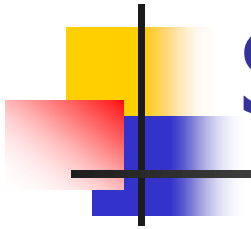
Triedy a objekty

Peter Borovanský
KAI, I-18

borovan 'at' ii.fmph.uniba.sk
http://dai.fmph.uniba.sk/courses/JAVA/

JVM, JRE, JDK



Statické metódy

doposiaľ sme (okrem pár skrytých prípadov – Random, Calendar, BigInteger) používali len statické metódy (System.*currentTimeMillis*), premenné a konštanty (Math.PI).

Statické metódy:

- predstavujú klasické procedúry/funkcie ako ich poznáme z C++,
- existujú automaticky, ak použijeme (importujeme) danú triedu,
- **existujú bez toho, aby sme vytvorili objekt danej triedy**,
- referencujú sa *menom*, napr. *vypis(pole),* alebo *menom triedy.meno metódy*, konštanty, napr. *Math.cos(fi), Math.PI, Systém.out.println(5),*
- ak aj metóda nemá argumenty, prázdne zátvorky sa do jej definície a do volania aj tak píšu (à la C++), napr. *System.out.println**();***

- syntax deklarácie statickej metódy je
  [**public**] **static** *typ meno***(***argumenty***) {** telo **}**

- ak ide o procedúru (nie funkciu), výstupný typ je **void**

Alan Kay
(OOP)

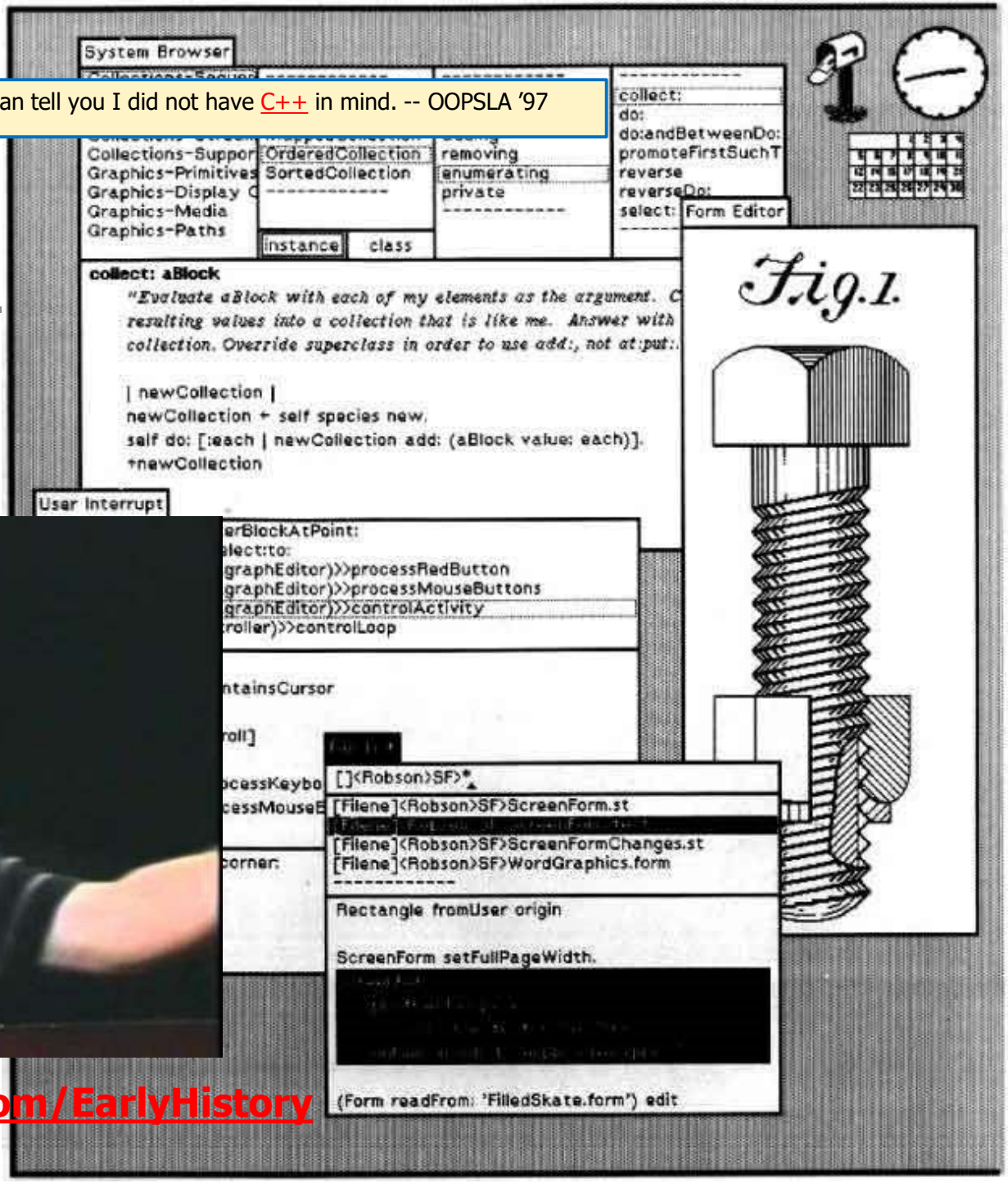
1. Everything is an object
2. Objects communicate by sending and receiving messages (in terms of objects)
3. Objects have their own memory
4. Every object is an instance of a class
5. The class holds the shared behavior for its instances
6. To eval a program list, control is passed to the first object and the remainder is treated as its message.

LISP

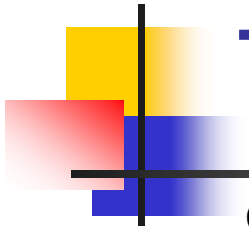
Actually I made up the term "object-oriented", and I can tell you I did not have C++ in mind. -- OOPSLA '97

Alan Kay
(Smalltalk '72)

Zaujímavé čítanie:
The Early History
Of Smalltalk



http://worrydream.com/EarlyHistoryOfSmalltalk/



Triedy a objekty

dnes bude:

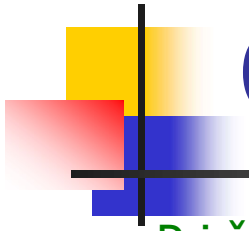
- prvá trieda/objekt (porovnanie konceptov a syntaxe s C++)
- konštruktory a metódy triedy
- preťažovanie konštruktorov a metód (vs. polymorfizmus)
- dedenie (nadtrieda a podtrieda) a veci súvisiace
- abstaktná trieda, abstraktná metóda
- triedne vs. statické metódy, premenné

cvičenia:

- vytvoriť malú hierarchiu tried/objektov
- vytvoriť abstraktnú triedu s podtriedami

literatúra:

- Thinking in Java, 3rd Ed. (http://www.ibiblio.org/pub/docs/books/eckel/TIJ-3rd-edition4.0.zip) – 4:Initialization & Cleanup,
- Naučte se Javu – úvod
  - http://interval.cz/clanky/naucte-se-javu-tridy-a-objekty-1/,
  - http://interval.cz/clanky/naucte-se-javu-tridy-a-objekty-2/,



OOP pojmy

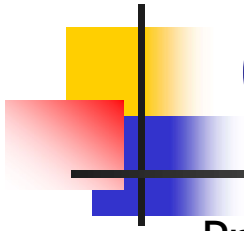
•všetko je objekt
•každý objekt má typ
•každý objekt má svoj kus pamäte
•program je hŕba objektov oznamujúcich si, čo robiť, posielaním správ
Alan Kay (Xerox Parc, Smalltalk, Macintosh)

Pri štúdiu ste sa už stretli s nasledujúcimi pojmami. Cieľom prednášky je ujasniť si ich význam, použitie a syntax v jazyku Java, nie ich preberať znova ...

- trieda – definícia abstraktného typu dát
- objekt – inštancia triedy – implementuje stav entity, vyváža jej metódy
- dedičnosť – podtrieda a nadtrieda, viacnásobné dedenie
- virtuálne metódy a dynamic binding (v C++)
- interface vs. abstraktná trieda
- ukrývanie (encapsulation) – public/private/protected/ ... toto nebude dnes
- preťažovanie (overloading) vs. polymorfizmus metód
- polymorfizmus – rôzne správanie objektov pri volaní metódy

V prednáške predpokladáme, že ste prešli školou procedurálneho programovania a statické metódy máme za sebou... Jedinú **statickú** metódu, ktorú uvidíte, je hlavný program main().

•z posielania správ sa stalo volanie metód
•miesto stavu objektu v správe, voláme metódu s referenciou na stav objektu – nie je to isté
Alan Kay (Xerox Parc, Smalltalk, Macintosh)



OOP vs. procedural

Procedurálne programovanie

- dekompozícia procesov/akcií na jednoduchšie
- klasická metóda rozdeľ-a-panuj

Objektovo-orientované

- dekompozícia problému na objekty/entity vystupujúce v ňom
- typ objektu (trieda) popisuje jeho stav a metódy
- objekt má stav, ktorý sa mení volaním metód

Malo svoje krízy, z ktorých sa liečilo

- no goto statement
- štruktúrované programovanie
- modulárne programovanie

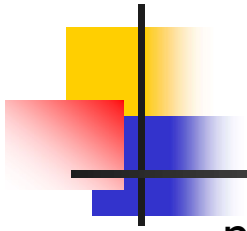
  zárodok enkapsulácie

Má svoje krízy, z ktorých sa lieči

- návrhové vzory
- SOLID princípy tvorby OO aplikácie
- Agile technikami
- Test-Driven Development

https://www.youtube.com/watch?v=QM1iUe6IofM

Definujte triedu na reprezentáciu komplexného čísla



Prvý objekt

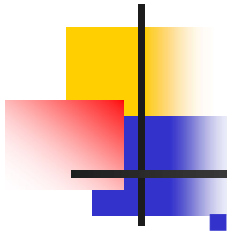
```
public class Complex {                                  // definícia triedy
  private double real, imag;                            // triedne premenné
  // private znamená, že ich nevidno mimo triedu
  public Complex(double _real, double _imag) {          // konštruktor
  // konštruktor má meno zhodné s triedou
       real = _real; imag = _imag;
  }
  public String toString() {                    // textová reprezentácia
       return "["+real+ "+" +imag+"*i]";
  }
}
```

Príklad použitia triedy Complex:

```
public static void main(String[] args) {
   Complex c1 = new Complex(1,0);   // 1
   Complex c2 = new Complex(0,1);   // i, i2 = -1
   System.out.println(c1);  // skryté volanie toString
   System.out.println(c2);
}  // nedeštruujeme objekt !!! urobí to sám
```

Súbor: Complex.java

this alias self



Prvý konštruktor

- konštruktor je metóda s menom zhodným s menom triedy, bez výstupného typu,
- konštruktor je najčastejšie je public. Môže byť private ? (premia?),
- trieda môže mať viacero preťažených konštruktorov (uvidíme neskôr),
- objekt triedy vytvoríme tak, že zavoláme konštruktor (resp. niektorý z konštruktorov) triedy pomocou **new**, príklad new Complex(1,0).
- výsledkom volania new (v prípade úspechu) je objekt danej triedy, t.j. Complex c1 = new Complex(1,0);
- a čo v prípade neúspechu ?
- **this** je referencia na aktuálny objekt v rámci definície triedy,
- cez **this.** sa dostaneme k triednym premenným, ak potrebujeme:

```
public class Complex {                                  // definícia triedy
  private double real, imag;                            // triedne premenné
                                                        // nie sú static
  public Complex(double real, double imag) {            // konštruktor
      this.real = real; this.imag = imag;
  }
```

Súbor: Complex.java

Nech je skryté, čo
má ostať skryté ...

Vlastnosti - properties

•K premenným reprezentujúcim stav objektu pristupujeme cez metódy, ktoré sprístupnia ich hodnotu (getter), a modifikujú (setter) na novú hodnotu.

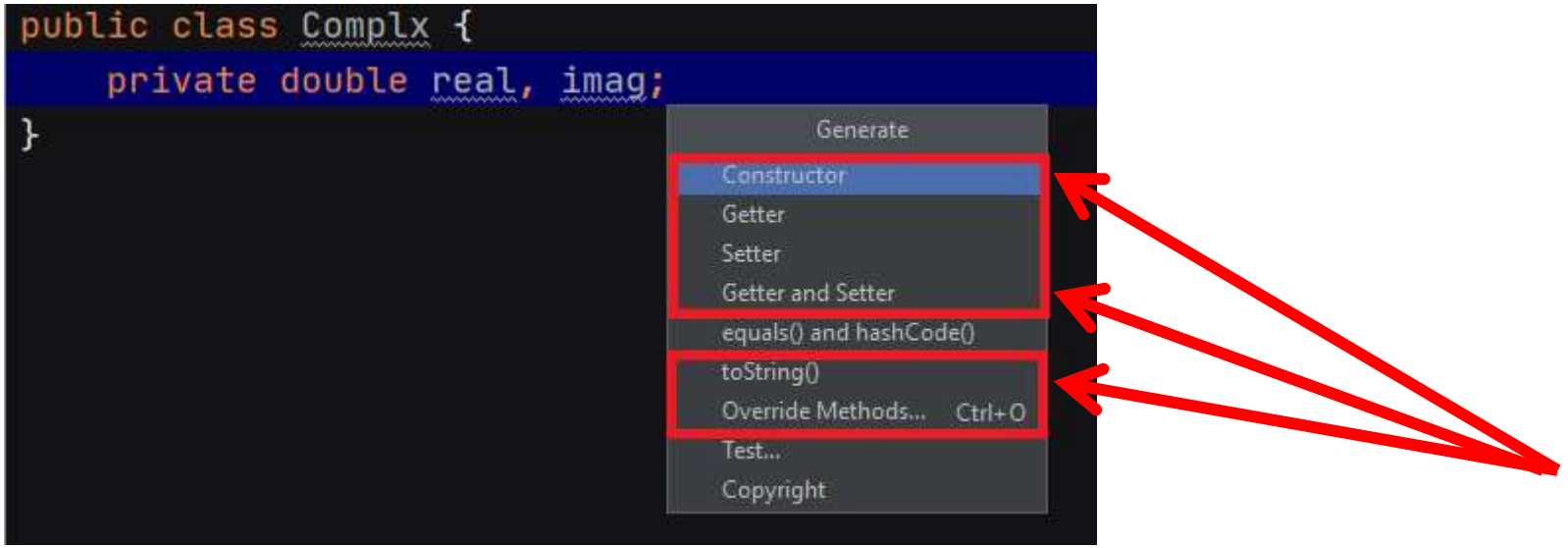
```
public class Complex {
  private double real, imag;                    // enkapsulácia
  ....                                          // ukrytie vnútornej reprezentácie

                                                // properties
  public double getReal() { return real; }      // getter
  public void setReal(double _real) { real = _real; }   // setter

  public double getImag() { return imag; }      // getter
  public void setImag(double imag) { this.imag = imag; } // setter
```

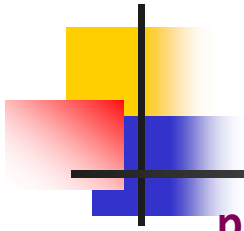
```
System.out.println(Math.sqrt(                   // použitie mimo triedy
        c1.getReal()*c1.getReal() +             // výpočet dĺžky k.čísla
        c1.getImag()*c1.getImag()));
```

Súbor: Complex.java

IntelliJ: ALT-Insert/ALT-Enter

Nechajte si vygenerovať
- konštruktory a
- get/set metódy
- toString()





Dostanete

```
public class Complex {
    private double real, imag;
    public Complex(double real, double imag) {
       super();
       this.real = real; this.imag = imag;
    }
    public double getReal() { return real; }
    public void setReal(double real) { this.real = real; }
    public double getImag() { return imag; }
    public void setImag(double imag) { this.imag = imag; }
    @Override
    public String toString() {
       return "Complex [real=“+real+", imag=“+ mag+"]";
    }
}
```

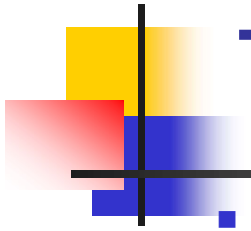
Súbor: Complx.java

Java Beans

V JAVE existuje koncept tzv. JAVA Beans, čo sú objekty tried napísaných pri dodržaní istých konvencií:

- majú **defaultným konštruktorom** bez argumentov, t.j. napr. `Complex()`
- majú gettery a settery - pre každú privátnu hodnotu - property *`Prop`* typu *`typ`*, disponuje metódami
  `public` *`typ`* `get`*`Prop`*`()` – vráti hodnotu `Prop : typ`, a
  `public void set`*`Prop`*`(`*`typ`* `x)` – nastaví hodnotu *`Prop`* na x : *`typ`*,
  napr. `Complex.getReal():Real`, alebo `Complex.setImag(x:Real)`
- a pre logické hodnoty poskytuje `public boolean is`*`Prop`*`()`
- a je serializovateľný

Tieto konvencie slúžia napísanie znovu použiteľných tried, napr. pri definícii vizuálnych komponentov a pod.



Triedne metódy

- nie sú statické (neobsahujú static)
- aplikujú sa vždy na objekt danej triedy
- ten však musí existovať pred aplikáciou

```
public class Complex {
 private double real, imag;
 public double abs() {                          // veľkosť vektora komp.čísla
    return Math.sqrt(real*real + imag*imag);
 }
 public void add(Complex c) {                   // súčet komplexných čísel
    real  += c.real;
    imag += c.imag;
 }
 public void mult(Complex c) {                  // súčin komplexných čísel
     double _real  = real*c.real-imag*c.imag;
     double _imag = real*c.imag+imag*c.real;
     real = _real;
     imag = _imag;
 }
```

```
System.out.println(c1.abs());

c1.add(c2);
c2.mult(c2);
System.out.println(c1);
System.out.println(c2);
```

```
[1.0+1.0*i]
[-1.0+0.0*i]
```

Súbor: Complex.java

Preťažovanie konštruktorov

Preťažovanie vie kompilátor rozhodnúť pred spustením programu, zo syntaxe.
Preťažovanie a virtual nesúvisia

Preťažiť môžeme konštruktor, metódu ale nie operátor ☺

```
public class Complex {
  private double real, imag;
  ....
 public Complex(double real, double imag) {
    this.real = real; this.imag = imag;
 }                                          // ďalší konštruktor rozpoznáme napr.
 public Complex() {                         // iným počtom argumentov
    real = 0; imag = 0;                     // vytvorí komplexné číslo [0,0]
 }
}
```

Konštruktor môže volať iný konštruktor tej istej triedy pomocou this()

```
public Complex() {                          // this(..) musí byť prvý príkaz
   this(0,0);                               // volanie Complex(double,double)
 }
```

Súbor: Complex.java

Preťažovanie metód

Preťažená metóda/konštruktor sa musí dať identifikovať (letmým pohľadom do programu) iným počtom resp. typom argumentov.

```
public class Complex {
  private double real, imag;
  ....
 public void mult(Complex c) { ... viď slide this-2 }
 public void mult(double r) {            // iný typ argumentov
    real   *= r;
    imag *= r;
 }
```

Príklady zakázaného preťaženia:

```
public double abs() { return Math.sqrt(real*real + imag*imag);   }
public int abs() { ... }                  // iný výstupný typ nestačí na rozlíšenie

public void mult(Complex c) { ... viď slide this-2 }
public Complex mult(Complex c) { ... }        // rozdiel proc/func tiež nestačí
```

Súbor: Complex.java

•Java nedovoľuje programátorovi preťažiť operátor, našťastie ☺
•ale niektoré preťažené sú ...

Preťažovanie operátorov

- 3 + 7
- 3.0 + 7
- 3 + 7.0
- 3.0 + 7.0

Preťažovanie vs. pretypovanie

- double+double

žiadne preťažovanie len pretypovanie

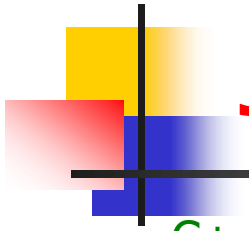
- int + int
- double + int
- int + double
- double + double

4 prekrývajúce sa operátory, žiadne pretypovanie len preťaženie

- int + int
- double + double

2 preťažené operátory,

- 3.0 + (double)7
- (double)3 + 7.0



JAVA class – zhrnutie pre C++

C++

- má struct{…}**;** a class{…}**;**
- class Complex{…};
  Complex c; // vytvorí objekt
- Complex cc = c; // kopíruje
- Complex *p = new Complex;
  p->real = … c.real

JAVA

- len class {…} aj to bez **;** na konci ☺
- class Complex{…}
  Complex c; // deklaruje referenciu
  c=new Complex();// vytvorí sa až tu
- Complex cc = c; // nekopíruje, ale
  Complex cc = c.clone(); // kopíruje
- neexistuje rozdieľ medzi objektom a pointrom (referenciou), preto k položkám a metódam objektu vždy pristupujeme pomocou “.”

Konštruktory nadtriedy

konštruktory triedy môžu byť preťažené

```
package SuperAndSub;

public class Nadtrieda {

    public Nadtrieda() {
            System.out.println("Konstruktor nadtriedy");
    }
    public Nadtrieda(int n) {
            System.out.println("Konstruktor nadtriedy n="+n);
    }
    public Nadtrieda(String s) {
            System.out.println("Konstruktor nadtriedy s="+s);
    }
    public void foo() {
            System.out.println("Nicnerobiaca funkcia foo v nadtriede");
    }
}
```

Súbor: Nadtrieda.java

Konštruktory podtriedy

super. verzus super()

```
package SuperAndSub;

public class Podtrieda extends Nadtrieda{
    public Podtrieda() {
        System.out.println("Konstruktor podtriedy");
    }
    public Podtrieda(int n) {
        System.out.println("Iny konstruktor podtriedy n="+n);
    }
    public Podtrieda(String s) {
        super(s+s);
        System.out.println("Konstruktor podtriedy s="+s);
    }
    public void foo() {
        System.out.println("Nicnerobiaca funkcia foo v podtriede");
        super.foo();
    }
}
```

konštruktor podtriedy najprv zavolá:
implicitný (bez arg.) konštruktor nadtriedy,
explicitne niektorý z konštruktorov
pomocou super(...)

// volanie konštruktora musí byť 1.príkaz

// volanie foo z nadriedy

Súbor: Podtrieda.java

Hlavný program

```
package SuperAndSub;

public class Main {
 public static void main(String[] args) {
    Nadtrieda nad = new Nadtrieda();
    Podtrieda pod = new Podtrieda();

    Nadtrieda nadInt = new Nadtrieda(10);
    Podtrieda podInt = new Podtrieda(100);

    Nadtrieda nadString = new Nadtrieda("wow");
    Podtrieda podString = new Podtrieda("wow");

    nadString.foo();
    podString.foo();
  }
}
```

Konstruktor nadtriedy

Konstruktor nadtriedy
Konstruktor podtriedy

Konstruktor nadtriedy n=10

Konstruktor nadtriedy
Iny konstruktor podtriedy n=100

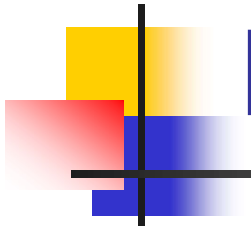
Konstruktor nadtriedy s=wow

Konstruktor nadtriedy s=wowwow
Konstruktor podtriedy s=wow

Nicnerobiaca funkcia foo v nadtriede

Nicnerobiaca funkcia foo v podtriede
Nicnerobiaca funkcia foo v nadtriede

Súbor: Podtrieda.java



Deštruktory

Deštruktory sú Jave implicitné.
Ak nemáme dôvod, nedefinujeme ich !
A ak aj máme, tak ich nevoláme...
Volá ich garbage collector a nemáme nad tým kontrolu...

```
public void finalize() {                    // deštruktor triedy sa volá finalize
    System.out.println("GC vola destruktor v podtriede");
}

for(int i=0; i<5000; i++) {                 // provokujeme garbage collector
                                            // aby začal zbierať „smeti“
   Nadtrieda nadInt = new Nadtrieda(i);
   Podtrieda podInt = new Podtrieda(i);
  ....
}
```

```
už začal...
GC vola destruktor v podtriede n=-890 s=null
GC vola destruktor v nadtriede n=0 s=null
GC vola destruktor v nadtriede n=890 s=null
```

Dedenie

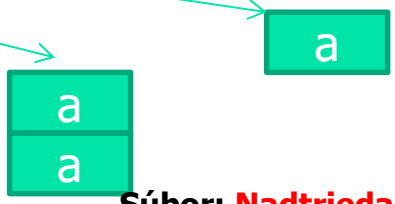
- má v JAVE syntax: [public] class Podtrieda **extends** Nadtrieda { ... }
- podtrieda obsahuje všetky premenné, konštanty a metódy nadtriedy,
- na predefinovanie metódy v podtriede nikde nepíšeme override,
- predefinovať môžeme každú metódu, všetko je virtual,

```
public class Nadtrieda {
  public int a;
  public Nadtrieda() { a = 0; }
  public int getA() { return a; }
  public void setA(int a) { this.a = a; }
}
```

```
public class Podtrieda extends Nadtrieda {
  public int a;  // prepíše či pridá ?
  public Podtrieda() { a = -1; }
  public int getA() { return a; }
  public void setA(int a) { this.a = a; }
  public int getSuperA() { return super.a; }
  public int getSuperGetA() { return super.getA(); }
}
```

```
public static void main(String[] args) {
  Nadtrieda x = new Nadtrieda(); x.setA(5);
  Podtrieda y = new Podtrieda(); y.setA(6);
  System.out.println(x.getA());
  System.out.println(y.getA());
  System.out.println(y.getSuperA());
  System.out.println(y.getSuperGetA());
```

5
6
0
0



Súbor: Nadtrieda.java, Podtrieda.java

Statické vs. dynamické typy

- definícia podtriedy class TPodtrieda(Tnadtrieda)
- Python je dynamicky typovaný jazyk, ako mnoho iných (moderných):
  - Javascript
  - PHP
  - Ruby
- znamená to, že typ hodnoty premennej je známy až počas behu programu

- Java je staticky typovaný jazyk, ako mnoho iných (slušných):
  - C, C++
  - Haskell
  - Scala
  - C# (bez dynamic)
  - Java (Reflection model)

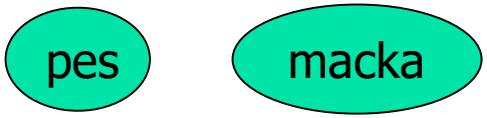
- znamená to, že typ hodnoty premennej je známy už počas kompilácie,
  aj keď programátor ich niekedy nemusí typy písať – kompilátor si domyslí

Ako to bolo v Pythone (Duck typing)

If it looks like a duck and quacks like a duck, it must be a duck ...

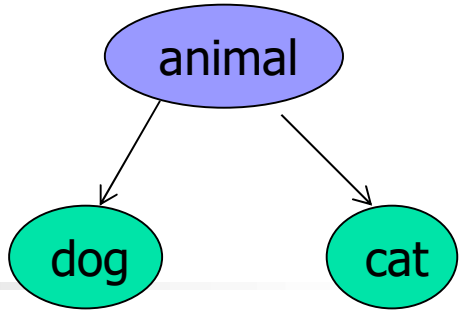
je forma dynamického typovania, dynamická náhrada virtuálnych metód

```
class pes():              # definujeme dve triedy bez akejkoľvek dedičnosti
    def zvuk(self):       # obe definujú metódu zvuk()
        return "haw-haw"

class macka():
    def zvuk(self):       # pes je vlastne mačka, lebo podná všetky jej metódy
        return "mnau-mnau"  # a mačka je tiež pes...

def zvuk(zviera):  # otázkou (statického programátora) je, akého typu je
    print(zviera.zvuk())  # premenná zviera, keď na ňu aplikujeme .zvuk()
                          # odpoveď: uvidí sa v RT podľa hodnoty premennej
farma = [pes(), macka()]   # heterogénny zoznam objektov

for zviera in farma:
    zvuk(zviera)
```



haw-haw
mnau-mnau

Ako to bude v Jave

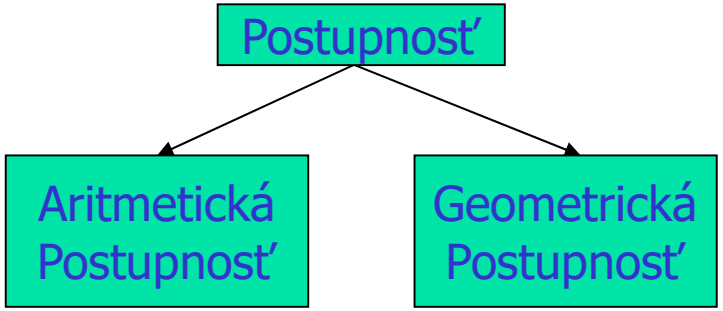



```
abstract class Animal {  // nikdy nemôžem vytvoriť objekt triedy Animal
   abstract void sound();  // teda zavolať new Animal()
}
class Dog extends Animal {
   public void sound() { System.out.println("haw-haw"); } }
class Cat extends Animal {
   public void sound() { System.out.println("mnaw-mnaw");}}

Animal[] animals = { new Dog(), new Cat() };
for(Animal a:animals) a.sound();
for(Animal a:animals){
   if (a instanceof Dog)
        System.out.println("it's a dog");
   else
        System.out.println("not a dog");
}
```

haw-haw
mnaw-mnaw

it's a dog
not a dog

Postupnosť




```
abstract class Postupnost {          // abstraktná trieda má abstraktnú
                                     // metódu, t.j. nemá inštancie
  protected long prvy;                        // prvý prvok postupnosti
  protected long aktualny;                    // aktuálny prvok potupnosti

  public long Prvy() {                        // skoč na prvý prvok
      aktualny = prvy;
      return aktualny;
  }
  abstract long Dalsi();                      // daj mi ďalší prvok

  public void printPostupnost(int n) {        // vytlač postupnosť
      System.out.print(Prvy());
      for(int i= 0; i<n; i++)
            System.out.print(", "+ Dalsi());  // volá sa nejaká ešte
    System.out.println();                     // neznáma motóda
  }
}
```

Súbor: Postupnost.java

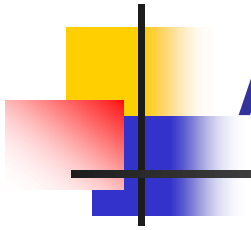
Aritmetická postupnosť

```
AritmetickaPostupnost r =
            new AritmetickaPostupnost(13,10);
r.printPostupnost(10);
```

13, 23, 33, 43, 53, 63, 73, 83, 93, 103, 113

```
public class AritmetickaPostupnost extends Postupnost { // podtrieda

protected long delta;                    // rozdiel' medzi posebeidúcimi prvkami

   AritmetickaPostupnost(int _delta) {              // konštruktor
      delta = _delta; prvy = 0;
    }

   AritmetickaPostupnost(int _prvy, int _delta) {   // ďalší konštruktor
      delta = _delta; prvy = _prvy;                 // preťaženie
    }

    public long Dalsi() {                // konkretizácia abstraktnej metódy
      aktualny += delta;
      return aktualny;
    }
}
```

Súbor: AritmetickaPostupnost.java



Abstraktná trieda/metóda

- abstraktná trieda obsahuje (môže obsahovať) abstraktnú metódu,
- abstraktná metóda má len hlavičku, jej telo bude definované v niektorej z podtried,
- abstraktná trieda nemôže mať inštancie, nie je možné vytvoriť objekt takejto triedy (lebo nepoznáme implementáciu abstraktnej metódy),
- kým nedefinujeme telo abstraktnej metódy, trieda je abstraktná,
- a nedá sa to oklamať:

```
public class ZlaPostupnosti extends Postupnost {
protected long delta;
                        // musí implementovať ZlaPostupnost.Dalsi()
  ZlaPostupnosti(int _delta) {
   delta = _delta; prvy = 0;
  }
}
```

Súbor: ZlaPostupnost.java

Geometrická postupnosť

```
GeometrickaPostupnost q =
            new GeometrickaPostupnost(1,2);
q.printPostupnost(10);
```

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024

```
public class GeometrickaPostupnost extends Postupnost {

    protected long quotient;                    // podieľ susedných prvkov

    GeometrickaPostupnost(int prvy, int quotient) {
       this.quotient = quotient;                // this je referencia na objekt
       this.prvy = prvy;                        // na ktorý bola metóda
    }                                           // aplikovaná

    public long Dalsi() {                       // konkretizácia abstrakcie
        aktualny *= quotient;
        return aktualny;
    }
}
```

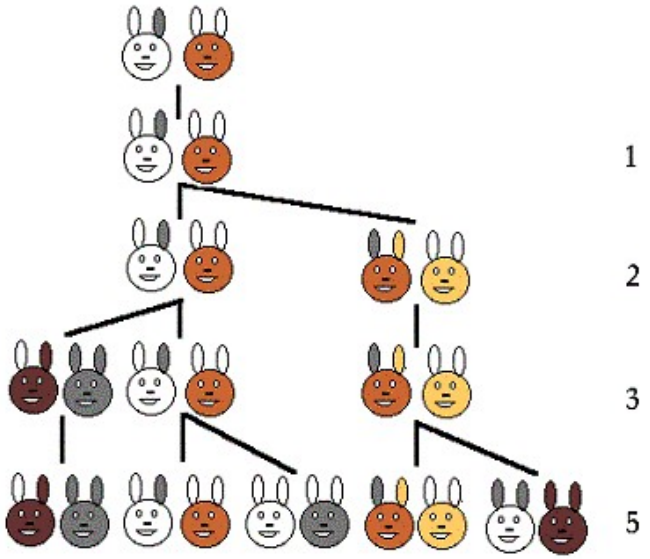
Súbor: GeometrickaPostupnost.java

Fibonacciho postupnosť

```
FibonaccihoPostupnost f =
            new FibonaccihoPostupnost(0,1);
f.printPostupnost(10);
```

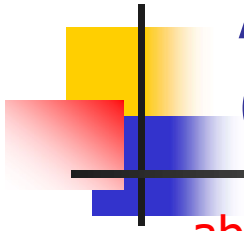
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

```
public class FibonaccihoPostupnost extends Postupnost {

  protected long predch;

  FibonaccihoPostupnost(long _prech, long _aktual) {
    predch = _prech;
    prvy = aktualny = _aktual;
  }

  public long Dalsi() {
      long pom = aktualny;
      aktualny += predch;
      predch = pom;
      return aktualny;
    }
}
```



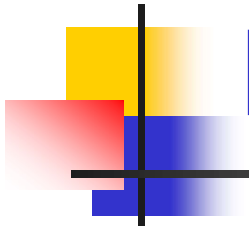
Súbor: FibonaccihoPostupnost.java

Tony Hoare: Abstraction arises from a recognition of *similarities between certain objects*, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences.



Abstrakcia
(bude na cvičení)

```
abstract public class Polynom {
    abstract Double valueAt(String[] vars, double[] values);  // hodnota
    abstract Polynom derive(String var);        // derivácia podľa premennej
}
public class Konstanta      extends Polynom {
    double m;                                    // reprezentácia konštanty
    public Konstanta (double m ){ this.m=m ; }          // konštruktor
    public Double valueAt(String[] vars, double[] values){ return m ; }
    public Polynom derive(String var){ return new Konstanta(0); }  // derivácia
    public String toString() { return String.valueOf(m);  } // textová reprezent.
}
public class Premenna       extends Polynom { ... }
public class Sucet          extends Polynom { ... }
public class Sucin          extends Polynom { ... }
```



Dedičstvo C++ vs. JAVA

- dedenie

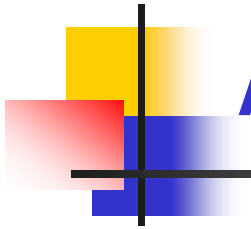
class TPodtrieda:public TNadtrieda{};

- ukrývanie premenných a metód v triede je podobne ako JAVA
- ukrývanie pri dedení public/private/protected dedenie "ťažšie témy"
- virtuálne metódy

- dedenie

class Podtrieda extends Nadtrieda {}

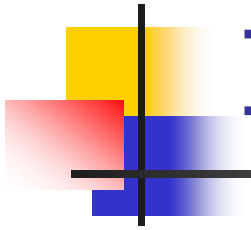
- public/private/protected/*nič* nič zodpovedá friendly
- zjednodušené len jedno dedenie: public môže prepísanať len public, private môže prepísanať len private, etc.
- (skoro) každá nestatická metóda môže byť predefinovaná bez syntatického upozornenia. V Jave je každá metóda virtuálna a má dynamic binding. Predefinovať nemožno len final metódu.



Abstract, virtual, interface

- iné použitie virtuálnej metódy – neupresnená metóda, ktorá bude dodefinovaná v podtriede, napr. virtual void vykresliMa();
- viacnásobne dedenie keďže to robí problémy (diamond problem), zaviedli virtuálne dedenie, čo je vlastne dedenie bez dedičstva...
- deštruktory a dealocate na odstránenie zbytočných objektov

- abstraktná metóda abstraktnej triedy
- alebo interface (uvidíme neskôr)
- nemá viacnásobné dedenie, ale virtuálne dedenie nahradil konceptom interface a trieda môže spĺňať/implementovať viacero interface
- má automatickú správu pamäte a deštruktory píšeme zriedka



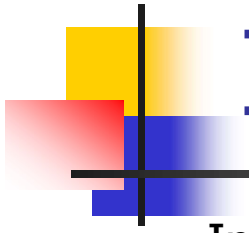
Interface

- je súbor metód, ktoré objekt danej triedy pozná, ... musí !
- ak trieda implementuje interface, t.j. každá jej inštancia pozná všetky metódy z inteface

Príklad: java.lang.Comparable

```
public interface Comparable<T> { // kto che byť Comparable
    int compareTo(T o);              // musí poznať compareTo
}
public class Student implements Comparable<Student> {
private String name;          // chýbajú gettery a settery
private int age;
public int compareTo(Student o) {
   if (this.age > ((Student) o).getAge()) return 1;
    else if (this.age < ((Student) o).getAge()) return -1;
    else return 0;
} }
```

Student.java



Interface ako typ

Iný príklad: implementujte haldu pomocou poľa, aby spĺňala:

```
interface HeapStringInterface {   // reprezentujte Max-heap
   public String first();         // vráti najväčší
   public String remove();        // odstráni najväčší
   public void insert(String str);// pridá prvok
}
```

- interface na rozdiel od triedy nemá inštancie, nejde urobiť new Comparable
- interface zodpovedá tomu, čo poznáme pod pojmom T Y P

```
interface Car {
   int speed = 50;// in km/h
   public void distance();
}
```

```
interface Bus {
   int distance = 100;// in km
   int speed = 40;// in km/h
   public void speed();
}
```

- interface teda môže obsahovať premenné, ale sú automaticky static a final, aj keď ich tak nedeklarujeme... ☹ škoda, čistejšie by bolo, keby to kompilátor vyžadoval, teda `final static int speed = 50;`

Car.java, Bus.java

Abstract vs. Interface

(rekapitulácia – tentokrát už v Jave)

aký je rozdiel medzi abstraktnou triedou a interface:

- abstract class XXX { … foo(…) ; } a interface XXX {… foo(…) ; }

1. trieda **dedí** od abstraktnej triedy, pričom trieda **implementuje** interface
2. rovnako **nejde urobiť new** od abstraktnej triedy ani od interface
3. abstraktná trieda môže predpísať defaultné správanie v neabstraktných metódach
4. abstraktná trieda vás donúti v podtriedach dodefinovať správanie abstraktných metód
5. trieda môže zároveň **implementovať viac interface**, ale nemôže dediť od viacerých

abstraktná trieda	**interface**
môže mať abstraktné aj neabstraktné metódy	len abstraktné public, takže **public abstract** ani nepíšeme
dve abstraktné triedy nemôžeme podediť do jednej	**interface** podporuje viacnásobné dedenie
môže mať final/non-final, static/non-static premenné	len static a final, takže k nim **static final** ani nepíšeme
môže mať statické metódy (napr. main), aj konštruktor	... nič z toho
abstraktná trieda môže implementovať interface	Interface nie je implementáciou abstraktnej triedy

Viditeľnosť metód/premenných

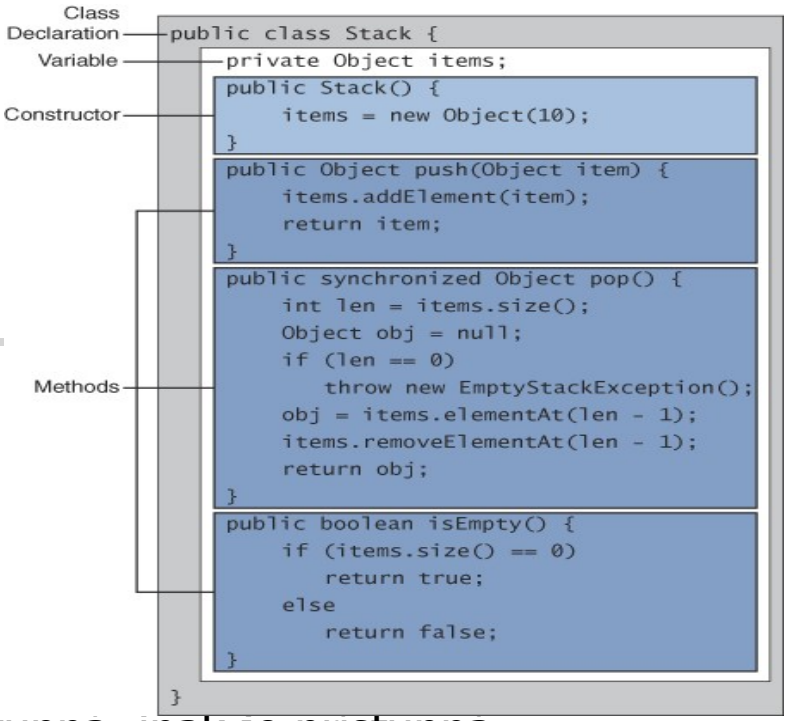
	Trieda	Package	Podtrieda	Inde
private	+	-	-	-
nič	+	+	-	-
protected	+	+	+	-
public	+	+	+	+

Príklady:

```
public final int MAX = 100;          // deklarácia viditelnej konštanty
protected double real, imag;         // lokálne premenné
void foo() { }                       // metódu vidno len v balíčku
private int goo() { }                // najreštriktívnejšie-fciu je len v triede
```

Deklarácia triedy

(rekapitulácia syntaxe)



class *MenoTriedy* {// MenoTriedy.java
TeloTriedy
}

- [**public**] trieda je voľne prístupna, inak je prístupná len v danom package
- [**abstract**] trieda nemôže byť inštanciovaná (asi obsahuje abstr.metódu) t.j. neexistuje objekt danej triedy
- [**final**] trieda nemôže mať podtriedy, „potomkov"
- [**extends** *supertrieda*] trieda je podtriedou inej triedy, dedičnosť
- [**implements** Interfaces{,}*] Interfaces sú implementované v tejto triede

Deklarácia metódy

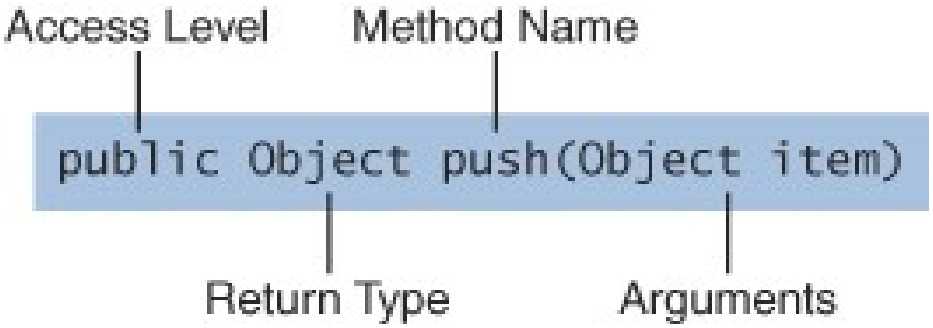
(rekapitulácia)

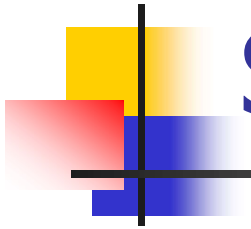
typ MenoMetódy(argumenty) {

telo metódy

}

- [**static**] triedna metóda, existuje nezávisle od objektov triedy
- [**abstract**] metóda, ktorá nie je implementovaná,bude v podtriede
- [**final**] metóda, ktorá nemôže byť predefinovaná, bezpečnosť
- [**native**] metóda definovaná v inom jazyku, „prilinkovaná“
- [**synchronized**] metóda synchronizujúca konkurentný prístup bežiacich threadov, neskôr...
- [**throws**] exceptions metóda produkujúca výnimky





Statické vs. triedne

v procedurálnom prístupe sme si zvykli definovať všetky metódy ako statické a nazývali sme ich procedúry a funkcie,

- volali sme ich cez meno triedy, explicitné či skryté, napr. Math.cos(fi), alebo len cos(fi),
- statická premenná triedy existuje v jedinej kópii,
- statická premenná *je ako globálna premenná* v rámci danej triedy,

v objektovom prístupe definujeme (*aj*) triedne metódy a triedne premenné,

- aplikujú sa na objekt triedy, ktorý musí byť vytvorený,
- inštancií triednej premennej existuje toľko, koľko je inštancií triedy,
- triedna premenná *je ako lokálna premenná* v rámci každej inštancie

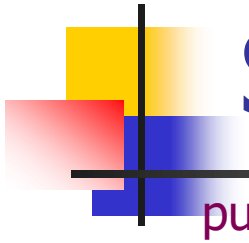
to, čo robí problémy, je miešanie statického a nestatického kontextu

Statické verzus triedne

(premenné aj metódy)

```
public class StaticVsClass {

  static int pocetInstancii = 0;          // statická premenná
  final static int MAX = 10;              // statická konštanta
  int indexInstancie;                     // triedna/nestatická premenná
  final int MIN = 7;                      // triedna/nestatická konštanta

 StaticVsClass() {                        // konštruktor
    indexInstancie = ++pocetInstancii;
 }
 static int rest() {                      // statická metóda
    return MAX-pocetInstancii;
 }
 int getIndex() {                         // nestatická metóda
    return indexInstancie;
 }
```

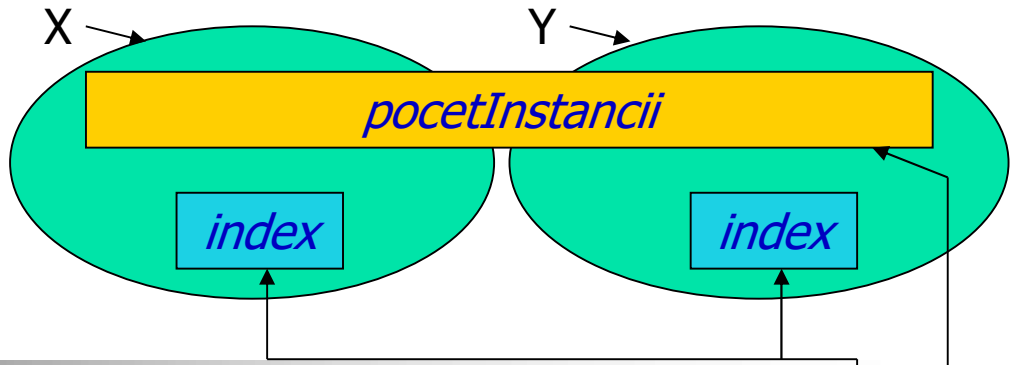
Súbor: StaticVsClass.java



Statické verzus nestatické

```
public static void main(String args[]) { // statický kontext
  int a = MAX +                          // referencia statickej premennej
          StaticVsClass.MAX +            // úplná referencia Trieda.var
          StaticVsClass.rest();          // referencia statickej metody
                                         // ... toto nejde !!!
  int b = StaticVsClass.MIN + // nestatická konštanta v statickom kontexte
          indexInstancie +    // nestatická premenná v statickom kontexte
          getIndex();         // nestatická metóda v statickom kontexte

  StaticVsClass X = new StaticVsClass(); // objekt triedy StaticVsClass

  int c = X.indexInstancie +// nestatická premenná v nestatickom kontexte
          X.MIN +           // nestatická konštanta v nestatickom kontexte
          X.getIndex();     // nestatická metóda v nestatickom kontexte
                            // ... aj toto ide !!
  int d = X.MAX +           // statická konštanta v nestatickom kontexte
          X.pocetInstancii + // statická premenná v nestatickom kontexte
          X.rest();          // statická metóda v nestatickom kontexte
```

Statické vs. nestatické




```
StaticVsClass X = new StaticVsClass(); // objekt triedy StaticVsClass
StaticVsClass Y = new StaticVsClass(); // objekt triedy StaticVsClass

System.out.println(X.getIndex());                    // 1
System.out.println(Y.getIndex());                    // 2

System.out.println(StaticVsClass.pocetInstancii);    // 2
System.out.println(X.pocetInstancii);                // 2
System.out.println(Y.pocetInstancii);                // 2

X.pocetInstancii = 17;
StaticVsClass.pocetInstancii = 13;
System.out.println(StaticVsClass.pocetInstancii);    // 13
System.out.println(X.pocetInstancii);                // 13
System.out.println(Y.pocetInstancii);                // 13
```

Súbor: StaticVsClass.java

Singleton návrhový vzor

```
public class Singleton {
    // tento konštruktor sa nedá zavolať zvonku, lebo je private. Načo teda je ?
private Singleton() { }                    // navyše nič moc nerobí...
    // môžeme ho zavolať v rámci triedy a vytvoríme tak jedinú inštanciu objektu
private static Singleton instance = new Singleton();

public static Singleton getInstance() {// vráť jedinú inštanciu
   return instance;
}
public String toString() { return "som jediny-jedinecny"; }
}
        public static void main(String[] args) {
        // v inej triede nejde zavolať Singleton object = new Singleton();
                Singleton object = Singleton.getInstance();
                System.out.println(object);
        }
```