

# Triedy a objekty

(voľné pokračovanie)



Peter Borovanský  
KAI, I-18

borovan 'at' ii.fmph.uniba.sk

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

Ladění je dvakrát těžší než psaní kódu.  
Takže když napíšete kód dle svých  
nejlepších znalostí, pak z definice nejste  
dost schopní na to, abyste jej odladili.  
-- Brian W. Kernighan (autor jazyka C)



## Zrnká múdrosti

---

Vždy jsem si přál, aby používání mého počítače bylo  
tak snadné jako používání mého telefonu.  
Přání se mi splnilo – už nechápu, jak používat telefon.  
-- Bjarne Stroustrup (autor C++)



# Hádanka

## z dielne majstrov

---

Vždy pište kód tak, jako by ten chlapík, co ho po vás bude udržovat, měl být násilnický psychopat, který bude vědět, kde bydlíte.

Čo robí tento príkaz ?

```
cislo = Integer.valueOf(String.valueOf(cislo) + Integer.valueOf(String.valueOf(0)));
```

```
cislo = 10*cislo;
```

```
cislo *= 10;
```



# Pozoruhodný kód

## z dielne majstrov

---

```
if (k > 9)
    return -1;
int[][] results = new int[][] {
    {19, 29, 12, 13, 14, 15, 16, 17, 18},
    {127, 128, 129, 139, 149, 123, 124, 125, 126},
    {1234, 1235, 1236, 1237, 1238, 1239, 1249, 1259, 1269},
    {12349, 12359, 12369, 12379, 12389, 12345, 12346, 12347, 12348},
    {123589, 123689, 123456, 123457, 123458, 123459, 123469, 123479, 123489},
    {1234567, 1234568, 1234569, 1234579, 1234589, 1234689, 1234789, 1235789, 1236789},
    {1234579, 12345689, 12345789, 12346789, 12356789, 12456789, 13456789, 23456789, 12345678}
};
if (k == 1)
    return b;
if (k == 9) {
    if (b == 9)
        return 123456789;
    return -1;
}
return results[k - 2][b - 1];
```

pre k a b nájdite najmenšie číslo, že ...

nie je zlé, ak ho nájdete ručne, ale vtedy sú vlastne vaše myšlienky súčasťou riešenia, a skúste ich preto popísať v README.TXT

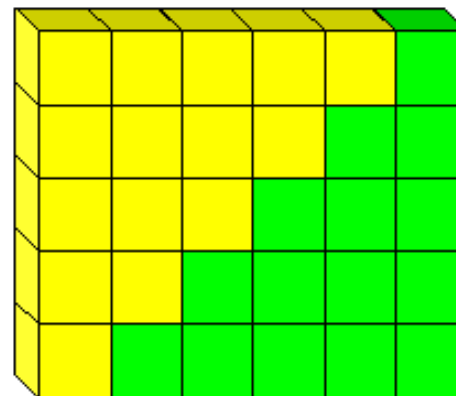
# Školníkové skrinky

z dielne majstrov

- ukazuje sa, že nepárny počet deliteľov majú len štvorce čísel  
1, 4, 9, 16, 25, ...
- málokto mal potrebu to dokázať...
- a vlastne úlohou bolo sčítať  $1^2 + 2^2 + 3^2 + \dots + n^2$
- málokto našiel/použil/pozná vzorec  $n(n+1)(2n+1)/6$

Skúsme jednoduchšie:

- dôkaz, že:  
$$2[1+2+ \dots + n] = n(n+1)$$

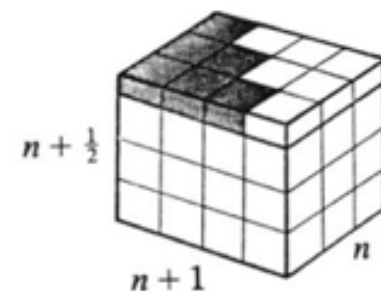
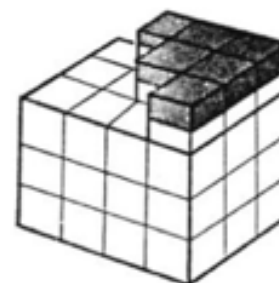
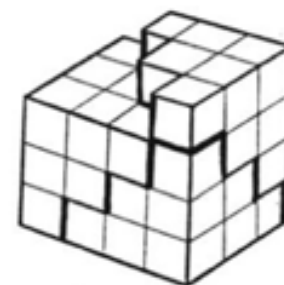
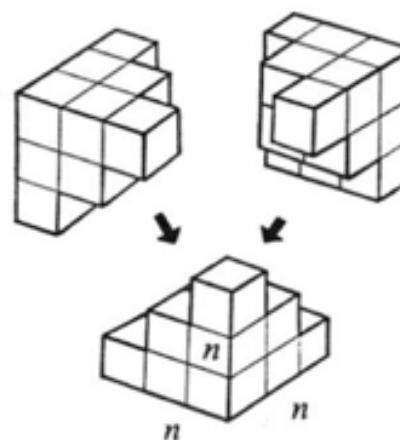
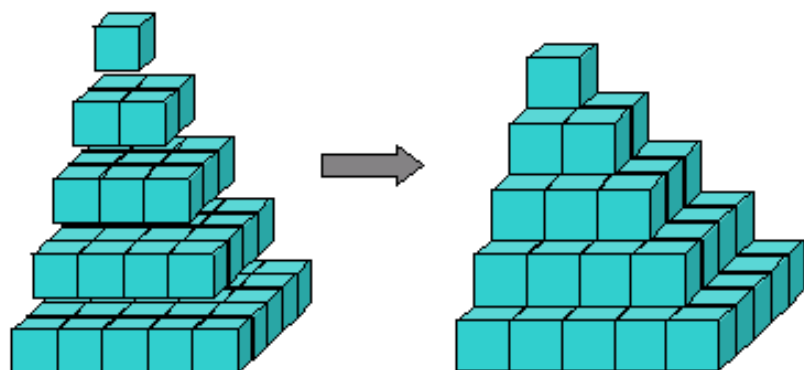


# Školníkové skrinky

z dielne majstrov

- dôkaz, že

$$6[1^2 + 2^2 + 3^2 + \dots + n^2] = n(n+1)(2n+1)$$

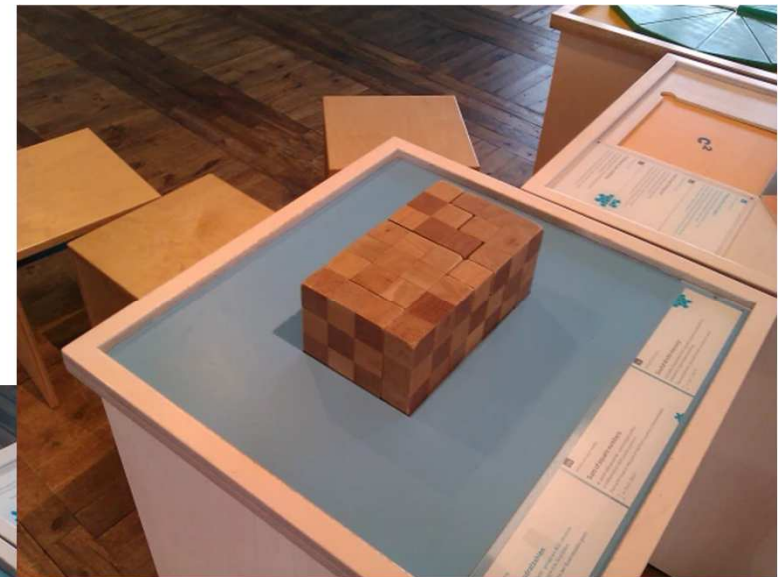
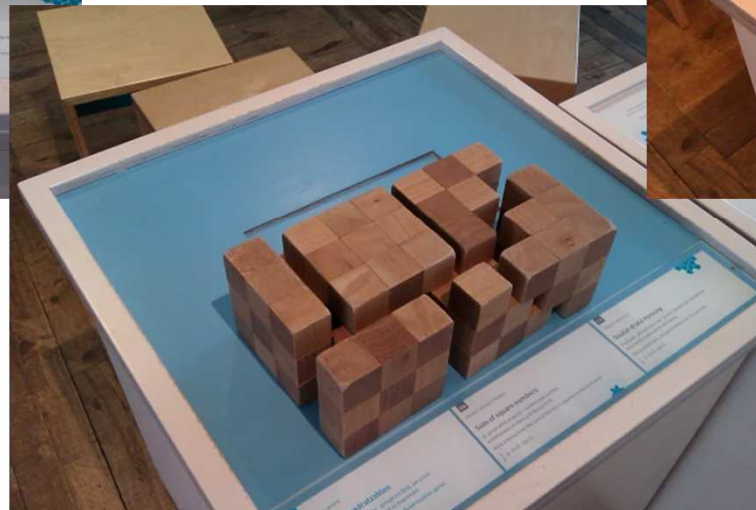
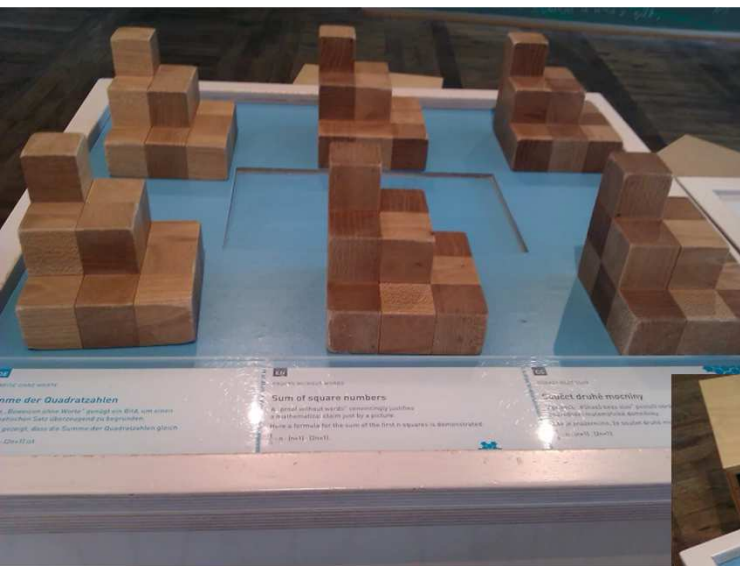


# Školníkové skrinky

z dielne majstrov

- dôkaz, že

$$6[1^2 + 2^2 + 3^2 + \dots + n^2] = n(n+1)(2n+1)$$

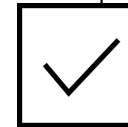


# Testy

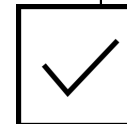
## z dielne majstrov

- CV03 – je bod v kruhu, guli ?

```
/**
 * definujte metodu, co zisti, ci bo b je v/na guli
 */
@Override
public boolean jeV(Bod3D b) {
    return false;
}
```



```
/**
 * definujte metodu, co zisti, ci bod b je v kruhu/na kruznici
 */
@Override
public boolean jeV(Bod2D b) {
    return false;
}
```



- nevedno, či je to zámer, alebo nepozornosť, ale ak nájdete chybu v teste a oznámite nám ju, tak na nej „zarobíte“ viac, ako keď ju obídete...



# BST

z dielne majstrov :)

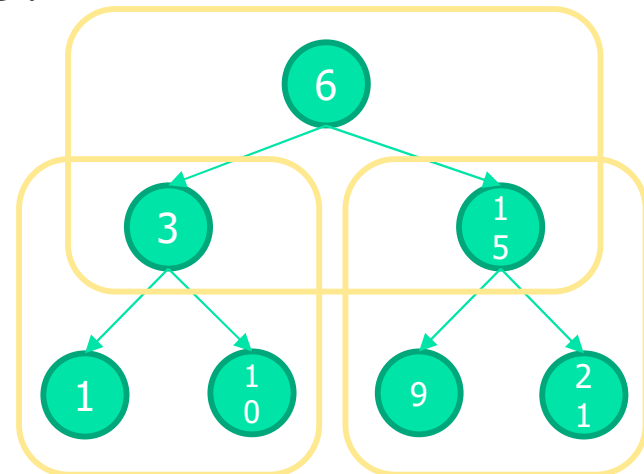
```
abstract class Tree {  
    abstract boolean isBST();  
    ...  
class Node extends Tree {  
    @Override  
    boolean isBST() {  
        if (left == null && right == null) return true;  
        if (left != null) {  
            if (data <= left.root()) return false;  
            if (!left.isBST()) return false;  
        }  
        if (right != null) {  
            if (data >= right.root()) return false;  
            if (!right.isBST()) return false;  
        }  
        return true;  
    } }  
}
```

```
class Leaf extends Tree {  
    @Override  
    boolean isBST() {  
        return true;  
    } }  
}
```

```
Tree c3 = new Node(new Leaf(1), 3, new Leaf(10));  
Tree c4 = new Node(new Leaf(9), 15, new Leaf(21));  
Tree c5 = new Node(c3, 6, c4);
```

```
@Override  
boolean isBST() {  
    return  
        (left == null || left.root() < data && left.isBST())  
        &&  
        (right == null || right.root() > data && right.isBST());  
}
```

Zlé riešenie, a vieme napísať fungujúce lineárne riešenie ?





# BST v štýle J17

## Sealed class/interface

---

Zapečatená (sealed) trieda/interface určuje, ktoré **jediné** podtriedy trieda môže mať, resp. ktoré triedy **jediné** implementujú zapečatený interface (**a žiadne iné**) – **idea zapečatenosti**

```
sealed interface Tree permits Node, Leaf {  
    boolean isBST();  
}
```

```
record Node(Tree left, int value, Tree right) implements Tree {  
    @Override  
    public boolean isBST() { // bez zmeny  
        return (left == null || left.root() < value && left.isBST())  
            && (right == null || right.root() > value && right.isBST());  
    }  
}
```

```
record Leaf(int value) implements Tree {  
    @Override  
    public boolean isBST() {  
        return true;  
    }  
}
```



# BST v štýle J17

## patterns

---

```
static int velkost1(Tree t) {  
    return  
        switch (t) {          // switch je príkaz !!!  
            case Node n ->  
                1 +  
                ((n.left() == null) ? 0:velkost1(n.left())) +  
                ((n.right() == null) ? 0:velkost1(n.right()));  
            case Leaf l -> 1;  
  
            // no default... lebo Tree je zapečatený  
  
        };  
}
```

```
// pred Java 17  
static int velkost2(Tree t) {  
    if (t instanceof Node) {  
        Node n = (Node) t;  
        return  
            1 +  
            ((n.left() == null)?0:velkost1(n.left())) +  
            ((n.right() == null)?0:velkost1(n.right()));  
    } else if (t instanceof Leaf) {  
        Leaf l = (Leaf)t;  
        return 1;  
    } else {  
        return 999;  
    }  
}
```



# Triedy a objekty

dnes bude:

- zhrnutie z minulej prednášky (abstrakcia a enkapsulácia)
- kompozícia objektov vs. dedenie
- nemeniteľná (immutable) trieda
- inkluzívny (triedny) polymorfizmus
- interface, typy a podtypy
- balíčkovanie - koncept package
- ukrývanie metódy/premennej: *private*, *protected*, *public* v package
- vnorené triedy

Niektoré dnešné príklady kódov v Jave nemajú hlbší (praktický) zmysel ako ilustrovať (niekedy až do absurdity) rôzne jazykové konštrukcie a princípy.

literatúra:

- Thinking in Java, 3rd Ed. (<http://www.ibiblio.org/pub/docs/books/eckel/TIJ-3rd-edition4.0.zip>)  
— 5: Hiding the Implementation, 7: Polymorphism
- Naučte se Javu – úvod
  - <http://interval.cz/clanky/naucte-se-javu-balicky/> ,
  - <http://interval.cz/clanky/naucte-se-javu-staticke-promenne-a-metody-balicky/>



# Dnešné 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?

# Deklarácia triedy

(rekapitulácia z minulej prednášky)

```
class MenoTriedy  
    TeloTriedy  
}
```

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

■ [public]

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

■ [abstract]

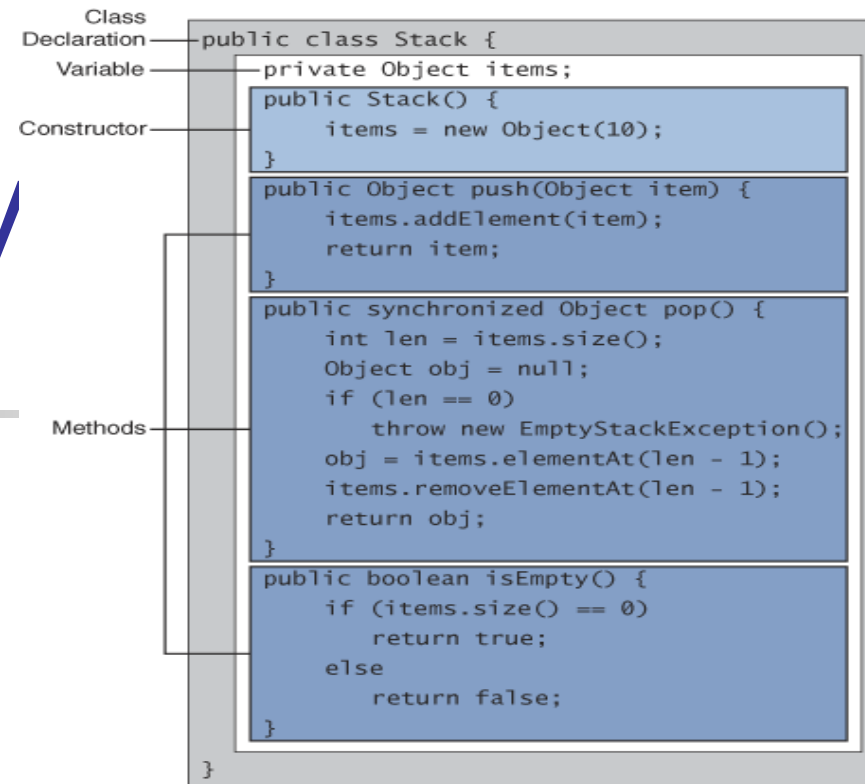
trieda **nemôže byť inštanciovaná** (asi obsahuje abstr. metódu) t.j. neexistuje objekt danej triedy

■ [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 z minulej prednášky)

→ *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

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

(rekapitulácia)

```
abstract public class Polynom {                                // úloha z cvičenia 3
    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 { ... }
```



# Singleton návrhový vzor

(rekapitulácia)

```
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 jedinecny"; }  
}  
  
public static void main(String[] args) {  
    // v inej triede nejde zavolať Singleton object = new Singleton();  
    Singleton object = Singleton.getInstance();  
    System.out.println(object);  
}
```



# Null Pointer Pattern

(návrhový vzor ako príklad abstraktnej triedy)

```
public abstract class AbstractStudent {  
    protected String name;  
    public abstract boolean isNull();  
    public abstract String getName();  
}
```

```
public class RealStudent extends  
    AbstractStudent {  
    public RealStudent(String name) {  
        this.name = name; }  
  
    @Override  
    public String getName() {  
        return name; }  
  
    @Override  
    public boolean isNull() {  
        return false; } }  
    
```

```
public class NullStudent extends  
    AbstractStudent {  
    @Override  
    public String getName() {  
        return "no name"; }  
  
    @Override  
    public boolean isNull() {  
        return true; } }  
    
```

# NullPointerException Pattern

(použitie)

```
public static AbstractStudent newStudent(String name) {  
    // vráti Realneho resp. Null študenta  
    // nikdy nevráti Abstraktného ...  
    if (name != null && name.length() > 0) // napr. podľa mena...  
        return new RealStudent(name);  
    else  
        return new NullStudent();  
} // vráti Abstraktný je vlastne zjednotením Realnych a Null štud.  
  
AbstractStudent[] group = { // pole Realnych resp. Null študentov  
    newStudent("Peter"),  
    newStudent(""),  
    newStudent("Pavel"),  
    newStudent(null) };  
  
for (AbstractStudent as : group)  
    System.out.println(as.getName());
```

Peter
no name
Pavel
no name

Craig Larman: Encapsulation is a mechanism used to *hide the data, internal structure, and implementation details* of an object. All interaction with the object is through a public interface of operations.



# Enkapsulácia

## (rekapitulácia)

---

```
... class Tovar {  
    public double cena;    // používajte gettery a settery miesto public
```

takto:

```
    private double cena;  
    public double getCena() {  
        return (cena);  
    }  
    public void setCena(double novaCena) {  
        cena = novaCena;  
    }  
                                     // používajte kontrolné testy  
    public void setCena(double novaCena) {  
        if (novaCena < 0) { // na odhalenie nekorektných hodnôt  
            sendErrorMessage(...); // toto nemôžete urobiť, ak  
            cena = novaCena; //pristupujete k public položkám priamo  
        }  
    }
```

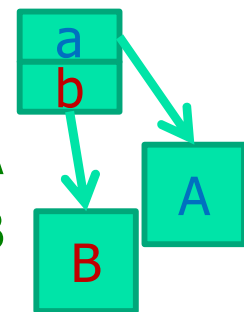


# Kompozícia objektov

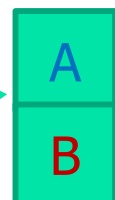
(agregácia objektov)

spojenie viacerých objektov do jedného, ktorý poskytuje funkcionality všetkých spojených objektov

```
class A { // trieda A so svojimi metódami, default.konštruktor
    public void doA() { ... } ... }
class B { // trieda B so svojimi metódami, default.konštruktor
    public void doB() { ... } ... }
class C { // trieda C spája triedy A + B
    A a = new A(); // vložená referencia (!) na objekt a typu A
    B b = new B(); // vložená referencia (!) na objekt b typu B
}
C c = new C(); // vytvorený objekt obsahujúci a:A aj b:B
c.a.doA(); // interné hodnoty a:A, b:B by mali byť skryté v C
c.b.doB(); // white-box
```



**Kompozícia v Java je vždy cez referenciu,**  
**v C++ je prostredníctvom hodnoty alebo referencie.**





# Kompozícia objektov

(druhý pokus, krajšie)

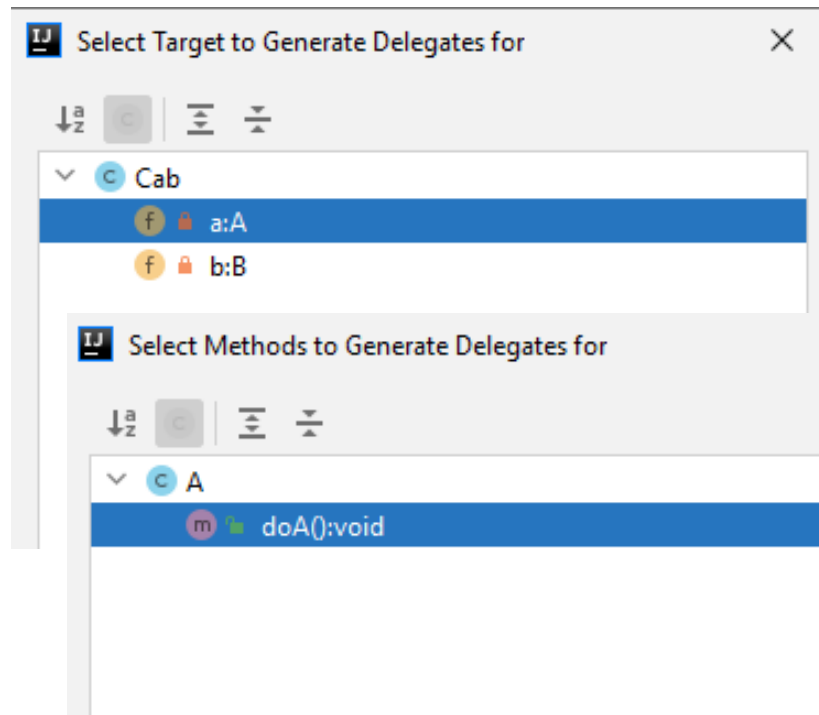
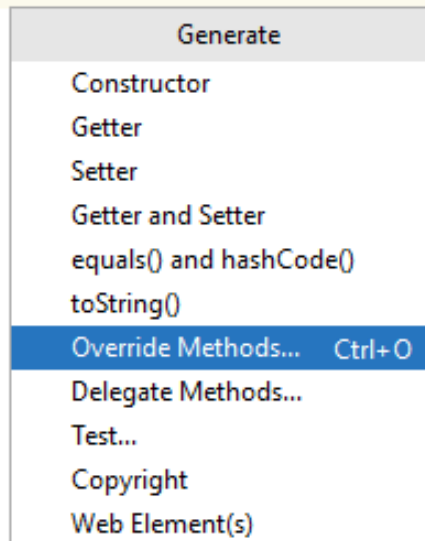
```
class A {                                // trieda A so svojimi metódami, default.konštruktor
    public void doA() { ... } ... }
class B {                                // trieda B so svojimi metódami, default.konštruktor
    public void doB() { ... } ... }
class C {                                // trieda C spája triedy A + B
    private A a = new A();               // vložená referencia (!) na objekt a typu A
    private B b = new B();               // vložená referencia (!) na objekt b typu B
    public void doA() { a.doA(); }        // delegovanie z triedy A do C
    public void doB() { b.doB(); }        // delegovanie z triedy B do C
}
C c = new C();                           // vytvorený objekt obsahujúci a:A aj b:B
c.doA();                                  // interné hodnoty a:A, b:B sú skryté v C
c.doB();                                  // black-box
```

Ak je ich veľa,  
trochu otravné

# IntelliJ vám pomůže

(Alt-Insert)

```
class Cab { // trieda C spája  
    private A a = new A(); //  
    private B b = new B(); //
```



```
public void doA() {  
    a.doA();  
}  
  
public void doB() {  
    b.doB();  
}
```



# Dedenie vs. Kompozícia

(všetko poznáme, aj tak nás to zaskočí)

```
public class Nadtrieda {  
    String[] pole = new String[100];  
    int counter = 0;  
    public void add(String s) { //pridaj 1  
        pole[counter++] = s;  
    }  
    public void addAll(String[] p) {  
        for(String s:p) // pridaj všetky  
            add(s);  
    }  
}
```

?

```
public static void main(String[] args) {  
    Podtrieda s = new Podtrieda();  
    s.addAll(new String[]{"Peter", "Pavol"});  
    System.out.println(s.getAddCount());  
}
```

```
public class Podtrieda extends Nadtrieda {  
    private int addCount = 0;  
    @Override  
    public void add(String s) { //pridaj 1  
        addCount++;  
        super.add(s);  
    }  
    @Override  
    public void addAll(String[] c) { // pridaj  
        addCount += c.length; // všetky  
        super.addAll(c);  
    }  
    public int getAddCount() {  
        return addCount; }  
}
```

// čo je výsledok ??? 2 alebo 4 ? [Podtried.java](#)





# To isté s kompozíciou

```
public class Kompozicia {                                     // „Podtrieda“ z predošlého slajdu
    private Nadtrieda n = new Nadtrieda(); // vložená nadtrieda
    private int addCount = 0;
    // nie je @Override
    public void add(String s) { //pridaj 1
        addCount++;
        n.add(s);
    }
    // nie je @Override
    public void addAll(String[] c) {
        addCount += c.length;
        n.addAll(c);
    }
    public int getAddCount() {
        return addCount; }
}
```

```
public static void main(String[] args) {
    Kompozicia s = new Kompozicia();
    s.addAll(new String[]{"Peter", "Pavol"});
    System.out.println(s.getAddCount());
}
// čo je výsledok ???    2 alebo 4 ???
```



# Dedenie vs. Kompozícia

- pri dedení je nadtrieda *zovšeobecnením*, obsahuje len spoločné metódy a atribúty všetkých podtried
- podtrieda je *konkretizáciou* s rozšírením o nové metódy, triedy a o novú funkcionálnosť všeobecných metód

+ nadtrieda sa ľahko modifikuje, dopĺňa, ...

- z podtriedy často vidíme detaily nadtriedy, a môžeme ich modifikovať, prepísať

**Riešenie:** poznaj a používaj

`final` – metóda nemôže byť prepísaná v podtriede

`private` – metóda/atribút nie je vidieť v podtriede

- prístup ku skomponovaným objektom je len cez interface (alias delegované metódy) nadtriedy,  
...teda, ak komponované objekty neurobíte public ☺ ale private

+ interné metódy/atribúty skomponovaných podtried sú dobre ukryté

- je náročnejšie definovať interface pre skomponované objekty, ako pri dedení (to je zadarmo)

# Dedenie vs. Kompozícia

```
public class QueueUsingInheritance<E>
    extends ArrayList<E> {

    public void enqueue(E e) {}
    public E dequeue() { return null; }
```

- front nie je podtyp zoznamu, preto nemá prečo byť od neho podedený
- podtyp, podtrieda je niečo užšie, špecializovanejšie, viac metódami
- podedený front disponuje metódami zoznamu, get, remove, čo by nemal

```
class Zloduch1<E> extends
QueueUsingInheritance<E> {

    public static void main(String[] args) {
        QueueUsingInheritance queue =
            new QueueUsingInheritance<String>();
        queue.enqueue("Jano");
        queue.remove(17);
```

```
public class QueueUsingComposition<E> {
    private List<E> q = new ArrayList<>();

    public void enqueue(E e) {}
    public E dequeue() { return null; }
```



Ak to urobíte takto, moc ste si nepomohli

```
public class QueueUsingCompositionZLE<E> {
    public List<E> q = new ArrayList<>();
    protected List<E> q = new ArrayList<>();
        List<E> q = new ArrayList<>();
    public void enqueue(E e) {}
    public E dequeue() { return null; }
```

```
class Zloduch2<E> extends
QueueUsingCompositionZLE<E> {

    public static void main(String[] args) {
        QueueUsingCompositionZLE queue =
            new QueueUsingCompositionZLE<String>();
        queue.enqueue("Jano");
        queue.q.remove(17);
```



# Immutable object

(nemeniteľná trieda – v prednáške .py)

- objekt, ktorého hodnotu (stav) nemôžeme zmeniť

```
public class Mutable {                                // tento asi nebude immutable :-)
    private int x;
    public Mutable(int x) { this.x = x; }              // konštruktor
    public int getX() { return x; }                   // getter
    public void setX(int x) { this.x = x; }            // setter
    @Override
    public String toString() { return "Mutable [x=" + x + "];" }
}
Mutable obj1 = new Mutable(77);
Mutable obj2 = obj1; // reference sharing, najväčšia katastrofa po NPE
System.out.println(obj1);                             Mutable [x=77]
System.out.println(obj2);                             Mutable [x=77]
obj1.setX(999);
System.out.println(obj1);                             Mutable [x=999]
System.out.println(obj2);                             Mutable [x=999]
```

[Mutable.java](#)



# Immutable object

(druhý pokus)

- objekt, ktorého hodnotu (stav) nemôžeme zmeniť

```
final class Immutable { // trieda je final, nemožno vytvoriť jej podtriedu
    private final int x; // stavovú premennú nemožno zmeniť, dostane
    public Immutable(int x) { this.x = x; } // hodnotu v konštruktore
    public int getX() { return x; } // má len getter, nie setter
    @Override
    public String toString() { return "Immutable [x=" + x + "]"; }
}
Immutable obj1 = new Immutable(77);
Immutable obj2 = obj1;
System.out.println(obj1);
System.out.println(obj2);
obj1 = new Immutable(999); // inak sa obj1 nedá zmeniť
System.out.println(obj1);
System.out.println(obj2);
```

Immutable [x=77]  
Immutable [x=77]  
Immutable [x=999]  
Immutable [x=77]



# Immutable object

(zhrnutie)

---

Immutable object:

- je z **final** triedy, aby nebolo možné zmeniť stav z objektu podedenej triedy
- triedne premenné sú **final**, ergo konštanty, získajú hodnotu v konštruktore
- logicky neponúka settery...

Používanie Immutable objects má svoje:

- **výhody**
  - patrí to medzi „best practices“
  - pri konkurentných výpočtoch (vláknach/threads) potrebujeme synchronizované dátové štruktúry, inak thread-safe, žiadne vlákno nemôže hodnotu zmeniť len skopírovať-a-zmeniť
- **aj nevýhody**
  - alokovanie/upratovanie pamäte je relatívne najdrahšia operácia VM

Vždy zvážte podľa konkrétnej aplikácie:

- nepatrné spomalenie v run-time vám môže ušetriť hodiny v debug-time...

Príklad: String, ...



# Dedenie je jedna z foriem polymorfizmu

---

Polymorfizmus je keď hodnota premennej môže patriť viacerým typom

- **univerzálny**

funkcia narába s (potenciálne) nekonečným počtom súvisiacich typov

- inkluzívny, dedenie, class-inheritance (dnes)
  - **objekt podtriedy sa môže vyskytnúť tam, kde sa čaká objekt nadtriedy**
- parametrický (na budúce)
  - generics: `public class ArrayStack<E> implements Stack<E>`

- **ad-hoc**

funkcia narába s konečným počtom (zväčša nesúvisiacich) typov

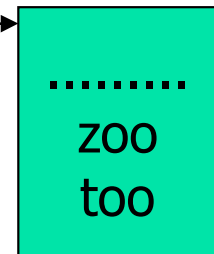
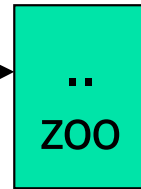
- preťažovanie (overloading) (už bolo dosť...)
  - `void foo(int x), void foo(float x)`
- pretypovanie (cast)



# Inkluzívny polymorfizmus

Mantra: **objekt podtriedy môže byť tam, kde sa čaká objekt nadtriedy**

```
public class Superclass {  
    public void zoo() { }  
}  
public class Subclass extends Superclass {  
    public void too() { }  
}  
public static void foo(Superclass x) { }  
public static void goo(Subclass x) { }
```



```
public static Superclass choo() { return new Superclass(); }  
public static Subclass hoo() { return new Subclass(); }
```

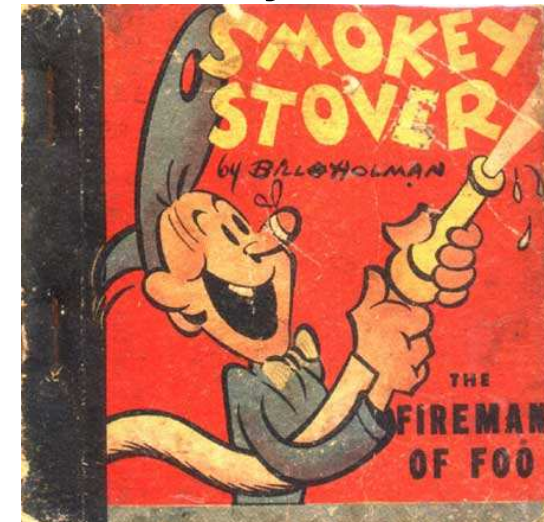
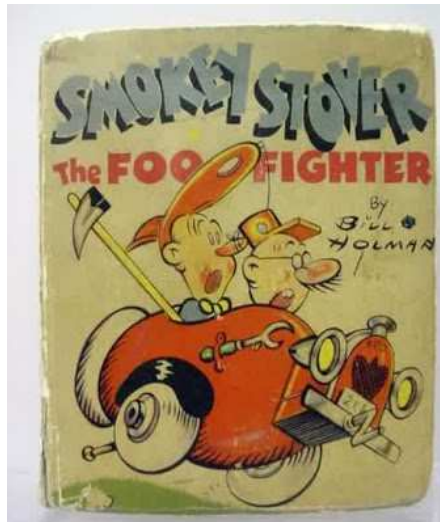
```
foo(new Subclass());  
goo(new Superclass()); ☹  
Superclass supcl = hoo();  
Subclass subcl = choo(); ☹
```

```
hoo().too();  
hoo().zoo();  
choo().too(); ☹  
choo().zoo();
```



# foo: /foo/ Term of disgust.

- Used very generally as a sample name for absolutely anything, esp. programs and files (esp. scratch files).
- When 'foo' is used in connection with 'bar' it has generally traced to the WWII-era Army slang acronym FUBAR ('Fucked Up Beyond All Repair' or 'Fucked Up Beyond All Recognition'), later modified to foobar.
- "Foo" and "bar" as metasyntactic variables were popularized by MIT and DEC, the first references are in work on LISP and PDP-1 and Project MAC from 1964 onwards.





# Interface

---

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

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 inštancie, nejde urobiť new Comparable
- interface zodpovedá tomu, čo poznáme pod pojmom  $\underline{T}$   $\underline{Y}$   $\underline{P}$

```
interface Car {                    interface Bus {
    int speed = 50; // in km/h      int distance = 100; // in km
    public void distance();          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](#)



# Viacnásobný interface

(náhrada za chýbajúce viacnásobné dedenie)

- trieda preto môže implementovať (splňať) viacero rôznych interface

```
class Vehicle implements Car, Bus {  
    public void distance() { // ale musí implementovať všetky  
        System.out.println("distance is " + distance);  
    }  
    public void speed() { // predpísané metódy zo všetkých  
        System.out.println("car speed is " + Car.speed);  
        System.out.println("bus speed is " + Bus.speed);  
    }  
}
```

```
Car c1 = this; // this je Vehicle, takže bude aj Car,  
Bus b1 = this; //                               aj Bus  
Bus b2 = c1;   ????  
Vehicle v = new Vehicle();  
System.out.println(v.speed);           ????  
System.out.println(((Car)v).speed);  
System.out.println(v.distance);
```



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



# Break

---



# Interface a dedenie

- podtrieda dedí z nadtriedy metódy a atribúty (dedenie = class inheritance)
- interface sa tiež môže dediť (z typu dostaneme jeho podtyp)
- hodnota podtypu je použiteľná všade tam, kde sa čaká nadtyp

ALE:

trieda implementujúca podtyp nie je podtriedou triedy implementujúcej nadtyp

- interface má len final a static premenné

```
interface NadInterface { // nadtyp
```

```
interface PodInterface extends NadInterface { // podtyp
```

```
// trieda implementujúca nadtyp
```

```
class NadInterfaceExample implements NadInterface
```

```
// trieda implementujúca podtyp
```

```
class PodInterfaceExample implements PodInterface
```

[NadInterface.java](#), [PodInterface.java](#)



# Interface vs. class inheritance

```
public interface NadInterface {  
    public void add(String s);  
    int aa = 9;
```

```
public interface PodInterface  
    extends NadInterface {  
    public void addAll(String[] p);  
    int bb = 10;
```

```
public class NadInterfaceExample  
    implements NadInterface {  
    public void add(String s) { }  
    public int a;
```

```
public class PodInterfaceExample  
    implements PodInterface {  
    public void add(String s) { }  
    public void addAll(String[] p) {}  
    public int b;
```

**NadInterfaceExample a PodInterfaceExample nie sú podtriedy**

NadInterfaceExample nie1 = nie; 😊

NadInterfaceExample nie2 = pie; 😞

PodInterfaceExample pie1 = nie; 😞

PodInterfaceExample pie2 = pie; 😊





# Interface vs. class inheritance

```
NadInterfaceExample nie = new NadInterfaceExample();  
PodInterfaceExample pie = new PodInterfaceExample();
```

```
pie.addAll(null); 😊
```

```
nie.addAll(null); ☹️
```

**NadInterfaceExample a PodInterfaceExample nie sú podtriedy**

```
NadInterfaceExample nie1 = nie; 😊
```

```
NadInterfaceExample nie2 = pie; ☹️
```

```
PodInterfaceExample pie1 = nie; ☹️
```

```
PodInterfaceExample pie2 = pie; 😊
```

```
System.out.print(pie.b); 😊
```

```
System.out.print(pie.a); ☹️
```

```
System.out.print(pie.bb); 😊
```

```
System.out.print(pie.aa); 😊
```

```
public interface NadInterface {  
    public void add(String s);  
    int aa = 9;
```

```
public interface PodInterface  
    extends NadInterface {  
    public void addAll(String[] p);  
    int bb = 10;
```

```
public class NadInterfaceExample  
    implements NadInterface {  
    public void add(String s) { }  
    public int a;
```

```
public class PodInterfaceExample  
    implements PodInterface {  
    public void add(String s) { }  
    public void addAll(String[] p) {}  
    public int b;
```

InterfaceExample.java



# Interface vs. class inheritance

```
NadInterface ni = new NadInterfaceExample(); // nie  
PodInterface pi = new PodInterfaceExample(); // pie
```

**NadInterface je nadtyp PodInterface**

```
NadInterface nie1 = ni; 😊  
NadInterface nie2 = pi; 😊  
PodInterface pie1 = ni; 😞  
PodInterface pie2 = pi; 😊
```

Uffff...

```
List<String> lst = new ArrayList<String>();
```

```
public interface NadInterface {  
    public void add(String s);  
    int aa = 9;
```

```
public class NadInterfaceExample  
implements NadInterface {  
    public void add(String s) { }  
    public int a;
```

```
public interface PodInterface  
extends NadInterface {  
    public void addAll(String[] p);  
    int bb = 10;
```

```
public class PodInterfaceExample  
implements PodInterface {  
    public void add(String s) { }  
    public void addAll(String[] p) {}  
    public int b;
```

InterfaceExample.java



# Package

z vašej C++ prednášky viete, že:  
„Čím viac sa program rozrastá, tým viac  
pribúda globálnych premenných. Je  
rozumné ich deliť do akýchsi rodín, spájať  
logicky zviazané premenné jedným  
priezviskom – **namespace**“

Package je adekvátny koncept v Java.

Definícia:

```
package balicek;    // subor Trieda.java patrí
    public class Trieda {    // do balíka balicek
        int sirka;
        int dlzka;
    }
```

Použitie balíčka:

```
import balicek.Trieda; // použi Trieda z balicek
alebo
import balicek.*;    // ber všetky triedy z balicek
... // a potom v programe ...
... Trieda o = new Trieda();
... o.dlzka = o.dlzka;
```

Nepoužitie balíčka:

```
balicek.Trieda o = new balicek.Trieda();
```

definícia:

```
namespace rozmery {
    int sirka;
    int dlzka;
}
```

použitie:

```
rozmery::sirka alebo
using namespace rozmery;
```

# Balíčkovanie

## Package java.lang

<a href="#"><u>String</u></a>	The String class represents character strings.
<a href="#"><u>StringBuffer</u></a>	A thread-safe, mutable sequence of characters.
<a href="#"><u>StringBuilder</u></a>	A mutable sequence of characters.
<a href="#"><u>System</u></a>	The System class contains several useful class fields and methods.

- v prostredí Eclipse existujú tri úrovne abstrakcie: project-package-class,
- project nemá podporu v jazyku Java,
- **package** je zoskupenie *súvisiacich typov*: napr. tried, interface, ...

Príklady už používaných balíčkov sme videli:

balík java.lang obsahuje o.i. triedy java.lang.Math, java.lang.System, ...

- použitie deklarujeme pomocou konštrukcie import:
  - použitie jednej triedy z balíčka **import** java.lang.Math;
  - všetkých tried z balíčka **import** java.lang.\*;
  - statické metódy/konštanty z triedy z balíčka **import static** java.lang.Math;

Prečo balíčkovat':

- aby súvisiace veci boli pokope (v adresári),
- aby v adresári bolo len rozumne veľa .java, .class súborov,
- aby sme si nemuseli vymýšľať stále nové unikátne mená tried,
- aby Java chránila prístup dovnútra balíčka (uvidíme),
- príprava pre vytvorenie archívneho .jar súboru



# Konvencie (nielen balíčkovania)

---

**Triedy**, napr.: *class Raster; class ImageSprite; package C*

- meno triedy je podstatné meno, každé podslovo začína veľkým písmenom (mixed case), celé meno začína veľkým písmenom.

**Balík**, napr.: *package java.lang;*

- malým písmenom.

**Metódy**, napr.: *run(); runFast(); getBackground();*

- mená metód sú slovesá, začínajú malým písmenom.

**Premenné**, napr. *int i; char c; float myWidth;*

- začínajú malým písmenom, mixed case, nezačínajú `_` resp. `$`. Jednopísmenkové mená sú na dočasné premenné.

**Konštanty**, napr. *static final int MIN\_WIDTH = 4; static final int MAX\_WIDTH = 999; static final int GET\_THE\_CPU = 1;*

- Veľkými, slová oddelené ("`_`").



# Vytvorenie balíčka

- pre pomenovanie balíčka sa používa inverzná doménová konvencia:
  - **package** sk.fmpi.prog4.java\_04;
- triedy balíčka sú potom organizované v jednom adresári:
  - <workspace>\sk\fmpi\prog4\java\_04\...
  - <workspace>/sk/fmpi/prog4/java\_04/...
- štandardné balíčky JavaSE začínajú s java. a javax.
- balíčky môžu mať podbalíčky, napríklad:
- **package** sk.fmpi.prog4.java\_04.One;
- **package** sk.fmpi.prog4.java\_04.Two;
- `import sk.fmpi.prog4.java_04.*;` sprístupní triedy balíčka, ale nie podbalíčkov – `import` nie je rekurzívny

```
package sk.fmpi.prog4.java_04;
import sk.fmpi.prog4.java_04.*;
public class Test {
    public static void main(String[] args) {
        Alpha nad = new Alpha();    // chyba
```

# Viditeľnosť metód/premenných



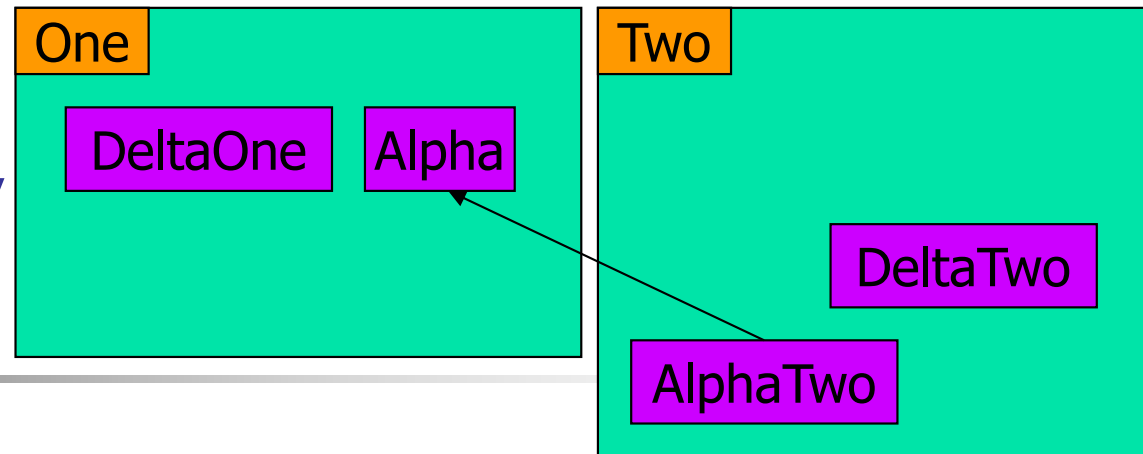
	<b>Trieda</b>	<b>Package</b>	<b>Podtrieda</b>	<b>Inde</b>
■ <b>private</b>	+	-	-	-
■ <b>nič</b>	+	+	-	-
■ <b>protected</b>	+	+	+	-
■ <b>public</b>	+	+	+	+

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
```

# Prístup na úrovni triedy



package **One**; // definuje triedy patriace do jedného balíka

```
public class Alpha {
    private      int iamprivate = 1;
                  int iampackage = 2;
    protected  int iamprotected = 3;
    public      int iampublic = 4;

    private void privateMethod() {}
    void packageMethod() {}
    protected void protectedMethod() {}
    public void publicMethod() {}
}
```

```
public static void main(String[] args) {
    Alpha a = new Alpha();
    // v rámci triedy vidno všetko
    a.privateMethod();
    a.packageMethod();
    a.protectedMethod();
    a.publicMethod();

    a.iamprivate
    a.iampackage
    a.iamprotected
    a.iampublic
}
```



Public method can be accessed from any other class.



# Prístup na úrovni package

```
package One; // ďalšia trieda z balíka One
```

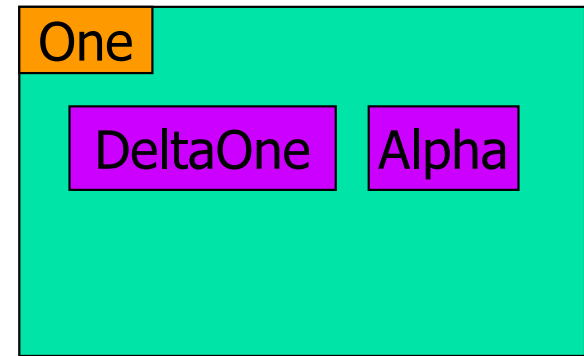
```
public class DeltaOne {
```

```
    public static void main (String[] args) {  
        Alpha a = new Alpha();
```

```
        //a.privateMethod(); // nevidno, lebo je private v triede Alpha  
        a.packageMethod();  
        a.protectedMethod();  
        a.publicMethod();
```

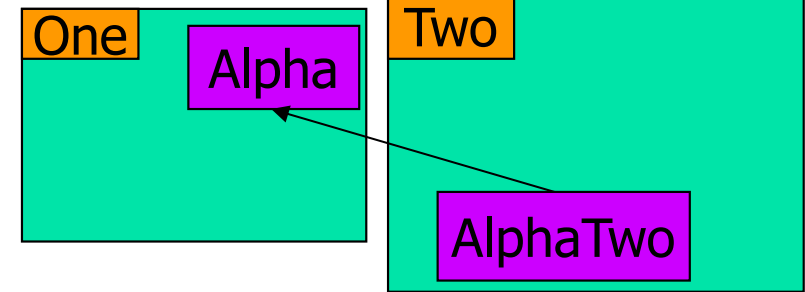
```
        // a.iamprivate // nevidno, lebo je private v triede Alpha  
        a.iampackage  
        a.iamprotected  
        a.iampublic
```

```
    }  
}
```



Package method can be accessed from any other class in the same package.

# Prístup z podtriedy



```
package Two;  
import One.*;
```

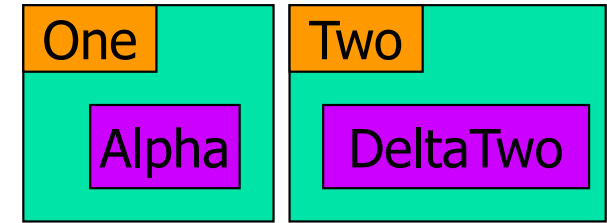
// iný balíček

// import.všetky triedy z One( Alpha a DeltaOne)

```
public class AlphaTwo extends Alpha { // podtrieda triedy Alpha  
    public static void main(String[] args) {  
        Alpha a = new Alpha(); // Alpha nie je podtrieda  
        // a.privateMethod(); // nevidno, lebo je private v triede Alpha  
        // a.packageMethod(); // nevidno, lebo sme v package Two  
        // a.protectedMethod(); // nevidno, aj keď sme v podtriede AlphaTwo,  
        // lebo a:Alpha nie je podtrieda AlphaTwo  
  
        a.publicMethod();  
        // a.iamprivate +  
        // a.iampackage +  
        // a.iamprotected + // to isté  
        a.iampublic;  
  
        // protected v AlphaTwo možno aplikovať len na  
        AlphaTwo a2 = new AlphaTwo(); // AlphaTwo, alebo jej podtriedu  
        a2.protectedMethod();  
        r = a2.iamprotected;  
    }  
}
```

Protected method declared in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class

# Prístup z okolitého sveta



```
package Two; // iný balíček
import One.Alpha; // importuj len triedu Alpha z balíčka One
```

```
public class DeltaTwo { // nemá nič s Alpha, AlphaTwo
    public static void main(String[] args) {
```

```
        Alpha a = new Alpha();
```

```
        //a.privateMethod();
        //a.packageMethod();
        //a.protectedMethod();
        a.publicMethod();
```

```
        int r =// a.iamprivate +
                // a.iampackage +
                // a.iamprotected +
                a.iampublic;
    }
}
```

public – použiteľná pre každého  
private – použiteľná len vo vnútri def.triedy  
protected – len vo vnútri triedy a v zdedených triedach

# Ako to bolo v Python

Enkapsulácia je základný princíp OOP (Rosumie tomu každý, až na G.Rossuma)

- všetko je public by default (katastrofa)
- protected – začínajú jedným \_ / ale to je len konvencia  
prefix \_ znamená, že nepoužívaj ma mimo podtriedy (doporučenie...)
- private – začínajú dvomi \_\_ / to nie je konvencia

class Bod:

    \_totoJeProtectedVariable = ...

    \_\_totoJePrivateVariable = ...

bod = Bod()

bod.\_totoJeProtectedVariable

bod.\_\_totoJePrivateVariable

bod.\_Bod\_\_totoJePrivateVariable

niekto mi to zakáže ??? ☹️

niekto mi to zakáže ??? 😊

niekto mi to zakáže ???

# Nevnorené triedy

- v definícii triedy sa môže nachádzať definícia inej triedy, ak ...
- ale to znamená, že súbor sa nemôže volať ako *Trieda.java* - lebo sú dve☺
- aj preto toto **nemôžeme** urobiť:

```
public class DveNevnoreneTriedy {  
}
```

```
public class Druha { // Druha musí byť definovaná vo vlastnom súbore  
}
```

- ale ak **nie sú** public (ale private, protected, nič, final, abstract), tak to je správne:

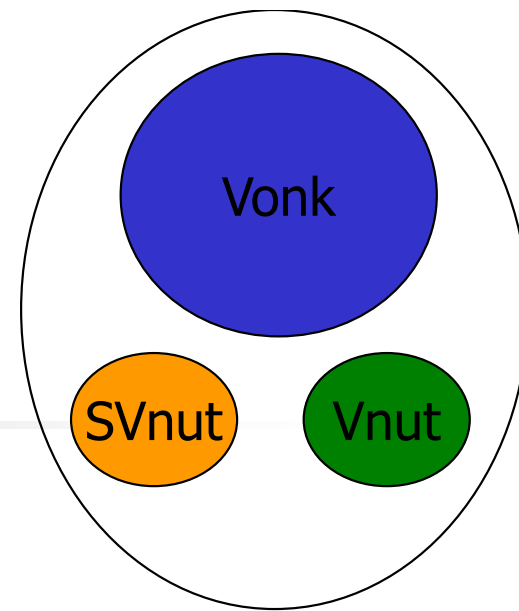
```
class Tretia {  
}
```

```
abstract class Stvrta {  
}
```

```
final class Piata {  
}
```



# Vnorené triedy



```
public class Vonkajsia {  
    public int a = 1;  
    public static int stat = 2;
```

```
    public class Vnutorna {  
        public int b = a;  
    }
```

// vnorená trieda môže byť public  
// protected, private aj nič

```
    public static class StatickaVnutorna {  
        public int c = stat+a+b;  
    }  
}
```

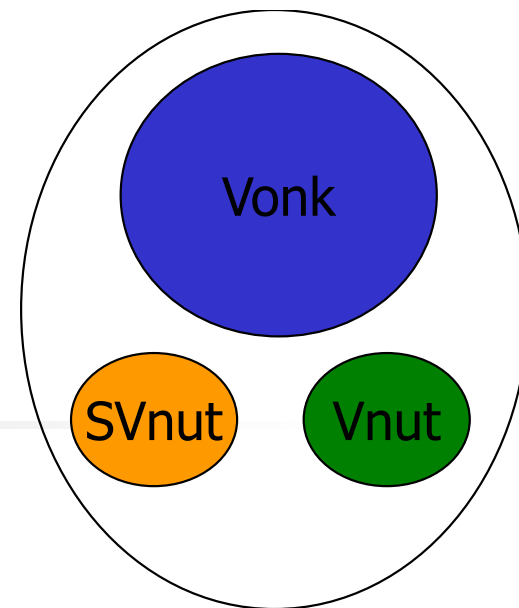
// aj statická

// chyba: nevidno ani a, ani b

**Do každej triedy dáme jej implicitný konštruktor:**

```
public ...() {  
    System.out.println("Vytvaram: "+getClass().getName());  
}
```

# Vnorené triedy



```
public class Vonkajsia {  
    public class Vnutorna { }  
    public static class StatickaVnutorna { }  
}
```

**Princíp vnútornej triedy: Vnútoraná trieda bez vonkajšej neexistuje**

```
Vonkajsia vonk = new Vonkajsia();
```

Vytvaram: Vonkajsia

```
// Vnutorna vnut1 = new Vnutorna(); -- chyba: nepozná Vnutornu triedu
```

```
// Vonkajsia.Vnutorna vnut2 = new Vonkajsia.Vnutorna();
```

-- chyba: Vnutorna bez Vonkajsej neexistuje

```
Vonkajsia.Vnutorna vnut3 = vonk.new Vnutorna();
```

Vytvaram: Vonkajsia\$Vnutorna

```
Vonkajsia.Vnutorna vnut4 = new Vonkajsia().new Vnutorna();
```

Vytvaram: Vonkajsia

Vytvaram: Vonkajsia\$Vnutorna

# Dedenie s vnorenými

```
public class PodVnutornou extends Vonkajsia.Vnutorna {  
}
```

**Princíp vnútornej triedy: Vnútná trieda bez vonkajšej neexistuje**

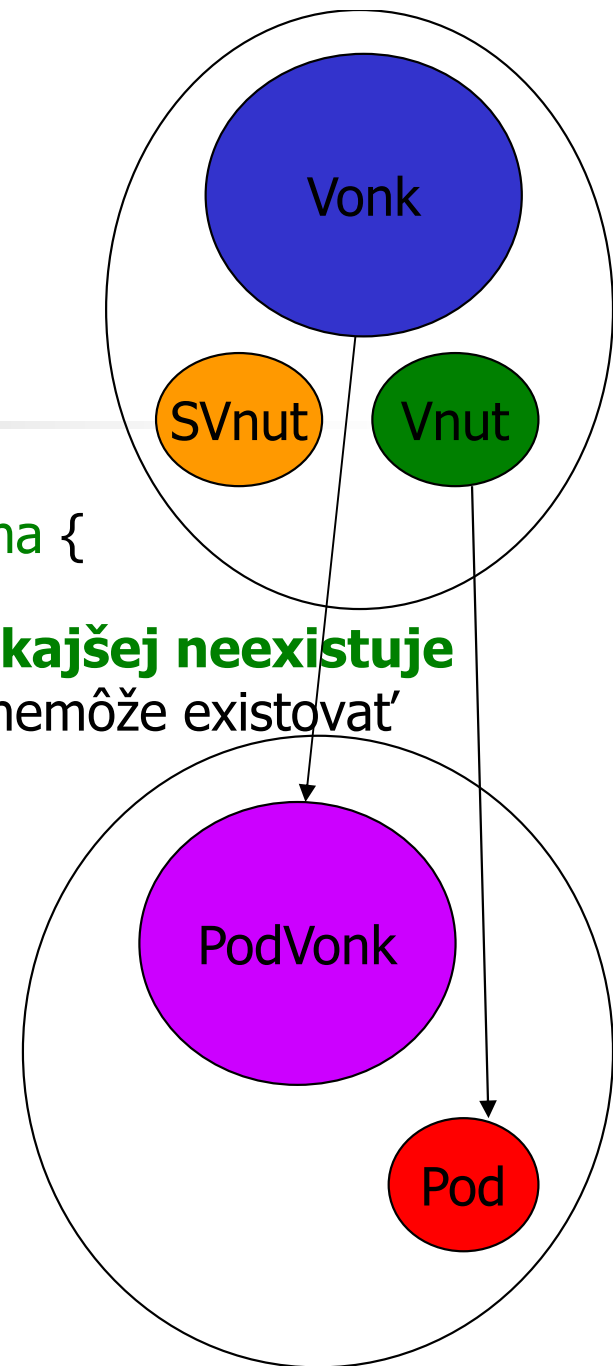
Preto PodVnutornou ako trieda, ktorá nemá Vonkajsiu, nemôže existovať

Ale toto môžeme:

```
public class PodVonkajsou extends Vonkajsia {  
  
    public class PodVnutornou extends Vnutorna {  
    }  
}
```

```
PodVonkajsou vonk = new PodVonkajsou();  
PodVnutornou vnut = vonk.new PodVnutornou();
```

```
}
```



Súbor: PodVnutornou.java

Súbor: PodVonkajsou.java





# A.java

---

```
package A;
```

```
class A {  
    A A(A A) {  
        A:  
        for (;;) {  
            if (A.A(A) == A) break A;  
        }  
        return A;  
    }  
}
```



**MEANING(LESS)**