Základy jazyka Java

(pro programátory v C++)

http://cgg.ms.mff.cuni.cz/~pepca/

Zdroje, literatura

- Bruce Eckel: ***Thinking in Java, 3rd edition***, *http://www.bruceeckel.com/* (starší vydání vyšlo též v češtině: nakladatelství Grada, 2-dílná kniha)
- Rebecca Hasti: ***Java Tutorial***, University of Wisconsin, *http://www.cs.wisc.edu/~hasti/cs368/JavaTutorial/*
- Marvin Solomon: ***Java for C++ programmers***, University of Wisconsin, *http://www.cs.wisc.edu/~solomon/cs537/java-tutorial.html*
- Bruce Eckel: ***Thinking in Patterns (with Java)***, *http://www.bruceeckel.com/*

On-line zdroje

- Stránky společnosti ***SUN Microsystems***:
  http://java.sun.com/
  http://developer.java.sun.com/
  (JDK download, množství dokumentace, tutorials, ..)
- **NetBeans IDE**: *http://www.netbeans.org/*
  (IDE zdarma, prapůvod na MFF, pracuje s libovolným JDK)
- **Borland jBuilder**:
  http://www.borland.com/jbuilder/
  (další populární IDE, škála verzí od osobní /zdarma/ až po profesionální)

On-line zdroje

- **Eclipse**: *http://www.eclipse.org/*
  (univerzální IDE zdarma, velmi populární)
- **IBM Developer kits**: vývojové prostředí zdarma (pro Windows, Linux, OS/2, AIX), asi nejrychlejší JVM
  http://www-106.ibm.com/developerworks/java/jdk/
- **BEA WebLogic JRockit**: *http://www.bea.com/*
  (další IDE s možností provozu zdarma)
- **jGRASP**: *http://www.jgrasp.org/*
  (IDE vhodné pro začátečníky, umí UML)
- **Kaffé**: *http://www.kaffe.org/*
  (neoficiální „JVM“ pod GNU licencí)

Základem syntaxe je C++

- většina operátorů i programových konstrukcí je přímo převzata z jazyka C++
  - příklad `S01Sort.java`
- snaha o maximální **bezpečnost** výpočtů v Javě:
  - zcela chybí „nečistá” práce s ukazateli
  - běh Javovských programů se může odehrávat v dobře izolovaném prostředí („sandbox”) – vhodné pro programování na Internetu
  - **interpretace** mezikódu („byte-code”) – možnost zavedení různých „run-time” kontrol..

Základní rysy jazyka Java

- důsledně **objektově-orientovaný** jazyk
  - příklad `S02Hello.java`
  - neexistují žádné globální proměnné ani funkce
  - příklad `S03Sort.java`
- maximální důraz je kladen na **bezpečnost**
- příprava spustitelného kódu v Javě:
  - `Test.java` → `Test.class`
    Překladač /javac/ vytváří tzv. byte-kód, **interpretuje** se pomocí JVM (Java Virtual Machine) /java/
- srovnej s postupem u C, C++:
  - `test.cc` → `test.o` → `test[.exe]`

Hlavní rozdíly Java vs. C++

- Java nemá preprocesor
- Java nemá šablony (templates) tak silné jako v C++
  - JDK 1.5 už má „generics”
- Java nemá typ „union” (bezpečnost)
  - ani „struct”, nový typ může být polem nebo se konstruuje pomocí „class”
- Java nedovoluje předefinovat operátory
- Java nedělá nebezpečné implicitní přetypování
  - přípustné konverze umožňuje, většinou se musí psát explicitně (výjimky: „upcast” číselných typů, ..)
- Java nemá nebezpečnou práci s ukazateli (&, ++)

Hlavní rozdíly Java vs. C++

- Java nemá příkaz „goto”
  - ale umí používat label u příkazů „break” a „continue”
- hierarchie tříd v Javě začíná třídou „Object”
  - univerzální předek
  - možnost používat univerzální reference
- Java má automatickou správu paměti (garbage coll.)
  - instance objektů a polí se v programu nemusí uvolňovat
- Java má slabší koncept destruktoru (finalize)
  - není zaručeno, kdy (a zda vůbec) bude zavolán
- Java má vestavěný mechanismus výjimek (Exception)
  - syntaktické kontroly jejich deklarace a ošetřování

Hlavní rozdíly Java vs. C++

- Java má podporu pro více výpočetních vláken (threads)
  - jazyk obsahuje přímo prostředky pro synchronizaci
- Java má vestavěný typ String
  - je součástí objektové hierarchie
- Java obsahuje zapouzdření všech jednoduchých typů
  - Boolean, Char, Int, ...
- všechny parametry se v Javě předávají hodnotou
  - u polí a objektů se jedná o reference, takto lze i přes parametry vracet spočítané hodnoty ven
- všechny metody se v Javě volají virtuálně (late binding)
  - základem je textový identifikátor (signatura) metody

Hlavní rozdíly Java vs. C++

- Java umožňuje přímo zacházet s objekty jazyka (RTTI, Reflection)
  - indentifikátory metod a jejich parametry
  - práce s třídami a protokoly
  - dynamické nahrávání nových tříd (i za běhu programu)
- Java je dobře přenositelná
  - UNIX, Windows, MacOS, OS/2, ..
  - mezikód je úplně přenositelný
- standarně jsou dodávány knihovny pro GUI, Internet, XML, regulární výrazy, rastrovou grafiku, kompresi, ...
  - grafické uživatelské rozhraní Swing (nezávislé na OS)
  - systém komponent JavaBeans

Rychlost JVM

- při použití **JIT** („Just In Time“) **kompilace** je srovnatelná s překládaným C++
- malá studie za pomoci **VBench/jBench** benchmarků (*http://cgg.ms.mff.cuni.cz/~pepca/bench/*):
  - Asus A7V600, Amd Athlon XP 2500+ (interně 2.0GHz), 512MB RAM (333MHz), disk IBM 40GB, WindowsXP Ho.
  - výběr tří reprezentativních testů (celkem jich je 15):
    - quick-sort pole délky 64MB (double[8*1024*1024])
    - hledání arbitrážní sekvence (dynamické programování; matice 12×12, rekurze, násobení, porovnávání, kopie pole)
    - waveletová transformace – lifting (T-S transformace, celočíselná aritmetika, aditivní operace, práce v poli)

Rychlost – výsledky

- dva překladače **C++** a čtyři **JVM**:
  - Intel Compiler 5.0.1, full optimizations
  - Microsoft Visual C++ 6.0, full optimizations
  - IBM JDK 1.3.1
  - BAE JRockit 1.4.1
  - Microsoft JVM 5.00.3802 JIT
  - Sun JDK 1.4.2beta HotSpot

system \ test	quick-sort	arbitrage	lifting
Intel C++	2.69s (100%)	35.08s (100%)	13.80s (100%)
Visual C++ 6.0	3.12s (116%)	42.83s (122%)	13.98s (101%)
IBM JDK 1.3.1	3.22s (120%)	41.56s (118%)	15.64s (113%)
BAE JRockit	3.47s (129%)	48.30s (138%)	16.45s (119%)
Microsoft JVM	4.11s (153%)	56.31s (161%)	18.48s (134%)
Sun 1.4.2beta	3.95s (147%)	61.09s (174%)	18.83s (136%)

Plán dalšího výkladu

- základní datové typy
- operátory, literály, programové konstrukce
- třídy I: dědičnost, polymorfismus, protokoly, ..
- modularita (packages), přístup, viditelnost
- vestavěné třídy (Object, String, Integer, BigInteger, ...)
- správa paměti, úklid paměti, destrukce objektů
- výjimky a jejich ošetřování
- multi-threading, synchronizace
- třídy II: vnitřní a anonymní třídy, statické tělo..
- RTTI, Reflection, nahrávání za běhu
- kontejnery, iterátory, ...

Základní datové typy

- osm **jednoduchých typů** + **reference** (odkaz na objekt – instanci třídy nebo pole)
- jednoduché typy:
  - **boolean** (hodnoty „**true**" a „**false**", nekompatibilní s číselnými typy, ukládá se většinou do 8 bitů)
  - **char** (jeden znak v UNICODE kódování, zabírá 16 bitů)
  - celočíselné typy (pouze se znaménkem!): **byte**, **short**, **int**, **long** (8, 16, 32 a 64 bitů)
  - plovoucí desetinná čárka: **float**, **double** (32 a 64 bitů, standard IEEE754)
- fiktivní typ **void** (nelze deklarovat proměnnou)

Typové konverze

- **implicitní** – při převodu na cílový typ, který je **nadmnožinou** zdrojového
  - int i = 12; long j = i; double d = i;
- **explicitní** – při převodu opačným směrem
  - je třeba pamatovat na ztrátu informace (přesnosti)
  - syntax jako v C++
  - i = (int)d; i = (int)j;
  - run-time při ztrátě nebo zkreslení dat zůstává tichý (pozor zejména na ořezání nejvýznamnějších bitů!)

Reference

- typ „**reference**”
  - odkaz na instanci třídy nebo pole
  - de facto je to „ukazatel”, jen nad ním nemáme takovou kontrolu, jako v C[++]
  - na instance tříd nebo pole se vždy přistupuje přes referenci
    - každá instance třídy nebo pole se alokuje na haldě (heap)
- **alokace** objektu/pole: operátor „**new**”
  - implicitní alokace při inicializaci pole
- **uvolnění** objektu/pole je zcela v režii správce paměti
  - systémový „garbage collector”

Pole = reference na objekt „pole“

- pole se **deklaruje** podobně jako v C++
  - „int[] a” je ekvivalentní „int a[]”
  - v Javě se tím však nealokuje místo v paměti!
- **alokace** pole v paměti, inicializace dat na samé „0”
  - a = new int[10];
  - lze použít opakovaně: „staré” pole se přestane používat
- **deklarace** s okamžitou **alokací** (+ inicializací)
  - int[] a = new int[10]; // inicializace na samé „0”
  - int[] a = { 12, 0, 35, 7, 16, 144, 99, -3, -8, 2 };

Pole

- pole se indexují vždy **od nuly** (jako v C[++])
- při každém přístupu do pole se **kontrolují meze!**
- **vícerozměrná pole** jsou vlastně „pole polí”
  - „int[][] m” (≡ „int m[][]”), „int[][][] v” (≡ „int v[][][]”), ...
  - interně se implementují jako pole referencí na méněrozměrná pole, atd.
  - v Javě lze vytvořit skutečně trojúhelníkovou matici!
- každé pole má člen „public final int **length**” obsahující velikost pole
  - for (int i = 0; i < a.length; i++) ...

Pole – přiřazování

- **přiřazovací příkaz** mezi poli pouze ukáže oběma proměnnými (referencemi) na stejný objekt
  - pozor – dále se data budou **sdílet !**
- **kopírování dat** v polích
  - System.**arraycopy**(**src**, **srcPos**, **dst**, **dstPos**, **count**)
  - „**src**" a „**dst**" jsou pole, „**srcPos**" a „**dstPos**" indexy a „**count**" počet kopírovaných prvků
  - pokud jde o pole složitějších typů, neprovádí se kopie do hloubky („deep copy"), ale jen kopie referencí
  - lze posunovat položky jediného pole („src == dst")

Operátory

- téměř kompletní sada operátorů z jazyka C++
  - chybí „sizeof”, čárka se používá jen v cyklu „for”
- aritmetické operátory
  - + - * / % unární „-”
  - „+” je použit též u vestavěného typu „String”
  - auto-inkrement a auto-dekrement „++”, „--”
- přiřazování „=”
  - každý binární operátor „#” má též formu „#=”
- relační operátory
  - == != < > <= >=

Operátory

- logické operátory (nad typem „boolean”)
  - **&& || !**
  - úsporné vyhodnocování !
- bitové operátory (nad číselnými typy)
  - **& | ^ ! ~**
  - lze je použít na „boolean”, způsobí úplné vyhodnocování
- bitové posuny (bit-shifts)
  - << >> >>> (logický posun, bez znaménkového rozšíření)
- podmíněný výraz „**? :**” (boolean-expr ? exp1 : exp2)

Literály

- jako v jazyce C[++]
  - celočíselné typy: prefixy „**0**”, „**0x**”, suffix „**L**”
  - pohyblivá čárka: suffixy „**F**” a „**D**”
  - znak (char): uzavírá se mezi apostrofy **'x'**
  - řetězec (String): uzavírá se mezi uvozovky **”xyz”**
  - „escape“ sekvence jsou podobné C[++]:
    - **\b \t \n \f \r \” \' **
    - oktalové kódy: **\0** až **\377**
    - hexadecimálně definovaný znak (UNICODE): **\uXXXX** (XXXX jsou hexadecimální číslice)
  - speciální literály: **false true null**

Klíčová slova

- Java verze 2 má 48 rezervovaných slov (nesmějí být použity jako identifikátory):
  - abstract boolean break byte case catch char class const continue default do double else extends final finally float for goto if implements import instanceof int interface long native new package private protected public return short static strictfp super switch synchronized this throw throws transient try void volatile while
  - „const“ a „goto“ nejsou aktuálně používány jako klíčová slova, ale nesmí se užívat jako identifikátory!

Identifikátory

- identifikátor se musí lišit od všech klíčových slov a speciálních literálů
  - první znak identifikátoru: „písmeno“, např. '**A**' až '**Z**', '**a**' až '**z**', '**_**', '**$**' (ale i písmena v jiných jazycích..)
  - další znaky mohou obsahovat písmena i číslice
  - možnost psát identifikátory v národních abecedách (česky, řecky, čínsky, japonsky, ..)
    - **totoJeČeskýIdentifikátor** αρετη **заведение12**

Řídící příkazy

- téměř kompletní sada z C++ (chybí pouze „goto”)
  - **if** (boolean-expr) statement [**else** statement2]
  - **while** (boolean-expr) statement
  - **do** statement **while** (boolean-expr)
  - **for** (init; boolean-expr; step) statement
  - **break** [label], **continue** [label]
  - **switch** (int-expr) statement
  - **case** int-val: statements; **default**: statements;
  - **return** value

Třídy, protokoly (interface)

- syntakticky se definice tříd podobá C++
  - nejsou odděleny deklarace od definic (kód metod se píše rovnou do těla třídy)
  - zcela chybí destruktory („finalizace“ je mnohem slabší)
  - atributy přístupu (public, ..) se uvádějí u každého členu
- tři typy „tříd“:
  - **interface** (protokol) – obsahuje jen hlavičky metod nebo konstantní členy
  - **abstract class** – třída, od které nelze vytvářet instance (může obsahovat abstraktní metody)
  - **class** – konkrétní třída, jejíž instance se používají

Dědičnost

- **interface** (protokol)
  - definuje „způsob komunikace s třídou“, množinu zpráv, kterým třída musí rozumět, ...
  - konkrétní třídy mohou „implementovat“ několik protokolů
- je dovolena pouze **jednoduchá dědičnost** mezi třídami
  - tím se eliminuje problém s rozhodováním, která implementace metody se má použít při vícenásobné dědičnosti (a problémy s virtuální dědičností) v C++
  - omezení OO návrhu není tak velké ...

Přístup, viditelnost

- „**public**“
  - člen/metodu vidí úplně všichni
- (package)
  - implicitní přístupová metoda, pro objekty z téhož balíku (**package**)
- „**protected**“
  - pouze pro potomky (dědice)
- „**private**“
  - pouze pro metody téže třídy (nelze ji tedy přeprogramovat u potomka – viz „final“)

Další atributy

- „**static**“
  - člen či metoda není svázán s konkrétní instancí třídy, ale patří k třídě jako celku („class data/methods“)
  - existuje od okamžiku, kdy se třída začne používat (viz „class loader“)
- „**final**“
  - **proměnná**: po prvním přiřazení se již nesmí měnit, je to vlastně konstanta (asi jako „const“ v C++)
  - může se přiřadit až v konstruktoru (v každém!)
  - pozor na reference! Referencovaný objekt se měnit smí!
  - **metoda**: nelze ji přeprogramovat u potomka (optimalizace – může se vkládat „inline“)

Další atributy

- „**abstract**“
  - neexistuje implementace této metody (jako „= NULL“ v C++)
  - konkrétní potomci ji nutně musí přeprogramovat
  - třída obsahující abstraktní metodu musí být sama označena jako abstraktní!
- „**volatile**“
  - používá se pouze u proměnných a při paralelním počítání (multi-threading)
  - uživatel takové proměnné nesmí pracovat s její lokální kopií, nebo musí pokaždé kopii synchronizovat s originálem (levnější mechanismus než „synchronize“)

Přístup k členům

- **tečková notace**
  - každá proměnná typu „objekt“ je referencí, tečka zde vlastně nahrazuje operátor „->“ z C++
  - příklad:

```
MyClass c = new MyClass();
c.member = 12;
c.set(“color“,0xFF00FF);
```

- **statické** členy a metody:
  - před tečkou se píše jméno třídy (viz „::“ v C++):

```
MyClass.setClassProperty(“weight“,2.5);
MyClass.ref = null;
```

Konstruktory

- koncept velice podobný C++
  - konstruktor nemá návratovou hodnotu, nelze v něm žádným způsobem zabránit vytvoření instance třídy
  - datové členy neinicializované v konstruktoru mají nulovou počáteční hodnotu (viz inicializace polí)
  - není-li definován žádný konstruktor, překladač doplní tzv. **implicitní konstruktor** (bez parametrů i příkazů)
  - je-li definován alespoň jeden explicitní konstruktor, implicitní konstruktor bez parametrů se nevytvoří
  - konstruktorů lze definovat více, musí se navzájem lišit počtem a/nebo typem parametrů!

Volání mezi konstruktory

- **vzájemné volání** mezi konstruktory jedné třídy
  - musí být prvním příkazem konstruktoru
  - použije se slovo „**this**“ (příklad: „{ this(12,“red“); … }“)
- volání **konstruktoru předka**
  - vlastně jde jen o upřesnění, který z jeho konstruktorů se má použít (vždy se některý musí zavolat !)
  - musí být prvním příkazem konstruktoru
  - použije se slovo „**super**“ (příklad: „{ super(-1); … }“)
  - použitý nadřízený konstruktor musí být přístupný (public, protected, ve stejné package, apod.)

Inicializace dat

- incializátor se uvede přímo **v deklaraci** proměnné

```
int size = 0;
double now = globalTime.getDoubleTime();
```

  - často se používá pro „final“ proměnné

- anonymní **blok kódu**
  - blok kódu uzavřený mezi složené závorky
  - takových bloků může definice třídy obsahovat několik

```
double sinA, cosA;
{
  double alpha = globalTime.getDoubleTime() * FREQ;
  sinA = Math.sin(alpha);
  cosA = Math.cos(alpha);
}
```

Parametry metod

- nelze definovat **implicitní hodnotu** parametru
- nelze používat **proměnný počet** parametrů („ ... “)
- předávání parametrů **hodnotou**
  - parametr je **lokální proměnnou** metody, do které se na začátku okopíruje hodnota aktuálního parametru
  - pozor na **objekty** (instance tříd, pole) – přiřadí se pouze reference! (objekt je sdílen volající a volanou metodou)
- „**final**“ parametry
  - nelze uvnitř metody měnit (přiřazovat do nich)
  - pozor na **objekty** – nelze přiřazovat referenci, ale lze měnit **vnitřní stav** předávaného objektu!

Dědičnost

- **třída** (class) může mít pouze **jednu třídu** jako předka
  - **class** DerivedClass **extends** MyClass { ... }
  - pokud není předchůdce uveden explicitně, stane se jím univerzální předek **Object** (vlastně java.lang.Object)
- **třída** (class) nebo **protokol** (interface) mohou implementovat **libovolné množství protokolů** (interface)
  - **class** AnotherClass **implements** Drawable, Cloneable { ... }
  - **interface** Drawable **extends** Serializable { ... }

Předefinování metod

- třída může **předefinovat implementaci** libovolných metod
  - výjimka: je-li metoda označena „**final**“, **nesmí** ji žádný z potomků **předefinovat** !
  - všechny metody dané třídy (i definované u předků) používají tuto novou implementaci !
    - pozor na **konstruktory !** Datové položky potomků ještě nejsou korektně inicializovány (jen default = 0) !
  - výjimka: „**private**“ metoda se sice novou implementací „překryje“, předchůdce ji však stále používá !
    - implicitní atribut „final“

Předefinování metod

- pozor na **přesný formát hlavičky** – častý zdroj chyb
  - překladač na to nemůže upozornit (viz **přetěžování**)
- vyvolání metody, jak ji používá **bezprostřední předek**, se dělá pomocí „**super**“
  - super.draw(12,-3)

- **přetěžování** metod (overloading)
  - několik metod může mít **stejný identifikátor**
  - musí se lišit v sadě parametrů (počet a/nebo typ)
  - při použití musí být jednoznačné, kterou variantu zavolat
    - typové konverze

Předefinování dat

- libovolné **datové položky** lze předefinovat
  - to platí i pro data označená u předka jako „**final**“ (konstanty) !
    - z konstanty (u předka) se může „stát“ obyčejná proměnná
  - ve skutečnosti vzniknou **dvě různé datové položky**
    - viditelnost: nová datová položka **překryje** tu původní
  - metody definované v určité třídě používají datovou položku známou (přístupnou) v této třídě
    - tj. není možné „podsunout“ předkovi jiná data – to by se k datům muselo důsledně přistupovat pomocí metod

Implementace protokolů

- každá **konkrétní třída** (class) musí implementovat všechny metody obsažené ve **sjednocení protokolů**, které se zavázala implementovat
  - protokoly mohou mít **neprázdný průnik**
    - model shodné sémantiky (stejná hlavička metody ≡ stejný význam)
  - implementace metod mohou být **zděděné** po předcích (i když se tam daný protokol nemusel deklarovat)
    - rozhodování je založeno výhradně na textové hlavičce metody
  - absence jakékoli implementace způsobí již chybu **při překladu**

Balíky (packages)

- mechanismus **omezující viditelnost** identifikátorů tříd (jediných globálních objektů v Javě)
  - přehlednost rozsáhlejších projektů, modularizace
  - jméno balíku souvisí s **adresářem**, ve kterém jsou soubory umístěny (cz.cuni.jagrlib … cz/cuni/jagrlib)
- každá třída je součástí nějakého balíku (package)
  - celý zdrojový soubor „xxx.java“ náleží do jediného balíku
  - explicitní deklarace „**package yyy**“ musí být prvním příkazem zdrojového souboru!
  - chybí-li explicitní deklarace, patří celý soubor do tzv. **anonymního balíku** (jediného v celém projektu)

Balíky

- **jméno balíku**: posloupnost jmen oddělených tečkami (hierarchie – adresáře na disku)
  - „java.lang.ref“, „org.w3c.dom“, „cz.cuni.jagrlib.piece“
  - konvence: začátek jména by měl odpovídat Internetové doméně (čtené odzadu)
  - konvence: jména balíků se píší malými písmeny (dříve se používala velké písmena)
- **viditelnost** v rámci stejného balíku
  - implicitní přístupová metoda (není-li uveden žádný atribut „public“, „private“ nebo „protected“)

Balíky

- použití třídy z **jiného balíku**
  - **tečková notace**: „java.lang.ref.Reference“, „cz.cuni.jagrlib.Plug“, atd.
- příkaz „**import**“ umožňuje **zkrácený zápis**
  - **import** fully.qualified.package.name.*;
    - importuje všechny veřejné třídy z daného balíku
  - **import** fully.qualified.package.name.**ClassName**;
    - importuje pouze danou třídu
    - vždy je samozřejmě třeba používat identifikátor třídy: ClassName.memberVariable
- implicitně je importován interní balík **java.lang.***

Veřejná třída

- jeden zdrojový soubor může obsahovat maximálně jednu **veřejnou definici třídy** („public class“ nebo „public interface“)
  - taková třída se musí jmenovat stejně jako zdrojový soubor. Příklad: „**MyClass.java**“ smí obsahovat definici třídy „**public class MyClass** ... { ... }“
  - kromě veřejné třídy může být ve zdrojovém souboru libovolné množství dalších tříd neoznačených „public“
  - pozn: „neveřejné“ třídy jsou viditelné pouze v balíku (package), kde jsou definovány

Překlad programu

- **překladač** („**javac**“) čte **zdrojové soubory** v jazyku Java (**.java**) a překládá je do **byte-kódu** (**.class**)
  - pro **každou třídu** je založen nový **.class** soubor (i pro třídy definované v jediném zdrojovém souboru)
  - **.class** soubory se ukládají do adresářů podle **balíků**, do kterých patří
    - interpret je pak může snadno najít !
- **hromadný překlad**
  - překladač umí najednou přeložit všechny třídy, které na sebe odkazují (automatické řešení závislostí, viz „make“)
  - překlad celého balíku: **javac cz.cuni.jagrlib.***

Běh programu

- **interpret** („**java**“) spouští zadanou třídu
  - tato třída musí obsahovat metodu
    public static void main (String[] args) { ... }
  - spuštění interpretu (JVM): **java MyClass**
- **hledání .class souborů**
  - až při běhu programu – teprve je-li daná třída potřeba – se hledá a zavádí do paměti její kód (.class)
  - jednoduché programy (jediný – anonymní balík) mohou být umístěny v jednom adresáři
  - jinak se při hledání používá analogie hierarchií balíků a adresářů operačního systému

Hledání .class souborů

- proměnná **CLASSPATH**
  - proměnná prostředí (environment) operačního systému
  - obsahuje posloupnost cest (adresářů), kde má postupně interpret kódy hledat (používá ji vlastně již překladač):

```
setenv CLASSPATH .:~pepca/MFF/java
set CLASSPATH=.;c:/jdk1.4.2/lib;c:/MFF/java
```

- stejnou funkci má i parametr na příkazové řádce
  - „**-cp** <list>“ nebo „**-classpath** <list>“
- formát archivu **.jar** (Java Archive)
  - komprimované .class soubory i s adresářovou strukturou

Applet

- objekt umístěný (a běžící) na **WWW stránce**
  - spouštěná třída je potomkem **java.applet.Applet**
  - přes HTML kód lze předat parametry (staticky)
  - interpret Javy běží v nejbezpečnějším režimu (sandbox)
    - není možné přistupovat na lokální disk, otevírat nové síťové kanály, apod.
  - integrace JVM do všech dnešních grafických prohlížečů

```
<applet code=Life.class id=Life width=800 height=650>
  <param name=cellsize value=5>
  <param name=colors value=30>
  <param name=delay value=200>
</applet>
```

Object

- **java.lang.Object** – univerzální předek
  - implicitní předek všech ostatních tříd i polí
  - nemusí se uvádět v klauzuli „extends“
  - nejčastější použití: univerzální reference
- vybrané **metody**:
  - **String toString ()** – vrací textovou reprezentaci instance (např. pro ladění)
  - **boolean equals (Object obj)** – porovnání dvou instancí, jejich „obsahu“
    - implicitní implementace: porovnání referencí
    - musí se předefinovat (porovnání datového obsahu)

Object – metody

- **int hashCode ()** – uživatelsky definovaná hašovací hodnota založená na datovém obsahu
  - spolu s „equals(obj)“ se používá v hašovacích tabulkách
- **protected Object clone ()** – vytvoření identické kopie instance (jen pro „implements Cloneable“)
  - musí se předefinovat
- **void wait (), void wait (long millis), void wait (long millis, int nanos), void notify (), void notifyAll ()** – pro synchronizaci (multi-threading)
- **void finalize ()** – vyvoláván při úklidu paměti (v okamžiku recyklace)
  - není zaručováno jeho vyvolání (není vlastně k ničemu)

String

- **java.lang.String** – řetězec znaků libovolné délky
  - literál typu String je řetězec znaků uzavřený do **uvozovek**: "řetězec", "můžeme použít ρυζνε абецеды"
  - instance třídy String jsou **konstantní** – jakmile je objekt vytvořen, už se nemůže měnit jeho hodnota
  - operátor „+“ vytváří vždy novou instanci
  - „equals“ a „hashCode“ jsou korektně implementovány
- je-li potřeba řetězec měnit, je tu třída **StringBuffer**
  - implicitně se používá ve výrazech typu: "m = " + i + " kg" new StringBuffer().append("m = ").append(i).append(" kg").toString()

Integer, Boolean, ...

- balík **java.lang** obsahuje obalovou třídu pro každý jednoduchý typ
  - **Boolean**, **Character**, **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**
  - použití: v kontejnerech (potřeba mít za předka Object)
- užitečné konstanty a funkce: **Integer**
  - **Integer.MAX_VALUE**, **Integer.MIN_VALUE**
  - **static Integer decode (String nm)** – čte lib. literál
  - **static int parseInt (String s)** – jednodušší forma
  - **static String toHexString (int i)** – výpis v „HEXu“

Character

- užitečné konstanty a funkce: **Character**
  - **static boolean isDigit (char ch), static boolean isLetter (char ch), ...**
  - **static boolean isWhitespace (char ch), static boolean isUpperCase (char ch), ...**
  - **static char toLowerCase (char ch), ...**

Double

- užitečné konstanty a funkce: **Double**
  - **Double.MIN_VALUE**, **Double.NaN**, **Double.NEGATIVE_INFINITY**, ...
  - **static long doubleToLongBits (double d)** – převod na binární (IEEE 754) reprezentaci
  - **static boolean isNaN (double d)**, **static boolean isInfinite (double d)**, ...
  - **static double parseDouble (String s)**, ...

BigInteger, BigDecimal

- **java.math.BigInteger** – celé číslo bez omezení délky
  - konstantní instance (viz String)
  - kromě běžných operací umí i další užitečné funkce: modulární aritmetika, práce s prvočísly, bitové operace, NSD, celočíselná mocnina, apod.
  - **konstruktory**: z pole „byte[]“, čtení řetězce (String), náhodný generátor, generování prvočísla
- **java.math.BigDecimal** – desetinné číslo s libovolnou přesností (dekadicky reprezentované, pevná desetinná tečka)
  - kromě běžných operací i plná kontrola zaokrouhlování

Správa paměti

- Java má **automatickou správu paměti**
  - všechny objekty na haldě (instance tříd a pole) se alokují explicitně operátorem „**new**“, ale nemusí se v programu uvolňovat
  - **uvolnění (úklid) paměti** provádí systém **sám**
  - není definováno, kdy (zda vůbec) bude paměť nějakého konkrétního objektu recyklována
- metoda „**finalize()**“ – volá se až před recyklací paměti
  - **není zaručeno**, že bude vůbec zavolána
  - **nepoužívat** místo destruktoru v C++ !

Destrukce objektu

- chceme-li, aby byla **na konci života** každé instance zavolána nějaká metoda, musíme ten mechanismus **implementovat sami**..
  - uzavření souborů na disku
  - úklid nestandardních prostředků (např. hardware)
  - může být obtížné/pracné určit okamžik, kdy již instance není používána
  - není definováno, kdy (zda vůbec) bude paměť nějakého konkrétního objektu recyklována
    - ale metodu „finalize()“ lze použít pro test, že byla destrukce v pořádku provedena (assert)

Úklid paměti (garbage collecting)

- z hlediska programátora je zcela **transparentní**
  - při real-time aplikacích můžeme někdy pozorovat malé „zpoždění“ – pozastavení běhu všech vláken programu
- **metody úklidu paměti**
  - počítání referencí (pozor na cyklické reference!)
  - „**stop-and-copy**“ přístup
    - průchod všech dostupných objektů a jejich přesun jinam
    - potřebuji cca 2× více paměti, po čase – zbytečné kopírování
  - „**mark-and-sweep**“, různé adaptivní varianty
    - označuji použité instance, nakonec mohu nepoužitou paměť recyklovat (nemusím kopírovat), triky: větší bloky paměti, …

Výjimky (exceptions)

- **asynchronní** výskyt **nestandardní situace**
  - aritmetika: dělení nulou, nepřípustný argument funkce …
  - použití prázdné reference („null“)
  - chyba v přístupu do pole (index mimo povolený rozsah)
  - chyba při vstupní/výstupní operaci (disk, síť, hardware)
  - jakákoli „výjimečná“ situace definovaná programátorem
- výjimky jsou v Javě pevně zapojeny do jazyka
  - **syntaktická kontrola** při překladu
  - výjimku specifikovanou v použité metodě musím buď **ošetřit**, nebo ji též **specifikovat** („předat výš“)

Throwable

- informace o výjimečné události se předává v objektu, který je potomkem třídy „**Throwable**“
  - záznam o stavu zásobníku
  - textová informace o tom, co se stalo (nepovinná)
    - getMessage(), toString()
  - původní příčina výjimky (pro „řetězení výjimek“)
- potomci třídy **Throwable**:
  - **Error** – vážná chyba, nespecifikuje se ani se neošetřuje
  - **Exception** – výjimka, specifikuje se a ošetřuje se
    - **RuntimeException** se nemusí specifikovat (může se objevit kdykoli při vykonávání programu)

Blok „try“ – „catch“ – „finally“

- obecné schéma chráněného (kontrolovaného) bloku:

```
try {
  // any „checked“ code
} catch ( Spec1 e1 ) {
  // handle exception of type Spec1
} catch ( Spec2 e2 ) {
  // handle exception of type Spec2
}
 ...
} finally {
  // this code will be executed every time!
}
```

Použití výjimek

- kontrolované výjimky (mimo **RuntimeExceptions**)
  - programátor **musí** v hlavičce metody specifikovat, které výjimky mohou uvnitř nastat

```
public int f() throws MyException, IOException { ... }
```

  - pokud kontrolovanou výjimku nespecifikuji ani neošetřím („**catch**“), již při překladu se ohlásí chyba

- **vyvolání** (nejen) uživatelské **výjimky**
  - příkaz „**throw**“ – syntax jako u příkazu „return“
    - návratový typ je potomkem „Throwable“ !
    - řízení se předá na **nejbližší** místo, které umí výjimku ošetřit

Výjimky – poznámky

- **dědičnost** – mohu specifikaci („throws“) **zužovat**, ale **nikoli rozšiřovat !**
- ošetřovat výjimky pouze tam, kde s daným problémem **mohu něco dělat**:
  - spočítat náhradní hodnotu
  - opravit data (parametry) a zopakovat výpočet, …
- **opakované vyvolání** výjimky (zevnitř bloku „**catch**“)
  - předávám ji výš (na nejbližší vyšší úroveň), viz „throw“
- **řetězení výjimek** – vyvolám novou výjimku, ale původní příčinu (včetně záznamu zásobníku) nechám
  - výjimky mají konstruktor s jedním parametrem „cause“

Vlákna (threads)

- programy v Javě běží standardně ve více vláknech
- **interface Runnable**
  - jediná metoda „public void run ()“
  - pro třídy, které „lze spustit“
- **class Thread**
  - algoritmus, který běží v samostatném vlákně
  - nastavování proirit
  - funkce související s plánovačem (vzdání se zbytku časového kvanta, čekání, odpočinek – spánek, ...)
  - yield(), wait(), sleep()

Kritická sekce

- chráněná („kritická“) sekce kódu se označuje klíčovým slovem „**synchronized**“
  - každá instance třídy má jeden synchronizační objekt („**monitor**“) zaručující exkluzivitu „synchronizovaných“ metod
  - třída jako celek má další monitor (pro statické metody)
  - „synchronized“ nepatří do signatury metody, takže se nemusí zachovávat při dědění
- **efektivita**
  - vstup do chráněné oblasti je časově náročnější
  - pro JDK ≥ 1.3 už není rozdíl tak dramatický

Malé kritické sekce

- **libovolný objekt** lze použít ke hlídání kritické sekce
  - „synchronized (instance) { ... }“
    - tento blok kódu je chráněný stejným monitorem, jako všechny „synchronized“ metody té instance
  - instance třídy **Object** (potřebuji jen monitor)
  - často je kvůli prevenci uváznutí (deadlock) žádoucí zamykat co nejkratší sekce kódu
    - „synchronized (this) { ... }“
- **uváznutí (deadlock)**
  - prevence: např. priorita (uspořádání) zámků
  - nepřehledný MT kód: velmi špatně se ladí

Spolupráce, synchronizace vláken

- s každou instancí objektu je spojen **semafor**
  - **čekání** na semafor: „**wait()**“
    - dvě formy: nekonečné i časově omezené čekání
  - **signalizace**: „**notify**“ nebo „**notifyAll**“
    - signalizující vlákno pokračuje bez čekání dále ve výpočtu
- čekání i signalizace se musí volat **v kritické sekci** příslušného objektu
  - čekající je sice uvolněn, musí však ještě počkat, než signalizující vlákno uvolní daný monitor..
  - nesprávné volání – „IllegalMonitorStateException“

Stavy vlákna

- **nové vlákno**
  - vlákno již existuje, ale ještě nebyl zavolán jeho kód
- **běžící vlákno**
  - potenciálně běžící vlákno, ale aktuálně může čekat na přidělení CPU (časového kvanta)
  - střídání „běžících“ vláken na CPU je nedeterministické!
- **blokované vlákno**
  - odpočívá (sleeping), čeká na nějakou událost (waiting)
  - může též čekat na I/O operaci (skryté čekání)
  - čeká, až bude moci vstoupit do chráněné oblasti (přidělení monitoru)

Interface ještě jednou

- slabší náhrada za chybějící výčtové typy
  - interface obsahující pouze definice konstant
  - konstanty jsou implicitně: „**public static final**“
  - „interface“ zdůrazňuje, že nemá smysl vytvářet instance

```
public interface WeekDay {
   byte SUNDAY    = 0;
   byte MONDAY    = 1;
   byte TUESDAY   = 2;
   byte WEDNESDAY = 3;
   byte THURSDAY  = 4;
   byte FRIDAY    = 5;
   byte SATURDAY  = 6;
}
```

Interface naposledy

- datové členy v „interface“ musí být „**final**“
  - to ale neznamená, že musí být inicializovány konstantním výrazem
  - instance třídy vůbec konstantní být nemusí..

```
public interface Rand {
   Random rand     = new Random();
   int rInt        = rand.nextInt();
   double rDouble = rand.nextDouble();
}
   ...
   System.out.println(“Rnd = “ + Rand.rand.nextInt());
   ...
```

Vnitřní třídy („inner classes“)

- velmi **silný prostředek** jazyka Java (od verze 1.1)
  - mohou nahradit vícenásobnou dědičnost (když potřebuji dědit více než jednu implementaci)
  - umožňují snadný a bezpečný přístup k privátním datům
  - usnadňují programování řízené událostmi
  - zpětné volání (obslužná rutina, „handler“, „callback“)
- **syntax**:
  - definice třídy („**vnitřní**“) uvnitř jiné třídy („**vnější**“)
  - instance vnitřní třídy je vždy spojená s instancí třídy vnější – má přístup ke všem členům (i privátním) !

Definice vnitřní třídy I

- definice třídy přímo **uvnitř vnější třídy**:

```
public class Parcel1 {
  private int val = 12;
  class PContents {
    public int value () { return val; }
  }
  public PContents cont () {
    return new PContents ();
  }
}
  ...   // in the same package:
  Parcel1 p = new Parcel1();
  Parcel1.PContents c1 = p.cont();
  Parcel1.PContents c2 = p.new PContents();
```

Přetypování (upcast) vnitřní třídy

- vnější třída vrací vnitřní třídu jako **službu** (**callback**), která může pracovat i s privátními členy:

```
public interface Contents {
  int value ();
}
public class Parcel2 {
  private int val = 12;
  protected class PContents implements Contents {
    public int value () { return val; }
  }
  public Contents cont () {
    return new PContents ();
  }
}
```

Veřejná služba

- služba se tak může používat i **vně daného balíku**
  - vnitřní třída je úplně skrytá (nikdo další o ní neví)

```
...
Parcel2 p = new Parcel2();
Contents c = p.cont();
System.out.println( c.value() );
...
```

- „**služební**“ vnitřní třída může být **potomkem libovolné** konkrétní nebo abstraktní **třídy**
  - z hlediska vnější třídy se tak vlastně jedná o vícenásobné dědění s implementací (rozdíl od implementace více protokolů, které nemohou mít implementaci)

Definice vnitřní třídy II

- definice třídy **uvnitř metody** vnější třídy
  - viditelná pouze uvnitř té metody, instance však žije dál !

```
public class Parcel3 {
  private int val = 12;
  public Contents cont () {
    class PContents implements Contents {
      public int value () { return val; }
    }
    return new PContents ();
  }
}
  ...
  Parcel3 p = new Parcel3();
  Contents c = p.cont();
```

Definice vnitřní třídy III

- **anonymní vnitřní třída**
  - nepojmenovaná třída je **dědicem** protokolu/třídy

```
public class Parcel4 {
  private int val = 12;
  public Contents cont () {
    return new Contents() {
      public int value () { return val; }
    };   // semicolon required (terminates the
   command)
  }
}
  ...
  Parcel4 p = new Parcel4();
  Contents c = p.cont();
```

Anonymní třídy

- **předkem anonymní třídy** může být libovolný interface nebo jiná třída
  - konstruktor předka se může volat i s parametry
- **konstruktor anonymní třídy** se píše formálně jako inicializace třídy

```
public class Parcel5 {
  public Ancestor anc ( final String s ) {
    return new Ancestor(12) {
      String local;
      { local = s; }
      public String sValue () { return local; }
    };
  }
}
```

Definice vnitřní třídy IV

- **statická vnitřní třída**
  - neobsahuje odkaz na instanci vnější třídy
  - může používat všechny statické členy vnější třídy
  - může obsahovat další vnitřní třídy (nejde u ne-statické)

```
public class Parcel6 {
  private static int val;
  static class PContents implements Contents {
    public int value () { return val; }
  }
  public static Contents cont () {
    return new PContents();
  }
}
```

Odkaz na vnější instanci

- jméno vnější třídy a „**.this**“
  - `Parcel5.this`
  - všechny odkazy (včetně implicitního odkazu z vnitřní třídy na vnější instanci) se samozřejmě účastní všech mechanismů správy paměti
  - tj. například nemůže být zrušena instance vnější třídy, pokud k ní ještě existuje některá instance její vnitřní třídy

Dědičnost mezi vnitřními třídami

- z pohledu dědičnosti jsou to obyčejné třídy
  - potomek vnitřní třídy musí být inicializován s odkazem na instanci původní vnější třídy

```
class Outer {
  class Inner {}
}

public class InheritInner extends Outer.Inner {
  InheritInner ( Outer o ) {
    Outer.super();
  }
}
```

- když dědíme z vnější třídy, její vnitřní třídy zůstanou beze změny

Statická inicializace třídy

- **blok kódu** uvnitř třídy označený slovem „**static**“
  - nepatří k instanci třídy, ale k třídě jako takové
  - provádí se v okamžiku, kdy je třída poprvé použita (zavedena – „ClassLoader“), opět jich může být několik

```
public class Sample {
  static int i;      // static, „class“ variable
  String s;          // instance variable
  Sample ( String s ) {
    this.s = s + i; // „i“ is always initialized !
  }
  static {
    i = 12;
  }
}
```

Běhová kontrola typů (RTTI)

- **přetypování** reference **směrem „dolů“** – z obecnější na více specializovanou třídu

```
List animals;   // list of references to „Object“
...
Cat cat = (Cat)animals.get(12);
...
```

  - ve skutečnosti se vždy provádí za běhu typová kontrola pomocí **RTTI** („run-time type identification“)
  - v případě chyby se vyvolá **ClassCastException**
  - pokud si není programátor jistý, může použít vestavěný predikát „**instanceof**“

Predikát „instanceof“

- zjišťuje, zda je daný objekt **instancí** nebo **potomkem** požadované třídy (interface, abstraktní třídy)
  - kontrola korektnosti přiřazení, přetypování

```
List animals;   // list of references to „Object“
...
Object item = animals.get(12);     // type-safe
Cat cat = (item instanceof Cat) ? (Cat)item : null;
...
```

  - pouze pro **statické testování**
    - predikát musí dostat (statický) **literál třídy**
    - dynamická varianta: Class.isInstance (Object obj)

Třída „Class“

- objekt reprezentující **třídu** v programu
  - **vytváření instancí** (objektů) dané třídy
  - **zavedení třídy** (její statické části) do paměti
  - **dynamická kontrola** typů a kompatibility – odkaz na třídu pomocí jejího textového identifikátoru
  - třída „Class“ je hojně používána **run-time systémem**
    - vytváření instancí objektů, vyvolávání jejich metod, ..
    - ukládá se vlastně do „**.class**“ souboru
  - protože se v programu **odrážejí** přímo **objekty** (třídy, jejich metody a proměnné, ..) téhož programu, říká se tomuto přístupu „**reflexe**“ (viz balík **java.lang.reflect**)

Jak získat instanci Class ?

- **staticky** (kontrola překladačem)
  - „**<type>.class**“: např. „MyClass.class“, „int.class“
  - „**<wrapper-type>.TYPE**“ pro jednoduché typy: např. „Boolean.TYPE“, „Integer.TYPE“
- **dynamicky** (jméno třídy se určuje až při běhu)
  - podle **jména** „**Class.forName (String className)**“
    - Class.forName(“MyClass“)
    - Class.forName(packageName+“.“+className)
  - z libovolné **instance** objektu „**object.getClass()**“
    - dog.getClass()
    - animals.get(12).getClass()

ClassLoader

- run-time objekt, který **zavádí** třídy do paměti
  - „**.class**“ soubory hledá v hierarchii adresářů na disku i v **JAR** archivech (viz CLASSPATH)
  - při neúspěchu vyvolá **ClassNotFoundException**
- pro **zavedení třídy** do paměti stačí získat její instanci (viz předchozí stránku), „MyClass.class“
- ClassLoader umí též **poskytovat datové soubory** (resource) ze stejných domén jako „.class“ kódy
  - možnost jednotného přístupu z disku nebo JAR archivu
  - findResource (String), getSystemResource (String), ..

Metody třídy Class

- **vytváření instancí**
  - Object **newInstance** ()
  - Constructor **getConstructor** (Class[] parameterTypes)
- **dědičnost**
  - Class **getSuperclass** (), Class[] **getInterfaces** ()
- **globální vlastnosti** (atributy) třídy
  - Package **getPackage** (), int **getModifiers** ()
  - boolean **isInterface** (), boolean **isPrimitive** ()
- práce s **vnitřními třídami** (inner classes)
  - Class[] **getClasses** (), Class **getDeclaringClass** ()

Metody třídy Class

- práce s **poli**
  - boolean **isArray** (), Class **getComponentType** ()
- **typová kontrola**
  - boolean **isAssignableFrom** (Class cls)
  - boolean **isInstance** (Object obj)
- přístup k **veřejným členům třídy**
  - Field **getField** (String name), Field[] **getFields** ()
  - Method **getMethod** (String name, Class[] parTypes), Method[] **getMethods** ()
  - Constructor **getConstructor** (Class[] paramTypes), Constructor[] **getConstructors** ()

Metody třídy Class

- přístup ke **všem členům třídy**, ne však zděděným
  - Field **getDeclaredField** (String name), Field[] **getDeclaredFields** ()
  - Method **getDeclaredMethod** (String name, Class[] parTypes), Method[] **getDeclaredMethods** ()
  - Constructor **getDeclaredConstructor** (Class[] paramTypes)
  - Constructor[] **getDeclaredConstructors** ()
- **ekvivalence** dvou tříd (nepočítá s dědičností):
  - operátor „==“
  - „boolean Class.**equals** (Class class)“

Balík „java.lang.reflect“

- spolu s třídou **java.lang.Class** obsahuje prostředky pro reflexi
  - třídy **Array**, **Field**, **Constructor**, **Method**
  - dynamické varianty většiny operací jazyka Java
  - zachovává se bezpečnost a všechna přístupová omezení
- hlavní metody třídy **Array**
  - Object **newInstance** (Class compType, int len)
  - Object **newInstance** (Class compType, int[] dims)
  - Object **get** (Object array, int index)
  - void **set** (Object array, int index, Object value)
  - int **getInt** (Object array, int index), ...

Metody třídy Field

- obecné metody:
  - String **getName** (), int **getModifiers** ()
  - Class **getType** (), Class **getDeclaringClass** ()
- čtení:
  - Object **get** (Object obj) – obecné čtení
  - int **getInt** (Object obj), float **getFloat** (Object obj), ...
- zápis:
  - void **set** (Object obj, Object value) – obecný zápis
  - void **setInt** (Object obj, int value),
    void **setFloat** (Object obj, float value), ...

Metody třídy Constructor

- obecné metody (platné i pro obyčejné metody):
  - String **getName** (), int **getModifiers** ()
  - Class **getDeclaringClass** ()
  - Class[] **getExceptionTypes** ()
  - Class[] **getParameterTypes** ()
- vytvoření nové instance dané třídy:
  - Object **newInstance** (Object[] initargs)

Metody třídy Method

- všechny obecné metody třídy „Constructor“, navíc:
  - Class **getReturnType** ()
- vyvolání metody
  - Object **invoke** (Object obj, Object[] args)

Kontejnery (container classes)

- běžně používané jednoduché datové struktury
  - **pole** (pevné nebo proměnlivé délky)
  - **seznam** (zachovává pořadí)
  - **množina** (prvky se nesmí opakovat)
  - **mapa**, slovník (množina dvojic „klíč → hodnota“)
- standardní přístupové metody a techniky (patterns)
  - **prvek**: univerzální „Object“
  - **Iterator**: procházení všech prvků kontejneru
  - **Comparator**: porovnávací třída (reprezentuje relaci)

Interface Collection

- **obecný kontejner** (některé metody jsou nepovinné)
  - boolean **add** (Object o), boolean **addAll** (Collection c)
  - boolean **remove** (Object o), boolean **removeAll** (Collection c)
  - void **clear** (), boolean **isEmpty** (), int **size** ()
  - boolean **contains** (Object o), boolean **containsAll** (Collection c)
  - boolean **retainAll** (Collection c)
  - Object[] **toArray** (), Object[] **toArray** (Object[] a)
  - Iterator **iterator** ()

Interface Iterator

- **sekvenční průchod** všemi prvky kontejneru
  - **není zaručeno pořadí** procházení (ani jeho opakovatelnost)
  - během procházení lze prvky **odstraňovat**
- metody:
  - boolean **hasNext** ()
  - Object **next** ()
    - může vrátit „null“ („null“ smí být prvkem kolekce)
  - void **remove** ()
    - odstraní prvek vrácený posledním voláním „next()“

Interface Comparator

- **porovnávání** dvou prvků
  - úplné (lineární) uspořádání
  - slouží ke třídění kolekcí
- metoda:
  - int **compare** (Object o1, Object o2)
    - vrací záporné číslo, nulu resp. kladné číslo pro „**o1 < o2**“, „**o1 = o2**“ resp. „**o1 > o2**“
    - symetrie: sgn(compare(x,y)) == –sgn(compare(y,x))
    - není vyžadována shoda s relací „equals()“ (ale obyčejně to platí):

      compare(x,y) == 0 ⇔ x.equals(y)

Interface List

- **uspořádaná kolekce** (sekvence)
  - typicky se dovolují duplikace prvků
  - speciální **ListIterator** (vkládání i odstraňování prvků, obousměrné procházení, libovolný počátek průchodu..)
- metody (navíc proti „Collection“):
  - Object **get** (int index)
  - int **indexOf** (Object o), int **lastIndexOf** (Object o)
  - Object **set** (int index, Object element)
  - List **subList** (int fromIndex, int toIndex)
  - ListIterator **listIterator** (int i), ...

Interface Set

- kolekce **prvků bez opakování** (množina)
  - pokud je dovolen i prázdný prvek („null“), smí být obsažen maximálně jednou
  - pozor na nekonstantní objekty („mutable“) !
- neobsahuje žádné metody navíc proti „Collection“

Interface Map

- **zobrazení**, **slovník**: množina dvojic „klíč → hodnota“
  - Object **put** (Object key, Object value)
  - void **putAll** (Map t)
  - Object **remove** (Object key)
  - Object **get** (Object key)
  - boolean **containsKey** (Object key)
  - boolean **containsValue** (Object value)
  - void **clear** (), boolean **isEmpty** (), int **size** ()
  - Set **entrySet** (), Set **keySet** ()
  - Collection **values** ()

Konkrétní implementace

- List
  - **ArrayList**, **LinkedList**, **Vector** (sync), **Stack** (sync)
- Set
  - **HashSet**, **LinkedHashSet**, **TreeSet** (i SortedSet)
- Map
  - **HashMap**, **Hashtable** (sync), **TreeMap** (i SortedMap)

Třída BitSet

- **bitové pole**
  - definiční obor: konečná podmnožina přirozených čísel
  - binární operace: **and**, **andNot**, **or**, **xor**
  - sada přístupových operací: **clear**, **get**, **set**
  - int **cardinality** (), int **length** ()
  - boolean **intersects** (BitSet set)
  - int **nextClearBit** (int fromIndex)
  - int **nextSetBit** (int fromIndex)

Třída Arrays

- **efektivní manipulace s poli** (jen statické metody)
  - List **asList** (Object[] a)
  - int **binarySearch** (<type>[] a, <type> key)
  - int **binarySearch** (Object[] a, Object key, Comparator)
  - boolean **equals** (<type>[] a, <type>[] b)
  - void **fill** (<type>[] a, <type> val)
  - void **sort** (<type>[] a)
  - void **sort** (<type>[] a, int fromIndex, int toIndex)
  - void **sort** (Object[] a, Comparator c)

Konec

Prezentace se ještě dolaďuje, později může být na stejném místě zveřejněna upravená verze ...