

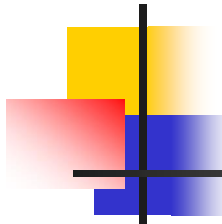
# Triedy a objekty

(na príkladoch)



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
- statické vs. triedne metódy, premenné

cvičenia:

- vytvoriť prvú triedu/objekt

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/>,



# 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 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**



# 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

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
- ukrývanie (encapsulation) – public/private/protected/ ... toto nebude dnes
- preťažovanie (overloading) vs. polymorfizmus metód
- dedičnosť – podtrieda a nadtrieda, viacnásobné dedenie
- virtuálne metódy a dynamic binding (v C++)
- 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().

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

# Prvý konštruktor

- konštruktor je metóda s menom zhodným s menom triedy, bez výstupného typu,
- konštruktor je najčastejšie je public. Môže byť private ? (prémia 1b),
- 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;  
  
    public Complex(double real, double imag) {  
        this.real = real; this.imag = imag;  
    }  
}
```

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

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  
  
    public double getReal() { return real; }           // properties  
    public void setReal(double _real) { real = _real; } // getter  
                                                    // setter  
  
    public double getImag() { return imag; }           // getter  
    public void setImag(double imag) { this.imag = imag; } // setter  
                                                    // setter  
  
    System.out.println(Math.sqrt(  
        c1.getReal()*c1.getReal() +  
        c1.getImag()*c1.getImag())); // použitie mimo triedy  
                                     // výpočet dĺžky k.čísla
```

Súbor: Complex.java

# Java Beans



- jazyk JAVA nemá v sebe podporu pre tzv. properties, ako ich poznáme z Delphi (Lazarus), C++ Builder, ...
- nakoniec, ani Pascal ich nikdy nemal, kým neprišiel Borland...

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

- s defaultným konštruktorom bez argumentov, t.j. napr. `Complex()`
- 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.čísla

// súčet komplexných čísel

// súčin komplexných čísel



# Pret'azovanie konštruktorov

Pret'azovanie vie kompilátor  
rozhodnúť pred spustením  
programu, zo syntaxe.  
Pret'azovanie a virtual nesúvisia

Pret'aziť 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**



# Pret'azovanie metód

Pret'azená 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 pret'azenia:

```
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 preťaženie sú ...

# Pretážovanie operátorov


## Pretážovanie vs. pretypovanie

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

- $\text{int} + \text{int}$
  - $\text{double} + \text{int}$
  - $\text{int} + \text{double}$
  - $\text{double} + \text{double}$
- 4 prekrývajúce sa operátory,  
žiadne pretypovanie len  
preťaženie

- $\text{double} + \text{double}$   
žiadne preťažovanie len  
pretypovanie

- $\text{int} + \text{int}$
  - $\text{double} + \text{double}$
- 2 preťažené operátory,
- $3.0 + (\text{double})7$
  - $(\text{double})3 + 7.0$



# 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 rozdiel medzi objektom a pointrom (referenciou), preto k položkám a metódam objektu vždy pristupujeme 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. verus 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();  
    }  
}
```

konštruktory podtriedy najprv zavolá:  
implicitný (bez arg.) konštruktory nadtriedy,  
explicitne niektorý z konštruktory  
pomocou super(...)  
// volanie konštruktory musí byť 1.príkaz  
// volanie foo z nadtriedy

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

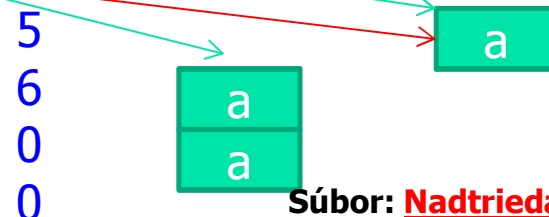
# Dedenie

- 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 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(); }  
}
```



Súbor: Nadtrieda.java, Podtrieda.java

```
class FarebnyBod(Bod) :
```



# 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.

# 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

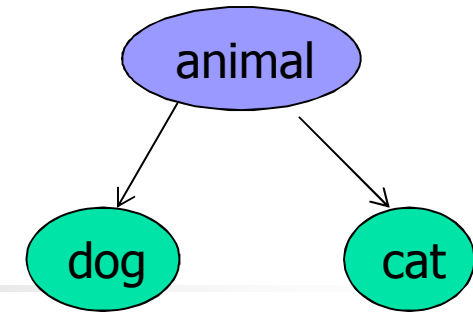
macka

```
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 Jave



```
abstract class Animal { // nikdy nemôem vytvori objekt triedy Animal
    abstract void sound(); // teda zavola 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");
```

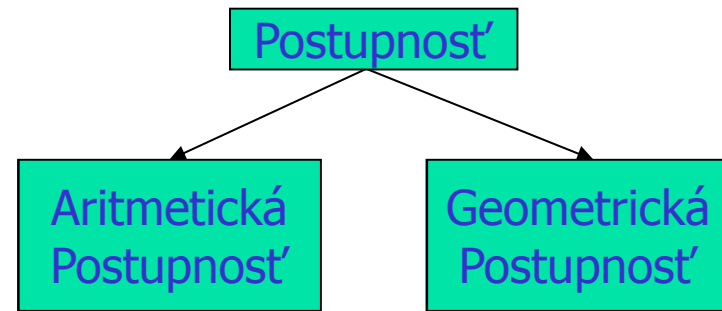
it's a dog  
not a dog

```
    else
```

```
        System.out.println("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 inšancie  
// 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; prvvy = 0;  
    }  
  
    AritmetickaPostupnost(int _prvy, int _delta) { // ďalší konštruktor  
        delta = _delta; prvvy = _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; prvý = 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);
```

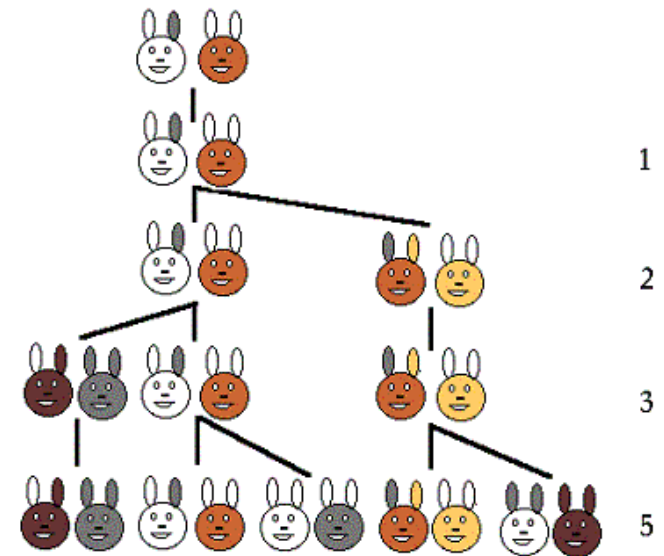
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89

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

```
    protected long predch;
```

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

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



Súbor: [FibonaccihoPostupnost.java](#)



# Dedičstvo C++ vs. JAVA

- dedenie

class TPodtrieda:public TNadtrieda{};

- dedenie

class Podtrieda extends Nadtrieda {}

- ukrývanie premenných a metód v triede je podobne ako JAVA

- public/private/protected/ *nič*  
nič zodpovedá friendly

- ukrývanie pri dedení  
public/private/protected dedenie  
"ťažšie témy"

- zjednodušené len jedno dedenie:  
public môže prepísať len public,  
private môže prepísať len private,  
etc.

- virtuálne metódy

- (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 spĺňať/implementovať viacero interface
- má automatickú správu pamäte a deštruktory píšeme zriedka

# Deklarácia triedy

(rekapitulácia syntaxe)

```
class MenoTriedy  
    TeloTriedy  
}
```

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

- **[public]**

- **[abstract]**

- **[final]**

- **[extends *supertrieda*]**

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

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

Class Declaration

```
public class Stack {
```

Variable

```
    private Object items;
```

Constructor

```
    public Stack() {  
        items = new Object(10);  
    }
```

Methods

```
    public Object push(Object item) {  
        items.addElement(item);  
        return item;  
    }  
  
    public synchronized Object pop() {  
        int len = items.size();  
        Object obj = null;  
        if (len == 0)  
            throw new EmptyStackException();  
        obj = items.elementAt(len - 1);  
        items.removeElementAt(len - 1);  
        return obj;  
    }  
  
    public boolean isEmpty() {  
        if (items.size() == 0)  
            return true;  
        else  
            return false;  
    }  
}
```

# 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

---

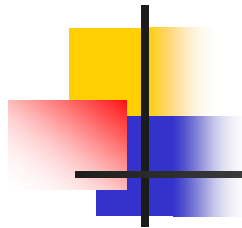
v procedurálnom prístupe sme si zvykli definovať všetky metódy ako statické a nazývali sme ich procedúry a funkcie,

- 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;  
    }  
}
```





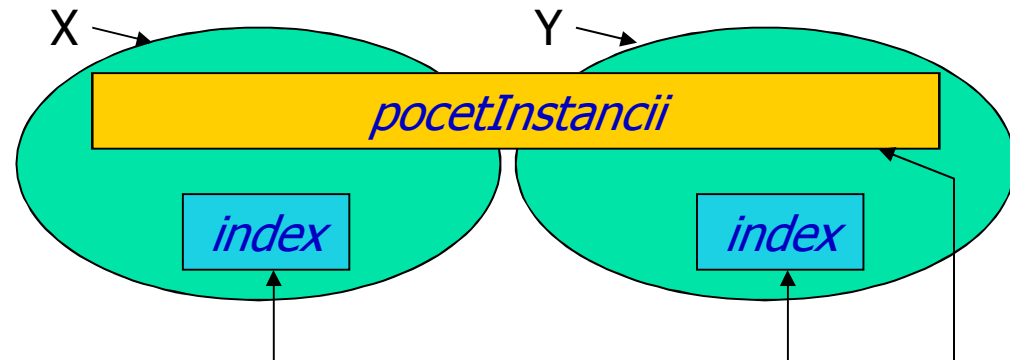
# Statické verzus nestatické

```
public static void main(String args[]) { // statický kontext
    int a = MAX + // referencia statickej premennej
        StaticVsClass.MAX + // úplná referencia Trieda.var
        StaticVsClass.rest(); // referencia statickej metódy
    // ... 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 konztruktor sa nedá zavola zvonku, lebo je private. Na o teda je ?  
    private Singleton() { } // navyze ni moc nerobí...  
    // môžeme ho zavola v rámci triedy a vytvoríme tak jedinú inztanciu objektu  
    private static Singleton instance = new Singleton();  
  
    public static Singleton getInstance() { // vrá jedinú inztanciu  
        return instance;  
    }  
    public String toString() { return "som jediny-jedinecny"; }  
}  
    public static void main(String[] args) {  
        // v inej triede nejde zavola Singleton object = new Singleton();  
        Singleton object = Singleton.getInstance();  
        System.out.println(object);  
    }
```