Java

Java

uvod

- Just Another Vague Acronim - JAVA
- 1991. tvorac Jave – James Gosling, *Sun Microsystems*
  - Stvorio jednostavan, platformski nezavistan jezik
  - Namenjen pokretanju elektronskih uređaja (Interaktivna TV, inteligentne rerne, telefoni,..)
- 1994. Java se ugrađuje u web browser WebRunner
- 1995. obavljuje se kod i dokumentacija Jave na Internetu

Java
osnovne karakteristike

- Objektna orijentacija
  - podržava sve koncepte objektno orijentisanog programiranja
  - sintaksa slična C++ ali su izbačeni složeni koncepti (pointeri)
- Prenosivost
  - Java programi se prevode u byte kod koji nije mašinski jezik nijednog konkretnog računara, već se izvršava na JVM
  - Java Virtuelna Mašina - je virtuelni računar koji može biti simuliran na bilo kom računaru
- Prirodna prilagođenost Internetu
  - Java programi mogu da se izvršavaju u Web browserima
  - Poseduju sigurnosne mehanizme
  - Mogu da se distribuiraju i izvršavaju na različitim mašinama
  - Podržava konkurentno programiranje
  - Omogućene klase za korisnički interfejs (API) koji omogućuje jedinstven izgled i korišćenje aplikacija

Java kao platforma

- Dizajniran da što manje zavisi od specifičnih karakteristika konkretnog računarskog sistema
- Jednom napisan i preveden program se izvršava na bilo kojoj platformi koja podržava Javu
- Interpretirani jezik, bajt-kod
- Java virtuelna mašina (JVM)
- Dve vrste Java programa
  - Apleti
    - izvršavaju se u okviru WWW čitača
    - automatska distribucija i instalacija
    - ograničene mogućnosti apleta iz razloga bezbednosti
  - Aplikacije

Java od početka

- Verzije

  1.0

  1.1 i

  Java 2 platforma (1.2,1.3,1.4…)

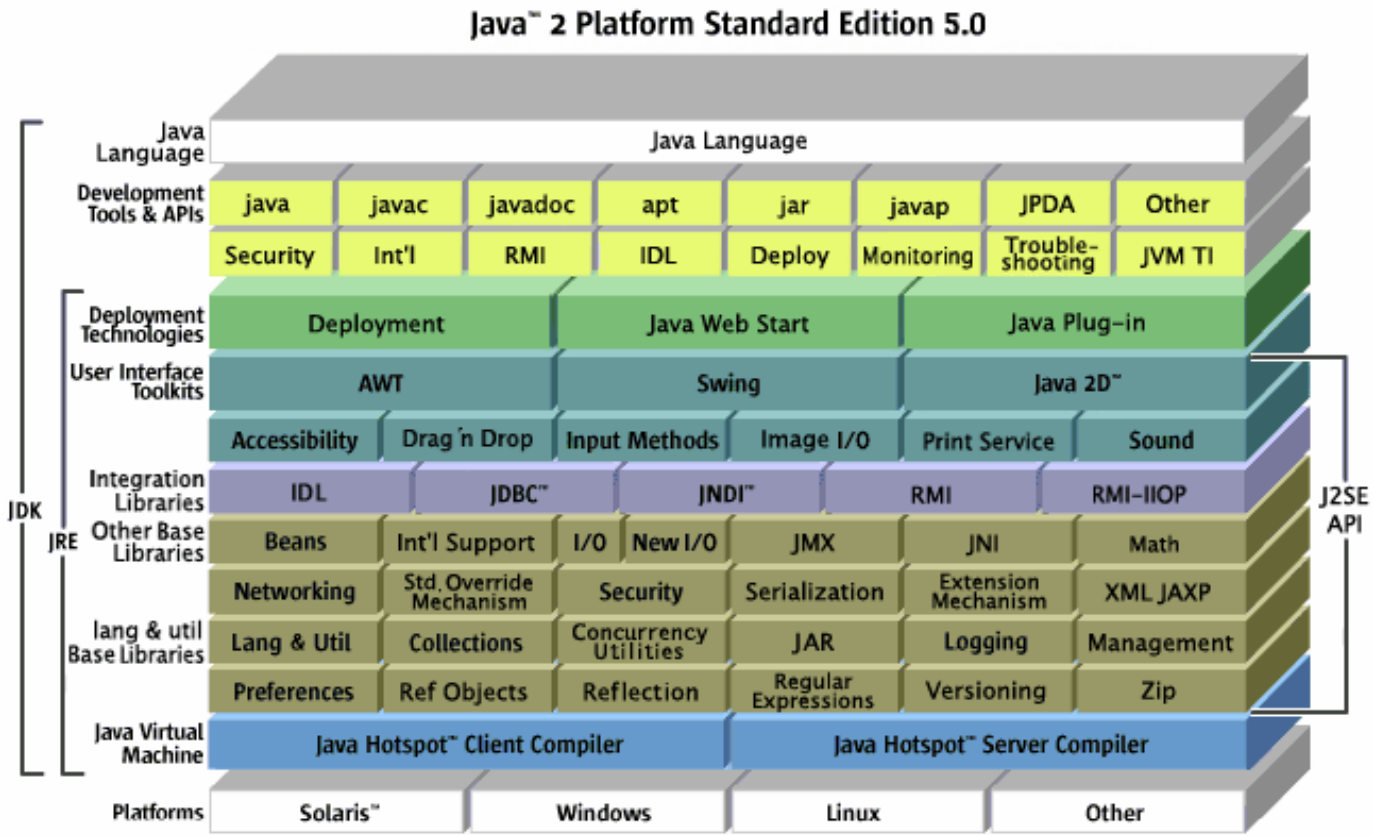
- Free download

  java.sun.com, www.sun.com

- Dokumentacija

Java 2 Platforma

- Javina platforma se sastoji od tri elementa: Java programskog jezika + Java API biblioteka + Javine virtalne mašine.
- Java 2 platforma je skup programa i sistemskih resursa koji su specifični za dati operativni sistem (Windows, Linux, UNIX, Mac, ...), a omogućuje prevođenje i izvršavanje Java programa.



Java 2 Platforma

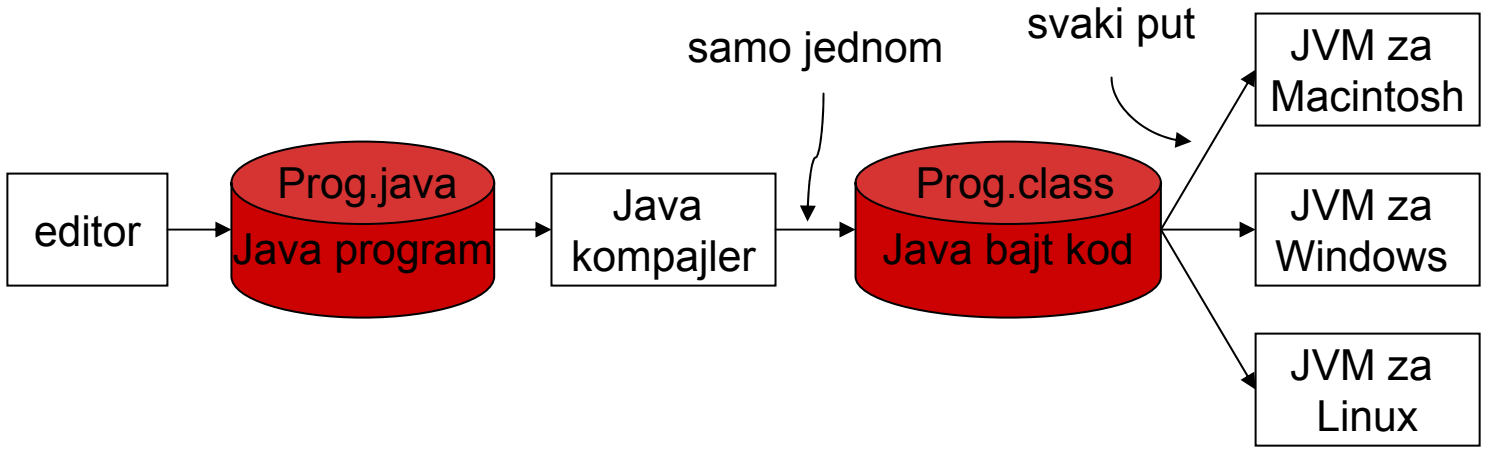
- **Java API (Application Programming Interface)** je skup već gotovih softwareskih komponenti napisanih u Java programskom jeziku koji su podijeljeni u pakete (eng. package), a svaki se sastoji od klasa (eng. class) koje enkapsuliraju (eng. encapsulate) određenu funkcionalnost. Evo nekoliko važnijih paketa:
- **java.applet** - Javini programi se mogu izvršavati unutar Web preglednika (npr. Internet Explorer, Mozilla, Netscape); ovaj paket enkapsulira tu funkcionalnost.
- **java.awt, javax.swing** - enkapsulira kreiranje korisničkog interfejsa, grafike i slike.
- **java.io**-input/output funkcionalnost za Java 2 platformu
- **java.lang** -fundamentalan za prevođenje i interpretiranje Javinih programa
- **java.net** - namenjena Internet aplikacijama.
- **java.security** - sigurnost na Internetu.
- **java.sql** - rad sa bazama podataka.

Kompajleri i interperteri

- Program pisan u nekom od viših programskih jezika potrebno je prevesti na mašinski jezik, ne bi li bio izvršen. To prevođenje vrši **kompajle**r (compiler) odgovarjućeg programskog jezika. Nakon što je program jednom preveden, program u mašinskom jeziku se može izvršiti neograničen broj puta, ali, naravno, samo na određenoj vrsti računara.
- Postoji alternativa. Umesto kompajlera, koji odjednom prevodi čitav program, moguće je koristiti **interpreter**, koji prevodi naredbu po naredbu prema potrebi. Interpreter je program koji se ponaša kao CPU s nekom vrstom dobavi-i-izvrši ciklusa. Da bi izvršio program, interpreter radi u petlji u kojoj uzastopno čita naredbe iz programa, odlučuje šta je potrebno za izvršavanje te naredbe, i onda je izvršava (oni se mogu koristiti za izvršavanje mašinskog programa pisanog za jednu vrstu računara na sasvim različitom računaru).

Java kompajler i interpreter

- Projektanti Jave su se odlučili za upotrebu kombinacije kompajliranja i interpretiranja. Programi pisani u Javi se prevode u mašinski jezik virtuelnog računara, tzv. **Java Virtual Machine**.
- Mašinski jezika za Java Virtual Machine se zove **Java bytecode**. (Sun Mycrosystems, začetnik Jave, razvio je CPU koji izvršava Java bajt kod u originalu, bez interpretiranja).
- Sve što je računaru potrebno da bi izvršio Java bajt kod jeste interpreter. Takav interpreter oponaša Java virtual machine i izvršava program.



Hello, world

```
public class HelloWorld {
    public static void main(String[ ] args) {
        System.out.println("Hello, World");
    } // kraj main metode
} // kraj klase HelloWorld
```

case sensitive
vodite računa o velikim
i malim slovima

- svaka Java aplikacija mora sadržati barem jednu klasu s metodom main(String[] args)
- počinje svoje izvršavanje pozivom metode **main**
- ovako napisan program se prevodi izvršavajući

  javac HelloWorld.java
- Ako nema grešaka prevodilac javac kreira datoteku HelloWorld.class koja sadrži bytecode instrukcije za JVM.
- JVM se pokreće sa

  java HelloWorld

Hello, world

- System.out.println("Hello, World");

  Java API (Aplication Programming Interface) je skup već gotovih softverskih komponenti napisanih u Java programskom jeziku koji su podeljeni u pakete (package), a svaki se sastoji od klasa koje enkapsuliraju određenu funkcionalnost. Klasa **System** nalazi se u paketu **java.lang**. Njeno puno ime je **java.lang.System**. Međutim, java.lang je jedini paket za koji se ne mora navesti puno ime klase. **out** je statička varijabla članica klase **System** tipa **PrintStream**; **println** je jedna od metoda klase **PrintStream**. Efekat cele naredbe je ispis stringa "Hello, World" na konzoli.

- Za API pogledati dokumentaciju Jave koja ide uz JDK (Java Development Kit)

  dirjave\docs\api\index.html

Java od početka

- Potrebni i skoro dovoljni uslovi
  1. prihvatiti OO način razmišljanja
     (‘misliti na engleskom’)
  2. na početku i na ‘kraju’ čitati dokumentaciju
     (‘ne bežati od engleskog’)

- U Javi je sve čime se manipuliše objekat neke klase♀, odnosno referenca na objekat

sledi

- Tipovi podataka
- Operatori
- Upravljačke strukture

tipovi podataka

- Strogo tipiziran jezik
  - Svaki podatak u svakom trenutku se zna kom tipu pripada
  - Prilikom deklaracije promenljive obavezno se navodi i njen tip
- Dva tipa podatka
  - Prosti tipovi podataka
    - int, char, byte, float, boolean
    - Svaki prost tip podatka ima tačno definisanu veličinu bez obzira na platformu na kojo se izvršava Java kod
  - Složeni tipovi podataka
    - Objekti
    - Nizovi

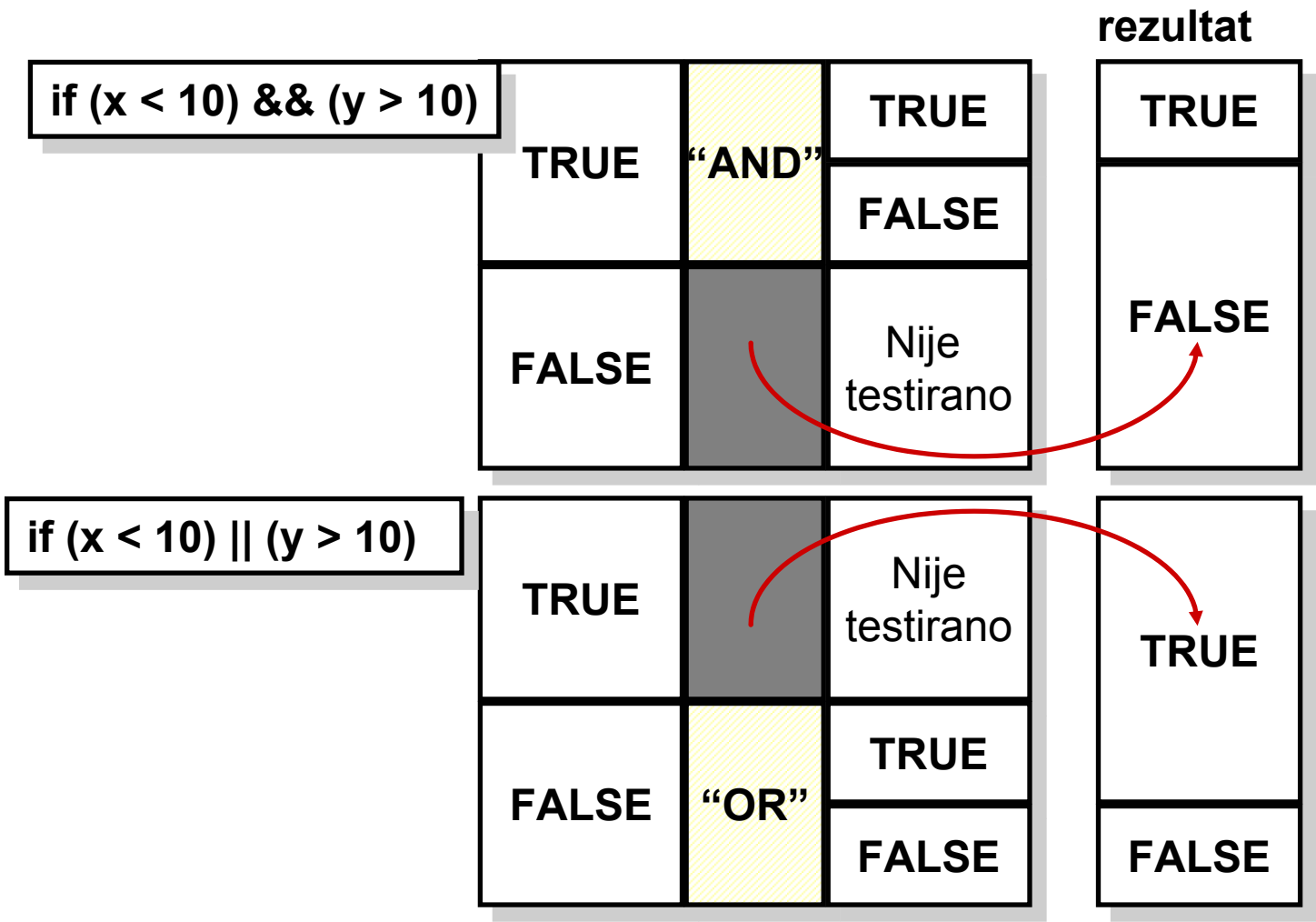
Java primitivni tipovi podataka

Tip Podataka	Opis	Default vrednost	Veličina	Minimalna vrednost Maximalna vrednost
Booelan	Logicki tip podatka sadrži *true ili false*	False	1 bit	- -
Char	Karakter (Unicode)	\u0000	16 bita	\u0000 \uFFFF
Byte	Ceo broj	0	8 bita	-128 127
Short	Ceo broj	0	16 bita	-32768 32767
Int	Ceo broj	0	32 bita	-2147483648 2147483647
Long	Ceo broj	0	64 bita	-9223372036854775808 9223372036854775807
Float	Realan broj sa pokretnim zarezom	0.0	32 bita	±1.40239846E-45 ±3.40282347E+38
Double	Realan broj u eksponencijalnom obliku	0.0	64 bita	±4.94065645841246544E-324 ±1.79769313486231570E+308

Java operatori

Tip operatora	Operator	Opis
Aritmetički operatori	++ , -- + , - , * , / , %	Inkrementiranje , dekrementiranje (unarni operatori) Sabiranje, Oduzimanje, Množenje, Deljenje, Deljenje po modulu
Operatori poređenje	== , != , < , > , <= , >= , %	Jednakost, Nejednakost, Manje od, Veće od, Manje ili jednako, veće ili jednako, deljenje po modulu
Logički operatori	&& , || , ^ , !	Logičko I, Logičko uključujuće ILI, Logičko isključujuće ILI, Negacija
Operatori za rad sa bitovima	& , ^ , ^- << , >>	Bitno I boolean I, Bitno logičko uključujuće ILI, Bitno logičko isključujuće ILI , Pomeranje bitova u levu stranu, pomeranje bitva u desnu stranu
Operatori dodele vrednosti	= += , - = , *= , /= , %=	Dodela vrednosti Dodela vrednosti sa primenom aritmeti;ke operacije
Operator za string objekte	+ Instanceo f	Konkatenacija stringova Operator za proveru pripadnosti nekog objekta klasi

Java operatori



Java

nizovi

- Način za čuvanje liste elemenata, koji pripadaju istom osnovnom tipu podatka ili klasi.
- Kreiranje niza u Javi podrazumeva
  - Deklaraciju promenljivih niza (navođenjem njegovog tipa i imena)
    `String reci[]; String[] reci;`
  - Kreiranje objekta niza
    - Korišćenjem operatora new
      - `String[] brojevi = new String [5];`
    - Direktnim inicijalizovanjem niza
      - `String[] brojevi = {“0”,”1”,”2”,”3”,”4”}`
  - Smeštanje podataka u niz `brojevi[1]=”1”;`
- Višedimenzioni nizovi se deklarišu na sledeći način
  - `Int brojevi = new int[10][10];`

Java

višedimenzionalni nizovi

```
int a[][] = { {1, 2, 3 },  {4, 5, 6 } };
```

```
int a[][] = new int[2][3];
```

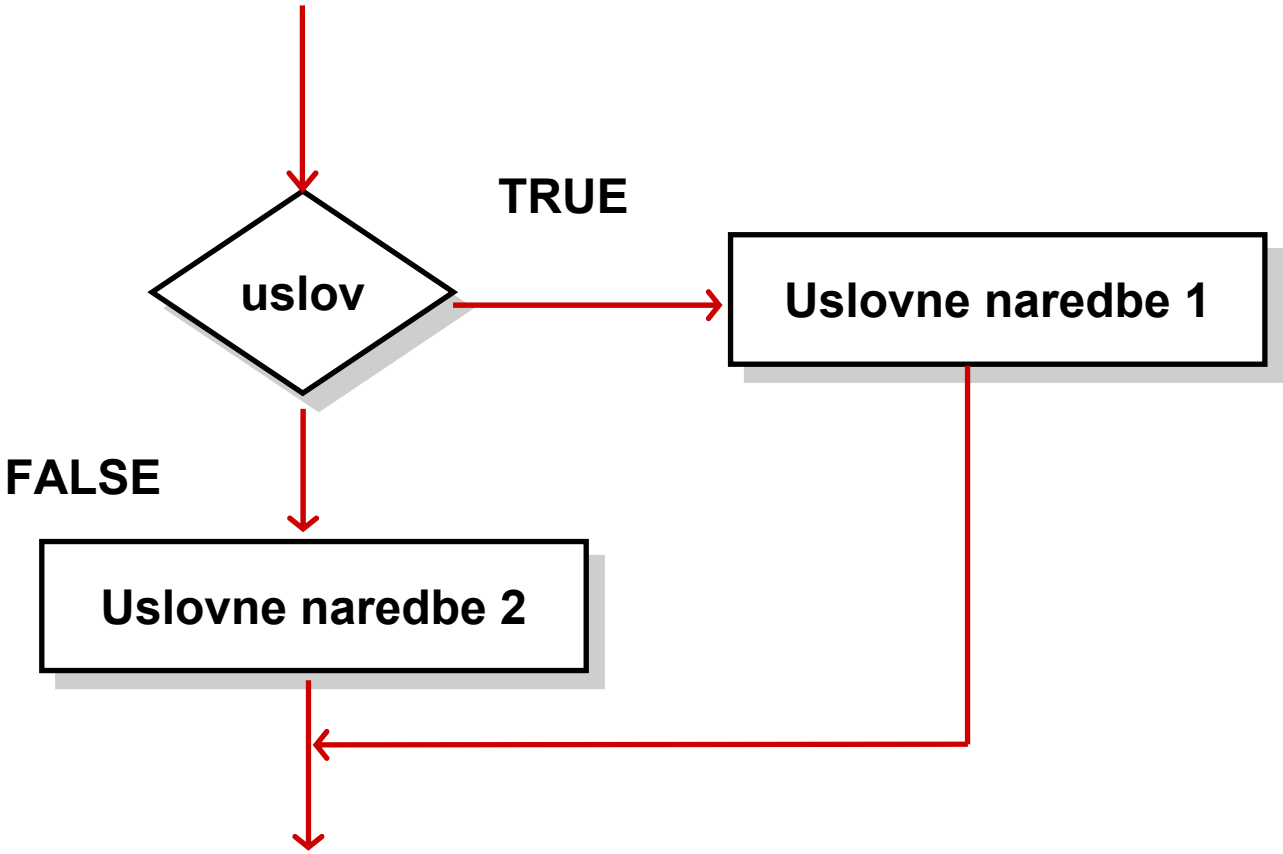
```
int a[][] = new int[2][];
for(int i = 0; i < a.length; i++) {
  a[i] = new int[3];
}
```

```
Automobil[][] a = {
   { new Automobil(), new Automobil() },
   { new Automobil(), new Automobil() }
};
```

Java

upravljačke strukture

- Naredbe izbora (selekcije)
  - `if/else,`
  - `switch`
- Naredbe iteracije (petlje)
  - `for,`
  - `while,`
  - `do while`
- Naredbe za obradu izuzetaka
  - `try/catch/finally`

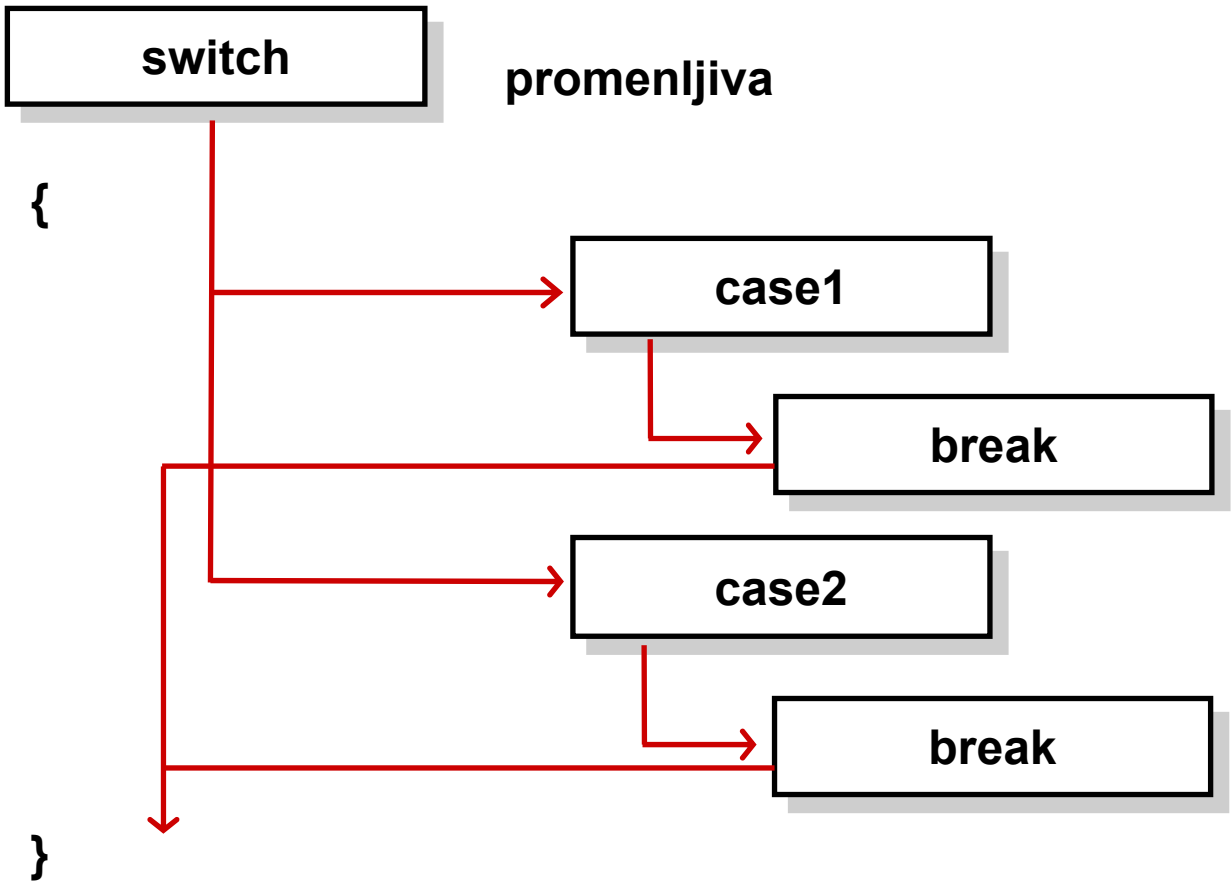

TRUE
uslov
Uslovne naredbe 1
FALSE
Uslovne naredbe 2

if

```
if (uslov)
 { uslovne_naredbe_1
  }
else
 { uslovne_naredbe_2
  }
```

- **if,else -** ključne reči
- **izraz -** može biti tačan ili netačan
- **uslovne_naredbe_1 -** izvršavaju se ako je izraz tačan
- **uslovne_naredbe_2 -** izvršavaju se ako je izraz netačan

switch



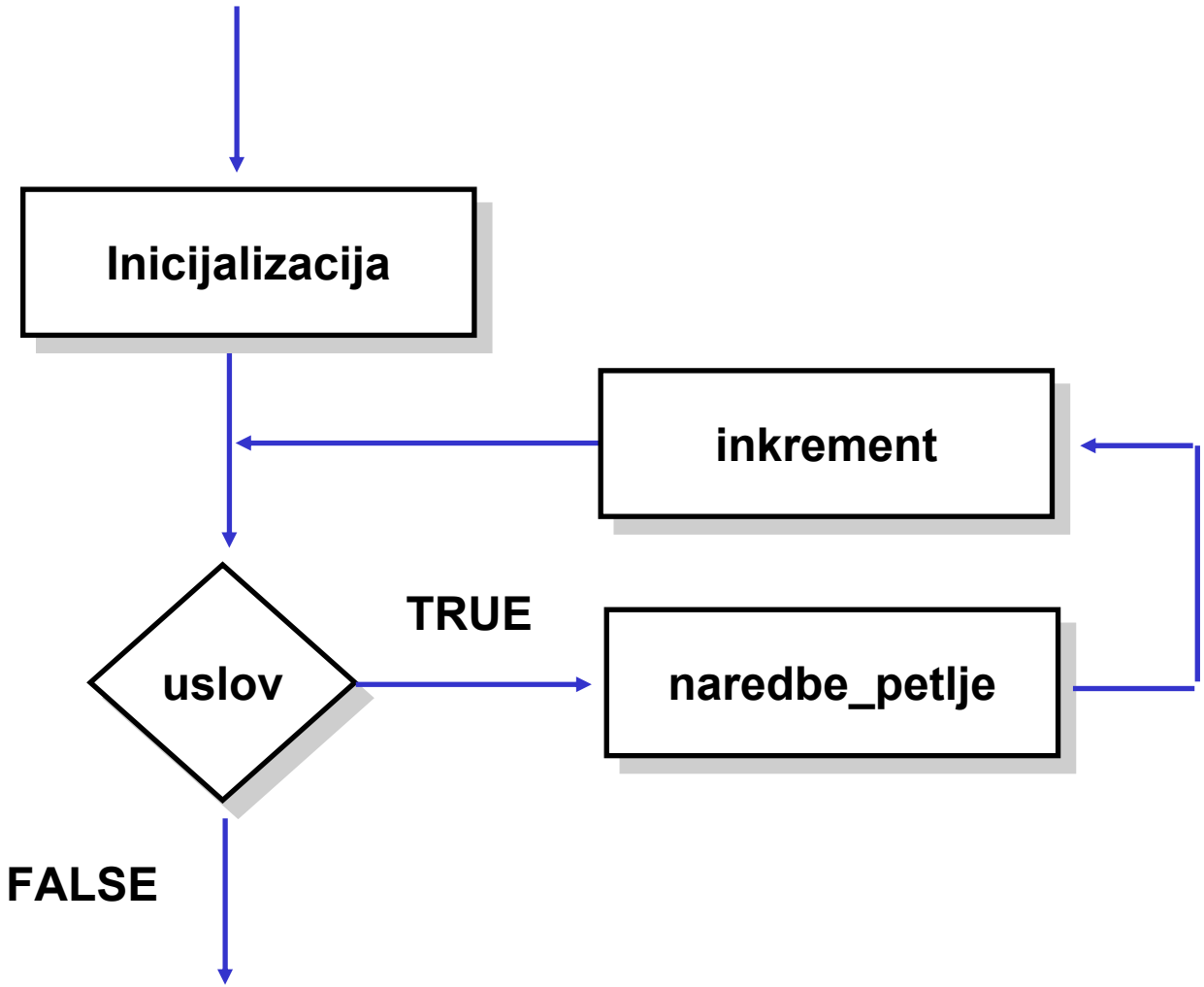
switch

```
switch (promenljiva)
 { case(vrednost1):
        uslovne_naredbe_1
    break;
   case(vrednost2):
        uslovne_naredbe_2
    break;
    ...
   default: uslovne_naredbe
 }
```

- **switch -** identifikuje promenljivu na osnovu koje program odlučuje gde dalje da ide
- **case -** početak svake grane
- **break -** obeležava kraj svake grane
- **default -** ako nijedna promenljiva (char, int, short, byte) naredbe switch ne odgovara ni jednoj vrednosti case

Java

for petlja

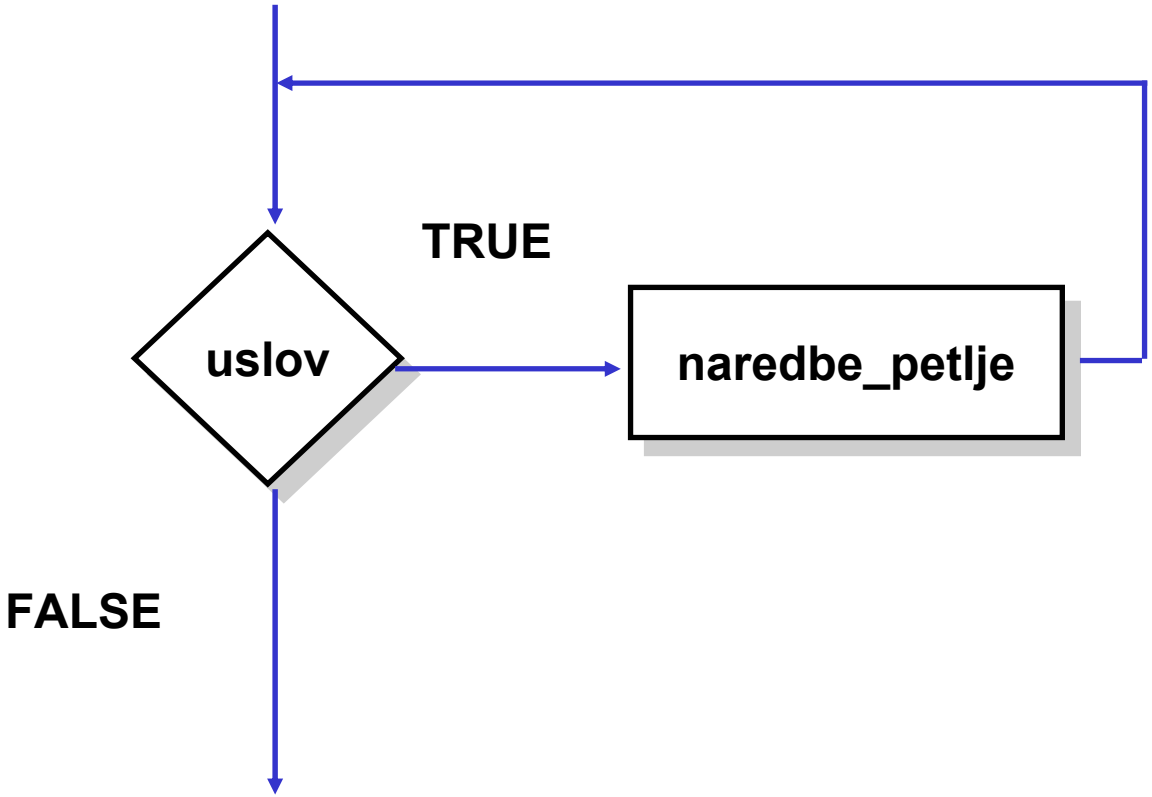


for petlja

```
for ( inicijalizacija ; izraz ; inkrement )
 {
   naredbe_petlje;
  }
```

- **Inicijalizacija -** postavlja početnu vrednost izraza
- **izraz -** testira se na početku svakog prolaza kroz petlju
- **Inkrement -** izvršava se na kraju svakog prolaza kroz petlju.

while



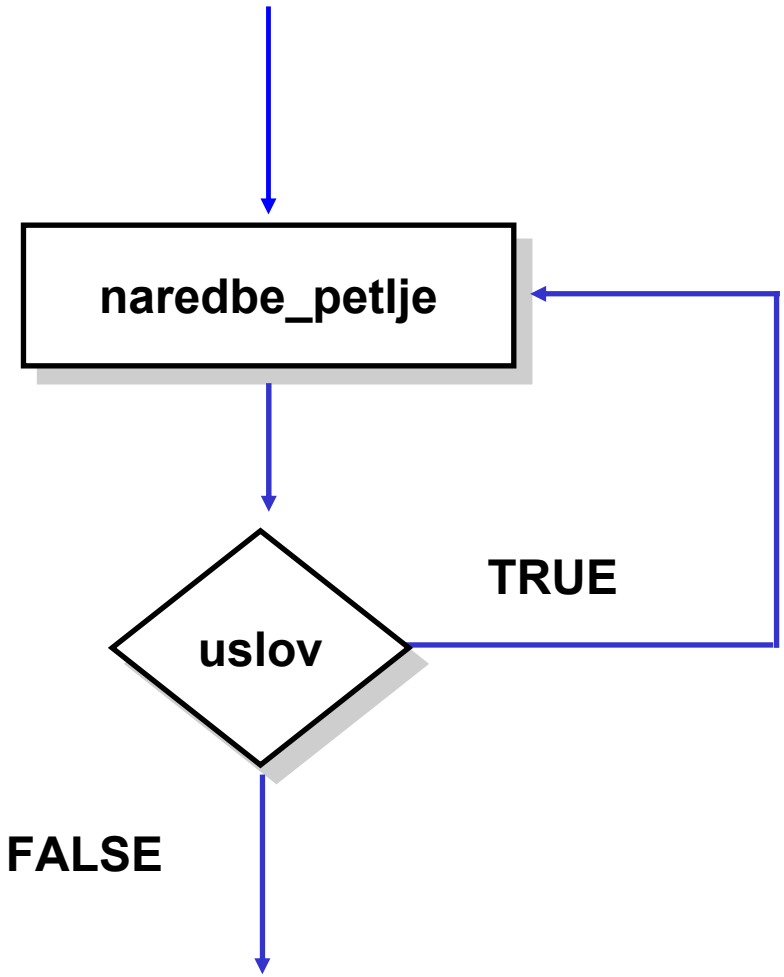
while

```
while( uslov )
 {
   naredbe_petlje;

  }
```

- **izraz -** testira se na početku svakog prolaza u petlju
- **naredbe_petlje –** izvršavaju se ako je uslov tačan.

do/while



do/while

```
do
 {
   naredbe_petlje;

 } while (uslov)
```

- **naredbe_petlje –** izvršavaju se sigurno jedanput i svaki sledeći put za koji je uslov zadovoljen.
- **izraz -** testira se posle prvog a pre svakog novog ulaska u petlju

try/catch/finaly

```
try {
      // izvršne_naredbe_1;
    } catch (Neki_izuzetak e)
       {
          // naredbe_za_obradu;
       }
finally {
          // izvršne_naredbe_2;
       }
```

- **try** – grupa naredbi u kojima se može pojaviti izuzetak ili abnormalni prekid (break, continue)
- **catch** - Označava početak obrade izuzetaka koji je tipa Neki_izuzetak ili podklase tog tipa
- **finally** – Naredbe koje se izvršavaju bez obzira na to da li je došlo do pojave izuzetaka ili je normalno izvršena grupa naredbi u `try`

JOŠ JEDNOM

- Standardno, **proceduralno** programiranje (npr. C):

  Program započinje izvršavanjem funkcije **main** koja izvršava postavljeni zadatak pozivanjem drugih funkcija. Program završava kad se izvrše sve instrukcije funkcije main. Osnovni građevni blok programa je, dakle, **funkcija**. Postavljeni zadatak se rešava tako da što se razbije na niz manjih zadataka od kojih se svaka može implementirati u jednoj funkciji, tako da je program niz funkcijskih poziva.

- U **objektno-orijentisanom** programiranju osnovnu ulogu imaju **objekti koji sadrže i podatke i funkcije (metode)**. Program se konstruiše kao skup objekata koji međusobno komuniciraju. Podaci koje objekat sadrži predstavljaju njegovo **stanje**, dok pomoću metoda on to stanje da može menja i komunicira sa drugim objektima.

Hello, World! u Javi

```
1  // Prvi program: HelloWorld.java
2  // Ispisuje tekst "Hello, World"
3
4  public class HelloWorld{
5  // main metoda kojom počinje izvršavanje svake Java aplikacije
6     public static void main(String[] args){
7           System.out.println("Hello, World");
8     } // kraj main metode
9   } // kraj klase HelloWorld
```

Objektno-orjentisano programiranje u Javi

- Dakle, u osnovi svega je objekat koji:
  - ima podatke
  - ima metode
  - ima jedinstvenu adresu u memoriji
  - predstavlja instancu neke konkretne klase
- Java je potpuno objektno-orjentisan jezik i zato je

  sav izvorni kod u Javi podeljen u klase

  (koje se nalaze u istoj ili različitim izvornim datotekama).
- Sve klase (i sistemske i naše) u Javi su direktno ili indirektno izvedene iz klase **Object**. Ako se eksplicitno ne navede nadklasa neke klase, onda je ona uvek **java.lang.Object**. Npr. HelloWorld se, zapravo, prevodi kao:

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

Klase

- Definisanje klase

```
// Tacka.java
    class Tacka{
        private double x;
        private double y;
        public Tacka(){
                x=0.0;
                y=0.0;
        }
        public Tacka(double a, double b){
                x=a;
                y=b;
        } //kraj konstruktora
        public double getX() { return x; }
        public double getY() { return y; }
    } //kraj klase Tacka
```

podaci članovi

konstruktori

funkcije članice metodi

Klase

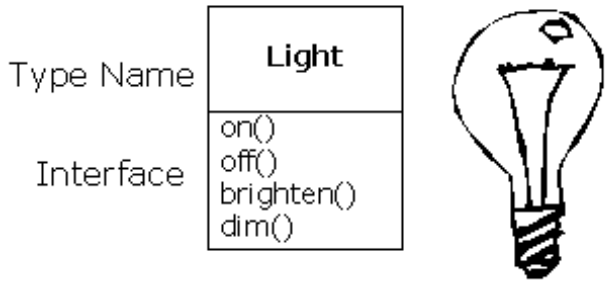
```
// Test.java
    public class Test {
    public static void main(String[ ] args){
        Tacka tc = new Tacka();
     System.out.println(tc.getX());     }
}
```

- Konstruktor ima isto ime kao i klasa
- Konstruktor nema povratne vrijednosti
- Klasa može imati više konstruktora koji se međusobno razlikuju po broju i tipu parametara koje uzimaju (overloading na delu)
- Konstruktor bez parametara je default konstruktor
- Konstruktor se uvijek poziva operatorom **new**

Metodi

- skup metoda - interface

```
returnType methodName(/* Argumenti */)
 {
  /* Telo metoda */
 }
```



- Metodi se definišu **samo kao deo klase**. Pozivi pogrešnih metoda za neki objekat se registruju pri kompjliranju.

```
int x = a.f(); // a je objekat odgovarajuce klase
```

- Poseban tretman imaju static metodi, kao i static podaci.

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nothing() { return; }
```

- Vraćanje sa bilo koje tačke, ali sa odgovarajućim tipom.

Modifikatori

- Uz varijable je moguće koristiti modifikatore
  - **static** -označava varijablu koja je zajednička svim objektima koji su instance date klase
  - **final** -definiše konstante

  npr. **public static final double pi=3.14**

- Uz metode možemo koristiti:
  - **static** - metoda koja je ista za svaki objekat date klase - kaže se još i class member npr. takva je metoda main
  - **final** - ne može se preraditi u nasleđenoj klasi
  - **native** - metoda koja je implementirana u drugom jeziku (najčešće C ili C++) - sve fundamentalne metode u API su takve
  - **synchornized** - koristi se u radu sa nitima

Statičke promenljive i metode

- Static označava nešto zajedničko svim instancama date klase. Npr. kada deklarišemo **static int x;** kao varijablu članicu klase, onda runtime environment kada naiđe na prvu instancu date klase alocira potrebnu memoriju za tu varijablu koju kasnije dele sve druge instance ove klase.

- Metode deklarisane sa static se tretiraju slično.

  public static void main(String[] args)

- Da bismo koristili statičku metodu neke klase nije potrebno imati instancu te klase. Statičkim metodama se pristupa sa:

  (ime_klase).(ime_statičke_metode)

  npr. **double x=Math.sqrt(2);**

  Math je klasa iz paketa java.lang u kojoj je sqrt statički metod

- Za statičke metode je važno napomenuti da one **ne mogu koristiti nestatičke** metode iste klase direktno, niti pristupati nestatičkim varijablama.

Statičke promenljive i metode

```
class Test {
    int x;              // nestatička javna varijabla
    static int y;

    public void print() { // nestatička metoda
            System.out.println("Hello, World");
    }
    public static void print_static() { // statička metoda
            System.out.println("Hello, World");
    }
    public static void main(String[] args) {
            x=3;                    //error
            print();                //error
            print_static();         //OK
            new Test().x=5;         //OK
            Test test=new Test(); //OK
             test.print();          //OK
    }
}
```

Statičke promenljive i metode

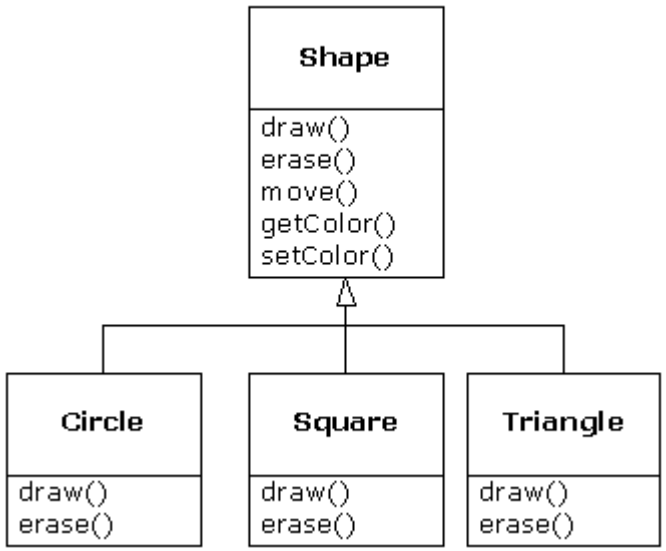
```
class StaticTest {
    static int i = 47;
}
     StaticTest st1 = new StaticTest();
     StaticTest st2 = new StaticTest();
     StaticTest.i++;
     st1.i++;
class StaticFun {
  static void incr() { StaticTest.i++; }
}
     StaticFun sf = new StaticFun();
     sf.incr();
     StaticFun.incr();
```

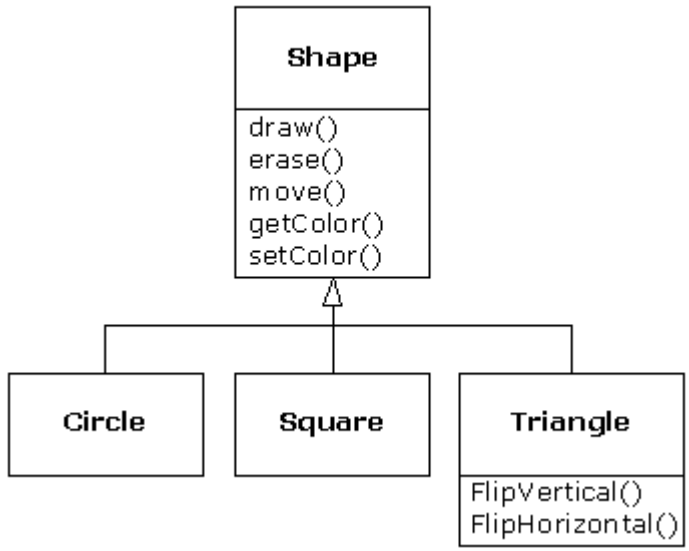
Nasleđivanje

- Dodatne karakteristike i funkcionalnosti, kao i drugačija ponašanja (overriding)
- Super klasa (nadklasa, roditeljska klasa)

  Podklasa (dete-klasa, izvedena klasa)

```
public class Kvadrat extends Pravougaonik
{
        //telo klase
}
```





Nasleđivanje

- Klasa može da ima **samo jednu nadklasu**
- Svaka klasa može da ima neograničen broj podklasa
- Podklase nisu ograničene na promenljive, konstruktore i metode klase koje nasleđuju od svoje roditeljske klase
- Podklase mogu dodati i neke druge promenljive i metode ili predefinisati stare metode i konstruktore.
- U deklaraciji podklase navode se razlike između nje i njene superklase.
- Ukoliko se u podklasi definiše metoda sa istim imenom, povratnom vrednošću i argumentima kao i metoda superklase tada se ovom metodom prepisuje (overriding) metoda superklase.
- **final** metodi ne dozvoljavaju overriding

Apstraktna klasa

- objedinjuje zajedničke karakteristike većeg broja klasa
- ima navedene sve metode koje će imati njene podklase, ali nisu definisani (implementirani).
- deklariše se upotrebom ključne reči **abstract**
- ne može biti instancirana
- ako u nekoj klasi **postoji apstraktna metoda tada i sama klasa mora da bude definisana kao apstraktna**, obrnuto ne mora da važi
- ukoliko podklasa neke apstraktne klase **nema implementirane sve nasleđene apstraktne metode** onda je i sama **podklasa apstraktna**

Apstraktna klasa

```
public abstract class GeometrijskiOblik {
    private Color c;
    public abstract double obim();
    public abstract double povrsina();
    public Color dajBoju ()
    {
        return c.getForeground() ;
    }
        ...
}
```

apstraktni metodi nemaju implementaciju,
tj. definisano telo

Interfejsi

- omogućavaju **VIŠESTRUKO NASLEĐIVANJE**
- **sve metode interfejsa su implicitno apstraktne**, a kao promenljive mogu se definistati samo statičke konstante

```
public interface Crtanje {
  //definicija interfejsa
  public void postaviBoju(Color c);
  public void isrctaj(DrawWindow prozor);
}
```

- nakon što se za neku podklasu definiše da je izvedena (extends) iz neke nadklase, mogu se opciono implemetirati (implements) jedan ili više interfejsa
- kada se u nekoj klasi definiše da implemetira neki interfejs tada se u njoj **moraju implementirati i sve metode interfejsa**
- jedan interfejs može biti izveden iz drugog

Interfejsi

```
class Krug extends GeometrijskiOblik implemets Crtanje {
   // deklaracija promenljivih
   public double x,y;
   public double r;

   // implemetacija nasledjenih apstraktnih metoda
   public double obim()     { return 2 * pi * r; }
   public double povrsina() { return pi * r*r; }

   // implemetacija metoda interfesa
   public void postaviBoju (Color c) { this.c = c; }
   public void isrctaj(DrawWindow dw) {
            ... //telo metode
   }
}
```

'Skrivanje'

- Zašto?

  Class creators vs. **client programmer**
- zaštita podataka i metoda

vidljivost / tip	Dostupno (vidljivo)			
	klasi	podklasi	paketu	bilo kome
private	X			
protected	X	X*	X	
public	X	X	X	X
package	X		X	

private

```
class Alpha {
    private int iamprivate;
    private void privateMethod() {
        System.out.println("privateMethod");
    }
}
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprivate = 10;      // illegal
        a.privateMethod();      // illegal
    }
}
```

vidljivost / tip	Dostupno (vidljivo)			
	klasi	podklasi	paketu	bilo kome
private	X			

private

```
Beta.java:9: Variable iamprivate in class Alpha not accessible from class
    Beta.
      a.iamprivate = 10;     // illegal
       ^
1 error
Beta.java:12: No method matching privateMethod()
found in class Alpha.
      a.privateMethod();        // illegal
1 error
```

private

- dozvoljeno je sledeće

```
class Alpha {
    private int iamprivate;
    boolean isEqualTo(Alpha anotherAlpha) {
      if (this.iamprivate==anotherAlpha.iamprivate)
            return true;
        else
            return false;
    }
}
```

protected

```
package Greek;
public class Alpha {
    protected int iamprotected;
    protected void protectedMethod() {
        System.out.println("protectedMethod");
    }
}
package Greek;
class Gamma {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iamprotected = 10;    // legal
        a.protectedMethod();    // legal
    }
}
```

* ako su u istom paketu

vidljivost / tip	Dostupno (vidljivo)			
	klasi	podklasi	paketu	bilo kome
protected	X	X*	X	

protected

```
package Latin;

import Greek.*;

class Delta extends Alpha {
    void accessMethod(Alpha a, Delta d) {
        a.iamprotected = 10;    // illegal
        d.iamprotected = 10;    // legal
        a.protectedMethod();    // illegal
        d.protectedMethod();    // legal
    }
}
```

public

```
package Greek;
public class Alpha {
    public int iampublic;
    public void publicMethod() {
        System.out.println("publicMethod");
    }
}
package Roman;
import Greek.*;
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampublic = 10;       // legal
        a.publicMethod();       // legal
    }
}
```

vidljivost / tip	Dostupno (vidljivo)			
	klasi	podklasi	paketu	bilo kome
public	X	X	X	X

package (default)

```
package Greek;
class Alpha {
    int iampackage;
    void packageMethod() {
        System.out.println("packageMethod");
    }
}
package Greek;
class Beta {
    void accessMethod() {
        Alpha a = new Alpha();
        a.iampackage = 10;     // legal
        a.packageMethod();     // legal
    }
}
```

vidljivost / tip	Dostupno (vidljivo)			
	klasi	podklasi	paketu	bilo kome
package	X		X	

Kreiranje objekta

- Sve čime se manipuliše je objekat neke klase♀ (čak i sama aplikacija/applet), odnosno **referenca na objekat**.

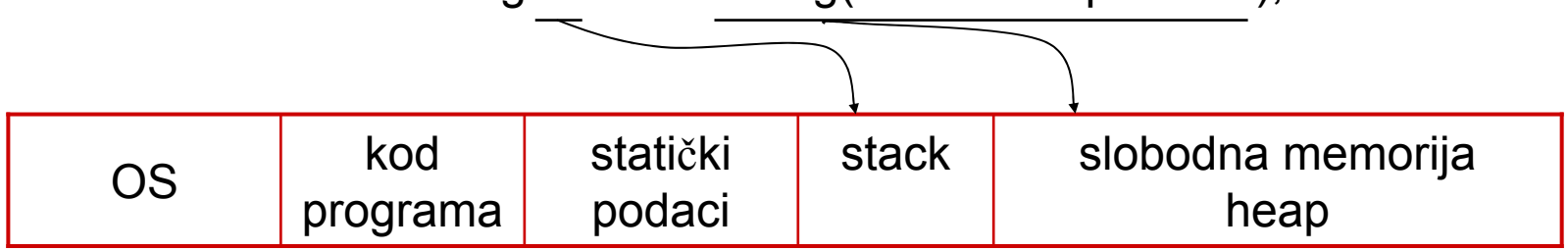
  Objekat – TV

  Referenca – daljinski

- Sa sobom nosite daljinski, a ne televizor.

  Možete kupiti daljinski i bez televizora.

```
String s; // kreirana je samo referenca
String s = new String("sad ste kupili i TV");
```



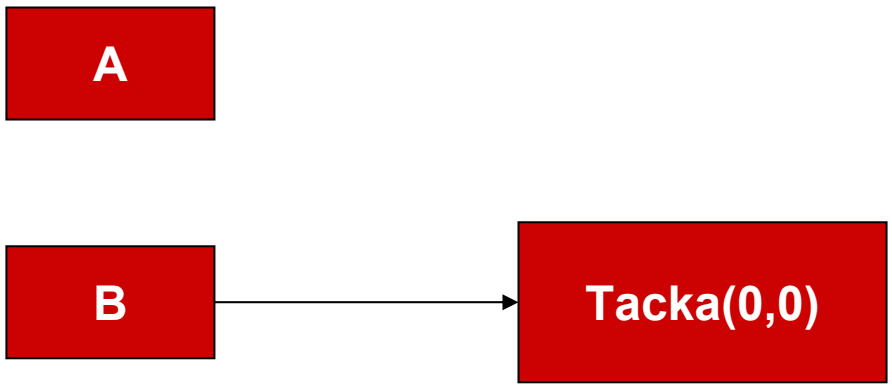
Raspodela memorije između OS i korisničkog programa

Primitivni tipovi – izuzetak

- Veličina primitivnih tipova se ne menja od računara do računara
- nema referenci i nalaze se na steku
- svi numerici su označeni

Reference

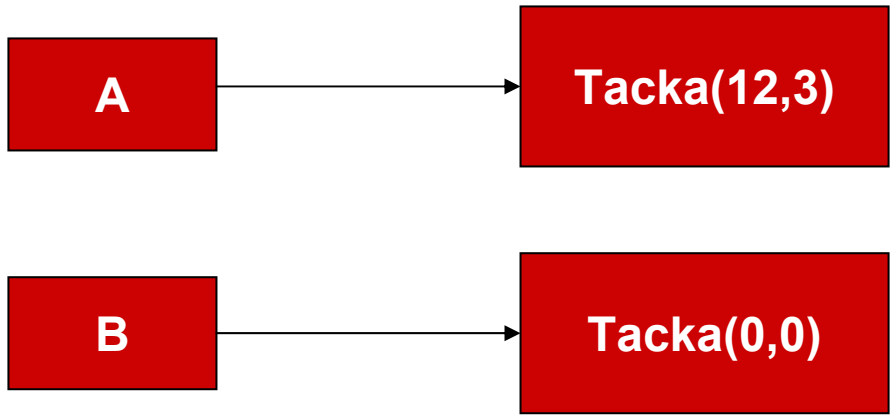
```
int x;                          // x je varijabla primitivnog tipa
Tacka A;                        // A je objektna varijabla (neinicijalizovana)
Tacka B = new Tacka(0,0); // B je objektna varijabla (inicijalizovana)
```



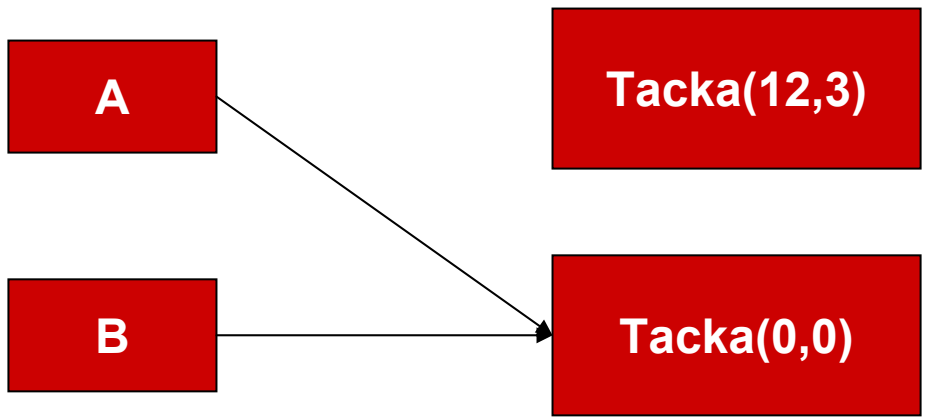

```
double z = A.getX();  // greska, objekat ne postoji
double w = B.getX(); // OK
```

Reference

A = new Tacka(12,3);



A = B;



Oblast važenja

- Oblast važenja (**scope**) podrazumeva vidljivost i životni vek ‘imena’ i Java je definiše slično C-u i C++-u

```
{ int x = 12;
  /* samo je x dostupno*/
  { int q = 96;
       /* x i q su dostupni */
   }
   /* samo x je dostupno a q ‘ne postoji’ */
}
```

- Za razliku od C-a u Javi nije dozvoljeno sledeće

```
{ int x = 12;
  {
    int x = 96; /* illegal */
  }
}
```

Oblast važenja

- **Životni vek objekata nije isti životnom veku primitivnih tipova**. Nakon kreiranja objekat postoji i posle }, jer je iz oblasti važenja ‘izašla‘ samo referenca

```
{
        String s = new String("a string");
}
```

- Kada objekat više nije potreban, tj. nije referenciran ni jednom referencom onda biva automatski oslobođen **garbage collector**-om

Inicijalizacija

```
class ATypeName {...};
TypeName a = new ATypeName();
class DataOnly {
  int i;
  float f;
  boolean b;
}
DataOnly d = new DataOnly();
d.i = 47;
d.f = 1.1f;
d.b = false;
myPlane.leftTank.capacity = 100;
```

- Default inicijalizacija za važi samo za podatke članove objekta, ali ne i za lokalne varijable metoda (kompajler javlja grešku)

Napomena o nizovima

- **Nizovi su uvek inicijalizovani** i ne može im se pristupiti van definisanog opsega (pri izvršavanju se proverava indeks)
- Pri kreiranju niza objekta kreira se niz referenci i svaka od tih referenci je automatski postavljena na null
- Ovo važi i za niz primitivnih tipova

Vidljivost imena

- Vidljivost imena
  - kako izbeći sukobljavanje imena iz različitih modula programa
  - C++ - funkcije u okviru klase, namespaces za globalne funkcije i promenljive
  - u Javi user biblioteke se imenuju obrnutim redoslem domena

    edu.firma.autor.utility.imebiblioteke
- Korišćenje postojećih komponenti
  - ako se koristi objekat klase koje se definiše u okviru istog fajla onda je dovoljan poziv (“forward referencing” nije obavezan)
  - traženje u drugom fajlu je delimično moguće
  - ako postoji definisana biblioteka
    import java.util.ArrayList;
    import java.util.*;

Još malo o objektima i referencama

Pošto se svim 'podacima' (osim onih koji su primitivnog tipa) pristupa preko reference očigledno je da:

- operator dodele vrednosti (=) i poređenja (==) 'rade' samo sa referencama na objekte (naravno, kada se primenjuju na objekte)

  za kopiranje stanja objekta korisiti se metoda clone(), a za poređenje metoda equals()

- promenjliva koja pripada složenom tipu (“reference type”) ne može nikada biti dodeljena promenljivoj primitivnog tipa i obrnuto

Ključna reč This

- Postavlja se pitanje: kako objekat pamti funkcije članice, odnosno kako objekat komunicira sa njima? Kompajler za svaki objekat inicira jednu pokazivačku varijablu u koju zapisuje adresu samog objekta. Ta varijabla ima status privatne varijable klase i naziva se **this**. Ova pokazivačka varijabla se koristi pri pozivu metoda objekta kao skriveni argument, koji služi da funkcija dobije poruku o adresi objekta koji obrađuje (koji ju je pozvao).
- Svaka nestatička metoda klase poziva se s instancom svoje klase. Npr.

```
Tacka A = new Tacka(3,3);
.........
double x = A.getX();
```

- Sintaksa poziva je **objektnaVarijabla.imeMetode(...)**. Pri pozivu, metoda dobija pored eksplicitnih parametara i jedan implicitni parametar, to je objekt na kojem je metoda pozvana. U svakoj metodi ključna riječ **this** referira na implicitni parametar.
- Implicitni parametar može biti koristan u više situacija. Npr, ako formalni argumenti konstruktora imaju ista imena kao i varijable članice koje inicijaliziraju, onda moramo koristiti this jer lokalne varijable skrivaju varijable članice.

Ključna reč This

- Implicitni parametar može biti koristan u više situacija. Npr, ako formalni argumenti konstruktora imaju ista imena kao i varijable članice koje inicijaliziraju, onda moramo koristiti this jer lokalne varijable skrivaju varijable članice.

```
class Tacka{                    // Tacka.java
        private double x;
        private double y;
        public Tacka(){
                x=0;
                y=0;
        }
        public Tocka(double x, double y){
                this.x=x;
                this.y=y;
        }
        public double getX(){ return x;}
        public double getY(){ return y;}
}
```

Ključna reč This

- Druga uloga ključne riječi this je poziv konstruktora.

```
public Tacka(){
        this(0,0);
}
```

- **this** ima ovakvu ulogu samo kad se pojavljuje kao prva naredba u konstuktoru

- Statičke metode za razliku od nestatičkih ne dobijaju implicitni parametar. To je razlog zbog kojeg one ne mogu pristupiti nestatičkim varijablama članicama, niti pozvati nestatičke metode.

Ključna reč Super

- U Javi ključna reč **super** se koristi da ukaže na klasu koja je na prvom višem nivou hijerarhije od klase, koja u okviru neke od svojih metoda, koristi ključnu reč super. Postoje dva slučaja korišćenja ključne reči super:
  - kada se želi pristupiti članicama nadređene klase
  - kada se želi pristupiti konstruktoru nadređene klase

  Opšti oblik pristupa konstruktoru nadređene klase je:

  Super(lista parametara) ili **Super()** za default konstruktore

Ključna reč Super

- Kad se konstruiše objekat klase koja proširuje neku dugu klasu (a takve su sve osim java.lang.Object), onda se prvo konstruiše deo objekta koji pripada nadklasi, a zatim onaj deo kojim klasa proširuje svoju nadklasu. Mehanizam konstrukcije nadklase je sledeći: prevodilac u svaki konstruktor klase ubacuje kao prvu naredbu poziv default konstruktoru nadklase (konstruktor bez argumenata). Taj mehanizam deluje rekurzivno, tako da se uvijek prvo konstriše najviša nadklasa.

- Ukoliko nadklasa nema default konstruktor, prevodilac će javiti grešku. S druge strane, default konstruktor ponekad nije dovoljan za konstrukciju nadklase. Stoga Java omogućava eksplicitan poziv konstruktora nadklase pomoću ključne reči **super**. Ona ima sličnu ulogu kao ključna reč **this**: dok this referira na klasu, **super** referira na nadklasu. Pomoću this možemo pristupiti varijabli članici klase (ako npr. skriva lokalna varijabla), a pomoću super možemo pristupiti varijabli članici nadklase (ako je npr. skriva varijabla članica klase). Kao što pomoću this možemo pozvati konstruktor klase, tako pomoću super možemo pozvati konstruktor nadklase. Naravno, ključna reč super mora se pojaviti prva u konstruktoru.

Ključna reč Super

```
class A{                                                    // B.java
   private int x;
   protected int y;
   public int z;
   public A( ){       x=y=z=0;
                      System.out.println("Objekat klase A je kreiran!");
   }}
public class B extends A{
   private int   w; // varijabla koja nije nasleđena od A
   protected int y; // prerađena varijabla
   public B(){ //konstruktor za B
            y=super.y+1; //vrednost sadržana u y je 1
            System.out.println("super.y=" + super.y);
            System.out.println("y=" + y);
            System.out.println("z=" + z);
    // w=super.x+1; // greška jer pristupamo privatnoj varijabli klase A
   // w=x+1;      // takođe greška jer pristupamo privatnoj varijabli klase
            System.out.println("Objekat klase B je kreiran!");
   }
   public static void main(String args[]){
            new B(); //ispisuje poruke iz konstruktora
   }}
```

Bitne napomene o polimorfizmu

```
public interface A{
     void nekaMetoda();
}
public class B {
   public void uciniNesto(A a){...}
   public void uciniNestoDrugo(A a){...}...
}
public class C implements A{
   public void nekaMetoda() { /* implementacija 1 */ }
}
public class D implements A{
   public void nekaMetoda() { /* implementacija 2 */ }
}
public class E {
   public static void main(String[] args){
            A x;        //možemo imati varijablu tipa interfejsa
            x=new C();  //referenca na konkretnu implementaciju
            A y=new D(); //drugačija implementacija interfejsa
            /*   A a=new A(); je greška */
            B b=new B();
            b.uciniNesto(x);
            b.uciniNesto(y);}}
```

Pravo pristupa metode je automatski public. S druge strane, u implementaciji metode public se mora navesti.

argument svaki objekt koji implementira interfejs A

Bitne napomene o polimorfizmu

- Metode klase B zavise samo od onoga što je deklarisano u A, a ne od konkretne implementacije kao što su C i D. Međutim, **b.uciniNesto(x)** i **b.uciniNesto(y)** koristi odgovarajuće implementacije interfejsa A date u C i D, respektivno.
- Linija koda: **B b=new B();**
  kreira objekat koji sadrži adresu gde se nalazi implementacija metode **uciniNesto()**, a ta metoda zavisio od metode **nekaMetoda()** deklarisanoj u A. Konkretno ona mora znati adresu na kojoj se nalazi implementacija **nekaMetoda()**. Ta se adresa ne razrešava tokom prevođenja programa, ali se razrešava na dva različita načina tokom izvođenjena programa i to u iduće dvije linije koda:
  b.uciniNesto(x);
  b.uciniNesto(y);
- Jedna adresa implementacije metode nekaMetoda() dolazi od objekta x (čiji tip je klasa C), a druga od objekta y (čiji tip je klasa D).

Bitne napomene o polimorfizmu

- Konkretno jezike možemo podeliti u dvije grupe prema tome **kada se funkcijski poziv** (poziv metode) **povezuje s adresom na kojoj se nalazi implementacija**:
  - **rano povezivanje** (eng. **early binding**) - tokom prevođenja programa. To je slučaj sa svakom funkcijom u C-u ili sa metodama koje su u Javi proglašene statičkim ili finalnim (privatne metode su po definiciji finalne - ne mogu se preraditi u podklasi).
  - **kasno povezivanje** (eng. **late binding**) - tokom izvršavanja programa. To je slučaj sa svim nestatičkim i nefinalnim metodama u Javi.

Implicitna i eksplicitna konverzija (cast)

- Pretvaranje jednog tipa podatka u drugi - konverzija
  - između primitivnih tipova
  - između objekata

- Koverzija primitivnih tipova
  - implicitni cast - manji tipovi u veće, i pri računanju vrednosti izaraza
    npr: int + double → double
    i pri dodeli vredosti izraza nekoj promenljivoj
    npr: double=float
  - eksplicitni cast – kopmajler ne dozvoljava skraćivanje vrednosti, npr double → float
    za tako nešto potrebno je direktno navođenje tipa na koji se svodi neka vrednost
    npr: int i = (int) 9.0/4.0;

Implicitna i eksplicitna konverzija (cast)

- Cast objekata je moguć u slučaju da radimo konverziju izmedju klasa koje pripadaju istom lancu nasledjivanja.

  Primer:

```
Class radnik{...}
Class rukovodilac extends radnik{
    void dodatak() {dohodak=dohodak*1.1}
}
…
radnik r=new radnik(…);
r.dodatak();  // nije dozvoljeno
rukovodilac s;
s=(rukovodilac)r;
s.dodatak(); //jeste dozvoljeno
```

- instanceof operator !

Apleti i aplikacije

- Apleti su mini programi priključeni Web stranama sa ograničenim mogućnostima korišćenja resursa računara na kojima se izvršavaju.
- Aplikacije samostalni Java programi koji se mogu pokrenuti i bez Web čitača i ne podležu ograničenjima koja važe za aplete.
- Glavna razlika je u main() metodu - aplikacije poseduju main() metod, dok ga apleti ne poseduju.

Primer apleta

```
import java.applet.*;
import java.awt.*;
public class HelloWorld extends Applet {
      Label helloLabel=new Label("Zdravo !");
      public void init() {
        setBackground(Color.yellow);
        add(helloLabel);
         }
   }
```

Fajl sa ovim kodom mora biti imenovan imenom klase izvedene iz Applet klase, dakle

HelloWorld.java

Naziv ovog fajla je nebitan. Npr. **vidiHelloWorld.html**

```
<html>
<title> HelloWorld primer </title>
<body>
<h1> Dakle,</h1>
 <applet code="HelloWorld.class" width=500 height=50>
 </applet>
</body>
</html>
```

testiranje apleta

1. Kompajliranjem

   javac HelloWorld.java

   se dobija bajtkod HelloWorld.class.

2. Aplet se startuje

   otvaranjem vidiHelloWord.html u Web browser-u

   ili

   zadavanjem komande

   appletviewer vidiHelloWorld.html

Aplet može

- crtati slike na web stranici
- kreirati novi prostor i crtati u njemu
- reprodukovati zvuk
- primati input od korisnika preko tastature ili miša
- povezivati se na server s kojeg je učitan, slati i primati podatke s tog servera

Aplet ne može

- pisati podatke na bilo koji disk na hostu
- čitati podatke s hostovog diska bez korisnikove dozvole. U nekim okruženjima, npr Netscape, applet ne može čitati podatke s korisnikovog diska čak ni uz dozvolu
- brisati datoteke
- čitati ili pisati po bilo kojem bloku memorije, čak i u nezaštićenim operativnim sistemima (sav pristup memoriji je strogo kontrolisan).
- povezivati se s hostovima na Internetu, osim sa hostom s kojeg dolazi (bar ne po defult sigurnosnoj šemi)
- pozivati direktno native API
- unositi viruse na host

hijerarhija klasa

java.lang.Object
- java.awt.Component
  - java.awt.Container
    - java.awt.Panel
      - **java.applet.Applet**

jdk1.2.2\docs\index.html →Java Platform 1.2 API Specification

životni ciklus apleta

primeri 1,1a

```
import java.applet.*;

public class LifeCycleApplet extends Applet
{           public void paint(Graphics g) {    }
            public void init() {    }
            public void start() {    }
            public void stop() {    }
            public void destroy() {    }
}
```

applet tag

```
< APPLET        [CODEBASE = codebaseURL]
                CODE = appletFile
                [ALT = alternateText]
                WIDTH = pixels
                HEIGHT = pixels
                [ALIGN = alignment]
                [VSPACE = pixels]
                [HSPACE = pixels]
>
[< PARAM NAME = appletParameter1 VALUE = value >]
[< PARAM NAME = appletParameter2 VALUE = value >]
. . .
[alternateHTML]
</APPLET>
```

izvršavanje apleta

- browser učitava HTML stranicu i pronalazi tag `<APPLET>`
- analizira je `<APPLET>` tražeći atribut `CODE` i eventualno `CODEBASE`.
- učitava .class datoteku appleta sa prethodno pronađenog URL-a
- konvertuje učitane bajtove u Java klasu
- instancira applet klasu kako bi formirao objekat učitanog appleta (to zahteva da applet ima konstruktor bez argumenata)
- poziva metod init()
- poziva metod start()
- dok se applet izvršava, browser mu šalje informacije o događajima koji su mu namijenjeni, npr. klik mišem, pritisak na tastere itd. preko appletove handleEvent() metode. Događaji koji rade update kažu appletu da se iznova prikaže (`repaint`)
- poziva metod `stop()`
- poziva metod `destroy()`.

user interface

- Korisnički interfejs je upravljan događajima (Event Driven).
- U verzijama 1.0 i 1.1 standard je AWT biblioteka (Abstract Window Toolkit).
- Od verzije 1.2 standard je Swing biblioteka.

awt

- AWT: biblioteka koja obezbeđuje upotrebu minimalnog skupa komponenti grafičkog interfejsa, a kojeg poseduju sve platforme koje podržavaju Javu.
- Izgleda “podjednako osrednje” na svim platformama.
- Paketi:
  - java.awt
  - java.awt.event
  - java.awt.image
  - java.awt.datatransfer

komponente

- Komponente su delovi korisničkog intefejsa (GUI - graphical user interface).
- U Javi, komponente su podklase java.awt.Component, a najčešće korišćene su:

Canvas	TextField	TextArea
Label	List	Button
Choice	Checkbox	Scrollbar

- Container - podklasa sa posebnom namenom
- Sve komponente se iscrtavaju samostalno (bez pisanja posebne paint() metode).

primer: Label

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

public class HelloContainer extends Applet {
    public void init() {
        Label l;
        l = new Label("Hello Container");
        this.add(l);
    }
}
```

`add` metod klase Container

Tri koraka u dodavanju komponente

- Deklarisanje komponente
- Inicijalizacija komponente
- Dodavanje komponente razmeštaju (layout)

U poslednjem primeru dodavanje labele je moglo biti izvedeno i ovako:

```
this.add(new Label("Hello Container"));
```

Nedostatak ovog kraćeg zapisa je taj što se izgubila referenca na labelu (varijabla l).

Iscrtavanje

- U appletu ne postoji paint(), a tekst se, ipak, ispisuje na ekranu. Komponente se same iscrtavaju. Svaki put kad se container kao što je applet ponovno iscrta, on pozove ne samo svoju vlastitu paint() metodu, nego i paint() metode svih svojih komponenti.
- Klasa java.awt.Label ima svoju paint() metodu koja zna kako se treba iscrtati.
- O iscrtavanju komponenti ne morate voditi računa dok god ne kreirate sopstvene klase komponenata ili izmenite izgled sistemskih komponenata

Metodi klase Label

- Podaci

  public final static int LEFT
  public final static int CENTER
  public final static int RIGHT

- Konstruktori

  public Label()
  public Label(String text)
  public Label(String text, int alignment)

- Metodi (bez nasleđenih)

  public void addNotify()
  public int getAlignment()
  public synchronized void setAlignment(int alignment)
  public String getText()
  public synchronized void setText(String text)

event driven programiranje u Javi

- Program se ne izvršava linearno
- Procedure se izvršavaju po pojavi nekog događaja (event) korisničkog interfejsa (klik mišem, pritisak tastera i sl.).
- Program ima inicijalizacioni blok i blokove koda koji reaguju na događaje korisničkog interfejsa.
- Događaji su predstavljeni objektima. Postoji nekoliko različitih vrsta događaja, a svaka vrsta je predstavljena klasom.
- Sve klase događaja korištene u AWT-u su podklase apstraktne klase java.awt.AWTEvent.
- Klase događaja koje opisuju tačno određene vrste događaja nalaze se u paketu java.awt.event.

event driven programiranje u Javi

- Program mora da otkrije događaj i odgovori na njega. Za svaku klasu događaja postoji interface koji određuje jednu ili više metoda za odgovor na događaje te klase.
- Na primer, za ActionEvent vezan je interface koji se zove ActionListener. On definiše metod

  public void actionPerfromed(ActionEvent evt)
- Objekt koji želi da odgovori na događaje mora da implementira ovaj interface.

```
public class MyApplet extends Applet implements
  ActionListener {
        . . .
    public void actionPerformed(ActionEvent evt) {
          . . . // odgovor na događaj
          }
          . . .
}
```

event driven programiranje u Javi

Događaje generišu komponente. Svaka komponenta koja može da generiše događaj ima definisanu metodu addActionListener, koja služi za prijavljivanje 'slušača ' događaja na komponenti. Kad komponenta stvori događaj, obaveštava sve prijavljene 'slušače' događaja pozivanjem njihovih actionPerformed() metoda.

Na primer, ako MyApplet kreira dugme commandButton i želi da reaguje kada korisnik klikne na dugme, onda mora pozvati metodu commandButton.addActionListener(this). Reč this se odnosi na sam aplet , a znači da će obaveštenje o događaju biti prosleđeno apletu pozivanjem actionPerfomed() metode apleta, odnosno pozivanjem njegovog ' obrađivača' događaja.

```
public void init() {
                . . .
                commandButton = new Button("Do It!");

commandButton.addActionListener(this);
                . . .
}
```

event driven programiranje u Javi

- Pozivanjem actionPerformed() metode, vrši se prosljeđivanje parametra tipa ActionEvent koji sadrži podatke o određenom događaju koji se dogodio.

- Metode za pristup ovim podacima su određene u ActionEvent klasi. Svaki tip događaja je predstavljen različitom klasom jer svaki zasebni tip događaja šalje 'slušačima ' različite podatke.

primer

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;

public class SimpleClick extends Applet implements MouseListener {
    StringBuffer buffer;
    public void init() {
    addMouseListener(this);
    buffer = new StringBuffer(); addItem("initializing... ");
    }
    public void start() { ...} public void stop() {...}
    public void destroy() {...}void addItem(String newWord) {...}
    public void paint(Graphics g) { ...}
    public void mouseEntered(MouseEvent event) {}
    public void mouseExited(MouseEvent event) {}
    public void mousePressed(MouseEvent event) {}
    public void mouseReleased(MouseEvent event) {}
    public void mouseClicked(MouseEvent event) {addItem("click!... ");}
}
```

šta još

- LayoutManager
- Crtanje

  http://laris.fesb.hr/java/java.htm
  http://student.math.hr/~vedris/

- Izuzeci

+

Niti