Programování v jazyce Java

```
<link rel="stylesheet"  href="http://cdnjs.cloudflare.com/ajax/libs/font-
awesome/3.1.0/css/font-awesome.min.css">
```

PB162 Programování v jazyce Java — jaro 2020

Základní informace

- jednotlivé sekce (1, 2…) neodpovídají přesně probírané látce v týdnech semestru (ale přibližně ano)
- přesné info k probírané látce v daném týdnu přednášky i cvičení najdete vždy v osnově předmětu PB162 jaro2020 v IS

1. Úvod do Javy

- slidy Úvod do studia | pro tisk HTML, PDF
- slidy Úvod do Javy | pro tisk HTML, PDF
- slidy První program, třída, objekt | pro tisk HTML, PDF
- slidy Balíky | pro tisk HTML, PDF
- slidy Spuštění programu z příkazové řádky | pro tisk HTML, PDF
- **slidy na doma** Opakovací témata | pro tisk HTML, PDF

2. Úvod do objektového programování, konstruktory

- slidy Konstruktory | pro tisk HTML, PDF
- slidy Zapouzdření | pro tisk HTML, PDF
- slidy Konvence | pro tisk HTML, PDF
- slidy Datové typy | pro tisk HTML, PDF
- slidy JavaDoc | pro tisk HTML, PDF

3. Statické proměnné a metody, neměnné objekty, přetěžování

- slidy Static | pro tisk HTML, PDF
- slidy Konstanty | pro tisk HTML, PDF

- slidy Výčtové typy | pro tisk HTML, PDF
- slidy Neměnné objekty | pro tisk HTML, PDF
- slidy Přetěžování | pro tisk HTML, PDF
- slidy Překrývání, třída Object | pro tisk HTML, PDF

4. Rozhraní

- slidy Rozhraní | pro tisk HTML, PDF
- slidy Rozšiřování rozhraní | pro tisk HTML, PDF
- slidy Testování, `JUnit` | pro tisk HTML, PDF

5. Dědičnost, viditelnost

- slidy Dědičnost | pro tisk HTML, PDF
- slidy Viditelnost | pro tisk HTML, PDF

6. Pole, porovnávání objektů, abstraktní třídy

- slidy Pole, třída `Arrays` | pro tisk HTML, PDF
- slidy Porovnávání objektů | pro tisk HTML, PDF
- slidy Abstraktní třídy | pro tisk HTML, PDF
- doplňkové téma Statické a výchozí metody rozhraní | pro tisk HTML, PDF

7. Dynamické datové struktury I

- slidy Kontejnery obecně, rozhraní `Collection` | pro tisk HTML, PDF
- slidy Seznam, množina, iterátory | pro tisk HTML, PDF

8. Dynamické datové struktury II

- slidy Mapy | pro tisk HTML, PDF
- slidy Uspořádané kolekce | pro tisk HTML, PDF
- slidy Porovnání kontejnerů, třída `Collections` | pro tisk HTML, PDF
- doplňkové téma Streamy a lambda výrazy | pro tisk HTML, PDF
- doplňkové téma Parametrické (generické) typy | pro tisk HTML, PDF

9. Výjimky

- slidy Výjimky | pro tisk HTML, PDF

10. Hlídané a vlastní výjimky

- slidy Hlídané a vlastní výjimky, blok `finally` | pro tisk HTML, PDF

11. Vstupy a výstupy

- slidy Vstupy a výstupy | pro tisk HTML, PDF

12. Soubory

- slidy Soubory | pro tisk HTML, PDF
- slidy `Path` a `Files` | pro tisk HTML, PDF

13. Konzultace, návaznosti, pokročilá témata

- slidy Java Archiver (`jar`) | pro tisk HTML, PDF
- slidy Navazující předměty | pro tisk HTML, PDF

První program v Javě, třída, objekt

Program "Hello World!"

- Abychom měli kam náš kód psát, vytvoříme třídu Demo s hlavní funkci main, která se zavolá při spuštění programu.

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

- Metoda main musí být **veřejná** (public), **statická** (static) a **nevrací žádnou hodnotu** (void). *Klíčová slova pochopíte časem, není to teď důležité.*
- Metoda musí mít parametry typu String (řetězec), které se předávají při spuštění z příkazového řádku do pole String[] args.

Motivace třídy I

- Jak reprezentovat složitou strukturu, aby se s ní dobře pracovalo?
- Příklad: Osoba s *jménem* a *rokem narození*

```
class Person {
   String name;
   int yearBorn;
}
```

- Části objektu nastavíme i zjistíme stejným způsobem jako v jazyce Python:

```
k.name = "Karel"; // set name to Karel
String karelsName = k.name; // get name value
```

Jednotlivé části (jméno, rok narození) nazýváme *atributy*.

Motivace třídy II

- Někdy bychom rádi měli funkce, které pracují přímo s částmi struktury.
- Pamatujeme si rok narození, ale co když chceme zjistit věk?
- Jak lehce zjistit informace o naší ~~struktuře~~ — *třídě*?

```
public class Person {
   private String name;
   private int yearBorn;
   public int getAge() {
     return 2018 - yearBorn;
   }
   public void printNameWithAge() {
     System.out.println("I am " + name + " and my age is " + getAge());
   }
}
```

V kódu třídy se nyní objevila klíčová slova `public` a `private`. Nemají vliv na funkcionalitu, ale na "viditelnost", na možnost či nemožnost z jiného kódu danou třídu nebo její vlastnost vidět a použít. Logicky `public` asi (určitě :)) půjde použít vždy a odevšad.

Vlastnosti třídy

- Třída představuje strukturu, která má *atributy* a *metody*.

Atributy

- jsou nositeli datového obsahu, údajů, "pasivních" vlastností objektů
- to, co struktura má, *z čeho se skládá*, např. auto se skládá z kol
- definují **stav** objektu, nesou **informace** o objektu

Metody

- jsou nositeli "výkonných" vlastností, *schopností* objektů něco udělat
- to, *co dokáže struktura dělat* — pes dokáže štěkat, osoba dokáže mluvit
- definují **chování** objektu (může být závislé na stavu)

Vytvoření konkrétní osoby

- Máme třídu Person, to je něco jako *abstraktní šablona* pro objekty—osoby.
- Jak vytvořím **konkrétní** osobu s jménem Jan?

```
public class Demo {
 public static void main(String[] a) {
   Person jan = new Person();
   jan.name = "Jan";
   jan.yearBorn = 2000;
   System.out.println(jan.name);
   System.out.println(jan.yearBorn);
 }
}
```

Poznámky k příkladu `Demo`

- **Třída** Person má vlastnost name a age, to jsou její *atributy*.
- **Objekt** jan typu Person má vlastnost name s hodnotou *Jan* a yearBorn s hodnotou *2000*.
- Klíčová slova `public` a `private` vám z Pythonu nejsou známá, zde v Javě i jiných jazycích označují "viditelnost" položky — jednoduše řečeno, co je veřejné a co soukromé. Soukromé atributy "vidíme" jen z metod třídy, v níž jsou uvedeny.

Objekt

- Objekt je jeden **konkrétní jedinec** příslušné třídy.
- Všechny vytvořené objekty nesou stejné vlastnosti, např. všechny objekty třídy Person mají vlastnost name.
- Vlastnosti mají však pro různé lidi různé hodnoty — lidi mají různá jména.
- Konkrétní objekt určité třídy se také nazývá *instance* (jedincem) své třídy.

Vytváření objektů

- Co znamená new Person()?
- Proč musíme psát Person jan = new Person() a ne jen `Person jan`?

```
Person jan = new Person();
// why not just:
Person jan;
```

- Pouhá deklarace proměnné objektového typu (`Person jan`) žádný objekt nevytvoří.
- K vytvoření slouží operátor `new`.

Co se děje při vytváření objektů přes `new`

- Alokuje se paměť v oblasti dynamické paměti, tedy na *haldě* (heap).
- Vytvoří se tam objekt a naplní jeho atributy výchozími hodnotami.
- Zavolá se speciální metoda objektu, tzv. konstruktor, který objekt dotvoří.

Konstruktor

- Slouží k "oživení" vytvořeného objektu bezprostředně po jeho vytvoření:
  - Jednoduché typy, jako například `int`, se vytvoří a inicializují samy a konstruktor nepotřebují.
  - Složené typy, *objekty*, je potřeba vždy zkonstruovat!
- V našem příkladu s osobou operátor `new` *vytvoří prázdný objekt typu Person* a naplní jeho atributy výchozími (default) hodnotami.
- Další přednáška bude věnována konstruktorům, kde se dozvíte víc.

Třída a objekt

Třída

- Je komplexní struktura, reprezentuje prvky z reálného světa (např. pes, člověk).
- Je určitý vzor pro tvorbu podobných objektů (konkrétních psů či lidí).
- Definice třídy sestává převážně z *atributů* a *metod* (říkáme jim také prvky nebo členy třídy).
- Skutečné objekty této třídy pak budou mít prvky, které byly ve třídě definovány.

Objekt

- Objekty jsou instancemi "své" třídy vytvořené dle definice třídy a obsahující atributy.
- Vytváříme je operátorem `new`.
- Odkazy na vytvořené objekty často ukládáme do proměnné typu té třídy, např. `Person jan = new Person();`

Komplexnější příklad I

Následující třída `Account` modeluje jednoduchý bankovní účet.

- Každý bankovní účet má jeden *atribut* `balance`, který reprezentuje množství peněz na účtu.

- Pak má *metody*:
  - `add` přidává na účet/odebírá z účtu
  - `writeBalance` vypisuje zůstatek
  - `transferTo` převádí na jiný účet

Komplexnější příklad II

```
public class Account {
   private double balance; // 0.0
   public void add(double amount) {
      balance += amount;
   }
   public void writeBalance() {
      System.out.println(balance);
   }
   public void transferTo(Account whereTo, double amount) {
      balance -= amount;
      whereTo.add(amount); // whereTo is another account
   }
}
```

- Metoda `transferTo` pracuje nejen se svým "mateřským" objektem, ale i s objektem `whereTo` předaným do metody.

Komplexnější příklad - definice vs. použití třídy

- Třída sama je definovaná v samostatném souboru `Account.java`.
- Její použití pak třeba v `Demo.java`.

```
public static void main(String[] args) {
   Account petrsAccount = new Account();
   Account ivansAccount = new Account();
   petrsAccount.add(100.0);
   ivansAccount.add(20.0);
   petrsAccount.transferTo(ivansAccount, 30.0);
   petrsAccount.writeBalance(); // prints 70.0
   ivansAccount.writeBalance(); // prints 50.0
}
```

`println` vs. `return`

- Pozor na rozdíl mezi vypsáním řetězce a jeho vrácením:

```
public void writeString() {
  System.out.println("Sample text"); // writes it
}
```

```
public String returnString() {
  return "Sample text"; // does not write it
}
= Programování v jazyce Java =
```

Profil

- Prakticky zaměřený bakalářský předmět
- Cílem je naučit základním principům objektového návrhu a programování.
- Všechny materiály k přednášce jsou v IS MU.

Předchozí předměty IB111

IB111 **Základy programování**, kde studenti získávají

- základní znalosti programování (Python)
- znalost základních příkazů, řídicí struktury, pole
- částečnou znalost objektového přístupu

Předchozí předměty PB071

PB071 **Principy nízkoúrovňového programování**, kde studenti získávají

- znalost syntaxe jazyka C
- znalost základních datových typů
- znalost vnitřních struktur

Předchozí předměty IB002

IB002 **Algoritmy a datové struktury I**, kde studenti získávají

- základy algoritmizace vč. datových struktur

Předpoklady obecně

Předpokládají se základní znalosti strukturované algoritmizace a programování, tj.:

- základní příkazy, sestavování jednoduchých výrazů;

- základní datové typy (celá a reálná čísla, logické proměnné, řetězce);
- základní řídicí struktury — větvení, cykly, procedury/funkce.

Návaznosti PV168

Na tento základní kurz PB162 navazují na úrovni Bc. studia:

PV168 **Seminář z jazyka Java** (jaro)

- náplní je zvládnutí Javy umožňující vývoj jednodušších praktických aplikací s GUI, databázemi, základy webových aplikací.
- V průběhu semestru se pracuje na uceleném projektu formou párového programování plus některých individuálních úloh.
- Učí kolektiv zkušených cvičících pod vedením Tomáše Pitnera, Luďka Bártka, Petra Adámka a Martina Kuby.

Návaznosti PB138

PB138 **Moderní značkovací jazyky** (jaro)

- náplní jsou XML a související technologie,
- prvky týmového vývoje (projekty, využití služeb hostování projektů, jako je **GitHub**).
- Učí kolektiv zkušených cvičících pod vedením Luďka Bártka a Tomáše Pitnera.

Návaznosti pokročilých předmětů PA165

PA165 **Vývoj aplikací v jazyce Java** (podzim)

- pokročilejší předmět spíše magisterského určení, předpokládá znalosti/zkušenosti z oblasti databází, částečně sítí a distribuovaných systémů, a také Javy zhruba v rozsahu PB162 a PV168.
- Náplní je zvládnutí netriviálních, převážně klient/server aplikací na platformě JavaEE.
- Přednáší *Petr Adámek*, *Tomáš Pitner*, *Bruno Rossi*, *Martin Kuba*, *Filip Nguyen*, *Matej Briškár*, *Tomáš Skopal*.

Návaznosti — webový vývoj

Problematice webových a mobilních aplikací se na FI věnují např.

- každý semestr PV226 Seminář Lasaris
- v jarním semestru PV219 Seminář webdesignu
- v podzimním semestru předmět PV247 Moderní uživatelská rozhraní
- v podzimním semestru předmět PV249 Vývoj v Ruby
- v jarním semestru PV239 Mobilní platformy

- v podzimním návazný PV256 Projekt z programování pro Android

Hodnocení a harmonogram předmětu

- Harmonogram přednášek i cvičení
- Hodnocení

O přednášejícím - Radek Ošlejšek

- pracovna **A305** (budova A1 FI) laboratoře Lasaris
- tel. `54949 6121` (z tlf. mimo budovu), kl. `6121` (volání v rámci fakulty i celé MU)
- e-mail: oslejsek(at)fi.muni.cz
- Web RO: Osobní stránka RO

O přednášejícím - Tomáš Pitner

- pracovna **A303** (budova A1 FI) laboratoře Lasaris,
- příp. kanc.správy Vědecko-technologického parku CERIT (1.NP/přízemí budovy A2);
- tel. `54949 5940` (z tlf. mimo budovu), kl. `5940` (volání v rámci fakulty i celé MU)
- e-mail: tomp(at)fi.muni.cz
- Web: Osobní web TP

Konzultační hodiny

- Primárním konzultačním bodem jsou vaši cvičící.
- Cvičení jsou vedena mj. právě z důvodu možnosti konzultací.
- Konzultace přímo s přednášejícími

  Tomáš Pitner
  : vždy v kanc. **A303**

- Út 10.00 — 11.30
- nebo jindy, dle dohody

Informační zdroje (knihy)

- Rudolf Pecinovský: Myslíme objektově v jazyku Java nebo
- Java 7 — Učebnice objektové architektury pro začátečníky Grada Publishing
- Rudolf Pecinovský: Java 5.0 — Novinky jazyka a upgrade aplikací (fulltext v PDF zdarma)
- Tomáš Pitner: *Java — začínáme programovat*, Grada Publishing, 2002, doprovodný web knihy (učebnice je orientovaná na Javu 1.4 a nižší; většina poznatků je platných i nadále, ale v Javě 5

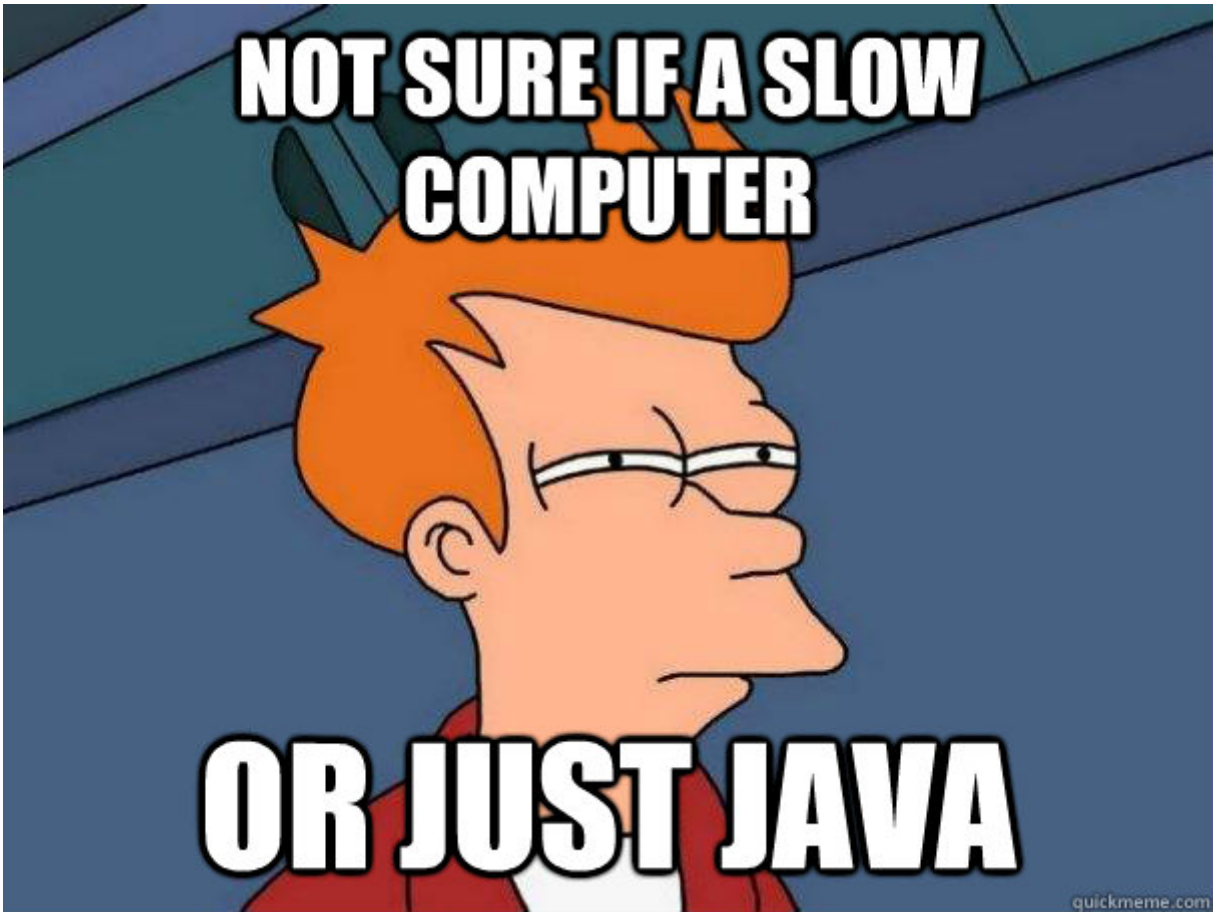
až 8 se objevila řada nových prvků)

- Pavel Herout: *Učebnice jazyka Java*, Kopp, 2000-2010, doprovodný web knihy
- příp. i Pavel Herout: *Java — grafické uživatelské rozhraní a čeština*, Kopp, 2001 — pro pokročilé

Informační zdroje (knihy)

- Bruce Eckel: Myslíme v jazyce Java — příručka programátora, Grada Publishing , 2000
- příp. Bruce Eckel: Myslíme v jazyce Java — příručka zkušeného programátora, Grada Publishing , 2000 — pro pokročilé
- Joshua Bloch: Java efektivně — 57 zásad softwarového experta, Grada Publishing
- Bogdan Kiszka: 1001 tipů a triků pro programování v jazyce Java, Computer Press, 2003
- Bruce Eckel: Thinking in Java Stáhnout zdarma (PDF) = Úvod do jazyka a prostředí Java =

Rychlost Javy



Java jako programovací jazyk

- Dle indexu TIOBE stále nejpopulárnější jazyk na světě s podílem 15 %, další jsou jazyk C a Python.
- Je jazykem *3. generace* (3GL) — imperativním jazykem vysoké úrovně.
- Je jazykem *univerzálním* — není určen výhradně pro specifickou aplikační oblast.

- Je jazykem *objektově-orientovaným* — program používá volání metod objektů (zasílání zpráv objektům).
- Je OO jazykem disponujícím *objektovými třídami*, což automaticky neplatí pro všechny jazyky s objekty.
- Ideovým předchůdcem Javy je *C++*.
- Svým způsobem je Java obdobou C++, ale zbaveným zbytečností a nepříjemností.

Java v budoucnu

- Pro tradiční typy serverových podnikových aplikací a systémů zůstává Java (Enterprise Edition) klíčovou platformou spolu s .NET
- Perspektivním směrem vývoje je zachování Java platformy (JVM, stávající knihovny, aplikace, aplikační prostředí)
- Rychle se vyvíjejí *skriptovací jazyky* na této platformě: *Groovy*, *JRuby*, *Jython*, a zejména v poslední době *Kotlin*…)

Mnoho jazyků bylo inspirovaných javou: *C#*, *Groovy*, *Ruby*, *Scala*, *Kotlin*…

Proč Java

- Java je jazyk pro vývoj a běh jednoduchých i rozsáhlých aplikací.
- Vývoj je efektivnější než na jejich předchůdcích (C++) a výsledné aplikace "běží všude".
- Silnou typovaností, běhovou bezpečnostní kontrolou, efektivní automatickou alokací a dealokací paměti (garbage collection), stabilními knihovnami vč. open-source a rozsáhlým souborem dobrých praktik nabízí aplikacím velmi vysokou robustnost.
- Nezavádí zbytečnosti a vede ke správným a dále uplatnitelným návykům.
- Je velmi perspektivní platformou pro vývoj open-source i komerčního SW, mj. pro *extrémně velké* množství volně dostupných knihoven.

Další charakteristiky

- Java *podporuje vytváření správných návyků* v objektovém programování a naopak systematicky brání přenosu některých špatných návyků z jiných jazyků.
- Program v Javě je *přenositelný* na úrovni *zdrojového i přeloženého* kódu.
- Přeložený javový program běží v tzv. Java Virtual Machine (JVM).
- Zdrojový i přeložený kód je tedy přenositelný mezi všemi obvyklými platformami (UNIX, Windows, Mac OS X).

Java pro programátora

Konkrétní možnosti:

- V Javě se dobře píší *vícevláknové aplikace* (multithreaded applications).
- Java má *automatické odklizení nepoužitelných objektů* (automatic garbage collection).
- Java je jednodušší než C++ (méně syntaktických konstrukcí, méně nejednoznačností v návrhu), což zlepšuje čitelnost a redukuje riziko chyb.

Platformy (specifikace) Java

- *Java SE* (Standard Edition) - pro běžná i serverová použití na běžných "plných" platformách (Linux, Mac OS, Win)
- *Java EE* (Enterprise Edition) - kromě samotného běhového prostředí (JVM) zahrnuje také množství dalších API. Existuje 20 implementací (aplikačních serverů) Java EE, jako je GlassFish Server, IBM WebSphere Application Server, Red Hat JBoss Enterprise Application Platform a další.
- *Java ME* (Micro Edition) - do přenosných, vestavěných a malých zařízení (mobily, televizory)
- *Java Card* - platforma pro bezpečné chytré karty

Aktuální verze

Stav k září 2019:

- *Java Standard Edition 12*, *11* a *8* (u zákazníků s *Long Term Support* pokračují i SE 6 a 7 a LTS 11)
- je stabilní verzí pro všechny platformy.
- U Java 7 běžná podpora skončila dubnem 2015.
- Aktuální informace najdete vždy na webu Oracle Oracle Technetwork/Java.
- K předpokládanému vývoji existuje Oracle roadmap

Licence Javy

- Na stránkách https://www.java.com/download/ je java dostupná Oracle Java SE pro všechny platformy.
- Pro testovací, vývojové, demonstrační nekomerční využití je Java SE zdarma.
- Pro komerční provoz je Oracle Java placená. Licenční poplatky se liší dle způsobu nasazení (desktop až cloud) od cca $2.5 do $25 měsíčně na jeden procesor a zahrnuje podporu - podobně jako např. RHEL.

Stažení Javy

- samotné *vývojové prostředí* (JDK), např. Java SE 8 JDK
- jen *běhové prostředí* (JRE), např. Java SE 8 JRE: to nám tady nestačí, chceme vyvíjet
- JDK v balíčku s grafickým (okénkovým) *integrovaným vývojovým prostředím* (IDE, Integrated Development Environment) NetBeans.

Připravili jsme pro vás tutoriál, jak Javu nainstalovat.

Budoucnost Javy

- Na závěr optimistického úvodu si pročtěte zajímavý článek analytika od *Forrester*: Java Is A Dead-End For Enterprise App Development = Balíky (packages) =

Organizace tříd do balíků

- Třídy zorganizujeme do balíků.
- V balíku jsou vždy umístěny *související* třídy.
- Co znamená *související*?
  - pracuje na nich **jeden tým**
  - jejich **objekty spolupracují**
  - jsou podobné **úrovni abstrakce**
  - jsou ze **stejné části reality**

Příklad: V balíku `geometry` jsou třídy reprezentující geometrické objekty (čtverec, trojúhelník, ...).

Světově unikátní pojmenování balíků

- Aby se zabránilo kolizím (stejná jména pro různé třídy)
- konstruují se jména balíků jako pokud možno světově unikátní
- byla zvolena obdoba doménových internetových jmen (taky unikátní)

Příklad jména balíku

- `cz.muni.fi.pb162` je možné a vhodné jméno balíku
- je světově unikátní, protože `cz.muni.fi` je obrácené doménové jméno fakulty (`fi.muni.cz`)
- `pb162` je identifikátor, jehož jedinečnost už si v rámci organizace FI "uhlídáme"
- Pozor, jiné konvence mají balíky ve standardní vestavěné knihovně Java Core API (např. `java.util`)
- Občas jsou výjimky i jinde, např. používalo se `junit.framework`, i když to nebylo Java Core API.

Příklad třídy v balíku

```java
package cz.muni.fi.pb162;
// class Person is in this package
public class Person {
   // attributes, methods
}
```

Všechna písmena názvu balíku by měla být dle konvencí *malá*, tedy nikoli `Cz.Muni.Fi.PB162` nebo tak něco.

Plný název třídy vč. balíku

- Na třídu v balíku se odvoláváme plným názvem `cz.muni.fi.pb162.Person`
- Pokud se odvoláváme na třídu ve stejném balíku (z jedné do druhé), pak stačí jen "holé" lokální jméno `Person`

Zápis třídy do zdrojového souboru

- Umístění do balíků souvisí s umístěním zdrojových souborů na disku
- Třída `Person` bude v souboru `Person.java`
- Tento soubor bude v adresáři `cz/muni/fi/pb162`
- Pozor na velká/malá písmena — v obsahu i názvu souboru i adresářů

Příslušnost třídy k balíku

- Deklarujeme ji syntaxí: `package název.balíku;`
- Uvádíme obvykle jako *první* deklaraci v zdrojovém souboru.
- Příslušnost k balíku musíme současně *potvrdit správným umístěním* zdrojového souboru do adresářové struktury.
- Neuvedeme-li příslušnost k balíku, stane se třída součástí tzv. *implicitního balíku* — prosím **nepoužívat!**

(Pseudo)hierarchie balíků

- Balíky obvykle organizujeme do jakýchsi pseudohierarchií, např.:
  - `cz.muni.fi.pb162`
  - `cz.muni.fi.pb162.banking`
  - `cz.muni.fi.pb162.banking.credit`
- Nicméně není to tak, že by např. třída `cz.muni.fi.pb162.banking.Account` byla současně v balíku `cz.muni.fi.pb162.banking` a taky třeba `cz.muni.fi.pb162`.
- Je-li třída v balíku `cz.muni.fi.pb162.banking`, je pouze v něm a žádném jiném.

- Není ani v `cz.muni.fi.pb162`, ani v `cz.muni.fi.pb162.banking.credit`.

Prostě buď *je ve stejném* balíku nebo *je v jiném* :-)

Použití tříd v různých balících

- Balíky slouží k logickému rozčlenění kódu
- Důsledkem je vzájemná "(ne)viditelnost" tříd
- Velmi zjednodušeně: třídy v jednom balíku "mají k sobě blíž"
- Jsou-li v různých balících, vůbec o sobě nemusí vědět

Odbočka: práva přístupu

- Kromě toho rozhoduje i to, jakou viditelnost (právo přístupu) použijeme
- Pro tento účel slouží modifikátory `public`, `protected`, `private`
- Objasníme později

Příklad vzájemného použití tříd

- Třídy ve stejném balíku: snadné vzájemné použití

Třída `Person` v balíku `cz.muni.fi.pb162`

```
package cz.muni.fi.pb162;
public class Person {
   // attributes, methods
}
```

Třída `Account` v tomtéž balíku

```
package cz.muni.fi.pb162;
public class Account {
   private Person owner; // owner of this Account is a Person
}
```

Příklad vzájemného použití tříd

- Třídy v jiném balíku: nutné plné jméno
- Třída `Account` v balíku `import cz.muni.fi.pb162.banking`
- použije pro třídu `Person` její plný název balíku

```
package cz.muni.fi.pb162.banking;
public class Account {
   private cz.muni.fi.pb162.Person owner; // full package name
}
```

Deklarace `import`

- Nebo lze pro vzájemné odvolávání pomocí deklarace `import`

```
package cz.muni.fi.pb162.banking;
import cz.muni.fi.pb162.Person;
public class Account {
   private Person owner; // class name imported above
}
```

Deklarace `import názevbalíku.NázevTřídy`

- Příklad `import cz.muni.fi.pb162.Person;`
- Umožní použít identifikátor třídy (v našem případě `Person`) v rámci jiné třídy.
- Píšeme obvykle ihned po deklaraci příslušnosti k balíku (`package názevbalíku;`).
- Import není nutné deklarovat mezi třídami téhož balíku!

Deklarace `import názevbalíku.*`

- Import **všech tříd** z balíku provedeme např. `import cz.muni.fi.pb162.*;`
- Doporučuje se "import s hvězdičkou" nepoužívat vůbec
  - jestli máme více * importů, a obojí z nich obsahují třídu `Person`, která z nich se použije?
  - (nepoužije se žádná, dostanete kompilační chybu)
- *Nebudeme používat*!
- Pozn.: Hvězdičkou **nezpřístupníme** třídy z *podbalíků* — např. `import cz.*` nezpřístupní třídu `cz.muni.fi.pb162.Person`. = Příkazy a řídicí struktury v Javě =

Příkazy a řídicí struktury v Javě

V Javě máme následující příkazy:

- Přiřazovací příkaz `=` a jeho modifikace (kombinované operátory jako je `+=` apod.)
- Řízení toku programu (větvení, cykly) `if`, `switch`, `for`, `while`, `do-while`
- Volání metody
- Návrat z metody příkazem `return`

- Příkaz je ukončen středníkem `;`

Přiřazení v Javě

- Operátor přiřazení `=` (assignment)
  1. na levé straně musí být *proměnná*
  2. na pravé straně *výraz přiřaditelný* (assignable) do této proměnné
- Rozlišujeme přiřazení
  1. *primitivních hodnot* a
  2. *odkazů na objekty*

Přiřazení primitivní hodnoty

- Na pravé straně je výraz vracející hodnotu primitivního typu:
  - číslo, logická hodnotu, znak
  - ale ne např. řetězec (to je objekt)
- Na levé straně je proměnná *téhož nebo širšího* typu jako přiřazovaná hodnota:
  - např. `int` lze přiřadit do `long`
- Při zužujícím přiřazení se také provede konverze, ale může dojít ke ztrátě informace:
  - např. `int` → `short` nebo i `int` → `float` nebo i `int` → `double` jsou zužující
- Přiřazením primitivní hodnoty se hodnota zduplikuje ("opíše") do proměnné na levé straně.

Přiřazení odkazu na objekt

- Konstrukci `=` lze použít i pro přiřazení do objektové proměnné, např. `Person z1 = new Person()`
- Co to udělalo?
  1. pravá strana vytvoří se nový objekt typu `Person` (`new Person()`)
  2. přiřazení přiřadilo jej do proměnné `z1` typu `Person`
- Nyní můžeme *odkaz* na tentýž vytvořený objekt například znovu přiřadit do `z2`: `Person z2 = z1;`
- Proměnné `z1` a `z2` ukazují nyní na fyzicky stejný (identický) objekt typu osoba!!!
- Proměnné objektového typu obsahují *odkazy* (reference) na objekty, tedy ne objekty samotné!!!

Volání metody

- Metoda objektu je vlastně procedura/funkce, která realizuje svou činnost primárně s proměnnými objektu.
- Volání metody určitého objektu realizujeme:
- `identifikaceObjektu.názevMetody(skutečné parametry)`, kde:

    - `identifikaceObjektu`, jehož metodu voláme
    - `.` (tečka)
    - `názevMetody`, jíž nad daným objektem voláme
- v závorách uvedeme *skutečné parametry* volání (záv. může být prázdná, nejsou-li parametry)

Návrat z metody

- Návrat z metody se děje:
    - Buďto automaticky posledním příkazem v těle metody
    - nebo explicitně příkazem `return`
- Oboje způsobí ukončení provádění těla metody a návrat, přičemž u `return` může být specifikována *návratová hodnota*
- Typ skutečné návratové hodnoty musí korespondovat s deklarovaným typem návratové hodnoty.

Větvení výpočtu — podmíněný příkaz

Podmíněný příkaz

neboli *neúplné větvení* pomocí `if`

```
if (logický výraz) příkaz
```

- Platí-li *logický výraz* (má hodnotu `true`), provede se *příkaz*.
- Neplatí-li, neprovede se nic.

Úplné větvení

Příkaz *úplného větvení*

`if (logický výraz) příkaz1 else příkaz2`

- Platí-li *logický výraz* (má hodnotu `true`), provede se *příkaz1*.
- Neplatí-li, provede se *příkaz2*.
- Větev `else` se *nemusí uvádět*.

Cyklus `while`, tj. s podmínkou na začátku

`while`

Tělo cyklu se provádí tak dlouho, **dokud** platí podmínka, obdoba v Pascalu, C a dalších

- V těle cyklu je jeden jednoduchý příkaz:

```
while (podmínka) příkaz;
```

Cyklus `while` se složeným příkazem

- Nebo příkaz složený z více a uzavřený ve složených závorkách:

```
while (podmínka) {
   příkaz1;
   příkaz2;
   příkaz3;
   ...
}
```

- Tělo cyklu se nemusí provést ani jednou — to v případě, že hned při prvním testu na začátku podmínka neplatí.

Doporučení k psaní cyklů/větvení

- Větvení, cykly: doporučuji vždy psát se **složeným příkazem v těle** (tj. se složenými závorkami)!!! Jinak hrozí, že se v těle větvení/cyklu z neopatrnosti při editaci objeví něco jiného, než chceme, např.:

```
while (i < a.length)
   System.out.println(a[i]);
   i++;
```

- Provede v cyklu jen ten výpis, inkrementaci již ne a program se tudíž zacyklí!!!

Doporučení k psaní cyklů/větvení

- Pišme proto vždy takto:

```
while (i < a.length) {
   System.out.println(a[i]);
   i++;
}
```

- U větvení obdobně:

```
if (i < a.length) {
   System.out.println(a[i]);
}
```

Příklad použití `while` cyklu

- Dokud nejsou přečteny všechny vstupní argumenty — vč. toho případu, kdy není ani jeden:

```
int i = 0;
while (i < args.length) {
   System.out.println(args[i]);
   i++;
}
```

Příklad `while` — celočíselné dělení

- Dalším příkladem (pouze ilustračním, protože na danou operaci existuje v Javě vestavěný operátor) je použití `while` pro realizaci celočíselného dělení se zbytkem.

```
public class DivisionBySubtraction {
   public static void main(String[] args) {
      int dividend = 10; // dělenec
      int divisor = 3; // dělitel
      int quotient = 0; // podíl
      int remainder = dividend;
      while (remainder >= divisor) {
         remainder -= divisor;
         quotient++;
      }
      System.out.println("Podil 10/3 je " + quotient);
      System.out.println("Zbytek 10/3 je " + remainder);
   }
}
```

Cyklus `do-while`, tj. s podmínkou na konci

- Tělo se provádí **dokud** platí podmínka (vždy aspoň jednou)
- obdoba `repeat` v Pascalu (podmínka je ovšem *interpretována opačně*)
- Relativně málo používaný — hodí se tam, kde něco musí aspoň jednou proběhnout

```
do {
   příkaz1;
   příkaz2;
   příkaz3;
   ...
} while (podmínka);
```

Příklad použití `do-while` cyklu

- Tam, kde pro úspěch algoritmu "musím aspoň jednou zkusit", např. čtu tak dlouho, dokud není z klávesnice načtena požadovaná hodnota.

```
float number;
boolean isOK;
// create a reader from standard input
BufferedReader in = new BufferReader(new InputStream(System.in));
// until a valid number is given, try to read it
do {
   String input = in.readLine();
   try {
      number = Float.parseFloat(input);
      isOK = true;
   } catch (NumberFormatException nfe) {
      isOK = false;
   }
} while(!isOK);
System.out.println("We've got the number " + number);
```

Příklad: Načítej, dokud není zadáno číslo

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
public class UntilEnteredEnd {
   public static void main(String[] args) throws IOException {
      BufferedReader input = new BufferedReader(
         new InputStreamReader(System.in));
      String line = "";
      do {
            line = input.readLine();
      } while (!line.equals("end"));
      System.out.println("Uživatel zadal " + line);
   }
}
```

Cyklus `for`

- Obdobně jako `for` cyklus v C/C++ jde de-facto o rozšíření cyklu `while`.
- Zapisujeme takto:

```
for(počáteční op.; vstupní podm.; příkaz po každém průch.)
   příkaz;
```

- Anebo obvykleji a bezpečněji mezi `{` a `}` proto, že když přidáme další příkaz, už nezapomeneme dát jej do složených závorek:

```
for (počáteční op.; vstupní podm.; příkaz po každém průch.) {
   příkaz1;
   příkaz2;
   příkaz3;
}
```

Příklad použití `for` cyklu

- Provedení určité sekvence určitý počet krát:

```
for (int i = 0; i < 10; i++) {
   System.out.println(i);
}
```

- Vypíše na obrazovku deset řádků s čísly postupně `0` až `9`.

Doporučení — asymetrické intervaly a pevný počet

- `for` se většinou užívá jako cyklus s pevným počtem opakování, známým při vstupu do cyklu. Tento počet nemusí být vyjádřený konstantou (přímo zadaným číslem), ale neměl by se v průběhu cyklu měnit.
- Používejte *asymetrické* intervaly (ostrá a neostrá nerovnost):
  - počáteční přiřazení `i = 0` a
  - inkrementaci `i++` je *neostrou nerovností*: `i` se na začátku *rovná* 0), zatímco
  - opakovací podmínka `i < 10` je *ostrou nerovností*: `i` už hodnoty 10 *nedosáhne*!
- Vytvarujte se složitých příkazů v hlavičce (kulatých závorkách) `for` cyklu.
- Je lepší to napsat podle situace před cyklus nebo až do jeho těla!

Doporučení — řídicí proměnná

- V cyklu `for` se téměř vždy vyskytuje tzv. *řídicí proměnná*,
- tedy ta, která je v něm inicializována, (obvykle) inkrementována a testována.
- Někteří autoři nedoporučují psát deklaraci řídicí proměnné přímo
  - do závorek cyklu `for (int i = 0; ⋯`
  - ale rozepsat takto: `int i; for (i = 0; ⋯`
- Potom je proměnná `i` přístupná ("viditelná") i za cyklem, což se však ne vždy hodí.

Vícecestné větvení `switch case default`

- Obdoba pascalského `select - case - else`
- Větvení do více možností na základě ordinální hodnoty, v novějších verzí Javy i podle hodnot jiných typů, vč. objektových.
- Chová se spíše jako `switch-case` v C, — zejména se chová jako C při "break-through"

Struktura `switch - case - default`

```
switch(výraz) {
   case hodnota1: prikaz1a;
                  prikaz1b;
                  prikaz1c;
                  ...
                  break;
   case hodnota2: prikaz2a;
                  prikaz2b;
                  ...
                  break;
   default:       prikazDa;
                  prikazDb;
                  ...
}
```

- Je-li *výraz* roven některé z *hodnot*, provede se sekvence uvedená za příslušným `case`.
- Sekvenci obvykle ukončujeme příkazem `break`, který předá řízení ("skočí") na první příkaz za ukončovací závorkou příkazu `switch`.

`switch` další info

- Řídicí *výraz* může nabývat hodnot
  - primitivních typů `byte`, `short`, `char` a `int`, dále
  - *výčtových typů* (`enum`),
  - typu `String` a některých dalších.
- Tutoriál Oracle Java: Switch statement

`switch` příklad s čísly

```
public class MultiBranching {
   public static void main(String[] args) {
      if (args.length == 1) {
         int i = Integer.parseInt(args[0]);
         switch (i) {
            case 1: System.out.println("jednicka"); break;
            case 2: System.out.println("dvojka"); break;
            case 3: System.out.println("trojka"); break;
            default: System.out.println("neco jineho"); break;
         }
         } else {
            System.out.println("Pouziti: java MultiBranching <cislo>");
         }
      }
   }
```

`switch` příklad se `String`

Převzato z tutoriálu Oracle

```
   switch (month.toLowerCase()) {
      case "january":
         monthNumber = 1;
         break;
      case "february":
         monthNumber = 2;
         break;
      case "march":
         monthNumber = 3;
         break;
      ...
```

`switch` příklad se společnými větvemi `case`

Převzato z tutoriálu Oracle

```
int month = 2;
int year = 2000;
int numDays = 0;
```

switch (month) { case 1: case 3: case 5: case 7: case 8: case 10: case 12: numDays = 31; break; case 4: case 6: case 9: case 11: numDays = 30; break; ...

Vnořené větvení

- Větvení `if - else` můžeme samozřejmě vnořovat do sebe.
- Toto je vhodný způsob zápisu:

```
if(podmínka_vnější) {
      if(podmínka_vnitřní_1) {
            ...
      } else {
            ...
      }
} else {
      if(podmínka_vnitřní_2) {
            ...
      } else {
            ...
      }
}
```

Vnořené větvení (2)

- Je možné "šetřit" a neuvádět složené závorky, v takovém případě se `else` vztahuje vždy k nejbližšímu neuzavřenému `if` , např. znovu předchozí příklad:

```
if(podmínka_vnější)
   if(podmínka_vnitřní_1)
      ...
   else // vztahuje se k if(podmínka_vnitřní_1)
else // vztahuje se k if(podmínka_vnější)
   if (podmínka_vnitřní_2)
      ...
   else // vztahuje se k if (podmínka_vnitřní_2) ...
```

- Tak jako u cyklů ani zde tento způsob zápisu (bez závorek) nelze v žádném případě doporučit!!!

Příklad vnořeného větvení

```
public class NestedBranching {
   public static void main(String args[]) {
      int i = Integer.parseInt(args[0]);
      System.out.print(i+" je cislo ");
      if (i % 2 == 0) {
         if (i > 0) {
            System.out.println("sude, kladne");
         } else {
            System.out.println("sude, zaporne nebo 0");
         }
      } else {
         if (i > 0) {
            System.out.println("liche, kladne");
         } else {
            System.out.println("liche, zaporne");
         }
      }
   }
}
```

Řetězené `if - else if - else`

- Časteji rozvíjíme pouze druhou (*negativní*) větev:

```
if (podmínka1) {
   ... // platí podmínka1
} else if (podmínka2) {
   ...  // platí podmínka2
} else if (podmínka3) {
   ...  // platí podmínka3
} else {
   ...  // neplatila žádná
}
```

- Opět je dobré všude psát složené závorky.

Příklad `if - else if - else`

```
public class MultiBranchingIf {
   public static void main(String[] args) {
      if (args.length == 1) {
         int i = Integer.parseInt(args[0]);
         if (i == 1)
         System.out.println("jednicka");
         else if (i == 2)
         System.out.println("dvojka");
         else if (i == 3)
         System.out.println("trojka");
         else
         System.out.println("jine cislo");
         } else {
            System.out.println("Pouziti: java MultiBranchingIf <cislo>");
         }
      }
   }
```

Příkazy `break`

- Realizuje "násilné" ukončení průchodu *cyklem* nebo *větvením* `switch`.
- Syntaxe použití `break` v *cyklu*:

```
int i = 0;
for (; i < a.length; i++) {
   if(a[i] == 0) {
      break; // skoci se za konec cyklu
   }
}
if (a[i] == 0) {
   System.out.println("Nasli jsme 0 na pozici "+i);
} else {
   System.out.println("0 v poli neni");
}
```

- Použití u `switch` jsme již viděli přímo v ukázkách pro `switch`.

Příkaz `continue`

- Používá se v těle cyklu.
- Způsobí přeskočení zbylé části průchodu tělem cyklu

```
for (int i = 0; i < a.length; i++) {
   if (a[i] == 5) continue; // pětku vynecháme
   System.out.println(i);
}
```

- Výše uvedený příklad vypíše čísla `1` , `2` , `3` , `4` , `6` , `7` , `8` , `9` , nevypíše hodnotu `5`.

Příklad na `break` i `continue`

```
public class BreakContinue {
   public static void main(String[] args) {
      if (args.length == 2) {
         int limit = Integer.parseInt(args[0]);
         int skip = Integer.parseInt(args[1]);
         for (int i = 1; i <= 20; i++) {
            if (i == skip)
               continue;
            System.out.print(i+" ");
            if (i == limit)
               break;
         }
         System.out.println("\nKonec cyklu");
      } else {
         System.out.println(
            "Pouziti: java BreakContinue <limit> <vynechej>");
      }
   }
}
```

Příklad je pouze ilustrativní — v reálu bychom `break` na ukončení cyklu v tomto případě nepoužili a místo toho bychom `limit` dali přímo jako horní mez `for` cyklu.

`break` a `continue` s návěštím

- Umožní ještě jemnější řízení průchodu vnořenými cykly:
  - pomocí návěští můžeme naznačit, který cyklus má být příkazem `break` přerušen nebo
  - tělo kterého cyklu má být přeskočeno příkazem `continue`.

```java
public class Label {
   public static void main(String[] args) {
      outer_loop:
      for (int i = 1; i <= 10; i++) {
         for (int j = 1; j <= 10; j++) {
            System.out.print((i*j)+" ");
            if (i*j == 25) break outer_loop;
         }
         System.out.println();
      }
      System.out.println("\nKonec cyklu");
   }
}
```

Spuštění programu z příkazové řádky

Základní životní cyklus javového programu

- Zdrojový kód každé veřejné (`public`) třídy je umístěn v jednom souboru
  - např. třída `Hello` je v `Hello.java`

Postup:

- vytvoření zdrojového textu (libovolným editorem)
- překlad (nástrojem `javac`)
- spuštění (nástrojem `java`)

Nástroje ve vývojové distribuci

- `javac`
  - překladač, tj. `Hello.java` → `Hello.class`
- `java` (nebo `jexec`)
  - spouštěč přeloženého bytecode
- `javadoc`
  - generátor dokumentace
- `jar`
  - správce archivů JAR (sbalení, rozbalení, výpis)

Pod Windows to jsou `.exe` soubory umístěné v podadresáři `bin`

Překlad "Ahoj!"

- Máme nainstalován Java **SDK** 8
- Jsme v adresáři `c:\devel\pb162`, v něm je soubor `Hello.java`
- Spustíme *překlad* — `javac Hello.java`
  - název souboru je včetně přípony `.java`
- Je-li program správně napsán, přeloží se "mlčky"
- Vytvoří se soubor `Hello.class`

Hello.java

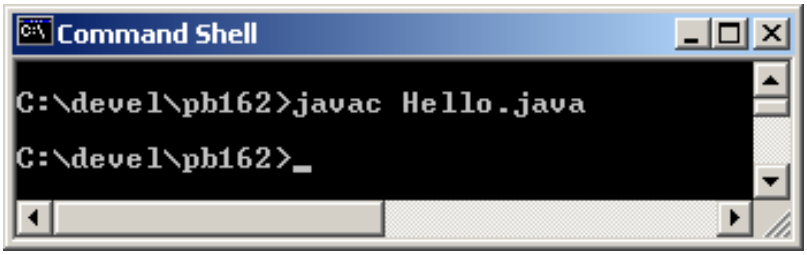
```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Ahoj!");
  }
}
```

Spuštění "Ahoj!"

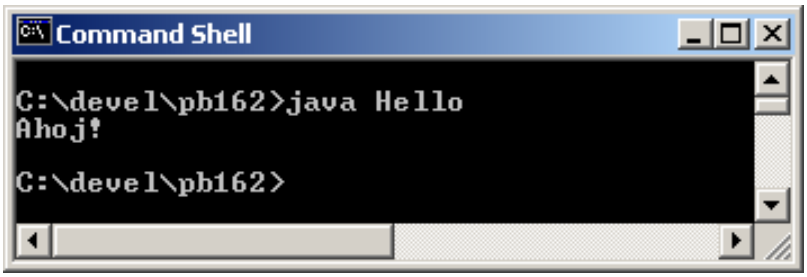
- Spustíme program Hello příkazem `java Hello`
  - název třídy je bez přípony `.class`
- Je-li program správně napsán a přeložen, vypíše se `Ahoj!`

Překlad & Spuštění

Překlad překladačem `javac` (úspěšný, bez hlášení překladače):



Spuštění voláním `java`:

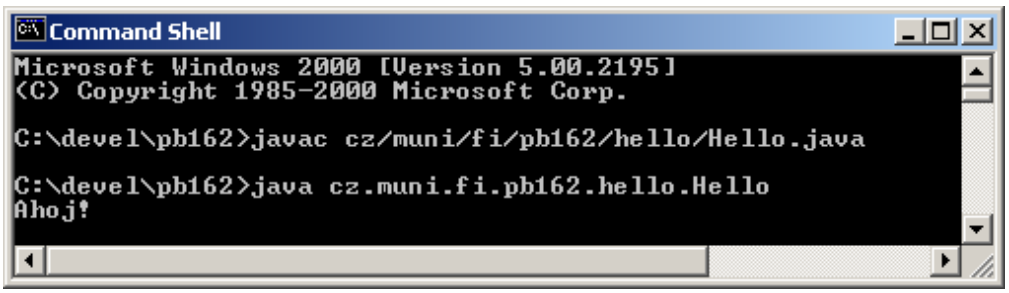


Co když je třída v adresáři (balíku)

Když je třída v balíku, tj. na začátku souboru je:

```
package cz.muni.fi.pb162.hello;
```

Kompilace a spuštění pak vypadá následovně:



Pro **maven** projekty (všechny projekty na cvičení) je nutno být ve adresáři `src/main/java`.

Praktické informace (aneb co je nutné udělat)

- Cesty ke spustitelným programům `PATH` musejí obsahovat i adresář `<JAVA_HOME>/bin`
  - Např. `⋯;C:\Program Files\Java\jdk9.0\bin`
- Systémové proměnné by měly obsahovat `JAVA_HOME=<adresář Javy>`
  - Např. `JAVA_HOME=C:\Program Files\Java\jdk9.0`
- Možné je nastavit i proměnnou `CLASSPATH=<cesty ke třídám>`
  - Např. `CLASSPATH=c:\devel\pb162` = Konstruktory =

Konstruktory

- Konstruktory jsou speciální *metody* volané při vytváření nových objektů (=instancí) dané třídy.
- V konstruktoru se typicky *inicializují atributy (proměnné) objektu.*
- Konstruktory lze volat jen ve spojení s operátorem **new** k vytvoření nového objektu.

Příklad konstruktoru

- Konstruktor se dvěma parametry, inicializuje hodnoty atributů:

```
public class Person {
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

- identifikátor `this` znamená, že se přistupuje k atributům objektu
- nastavují se dle hodnot `name, age` předaných do konstruktoru

Konstruktory — použití

- Příklad využití tohoto konstruktoru:

```
...
Person pepa = new Person("Pepa from Hongkong", 105);
...
```

- Toto volání vytvoří objekt `pepa` a naplní ho jménem a věkem.
- Následně je možné získávat hodnoty proměnných objektů pomocí tečkové notace, např. `pepa.age`.

Výchozí (default) konstruktor

- Co když třída nemá definovaný žádný konstruktor?
- Vytvoří se automaticky *výchozí (default)* konstruktor:

```
public Person() { }
```

- Použití konstruktoru pak vypadá následovně:

```
Person p = new Person();
```

- Výchozí (default) konstruktor se vytvoří pouze v případě, že žádný jiný konstruktor v třídě neexistuje.

Nevytvoření objektu

- Co když volám nad proměnnou metody bez vytvoření objektu?

```
Person p = null;
System.out.println(p.getName());
```

- Kód po spuštění "spadne", neboli zhavaruje nebo předčasně skončí.
- Java sa snaží pád programu popsat pomocí výjimek.
- Výjimky mají své *jméno*, obvykle i určitý textový popis dokumentující příčinu havárie.

```
Exception in thread "main" java.lang.NullPointerException
```

- Výjimky budou probírány později.

Návratový typ konstruktoru?

- Jaký je návratový typ konstruktoru?
- "prázdný" typ `void`? NIKOLI!
- konstruktory vracejí odkaz na vytvořený objekt
- *návratový typ nepíšeme*, typem je odkaz na nově vytvořený objekt

Proměnná objektového typu

- Bavíme se o proměnných *lokálních* ve kódu metod.
- Proměnná objektového typu se deklaruje např. `Person p;`
- Deklarace proměnné objektového typu sama o sobě *žádný objekt nevytváří*
- Takové proměnné jsou pouze *odkazy* na *dynamicky vytvářené objekty*
- Vytvoření objektu se děje až operátorem `new` dynamicky, instance se vytvoří až za běhu programu
- V Javě sa celé objekty do proměnné *neukládají*, jde vždy o uložení pouze *odkazu* (adresy) na objekt

Přiřazení proměnné objektového typu

- Přiřazením takové proměnné pouze *zkopírujeme odkaz*
- Na jeden objekt se odkazujeme nadále ze dvou míst
- **Nezduplikujeme** tím objekt

Příklad kopie odkazu na objekt

- Proměnné `jan` a `janCopy` ukazují na ten tentýž objekt ⇒ změna objektu se projeví v obou:

```java
public static void main(String[] args) {
  Person jan = new Person("Jan");
  Person janCopy = jan;
  janCopy.name = "Janko"; // modifies jan too
  System.out.println(jan.name); // prints "Janko"
}
```

Konstruktory — shrnutí

Jak je psát a co s nimi lze dělat?

- neuvádí se *návratový typ*
- mohou a nemusí mít *parametry*
- když třída nemá žádný konstruktor, automaticky se vytvoří *výchozí*
- může jich být *více* v jedné třídě, reálně se používá

Konvence

Konvence obecně

- Slouží k *ustálení zvyklostí*, jak psát kód
- Konvence pro různé programovací jazyky se obvykle částečně liší.
- *Nejsou striktně vyžadované* překladačem, tzn. kód může být přeložitelný a funkční i při porušení konvencí
- V Javě se dodržují *víceméně všude a všemi vývojáři*, ti většinou nemají moc vlastních odlišných konvencí
- V Javě se na nich *hodně lpí* a jejich nedodržování je neslušnost
- Podstatnou kategorií konvencí jsou *jmenné konvence* pro pojmenovávání tříd, proměnných atd.

Jmenné zásady v Javě

- *Nepoužíváme diakritiku* (problémy s editory, přenositelností a kódováním znaků)
- Používáme *výhradně angličtinu* (čeština/slovenština dělá problémy cizojazyčným kolegům v týmu)
- Je-li jméno složenina více slov, pak je *nespojujeme podtržítkem*: `This_is_bad_in_Java`
- Používáme tzv. *camelCase*, "velbloudí" střídání velkých a malých písmen: `myVeryLongMethodNameIsOK()`
- Konvence jsou jiné pro jména balíků, tříd, metod, proměnných atd. viz dále

Konvence názvů proměnných

- vztahují se na lokální proměnné v metodách i na atributy
- jména proměnných **začínají malým písmenem**
- Příklady
  - `age`
  - `temporalName`

Konvence názvů metod

- platí pro všechny metody obecně
- jména metod **začínají malým písmenem**
- názvy metod vždy obsahují závorky, v kterých mohou, ale nemusí, být parametry
- Příklady
  - `calculateAge()`
  - `print(String stringToBePrinted)` — `stringToBePrinted` je parametr
  - `toString()`

Konvence názvů tříd

- **začínají velkým písmenem**
- Příklady
  - `Person`
  - `MeasurableGrid`

Konvence názvů balíků

- všechno malými písemny
- jednotlivá slova reprezentují složky názvu (a tím adresáře, kde jsou třídy balíku uloženy)
- slova jsou oddělena tečkou
- Příklady
  - `cz`
  - `cz.muni.fi`
  - `geometry` (není ideální, protože není světově unikátní)

Konvence názvů konstant

- Konstantou rozumíme hodnotu, která se *nemění*
- Název konstanty se píše *velkými písmeny*

- Konstanta je jediná výjimka, kde v názvu používáme znak `_`
- Příklady
  - `SIZE`
  - `MAXIMUM_AGE`
  - `DEFAULT_USER_NAME`
- Deklarace typicky obsahuje modifikátory `public static final`
- V celé podobě například `public static final int MAXIMUM_AGE = 100;`
- Je dobře možné i s omezenou viditelností `private static final int MAXIMUM_AGE = 100;`

Jmenné konvence — testík

- Co následující identifikátory mohou být - Třída? Metoda? Lokální proměnná? Atribut? Konstanta?
  - `Dog`
  - `dog`
  - `dog()`
  - `DOG`

Jmenné konvence — závěrem

- Dodržování jmenných konvencí *výrazně zlepšuje čitelnost* i cizího kódu
- Je *základem psaní* srozumitelných programů
- Bude *vyžadováno a hodnoceno* v úlohách i písemkách
- Poměrně málo často se v názvech tříd či proměnných používají *číslice* — spíše výjimečně
- Jedině tam, kde jde o zvláštní konkrétní význam daného čísla, např. *Counter32bit*, *Vertex2D*. = Datové typy v Javě =

Úvod k datovým typům v Javě

- Existují dvě základní kategorie datových typů: **primitivní** a **objektové**

Primitivní

- v proměnné je uložena přímo hodnota
- např. `int`, `long`, `double`, `boolean`, …

Objektové

- musí se nejdřív zkonstruovat (použitím `new`)
- do proměnné se uloží pouze odkaz
- např. `String`, `Person`, …

Datové typy primitivní

integrální typy

zahrnují typy *celočíselné* (byte , short , int a long) a typ char

čísel s pohyblivou řádovou čárkou

float a double

logických hodnot

boolean

Výchozí (default) hodnoty

- Každý typ má svou výchozí (default) hodnotu, na kterou je nastaven, není-li hned přiřazena jiná.
- Dle Java Language Specification: Each *class variable*, *instance variable*, or array component is initialized with a default value when it is created (§15.9, §15.10):

Type	Default value
`byte`	`(byte)0`
`short`	`(short)0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`
`char`	`'\u0000'`
`boolean`	`false`
reference types	`null`

Na co se vztahují výchozí hodnoty

- Automatické nastavení proměnných na výchozí hodnoty se tedy vztahuje na proměnné objektů a tříd (atributy) a prvky polí.
- Nevztahuje se na lokální proměnné a parametry, ty musejí být před prvním použitím nastaveny, inicializovány.

Příklad výchozí hodnoty

```
int i; // automatically i = 0
```

Více informací najdete na: The Java Tutorials: Primitive Data Types.

Zajímavosti a odlišnosti

- V Javě neexistuje možnost typy rozsahově či interpretačně modifikovat (žádné unsigned int apod.)
- Pro velká čísla lze v nových verzích Javy použít notaci s podtržítkem k oddělení řádů po tisících:

```
private int bigNumber = 123_456_789;
```

Datové typy objektové

- objektovými typy v Javě jsou všechna ostatní typy
- **třídy**
- **rozhraní** ("téměř totéž, co třídy")
- **pole** (ano, v Javě jsou samotná pole objekty)

Výchozí hodnota objektového typu je `null` — tzv. "ukazatel na nic".

Příklad použití objektového typu

```
Person p; // p is null automatically
p = new Person(); // now p references to an object
```

- Objektový typ je všechno, kde se používá operátor `new`.

Shrnutí

Základní rozdíl je v práci s proměnnými.

primitivní typy
: přímo obsahují danou *hodnotu*

objektové typy
: obsahují pouze *odkaz* na příslušný objekt

  - Důsledek: dvě objektové proměnné mohou nést odkaz na **tentýž objekt**

Přiřazení primitivní proměnné

- Hodnota proměnné se nakopíruje:

```
double a = 1.23456;
double b = a;
a += 2;
// a is 3.23456
// b is 1.23456
```

Přiřazení objektové proměnné

- Objektové proměnné ukazují na stejný objekt:

```
public class Counter {
   private double value;
   public Counter(double v) {
      value = v;
   }
   public void add(double v) {
      value += v;
   }
}
...
Counter c1 = new Counter(1.23456);
Counter c2 = c1;
c1.add(2);
// c1 has value 3.23456
// c2 has value 3.23456
```

Operátor ==

- Pro primitivní typy porovnává hodnoty:

```
1 == 1 // true
1 == 2 // false
```

- Pro objektové typy porovnává odkazy:

```
Counter c1 = new Counter(1.23456);
Counter c2 = c1;
c1 == c2 // true
c1 == new Counter(1.23456) // false
```

- Na porovnání *hodnot* objektových typů se používá `equals`, probereme později.

Použití u metod — primitivní typy

- Java funguje na principu "pass-by-value", tj. změna v metodě se neprojeví:

```
public static void main(String[] args) {
  int i = 1;
  passByValue(i);
  System.out.println(i); // 1
}

private static void passByValue(int i) {
  i = 4;
}
```

Použití u metod — objektové typy

- Odkazy na objekty fungují obdobně — změna objektu na jiný se neprojeví.
- Modifikace téhož objektu však ano!

```
public static void main(String[] args) {
  Dog d = new Dog("Max");
  passByValue2(d);
  System.out.println(d); // Charlie
}

private static void passByValue2(Dog d) {
  d.setName("Charlie");
  d = new Dog("Alex");
}
```

Pole v zkratce

- Vytvoření, naplnění a získání hodnot vypadá následovně:

```
int[] array = new int[2];
array[0] = 1;
array[1] = 4;
System.out.println("First element is: " + array[0]);
```

- Deklarace: *typ*[] *jméno* = `new` *typ* [*velikost*];
- *typ* může být i objektový: `Person[] p = new Person[3];` = Zapouzdření =

Co je zapouzdření

- Naprosto zásadní vlastnost objektového přístupu, možná nejzásadnější
- Jde o *spojení dat a práce s nimi* do jednoho celku - objektu
- Data jsou v atributech objektu, práce je umožněna díky metodám objektu
- Data by měla být zvenčí (jinými objekty) přístupná jen prostřednictvím metod
- Data jsou tedy "skryta" uvnitř, zapoudřena
- To zajistí větší bezpečnost a robustnost, přístup k datům máme pod kontrolou

Motivace zapouzdření

```
class Person {
   String name;
   int age;

   Person(String inputName, int inputAge) {
     name = inputName;
     age = inputAge;
   }
 }
```

Co když:

- Nechceme, aby kdokoli mohl modifikovat atributy `name`, `age` po vytvoření objektu
- Většinou ale chceme, aby konstruktor a třídu mohl používat každý

Řešení — práva přístupu

- Nastavíme *viditelnost* nebo též *práva přístupu* pomocí modifikátorů třídy, metody nebo atributu
- Nechceme modifikovat atributy `name`, `age` po vytvoření objektu? ⇒ použijeme klíčové slovo `private`
- Chceme, aby mohl konstruktor a třídu používat skutečně každý? ⇒ použijeme klíčové slovo `public`

```
public class Person {
  private String name;
  private int age;

  public Person(String inputName, int inputAge) {
    name = inputName;
    age = inputAge;
  }
}
```

4 typy viditelnosti v zkratce

- `public` = veřejný, může používat každý i mimo balík ⇒ používejte na třídy a (některé) metody
- `private` = soukromý, nemůže používat nikdo mimo třídy ⇒ používejte na atributy
- `protected` = chráněný ⇒ používá se při dědičnosti, vysvětlíme později
- *modifikátor neuveden*, pak jde o možnost přístupu "package local"
  - v rámci balíku se chová jako `public`, mimo něj jako `private`
  - v našem kurzu tento typ nebudeme používat
- Ujistěte se, že vždy máte zadefinovaný práva přístupu/typ viditelnosti.
- V drtivé většině budete používat `public` a `private`.

Každá **veřejná** třída musí být v souburu se stejným jménem.

Metody `get` & `set` — motivace

```
public class Person {
   private String name;
   private int age;

   public Person(String inputName, int inputAge) {
     name = inputName;
     age = inputAge;
   }
 }
```

- Klíčové slovo `public` umožňuje použít třídu `Person` všude

```
Person p = new Person("Marek", 23); // even from another class/package
```

- Klíčové slovo `private` zabraňuje získat hodnotu atributů `p.name`, `p.age`.

Metody `get`

- Chci *získat hodnotu* atributu i po vytvoření objektu,
- ale *zabránit jeho modifikaci*?
- Do třídy přidáme metodu, která bude *veřejná* a po zavolaní vrátí hodnotu atributu.

```
public int getAge() {
  return age;
}
```

- Takové metody se slangově nazývají "gettery".
- Mají návratovou hodnotu podle typu vraceného atributu.
- Název metody je vždy `get` + *jméno atributu* s velkým písmenem (`getAge`, `getName`, ...).

Metody `set`

- Chci-li nastavit hodnotu atributu i po vytvoření objektu:

```
public void setAge(int updatedAge) {
  age = updatedAge;
}
```

- Metoda je *veřejná* a po jejím zavolání *přenastaví* původní hodnotu atributu.
- Takové metody se slangově nazývají **settery**.
- Mají návratovou hodnotu typu `void` (nevrací nic).
- Název metody je vždy `set` + *jméno atributu* s velkým písmenem (`setAge`, `setName`, ...).

Příklad atribut a `get` & `set`

```
public class Person {
  private String name; // attribute
  public String getName() { // its getter
    return name;
  }
  public void setName(String newName) { // its setter
    name = newName;
  }
}
```

Viditelnost atributů

- Není lepší udělat atribut `public`, namísto vytváření metod `get` a `set`?
- Není, neumíme pak řešit tyhle problémy:
- Co když chci jenom získat hodnotu atributu, ale zakázat modifikaci (mimo třídy)? ⇒ *Řešení:* odstraním metodu `set`
- Chci nastavit atribut věk (v třídě `Person`) pouze na kladné číslo? ⇒ *Řešení:* upravím metodu `set`:
  - `if (updatedAge > 0) age = updatedAge;`
- Chci přidat kód provedený při získávání/nastavování hodnoty atributů? ⇒ *Řešení:* upravím metodu `get`/`set`
- Gettery & settery se dají ve vývojových prostředích (NetBeans, IDEA) generovat automaticky.

Využití `this`

```
public void setAge(int updatedAge) {
  age = updatedAge;
}
```

- Mohli bychom nahradit jméno parametru `updatedAge` za `age`?
- Ano, ale jak bychom se potom dostali k atributu objektu?
- Použitím klíčového slova `this`:

```
public void setAge(int age) {
  this.age = age;
}
```

- `this` určuje, že jde o atribut objektu, nikoli parametr (lokální proměnnou)

Korektní použití třídy I

```
public class Person {
   private String name;
   public Person(String name) {
     this.name = name;
   }
   public void writeInfo() {
     System.out.println("Person " + name);
   }
   public String getName() {
     return this.name;
   }
}
```

Korektní použití třídy II

- Vytvoříme dvě instance (konkrétní objekty) typu `Person`.

```
public class Demo {
   public static void main(String[] args) {
      Person ales = new Person("Ales");
      Person beata = new Person("Beata");
      ales.writeInfo();
      beata.writeInfo();
      String alesName = ales.getName(); // getter is used
      // String alesName = ales.name; // forbidden
   }
}
```

Třída Account — připomenutí

```
public class Account {
  private double balance;
  public void add(double amount) {
    balance += amount;
  }
  public void writeBalance() {
    System.out.println(balance);
  }
  public void transferTo(Account whereTo, double amount) {
    balance -= amount; // change the balance
    whereTo.add(amount);
  }
}
```

- Co je zde malý nedostatek?
- metoda `transferTo` přistupuje přímo k `balance`
- ale přístup k `balance` by měl být nějak lépe kontrolován (např. zda z účtu neberu více, než smím)!

Třída Account — řešení

- řešení: znovupoužijeme metodu `add`, která se o kontrolu zůstatku postará
- (i když to třeba ještě teď neumí!)

```
public void transferTo(Account whereTo, double amount) {
  this.add(-amount);
  whereTo.add(amount);
}
= Dokumentace javového kódu =
```

Dokumentace javových programů I

- Dokumentace je *nezbytnou součástí* javových programů.
- Existuje mnoho druhů dokumentací dle účelu:
  - instalační (pokyny pro nasazení produktu)
  - systémová (konfigurace, správa produktu)
  - uživatelská (pro užívání produktu)
  - vývojářská neboli programátorská (pro údržbu, rozšiřování a znovupoužití)

Dokumentace javových programů II

- Zde se budeme věnovat především dokumentaci *programátorské*.
- To znamená pro ty, kteří budou náš kód využívat ve svých programech, rozšiřovat jej, udržovat jej.
- Programátorské dokumentaci se říká *dokumentace API*, *javadoc*.
- Nejlepším příkladem je přímo dokumentace Java Core API

Pravidla komentování

Při psaní dokumentace dodržujeme tato pravidla:

- Jedná se hlavně o dokumentaci *pro ostatní*, tudíž prioritně dokumentujeme to, co je pro použití ostatními
- Dokumentujeme především celé třídy a jejich `public` a `protected` metody.
- Ostatní věci dle potřeby, například těžce pochopitelné nebo nelogické pasáže kódu.
- Dokumentaci píšeme *přímo do zdrojového kódu* ve *speciálních dokumentačních komentářích*.
- Dovnitř metod nepíšeme většinou *žádné* komentáře, ale můžeme (obtížné části).
- Ideální ovšem je, když *uvnitř metod* žádný komentář být nemusí, neboť účel a funkčnost je z kódu zřejmá :-)
- Ve výuce (ale často i v praxi) budou komentáře ve vašem projektu vynucovány nástrojem *checkstyle*.

Typy komentářů

- *řádkové* - od značky `//` do konce řádku, nepromítnou se do dokumentace

```
// inline comment, no javadoc generated
```

- *blokové* - mohou být na libovolném počtu řádků, nepromítnou se do dokumentace

```
/* block comment, no javadoc generated */
```

- *dokumentační* - libovolný počet řádků, promítnou se do dokumentace

```
/** documentary comment, javadoc generated */
```

- *Řádkové* a *blokové* komentáře by měly být pouze *dočasné*, ve výsledném kódu by měly být komentáře pouze *dokumentační*.

Nástroj `javadoc`

- Slouží ke strojovému generování dokumentace z dokumentačních komentářů a ze samotné struktury programu
- Dokumentace se vygeneruje do sady HTML souborů, takže to bude vypadat jako v Java Core API.
- Používá speciální značky se znakem zavináč, např. `@author`, v dokumentačních komentářích

Značky nástroje `javadoc`

- Javadoc používá značky vkládané do dokumentačních komentářů; hlavními jsou:

`@author`
: specifikuje autora

`@param`
: popisuje jeden parametr metody

`@return`
: popisuje co metoda vrací

`@throws`
: popisuje informace o výjimce, kterou metoda propouští ("vyhazuje")

a další…

Co dokumentujeme

- Dokumentujeme především *celé třídy*, zejména veřejné
- Jejich `public` a `protected` *metody*
- Lze dokumentovat i *celý balík* a to sepsáním souboru `package-info.html` v příslušném balíku
- V pořádných knihovnách (API) dokumentace k balíkům je

Komentář třídy

```
/**
 * This class represents a point in two dimensional space.
 *
 * @author Petr Novotny
 **/
 public class Vertex2D { ... }
```

Komentář metody I

- Zkrácený oficiální komentář metody toString():

```
/**
 * Returns a string representation of the object. In general, the
 * {@code toString} method returns a string that
 * "textually represents" this object. The result should ...
 *
 * @return  a string representation of the object.
 */
 public String toString() { ... }
```

- Část `{@code toString}` značí formátování, vypíše to `toString` neproporcionálním písmem.

Komentář metody II

```
/**
 * Returns the smaller of two int values.
 * If the arguments have the same value, the result is that same value.
 *
 * @param   a   an argument.
 * @param   b   another argument.
 * @return  the smaller of {@code a} and {@code b}.
 */
public int min(int a, int b) {
    return (a <= b) ? a : b;
}
```

Kde uvádíme dokumentační komentáře

- Dokumentační komentáře uvádíme:
  - Před hlavičkou *třídy* — pak komentuje třídu jako celek.
  - Před hlavičkou *metody* — pak komentuje příslušnou metodu.

- Doporučení Sun/Oracle k psaní dokumentačních komentářů — How to Write Doc Comments for the Javadoc Tool

Problémy dokumentování

- Komentáře lžou — změním kód a zapomenu upravit komentář

```
/**
 * Calculates velocity.
 */
System.out.println(triangle.getArea());
```

- Zbytečné komentáře

```
private int size; // creates size
```

- Ideální je psát kód a názvy tříd a metod tak, aby se komentáře ani nemusely číst = Konstanty =

Konstanty v Javě

- Konstanty slouží pro pojmenování určitých konkrétních hodnot se zvláštním významem v programu — tzv. *magic numbers*, kde nemusí být na první pohled zřejmé, proč je to zrovna právě ta hodnota.
- Proto si ji rozumně pojmenujeme a pak používáme tento identifikátor místo přímé hodnoty.
- Může jít i o objektové typy (např. konstanta typu `Person`).

Všeobecně platí: raději víc konstant než míň.

Definice konstanty

- Konstanty jsou vždy:
  - statické (`static`) — stačí nám jedna pro celou třídu
  - neměnné (`final`) — je to konstanta, tudíž pomocí `final` zajistíme neměnnost
- Konstanta může být:
  - *privátní* (dobře možné, když ji nechceme používat mimo třídu)
  - *veřejná* (nicméně asi obvyklejší, většinou má širší použití)

Příklad konstanty

- Obvykle se nastavuje takto přímo přiřazením hodnoty

```
public static final int MAX_PEOPLE_COUNT = 100;
public boolean maxPeopleCountReached() {
  return peopleCount >= MAX_PEOPLE_COUNT;
}
...
int constant = Person.MAX_PEOPLE_COUNT;
```

Klíčové slovo `final`

- Slovo `final` způsobuje, že daná hodnota se v proměnné nemůže změnit.
- V objektové proměnné je uložena adresa (odkaz),
- `final` odkaz se tedy změnit nemůže, ale vnitřek (atributy) objektu ano
- Proto se může kombinovat s neměnnými (immutable) objekty, u nichž se vnitřek nemění

Příklad `final` objektová proměnná

```
final int i = 1;
i = 2; // cannot be done

final Person p = new Person("Honza");
p = new Person("Pavel"); // cannot be done
p.setName("Pavel"); // dirty hack
```

Konstanty mají kromě slova `final` i slovo `static` protože ji chceme *právě jednou*. = Výčtové typy =

Motivace k výčtovému typu

Chceme reprezentovat dny v týdnu.

```
public static final int MONDAY = 0;
public static final int TUESDAY = 1;
public static final int WEDNESDAY = 2;
```

- Problémem je, že nemáme žádnou kontrolu:
  - typovou: metoda přijímající *den* má parametr typu int, takže bere libovolné číslo, třeba `2000`, a to nebude fungovat.
  - hodnotovou: dva dny v týdnu mohou omylem mít stejnou hodnotu a překladač nám to taky neodchytí.

Výčtový typ

- Typově bezpečná cesta, jak vyjmenovat a používat pojmenované konečné výčty prvků.
- Proměnná určitého výčtového typu může pak nabývat vždy jedné hodnoty z daného výčtu.
- Definice výčtového typu "den v týdnu":

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

Příklad použití výčtu

- Velmi příjemné použití ve větvení `switch`

```
public String tellItLikeItIs(Day day) {
  switch (day) {
    case MONDAY:
      return "Mondays are bad.";
    case FRIDAY:
      return "Fridays are better.";
    case SATURDAY:
    case SUNDAY:
      return "Weekends are best.";
    default:
      return "Midweek days are so-so.";
  }
}
```

- Klíčové slovo `break` může být vynecháno, protože `return` způsobí okamžitý návrat z funkce.

Výčty vs. třídy

- Výčtový typ se koncepčně velmi podobá *třídě*, de facto *je to třída*.
- Výčet má však jen *pevně daný počet prvků* (instancí).
- Každý námi definovaný výčtový typ je potomkem třídy java.lang.Enum.
- Podobně jako jiné třídy má vestavěné metody a může mít *další metody*, *konstruktory* apod.

Hezký příklad najdete na The Java™ Tutorials — Enum Types = Neměnné objekty =

Neměnné objekty

- Neměnný (*immutable*) objekt nemůže být po jeho vytvoření modifikován

- Bezpečně víme, co v něm až do konce života bude
- Tudíž může být souběžně používán z více míst, aniž by hrozily nekonzistence
- Jsou prostředkem, jak psát robustní a bezpečný kód
- K jejich vytvoření nepotřebujeme žádný speciální nástroj

Příklad neměnného objektu

```
public class Vertex1D {
  private int x;
  public Vertex1D(int x) {
    this.x = x;
  }
  public int getX() {
    return x;
  }
}
...
Vertex1D v = new Vertex1D(1);
// x of value 1 cannot be changed anymore
```

Výhody a nevýhody

Výhody

- je to vláknově bezpečné (*thread safe*) — objekt může být *bezpečně* používán více vlákny naráz
- programátor má jistotu, že se mu obsah objektu *nezmění* — silný předpoklad
- kód je *čitelnější*, *udržovanější* i *bezpečnější* (např. útočník nemůže změnit náš token)

Nevýhody

- chceme-li objekt byť jen drobně změnit, musíme vytvořit nový
- to stojí čas a paměť

Porovnání

- Neměnný objekt vs. konstanta
  - konstanta je jenom jedna (je statická)
  - neměnných objektů může být víc s různými hodnotami
  - kdyby jich bylo více a měly stejnou hodnotu, postrádá smysl
- Neměnný objekt vs. `final`
  - `final` prakticky zakáže změnu odkazu
  - nelze tedy do dané proměnné přiřadit odkaz na jiný objekt
  - samotný objekt ale může být dál "vevnitř" modifikován

Příklad `final`

```
final Person p = new Person("Marek");
// p = new Person("Jan"); // cannot be done
p.setName("Haha I changed it"); // this works!
```

Vestavěné neměnné třídy

- Neměnnou třídou v Javě je `String`.
- Má to řadu dobrých důvodů - tytéž jednou definované řetězce lze používat souběžně z více míst programu
- Nicméně i negativní stránky - někdy větší režie spojená s nemodifikovatelností:

```
String s = "Hello " + "World";
```

- Kód vytvoří 3 objekty: `"Hello "`, `"World"` a `"Hello World"`.
- Cože? To je tak neefektivní?
- Pokud by vysloveně vadilo, lze místo `String` použít `StringBuilder`, který je modifikovatelný (mutable), viz dále

String pod lupou

- Podívejme se na rozdíl `"Hello"` a `new String("Hello")`.

  `new String("Hello")`
  : vytvoří pokaždé nový objekt

  `"Hello"`
  : funguje na principu: *jestli taký objekt zatím neexistuje, tak ho vytvořím, jinak vrátím odkaz na již existující objekt* (uložen v paměti *String constant pool*)

```
String s1 = "Hello";
String s2 = "Hello";
boolean bTrue = (s1 == s2); // true, identical objects

s1 = new String("Hello");
s2 = new String("Hello");
boolean bFalse = (s1 == s2); // false, different objects though with same value
```

Metody třídy `String`

- `char charAt(int index)` — vráti prvek na daném indexu

- `static String format(String format, Object… args)` — stejné jako `printf` v C
- `boolean isEmpty()` — vráti `true` jestli je prázdný
- `int length()` — velikost řetězce
- `matches`, `split`, `indexOf`, `startsWith` …

Více metod najdete v dokumentaci třídy String.

- Doporučujeme javadoc prostudovat, používaním existujících metod jsi ušetříte spoustu práce!

Třída `StringBuilder`

```
StringBuilder builder = new StringBuilder("Hello ");
builder.append("cruel ").append("world"); // method chain
builder.append("!");
String result = builder.toString();
```

- `StringBuilder` se průběžně modifikuje, přidáváme do něj další znaky
- Na závěr vytvoříme výsledný řetězec
- `StringBuilder` není *thread safe*, proto existuje její varianta `StringBuffer`.

Objektové obálky nad primitivními hodnotami

- Java má primitivní typy — `int`, `char`, `double`, …
- Ke každému primitivnímu typu existuje varianta objektového typu — `Integer`, `Character`, `Double`, …
- Tyto objektové typy se nazývají *wrappers*.
- Objekty jsou neměnné
- Při vytváření takových objektů není nutné používat `new`,
  - využije se tzv. *autoboxing*, např. `Integer five = 5;`
  - Obdobně autounboxing, `int alsoFive = five;`

```
Integer objectInt = new Integer(42);
Integer objectInt2 = 42;
```

Konstanty objektových obálek

- Wrappers (např. `Double`) mají různé konstanty:
  - `MIN_VALUE` je minimální hodnota jakou může `double` obsahovat
  - `POSITIVE_INFINITY` reprezentuje konstantu kladné nekonečno

    - `NaN` je zkratkou Not-a-Number — dostaneme ji např. dělením nuly
- Protože konstanty jsou statické, jejich hodnoty získáme přes název třídy:

```
double d = Double.MIN_VALUE;
d = Double.NEGATIVE_INFINITY;
```

Metody objektových obálek

- např. pro `Double` existuje `static double parseDouble(String s)` — udělá ze `String` číslo, z `"1.0"`, vytvoří číslo `1.0`
- obdobně pro `Integer` a další číselné typy
- pro převody na číselné typy dále `int intValue()` převod `double` do typu `int`
- `boolean isNaN()` — test, jestli není hodnotou číslo

Více konstant a metod popisuje javadoc.

Víc hodnot

- Test: Objektové typy (`Integer`) mají od primitivních (`int`) jednu hodnotu navíc — uhádněte jakou!
- je to `null`
- `Integer` je objektový typ, proměnná je odkaz na objekt

Rozdíl od primitivních typů

Proč tedy vůbec používat primitivní typy, když máme typy objektové?

`int i = 1`

- zabere v paměti právě jen 32 bitů
- používáme přímo danou paměť, jednička je uložena přímo v `i`

`Integer i = 1`

- je třeba alokace paměti pro objekt, zkonstruování objektu s obsahem `1`
- v proměnné `i` je pouze odkaz, je to (nepatrně) pomalejší
- Výkon může být u velkého počtu objektů problém, např. vytvoření miliónu proměnných typu `Integer` namísto `int` může mít dopad na výkon a zcela jistě zabere dost paměti, asi zbytečně

Doporučení

- Používejte *hlavně primitivní typy*
- Využívejte metody objektových typů, hlavně statické, kde není třeba mít objekt

- Řada objektových jazyků vůbec primitivní typy jako v Javě nemá, vše jsou objekty

Přechod mezi primitivními a objektovými typy

- Java podporuje automatické balení (boxing) a vybalení (unboxing) mezi primitivními typy a wrappery.
- Proto je následující kód je naprosto v pořádku

```
int primitiveInt = 42;
Integer objectInt = primitiveInt;
primitiveInt = new Integer(43);
```

- Jak už bylo řečeno, použití *primitivního typu* je obvykle lepší nápad - jsme v Javě :-)

Zvláštnosti

- Zajímavost, anebo spíš podraz Javy:

```
Integer i1 = 100; // between -127 and 128
Integer i2 = 100;
boolean referencesAreEqual = (i1 == i2); // true

i1 = 300;
i2 = 300;
boolean referencesNotEqual = (i1 == i2); // false
```

- Poučení: objekty pomocí `==` obvykle neporovnáváme (budeme se učit o `equals`). = Přetěžování metod =

Přetěžování metod

- Jedna třída může mít více metod se *stejnými názvy, ale různými parametry*.
- Pak hovoříme o tzv. *přetížené* (*overloaded*) metodě.
- I když jsou to teoreticky úplně různé metody, jmenují se stejně, proto by *měly dělat něco podobného*.

Přetěžování — příklad I

```
public void transferTo(Account whereTo, double amount) {
  this.add(-amount);
  whereTo.add(amount);
}
public void transferTo(Account whereTo) {
  whereTo.add(balance);
  balance = 0;
}
```

- První metoda převede na účet příjemce `amount` peněz.
- Druhá metoda převede na účet *celý zůstatek* (`balance`) z účtu odesílatele.
- Nedala by se jedna metoda volat pomocí druhé?

Přetěžování — příklad II

```
public void transferTo(Account whereTo, double amount) {
  this.add(-amount);
  whereTo.add(amount);
}
public void transferTo(Account whereTo) {
  transferTo(whereTo, balance);
}
```

- Toto je *jednodušší, přehlednější*, udělá se tam potenciálně méně chyb.
- Kód se neopakuje, tudíž se neopakuje ani případná chyba
- Je to přesně postup divide-et-impera, rozděl a panuj, dělba práce mezi metodami!

Přetěžování — příklad III

Zkusme naší metodu lépe popsat:

```
public void transferTo(Account whereTo, double amount) {
  this.add(-amount);
  whereTo.add(amount);
}
public void transferAllMoneyTo(Account whereTo) {
  transferTo(whereTo, balance);
}
```

- Převod celého zůstatku jsme napsali jako *nepřetíženou* metodu, která popisuje přesně, co dělá.
- Z názvu metody je zřejmé, co dělá — netřeba ji komentovat!

Přetěžování konstruktorů

- Přetěžovat můžeme i konstruktory
- Můžeme tak mít více konstruktorů v jedné třídě
- Pro vzájemné volání konstruktorů použijeme klíčové slovo `this`
- Používá se hodně často, častěji než přetěžování jiných metod

```
public Person() {
  this("Default name"); // calls second constructor
}
public Person(String name) {
  this.name = name;
}
```

Přetěžování — jak ne

- Proč nelze přetížit metodu *pouze změnou typu návratové hodnoty*?
- Která metoda se zavolá?

```
public int getNumber() {
  return 5;
}
public short getNumber() { // smaller int
  return 6;
}
...
long bigInt = getNumber(); // 5 or 6?
```

- V Javě se číselné typy proměnných přetypují automaticky.
- Mělo by dojít k přetypování `int` na `long`, nebo `short` na `long`?

Obdobný příklad

- Nelze také přetížit uvedením a neuvedením návratové hodnoty
- Protože vracenou hodnotu stejně nemusíme použít:

```
new String("Sss").isEmpty(); // result is omitted
```

- Opět nevíme, která metoda se zavolá:

```
public void getNumber() {
  // do nothing
}
public int getNumber() { // smaller int
  return 6;
}
...
getNumber(); // which one is called?
```

Vracení odkazu na sebe

- Metoda může vracet odkaz na objekt, nad nímž je volána pomocí `this`:

```
public class Account {
  private double balance;

  public Account(double balance) {
    this.balance = balance;
  }

  public Account transferTo(Account whereTo, double amount) {
    add(-amount);
    whereTo.add(amount);
    return this; // return original object
  }
}
```

Řetězení volání

- Vracení odkazu na sebe lze využít k *řetězení volání*:

```
Account petrsAccount = new Account(100);
Account ivansAccount = new Account(100);
Account robertsAccount = new Account(1000);

// we can chain methods
petrsAccount
   .transferTo(ivansAccount, 50)
   .transferTo(robertsAccount, 20);
```

- Stejný princip se dost často využívá u `StringBuilder` metody `append`. = Překrývání metod =

Třída `Object`

- I když v Javě vytvoříme prázdnou třídu, obsahuje 'skryté' metody.

```
public class Person { }
...
Person p = new Person();
p.toString(); // ???
```

- Tyto vestavěné (tj. bez našeho vlivu existující) metody jsou ve skutečnosti *poděděny* z třídy `Object`.
- Seznam všech vestavěných metod najdete v javadocu třídy `Object`.

Některé metody třídy `Object`

- `getClass()` — vrátí název třídy
- `equals`, `hashCode` — bude probíráno později
- `clone()` — vytvoří identickou kopii objektu (*deep copy*)
  - tahle metoda však může způsobovat problémy (vysvětlíme později)
  - proto ji **nepoužívejte**
- `toString()` — vrátí textovou reprezentaci objektu

```
Person p = new Person();
System.out.println(p);
// it simply does this:
System.out.println(p.toString());
```

Překrytí metody

- Podívejme se blíž na metodu `String toString()`.
- Co kdybychom chtěli její chování změnit?
- Zkusme v naší třídě implementovat metodu se stejným jménem:

```
public class Person {
  public String toString() {
    return "it works";
  }
}
Person p = new Person();
System.out.println(p); // it works
```

Metoda `toString()`

- Javadoc říká, že každá třída by měla tuhle metodu překrýt.
- Co se stane, když ji nepřekryjeme a přesto ji zavoláme?

- Použije se výchozí implementace z třídy `Object`:

`getClass().getName() + '@' + Integer.toHexString(hashCode())`

```
Person p = new Person();
System.out.println(p); // Person@14ae5a5
```

- Vypíše se jméno třídy, zavináč, a pak hexadecimálně nějaký podivný hash.

Anotace `@Override` — motivace

- Bylo by fajn mít kontrolu nad tím, že překrýváme skutečně existující metodu.
- Najdete chybu?

```
public class Person {
  public String toStrng() {
    return "not working";
  }
}
Person p = new Person();
System.out.println(p); // Person@14ae5a5
```

Anotace `@Override`

- Použijeme proto **anotaci**, která kompilátoru říká: *přepisuji existující metodu*.
- Anotace se píše před definici metody:

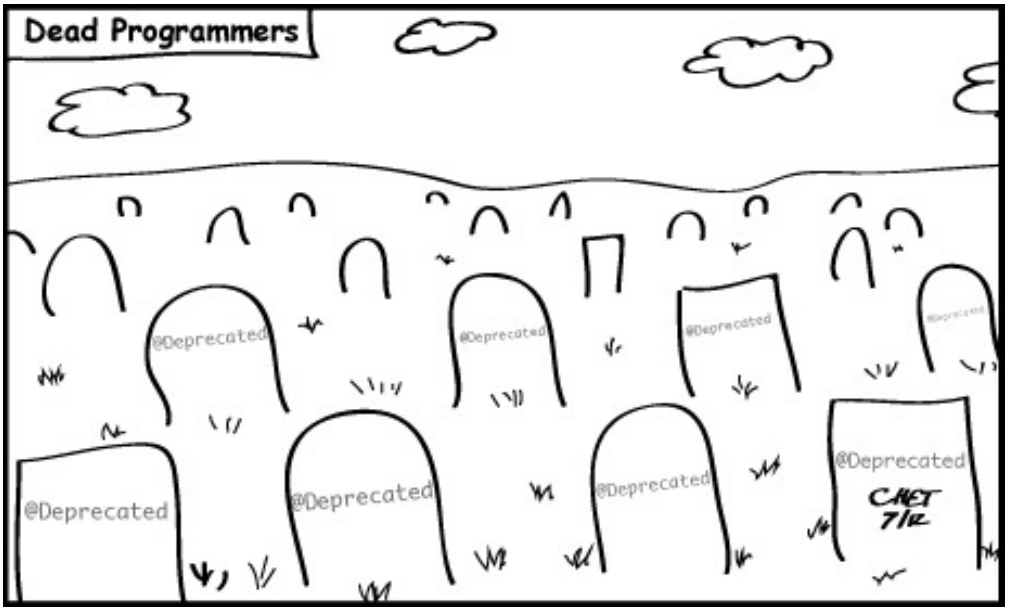
```
@Override
public String toString() {
  return "it works again";
}
```

- Kdybychom udělali chybu např. v názvu překrývané metody, kód by nešel přeložit.

Vždy používejte anotaci `@Override`, když přepisujete metodu.

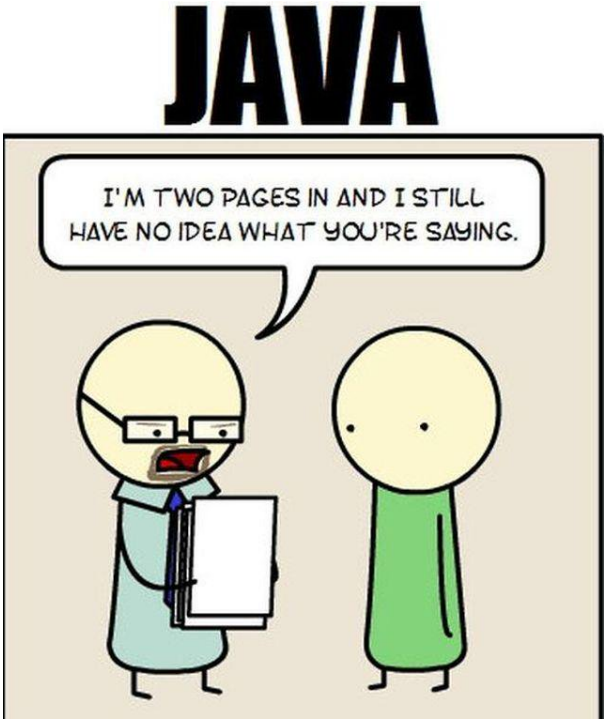
Jiné notace

Existují i jiné anotace, například `@Deprecated` naznačuje, že by se daná věc už neměla používat.



Statické proměnné a metody

Java je ukecaná



Úvod

- Se statickou metodou jsme se setkali už u úplně prvního programu - Hello, world!

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

- `public` reprezentuje viditelnost — metoda je veřejná
- `void` reprezentuje návratový typ — nic nevrací
- Co reprezentuje `static`?
- Metoda static v Javě (jinde může být chápáno jinak) říká, že metoda nepotřebuje pro své fungování žádný konkrétní existující objekt, s nímž by pracovala
- To přesně potřebujeme u `main`, neboť žádný objekt dosud nemáme

Dosud byly nestatické metody a atributy

- Dosud jsme zmiňovali *atributy (proměnné)* a *metody* objektu.
- Jméno (atribut `String name`) patří přímo jedné osobě
- Metoda `toString` vrátí `Person jméno této osoby`

```
public class Person {
  private String name; // name of this person
  ...
  public String toString() {
    return "Person " + name; // returns name of this person
  }
}
```

Jak fungují statické metody a atributy

- Lze deklarovat také metody a proměnné patřící *celé třídě*, tj. *všem objektům třídy*.
- Taková proměnná (nebo metoda) existuje pro jednu třídu jen *jednou*.
- Označujeme ji `static`.

Použití statických metod

- Chceme metodu `max`, která vrací maximální hodnotu dvou čísel
- K tomu žádné objekty nepotřebujeme
- Uděláme ji jako statickou

```
public class Calculate {
  public static int max(int a, int b) {
    if (a > b) return a;
    return b;
  }
}
```

Jde i jako nestatická?

- Ano, takto. Vynecháme `static`.

```
public class Calculate {
  public int max(int a, int b) {
    if (a > b) return a;
    return b;
  }
}
```

- Na spuštění `max` budu nyní potřebovat objekt třídy `Calculate`

```
Calculate c = new Calculate();
int max = c.max(1, 2); // method needs an object `c`
```

- Ovšem ten objekt `c` je tam úplně k ničemu, s žádnými jeho atributy se nepracuje a ani žádné nemá

Řešení

- Uděláme metodu statickou.
- Pak metoda patří celé třídě a zavoláme ji názvem třídy bez konkrétního objektu

```
public class Calculate {
  public static int max(int a, int b) {
    if (a > b) return a;
    return b;
  }
}
...
int m = Calculate.max(1, 2);
```

Zpříjemnění použití statických metod

- Někdy v kódu často používáme statické metody určité třídy, např. naše `Calculator.max`

- Pro kratší zápis lze pak využít deklaraci `import static` dané metody
- nebo všech metod přes *, např.

```
import static cz.muni.fi.pb162.Calculator.*;
...
int m = max(1, 2);
```

Typické použití statických metod

- Velmi často se používají v obdobných situacích, jako výše uvedené `max`
- Tzn. jednoduše pro implementaci funkce, která nevyužívá žádné atributy (data objektu), pouze dostane vstupy a něco vrátí
- Pak ani logicky žádný objekt nepotřebuje
- Příklady: metody třídy `java.util.Arrays`

Statické proměnné (atributy třídy)

- Doposud jsem měli pouze proměnné (atributy) patřící konkrétnímu objektu
- Např. ve třídě `Person`, která reprezentuje člověka, má každý člověk své (obvykle i odlišné) jméno
- Někdy je ale situace, kdy pro celou třídu stačí určitý údaj jenom jednou
- Příklad: chceme jsi pamatovat, kolik lidí se nám během chodu programu vytvořilo
- Jak to udělat?

Počítaní lidí

- Vytvoříme *statickou* proměnnou peopleCount a každý člověk ji při svém vzniku zvýší o jedna.

```
public class Person {
   private String name;
   private static int peopleCount = 0;
   public Person(String name) {
      this.name = name;
      peopleCount++;
   }
   public static int howManyPeople() {
      return peopleCount;
   }
}
```

- Logicky na vrácení počtu lidí stačí *statická metoda* `howManyPeople()`

Počítaní lidí II

- Použití bude vypadat následovně

```
Person.howManyPeople(); // 0
Person jan = new Person("Jan");
Person.howManyPeople(); // 1
Person anna = new Person("Anna");
Person.howManyPeople(); // 2
```

Více informací: Java tutorial — Class variables

Voláni statické metody

Můžeme volat statickou metodu nad konkrétním objektem (instancí) dané třídy?

```
Person anna = new Person("Anna");
anna.howManyPeople();
```

- Ano, není to problém.
- Přes třídu `Person` je to však správnější.

Voláni nestatické metody

- Můžeme volat nestatickou metodu jako statickou?

```
Person.getName();
```

- Logicky ne!
- Co by mohlo volání `Person.getName()` vrátit? Nedává to smysl.
- Jde nám přece o jméno konkrétního člověka, tj. atribut v konkrétním objektu `Person`
- Atribut `name` se nastaví až při zavolání konstruktoru

Přístup ze statické metody k nestatickému atributu?

- Obdobně platí pro atributy, tj. NELZE toto:

```
public class NonStaticTest {
  private int count = 0;
  public static void main(String args[]) {
    count++; // compiler error
  }
}
```

- Java ohlásí při překladu chybu: *non-static variable count cannot be referenced from a static context*.
- Metoda `main` je statická — může používat pouze statické metody a proměnné.

Přístup k nestatickému atributu

- Jedině po vytvoření konkrétní instance:

```
public class NonStaticTest {
  private int count = 0;
  public static void main(String args[]) {
    NonStaticTest test = new NonStaticTest();
    test.count++;
  }
}
```

Všimněte si, že ve třídě mohu vytvořit objekt té stejné třídy.

- Dalším řešením by bylo udělat `count` statický.

Problémy se statickými metodami? NE

- Ano, použití `static` není tak prosté, jak jsme dosud prezentovali :-)
- *Statické metody* většinou problém nejsou
- Jednoduše slouží k realizaci nějaké činnosti, které stačí předané vstupy (parametry) a která nepotřebuje žádný "svůj" objekt s atributy
- Důkazem je řada použití statických metod v Java Core API, kde jsou třídy, které mají *jen statické metody*
- Takovým třídám se říká *utility classes* (jakési "účelové" třídy)

Problémy se statickými proměnnými? ANO

- U statických proměnných je to složitější
- Jejich použití musí být hodně dobře zdůvodněné
- Opravdu potřebujeme danou hodnotu *pro danou třídu právě jednou*???

- Nestane se dobudoucna, že jich budeme potřebovat více???

Možné řešení — singleton

- Velmi často je daleko lepší aplikovat *návrhový vzor singleton* (jedináček)
- Tj. vyrobit si třídu, která bude soustřeďovat (jako nestatické atributy) ta data, která jsme nechtěli dávat do ostatních tříd
- Např. náš výše uvedený počet osob, kromě toho i další údaje, např. nějaký další souhrn za všechny osoby, třeba proměrný plat, věk…
- U singletonu musíme zajistit, aby objekt existoval nejvýše jedenkrát, a to jakmile ho potřebujeme.
- Musíme tedy zabránit přímému vytváření objektů zvnějšku pomocí `new`.
- Na získání instance zvnějšku se pak použije volání statické metody, obvykle něco jako `getInstance()`.

Příklad singletonu

```
public class PersonCounter {
   // here will be the singleton instance (object)
   private static PersonCounter counter = null;
   private int peopleCount = 0;
   // "private" prevents creation of instance via new PersonCounter()
   private PersonCounter() {}
   // creates the singleton unless it exists
   public static PersonCounter getInstance() {
      if(counter == null) {
         counter = new PersonCounter();
      }
      return counter;
   }
   public void addPerson() {
      peopleCount++;
   }
   public int howManyPeople() {
      return peopleCount;
   }
}
```

Import statických prvků

- Už jsme ukázali výše, že statické třídy i metody můžeme importovat:

```
import static java.lang.System.out;
public class HelloWorld {
  public static void main(String[] args) {
    out.println("Hello World!");
  }
}
```

- relevantní pouze pro *statické metody a proměnné*
- vhodně použitelné pro některé věci z Java Core API, např. Math

Více informací: Wikipedia:Static import

Problémy importu statických prvků

```
import static java.lang.System.out;
// developer is reading the code
out.println("Hello World!"); // what is out?
// few lines above:
PrintStream out;
// ahh ok, I thought it was System.out
```

- Takže někdy pak nevíme, jestli jde o statický import a nebo jen o lokální proměnnou/metodu.
- A jakmile něco nevíme na první pohled, JE TO ŠPATNĚ! :-) = Rozšiřování (dědičnost) rozhraní =

Rozšiřování (dědičnost) rozhraní

- Rozhraní může převzít (můžeme říci též *dědit*) metody z existujících rozhraní.
- Říkáme, že rozhraní mohou být *rozšiřována* (*extended*).
- Rozšířené rozhraní by mělo *nabízet něco navíc* než to výchozí (rozšiřované).
- Příklad: Každá třída implementující `WellInforming` musí pak implementovat i metody z rozhraní `Informing`:

```
interface Informing {
  String retrieveInfo();
}
interface WellInforming extends Informing {
  String detailedInfo();
}
public class Person implements WellInforming {
  public String retrieveInfo() { ...  }
  public String detailedInfo() { ... }
}
```

Kde použít `implements` a kde `extends`

- Ztrácíte se v klíčových slovech?
- `implements` = třída implementuje rozhraní
  - ve třídě musím napsat kód (obsah) metod předepsaných rozhraním
- `extends` = když dědím ze třídy nebo rozšiřuji rozhraní
  - *dědím* metody automaticky

Vícenásobé rozšiřování rozhraní

- Rozhraní může dědit z více rozhraní zároveň:

```
public interface Informing { String retrieveInfo(); }
public interface Screaming { String scream(); }

public interface WellInforming extends Informing, Screaming {
  String detailedInfo();
}
```

- Každá třída implementující `WellInforming` musí implementovat všechny 3 metody.

Řetězení rozšiřování (dědičnosti)

- Dědičnost můžeme řetězit:

```
public interface Grandparent { int method1();  }
public interface Parent extends Grandparent { int method2(); }
public interface Child extends Parent { int method3(); }
```

- `Grandparent` pak obsahuje jednu metodu, `Parent` dvě, `Child` tři.

Kdy je vícenásobné rozšiřování možné

- Úplně stejné metody ze dvou rozhraní jsou OK

```
public interface A {
  void someMethod();
}
public interface B {
  void someMethod();
}
public interface AB extends A, B {
  // it is OK, methods have same signature
}
```

Kdy je vícenásobné rozšiřování možné

- Stejné metody s různými parametry ze dvou rozhraní jsou také OK

```
public interface A {
  void someMethod();
}
public interface B {
  void someMethod(int param);
}
public interface AB extends A, B {
  // it is OK, methods have different params
}
```

Kdy vícenásobné rozšiřování není možné

- Dvě metody lišící se jen návratovým typem nejsou OK
- Třída implementující rozhraní `AB` by musela mít dvě metody lišící se jen návratovým typem, a to nejde

```
public interface A {
  void someMethod();
}
public interface B {
  int someMethod();
}
public interface AB extends A, B {
  // cannot be compiled
}
= Rozhraní =
```

Motivace

- Jsou objektové jazyky, kde vůbec rozhraní nemáme.

- Jsou dokonce objektové jazyky, kde nemáme ani třídy (JavaScript).
- Většinou ale *třídy jsou*, u většiny OO jazyků: Java, C++, C#, Python, Ruby…
- Přímočaré řešení je pak použít na implementaci nějaké potřebné funkčnosti třídu.
- Problém je, že někdy máme takový požadavek na funkčnost, který se jednou třídou špatně implementuje, resp. evidentně potřebujeme více *zcela různorodých tříd*, které budou tento požadavek plnit.
- Příklad: požadavkem je, aby objekty uměly o sobě sdělit informace, tedy aby měly třeba metodu `retrieveInfo()`.
- Zcela jistě existuje mnoho typů objektů, které o sobě umějí informovat — od psa až po laserovou tiskárnu :-)

Příklad jednoduchého rozhraní

- Velmi jednoduché rozhraní s jedinou metodou:

```
// Informing = can describe information about itself
public interface Informing {
  // this method is used for describing
  String retrieveInfo();
}
```

Přesněji k rozhraní

- Rozhraní (`interface`) je *seznam metod* (budoucích) tříd objektů.
- *Neobsahují vlastní kód* těchto metod.
- Je to tedy *seznam hlaviček metod* s popisem, co přesně mají metody dělat, typicky dokumentačním komentářem.
- Rozhraní by nemělo příliš smyslu, kdybychom neměli třídy, které jej naplňují, realizují, přesněji *implementují*
- Říkáme, že třída *implementuje* rozhraní, pokud obsahuje *všechny metody*, které jsou daným rozhraním předepsány.
  - třída *implementuje rozhraní* = objekt dané třídy *se chová jak* rozhraní předepisuje,
  - např. osoba nebo pes se chová jako běhající

Deklarace rozhraní

- Stejně jako třída, jedině namísto slova `class` je `interface`.
- Všechny *metody v rozhraní přebírají viditelnost* z celého rozhraní:
  - viditelnost hlaviček metod není nutno psát;
  - rozhraní v našem kurzu budou pouze `public` (není to vůbec velká chyba tak dělat stále).

- Těla metod v deklaraci rozhraní se nepíší vůbec, ani složené závorky, jen středník za hlavičkou.

```
// Informing = can describe information about itself
public interface Informing {
  // this method is used for describing
  String retrieveInfo();
}
```

Implementace rozhraní

- Třídy `Person` a `Donkey` implementují rozhraní `Informing`:

```
public class Person implements Informing {
  public String retrieveInfo() {
    return "People are clever.";
  }
}
public class Donkey implements Informing {
  public String retrieveInfo() {
    return "Donkeys are simple.";
  }
}
```

Když třída implementuje rozhraní, musí implementovat *všechny* jeho metody!

Přetypování třídy na rozhraní

```
public void printInfo(Informing informing) {
  System.out.println("Now you learn the truth!");
  System.out.println(informing.retrieveInfo());
}
...
Person p = new Person();
printInfo(p);
...
Donkey d = new Donkey();
printInfo(d);
```

- Parametr metody je typu rozhraní, můžeme tam tedy použít *všechny třídy, které ho implementují*.
- To je velice časté a užitečné používat jako typ parametru rozhraní.

Přetypování obecně

- Píše se `(typ) hodnota`.
- Ve skutečnosti se nejedná o změnu obsahu objektu, nýbrž pouze o potvrzení (*běhovou typovou kontrolu*), že běhový typ objektu je ten požadovaný.

```
Person p = new Person();
Informing i = (Informing) p;
Object o = (Object) i;
```

U *primitivních typů* se jedná o skutečný převod, např. `long` přetypujeme na `int` a ořeže se tím rozsah.

Proměnná typu rozhraní

- Můžeme taky vytvořit proměnnou typu rozhraní:

```
public class Person implements Informing {
  public String retrieveInfo() {
    return "People are clever.";
  }
  public void emptyMethod() { }
}
...
Informing i = new Person();
i.retrieveInfo(); // ok
i.emptyMethod(); // cannot be done
```

- Proměnná `i` může používat pouze metody definované v rozhraní (ztrácí metody v třídě `Person`).
- To je opět velice časté a užitečné používat jako typ proměnné rozhraní.

Příklad z reálného světa

- máme různé tiskárny, různých značek
- nechceme psát kód, který ošetří všechny značky, všechny typy
- chceme použít i budoucí značky/typy
- vytvoříme pro všechny jedno uniformní rozhraní:

```
public interface Printer {
  void printDocument(File file);
  File[] getPendingDocuments();
}
```

- náš kód bude používat tohle rozhraní, každá tiskárna která ho implementuje, bude kompatibilní
- budoucí tiskárny, které rozhraní implementují ho budou moct používat také

Anotace `@Override`

- Pro jistotu, že *přepisujeme metodu předepsanou rozhraním* a ne jinou, použijeme znovu anotaci `@Override`:

```
public class Person implements Informing {
  @Override
  public String retrieveInfo() {
    return "People are clever.";
  }
}
```

Implementace více rozhraní I

- Jedna třída může implementovat *více rozhraní*.
- Jednoduše v případě, kdy objekty dané třídy toho "mají hodně umět".
- Příklad: Třída `Person` implementuje 2 rozhraní:

```
public interface Informing { String retrieveInfo(); }
public interface Screaming { String scream(); }

public class Person implements Informing, Screaming {
  public String retrieveInfo() { ... }
  public String scream() { ... }
}
```

Implementace více rozhraní II

- Co kdyby obě rozhraní měla stejnou metodu?

```
public interface Informing { String retrieveInfo(); }
public interface Screaming { String retrieveInfo(); }
```

- Mají-li úplně stejnou hlavičku, je to v pořádku:

```
public class Person implements Informing, Screaming {
  @Override
  public String retrieveInfo() { ... }
}
```

Implementace více rozhraní III

- Mají-li stejný název i parametry, ale různý návratový typ, je to PROBLÉM.

```
public interface Informing { String retrieveInfo(); }
public interface Screaming { void retrieveInfo(); }
public class Person implements Informing, Screaming { ... }
```

- To samozřejmě nelze — viz totéž u přetěžování metod:

```
Person p = new Person();
// do we call method returning void or
// string and we ignore the result?
p.retrieveInfo();
```

Nesnesou se tedy metody lišící se *pouze návratovým typem*.

Rozhraní — vtip

Metody i samotné rozhraní by mělo obsahovat kvalitní dokumentaci s detailním popisem.

> Rozhraní je jako vtip. Když ho třeba vysvělovat, není tak dobré.

Zajímavost — rozhraní bez metod

- Občas se kupodivu používají i prázdná rozhraní, *nepředepisující žádnou metodu*.
- Úplně bezúčelné to není — deklarace, že třída implementuje určité rozhraní, poskytuje typovou informaci o dané třídě.
- Např. `java.lang.Cloneable`, `java.io.Serializable`. = Testování jednotek, *JUnit* =

Testování software

- Účelem testování je obecně vzato zajistit bezchybný a spolehlivý software.
- Testování je naprosto *klíčová součást* SW vývoje.
- Je to rozsáhlá disciplína softwarového inženýrství sama o sobě.
- Některé postupy vývoje jsou přímo *řízené testy* (Test-driven Development).

- Zde v PB162 se zatím budeme věnovat jen *testování jednotek* programu.
- Bližší info v řadě předmětů na FI, např. PV260 Software Quality

Typy testování

- *Testování jednotek* (malé, ale ucelené kusy, např. třídy, samostatně testované) — dělá vývojář nebo tester
- *Integrační testování* (testování, jak se kus chová po zabudování do celku) — dělá vývojář nebo tester často ve spolupráci s architektem řešení
- *Akceptační testování* (při přijetí kódu zákazníkem) — dělá přebírající
- *Testování použitelnosti* (celá aplikace obsluhovaná uživatelem) — dělá uživatel či specialista na UX
- *Bezpečnostní testování* (zda neobsahuje bezpečnostní díry, odolnost proti útokům, robustnost) — dělá specialista na bezpečnost
- …

Testování jednotek

- Testování jednotek (*unit testing*) testuje jednotlivé elementární části kódu
  - elementární části = třídy a metody
- V Javě se nejčastěji používá volně dostupný balík JUnit.
- V nových produktech se používají verze JUnit 4.x nebo JUnit 5.
- Elementárním testem v JUnit je *testovací metoda* opatřena anotací `@Test`.

Jednoduchý příklad `JUnit` testu

- `@Test` před metodou označí tuto metodu za *testovací*.
- Metoda se nemusí nijak speciálně jmenovat (žádné `testXXX` jako dříve není nutné).

```
@Test
public void minimalValueIs5() {
  Assert.assertEquals(5, Math.min(7, 5));
}
```

- Metoda `assertEquals` bere 2 parametry
  - očekávanou (expected) hodnotu — v příkladu `5` a
  - skutečnou (actual) hodnotu — v příkladu `Math.min(7, 5)`.
- Pokud hodnoty nejsou stejné, test selže.
- Může přitom vydat hlášku, co se vlastně stalo.

Příklad Calculator — testovaná třída

- Testovaná třída `Calculator`, jednoduchá sčítačka čísel:

```
public class Calculator {
  public int evaluate(String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
      sum += Integer.valueOf(summand);
    return sum;
  }
}
...
new Calculator().evaluate("1+2"); // returns 3
```

Řetězec `"\\+"` je pouhý regulární výraz reprezentující znak `+`.

Příklad Calculator — testovací třída

- Třída testující `Calculator`:

```
public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    Assert.assertEquals(6, sum);
  }
}
```

Zdroj: junit wiki.

`assert` metody

- Jak bylo vidět, pro ověření, zda během provádějí testu platí různé podmínky, jsem používali volání `Assert.assertXXX`.
- Z jejich názvů je intuitivně patrné, co vlastně ověřují.
- Jsou realizovány jako statické metody třídy `Assert` z JUnit:
  - `assertTrue`
  - `assertFalse`
  - `assertNull`
  - ...

Import statických metod `Assert`

- Abychom si ušetřili psaní názvu třídy `Assert`, můžeme potřebné statické metody importovat

```
import static org.junit.Assert.assertEquals;
public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

Javové testování bude podrobně probíráno v dalším kurzu — PV168. = Dědičnost =

Vtip

> Dědičnost. Nejlepší objektově-orientovaný způsob, jak zbohatnout.

Dědičnost

- Objektové třídy jsou obvykle **podtřídami**, tj. speciálními případy jiných tříd:

```
class Subclass extends Superclass
```

```
class DogKeeper extends Person {
// methods & attributes for DogKeeper
// in addition to Person
}
```

- Všechny objekty typu `DogKeeper` jsou současně typu `Person`.

```
Person p = new DogKeeper("Karel");
```

V Javě dědí **každá** třída od třídy `Object`.

Definice

- **Nadtřída** = superclass, "bezprostřední předek", "rodičovská třída"
- **Podtřída** = subclass, "bezprostřední potomek", "dceřinná třída"

    - je specializací své nadtřídy
    - přebírá vlastnosti nadtřídy
    - zpravidla přidává další vlastnosti, čímž nadtřídu *rozšiřuje* (`extends`)

Správné použití

- Dědičnost by měla splňovat vztah **je (is a)** — Chovatel psa *je* osoba.
- Při atributech třídy se používá vztah **má (has a)** — Osoba *má* jméno.

Proč používat dědičnost?

- Abychom zohlednili **konceptuální vztah** *obecnější* vs. *speciálnější* typ.
- Abychom se **vyhnuli opakování kódu** a dosáhli *znovupoužití* (= kód metod a atributů se podědí, nemusíme jej znovu psát).
- Mělo by platit oboje, aby mělo smysl dědičnost použít.

Tranzitivní dědění

Dědění může být vícegenerační:

```
public class Manager extends Employee { ... }

public class Employee extends Person { ... }
```

Manažer podědí metody a atributy ze třídy *Zaměstnanec* i *Osoba*.

Vícenásobná dědičnost

- Java vícenásobnou dědičnost u tříd **nepodporuje**!
- Důvodem je problém typu diamant:

```
class DoggyManager extends Employee, DogKeeper { }
class Employee  { public String toString() { "Employee";  } }
class DogKeeper { public String toString() { "DogKeeper"; } }

DoggyManager().toString(); // we have a problem!
```

- Vícenásobná dědičnost je možná jedině u rozhraní (metody zatím nemají definovanou implementaci).

Dědičnost a vlastnosti tříd

- Dědičnost (v kontextu Javy) znamená:

1. potomek dědí **všechny** vlastnosti nadtřídy
2. poděděné vlastnosti potomka se **mohou změnit** (např. překrytím metody)
3. potomek může **přidat** další vlastnosti

- **Pozn.:** *Vlastnosti třídy* = metody & atributy třídy

Dědičnost vs. rozhraní

Dědičnost

- vyhýbáme se duplikaci kódu
- kód je kratší
- když potřebuji upravit předka, musím upravit změny ve všech potomcích, co může být netriviální

Použití rozhraní

- méně závislostí, více kódu
- více používné v praxi

Příklad s `Account`

- Vylepšíme třídu `Account` tak, aby zůstatek nebyl nižší než minimální zůstatek
- Realizujeme tedy změnu metody `debit` pomocí jejího *překrytí* (overriding)
- Stará třída `Account`:

```
public class Account implements Informing {
   private int balance;
   ...
   public boolean debit(int amount) {
      if(amount <= 0) return false;
      balance -= amount;
      return true;
   }
}
```

Nová třída `CheckedAccount`

```
public class CheckedAccount extends Account {
  private int minimalBalance;
  public CheckedAccount(Person owner, int minBal, int initBal) {
    super(owner, initBal); // calling Account constructor
    if(initBal < minBal) { // is initial balance sufficient?
       throw new IllegalArgumentException("low initial balance");
    }
    this.minimalBalance = minBal;
  }

  @Override
  public boolean debit(int amount) {
    // check min. balance
    if(getBalance() - amount >= minimalBalance) {
       return super.debit(amount); // the debit is deducted
    } else return false;
  }
}
```

Co tam bylo nového

- Klíčové slovo `extends` značí, že třída `CheckedAccount` je potomkem (*subclass*) třídy `Account`.
- Konstrukce `super.metoda(⋯);` značí, že je volána metoda předka, tedy třídy `Account`.
- Kdyby se `super` nepoužilo, zavolala by se metoda `debit` třídy `CheckedAccount` a program by se zacyklil!

V konstruktoru potomka je **vždy** nutno zavolat konstruktor předka, protože se v něm inicializují vlastnosti dané třídy.

Konstruktor předka je nutno zavolat jako **první**, protože by se pak dali použít vlastnosti, které zatím nejsou nainicializované.

Kompozice

Často se používá *skládání* (kompozice) objektů, kdy objekt **nedědí**, ale nese odkaz na jiný objekt.

```
public class CheckedAccount {

  private int minimalBalance;
  private Account account;

  public CheckedAccount(Person owner, int minBal, int initBal) {
    account = new Account(owner, initBal);
    ...
  }
  // account.debit(amount)
}
```

Kompozicí se zabývá navazující kurz *PV168 Seminář Javy*. = Viditelnost (práva přístupu) =

Základní principy OOP

- **Dědičnost** umožňuje vztah *X is Y* (X je Y) mezi objekty.
  - Umožňuje rozšířit již existující třídu, tedy vytvořit její podtřídu (podtyp), která od svého rodiče zdědí jeho atributy a metody
  - *Proč?* Znovupoužití kódu.
- **Polymorfismus** je schopnost objektu měnit chování počas běhu.
  - `Object o = new String("String, but can be Object")`.
  - Na místo, kde je očekávána instance třídy `Object`, je možné dosadit instanci jakékoli její podtřídy. Odkazovaný objekt se chová podle toho, jaké třídy je instancí.
- **Zapouzdření** je zabalení dat a metod do jedné komponenty (třídy), souvisí s viditelností objektů.
  - Cokoli, co nemusí být viditelné, nemá být viditelné.
  - *Proč?* Vede ke skrývání dat, informací.

Viditelnost

- Použití *tříd* i jejich metod a atributů lze regulovat (uvedením tzv. *modifikátoru přístupu*).
- Nastavením správné viditelnosti jsme schopni docílit **zapouzdření**.
- Omezení viditelnosti je *kontrolováno při překladu* → není-li přístup povolen, nelze program přeložit.
- Metody i atributy uvnitř třídy mohou mít viditelnost stejnou jako třída nebo nižší.

Typy viditelnosti / přístupu

Existují čtyři možnosti:

- `public` = veřejný

- `protected` = chráněný
- *modifikátor neuveden* = říká se *privátní v balíku* (package-private)
- `private` = soukromý

Tabulka viditelností

Table 1. Access Levels table

Modifier	Class	Package	Subclass (diff. package)	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

- např. atribut typu `private` je viditelný pouze v rámci dané třídy

Třídy nemohou být `protected`!

Použití typů viditelnosti v tomto kurzu

public
: třídy/rozhraní, metody, konstanty

private
: atributy, metody, konstanty

protected
: pravděpobně nebudeme používat, výjimečně metody, atributy

package-private
: pravděpobně nebudeme používat

Veřejný, `public`

- Přístupné odevšad.

```
public class Account { ... }
```

- U třídy `Account` lze např.
  - vytvořit objekt typu `Account` v metodě jiné třídy
  - deklarovat podtřídu třídy `Account` ve stejném i jiném balíku
- ne všechny vlastnosti uvnitř `Account` musejí vždy být veřejné

- veřejné bývají obvykle některé *konstruktory* a některé *metody*
- veřejné jsou typicky metody předepsané implementovaným *rozhraním*
- třídy deklarované jako *veřejné* musí být umístěné do souboru s totožným názvem: `Account.java`

Soukromý, `private`

- Viditelné **jen** v rámci třídy.

```
public class Account {
   private String owner;
   ...
   public void add(Account another) {
     another.owner; // can be accessed!
     ...
   }
}
```

- K atributu `owner` nelze přistoupit v podtřídě, pouze v dané třídě.
- Pro zpřístupnění proměnné pro "vnější" potřeby je nutno použít gettery/settery.
- Skrýváme konkrétní implementaci datové položky.
- Např. metoda `getAge()` nemusí existovat jako proměnná, ale může se v případě volání spočítat.

Volbou `private` nic zásadně nepokazíme.

Soukromé třídy

Třídy mohou mít viditelnost `private`.

Proč by někdo chtěl privátní třídu?

```
public class SomeClass {
  private class InnerDataStructure { ... }
  // code using InnerDataStructure
}
```

- Používá se u vnořených tříd (tříd uvnitř tříd).
- Mimo rozsah předmětu, nebudeme používat!

Ve stejném souboru může být libovolný počet deklarací neveřejných tříd. Není to však hezké.

Lokální v balíku, `package-private`

- Přístupné jen ze tříd **stejného balíku**, používá se málo.
- Jsou-li podtřídy v jiném balíku, třída není přístupná!

```
package cz.some.pkg;

class Account {
// package-private class
// available only in cz.some.pkg
}
```

- Svazuje viditelnost s organizací do balíků (ta se může měnit častěji než např. vztah *nadtřída-podtřída*).
- Občasné využití, když nechceme mít konstruktor `private` nebo rozhraní `public`.

Chráněný, `protected`

- Viditelnost `protected` , tj. přístupné jen z *podtříd* a *tříd stejného balíku*.

```
public class Account {
   // attribute can be protected (but it is better to have it private)
   protected float creditLimit;
}
```

- U *metod* tam, kde se nutně očekává použití z podtříd nebo překrývání.
- Vcelku často u *konstruktorů* — často se volá právě ze stejné (pod)třídy.

Shrnutí viditelnosti

Obvykle se řídíme následujícím:

metoda

- obvykle `public`, je-li užitečná i mimo třídu či balík
- `protected` je-li je vhodná k překrytí v případných podtřídách
- jinak `private`

atribut

- obvykle `private`
- výjimečně `protected`, je-li potřeba přímý přístup v podtřídě

třída

- obvykle `public`

- výjimečně `package-private` nebo `private`

Abstraktní třídy

Motivace

- Java disponuje rozhraními.
- Pak máme třídu(y) implementující určité rozhraní.
- Někdy je vhodné určité rozhraní implementovat pouze *částečně*:
  - *Rozhraní* = Specifikace
  - *Abstraktní třída* = Částečná implementace rozhraní (stačí mít hotové některé metody) a současně předek konkrétních tříd, tedy plných implementací
  - *Neabstraktní třída* = Úplná implementace rozhraní (musí mít hotové všechny metody)

Zápis abstraktní třídy

- Abstraktní třída je označena klíčovým slovem `abstract` v hlavičce, např.:

```
public abstract class AbstractSearcher
```

- Název začínající na `Abstract` není povinný ani nutný.
- Abstraktní třída má obvykle alespoň jednu *abstraktní metodu*, deklarovanou např.:

```
public abstract int indexOf(double d);
```

- Od abstraktní třídy *nelze vytvořit instanci*, (chybí implementace některých metod) nelze napsat např.:

```
Searcher ch = new AbstractSearcher(...);
```

Reálný příklad: rozhraní → abstraktní třída → neabstraktní třída

- `Searcher` = rozhraní — specifikuje, co má prohledávač umět
- `AbstractSearcher` = abstraktní třída — předek konkrétních plných implementací prohledávače
- `LinearSearcher` = konkrétní třída — plná implementace prohledávače

Searcher

`Searcher` je rozhraní = specifikuje, co má prohledávač umět

```
public interface Searcher {
   // Set the array for later searching
   void setData(double[] a);
   // Check whether array contains d element
   boolean contains(double d);
   // Return the position of d in the array (or -1 if not found)
   int indexOf(double d);
}
```

AbstractSearcher

`AbstractSearcher` je abstraktní třída = předek konkrétních plných implementací prohledávače

```
// this class implements Searcher only partially
public abstract class AbstractSearcher implements Searcher {
   // array, its getters and setters are implemented
   private double[] array;
   public void setData(double[] a) { array = a; }
   public double[] getData() { return array; }
   // we can call indexOf now - it will be implemented leter
   public boolean contains(double d) {
      return indexOf(d) >= 0;
   }
   // finding the position of d is NOT implemented yet!
   public abstract int indexOf(double d);
}
```

LinearSearcher

`LinearSearcher` je konkrétní třída = plná implementace prohledávače, pomocí lineárního prohledání

```
public class LinearSearcher extends AbstractSearcher {
   // class has to implement all abstract methods
   public int indexOf(double d) {
      double[] data = getData();
      for(int i = 0; i < data.length; i++) {
         if(data[i] == d) {
            return i;
         }
      }
      return -1;
   }
}
= Pole, třída `Arrays` =
```

Opakování

- Vytvoření, naplnění a získání hodnot vypadá následovně:

```
int[] array = new int[2];
array[0] = 1;
array[1] = 4;
System.out.println("First element is: " + array[0]);
```

- Deklarace: *typ*[] *jméno* = `new` *typ* [*velikost*];
- *typ* může být i objektový: `Person[] p = new Person[3];`

Velikost pole

- *velikost* pole je daná při jejím vytvoření a **nelze ji měnit**
- V budoucnu budeme probírat *kolekce* (seznam, slovník), což je mocnější složený datový typ než pole
  - jejich počty prvků se mohou dynamicky měnit

Kopírování odkazu

- Přiřazení proměnné objektového typu (což je i pole) vede pouze k **duplikaci odkazu**, nikoli celého odkazovaného objektu.
- Modifikace pole přes jednu proměnnou se pak projeví i té druhé.

```
int[] array = new int[] {1, 4, 7};
int[] array2 = array;
array[1] = 100;
System.out.println(array[1]); // prints 100
System.out.println(array2[1]); // prints 100
```

Kopie pole

Pomocí Arrays.copyOf můžeme vytvořit kopii pole Kopie vznikne tak, že se vytvoří nové pole a do něj se nakopírují položky z původního pole.

```
int[] array = new int[] {1, 4, 7};
int[] array2 = Arrays.copyOf(array, array.length);
array[1] = 100;
System.out.println(array[1]); // prints 100
System.out.println(array2[1]); // prints 4
```

Metoda `copyOf` bere dva parametry — původní pole a počet prvků, kolik se má nakopírovat.

Kopie u objektů I

- Obdobně to funguje i objektů.

```
Person[] people = new Person[] { new Person("Jan"), new Person("Adam")};
Person[] people2 = Arrays.copyOf(people, people.length);
people[1] = new Person("Pepa");
System.out.println(people[1].getName()); // prints Pepa
System.out.println(people2[1].getName()); // prints Adam
```

- Co kdybychom změnili jenom *jméno* (atribut objektu)?

Kopie u objektů II

- Do cílového pole se zduplikují jenom *odkazy na objekty* `Person`, nevytvoří se kopie objektů `Person`!

```
Person[] people = new Person[] { new Person("Jan"), new Person("Adam")};
Person[] people2 = Arrays.copyOf(people, people.length);
people[1].setName("Pepa"); // changes Adam to Pepa
System.out.println(people[1].getName()); // prints Pepa
System.out.println(people2[1].getName()); // prints Pepa
```

- Jinými slovy, pole mají sice *různý odkaz* (šipku), ale na **stejný objekt**.

- V předešlém příkladu jsme změnili odkaz na jiný objekt.
- Teď jsme změnili obsah objektu, na který ukazují oba odkazy.

Třída `Arrays`

- nabízí jen statické metody a proměnné, tzv. utility class
- nelze od ní vytvářet instance
- metody jsou implementovány pro všechny primitivní typy i objekty
  - pro jednoduchost použijeme pole typu `long`

Javadoc třídy `Arrays`

Metody třídy `Arrays` I

- `String toString(long[] a)`
  - vrátí textovou reprezentaci
- `long[] copyOf(long[] original, int newLength)`
  - nakopíruje pole `original`, vezme prvních `newLength` prvků
- `long[] copyOfRange(long[] original, int from, int to)`
  - nakopíruje prvky `from`-`to`
  - nové pole vrátí
- `void fill(long[] a, long val)`
  - naplní pole `a` hodnotami `val`

Metody třídy `Arrays` II

- `boolean equals(long[] a, long[] a2)`
  - `true` jestli jsou pole stejná
- `int hashCode(long[] a)`
  - haš pole
- `void sort(long[] a)`
  - setřídí pole
- `⋯ asList(⋯)`
  - z pole vytvoří kolekci (budou probírány později)

Příklad

```
long[] a1 = new long[] { 1L, 5L, 2L };
a1.toString(); // [J@4c75cab9
Arrays.toString(a1); // [1, 5, 2]

long[] a2 = Arrays.copyOf(a1, a1.length);
Arrays.equals(a1, a2); // true

Arrays.fill(a2, 3L); // [3, 3, 3]
Arrays.sort(a1); // [1, 2, 5]
```

Porovnávání v Javě

Porovnávání a pořadí (uspořádání)

- Obecně rozlišujeme, zda chceme zjišťovat *shodnost* (*rovnost*, *ekvivalenci*):
  - mezi dvěma *primitivními hodnotami*
  - mezi dvěma *objekty*
- U *primitivních hodnot* jsou rovnost i pořadí určeny **napevno** a nelze je změnit.
- U *objektů* lze porovnání i uspořádání programově určovat.

Rovnost primitivních hodnot

- Rovnost primitivních hodnot zjišťujeme pomocí operátorů:
  - `==` (rovná se)
  - `!=` (nerovná se)
- U integrálních typů funguje bez potíží
- U čísel floating-point (`double`, `float`) je třeba porovnávat s určitou tolerancí

```
1 == 1 // true
1 == 2 // false
1 != 2 // true

1.000001 == 1.000001 // true
1.000001 == 1.000002 // false
Math.abs(1.000001 - 1.000002) < 0.001 // true
```

Uspořádání primitivních hodnot

- Uspořádání primitivních hodnot funguje pomocí operátorů `<`, `⇐`, `>=`, `>`
- U *primitivních hodnot* nelze koncept uspořádání ani rovnosti programově měnit.

Uspořádání není definováno na typu `boolean`, tj. neplatí `false < true`!

```
1.000001 <= 1.000002  // true
```

Jak chápat rovnost objektů

Identita objektů, `==`

vrací `true` při rovnosti **odkazů**, tj. když oba odkazy **ukazují na tentýž objekt**

Rovnost obsahu, metoda `equals`

vrací `true` pri logické ekvivalenci objektů (musí být explicitně nadefinované)

```
Person pepa1 = new Person("Pepa");
Person pepa2 = new Person("Pepa");
Person pepa3 = pepa1;

pepa1 == pepa2; // false
pepa1 == pepa3; // true
```

Porovnávání objektů pomocí `==`

- Porovnáme-li dva objekty prostřednictvím operátoru `==` dostaneme rovnost jen v případě, jedná-li se o dva odkazy na tentýž objekt.
- Jedná-li se o dva byť *obsahově stejné objekty*, ale existující samostatně, pak `==` vrátí `false`.

Objekty jsou identické = odkazy obsahují stejnou adresu objektu.

Porovnávání objektů dle obsahu

Rovnost obsahu, metoda `equals`:

- Dva objekty jsou *rovné* (*rovnocenné*), mají-li stejný obsah.
- Na zjištění rovnosti se použije metoda `equals`, kterou je potřeba překrýt.
- Pro nadefinování rovnosti bude hlavička metody **vždy** vypadat následovně:

```
@Override
public boolean equals(Object o)
```

- Parametrem je objekt typu `Object`.
- Jestli parametr není objekt typu `<class-name>`, obvykle je potřeba vrátit `false`.
- Pak se porovnají jednotlivé vlastnosti objektů a jestli jsou stejné, metoda vráti `true`.

Porovnávání objektů — komplexní číslo

Dvě komplexní čísla jsou stejná, když mají stejnou reálnou i imaginární část.

```
public class ComplexNumber {
  private int real, imag;

  public ComplexNumber(int r, int i) {
    real = r; imag = i;
  }
  @Override
  public boolean equals(Object o) {
    if (this.getClass() != o.getClass()) return false;

    ComplexNumber that = (ComplexNumber) o;
    return this.real == that.real
        && this.imag == that.imag;
  }
}
```

Porovnávání objektů — osoba I

Popis kódu na následujícím slajdu:

- Dvě osoby budou stejné, když mají stejné jméno a rok narození.
- Rovnost nemusí obsahovat porovnání všech atributů (porovnání `age` je zbytečné, když máme `yearBorn`).
- `String` je objekt, proto pro porovnání musíme použít metodu `equals`.
- Klíčové slovo `instanceof` říká "mohu pretypovat na daný typ".

Metoda `equals` musí být reflexivní, symetrická i tranzitivní (javadoc).

Porovnávání objektů — osoba II

```
public class Person {
  private String name;
  private int yearBorn, age;

  public Person(String n, int yB) {
    name = n; yearBorn = yB; age = currentYear - yB;
  }
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Person)) return false;

    Person that = (Person) o;
    return this.name.equals(that.name)
        && this.yearBorn == that.yearBorn;
  }
}
```

Porovnávání objektů — použití

```
ComplexNumber cn1 = new ComplexNumber(1, 7);
ComplexNumber cn2 = new ComplexNumber(1, 7);
ComplexNumber cn3 = new ComplexNumber(1, 42);
cn1.equals(cn2); // true
cn1.equals(cn3); // false

Person karel1 = new Person("Karel", 1993);
Person karel2 = new Person("Karel", 1993);

karel1.equals(karel2); // true

karel1.equals(cn1); // false
cn2.equals(karel2); // false
```

Chybějící `equals`

Co když zavolám metodu `equals` aniž bych ji přepsal?

Použije se původní metoda `equals` ve třídě `Object`:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

Původní `equals` funguje přísným způsobem — rovné jsou jen identické objekty.

Jak porovnat typ třídy

Je `this.getClass() == o.getClass()` stejné jako `o instanceof Person`?

- Ne! Jestli třída `Manager` dědí od `Person`, pak:

```
manager.getClass() == person.getClass() // false
manager instanceof Person // true
```

- Co tedy používat?
  - `instanceof` porušuje symetrii `x.equals(y) == y.equals(x)`
  - `getClass` porušuje tzv. *Liskov substitution principle*
- Záleží tedy na konkrétní situaci.

Metoda `hashCode`

- Při překrytí metody `equals` nastává dosud nezmíněný problém.
- Jakmile překryjeme metodu `equals`, měli bychom současně překrýt i metodu `hashCode`.
- Metoda `hashCode` je také ve třídě `Object`, tudíž ji obsahuje každá třída.

```
@Override
public int hashCode()
```

- Metoda vrací celé číslo pro daný objekt tak, aby:
  - pro dva stejné (`equals`) objekty musí **vždy** vrátit *stejnou hodnotu*
  - jinak by metoda měla vracet _různé hodnoty (není to nezbytné a ani nemůže být vždy splněno)
  - složité třídy mají více různých objektů než je všech hodnot typu `int`

Příklad `hashCode` I

```
public class ComplexNumber {
  private int real;
  private int imag;

  ...

  @Override
  public boolean equals(Object o) { ... }

  @Override
  public int hashCode() {
    return 31*real + imag;
  }
}
```

Příklad hashCode II

```
public class Person {
  private String name;
  private int yearBorn, age;
  ...
  @Override
  public boolean equals(Object o) { ... }

  @Override
  public int hashCode() {
    int hash = name.hashCode();
    hash += 31*hash + yearBorn;
    return hash;
  }
}
```

Obecný hashCode

Nejlépe je vytvářet metodu následujícím způsobem (31 je prvočíslo):

```
  @Override
  public int hashCode() {
    int hash = attribute1;
    hash += 31 * hash + attribute2;
    hash += 31 * hash + attribute3;
    hash += 31 * hash + attribute4;
    return hash;
  }
```

A nebo ji generovat (pokud víte, co to dělá :-))

Proč `hashCode`

- Metoda se používá v hašovacích tabulkách, využívá ji například množina `HashSet`.
- Při zjištění, jestli se prvek X nachází v množině, metoda vypočítá její haš (*hash*).
- Pak vezme všechny prvky se stejným hašem a zavolá `equals` (haš mohl být stejný náhodou).
- Jestli má každý objekt **unikátní** haš, pak je tato operace **konstantní**.
- Jestli má každý objekt **stejný** haš, pak je operace `contains` **lineární**!

Jestli se `hashCode` napíše špatně (nevrací pro stejné objekty stejný haš) nebo zapomene — množina nad danou třídou přestane fungovat!

Uspořádání objektů

- Budeme probírat později
- V Javě neexistuje přetěžování operátorů `<, ⇐, >, >=`
- Třída musí implementovat rozhraní `Comparable` a její metodu `compareTo` = Výchozí a statické metody rozhraní =

Výchozí a statické metody rozhraní

- Jde o možnosti, jak **do rozhraní přímo implementovat funkčnost**, nenechávat to až na třídy.
- Popírá základní princip, že *v rozhraní funkční kód metod není.*
  - ⇒ má omezené použití a nemělo by se zneužívat
- Existují dva základní typy metod:
  - statické — `static`
  - výchozí — `default`

Statické metody

- Rozhraní může obsahovat *statické metody*.
- Statické metody smějí pracovat jen s dalšími statickými metodami a proměnnými.
- Nesmějí pracovat s metodami a atributy objektu.

```
interface A {
  void methodToImplement();
  static void someStaticMethod() {
    /* code inside */
  }
}
...
A.someStaticMethod();
```

Výchozí metody — motivace

- Nechť existuje rozhraní, které implementuje 10 tříd.
- Do rozhraní chceme přidat novou metodu.
- Metoda musí být (bohužel) implementována ve všech rozhraních!
- Co kdyby rozhraní poskytovalo i svou **výchozí implementaci**, kterou by třídy nemuseli implementovat?
- výchozí = `default`

Oracle The Java Tutorial: Default Methods

Výchozí metody — příklad

Výchozí metodu můžeme samozřejmě ve třídách překrýt.

```
interface Addressable {

  String getStreet();

  String getCity();

  default String getFullAddress() {
    return getStreet() +", " + getCity();
  }
}
```

Zdroj: Java SE 8's New Language Features

Výchozí metody — použití

Výchozí metody používáme, když chceme:

- **přidat novou metodu do existujícího rozhraní**
  - všechny třídy implementující rozhraní pak nemusí implementovat novou metodu
- **nahradit abstraktní třídu za rozhraní**
  - abstraktní třída vynucuje dědičnost
  - preferujeme implementaci rozhraní před dědičností tříd

Statické a výchozí metody

Statické metody se mohou v rozhraní využít při psaní výchozích metod:

```
interface A {
  static void someStaticMethod() {
    /* some stuff */
  }
  default void someMethod() {
    // can call static method
    someStaticMethod();
  }
}
```

Rozšiřování rozhraní s výchozí metodou

- Mějme rozhraní `A` obsahující výchozí metodu `defaultMethod()`.
- Definujme-li rozhraní `B` jako rozšíření rozhraní `A`, mohou nastat 3 různé situace:
  1. Jestliže výchozí metodu `defaultMethod()` v rozhraní `B` nezmiňujeme, pak se podědí z `A`.
  2. V rozhraní `B` uvedeme metodu `defaultMethod()`, ale *jen její hlavičku* (ne tělo). Pak ji nepodědíme, stane se **abstraktní** jako u každé obyčejné metody v rozhraní a každá třída implementující rozhraní `B` ji *musí sama implementovat*.
  3. V rozhraní `B` implementujeme metodu znovu, čímž se původní výchozí metoda překryje — jako při dědění mezi třídami.

Více výchozích metod — chybně

Následující kód se **nezkompiluje**:

```
interface A {
   default void someMethod() { /*bla bla*/ }
}
interface B {
   default void someMethod() { /*bla bla*/ }
}
class C implements A, B {
   // compiler does not know which default method should be used
}
```

Více výchozích metod — překryté, OK

Následující kód je zkompiluje:

```
interface A {
   default void someMethod() { /*bla bla*/ }
}
interface B {
   default void someMethod() { /*bla bla*/ }
}
class D implements A, B {
   @Override
   public void someMethod() {
   // now we can define the behaviour
   A.super.someMethod();
   }
}
```

Jedna metoda výchozí, druhá abstraktní

- Následující kód se opět nezkompiluje.
- Jedno rozhraní default metodu má a druhé ne.

```
interface A { void someMethod(); }
interface B { default void someMethod() { /* whatever */ } }
class E implements A, B {
 // compiler should or should not use default method?
}
= Kontejnery obecně, rozhraní `Collection` =
```

Kontejnery (dynamické datové struktury)

Co jsou kontejnery, kolekce (collections)?

- *dynamické datové struktury* vhodné k ukládání proměnného počtu objektů (přesněji odkazů na objekty)
- jsou automaticky ukládané v operační paměti (ne na disk)

Proč je používat?

- pro uchování **proměnného počtu** objektů (počet prvků se může měnit — zvyšovat, snižovat)
- oproti polím nabízejí efektivnější algoritmy přístupu k prvkům

Kvalitní seznámení s kontejnery najdete na stránkach Oracle

Základní kategorie

Základní kategorie jsou dány tím, které **rozhraní** daný kontejner implementuje:

Seznam (`List<E>`)

lineární struktura, každý prvek má svůj číselný index (pozici)

Množina (`Set<E>`)

struktura bez duplicitních hodnot a (obecně) bez uspořádání

Mapa, slovník, asociativní pole (`Map<K,V>`)

struktura uchovávající dvojice (klíč → hodnota), rychlý přístup přes klíč

Vlastní implementace se dá vytvořit implementováním zmíněných rozhraní.

Typové parametry

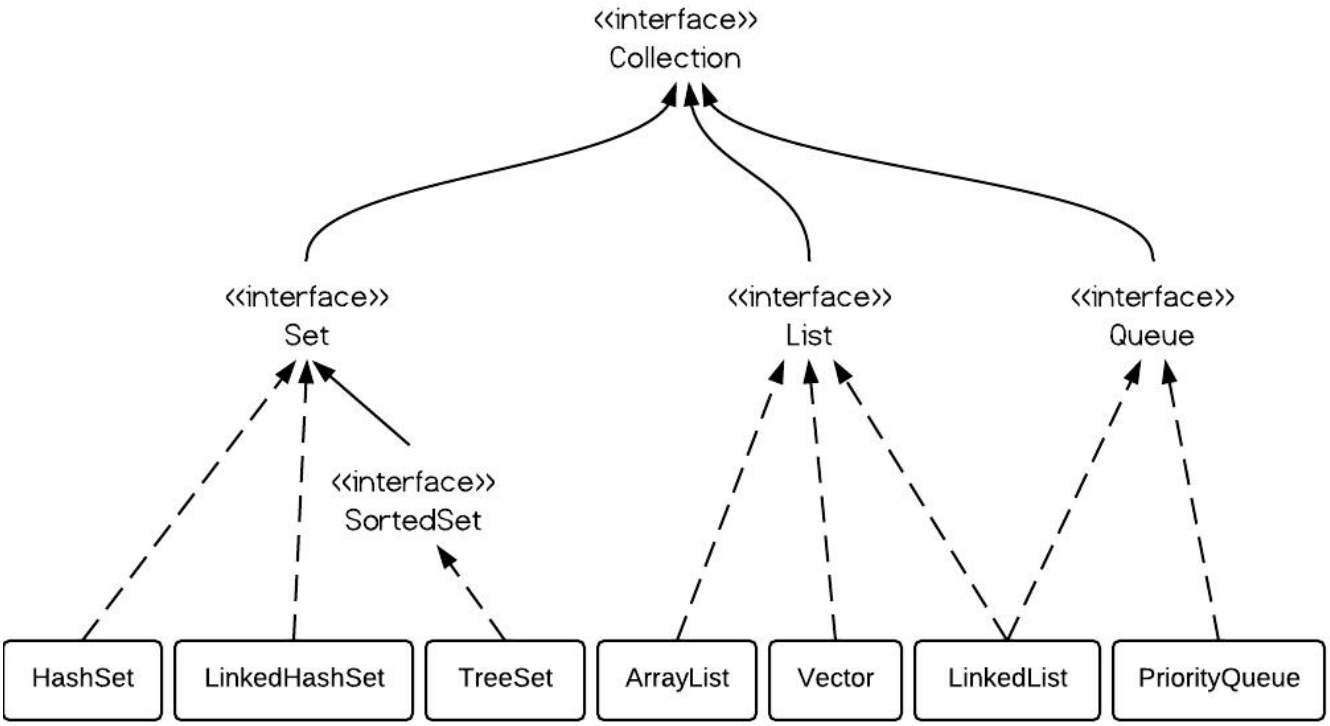
- v Javě byly dříve kontejnery koncipovány jako *beztypové*
- teď mají *typové parametry* ve špičatých závorkách (např. `Set<Person>`)
- určují, jaký typ položek se do kontejneru smí dostat

```
List objects; // without type, old!
List<String> strings; // with type String
```

Kontejnery bez typů nepoužívat! (Vždy používejte špičaté závorky.)

Hierarchie kolekcí

Třídy jsou součástí Java Core API (balík `java.util`).



`Collection`

- Mezi třídy typu rozhraní `Collection` patří:
  - rozhraní `List, Set,`... (ne však `Map`!)
  - konkrétní implementace `ArrayList, LinkedList, HashSet,`...
- Vytvoření **prázdné** kolekce:

```
List<String> listOfStrings = new ArrayList<>();
Collection<String> collection = new HashSet<>();
List<String> listWithValues = List.of("so", "cool");
```

Metody `Collection` I

Kolekce prvků typu E (tj. `Collection<E>`) má následující metody:

- `boolean add(E e)`
  - přidá prvek `e` do kolekce
  - `true` jestli se skutečne přidal (využití u množin)
- `int size()`
  - vrátí počet prvků v kolekci
- `boolean isEmpty()`
  - `true` jestli je kolekce prázdná (velikost je 0)

Příklad metod `Collection` I

```
Collection<String> set = new HashSet<>();
set.add("Pacman"); // true
set.add("Pacman"); // false
set.toString(); //  ["Pacman"]

set.size(); // 1
set.isEmpty(); // false
```

Metody `Collection` II

- `void clear()`
  - odstraní všechny prvky z kolekce
- `boolean contains(Object o)`
  - `true` jestli se `o` nachází v kolekci
  - na test rovnosti se použije `equals`

- `boolean remove(Object o)`
  - odstraní prvek z kolekce
  - `true` jestli byl prvek odstraněn

Příklad metod `Collection` II

```
List<String> list = List.of("Hello", "world");
list.toString(); //  ["Hello", "world"]

list.contains("Hello"); // true
list.remove("Hello");   // true

list.toString(); //  ["world"]
list.contains("Hello"); // false

list.clear();
list.toString(); //  []
```

Metody `Collection` III

- `Iterator<E> iterator()`
  - vrací něco, co se dá iterovat (procházet for-each cyklem)
- `T[] toArray(T[] a)`
  - vrátí pole typu `T`
  - konverze kolekce na pole
  - použití:

```
Collection<String> c = List.of("a", "b", "c");
String[] stringArray = c.toArray(new String[0]);
// stringArray contains "a", "b", "c" elements
```

Metody `Collection` IV

- `boolean containsAll(Collection<?> c)`
  - `true` jestli kolekce obsahuje všechny prvky z `c`

Metody vracející `true`, když byla kolekce změněna:

- `boolean addAll(Collection<E> c)`
  - přidá do kolekce všechny prvky z `c`
- `boolean removeAll(Collection<?> c)`

    - udělá rozdíl kolekcí (`this - c`)
- `boolean retainAll(Collection<?> c)`
    - udělá průnik kolekcí

Výraz `Collection<?>` reprezentuje kolekci jakéhokoliv typu.

Příklad metod `Collection` IV

```
Collection<String> c1 = List.of("A", "A", "B");
Collection<String> c2 = Set.of("A", "B", "C");
c1.containsAll(c2); // false
c2.containsAll(c1); // true

c1.retainAll(c2); // true
// c1 ["A", "B"]

c1.removeAll(c2); // true
// c1 []

c1.addAll(c2); // true
// c1 ["A", "B", "C"]
```

Seznam, množina, iterátory

Potomci `Collection`

Podívejme se na potomky třídy `Collection`, konkrétně:

- rozhraní `List`, implementace:
    - `ArrayList`
    - `LinkedList`
- rozhraní `Set`, implementace:
    - `HashSet`

Seznam, `List`

- něco jako dynamické pole
- každý uložený prvek má svou pozici — **číselný index**
- index je celočíselný, nezáporný, typu `int`
- možnost procházení seznamu dopředně i zpětně
- lze pracovat i s *podseznamy*:

```
List<E> subList(int fromIndex, int toIndex)
```

Implementace seznamu — `ArrayList`

- nejpoužívanější implementace seznamu
- využívá **pole** pro uchování prvků
- při zvětšování/zmenšování se vytváří nové pole a prvky se musejí přesouvat (gif)
- rychlý přístup k prvkům dle indexu
- pomalé operace přidávání a odebírání prvků blíže k začátku seznamu (pole, v němž je seznam, se musí realokovat)

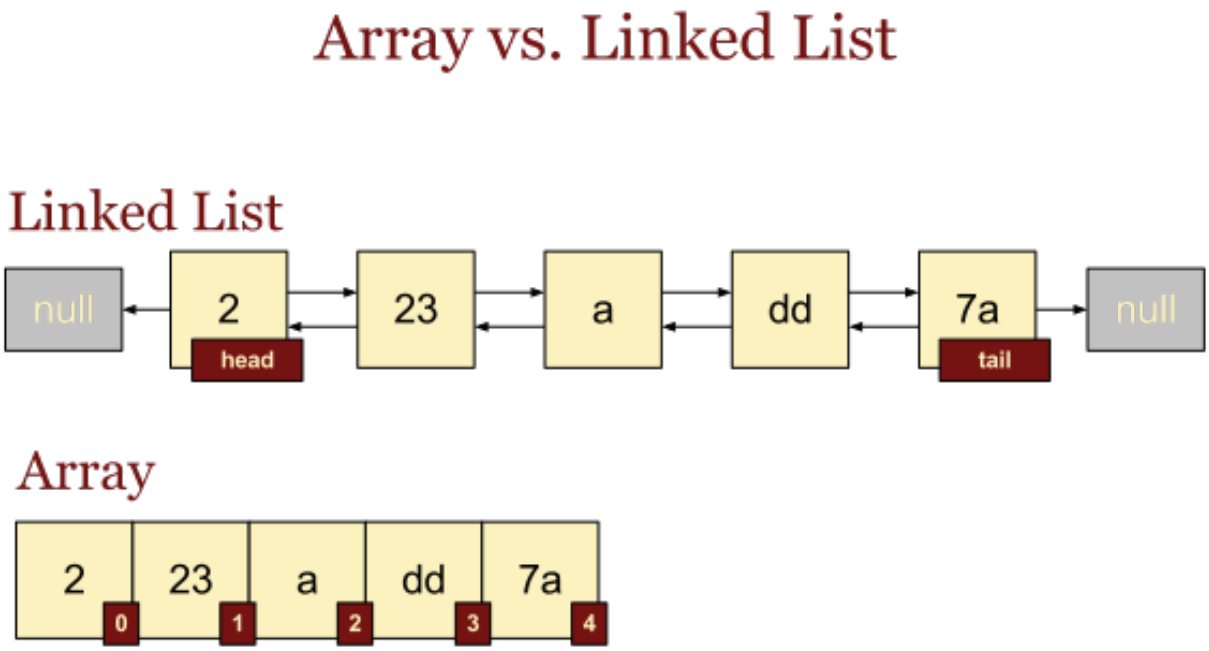
Javadoc třídy `ArrayList`

Implementace seznamu — `LinkedList`

- druhá nejpoužívanější implementace seznamu
- využívá **zřetězený seznam** pro uchování prvků
- pomalejší operace přístupu k prvkům dle indexu "uvnitř" seznamu
- rychlejší operace přidávání a odebírání prvků na začátku a na konci, resp. blízko nich

Javadoc třídy `LinkedList`

ArrayList vs. LinkedList



Kontejnery ukladají pouze jeden typ, v tomto případě `String`.

Výkonnostní porovnání seznamů

- Operace nad ArrayList vs LinkedList

```
add 100000 elements                        | 12 ms     | 8 ms
remove all elements from last to first     | 5 ms      | 9 ms
add 100000 elements at 0th position        | **1025 ms**   | 18 ms
remove all elements from 0th position      | **1014 ms**   | 10 ms
add 100000 elements at random position     | 483 ms    | **34504 ms**
remove all elements from random position   | 462 ms    | **36867 ms**
```

Konstruktory seznamů

- `ArrayList()`
  - vytvoří prázdný seznam (s kapacitou 10 prvků)
- `ArrayList(int initialCapacity)`
  - vytvoří prázdný seznam s danou kapacitou
- `ArrayList(Collection<? extends E> c)`
  - vytvoří seznam a naplní ho prvky kolekce `c`

Kapacita reprezentuje interní kapacitu, **neznamená** to počet `null` prvků v nové kolekci!

Vytváření kolekcí

Statické "factory" metody:

- `List.of(elem1, elem2, ⋯)`
  - vytvoří seznam a naplní ho danými prvky
  - vrátí **nemodifikovatelnou** kolekci
- analogicky `Set.of, Map.of`
- jestli chceme kolekci modifikovatelnou, musíme vytvořit novou:

```
List<String> modifiableList = new ArrayList<>(List.of("Y", "N"));
```

Na zamyšlení

- Jak udělám se seznamu typu `List` kolekci `Collection`?

```
// change the type, it is its superclass
Collection<Long> collection = list;
```

- Jak udělám z kolekce `Collection` seznam `List` ?

```
// create new list
List<Long> l = new ArrayList<>(collection);
```

Metody rozhraní `List` I

Rozhraní `List` dědí od `Collection`.

Kromě metod v `Collection` obsahuje další metody:

- `E get(int index)`
  - vrátí prvek na daném indexu
  - `IndexOutOfBoundsException` je-li mimo rozsah
- `E set(int index, E element)`
  - nahradí prvek s indexem `index` prvkem `element`
  - vrátí předešlý prvek

Metody rozhraní `List` II

- `void add(int index, E element)`
  - přidá prvek na daný index (prvky za ním posune)
- `E remove(int index)`
  - odstraní prvek na daném indexu (prvky za ním posune)
  - vrátí odstraněný prvek
- `int indexOf(Object o)`
  - vrátí index **prvního** výskytu `o`
  - jestli kolekce prvek neobsahuje, vrátí -1
- `int lastIndexOf(Object o)` pro index **posledního** výskytu

Příklad použití seznamu

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("C");
list.add(1, "B");
// ["A", "B", "C"]

list.get(2); // "C"
list.set(1, "D"); // "B"
list.indexOf("D"); // 1
```

Množina, `Set`

- odpovídá matematické představě množiny
- prvek lze do množiny vložit nejvýš *jedenkrát*
- při porovnávání rozhoduje rovnost **podle výsledku volání** `equals`
- umožňuje rychlé dotazování na přítomnost prvku
- provádí rychle atomické operace (se složitostí *O(1), O(log(n))*):
  - vkládání prvku — `add`
  - odebírání prvku — `remove`
  - dotaz na přítomnost prvku — `contains`

Množiny jsou primárně bez pořadí, bez uspořádání, existuje však i množina s uspořádáním.

Equals & hashCode — opakování

Equals

zjistí, jestli jsou objekty logicky stejné (porovnání atributů).

HashCode

vrací pro logicky stejné objekty stejné číslo, **haš**.

Hash

je *falešné ID* — pro různé objekty může `hashCode` vracet stejný haš.

Implementace množiny — `HashSet`

- Ukládá objekty do hašovací tabulky podle haše
- Ideálně konstantní operace (tj. sub-logaritmická složitost)
- Když má více prvků stejný haš, existuje více způsobů řešení
- Pro (ne úplně ideální) `hashCode` x + y vypadá tabulka následovně:

```
haš  | objekt
0    | [0,0]
1    | [1,0]
2    |
3    | [2,1]
```

Javadoc třídy `HashSet`

`HashSet` pod lupou

- `boolean contains(Object o)`
  - vypočte haš tázaného prvku `o`
  - v tabulce najde objekt uložený pod stejným hašem
  - objekt porovná s `o` pomocí `equals`
- Co když mají všechny objekty stejný haš?
  - Množinové operace budou velmi, velmi pomalé.
- Co když porušíme *kontrakt* metody `hashCode` (pro stejné objekty vrátí **různá** čísla)?
  - Množina přestane fungovat jako množina, bude obsahovat duplicity!

Další implmentací množiny je `LinkedHashSet` = Hash Table + Linked List

Další lineární struktury

zásobník

třída `Stack`, struktura LIFO

fronta

třída `Queue`, struktura FIFO

- fronta může být také *prioritní* — `PriorityQueue`

oboustranná fronta

třída `Deque` (čteme "deck")

- slučuje vlastnosti zásobníku a fronty
- nabízí operace příslušné oběma typům

Starší typy kontejnerů

- Existují tyto starší typy kontejnerů (za → uvádíme náhradu):
  - `Hashtable` → `HashMap`, `HashSet` (podle účelu)

- `Vector` → `List`
- `Stack` → `List` nebo lépe `Queue` či `Deque`

Kontejnery a primitivní typy

- Kontejnery ukládájí pouze odkazy na objekty, **neukládají primitivní typy**.
- Proto používáme jejich objektové protějšky — `Integer, Char, Boolean, Double`...
- Java automaticky dělá tzv. **autoboxing** — konverzi primitivního typu na objekt
- Pro zpětnou konverzi se analogicky dělá tzv. **unboxing**

```
List<Integer> list = new ArrayList<>();
list.add(new Integer(1));
list.add(1); // autoboxing
int primitiveType = list.get(0); // unboxing
```

Procházení kolekcí

Základní typy:

for-each cyklus

- jednoduché, intuitivní
- nepoužitelné pro modifikace samotné kolekce

iterátory

- náročnější, užitečnější
- modifikace povolena

lambda výrazy s `forEach`

- strašidelné

For-each cyklus I

- Je rozšířenou syntaxí cyklu `for`.
- Umožňuje procházení kolekcí i **polí**.

```
List<Integer> numbers = List.of(1, 1, 2, 3, 5);
for(Integer i: list) {
   System.out.println(i);
}
```

For-each cyklus II

- For-each neumožňuje modifikace kolekce.
- Jestli kolekci změníme, nemůžeme pokračovat v iterování — dojde k vyhození `ConcurrentModificationException`.
- Odstranění prvku a vyskočení z cyklu však funguje:

```
Set<String> set = Set.of("Donald Trump", "Barrack Obama");
for(String s: set) {
  if (s.equals("Donald Trump")) {
    set.remove(s);
    break;
  }
}
```

Iterátory

- Sekvenční procházení prvků kolekce v *neurčeném pořadí* nebo *uspořádání* (u uspořádaných kolekcí)
- Každý iterátor musí implementovat velmi jednoduché rozhraní `Iterator<E>`
- Běžné použití pomocí `while`:

```
Set<Integer> set = Set.of(1, 2, 3);
Iterator<Integer> iterator = set.iterator();
while(iterator.hasNext()) {
  Integer element = iterator.next();
  ...
}
```

Metody iterátorů

- `E next()`
  - vrátí následující prvek
  - `NoSuchElementException` jestli iterace nemá žádné zbývající prvky
- `boolean hasNext()`
  - `true` jestli iterace obsahuje nějaký prvek
- `void remove()`
  - odstraní prvek z kolekce
  - maximálně jednou mezi jednotlivými voláními `next()`

Iterátor — příklad

Pro procházení iterátoru se dá použít i `for` cyklus:

```
Set<String> set = Set.of("Donald Trump", "Barrack Obama", "Hillary Clinton");

for (Iterator<String> iter = set.iterator(); iter.hasNext();) {
  String element = iter.next();
  if (!element.equals("Barrack Obama")) iter.remove();
}
```

Roli iterátoru plnil dříve *výčet* (`Enumeration`) — nepoužívat. = Porovnání kontejnerů, třída `Collections` =

Pomocná třída `Collections`

- Java Core API v balíku `java.util` nabízí třídu `Collections`
- nabízí jen statické metody a proměnné (tzv. utility class), nelze od ní vytvářet instance
- nabízí škálu užitečných metod pro práci s kontejnery

Javadoc třídy `Collections`

Souběžný přístup

- moderní kontejnery **nejsou synchronizované**
- jinak řečeno, nepřipouštějí souběžný přístup z více vláken
- standardní (nesynchronizovaný) kontejner lze však **zabalit**
- `synchronizedSet, synchronizedList, synchronizedCollection, ⋯`
- metoda vrátí novou synchonizovanou kolekci

```
List<String> list = new ArrayList<>();
List<String> syncedList = Collections.synchronizedList(list);
```

Získání nemodifikovatelných kontejnerů

- kontejnery jsou standardně modifikovatelné (read/write)
- nemodifikovatelné kontejnery se používají při vracení hodnot z metod
- `unmodifiableSet, unmodifiableList, unmodifiableCollection, ⋯`
- metoda vrátí novou nemodifikovatelnou kolekci

```
Set<String> set = Set.of("Donald Trump", "Barrack Obama", "Hillary Clinton");
return Collections.unmodifiableSet(set);
```

Getter na kolekci je vždy nemodifikovatelný!

Prázdné kontejnery

- třída obsahuje konstanty `EMPTY_SET, EMPTY_LIST, EMPTY_MAP`
- metody `emptyList(), emptyMap(), emptyIterator(), ⋯`
- preferujeme metody, protože konstanty postrádají typ, tj. chybí jim typová kontrola
- vrácené kolekce jsou **nemodifikovatelné**
- šetříme vytváření nové kolekce

```
Collections.<String>emptyList();
```

Metody v `Collections` I

- `Collections.binarySearch`
  - binární vyhledávání v kontejneru
- `Collections.reverseOrder, rotate`
  - obrácení, rotace pořadí prvků
- `Collections.swap`
  - prohazování prvků
- `Collections.shuffle`
  - náhodné zamíchání prvků

Metody v `Collections` II

- `Collections.sort`
  - uspořádání (přirozené, anebo pomocí komparátoru)
- `Collections.min, max`
  - minimální, maximální prvek (s definovaným uspořádaním)
- `Collections.nCopies`
  - vytvoří kolekci n stejných prvků
- `Collections.frequency`
  - kardinalita dotazovaného prvku v dané kolekci

Srovnání implementací kolekcí

- `ArrayList` — na bázi pole
  - rychlý přímý přístup (přes index)
- `LinkedList` — na bázi lineárního zřetězeného seznamu
  - rychlý sekvenční přístup (přes iterátor)
- `HashMap`, `HashSet` — na bázi hašovacích tabulek
  - rychlejší, ale neuspořádané
  - lze získat iterátor procházející klíče uspořádaně
- `TreeMap`, `TreeSet` — na bázi vyhledávacích stromů
  - pomalejší, ale uspořádané
- `LinkedHashSet`, `LinkedHashMap` — spojení výhod obou

Kontejnery a jejich rozhraní

- `Set` (množina)
  - `HashSet` (založena na hašovací tabulce)
  - `TreeSet` (černobílý strom)
  - `LinkedHashSet` (zřetězené záznamy v hašovací tabulce)
- `List` (seznam)
  - `ArrayList` (implementován pomocí pole)
  - `LinkedList` (implementován pomocí zřetězeného seznamu)
- `Deque` (fronta - obousměrná)
  - `ArrayDeque` (fronta pomocí pole)
  - `LinkedList` (fronta pomocí zřetězeného seznamu)
- `Map` (asociativní pole/mapa)
  - `HashMap` (založena na hašovací tabulce)
  - `TreeMap` (černobílý strom)
  - `LinkedHashMap` (zřetězené záznamy v hašovací tabulce)

Kontejnery a výjimky

Při práci s kontejnery může vzniknout řada *výjimek*.

Některé z nich i s příklady:

- `IllegalStateException`
  - vícenásobné volání `remove()` bez volání `next()` v iterátoru

- `UnsupportedOperationException`
  - modifikace **nemodifikovatelné** kolekce
- `ConcurrentModificationException`
  - iterovaný prvek (for-each cyklem) byl odstraněn

Většina výjimek je *běhových* (runtime), tudíž není nutné je řešit. (Samozřejmě je ale třeba psát kód tak, aby nevznikaly. :))

Nepovinné metody

- funkcionalita kontejnerů je předepsána *rozhraním*
- některé metody rozhraní jsou *nepovinné* — třídy jej nemusí implementovat
  - např. `add`, `clear`, `remove`
  - metoda existuje, ale nelze ji použít, protože volání *vyhodí výjimku* `UnsupportedOperationException`
- Důvod?
  - např. nehodí se implementovat zápisové operace, když kontejnery budou read-only (`unmodifiable`) = Uspořádané kolekce =

Úvod

Motivace

- chceme prvky v kolekci uspořádané, ale nechceme to dělat "ručně"
- v kolekci typu `String` chceme jména od K po M

Implementace

- uspořádání dáné třídy musí být definováno ve třídě

Rozhraní `Comparable<T>`

- rozhraní slouží k definování **přirozeného** (defaultního) **uspořádání** třídy
- třída implementuje rozhraní ⇒ objekty jsou vzájemně *uspořádatelné*
- použití zejména u uspořádaných kontejnerů
- předepisuje jedinou metodu `int compareTo(T o)`
- `T` = typ objektu, název třídy

Javadoc třídy `Comparable<T>`

Metoda `compareTo`

```
int compareTo(T that)
// used as e1.compareTo(e2)
```

- metoda porovná 2 objekty — `this` (e1) a `that` (e2)
- vrací celé číslo, pro které platí:
  - číslo je záporné, když e1 < e2
  - číslo je kladné, když e1 > e2
  - 0, když nezáleží na pořadí
- na samotném čísle nezáleží, je v pořádku používat pouze hodnoty -1, 0, 1

Implementace `Comparable<E>`

```
public class Point implements Comparable<Point> {
  private int x;
  // ascending order
  public int compareTo(Point that) {
    return this.x - that.x;
  }
}
...
new Point(1).compareTo(new Point(4)); // -3
```

Existuje i beztypové rozhraní `Comparable`, to ale nebudeme používat!

`compareTo` vs. `equals`

- chování `compareTo` by mělo být konzistentní s `equals`
- pro rovné objekty by `compareTo` mělo vrátit `0`
- není to však nutnost
  - např. třída **`BigDecimal`** pro přesné hodnoty podmínku porušuje
  - pro stejné hodnoty s rozdílnou přesností — např. 4.0 a 4.00
- `compareTo` na rozdíl od `equals` nemusí vstupní objekt přetypovávat a může vyhazovat výjimku

Více uspořádání

Co kdybychom chtěli více typů uspořádání, nebo alternativu k přirozenému uspořádání?

Nemůžeme nadefinovat stejnou metodu víckrát.

- rozhraní Comparator<T> slouží k definování uspořádání zvnějšku — pomocí objektu jiné třídy
- předepisuje jedinou metodu `int compare(T o1, T o2)`

- uspořádání funguje nad objekty typu `T`
- návratová hodnota `compare` funguje stejně jako u `compareTo`
- funguje jako alternativa pro další uspořádání

Příklad komparátoru

Třída `String` má definované přirozené uspořádání lexikograficky.

Definujme lexikografický komparátor, který ignoruje velikost písmen:

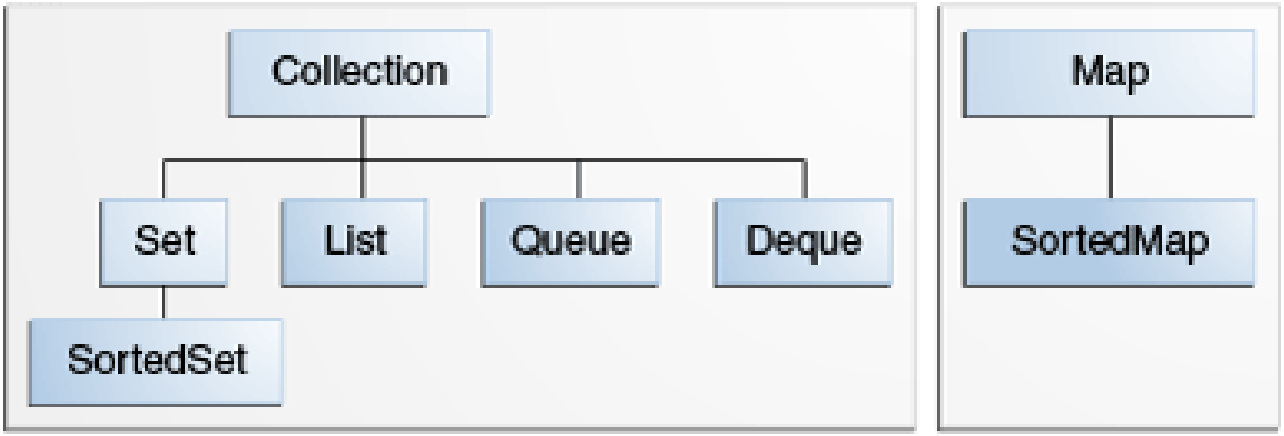
```
public class IgnoreCaseComparator implements Comparator<String> {
  public int compare(String o1, String o2) {
    return o1.toLowerCase().compareTo(o2.toLowerCase());
  }
}
...
new IgnoreCaseComparator().compare("HI", "hi"); // 0
```

Skutečné použití

- metody pro uspořádání programátor v kódu obvykle nepoužívá
- namísto toho používá **uspořádané kolekce**
- prvky v kolekcích jsou řazeny automaticky
- je nutno definovat přirozené uspořádání nebo použít komparátor, aby kolekce vědela, podle jakých pravidel prvky seřadit

Hierarchie rozhraní kolekcí

Budeme se zabývat rozhraními `SortedSet` a `SortedMap`.



`SortedSet, SortedMap`

`SortedSet`

- rozhraní pro uspořádané množiny
- všechny vkládané prvky musí implementovat rozhraní `Comparable` (nebo použít komparátor)
- implementace `TreeSet`

`SortedMap`

- rozhraní pro uspořádané mapy
- všechny vkládané **klíče** musí implementovat rozhraní `Comparable` (nebo použít komparátor)
- implementace `TreeMap`

Konstruktory `TreeSet`

- `TreeSet()`
  - vytvoří prázdnou množinu
  - prvky jsou uspořádány podle **přirozeného uspořádání**
- `TreeSet(Collection<? extends E> c)`
  - vytvoří množinu s prvky kolekce `c`
  - prvky jsou uspořádány podle **přirozeného uspořádání**
- `TreeSet(Comparator<? super E> comparator)`
  - vytvoří prázdnou množinu
  - prvky jsou uspořádány podle **komparátoru**
- `TreeSet(SortedSet<E> s)`
  - vytvoří množinu s prvky i uspořádáním podle `s`

Příklad `TreeSet` I

Definice přirozeného uspořádání:

```
public class Point implements Comparable<Point> {
  ...
  public int compareTo(Point that) {
    return this.x - that.x;
  }
}
```

Příklad `TreeSet` II

Použití:

```
SortedSet<Point> set = new TreeSet<>();
set.add(new Point(3));
set.add(new Point(3));
set.add(new Point(-1));
set.add(new Point(0));
System.out.println(set);
// prints -1, 0, 3
```

Jiný příklad `TreeSet`

Třída `String` má definované přirozené uspořádání lexikograficky.

```
SortedSet<String> set = new TreeSet<>();
set.add("Bobik");
set.add("ALIK");
set.add("Alik");
System.out.println(set); // [ALIK, Alik, Bobik]

SortedSet<String> set2 = new TreeSet<>(new IgnoreCaseComparator());
set2.addAll(set);
System.out.println(set2); // [ALIK, Bobik]
```

`TreeSet` pro porovnávání prvků používá `compareTo` / `compare`, proto má druhá množina pouze 2 prvky!

`TreeSet` pod lupou

- implementována jako *červeno-černý vyvážený vyhledávací strom*
  - ⇒ operace `add, remove, contains` jsou v *O(log n)*
- hodnoty jsou uspořádané
  - prvky jsou procházeny v přesně definovaném pořadí

Javadoc třídy `TreeSet`

`TreeMap`

- množina klíčů je de facto `TreeSet`
- hodnoty nejsou uspořádány
- uspořádání lze ovlivnit stejně jako u uspořádané množiny
- implementace stromu a složitost operací je stejná

Javadoc třídy `TreeMap`

Příklad `TreeMap`

Klíče jsou unikátní a uspořádané, hodnoty nikoliv.

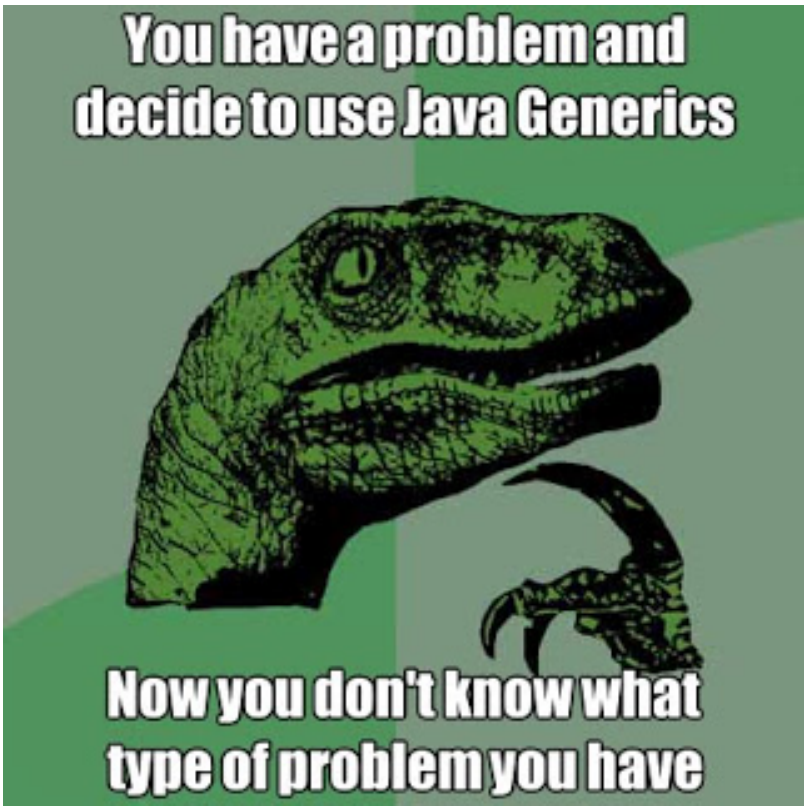
```
SortedMap<String, Integer> population = new TreeMap<>();
population.put("Brno", -1);
population.put("Brno", 500_000);
population.put("Bratislava", 500_000);

System.out.println(population);
// {Bratislava=500000, Brno=500000}
```

Programování v jazyce Java

Generické typy, typové parametry

- Generické typy = něco *obecně použitelného* , *zobecnění*
- Třídy v Javě mají společného předka, třídu `Object` (každý objekt je instancí třídy `Object`).
- Potřebujeme-li pracovat s nějakými objekty, o kterých neznáme typ, můžeme využít společného předka a pracovat s ním.
- To umožňuje snadnou implementaci kolekcí, ale například i využití reflexe.

Vtip



Deklarace seznamu bez a s generiky

```
// no generics (obsolete)
public interface List { ... }
// generic type E
public interface List<E> { ... }
```

- do špičatých závorek umístíme symbol — seznam bude obsahovat prvky `E` (předem neznámého) typu
- je doporučováno používat velké, jednopísmenné deklarace
- písmeno vystihuje použití — `T` je **t**ype, `E` je **e**lement
- `E` nahradíme jakoukoliv třídou nebo rozhraním

Jednoduché využití v metodách

```
E get1(int index);
Object get2(int index);
```

- `get1` vrací pouze objekty, které jsou typu `E` — je vyžadován speciální typ
- `get2` vrací libovolný objekt, tj. musíme pak přetypovávat

```
boolean add(E o);
```

- přidává do seznamu prvky typu `E`

Výhody generik

```
List numbers1 = new ArrayList();
numbers1.add(1);
numbers1.add(new Object()); // allowed, unwanted
Integer n = (Integer) numbers1.get(0);

List<Integer> numbers2 = new ArrayList<>();
numbers2.add(1);
numbers2.add(new Object()); // won't compile!
n = numbers2.get(0);
```

- do seznamu `numbers1` lze vložit **libovolný objekt**
- při získávání objektů se spoléháme na to, že se jedná o číslo
- do `numbers2` nelze obecný objekt vložit, je nutné vložit číslo

Motivace

- Chceme seznam různých typů seznamů, tak jej vytvoříme následovně:

```
List<List<Object>> listOfDifferentLists;
```

- Máme problém — seznam čísel není seznamem objektů:

```
List<Number> numbers = new ArrayList<Number>();
List<Object> general = numbers; // won't compile!
List<? super Number> general2 = numbers; // solution
```

Do seznamu, který obsahuje nejvýše čísla lze vkládat pouze objekty, které jsou **alespoň čísly**.

Žolíci (wildcards) I

Generika poskytují nástroj zvaný *žolík* (wildcard) , který se zapisuje jako `<?>`.

```
List<Number> numbers = new ArrayList<Number>();
List<?> general = numbers; // OK
general.add("Not a number"); // won't compile!
```

- `List<?>` říká, že jde o seznam **neznámých** prvků.
- Jelikož nevíme, jaké prvky v seznamu jsou, **nemůžeme do něj ani *žádné* prvky přidávat**.
- Jedinou výjimkou je žádný prvek `null`, který lze přidat kamkoliv.

Abstraktní třída `Number` reprezentuje numerické primitivní typy (`int`, `long`, `double`, ⋯)

Žolíci (wildcards) II

- Ze seznamu neznámých objektů můžeme prvky číst.
- Každý prvek je alespoň instancí třídy `Object`:

```
public static void printList(List<?> list) {
  for (Object e : list) {
    System.out.println(e);
  }
}
```

Žolíci a polymorfismus I

Nasledující metoda dělá sumu ze seznamu čísel:

```
public static double sum(List<Number> numbers) {
  double result = 0;
  for (Number e : numbers) {
    result += e.doubleValue();
  }
  return result;
}
...
List<Number> numbers = List.of(1,2,3);
sum(numbers); // it works
List<Integer> integers = List.of(1,2,3);
sum(integers); // won't compile!
```

Žolíci a polymorfismus II

- `Integer` je `Number` a přesto seznam `List<Integer>` nelze použít!
- Nechceme `List<Number>`, řešením je **seznam neznámých prvků, které jsou nejvýše čísly**.

```
public static double sum(List<? extends Number> numbers) { ... }
```

- Toto použití žolíku má uplatnění i v rozhraní `List<E>`, např. v metodě `addAll`:

```
boolean addAll(Collection<? extends E> c);
```

- Uvědomte si následující — žolík je zkratka pro neznámý prvek rozšiřující `Object`.

Žolíci a dědičnost

Další použití žolíků:

- Parametrem metody je instance třídy, která je **v hierarchii mezi třídou specifikovanou naším obecným prvkem `E` a třídou `Object`**.
- Například chceme setřídit množinu celých čísel.
- Existuje třídění podle:
  - hodnoty metody `hashCode()` — na úrovni třídy `Object`
  - čísla — na úrovni třídy `Number`
  - celého čísla — na úrovni třídy `Integer`
- Konstruktor stromové setříděné mapy:

```
public TreeSet(Comparator<? super E> c);
```

Žolíci a více typů

- Deklarace obecného rozhraní setříděné mapy:

```
public interface SortedMap<K,V> extends Map<K,V> { ... }
```

- Je-li třeba použít více nezávislých obecných typů, zapíšeme je do zobáčků jako seznam hodnot oddělených čárkou.
- `K` je **key**, `V` je **value**.
- Je možné použít i žolíků, viz následující příklad konstruktorů stromové mapy:

```
public TreeMap(Map<? extends K, ? extends V> m);
public TreeMap(SortedMap<K, ? extends V> m);
```

Generické metody

- Pro používání generik a žolíků v metodách platí stále stejná pravidla.
- Generická metoda = metoda **parametrizována alespoň jedním obecným typem**.
- Obecný typ nějakým způsobem váže typy proměnných a/nebo návratové hodnoty metody.
- Příklad statické metody, která přenese prvky z pole do seznamu (pole i seznam musí mít stejný typ):

```
static <T> void arrayToList(T[] array, List<T> list) {
  for (T o : array)  list.add(o);
}
```

- Ve skutečnosti nemusí být seznam `list` **téhož typu** — stačí, aby jeho **typ byl nadtřídou** typu pole `array`.
- Např. `Integer[] array` a `List<Number> list`
  - prvky z pole do seznamu se dají kopírovat (i když typy nejsou stejné!), protože `Integer` je `Number`

Generics metody vs. wildcards

- Chceme, aby typ u generické metody spojoval parametry nebo parametr a návratovou hodnotu.
- Ne úplně správné (funkční) použití generické metody:

```
static <T, S extends T> void copy(List<T> destination, List<S> source);
```

- Lepší zápis, `T` spojuje dva parametry metody a přebytečné `S` je nahrazené žolíkem:

```
static <T> void copy(List<T> destination, List<? extends T> source);
```

Metody jsou `public`, viditelnost je vynechána kvůli lepší přehlednosti.

Pole

- Pro pole nelze použít parametrizovanou třídu.
- Při vkládání prvků do pole runtime systém kontroluje pouze *typ vkládaného prvku*.
- Do pole řetězců bychom pak mohli vložit pole čísel a pod.

```
// generic array creation error
public <T> T[] returnArray() {
  return new T[10];
}
```

- Jde však použít třídu s žolíkem, který není vázaný:

```
List<?>[] pole = new List<?>[10];
```

Vícenásobná vazba generik I

- Uvažujme následující metodu, která vyhledává maximální prvek kolekce.

```
static Object max(Collection<T> c);
```

- Prvky kolekce musí implementovat rozhraní `Comparable`, což není syntaxí vůbec podchyceno.
  - Zavolání této metody proto může vyvolat výjimku `ClassCastException`!
- Chceme, aby prvky kolekce implementovali rozhraní Comparable.

```
static <T extends Comparable<? super T>> T max(Collection<T> c);
// if generics are removed
static Comparable max(Collection c); // does not return Object!
```

Vícenásobná vazba generik II

- Signatura metody se změnila — má vracet `Object`, ale vrací `Comparable`!
  - Metoda musí vracet `Object` kvůli zpětné kompatibilitě.
- Využijeme tedy **vícenásobnou vazbu**:

```
static <T extends Object & Comparable<? super T>> T max (Collection<T> c);
```

- Po výmazu má metoda správnou signaturu, protože v úvahu se bere první zmíněná třída.
- Obecně lze použít více vazeb, například když je obecný prvek implementací více rozhraní.

Závěr

- Generiky mají i další využití, například u reflexe.
- Tohle však již překračuje rámec začátečnického seznamování s Javou.
- Slidy vychází z materiálů
  - Javy firmy Sun
  - *Generics in the Java Programming Language* od Gilada Brachy = Streamy a lambda výrazy =

Lambda výrazy

Anonymní metody, umožňují elegantní zápis pomocí symbolu šipky.

- `i -> i + 1`
  - inkrementuj číselní hodnotu
- `(Person p) -> (p.getAge() >= 18)`
  - vrať `true`, je-li osoba dospělá
- levá část obsahuje vstupní parametry
- pravá část použití a návratovou hodnotu
- dříve realizováno pomocí rozhraní s jednou metodou

Kolekce typu `Stream`

Proud (`java.util.stream.Stream`) je homogenní lineární struktura prvků (např. seznam).

- hodí se pro **snadné řetězení více operací**
- existují 2 typy operací:
  - *intermediate operations* ("přechodné") — transformuje proud do dalšího proudu
  - *terminal operation* ("terminální", "koncová") — vyprodukuje výsledek nebo má vedlejší účinek

- jedná se o "lazy collections" — prvky se vyhodnotí, až když se zavolá terminální operace a použije se její výsledek

Neplést jsi se vstupně/výstupními proudy (`java.io.InputStream`/`OutputStream`)

Příklad

```
List<String> list = List.of("MUNI", "VUT", "X");
int count = list.stream()
                .filter(s -> s.length() > 1)
                .mapToInt(s -> s.length())
                .sum();
// count is 7
```

- kolekci tranformujeme na proud
- použijeme pouze řetězce větší než 1
- transformujeme řetězec na přirozené číslo (délka řetězce)
- zavoláme terminální operaci, která čísla sečte

Vytvoření `Stream`

`Stream` obvykle vytváříme z prvků kolekce, pole, nebo vyjmenováním.

```
Stream<String> stringStream;

List<String> names = new ArrayList<>();
stringStream = names.stream();

String[] names = new String[] { "A", "B" };
stringStream = Stream.of(names);

stringStream = Stream.of("C", "D", "E");
```

Odkazy na metody

Lambda výraz, který pouze volá metodu, se dá zkrátit **vytvořením odkazu na** (danou) **metodu**.

- Zjištění délky řetězce `s → s.length()`:

```
String::length
```

- Vypsání prvků `s → System.out.println(s)`:

```
System.out::println
```

Příklad použití `Stream`

```
List<String> names = ...
names.stream()
     .map(String::toUpperCase)
     .forEach(System.out::println);
```

1. nejdříve vytvoří ze seznamu proud
2. pak každý řetězec převede pomocí `toUpperCase` (průběžná operace)
3. na závěr každý takto převedený řetězec vypíše (terminální operace)

Odpovídá sekvenční iteraci:

```
for(String name : names) {
   System.out.println(name.toUpperCase());
}
```

Přechodné metody třídy `Stream` I

Přechodné metody vrací proud, na který aplikují omezení:

- `Stream<T> distinct()`
  - bez duplicit (podle `equals`)
- `Stream<T> limit(long maxSize)`
  - proud obsahuje maximálně `maxSize` prvků
- `Stream<T> skip(long n)`
  - zahodí prvních `n` prvků
- `Stream<T> sorted()`
  - uspořádá podle přirozeného uspořádání

Přechodné metody třídy `Stream` II

- `Stream<T> filter(Predicate<? super T> predicate)`
  - `Predicate` = lambda výraz s jedním parametrem, vrací `booolean`
  - zahodí prvky, které nesplňují `predicate`
- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`
  - `mapper` je funkce, která bere prvky typu `T` a vrací prvky typu `R`

    - může vracet stejný typ, např. `map(String::toUpperCase)`
- existují mapovací funkce `mapToInt, mapToLong, mapToDouble`
    - vracejí speciální `IntStream, ..`
    - obsahuje další funkce — např. `average()`

Terminální metody třídy `Stream` I

Terminální anebo ukončovací metody.

- `long count()`
    - vrací počet prvků proudu
- `boolean allMatch(Predicate<? super T> predicate)`
    - vrací `true`, když všechny prvky splňují daný predikát
- `boolean anyMatch(Predicate<? super T> predicate)`
    - vrací `true`, když alespoň jeden prvek splňuje daný predikát
- `void forEach(Consumer<? super T> action)`
    - aplikuje `action` na každý prvek proudu
    - např. `forEach(System.out::println)`

Terminální metody třídy `Stream` II

- `<A> A[] toArray(IntFunction<A[]> generator)`
    - vytvoří pole daného typu a naplní jej prvky z proudu
    - `String[] stringArray = streamString.toArray(String[]::new);`
- `<R,A> R collect(Collector<? super T,A,R> collector)`
    - vytvoří kolekci daného typu a naplní jej prvky z proudu
    - `Collectors` je třída obsahující pouze statické metody
    - použití:

```
stream.collect(Collectors.toList());
stream.collect(Collectors.toCollection(TreeSet::new));
```

Příklad

```
int[] numbers = IntStream.range(0, 10) // 0 to 9
  .skip(2) // omit first 2 elements
  .limit(5) // take only first 5
  .map(x -> 2 * x) // double the values
  .toArray(); // make an array [4, 6, 8, 10, 12]

List<String> newList = list.stream()
  .distinct() // unique values
  .sorted() // ascending order
  .collect(Collectors.toList());

Set<String> set = Stream.of("an", "a", "the")
  .filter(s -> s.startsWith("a"))
  .collect(Collectors.toCollection(TreeSet::new));
  // [a, an]
```

Možný výsledek `Optional`

- `Optional<T> findFirst()`
  - vrátí první prvek
- `Optional<T> findAny()`
  - vrátí nějaký prvek
- `Optional<T> max/min(Comparator<? super T> comparator)`
  - vrátí maximální/minimální prvek

V jazyce Haskell má `Optional` název `Maybe`.

Použití `Optional`

`Optional<T>` má metody:

- `boolean isPresent()` — `true`, jestli obsahuje hodnotu
- `T get()` — vrátí hodnotu, jestli neexistuje, vyhodí výjimku
- `T orElse(T other)` — jestli hodnota neexistuje, vrátí `other`

```
int result = List.of(1, 2, 3)
  .stream()
  .filter(num -> num > 4)
  .findAny()
  .orElse(0);
// result is 0
```

Paralelní a sekvenční proud

```
List<Integer> integerList = List.of(1, 2, 3, 4, 5);
integerList.parallelStream()
           .forEach(i -> System.out.print(i + " "));
// 3 5 4 2 1
```

```
List<Integer> integerList = List.of(1, 2, 3, 4, 5);
integerList.stream()
           .forEach(i -> System.out.print(i + " "));
// 1 2 3 4 5
```

Konverze na proud — příklad I

Jak konvertovat následující kód na proud?

```
Set<Person> owners = new HashSet<>();
for (Car c : cars) {
  owners.add(c.getOwner());
}
```

Hint `x -> x.getOwner()` se dá zkrátit na `Car::getOwner`.

```
Set<Person> owners = cars.stream()
          .map(Car::getOwner)
          .collect(Collectors.toSet());
```

Konverze na proud — příklad II

Jak konvertovat následující kód na proud?

```
Set<Person> owners = new HashSet<>();
for (Car c : cars) {
  if (c.hasExpiredTicket()) owners.add(c.getOwner());
}
```

```
Set<Person> owners =
cars.stream()
    .filter(c -> c.hasExpiredTicket())
    .map(Car::getOwner)
    .collect(Collectors.toSet());
```

Funkcionální rozhraní

- Podpora v knihovnách (Java Core API)
- funkcionální rozhraní v balíku `java.util.functions`
- většinou jako generická (typově parametrizovaná) rozhraní, např.:

Predicate<T>
: s jednou metodou `boolean test(T t)`

Supplier<T>
: s jednou metodou `void get(T t)`

Consumer<T>
: s jednou metodou `void accept(T t)`

Dokumentace

- Benjamin Winterberg: Java 8 Stream Tutorial
- Amit Phaltankar: Understanding Java 8 Streams API
- Oracle Java Documentation: Lambda Expressions = Mapy = = Mapy =

`Map`

Asociativní pole, mapa, slovník

- ukládá **dvojici** *klíč — hodnota*
- umožnuje rychlé vyhledání hodnoty podle klíče
- klíče v mapě jsou vždy unikátní
- mapa je **kontejner** — dynamická datová struktura
- mapa rozhodně **není** `Collection<E>`
- implementuje rozhraní `Map<K,V>`
  - **K** = objektový typ klíče, **V** = objektový typ hodnoty
  - např. mapa ID a osob — `Map<Long, Person>`

Příklad `Map`

Následující mapa ukládá značky aut (klíče) a počty kusů (hodnoty):

```
Map<String, Integer> vehicles = new HashMap<>();
vehicles.put("BMW", 2);
vehicles.put("Audi", 4);
vehicles.put("Opel", 1);

vehicles.get("BMW"); // 2
```

Metody `Map` I

- `int size()` — velikost mapy
- `void clear()` — vyprázdní mapu
- `boolean isEmpty()` — `true`, když je mapa prázdná
- **`boolean containsKey(Object key)` — dotaz na přítomnost klíče**
- `boolean containsValue(Object value)` — dotaz na přítomnost hodnoty
- `V remove(Object key)` — odstraní klíč, vrací hodnotu (nebo `null`)
- `V replace(K key, V value)` — nahradí existující klíč hodnotou

K = objektový typ klíče (**k**ey)

V = objektový typ hodnoty (**v**alue)

Metody `Map` II

- `V put(K key, V value)`
  - vloží dvojici *klíč* — *hodnota* do mapy
  - jestli daný klíč už existuje, hodnota je **přepsána**
  - vrací přepsanou hodnotu nebo `null`
- `V putIfAbsent(K key, V value)`
  - vloží dvojici pouze v případě, že klíč zatím v mapě neexistuje
- `V get(Object key)`
  - výběr hodnoty odpovídající zadanému klíči
  - jestli klíč neexistuje, vrací `null`
- `V getOrDefault(Object key, V defaultValue)`
  - vrací hodnotu daného klíče nebo defaultní hodnotu

Metody `Map` III

- `Set<K> keySet()`
  - vrací **množinu** všech klíčů

    - Proč množina? Každý klíč je v mapě maximálně jednou
- `Collection<V> values()`
    - vrací **kolekci** všech hodnot (může obsahovat duplicity)
- `Set<Map.Entry<K,V>> entrySet()`
    - vrací množinu typu `Map.Entry` pro iteraci kolekce
    - obsahuje metody `getKey(), getValue()`

Pro vkládání mapy do mapy existuje `putAll`.

Příklad iterace mapy

```
Map<Integer, String> map = Map.ofEntries(
  entry(1, "a"),
  entry(2, "b")
  );

for (Map.Entry<Integer, String> entry : map.entrySet()) {
  System.out.println("key: " + entry.getKey());
  System.out.println("value: " + entry.getValue());
}
```

Implementace mapy — `HashMap`

- `HashMap` je implementována pomocí hašovací tabulky
- haš je zahašovaný klíč, hodnota tabulky je dvojice *(klíč, hodnota)*
- Složitost základních operací
    - v praxi závisí na kvalitě hašovací funkce (metody `hashCode`) na ukládaných objektech
    - teoreticky se blíží složitosti *konstantní*

Kolekce `HashSet` je implementována pomocí `HashMap` — klíč je prvek, hodnota je "dummy object".

Javadoc třídy `HashMap` = Výjimky =

Vtip

Výjimky

- **Výjimky** jsou mechanizmem umožňujícím reagovat na nestandardní (tj. chybové) běhové chování programu, které může mít různé příčiny:
  - chyba okolí: uživatele, systému
  - vnitřní chyba programu: tedy programátora.
- Proč výjimky?
  - Mechanizmus, jak psát robustní, spolehlivé programy odolné proti chybám "okolí" i chybám v samotném programu.

Výjimky v Javě fungují podobně jako v dalších objektových jazycích (C++, Python).

Vytvoření výjimky

- Výjimka, `Exception` je objektem třídy (nebo podtřídy) `java.lang.Exception`.
- Existují 2 základní konstruktory:

Constructor	Description
NullPointerException()	Constructs a NullPointerException with no detail message.
NullPointerException(String s)	Constructs a NullPointerException with the specified detail message.

Preferujeme konstruktor se zprávou.

Vyhození výjimky

Objekt výjimky je vyhozen:

1. automaticky běhovým systémem Javy, nastane-li nějaká běhová chyba — např. dělení nulou
2. samotným programem použitím klíčového slova `throw`, zdetekuje-li nějaký chybový stav, na nějž je třeba reagovat
   - např. do metody je předán špatný argument

```
if (x <= 0) {
  throw new IllegalArgumentException("x was expected to be positive");
}
```

Co se stane s vyhozenou výjimkou?

Vyhozený objekt výjimky je buďto:

1. **Zachycen** v rámci metody, kde výjimka vznikla (v bloku `catch`).
2. Výjimka **propadne** do nadřazené (volající) metody, kde je buďto v bloku `catch` zachycena nebo opět propadne atd.
   - Výjimka tedy "putuje programem" tak dlouho, než je zachycena.
   - Pokud není nikde zachycena, program skončí s hlášením o výjimce.

Jak reagovat na výjimku

- Jestli výjimku nezachytíme, způsobí pád programu.
- Proto existuje mechanismus `try-catch` bloku, který umožňuje reagovat na vyhození výjimky.

`try`
: blok vymezující místo, kde může výjimka vzniknout

`catch`
: blok, který se vykoná, nastane-li výjimka odpovídajícího typu

Try & catch

Příklad zachycení výjimky I

- Blok `catch` výjimku zachytí.
- Vyhození výjimky programátorem, výjimka je zachycena v `catch` bloku:

```
try {
  throw new NullPointerException();
} catch(NullPointerException e) {
   System.out.println("NPE was thrown and I caught it");
}
```

Příklad zachycení výjimky II

- Vyhození výjimky chybou programu (a její zachytění):

```
int[] array = new int[] { 16, 25 };
try {
   int x = array[2];
} catch(ArrayIndexOutOfBoundsException e) {
   System.err.println("Only index 0 or 1 can be used");
}
```

Try a catch pod lupou

- Jak funguje `try` blok?
  - vykonává se příkaz za příkazen
  - jestli dojde k vyhození výjimky, **další kód v `try` se přeskočí** a kontroluje se `catch` blok
  - jestli **nedojde** k vyhození výjimky, kód se vykoná a bloky s `catch` se ignorují
- Jak funguje `catch` blok?
  - syntax je `catch(ExceptionType variableName) { ⋯ }`
  - jestli se typ výjimky **zhoduje anebo je nadtřídou**, vykoná se kód `catch` bloku
  - jestli se typ výjimky **nezhoduje**, výjimka není zachycena
  - `catch` bloků může být víc, pak se prochází postupně
  - vždy se vykoná maximálně jeden `catch` blok

Příklad více `catch` bloky

```
try {
   String s = null;
   s.toString(); // NPE exception is thrown
   s = "This will be skipped";
} catch(IllegalArgumentException iae) {
   System.err.println("This will not be called");
} catch(NullPointerException npe) {
   System.err.println("THIS WILL BE CALLED");
} catch(ArrayIndexOutOfBoundsException e) {
   System.err.println("This entire block will be skipped");
}
```

Proměnná v `catch`

- Catch blok kromě typu výjimky obsahuje i proměnnou, která se dá použít v rámci bloku:

```
try {
   new Long("xyz");
} catch (NumberFormatException e) {
   System.err.println(e.getMessage());
}
```

Sloučení `catch` bloků

```
try {
   Person p = new Person(null);
} catch(NullPointerException e) {
   System.err.println("Invalid name.");
} catch(IllegalArgumentException e) {
   System.err.println("Invalid name.");
}
```

Operátor `|` sloučí stejné `catch` bloky:

```
try { ... }
  catch(NullPointerException | IllegalArgumentException e) {
   System.err.println("Invalid name.");
}
```

Reakce na výjimku — možnosti

Jak můžeme na vyhozenou výjimku reagovat?

Napravit příčiny vzniku chybového stavu

- opakovat akci (např. znovu nechat načíst vstup)
- poskytnout náhradu za chybný vstup (např. implicitní hodnotu)

Operaci neprovést

- sdělit chybu výše tím, že výjimku *propustíme* z metody (propadne z ní)

Oracle Java Tutorials: Lesson: Handling Errors with Exceptions

Kaskády bloků `catch`

- Pokud `catch` řetězíme, musíme respektovat, že výjimka bude zachycena nejbližším příhodným `catch`.
- Překladač si ohlídá, že kód neobsahuje *nedosažitelné* catch-bloky, např:

```
try {
  ...
} catch (Exception e) {
  ...
} catch (IllegalArgumentException e) {
  // won't compile, unreachable code
}
```

Výjimka z podtřídy (speciálnější) musí být zachycována dříve než výjimka obecnější.

Výjimky — jak ne I

```
try {
  ...
} catch ( Exception e ) {
  ...
}
```

Problém: Zachytáváme všechny výjimky, některé výjimky ale vždy chceme propouštět.

Řešení: Použít v `catch` speciálnější typ třídy.

Výjimky — jak ne II

```
try {
  ...
} catch ( NullPointerException e ) {
}
```

Problém: Prázdny `catch` blok — nedozvíme se, že výjimka byla vyhozena.

Řešení: Logovat, vypsat na chybový výstup nebo použít `e.printStackTrace();`

Výjimky — jak ne III

```
try {
  throw new NoSuchMethodException();
} catch ( NoSuchMethodException e ) {
  throw e;
}
```

Problém: Kód zachytí a následně vyhodí stejnou výjimku.

Řešení: Blok `catch` smazat — je zbytečný.

Hlídané a vlastní výjimky, blok `finally`

Blok `finally`

- Blok `finally` obvykle následuje po blocích `catch`.
- Zavolá se vždy, tj.
  - když je výjimka **zachycena** blokem `catch`
  - když je výjimka **propuštěna** do volající metody
  - když výjimka nenastane

Používá se typicky pro uvolnění (systémových) zdrojů, např. uzavření souborů.

Příklad na `finally`

- Zavírání vstupu (IO bude probíráno později):

```
InputStream is = null;
try {
  // trying to read from the file
  is = new FileInputStream("data.bin");
} catch (IOException e) {
   e.printStackTrace();
} finally {
   if (is != null) is.close();
}
```

Blok `finally` bez `catch`

- Dokonce existuje možnost try-finally
  - pro případy typu "potřebuji zavřít soubor jestli se výjimka vyhodí anebo ne"

```
InputStream is = null;
try {
  // trying to read from the file
  is = new FileInputStream("data.bin");
} finally {
  if (is != null) is.close();
}
```

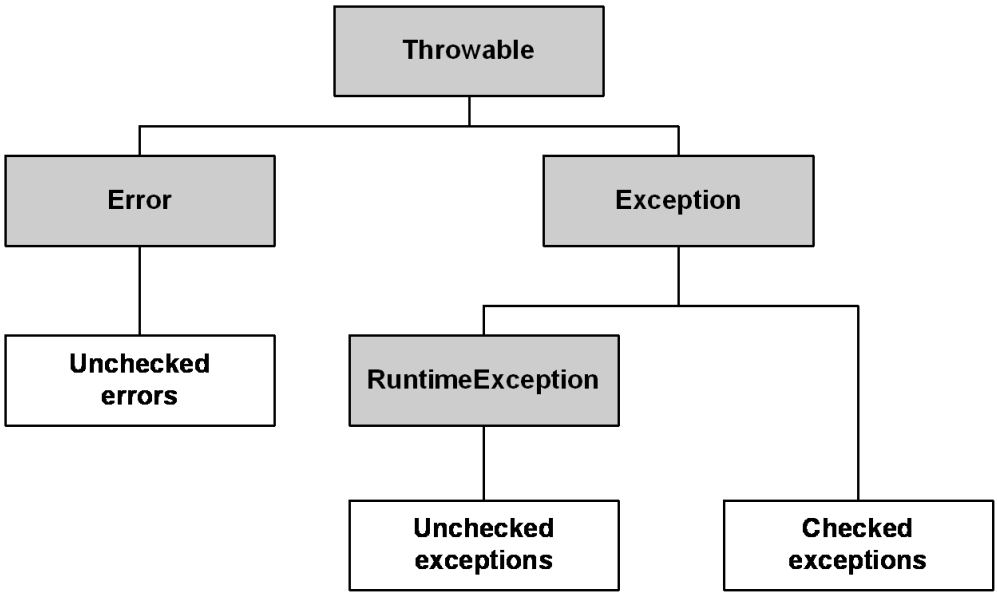
Na zamyšlení

- Příkaz `return` udělá okamžitý návrat z metody.
- Blok `finally` se vždy vykoná.
- Co vrátí následující kód?

```
try {
  return 5;
} finally {
  return 4;
}
```

Řešení: Obsah `finally` se vykoná skutečně vždy.

Hierarchie výjimek I



Hierarchie výjimek II

Throwable — obecná třída pro vyhazovatelné objekty, dělí se na:

- **Error** — chyby způsobené prostředím
  - `OutOfMemoryError, StackOverFlowError`
  - je téměř nemožné se z nich zotavit
  - překladač o nich neví
- **Exception** — chyby způsobené aplikací
  - `ClassCastException, NullPointerException, IOException`
  - je možné se z nich zotavit
  - překladač neví **jenom** o podtřídě **RuntimeException**

Hlídané & nehlídané výjimky

Existují 2 typy výjimek (**Exception**):

- běhové (**ne**hlídané), *unchecked*:
  - dědí od třídy `RuntimeException`
  - `NullPointerException, IllegalArgumentException`
  - netřeba je definovat v hlavičkách metody
  - reprezentují **chyby v programu**
- hlídané, *checked*:
  - dědí přímo od třídy `Exception`
  - `IOException, NoSuchMethodException`
  - je nutno definovat v hlavičkách metody použitím `throws`
  - reprezentují **chybový stav** (zlý vstup, chybějící soubor)

Metody propouštějící výjimku I

- Výjimky, které nejsou zachyceny pomocí `catch` mohou z metody propadnout výše.
- Tyhle výjimky jsou indikovány v hlavičce metody pomocí `throws`:

```
public void someMethod() throws IOException {
   ...
}
```

- Pokud hlídaná výjimka nikde v těle nemůže vzniknout, překladač ohlásí chybu.

```
// Will not compile
public void someMethod1() throws IOException { }
```

Metody propouštějící výjimku II

Pokud `throws` u **hlídaných výjimek** chybí, program se nezkompiluje:

```
// Ok
public void someMethod1() {
  throw new NullPointerException("Unchecked exception");
}
// Will not compile
public void someMethod1() {
  throw new IOException("Checked exception");
}
```

Pozor na rozdíl mezi `throw` a `throws`

Příklad s propouštěnou výjimkou

```
public static void main(String[] args) {
  try {
    openFile(args[0]);
    System.out.println("File opened successfully");
  } catch (IOException ioe) {
    System.err.println("Cannot open file");
  }
}
private static void openFile(String filename) throws IOException {
  System.out.println("Trying to open file " + filename);
  FileReader r = new FileReader(filename);
  // success, now do further things
}
```

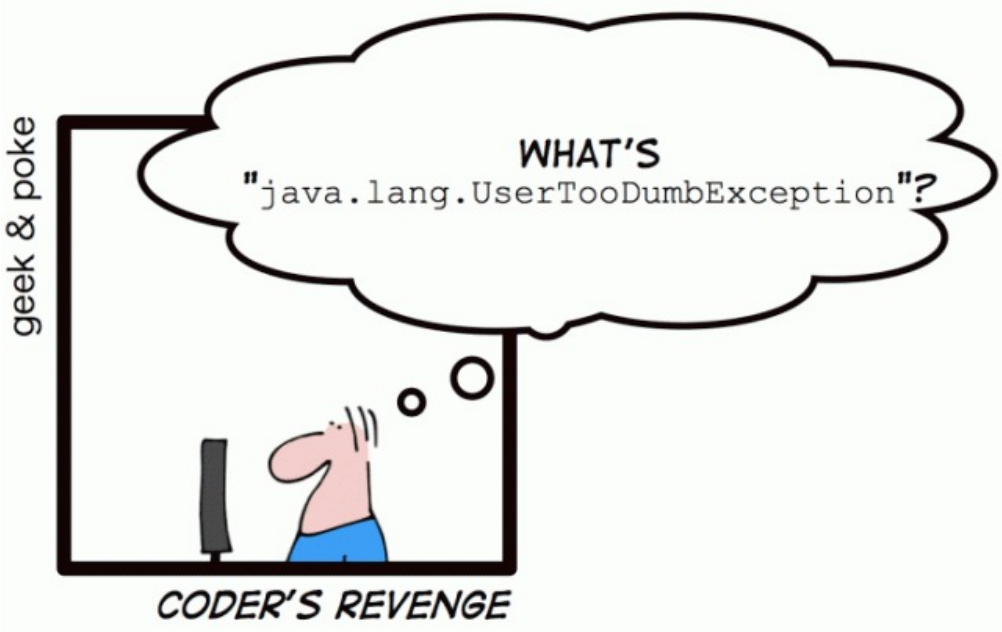
Stručné shrnutí

- u hlídaných výjimek **musí** být `throws`
- u nehlídaných výjimek **může** být `throws`

```
// throws is not neccessary
public void someMethod1() throws NullPointerException {
  throw new NullPointerException("Unchecked exception");
}
// throws is neccessary
public void someMethod2() throws IOException {
  throw new IOException("Checked exception");
}
```

Vytváření vlastních výjimek

- Třídy výjimek si můžeme definovat sami.
- Bývá zvykem končit názvy tříd výjimek na `Exception`:



Konstruktory výjimek

Ideální je definovat 4 konstruktory:

Constructor	Description
IllegalArgumentException()	Constructs an IllegalArgumentException with no detail message.
IllegalArgumentException(String s)	Constructs an IllegalArgumentException with the specified detail message.
IllegalArgumentException(String message, Throwable cause)	Constructs a new exception with the specified detail message and cause.
IllegalArgumentException(Throwable cause)	Constructs a new exception with the specified cause and ...

Příklad vlastní výjimky

- Vytvoření **hlídané** výjimky (nehlídaná dědí od `RuntimeException`):

```
public class MyException extends Exception {
  public MyException(String s) {
    super(s);
  }
  public MyException(Throwable cause) {
    super(cause);
  }
  ...
}
```

U výjimek stejně jako u jiných tříd můžeme mít atributy, konstruktory, atd.

Použití vlastní výjimky I

- Použití konstruktoru se `String message`:

```
public void someMethod(Thing badThing) throws MyException {
  if (badThing.happened()) {
    throw new MyException("Bad thing happened.");
  }
}
```

Použití vlastní výjimky II

- Konstruktor s `Throwable` se používá na obalování výjimek (i errorů).
- Výhodou je, že při volání `someMethod()` nemusíme řešit všechny možné typy výjimek:

```
public void someMethod() throws MyException {
  try {
    doStuff();
  } catch (IOException | IllegalArgumentException e) {
    throw new MyException(e);
  }
}
public void doStuff() throws IOException, IllegalArgumentException {
  ...
}
```

- Výjimky se dají zachytávat a řetězit:

```
try {
  int[] array = new int[1];
  a[4] = 0;
  System.out.println("Never comes to here");
} catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("ArrayIndexOutOfBoundsException has been thrown, continue in
code");
  // Puts chain of previous exception
  throw new MyException("Exception occured", e);
} finally {
  System.out.println("This always happens");
}
= Vstupy a výstupy v Javě =
```

Koncepce vstupně/výstupních operací v Javě

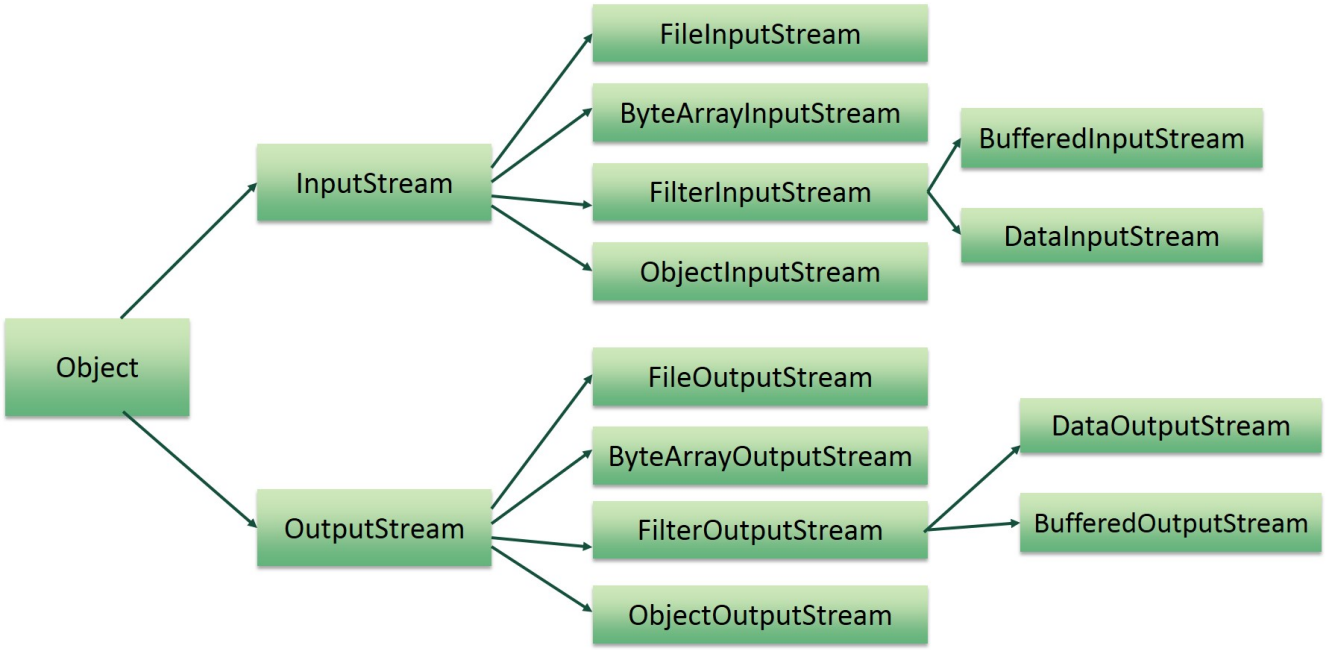
- V/V operace jsou založeny na *vstupně/výstupních proudech* (streams).
- Tím pádem je možno značnou část logiky programu psát nezávisle na tom, o který *konkrétní typ* V/V zařízení jde.
- Současně s tím jsou díky tomu V/V operace plně *platformově nezávislé*.

Table 2. Vstupně/výstupní proudy

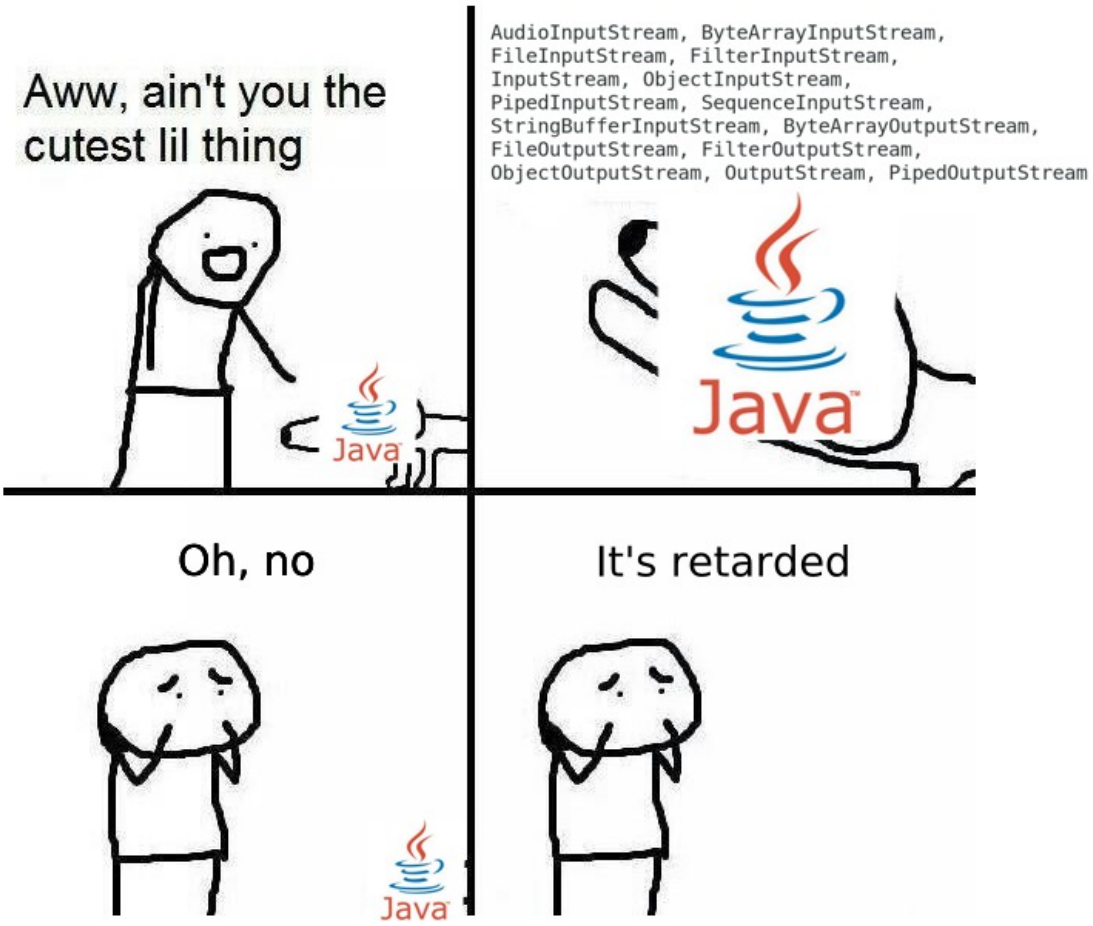
typ dat	vstupní	výstupní
binární	*InputStream*	*OutputStream*
znakové	*Reader*	*Writer*

Vstupy a výstupy v Javě

Zdroj: http://www.tutorialspoint.com/java/



Je toho příliš mnoho?



API proudů

- Téměř vše ze vstupních/výstupních tříd a rozhraní je v balíku *java.io*.
- Počínaje Java 1.4 se rozvíjí alternativní balík *java.nio (New I/O)*, zde se ale budeme věnovat klasickým I/O z balíku *java.io*.
- Blíže viz dokumentace API balíků java.io, java.nio.

Skládání vstupně/výstupních proudů

Proudy jsou koncipovány jako "stavebnice" — lze je spojovat za sebe a tím přidávat vlastnosti:

```
// casual input stream
InputStream is = System.in;
// bis enhances stream with buffering option
BufferedInputStream bis = new BufferedInputStream(is);
```

Neplést si **streamy** (proudy dat) s **lambda streamy**!

Stručné shrnutí

Closable	bytes closable	characters closable	lines closable
Input	InputStream	InputStreamReader	BufferedReader
Output	OutputStream	OutputStreamWriter	BufferedWriter

Základem znakových vstupních proudů je abstraktní třída *Reader* (pro výstupní *Writer*).

Konverze binárního proudu na znakový

Ze vstupního binárního proudu *InputStream* (čili každého) je možné vytvořit znakový *Reader*.

```
// binary input stream
InputStream is = ...
// character stream, decoding uses standard charset
Reader reader = new InputStreamReader(is);
// charsets are defined in java.nio package
Charset charset = java.nio.Charset.forName("ISO-8859-2");
// character stream, decoding uses ISO-8859-2 charset
Reader reader2 = new InputStreamReader(is, charset);
```

- Podporované názvy znakových sad naleznete na webu IANA Charsets .
- Obdobně pro výstupní proudy — lze vytvořit *Writer* z *OutputStream*.

Na zjištění, jestli je možné z čtenáře číst, se používa metoda `reader.ready()`.

Konverze znakového proudu na "buffered"

```
InputStreamReader isr = new InputStreamReader(is);
// takes another Reader and makes it bufferable
BufferedReader br = new BufferedReader(isr);
// BufferedReader supports read by line
String firstLine = br.readLine();
String secondLine = br.readLine();
```

Konverze souboru na proud

Soubor se dá lehce transformovat na proud znaků.

```
File file = new File("some_file.txt");

// file converted to InputStream
FileInputStream fis = new FileInputStream(file);

// file converted to OutputStream
FileOutputStream fis = new FileOutp2utStream(file);
```

Soubor se otevřel, proto je nutno ho pak zavřít! (později)

Znakové výstupní proudy

- Jedná se o protějšky k vstupním proudům, názvy jsou konstruovány analogicky (např. *FileReader* → *FileWriter*).
- Místo generických metod *read* mají *write(...)*.

```
OutputStream os = System.out;
os.write("Hello World!");
// we have to use generic newline separator
os.write(System.lineSeparator());

// bw has special method for that
BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(os));
bw.newline();
```

Zavírání proudů a souborů

- **Soubory** zavíráme vždy.
- **Proudy** nezavíráme.
- Když zavřeme `System.out`, metoda `println` pak přestane vypisovat text.

Povinné zavírání proudů

- Při otevření souboru (a konverzi na proud) se musíme postarat o dodatečné uzavření souboru.
- Před *Java 7* se to muselo dělat blokem *finally*:

```
public String readFirstLine(String path) throws IOException {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
     return br.readLine();
  } finally {
     if (br != null) br.close();
  }
}
```

Nově sa dá použít tzv. ***try-with-resources***:

```
public String readFirstLine(String path) throws IOException {
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

Více proudů

Pomocí *try-with-resources* lze ošetřit i více proudů současně — zavřou se pak všechny.

```
try (
     ZipFile zf = new ZipFile(zipFileName);
     BufferedWriter writer = new BufferedWriter(outputFilePath, charset)
    ) {
    ...
}
```

Obecně lze do hlavičky *try-with-resources* dát nejen proud, ale cokoli, co implementuje *java.io.Closeable*.

Výpis textu *PrintStream* a *PrintWriter*

PrintStream

je typ proudu standardního výstupu *System.out* (a chybového *System.err*).

- Vytváří se z binárního proudu, lze jím přenášet i binární data.
- Většina operací nevyhazuje výjimky, čímž uspoří neustálé hlídání (try-catch).
- Na chybu se lze zeptat pomocí *checkError()*.

PrintWriter

pro znaková data

- Vytváří se ze znakového proudu, lze specifikovat kódování.

Příklad s nastavením kódování:

```
PrintWriter writer = new PrintWriter(new OutputStreamWriter(output, "UTF-8"));
```

Serializace objektů I

Postupy ukládání a rekonstrukce objektů:

serializace

postup, jak z objektu vytvořit sekvenci bajtů perzistentně uložitelnou na paměťové médium (disk) a později restaurovatelnou do podoby výchozího javového objektu.

deserializace

je právě zpětná rekonstrukce objektu

Aby objekt bylo možno serializovat, musí implementovat **prázdné** rozhraní *java.io.Serializable*.

Serializace objektů II

- Proměnné objektu, které nemají být serializovány, musí být označeny modifikátorem, klíčovým slovem, *transient*.
- Pokud požadujeme "speciální chování" při (de)serializaci, musí objekt definovat metody:

```
private void writeObject(ObjectOutputStream stream) throws IOException

private void readObject(ObjectInputStream stream) throws IOException,
ClassNotFoundException
```

ObjectInputStream je proud na čtení serializovaných objektů.

Odkazy

Java Oracle Tutorial — essential Java I/O = Soubory =

Hlavní balíky V/V operací

- Základní věci jsou v balících java.io, java.nio.file.
- Základem je třída java.io.File

Práce se soubory přes objekty File

- Objekt třídy File je de facto nositelem jména souboru, jakási "brána" k fyzickým souborům na

disku.

- Nejde tedy o datovou strukturu nesoucí např. obsah souboru.
- Používá se jak pro soubory, tak *adresáře*, *linky* i soubory identifikované *UNC jmény*
- Je plně platformově nezávislé.

Odlišnosti systémů souborů

Na odstínění odlišností jednotlivých systémů souborů lze použít vlastností (uvádíme jejich hodnoty pro JVM pod systémem MS Windows):

File.separatorChar

`\\` jako char

File.separator

totéž jako String

File.pathSeparatorChar

`:` jako char

File.pathSeparator

totéž jako String

System.getProperty("user.dir")

adresář uživatele, pod jehož UID je proces JVM spuštěn

Vytvoření objektu File

Pro vytvoření objektu třídy File konstruktorem (NEJEDNÁ SE PŘÍMÉ VYTVOŘENÍ SOUBORU NA DISKU!) máme několik možností:

new File(String _filename_)

zpřístupní v aktuálním adresáři soubor s názvem *filename*

new File(File _baseDir_, String _filename_)

zpřístupní v adresáři *baseDir* soubor s názvem *filename*

new File(String _baseDirName_, String _filename_)

zpřístupní v adresáři se jménem *baseDirName* soubor s názvem *filename*

new File(URL _url_)

zpřístupní soubor se souborovým (file:) URL *url*

Existence a povaha souboru

boolean exists()

vrátí true, právě když zpřístupněný soubor (nebo adresář) existuje

boolean isFile()

test, zda jde o soubor a nikoli adresář

boolean isDirectory()

test, zda jde o adresář

Přístupová práva k souboru

boolean canRead()

mám právo čtení souboru?

boolean canWrite()

mám právo zápisu souboru?

Vytvoření souboru/adresáře

boolean createNewFile()

zkusí vytvořit soubor *soubor* a vrací true, právě když se podaří vytvořit.

boolean mkdir()

obdobně pro adresář

boolean mkdirs()

navíc si umí dotvořit i příp. neexistující adresáře na cestě

Vytvoření dočasného souboru

static File createTempFile(String _prefix_, String _suffix_)

Vytvoření dočasného (temporary) souboru — skutečně fyzicky vytvoří dočasný soubor ve standardním, pro to určeném, adresáři (např. c:/temp) s uvedeným prefixem a sufixem názvu

static File createTempFile(String _prefix_, String _suffix_, File _directory_)

dtto, ale vytvoří dočasný soubor v zadaném adr. directory

Smazání, přejmenování

boolean delete()

zrušení souboru nebo adresáře

boolean renameTo(File _dest_)

přejmenuje soubor nebo adresář (neumí přesun souboru/adresáře)

Další vlastnosti

long length()

délka (velikost) souboru v bajtech

long lastModified()

čas poslední modifikace v ms od začátku éry — tj. ve stejných jednotkách a škále jako systémový čas vracený System.currentTimeMillis().

String getName()

jen jméno souboru (tj. poslední část cesty)

String getPath()

celá cesta k souboru i se jménem

String getAbsolutePath()

absolutní cesta k souboru i se jménem

String getParent()

adresář, v němž je soubor nebo adresář obsažen

- Blíže viz dokumentace API třídy File.

Práce s adresáři

- Klíčem je opět třída File, použitelná i pro adresáře
- Jak např. získat (filtrovaný) seznam souborů v adresáři?
- Pomocí metody File[] listFiles(FileFilter ff) nebo podobné File[] listFiles(FilenameFilter fnf).
- FileFilter je rozhraní s jedinou metodou boolean accept(File pathname)
- obdobně FilenameFilter
- Viz Popis API java.io.FilenameFilter. = Balík New Input/Output (nio) =

Balík java.nio

- Třída Path jako nová a mocnější reprezentace cesty k souboru
- Pomocná třída Paths
- Pomocná třída Files pro pokročilejší manipulaci se soubory

Path

- Nástupce File, konceptuálně zhruba totéž, ale s více možnostmi
- Instance jsou nemodifikovatelné a vláknově bezpečné.
- Podporuje více systémů souborů na jednom počítači
- Nabízí metody jako getFileName, getParent, getRoot a subpath.
- Objekt Path je porovnatelný, iterovatelný a sledovatelný (Comparable<Path>, Iterable<Path>,

Watchable).

- Zejména sledovatelnost je novou vlastností, umožňuje reagovat na změny v systému souborů (např. v adresáři).

Zajímavé metody Path

- Kompletní dokumentace Path API
- Užitečné metody:

resolve
: umožňuje vyhodnotit danou cestu vůči jiné (např. relativní cestu vůči aktuálnímu adresáři)

relativize
: naopak relativizuje, vytvoří relativní z absolutní, když zadáme výchozí adresář.

startsWith, endWith
: podobně jako u řetězců, ale funguje na úseky cesty.

Files

- Typická "utility class", třída nabízející statické metody.
- Týkají se souborových systémů, souborů, adresářů atd.
- Nabízí metody pro:
  - kopírování
  - mazání
  - procházení (traverzace) systému souborů
  - přístup k metadatům souborů (čas, práva, uživatel)
  - přímé vytváření proudů (např. newBufferedReader)
- Další v tutoriálu Oracle File Operations
- Kompletní dokumentace Files API = Návazné předměty =

Co dál studovat?

Na tento základní kurz PB162 navazují na úrovni Bc. studia:

PV168 Seminář z programování v jazyce Java (jarní semestr)

- Náplní je zvládnutí Javy umožňující vývoj jednodušších praktických aplikací:
  - tvorba grafického uživatelského rozhraní
  - obsluha databází
  - základy webových aplikací

- V průběhu semestru se pracuje na uceleném projektu formou párového programování plus některých individuálních úloh.
- Učí kolektiv zkušených cvičících pod vedením Tomáše Pitnera, Luďka Bártka, Petra Adámka a Martina Kuby.

PB138 Moderní značkovací jazyky (jarní semestr)

- Náplní jsou XML a související technologie
- Prvky týmového vývoje (projekty, využití služeb hostování projektů, jako je GitHub).
- Učí kolektiv zkušených cvičících pod vedením Luďka Bártka a Tomáše Pitnera.

Pokročilé předměty

Na *Seminář z Javy* navazují na FI i pokročilejší kurzy:

PA165 Vývoj programových systémů v jazyce Java (podzimní semestr)

- Pokročilý předmět spíše magisterského určení, předpokládá znalosti/zkušenosti z oblasti databází, částečně sítí a distribuovaných systémů a také Javy zhruba v rozsahu PB162 a PV168.
- Náplní je zvládnutí netriviálních, převážně klient/server aplikací na platformě JavaEE.
- Přednáší a cvičí P. Adámek, T. Pitner, B. Rossi, M. Kuba, F. Nguyen, M. Kotala, J. Uhlíř, J. Čecháček.

Webové a mobilní aplikace

Problematice webových a mobilních aplikací se na FI věnují např.

- každý semestr PV226 Seminář Lasaris;
- v jarním semestru PV219 Seminář webdesignu;
- v podzimním semestru předmět PV249 Vývoj v Ruby;
- v jarním semestru PV239 Mobilní platformy a
- v podzimním návazný PV256 Projekt z programování pro Android; = Distribuce aplikací přes balíčky JAR =

Vtip

Nástroj JAR

- Javové programy se distribuují k uživateli různými způsoby.
- Ať už je způsob jakýkoli, většinou se však kód pro účel šíření balí pomocí nástroje `jar` (*Java ARchiver*).
- Distribucí nemyslíme použití nástroje typu "InstallShield"..., ale spíše něčeho podobného `tar` / `ZIP`.
- Java na sbalení množiny souborů zdrojových i přeložených (`.class`) a dalších nabízí nástroj `jar` .
- Sbalením vznikne soubor (archív) `.jar` formátově podobný `ZIP` u (obvykle *je* to ZIP formát), ale nemusí být komprimován.
- Kromě souborů obsahuje i metainformace (tzv. *MANIFEST*)
- Součástí archívu nejsou jen `.class` soubory, ale i další zdroje, např. *obrázky*, *soubory s národními variantami řetězců* (resouce bundles), *zdrojové texty* programu, *dokumentace* ...

Spuštění jar

- Spuštění: `jar {ctxu} [vfm0M] [jar-file] [manifest-file] [-C dir] files`
  - `c` - vytvoří archív
  - `t` - vypíše obsah archívu
  - `x` - extrahuje archív
  - `u` - aktualizuje obsah archívu

- Volby:
  - `v` - verbose
  - `0` - soubory nekomprimuje
  - `f` - pracuje se se souborem, ne se "stdio"
  - `m` - přibalí metainformace z `manifest-file`
- Parametr `files` uvádí, které soubory se sbalí, mohou být i nejavové (např. dokumentace API nebo datové soubory)

`jar` - příklad

- Vezměme následující zdrojový text třídy `JarDemo` v balíku `tomp.ucebnice.jar`, tj. v adresáři `c:\tomp\pb162\java\tomp\ucebnice\jar`
- Vytvoříme archív se všemi soubory z podadresáře `tomp/ucebnice/jar` (s volbou `c` - create, `v` - verbose, `f` - do souboru):
- `jar cvf jardemo tomp/ucebnice/jar`
- Vzniklý `.jar` soubor lze prohlédnout/rozbalit také běžným nástrojem typu `unzip`, `gunzip`, *WinZip*, *PowerArchiver* nebo souborovým managerem.
- Tento archív rozbalíme v adresáři `/temp` následujícím způsobem:
- `jar xvf jardemo`

Rozšíření `.jar` archívů

- Formáty vycházející z `JAR`:
  - webové aplikace → `.war`
  - enterprise (EJB) aplikace → `.ear`
- Liší se podrobnějším předepsáním adresářové struktury a dalšími povinnými metainformacemi.

Tvorba spustitelných archívů

- Vytvoříme `jar` s manifestem obsahujícím tento řádek: `Main-Class: NázevSpouštěnéTřídy`
- poté zadáme: `java -jar NázevJARu.jar`
- a spustí se metoda `main` třídy `NázevSpouštěnéTřídy`.

Spuštění archívu - příklad

- Spuštění aplikace zabalené ve spustitelném archívu je snadné:

```
java -jar jardemo.jar
```

- a spustí se metoda `main` třídy `tomp.ucebnice.jar.JarDemo`:

Další příklad spuštění jar

- `jar tfv svet.jar | more`
- vypíše po obrazovkách obsah (listing) archívu `svet.jar`