrat se zanka zaključi in z njo tudi metoda `run` ter s tem tudi sama nit. Ob vsaki ponovitvi zanke počakamo nekaj časa (tudi tu je interval izbran naključno med 0 in 100 milisekundami), da se zanka ne izvaja prehitro.

```
public void run()
{
  int vrednost = 0;
```

```
    do {
      try {
        sleep((int)(Math.random()*100));
      }
      catch (InterruptedException e) {
      }
      vrednost = polica.vzemi();
    } while(vrednost < 10);
  }
```

Tudi v potrošniku smo definirali atribut `polica`, ki je referenca na odložišče, torej na isti objekt, kot ga uporablja proizvajalec za hranjenje števil. Referenco na ta objekt smo podali on klicu konstruktorja potrošnika.

```
private Odlozisce polica;

public Potrosnik(Odlozisce o)
{
  this.polica = o;
}
```

Na koncu napišimo še razred, katerega metoda `main` poskrbi za ustvarjanje odložišča, proizvajalca in potrošnika ter za zagon obeh niti:

```
public class PPAsinhrono
{
  public static void main(String[] args)
  {
    Odlozisce odl = new Odlozisce();
    Proizvajalec pro = new Proizvajalec(odl);
    Potrosnik pot = new Potrosnik(odl);
    pro.start();
    pot.start();
  }
}
```

Obe niti, proizvajalec in potrošnik, sta v tem primeru nesinhronizirani, zato smo razred tudi tako poimenovali. Cel program je v datoteki `PPAsinhrono.java`. Preizkusite delovanje programa in preverite, ali rešitev ustreza problemu, kot smo si ga zamislili.

```
public class PPAsinhrono
{
  public static void main(String[] args)
  {
    Odlozisce odl = new Odlozisce();
    Proizvajalec pro = new Proizvajalec(odl);
    Potrosnik pot = new Potrosnik(odl);
    pro.start();
    pot.start();
  }
}
```

```
class Odlozisce
{
  private int vsebina;

  public int vzemi()
  {
    int v = vsebina;
    vsebina = 0;
    System.out.println("Vzeto: " + v);
    return v;
  }

  public void postavi(int v)
  {
    vsebina = v;
    System.out.println("Postavljeno: " + v);
  }
}

class Proizvajalec extends Thread
{
  private Odlozisce polica;

  public Proizvajalec(Odlozisce o)
  {
    this.polica = o;
  }

  public void run()
  {
    for (int i = 1; i <= 10; i++) {
      polica.postavi(i);
      try {
        sleep((int)(Math.random()*100));
      }
      catch (InterruptedException e) {
      }
    }
  }
}

class Potrosnik extends Thread
{
  private Odlozisce polica;

  public Potrosnik(Odlozisce o)
  {
    this.polica = o;
  }

  public void run()
  {
    int vrednost = 0;
    do {
      try {
        sleep((int)(Math.random()*100));
      }
      catch (InterruptedException e) {
      }
      vrednost = polica.vzemi();
    } while(vrednost < 10);
  }
}
```

Program PPAsinhrono.java

Proizvajalec in potrošnik delita podatke preko objekta razreda `Odlozisce`. Čeprav naj bi potrošnik prebral vsako vrednost natanko enkrat, pa pri izvajanju programa opazimo, da ni vedno tako. Pojavita se lahko dve vrsti problemov:

- Proizvajalec je hitrejši od potrošnika in generira dve števili, preden uspe potrošnik prebrati prvo število. V tem primeru potrošnik izpusti število.
- Potrošnik je hitrejši od proizvajalca in že drugič prebere število, preden uspe proizvajalec zgenerirati novo število. Tako potrošnik prebere število 0, čeprav le-to ni med števili proizvajalca.

Obema opisanima problemoma se lahko izognemo, če proizvajalčevo shranjevanje novega števila v odložišče časovno uskladimo s porabnikovim branjem tega števila. Program popravimo tako, da bo delovanje obeh niti usklajeno.

Najprej moramo zagotoviti, da obe niti ne dostopata istočasno do vrednosti v odložišču. Sočasen dostop lahko preprečimo z uporabo zaklepanja objekta. V našem primeru sta kritični metodi `vzemi` in `postavi` v razredu `Odlozisce`. Obe torej označimo kot sinhronizirani:

```
public class Odlozisce
{
  private int vsebina;

  public synchronized int vzemi()
  {
    // ...
  }

  public synchronized void postavi(int v)
  {
    // ...
  }
}
```

Ko vstopimo v sinhronizirano metodo, nit, ki je klicala to metodo, zaklene objekt, katerega metodo je poklicala. Tako ostale niti ne morejo klicati nobene sinhronizirane metode istega objekta, dokler je ta objekt zaklenjen. Nit, ki je zaklenila objekt, lahko poljubno kliče tudi ostale sinhronizirane metode tega objekta, saj ima že kontrolo nad njegovo ključavnico.

Vendar pa s tem problema nismo rešili, saj lahko proizvajalec še vedno prehiteva potrošnika ali obratno. Zato moramo obe niti tudi časovno uskladiti. Proizvajalec mora na nek način obvestiti potrošnika, da je nova vrednost na voljo, potrošnik pa mora na nek način obvestiti proizvajalca, da lahko pripravi novo vrednost. Za to lahko v sinhronizirani metodi uporabimo metodi `wait` in `notify` (ali `notifyAll`). Prva omogoča niti, da počaka, dokler ni izpolnjen določen pogoj, druga pa obvesti čakajoče niti (eno ali vse), ko se ta pogoj spremeni. Vse omenjene metode so metode razreda `Object`, zato jih podedujejo vsi objekti v Javi.

V razred `Odlozisce` moramo torej dodati še en privatni atribut, ki pove, ali je nova vrednost na voljo (resnično, ko je nova vrednost nastavljena in še ne prebrana, in neresnično, kadar je vrednost prebrana in še ni ponovno nastavljena):

```
private boolean naVoljo = false;
```

Program želimo napisati tako, da bo potrošnik čakal, dokler proizvajalec ne bo postavil nove vrednosti v odložišče in o tem obvestil potrošnika. Podobno pa naj tudi proizvajalec čaka, dokler potrošnik ne prebere vrednosti iz odložišča in o svoji aktivnosti obvesti proizvajalca. Šele takrat lahko proizvajalec postavi novo vrednost v odložišče.

Metodi `vzemi` in `postavi` moramo spremeniti tako, da čakata in obveščata druga drugo o svojih aktivnostih.

Koda metode `vzemi` se tako začne z zanko, katera se ponavlja, dokler v odložišču ni na voljo nove vrednosti (dokler proizvajalec ne zgenerira nove vrednosti). V zanki kliče metodo `wait`, ki povzroči čakanje te metode, dokler je proizvajalec ne obvesti, da je zaključil svoje delo (metoda `wait` povzroči, da potrošnik sprosti ključavnico nad objektom odložišče, ki jo pridobi ob vstopu v sinhronizirano metodo, ter tako omogoči proizvajalcu, da zaklene ta objekt in nastavi vrednost v odložišču). Ko proizvajalec nastavi vrednost v odložišču, s klicem metode `notify` o tem obvesti potrošnika. Metoda, ki ima izključen nadzor nad deljenim objektom, lahko namreč pokliče metodo `notify` ali `notifyAll`, ki sporoči čakajočim nitim (eni ali vsem), da lahko prekinejo s čakanjem. Potrošnik nato prekine svoje stanje čakanja (vrednost spremenljivke `naVoljo` se je med tem tudi spremenila na `true`), izstopi iz `while` zanke ter vrne vrednost iz odložišča.

```
public synchronized int vzemi()
{
  while(naVoljo == false) {
    try {
      wait();
    }
    catch (InterruptedException e) {
    }
  }
  int v = vsebina;
  vsebina = 0;
  System.out.println("Vzeto: " + v);
  naVoljo = false;
  notify();
  return v;
}
```

Metoda `wait` lahko sproži izjemo `InterruptedException`, kadar jo prekine druga nit, zato jo postavimo v `try`/`catch` blok. Izjemo prestrežemo, a ne ukrepamo, saj je tak dogodek pričakovan.

Podobno spremenimo tudi kodo metode `postavi`, kjer na začetku v zanki čakamo, da potrošnikova nit prebere trenutno vrednost iz odložišča ter obvesti proizvajalca, da lahko v odložišče postavi novo vrednost. Tu gre za primer, ko proizvajalec čaka na potrošnika, da opravi svoje delo, zato se `while` zanka ponavlja, dokler je vrednost `naVoljo` resnična.

```
public synchronized void postavi(int v)
{
  while(naVoljo == true) {
    try {
      wait();
    }
    catch (InterruptedException e) {
    }
  }
  vsebina = v;
  System.out.println("Postavljeno: " + v);
  naVoljo = true;
  notify();
}
```

Tako dopolnjena koda je v datoteki `PPSinhrono.java`, ki smo jo tako poimenovali zaradi usklajenega dela proizvajalca in potrošnika. Preizkusite delovanje programa in preverite, ali obe niti res delujeta usklajeno in v pravilnem zaporedju (vrstni red izpisov bi moral biti vedno enak).

```
public class PPSinhrono
{
  public static void main(String[] args)
  {
    Odlozisce odl = new Odlozisce();
    Proizvajalec pro = new Proizvajalec(odl);
    Potrosnik pot = new Potrosnik(odl);
    pro.start();
    pot.start();
  }
}

class Odlozisce
{
  private int vsebina;
  private boolean naVoljo = false;

  public synchronized int vzemi()
  {
    while(naVoljo == false) {
      try {
        wait();
      }
      catch (InterruptedException e) {
      }
    }
    int v = vsebina;
    vsebina = 0;
    System.out.println("Vzeto: " + v);
    naVoljo = false;
    notify();
```

```
    return v;
  }

  public synchronized void postavi(int v)
  {
    while(naVoljo == true) {
      try {
        wait();
      }
      catch (InterruptedException e) {
      }
    }
    vsebina = v;
    System.out.println("Postavljeno: " + v);
    naVoljo = true;
    notify();
  }
}

class Proizvajalec extends Thread
{
  private Odlozisce polica;

  public Proizvajalec(Odlozisce o)
  {
    this.polica = o;
  }

  public void run()
  {
    for (int i = 1; i <= 10; i++) {
      polica.postavi(i);
      try {
        sleep((int)(Math.random()*100));
      }
      catch (InterruptedException e) {
      }
    }
  }
}

class Potrosnik extends Thread
{
  private Odlozisce polica;

  public Potrosnik(Odlozisce o)
  {
    this.polica = o;
  }

  public void run()
  {
    int vrednost = 0;
    do {
      try {
        sleep((int)(Math.random()*100));
      }
      catch (InterruptedException e) {
      }
      vrednost = polica.vzemi();
    } while(vrednost < 10);
  }
}
```

Program PPSinhrono.java

8. Literatura

U. Mesojedec in B. Fabjan: *Java 2: temelji programiranja*, Ljubljana: Pasadena, 2004.

J. Farrell: *Java Programming*, Third Edition, Thomson Course Technology, 2006.

R. Morelli: *Java, Java, Java: Object-oriented problem solving*, Prentice hall, 2000.

B. McLaughlin in D. Flanagan: *Java 1.5 Tiger*, O'Reilly Media, Inc., 2004.

B. Eckel: *Thinking in Java*, Prentice Hall PTR, 1998.

Spletni vir: *The Java Tutorial*, http://java.sun.com/docs/books/tutorial/index.html

Spletni vir: *Painting in AWT and Swing*,
http://java.sun.com/products/jfc/tsc/articles/painting/index.html