

An abstract blue-toned image featuring a globe with puzzle pieces and wavy lines, suggesting a digital or network theme.

# Paketi i kontrola pristupa

Programski jezici II

# Struktura Java datoteke sa izvornim kodom

- svaka definicija klase koja se koristi u programu trebala bi se nalaziti u zasebnoj datoteci
- moguće je da se u jednoj datoteci nađe više definicija različitih klasa
- naziv datoteke treba da bude identičan nazivu klase, s tim što datoteka treba da ima ekstenziju .java

# Struktura Java datoteke sa izvornim kodom

```
package net.etfbl.paket; // 1

import java.util.*; // 2
import java.text.DecimalFormat; // 3

public class KlasaPrva { // 4
}

interface InterfejsPrvi{ // 5
}

class KlasaDruga { // 6
}

interface InterfejsDrugi{ // 7
}

enum Enumeracija { JANUAR, FEBRUAR, MART} // 8
```

# Struktura Java datoteke sa izvornim kodom

- elementi Java datoteke sa izvornim kodom:
  - deklaracija paketa – ako se klasa nalazi u odgovarajućem paketu, to mora biti specificirano korištenjem ključne riječi **package** iza koje se navodi naziv paketa – ovaj element Java datoteka sa izvornim kodom nije obavezan
  - deklaracija uvoza (**import**) – prije deklaracije tipova u izvornom kodu neophodno ih je uvesti (import-ovati) – ovaj element Java datoteka sa izvornim kodom nije obavezan, a ako postoji može postojati jedna ili više deklaracija uvoza – potrebno je razlikovati dvije vrste deklaracija uvoza (statički i nestatički)
  - deklaracija tipova – klasa, interfejsa i enumeracija – ovaj element Java datoteka sa izvornim kodom je obavezan i može postojati jedna ili više deklaracija tipova

# Struktura Java datoteke sa izvornim kodom

- bitno pravilo – klasa koja ima public vidljivost mora biti definisana u datoteci sa nazivom koji je identičan nazivu klase i ekstenzijom .java
  - ovo pravilo određuje da izvorna datoteka ne može sadržavati više od jedne public klase, tj. ako se u jednoj izvornoj datoteci nalaze definicije više klasa, onda samo jedna može biti public i izvorna datoteka će imati naziv koji odgovara nazivu public klase (sa ekstenzijom .java)
  - svaka definicija klase iz iste izvorne datoteke kompajlira se u zasebnu datoteku sa .class ekstenzijom
- bitno – sav izvorni kod je enkapsuliran u klasama ili interfejsima, osim deklaracije paketa i deklaracija uvoza.

# Paketi

- paketi su mehanizam za enkapsulaciju pomoću kojeg se grupišu klase, interfejsi, enumeracije i podpaketi, tj. paketi su mehanizam za hijerarhijsko organizovanje programa u module
- postoji više razloga za korištenje paketa – najznačajniji su:
  - jednostavno utvrđivanje da su određeni tipovi (koji se nalaze u paketu) povezani
  - lakše pronalaženje tipova koji obezbjeđuju određene funkcionalnosti
  - izbjegavanje konflikta kod naziva tipova (paket kreira novi prostor imena – eng. namespace)
  - obezbjeđivanje kontrole pristupa



# Deklaracija paketa

- korištenjem ključne riječi **package** iza koje se navodi naziv paketa, u datoteci sa izvornim kodom, Java klasa ili klase (interfejs, interfejsi, enumeracije) se smješta(ju) u odgovarajući paket
- može postojati samo jedna deklaracija paketa u datoteci sa izvornim kodom, tj. jedna klasa se ne može nalaziti u više paketa
- za klase kod kojih u datoteci sa izvornim kodom ne postoji deklaracija paketa kaže se da se nalaze u korijenskom (eng. root) ili implicitnom paketu – ovaj paket nema posebno ime

# Uvoz paketa

- korištenjem ključne riječi import iza koje se navodi naziv paketa ili klase, u datoteci sa izvornim kodom, omogućava se korištenje uvezene klase ili svih klasa iz specificiranog paketa
- korištenje tipova (klasa i interfejsa) moguće je i bez uvoza, ali se u tom slučaju navodi puno kvalifikovano ime tipa
- uvoz značajno pojednostavljuje pisanje koda i čini ga čitljivijim
- deklaracija uvoza u datoteci sa izvornim kodom se mora nalaziti iza deklaracije paketa
- postoje dva oblika deklaracije uvoza: uvoz jednog tipa i uvoz na zahtjev
  - uvoz jednog tipa omogućava korištenje datog tipa, bez navođenja punog kvalifikovanog imena
  - uvoz na zahtjev omogućava korištenje bilo kojeg tipa iz specificiranog paketa, bez navođenja punog kvalifikovanog imena



# Uvoz paketa

- deklaracijom uvoza ne uvoze se podpaketi, rekurzivno
- sve datoteke sa izvornim kodom implicitno uvoze java.lang paket
  - na primjer, moguće je koristiti klasu String bez navođenja punog kvalifikovanog imena klase, iako paket java.lang ili klasa java.lang.String nisu eksplicitno uvezene
  - Object – osnovna klasa, korijenska u hijerarhiji Java klasa
  - Class – instance ove klase predstavljaju klase u run time-u
  - Okružujuće klase – Integer, Long,...
  - StringBuffer, Math, ClassLoader, Process, Runtime, SecurityManager, System, Thread,...
- kod uvoza tipova sa istim imenom (iz različitih paketa) javlja se problem neodređenosti imena

```
import java.awt.Rectangle;  
import net.etfbl.paket.Rectangle;
```

- jedini način referenciranja određenog tipa jeste navođenjem njegovog punog kvalifikovanog imena

# Statički uvoz

- slično uvozu tipova, moguće je uvesti i statičke članove kompleksnih tipova
- ovaj mehanizam se naziva statički uvoz (statički import)
- statički uvoz omogućava korištenje uvezenih kratkih naziva statičkih članova, bez navođenja njihovog punog kvalifikovanog imena
- postoje dva oblika deklaracije statičkog uvoza:

- statički uvoz jednog člana

```
import static <puno kvalifikovano ime tipa>.<naziv statičkog člana>; // 1
```

```
import static <puno kvalifikovano ime tipa>.*; // 2
```

- statički uvoz na zahtjev

# Statički uvoz

```
import static java.lang.Math.random; // 1
import static java.lang.Math.*;      // 2

public class StatickiImport {
    public static void main(String[] args) {
        double r = random();        // 3
        double pi = PI;              // 4
    }
}
```

# Statički uvoz

- identifikatori klase mogu maskirati statičke članove koji se uvoze
  - primjer – uvezena statička metoda `random` iz klase `java.lang.Math` maskirana je metodom `random` klase `StatickiImport`

```
import static java.lang.Math.random;           // 1
import static java.lang.Math.*;

public class StatickiImport {
    public static void main(String[] args) {
        double r = random();                     // 2
        double r2 = java.lang.Math.random();    // 3
        System.out.println(r);                  // 4
    }
    public static double random() {              // 5
        return 1.2;
    }
}
```

- konflikt imena se može javiti i u slučaju kada se putem dva ili više statička uvoza uveze statička metoda koja ima isti potpis ili statički atribut istog imena – i u ovim slučajevima jedini način pristupa određenoj statičkoj metodi ili statičkom atributu jeste navođenjem njenog/njegovog punog kvalifikovanog imena

# Paketi

- organizacija – kao direktorijumi i datoteke
- prevođenje:

```
javac paket1\Test.java
```

```
javac paket2\paket3\Test2.java
```

- pokretanje:

```
java paket1.Test
```

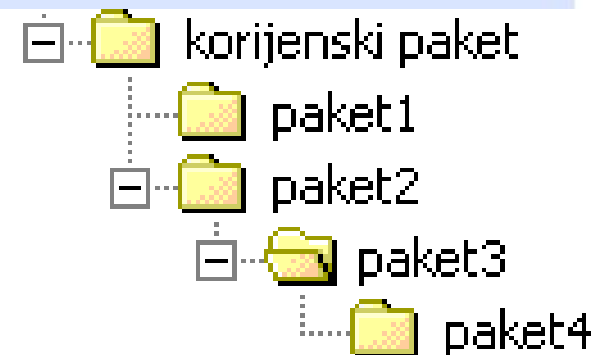
- “kod” klase Test2 mora početi sa:

```
package paket2.paket3;
```

```
class Test2{...}
```

```
package paket2.paket3.paket4;
```

```
public class Klasa234 {  
  
}
```



```
package paket1;
```

```
public class Klasa1 {  
  
}
```

# Paketi

- kreiranje paketa

```
package imePaketa;  
...  
public class MojaKlasa { ... }
```

- korišćenje paketa

```
import imePaketa.MojaKlasa;  
import imePaketa.*;
```

```
...  
MojaKlasa m = new MojaKlasa();
```

```
imePaketa.MojaKlasa m = new  
imePaketa.MojaKlasa();
```



# Paketi

- direktorijumi
  - hijerarhija paketa se poklapa sa hijerarhijom direktorijuma:

`paket1.paket2.Proba -> paket1\paket2\Proba.class`



# CLASSPATH

- CLASSPATH *environment* varijabla predstavlja spisak foldera i JAR arhiva na fajl sistemu gdje JVM traži klase i druge resursne datoteke
- JVM će klase i druge resursne datoteke tražiti u standardnim Java bibliotekama i na putanjama navedenim u CLASSPATH varijabli
- ako vrijednost CLASSPATH varijable nije podešena tekući direktorijum se smatra podrazumijevanom vrijednošću CLASSPATH varijable

```
CLASSPATH=.;D:\Java\jdk1.6.0_17\jre\lib\rt.jar;
```

- u slučaju da je vrijednost CLASSPATH varijable podešena, potrebno je eksplicitno navesti tekući direktorijum (znak „.”) kako bi JVM klase i druge resursne datoteke tražila i u tekućem direktorijumu
- kao alternativa, može se koristiti -classpath opcija (ili -cp) kao argument JDK komandnim alatima
- CLASSPATH varijabla vidljiva za sve aplikacije na datoj platformi

# Paketi

- problem davanja imena paketima – problem 2 nezavisna programera koji koriste isto ime paketa i ime klase
- konvencija davanja imena
  - reverzna imena domena kao prefiks imena paketa
  - net.etfbl.paket za paket koji je pisan na ETF-u

| Ime domena        | Prefiks imena paketa |
|-------------------|----------------------|
| etf-bl.net        | net.etf_bl           |
| student.etfbl.int | int_.etfbl.student   |
| paket.24sata.info | info._24sata.paket   |

# JAR

- JAR (*Java ARchive*) arhive su arhive sa klasičnim ZIP formatom i podrazumijevanom ekstenzijom .jar
- sve klase iz standardnih Java biblioteka, prilikom instalacije JDK paketa, smještene su u %JAVA\_HOME%\jre\lib\rt.jar, gdje je %JAVA\_HOME% putanja na kojoj se nalazi korijenski direktorijum unutar kojeg je instaliran JDK ili JRE paket
- JAR arhive se kreiraju korištenjem JAR alata koji dolazi u okviru JDK paketa
- tipična JAR arhiva sadrži .class datoteke i sve druge resurse koji su neophodni za funkcionisanje aplikacije (npr., slike i audio datoteke), kao i manifest datoteku
- Manifest datoteka se automatski kreira i uključuje u arhivu
  - može sadržavati korisne informacije, poput informacije u kojoj klasi se nalazi osnovna, main metoda

```
jar cvf arhiva.jar Klasa1.class Klasa2.class
jar cvmf arhiva.jar mf.txt Klasa1.class Klasa2.class
jar cvfe arhiva.jar Klasa1 Klasa1.class Klasa2.class
java -jar arhiva.jar
```

# Sistemske osobine

- Java izvršno okruženje ima informacije o nazivu operativnog sistema, verziji JDK-a, JRE-a, i različitim informacijama zavisnim od platforme, poput oznake za kraj linije
- ove informacije se čuvaju kao kolekcija osobina vezana za platformu na kojoj je Java izvršno okruženje instalirano
- svaka osobina je definisana parom ime-vrijednost.
  - primjer – naziv operativnog sistema čuva se u osobini sa imenom „os.name“

```
public class SistemskeOsobine {  
    public static void main(String[] args) {  
        String nizOsobina[] = {"os.name", "java.version"};  
        Properties osobine = System.getProperties();  
        for (String kljuc: nizOsobina) {  
            String vrijednost = osobine.getProperty(kljuc);  
            System.out.println(kljuc + " = " + vrijednost);  
        }  
    }  
}
```

- osobine se čuvaju u heš tabeli, a može im se pristupiti putem `java.util.Properties` klase, koja je klasa nasljednica `java.util.Hashtable` klase

# Sistemske osobine

- java.version
- java.vendor
- java.vendor.url
- java.home
- java.vm.specification.version
- java.vm.specification.vendor
- java.vm.specification.name
- java.vm.version
- java.vm.vendor
- java.vm.name
- java.specification.version
- java.specification.vendor
- java.specification.name
- java.class.version
- java.class.path
- java.library.path
- java.io.tmpdir
- java.compiler
- java.ext.dirs
- os.name
- os.arch
- os.version
- file.separator
- path.separator
- line.separator
- user.name
- user.home
- user.dir



# Pravila vidljivosti

- Java obezbeđuje eksplicitne modifikatore pristupa radi kontrole pristupa članovima klase od strane eksternih klijenata
  - klasni opseg vidljivosti
  - opseg vidljivosti bloka

# Klasni opseg vidljivosti

- klasni opseg vidljivosti određuje kako se pristupa članovima klase (uključujući i one koji su nasljeđeni) unutar same klase

```
class Roditeljska {  
    int promURod;  
    static int statickaPromURod;  
    void metodaInstanceURod() {}  
    static void statickaMetodaURod() {}  
}  
class Osnovna extends Roditeljska {  
    int prom;  
    static int statickaProm;  
    void metodaInstance() {}  
    static void statickaMetoda() {}  
}
```

# Klasni opseg vidljivosti

| Član klase               | Nestatički kontekst unutar klase<br>Osnovna  | Statički kontekst unutar klase<br>Osnovna  |
|--------------------------|--|--|
| Promjenljiva<br>instance | prom<br>this.prom<br>promURod<br>this.promURod<br>super.promURod   | Nije moguće  |
| Metoda<br>instance       | metodaInstance()<br>this.metodaInstance()<br>metodaInstanceURod()<br>this.metodaInstanceURod()<br>super.metodaInstanceURod()   | Nije moguće  |
| Statička<br>promjenljiva | statickaProm<br>this.statickaProm<br>Osn.statickaProm<br>statickaPromURod<br>this.statickaPromURod<br>super.statickaPromURod<br>Osn.statickaPromURod<br>Roditeljska.statickaPromURod                                 | statickaProm<br><br>Osn.statickaProm<br>statickaPromURod<br><br>Osn.statickaPromURod<br>Roditeljska.statickaPromURod                     |
| Statička<br>metoda       | statickaMetoda()<br>this.statickaMetoda()<br>Osn.statickaMetoda()<br>statickaMetodaURod()<br>this.statickaMetodaURod()<br>super.statickaMetodaURod()<br>Osn.statickaMetodaURod()<br>Roditeljska.statickaMetodaURod() | statickaMetoda()<br><br>Osn.statickaMetoda()<br>statickaMetodaURod()<br><br>Osn.statickaMetodaURod()<br>Roditeljska.statickaMetodaURod() |

# Klasni opseg vidljivosti

- iz statičkog konteksta može se pristupiti samo statičkim članovima
- kako se statički kod ne izvršava u kontekstu objekta, tako i reference `this` i `super` nisu dostupne iz statičkog konteksta
- iz nestatičkog konteksta su uvijek dostupni i statički članovi
  - može se primijetiti da je pristup statičkim članovima preko imena klase identičan iz same klase i eksterne klase

# Opseg vidljivosti bloka

- dijelovi programskog koda su organizovani u blokove korištenjem vitičastih zagrada „{“ i “}”
- blokovi mogu biti ugnježdeni
- deklaracija lokalne promjenljive može se pojaviti bilo gdje unutar bloka
- opseg vidljivosti definisan je vitičastim zagradama, tj. promjenljive postoje od deklarisanja do izlaska iz bloka
- lokalne promjenljive metode obuhvataju formalne parametre metode, kao i promjenljive koje su deklarisanne unutar tijela metode
  - lokalne promjenljive u tijelu metode se kreiraju iznova, pri svakom pozivu metode

# Opseg vidljivosti bloka

```
public class OpsegVidljivosti {  
    private int globalnaPromjenljiva;    // vidljiva do 3  
  
    public void metoda(int parametar) {  
        int promjenljiva1;                // vidljiva do 2  
        if (globalnaPromjenljiva > 0) {  
            int promjenljiva3;            // vidljiva do 1  
        }                                // 1  
        int promjenljiva2;                // vidljiva do 2  
    }                                    // 2  
}                                        // 3
```

- bitno je napomenuti da objekti nemaju isti opseg vidljivosti kao primitivni tipovi i reference
  - objekat će postojati na heap-u sve dok ne bude uklonjen od strane garbage collector-a



# Modifikatori pristupa za deklaraciju top-level tipova

- modifikator pristupa public može se koristiti za deklaraciju top-level tipova (klasa, interfejsa i enumeracija) u paketima, kako bi one mogle biti dostupne kako iz svog paketa, tako i iz drugih

| Modifikator pristupa | Top-level tip                |
|----------------------|------------------------------|
| Public               | Dostupan iz svih paketa      |
| Podrazumijevani      | Dostupan samo iz svog paketa |

- ako se ne koristi modifikator pristupa public deklarirani tip će biti dostupan samo unutar svog paketa – paketski ili podrazumijevani pristup

# Modifikatori pristupa za deklaraciju top-level tipova

```
package net.etfbl.paket;

public class JavnaKlasa {
    public static void ispis() {
        PaketskaKlasa pk = new PaketskaKlasa();
        System.out.println("Metoda ispis iz objekta
klase JavnaKlasa...");
        pk.ispis();
    }

    public static void main(String[] args) {
        ispis();
    }
}

class PaketskaKlasa{
    void ispis() {
        System.out.println("Metoda ispis iz objekta
klase PaketskaKlasa...");
    }
}
```

# Ostali modifikatori za top-level i ugnježdene tipove

- modifikatori: abstract i final
- modifikator abstract se koristi u deklaraciji klase kako bi označio da klasa ne može biti instancirana, tj. kako bi označio da je klasa apstraktna

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

- u opštem slučaju, apstraktne klase su klase koje predstavljaju opštu apstrakciju, čije instanciranje ne bi imalo praktičnu primjenu, dok bi njena specijalizacija mogla imati praktičnu primjenu

# Ostali modifikatori za top-level i ugnježdene tipove

- svaka klasa može biti deklarirana kao apstraktna klasa, ali klase koje imaju jednu ili više apstraktnih metoda (metode deklarirane ključnom riječju `abstract` i koje nemaju tijelo) moraju biti deklarirane kao apstraktne klase
- kako nisu u potpunosti implementirane, ne mogu biti ni instancirane
- ovakve klase se mogu koristiti kako bi specificirale određeno ponašanje, ali se klasama nasljednicama ostavlja da obezbijede odgovarajuću implementaciju
- klase nasljednice, da bi mogle biti instancirane, moraju obezbijediti implementaciju svih apstraktnih metoda nasljeđenih iz apstraktne klase – u suprotnom, i klase nasljednice moraju biti deklarirane kao apstraktne
- interfejsi specificiraju samo apstraktne metode, bez implementacije bilo koje metode – interfejsi su po svojoj prirodi implicitno apstraktni i ne mogu biti instancirani
- interfejs je moguće deklarirati kao apstraktan, korištenjem ključne riječi `abstract`, ali je to nepotrebno i predstavlja redundansu, dok enum tipovi ne mogu biti deklarirani kao apstraktni

# Ostali modifikatori za top-level i ugnježdene tipove

- modifikator final se koristi u deklaraciji klase kako bi označio da klasa ne može biti nasljeđena, tj. kako bi označio da ne mogu postojati klase nasljednice date klase
- iz ovog razloga metode deklarisanе u ovakvoj klasi ne mogu biti redefinisane, tj. ponašanje klase ne može biti promijenjeno njenim nasljeđivanjem
- samo klase koje su u potpunosti definisane, tj. kod kojih su sve metode implementirane, mogu biti deklarisanе kao final klase - prema tome, klasa ne može u isto vrijeme biti deklarisanа i kao abstract i kao final
- klase deklarisanе kao final i interfejsi se mogu posmatrati kao potpune suprotnosti u pogledu implementacije
  - sve metode interfejsa moraju biti implementirane od strane klase koja implementira dati interfejs, dok metode u klasi deklarisanоj kao final ne mogu biti redefinisane
  - apstraktna klasa predstavlja kompromis između interfejsa i klase deklarisanih kao final klase

# Modifikatori pristupa članova

- navođenjem modifikatora pristupa članova, klasa kontroliše koji od njih su dostupni drugim klasama (klijentima)
- ovi modifikatori omogućavaju klasi da definiše ugovor putem kojeg klijenti znaju koje servise klasa pruža
- modifikatori pristupa članova su: public, protected, private i nespecificirani (prijateljski, friendly) pristup
- modifikatori pristupa članova klase imaju značenje samo ako je klasa ili klasa nasljednica dostupna klijentima (drugim klasama)
  - ako klasa nije deklarirana kao public klasa, onda public modifikator pristupa članova date klase nema nikakvo značenje za klase iz drugih paketa
- jedan član klase može imati samo jedan modifikator pristupa



# Modifikatori pristupa članova

- modifikator `public` – označava da je član klase vidljiv za sve klase u bilo kojem paketu programa – ovo važi i za članove instance i za statičke članove
- modifikator `protected` – označava da je član klase vidljiv samo za klase u istom paketu i za sve klase nasljednice u bilo kojem paketu – sve ostale klase ne mogu pristupiti `protected` članovima
- modifikator `private` – označava da je član klase vidljiv samo unutar svoje klase – ovo važi i za klase nasljednice, bez obzira da li se one nalaze u istom ili drugom paketu
  - preporuka je da se pomoćne metode deklarišu kao privatne
- paketski ili prijateljski (`friendly`) pristup – ako modifikator pristupa člana nije specificiran – podrazumijevani modifikator označava da je član klase vidljiv samo od strane klasa u istom paketu
  - podrazumijevani modifikator je restriktivniji od `protected` modifikatora

# Ostali modifikatori članova

- ostali modifikatori članova klase:
  - static
  - final
  - abstract
  - synchronized
  - native
  - transient
  - volatile

# Ostali modifikatori članova

- modifikator static – statički članovi pripadaju klasi u kojoj su deklarirani i nisu dio niti jedne instance klase – deklaracija statičkih članova vrši se pomoću modifikatora static
- modifikator final – promjenljiva u čijoj deklaraciji se nalazi modifikator final je konstanta
  - njena vrijednost ne može biti promijenjena nakon inicijalizacije, tj. vrijednost promjenljive primitivnog tipa ne može biti promijenjena, kao ni vrijednost reference (final reference uvijek referenciraju isti objekat)
  - ključna riječ final ne može uticati na to da stanje objekta koji referencira final referenca ne bude izmijenjeno

# Ostali modifikatori članova

- promjenljive koje su static i final se najčešće koriste za definisanje konstanti
- promjenljive definisane unutar interfejsa su implicitno final promjenljive
- final promjenljiva ne mora biti inicijalizovana pri deklaraciji, ali mora biti inicijalizovana prije nego što bude korištena – onda inicijalizacija mora biti izvršena u svakom konstruktoru (ako ih je više) ili u inicijalizacionom bloku
- final promjenljive obezbjeđuju da vrijednost ne može biti promijenjena, dok final metode obezbjeđuju da ponašanje ne može biti promijenjeno
- Vrijednost final promjenljive ne mora biti poznata u vrijeme kompajliranja programa

# Modifikator abstract

- modifikator abstract može biti korišten pri deklaraciji metoda, ali ne i pri deklaraciji promjenljivih
- metode koje u svojoj deklaraciji imaju ključnu riječ abstract nazivaju se apstraktnim metodama
- apstraktna metoda nema implementaciju, tj. tijelo metode nije definisano
- modifikator pristupa apstraktne metode ne može biti private, jer u tom slučaju ne bi bilo moguće da klasa koja nasljeđuje apstraktnu klasu redefiniše apstraktnu metodu i obezbijedi njenu implementaciju
- samo metode instance mogu biti deklarirane kao apstraktne – kako statička metoda ne može biti redefinisana, nije dozvoljeno ni njeno deklarisanje kao apstraktne metode
- final metoda ne može biti apstraktna, jer ne može biti nekompletna
- metode specificirane u interfejsu su implicitno apstraktne metode, tako da modifikator abstract nije potrebno ni navoditi u deklaraciji metode interfejsa



# Modifikator synchronized

- modifikator synchronized može biti korišten pri deklaraciji metoda, ne i promjenljivih
- metode koje u svojoj deklaraciji imaju ključnu riječ synchronized nazivaju se sinhronizovanim metodama



# Modifikator native

- native metode su metode čija implementacija nije definisana u Java programskom jeziku, već u nekom drugom programskom jeziku (npr., C, C++, ...)
- ovakva metoda se može deklarirati kao član Java klase, i to pomoću ključne riječi native

```
public class NativeMetode {  
    static {  
        System.loadLibrary("NativeBiblioteka");  
    }  
    public native void metoda() throws Exception;  
}
```

- Iako native metoda može specificirati i izuzetke u throws klauzuli, kompajler ih ne može provjeriti, jer metoda nije implementirana u Javi



# Modifikator transient

- modifikator transient može biti korišten pri deklaraciji promjenljivih, ne i metoda
- promjenljive koje u svojoj deklaraciji imaju ključnu riječ transient nazivaju se tranzientnim promjenljivim
- ove promjenljive se koriste kada je potrebno sačuvati stanje objekta koristeći serijalizaciju

# Modifikator volatile

- modifikator volatile može biti korišten pri deklaraciji promjenljivih, ne i metoda
- ovaj modifikator se koristi kada je potrebno upozoriti kompajler da ne vrši optimizacije nad poljem koje je deklarirano kao volatile, a koje bi mogle dovesti do nepredvidivih rezultata kada se polju pristupa od strane više niti