Java

Objekti i klase

Usage Modifiers

- 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

final

- The final modifier may be applied to a class, a method, or a variable. It means, in general, that the element is final.
  - `final` variable - means the value of the variable is constant, and cannot be changed.
  - `final` class - means the class cannot be extended,
  - `final` method cannot be overridden
- Primer

```
class Calculator {
   final int dime = 10;
   int count = 0;
   Calculator (int i) { count = i; }
}
class RunCalculator {
   public static void main(String[] args) {
        final Calculator calc = new Calculator(1);
        calc = new Calculator(2);
        calc.count = 2;
        calc.dime = 11;
        System.out.println("dime: " + calc.dime);}
}
```

final

- If you declare a final method inside a non-final class, it will be legal to extend the class, but you cannot override the final method of the parent class in the subclass.
- Similarly, you can pass the final variable to a method through arguments, but you cannot change their value even inside the method.

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 StaticCodeExample {
   static int counter=0;
   static {
        counter++;
   System.out.println("Static Code block: counter: " + counter);
   }
   StaticCodeExample() {
        System.out.println("Constructor: counter: " + counter);
   }
   static {
        System.out.println("This is another static block");
   }
}
public class RunStaticCodeExample {
   public static void main(String[] args) {
   StaticCodeExample sce = new StaticCodeExample();
        System.out.println("main: " + sce.counter);
   }
}
```

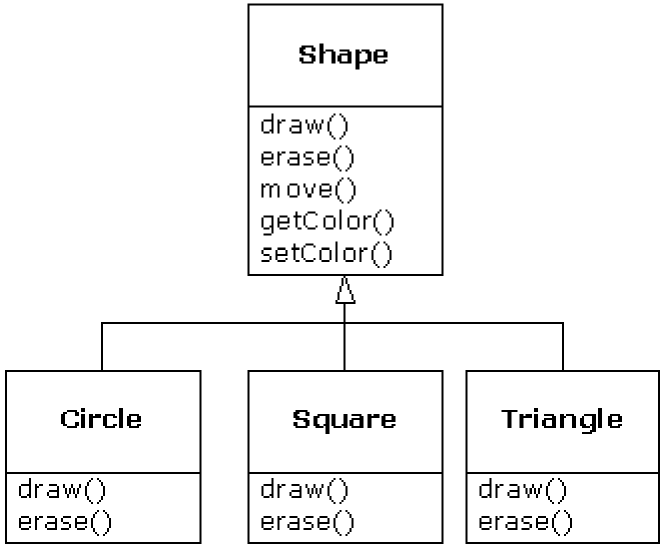
Statičke promenljive i metode

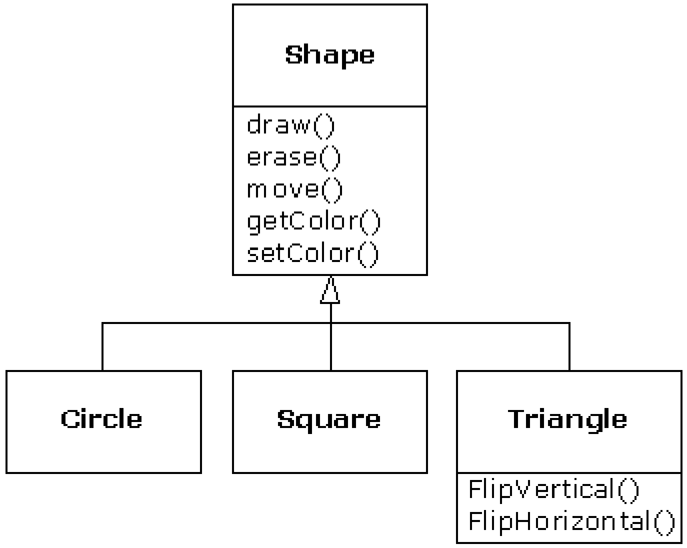
```
class StaticTest {
    static int i = 47;
}
   StaticTest st1 = new StaticTest();
   StaticTest st2 = new StaticTest();
   StaticTest.i++;
   st1.i++;
class StaticFun {
  static void incr() { StaticTest.i++; }
}
   StaticFun sf = new StaticFun();
   sf.incr();
   StaticFun.incr();
```

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

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

Objektno-orjentisano programiranje u Javi

- 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");}
 }
```

Apstraktna klasa

- objedinjuje zajedničke karakteristike većeg broja klasa, ima navedene sve metode koje će imati njene podklase
    - neki od njenih metoda **mogu** biti proglašeni apstraktnim i tada ne mogu biti definisani (implementirani); apstraktna klase ne mora imati ni jedan apstraktan metod
    - 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
    - uz apstraktnu klasu mogu se koristiti samo modifikatori `public, static i final`
- ukoliko podklasa neke apstraktne klase **nema implementirane sve nasleđene apstraktne metode** onda i sama **podklasa mora biti proglašena apstraktnom**

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

Apstraktna klasa

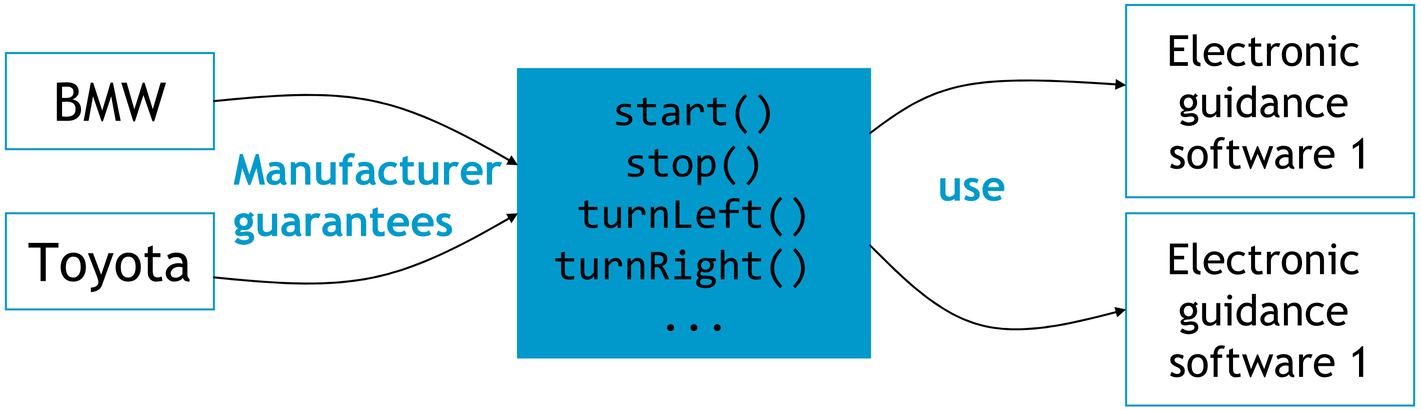
```
package paketic;

public abstract class absLine {
    int i;
    public void a(){}
    public abstract int b();
}
class conLine extends absLine {
    int kk;
    // protected int b() {return 1;}
    // public int b() {return 1;}
}
```

Interfejsi

- iz sun-ovog tutorial-a

  There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts.



imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software (Java, of course) that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, make computer systems that receive GPS (Global Positioning Satellite) position data and wireless transmission of traffic conditions and use that information to drive the car.

Interfejsi

- U Javi interfejs je reference tip (kao i klasa)
    - koji sadrži metode koji su implicitno `public` i `abstract`, i podatke koji su implicitno `public`, `static` i `final` (tačnije ono što je implicitno podrazmevano ne mora se, a može navesti)
    - koji se **ne može instancirati**
    - može se implementirati na klasu
    - može biti proširen - tačnije moguće je jedan interfejs izvesti iz drugog
- nakon što se za neku podklasu definiše da je izvedena (`extends`) iz neke nadklase, mogu se opciono implemetirati (`implements`) jedan ili više interfejsa (simulacija **višestrukog nasleđivanja**)
- 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

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

class Krug extends GeometrijskiOblik implemets Crtanje {
   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
   }
}
```

Interfejsi

Interfaces as APIs

The robotic car example shows an interface being used as an industry standard **Application Programming Interface (API)**. APIs are also common in commercial software products.

Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product. An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's API is made public (to its customers), its implementation of the API is kept as a closely guarded secret—in fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

Interfejsi PRIMER

Relatable – interfejs kojim se definiše poređenje objekata iste klase.

```
    public interface Relatable {
// this and other must be instances of the same class
// returns 1, 0, -1 if this is greater than, equal to, or less than other
       public int isLargerThan(Relatable other);
    }
...
public class RectanglePlus implements Relatable {
    public int width = 0;
    public int height = 0;
    public Point origin;

    public RectanglePlus() {origin = new Point(0, 0);}
    public RectanglePlus(Point p) {origin = p;}
    public RectanglePlus(int w, int h) {
        origin = new Point(0, 0); width = w; height = h;}
    public RectanglePlus(Point p, int w, int h) {
        origin = p; width = w; height = h;}
    public void move(int x, int y) { origin.x = x; origin.y = y;}
    public int getArea() { return width * height; }
```

Interfejsi PRIMER

```
// a method to implement Relatable
    public int isLargerThan(Relatable other) {
        RectanglePlus otherRect = (RectanglePlus)other;
        if (this.getArea() < otherRect.getArea())
                return -1;
        else if (this.getArea() > otherRect.getArea())
                return 1;
        else
                return 0;
    }
}
```

You can use interface names anywhere you can use any other data type name. If you define a **reference variable whose type is an interface, any object you assign to it must be an instance of a class that implements the interface**.

Zašto je argument definisan sa `Realatible`?
Sledeći slajd

Interfejsi PRIMER

finding the largest object in a pair of objects, for any objects that are instantiated from a class that implements Relatable:

```
 public Object findLargest(Object object1, Object object2) {
            Relatable obj1 = (Relatable)object1;
            Relatable obj2 = (Relatable)object2;
            if ( (obj1).isLargerThan(obj2) > 0)
               return object1;
            else
               return object2;
         }
```

By casting `object1` to a `Relatable` type, it can invoke the `isLargerThan` method.
If you make a point of implementing `Relatable` in a wide variety of classes, the objects instantiated from any of those classes can be compared with the `findLargest()` method—provided that both objects are of the same class.

DOMAĆI: **Definisati nekoliko klasa izvedenih iz apstraktne klase GeometrijskiOblik (po modelu Pravougaonik, definisati Krug i Trougao) implementirati im Realtable i napraviti Test klasu u kojoj bi se definisalo više primeraka različitih oblika i odredilo koji je najveći.**

Interfejsi vs Apstarktne klase

Unlike interfaces, abstract classes

- can contain fields that are not static and final,
- and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation.

If an abstract class contains only abstract method declarations, it should be declared as an interface instead.

Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way.

...

'Skrivanje'

- Zašto?

  Class creators vs. **client programmer**
- zaštita podataka i metoda
- Access modifiers, also called visibility modifiers, determine the accessibility scope of the Java elements they modify. If you do not explicitly use an access modifier with a Java element, the element implicitly has the **default access** modifier. The explicit access modifiers may be used with a class and its members (that is, instance variables and methods). They cannot be used with the variables inside a method.

'Skrivanje'

vidljivost / tip	Dostupno (vidljivo)			
	klasi	podklasi	paketu	bilo kome
`private`	X			
`protected`	X	X	X	
`public`	X	X	X	X
Nothig (default) Package Friendly	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
    }
}
```

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	