OBJEKTNO PROGRAMIRANJE 2

Oznaka predmeta: OP2

Predavanje broj: 02

Nastavna jedinica: JAVA

Nastavne teme:

Paketi, CLASSPATH, JAR arhive, nasleđivanje, modifikatori pristupa, redefinisanje metoda, apstraktne klase, interfejsi, unutrašnje klase, polimorfizam, izuzeci, klasa objekt, klasa String.

Predavač: prof. dr Perica S. Štrbac, dipl. ing.

Literatura:

Eckel B., *Thinking in Java*, 2nd edition, Prentice-Hall, New Jersey 2000.

Cay S. Horstmann and Gary Cornell*: "Core Java, Advanced Features", Vol. 2, Prantice Hall, 2013.*

The Java Tutorial, Sun Microsystems 2001. *http://java.sun.com*

Branko Milosavljević, Vidaković M, *Java i Internet programiranje*, GInT, Novi Sad 2002.

Paketi

- Java programi se sastoje isključivo iz klasa.
- Broj klasa koje čine program može biti relativno velik, pa je uvođenje organizacije u takav skup klasa neophodno.
- Paketi su način da se klase grupišu po nekom kriterijumu.
- Paketi mogu da sadrže klase ili potpakete, analogno odnosu direktorijuma i datoteka u okviru fajl-sistema.
  - Svaka klasa mora da pripada nekom paketu.
- Ako se ne navede kom paketu pripada data klasa, podrazumeva se da pripada tzv. korenskom ili implicitnom paketu.
- Korenski paket nema posebno ime.
  - On može da sadrži klase i potpakete, koji sa svoje strane takođe mogu da sadrže klase i potpakete. Slika 2.1 prikazuje strukturu paketa nekog programa.

Slika 2.1 Struktura paketa u programu.

Paketi

- Paketi i klase su u okviru fajl-sistema zaista i organizovani kao direktorijumi i datoteke:
  - paketi su predstavljeni direktorijumima
  - klase se nalaze u odgovarajućim datotekama
- Klasa koja se nalazi u nekom paketu (osim korenskog), mora u okviru svoje datoteke imati odgovarajuću deklaraciju, kao u sledećem primeru:

```
package paket1;
class Automobil { ... }
```

- Deklaracija package se mora nalaziti na samom početku teksta datoteke.
- Datoteka *Automobil.java* mora biti smeštena u direktorijum *paket1* koji se nalazi u korenskom direktorijumu aplikacije.
- Naziv korenskog direktorijuma nije važan, niti je važno gde se on nalazi u okviru fajl-sistema. Prevođenje klase *Automobil* se mora obaviti komandom:

```
D:\temp\korenski paket\>javac paket1\Automobil.java
```

- Komanda za prevođenje se poziva iz korenskog direktorijuma projekta, a kao parametar navodi se ime *.java* datoteke, zajedno sa *relativnom putanjom* do nje. Npr. komanda prevođenja za klasu *Tocak u paketu 3:*

```
D:\temp\korenski paket\>javac paket2\paket3\Tocak.java
```

Paketi

- Tekst klase *Tocak* obavezno mora početi odgovarajućom deklaracijom:

```
package paket2.paket3;
class Tocak { ... }
```

- Za separaciju imena paketa u okviru Java programa koristi se tačka, a ne kosa crta ili obrnuta kosa crta.
- Prilikom pokretanja programa navodi se ime one klase koja sadrži metodu *main*.
  - Prilikom navođenja imena ove klase mora se navesti njeno puno ime, uključujući i paket u kome se klasa nalazi.

  Primer, ukoliko klasa *Tocak* poseduje metodu *main*, i želimo da odatle počne izvršavanje programa, program moramo pokrenuti pomoću sledeće komande:

```
D:\temp\korenski paket\>java paket2.paket3.Tocak
```

- Važno je sa kog mesta se poziva Java interpreter: ovde je to korenski direktorijum aplikacije.
- Svaka prevedena klasa se u okviru aplikacije vidi u okviru paketa čija je putanja jednaka relativnoj putanji do odgovarajućeg direktorijuma.

Paketi

- Programski jezik Java stiže sa velikim brojem klasa grupisanim u pakete.
  - Te klase su dostupne kao i klase koje sami pišemo.
- Klasa *Vector* koja se nalazi u paketu *java.util* je u programima dostupna kao **java.util.Vector**.
  - Kako bi svako pominjanje ove klase u tekstu programa zahtevalo navođenje pune putanje do nje (odnosno navođenje odgovarajućeg paketa), to bi program učinilo manje čitljivim. Zato je moguće na početku teksta klase deklarisati da se koristi ta-i-ta klasa koja se nalazi u tom-i-tom paketu.

```
package paket1;
import java.util.Vector;
class Automobil { ... }
```

Nadalje se u tekstu klase Automobil klasa Vector koristi samo navođenjem njenog imena, bez imena paketa u kome se nalazi.

- Deklaracija **import** se mora nalaziti između (opcione) **package** deklaracije i definicije klase.

Paketi

- Ukoliko koristimo više klasa iz istog paketa, moramo svaku od njih navesti u odgovarajućoj import deklaraciji.
- Drugi način je da se importuju sve klase iz datog paketa koristeći džoker-znak za sve klase *:

```
package paket1;
import java.util.*;
class Automobil { ... }
```

- Ovakav način importovanja ne obuhvata i sve potpakete importovanog paketa!
- Nije dozvoljeno korišćenje džoker znakova kao u primeru:

```
import java.util.Vec*;  // nije dozvoljeno!
```

- Klase koje se nalaze u paketu *java.lang* nije potrebno importovati.
  - Odgovarajuća **import** deklaracija se podrazumeva.
- Korišćenje klase *Vector* iz paketa *java.util* u prethodnom primeru bi značilo da se odgovarajuće stablo direktorijuma *java\util\...* koje sadrži kompajlirane klase mora kopirati unutar strukture direktorijuma svake aplikacije koja to i koristi.

CLASSPATH

- Prethodnim bi se bespotrebno zauzimao prostor i komplikovalo održavanje softvera. Zato postoji način da se paketi sa klasama koji se koriste iz više aplikacija čuvaju na jednom mestu, a sve aplikacije će pomoću odgovarajućeg mehanizma te klase videti kao da je struktura direktorijuma iskopirana u okviru svake aplikacije.
- U pitanju je mehanizam sličan korišćenju PATH promenljive okruženja (*environment variable*).
- Java interpreter za ovu svrhu koristi promenljivu okruženja koja se naziva CLASSPATH. Ona sadrži listu direktorijuma u kojima treba tražiti klase koje se koriste.

  Primer: ukoliko je cela *java\...* hijerarhija paketa smeštena u direktorijum *C:\java\lib*, vrednost CLASSPATH promenljive bi glasila:

  ```
  CLASSPATH=C:\java\lib
  ```

  čime bi sve klase smeštene po svojim paketima unutar direktorijuma *C:\java\lib* bile vidljive za sve Java aplikacije.
- Ukoliko CLASSPATH treba da sadrži više direktorijuma, oni se navode jedan za drugim, sa tačkom-zarez (na LINUX-u dvotačka) kao separatorom.

  ```
  CLASSPATH=C:\java\lib;D:\mojalib
  ```

CLASPATH

- Ukoliko se u CLASSPATH doda direktorijum *D:\temp\korenski paket* iz prethodnih primera, na primer komandom:

  C:\>set CLASSPATH=%CLASSPATH%;D:\temp\korenski paket

  tada se program može pokrenuti sa bilo kog mesta u okviru fajl sistema, jer će klase biti vidljive preko CLASSPATH-a.

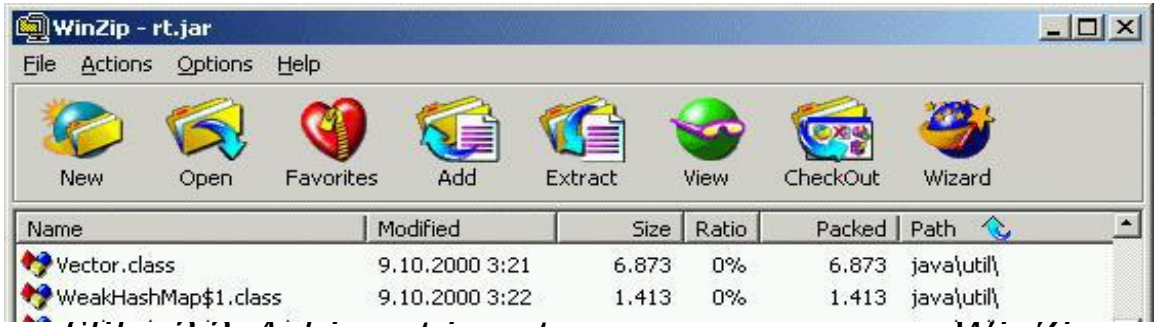
  Primer: komanda:

  C:\>java paket2.paket3.Tocak

  će pokrenuti program iako se poziv ne izvodi iz direktorijuma *D:\temp\korenski paket*.

JAR (Java archive)

- Distribucija biblioteka klasa smeštenih u svoje pakete nije preterano elegantna u slučaju većeg broja klasa i paketa, jer se povećava broj datoteka i direktorijuma koje treba instalirati i navesti u CLASSPATH-u.
- Zbog toga je omogućeno arhiviranje biblioteka u tzv. JAR arhive koje sadrže klase u svojim paketima arhivirane u klasičnom *zip* formatu. Podrazumevana ekstenzija im je *.jar* (može biti i *.zip*).
- Ovakve arhive se mogu generisati alatkom jar koja je sastavni deo JDK paketa, ali mogu i bilo kojim drugim programom koji može da generiše *zip* arhive (ekstenzija se može promeniti kasnije).
- Sve klase iz osnovne Java biblioteke su, prilikom instalacije JDK paketa, smeštene u datoteku *%JAVA_HOME%\jre\lib\rt.jar*, gde je JAVA_HOME direktorijum gde je instaliran JDK paket. Ovu datoteku možemo otvoriti, recimo, programom *WinZip*, kao na slici 2.2.



Slika 2.2. Arhiva rt.jar otvorena programom WinZip

JAR (Java archive)

- Umesto da u okviru CLASSPATH-a navodimo direktorijum u kome se nalazi raspakovan sadržaj arhive *rt.jar*, možemo navesti samu datoteku *rt.jar* (sa svojom putanjom) i dobićemo isti efekat.

  Primer: `CLASSPATH=C:\jdk1.3\jre\lib\rt.jar`
- CLASSPATH može da sadrži nazive direktorijuma i *zip* arhiva u kojima se nalaze deljene biblioteke. Korišćenje direktorijuma u arhivama je u ovom slučaju potpuno ravnopravno.
- U dosadašnjim primerima je naglašavano da se prilikom pokretanja programa moramo nalaziti u korenskom direktorijumu aplikacije. To je, zapravo, posledica činjenice da se *tekući direktorijum* u kome se nalazimo nalazi u CLASSPATH-u kada on nije definisan, kao da je `CLASSPATH=.`

  gde je tačka (.) oznaka za tekući direktorijum.
- Ako se CLASSPATH promenljiva definiše, tekući direktorijum se mora eksplicitno navesti u CLASSPATH-u.
- Još jedna komponenta CLASSPATH-a se podrazumeva, a to je biblioteka *rt.jar*
  - Nju ne moramo navoditi čak ni kada definišemo promenljivu CLASSPATH.

  Svaki CLASSPATH uvek sadrži komponentu:

  `CLASSPATH=C:\jdk1.3\jre\lib\rt.jar`

CLASSPATH

- Ukoliko CLASSPATH uopšte nije definisan, onda on obuhvata i tekući direktorijum, tako da se može reći da CLASSPATH u tom slučaju glasi:

  ```
  CLASSPATH=.;C:\jdk1.3\jre\lib\rt.jar
  ```

  ovaj podrazumevani skup komponenti CLASSPATH-a uveden je tek od Java verzije 1.2.

- U praksi se koristi i IBM-ov kompajler *jikes*, koji je znatno brži. *jikes* nije deo standardne JDK instalacije i mora se instalirati posebno. Do svoje verzije 1.02 on se ponaša kao Java 1.1 kompajler, tako da nema podrazumevanih komponenti u CLASSPATH-u.

  Kako je prilično nezgodno menjati sadržaj CLASSPATH promenljive naizmenično za kompajliranje *jikes*-om i pokretanje *java*-om, problem se može prevazići korišćenjem promenljive okruženja JIKESPATH koju koristi isključivo *jikes*.

  Ona ima isto značenje kao CLASSPATH do Java verzije 1.1. Dakle, ako CLASSPATH ima sadržaj: `CLASSPATH=.;D:\nekamojabibl.zip`

  JIKESPATH bi trebalo da ima sledeći sadržaj:

  ```
  JIKESPATH=.;C:\jdk1.3\jre\lib\rt.jar;D:\nekamojabibl.zip
  ```

Nasleđivanje

- Nasleđivanje, kao jedan od osnovnih koncepata objektno-orijentisanog programiranja, postoji i u Javi.
- Kada jedna klasa nasleđuje drugu, potrebno je to naglasiti u okviru teksta klase klazulom **extends** kao u sledećem primeru, gde klasa *BorbeniAvion* nasleđuje klasu *Avion*:

```
class Avion {
   Krilo levo, desno;
   void poleti() { ... }
   void sleti() { ... }
}
class BorbeniAvion extends Avion {
   Top top;
   Bomba[] bombe;
   void poleti() { ... }
   void pucaj() { ... }
}
```

- Java ne dopušta višestruko eksplicitno nasleđivanje klasa (što može u jeziku C++). Dakle klasa može da nasledi najviše jednu klasu eksplicitno.

Modifikatori pristupa

- U Javi postoje sledeća tri modifikatora pristupa:
  - **public**: označava da su atribut ili metoda vidljivi za sve klase u programu
  - **protected**: atribut ili metoda su vidljivi samo za klase naslednice
  - **private**: atribut ili metoda su vidljivi samo unutar svoje klase
- Nespecificiran (tzv. *friendly*): atribut ili metoda su vidljivi za klase iz istog paketa
- Modifikatori pristupa se navode ispred definicije metode ili atributa.

  Primer:

```
class Avion {
   protected Krilo levo, desno;
   public void poleti() { ... }
   public void sleti() { ... }
}
```

- Na sličan način modifikatori pristupa se mogu primeniti i na celu klasu, na primer:

```
public class Avion { ... }
```

Redefinisanje metoda

- Redefinisanje metoda (method overriding) je postupak kada klasa naslednica redefiniše telo metode nasleđene od roditeljske klase.
- U Javi se to specificira prostim navođenjem nove definicije metode u klasi naslednici.

```
class A {
  void metoda1() { System.out.println("metoda1 klase A"); }
  void metoda2() { System.out.println("metoda2 klase A"); }
}
class B extends A {
  void metoda1() { System.out.println("metoda1 klase B"); }
}
```

Izvršavanjem koda	Dobija se ispis na konzoli

`A varA = new A();`
`B varB = new B();`
`varA.metoda1();`
`varB.metoda1();`
`varA.metoda2();`
`varB.metoda2();`
	`metoda1 klase A`
	`metoda1 klase B`
	`metoda2 klase A`
	`metoda2 klase A`

- Metoda metoda1 je redefinisana u klasi B, tako da je promenjen ispis na konzolu, dok metoda2 nije redefinisana, pa se za klasu B preuzima implementacija metode iz klase A).

Apstraktne klase

- Apstraktne klase su klase koje ne mogu imati svoje instance (objekte).
- Razlog za to je što je implementacija neke od metoda izostavljena.

  Primer:

```
public abstract class A {
   public void metoda1() { ... }
   public abstract void metoda2();
   private int i;
}
```

  metoda *metoda2* je proglašena za apstraktnu korišćenjem ključne reči **abstract**. Njena implementacija nije navedena. Samim tim klasa je apstraktna pa se i za nju to mora navesti navođenjem ključne reči **abstract** ispred **class**.
- Dakle u slučaju da je klasa A apstraktna onda iskaz poput:

```
A x = new A();  //nije dopušten
```

Interfejsi

- Interfejsi su poseban koncept u Javi: nisu u pitanju klase, mogu da sadrže deklaracije apstraktnih metoda, konstanti i statičkih atributa.

```
interface Instrument {
   void sviraj();
   void nastimaj();
 }
```

- Interfejsi podsećaju na apstraktne klase. Veza između klasa i interfejsa je sledeća: kaže se da klasa *implementira* (a ne nasleđuje) interfejs.
- Klasa može da implementira više interfejsa istovremeno. Klasa koja nasleđuje drugu klasu može da implementira i interfejse. Jedan interfejs može da *nasledi* drugi interfejs.

```
class Klarinet implements Instrument {
   void sviraj() { ... }
   void nastimaj() { ... }
}
```

- Klasa koja implementira interfejs obavezna je da redefiniše sve metode interfejsa inače kompajler neće dopustiti prevođenje te klase.

Unutrašnje klase

- Od Java verzije 1.1 klasa može, osim atributa i metoda, da poseduje i tzv. unutrašnje klase (*inner classes*).

```
class Spoljasnja {
   void metoda() { ... }
   class Unutrasnja {
      int metoda2() { ... }
   }
}
```

- Klasa *Unutrasnja* je vidljiva samo unutar klase *Spoljasnja*, mada se to može promeniti modifikatorima pristupa na uobičajen način.
- Instanca unutrašnje klase se može kreirati i izvan nje, ali samo preko instance spoljašnje klase, kao u sledećem primeru:

```
Spoljasnja spo = new Spoljasnja();
Spoljasnja.Unutrasnja unu = spo.new Unutrasnja();
```

- Sledeći izraz (pokušaj konstrukcije instance unutrašnje klase bez instance spoljašnje klase) nije dozvoljen:

```
Spoljasnja.Unutrasnja unu = new Spoljasnja.Unutrasnja();
//nije dozvoljeno
```

Polimorfizam

- Polimorfizam je koncept koji omogućuje objektima da ispolje različito ponašanje, zavisno od njihove klase, bez obzira što se oni koriste kao instance nekog zajedničkog roditelja.
- Neka su date sledeće tri klase:

```
abstract class Instrument {
        abstract void sviraj();
}
class Violina extends Instrument { void sviraj() { ... } }
class Klarinet extends Instrument{ void sviraj() { ... } }
```

- Dakle, primer definiše tri klase: klasa *Instrument* je apstraktna klasa (njena metoda *sviraj* je apstraktna), a klase *Violina* i *Klarinet* nasleđuju klasu *Instrument* i implementiraju (redefinišu) apstraktnu metodu.
- Posmatrajmo sada klasu *Muzicar*:

```
class Muzicar {
   void sviraj(Instrument i) {
      i.sviraj();
   }
}
```

Polimorfizam

- Klasa *Muzicar* ima metodu *sviraj* koja kao parametar ima instancu klase *Instrument, a* klasa *Instrument* ne može imati instance, jer je apstraktna.
- Ova metoda će ipak biti upotrebljiva, jer se njoj kao parametar može proslediti instanca neke klase koja nasleđuje klasu *Instrument* u ovom slučaju instance klasa *Violina* i *Klarinet*.

  Iskaz
  ```
  Muzicar m = new Muzicar();
  m.sviraj(new Klarinet());
  ```
  će izazvati pozivanje metode *sviraj* klase *Klarinet* (iako se to nigde eksplicitno ne navodi u metodi sviraj klase *Muzicar*).

  Iskaz
  ```
  m.sviraj(new Violina());
  ```
  će izazvati pozivanje metode *sviraj* klase *Violina* po istom principu.
- Poziv metode *sviraj* klase *Muzicar* će imati različite efekte zavisno od stvarnog argumenta.
- Određivanje koja metoda će se pozvati se obavlja *u toku izvršavanja programa*. U primeru ovo specijalno ponašanje metoda nije ničim naglašeno u tekstu programa.
  - U terminologiji jezika C++, sve metode u Javi su virtuelne, pa se ta osobina ne mora naglašavati posebno u programu.

Izuzeci

- Izuzeci su mehanizam za kontrolu toka programa koji se koristi za obradu grešaka nastalih u toku izvršavanja programa.
- Segment programskog koda za koji smatramo da može da izazove izuzetak možemo da smestimo u tzv. *try/catch* blok, kao u sledećem primeru:

```
try {
   // kod koji može da izazove izuzetak
}
catch (Exception ex) {
     System.out.println("Desio se izuzetak: " + ex);
}
```

- Izuzetak može biti, na primer, deljenje nulom, pristup elementu niza koji je izvan granice niza, itd.
- Ukoliko se prilikom izvršavanja koda koji se nalazi u *try* bloku desi izuzetak, tok izvrašavanja programa se automatski prebacuje na početak *catch* bloka. Nakon izvršavanja koda u *catch* bloku, program dalje nastavlja rad.

Izuzeci

- U okviru *catch* bloka informacije o samom izuzetku koji se dogodio su dostupne preko objekta klase *Exception* ili neke njene naslednice.
- U primeru je to objekat *ex*. Različite vrste izuzetaka su predstavljene različitim *exception* klasama, na primer:
  - svi izuzeci prilikom izvršavanja aritmetikih operacija (deljenje nulom, *overflow*, itd.) su predstavljene klasom *ArithmeticException.*
  - pristup elementu čiji je indeks izvan granice niza je predstavljen klasom *ArrayIndexOutOfBoundsException*, itd.
- Klasa *Exception* je zajednički predak svim *exception* klasama. Jedan *try* blok može imati više sebi pridruženih *catch* blokova, kao u sledećem primeru:

```
try {
   // kod koji može da izazove izuzetak
}
catch (ArithmeticException ex) { System.out.println("Delite
   nulom");
}
catch (ArrayIndexOutOfBoundsException ex)
   {   System.out.println("Pristup van granica niza");
}
catch (Exception ex) {  System.out.println("Svi ostali
   izuzeci");  }
finally              {  // kod koji se izvršava u svakom
   slučaju   }
```

Izuzeci

- Kada se dogodi izuzetak, niz *catch* blokova se sekvencijalno obilazi i ulazi se u onaj *catch* blok čiji parametar odgovara izuzetku koji se dogodio.
- Red odgovora na izuzetak u ovom slučaju znači: u pitanju je klasa kojoj *exception* objekat pripada, ili njen predak. Kada poslednji *catch* blok hvata izuzetak klase *Exception*, to znači da će svi izuzeci biti obrađeni, jer je klasa *Exception* zajednički roditelj.
- Blok *finally* se ne mora navesti. On sadrži blok koda koji će se izvršiti u svakom slučaju, desio se izuzetak ili ne.
- Moguće je definisati i nove vrste izuzetaka definicijom odgovarajuće klase. Može se definisati nova vrsta izuzetka predstavljenog klasom *MojException* koja je data u primeru:

```
public class MojException extends Exception {
   public MojException() {           super();         }
   public MojException(String msg) { super(msg);      }
}
```

- Klasa *MojException* ima dva konstruktora koji pozivaju odgovarajuće konstruktore roditeljske klase *Exception*. Pisanje ovakvih konstruktora nije obavezno, ali je obavezno naslediti klasu *Exception* (ili nekog njenog potomka).

Izuzeci

- Ovakav korisnički izuzetak može biti izazvan samo programski, pomoću ključne reči **throw**, kao u sledećem primeru:

```
if (errorCheck())
throw new MojException("Houston, we have a problem.");
```

- Programski kod koji sadrži ovakvu **throw** naredbu mora biti smešten unutar *try* bloka koji hvata izuzetak *MojException* .
  - O uspunjenosti tog uslova se brine kompajler.

Ovo bi moglo da izgleda na sledeći način:

```
try {
   if (errorCheck())
      throw new MojException("Houston, we have a problem.");
}
catch (MojException ex) {
   System.out.println("Exception: " + ex);
}
```

Izuzeci

- Drugi način da obradimo nastanak ovakvog izuzetka je da metodu u kojoj se nalazi **throw** naredba označimo kao metodu u kojoj može da nastane izuzetak date vrste.

```
public void metoda() throws MojException {
  ...
  if (errorCheck())  throw new MojException("Houston, we have a
  problem");
  ...
}
```

- Sada poziv ovakve metode mora biti u odgovarajućem *try* bloku, ili metoda koja sadrži ovaj poziv mora isto biti označena da može da izazove izuzetak.

```
public void m1() {
   try {  metoda();  }
   catch (MojException ex) {  System.out.println("Exception: " +
   ex); }
}
```

ili:

```
public void m1() throws MojException {          metoda();       }
```

- Na ovaj način odgovornost za obradu izuzetka se može propagirati sve do metode *main* od koje počinje izvršavanje programa.

Klasa Object

- Klasa *Object* predstavlja osnovnu klasu u hijerarhiji Java klasa, u smislu da sve klase implicitno nasleđuju klasu *Object*. Klasa kod koje je izostavljena klauzula **extends** podrazumeva se da nasleđuje klasu *Object*.
- Klasa *Object* nije apstraktna, tako da je moguće kreirati objekat ove klase.
- Ona definiše neke metode koje se relativno često koriste

  public boolean equals(Object o);

- Koristi se prilikom poređenja objekata; poređenje tipa **(a == b)** je zapravo poređenje referenci. Poređenje **(a.equals(b))** vraća rezultat zavisno od implementacije metode *equals* klase kojoj pripada objekat *a*.
- Klasa *Object* definiše podrazumevano poređenje objekata koje se svodi na poređenje referenci.

public int hashCode(); Izračunava *hash* vrednost za dati objekat. Koristi se najviše u *hash*-tabelama.

$$h(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

public String toString(); Vraća string reprezentaciju objekta. Ukoliko se ne redefiniše, poziva se implementacija iz klase *Object* koja vraća prilično 'nerazumljiv' rezultat.

Klasa String

- Klasa *String* se nalazi u paketu *java.lang* i predstavlja string kao tip podatka (rekosmo da stringovi nemaju odgovarajući primitivni tip u Javi).
- Klasa String je kao i svaka druga Java klasa osim što ima donekle specijalan tretman od strane kompajlera.
  - Vrednosti primitivnih tipova se mogu predstaviti u Java programu odgovarajućim konstantama.

    Primeri: **16** je vrednost tipa **int**,

    vrednost **16L** označava vrednost tipa **long**,

    'x' je vrednost tipa **char**, itd.
  - Objektima se ne može pridružiti vrednost koja se može predstaviti literalom, osim u slučaju objekata klase *String*.

    Primer:

    "tekst" u Java programu predstavlja objekat klase *String* čija je vrednost inicijalizovana na dati tekst.
  - Ispravno je sledeće pisanje:

```
String x = "tekst";
String x = new String("tekst");
```

Klasa String

- Java ne omogućuje redefinisanje operatora (kao što je to moguće u jeziku C++). Međutim, operator + može da se upotrebi za konkatenaciju stringova.
- Kada naiđe na izraz poput

```
"ime"  + "prezime"    ili
x      + "prezime"    ili
x      + y
```

kompajler će generisati kod koji će izvršiti konkatenaciju stringova i vratiti rezultat u obliku novokreiranog objekta klase *String*.

- Konkatenacija može da obuhvati i primitivne tipove ili objekte drugih klasa.

```
int a = 10;
String x = "vrednost: " + a;
```

rezultira kreiranjem novog *String* objekta čiji sadržaj je **"vrednost: 10"**.

U slučaju konkatenacije stringa sa objektom neke druge klase, kao na primer:

```
Automobil a = new Automobil();
String x = "Moj auto je: " + a;
```

biće pozvana metoda *toString* klase *Automobil*, pa će se zatim izvršiti konkatenacija stringova pomoću dobijenog stringa.

Klasa String

- Metoda koja kao parametar ima tip *String*, može u svom pozivu da primi i objekat neke druge klase, pri čemu će kompajler automatski generisati kod koji poziva metodu *toString()* objekta, i zatim taj rezultat prosleđuje metodi koja se poziva.

```
public void handleMessage(String message) { ... }
```

i njen poziv

```
handleMessage(new Automobil());
```

Ovaj poziv metode *handleMessage* biće preveden u sledeći poziv:

```
handleMessage(new Automobil().toString());
```

- Parametri metoda tipa *String* se ponašaju kao da se prenose po vrednosti a ne po referenci, iako su u pitanju objekti, a ne primitivni tipovi.

```
public void handleMessage(String message) { message += "dodaj"; }
```

neće izazvati promenu objekta koji je prosleđen kao parametar, jer će se u telu metode, prilikom konkatenacije, generisati novi *String* objekat koji će biti dodeljen lokalnoj kopiji reference *message*. Po povratku iz metode uništava se promenjena lokalna kopija reference, i ostaje samo originalna referenca koja i dalje ukazuje na stari *String* objekat.