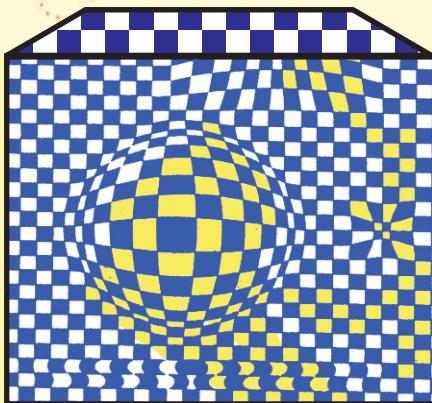
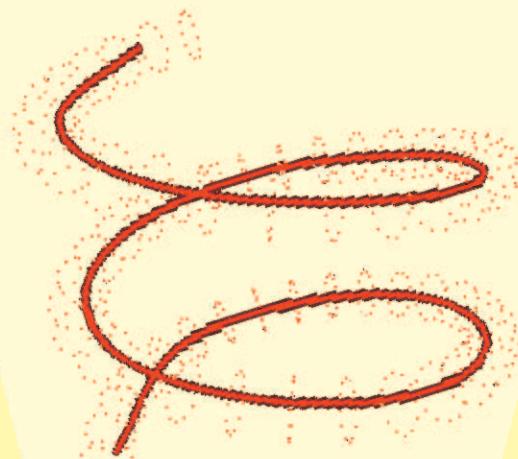


prof. dr. DŽENAN RIĐANOVIĆ

MAGIČNE KUTIJE:

SPIRALNI PRISTUP UČENJU JAVA PROGRAMSKOG JEZIKA I SWING GRAFIČKE BIBLIOTEKE



MAGIČNE KUTIJE:
SPIRALNI PRISTUP UČENJU JAVA PROGRAMSKOG JEZIKA
I SWING GRAFIČKE BIBLIOTEKE

Dženan Riđanović
Magične kutije:
Spiralni pristup učenju Java programskog jezika
i Swing grafičke biblioteke

Izdavač
TDP d.o.o., Sarajevo

Za izdavača
Narcis Pozderac

Recenzenti
Dr. Diana Protić-Tkalčić
Dr. Novica Nosović

Dizajn korica
Ema Mazrak

Računarska obrada i štampa
TDP d.o.o., Sarajevo

2005 © Dženan Riđanović
<http://drdb.fsa.ulaval.ca/>

CIP - Katalogizacija u publikaciji
Nacionalna i univerzitetska biblioteka
Bosne i Hercegovine, Sarajevo

004.438 JAVA(035)

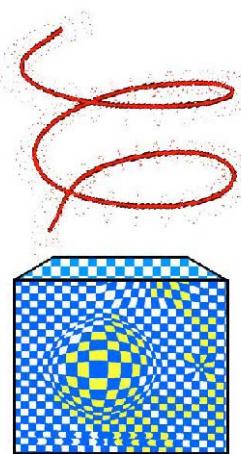
RIĐANOVIĆ, Dženan
Magične kutije : spiralni pristup učenju Java
programskega jezika i Swing grafičke biblioteke /
Dženan Riđanović. - Sarajevo : TDP, 2005. - 140
str. : graf. prikazi ; 24 cm

ISBN 9958-9214-2-1

COBISS.BH-ID 14405894

Dženan Riđanović

**Magične kutije:
Spiralni pristup učenju
Java programskog jezika
i Swing grafičke biblioteke**



Maj 2005

SADRŽAJ

PREDGOVOR	7
O SPIRALNOM PRISTUPU	9
SOFTVER	11
POGLAVLJE 1: OSNOVE JAVA PROGRAMSKOG JEZIKA	13
POGLAVLJE 2: RAZVOJNO OKRUŽENJE	21
POGLAVLJE 3: OKVIR ILI PROZOR	29
DIO A: APLIKACIJA	31
DIO B: APLET	36
POGLAVLJE 4: PANEL	49
DIO A: PRAZAN PANEL	51
DIO B: PLAVI PANEL U DIJAGRAMU	58
DIO C: ŽUTA KUTIJA U DIJAGRAMU	63
POGLAVLJE 5: DOGAĐAJI	67
POGLAVLJE 6: MENI	79
POGLAVLJE 7: DINAMIČKA MANIPULACIJA OBJEKATA	89
DIO A: KUTIJE U DIJAGRAMU	91
DIO B: MENI BAR	96
DIO C: BAR ZA POMJERANJE RADNE POVRŠINE	101
POGLAVLJE 8: BAR ZA ALATE	105
POGLAVLJE 9: O APLIKACIJI	115
POGLAVLJE 10: INTERNACIONALIZACIJA APLIKACIJE	131

PREDGOVOR

Objektno orijentisani (OO) pristup se dokazao i u praksi i u teoriji. Ovaj pristup je povećao produktivnost u dizajnu i programiranju softvera i informacionih sistema. Baziran je na dva fundamentalna koncepta: nasljeđivanju i sakrivanju informacija. Nasljeđivanje se koristi kako bi se dodala nova funkcionalnost na temelju već postojeće. Sakrivanje informacija se koristi kako bi se informacije zaštitile i kako bi se obezbijedile korisničke operacije koje su imune na promjene reprezentacije tih informacija. Nasljeđivanje se zasniva na iskustvu drugih, dok sakrivanje informacija omogućava njihovo mijenjanje bez uticaja na korisnike.

Jedan od najpopularnijih OO programskih jezika je Java. Prisustvo Java interpretera na korisničkoj radnoj stanici, koji se naziva Java "Virtuelna Mašina" (VM), odstranjuje ograničenja vezana za određeni operativni sistem. Java kôd se analizira i generiše se "bajt kôd" kako bi se očuvala portabilnost kôda. Java dozvoljava upotrebu aplikacija na raznim platformama kao što su Linux, Windows ili Macintosh.

Postoji nekoliko razloga zašto je Java postala popularna i u akademskim krugovima i na tržištu. Prvo, Java je OO programski jezik. Drugo, Java kôd je portabilan. Treće, Java aplikacije se smještaju na server i mogu se izvršavati i na klijent računaru i na serveru. Četvrti, Java programski jezik se razvio poslije nastanka Internet-a. Npr. Java kôd koji se koristi u ovoj knjizi, je dostupan na serveru <http://drdb.fsa.ulaval.ca/> i može se pokrenuti sa Interneta.

Kao što je navedeno, korištenje OO pristupa i Java tehnologije osigurava mnogo značajnih prednosti. Međutim, razvijanje aplikacija na ovaj način zahtijeva specifična teoretska i tehnička znanja koja se mogu usvojiti samo kroz dug i naporan proces učenja. Kako bi pojednostavili proces učenja razvijanja softvera koristeći Java programski jezik, koji uključuje odredene kompleksne mehanizme, preporučujem korištenje iterativnog ili "spiralnog pristupa" u učenju. Ovaj pristup omogućava "evolutivno" učenje koncepata kroz ponavljanje ciklusa razvijanja softverskih verzija od veoma jednostavnih do kompleksnijih. Kroz ovaj postepeni proces, student razvija svoje samopouzdanje. Koristio sam ovaj metod zadnjih 8 godina sa mojim studentima sa odličnim rezultatima.

Ova knjiga je specijalno pripremljena za one koji žele da nauče nove koncepte objektno orijentisanog pristupa kroz praksu. Prema tome, i profesionalac

koji želi da proširi svoje znanje, kao i početnik u razvijanju softvera i informacijskim sistemima, može imati koristi od ove knjige.

Java standardni skup alata (J2SE ili JDK ili SDK) se može besplatno snimiti sa Interneta sa web stranice <http://java.sun.com/>. Integrисана razvojna okruženja (IDE) kao što su JBuilder (<http://www.borland.com/>) i Eclipse (<http://www.eclipse.org/>) se mogu korsititi za editovanje, kompiliranje i pokretanje kôda. Personalna (Foundation) verzija JBuilder-a i Open Source Eclipse su dostupni besplatno.

O SPIRALNOM PRISTUPU

Učenje novog OO programskog jezika kao što je Java predstavlja težak zadatak. Postoje već mnoge knjige o Java jeziku. Većina ovih knjiga imaju mnogo stranica i pokrivaju veliki broj tema.

Java je kompleksna tehnologija i vi se morate zapitati koji je najlakši i najbrži način da počnete učiti takvu tehnologiju. Meni je trebalo više od deset godina učenja i predavanja da otkrijem da inicijalno učenje nove tehnologije treba biti orijentisano prema projektnom zadatku a ne prema temama. Većina Java knjiga su organizovane prema temama. Samo je potreban brzi pregled sadržaja gotovo svake Java knjige i shvatićete da skoro svako poglavlje počinje sa novom temom.

U mojim predavanjima, koristim spiralni pristup koji se sastoji od jednog projekta razvijenog u više spirala. U ovoj knjizi, projekat je jednostavan grafički softver koji dozvoljava korisniku da kreira i pomjera "kutije" po ekranu. Čak i za ovako jednostavan projekt, potrebno je koristiti mnogo spirala. Prva spirala kreira vrlo jednostavnu Java klasu koja ne radi ništa korisno. Ali, postoje osnovni koncepti koji su korišteni u ovako jednostavnom zadatku i svaki od njih je objašnjen u Poglavlju 1.

Sa svakom novom spiralom, projekat raste i upoznaju se novi koncepti. Nova spirala je objašnjena u odnosu na prethodnu. Spirale mogu tretirati istu temu nekoliko puta, ali svaki naredni put sa poboljšanom i novom verzijom softvera.

Kao univerzitetski profesor, shvatio sam da studenti bez ikakvog iskustva u programiranju mogu naučiti osnove OO programiranja u Java jeziku za deset sedmica.

Želio bih se zahvaliti Rémy Mathieu za vrijedan rad na francuskoj verziji ove knjige i Vensadi Okanović za mnoge korekcije na našem jeziku.

*Dženan Riđanović
Sarajevo, 25. maj 2005.*

SOFTVER

Knjiga sadrži CD sa softverom potrebnim za editovanje i izvršavanje Java kôda. CD se sastoji od 4 direktorija:

```
eclipse
    eclipse-SDK-2.1.1-win32.zip
jBuilder
    jbx-windows.zip
magicneKutije
    code
        ch01
        ...
        ch10
        readme.txt
sdk
    j2se1.4.2
    j2sdk-1_4_2_01-windows-i586.exe
```

Koristite web stranicu knjige na mom serveru, za skidanje novih verzija softvera: <http://drdb.fsa.ulaval.ca/mk/>.

Prvo se mora instalirati Java, pa onda Eclipse ili JBuilder. Pročitajte osnovna uputstva na: <http://drdb.fsa.ulaval.ca/mk/>.

Kôd spirale 10 na CD-u se izvršava bez problema sa Java verzijom sa CD-a. Da bi izbjegli probleme sa novom Java verzijom, pokupite novi kôd spirale 10 sa mog servera: <http://drdb.fsa.ulaval.ca/mk/>. Novi kôd spirale 10 radi i sa Java verzijom sa CD-a, a i sa novom Java verzijom. Poglavlje 10 objašnjava novu verziju kôda.

Kôd spirala se može izvršiti i na Internetu, kao apleti ili aplikacije: <http://drdb.fsa.ulaval.ca/mk/>.

Naći će se dosta korisnih web linkova na stranici: <http://drdb.fsa.ulaval.ca/>.

Provjerite, s vremena na vrijeme, web stranicu knjige za zadnje novosti.

POGLAVLJE 1

OSNOVE JAVA PROGRAMSKOG JEZIKA

Ovo poglavlje uvodi osnovne notacije Java programskog jezika. Preciznije, cilj ovog poglavlja je upoznavanje sa: 1) konceptom objekta i klase, i, 2) konceptom nasljedivanja.

Izvršavanje aplikacije prezentirane u ovom poglavlju neće dati nikakve vizuelne rezultate.



Osnovne strukture

Objekat

Prvi zadatak u dizajnu softvera ili informacionih sistema se sastoji od spoznaje realnog svijeta u kojem korisnici komuniciraju. Moramo biti sigurni da finalni softverski proizvod što bolje reflektuje tu realnost.

Sa tačke gledišta podataka i operacija na podacima, možemo reći da se naš cilj sastoji od organizovanja podataka kao objekata koji predstavljaju realni svijet.

Objekti su **apstrakcija realnosti**.

Ovi objekti moraju odgovarati potrebama korisnika. Objekti su apstrakcija realnog svijeta. Oni enkapsuliraju dio znanja realnog svijeta u kojem korisnici rade. Kao i živa bića, objekti se radaju, žive i umiru.

Klasa

Kako bismo grupisali objekte, pokušavamo da prirodno udružimo elemente koji liče jedni drugima.

Svaki objekat **pripada** određenoj klasi.

Klasa je opis skupa objekata koji dijele istu definiciju. Svaki objekat pripada određenoj klasi. Generalne osobine predstavljaju klasu, dok specifične karakteristike čine objekte.

Kreiranje objekta se dešava u procesu instanciranja klase. Dakle, svaki objekat je instanca klase.

Objektno orijentisani jezici, kao što je Java, smanjuju razliku između koncepata poznatih ljudima i jezika koji je poznat računarima. To je jedan od razloga zašto je lakše razvijati aplikacije sa objektno orijentisanim jezicima nego sa tradicionalnim jezicima.

Klasa: Specifikacija podataka i operacija.

Koncept objekta je vezan uz koncept klase. Klasa opisuje format podataka i operacije nad podacima. Nemoguće je pisanje Java kôda bez korištenja ovog osnovnog koncepta.

Ovdje je prezentiran prvi primjer Java klase:

Klasa Start

```
1: public class Start {  
2:  
3:     public Start() {  
4:         super();  
5:     }  
6:  
7:     public static void main(String[] args) {  
8:         new Start();  
9:     }  
10:  
11: }
```

U ovom primjeru kôd koji je prikazan definiše klasu *Start*. Zaglavlje ili potpis klase je prikazano na liniji 1:

```
1: public class Start {
```

Ključna riječ *public* dozvoljava pristup ovoj klasi iz bilo koje druge klase. Ključna riječ *class* dolazi prije imena klase. Otvorena vitičasta zagrada ”{” počinje tijelo klase koje se završava sa zatvorenom vitičastom zagradom ”}”.

Treba primijetiti sljedeće:

Ime klase počinje velikim slovom.

- ime klase počinje velikim slovom;
- kako bismo objasnili Java kôd, dodjeljujemo svakoj liniji broj sa lijeve strane kôda. Ovi brojevi zajedno sa ":" **nisu** dio Java programskega jezika;

Metode

Klasa *Start* nema atributa. Njeno tijelo se sastoji od dva metoda. Metode su operacije koje se mogu izvršiti nad objektima.

Konstruktor

Konstruktor:

Specijalna metoda klase koja služi za kreiranje objekata.

Svaka klasa ima specijalnu metodu za kreiranje objekata. Ta metoda se naziva konstruktor klase. Klasa može imati više od jednog konstruktora, a razlike tih konstruktora su vidljive u argumentima metoda.

Ime konstruktora = Ime klase

Konstruktor ima isto ime kao i klasa. Karakteri ”()” slijede njegovo ime. Unutar navedenih zagrada optionalno se dodaju argumenti.

U našem primjeru, linije od 3 do 5 pokazuju definiciju konstruktora klase *Start*:

```
3: public Start() {  
4:     super();  
5: }
```

U suprotnosti od konstruktora, prvi karakter imena **metoda** mora počinjati **malim slovom**.

Generalno, u klasi, bilo koje ime može biti dano metodama. Za razliku od konstruktora, ime metode mora početi sa malim slovom. Preporučujem da pratite ove programerske standarde jer oni povećavaju preglednost kôda.

main Metoda

Kao i konstruktor, *main* metoda je različita od ostalih metoda. U našem primjeru, *main* metoda je definisana na sljedeći način:

```
7: public static void main(String[] args) {  
8:     new Start();  
9: }
```

Kada se pokrene aplikacija, Java VM poziva **main** metodu.

Java VM poziva *main* metodu kada se aplikacija pokrene. U našem primjeru sljedeća linija kôda

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

služi kao početna tačka aplikacije.

Linija 8,

```
8: new Start();
```

nam pokazuje način na koji se poziva konstruktor *Start()* koji kreira objekat prema definiciji klase *Start*.

Za pozivanje konstruktora klase koristi se ključna riječ *new*. Ova riječ mora biti uvijek navedena i mora uvijek prethoditi imenu konstruktora objekata.

Nakon kreiranja objekta dozvoljeno nam je pozivanje metoda na tom objektu.

Nasljedivanje

Kreiranje objekta prema definiciji klase *Start* je realizovano pokretanjem kôda koji je unutar konstruktora *Start()*. Taj kôd se zasniva na nasljđivanju klasa.

Sve klase su povezane hijerarhijskom strukturom na čijem vrhu se nalazi klasa *Object*.

Java koristi hijerarhijsku strukturu između klasa. Na vrhu piramide ćemo uvijek naći klasu pod imenom *Object*. Sve ostale klase su nasljednici ove klase.

U našem primjeru, klasa *Start* implicitno nasljeđuje klasu *Object*. Konstruktor klase *Start* poziva konstruktor klase *Object* kao što je navedeno na liniji 4:

```
4: super();
```

Modifikatori

Moguće je pozvati metodu, kao što je metoda *main*, bez prethodnog kreiranja objekta klase. To omogućava ključna riječ *static* u zagлавlu metode:

```
7: public static void main(String[] args)
```

Termini *public*, *static* i *void* su ustvari modifikatori metode i dio su liste rezervisanih riječi Java programskog jezika.

Metoda *main* koja ima modifikator *public* može biti pozvana izvan klase *Start*. Java poziva ovu metodu kada se aplikacija pokrene, na sljedeći način:

```
Start.main(...);
```

Ovaj primjer nam pokazuje kako metoda *main* može poslužiti kao početna tačka naše aplikacije izvana klase *Start*.

Generalna sintaksa pozivanja metoda, koja ne zahtijeva kreiranje objekata, je:

```
<Ime klase>.<ime metode>([parametri])
```

Metoda može vratiti, na mjestu poziva metode, određenu vrijednost. Ako metoda ne vraća nikakvu vrijednost koristi se modifikator *void*.

Kako bi se mogla dati kontrola eksternom okruženju Java aplikacije, veoma je važno da se deklarišu modifikatori *public*, *static* i *void* u metodi *main*, jer:

- metoda *main* mora biti javna jer se poziva izvan aplikacije;
- ne postoji kreiran objekat, tako da je opravdana upotreba ključne riječi *static* u specifikaciji metode *main*. Pozivanje metode *main* je određeno sa imenom pripadajuće klase;
- Java VM ne prima nikakvu vrijednost poslije završetka izvršavanja aplikacije, čime se takođe opravdava upotreba modifikatora *void*.

Parametri metoda

Kako bi se uspostavila komunikacija između različitih dijelova aplikacije Java koristi parametre metoda.

U našem primjeru metoda *main* sadrži jedan parametar:

```
String[] args
```

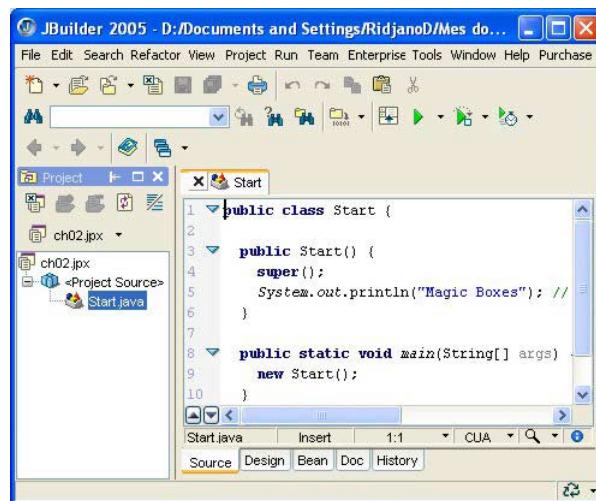
Termin "String" pravi referencu na predefinisanu Java klasu koja dozvoljava da radite sa tekstrom. Uglaste zagrade [] označavaju niz tekstova, tako da parametar *args* može da ima više vrijednosti.

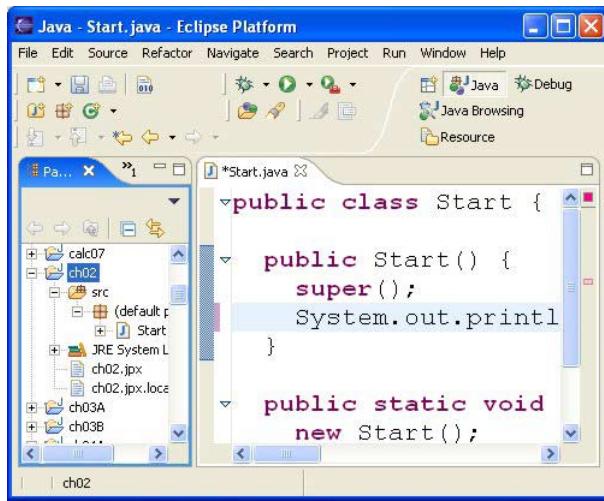
POGLAVLJE 2

RAZVOJNO OKRUŽENJE

U ovom poglavlju će biti objašnjeno osnovno korištenje JBuilder razvojnog okruženja.

Java se može snimiti besplatno sa web adrese <http://java.sun.com/>. Integrисано развојно окруžење као што је JBuilder (<http://www.borland.com/>) или Eclipse (<http://www.eclipse.org/>) се може користити за editovanje, компилiranje и покretanje кода. Personalna (Foundation) verzija JBuilder-a i Eclipse su dostupni бесплатно. Mnogo vrlo dobrih objašњења о Eclipse se може наћи на <http://eclipse-tutorial.dev.java.net/>.





Pripremanje aplikacionog okruženja

Kako bismo implementirali aplikaciju prezentiranu u Poglavlju 1, prvo morate pripremiti aplikaciono okruženje.

Direktorij

Kako bismo grupisali sve primjere ove knjige na jednu lokaciju, preporučujemo kreiranje direktorija *magicBoxes* u kojem ćete držati Java kôd. A zatim ćete za svako poglavlje kreirati odgovarajući direktorij. Finalni rezultat je sljedeći:

Prema standardima, imena direktorija počinju sa **malim početnim** slovom.

- *magicBoxes*
 - *code*
 - *ch01*
 - *ch02*
 - *ch03A*
 - *ch03B*
 - ...

Primijetite da ime direktorija *ch01* određuje programski kôd Poglavlja 1.

Projekat

Onda kada je struktura direktorija uspostavljena (prije početka razvoja Java klase) kreira se ”projekat” aplikacije. U Poglavlju 2 korištena projektna datoteka će se zvati *ch02.jpx*.

Projekat:

Dozvoljava čuvanje opcija kao i referenci na aplikacione datoteke.

Koncept projekta dozvoljava grupisanje i čuvanje različitih datoteka kao i opcija koje se koriste u aplikaciji pod projektnim imenom.

Projekat kreiramo kada kliknemo na meni *File* lociran na vrhu prvog prozora JBuilder razvojnog okruženja, a zatim kliknemo na opciju menija *New Project... :*

Polje *Name* dozvoljava da dodijelite ime projektu koji smo definisali (ch02). Ukoliko kliknete na dugme ..., možete odrediti putanju do vašeg direktorija, a polje *Directory* dozvoljava specifikaciju lokacije projektne datoteke.

Onda kada se definiše projektno ime i putanja direktorija ostaje da se uradi definicija putanje direktorija u kojem ćeće čuvati kôd (*.java) i kompilirane bajt datoteke (*.class).

Možete vidjeti novi sadržaj *ch02* direktorija kada se završi kreiranje projekta:

```
- magicBoxes
  - code
    - ch02
      ch02.jpx
```

Uz projektnu datoteku *ch02.jpx*, JBuilder može kreirati i *ch02.html* datoteku. Ova datoteka dozvoljava prezentiranje projektnih informacija u HTML formatu.

Osnovne datoteke

Pored projektnih informacija sačuvanih u datoteci sa ekstenzijom ”.jpx” Java kôd mora biti sačuvan u osnovnim datotekama. U Poglavlju 1 koristili smo datoteku pod imenom *Start.java*. Ekstenzija ”.java” govori kakvog je tipa datoteka sa kojom radimo.

Sljedeći koraci će kreirati osnovnu datoteku :

- 1- Selekirajte meni *File* a zatim kliknite na opciju *New...*;
- 2- U prozoru koji se pojavio, kliknite na polje ”*General*”, selektujte ”*Class*” ikonu a zatim kliknite na ”*OK*”;
- 3- Dodajte ime klase u polju ”*Class Name*”. U našem primjeru ime klase je *Start*. Nakon selektovanja opcija ”*Public*” i ”*Generate default constructor*”, JBuilder automatski generiše sljedeći Java kôd:

```
public class Start {
    public Start() {
    }
}
```

Preostaje dodavanje odgovarajućeg Java kôda kao što ćemo prezentirati u primjeru Dijela A Poglavlja 3.

Kompilacija klase

Kompilirana klasa:
<Ime klase>.class

Kada se klasa kompilira rezultirajući ”bajt kôd” se čuva u datoteci sa istim imenom kao i klasa ali sa ekstenzijom ”.class”. U našem prijašnjem primjeru osnovna datoteka *Start.java* je služila za kreiranje druge datoteke pod istim nazivom *Start.class* ali drugom ekstenzijom.

Kako bi održavanje datoteka bilo lakše, strogo preporučujemo da fizički odvojite datoteke tipa ”.class” od osnovnih datoteka ”.java”. Kako bismo ovo uradili, kreirali smo direktorij *classes* za kompilirane datoteke i direktorij *src* za osnovne datoteke. Naš primjer ima sljedeću direktorijsku strukturu:

```
- magicBoxes
  - code
    - ch02
      - classes
      - src
```

Primijetite da su poddirektoriji direktorija *classes* automatski kreirani od strane JBuilder-a tokom kompilacije. Imena ovih poddirektorija odgovaraju onima koji se nalaze u direktoriju *src*.

Osobine projekta

Opcija menija *Project properties...* u meniju *Project* dozvoljava definisanje osobina projekta. Kada izaberete ovu opciju, JBuilder prezentira mnogobrojne osobine startujući sa poljem *Paths*:

Sljedeća putanja,

```
... \magicBoxes\code\ch02\src
```

dodata u polje *Source* mora odgovarati lokaciji osnovnih datoteka, kao što je npr. *Start.java*.

Primjer putanje u polju *Output path*,

```
... \magicBoxes\code\ch02\classes
```

odgovara pravoj lokaciji datoteka tipa ”class”, kao što je to npr. *Start.class*.

Funkcije dugmadi ”...” i ”Edit” dozvoljavaju veoma lagano modifikovanje navedene direktorijske putanje.

U istom prozoru gdje su specificirane osobine projekta polje *Run* dozvoljava označavanje početne klase aplikacije (klase u kojoj se nalazi metoda *main*).

Specifikacija navedene klase dozvoljava JBuilder-u da tokom pokretanja aplikacije zna koju metodu da izvrši. Ukoliko ova osobina nije podešena, JBuilder neće pokrenuti aplikaciju i mi moramo selektovati klasu koja ima definisan *main* metodu.

Primijetite da pokretanje aplikacije može biti izvedeno na tri različita načina:

- Klikanjem na "Run" dugme u prvoj sekciji JBuilder-a;
- Korištenjem menija Run i selektovanjem opcije "Run Project";
- Pritisom na funkcionalnu tipku F9.

Paket

Paket: kolekcija klasa

Razvijanje Java aplikacije zahtijeva određeno grupisanje klasa. Koncept paketa (*package*) nam dozvoljava da grupišemo klase.

Na primjer, želimo da grupišemo određene klase u paket *mb.view* a neke druge klase u paket *mb.model*. U konkretnim terminima imena paketa odgovaraju putanji direktorija koji se nalaze u *src* direktoriju (prema tome i *classes* direktoriju).

Ključna riječ *package* dolazi prije putanja direktorija u kojem se nalaze osnovne datoteke. U našem primjeru (Dio A, Poglavlje 3), morate uključiti sljedeću instrukciju prije definicije klase:

```
package mb;
```

Preciznije, morate imati sljedeće Java instrukcije:

```
package mb;  
...  
public class App {  
    .  
    .  
    .  
}
```

package instrukcija pravi referencu na jedan ili više direktorija.

Ovaj zadnji primjer određuje egzistenciju klase *App* (*App.java*) u direktoriju *mb*.

Biblioteke predefinisanih komponenti

Da bi se olakšalo razvijanje aplikacija, kompanija kao što je Sun, predefiniše biblioteke (pakete) komponenata. Na primjer, Sun je definisao kolekciju komponenta za grafički korisnički interfejs nazvanu *Swing* i lociranu u paketu *javax.swing*.

Eksterne Klase

Uključiti klasu sa neke druge lokacije:

```
import
```

Kada želimo da uključimo klasu koja pripada nekom drugom paketu, moramo se eksplisitno pozvati na tu željenu klasu. Instrukcija *import*, kao što je prikazano u narednom primjeru, dozvoljava upravo da to postignemo:

```
import javax.swing.JFrame;
```

U ovom primjeru klasa *JFrame* je uključena iz paketa *javax.swing*.

Takođe primijetite da je paket *java.lang* uvijek uključen čak kad ni jedna *import* instrukcija nije specificirana.

POGLAVLJE 3

OKVIR ILI PROZOR

Aplikacioni kontekst određuje okvir (ili prozor) aplikacije ili apleta. Rezultat postignut u prvom dijelu (Dio A) ovog poglavlja je aplikacioni okvir koji neće sadržati nijednu grafičku komponentu.



U drugom dijelu ovog poglavlja (Dio B) prvo ćemo promjeniti dva stanja aplikacionog okvira zatim ćemo dodati naslov aplikaciji i finalno odrediti tačnu širinu i dužinu okvira.



Ovo poglavlje će prezentirati osnovne koncepte Java apleta. Dodaćemo Java kôd koji omogućava da se naša aplikacija koristi unutar Internet pretraživača.

DIO A:

APLIKACIJA

Eksterne klase

U prvom dijelu ovog poglavlja nastavljamo sa razvijanjem naše aplikacije iz Poglavlja 1. U narednom kôdu možete vidjeti da klasa *Start* sada poziva konstruktora nove klase nazvane *App*:

```
1: package mb; // +
2:
3: public class Start {
4:
5:     public Start() {
6:         super();
7:         // System.out.println("Magic Boxes"); // -
8:     }
9:
10:    public static void main(String[] args) {
11:        new App();
12:    }
13:
14: }
```

Nasljedivanje

Prva i jedina instrukcija unutar polazne tačke aplikacije poziva konstruktora *App* klase.

```
11: new App();
```

Kao što smo već napomenuli, svaki konstruktor (osim klase *Object*) uvijek poziva konstruktor roditeljske klase ključnom riječi *super*.

Prema tome, prva instrukcija konstruktora klase *App*

```
25: public App() {
```

```
26:     super() ;  
27:     appFrame = new JFrame() ;  
28:     appFrame.setVisible(true) ;  
29: }
```

eksplicitno poziva konstruktora roditeljske klase. Ova veza dozvoljava kopiranje atributa i korištenje javnih metoda klase *Object* u kreiranom objektu.

Okvir

Cilj klase *App* je da definiše okvir naše aplikacije. Slijedi kôd klase *App*:

```
15: // {+}  
16:  
17: package mb;  
18:  
19: import javax.swing.JFrame;  
20:  
21: public class App {  
22:  
23:     private JFrame appFrame;  
24:  
25:     public App() {  
26:         super();  
27:         appFrame = new JFrame();  
28:         appFrame.setVisible(true);  
29:     }  
30:  
31: }
```

Instrukcija na liniji 17

```
17: package mb;
```

definiše lokaciju datoteke klase *App* u paketu *mb*.

Lokacija svih korištenih klasa izvan "paketa" mora biti eksplisitno navedena pomoću instrukcije *import*.

Druga instrukcija klase *App* referencira, preko sljedeće *import* instrukcije,

```
19: import javax.swing.JFrame;
```

klasu *JFrame* koju ćemo trebati radi definisanja atributa *appFrame* na liniji 23. Klasa *JFrame* je dio paketa *javax.swing*. Konkretno, kôd klase *JFrame*

me nije dio aplikacionog kôda kao takvog. Korištenje metoda koje su implementirane u eksternim klasama, kao što je *JFrame*, značajno olakšava razvijanje aplikacije.

Tipovi podataka i atributi klase

Tipovi podataka =

Definicija podataka

Kao u svim programskim jezicima, Java koristi "osnovne" tipove podataka kako bi reprezentovala format i strukturu varijabli koje će biti korištene. Ovi tipovi podataka su grupisani u sljedeće kategorije:

Kategorija	Java osnovni tipovi
4- Prirodni brojevi	byte, int, long, short
5- Realni brojevi	double, float,
6- Karakter	char
7- Boolean	boolean

Na primjer, definicija eksterne metode *setVisible* (korištene na liniji 28)

```
public void setVisible(boolean b)
```

definiše parametar *b* koji odgovara tipu podatka boolean. Za ovaj tip dozvoljene su samo dvije vrijednosti: *true* i *false*. Parametar *b* se inicijalizira na vrijednost *true* kada se metoda *setVisible* prvi put pozove:

```
28: appFrame.setVisible(true);
```

Sljedeća linija

```
23: private JFrame appFrame;
```

Swing: Java paket koji definiše grafički korisnički interfejs.

definiše atribut *appFrame* tipa *JFrame*. U ovom primjeru, tip podatka odgovara predefinisanoj Java klasi *JFrame* iz paketa *javax.swing*.

Osnovni tip:

direktni pristup vrijednosti atributa.

Tip klase:

pristup referenci na objekat.

Sa osnovnim tipom podatka imate direktni pristup vrijednosti varijable. Sa tipom klase imate pristup referenci na objekat.

Atribut *appFrame*, na liniji 23, je definisan u toku izvršavanja konstruktora klase *App*. Ovaj atribut je tipa *JFrame*. Na liniji 23, *appFrame* ne referencira objekat tako da je njegova vrijednost *null*. Na liniji 27, ta se vrijednost mijenja sa *null* na kreirani objekat:

```
27: appFrame = new JFrame();
```

Nezavisno nasljeđivanje u hijerarhiji

Objekat *appFrame* posjeduje nasljeđivanje koje je različito i nezavisno od klase kojoj pripada (*App*) .

AWT : Java alati korišteni za razvijanje višeplatformskog grafičkog interfejsa.

Klasa *JFrame* nasljeđuje klasu *Frame* kao i sve ostale klase u hijerarhiji nasljeđivanja (*JFrame* → *Frame* → *Window* → *Container* → *Component* → *Object*). *Frame* klasa je dio Java AWT-a (Abstract Window Toolkit). AWT predstavlja skup alata koji dozvoljavaju razvijanje grafičkog korisničkog interfejsa potpuno neovisno od platforme.

Modifikator *private*

Ključna riječ *private* korištena u definiciji atributa *appFrame*

```
23: private JFrame appFrame;
```

je modifikator koji vrši restrikciju pristupa atributu *appFrame*.

Na primjer, ukoliko bismo promijenili modifikator konstruktora klase *App* na *private* klasa *Start* ne bi mogla pozvati konstruktor na liniji 11. U tom slučaju desila bi se kompilacijska greška indicirajući da konstruktor klase *App* ima privatni pristup.

Prikazivanje okvira aplikacije

Nakon definicije okvira *appFrame*, na liniji 27, ostaje da se isti prikaže. Sljedeća linija radi upravo to:

```
28: appFrame.setVisible(true);
```

Ova naizgled jednostavna instrukcija sadrži nekoliko veoma važnih notacija.

Pozivanje metode objekta

Linija 28 je bazirana na konceptu korištenja metode objekta. Generalna sintaksa relacije između objekta i metode je prikazana na sljedećoj liniji:

```
<ime objekta>.<ime metode>([parametri])
```

U našem primjeru odmah možemo primijetiti da metoda *setVisible* ne pripada ni klasi *Start* ni klasi *App*.

Integrисano razvojno okruženje često obezbjeđuje traženje pored ostalog i metoda koje nam nisu poznate. Da bismo našli metodu *setVisible* unutar JBuilder razvojnog okruženja kliknite na meni *Help* a zatim na *Help Topics*.

Osobine

Osobina dozvoljava pohranjivanje specifičnog stanja objekta.

Osobina je sačuvano stanje specifične vrijednosti objekta. U našem primjeru korištenja metode *setVisible*, osobina je asocirana sa objektom *appFrame* kako bi se odredilo da li objekat treba prikazati na korisničkom ekranu. Ovo stanje je sačuvano u privatnom atributu nazvanom *visible*. Ovaj atribut sadrži boolean vrijednost *true* odnosno *false*.

Osobina je privatni atribut kojim se pristupa specijalnim metodama čije ime počinje sa *"is"*, *"get"* ili *"set"*.

Kako bi se modifikovala vrijednost osobine objekta koriste se metode za čitanje i pisanje. Ove metode počinju sa prefiksima *"is"*, *"get"* ili *"set"*. Ime atributa se koristi kao sufiks u imenu metode i počinje sa velikim početnim slovom.

Prema navedenim pravilima, možemo primijetiti da metoda *setVisible* dozvoljava programeru mijenjanje osobine *visible* objekta *appFrame*.

Takođe primijetite da *"set"* metoda mora imati modifikator *void*. Vrijednost koja dozvoljava modifikaciju osobine objekta je parametar metode.

DIO B:

APLET

Eliminisanje klase *Start*

U prethodnoj sekciji smo koristili dvije klase za definisanje okvira aplikacije. Definisanje *main* metode u klasi *Start* je poslužilo kao početna tačka aplikacije. Instrukcija na liniji 11 je pozvala konstruktor klase *App*.

Primijetite da konstruktor klase *Start* nije dio aplikacije. Takođe klasa *Start* se koristila isključivo za pokretanje aplikacije. Konsekventno, odlučujemo da eliminišemo klasu *Start* i uključimo sve njene funkcije u klasu *App*. Sada nova klasa *App* služi i kao početna tačka aplikacije:

```
1: package mb;
2:
3: import javax.swing.JApplet; // +
4: import javax.swing.JFrame;
5:
6: public class App extends JApplet { // &
7:
8:     private JFrame appFrame;
9:
10:    public App() {
11:        super();
12:        appFrame = new JFrame();
13:        appFrame.setTitle("Empty window"); // +
14:        appFrame.setSize(400, 300); // +
15:        // appFrame.setVisible(true); //-
16:    }
17:
18:    // {+}
19:    // Start.java // -
20:
21:    public void init() {
```

```

22:         appFrame.setLocation(80, 80);
23:     }
24:
25:     public void start() {
26:         appFrame.setVisible(true);
27:     }
28:
29:     public void stop() {
30:         appFrame.setVisible(false);
31:     }
32:
33:     public void destroy() {
34:         appFrame.dispose();
35:     }
36:
37:     public static void main(String[] args) {
38:         App app = new App();
39:         app.init();
40:         app.start();
41:     }
42:
43: }
```

Vidimo pored ostalog da *main* metoda (linije od 37 do 41) kreira, inicijalizira i pokreće aplikaciju.

Instrukcija *super* na liniji 11 sada poziva konstruktora klase *JApplet* umjesto konstruktora klase *Object* kao što je to pokazano u Dijelu A ovog poglavlja.

Nasljedivanje

Kako bismo mogli pokrenuti aplikaciju unutar Internet pretraživača, povezujemo klasu koja pokreće našu aplikaciju, sa klasom *JApplet*.

Klasa *App* je sada integralni dio sljedeće strukture nasljedivanja: *App* → *JApplet* → *Applet* → *Panel* → *Container* → *Component* → *Object*.

Ova nova uspostavljena veza nasljedivanja između klase *App* i klase *JApplet* je definisana ključnom riječi *extends* na početku klase *App*:

```
6: public class App extends JApplet {
```

Termin *extends* definiše hijerarhijsku relaciju nasljedivanja između dječije klase i roditeljske klase.

Navedena ključna riječ definiše hijerarhijsku relaciju nasljeđivanja između dvije klase. Ova veza je uspostavljena sa imenom klase koja je specificirana nakon ključne riječi *extends*. Prema tome na liniji 6 *JApplet* klasa se naziva roditeljska klasa (ili superklasa) dok se klasa *App* naziva dječija klasa (ili podklasa). Takođe kažemo da klasa *App* nasljeđuje klasu *JApplet*.

Pored mogućnosti pristupa vlastitim osobinama i metodama klasa *App* može koristiti osobine i metode klase *JApplet*, kao i one koji se mogu naći u hijerarhiji iznad. Prema tome klasa *JApplet* nasljeđuje klasu *Applet*. Ovaj mehanizam nasljeđivanja se uspostavlja sve do klase *Object*.

Komentari

Kao i ostali programski jezici, Java dozvoljava dodavanje komentara određenim dijelovima programa koji zahtijevaju posebnu pažnju. Kao rezultat biće olakšano drugoj osobi da bolje razumije kôd.

Dodavanje komentara se može uraditi na jedan od dva načina:

```
// Komentar u jednoj liniji
```

ili

```
/* Prva linija komentara  
 Druga linija komentara  
 .  
 .  
 .  
 Zadnja linija komentara */
```

Komentar se može naći u produžetku specifične instrukcije Java kôda, kao što je prikazano na liniji 13:

```
13: appFrame.setTitle("Empty window"); // +
```

Lanac karaktera “//”, “/*” ili “/**” indicira da informacija koja slijedi služi za dokumentaciju. Primijetite da se karakteri “/**” mogu zamjeniti sa “/*”.

Konvencije

Ova knjiga koristi iterativni razvoj aplikacije kroz spirale. Počeli smo sa malom Java aplikacijom koja je veoma jednostavna i zatim smo postepeno dodavali ili

modificirali odgovarajući Java kôd. Kako bismo bolje prezentirali modifikacije urađene u ovom poglavlju, ili poredili trenutno poglavlje sa prethodnim, koristimo sljedeće komentare ili konvencije:

// + Dodavanje instrukcije

Na primjer:

```
13: appFrame.setTitle("Empty window"); // +
14: appFrame.setSize(400, 300); // +
```

// & Modifikacija instrukcije

// - Brisanje instrukcije

Na primjer:

```
19: // Start.java // -
```

Ovaj komentar označava da je klasa *Start* izbrisana iz aplikacije. Naravno kako Java ne bi interpretirala ono što je pronađeno lijevo od karaktera “// -” obavezno je dodati karaktere “//” na početku linije kôda.

Modifikacije nad blokom instrukcija

Kada se dodaje blok instrukcija, koristićemo sljedeću konvenciju:

// {+

Dodane Java instrukcije

// +}

Ovaj način izbjegava ubacivanje karaktera “// +” na svaku dodatu liniju kôda.

Ista stvar se odnosi i na blok instrukcija koji zahtijeva modifikaciju. Konvencija koja će se koristiti u ovoj situaciji je sljedeća:

// {&

Modificirane Java instrukcije

// &}

Za razliku od dodavanja i modificiranja instrukcija, brisanje će koristiti sljedeću konvenciju:

```
/* {-  
Obrisane Java instrukcije  
-} */
```

Dodavanje kôda na kraj klase ili dodavanje nove klase koristi sljedeću konvenciju:

```
// {+}      Dodavanje novih linija Java kôda na kraj  
klase
```

Na primjer:

```
18: // {+}
```

Kada su karakteri “// {+}” stavljeni na prvu liniju iznad definicije klase komentar označava da smo dodali novu klasu u našu aplikaciju.

Modifikacija stanja objekta

Osobina *title*

Dvije instrukcije su dodata u konstruktor klase *App*. Linija 13

```
13: appFrame.setTitle("Empty window"); // +
```

specificira naslov aplikacije. Ova instrukcija modifica osobinu *title* koja je definisana u klasi *JFrame*. Pošto je ova klasa dio hijerarhije nasljeđivanja objekta *appFrame*, dozvoljeno je korištenje ove osobine.

Osobina *size*

Ista napomena važi i za liniju 14. Ovdje je osobina *size* definisana u klasi *Component* i prema tome metoda koja je asocirana sa ovom osobinom je naslijeđena iz klase *Component*. Metoda *setSize* dozvoljava redefinisanje širine i visine aplikacionog okvira. Tokom konstrukcije objekta okvira kao što je *appFrame*, širina i visina okvira imaju vrijednost nula. Tada se vidi samo naslov okvira.

Kada se koristi metoda `setSize`, preko parametara se prenose vrijednosti koje odgovaraju broju piksela. Širina naslova aplikacije se automatski prilagođava novoj širini objekta.

Primijetite da se dva prirodna broja šalju kao parametri metode `setSize`. Pošto postoje dvije metode `setSize` u klasi `Component`:

```
public void setSize(Dimension d)  
  
public void setSize(int width, int height)
```

parametri određuju koja će metoda biti pozvana. U našem primjeru je to druga metoda.

Pozivanjem metode `setSize` na liniji 14

```
14: appFrame.setSize(400, 300); // +
```

predajemo vrijednosti 400 i 300 kao parametre metodi `setSize`. Ova metoda modifcira osobinu `size` koja je tipa `Dimension`. Konkretno osobina `size` je određena sa:

- `width`: širinom okvira;
- `height`: visinom okvira.

Internet pretraživač i Java radno okruženje

Generalni pregled

Aplet: kolekcija povezanih Java klasa koje može izvršavati Internet pretraživač.

Generalno govoreći, možemo definisati Java aplet kao kolekciju povezanih klasa koje može izvršavati Internet pretraživač. Aplet se pokreće od strane HTML stranice. Ime "početne" klase apleta se onda specificira na toj stranici.

Aplikacija: kolekcija povezanih Java klasa koje može izvršavati korisnički računar.

Razlika između apleta i aplikacije je samo u "početnoj" klasi. Primijetite da je termin aplikacija u ovoj knjizi definisan kao kolekcija povezanih klasa koje dozvoljavaju izvršavanje serije instrukcija na korisnikovom računaru.

Kako bi prepoznao i izvršio instrukcije koje čine Java aplet, Internet pretraživač mora biti opremljen sa Java radnim okruženjem (JRE). JRE je dio Java standarnog skupa razvojnih alata (SDK).

Aplet / Server

Za početak, pretpostavimo da se aplet nalazi na serveru koji je povezan sa Internetom. Kada se HTML stranica zatraži od korisnika, pretraživač traži od servera datoteku u formi zahtjeva. Ukoliko je datoteka nađena ista se učita i izvršava unutar pretraživača.

Kada HTML stranica zatraži Java aplet, desi se drugi zahtjev. Ovaj put se učitaju sve ”*byte code*” datoteke koje su dio samog apleta. Ove datoteke se nalaze na istom serveru gdje je prethodno učitana HTML stranica.

Pošto učitavanje apleta može obuhvatiti nekoliko klasa vrijeme potrebno za učitavanje može biti relativno dugo. Kako bi se ubrzao proces učitavanja sve ”*.class*” datoteke mogu biti kompresovane u jednu datoteku nazvanu *jar* datoteka. Ta *jar* datoteka se pohranjuje na aplikacionom serveru.

HTML dodaci

Kao što je već napomenuto, datoteka *start.html*, locirana u *classes* direktoriju aplikacije, se učitava kao početna web stranica. Jednom učitana stranica će pozvati učitavanje i izvršavanje apleta.

U datoteci *start.html* specijalne sekcije su dodate za ispravno funkcioniranje dva glavna pretraživača na tržištu.

```
57: <OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-  
00805F499D93"  
58: width="400"  
59: height="30"  
60: align="baseline"  
61: codebase="http://java.sun.com/j2se/1.4.2/  
download.html">  
62: <PARAM NAME="code" VALUE="mb.App.class">  
63: <PARAM NAME="codebase" VALUE=".">   
64: <PARAM NAME="archive" VALUE="mb.jar">  
65: <PARAM NAME="type" VALUE="application/x-java-  
applet;version=1.4">  
66:
```

```

67:      <COMMENT>
68:          <EMBED
69:              type="application/x-java-
applet;version=1.4"
70:                  width="400"
71:                  height="30"
72:                  align="baseline"
73:                  code="mb.App.class"
74:                  codebase=". "
75:                  archive="mb.jar"
76:                  pluginspage="http://java.sun.com/
j2se/1.4.2/download.html">
77:          </EMBED>
78:      </COMMENT>
79:
80:  </OBJECT>

```

Sve instrukcije unutar *EMBED* bloka (linije od 68 do 77) nisu prepoznate od Explorer-a, dok ih Netscape prepoznaje. Obrnuto, Netscape ne prepoznaje *OBJECT* i *COMMENT*.

Ove sekcije (linije od 57 do 80) su specijalno namijenjene za dodatne funkcionalnosti koje nisu dostupne osnovnim pretraživačima. Emuliranje aplikacione sposobnosti koje pretraživač ne može izvesti je omogućeno pomoću specijalnog dodatka aplikaciji nazvanog ”*plugin*”.

U prezentiranom primjeru, ukoliko to već niste uradili, morate dodati JRE vašem pretraživaču. Pretraživač onda verificira da ima taj *plugin*. Ukoliko ga nema, pitaće korisnika da učita *plugin* sa adresu specificirane na liniji 61 ili 76. Adresa dozvoljava korisniku da snimi Java Runtime Environment ili Java SDK sa Sun-ovog Java servera.

Parametar *type* na liniji 65 i 69 označava da želimo da radimo sa JRE ili SDK u verziji 1.4.

Na linijama 64 i 75 parametar *archive* optionalno indicira ime *jar* datoteke. Ukoliko se ne bi koristila ta opcija sve ”*.class*” datoteke moraju biti smještene na server.

Uz optionalnu *jar* datoteku server mora imati i osnovnu HTML stranicu. Na primjer, server može imati sljedeći direktorij sa datotekama:

- magicBoxes
 - mb.jar
 - start.html

Lokacija klase koja pokreće izvršavanje apleta je navedena u HTML stranici.

Kao što smo već naveli, aplet kao i standardna Java aplikacija ima klasu koja služi za pokretanje (tzv. "startup" klasa). Ta klasa se specificira u HTML stranici kao što je to prikazano u narednim linijama kôda:

```
62: <PARAM NAME="code" VALUE="mb.App.class">
```

ili

```
73: code="mb.App.class"
```

Ovaj dio HTML kôda indicira ime klase Java apleta. U ovom primjeru to je klasa *App* unutar paketa *mb*.

Osobine *width* i *height* na linijama 58, 59, 70 i 71 definišu dužinu i visinu prostora koji se koristi od strane apleta unutar HTML stranice. U našem primjeru se koristi fiktivan prostor radi poruke koja indicira učitavanje apleta sa servera.

Pristup podacima na drugom računaru

Prema određenim sigurnosnim dozvolama, Java može dozvoliti apletu da čita (i čuva) datoteke sa korisničkog računara.

Drugi važan aspekt je u vezi sa pristupom apleta podacima na korisničkom računaru. Sigurnosna ograničenja ne dozvoljavaju apletu da posjeduje neograničen pristup datotekama. Čak iako Java dozvoljava prema određenim sigurnosnim dozvolama čitanje i čuvanje datoteka na računaru mi moramo naglasiti da aplet generalno nema pristup datotekama na korisničkom računaru.

Konkretno, izvršava se verifikacija "byte code" datoteka prije učitavanja na računar, kako bi se moglo provjeriti da li je ijedna Java konvencija prekršena. Na primjer, provjerava se da li kôd pristupa memorijskim lokacijama koje pripadaju drugim aplikacijama ili datotekama.

Norme izvršavanja *Apleta*

Mnoge funkcije apleta su sadržane u klasi *Applet*. Zahtijeva se da klasa koja nasljeđuje klasu *JApplet* mora definisati metode *init*, *start*, *stop* i *destroy* koje su predefinisane u klasi *Applet*. Ove metode se moraju ponovo napisati, odnosno redefinisati, prema tim zahtjevima.

Poredak izvršavanja metoda apleta je sljedeći:

- 1- Konstruktor klase koja pokreće aplet;
- 2- Metoda *init*;
- 3- Metoda *start*;
- 4- Metoda *stop*;
- 5- Metoda *destroy*.

Čim se Java aplet učita, izvršava se konstruktor klase koja pokreće aplet. U našem primjeru to je *App* konstruktor:

```
10: public App() {  
11:     super();  
12:     appFrame = new JFrame();  
13:     appFrame.setTitle("Empty window"); // +  
14:     appFrame.setSize(400, 300); // +  
15:     // appFrame.setVisible(true); // -  
16: }
```

Primijetite da se sljedeća *main* metoda

```
37: public static void main(String[] args) {  
38:     App app = new App();  
39:     app.init();  
40:     app.start();  
41: }
```

ne koristi u izvršavanju apleta zbog toga što JRE, uključen u Internet pretraživač, automatski poziva konstruktor klase koja pokreće aplet. Međutim, kako bi takođe mogli pokrenuti aplet kao aplikaciju, preporučujemo da uključite metodu *main* kao što je to urađeno u *App* klasi.

Metoda *init*.

Onda kada izvršavanje konstruktora prestane poziva se *init* metoda. To je funkcija koja sadrži određene inicijalizacije kao što su redefinisanje veličine grafičkog interfejsa, kreiranje određenih instanci, inicijalizacija određenih parametara, učitavanje slika, i tako dalje. U našem primjeru okvir aplikacije se pozicionira unutar metode *init*:

```
21: public void init() {  
22:     appFrame.setLocation(80, 80);  
23: }
```

Primijetite da programer uvijek mora uključiti ovu metodu u klasu koja pokreće aplet čak iako metoda nema ni jednu instrukciju. Ista primjedba se odnosi i na metode *start*, *stop* i *destroy*.

Metoda *start*.

Nakon što se završila metoda *init*, pretraživač pokreće metodu *start*. U našem primjeru metoda *start* izgleda ovako:

```
25: public void start() {  
26:     appFrame.setVisible(true);  
27: }
```

Treba primijetiti da metoda *start* može biti pozvana drugi put kada pretraživač restartuje suspendovan aplet nakon izvršene metode *stop*. Prestanak izvršavanja apleta se može desiti kada korisnik klikne na drugi link i napusti web stranicu. Ukoliko se korisnik vrati na stranicu, ponovo se poziva *start* metoda. U pretvodno opisanom scenariju se jasno vidi razlika između metode *init* i *start*. Metoda *init* se poziva samo jednom u toku "životnog vijeka" apleta, dok se metoda *start* može pozivati više puta.

U našem primjeru, ova metoda služi za restartovanje metode *setVisible* dozvoljavajući (ponovno) prikazivanje aplikacionog okvira.

Metoda *stop*.

Kao što je već napomenuto kada korisnik napusti web stranicu koja sadrži aplet, pretraživač suspenduje aplet pozivajući metodu *stop*. Programer mora uključiti metodu *stop* u klasi koja pokreće aplet. U našem primjeru metoda *stop* izgleda ovako:

```
29: public void stop() {  
30:     appFrame.setVisible(false);  
31: }
```

Metoda *destroy*.

Konačno, kada aplet više nije potreban, poziva se metoda *destroy* koja oslobođava korištene resurse.

Generalno, aplet više nije potreban kada korisnik klikne na "Back" dugme pretraživača ili kada zatvori pretraživač (X dugme desno u vrhu prozora ili opcija *Close* odnosno *Exit* u *File* meniju).

Bez uloženja u detalje, veoma je važno osloboditi resurse operativnog sistema korisničkog računara. Puno zauzetih resursa mogu napraviti probleme računaru kao i krajnjem korisniku.

U našem primjeru koristimo sljedeću *destroy* metodu:

```
33: public void destroy() {  
34:     appFrame.dispose();  
35: }
```

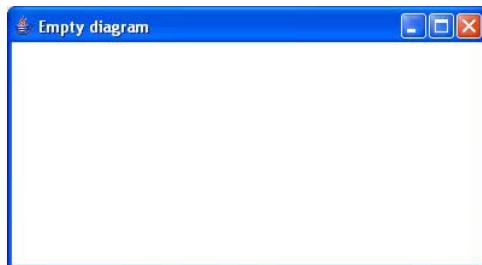
Metoda *dispose* na liniji 34 oslobađa sve resurse operativnog sistema zauzete od strane objekta *appFrame*.

POGLAVLJE 4

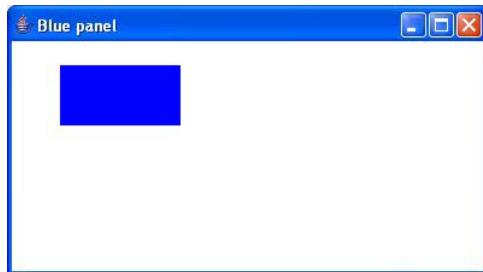
PANEL

OSNOVNI OKVIR APLIKACIJE MOŽE SADRŽAVATI RAZLIČITE KOMPONENTE I KONTEJNERE. KONTEJNER OPET MOŽE SADRŽAVATI KOMPONENTE I KONTEJNERE. JEDAN OD OVIH KONTEJNERA JE PANEL. GENERALNO, PANELI DOZVOLJAVAJU DA DEFINIŠETE ODREĐENA APLIKACIONA GRAFIČKA PODRUČJA U KOJA MOŽETE DODAVATI OBJEKTE PREMA VAŠIM POTREBAMA USVOJENIM U FAZI DIZAJNA.

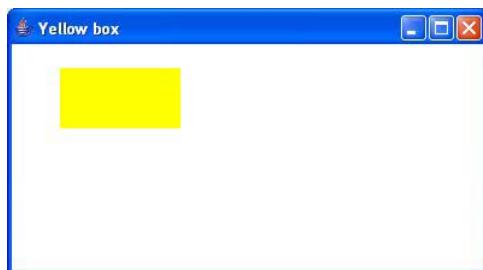
Prvi dio ovog poglavlja (Dio A) pokazuje na koji način možete kreirati prazan panel. U primjeru ćemo detaljno vidjeti kako se dio aplikacionog okvira može prekriti sa panelom bijele boje.



U drugoj sekciji ovog poglavlja (Dio B) nastavljam razvoj grafičkog korisničkog interfejsa definisanjem panela sa plavom bojom smještenim unutar bijelog panela.



*Konačno, treći dio sekcije (Dio C) završava sa prezentacijom koncepta specijalizacije klase sa kreiranjem nove klase **Box**. Ova sekcija takođe pokazuje i kako kreirati vlastite osobine, u kontrastu sa prethodnim poglavlјem gdje smo koristili osobine koje pripadaju eksternim klasama definisanim od strane drugih programera.*



DIO A:

PRAZAN PANEL

Nasljeđivanje

Kao što možemo primijetiti, klasa *App* prezentirana u ovoj sekciji

```
1: package mb;
2:
3: import javax.swing.JApplet;
4: // import javax.swing.JFrame; // -
5: import mb.view.AppFrame; // +
6:
7: public class App extends JApplet {
8:
9:     private AppFrame appFrame; // &
10:
11:    public App() {
12:        super();
13:        appFrame = new AppFrame(); // &
14:        // appFrame.setTitle("Empty window"); // -
15:        // appFrame.setSize(400, 300); // -
16:    }
17:
18:    public void init() {
19:        appFrame.setLocation(80, 80);
20:    }
21:
22:    public void start() {
23:        appFrame.setVisible(true);
24:    }
25:
26:    public void stop() {
```

```
27:     appFrame.setVisible(false);
28: }
29:
30: public void destroy() {
31:     appFrame.dispose();
32: }
33:
34: public static void main(String[] args) {
35:     App app = new App();
36:     app.init();
37:     app.start();
38: }
39:
40: }
```

je veoma slična onoj prezentiranoj u Dijelu B prethodnog poglavlja. Na liniji 9, umjesto definisanja objekta *appFrame* tipa *JFrame*, koristimo klasu tipa *AppFrame*:

```
41: // {+
42:
43: package mb.view;
44:
45: import java.awt.Color;
46: import java.awt.Container;
47: import javax.swing.JFrame;
48: import javax.swing.JPanel;
49:
50: public class AppFrame extends JFrame {
51:
52:     private JPanel diagram;
53:
54:     public AppFrame() {
55:         super();
56:         diagram = new JPanel();
57:         diagram.setBackground(Color.WHITE);
58:         this.setTitle("Empty Panel");
59:         this.setSize(400, 300);
60:         Container cp = this.getContentPane();
61:         cp.add(diagram);
62:     }
63:
64: }
```

Primijetite da klasa *App* sadrži instrukciju *import* kako bi mogla pristupiti klasi *AppFrame* jer ove klase ne pripadaju istom paketu.

Nova relacija je uspostavljena između klase *AppFrame* i *JFrame* ključnom riječi *extends* u potpisu klase *AppFrame*:

```
50: public class AppFrame extends JFrame
```

Kao što je već napomenuto, klasa *AppFrame* može pored pristupa vlastitim osobinama i metodama pristupiti na isti način osobinama i metodama koje su tvorci klase *JFrame* (kao i svim onima sadržanim u hijerarhiji iznad klase *JFrame*) koje smo omogućili nasljeđivanjem: *AppFrame* → *JFrame* → *Frame* → *Window* → *Container* → *Component* → *Object*.

Upozorenje: Riječi *appFrame* i *AppFrame* nemaju isto značenje.

Klasa *AppFrame* i objekat *appFrame* u našem primjeru predstavljaju konceptualno dvije različite stvari koje se ne smiju miješati jedna sa drugom.

this

Kao što smo vidjeli u Poglavlju 1, kako bismo koristili podatke i metode određenog objekta, moramo prvo kreirati taj objekat. U našem primjeru to radi konstruktor *AppFrame* (linija 54). Pošto objekat klase *AppFrame* nema imena unutar svoje klase, imamo problem u osnivanju relacije između trenutnog objekta i njegovih metoda.

this: trenutni objekat

Kako bismo riješili ovaj problem, koristimo ključnu riječ *this*. U našem primjeru, ime trenutnog objekta koji je nepoznat unutar definicije klase, se mijenja sa ključnom riječi *this*. Relacija između objekta i metode je ostvarena samo tokom izvršavanja instrukcija koje su u našem primjeru na linijama 58 i 59:

```
58:     this.setTitle("Empty diagram");  
59:     this.setSize(400, 300);
```

Specijalizacija klase

Grafički korisnički interfejs kao što je okvir je **kontejner**. Ovi kontejneri mogu "sadržavati" druge kontejnere kao što su paneli.

Okviri su kontejneri koji mogu sadržavati druge grafičke komponente. U prethodnom poglavlju smo definisali osnovni kontejner koji predstavlja aplikacioni okvir. Ovaj kontejner može sadržavati menije, slike kao i druge kontejnere kao što su paneli.

Dio B prethodnog poglavlja je definisao klasu *App* unutar paketa *mb*. Cilj ove klase je bio da prikaže prazan aplikacioni okvir (objekat *appFrame*). U ovoj sekciji ćemo koristiti klasu *AppFrame* da odvojimo definiciju objekta (klasa *AppFrame*) od njegovog korištenja (metode *setVisible* u klasi *App*). Na ovaj način želimo da specijaliziramo objekat *appFrame*, prethodno tipa *JFrame*. Od sada, njegov tip će odgovarati klasi *AppFrame* koja nasljeđuje klasu *JFrame*.

Definicija klase *AppFrame* uključuje kreiranje panela koji će biti dodat u okvir. U našem primjeru, linija 52

```
52: private JPanel diagram;
```

definiše atribut tipa *JPanel* sa imenom *diagram*.

Onda je sa linijom 56

```
56: diagram = new JPanel();
```

kreiran objekat tipa *JPanel*. Atribut *diagram* predstavlja njegovu referencu. Hijerarhija nasljeđivanja objekta *diagram* je sljedeća: *JPanel* → *JComponent* → *Container* → *Component* → *Object*.

Generalno se paneli koriste za grupisanje određenih grafičkih komponenata kao što su dugmad, polja za provjere, nazivi ili tekstualna polja. Paneli se mogu dodati unutar drugih kontejnera kao što je naš aplikacioni okvir.

Predefinisani paneli

Klasa *AppFrame* nasljeđuje klasu *JFrame*:

```
50: public class AppFrame extends JFrame
```

Interna arhitektura objekta *appFrame* je definisana na sljedeći način:

Objekat definisan klasom čija je roditeljska klasa *JFrame* nasljeđuje isključivo jedan kontejner. Ovaj kontejner je instanca klase *JRootPane*.

- objekat *appFrame* nasljeđuje klasu *JFrame*
- sastavljen je od jedinstvenog kontejnera definisanog klasom *JRootPane* koja je sastavljena od

- komponente *glassPane*
- komponente *layeredPane* koja je sastavljena od
- komponente [*menuBar*]
- i komponente *contentPane*

Objekat definisan klasom *JRootPane* odgovara panelu koji zauzima isti grafički prostor koji je zauzeo objekat *appFrame*, osim prostora koji je zauzet naslovom aplikacije.

Klasa *JRootPane* je sastavljana od dva objekta ili panel komponente: *glassPane* i *layeredPane*. U ovoj knjizi ne koristimo panel *glassPane*. Generalno, možemo reći da panel *glassPane* dozvoljava dodavanje specijalnih grafičkih efekata.

U našem primjeru, koristi se podkomponenta *layeredPane* u prostoru zauzettim objektom koji je definisan klasom *JRootPane*. Objekat *layeredPane* je ustvari kontejner organizovan na takav način da svaka dodata komponenta prekriva drugu koja je prethodno dodata. Na primjer, u narednim poglavljima ćemo dinamički dodati kutiju koristeći klik miša na dijagramu. Kada se doda druga kutija klikanjem na prostor dijagrama lijevo od prve kutije, dio prve kutije je prekriven drugom, novom kutijom.

Pošto se komponenta *MenuBar* u ovom momentu ne koristi u našem primjeru, komponenta *contentPane* prekriva sav prostor zauzet objektom *layeredPane*. Ukoliko bi se *MenuBar* koristio, našao bi se iznad objekta *contentPane*. Tada bi objekat *contentPane* okupirao ostali slobodan prostor. Primijetite da panel *contentPane* (kao i svaki panel) je ustvari objekat definisan karakteristikama klase *Container*.

Kako bismo poštivali pravila kreiranja asocirana sa klasom *JFrame*, sve grafičke komponente koje su dio objekta *layeredPane* moraju biti dodate objektu *contentPane*. Na primjer, kako bismo mogli dodati panel *diagram* okviru naše aplikacije (*appFrame*), koriste se instrukcije:

```
60: Container cp = this.getContentPane();
61: cp.add(diagram);
```

Na liniji 60 dobijamo referencu na objekat *cp*. Na liniji 61, panel *diagram* se dodaje panelu *cp*.

Generalno, uvijek bismo trebali dodavati grafičke komponente objektu *contentPane* umjesto objektu *appFrame* zato što objekat koji nasljeđuje klasu *JFrame* mora sadržati samo jednu dječiju komponentu koju je Java automatski definisala prema pravilima klase *JRootPane*. Suprotno tome, trebali bismo referencirati direktno objekat kao što je *appFrame* kada želimo modificirati njegove osobine kao vidljivost, naslov ili veličinu:

```
23: appFrame.setVisible(true);  
58: this.setTitle("Empty Panel");  
59: this.setSize(400, 300);
```

Ključna riječ *this* na linijama 58 i 59 referencira trenutni objekat tokom izvršavanja. U našem primjeru, pošto su ove instrukcije locirane u konstruktoru *AppFrame*, trenutni objekat je *appFrame* referenciran u klasi *App*:

```
9: private AppFrame appFrame; // &  
13: appFrame = new AppFrame(); // &
```

Konstante klase

Metoda *setBackground* na liniji 57

```
57: diagram.setBackground(Color.WHITE);
```

dodaje bijelu pozadinsku boju novom kreiranom panelu. U konkretnim terminima, ovo dodjeljivanje modificira stanje osobine *background* klase *Component*.

Modifikatori *final* i *static*

Sa modifikatorom *final*, koncept varijable je zamijenjen terminom **konstanta**. Ova definicija znači da se dodijeljena vrijednost ne može mijenjati.

Kada se u definiciji atributa koristi ključna riječ *final*, kažemo da se radi o konstanti, a ne o varijabli. Prema tome, vrijednost koju smo zadali konstanti će uvijek biti ista.

Primijetite da je vrijednost koja je data kao parametar metodi *setBackground* ustvari konstanta. Definicija atributa WHITE u klasi *Color* je:

```
public static final Color WHITE;
```

Sa ključnom riječi *static*, atribut postaje dio klase a ne dio objekta.

Ključna riječ *static* označava da konstanta WHITE neće biti asocirana sa svakim kreiranim objektom, nego će biti konstanta klase, jedinstvena za sve objekte koji će je koristiti. Generalno, referenca na konstantu koja pripada klasi mora zadovoljiti sljedeću sintaksu:

<Naziv klase>.<NAZIV KONSTANTE>

Prema uspostavljenim standardima, svi karakteri **konstante** moraju biti **velika slova**.

Primijetite da je konstanta u našem primjeru definisana velikim slovima (*WHITE*) kako bi se poštovao standard programskog jezika Java.

DIO B:

PLAVI PANEL U DIJAGRAMU

Specijalizacija klase (Prvi dio)

U prethodnim sekcijama, umjesto definisanja objekta *appFrame* tipa *JFrame* i uključivanja tog objekta u klasu *App*, odabrali smo da definišemo klasu *AppFrame* koja nasljeđuje klasu *JFrame*. Navedeno dozvoljava enkapsulaciju kôda objekta *appFrame*.

Objekat *appFrame* nije više potreban kao standardni objekat tipa *JFrame* već je sada objekat tipa *AppFrame* sa specifičnim definicijama sadržanim u klasi *AppFrame*. Ovaj specifični kôd dozvoljava definisanje atributa *diagram* i modifikaciju osobina *title* i *size*.

Ukoliko nastavimo našu analizu prezentiranog kôda, primijetićete da u klasi *AppFrame* dijela A ovog poglavlja, objekat *diagram*, tipa *JPanel*, takođe sadrži prilagođen kôd:

```
57: diagram.setBackground(Color.WHITE);
```

Ova instrukcija nam dozvoljava prilagođavanje objekta klase *JPanel*. Kako bismo mogli izvesti fleksibilnije prilagođenje, upoznaćemo u ovoj sekciji novu klasu *Diagram*:

```
64: package mb.model;
65:
66: import java.awt.Color;
67: import javax.swing.JPanel;
68:
69: public class Diagram extends JPanel {
70:
71:     private JPanel box;
72:
73:     public Diagram() {
74:         super();
75:         box = new JPanel();
76:         box.setBackground(Color.BLUE);
```

```
77:     box.setSize(100, 50);
78:     box.setLocation(40, 20);
79:     this.setBackground(Color.WHITE);
80:     this.setLayout(null);
81:     this.add(box);
82: }
83:
84: }
```

Linija 57, koja je prethodno prikazana, je pomjerena na liniju 79. Primijetite da će linije od 76 do 78 biti pomjerene (Dio C) u novu klasu, pošto ove linije sadrže prilagođavanja, ovaj put na nivou objekta *box*.

Podjela aplikacije u pakete

Dobra osobina razvijanja softvera je grupisanje klasa slične prirode. U našem primjeru smo grupisali klase koje služe za kreiranje grafičkog korisničkog interfejsa u paket *mb.view*. Prema tome, klasa *AppFrame* je dio ovog paketa. Međutim klasa *Diagram* pripada klasama koje služe za definisanje "modela" ili interne strukture grafičkih komponenti. Ove klase su smještene u paketu nazvanom *mb.model*.

Naredne instrukcije određuju u kojim paketima se nalaze navedene klase:

```
1: package mb;
38: package mb.view;
64: package mb.model;
```

Pošto klasa *AppFrame* koristi objekat *diagram* definisan u drugom paketu, neophodno je koristiti sljedeću import instrukciju:

```
44: import mb.model.Diagram; // +
```

za pristup klasi *Diagram*.

Takođe primijetite da su instrukcije *import* na linijama 45 i 48 prethodne sekciije pomjerene u klasu *Diagram*:

```
66: import java.awt.Color;
67: import javax.swing.JPanel;
```

Osobina *location*

Instrukcija na liniji 78

```
78: box.setLocation(40, 20);
```

koristi metodu *setLocation* kako bi se modifikovala vrijednost osobine *location*. Ova metoda modificira koordinate panela *box* u *x* = 40 i *y* = 20 od gornjeg lijevog ugla kontejnera (objekta *diagram*).

Koordinatori izgleda

Grafičke komponente koje obezbeđuje Java dozvoljavaju razvijanje grafičkog korisničkog interfejsa koji može raditi na bilo kojoj platformi. Ove komponente dozvoljavaju programeru, pored ostalog, da koristi koordinatore za izgled grafičkog korisničkog interfejsa koji su u mogućnosti da precizno postave grafičke komponente (na primjer dugmad, menije, tekstualna polja) u grafički kontejner. Kao što smo vidjeli, ovi kontejneri mogu biti paneli ili okviri.

Koordinatori izgleda:

Eliminišu potencijalne probleme prikazivanja grafičkih komponenata na različitim platformama.

Koordinatori izgleda su kreirani da se izbjegne različito prikazivanje grafičkih komponenti na različitim platformama. Njihov cilj je eliminisanje prepostavki o veličini kontejnera i komponenti. Davanje eksplisitnih koordinata komponenti nije preporučljivo.

Koordinator *BorderLayout*

Klasa *AppFrame* nasljeđuje klasu *JFrame*, koja koristi koordinator *BorderLayout*.

Vrlo jednostavan za korištenje, ovaj koordinator koristi pet regija u kojih je moguće smjestiti grafičke komponente. Ovih pet regija odgovara geografskim orijentirima: Istok, Zapad, Sjever, Jug, Centar.

Na primjer, unutar okvira, panel se može locirati u centralnu regiju, a dugmad u regiju iznad (Sjever) ili regiju ispod (Jug).

Komponente koje se koriste u **kontejneru** mogu koristiti **druge koordinatore izgleda** za razliku od onog koji koristi kontejner.

Ukoliko nastavimo sa primjerom malo detaljnije, moguće je da panel, smješten u centralnu regiju, koristi druge koordinatorne izgleda. Generalno, možemo reći da se nekoliko različitih koordinatora može koristiti za kreiranje kontejnera kao što je objekat *appFrame*.

Kada analiziramo različite instrukcije u našoj aplikaciji, nигде ne nalazimo eksplisitnu referencu na koordinator *BorderLayout*. Kada se referenca ne naveđe, Java odlučuje koji će se koordinator koristiti.

Kada je panel *diagram* ubačen u okvir sa sljedećom instrukcijom

```
57: cp.add(diagram);
```

Java koristi koordinator *BorderLayout* za daljnje grafičko pozicioniranje. U takvim situacijama, koristi se centralna regija. Pošto se ostale regije ne koriste, prostor zauzet ovim regijama postaje minimalan. Prema tome, centralna regija zauzima najveći dio prostora dostupnog okviru.

Osobina *layout*

Metoda *setLayout* eksplisitno određuje koji koordinator kontejnera mora koristiti. Na primjer, bismo mogli imati eksplisitno definisan panel *cp* sa koordinatorom *BorderLayout* na sljedeći način:

```
cp.setLayout(new BorderLayout());
```

Panel *diagram* koristi metodu *setLayout* kako bi eksplisitno odredio koji će se koordinator koristiti kada se grafička komponenta stavi na dijagram.

```
80: this.setLayout(null);
```

Terminom *null* specificiramo da se neće koristiti nijedan koordinator. Grafička komponenta dodana u kontejner *diagram* treba da koristi relativne koordinate *x* i *y* od gornjeg lijevog ugla kontejnera. Ovo vam dozvoljava da sačuvate poziciju kao i veličinu objekta čak i kad se veličina kontejnera promjeni.

Sljedeće instrukcije pozicioniraju novi panel nazvan *box* unutar panela *diagram*:

```
81: this.add(box);
```

Naredna metoda *setLocation*

```
78: box.setLocation(40, 20);
```

određuje poziciju gornjeg lijevog ugla panela *box* (relativne pozicije *x* = 40 i *y* = 20) u odnosu na panel *diagram*(*x* = 0 i *y* = 0). Veličina panela *box* je specificirana na liniji 77:

```
77: box.setSize(100, 50);
```

DIO C:

ŽUTA KUTIJA U DIJAGRAMU

Definisanje i pridruživanje atributa

Linija 8 našeg primjera

```
8: private AppFrame appFrame = new AppFrame(); // &
```

definiše atribut `appFrame` i odmah ga povezuje sa kreiranim objektom. Ova instrukcija zamjenjuje dvije instrukcije primjera u Dijelu B:

```
8: private AppFrame appFrame;
```

```
12: appFrame = new AppFrame();
```

Ista napomena važi i u ovoj sekciji prilikom definisanja sljedećih atributa:

```
46: private Diagram diagram = new Diagram(); // &
```

```
66: private Box box = new Box(); // &
```

Specijalizacija klase (Dio 2)

U drugom dijelu, smo vidjeli da se kôd koji se odnosi na objekat poslije njegovog kreiranja može smjestiti u odvojenu klasu.

Atribut `box` više nije tipa `JPanel` (predefinisane Java klase) kao što je navedeno u prethodnoj sekciji:

```
71: private JPanel box;
```

nego je tipa `Box`:

```
66: private Box box = new Box(); // &
```

Klasa `Box` služi za definisanje drugog panela aplikacije. Primijetite da su u zadnjoj sekciji sljedeće instrukcije

```
76: box.setBackground(Color.BLUE);  
77: box.setSize(100, 50);  
78: box.setLocation(40, 20);
```

pomjerene u konstruktor klase *Box*:

```
92: this.setSize(100, 50);  
93: this.setColor(Color.YELLOW);  
94: this.setLocation(40, 20);
```

Takođe primijetite da konstruktor poziva, na liniji 93, metodu *setColor* umjesto direktnog korištenja metode *setBackground*.

Ova modifikacija definiše koncept osobine.

Osobine

U Poglavlju 3, smo se upoznali sa osnovnim konceptom osobine. Nakon toga, smo pokazali osobinu *visible* lociranu u Java klasi nazvanoj *Component*.

Deklaracija atributa osobine je opcionalna.

Sa *get* i *set* metodama smo napomenuli da je osobina ustvari privatni atribut. U primjeru prezentiranom u ovoj sekciji, smo definisali osobinu *color* unutar klase *Box*. Primijetite da analizirajući kôd klase *Box*, ne nalazimo atribut *color*. Metoda *setColor* poziva metodu *setBackground*

```
102: this.setBackground(aColor);
```

kako bi direktno promjenila boju panela. U ovom slučaju, ne moramo definisati sljedeći atribut

```
private Color color;
```

zato što metoda *setBackground* direktno mijenja poslatu boju, kao parametar, pozadini panela. Čak iako atribut *color* nije definisan, možemo diskutovati o postojanju osobine *color*.

JavaBeans

Osobina =
get i *set* metode

Upravo smo vidjeli da je osobina u stvarnosti stanje objekta komponente koje se može dobiti sa *get* metodom i promijeniti sa *set* metodom. U većini slučajeva osobina ima odgovarajući privatni atribut. U najjednostavnijoj formi, "JavaBean" je klasa koja mora zadovoljavati sljedeće karakteristike:

- 1- Klasa je javna;
- 2- Osnovni (bez parametara) konstruktor je javni.
Naravno, komponenta može sadržavati druge konstruktore.
- 3- Osobine prave referencu na specijalne javne metode ("*is*", "*get*" i "*set*") koje zadovoljavaju dole navedena programerska pravila.

Samo specijalne javne metode mogu modificirati stanje osobine.

Kada atribut osobine postoji, on mora imati modifikator *private*. Prema tome konsultacija ili modifikacija stanja osobine može biti izvedena koristeći samo specijalno namijenjene metode. Naravno, modifikator *public* mora biti prisutan u potpisu ovih metoda.

Čitanje stanja osobine: metoda "*is*" ili "*get*".

Metoda koja čita vrijednost osobine mora imati sljedeću sintaksu:

```
getImeOsobine
```

Primijetite da osobina tipa *boolean* koristi sljedeću sintaksu:

```
isImeOsobine
```

Metode za čitanje ("*is*" ili "*get*") ne koriste ni jedan parametar i vraćaju objekt tipa identičnog onome koji ima atribut osobine (ukoliko se atribut eksplicitno koristi). Na primjer, možete vidjeti potpis metode *getColor* na liniji 97:

```
97: public Color getColor() {
```

Modifikacija ili mijenjanje stanja osobine: metoda "*set*".

Za metodu koja služi za modificiranje stanja osobine, sintaksa se definiše kao:

```
setImeOsobine
```

Osobina je sastavljena od privatnog atributa kao i od javnih metoda koje počinju sa prefiksima *get (is)* ili *set*.

Primijetite da je prvo slovo imena atributa osobine malo početno slovo. U metoda, prvi karakter imena osobine počinje sa velikim početnim slovom, pri čemu ime osobine služi kao sufiks imena metode.

U klasi *Box*, metoda *setColor* se koristila za promjenu boje kutije. Ova metoda, kao i sve druge ”*set*” metode, ne vraćaju nikakvu vrijednost i konsekventno sadrže modifikator *void*. Potpis metode *setColor* je prema tome:

```
101: public void setColor(Color aColor) {
```

Klasa parametra metode ”*set*” je jednaka tipu osobine.

Parametar metode je lokalna varijabla. Vidljivost lokalne varijable je ograničena unutar metode u kojoj je definisana. Bilo koje ime parametra može biti korišteno. Tip parametra odgovara tipu osobine.

Osobina **samo za čitanje**: bez metode ”*set*”.

Kao i atribut koji služi za čuvanje stanja osobine, metode ”*is*”, ”*get*” i ”*set*” su takođe opcionalne. Ukoliko je osobina predodređena samo za čitanje, dozvoljeno je definisanje samo metode ”*get*” (ili ”*is*”). Obrnuto, kada okolnosti diktiraju da se osobina ne može čitati, onda je dozvoljeno definisanje samo metode ”*set*”. Ali u svakom slučaju, jedna od tri navedene javne metode mora biti definisana kako bismo imali osobinu.

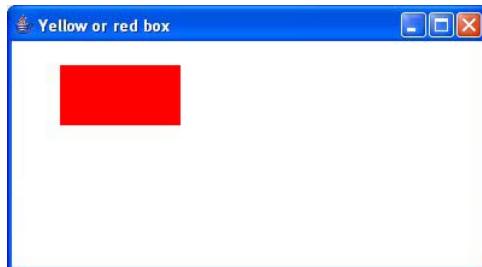
POGLAVLJE 5

DOGAĐAJI

Ovo poglavlje koristi osnovni grafički korisnički interfejs prezentiran u prethodnom poglavlju kako bi se čitalac upoznao sa akcijama miša na tom interfejsu.

Cilj ovog poglavlja je da korisniku omogući da klikne lijevim dugmetom miša na prikazanu kutiju i promijeni boju kutije iz žute u crvenu i obratno.

Rad sa događajima, koji su prezentirani u ovom poglavlju, zahtijeva kontrolni mehanizam koji je adaptiran za svaki tip događaja.



Novi Java programski elementi

Ovo poglavlje će početi sa prezentacijom novih Java programskih elemenata.

Opcionalno korištenje ključne riječi *this*

Ključna riječ *this* je referenca na trenutni objekat. Ona može biti izostavljena, kao što je to pokazano na narednoj liniji kôda:

```
181: return selected;
```

U prikazanom primjeru smo mogli koristiti ključnu riječ *this* kao:

```
return this.selected;
```

Atributi i metode klase mogu biti korišteni bez ključne riječi *this* pošto je njena referenca implicitna kada se kôd nalazi u samoj klasi.

Ukoliko metoda ima parametar koji ima isto ime kao i atribut klase, onda moramo koristiti ključnu riječ *this* kako bi razlikovali atribut od parametra. Na primjer, prezentirana klasa *Tool* ima definisan atribut *name* dok konstruktor u svom potpisu ima parametar istog imena. Kako bismo mogli razlikovati navedeni parametar od atributa ključna riječ *this* je neophodna za korištenje atributa.

```
public class Tool {  
    private String name = "";  
  
    public Tool(String name) {  
        super();  
        this.name = name;  
    }  
}
```

Drugi način imenovanja parametra, koji se koristi za modifikaciju atributa, je korištenje imena parametra sa malim početnim slovom ”a”. Prvi karakter imena atributa je onda napisan velikim slovom. U našem slučaju, klasa *Tool* se može definisati na sljedeći način:

```
public class Tool {  
    private String name = "";  
  
    public Tool(String aName) {  
        super();  
        name = aName;
```

```
    }  
}
```

Opcionalno korištenje imena klase

Kao i ključna riječ *this*, ime klase koje dolazi prije referenciranja na njene metode ili konstante definisane modifikatorom *static*, može biti izostavljeno. Na primjer, u sljedećoj instrukciji:

```
187: this.setBackground(SELECTION_COLOR);
```

mogli smo koristiti ime klase *Box* na sljedeći način:

```
this.setBackground(Box.SELECTION_COLOR);
```

Generalno, konstante i metode definisane sa modifikatorom *static* mogu biti referencirane bez korištenja klase pošto je referenca implicitna u toj samoj klasi.

Slanje poruka između objekata

Poruka ili parametar metode.

Objekti moraju međusobno razmjenjivati poruke kako bi komunicirali jedni sa drugima. Tehnički govoreći, radi se o parametrima (ili argumentima) metoda.

Za razliku od metode *main* koja ima modifikator *void*, metoda *isSelected*

```
180: public boolean isSelected() {  
181:   return selected;  
182: }
```

Instrukcija *return* je instrukcija koja je odgovorna za slanje reference objekta ili vrijednosti tamo gdje je pozvana metoda.

sadrži u okviru svog potpisa (linija 180) tip povratne vrijednosti, koji dolazi prije imena metode. U ovom primjeru, tip podatka je *boolean*. Vrijednost tipa *boolean* se vrati tamo gdje je pozvana metoda ključnom riječi *return*. Prema tome, u našem primjeru, vrijednost atributa *selected* je vraćena, ključnom riječi *return* na liniji 181, sljedećem izrazu na liniji 121:

```
121: box.setSelected(!box.isSelected());
```

Znak uskličnik ”!”, koji je zapravo prvi karakter unutar zagrada metode *setSelected*, pravi negaciju vraćene *boolean* vrijednosti. Na primjer, ukoliko bi metoda *isSelected* vratila vrijednost *true*, vrijednost *false* bi bila ona koja bi se prenijela kao parametar metodi *setSelected*.

Na liniji 184:

```
184: public void setSelected(boolean aSelected) {
```

vidimo u parametru metode *setSelected* deklaraciju argumenta *aSelected* tipa *boolean*. Parametar *aSelected* poprima novu vrijednost na mjestu gdje je metod pozvan, kao na liniji 121.

Varijabla nije neophodno atribut klase.

Primijetite da parametar *aSelected* predstavlja lokalnu varijablu, a ne atribut, pošto vidljivost te varijable ne prelazi granice metode *setSelected*. Definicija atributa je uvijek locirana unutar klase, a izvan metoda te klase.

Instrukcija dodjeljivanja

U Dijelu C prethodnog poglavlja upoznali smo se sa konceptom dodjeljivanja objekta, odnosno vrijednosti, atributima tokom same definicije.

```
66: private Box box = new Box();
```

U ovom poglavlju ćemo pokazati drugačije instrukcije dodjeljivanja vrijednosti. Na primjer:

```
152: private Color currentColor = Color.YELLOW; // +
153: private boolean selected = false; // +
```

Možemo dodjeliti objekat ili vrijednost atributu tokom njegove deklaracije.

Prvi izraz deklariše atribut *currentColor* i inicijalizira ovaj objekat sa referencom na konstantu *YELLOW* koja pripada klasi *Color*. Druga instrukcija deklariše atribut *selected* i inicijalizira ga na vrijednost *false*. Instrukcija dodjeljivanja se može koristiti i unutar metode (na primjer *setCurrentColor*) kao što je prikazano na liniji 170:

```
170: currentColor = aColor;
```

Ovdje dodjeljujemo objekat *aColor*, primljen kao parametar, atributu *currentColor*.

if Instrukcija

Unutar zagrada ”*if*” instrukcije, izraz tipa *boolean* odlučuje koja će se grupa instrukcija izvršiti; ukoliko je izraz *true*, prva grupa instrukcija će se izvršiti; ukoliko je izraz *false*, druga grupa instrukcija će se izvršiti.

Linije od 186 do 191 prikazuju primjer ”*if*” instrukcije:

```
186: if (selected) {  
187:     this.setBackground(SELECTION_COLOR);  
188: }  
189: else {  
190:     this.setBackground(currentColor);  
191: }
```

Na liniji 186 vrijednost atributa *selected*, tipa *boolean*, se ispituje i ukoliko je atribut jednak vrijednosti *true*, izvršiće se linija 187; u drugom slučaju linija 190, koja je dio *else* izraza, će dobiti kontrolu izvršavanja. Primijetite da je izraz *else* optionalan, kao što je prikazano u sljedećim linijama kôda:

```
169: if (aColor != SELECTION_COLOR) {  
170:     currentColor = aColor;  
171:     this.setBackground(aColor);  
172: }
```

Ukoliko više od jedne instrukcije postoji u bloku ”*true*” odnosno ”*false*”, linije kôda moraju biti grupisane unutar zagrada ”{}” kao što je to prikazano u prethodnom primjeru. Zbog razloga čitljivosti i potencijalnih promjena, preporučeno je da se uvijek koriste vitičaste zgrade čak i kad postoji samo jedna instrukcija u nekim od blokova.

Metoda *repaint*

Korištenje metode *repaint*, kao na liniji 173

```
173: this.repaint();
```

je ponekad korisno kako bi se Java forsirala da ”osvježi” kontejner na ekranu. Primijetite da je metoda *repaint*, u našem primjeru, naslijedena od klase *Container*.

Specifikacija naspram implementacije

Kao što smo vidjeli u Poglavlju 1, klase je sastavljena, pored ostalog, i od javnih metoda. Ove metode zapravo dozvoljavaju pristup privatnim atributima objekta. Na primjer, klasa *Box* prezentirana u ovom poglavlju sadrži sljedeće javne metode:

```
public Box()  
  
public Color getCurrentColor()  
public void setCurrentColor(Color aColor)  
public boolean isSelected()  
public void setSelected(boolean aselected)
```

Specifikacija klase = potpisi javnih metoda klase.

Navedeni potpisi metoda su dijelovi "specifikacije" klase *Box*. Programer koji će koristiti klasu *Box* ne mora da razumije kôd koji je dio "implementacije" odnosno logike izvršavanja ovih metoda. Sve što treba da zna je specifikacija klase.

Implementacija = interni kôd lociran unutar zagrada "()" metoda.

Na primjer, interni kôd metode *setCurrentColor* (Color aColor)

```
169: if (aColor != SELECTION_COLOR) {  
170:     currentColor = aColor;  
171:     this.setBackground(aColor);  
172: }  
173: this.repaint();
```

nije neophodan vanjskim korisnicima te metode.

Nasljeđivanje specifikacije i implementacije klase

Naredna linija

```
45: public class AppFrame extends JFrame  
    implements WindowListener { // &
```

Ključna riječ **extends** dozvoljava nasljeđivanje specifikacije i implementacije klase.

prikazuje relaciju nasljeđivanja između klase *AppFrame* naše aplikacije i predefinisane Java klase *JFrame*.

Ova relacija nasljeđivanja je omogućena ključnom riječi *extends*. U našem primjeru, ova relacija dozvoljava klasi *AppFrame* da naslijedi specifikaciju i implementaciju roditeljske klase.

Nasljeđivanje po "ugovoru"

Drugi dio instrukcije, prethodno prikazane na liniji 45, počinje sa ključnom riječi *implements*, a zatim slijedi ime jednog ili više Java interfejsa. Termin interfejs se u ovom kontekstu ne smije brkati sa konceptom grafički korisnički interfejs.

Nadziranje događaja se obezbjeđuje kroz Java mehanizam nazvan *interface*.

Na primjer, klik miša unutar kutije naše aplikacije se smatra događajem pošto se izvršava predefinisana akcija predviđena od programera aplikacije.

Na primjer, Java interfejs *WindowListener*, korišten na liniji 45 sadrži sljedećih sedam specifikacija:

```
public void abstract
    windowOpened(WindowEvent e);
public void abstract
    windowClosing(WindowEvent e);
public void abstract
    windowClosed(WindowEvent e);
public void abstract
    windowIconified(WindowEvent e);
public void abstract
    windowDeiconified (WindowEvent e);
public void abstract
    windowActivated(WindowEvent e);
public void abstract
    windowDeactivated(WindowEvent e);
```

Definicije metoda, koje su dio Java interfejsa, sadrže modifikator *abstract*. Ovaj modifikator indicira da ne postoji implementacija unutar te metode.

Programer mora da zadovolji implementaciju svih abstraktnih metoda koje su navedene u Java interfejsu.

Ugovor između programera i Java programskog jezika sa sastoji od implementacije svih metoda klase koja implementira Java interfejs. Uloga programera je da napiše kôd u funkciji koju partikularni događaj treba.

Sve metode Java interfejsa moraju biti uključene u klasu koja implementira ovaj interfejs.

U našem primjeru, metoda *windowClosing* klase *AppFrame* sadrži specifični kôd koji dozvoljava reakciju na akciju miša (klik na dugme X okvira). Čak iako partikularna akcija nije očekivana mi je moramo uključiti u klasu *AppFrame*. U drugom slučaju, Java će signalizirati grešku tokom kompilacije.

Kada se desi događaj, Java kreira objekat koji je povezan sa tim događajem.

Tokom izvršavanja aplikacije, kada korisnik zatvori okvir, Java kreira objekat klase *WindowEvent*. Ovaj objekat se šalje metodi *windowClosing* i onda se ova metoda izvrši:

```
69: public void windowClosing(WindowEvent e) {  
70:     this.exit();  
71: }
```

Metoda *exit* se onda poziva u klasi *AppFrame*. Njena prva instrukcija je:

```
61: this.dispose();
```

Metoda *dispose* oslobađa resurse povezane sa aplikacionim okvirom. Prema relaciji nasljeđivanja u našem primjeru, metoda *dispose* dolazi iz klase *Frame*.

Nakon izvršavanja metode *dispose*, sljedeća instrukcija je

```
62: System.exit(0);
```

Klasa *System* se nalazi u paketu *java.lang*. Metoda *exit* ove klase stope izvršavanje Java VM. Argument indicira statusni kôd normalnog ili nenormalnog funkcionisanja aplikacije. Prema standardima, status različit od nule indicira nenormalno završavanje aplikacije.

Instrukcija na liniji 51

```
51: this.addWindowListener(this); // +
```

dodaje aplikacionom okviru objekat koji je odgovoran za reakciju na klik miša. Kada se na ovom objektu desi akcija miša, Java ustanovi da li je objekat dio objekata povezanih sa Java interfejsom *WindowListener*.

Ukoliko pogledamo detaljnije, vidjećemo ključnu riječ *this* na dva mesta u instrukciji na liniji 51. Prvi put, ključna riječ *this* pravi referencu na trenutni

objekat klase *AppFrame*, odnosno *appFrame* u klasi *App*. U ovom slučaju, *this* odgovara "izvoru" događaja, tj. odakle je potekao događaj.

U drugom slučaju, *this* je "destinacija" događaja. Drugim riječima, *this* određuje destinacioni objekat koji će odlučiti kako reagovati na događaj izvora. Izvorni i destinacioni objekti ne moraju biti isti.

Java interfejs *MouseListener*

Termin *MouseListener* korišten u klasi *Diagram*

```
102: public class Diagram extends JPanel  
      implements MouseListener { // &
```

određuje drugi Java interfejs korišten u našem primjeru. Napomene koje se odnose na Java interfejs *WindowListener* se takođe odnose i na *MouseListener*. Šta razdvaja ova dva interfejsa je priroda događaja. U prvom slučaju, *WindowListener* reaguje na događaj koji je povezan sa akcijama na objektu *appFrame*. Primjer koji smo vidjeli sadrži akciju zatvaranja aplikacije koristeći metodu *windowClosing*.

Što se tiče Java interfejsa *MouseListener*, on reaguje na klik miša na objekat klase *Box* koji je dio objekta *diagram*. Objekat *diagram* je stavljen u kolekciju objekata koji implementiraju *MouseListener*:

```
108: box.addMouseListener(this); // +
```

Asocijacija objekta *box* sa Java interfejsom *MouseListener* indicira potrebu da Java pošalje objekat tipa *MouseEvent* metodi *mousePressed* destinacionog objekta *this* (*Diagram*), čim Java detektuje klik miša na objektu *box*.

Pregled poglavlja

Ključna riječ *extends* dozvoljava kreiranje relacije nasljedivanja. I specifikacije i implementacije se nasljeđuju od predaka. Drugim riječima, definicije podataka se kopiraju dok se metode referenciraju. Ove reference dozvoljavaju pristup, tokom izvršavanja aplikacije, samim metodama.

Ključna riječ *implements* dozvoljava uspostavljanje veze sa Java interfejsima. Sve metode uključene u specifikaciji Java interfejsa moraju biti implementirane. U tom slučaju četiri faze moraju biti zadovoljene:

4 faze implementacije Java interfejsa.
--

- 1- Dodati instrukcije *import* kako bismo pristupili potrebnom Java interfejsu, kao i određenim klasama koje definišu tipove parametara Java interfejsa. Na primjer:

```
40: import java.awt.event.WindowListener; // +
41: import java.awt.event.WindowEvent; // +
98: import java.awt.eventMouseListener; // +
99: import java.awt.event.MouseEvent; // +
```

- 2- Povezati Java interfejs sa klasom gdje će metode interfejsa biti implementirane. Na primjer:

```
102: public class Diagram extends JPanel
      implements MouseListener { // &
```

- 3- Kako bismo povezali izvorni i destinacioni objekat događaja, dodati destinacioni objekat u kolekciju objekata odgovornih za reakciju na dogadaj. Na primjer:

```
108: box.addMouseListener(this); // +
```

- 4- Implementirati metode koje su dio specifikacije Java interfejsa. Java automatski kreira specijalni objekat koji identificira događaj koji se desio. Ovaj novi objekat služi kao parametar različitim metodama Java interfejsa.

Trag izvršavanja

Prezentiraćemo ovdje korake izvršavanja kada neko odluči da klikne na žutu kutiju dijagrama.

Prvo, pretpostavljamo da je izvor događaja definisan (*box*) i da je destinacija događaja (*this*, t.j., *diagram*) povezana sa objektom *box* na sljedeći način:

```
108: box.addMouseListener(this); // +
```

Kada korisnik odluči da klikne na žutu kutiju, odnosno pravougaonik, Java kreira objekat klase *MouseEvent*. Ovaj objekat se onda šalje metodi *mousePressed*

```
120: public void mousePressed(MouseEvent e) {
121:     box.setSelected(!box.isSelected());
122: }
```

i Java automatski izvršava ovaj metod.

Kada Java uspješno startuje metod `mousePressed`, izvršava se linija 121, odnosno pozove se metod `isSelected`. Vrijednost `false` je vraćena a onda transformisana u `true` (operatorom negacije ”!”) prije predavanja kontrole metodi `setSelected`.

Na liniji 185,

```
185: selected = aSelected;
```

dodjeljuje se vrijednost `true` atributu `selected` pošto je parametar `aSelected` jednak `true`.

Pošto je atribut `selected` sada jednak `true`, rezultat instrukcije `if` na liniji 186

```
186: if (selected) {
```

će imati efekat predavanja kontrole bloku `true`. Prema tome linija 187

```
187: this.setBackground(SELECTION_COLOR);
```

će promijeniti boju u crvenu. Efektivno ova boja odgovara vrijednosti konstante `SELECTION_COLOR` na liniji 148.

```
148: public static final Color  
      SELECTION_COLOR = Color.RED; // +
```

Nakon toga, drugi klik miša će uzrokovati promjenu kutije u orginalnu, žutu boju. Sljedeći klik će vratiti crvenu boju kutiji i tako redom...

POGLAVLJE 6

MENI

Meniji služe za grupisanje semantički povezanih funkcija prema aplikacionoj logici.

Ovo poglavlje prezentira i dodaje navedenu funkcionalnost našem osnovnom grafičkom korisničkom interfejsu koje smo izlagali u prethodnom poglavlju.

Meni bar naše aplikacije je sastavljen od tri menija: File, Edit i View.

File

Exit

Meni File ima samo jednu opciju, Exit, koja služi za izlaz iz aplikacije.

Edit

Select Box

Deselect Box

Meni Edit dozvoljava korisniku da selektuje kutiju sa opcijom Select Box. Boja kutije će onda postati crvena. Opcija Deselect Box je druga opcija ovog menija. Sa ovom opcijom, kutija poprima originalnu žutu boju.

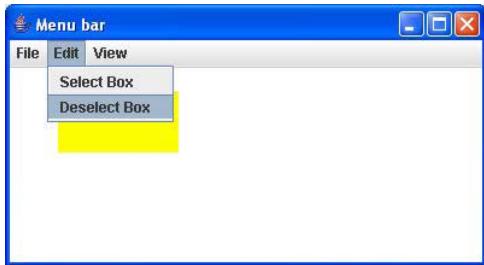
View

Hide Box

View Box

Meni View sadrži opciju Hide Box koja dozvoljava da sakrijete kutiju. Druga dostupna opcija je View Box, koja služi za ponovno prikazivanje sakrivene kutije.

Nova klasa AppMenu je dodata u ovom poglavlju i proširuje funkcionalnost aplikacije prethodnog poglavlja sa meni barom i spomenutim opcijama.



Relacija Meni bar / Java događaji

Java interfejs *ActionListener*

Kreiranje objekta *appMenu*, koji predstavlja naš meni bar, je urađeno na liniji 47:

```
47: private AppMenu appMenu = new AppMenu(this); // +
```

Na liniji 100, u potpisu klase *AppMenu*,

```
100: public class AppMenu extends JMenuBar  
      implements ActionListener {
```

koristi se novi Java interfejs nazvan *ActionListener*. Ovaj interfejs pokreće odgovarajuću akciju kada se jedna od opcija menija izabere od strane korisnika.

Relacija između selektovanja opcije menija i izvršavanja određenih Java instrukcija mora biti uspostavljena od strane programera. Njegova uloga je implementiranje mehanizma za detekciju događaja određenog Java interfejsa.

Objekat *menuFileExit* definisan na liniji 109

```
109: private JMenuItem menuFileExit =  
      new JMenuItem("Exit");
```

Svaka opcija menija mora biti povezana sa Java interfejsom *ActionListener*.

odgovara opciji *Exit* u meniju *File* i on je povezan sa interfejsom *ActionListener* na sljedeći način:

```
122: menuFileExit.addActionListener(this);
```

Navedena instrukcija kreira referencu izmedu objekta kao opcije menija i Java interfejsa *ActionListener* implementiranog u klasi *AppMenu*.

Kada se selektuje opcija menija, Java šalje automatski objekat klase *ActionEvent*, koji identificira događaj, metodi *actionPerformed*.

Prva instrukcija metode *actionPerformed* je

```
169: String command = e.getActionCommand();
```

Ova instrukcija poziva metodu *getActionCommand* na objektu *e* koji odgovara instanci izvršenog događaja. Rezultat metode *getActionCommand* prepoznaje ime izabrane opcije menija kao izvora događaja. Ovo se ime sačuva u lokalnoj varijabli *command*.

Instrukcija *switch*

Metoda *actionPerformed* koristi instrukciju *switch*. Generalna forma ove instrukcije je:

```
switch (expression) {  
    case CONSTANT_1:  
        statement(s)  
    case CONSTANT_2:  
        statement(s)  
    .  
    .  
    .  
    case CONSTANT_N:  
        statement(s)  
    default:  
        statement(s)  
}
```

Blok instrukcijâ *default* je opcionalan.

Nakon evaluacije izraza u zagradi *switch* instrukcije, dobijeni rezultat se poredi sa vrijednostima konstanti u pojedinim *case* blokovima. Ukoliko se ne pronađe odgovarajuća vrijednost, izvršavaju se instrukcije *default* bloka. Blok instrukcijâ *default* je opcionalan. Konstante u pojedinim *case* blokovima moraju biti numeričkog oblika.

Instrukcija *break* terminira izvršavanje instrukcije *switch*.

Kada se pronađe da određena konstanta odgovara evaluiranom izrazu, instrukcije tog bloka počinju sa izvršavanjem. Na kraju izvršavanja zadnje instrukcije tog bloka, počinje izvršavanje prve instrukcije narednog bloka ukoliko nije specificirana instrukcija *break*. Onda kada tok izvršavanja dođe do instrukcije *break*, izvršavanje uslovne instrukcije *switch* terminira.

U našem primjeru instrukcija *switch* počinje na liniji 170

```
170: switch (this.getMenuCommand(command)) {
```

i sadrži sljedeći izraz:

```
this.getMenuCommand(command)
```

Parametar *command* metode *getMenuCommand*, kao što smo već vidjeli, sadrži ime odabrane opcije menija. Pozvana je metoda *getMenuCommand* na linijama od 158 do 166.

```
158: private int getMenuCommand(  
159:     String aCommand) {  
160:     int action = -1;  
161:     if (aCommand.equals("Exit"))  
162:         action = EXIT;  
163:     else if (aCommand.equals("Select"))  
164:         action = SELECT;  
165:     else if (aCommand.equals("Deselect"))  
166:         action = DESELECT;  
167:     else if (aCommand.equals("Hide"))  
168:         action = HIDE;  
169:     else if (aCommand.equals("Show"))  
170:         action = SHOW;  
171:     return action;  
172: }
```

Ova metoda vraća, na liniji 165, vrijednost broja koji odgovara odabranoj opciji menija.

Na primjer, ukoliko je odabrana opcija *Exit* menija *File*, izraz *if* instrukcije na liniji 160 vraća vrijednost *true* i izvršava se naredna instrukcija

```
action = EXIT;
```

Pošto vrijednost konstante *EXIT* sadrži vrijednost 1

```
102: private static final int EXIT = 1;
```

povratna vrijednost na liniji 165 ima vrijednost 1.

Pošto rezultat metode *getMenuCommand* odgovara vrijednosti prvog *case* bloka na liniji 171

```
171: case EXIT:
```

izvršava se instrukcija tog bloka:

```
172: appFrame.exit();
```

Primijetite da objekat *appFrame* odgovara aplikacionom okviru. U narednoj sekciji ćemo vidjeti kako dobiti referencu na aplikacioni okvir u klasi *AppMenu*.

Nakon svega, *break* instrukcija na liniji 173

```
173: break;
```

terminira izvršavanje *switch* instrukcije.

Povezivanje objekata

Mogućnost modificiranja boje prikazane kutije iz žute u crvenu, preko opcije menija "Select Box", ukazuje na prisustvo funkcionalne veze između različitih objekata. Ovi objekti su aplikacioni okvir, meni bar, opcija menija "Select Box", dijagram i kutija. Meni bar i dijagram su dijelovi aplikacionog okvira. Opcija menija "Select Box" je dio meni bara, i kutija je dio dijagrama.

Cilj se sastoji od ostvarivanja relacije između objekta *menuEditSelect* i objekta *box*.

Početna tačka je objekat *menuEditSelect* pošto je ovo objekat na kojem će se događaj desiti. Završna tačka je objekat *box* pošto se njegova boja mora promjeniti iz žute u crvenu.

Pošto navedena opcija menija postoji u klasi *AppMenu*, automatski imamo pristup objektu *menuEditSelect*.

Prva veza koju moramo ostvariti je veza između objekata *appMenu* i *appFrame*. Ova veza je ostvarena kada se pozove konstruktor klase *AppMenu* u klasi *AppFrame*:

```
47: private AppMenu appMenu = new AppMenu(this);
```

U suprotnosti sa svim konstruktorima koje smo do sada vidjeli, poslana je referenca na objekat višeg nivoa (ovdje je to objekat *appFrame* reprezentiran ključnom riječi *this*) kao parametar pošto će se ovaj objekat koristiti od strane menija (*appMenu*).

Kako bismo korektno koristili meni bar, objekat *appMenu* mora efektivno imati referencu na objekat *appFrame*, pošto ovaj zadnji sadrži objekat *diagram* koji dozvoljava pristup objektu *box*.

Konstruktor *AppMenu* koji sadrži parametar tipa *AppFrame* je prezentiran ovdje:

```
145: public AppMenu(AppFrame aAppFrame) {  
146:     this();  
147:     this.setAppFrame(aAppFrame);  
148: }
```

Kada se ključna riječ *this* koristi unutar konstruktora kao metoda ona pokazuje na konstruktor klase.

Instrukcija na liniji 146 pokazuje interesantnu specifičnost ključne riječi *this* unutar konstruktora. Kada se ključna riječ *this* koristi unutar konstruktora kao metoda, ona pravi referencu na konstruktor klase. U našem primjeru, linija 146 poziva osnovni konstruktor klase *AppMenu*. (linije od 119 do 143).

Izvršavanje metode "set" na liniji 147 dozvoljava čuvanje, pomoću osobine *appFrame*, objekta koji je poslan kao parametar konstruktoru *AppMenu*.

Nakon što smo referencirali objekat *appFrame* u klasi *AppMenu*, moguće je pristupiti objektu *diagram*, pošto je on sadržan unutar objekta *appFrame*. Na isti način moguće je pristupiti objektu *box* pošto se on nalazi unutar objekta *diagram*.

Čak iako objekti *box* i *diagram* postoje unutar objekta *appFrame*, dvije "get" metode moraju biti korištene, kako bi se pronašle reference na navedene objekte, pošto su ovi objekti atributi koji sadrže modifikator *private* u njihovim definicijama:

```
48: private Diagram diagram = new Diagram();
200: private Box box = new Box();
```

Prvo, metoda *getDiagram*, koja je locirana unutar klase *AppFrame*, dobija referencu na objekat *diagram*:

```
62: public Diagram getDiagram() {
63:     return diagram;
64: }
```

Druge, referenca na objekat *box* dolazi od metode *getBox* locirane unutar klase *Diagram*:

```
212: public Box getBox() {
213:     return this.box;
214: }
```

Konačno, instrukcija na liniji 175

```
175: appFrame.getDiagram().getBox().
setSelected(true);
```

je instrukcija koja dozvoljava modificiranje boje panela *box*. Osobina *selected*, asocirana sa klasom objekta *box*, tada postaje jednaka vrijednosti *true*. To je indikator koji nam govori da li je objekat selektovan (vrijednost *true*) ili ne (vrijednost *false*):

```
273: public void setSelected(
    boolean aSelected) {
274:     selected = aSelected;
275:     if (selected) {
276:         this.setBackground(SELECTION_COLOR);
277:     }
278:     else {
```

```
279:         this.setBackground(currentColor);  
280:     }  
281:     this.repaint();  
282: }
```

Razvijanje meni bara

Nekoliko instrukcija se mora koristiti u konstrukciji meni bara. Naredna linija

```
100: public class AppMenu extends JMenuBar  
      implements ActionListener {
```

pokazuje relaciju nasljeđivanja klase *AppMenu* sa roditeljskom klasom *JMenuBar*.

Sljedeći atributi definišu tri menija aplikacije:

Kreiranje menija.

```
108: private JMenu menuFile = new JMenu("File");  
110: private JMenu menuEdit = new JMenu("Edit");  
113: private JMenu menuView = new JMenu("View");
```

Parametri korišteni u konstruktoru *JMenu* odgovaraju imenima tri različita menija.

Sljedećih pet atributa definišu opcije menija koje su locirane u prethodno definisanim menijima:

Kreiranje opcija menija.

```
109: private JMenuItem menuFileExit =  
      new JMenuItem("Exit");  
111: private JMenuItem menuEditSelect =  
      new JMenuItem("Select Box ");  
112: private JMenuItem menuEditDeselect =  
      new JMenuItem("Deselect Box ");  
114: private JMenuItem menuViewHide =  
      new JMenuItem("Hide Box");  
115: private JMenuItem menuViewShow =  
      new JMenuItem("Show Box");
```

Određivanje imena akcije za svaku opciju menija.

Metoda *setActionCommand* korištena u sljedećim opcijama menija

```
121: menuFileExit.setActionCommand("Exit");

124: menuEditSelect.setActionCommand("Select");
126: menuEditDeselect.setActionCommand(
        "Deselect");
129: menuViewHide.setActionCommand("Hide");
131: menuViewShow.setActionCommand("Show");
```

daje interno ime akcije za svaku opciju menija. To ime akcije dozvoljava metodi *getActionCommand* na liniji 169

```
168: public void actionPerformed(
        ActionEvent e) {
169:     String command = e.getActionCommand();
170:     switch (this.getMenuCommand(command)) {

158:     private int getMenuCommand(
            String aCommand) {
159:         int action = -1;
160:         if (aCommand.equals("Exit"))
                action = EXIT;
161:         else if (aCommand.equals("Select"))
                action = SELECT;
162:         else if (aCommand.equals("Deselect"))
                action = DESELECT;
163:         else if (aCommand.equals("Hide"))
                action = HIDE;
164:         else if (aCommand.equals("Show"))
                action = SHOW;
165:         return action;
166:     }
```

identifikaciju opcije izabrane od strane korisnika. Na primjer, kada korisnik odluči da selektuje opciju "Select Box", lokalna varijabla *command* definisana na liniji 169 dobija vrijednost "Select" od metode *getActionCommand*. Tada na liniji 170, metoda *getMenuCommand* vraća konstantu SELECT definisanu na liniji 103:

```
103: private static final int SELECT = 2;
```

Dodatno, sljedeće instrukcije:

```
122: menuFileExit.addActionListener(this);
125: menuEditSelect.addActionListener(this);
```

```
127: menuEditDeselect.addActionListener(this) ;  
130: menuViewHide.addActionListener(this) ;  
132: menuViewShow.addActionListener(this) ;
```

povezuju svaku opciju menija sa funkcionalnošću koju obezbjeđuje interfejs *ActionListener*. Ovo znači da u momentu kada korisnik izabere jednu od pomenutih opcija menija, Java automatski reaguje slanjem objekta, koji identificira događaj, metodi *actionPerformed*. Tada će akcija koja odgovara selektovanoj opciji biti izvršena.

Opcija menija u tom kontekstu je "izvor" događaja. Objekat koji odgovara lokализaciji Java interfejsa je "destinacija" događaja. Na primjer, na liniji 122 izvorni objekat je *menuFileExit*, a destinacioni objekat je referenciran sa ključnom riječi *this* (*appMenu*).

Instrukcije koje slijede dodaju opcije menijima:

Veza meniji/opcije.

```
134: menuFile.add(menuFileExit) ;  
135: menuEdit.add(menuEditSelect) ;  
136: menuEdit.add(menuEditDeselect) ;  
137: menuView.add(menuViewHide) ;  
138: menuView.add(menuViewShow) ;
```

Onda kada su meniji definisani, ostaje da definišemo meni bar:

Veza meni bar/meniji.

```
140: this.add(menuFile) ;  
141: this.add(menuEdit) ;  
142: this.add(menuView) ;
```

Primijetite da je meni bar reprezentiran sa ključnom riječi *this*.

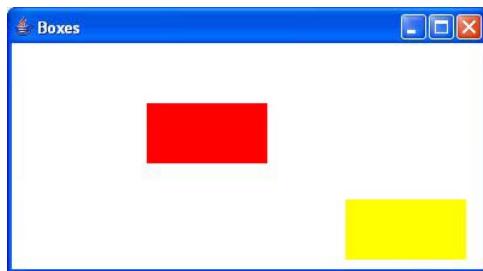
U klasi *AppFrame*, meni bar je dodat, sljedećom instrukcijom

```
53: this.setJMenuBar("this.appMenu") ; // +
```

POGLAVLJE 7

DINAMIČKA MANIPULACIJA OBJEKATA

Prvi dio ovog poglavlja objašnjava kako dozvoliti korisniku da dinamički dodaje velik broj kutija unutar dijagrama. Ova nova funkcionalnost povećava efektivnost grafičkog korisničkog interfejsa prezentiranog u Poglavlju 5. Sjetite se da je prije bilo moguće dodavanje samo jedne kutije a zatim selektovanje i deseletovanje te kutije.

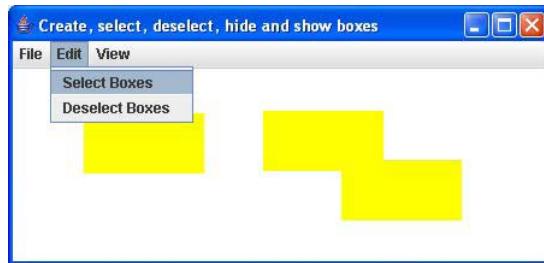


Ovdje će korisnik imati slobodu da kreira proizvoljan broj kutija. Kutije se mogu dodavati bilo gdje u dijagramu. Veličina tih kutija je fiksirana i ostaje ista kao ona korištena u Poglavlju 5.

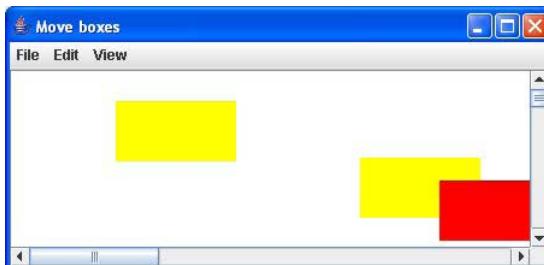
Kada je kutija kreirana njena boja će biti žuta sve dok je korisnik ne selektuje pri čemu boja postaje crvena. Ponovnim klikanjem na selektovanu kutiju se mijenja boja iz crvene u originalnu žutu.

Kako bi prezentirali novi koncept, što je moguće jednostavnije, odlučili smo da privremeno isključimo kôd meni bara dodat u Poglavlju 6.

Nakon što objasnimo ponašanje dogadaja u dva različita izvorna objekta (diagram i box) ponovo ćemo integrisati, u drugom dijelu ovog poglavlja, meni bar prezentiran u prethodnom poglavlju. Ovdje moramo prilagoditi nekoliko opcija menija tako da akcija izvedena na jednoj kutiji može biti primijenjena na sve kutije dijagrama.



U trećem dijelu ovog poglavlja, pokazat ćemo kako dinamički pomjerati kutiju unutar grafičkog korisničkog interfejsa. Pomjeranjem miša, dok se drži pritisnuto dugme miša iznad kutije, dozvoljava korisniku da pomjeri kutiju na novu lokaciju u dijagramu. Dva bara za pomjeranje radne površine su takođe dodata grafičkom interfejsu. Ovim se dozvoljava da radna površina dijagrama može biti povećana u slučaju da prostor postane nedovoljan za prikazivanje kutija.



DIO A: *KUTIJE U DIJAGRAMU*

Cilj ove sekcije je da se dozvoli korisniku dinamičko dodavanje proizvoljnog broja kutija. Gornji lijevi ugao svake kreirane kutije odgovara poziciji kursora miša unutar panela *diagram*. Kutija se kreira klikom miša.

Interfejs *MouseListener*

Izvor i destinacija događaja

Pošto želimo da akcija miša kreira kutiju, koristićemo, kao u Poglavlju 5, interfejs *MouseListener*:

```
90: public class Diagram extends JPanel  
      implements MouseListener {
```

Prvi izvor događaja: panel.

U Poglavlju 5 je samo jedan tip objekta bio vezan za Java interfejs *MouseListener*. Ovdje moramo povezati dva tipa objekata sa navedenim interfejsom. Prvo moramo predvidjeti događaj kada se klikne mišem na panel *diagram*. Ova akcija će kreirati novu kutiju čiji će gornji lijevi ugao odgovarati poziciji gdje se kliknulo mišem. Instrukcija koja kreira ovu prvu vezu je sljedeća:

```
97: this.addMouseListener(this);
```

Drugi izvor događaja: kutija.

Kada je kutija kreirana i postavljena na odgovarajuće mjesto u dijagramu, aplikacija mora biti u mogućnosti da detektuje drugu akciju izvedenu na kutiji. Kako bismo kreirali ovu novu funkciju, moramo ostvariti vezu između svake kutije u dijagramu i interfejsa *MouseListener*. Linija 113

```
113: newBox.addMouseListener(this);
```

sadrži instrukciju koja dozvoljava upravo ovu mogućnost. Konkretno, objekat

`newBox` služi kao izvor događaja.

Programer mora **specificirati** destinaciju događaja gdje je implementiran Java interfejs.

Kao što smo vidjeli u Poglavlju 5, dva objekta moraju biti uzeta u obzir kako bismo kreirali asocijaciju između objekta i Java interfejsa. Implicitirani objekat (izvor događaja) mora biti indiciran kao i objekat (klasa) u kojem je implementiran Java interfejs. Ovaj zadnji objekat zovemo destinacija događaja.

Programer aplikacije mora specificirati definiciju metoda povezanih sa Java interfejsom. Ključna riječ `this` na liniji 113 definiše destinaciju trenutnog objekta koji će obraditi događaj miša kada se on desi na kutiji. Ista napomena važi i za liniju 97. U ovom slučaju, destinacija je ponovo predstavljena, na kraju linije, ključnom riječi `this`.

Kao što smo vidjeli, dva izvora događaja moraju biti definisana, jedan na dijagramu i drugi na kutiji. Oni se međusobno isključuju pošto, u našoj aplikaciji, radna površina ne može doći u konflikt sa površinom zauzetom od strane kutije.

Kako bismo specificirali proces na izvoru događaja, moramo verificirati prirodu objekta kao što je prikazano na linijama 121 i 124 ispod:

Verifikacija jednakosti dva objekta i verifikacija koja instanca pripada kojoj klasi.

```
121: if (e.getSource() == this) {  
122: else if (e.getSource() instanceof Box) {
```

Metoda `getSource` vraća objekat izvora događaja.

Metoda `getSource` vraća referencu izvora događaja, tj. referencu objekta na kojem se desio klik miša. Na primjer, ukoliko se radi o objektu kutije (instanca klase `Box`) izraz `if` instrukcije na liniji 124 će vratiti vrijednost `true` i sljedeće dvije linije će biti izvršene:

```
125: Box clickedBox = (Box) (e.getSource());  
126: clickedBox.setSelected(  
    ! (clickedBox.isSelected()));
```

Prilikom dodjeljivanja reference, klasa objekta koji se dodjeljuje može biti promijenjena prije same dodjele.

Na liniji 125, čak iako objekat izvora događaja ustvari predstavlja instancu klase `Box`, metoda `getSource` vraća objekat tipa `Object` (klasa na vrhu hijerarhi-

je nasljeđivanja). Prema tome, ovaj objekat mora biti konvertovan prije nego što se referenca dodijeli objektu *clickedBox*. Iako objekat koji dobija referencu na izvor događaja pripada klasi *Box*, metoda *getSource* ne može pokriti sve moguće tipove objekata i zbog toga ne vraća objekat tipa *Box*. Obaveza programera je da forsira konverziju (u Java programskom jeziku nazvano *casting*).

Da bismo završili sa obradom događaja miša na liniji 126 našeg primjera, boolean vrijednost osobine *selected*, osobine kutije na kojoj se desio događaj, je promijenjena u suprotnu vrijednost.

Poredak metoda Java interfejsa *MouseListener*

Svaka metoda interfejsa *MouseListener* posjeduje predefinisano ponašanje.

mouseEntered

mouseExited

Onda kada kurzor miša (bez klikanja) nađe na površinu dijagrama, reaguje metoda *mouseEntered*. Ukoliko je zatim kurzor pozicioniran na kutiju, reaguje metoda *mouseExited* zato što se prelazi sa površine dijagrama na površinu zauzetu od strane kutije. Nakon toga, ponovo reaguje metoda *mouseEntered* zato što je drugi izvor događaja povezan sa kutijom. Kada kurzor napušta površinu kutije vraćajući se, na primjer, ponovo na radnu površinu dijagrama, ponovo reaguje metoda *mouseExited*.

mousePressed

Onda kada se izvede akcija miša, kao što je klik miša na dijagram, reaguje metoda *mousePressed*. U našem primjeru, ukoliko je izvor događaja ustvari dijagram kao što je prikazano na liniji 121,

```
121: if (e.getSource() == this) {
```

biće pokrenuta metoda *addNewBox*:

```
122: this.addNewBox(e.getPoint());
```

Ova metoda definiše novu kutiju na dijagramu. Kao što smo već vidjeli, ukoliko se desi klik miša na kutiju, reaguje metoda *mousePressed* i izvršavaju se linije 125 i 126:

```
125: Box clickedBox = (Box) (e.getSource());
```

```
126: clickedBox.setSelected(
```

```
! (clickedBox.isSelected()));
```

MouseReleased

Nakon metode *mousePressed*, reaguje metoda *mouseReleased* na puštanje dugmeta miša.

MouseClicked

Na kraju se poziva metoda *mouseClicked*. To je događaj sastavljen od dva dogadaja *mousePressed* i *mouseReleased*.

Definisanje koordinata panela

Metoda *addNewBox* je pozvana unutar metode *mousePressed*,

```
122: this.addNewBox(e.getPoint());
```

i ona kreira novi objekat tipa *Box*.

Parametar metode *addNewBox* koristi metodu *getPoint* koja pripada eksternoj klasi *MouseEvent*. Ova metoda vraća x i y koordinate pozicije cursora miša na izvornom objektu. U našem primjeru, panel *diagram* djeluje kao izvor događaja. Gornji lijevi ugao panela ima koordinate x = 0 i y = 0.

Koordinate vraćene sa metodom *getPoint* u stvari predstavljaju objekat kompatibilan sa osobinom *point* klase *MouseEvent*. Pošto metoda *addNewBox* na liniji 122 koristi metodu *getPoint* kao parametar, definicija parametra sa specifikacijske tačke gledišta mora biti tipa *Point*, kao što možemo vidjeti na liniji 110:

```
110: private void addNewBox(Point aPoint) {
```

Sve kutije imaju istu širinu i dužinu.

```
149: public static final Dimension BOX_SIZE =
      new Dimension(100, 50);
```

Prepoznavanje objekta

U primjeru naše aplikacije, svi novi objekti su tipa *Box* i kreirani su sa sljedećom instrukcijom:

```
111: Box newBox = new Box();
```

Ova instrukcija koristi isto ime *newBox* za sve nove kutije. Da li će postojati problem prilikom prepoznavanja objekata?

getSource dozvoljava prepoznavanje objekta koji je izvor dogadaja.

Odgovor je negativan jer Java programski jezik obezbeđuje način za dobijanje reference na objekat koji pokreće događaj. Metoda *getSource* se koristi za traženje reference prema trenutnom objektu kao što je prikazano na liniji 125:

```
125: Box clickedBox = (Box) (e.getSource());
```

Ime objekta nema neku važnost, jer objekat koji nas interesuje u realnosti je onaj u asocijaciji sa događajem koji se desio sa zadnjim klikom miša.

Objekat kome se dodijeli referenca drugog objekta postaje taj objekat.

Onda kada je objekat prepoznat, dodjeljujemo njegovu referencu privremenom objektu nazvanom *clickedBox*. Ovaj objekat je u stvari lokalna varijabla (lokalna u metodi *addNewBox*) čija interna vrijednost odgovara referenci kliknutog objekta. Ista varijabla može sadržavati različite reference u različitim trenucima.

Kada se klikne na drugu kutiju, vrijednost *clickedBox* objekta će biti modifirana tako da može referencirati tu drugu kutiju. Ovo prepoznavanje objekta će dozvoliti modifikaciju osobine *selected* tokom izvršenja sljedeće instrukcije:

```
126: clickedBox.setSelected(  
        !(clickedBox.isSelected()));
```

DIO B: *MENI BAR*

U drugom dijelu ovog poglavlja, naš cilj se sastoji od dodjeljivanja funkcionalnosti meni bara, prezentiranog u poglavlju 6, svim kutijama grafičkog korisničkog interfejsa. Realizacija cilja se sastoji od zamjenjivanja sljedećih instrukcija

```
175: appFrame.getDiagram().getBox() .  
      setSelected(true);  
178: appFrame.getDiagram().getBox() .  
      setSelected(false);  
181: appFrame.getDiagram().getBox() .  
      setVisible(false);  
184: appFrame.getDiagram().getBox() .  
      SetVisible(true);
```

sa ovim instrukcijama:

```
165: appFrame.getDiagram().selectAllBoxes();  
168: appFrame.getDiagram().deSelectAllBoxes();  
171: appFrame.getDiagram().hideSelectedBoxes();  
174: appFrame.getDiagram().showSelectedBoxes();
```

Na primjer, u Poglavlju 6, *Box* metoda *setSelected* korištena na liniji 175 je zamijenjena sa *Diagram* metodom *selectAllBoxes*:

```
210: public void selectAllBoxes() {  
211:     Component[] components =  
          this.getComponents();  
212:     Box box;  
213:     for (int i = 0;  
          i < components.length; i++) {  
214:         if (components[i] instanceof Box) {  
215:             box = (Box) components[i];  
216:             box.setSelected(true);  
217:         }  
218:     }  
219: }
```

Metode `selectAllBoxes`, `deSelectAllBoxes`, `hideSelectedBoxes` kao i `showSelectedBoxes` sadržavaju seriju instrukcija koje zahtijevaju detaljnije objašnjenje.

Nizovi

U Poglavlju 1, smo površno objasnili koncept niza. Koncept se sastojao od niza `args` koji služi kao parametar metode `main`.

Generalno, možemo definisati niz kao kolekciju podataka ili referenci objekata čuvanih u pojedinim celijama. Svaki objekat se naziva element niza. Svaki element mora biti istog tipa.

Linija 211

```
211: Component[] components = this.getComponents();
```

definiše lokalnu varijablu nazvanu `components`.

Redni brojevi elemenata niza počinju sa indeksom nula.

Elementi niza su poredani po indeksima počevši od vrijednosti nula.

Broj elemenata niza, koji čine sam niz, se naziva veličina niza. Na liniji 211 metoda `getComponents` ima funkciju pristupa svakoj komponenti kontejnera `this` (diagram).

Primijetite da u našem primjeru, kutije reprezentiraju komponente. Tokom korištenja metode `add` na liniji 205

```
205: this.add(newBox);
```

referenca na objekat `newBox` je automatski dodata listi internih objekata (komponenti) koji čine dio kontejnera `this` (diagram).

Moguće je određivanje veličine niza, koja je jednaka vrijednosti "N", gdje "N" predstavlja broj kutija koje sadrži kontejner `diagram`.

Takođe u našem primjeru, tip niza `components` će biti tip asociran sa klasom `Component`. Tehnički tip niza se piše na sljedeći način:

```
211: Component[] ...
```

Što se tiče pristupa jednom od elemenata niza, izraz koji određuje vrijednost indeksa se evaluira da bi se dobio cijeli broj. Na primjer sljedeća linija

```
214: if (components[i] instanceof Box) {
```

pristupa elementu niza *components* i to pristupanje je određeno evaluiranjem indeksne varijable *i*. Na primjer, ukoliko je ova varijabla jednaka 8, pristupće se 9-om elementu niza.

Instrukcija *for*

Petlja izvršavanja

U našoj aplikaciji, kada korisnik pokrene akciju klikom miša na opciju menija "Select Boxes", sljedeća instrukcija počinje sa izvršavanjem:

```
165: appFrame.getDiagram().selectAllBoxes();
```

Kako bismo selektovali sve kutije dijagrama, neophodno je izvršavanje jedne instrukcije nekoliko puta u vremenu. U ovom kontekstu, prikazane su instrukcije neophodne za selektovanje samo jedne kutije:

```
212: Box box;  
215: box = (Box) components[i];  
216: box.setSelected(true);
```

Linija 212 kreira lokalnu varijablu nazvanu *box*. Linija 215 kopira u objekat *box* referencu, koja ovisi o vrijednosti varijable *i*, na kutiju u nizu *components*.

Nakon što smo ostvarili referencu na specifičnu kutiju, ostaje da se pozove metoda *setSelected* klase *Box* kako bismo dodijelili vrijednost *true* osobini *selected* referenciranog objekta. Ova operacija mora biti izvedena za sve kutije u dijagramu. Da bismo to postigli, moramo kreirati petlju koristeći *for* instrukciju.

Prikazana je generalna forma navedene instrukcije:

```
for (inicijalizacija; boolean izraz; brojač) {  
    instrukcije u petlji;  
}
```

Inicijalizacija (prvi blok).

Prije početka izvršavanja instrukcija unutar *for* petlje, neophodno je startovati jednu ili više instrukcija u prvom bloku. Ukoliko postoji više od jedne instrukcije tada se one moraju odvojiti zarezima. Na primjer:

```
x = 5, y = 8;
```

Karakter ";" indicira kraj prvog od tri bloka *for* instrukcije. Primijetite da je incijalizacijska instrukcija izvršena samo jednom.

Boolean izraz (drugi blok)

Nakon faze incijalizacije, vrijednost boolean izraza drugog bloka određuje da li instrukcije unutar petlje trebaju biti izvršene. Kao i prvi blok, drugi blok takođe završava sa karakterom tačka-zarez.

Brojač (treći blok)

Kada rezultat evaluacije boolean izraza postane jednak *true*, sistem izvršava prvi put instrukcije unutar *for* petlje. Onda kada se izvrši zadnja instrukcija, kontrola se prenosi na treći odnosno zadnji blok. Generalno se instrukcije u trećem bloku *for* instrukcije koriste za mijenjanje vrijednosti indeksne varijable koja kontroliše, u drugom bloku, izlaz iz petlje.

Kada se izvrše instrukcije trećeg bloka, kontrola se ponovo daje drugom bloku kako bi se ponovno evaluirao boolean izraz. Ukoliko je rezultat *false*, *for* instrukcija terminira i kontrola se prenosi na instrukcije koje slijede iza *for* instrukcije.

Primjer *for* instrukcije:

```
213: for (int i = 0;  
           i < components.length; i++) {  
214:     if (components[i] instanceof Box) {  
215:         box = (Box) components[i];  
216:         box.setSelected(true);  
217:     }  
218: }
```

Brojač koji je dio trećeg bloka koristi novu formu instrukcije dodjeljivanja.

Operatori ++ i --

Operatori "++" i "--" dodaju i oduzimaju vrijednost 1 od vrijednosti koja sadrži varijabla na koje se pomenuti operatori odnose. Na primjer:

```
i++;
```

Kada se operator "++" stavi poslije varijable, vrijednost varijable se koristi prije dodavanja vrijednosti 1. Na primjer:

```
int a = 10;  
int b;  
b = a++;
```

Nakon izvršenja zadnje linije, varijabla `a` ima vrijednost 11, a `b` ima vrijednost 10. Ukoliko se operator ++ postavi prije varijable `a`, varijable `a` i `b` imaju vrijednost 11.

Primijetite da ove napomene ne stvaraju nikakvu konfuziju u našem primjeru `for` petlje na liniji 213.

```
213: for (int i = 0;  
        i < components.length; i++) {
```

Mogli smo upotrijebiti izraz `++i` i ostvarili bismo isti rezultat.

Vidljivost varijabli

Vidjeli smo na liniji 213 unutar `for` instrukcije inicijalizaciju indeksne varijable `i`. Pošto je ova varijabla definisana unutar `for` instrukcije, njena vidljivost, odnosno trajanje, neće biti moguće izvan navedene instrukcije.

Generalno, zagrade {} određuju vidljivost varijabli. Na primjer, niz na liniji 211

```
211: Component[] components = this.getComponents();
```

može biti korišten samo unutar metode `selectAllBoxes`.

Veličina niza

Drugi blok `for` instrukcije naše aplikacije, sadrži sljedeći boolean izraz:

```
i < components.length;
```

Ovaj izraz verificira da li su sve kutije selektovane. Kada `i` bude jednaka veličini niza (broju kutija), izraz će vratiti boolean vrijednost `false` i `for` petlja terminira.

DIO C: BAR ZA POMJERANJE RADNE POVRŠINE

U trećem i zadnjem dijelu ovog poglavlja, nastavljamo sa prezentacijom koja se tiče dinamičke manipulacije objekata grafičkog korisničkog interfejsa. Ovdje ćemo se koncentrisati uglavnom na aspekt promjene pozicije kutije na dijagramu. Počinjemo tako što ćemo pokazati kako dodati veću radnu površinu uprkos fizičkim ograničenjima korisničkog monitora.

Bar za pomjeranje radne površine

Linija 58, u ovom dijelu,

```
58: cp.add(new JScrollPane(diagram));
```

proširuje funkcionalnost panela *diagram* prije njegovog samog dodavanja aplikacionom okviru. Objekat *diagram* služi kao početna tačka konstrukcije novog objekta prema definiciji klase *JScrollPane*.

Metoda *getPreferredSize* je eksplicitno definisana u klasi *Diagram*. Ona zamjenjuje metodu *getPreferredSize* naslijedenu iz klase *JComponent*:

```
264: public Dimension getPreferredSize() {  
265:     return new Dimension(1600, 1600);  
266: }
```

Ova definicija dozvoljava eksplicitno forsiranje veličine kontejnera *diagram*. Pošto je *getPreferredSize* definisana na ovaj način, Java više ne određuje osnovnu veličinu panela *diagram*, nego to određuje ova specijalizacija metode *getPreferredSize*.

Na ovaj način se dodjeljuje veličina 1600 piksela sa 1600 piksela objektu *diagram* lociranom unutar objekta *appFrame* koji sam ima veličinu 600 puta 400 (linija 56).

```
56: this.setSize(600, 400); // &
```

Pošto je veličina dijagrama veća od veličine objekta `appFrame`, dodat je bar za pomjeranje radne površine desno i ispod okvira grafičkog korisničkog interfejsa.

Dinamičko pomjeranje kutija

MouseListener interfejs

Novi interfejs `MouseListener`, korišten u potpisu klase `Diagram`,

```
193: public class Diagram extends JPanel  
      implements MouseListener, MouseMotionListener  
{ // &
```

je izvor mehanizma koji dozvoljava pomjeranje, uz pomoć miša, bilo koje kutije dijagrama. Naravno, objekat koji je odgovoran za reakciju na događaj mora biti povezan sa Java interfejsom `MouseListener` kao što je prikazano na liniji 210:

```
210: newBox.addMouseListener(this); // +
```

MouseMotionListener metode:

- `mouseDragged`;
- `mouseMoved`.

Kada se desi događaj ovakve prirode, Java kreira objekat prema definicijama sadržanim u klasi `MouseEvent`. Ovaj objekat se onda šalje metodama `mouseDragged` i `mouseMoved` koje su dio interfejsa `MouseMotionListener`.

Prva stvar koju moramo uraditi kada želimo pomjeriti kutiju je kliknuti i držati lijevi klik miša nad željenom kutijom. Pritisak miša indicira Java programskom jeziku da se desio događaj i Java kreira objekat klase `MouseEvent`. Važno je napomenuti da dva interfejsa, `MouseListener` i `MouseMotionListener`, korištena u klasi `Diagram` reaguju na pritisak miša.

Prvo reaguje na klik miša metoda `mousePressed` koja pripada interfejsu `MouseListener`. Jedna od instrukcija je:

```
276: mousePressed = e.getPoint(); // +
```

Primijetite definiciju atributa `mousePressed` koja je prethodno urađena na liniji 197:

```
197: private Point mousePressed = new Point(0, 0);  
// +
```

Instrukcija na liniji 276 čuva vrijednost koordinata x i y lociranih unutar kutije tamo gdje smo kliknuli. Primijetite da su vrijednosti koordinata x i y relativne u odnosu na gornji lijevi ugao objekta *clickedBox* koji odgovara koordinatama $x = 0$ i $y = 0$.

Držeći pritisnuto lijevo dugme miša, laganim pomjeranjem miša dozvoljavamo Java programskom jeziku da pošalje novi objekat, sličan onom koji je poslan kao parametar metodi *mousePressed*, metodi *mouseDragged* interfejsa *MouseMotionListener*:

```
286: public void mouseDragged(MouseEvent e) {  
287:     if (e.getSource() instanceof Box) {  
288:         Box box = (Box) e.getSource();  
289:         // box location (x, y) and size (r)  
         // before the box has been dragged  
290:         Rectangle r = box.getBounds();  
291:         int x = r.x + e.getX() -  
                 mousePressed.x;  
292:         int y = r.y + e.getY() -  
                 mousePressed.y;  
293:         if (x < 0) x = 0;  
294:         if (y < 0) y = 0;  
295:         box.setLocation(x, y);  
296:     }  
297: }
```

Linija 287 dozvoljava prvo da verificiramo da selektirani objekat odgovara instanci klase *Box* a ne instanci klase *Diagram*.

Na liniji 288, pravimo kopiju reference izvornog događaja. Primijetite da je navedeno referenciranje važno zato što dozvoljava da dobijemo karakteristike objekta prije samog pomjeranja.

Na liniji 290, čuvamo vrijednosti veličine i lokacije kutije prije njenog pomjeranja unutar dijagrama. Nakon toga, linije 291 i 292 izračunavaju nove koordinate kutije.

Predstavljen je primjer situacije kada imamo kutiju lociranu na poziciji (130, 450) relativnu u odnosu na gornji lijevi ugao panela *diagram*. Pritisnuti klik miša se desio na poziciji (10, 40) relativno u odnosu na gornji lijevi ugao kutije. Pomjeranje miša je urađeno tako što se kutija pomjerila 50 grafičkih tačaka desno (po koordinati x). Prikazana je kalkulacija vrijednosti x :

$$x = 130 + 50 - 10 = 170$$

U realnosti, metod *mouseDragged* mora biti izvršen nekoliko puta u sekundi kako bismo dobili vrijednost varijable *x* našeg primjera. Kalkulacija *e.getX()* mora biti izvedena nekoliko puta kako bismo dobili konstantno prikazivanje nove pozicije kutije, tako da korisnik ima utisak da se radi o jednom pomjeranju, a ne o nekoliko sukcesivnih pomjeranja i prikazivanja kutije na različitim mjestima.

Kada se kurzor miša postavi izvan dijagrama (lijeva i gornja vrijednost), vrijednosti *x* i *y* postaju negativne vrijednosti. Ukoliko se desi ovaj slučaj navedene vrijednosti se postavljaju na nulu instrukcijama na linijama 293 i 294.

Onda kada se kalkulacija nove pozicije završi, instrukcija na liniji 295 konačno sačuva novu lokaciju koja zapravo mijenja poziciju kutije unutar dijagrama.

Metoda *mouseMoved*

Metoda *mouseMoved* konstituira drugu metodu koja je dio specifikacije interfejsa *MouseMotionListener*. Primjer prezentiran u ovom poglavlju ne sadrži kôd metode *mouseMoved*:

```
299: public void mouseMoved(MouseEvent e) { }
```

Ipak, želimo da napomenemo da ova metoda posjeduje karakteristike izvršavanja koje su slične metodi *mouseDragged*. Metoda *mouseMoved* reaguje kada jednostavno pokrećete kurzor miša iznad prostora koji je okupirao objekat. Dakle, nije potreban klik miša koji je u slučaju metode *mouseDragged* neophodan.

POGLAVLJE 8

BAR ZA ALATE

Tokom dizajna grafičkog korisničkog interfejsa važno je napraviti razliku između kreiranja objekata i korištenja istih. Kako bismo predstavili navedenu razliku, definisali smo bar za alate koji nam omogućava da radimo na dva navedena načina:

- 1- "Selekcija" (slika ruke)
- 2- "Dodavanje" (slika kutije)

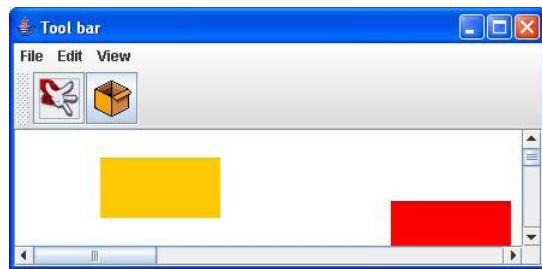
U prvom načinu rada, tj. kada korisnik klikne na sliku ruke smještene na baru za alate, moguće je pomjeranje kutije na željenu poziciju unutar dijagrama. Takođe je moguće deseletovati odnosno promijeniti "stanje" selektovane kutije klikom miša. Kada se klikne mišem na dijagram, a ne na kutiju, sve su kutije "deselektovane".

U drugom načinu rada, tj. kada korisnik klikne na sliku kutije smještene na baru za alate, kutije mogu biti dodate bilo gdje na dijagramu sa jednostavnim klikom miša.

Naravno, spomenuta dva načina rada bara za alate su međusobno isključiva.

Što se tiče slika koje su smještene na baru za alate, morali smo ikone staviti u odgovarajući dio eksterne memorije. Konsekventno, znajući da će se aplikacija eventualno instalirati na drugom računaru ili premjestiti u drugi direktorij na istom računaru, važno je

pokazati kako razviti takav sistem bez kontrolisanja puta direktorija tokom programiranja.



Specifikacija i kreiranje bara za alate

Potpis nove klase `AppToolBar` na liniji 195

```
195: public class AppToolBar extends JToolBar  
      implements ActionListener {
```

implementira interfejs `ActionListener`.

Kao što smo već vidjeli u prethodnom poglavlju, interfejs `ActionListener` preko jedine metode `actionPerformed` efikasno kontroliše događaj.

Definicija i relacija komponenata bara za alate

Na konkretni način, želimo da jedno dugme na baru za alate odredi događaj. Definicija dugmadi je prikazana na narednim linijama kôda:

```
203: private JButton selectButton = new  
      JButton(this.getIcon(SELECT_IMAGE));  
204: private JButton boxButton = new  
      JButton(this.getIcon(BOX_IMAGE));
```

Tokom definicije, želimo da određene slike pridružimo dugmadima bara za alate. Ove slike su zapravo specijalne datoteke kao što su *"gif"* datoteke. U našem primjeru, ove datoteke su smještene u poddirektorij nazvan *"images"* koji se nalazi pod direktorijem *mb/view*. Ime datoteke zajedno sa navedenim poddirektorijem je definisano u obliku konstanti `SELECT_IMAGE` i `BOX_IMAGE`:

```
200: private static final String  
      SELECT_IMAGE = "images/selectButton.gif";  
201: private static final String  
      BOX_IMAGE = "images/boxButton.gif";
```

Na prethodno prikazanoj liniji 203, vidjeli smo da definicija dugmeta `selectButton` poziva metod `getIcon`:

```
237: private ImageIcon getIcon(String anImage) {  
238:   URL url = this.getClass().  
      getResource(anImage);  
239:   ImageIcon imageIcon = new ImageIcon(url);  
240:   return imageIcon;  
241: }
```

Ova metoda zapravo uspostavlja referencu na resurs slike koji može biti korišten od strane Java razvojnog okruženja.

Pošto metoda `getImageIcon` na liniji 203 koristi kao parametar lokalni put direktorija sadržan u konstanti `SELECT_IMAGE` ovaj način nas primorava da smjestimo direktorij `"images"` u isti direktorij (`view`) gdje je smještena klasa koja je zapravo definisala metodu `getImageIcon`.

Sa ovakvim pristupom bilo koja datoteka korištena od strane aplikacije može biti dinamično locirana čak iako se aplikacija pomjeri na drugu fizičku lokaciju ili instalise na drugi računar.

Uvijek je bolje specificirati relativni direktorijski put a ne absolutni.

Ova dinamična procedura traženja direktorijskog puta datoteke nam pokazuje da nije neophodno specificirati čitav direktorijski put (absolutnu referencu). Preporučeno je korištenje relativne reference kako bi aplikacija bila što je moguće više fleksibilna.

Jednom kada se dugmad bara za alate definišu, ostaje da se urade sljedeće tri faze:

3 faze neophodne za dodjeljivanje dugmadi baru za alate.

1- Dodjeljivanje imena akcije za svako dugme bara za alate:

```
211: selectButton.setActionCommand("Select");  
214: boxButton.setActionCommand("Box");
```

Ovo interno dodjeljivanje će dozvoliti, preko `switch` instrukcije u metodi `actionPerformed`,

```
250: public void actionPerformed(ActionEvent e) {  
251:     String command = e.getActionCommand();  
252:     switch (this.getCommand(command)) {  
253:         case SELECT:  
254:             this.setSelectedButton(SELECT);  
255:             break;  
256:         case BOX:  
257:             this.setSelectedButton(BOX);  
258:             break;  
259:     }  
260: }
```

prepoznavanje izvora događaja, tj. događaja na jednom od dugmadi bara za alate.

2- Povezivanje dugmadi bara za alate sa Java interfejsom *ActionListener*:

```
212: selectButton.addActionListener(this);
```

```
215: boxButton.addActionListener(this);
```

3- Dodjeljivanje dugmadi baru za alate:

```
217: this.add(selectButton);
```

```
218: this.add(boxButton);
```

Osobine

U poglavlju 4 objasnili smo koncept osobina definisanih u klasi. Kako bismo bolje razumjeli ovaj fundamentalni koncept, želimo da pokažemo konkretniji način adekvatnog korištenja osobina.

U našoj aplikaciji osobina *selectedButton* klase *AppToolBar* je definisana na sljedeći način:

```
206: private int selectedButton = SELECT;
```

```
221: public int getSelectedButton() {
```

```
222:     return selectedButton;
```

```
223: }
```

```
224:
```

```
225: private void setSelectedButton(int aButton) {
```

```
226:     selectedButton = aButton;
```

```
227:     if (selectedButton == SELECT) {
```

```
228:         selectButton.setSelected(true);
```

```
229:         boxButton.setSelected(false);
```

```
230:     }
```

```
231:     else if (selectedButton == BOX) {
```

```
232:         boxButton.setSelected(true);
```

```
233:         selectButton.setSelected(false);
```

```
234:     }
```

```
235: }
```

Osobina *selectedButton* čuva stanje relativno u odnosu na trenutni način rada. Dva stanja su moguća:

Stanje Objasnjenje

1 "Selekcija"

2 "Dodavanje"

Vrijednosti ovih različitih stanja su konkretno definisani preko sljedećih konstanti:

```
197: public static final int SELECT = 1;  
198: public static final int BOX = 2;
```

Tokom definisanja bara za alate, linija 206 inicijalizira osobinu *selectedButton* na stanje 1. Ukoliko korisnik želi da doda nekoliko kutija on prvo mora da klikne na sliku kutije (dugme) smještenu unutar bara za alate. Ova akcija će modificirati stanje osobine *selectedButton* na vrijednost 2.

Detaljnije, pokretanje događaja će uzrokovati izvršavanje metode *actionPerformed*:

```
250: public void actionPerformed(ActionEvent e) {  
251:     String command = e.getActionCommand();  
252:     switch (this.getCommand(command)) {  
253:         case SELECT:  
254:             this.setSelectedButton(SELECT);  
255:             break;  
256:         case BOX:  
257:             this.setSelectedButton(BOX);  
258:             break;  
259:     }  
260: }
```

Prva instrukcija metode *actionPerformed* (linija 251) čuva, u lokalnoj varijabli *command*, vrijednost osobine *actionCommand* povezane sa izvorom događaja. Pošto ovaj objekat odgovara objektu *boxButton*, važno je napomenuti da je osobina *actionCommand* prethodno bila inicijalizirana na vrijednost "Box" tokom izvršavanja konstruktora:

```
214: boxButton.setActionCommand("Box");
```

U drugoj instrukciji metode *actionPerformed* sljedeća *switch* instrukcija

```
252: switch (this.getCommand(command)) {
```

poziva metodu *getCommand*:

```
243: private int getCommand(String aCommand) {  
244:     int action = -1;  
245:     if (aCommand.equals("Select"))  
         action = SELECT;  
246:     else if (aCommand.equals("Box"))  
         action = BOX;  
247:     return action;
```

```
248: }
```

Metoda `getCommand` dobija vrijednost "Box". Linija 247 konačno vraća vrijednost lokalne varijable `action`. U našem primjeru ova varijabla je jednaka vrijednosti 2 zbog dodjeljivanja konstante `BOX` varijabli `action` (linija 246).

Pošto je vrijednost vraćena pomoću metode `getCommand` jednaka vrijednosti 2, drugi blok instrukcije `switch` na liniji 252 će biti izvršen:

```
256: case BOX:  
257:     this.setSelectedButton(BOX);  
258:     break;
```

Na liniji 257, vrijednost 2 će biti poslana metodi `setSelectedButton`. Vrijednost ovog parametra će dozvoliti pokretanje tri instrukcije metode `setSelectedButton`:

```
226: selectedButton = aButton;  
  
232: boxButton.setSelected(true);  
233: selectButton.setSelected(false);
```

Na liniji 226 trenutna aktivnost će promijeniti način rada iz stanja 1 (način "selekcijske") u stanje 2 (način "dodavanje").

Linije 232 i 233 inicijaliziraju osobinu `selected` povezani sa svakim dugmetom bara za alate. Prije prvog klika miša na jedan od ovih objekata bara za alate, osobina `selected` ima vrijednost `false` zato što nijedna inicijalizacija u ovom smislu nije izvedena.

Primijetite da tokom prvog aplikacionog pokretanja, prvo lijevo dugme unutar bara za alate je osnovno dugme selektovano od strane Java programskog jezika (svijetlo plavi okvir oko ikone) čak iako je stanje osobine `selectedButton` jednako `false`. Ovaj način rezonovanja implicira da programer mora da inicijalizira vrijednost osobine `selectedButton` na vrijednost "1". Ukoliko bi osobina `selectedButton` bila jednaka vrijednosti "2", korisnik bi mogao odmah dodavati kutije na dijagram, iako je aktivno dugme bara za alate bilo ono sa slikom ruke.

Generalno, vrijednost osobine `selected` indicira trenutno stanje bara za alate (svijetlo plavi okvir oko slike).

Pozicioniranje bara za alate

Koordinator izgleda *BoxLayout* se koristi za pozicioniranje objekata unutar bara za alate.

Dodjeljivanje objekata *selectButton* i *boxButton* se vrši, u klasi *JToolBar*, uz pomoć osnovnog koordinatora *BoxLayout*.

Ovaj koordinator sekvencijalno organizira dugmad bara u liniji komponenti jednake veličine. Pozicija objekata unutar bara za alate je određena počevši od gornjeg lijevog ugla bara, lijevo na desno i odozgo prema dole.

Nakon što su objekti *selectButton* i *boxButton* stavljeni u objekat *appToolBar*, sljedeći korak se sastoji od smještanja bara za alate unutar aplikacionog okvira, tj. unutar objekta *cp*.

Pošto *cp* koristi osnovni koordinator *BorderLayout*, instrukcija na liniji 59

```
59: cp.add(appToolBar, "North"); // +
```

pozicionira bar za alate u regiju "North" objekta *cp*. Konsekventno, aplikacijski okvir nakon sljedećih instrukcija:

```
55: this.setJMenuBar(this.appMenu);
```

```
59: cp.add(appToolBar, "North"); // +
```

```
60: cp.add(new JScrollPane(diagram), "Center"); // &
```

sadrži meni bar i kontejner, koji je opet sastavljen od bara za alate na sjeveru i dijagrama na jugu.

Relacija između bara za alate i aplikacionog okvira

Pošto moguće operacije moraju biti uspostavljene u funkciji različitih načina rada koje obezbjeđuje bar za alate (način "dodavanje" ili "selekcije"), jasno je da prva relacija koja mora biti uspostavljena je relacija između objekata *diagram* i *appToolBar*. Ova relacija je omogućena sljedećim instrukcijama:

```
50: private Diagram diagram = new
Diagram(appToolBar);
```

Osobina *appToolBar* klase *Diagram* prima ovaj parametar u svom konstruktoru:

```
289: public Diagram(AppToolBar aAppToolBar) {
```

```
290:     this();  
291:     this.setAppToolBar(aAppToolBar);  
292: }
```

Metoda `setAppToolBar` je pozvana na liniji 291. Ova metoda je realizirana na sljedeći način:

```
298: public void setAppToolBar(AppToolBar  
aAppToolBar) {  
299:     appToolBar = aAppToolBar;  
300: }
```

Metoda dodjeljuje referencu objekta bara za alate osobini `appToolBar` klase *Diagram*. Onda kada smo ostvarili ovu relaciju, možemo nastaviti na verifikaciji operacija trenutnog načina rada kao što je to pokazano na narednim linijama:

```
366: if (appToolBar.getSelectedButton() ==  
        AppToolBar.BOX) { // +  
369: else if (appToolBar.getSelectedButton() ==  
            AppToolBar.SELECT) { // +  
374: if (appToolBar.getSelectedButton() ==  
        AppToolBar.SELECT) { // +
```

Ove tri operacije dozvoljavaju da prepoznamo koju aktivnost korisnik zaista želi da uradi. Bez prethodno ostvarene reference na objekat bara za alate bilo bi nemoguće verificirati trenutni način rada u klasi *Diagram*.

POGLAVLJE 9

O APLIKACIJI

Kada razvijate grafički korisnički interfejs korisno je da dodate meni za pomoć. Ovaj meni je dizajniran za nove korisnike kako bi se mogli na što lakši način upoznati sa operacijama koje nudi aplikacija.

Nekoliko opcija može sadržavati ovaj meni. U ovom poglavlju ćemo prezentirati opciju "About...". Klikajući na ovu opciju menija možemo vidjeti aplikacijski prozor nazvan "About...". Ovaj prozor je postavljen preko osnovnog okvira aplikacije.

Unutar ovog prozora ćemo vidjeti sliku zajedno sa par linija teksta. Dugme "OK" dozvoljava zatvaranje ovog prozora. Korištenje aplikacionih funkcija je blokirano dok se ne klikne mišem na navedeno dugme.

Pored menija za pomoć, dodati ćemo meniju "Edit" opciju "Delete Selected Boxes". Takođe će svaka kutija imati vlastiti naziv koji će se moći mijenjati.



Održavanje meni bara

Konstante opcija u meniju

U klasi *AppMenu* morali smo modificirati vrijednost konstanti opcija menija prezentiranih u prethodnim poglavljima. Kako bismo dodali opciju "About..." (meni "Help") i opciju "Delete Selected Boxes" (meni "Edit") dodate su dvije konstante, *ABOUT* i *DELETE*, i modifikovane su prethodne konstante sa novim brojevima:

```
95: private static final int EXIT      = 11;    // &
96: private static final int SELECT   = 21;    // &
97: private static final int DESELECT = 22;    // &
98: private static final int DELETE   = 23;    // +
99: private static final int HIDE     = 31;    // &
100: private static final int SHOW    = 32;    // &
101: private static final int ABOUT   = 41;    // +
```

Konstruktor *AppMenu*

Pored konstanti, dodavanje opcija "About..." (meni "Help") i "Delete Selected Boxes" (meni "Edit") zahtijeva dodatni kôd unutar konstruktora meni bara:

```
108: private JMenuItem menuEditDelete =
       new JMenuItem ("Delete Selected Boxes"); // +
112: private JMenu menuHelp = new JMenu("Help"); // +
113: private JMenuItem menuHelpAbout =
       new JMenuItem("About..."); // +
126: menuEditDelete.setActionCommand("Delete"); // +
127: menuEditDelete.addActionListener(this); // +
134: menuHelpAbout.setActionCommand("About"); // +
135: menuHelpAbout.addActionListener(this); // +
140: menuEdit.addSeparator(); // +
141: menuEdit.add(menuEditDelete); // +
144: menuHelp.add(menuHelpAbout); // +
149: this.add(menuHelp); // +
```

Metoda `addSeparator` na liniji 140 dodaje horizontalnu liniju između opcije "Deselect Boxes" i opcije "Delete Selected Boxes".

Metoda `getCommand`

Kako bismo uključili opcije "About..." i "Delete Selected Boxes" dvije `if` instrukcije su dodata (linije 184 i 187) unutar metode `getCommand`:

```
179: private int getCommand(String aCommand) {  
180:     int action = -1;  
181:     if (aCommand.equals("Exit"))  
182:         action = EXIT;  
183:     else if (aCommand.equals("Select"))  
184:         action = SELECT;  
185:     else if (aCommand.equals("Deselect"))  
186:         action = DESELECT;  
187:     else if (aCommand.equals("Delete"))  
188:         action = DELETE; // +  
189:     else if (aCommand.equals("Hide"))  
190:         action = HIDE;  
191:     else if (aCommand.equals("Show"))  
192:         action = SHOW;  
193:     else if (aCommand.equals("About"))  
194:         action = ABOUT; // +  
195:     return action;  
196: }
```

Primijetite da je korištenje ključnih riječi "`else if`" na jednoj liniji ekvivalentno korištenju ključnih riječi stavljenih na dvije odvojene linije:

```
if (aCommand.equals("Exit"))           action = EXIT;  
else  
    if (aCommand.equals("Select"))      action = SELECT;  
    else  
        if (aCommand.equals("Deselect")) action = DESELECT;  
        else  
            if (aCommand.equals("Delete")) action = DELETE;  
            else  
                if (aCommand.equals("Hide"))   action = HIDE;  
                else  
                    if (aCommand.equals("Show"))  action = SHOW;  
                    else  
                        if (aCommand.equals("About")) action = ABOUT;
```

Izbor korištenja je ostavljen programerima iako preporučujemo korištenje ”*else if*” instrukcija na jednoj liniji ukoliko se one uzastopno ponavljaju kao što je pokazano na linijama 181 do 187.

Metoda *deleteSelectedBoxes*

Dva bloka *case* instrukcija su dodata *switch* instrukciji metode *actionPerformed*:

```
204: case DELETE:  
205:     appFrame.getDiagram().deleteSelectedBoxes();  
206:     break;  
  
215: case ABOUT:  
216:     this.displayAbout();  
217:     break;
```

Razvijanje metode *deleteSelectedBoxes*, korištene u *case* bloku na liniji 205, u osnovi prati ista pravila dizajna kao kod metode *hideSelectedBoxes* unutar klase *Diagram*. Sljedeće linije prikazuju metodu *deleteSelectedBoxes* koja je takođe locirana u klasi *Diagram*:

```
498: public void deleteSelectedBoxes() {  
499:     Component[] components = this.getComponents();  
500:     Box box;  
501:     for (int i = 0; i < components.length; i++) {  
502:         if (components[i] instanceof Box) {  
503:             box = (Box) components[i];  
504:             if (box.isSelected()) {  
505:                 this.remove(box);  
506:             }  
507:         }  
508:     }  
509:     this.repaint();  
510: }
```

Jedina razlika između ove metode i metode *hideSelectedBoxes* je na liniji 505 gdje smo zamijenili instrukciju *box.setVisible(false)* sa instrukcijom *this.remove(box)*.

Metoda *repaint* je korištena za ponovno prikazivanje grafičkog korisničkog interfejsa.

Metoda `remove` na liniji 505 briše referencu objekta `box` lociranog u objektu `diagram`. Pošto se instrukcija na liniji 505 nalazi pod `for` petljom svi prethodno selektovani objekti biće obrisani. Metoda `repaint` (linija 509) se koristi za ponovno prikazivanje odnosno osvježavanje dijagrama.

Metoda `displayAbout`

Metoda `displayAbout`:

```
166: private void displayAbout() {  
167:     AppAbout about = new AppAbout(appFrame);  
168:     Dimension aboutSize = about.  
getPreferredSize();  
169:     Dimension frameSize = appFrame.getSize();  
170:     Point frameLocation = appFrame.getLocation();  
171:     int x = ((frameSize.width - aboutSize.width)  
               / 2) + frameLocation.x;  
172:     int y = ((frameSize.height - aboutSize.height)  
               / 2) + frameLocation.y;  
173:     about.setLocation(x, y);  
174:     about.setModal(true);  
175:     about.setVisible(true);  
176: }
```

Linija 167 definiše lokalnu varijablu `about`. Vratićemo se u narednim sekcijama na objašnjavanje korištenja ovog objekta.

Metoda `getPreferredSize` daje preferiranu veličinu kontejnera koju su zadali tvorci Java programskog jezika.

Metoda `getPreferredSize` na liniji 168 daje "preferiranu" veličinu kontejnera `about`. Efektivno, tvorci Java programskog jezika su definisali preferirane dimenzije svakog tipa kontejnera. U našem primjeru smo dobili preferiranu širinu i dužinu prozora `about`. Lokalna varijabla `aboutSize` će služiti za kalkulaciju koordinata `x` i `y` relativnih u odnosu na gornji lijevi ugao aplikacijskog prozora. U narednoj liniji, u odnosu na liniju 168, se koristi metoda `getSize` za dobijanje trenutne vrijednosti širine i dužine prozora.

Linije 171 i 172 izračunavaju poziciju `about` prozora u odnosu na aplikacioni prozor.

Instrukcija na liniji 173 pozicionira gornji lijevi ugao prozora na koordinate `x` i `y` dobijene gornjom kalkulacijom.

Prozor je **modalni** kada korisnik ne može pristupiti drugim prozorima aplikacije.

Metoda *setModal* na liniji 174 kontroliše operacije kada se prozor prikaže na ekranu. U našem primjeru želimo primorati korisnika da zatvori prozor prije nego bilo šta drugo uradi. Predavajući parametar *true* metodi *setModal*, ne dozvoljavamo korištenje aplikaciono dok korisnik ne klikne na dugme "OK". Ovaj tip operacije nazivamo modalni prozor. Suprotno, kažemo da je prozor nemodalni kada on može ostati otvoren u isto vrijeme sa drugim prozorima dozvoljavajući korisniku da se prebacuje sa jednog na drugi prozor.

Metoda *setVisible* na liniji 175 daje vrijednost *true* osobini *visible*. Ova promjena prikazuje *about* prozor na korisničkom ekranu.

Razvijanje prozora

Kao što smo već napomenuli, instrukcija na liniji 167

```
167: AppAbout about = new AppAbout(appFrame);
```

kreira objekat klase *AppAbout*. Potpis klase je sljedeći:

```
234: public class AppAbout extends JDialog
      implements ActionListener {
```

JDialog klasa definiše prozor dijaloga.

Prije prelaska na detalje kôda objekta *about*, navećemo vrlo važne koncepte u vezi sa mehanizmom izuzetaka koji nam dozvoljavaju da imamo kontrolu nad greškama.

Izuzeci

Svi programi mogu da u toku izvođenja izazovu greške. Java posjeduje efikasan način kontrolisanja takvih grešaka omogućavajući mehanizam za lokalizaciju i procesiranje takvih situacija.

Izuzetak je objekat kreiran od strane Java programskog jezika na mjestu "neuobičajene" situacije proizvedene tokom izvršavanja aplikacije.

Izuzetak je ništa više nego objekat kreiran od strane Java programskog jezika opisujući nastalu neuobičajenu situaciju. Konkretno, izuzetak nastaje tokom izvršavanja instrukcija. Izuzeci se mogu detektovati a zatim manipulisati. Kada

mehanizam detekcije izuzetaka nije implementiran aplikacija abnormalno terminira. Tada se prikazuje poruka o indikaciji koji izuzetak je terminirao aplikaciju i gdje.

Dva toka izvršavanja:

- normalni tok;
- tok izuzetaka.

Manipulacija izuzecima je veoma važna tokom razvijanja aplikacionog softvera, jer dozvoljava programeru da podijeli program na normalni tok izvršavanja i tok izvršavanja kada nastane izuzetak.

Za ilustraciju mehanizma otkrivanja izuzetaka predstavljen je primjer:

```
269: public AppAbout (AppFrame parent) {  
270:     super (parent);  
271:     try {  
272:         this.init();  
273:     }  
274:     catch (Exception e) {  
275:         System.out.println(e.getMessage());  
276:     }  
277:     this.pack();  
278: }
```

U ovom primjeru prikazan je konstruktor klase *AppAbout*. Kada počne izvršavanje konstruktora, linija 270 poziva konstruktor roditeljske klase *JDialog*. Različiti konstruktori klase *JDialog* zahtijevaju kao parametar referencu na objekat okvira u kojem se prikazuje sam prozor dijaloga.

Zatim se izvršava instrukcija *try* i ukoliko se ne desi izuzetak u metodi *init*, završava se blok instrukcija u instrukciji *try* i kontrola izvršavanja se prenosi na liniju 277, odnosno naredna metoda koja se poziva je metoda *pack*.

Metoda **pack**: dozvoljava Java programskom jeziku izračunavanje "preferirane" veličine kontejnera koji se nalazi unutar drugog kontejnera.

Primijetite da metoda *pack* dozvoljava Java programskom jeziku izračunavanje preferirane veličine prozora koji se nalazi unutar drugog kontejnera. Izračunata veličina je konsekventno manja od veličine prvog kontejnera.

Što se tiče same metode *init*, vratićemo se malo kasnije objašnjenu njenih karakteristika.

Svaka **catch** instrukcija manipulira specifičnim izuzetkom koji se desio tokom izvršavanja jedne od instrukcija *try* bloka.

Takođe primijetite blok *catch* instrukcija na liniji 274 koje su dio *try* instrukcije. Ukoliko se desi izuzetak tokom izvršavanja metode *init*, pozvane na liniji 272, kontrola toka izvršavanja se prenosi na instrukciju *catch*. U drugom slučaju, kontrola toka se normalno nastavlja sa izvršavanjem na liniji 277.

Sintaksa instrukcije *catch*, koja je integralni dio *try* instrukcije, je:

```
catch (ExceptionClass localVariable)
```

Ukoliko je izuzetak instanca klase indicirane u *catch* bloku, biće izvršene instrukcije koje su dio ovog bloka.

Ukoliko je detektirani izuzetak instanca klase navedene u *catch* instrukciji taj blok će biti izvršen. U našem primjeru samo jedna instrukcija je dio ovog bloka:

```
275: System.out.println(e.getMessage());
```

Metoda *getMessage* vraća tekst koji objašnjava uzrok nastanka izuzetka. Ova metoda je dio klase *Throwable* iz koje su svi objekti izuzetaka izvedeni.

Nakon izvršene instrukcije na liniji 275, Java vraća kontrolu normalnom toku izvršavanja. Konkretno, to su instrukcije iza *try catch* instrukcije. U našem primjeru kontrola se vraća metodi *pack* na liniji 277.

Nekoliko *catch* blokova mogu biti dio *try* instrukcije. Parametar koji je objekat klase izuzetka određuje koji izuzetak obrađujemo. Ova procedura dozvoljava izvršavanje instrukcija u funkciji tipa izuzetka.

Kada se otkrije izuzetak koji nije integralni dio *try* instrukcije, kontrola se prenosi metodi višeg nivoa (pozivna metoda) i ova procedura se propagira do *main* metode. I ukoliko ni tada ne postoji neki mehanizam detekcije, aplikacija terminira abnormalno i prikazuje se poruka izuzetka.

Sljedeća *init* metoda demonstrira mehanizam propagacije izuzetka:

```
280: private void init() throws Exception {  
281:     this.setTitle("About...");  
282:     this.setResizable(false);  
283:  
284:     Container cp = this.getContentPane();  
285:     cp.setLayout(borderLayout1);  
286:     cp.add(northPanel, "North");  
287:     cp.add(southPanel, "South");  
288:     cp.add(centerPanel, "Center");  
289:     cp.add(westPanel, "West");  
290:     cp.add(eastPanel, "East");  
291: }
```

```
292:     centerPanel.setLayout(borderLayout2);
293:     centerPanel.add(imagePanel, "North");
294:     centerPanel.add(textPanel, "South");
295:
296:     imagePanel.setBackground(Color.red);
297:     imagePanel.add(imageLabel);
298:
299:     textPanel.setLayout(gridLayout);
300:     textPanel.setBackground(Color.orange);
301:     gridLayout.setRows(5);
302:     gridLayout.setColumns(1);
303:     label0.setText(empty);
304:     label1.setText(product);
305:     label2.setText(version);
306:     label3.setText(date);
307:     label4.setText(author);
308:     textPanel.add(label0);
309:     textPanel.add(label1);
310:     textPanel.add(label2);
311:     textPanel.add(label3);
312:     textPanel.add(label4);
313:
314:     southPanel.setLayout(flowLayout);
315:     southPanel.add(button);
316:     button.addActionListener(this);
317: }
```

U ovom primjeru ne vidimo da je instrukcija *try* dio integralnog koda *init* metode. Kada se desi izuzetak kontrola se vraća pozivnoj metodi koja je zapravo *AppAbout* konstruktor.

Primijetite da Java zahtijeva od programera eksplicitnu deklaraciju, u potpisu njegove metode, da želi koristiti navedeni mehanizam propagacije. Na primjer, potpis metode *init*

```
280: private void init() throws Exception {
```

sadrži riječ *throws* koja forsira metodu koja ju je pozvala, tj. pozivnu metodu da ili uhvati izuzetak ili napravi propagaciju sa izuzetkom. U našem slučaju, posto nema *try* instrukcije unutar metode *init*, Java propagira analizu izuzetka pozivnoj metodi odnosno konstruktoru *AppAbout*.

Nekoliko klasa izuzetaka mogu biti dio verifikacije. Ovdje, u metodi *init*, koristimo samo jednu koja pokriva, u smislu hijerarhije, nekoliko drugih klasa izuzetaka, a to je klasa *Exception*.

Primijetite da je mehanizam izuzetka dodat u metodi *getImageIcon* koja se nalazi u klasi *AppToolBar* prethodnog poglavlja. Prikazana je metoda korištena u ovom poglavlju:

```
391: private ImageIcon getImageIcon(String anImage) {  
392:     // {+  
393:     URL url = null;  
394:     ImageIcon imageIcon = null;  
395:     try {  
396:         // +}  
397:         url = this.getClass().getResource(anImage);  
398:         imageIcon = new ImageIcon(url);  
399:     // {+  
400:     }  
401:     catch (Exception e) {  
402:         System.out.println(e.getMessage() +  
        " // no URL");  
403:     }  
404:     // +}  
405:     return imageIcon;  
406: }
```

Ova metoda je unutar klase *AppAbout* korištena na isti način.

Detalji prozora "About"

Linija 281:

```
281: this.setTitle("About...");
```

daje prozoru (*this*) naslov "About...". Vrijednost *false* je poslana metodi *setResizable* u sljedećoj liniji

```
282: this.setResizable(false);
```

i ona onemogućava korisnika da modifcira veličinu prozora. Definisano je neko-liko objekata klase *JPanel* u aplikacionom prozoru:

```
244: private JPanel northPanel = new JPanel();
```

```
245: private JPanel southPanel = new JPanel();  
246: private JPanel centerPanel = new JPanel();  
247: private JPanel westPanel = new JPanel();  
248: private JPanel eastPanel = new JPanel();  
249:  
250: private JPanel imagePanel = new JPanel();  
251: private JPanel textPanel = new JPanel();
```

Klasa *JPanel* je ništa drugo nego grafički kontejner korišten veoma često za definisanje regija prozora:

Linija 285

```
285: cp.setLayout(borderLayout1);
```

dodjeljuje koordinator *BorderLayout* objektu *cp*. Referenca na *BorderLayout* je uradena pomoću prethodno definisanog objekta *borderLayout1*:

```
264: private BorderLayout borderLayout1 =  
      new BorderLayout();
```

Jednom kada je koordinator definisan moguće je dodjeljivanje panela regijama prozora:

```
286: cp.add(northPanel, BorderLayout.NORTH);  
287: cp.add(southPanel, BorderLayout.SOUTH);  
288: cp.add(centerPanel, BorderLayout.CENTER);  
289: cp.add(westPanel, BorderLayout.WEST);  
290: cp.add(eastPanel, BorderLayout.EAST);
```

Ovo dozvoljava pored ostalog i uokviravanje ivica prozora sivom bojom. Dugme "OK" zatim zauzima regiju *SOUTH*. Kako biste mogli dodati sliku i tekst iznad dugmeta neophodno je razviti regiju *CENTER* uz ponovnu pomoć koordinatora *BorderLayout*:

```
292: centerPanel.setLayout(borderLayout2);
```

Objekat *borderLayout2* je prethodno definisan na sljedeći način:

```
265: private BorderLayout borderLayout2 =  
      new BorderLayout();
```

Slika će okupirati regiju *NORTH*, a tekst regiju *SOUTH*:

```
293: centerPanel.add(imagePanel, BorderLayout.NORTH);  
294: centerPanel.add(textPanel, BorderLayout.SOUTH);
```

Radna površina kontejnera namijenjenog za sliku postaje crvene boje:

```
296: imagePanel.setBackground(Color.RED);
```

Sljedeći korak se sastoji od dodjeljivanja slike kontejneru *imagePanel*:

```
297: imagePanel.add(imageLabel);
```

Ova slika je referencirana od strane objekta *imageLabel*:

```
260: private JLabel imageLabel = new
JLabel(imageIcon);
```

Klasa *JLabel* dozvoljava da definišete objekte koji služe za prikazivanje teksta ili slike. Definicija objekta *imageLabel*, koji predstavlja kontejner slike, referencira objekat *imageIcon*. Ovaj zadnji navedeni objekat nam finalno dozvoljava uspostavljanje relacije sa slikom čiji je put specificiran konstantom *IMAGE_PATH*:

```
253: private ImageIcon imageIcon =
this.getImageIcon(IMAGE_PATH);
```

```
236: private static final String IMAGE_PATH =
"images/aboutMagic.gif";
```

Koordinator izgleda *GridLayout*.

Objekat *textPanel* je korišten radi prikazivanja pet linija teksta pozicioniranih ispod slike. Ovaj kontejner koristi koordinator *GridLayout*:

```
266: private GridLayout gridLayout = new
GridLayout();
```

```
299: textPanel.setLayout(gridLayout);
```

Koordinator izgleda *GridLayout* nam pomaže da organizujemo sadržaj objekta *textPanel* u matricu čije su ćelije jednake veličine i gdje svaka ćelija predstavlja komponentu ili kontejner. Pozicija objekata unutar *textPanel* efektivno startuje od gornjeg lijevog ugla objekta *textPanel*, idući lijevo ka desno (kolona po kolona) i od vrha prema dnu (red po red). Ovaj zadatak je izведен po principu objekat po objekat. Prvo je pozadina kontejnera obojena bojom narandže, a zatim je uspostavljena matrica od pet redova i jedne kolone:

```
300: textPanel.setBackground(Color.ORANGE);
```

```
301: gridLayout.setRows(5);
```

```
302: gridLayout.setColumns(1);
```

Kako bismo dodali tekst u kontejner *textPanel* definisali smo pet objekata tipa *JLabel*:

```
255: private JLabel label0 = new JLabel();  
256: private JLabel label1 = new JLabel();  
257: private JLabel label2 = new JLabel();  
258: private JLabel label3 = new JLabel();  
259: private JLabel label4 = new JLabel();
```

Kao što smo već napomenuli, klasa *JLabel* dozvoljava definiciju objekata koji služe za prikazivanje teksta ili slike. U ovom kontekstu želimo da prikažemo pet linija teksta:

```
238: private String empty    = "";  
239: private String product = " Magic Boxes";  
240: private String version = " Java";  
241: private String date   = " 1997 - 2004";  
242: private String author  = " Dzenan Ridjanovic";
```

Metodom *setText* dodjeljujemo tekst prethodno definisanim objektima:

```
303: label0.setText(empty);  
304: label1.setText(product);  
305: label2.setText(version);  
306: label3.setText(date);  
307: label4.setText(author);
```

Ostaje nam da dodamo tekst objektu *textPanel*:

```
308: textPanel.add(label0);  
309: textPanel.add(label1);  
310: textPanel.add(label2);  
311: textPanel.add(label3);  
312: textPanel.add(label4);
```

Koordinator izgleda *FlowLayout*.

Iako objekat *southPanel* okupira u objektu *cp* regiju *SOUTH* organizovanu pomoću koordinatora *BorderLayout*, sljedeća instrukcija dodaje ovoj regiji novi koordinator *FlowLayout*:

```
314: southPanel.setLayout(flowLayout);
```

```
267: private FlowLayout flowLayout =
```

```
new FlowLayout();
```

Pomoću koordinatora *FlowLayout* komponente se dodaju u centru jedna po jedna, lijevo ka desno i odozgo prema dole. Koordinator koristi metodu *preferredSize* za izračunavanje veličina pojedinih komponenti. U jednoj liniji koordinator dodaje onoliko komponenti koliko je to moguće, a zatim nastavlja sa dodavanjem na sljedećoj liniji. Generalno se koordinator *FlowLayout* koristi za prikazivanje dugmadi.

Dugme *button*:

```
262: private JButton button = new JButton("OK");
```

Naziv "OK" je tokom kreiranja dodat dugmetu. Sljedeća linija dodaje objekat *button* kontejneru *southPanel* koji je organizovan pomoću koordinatora *FlowLayout*:

```
315: southPanel.add(button);
```

Kako bismo dozvolili da klik na dugme "OK" zatvori prozor *about* moramo koristiti mehanizam događaja. Konkretno, mehanizam događaja se obezbjeđuje implementiranjem interfejsa *ActionListener* u klasi *AppAbout* na sljedeći način:

```
234: public class AppAbout extends JDialog  
      implements ActionListener {
```

Instrukcija na narednoj liniji

```
316: button.addActionListener(this);
```

specificira objekat *button* kao izvor događaja koji će obraditi implementirana metoda *actionPerformed* interfejsa *ActionListener* locirana u objektu *this* (*about*).

Kada se desi klik miša, Java kreira instancu klase *ActionEvent* i šalje isti kao parametar metodi *actionPerformed*:

```
332: public void actionPerformed(ActionEvent e) {  
333:     if (e.getSource() == button) {  
334:         this.setVisible(false);  
335:         this.dispose();  
336:     }  
337: }
```

Unutar metode *actionPerformed* linija 333 verificira da li izvor događaja odgovara objektu *button*. Ukoliko odgovara, sljedeća linija dodjeljuje vrijed-

nost *false* osobini *visible*, a zatim slijedi izvršavanje metode *dispose*. Ova metoda oslobađa sve resurse zauzete od strane prozora.

Naziv kutije

Naredne linije

```
597: public static final Color TITLE_COLOR =
      Color.WHITE; // +
598: public static final String TITLE_TEXT = "Box"; // +
603: private JTextField titleText =
      new JTextField(TITLE_TEXT); // +
611: titleText.setBackground(TITLE_COLOR);
612: this.setLayout(new BorderLayout());
613: this.add(titleText, "North");
614: this.doLayout();
```

su dodate klasi *Box*. Prvo se na liniji 611 definiše boja podloge koja će se koristiti za naziv. Zatim se definiše koordinator *BorderLayout* kako bi se, u narednoj liniji, odredila pozicija naslova unutar kutije koja će biti smještena u regiju *NORTH*. Osnovni naslov je sadržan u objektu *TITLE_TEXT*. Metoda *doLayout* na liniji 614 prikazuje naslov unutar kutije.

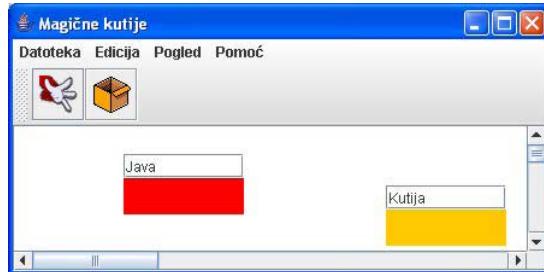
Jednom kada se kutija prikaže biće moguće modificiranje njenog naslova. Sa jednim klikom miša u prostor zauzet od strane naslova biće prikazan kurzor koji će dozvoliti lagano modificiranje naslova. Sa duplim klikom možemo selektovati naslov kutije i obrisati ga u potpunosti.

POGLAVLJE 10

INTERNACIONALIZACIJA APLIKACIJE

U ovom poglavlju želimo da pokažemo kako razviti internacionalnu verziju naše aplikacije (apleta). Iako je korišten mali broj prirodnih jezika, koncepti koji su prezentirani u ovom poglavlju mogu biti korišteni za bilo koje kulturno područje.

Prezentiraćemo takođe dodatnu funkcionalnost: napomene ispod svake opcije bara za alate, koje znatno olakšavaju rad manje iskusnim korisnicima.



Personalizacija grafičkog korisničkog interfejsa prema prirodnom jeziku

Definisanje datoteke prema specifičnom korisniku

Vrlo često se desi da grafički korisnički interfejs razvijen za veliki broj korisnika zahtijeva određene specijalizacije prema specifičnom korisniku. Na primjer, jezik korisnika može zahtijevati različit prikaz menija i njegovih opcija.

Java obezbjeđuje mehanizam selekcije klasa prema prirodnom jeziku korisnika.

U našem primjeru, grupisali smo sve riječi koje će biti prikazane na grafičkom korisničkom interfejsu kada aplikacija (ili applet) bude isporučena krajnjem korisniku. Za korisnika iz engleskog govornog područja, navedene riječi će biti sadržane u parovima ključ=vrijednost u datoteci *StringRes_en.properties*:

```
about=About...
aboutTitle=About
appTitle=Magic Boxes
author=Dzenan Ridjanovic
boxTip>Create a box.
boxTitle=Box
date=1997 - 2005
deleteSelectedBoxes>Delete Selected Boxes
deselectBoxes=Deselect Boxes
edit>Edit
exit=Exit
file=File
help=Help
hideSelectedBoxes=Hide Selected Boxes
ok=OK
empty=
product=Magic Boxes
selectBoxes>Select Boxes
selectTip>Select a box.
showSelectedBoxes>Show Selected Boxes
version=Java
view=View
```

Što se tiče korisnika iz francuskog govornog područja koristićemo datoteku *StringRes_fr.properties*:

```
about=À propos de...
aboutTitle=À propos de
appTitle=Boîtes magiques
author=Dzenan Ridjanovic
boxTip=Créer une boîte.
boxTitle=Boîte
date=1997 - 2005
deleteSelectedBoxes=Supprimer les boîtes
sélectionnées
deselectBoxes=Désélectionner les boîtes
edit=Édition
exit=Sortie
file=Fichier
help=Aide
hideSelectedBoxes=Cacher les boîtes sélectionnées
ok=OK
empty=
product=Boîtes magiques
selectBoxes=Sélectionner les boîtes
selectTip=Sélectionner une boîte.
showSelectedBoxes=Montrer les boîtes sélectionnées
version=Java
view=Vue
```

Konačno, u slučaju korisnika iz Bosne i Hercegovine koristićemo datoteku *StringRes_ba.properties*:

```
about=O...
aboutTitle=O
appTitle=Magi\u010dne kutije
author=D\u017eenan Ri\u0111anovi\u0107
boxTip=Kreiraj kutiju.
boxTitle=Kutija
date=1997 - 2005
deleteSelectedBoxes=Izbri\u0161i izabrane kutije
deselectBoxes=Anuliraj izabrane kutije
edit=Edicija
exit=Izlaz
file=Datoteka
help=Pomo\u0107
hideSelectedBoxes=Sakrij izabrane kutije
ok=OK
empty=
```

```
product=Magi\u010dne kutije
selectBoxes=Izaberi kutije
selectTip=Izaberi kutiju.
showSelectedBoxes=Poka\u017ei izabrane kutije
version=Java
view=Pogled
```

Specijalni karakteri su definisani prema Unicode (\u) sekvenci:

\u0106
\u0107
\u010c
\u010d
\u0110
\u0111
\u0160
\u0161
\u017d
\u017e

Moje ime, Dženan Riđanović, postaje "D\u017eenan Ri\u0111anovi\u0107".

Detaljnije informacije o Unicode načinu kodiranja možete potražiti na web adresi www.unicode.org.

Pored navedenih kreira se i osnovna datoteka, *StringRes.properties*.

Za svaku datoteku postoji odgovarajuća klasa koja podržava datoteku. Klasa *StringRes* će biti korištena onda kada korisnik ne pripada nijednom području koje smo prethodno definisali (*StringRes_en*, *StringRes_fr*, i *StringRes_ba*).

Identifikacija korisnika

Na nekoliko linija unutar aplikacije pozivamo metod *getPara* kako bismo dobili aplikacione parametre. Na primjer, imamo sljedeću instrukciju:

```
52: private Para para = Para.getPara();
```

Metodu *getPara* definiše klasa *Para*.

Specifikacija jezika korisnika je urađena na liniji 735. U ovom slučaju kreiran je objekat *locale* sa parametrom "ba". U klasi *Locale* dostupna je nekolicina konstanti za različite jezike. Npr. imamo sljedeće konstante:

- CHINESE;
- FRENCH;
- GERMAN;
- ITALIAN;
- KOREAN;
- SIMPLIFIED_CHINESE;
- TRADITIONAL_CHINESE.

Primijetite da je objekat *Locale* korišten kako bismo mogli odrediti geografsku, političku ili kulturnu regiju eventualnog korisnika. Koriste se dva konstruktora za definisanje objekta klase *Locale*:

```
Locale(String language)
Locale(String language, String country)
```

Prvi argument oba konstruktora odgovara kôdu jezika. Npr. kôd "fr" se koristi za korisnike francuskog govornog područja i "en" za korisnike engleskog govornog područja. Drugi parametar odgovara kôdu zemlje.

```
718: package mb.config;
719:
720: import java.util.Locale;
721: import java.util.ResourceBundle;
722: import java.util.MissingResourceException;
723: import javax.swing.ImageIcon;
724: import java.net.URL;
725:
726: public class Para {
727:
728:     private static Para para;
729:
730:     private static Locale locale;
731:     private static ResourceBundle
interStringDispenser;
732:
733:     {
734:         try {
735:             locale = new Locale("ba");
736:             // locale = Locale.ENGLISH;
737:             // locale = Locale.FRENCH;
738:             interStringDispenser =
ResourceBundle.getBundle("mb/config.
StringRes", locale);
739:     }
```

```
740:         catch (Exception e) {
741:             System.out.println(e.getMessage());
742:         }
743:     }
744:
745:     private Para() {
746:         super();
747:     }
748:
749:     public static Para getPara() {
750:         if (para == null) {
751:             para = new Para();
752:         }
753:         return para;
754:     }
755:
756:     public String getT(String key) {
757:         String lookFor = null;
758:         try {
759:             lookFor = interStringDispenser.getString(key);
760:         }
761:         catch (MissingResourceException e) {
762:             System.out.println(e.getMessage() +
763:                 " // Missing string: " + key);
764:             lookFor = "Missing string: " + key;
765:         }
766:         return lookFor;
767:     }
768:
769:     public ImageIcon getImageIcon(String anImage) {
770:         URL url = null;
771:         ImageIcon imageIcon = null;
772:         try {
773:             url = this.getClass().getResource(anImage);
774:             imageIcon = new ImageIcon(url);
775:         }
776:         catch (Exception e) {
777:             System.out.println(e.getMessage() + " // "
no URL");
778:         }
779:         return imageIcon;
780:     }
```

```
779:  
780: }  
781:
```

Metoda `getBundle`, u predefinisanoj klasi `ResourceBundle` na liniji 738, dobija referencu na klasu na osnovu imena osnovne klase (`StringRes`) koje se povezuje sa karakterom `"_"` i onda sa vrijednosti objekta `locale`. U našem primjeru, pošto se radi o korisniku iz BiH, objekat `interStringDispenser` tipa `ResourceBundle` dobija referencu na klasu `StringRes_ba` koja koristi `StringRes_ba.properties` datoteku.

Metoda `getBundle` traži odgovarajuću klasu jezika. Ukoliko ova klasa ne postoji, zadnji sufiks se briše iz imena klase i pokreće se ponovno pretraživanje. Ista procedura se odnosi i na predhodni sufiks i tako redom. Prikazani su traženi sufksi:

- 1- Base class + `"_"` + language + `"_"` + country
- 2- Base class + `"_"` + language
- 3- Base class

U našem primjeru, traženje odgovara drugom nivou pošto ne postoji sufiks sa imenom zemlje. Ukoliko korisnik nije iz područja čiji sufksi odgovaraju `"en"`, `"fr"` i `"ba"` onda se koristi osnovna klasa `StringRes`. Primijetite da je lokalni put direktorija `mb/config` dodat prije imena ove osnovne klase kao na liniji 738..

Traženje teksta

Nakon što smo kreirali objekat, zavisno od jezika korisnika na liniji 735, koristimo metodu `getT` klase `Para` da bismo našli vrijednost teksta po internom ključu. Na primjer, ključ `"file"` odgovara vrijednosti `"File"` u datoteci `StringRes_en.properties` (korisnik iz engleskog govornog područja) odnosno vrijednosti `"Fichier"` u datoteci `StringRes_fr.properties` (korisnik iz francuskog govornog područja). Konkretno, sljedeća linija kôda

```
57: appTitle = para.getT("appTitle");
```

poziva metodu `getT`. Pretpostavljajući da se radi o engleskom korisniku sljedeća instrukcija

```
759: lookFor =
        interStringDispenser.getString(key);
```

pretražuje datoteku *StringRes_en.properties* i vraća vrijednost "Magic Boxes" objektu *lookFor*.

Kôd prethodnog poglavlja je modifciran kako bi se mogao koristiti mehanizam personalizacije prema jeziku korisnika. Na primjer, klasa *Box* sadrži sljedeće linije:

```
659: private static String boxTitle;  
660:  
661: {  
662:     boxTitle = para.getT("boxTitle");  
663: }  
  
674: private JTextField titleText =  
      new JTextField(boxTitle);
```

Linija 674 odgovara sljedećim linijama Poglavlja 9:

```
598: public static final String TITLE_TEXT =  
      "Box";  
  
603: private JTextField titleText =  
      new JTextField(TITLE_TEXT);
```

Napomene na alatkama

Na baru za alate, napomene se prikazuju ispod specifičnog dugmeta. Kada se miš nadnese iznad alatke koja ima sliku kutije prikaže se napomena. Ova funkcionalnost je dodata dugmetu *boxButton* na sljedeći način:

```
444: boxButton.setToolTipText(boxTip);
```

Funkcionalnost se sastoji od jednostavnog definisanja osobine *toolTipText* sa opisom sadržanim u objektu *boxTip* na sljedeći način:

```
412: private static String boxTip;
```

```
416: boxTip = para.getT("boxTip");
```

Grupisanje često korištenih osnovnih datoteka

Metodom *getPara*

```
private Para para = Para.getPara();
```

ostvarujemo referencu na objekat definisan klasom *Para*. Primijetite da je ovaj objekat kreiran samo jedanput tokom izvršavanja aplikacije. Prema tome, samo kada je referenca jednaka null vrijednosti, objekat *para* se kreira, kao što je pokazano u metodi *getPara*:

```
749: public static Para getPara() {  
750:     if (para == null) {  
751:         para = new Para();  
752:     }  
753:     return para;  
754: }
```

Pošto se klasa *Para* koristi u gotovo svim klasama, može biti korisno grupisati sve često korištene osnovne datoteke aplikacije u direktorij gdje je smještena klasa *Para*. Kao što smo vidjeli u prethodnom poglavlju, nekoliko klasa mogu zvati datoteke sa ”*gif*” ekstenzijom. U našem primjeru su to klase *AppAbout* i *AppToolBar*. Kako bismo izbjegli ponavljanje ”*gif*” datoteka unutar nekoliko paketa, grupisali smo iste u direktorij *images* unutar direktorija *mb/config*.



O autoru

Dženan Riđanović je profesor informatike i informacionih sistema na Laval Univerzitetu u Kvebeku, Kanada. Dobio je magistarsku diplomu iz informatike na Univerzitetu u Merilendu, SAD, kao Fulbrajтов stipendista. Završio je doktorat iz informacionih sistema na Univerzitetu u Minesoti, SAD. Radio je kao direktor istraživanja u kompaniji Silverrun, gdje je vodio razvoj CASE alata za dizajn baza podataka. Bio je jedan od dva osnivača kompanije OpenPole, koja se specijalizira u razvoju dinamičkih web aplikacija. U zadnje vrijeme, posvetio se razvoju Open Source framework-a za baze podataka i dinamičke web aplikacije, koristeći spiralni pristup, Java tehnologije i XML.

O knjizi

Magične kutije su knjiga o objektno orijentisanom (OO) programskom jeziku, Java, koji je postao popularan u akademskim krugovima, a i na tržištu. Knjiga je specijalno pripremljena za one koji žele da nauče nove OO koncepte kroz praksu. Prema tome, i profesionalac koji želi da proširi svoje znanje, kao i početnik u razvijanju softvera i informacionih sistema, može imati koristi od Magičnih kutija. Proces učenja OO programskog jezika je kompleksan. Kako bih pojednostavio ovaj proces, ja koristim u ovoj knjizi iterativni ili "spiralni pristup" prezentaciji novih koncepcata. Ovaj pristup omogućava "evolutivno" učenje kroz ponavljanje ciklusa razvijanja softverskih verzija od veoma jednostavnih do kompleksnijih.

O CD-u

Magične kutije sadrže CD sa softverom potrebnim za editovanje i izvršavanje Java kôda koji se koristi u knjizi. Taj se kôd može pokrenuti i na Internetu sa mog servera: <http://drdb.fsa.ulaval.ca/mk/>.

ISBN 9958-9214-2-1

9 789958 921421