

Triedy a objekty



Peter Borovanský
KAI, I-18

borovan 'at' ii.fmph.uniba.sk

<http://dai.fmph.uniba.sk/courses/JAVA/>



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
- abstraktná 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) (<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/>,

Status quo

CV02

- 69 riešení, 27 plné 2 body, priemer 1.66 ☺
- body za hru boli zapísané (2x)

```
public static boolean podobne(String[] a, Integer[] b) {
    if (a == null && b == null) return true;
    if (a == null || b == null) return false;
    if (a.length != b.length) return false;

    for (int i = 0; i < a.length; i++) ...
```

DÚ01

- ?? riešení, ?? plné ? body, priemer ?? ☺

```
public static boolean podobne2(String[][] a, Integer[][] b) {
    if (a == null && b == null) return true;
    if (a == null || b == null) return false;
    if (a.length != b.length) return false;

    for (int i = 0; i < a.length; i++) ...
```

```
// generická statická metóda – out of scope
public static <T> boolean obeNull(T[] a, T[] b) {
    return a == null && b == null;
}

// generická statická metóda – out of scope
public static <T> boolean roznePolia(T[] a, T[] b) {
    if (a == null && b == null) return false;
    if (a == null && b != null) return true;
    if (a != null && b == null) return true;
    return a.length != b.length;
}
```

```
public static boolean podobne(String[] a, Integer[] b) {
    if (obeNull(a,b)) return true;
    if (roznePolia(a,b)) return false;

    for (int i = 0; i < a.length; i++) ...
```

```
public static boolean podobne2(String[][] a, Integer[][] b) {
    if (obeNull(a,b)) return true;
    if (roznePolia(a,b)) return false;

    for (int i = 0; i < a.length; i++) ...
```

CV02 – Podobné

pomocou generickej statickej metódy

```
// generická statická metóda
public static <T> boolean obeNull(T[] a, T[] b) {
    return a == null && b == null;
}
// generická statická metóda
public static <T> boolean roznePolia(T[] a, T[] b) {
    if (a == null && b == null) return false;
    if (a == null && b != null) return true;
    if (a != null && b == null) return true;
    return a.length != b.length;
}
```

```
public static boolean podobne2(String[][] a, Integer[][] b) {
    if (obeNull(a,b)) return true;
    if (roznePolia(a,b)) return false;
    for (int i = 0; i < a.length; i++) {
        if (obeNull(a[i],b[i])) return true;
        if (roznePolia(a[i],b[i])) return false;
        for (int j = 0; j < a[i].length; j++)
            if (!compare(a[i][j], b[i][j]))
                return false;
    }
    return true;
}
```

Krátkodobé ciele



Daily Coding Problem

Good morning! Here's your coding interview problem for today.

This problem was asked by Google.

Explain the difference between composition and inheritance. In which cases would you use each?

dnes:

- trieda a jej inštancia vs. data-class (record),
- konštruktor, getter/setter
- rozdiel medzi statickou a triednou metódou
- virtuálna metóda v C++ a Java
- abstraktná trieda vs. interface



Statické metódy

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

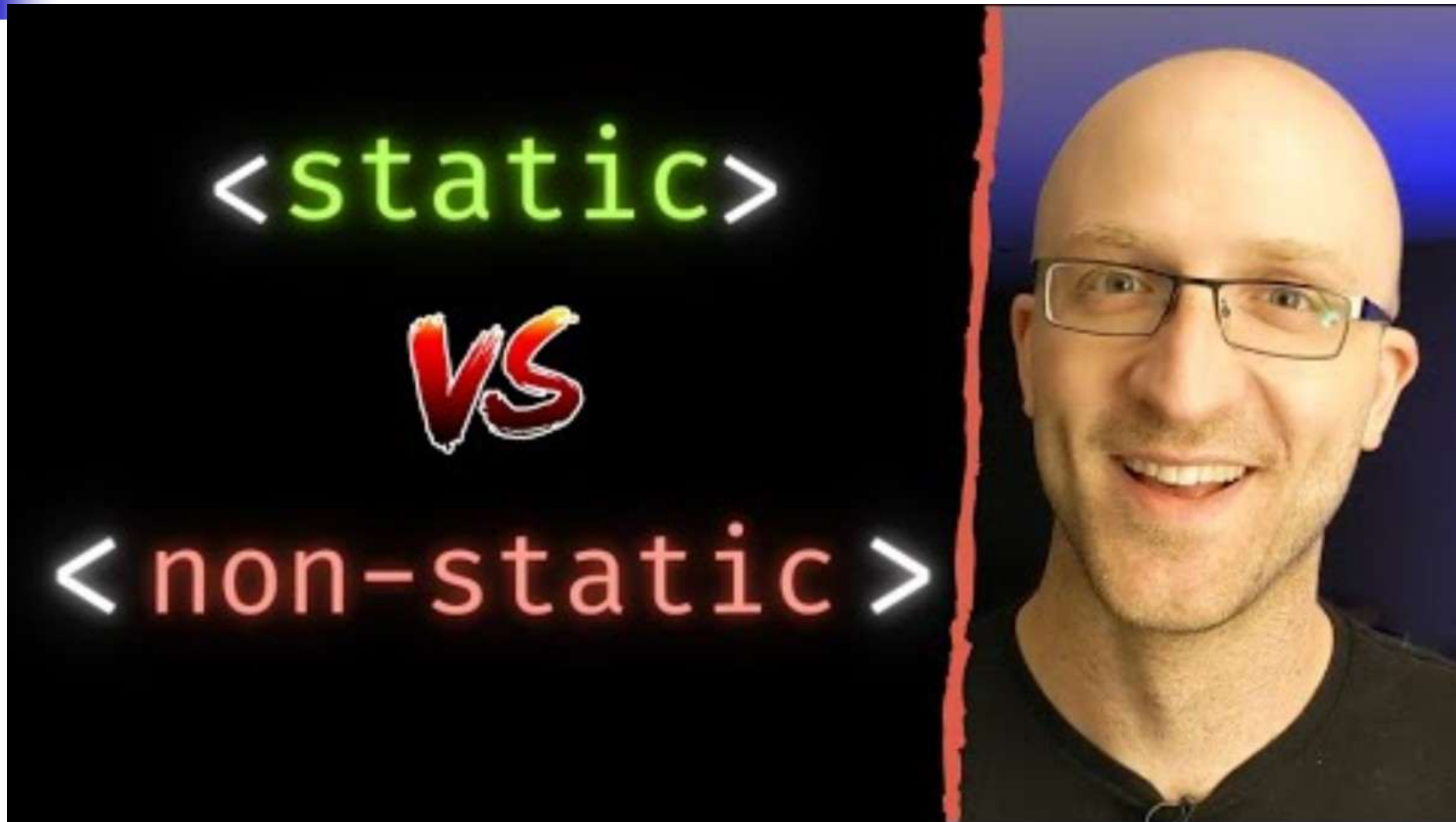
Statické metódy:

- 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*, *System.out.println(5)*
- ak aj metóda nemá argumenty, prázdne zátvorky sa do jej definície a do volania aj tak píšú (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**

Static vs. non-static



<https://www.youtube.com/watch?v=-Y67pdWHr9Y>



Alan Kay

(OOP)

The best way to predict the future is
to invent it.

Alan Kay



1. Everything is an object
2. Objects communicate by sending and receiving messages (in terms of objects)
3. Objects have their own memory (state)
4. Every object is an instance of a class
5. The class holds the shared behavior for its instances

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

6. To eval a program list, control is passed to the first object and the remainder is treated as its message.



OOP pojmy

- všetko je objekt
 - každý objekt má typ
 - každý objekt má svoj kus pamäte
 - program je hĺba 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

Malo svoje krízy, z ktorých sa liečilo

- no goto statement
- štruktúrované programovanie
- modulárne programovanie
zárodok enkapsulácie

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

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
```

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 {
    private double real, imag;
```

```
// definícia triedy
// triedne premenné
// nie sú static
// konštruktor
```

```
    public Complex(double real, double imag) {
        this.real = real; this.imag = imag;
    }
```

musí byť public?

Nech je skryté, čo
má ostať skryté ...

Vlastnosti - properties

- K premenným reprezentujúcim stav objektu prístupujeme 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(  
        c1.getReal()*c1.getReal() + // použitie mimo triedy  
        c1.getImag()*c1.getImag())); // výpočet dĺžky k.čísla  
}
```

IntelliJ: ALT-Insert/ALT-Enter

Nechajte si vygenerovať

- konštruktory a
- get/set metódy
- toString()

```
public class Complex {  
    private double real, imag;  
}
```





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+"]";
    }
}
```

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 **getter** a **setter** - pre každú privátnu hodnotu - property *Prop* typu *typ*, disponuje metódami
`public typ getProp()` – vráti hodnotu `Prop : typ`, a
`public void setProp(typ x)` – nastaví hodnotu `Prop` na `x : typ`,
napr. `Complex.getReal() : Real`, alebo `Complex.setImag(x : Real)`
- a pre logické hodnoty poskytuje `public boolean isProp()`
- 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() {  
        return Math.sqrt(real*real + imag*imag);  
    }  
    public void add(Complex c) {  
        real += c.real;  
        imag += c.imag;  
    }  
    public void mult(Complex c) {  
        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]
```

```
// veľkosť vektora komp.čísła
```

```
// súčet komplexných čísel
```

```
// súčin komplexných čísel
```

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)  
}
```

Preťažovanie metód

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

```
public class Complex {  
    private double real, imag;  
    ....  
    public void mult(Complex c) { ... vid' 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) { ... vid' slide this-2 }  
public Complex mult(Complex c) { ... } // rozdiel proc/func tiež nestačí
```

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

Preťažovanie operátorov

Preťažovanie vs. pretypovanie

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

- `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,
- $3.0 + (\text{double})7$
 - $(\text{double})3 + 7.0$

IntelliJ JDK15

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Haskell Help TestJava15 - Complex.java

TestJava15 src Complex.java

- Project
- TestJava15 D:\TestJava15
- External Libraries
 - < 15 > C:\Program Files\Java\jdk-15.0.2
 - java.base library root
 - java.compiler library root
 - java.datatransfer library root
 - java.desktop library root
 - java.instrument library root
 - java.logging library root
 - java.management library root
 - java.management.rmi library root
 - java.naming library root
 - java.net.http library root
 - java.prefs library root
 - java.rmi library root
 - java.scripting library root
 - java.se library root
 - java.security.jgss library root
 - java.security.sasl library root
 - java.smartcardio library root
 - java.sql library root
 - java.sql.rowset library root
 - java.transaction.xa library root
 - java.xml library root
 - java.xml.crypto library root
 - jdk.accessibility library root
 - jdk.aot library root
 - jdk.attach library root

Project Structure

Project Settings

Project

Modules

Libraries

Facets

Artifacts

Platform Settings

SDKs

Global Libraries

Problems

Project name:

TestJava15

Project SDK:

This SDK is default for all project modules.

A module specific SDK can be configured for each of the modules as required.

15 java version "15.0.2"

Edit

Project language level:

This language level is default for all project modules.

A module specific language level can be configured for each of the modules as required.

15 (Preview) - Sealed types, records, patterns, local enums and interfaces

Project compiler output:

This path is used to store all project compilation results.

A directory corresponding to each module is created under this path.

This directory contains two subdirectories: Production and Test for production code and test sources, respectively.

A module specific compiler output path can be configured for each of the modules as required.

D:\TestJava15\out

Experimental Feature Alert



You must accept the terms of legal notice of the beta Java specification to enable support for "15 (Preview) - Sealed types, records, patterns, local enums and interfaces".

The implementation of an early-draft specification developed under the Java Community Process (JCP) is made available for testing and evaluation purposes only and is not compatible with any specification of the JCP.

Accept

Decline

Data class/record v JDK14+

definujete tzv. data-class, klíč. Slovo record:
`record Complex(double real, double imag) { }`

potom už môžete používať plný komfort:

```
public class Main {  
    public static void main(String[] args) {  
        Complex c = new Complex(1, 0);  
        Complex c1 = new Complex(1, 0);  
        if (c.real() > c.imag()) {  
            System.out.println(c);  
        }  
        if (c.equals(c1)) {  
            System.out.println("equals");  
        }  
    }  
}
```

skryte sa vygeneruje konštruktor,
getter, equals, hashCode, toString, ...:

```
public final class Complex {  
    private double real;  
    private double imag;  
    public Complex(double real,  
                   double imag) {  
        this.real = real; this.imag = imag; }  
  
    double real() { return this.real; }  
    double imag() { return this.imag; }  
  
    public Boolean equals() ...  
    public int hashCode()...  
    public String toString() {...}  
}
```

Súbor: `Complex_record.java`

Data class/record v JDK14+

```
record Complx_record(double real, double imag) {  
    Complx_record() { ... } // tzv. kompaktny konstuktor  
  
    // tzv. kanonicky konstruktor, a moze byt jeden, nie oba konstruktory  
    Complx_record(double real, double imag) { this.real = real; this.imag = imag; }  
  
    //data class/record moze mat staticku premenu  
    static int counter = 0;  
    //data class/record moze mat staticku metodu  
    static int getCounter() { return counter; }  
    // aj staticku sekciu  
    static {  
        int pocitadlo = 0;  
    }  
    //data class/record NEmoze mat triednu premennu, jedine hodnoty v objekte su z konstruktora  
    //int index = 0;  
  
    //data class/record napriek tomu moze mat triednu metodu  
    double abs() { return Math.sqrt(real*real + imag*imag); }  
}
```

Data class/record v JDK14+

```
record Dvojica(Integer first, Integer second) {  
  
    static int pocetInstancii = 0;  
  
    public Dvojica {  
        pocetInstancii++;  
    }  
  
    public static void main(String[] args) {  
        var x = new Dvojica(1,2);  
        System.out.println(x);                               // Dvojica[first=1, second=2]  
        // x.second *= 2*x.first;  
        var y = new Dvojica(x.first, 2*x.first);  
        System.out.println(x.equals(y));                     // true  
        System.out.println(x == y);                          // false  
        System.out.println("pocet instancii " + pocetInstancii); // pocet instancii 2  
    }  
}
```

Data class/record v JDK14+

```
record Pair<A,B>(A first, B second) {  
  
    static int pocetInstancii = 0;  
  
    public Pair {  
        pocetInstancii++;  
    }  
  
    public static void main(String[] args) {  
        var x = new Pair<>(1,2);  
        System.out.println(x);           // Pair[first=1, second=2]  
        // x.second *= 2*x.first;  
        var y = new Pair<>(x.first, 2*x.first);  
        System.out.println(x.equals(y)); // true  
        System.out.println(x == y);     // false  
        var z = new Pair<>(Math.PI, Math.E);  
        System.out.println(z);           // Pair[first=3.14..., second=2.71..]  
        var w = new Pair<>(y,z);  
        System.out.println(w);           // Pair[first=Pair[first=1, second=2],  
                                           second=Pair[first=3.14, second=2.71]]  
        System.out.println("pocet instancii " + pocetInstancii); // 4  
    }  
}
```

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 😊
 - `record/data` classes existujú JDK14+
 - `record Complex(double real, double imag) {}`
 - <https://cr.openjdk.java.net/~briangoetz/amber/datum.html>
- `class Complex{...}`
`Complex c; // deklaruje referenciu`
`c=new Complex(); // vytvorí sa až tu`
- `Complex cc = c; // nekopíruje, ale`
`Complex cc = c.clone(); // kopíruje`
- neexistuje rozdiel medzi objektom a pointrom (referenciou), preto k položkám a metódam objektu vždy prístupujeme pomocou ``.``



Konštruktory nadtriedy

```
package SuperAndSub;
```

konštruktory triedy môžu byť preťažené

```
public class Nadtrieda {
```

```
    public Nadtrieda() {
```

```
        System.out.println("Konštruktor nadtriedy");
```

```
    }
```

```
    public Nadtrieda(int n) {
```

```
        System.out.println("Konštruktor nadtriedy n="+n);
```

```
    }
```

```
    public Nadtrieda(String s) {
```

```
        System.out.println("Konštruktor nadtriedy s="+s);
```

```
    }
```

```
    public void foo() {
```

```
        System.out.println("Nicnerobiaca funkcia foo v nadtriede");
```

```
    }
```

```
}
```

Konštruktory podtriedy

super. verzus super()

```
package SuperAndSub;
```

```
public class Podtrieda extends Nadtrieda{
    public Podtrieda() {
        System.out.println("Konštruktor podtriedy");
    }
    public Podtrieda(int n) {
        System.out.println("Iny konštruktor podtriedy n="+n);
    }
    public Podtrieda(String s) {
        super(s+s);
        System.out.println("Konštruktor 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 konštruktoru musí byť 1.príkaz

// volanie foo z nadtriedy

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



Deštruktory

Deštruktory sú Java 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 „smeť“
    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

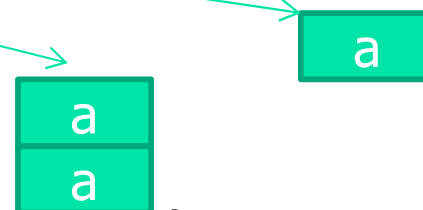
- má v JAVE syntax: `[public] class Podtrieda extends Nadtrieda { ... }`
- podtrieda obsahuje všetky premenné, konštanty a metódy nadtriedy,
- na predefinovanie metódy v podtriede nikde nepíšeme `override`,
- predefinovať môžeme každú metódu, všetko je `virtual`,

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

```
public 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());  
}
```

```
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(); }  
}
```

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...
```

pes

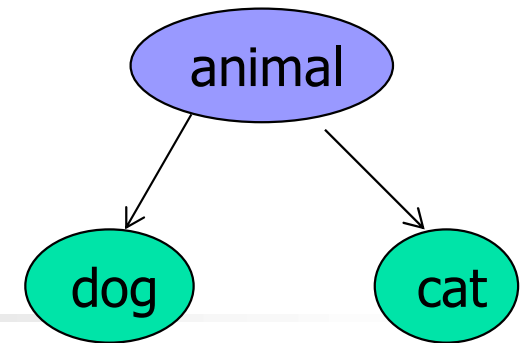
macka

```
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 Java



```
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();
```

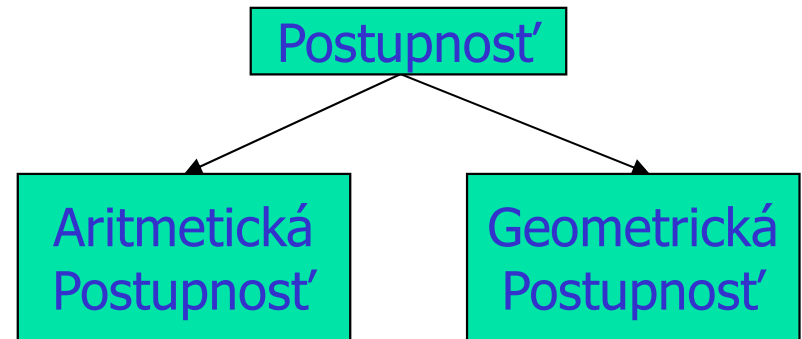
→ haw-haw
→ mnaw-mnaw

```
for(Animal a:animals){
    if (a instanceof Dog)
        System.out.println("it's a dog");
    else
        System.out.println("not a dog");
}
```

→ it's a dog
→ not a dog

```
}
```

Postupnosť



```
abstract class Postupnosť {
```

```
protected long prvy;  
protected long aktualny;
```

```
public long Prvy() {  
    aktualny = prvy;  
    return aktualny;  
}
```

```
abstract long Dalsi();
```

```
public void printPostupnosť(int n) {  
    System.out.print(Prvy());  
    for(int i= 0; i<n; i++)  
        System.out.print(", "+ Dalsi());  
    System.out.println();  
}
```

```
// abstraktná trieda má abstraktnú  
// metódu, t.j. nemá inštancie  
// prvý prvok postupnosti  
// aktuálny prvok postupnosti
```

```
// skoč na prvý prvok
```

```
// daj mi ďalší prvok
```

```
// vytlač postupnosť
```

```
// volá sa nejaká ešte  
// neznáma metóda
```

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; prvý = 0;  
    }  
  
    AritmetickaPostupnosť(int _prvý, int _delta) { // ďalší konštruktor  
        delta = _delta; prvý = _prvý; // preťaženie  
    }  
  
    public long Dalsi() { // konkretizácia abstraktnej metódy  
        aktualny += delta;  
        return aktualny;  
    }  
}
```



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;  
    }  
}
```



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;
```

```
    // podiel' 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;
```

```
    }
```

```
}
```

Fibonacciho postupnost'

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

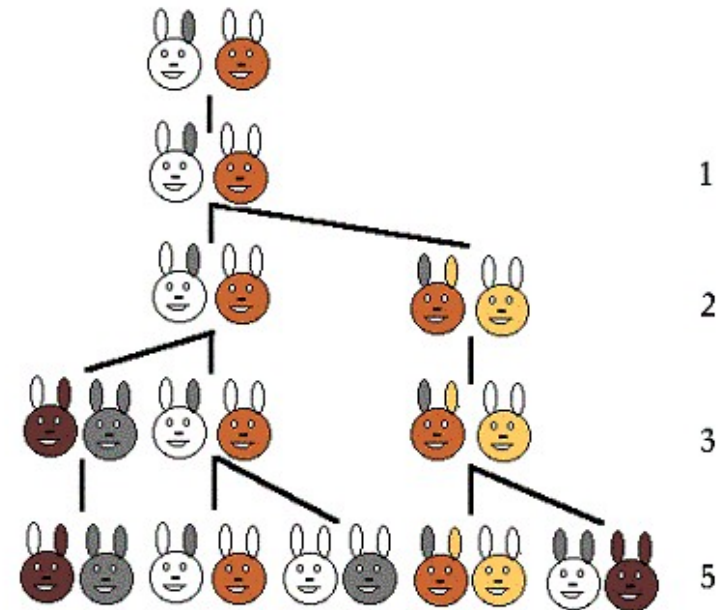
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

```
public class FibonaccihoPostupnost extends Postupnost {
```

```
    protected long predch;
```

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

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



Abstrakcia

výraz-operátor/konštantá

```
abstract class Expression {  
    abstract long value();  
}
```

```
public class Constant extends Expression {  
    long constant;  
  
    public Constant(long constant) {  
        this.constant = constant;  
    }  
  
    public long value() {  
        return constant;  
    }  
  
    @Override  
    public String toString() {  
        return ""+constant;  
    }  
}
```

```
public class Operator extends Expression {  
    private char op;  
    private Expression left, right;  
    public Operator(char op, Expression left, Expression right) {  
        this.op = op; this.left = left; this.right = right;  
    }  
    @Override  
    long value() {  
        long leftvalue = (left==null)?0:left.value();  
        long rightvalue = (right==null)?0:right.value();  
        switch (op) {  
            case '+': return leftvalue + rightvalue;  
            case '*': return leftvalue * rightvalue;  
            default: return 0;  
        }  
    }  
    @Override  
    public String toString() {  
        String leftString = (left==null)?"":left.toString();  
        String rightString = (right==null)?"":right.toString();  
        if (op == '*') {  
            if (left instanceof Operator && ((Operator) left).op == '+')  
                leftString = "(" + leftString + ")";  
            if (right instanceof Operator && ((Operator) right).op == '+')  
                rightString = "(" + rightString + ")";  
        }  
        return leftString + op + rightString;  
    }  
}
```

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 prepísať len public,
private môže prepísať len private,
etc.

- (skoro) každá nestatická metóda môže byť predefinovaná bez syntaktického 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 spíňať/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;
    }
}
```

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 inštalácie, 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;**

Abstract vs. Interface

(rekapitulácia – tentokrát už v Java)

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;  
protected double real, imag;  
void foo() { }  
private int goo() { }
```

```
// deklarácia viditelnej konštanty  
// lokálne premenné  
// metódu vidno len v balíčku  
// najreštriktívnejšie-fciu je len v triede
```

Deklarácia triedy

(rekapitulácia syntaxe)

```
class MenoTriedy
```

```
    TeloTriedy
```

```
}
```

- [public]

- [abstract]

- [final]

- [extends *supertrieda*] trieda je podtriedou inej triedy, dedičnosť

- [implements Interfaces{,}*] Interfaces sú implementované v tejto triede

```
{// MenoTriedy.java
```



trieda je voľne prístupná, inak je prístupná len v danom package

trieda nemôže byť inštanciovaná (asi obsahuje abstr.metódu) t.j. neexistuje objekt danej triedy

trieda nemôže mať podtriedy, „potomkov“

trieda je podtriedou inej triedy, dedičnosť

Interfaces sú implementované v tejto triede

Deklarácia metódy

(rekapitulácia)

```
typ MenoMetódy(argumenty) {  
    telo metódy  
}
```

- **[static]**
- **[abstract]**
- **[final]**
- **[native]**
- **[synchronized]**
- **[throws]** exceptions

triedna metóda, existuje nezávisle od objektov triedy
metóda, ktorá nie je implementovaná, bude v podtriede
metóda, ktorá nemôže byť predefinovaná, bezpečnosť
metóda definovaná v inom jazyku, „prilinkovaná“
metóda synchronizujúca konkurentný prístup
bežiacich threadov, neskôr...
metóda produkujúca výnimky

```
Access Level | Method Name  
|            |  
public Object push(Object item)  
|           |  
Return Type | Arguments
```



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;  
    }  
}
```

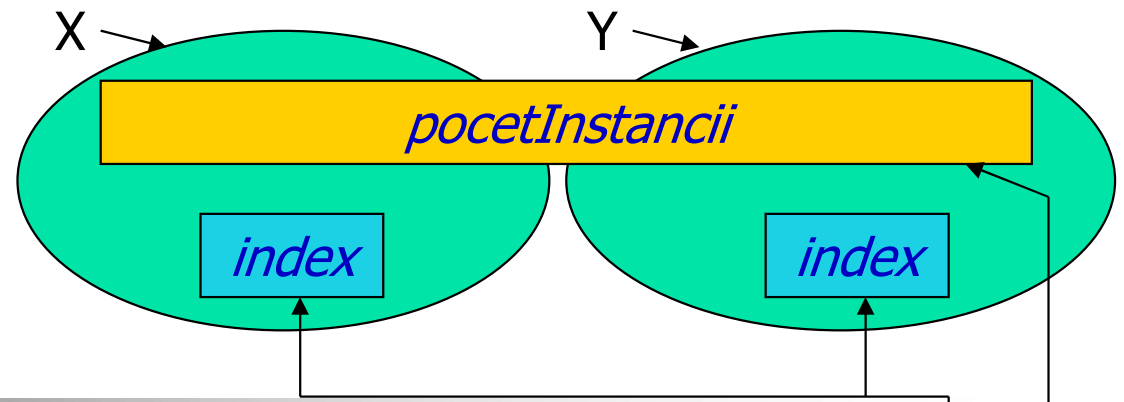
Statické verzus nestatické

```
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
```



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);
    }
```