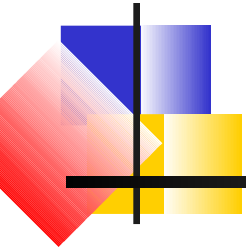
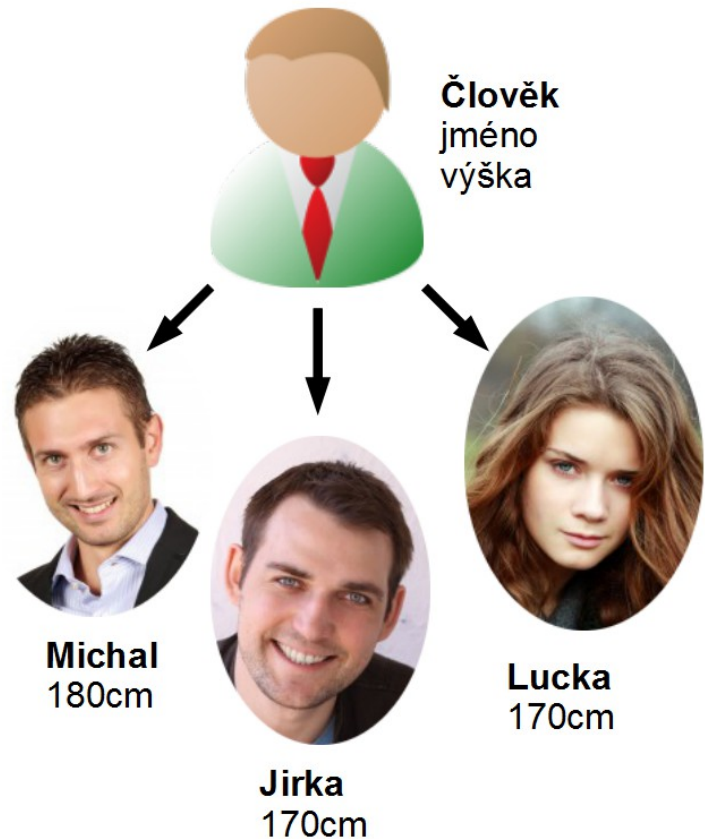


Úvod do programovacího jazyka Java



Třída vs. objekt

- Třída je předpisem (typem objektu), na jehož základě lze vytvořit celou řadu instancí třídy (objektů) s vlastními hodnotami atributů ve vlastní paměťové oblasti.
- Na obrázku vpravo se nachází:
 - Jedna třída Člověk, která má dva atributy: jméno a výška.
 - Tři objekty (instance) této třídy.



Třída v Javě I.

- Třída v Javě = soubor se stejným názvem na disku.
- Třída se skládá z atributů a metod.
- Konvence při tvorbě tříd:
 - Třídy mají první písmeno velké.
 - Každá třída by měla být v nějakém balíčku.
 - Atributy a metody mají první písmeno malé.
 - V názvech tříd, atributů a metod není doporučeno používat diakritiku.

název balíčku

```
package cz.skoleni.helloworld;
```

název třídy

```
public class Clovek {
```

začátek třídy

```
    public String jmeno;
```

název atributu

```
    public int vyska;
```

typ atributu

```
}
```

konec třídy



Třída v Javě II.

- Se stavem objektu (atributy) lze manipulovat voláním metod.

```
package cz.skoleni.java.helloworld;

public class Clovek {
    public String jmeno;
    public int vyska;
    public void print() {
        System.out.println("Člověk:");
        System.out.println("jméno: " + jmeno);
        System.out.println("výška: " + vyska);
    }
}
```

název metody

začátek metody

výpis na konzoli

konec metody



Třída v Javě III.

- Každá třída nemusí mít vlastní soubor (i když tomu tak je v drtivé většině případů). Existují i vnitřní třídy, anonymní třídy a od Java SE 8 také lambda výrazy.
 - <http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html>



Metody I.

- Metoda může mít vstupy anebo výstupy.
 - Počet vstupů není omezený, každý vstup má typ a název parametru.
 - Metoda může mít maximálně jeden výstup nějakého typu. Výstupem může být primitivní datový typ, String nebo jakýkoli jiný objekt. K vrácení výstupu z metody se používá příkaz `return`. Po zavolání tohoto příkazu se metoda zároveň ukončí.
 - V případě, že metoda nemá žádný výstup, tak vrací `void`. Pro předčasné ukončení metody se používá příkaz `return`;

návratový
typ metody

```
public String txtClovek(String jednotka) {  
    return jmeno + ", výška: " + vyska + " " + jednotka;  
}
```

parametry

ukončení metody



Metody II.

- Uvnitř jedné třídy můžete mít více metod které mají stejný název, ale liší se počtem nebo typem parametrů:

```
public String txtClovek(String jednotka) {  
    return jmeno + ", výška: " + vyska + " " + jednotka;  
}
```

```
public String txtClovek() {  
    return jmeno + ", výška: " + vyska;  
}
```



Single point of exit ... or not?

- Vedou se debaty nad tím, jestli se má příkaz `return`; používat max. jednou (single point of exit), nebo jestli se může používat vícekrát. V některých firmách je dokonce zakázáno používat více příkazů `return`; v jedné metodě.
 - Každopádně by se tento příkaz neměl zneužívat a měl by se jeho výskyt minimalizovat.
 - Nicméně jeho použití dokáže zvýšit čitelnost kódu (zejména v malých metodách).
- Jedna z mnoha diskuzí na toto téma:
 - <http://stackoverflow.com/questions/36707/should-a-function-have-only-one-return-statement>



Main metoda

- Pro běh aplikace je nutné v nějaké třídě nadefinovat metodu `main()`.
- Tato metoda obsahuje pole argumentů, které je možné aplikaci při spuštění předat:
 - `java -jar aplikace.jar arg1 arg2 arg3`

```
package cz.skoleni.java.helloworld;
```

```
public class Aplikace {
```

```
metoda main → public static void main(String[] args) {  
    System.out.println("Hello World!");  
}  
}
```

Tvorba objektů, přístup k členským složkám objektu

- V jazyce Java je vše objektem (kromě primitivních datových typů)
- Nový objekt (instance třídy) se vytvoří voláním konstruktoru pomocí operátoru **new**
- K metodám a atributům objektu (instance) se přistupuje pomocí operátoru „tečka“

uložení objektu
do pomocné
proměnné clovek

vytvoření instance
třídy (objektu)

```
Clovek clovek = new Clovek();  
clovek.jmeno = "Jirka Pinkas";  
clovek.vyska = 170;  
clovek.print();
```

volání metody

přístup
k atributu



Primitivní datové typy I.

- V Javě existuje 8 primitivních datových typů.
- Nejsou to objekty, tudíž není nutné při jejich tvorbě používat operátor new.

typ	popis	velikost	min. hodnota	max. hodnota
byte	celé číslo	8 bitů	-128	+127
short	celé číslo	16 bitů	-32768	+32767
int	celé číslo	32 bitů	-2147483648	+2147483647
long	celé číslo	64 bitů	-9223372036854775808	+9223372036854775807
float	reálné číslo	32 bitů	-3.40282e+38	+3.40282e+38
double	reálné číslo	64 bitů	-1.79769e+308	+1.79769e+308
char	znak UNICODE	16 bitů	/u0000	/uFFFF
boolean	logická hodnota	<u>1</u> bit	-	-



Primitivní datové typy II.

- K základním neobjektovým typům Javy lze přistupovat také jako k objektům (pomocí jejich objektové reprezentace, tzv. wrapper classes, které umožňují neobjektovou hodnotu zabalit do objektu)
- Wrapper třídy obsahují řadu metod pro různé konverze

Primitive type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Konverze String → int:

```
int cislo = Integer.parseInt("10");
```



BigDecimal

- Double není vhodný datový typ pro práci s finančními operacemi. Při násobení double proměnných dochází ke ztrátě informace. Z toho důvodu se používá pro uchovávání peněz třída BigDecimal:
 - http://www.opentaps.org/docs/index.php/How_to_Use_Java_BigDecimal:_A_Tutorial
 - <http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal1.html>



Podtržítka v číslech (od Java 7)

- Pokud potřebujete v kódu zapsat velké číslo, pak od Java 7 můžete použít jeden z těchto způsobů:

```
int milion1 = 1000000;
```

```
int milion2 = 1_000_000;
```



Příkazy, komentáře, Javadoc

- Každý příkaz v Javě končí středníkem. Obvykle se na jeden řádek píše jeden příkaz.
- Komentáře jsou v Javě dvou typů:

`// komentář na jeden řádek`

`/* komentář`

`* na více`

`* řádků`

`*/`

- Ještě se v kódu setkáte s následujícím kusem textu, který vypadá jako komentář na více řádků, začíná ale dvěma hvězdičkami:

`/**` ← **POZOR!**

`* toto je Java dokumentace (Javadoc)`

`*/`

Javadoc = Java dokumentace

hledejte v Googlu „java api 7“

balíčky

třídy

The screenshot displays the Java Platform Standard Ed. 7 API documentation. On the left, a sidebar lists various packages and classes. The main content area shows the details for the `Class String` in the `java.lang` package. The page includes navigation tabs (Overview, Package, Class, Use, Tree, Deprecated, Index, Help) and a summary section with links for Prev Class, Next Class, Frames, and No Frames. The class details section shows the inheritance hierarchy (java.lang.Object, java.lang.String) and the implemented interfaces (Serializable, CharSequence, Comparable<String>). The class declaration is shown as `public final class String extends Object implements Serializable, Comparable<String>, CharSequence`. A description of the String class is provided, followed by an example code snippet: `String str = "abc";` and a note that it is equivalent to a string literal.

Java™ Platform Standard Ed. 7

Overview Package **Class** Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.lang

Class String

java.lang.Object
java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

`public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence`

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:



Javadoc ve vlastním kódu

- Javadoc je obecný způsob tvorby Java dokumentace, tudíž je možné ho využít i pro dokumentování vlastních kódů.
- Pomocí Javadocu je možné přidat dokumentaci ke třídě, atributu a metodě. U každého z nich je možné popsat, co bude příslušná třída / atribut / metoda dělat. U metody je možné navíc popsat vstupy a výstupy z metody.
- Následně je možné pro aplikaci vygenerovat HTML podobu Javadocu, která je svou podobou podobná oficiální Java dokumentaci.



Balíčky

- Balíčky jsou mechanismem sdružení logicky souvisejících tříd do skupin (obdoba namespace nebo unit v jiných programovacích jazycích). Balíčky jsou fyzicky adresáře.
- Balíčky se pojmenovávají podle následujícího vzoru (obrácená www adresa):
 - **`cz.nazev-firmy.nazev-projektu.cast_projektu`**
 - Příklad: `cz.skoleni.java.helloworld`
 - Na disku bude adresář `cz`, v něm adresář `javaskoleni`, v něm adresář `reference`, v něm adresář `databaze` a teprve v tomto adresáři budou třídy.
- Od Java 1.4 je důrazně doporučeno, aby každá třída byla v nějakém balíčku. Pokud se třída nenachází v žádném balíčku, pak se o takové třídě také říká, že je v „defaultním balíčku“.



Plný název třídy

- Třída je přímo viditelná pouze v rámci balíčku, ve kterém se nachází.
- V případě, že chcete použít třídu z jiného balíčku, pak je nutné:
 - buď použít plný název třídy,
 - nebo třídu naimportovat.
- **Plný název třídy** se skládá ze všech balíčků, ve kterých se třída vyskytuje a samotného názvu třídy.
 - Příklad: `cz.skoleni.java.helloworld.Aplikace`
- Plný název třídy se hodí v ojedinělých případech, kdy potřebujete v jedné třídě pracovat se dvěma stejně pojmenovanými třídami (například kdybyste chtěli provést konverzi mezi `java.util.Date` a `java.sql.Date`)



Import

- Tvorba importů se v moderních vývojových prostředích provádí téměř automatickým způsobem, jenom je nutné si dát pozor na to, abyste naimportovali správnou třídu.
- Příklady importů:

```
// naimportuje jednu vybranou třídu
```

```
import cz.skoleni.java.helloworld.Clovek;
```

```
// naimportuje všechny třídy z jednoho balíčku
```

```
import cz.skoleni.java.helloworld.*;
```

- Importování tříd je jenom zjednodušení pro programátora, aby nemusel všude psát plný název třídy. Ve fázi kompilace se importy smažou a do výsledného bytecode se uloží plné názvy tříd.



Inicializace atributů a proměnných

- U pomocných proměnných v metodách Java kontroluje při překladu, jestli byla proměnná inicializována. Pokud ne, pak se překlad ukončí chybou.
- U atributů Java používá implicitní inicializaci:
 - Celočíselné typy = 0
 - Reálné typy = 0.0
 - boolean = false
 - char = \u0000 (prázdný znak)
 - Jakýkoli objekt = null
- Při definici proměnné či atributu lze rovnou provést i inicializaci:

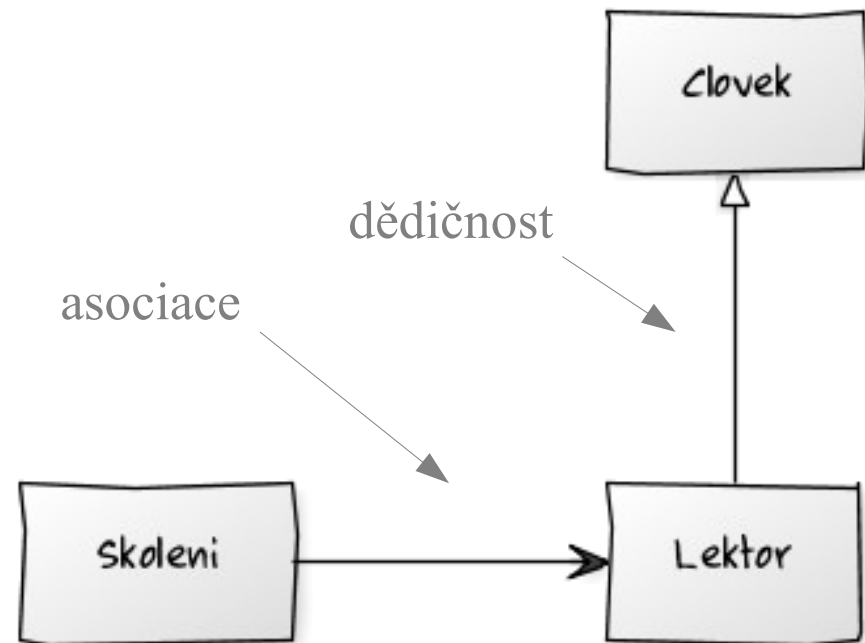
```
int cislo = 10;
```

Asociace, dědičnost

- Reálná aplikace se skládá z celé řady tříd, které jsou pospojované dvěma typy vazeb:
 - **Asociace** (vazba typu „má“) - školení má lektora
 - **Dědičnost** (vazba typu „je“) - lektor je člověk

Poznámka:

Tento graf se nazývá diagramem tříd a je to jeden z několika typů grafů UML (Unified Modeling Language)





Asociace

- Aplikace se obvykle skládá z řady tříd, které jsou zjednodušenou šablonou objektů z reálného světa.
- Tak jako každý z objektů reálného světa plní nějakou specifickou funkci a má konkrétní stav, tak i každá třída je obdobným způsobem omezená.
- V reálném světě spolu objekty interagují a mají mezi sebou typ vazby, kdy jeden objekt obsahuje jiný nebo se skládá z více objektů. Takový typ vazby se nazývá asociace a v Java kódu je zachycen jako atribut nějakého typu:

```
public class Skoleni {  
    public Lektor lektor;  ← název atributu  
}
```

← typ atributu



Dědičnost

- Dalším typem vazby je dědičnost. Tato vazba se používá pro zamezení duplikování kódu a jedná se o velice těsný typ vazby.
- Mezi dvěma třídami je při použití této vazby vztah **rodič (předek) ↔ potomek**.
- Potomek zdědí všechny vlastnosti předka.
- V Javě existuje pouze tzv. jednoduchá dědičnost. To znamená, že třída může mít maximálně jednoho předka.

```
public class Lektor extends Clovek {  
    public int rokZahajeniKariery;  
}
```

třída Lektor
je potomkem
třídy Clovek



Abstraktní metoda a třída

- Občas se dostanete do situace, kdy chcete v předkovi použít metodu, jejíž implementaci dodáte později v potomkovi. K tomu slouží abstraktní metoda.
 - Metoda je abstraktní, pokud neobsahuje tělo a je označena klíčovým slovem `abstract`.
 - Třída je abstraktní, pokud obsahuje alespoň jednu abstraktní metodu. V případě, že třídu neoznačíte klíčovým slovem `abstract`, tak Vás Java na takovou situaci sama upozorní a nepovolí takový kód ani zkompilovat.
- Není možné vytvořit instanci abstraktní třídy.



Konstruktor I.

- Konstruktor je speciální metodou, která se volá při vytváření instance třídy (konstrukci objektu).
 - Oproti obyčejné metodě se liší tím, že má stejný název jako třída a nemá návratový typ.
 - Pokud nevytvoříte vlastní konstruktor, tak kompilátor vytvoří vlastní konstruktor bez parametrů.
- Používá se, když chcete při vytvoření objektu rovnou provést inicializaci jeho atributů.

```
public Clovek(String jmeno, int vyska) {  
    this.jmeno = jmeno;  
    this.vyska = vyska;  
}
```



Konstruktor II.

- Můžete mít více konstruktorů v jedné třídě. Tyto konstruktory se mohou navzájem volat pomocí klíčového slova `this()`.

```
public Clovek(String jmeno) {  
    this(jmeno, 0);  
}
```

- Konstruktory se nedědí. Když chcete z konstruktoru potomka zavolat konstruktor předka, tak k tomu použijete klíčové slovo `super()`:

```
public Ucastnik(String jmeno) {  
    super(jmeno);  
}
```



this, super, null

- **this** = Odkaz na instanci třídy, ve které se vyskytuje
 - Přístup k atributu třídy v metodě, do které vstupuje stejně pojmenovaný parametr.
 - Volání konstruktoru třídy z jiného konstruktoru té samé třídy.
- **super** = Odkaz na metodu nebo atribut, který je definován v otcovské třídě
 - Přístup k atributu / metodě předka, která je stejně pojmenovaná v potomkovi.
 - Volání konstruktoru předka v potomkovi.
- **null** = Speciální hodnota pro odkaz, který neukazuje do žádné oblasti paměti (instance třídy není vytvořena).



Operátory I.

- **Aritmetické:** +, -, *, /
- **Inkrementace** (++), **dekrementace** (--)
 - **prefix** ($y = ++x$) ... nejprve se provede inkrementace a poté přiřazení.

```
int x = 10;  
int y = ++x;  
// hodnota y je 11
```
 - **postfix** ($y = x++$) ... nejprve se provede přiřazení a poté inkrementace.

```
int x = 10;  
int y = x++;  
// hodnota y je 10
```



Operátory II.

- **Operátor přiřazení (=):** `b += a; c *= b;`

`b += a; // to samé jako b = b + a`

`b -= a; // to samé jako b = b - a`

`b *= a; // to samé jako b = b * a`

`b /= a; // to samé jako b = b / a`

- **Relační operátory:** `==, !=, >, <, >=, <=`
- **Logické operátory:** `&&` (and), `||` (or), `!` (not)
- **Operátor zřetězení (+):**
 - Cokoli plus String je String:

```
int a = 10;
```

```
String s = a; // CHYBA!
```

```
String s = a + ""; // FUNGUJE
```

Mnohem hezčí je ale použít:

```
String s = Integer.toString(a);
```

nebo:

```
String s = String.valueOf(a);
```



Operátory III.

- **Bitové operátory:**

- & (and), | (or), ^ (xor)
- posun doleva (<<), posun doprava (>>)

- **Operátor podmínkový, ternální operátor (? :)**

- <booleovský výraz> ? <hodnota1> : <hodnota2>
- pokud je výraz true, vrátí se hodnota1, jinak hodnota2

```
String pohlavi = clovek.getPohlavi();
```

```
System.out.println(  
    "m".equals(pohlavi) ? "muž" : "žena");
```



Operátor přetypování

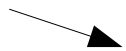
- Java je silně typový jazyk a neumožní Vám například uložit do proměnné typu `int` číslo typu `long` (`int` má menší rozsah než `long`, tudíž by mohlo dojít ke ztrátě informace). Někdy je ale ale něco takového zapotřebí (a mnohem více u vlastních objektů než u primitivních datových typů).

- V případě kompatibilních typů se nemusí nic řešit:

```
long cisloLong2 = cisloInt;
```

- V případě nekompatibilních typů je nutné provést přetypování:

```
int cisloInt = (int) cisloLong;
```





Operátor instanceof

- Pokud nevíte jakého typu je objekt, můžete to zjistit pomocí operátoru instanceof:

```
Object obj = new ArrayList();  
if(obj instanceof ArrayList) {  
    ArrayList list = (ArrayList) obj;  
}
```

- Poznámka: Kdybyste provedli přetypování na nekompatibilní typ, pak se vyhodí výjimka typu `ClassCastException`



Třída Object

- Třída Object je na vrcholu hierarchie tříd v Javě. Každá třída je potomkem třídy Object.
 - V případě, že Vaše třída nemá žádného předka, tak kompilátor doplní v definici třídy `extends Object`.
- Tato třída definuje základní stav a chování objektu, dává mu možnost porovnání s jiným objektem, zkonvertovat ho na řetězec apod.
- Metoda `toString()`
 - Vrací řetězcovou reprezentaci objektu.
 - Zavolá se automaticky když chcete překonvertovat objekt na String, například:

```
System.out.println(clovek);
```



Porovnávání: == vs. equals()

- Operátor ==

- Zjišťuje, jestli dvě reference ukazují na stejnou část paměti (jedná se o stejný objekt).

```
Clovek clovek = new Clovek();
```

```
Clovek jirka = clovek;
```

```
System.out.println(clovek == jirka); // vrati true
```

- Metoda equals()

- Slouží k porovnání obsahu dvou objektů (jestli obsahují stejnou informaci).

```
String s1 = new String("test");
```

```
String s2 = new String("test");
```

```
System.out.println(s1 == s2); // vrati false
```

```
System.out.println(s1.equals(s2)); // vrati true
```



String I.

- String je objektem, nikoli primitivním datovým typem. Je možné s ním ale pracovat oběma způsoby. Porovnávání takových objektů pomocí `==` ale dává zvláštní výsledky:

```
String s1 = "test";  
String s2 = new String("test");  
String s3 = "test";  
System.out.println(s1 == s2); // vrati false  
System.out.println(s1 == s3); // vrati true  
System.out.println(s1.equals(s2)); // vrati true  
System.out.println(s2.equals(s3)); // vrati true
```

- Při porovnávání Stringů byste měli vždy používat metodu `equals()`, abyste se tomuto problému vyhnuli!

String II.

- V Javě existují dvě části paměti:

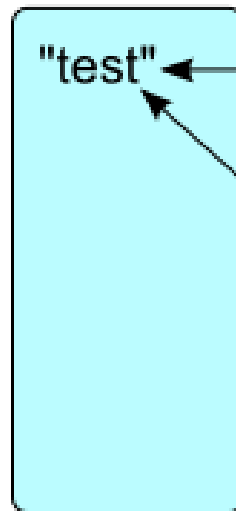
- **Zásobník (stack)**

- velice rychlý a optimalizovaný, ale malý.

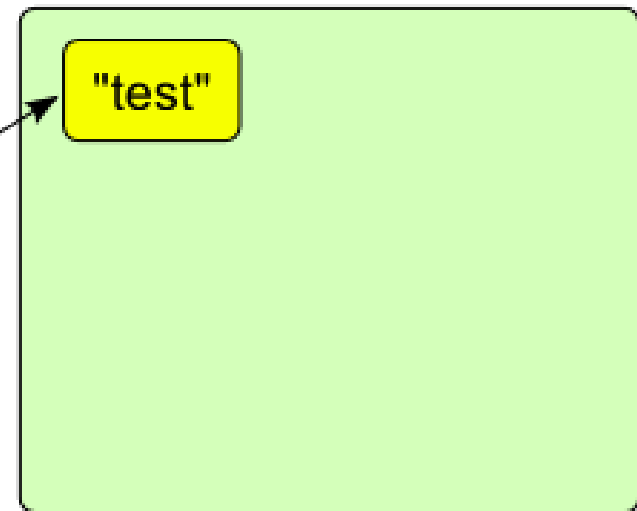
- **Halda (heap)**

- velká, ale operace na ní jsou pomalejší.

zásobník



halda



- Z důvodu optimalizace se všechny primitivní datové typy (a String, který byl vytvořen jako primitivní datový typ) ukládají do zásobníku. Všechny objekty se ukládají do haldy.



String III. - best practices

- Pro porovnávání Stringů používat vždy metodu `equals()`.
- Nepoužívat String objektovým způsobem. Je to pomalejší a nemá to většinou žádný přínos.
- Při spojování Stringů pomocí operátoru `+` dochází k neefektivní tvorbě Stringů v zásobníku. Lepší je použít třídu `StringBuilder` (nebo starší a pomalejší `StringBuffer`):

```
String s = "Jirka" + " " + "Pinkas"; // neefektivní
StringBuilder sb = new StringBuilder();
sb.append("Jirka");
sb.append(" ");
sb.append("Pinkas");
String s2 = sb.toString();
```



Modifikátory viditelnosti I.

- V Javě existují čtyři modifikátory viditelnosti, pomocí kterých je možné omezit viditelnost třídy / metody / atributu (dále budou souhrnně označovány jako elementy):
 - **public** – Element je viditelný v rámci celé aplikace či jiných aplikací, které ji mohou používat jako knihovnu.
 - **private** – Element je viditelný pouze v rámci třídy, ve které je definovaný.
 - **default** – Element je viditelný v rámci balíčku, ve kterém je definovaný. **Default není klíčové slovo**, ale použijete jej, když neuvedete žádný z jiných modifikátorů viditelnosti.
 - **protected** – Element je viditelný v rámci balíčku, ve kterém je definovaný a navíc v rámci všech potomků třídy, ve které se element nachází (čili v rámci dědické hierarchie).



Modifikátory viditelnosti II.

- Best practices:

- **public** – velice dobře si rozmyslete, co nastavíte jako public. Pokud Váš kus kódu bude využívat i někdo jiný, tak bude mít k dispozici i Vaše veškeré public elementy. A pokud navíc nebudou dobře odokumentované, tak bude mít problém Váš kód vůbec používat.
- **private** – všechny atributy by měly být private (nebo protected). Také vše ostatní co chcete skrýt uvnitř třídy by mělo být private (pomocné elementy), aby k tomu nikdo jiný neměl přístup.
- **default** – používá se hodně při testování Vašich tříd pomocí JUnit, aby Vaše elementy byly v rámci balíčku dostupné testovacím třídám.
- **protected** – moc se nepoužívá.



Gettery / settery I.

- Atributy by měly mít vždy viditelnost `private` nebo `protected`, aby k nim nebyl z jiné třídy přímý přístup. Jak ale získat či nastavit hodnotu atributu, který není `public`? Pomocí getteru a setteru.
- Gettery a settery jsou standardizované metody, které slouží pro práci s atributy:

```
private String jmeno;  
public String getJmeno() {  
    return jmeno;  
}  
public void setJmeno(String jmeno) {  
    this.jmeno = jmeno;  
}
```



Gettery / settery II.

- V případě, že je atribut typu boolean, můžete se setkat s následující variantou getteru:

```
private boolean active;  
public boolean isActive() {  
    return active;  
}
```

- Settery se dají použít pro ošetření vstupů:

```
public void setPohlavi(String pohlavi) {  
    if(!"m".equals(pohlavi) && !"z".equals(pohlavi)) {  
        throw new RuntimeException  
            ("Pohlaví může být pouze: [m|z]"); }  
    this.pohlavi = pohlavi;  
}
```



Polymorfismus I.

- V rámci dědičnosti je ještě jeden silný mechanismus, který jsme dosud nezmínili – **polymorfismus**. V potomkovi lze předefinovat chování metody definované v předkovi.
- Příklad:
 - Ve třídě `Object` je definována metoda `toString()`. Tato metoda se zavolá pokaždé, když chceme textovou reprezentaci objektu. Například když zavoláme následující kus kódu:

```
System.out.println(clovek);
```
 - V současnosti se vypíše nějaký defaultní text, který nám ale nevyhovuje a chceme ho změnit. Místo tohoto textu budeme chtít vypsát na obrazovku jméno a výšku člověka.



Polymorfizmus II.

- Ve třídě Object má metoda toString() následující implementaci:

```
public String toString() {  
    return getClass().getName() + "@" +  
        Integer.toHexString(hashCode());  
}
```

- Pro změnu tohoto chování musíme vytvořit v třídě Clovek tuto metodu:

```
@Override  
public String toString() {  
    return jmeno + ", " + vyska + " cm";  
}
```

- Tím změníme chování metody toString() v předkovi.



Anotace

- Na předcházející stránce byla použita anotace `@Override`. Tato anotace je nepovinná.
- Anotace jsou v jazyce Java od verze 5 a jedná se o rozšíření jazyka o další informace.
- Pomocí anotace `@Override` říkáte, že chcete pomocí polymorfizmu předefinovat chování metody. Tuto anotaci není možné použít na metodě, která neodpovídá stejně pojmenované metodě v předkovi.
- Existují i další anotace – například anotace `@Deprecated`, která slouží k označení metod, které jsou zastaralé a neměly by se používat.
- S anotacemi se setkáte v celé řadě pokročilejších frameworků.



Statické členské složky tříd I.

- Statické členy tříd (atributy, metody, vnořené třídy) se definují s klíčovým slovem **static**, např.:
`private static int pocetInstanci;`
- Patří ke třídě, nikoliv k instanci třídy (objektu).
Existují, i když není vytvořena žádná instance třídy.
- Ke statickým členům lze přistupovat přes jméno třídy:
`Osoba.getPocetInstanci();`
`Proces.PRIORITY;`
- Ve statických metodách lze používat pouze statické atributy, v instančních metodách všechny atributy.



Statické členské složky tříd II.

- Statické metody a atributy byste měli používat co nejméně (v opačném případě neprogramujete objektově).
- Vhodné použití statických metod a atributů:
 - Atributy: konstanty (viz. další snímek), nebo jako součást Singletonu.
 - Metody: utility metody (například metody z třídy `java.lang.Math`), nebo jako součást Singletonu.
 - <http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html>
- Singleton:
 - http://en.wikipedia.org/wiki/Singleton_pattern



Final

- K definování konstant se používá klíčové slovo **final** (obvykle navíc kombinované s klíčovým slovem **static**).
- Konstanty se pojmenovávají VELKÝMI_PÍSMENY.

Příklad:

```
public static final String POHLAVI_MUZ = "m";  
public static final String POHLAVI_ZENA = "z";
```

- Klíčové slovo **final** má ještě širší použití:
 - Při definování **final** u třídy zamezíte tvorbu potomků takové třídy.
 - Při definování **final** u metody zamezíte v potomcích třídy predefinování metody pomocí polymorfizmu.



Cykly I.

- **Cyklus s podmínkou na začátku:**

```
while (condition) {  
    // tělo cyklu - nemusí proběhnout ani jednou  
}
```

← Složené závorky jsou nepovinné, ale vždy je používejte!

- **Cyklus s podmínkou na konci:**

```
do {  
    // tělo cyklu - proběhne minimálně jednou  
} while (condition);
```

- **Pozor! Podmínka musí být typu boolean (true nebo false)**



Cykly II.

- **For cyklus** – pro pevně daný počet opakování:

```
for (výraz start; výraz ukončení; výraz iterování) {  
    // běh cyklu  
}
```

// příklad:

```
for (int i = 0; i < pole.length; i++) {  
  
}
```

- **Foreach cyklus** – pro sekvenční průchod všech prvků:

```
for (TypPrvku promenna : pole) {  
    // běh cyklu  
}
```



Podmíněný příkaz

```
if(condition1) {  
    // když je condition1 rovno true  
} else if(condition2) {  
    // když je condition2 rovno true  
}  
... // else if se může opakovat  
else {  
    // když nebyla splněna ani jedna podmínka  
}
```

- else if a else bloky jsou nepovinné



Speciální příkazy uvnitř cyklu

- Uvnitř cyklu můžete použít tyto speciální příkazy:
 - `break;` - ukončí provádění cyklu
 - `continue;` - skočí na konec aktuálně prováděného cyklu



Switch I.

- Tento příkaz se používá jako náhrada složitých podmíněných příkazů:

```
int key = 10;
switch (key) {
    case 10: příkazy ... break;
    case 20: příkazy ... break;
    default: příkazy ... break;
}
```

- Dovnitř vstupuje klíč, který se porovnává s hodnotami uvedenými za klíčovým slovem case. V případě, že nějaký case odpovídá klíči pokračuje se vykonáváním příkazů dokud se nenarazí na klíčové slovo break. Poté se vykonávání příkazu switch ukončí.



Switch II.

- V příkazu `switch` je povinné mít minimálně jeden `case`.
- V případě, že se nevykoná žádný `case`, vykonají se příkazy za klíčovým slovem `default`.
- Do Java SE 7 byl `switch` velice omezený, jako klíč podporoval pouze atribut typu `int` nebo jiný, který je možné jednoznačně na `int` přetypovat: `byte`, `short` a `char`. Od Java SE 7 je možné porovnávat pomocí `switch` příkazu i objekty typu `String`.



Statické pole I.

- Statické pole má pevný počet prvků, který není možné za běhu změnit (toto omezení nemá dynamické pole, viz. dále).
- Dva druhy zápisu definice pole:
 - `TypPrvku [] nazevPole;` – tento způsob je doporučený
 - `TypPrvku nazevPole [];`
- V poli může být jakýkoli typ prvků (`int`, `String`, `Clovek`, ...)
- Pole je vždy číslované od 0 do $n-1$, kde n je počet prvků.
 - `Ucastnik [] ucastnici = new Ucastnik [10];`
 - Vytvoří se pole s počtem prvků 10 přístupných přes index 0 až 9.
 - Hodnoty prvků v poli jsou inicializované stejně jako při implicitní inicializaci atributů.



Statické pole II.

- Nastavení prvku na index v poli:
 - `ucastnici[0] = ucastnik1;`
 - `ucastnici[1] = ucastnik2;`
- Získání prvku z indexu v poli:
 - `ucastnici[0];`
- Více rozměrů pole:
 - `int[][] matice2d;`
 - `int[][][] matice3d;`
- Jednodušší tvorba pole:
 - `int[] pole = new int [] {1,2,3,4,5}`
 - Při vytvoření pole se rovnou provede inicializace. Pole má takovou velikost, kolik prvků dáme do složených závorek.



Statické pole – arraycopy()

- Pokud potřebujete zkopírovat hodnoty z jednoho pole do druhého, pak můžete použít `System.arraycopy()`:

```
char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',  
                    'i', 'n', 'a', 't', 'e', 'd' };  
  
char[] copyTo = new char[7];  
System.arraycopy(copyFrom, 2, copyTo, 0, 7);  
System.out.println(new String(copyTo));
```



Statické pole – třída Arrays

- Třída `java.util.Arrays` obsahuje několik pokročilejších metod pro práci se statickým polem:
 - `binarySearch()` - binární vyhledávání v poli
 - `copyOf()` - pokročilejší kopírování pole
 - `equals()` - zjistí, jestli dvě pole obsahují stejné prvky
 - `fill()` - vyplní pole zadanými hodnotami
 - `sort()` - utřídí pole
 - `toString()` - provede transformaci pole na `String`
 - ...



Statické pole – varargs I.

- Statické pole můžete použít jako parametr metody. Bez použití varargs:

```
public static String constructFullName(String[] parts) {  
    StringBuilder fullName = new StringBuilder();  
    for (String part : parts) {  
        fullName.append(part);  
        fullName.append(" ");  
    }  
    return fullName.toString().trim();  
}  
  
public static void main(String[] args) {  
    String fullName = constructFullName(new String[] {"Jirka", "Pinkas"});  
    System.out.println(fullName);  
}
```

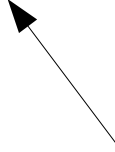


Statické pole – varargs II.

- Nebo můžete použít varargs (od Java SE 5):

```
public static String constructFullName(String ... parts) {  
    StringBuilder fullName = new StringBuilder();  
    for (String part : parts) {  
        fullName.append(part);  
        fullName.append(" ");  
    }  
    return fullName.toString().trim();  
}
```

Varargs parametr



```
public static void main(String[] args) {  
    String fullName = constructFullName("Jirka", "Pinkas");  
    System.out.println(fullName);  
}
```

Použití





Výčtový typ – enum

- V Javě existuje výčtový typ:

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,  
    THURSDAY, FRIDAY, SATURDAY  
}
```

- <http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>