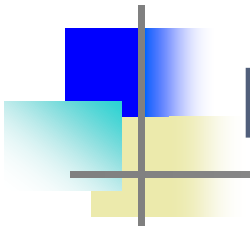
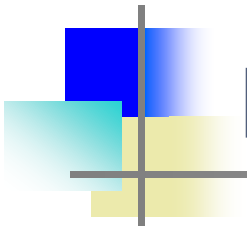


Rámcové zadanie

Produkty: tovary či služby



Produkty: tovary či služby

Výroba produktov je značne zložitá činnosť, ktorá vyžaduje veľa plánovania a riadenia. To je prakticky nepredstaviteľné bez zodpovedajúcej softvérovej podpory. Plánovanie a riadenie výrobných procesov je špecifický druh plánovania podnikových zdrojov. Zahŕňa smerovanie (materiálov), rozvrhovanie, dispečing a následné zhodnotenie. Mnohí výrobcovia sa usilujú byť úsporní (lean manufacturing), čo znamená nevyrábať to, čo nie je potrebné a vyrábať práve včas. Priame mapovanie entít reálneho sveta do objektov v modeli podporovanom počítačom aplikujte vo svojom programe - v jazyku Java.

Vypracujte konkretizáciu rámcovej témy do podoby zámeru projektu súvisaceho s produktami a to tovarmi, alebo službami.

Prednáška 3: Atribúty - deklarácia, typy, menné konvencie, použitie a modifikátory prístupu. Odkazy na objekty, referencovanie, priraďovanie objektových premenných, rekurzia, zreťazenie, agregácia

Ján Lang

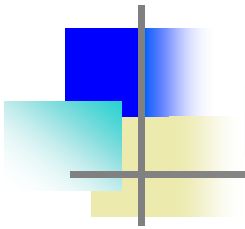
kanc. 4.34, lang@fiit.stuba.sk, http://www2.fiit.stuba.sk/~lang/zoop/

Ústav informatiky, informačných systémov a softvérového inžinierstva

Fakulta informatiky a informačných technológií

Slovenská technická univerzita v Bratislave

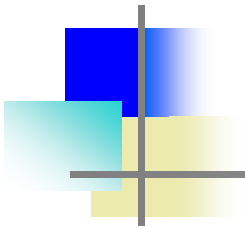
03. október 2023



public class Student

...courseAdministrationSystem_4

- private int ID;
- private double energia;
- private String firstName;
- private String middleName;
- private String lastName;
- private int birthDay, birthMonth, birthYear;
- private Poloha p;



Konštruktor

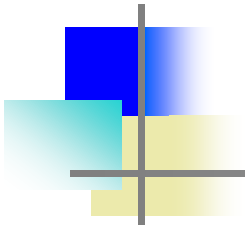
- Je operácia/metóda s menom zhodným s menom triedy, bez výstupného typu bez návratového typu
- Je operácia na inicializáciu objektu
- Je operácia volaná v okamihu vytvárania objektu
- Default je implicitný konštruktor

```
Integer i = new Integer();
```

- Objekt triedy vytvoríme tak, že zavoláme konštruktor (resp. niektorý z konštruktorov) triedy pomocou kľúčového slova `new`

Konštruktor (implicitný vs. explicitný)

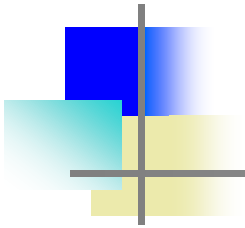
```
public class Student {

        Student() {
          //TODO: implementation part
          }

}
```



Konštruktor

Mám preto toto:

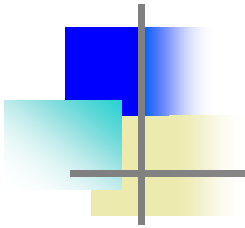
```
public class MojProgram {

public static void main(String[] args) {
        Student jano = new Student();
```



Konštruktor

Chcem aj toto:

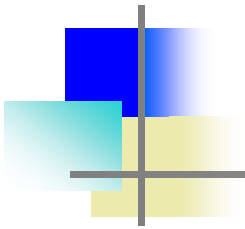
```
public class MojProgram {

public static void main(String[] args) {
        Student jano = new Student("Jano");
```



Konštruktor

Chcem ešte aj toto:

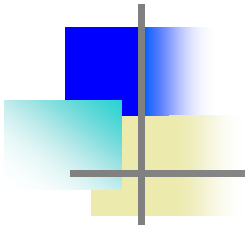
```
public class MojProgram {

public static void main(String[] args) {
        Student jano = new Student("Jano");
        Student peter1 = new Student("Peter", 100);
```



Konštruktor

??? Prečo nefunguje toto:

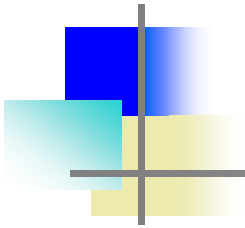
```
public class MojProgram {

public static void main(String[] args) {
        Student fero = new Student();
```



Konštruktor

??? Bude fungovať toto:

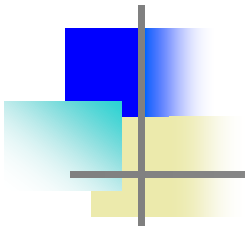
```
public class MojProgram {

public static void main(String[] args) {
        Student dano = new Student(100, 100);
```



Konštruktor

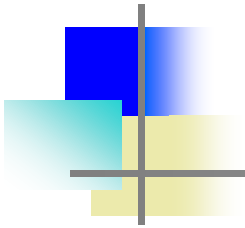
??? Môžem chcieť toto:

```
public class MojProgram {

public static void main(String[] args) {
        Student peter2 = new Student(fero);
```



Konštruktor s parametrami

- Je možné definovať konštruktor s parametrami
- Pre inicializáciu atribútov vznikajúceho objektu je možné použiť kľúčové slovo `this`
- Viac konštruktorov **v jednej triede = preťaženie konštruktorov**



...ku konštruktorom

- Kľúčové slovo `this` má ešte jedno špeciálne použitie v preťažených konštruktoroch.
- Pomocou neho, v zátvorkách s parametrami, môže konštruktor vyvolať iný konštruktor rovnakej triedy – ten, ktorého parametre vyhovujú čo do počtu aj poradia typov. Volanie iného konštruktora pomocou `this` musí byť prvý príkaz vo volajúcom konštruktore

...ku konštruktorom

```
class Clovek {
  String meno;
  String priezvisko;

  Clovek(String meno, String priezvisko) {
    this.meno = meno;
    this.priezvisko = priezvisko;
  }

  Clovek(Clovek c) {
    this.meno = c.meno;
    this.priezvisko = c.priezvisko;
  }

  Clovek() {
    meno = "Niekto";
    priezvisko = "Hocijaky";
  }

  void vypis() {
    System.out.println("Meno a priezvisko: " + meno + " " + priezvisko);
  }

  public static void main(String[] args) {
    Clovek prvy = new Clovek("Adam","Prvy");
    Clovek druhy = new Clovek(prvy);
    Clovek treti = new Clovek();
    prvy.vypis();
    druhy.vypis();
    treti.vypis();
  }
}
```

Privátny konštruktor

Prostredie s jedným vláknom (thread), niťou:

- Inštancia triedy vytvorená v čase načítania triedy
- Inicializácia statickým blokom
- „Lenivá" inicializácia. Inštancia triedy vytvorená v rámci globálnej metódy

Riešenia pre viacvláknové implementácie:

- Synchronizovaná globálna metóda
- Dvojnásobná kontrola uzamknutia
- Bill Pugh riešenie s použitím statickej vnútornej triedy

- Návrhový vzor: Singleton

Privátny konštruktor

- Inštancia triedy vytvorená v čase načítania triedy

- **package sk.stuba.fiit.privateCon;**
- public class A {
  - private static A instance = new A();

  - private A() {
  - }

  - public static A getInstance() {
  - return instance;
  - }
- }

Privátny konštruktor

- **package sk.stuba.fiit.privateCon;**
- public class A {
  - private static **volatile** A instance = new A();
  - private A() {
  - }
  - public static A getInstance() {
  - return instance;
  - }
- }

- V zásade sa volatile používa na označenie toho, že hodnota premennej bude modifikovaná rôznymi vláknami.

Privátny konštruktor

```
package sk.stuba.fiit.privateCon;

public class Hlavna {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //new A();
        System.out.println(A.getInstace());
        System.out.println(A.getInstace());
    }

}
```

- V zásade sa volatile používa na označenie toho, že hodnota premennej bude modifikovaná rôznymi vláknami.

Privátny konštruktor

```
package sk.stuba.fiit.lazySingleton;
public class B {
    private static B instance;

    private B() {
    }

    public static B getInstance() {
        if(instance == null) {
            instance = new B();
        }
    return instance;
    }
}
```

„Lenivá" inicializácia. Inštancia triedy vytvorená v rámci globálnej metódy

Privátny konštruktor

```
1  package sk.stuba.fiit.lazySingleton;
2
3  public class Main {
4      public static void main(String[] args) {
5          // TODO Auto-generated method stub
6          B s1 = B.getInstance();
7          System.out.println(s1);
8          //A s2 = new A();
9          //System.out.println(s2);
10     }
11 }
```

```
    return instance;
  }
}
```

Privátny konštruktor - odporúčané

```
public class ThreadSafeSingleton {
    private static ThreadSafeSingleton instance;
    private ThreadSafeSingleton(){}

    public static synchronized ThreadSafeSingleton
    getInstance(){
        if(instance == null){
            instance = new ThreadSafeSingleton();
        }
    return instance;
    }
}
```

- Synchronizovaná globálna metóda

Preťažený konštruktor

- Trieda môže mať viacero preťažených konštruktorov
- Objekt nevytvoríme iba deklaráciou premennej typu

```
ComplexNumber c1;
```

- Tým vytvoríme len premennú typu
- Objekt triedy pomocou konštruktora vytvoríme konštrukciou kde použijeme kľúčové slovo `new`

```
c1 = new ComplexNumber();
```

- Je možná aj kombinácia deklarácie a inicializácie

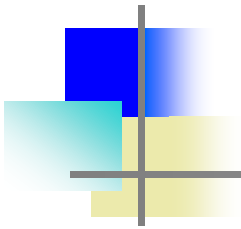
```
ComplexNumber c1 = new ComplexNumber();
```

- Výsledkom volania konštruktora je fyzická inštancia triedy charakteristická svojim **stavom, povahou a identitou**

Trieda - VARIABLE DEFINITION

- Uchovávanie dát reprezentované premennými
  - Premenné inštancií - budú uchovávať charakteristiky jednotlivých inštancií/objektov
  - Premenné tried - premenné, ktoré **nie sú** viazané na objekt ale triedu
  - Finálne premenné - po prvotnej inicializácii nemôžu byť zmenené

- Platnosť premenných
  - Lokálne premenné - deklarované v metóde či v bloku
  - Parametre - argumenty metód, premenné v slučkách
  - Premenné inštancií - deklarované pri definícii triedy

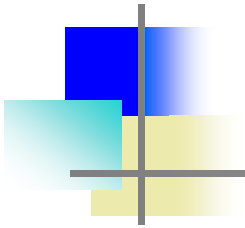


Premenné inštancií

- Definované v triedach napr.

```
int id;
private double energia;
String firstName;
String middleName;
String lasttName;
int birthYear, birthMonth, birthDay;
```

- Inak tiež nazývané **členské premenné** príp. **polia triedy**.



Premenné tried

- Sú statické premenné
- Premenná zdieľaná medzi všetkými inštanciami tej triedy
- Nepotrebujeme vytvárať objekt aby sme ich mohli inicializovať

Premenné tried

Chcem toto:

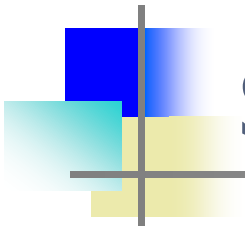
public class Student {

- private int ID;
- private double energia;
- private String firstName;
- private String middleName;
- private String lastName;
- private int birthDay, birthMonth, birthYear;
- private Poloha p;

- **static int *pocitadlo;***

Premenné tried

```
public class Student {

        Student() {
          pocitadlo++;
          this.pocitadlo++;
           }

}
```



Statické vs. nestatické vlastnosti

Procedurálna paradigma

- Tým, že je premenná typu statická existuje jej jediná inštancia
- Definované ako statické aj procedúry resp. funkcie
- Volané menom príp. s uvedením parametrov

Objektovo-orientovaná paradigma

- Počet inštancií zodpovedá počtu inštancií tried
- Nestatická inštancia vlastnosti triedy má lokálny charakter v rámci inštancie triedy - objektu

Statické metódy

- Doteraz používaná metóda

```
public static void main(String[] args) { ...telo... }
```

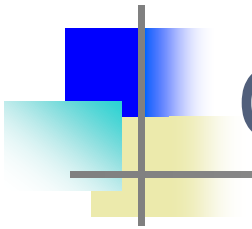
- syntax deklarácie statickej metódy

```
[public] static typ meno (argumenty) {...telo... }
```

- Sú **k dispozícii okamžite** ako náhle sprístupníme menný priestor v ktorom sa vyskytuje. Existujú aj bez toho, aby sme vytvorili objekt danej triedy
- **referencujú sa menom**, alebo menom triedy.meno_metódy či pomenovaním konštanty
- Statické metódy vidia len statické premenné a môžu volať len statické metódy (bez vytvorenia objektu)

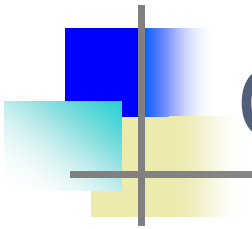
Statické metódy dostupné v Java API

```
package cvicenie03;

import java.lang.Math;

public class Hlavny {

    public static void main(String [] args) {
        System.out.println(
        Math.pow(((55.4 - 20) + 4 * 5.1 - (44 - 12)),
        3));
    }
}
```



Objekt???

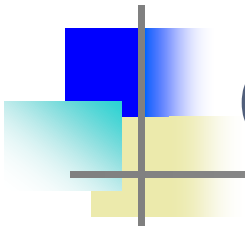
Kde však máme objekt?



Objekt???

Kde však máme objekt?

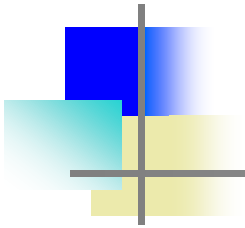
```
public static void main(String [] args) {
    System.out.println(
    Math.pow(((55.4 - 20) + 4 * 5.1 - (44 - 12)),
    3));
}
```



Objekt???

Kde však máme objekt?

- `public static void main(String [] args) {`
  - `System.out.println(`
  - **`Math.pow`**`(((55.4 - 20) + 4 * 5.1 - (44 - 12)), 3));`
- `}`
  - ...dostupné priamo v mennom priestore



Finálne premenné

- Sú premenné, ktoré môžu byť inicializované iba raz
- Hneď pri definícii, resp. neskôr v metóde

Finálne premenné

Chcem pridať toto:

public class Student {

- ...
- private final int MINIMUM = 1; // nestatická konštanta
- private static int *pocetKreditov = 0; //* statická premenná
- protected final static int *MAXIMUM* = 100; // statická konštanta

Finálne, statické premenné

??? Môžem chcieť toto:

```
public class Student {
    ...
    int pocetKreditov() {
        return MAXIMUM;
    }
```

Finálne, statické premenné

```
public class Student {
    ...
    static int pocetKreditov() {
        return MAXIMUM;
    }
```

Statické vs. triedne

```
public class PremenneAMetody {
        static int pocetLudi = 0;  //statická premenná
        final static int MAXIMUM = 100;  //statická konštanta
        int ID_instacie;  //nestaticka premenná
        final int MINIMUM = 1;  //konštanta

        public PremenneAMetody() { //konštruktor
          ID_instacie = ++pocetLudi;
        }
        static int volne() {  //statická metóda
          return MAXIMUM-pocetLudi;
        }
        int getID() { //nestatická metóda
          return ID_instacie;
        }
    }
```

Statické vs. triedne

```
... statický kontext metódy
public static void main(String [] args) {
```

- **`MAXIMUM`** // referencia statickej premennej
- `PremenneAMetody.MAXIMUM` //úplná referencia
- `PremenneAMetody.`*`volne`* //referencia statickej metódy

Čo nebude možné:

- `PremenneAMetody.MINIMUM` // nestatická konštanta v statickom kontexte
- `ID_instacie` //nestatická premenná v statickom kontexte
- `getID()` //nestatická metóda v statickom kontexte

Statické vs. triedne

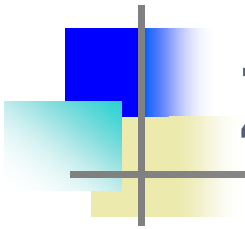
```
premenneAMetody X = new premenneAMetody();
```

objekt triedy `premenneAMetody`

- X.ID_instacie //nestatická premenná v nestatickom kontexte
- X. MINIMUM //nestatická konštanta v nestatickom kontexte
- X.getID() //nestatická metóda v nestatickom kontexte

Ide aj:

- ... //statická konštanta v nestatickom kontexte
- ... //statická premenná v nestatickom kontexte
- ... //statická metóda v nestatickom kontexte



Základné OOP pojmy

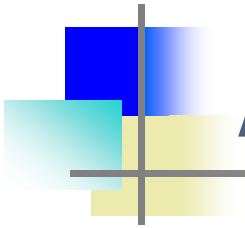
- Trieda – definícia abstraktného typu dát
- Objekt – inštancia triedy – implementuje stav entity, poskytuje navonok funkcionalitu prostredníctvom metód a ich implementácií, istá forma rozhrania
- Zapuzdrenie, obalenie (encapsulácia)
- Preťažovanie
- Prekonávanie
- Dedičnosť
- Polymorfizmus

K zapuzdreniu

```
public int getID() { // nestatická metóda
    return ID;
}

public double getEnergia() {
    return energia;
}

public void setEnergia(double energia) {
    this.energia = energia;
}

public void setbirthDay(int i) {
    this.birthDay = i;
}
```

K zapuzdreniu

```
void najedzSa() {
    this.energia++;
}

void vypis() {
    System.out.println("Volam sa: " + this.firstName);
}

public String toString() {
    return "Volam sa: " + this.firstName;
}
```



Agregácia

- Objekt jednej triedy obsahuje objekt inej triedy
- V Jave referenciou
- Realizuje sa prostredníctvom atribútov tried

konštruktor má meno zhodné s pomenovaním triedy

Príklad

- Naprogramujte triedu na reprezentáciu komplexného čísla

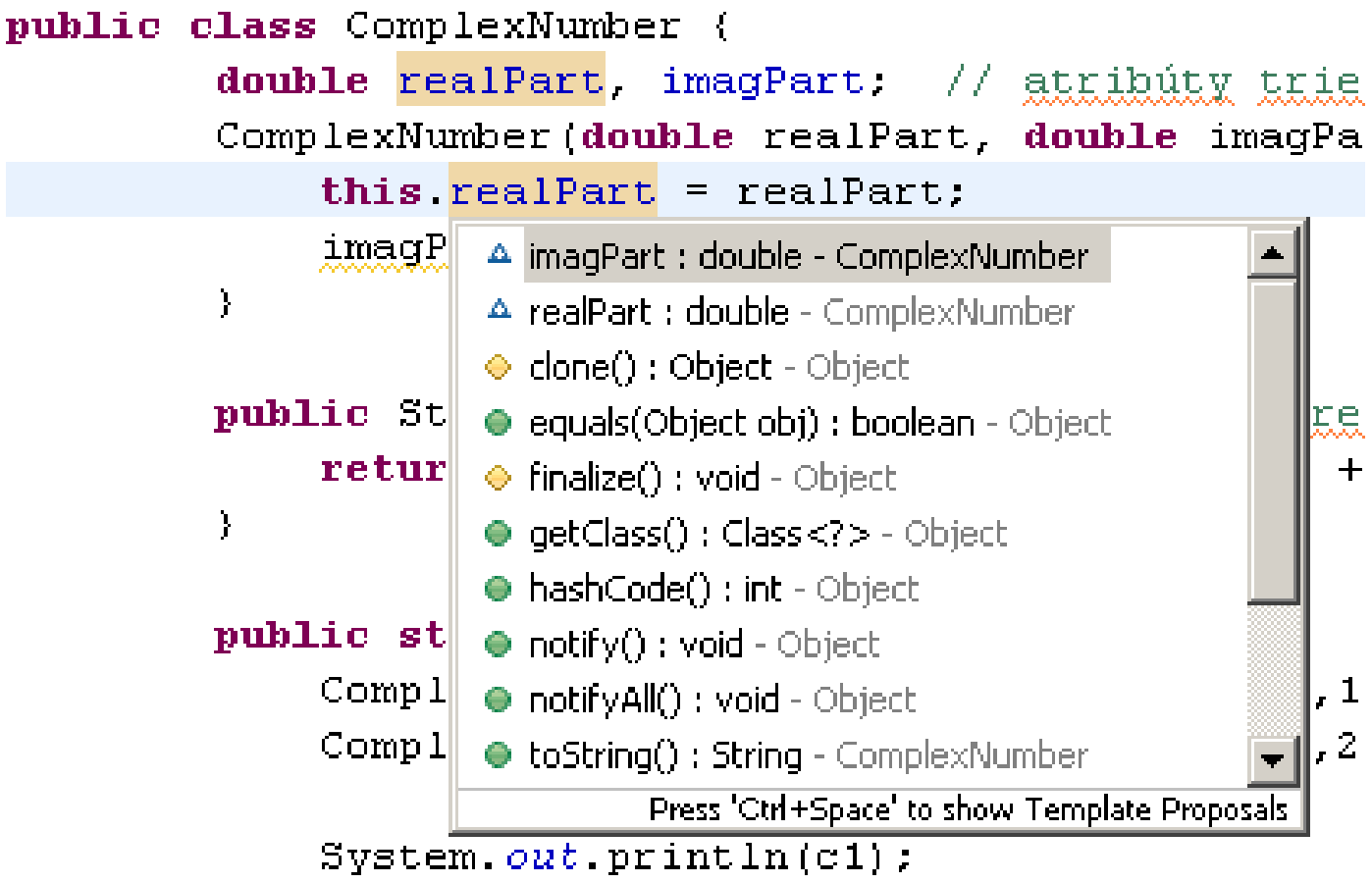
```
public class ComplexNumber {
    double realPart, imagPart; // atribúty triedy
    ComplexNumber(double cReal, double cIimag) {
        realPart = cReal;
        imagPart = cIimag;
    }

    public static void main(String[] args) {
        ComplexNumber c1 = new ComplexNumber(0,1);
        ComplexNumber c2 = new ComplexNumber(1,2);

        System.out.println(c1);
        System.out.println(c2);
    }
}
```

Kľúčové slovo `this`

- `this` je referencia na aktuálny objekt v rámci definície triedy
- V rámci triedy sa môžeme konštrukciou `this.` dostať k atribútom danej triedy
- Vhodné napr. pri inicializácii vznikajúcej inštancie, v konštruktore

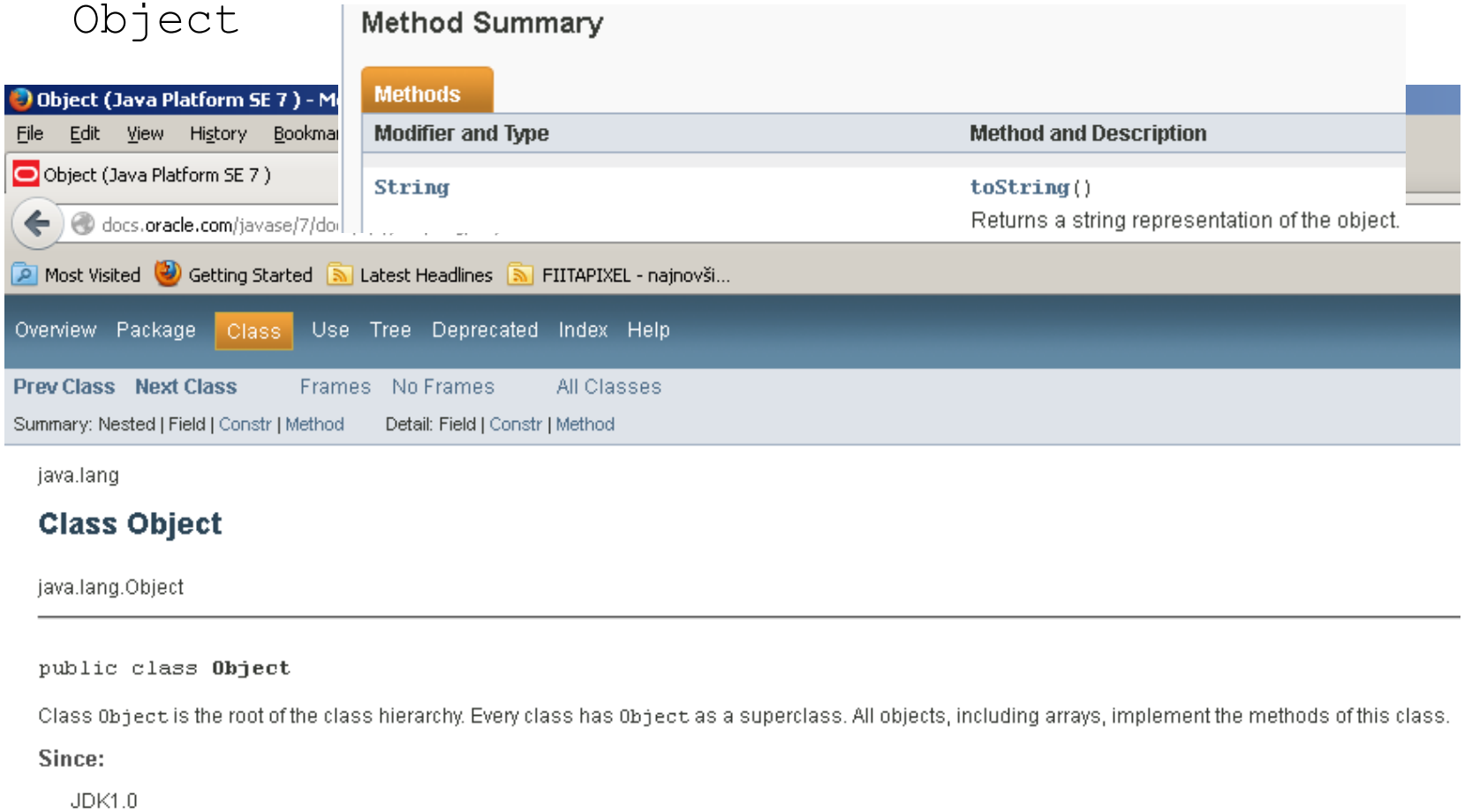
```
public class ComplexNumber {
    double realPart, imagPart;
    ComplexNumber(double realPart, double imagPart) {
        this.realPart = realPart;
        this.imagPart = imagPart;
    }
}
```

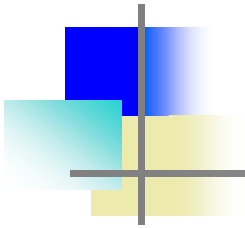
Eclipse IDE


public class ComplexNumber {
double realPart, imagPart; // atribúty trie
ComplexNumber(double realPart, double imagPa
this.realPart = realPart;
imagP
}
public St
retur
}
public st
Compl
Compl
System.out.println(c1);
imagPart : double - ComplexNumber
realPart : double - ComplexNumber
clone() : Object - Object
equals(Object obj) : boolean - Object
finalize() : void - Object
getClass() : Class<?> - Object
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
toString() : String - ComplexNumber
Press 'Ctrl+Space' to show Template Proposals

Príklad

- Nie príliš čitateľný výstup predchádzajúceho príkladu. Výsledok volania metódy nadtypu. Volania metódy `toString` triedy `Object`





Príklad

- Riešenie v podobe prekonania metódy nadtypu:

```
public String toString() {
  return "[" + realPart + "+" + imagPart + "*i]";
}
```

Metóda `toString()`

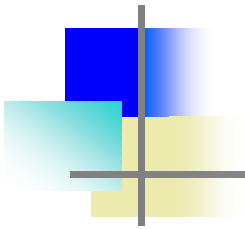
System.*out.*println(jano);

Java, API PrintStream

- public void println(Object x) Prints an Object and then terminate the line. This method calls at first String.valueOf(x) to get the printed object's string value, then behaves as though it invokes print(String) and then println().

Java, API String:

- valueOf(Object obj) Returns the string representation of the Object argument. If the argument is null, then a string equal to "null"; otherwise, the value of **obj.toString()** is returned.



Metóda `toString()`

- The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

- getClass().getName() + '@' + Integer.toHexString(hashCode())

Metóda `toString()`

- System.*out.*println(jano);

  ...System.*out vracia* PrintStream
- PrintStream.println(jano);
- PrintStream.println(Object obj);

  ... his method calls at first String.valueOf(obj) to get the printed object's string value
- String String.valueOf(obj);

  ... if the argument is null, then a string equal to "null"; otherwise, the value of obj.toString() is returned.
- obj.toString()
- Object.toString()

- getClass().getName() + '@' + Integer.toHexString(hashCode())

  ...a string representation of the object.

Metóda `toString()`

- c06:SprinklerSystem.java TIJ

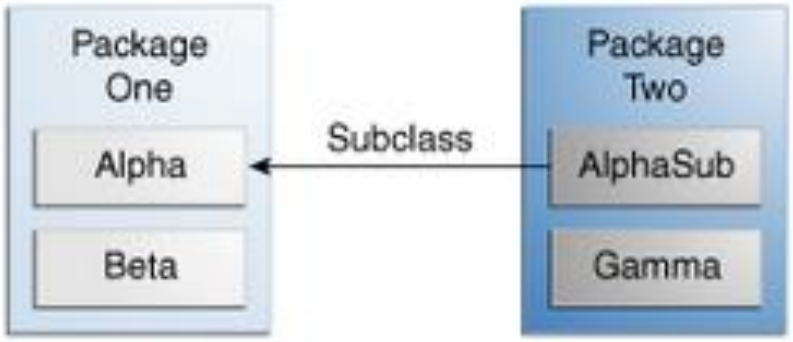
```
package sk.stuba.fiit.c06SpriklerSystemTIJ;

public class WaterSource {
    private String s;

    WaterSource() {
        System.out.println("WaterSource()");
        s = new String("Constructed");
    }

    public String toString() {
      return s;
    }
}
```

Metóda `toString()`

```
public class SprinklerSystem {
    private String valve1, valve2, valve3, valve4;
    private WaterSource source;
    private int i;
    private float f;
    public String toString() {
    return "valve1 = " + valve1 + "\n" + "valve2 = " +
      valve2 + "\n"
    + "valve3 = " + valve3 + "\n" + "valve4 = " +
      valve4 + "\n"
    + "i = " + i + "\n" + "f = " + f + "\n" + "source =
      " + source;
    }
    public static void main(String[] args) {
        SprinklerSystem sprinklers = new
          SprinklerSystem();
        System.out.println(sprinklers);
    }
}
```

Modifikátory prístupu

- bez uvedenia modifikátora, public, protected a private

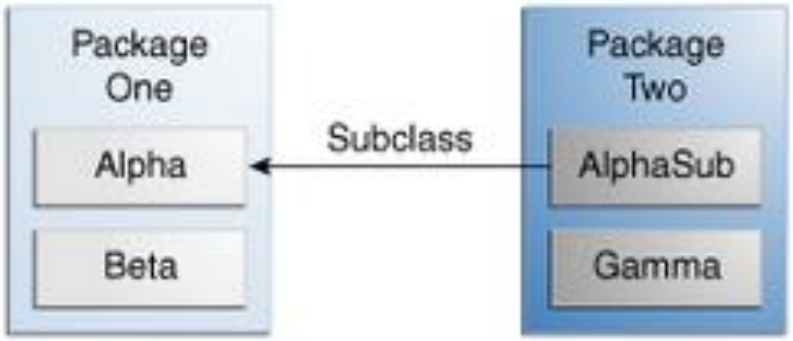


Modifier	Alpha	Beta	Alphasub	Gamma
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

- Atribúty triedy Alpha sú v rámci triedy Alpha viditeľné s uvedením ľubovoľného modifikátora prístupu či bez neho
- Atribúty triedy Alpha sú v rámci triedy Beta viditeľné s uvedením ľubovoľného modifikátora prístupu či bez neho s výnimkou privátnych atribútov
- Atribúty triedy Alpha sú v rámci triedy Alphasub viditeľné s uvedením modifikátora `public` a `protected`

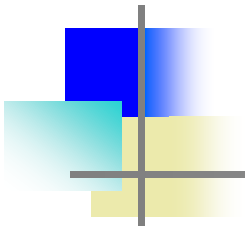
Modifikátory prístupu

- bez uvedenia modifikátora, public, protected a private



Modifier	Alpha	Beta	Alphasub	Gamma
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- Atribúty triedy Alpha ... Čo je videteľné v triede Gamma?



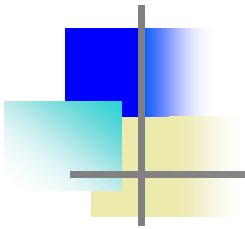
Balíčky

- Balíčky, Package
- Menný priestor, Namespace - slúži k tomu aby logicky združoval typy, ktoré spolu súvisia. Dôvod zložitosť programov, ich rozsah, statické vlastnosti - globálne vlastnosti
- Obyčajne je v balíčku viac tried

Súbor Classs.java

```
package sk.stuba.fiit;
public class Classs {
   ...
}
```

- Súbor Classs.java patrí do balíčka sk.stuba.fiit



Balíčky

- Aplikáciu balíčka v inom súbore si vyžaduje jeho sprístupnenie kľúčovým slovom `import`

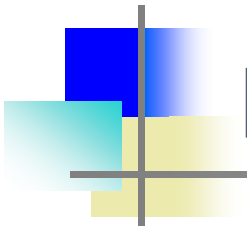
```
import sk.stuba.fiit.*;
```

- Znak `*` znamená že takto uvedená konštrukcia sprístupní menný priestor celého balíčka. Mám prístup ku všetkým triedam, ktoré obsahuje
- V prípade, že z balíčka vyžadujem sprístupniť len konkrétny typ musím to v deklarácii uviesť

```
import sk.stuba.fiit.Classs;
```

- Po tomto budem môcť pracovať len s triedou `Classs` z balíčka `sk.stuba.fiit`
- Importovanie len statických metód/konštánt napr.

```
import static java.lang.Math;
```



Balíčky

- Pracovať možno aj bez „importovania“ balíčka
- Ak nesprístupním požadovaný menný priestor budem musieť napr. pre vytvorenie inštancie typu pristupovať s uvedením plnohodnotnej cesty

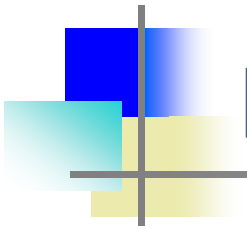
```
sk.stuba.fiit.Classs c = new sk.stuba.fiit.Classs();
```

Balíčky

- Pracovať možno aj bez „importovania“ balíčka
- Ak nesprístupním požadovaný menný priestor budem musieť napr. pre vytvorenie inštancie typu pristupovať s uvedením plnohodnotnej cesty

```
sk.stuba.fiit.Classs c = new sk.stuba.fiit.Classs();
```

```
package courseAdministrationSystem_4;

//import sk.stuba.fiit.privateCon.A;

public class Main {
    public static void main(String[] args) {

        System.out.println(sk.stuba.fiit.privateCon.A.getInstace());
        //System.out.println(A.getInstace());
        //System.out.println(A.getInstace());
        //System.out.println(A.getInstace());

    }
}
```



Balíčky

- Na balíčky sa možno zjednodušene pozerať ako na knižnice tried. Java ako programovací jazyk nezávislý na platforme, má možnosť vytvárania knižníc už zakomponovanú do jazyka.
- Každý balík je obvykle reprezentovaný adresárom, ktorý obsahuje preložené zdrojové súbory (.class) tried balíčka
- Vnorovaním adresárov vzniká hierarchia balíčkov. Mená adresárov sú rovnaké ako mená balíčkov. Cestu k balíčkom udáva systémová premenná CLASSPATH, ktorá štandardne obsahuje cestu k balíčkom Java Core API a do aktuálneho adresára
- Meno premennej alebo metódy alebo triedy vrátane mena balíčka predstavuje úplné meno, ktoré môžeme kedykoľvek použiť, keď chceme zabrániť nejednoznačnostiam.
- Za situácie, keď nepoužívame import balíčkov, musíme použiť úplne meno. Aby sme nepoužívali úplné mená, Java poskytuje možnosť pomocou kľúčového slova import rozšíriť rozsah platnosti identifikátorov na požadovanú triedu.
- Defaultný balíček

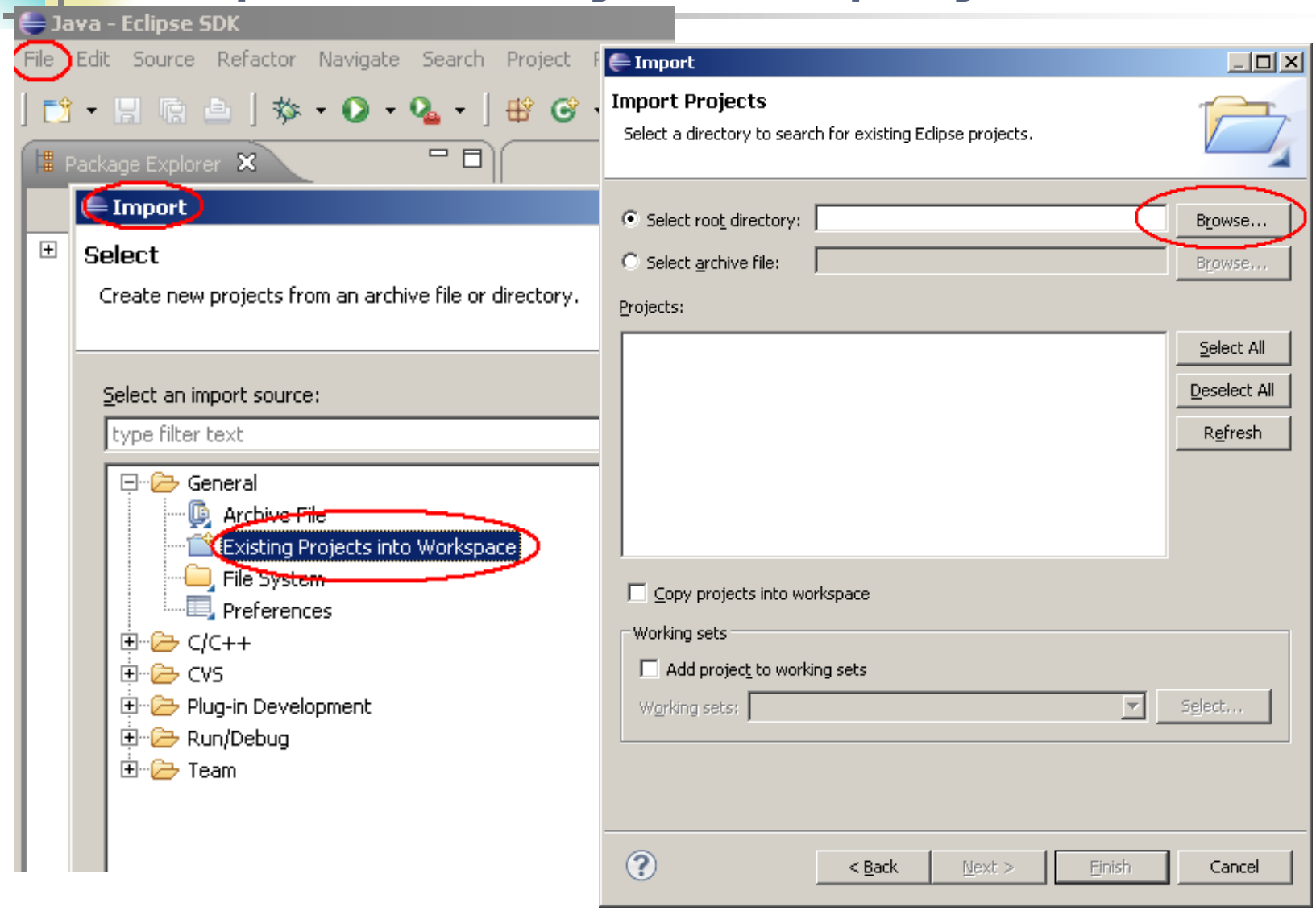
Implicitné baličky

- Aj bez kľúčového slova import sú importované tri balíčky:
  - java.lang
  - default - balík, v ktorom sú všetky triedy nepatriace do žiadneho balíčka
  - aktuálny balík

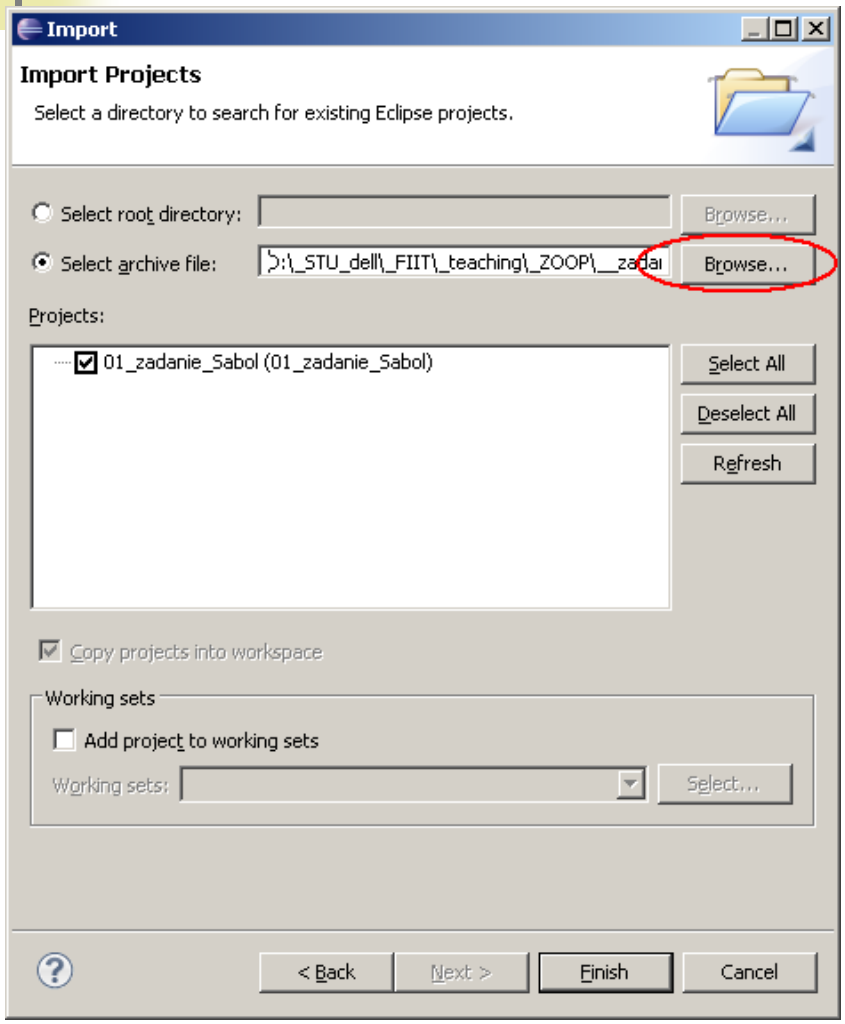
Benefity vytvárania balíčkov resp. prečo...?

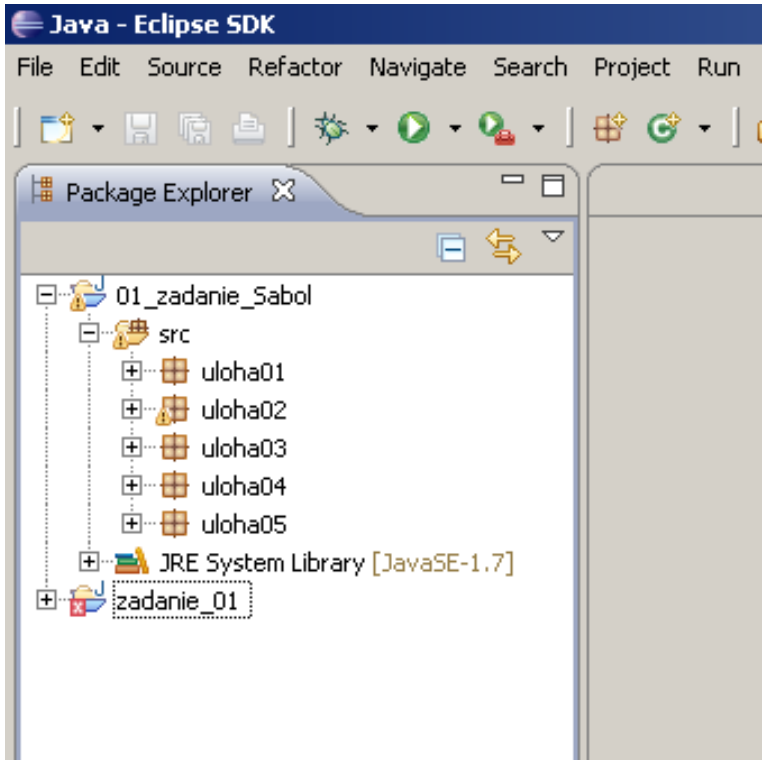
- Aby súvisiace veci boli pokope (v adresári)
- Aby v adresári bolo len rozumne veľa .java, .class súborov
- Aby sme si nemuseli vymýšľať stále nové unikátne mená tried
- Aby Java chránila prístup dovnútra balíčka

Import existujúceho projektu



Import existujúceho projektu





TODO

- Aj vy môžete pomôcť vylepšiť tento predmet študentom pre nasledujúci akademický rok. Vaše odporúčanie, komentár či otázka.