PB162 — Programování v jazyce Java

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 PB112 jaro2024 v IS

1. Úvod do Javy

- Jak přejít od Pythonu, C neb JavaScriptu k Javě?
- slidy Cíle předmětu | pro tisk HTML, PDF
- slidy Přechod k Javě | pro tisk HTML, PDF
- slidy První program, třída, objekt | pro tisk HTML, PDF
- slidy Proměnné, deklarace | pro tisk HTML, PDF
- slidy Balíky | pro tisk HTML, PDF
- slidy Spuštění programu | pro tisk HTML, PDF
- **slidy na doma** Základní příkazy - volání metod, přiřazení, návrat | pro tisk HTML, PDF
- slidy Řídicí struktury - větvení, cykly | 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 a záznamy | pro tisk HTML, PDF
- slidy Přetěžování metod | pro tisk HTML, PDF
- slidy Překrývání metod, třída Object | pro tisk HTML, PDF
- slidy Likvidace objektů | pro tisk HTML, PDF

4. Rozhraní

- slidy Rozhraní | pro tisk HTML, PDF
- slidy Výchozí a statické metody 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
- slidy Polymorfismus | pro tisk HTML, PDF
- slidy Konstruktory - tipy | pro tisk HTML, PDF
- slidy Potíže s dědičností | pro tisk HTML, PDF

6. Pole, porovnávání objektů, abstraktní třídy, moduly

- slidy Pole, třída `Arrays` | pro tisk HTML, PDF
- slidy Porovnávání objektů | pro tisk HTML, PDF
- slidy Pomocná třída Objects | pro tisk HTML, PDF
- slidy Abstraktní třídy | pro tisk HTML, PDF
- slidy Moduly | pro tisk HTML, PDF

7. Kontejnery: `Collection`, `Set`, `List`, `Iterator`

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

8. Kontejnery: `Map`, `SortedMap`, `Collections`; lambda výrazy, proudy

- slidy Mapy | pro tisk HTML, PDF
- slidy Uspořádané kolekce | pro tisk HTML, PDF
- slidy Porovnání kontejnerů, třída `Collections` | pro tisk HTML, PDF
- slidy Lambda výrazy | pro tisk HTML, PDF
- slidy Proudy (Stream) | pro tisk HTML, PDF
- slidy Parametrické (generické) typy | pro tisk HTML, PDF

9. Výjimky

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

10. Vnořené třídy

- slidy Vkládání závislostí | pro tisk HTML, PDF
- slidy Vnořené a vnitřní třídy | pro tisk HTML, PDF

11. Vstupy a výstupy, soubory

- slidy Vstupy a výstupy | pro tisk HTML, PDF
- slidy Soubory | pro tisk HTML, PDF
- slidy `Path` a `Files` | pro tisk HTML, PDF
- slidy Kódování znaků | pro tisk HTML, PDF

12. Konzultace, návaznosti, pokročilá témata

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

Program "Hello World!"

- Abychom měli kam náš kód psát, vytvoříme třídu Demo s hlavní funkci main, která se zavolá při spuštění programu.
- V Javě nestačí výkonný kód (příkazy) umístit jen tak do zdrojového souboru.

- V Javě sice existuje možnost interaktivní práce "REPL" (*read-eval-print-loop*), ale moc se nepoužívá.
- V praxi tedy většinou napíšeme kód aspoň do statické metody `main` v nějaké třídě.

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

Program "Hello World!" - proč a jak `main`

```
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());
   }
}
```

Modifikátory `public` a `private`

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` 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 *šablona* pro objekty — osoby.
- Jak vytvořím **konkrétní** osobu s jménem Jan?

```java
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 (`private`) 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.

Deklarace vs vytváření objektů

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

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

Deklarace sama nic nevytvoří

- Pouhá deklarace proměnné objektového typu (`Person jan`) žádný objekt nevytvoří.
- Pouze nám to pojmenuje místo pro *odkaz*, který následně naplníme odkazem na skutečně vytvořený objekt.

- K vytvoření tohoto objektu 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.

Repl.it demo k třídám a objektům

- https://repl.it/@tpitner/PB162-Java-Lecture-01-objects

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 vs objekt

Třída

- Reprezentuje obecně více prvků 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
}
```

Proměnné

- Obecně slouží k pamatování si hodnot v paměti během chodu programu.
- Některé - a to lokální proměnné - si pamatují jen během volání podprogramu (metody) a pak zmizí.
- Jiné - objekty a prvky objektů - přetrvávají, jak dlouho potřebujeme, nejdéle do konce běhu programu.
- Trvalé ukládání dat už se děje jinde - na vnějších pamětech, discích.

Pojmenování proměnných

- Proměnné se stejně jako pro metody a třídy pojmenovávají pomocí *identifikátorů*.
- Identifikátor podobně jako např. v Pythonu musí:
- začínat *písmenem* nebo znakem podtržítka `_`, které ovšem raději nepoužívejme, není to v Javě zvykem

Konvence v Javě na rozdíl od Pythonu

názvy tříd a dalších typů, třeba výčtů, a konstant

začínáme písmenem velkým — třída `Person`, konstanta `MAX_COUNT`

atributy, proměnné, metody…

malým písmenem — atribut `numHeads`, metoda `print`

podtržítka

většinou nepoužíváme

víceslovné názvy

namísto podtržítek jako v Pythonu (`my_variable`) píšeme v "CamelCase" (`myVariable`) kromě

konstant, kde píšeme velkými písmeny a slova oddělujeme podtržítky (`MAX_COUNT`)

Kategorie proměnných

V Javě jsou proměnné realizovány jako místa v paměti nesoucí hodnotu nebo odkaz na objekt. Rozlišujeme tyto kategorie podle místa výskytu proměnné:

- **atributy** (nebo též *proměnné objektu*, *instanční proměnné*)
- **statické proměnné** (nebo též *statické atributy* nebo *proměnné třídy*)
- **lokální proměnné** (místní proměnné, *proměnné v metodě*)

Lokální proměnné

- Ve příkladu s bankovními účty se v metodě `main` objevily proměnné (účty), které nebyly atributy objektů, ale pracovalo se s nimi pouze v metodě samotné.
- Takové proměnné se označují podobně jako v jiných jazycích jako *lokální*.
- Podobně jako atributy mají i tyto proměnné svůj datový typ - primitivní nebo objektový.
- V případě primitivního pak proměnná nese přímo hodnotu (například číslo nebo `boolean`).
- V případě objektového nese odkaz na objekt.
- V tomto ohledu se to tedy nijak neliší od atributů.

```
public static void main(String[] args) {
   Account petrsAccount = new Account();
   petrsAccount.add(100.0);
}
```

Lokální v Javě = automatické

- Liší se však okamžikem vytvoření: vytvoří se při každém zavolání metody (v případě objektové proměnné se samozřejmě vytvoří jen ten odkaz, nikoli celý objekt).
- Proto jsou také označovány jako *automatické*.
- Technicky jsou vytvořeny alokací místa na zásobníku, tedy podobným mechanismem, jako se ukládají návratové adresy při volání metod.

Deklarace

- Úplně klasicky vypadá například deklarace lokální proměnné takto:
- primitivní typy (čísla, `boolean`…): `int i = 2`, `boolean isOK = true`
- objektové typy:

```
Person jan = new Person("Honza");
// compiles iff Employee is a subclass of Person
Person petr = new Employee("Petr");
```

Odvození typu

- Doposud jsme viděli, že na levé straně byl uveden deklarovaný typ proměnné.
- V novějších verzích Javy (8+) lze využít tzn. *odvození typu* (type inference).
- Z typu výrazu na pravé straně odvodíme typ proměnné na levé straně:

```
var petr = new Employee("Petr Servus");
// so this does not compile - Employee expected:
petr = new Person("Petr Svatý");
```

Odvození typového parametru

- Kromě výše uvedených jednoduchých situací uvidíme později další.
- U tzv. parametrizovaných typů, např. seznamů, lze odvodit typ prvku seznamu z jedné strany na druhou - i zleva doprava.

```
List<Person> listPeople = new ArrayList<>();
// or the type on the left is inferred to ArrayList<Person>
var listPeople = new ArrayList<Person>();
```

Odvození typu lambda výrazu

- Lambda výrazy (konstrukty funkcionálního paradigmatu) jsou také typované.
- Funguje zde odvození typů - zde se odvodí, že v seznamu jsou osoby:

```
var listPeople = new ArrayList<Person>();
// type of the list item is Person
// inferred type of lambda is `(Person p) -> p.print()`:
listPeople.forEach(p -> p.print());
```

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

```
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. Stejně tak raději žádná podtržítka v názvech.

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 jiném balíku `cz.muni.fi.pb162.banking`
- použije pro třídu `Person` její plný název balíku
- nebo lze použít `import` názvu třídy

```
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;
```

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

`import` sám nezajistí viditelnost

- Když bude Person neveřejná, nebude vidět do jiného balíku
- `import` je pak k ničemu
- následující soubor `Account.java` se nezkompiluje:

```java
package cz.muni.fi.pb162;
... // in file cz/muni/fi/pb162/Person.java
class Person { // package-local class
   private Person owner; // owner of this Account is a Person
}
... // in file cz/muni/fi/pb162/banking/Account.java
package cz.muni.fi.pb162.banking;
import cz.muni.fi.pb162.Person;
public class Account {
   // class Person imported but invisible from here
   private Person owner;
}
```

Deklarace `import` třídy

- 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` celého balíku

- Import **všech tříd** z balíku provedeme např. `import cz.muni.fi.pb162.*;`
- Doporučuje se "import s hvězdičkou" nepoužívat vůbec — máme-li více `*` importů:
  - Problém 1: a obojí z nich obsahují třídu `Person`, která z nich se použije? (nepoužije se žádná, dostanete kompilační chybu)
  - Problém 2: nepoznáme na první pohled, ze kterého balíku identifikátor pochází
  - Řešení: *Nebudeme to používat*!
- Pozn.: Hvězdičkou **nezpřístupníme** třídy z *podbalíků* — např. `import cz.*` nezpřístupní třídu `cz.muni.fi.pb162.Person`.

Základní životní cyklus javového programu

- V Javě sice existuje interaktivní (tzv. REPL *read-eval-print-loop*) prostředí,
- většinou však kód napíšeme do souboru, uložíme, přeložíme a spustíme.
- 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`)
- (anebo totéž jednodušeji zelenou šipkou v některém z IDE, třeba v IntelliJ IDEA)

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 `java`, `javac`... umístěné v podadresáři `bin` instalace Javy.

Překlad "Ahoj!"

- Máme nainstalován Java **SDK** 8 (jen příklad, funguje i s dalšími verzemi)
- 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) {
```

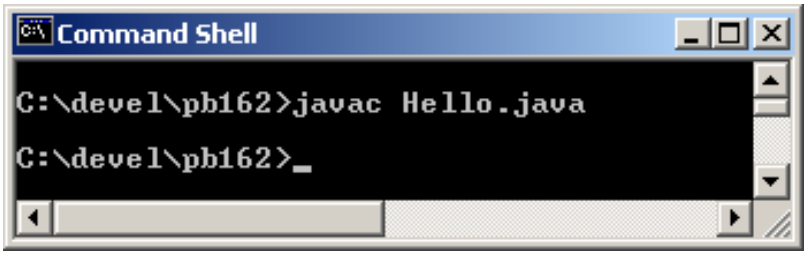
```
    System.out.println("Ahoj!");
  }
}
```

Spuštění "Ahoj!"

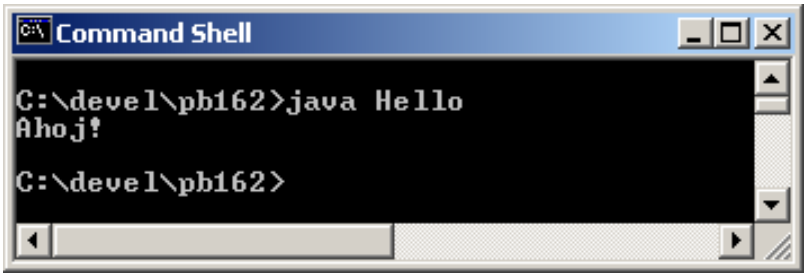
- Spustíme program Hello příkazem `java Hello`, název třídy je bez přípony `.class`
- V nejnovějších verzích Javy lze spustit rovnou včetně překladu: `java Hello.java`, pak se přeloží s hned spustí, viz dále.
- Je-li program správně napsán a přeložen, vypíše se `Ahoj!`

Překlad & Spuštění

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



Spuštění voláním `java`:

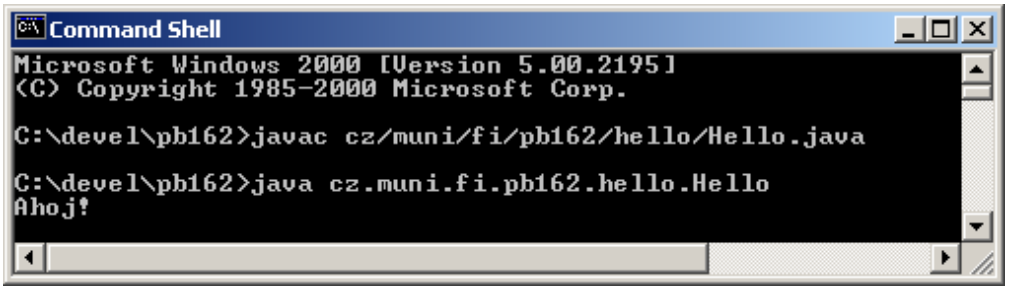


Co když je třída v adresáří (balíku)

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`.

Od Java 11: překlad a spuštění v jednom

- Počínaje Java 11 lze na jednoduché(!) programy použít přímý postup:
- `java HelloWorld.java` → program se přeloží a následně spustí
- funguje dokonce i tehdy, nebude-li se třída s metodou `main` jmenovat `HelloWorld`
- blíže také 330: Launch Single-File Source-Code Programs

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`

Spuštění sbaleného programu

- Java má formát JAR (Java ARchive) pro distribuci programů složených z mnoha tříd.
- Je podobný ZIP s pomocnými soubory navíc.
- V archívu se deklaruje, která třída (=její metoda `main`) se spouští.
- Lze pak spouštět jakoby celý archív: `java -jar myapp.jar`

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)
  - na levé straně musí být *proměnná*
  - na pravé straně *výraz přiřaditelný* (assignable) do této proměnné
- Rozlišujeme přiřazení
  - *primitivních hodnot* a
  - *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
- `Person z1 = new Person()`
- Co to udělalo?
  1. na pravé straně se vytvoří nový objekt typu `Person` (`new Person()`)
  2. přiřazení jej přiřadilo do proměnné `z1` typu `Person`

Kopie odkazu na objekt

- 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.

Přehled

- Větvení `if-else`
- Cyklus `while` - kde stačí vstupní po
- Cyklus `for` - vstupní a pokračovací podmínka, akce po každém provedení - obdoba téhož v C/C++
- Cyklus `for` ve variantě *iterace* po prvcích pole, seznamu, množiny… - tedy něco jako `foreach`
- Cyklus `do-while` - pokračovací podmínka se testuje na konci těla cyklu
- Vícecestné větvení `switch` - `case` - `default` - podobné jako v C/C++

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.

Příklad podmíněného příkazu

Tedy javové `if`:

```
if (name.equals("Debora"))
   System.out.println("Hi, Debora");
```

je ekvivalentní `if` v Pythonu:

```
if name == 'Debora':
   print('Hi, Debora')
```

Rozdíly:

- v Javě stejně jako v C/C++ musejí být závorky kolem podmínky
- v Pythonu je za podmínkou dvojtečka `:`
- a povinné odsazení příkazu/ů, které se podmíněně provedou

Příklad podmíněného bloku příkazů

V Javě nutno uzavřít do bloku `{}`:

```
if (name.equals("Debora")) {
   System.out.println("Hi, Debora");
   System.out.println("I am your friend!");
}
```

V Pythonu stačí odsadit:

```
if name == 'Debora':
   print('Hi, Debora')
   print('I am your friend!')
```

Úplné větvení

- `if (logický výraz) příkaz1 else příkaz2`
- Platí-li *logický výraz*, provede se *příkaz1*.
- Neplatí-li, provede se *příkaz2*.

Úplné větvení

V Javě:

```
if (name.equals("Debora"))
   System.out.println("Hi Debora");
else
   System.out.println("Who are you?");
```

V Pythonu:

```
if name == 'Debora':
   print('Hi Debora!')
else:
   print('Who are you?')
```

Lépe do bloků

Je-li ve větvích po jednom příkazu, lze nechat tak. Lepší je ovšem vždy závorkovat, uzavřít bloky:

```
if (name.equals("Debora")) {
   System.out.println("Hi Debora");
} else {
   System.out.println("Who are you?");
}
```

Postupné větvení

V Javě není speciální konstrukce "elseif", která v negativním případě zkouší další větev. Jednoduše se použije `else` a za ním `if`.

```
if (name.equals("Debora"))
   System.out.println("Hi Debora");
else if(name.equals("Joshua"))
   System.out.println("Hi Joshua");
else
```

```
   System.out.println("Who are you?");
```

Zatímco v Pythonu ano:

```
if name == 'Debora':
   print('Hi Debora!')
elif name == 'Joshua':
   print('Hi Joshua!')
else:
   print('Who are you?')
```

Rozepsané postupné větvení

Výše uvedená javová konstrukce přepsaná s blokovými závorkami:

```
if (name.equals("Debora")) {
   System.out.println("Hi Debora");
} else {
   if(name.equals("Joshua")) {
      System.out.println("Hi Joshua");
   } else {
      System.out.println("Who are you?");
   }
}
```

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);
```

Totéž s možností ukončení

- Použití příkazu `break`
- Realizuje "násilné" ukončení průchodu *cyklem* (nebo *větvením* `switch`).
- Doplnění `break` do *cyklu*:

```
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();
   // if the input is empty then finish
   if(input.length() == 0) break;
   try {
      number = Float.parseFloat(input);
      isOK = true;
   } catch (NumberFormatException nfe) {
      isOK = false;
   }
} while(!isOK);
System.out.println("We've got the number " + number);
```

`break` v cyklu a podmínce

- `break` ukončí cyklus `for` podobně jako předtím `do-while`:

```
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");
 }
```

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:

```java
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; ...

Použití nové syntaxe větví →

- V nových verzích Java 14+ lze použít namísto otravného ukončování větví pomocí `break` (což je kdysi poděděné z C) nové syntaxe s šipkou `→`.

```
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);
    case TUESDAY                -> System.out.println(7);
    case THURSDAY, SATURDAY     -> System.out.println(8);
    case WEDNESDAY              -> System.out.println(9);
}
```

- Ve vybrané větvi se provede příkaz nebo blok, je-li uveden v `{ }`.
- Jednu větev lze vybrat více výrazy současně (např. `MONDAY, FRIDAY, SUNDAY`).

Výraz `switch`

- switch nemusíme používat jen jako příkaz vícecestného větvení
- zejména ve výše uvedených příkladech, kdy se v každé větvi provedlo jen přiřazení do stejné proměnné, je lepší použít switch v nové formě jako jakýsi *rozšířený podmíněný výraz*
- funguje v Javě 14+

```
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

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>");
         }
      }
   }
```

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`.

```
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");
    }
}
```

Repl.it demo k řídicím strukturám

- https://repl.it/@tpitner/PB162-Java-Lecture-03-control-structures

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.

Konstruktory v Pythonu a Javě

- V Pythonu jsou to metody **def *init*(self):**
- V Javě se jmenují přesně stejně jako jejich třída (a bez návratového typu)
- Konstruktorů v jedné třídě může být více - musí se pak lišit počtem, evt. typem parametrů

Příklad třídy s konstruktorem v Pythonu

```
class Person:
    # default constructor
    def __init__(self, name):
        self.name = name
```

Příklad třídy s konstruktorem v Javě

```
public class Person {
  private String name;

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

Rozdíly Python vs Java

Java

identifikátor `this` znamená, že se přistupuje k atributům objektu. *Není nutné* používat tam, kde se neshoduje jméno atributu a parametru.

Python

identifikátor `self` znamená totéž, ale musí se používat *vždy při přístupu* k atributu nebo metodě.

Konstruktory — použití

Python

```
pepa = Person("Pepa from Hongkong");
```

Java

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

- Toto volání vytvoří objekt `pepa` a naplní ho jménem.
- Následně je možné získávat hodnoty proměnných objektů pomocí tečkové notace, např. `pepa.name`.
- V tomto případě by nebylo možné volat `Person pepa = new Person();`, protože existující konstruktor má jeden parametr.

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.

Pouhá deklarace proměnné

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

- Výrazný rozdíl oproti C++: v Javě vůbec nepůjde přeložit.
- Nevytvoří žádný objekt a překladač ví, že proměnná `p` neukazuje nikam.
- Tudíž veškerá volání `p.getName()` a podobně by byla nefunkční.

Nevytvoření objektu

- Toto již přeložit půjde - když do odkazu přiřadím `null`.
- Nicméně, co se stane, když zavolám nad odkazem `null` metody?

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

- Kód po spuštění "spadne", neboli zhavaruje, předčasně skončí.
- Java sa snaží pád programu popsat pomocí *výjimek* (exceptions).
- 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 fakticky 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 se 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:

```
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"
}
```

Přiřazení `janCopy.name = "Janko"` bude možné jen tehdy, nebude-li atribut `name` privátní, jinak bychom museli mít něco jako `janCopy.setName("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

Návrhové vzory

- Návrhové vzory jsou osvědčené způsoby objektové dekompozice v jasně popsaných situacích
- Jsou použitelné pro libovolný objektově orientovaný jazyk
- Jejich aplikace ale vyžaduje návrhová rozhodnutí, která mohou být ovlivněna vlastnostmi programovacího jazyka
- Mnohé si postupně stručně představíme s cílem
  - demonstrovat, že objektová dekompozice Java Core API není náhodná,
  - motivovat vás l používání vozrů při dekompozici vašeho kódu.

Vytvářecí návrhové vzory

Speciální podskupina návrhových vzorů, která nabízí alternativní způsoby k vytváření objektů, než je prosté volání konstruktoru

- Singleton: Vytvoření jediné instance třídy, kterou všichni sdílí a snadno k ní přistupují odkudkoli.
- Builder: Konstrukce složitého objektu po kouscích (např. vytvoření grafu přidáváním uzlů a hran).
- Prototype: Namísto vytváření nového objektu naklonuj existující objekt.
- Abstract Factory: Jednotné místo pro vytváření vzájemně kompatibilních instancí různých tříd.
- Factory Method: Přenechání konstrukce objektu podtřídě.
- Dobrá praxe dle Josh Bloch: Effective Java

Co je zapouzdření

- Naprosto zásadní vlastnost objektového přístupu, asi 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;
   }
 }
```

- Obvykle nechceme, aby kdokoli mohl modifikovat atributy `name`, `age` po vytvoření objektu, ale pouze prostřednictvím metod této třídy.
- Dosti často chceme, aby třídu a (některý) konstruktor 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 přístup "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 souboru se stejným jménem.

Metody `get` a `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 zavolaní *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);
```

```
}
```

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 na rozdíl např. od C nebo Pythonu *nespojujeme podtržítkem*: `This_is_bad_in_Java`.
- Používáme tzv. *camelCase*, "velbloudí" střídání velkých a malých písmen: `myVeryLongMethodNameIsOK()`.
- Delší jména až tak nevadí, podstatná je čitelnost.
- 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*, *záznamů* a *výčtů*

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

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 a prvků výčtu

- Konstantou rozumíme hodnotu, která se *nemění*.
- Totéž prvek výčtu, například `SPRING` je prvkem výčtu `Season`.
- 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*.

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`, …
- mezi objektové patří i výčtové typy (`enum`) podobné třídám (de facto jsou to specifické třídy)
- zcela nově pak Java nabízí i typy `record` (záznam) = třídy neměnitelných objektů

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.

Repl.it demo k primitivním typům a objektům

- https://repl.it/@tpitner/PB162-Java-Lecture-02-primitive-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. "odkaz 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 po přiřazení *odkazují* (ukazují) na stejný objekt.
- Nevznikají přitom žádné kopie objektů, pouze odkaz (adresa) se zkopíruje do proměnné vlevo.

```
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];`

Řetězce ve zkratce

- `String` je velice často užívaný typ, jeho hodnotami jsou posloupnosti znaků.
- Jedná se o typ objektový, nikoli primitivní.
- Proměnné typu `String` tedy odkazují na objekty, které teprve obsahují znaky řetězce.
- Řetězec smí být prázdný (velikost neboli délka = 0), zapisujeme `""`.
- Prázdný řetězec je pochopitelně něco jiného než neexistující řetězec (odkaz `null`).

Vytváření řetězců

- Pro vytváření řetězců nemusíme používat operátor `new`; stačí napsat:

```
String s = "Hello!";
System.out.println("Greeting is " + s);
// this is also possible:
s = new String("Hello!");
```

Porovnání řetězců

- Z objektové povahy řetězců plyne, že k jejich porovnávání musíme přistupovat jinak, než např. k porovnávání čísel.
- Většinou potřebujeme zjistit, zda jsou dva řetězce stejné, tzn. mají:
  - stejný počet znaků (nebo jsou oba prázdné)
  - na stejných pozicích mají stejné znaky
- Takovou shodu ověříme pomocí metody `equals`, nikoli pomocí `==`

```
String s = "Hello!";
String t = "Hell" + "o!";
System.out.println("Strings are equal? " + s.equals(t));
```

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.
- Komentáře říkají nejen *jak je kód psán a co dělá*, ale také a zejména *proč se to tak dělá*.
- 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*.

Připomenutí z Pythonu

```
# Simple comment, not going to documentation
def my_function():
    '''docstring comment of this function -
    going to help and doc'''
    return None

print("Using __doc__:")
print(my_function.__doc__)

print("Using help:")
help(my_function)
```

- Namísto *docstringů* používáme v Javě dokumentační komentáře `/** zde je */`

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

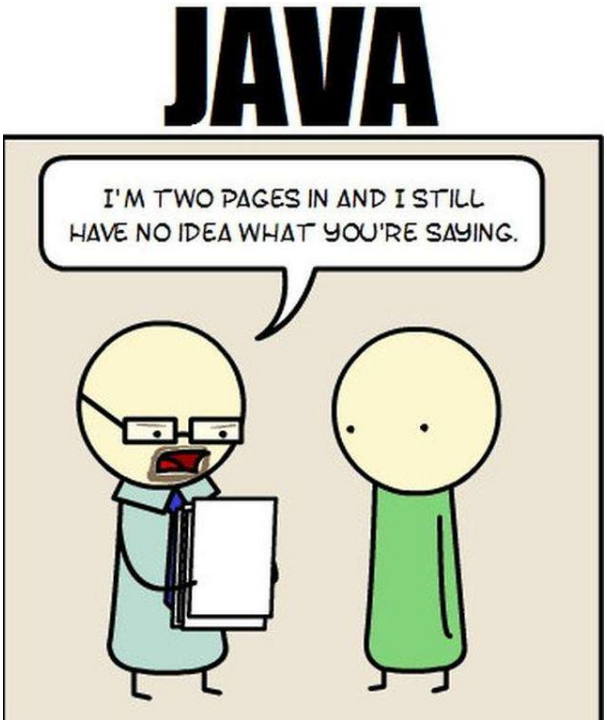
```
/**
 * Calculates velocity.
 */
System.out.println(triangle.getArea());
```

- Zbytečné komentáře

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

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

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 `main` je veřejná, chceme spouštět
- `void` reprezentuje návratový typ — že `main` nic nevrací

Co znamená `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.
- Sémantika `static` je +/- stejná jako v Pythonu.

Dosud byly metody a atributy *nestatické*

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

```java
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 atributy patřící *celé třídě*, tj. *všem objektům třídy*.
- Taková proměnná (nebo metoda) existuje pro jednu třídu jen *jednou*.
- Označujeme ji `static`.

Podobně v Pythonu

- Statické proměnné/atributy existují i v Pythonu.
- Označují se jako *proměnné třídy* (takto můžeme označovat i v Javě), zatímco atributy jsou *proměnné objektu*.
- V Pythonu se poznají tak, že se *neoznačují* `self`.

```python
# Class for Computer Science Student
class CSStudent:
    stream = 'cse'                  # Class Variable
    def __init__(self,name):
        self.name = name            # Instance Variable
```

Použití statických metod

- Chceme metodu `max`, která vrací maximální hodnotu dvou čísel
- K tomu žádné objekty nepotřebujeme
- Uděláme ji jako statickou
- Kdekoli zavoláme `Calculate.max(-2, 8)`

```java
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;
  }
}
```

Spuštění nestatické metody `max`

- 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 — třeba dvě, tři, deset???

Možné řešení — singleton

- Často je lepší aplikovat *návrhový vzor Singleton* (jedináček)
- Jedná se běžnou třídu, např. `PersonCounter`, která má své běžné nestatické atributy a metody zajišťující potřebnou funkcionalitu
  - Např. metody `void increaseNumPeople()` a atribut `int peopleCount` zajišťující počítání vytvořených lidí.

```
public class PersonCounter {
   private int peopleCount = 0;
   public void increaseNumPeople() {
      peopleCount++;
   }
   public int howManyPeople() {
      return peopleCount;
   }
}
```

Možné řešení — singleton (pokr.)

- Singleton ale navíc zajišťuje vytvoření jediné sdílené instance (např. jediného počítadla lidí pro celý systém):
  - Zakáže se volání konstruktoru (např. tak se vytvoří jediný bezparametrický **privátní** konstruktor)
  - Místo volání konstruktoru se nabídne veřejná metoda `getInstance()`, která zjistí, jestli (jediná) instance již existuje. Pokud ne, vytvoří jí (voláním privátního konstruktoru) a uloží do statického atributu třídy. Pokud již existuje, rovnou se instance vrátí.
- A dále zajišťuje jednoduchý přístup k jediné instanci odkudkoliv:
  - Metoda `getInstance()` je statická, tj. kdokoliv odkudkoliv může zavolat `PersonCounter.getInstance().increaseNumPersons()`.
  - Pokud chceme automaticky počítat vytvoření instance třídy `Person` z předchozího příkladu, lze toto volání jednoduše přidat konstruktorů třídy `Person`

Kompletní 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 increaseNumPeople() {
        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! :-)

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.
- Třeba `static final int MAX_CAPACITY = 12` může znamenat konstantu, která se může v dalších verzích programu zvětšit.
- Konstanty v Javě jsou vždy umístěny v některé ze tříd či rozhraní - nemohou jen tak globálně "plavat nad vším".
- 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.

Připomenutí: konstanty v Pythonu

```
Create a constant.py:
# declare constants
PI = 3.14
GRAVITY = 9.8
```

- Jmenná konvence (velká písmena) je stejná jako v Javě.
- V Pythonu je to jen konvence, lze znovu přiřadit, tj. `PI = 3.2` je technicky možné.
- V Javě bráníme opakovanému přiřazení označením symbolu jako `final`.

Definice konstanty

- Konstanty jsou vždy:

  Statické (`static`)

  stačí nám jedna hodnota pro celou třídu, nemá smysl, aby každý objekt měl svou stejnou kopii.

  Neměnné (`final`)

  je to konstanta, tudíž pomocí `final` zajistíme neměnnost

- Konstanta může být:

  `private`

  dobře možné, když ji nechceme používat mimo třídu

  `public`

  nicméně asi obvyklejší, většinou má širší použití

Příklad konstanty

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

int count = Person.MAX_PEOPLE_COUNT;
```

Všimněte si, že uvádíme i název třídy, ve které je konstanta definovaná `Person.MAX_PEOPLE_COUNT`. Je to sice delší, ale je to tak vlastně dobře: umístění do tříd je zapouzdřením konstanty - hned vidíme, kam patří a co znamená. Můžeme dokonce zkrátit její název `Person.MAX_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
```

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 `enum`

- 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
}
```

Jak to bylo v Pythonu?

- Zde bylo třeba podědit ze třídy `Enum` a ručně ohodnotit jednotlivé prvky výčtu:

```
class State(Enum):
    INIT = 1
    RUNNING = 2
    STOPPED = 3
```

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, 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.

Porovnání `enum`

Lze snadno a bezpečně testovat rovnost pomocí `==`:

- `day == MONDAY`
- `day.equals(MONDAY)` - funguje stejně, pouze selže při `day == null`
- `MONDAY.equals(day)` - bezpečněji, ale stejně lépe psát `day == MONDAY`

Díky tomu, že každý výčet implementuje `Comparable<E>`, lze prvky i řadit. Platí tedy, že `MONDAY` je před `TUESDAY` atd.

Výčty vs. třídy

- Výčtový typ se koncepčně velmi podobá *třídě*, de facto *je to třída*.
- Výčet má však jen *pevně daný počet prvků* (instancí).
- Pro každý prvek daného typu `enum` je získatelné jeho ordinální číslo (pořadí) metodou `int ordinal()`.
- 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.

Výčtový typ s dalšími vlastnostmi

- Využijeme, že `enum` je de facto třída.
- Přidáme *atributy*, případně *metody* nebo i *konstruktor*.
- Dobře se využije možnost definovat jednotlivé prvky výčtu s inicializací jeho atributů.
- Lze i překrýt metody jako `toString` a vracet jiné textové reprezentace
- *Nelze* však překrýt `equals`, porovnání proto zůstává takové, že *x.equals(y)* $<\Rightarrow$ *x* == *y*

Příklad (1)

```
enum OrderStatus {
    // 3 instances and no more ever
    // they might by initialized with parameters
    ORDERED(5), PREPARING(2), READY(0);
    // enum may have attributes (not necessarily final)
    int timeToReady;
    OrderStatus(int timeToReady) {
        this.timeToReady = timeToReady;
    }
    // may override toString
    @Override public String toString() { return "ready in " + timeToReady; }
    // this would not compile! must NOT override equals
```

```
    @Override public boolean equals(Object o) {
        if(o instanceof OrderStatus status)
            return timeToReady == status.timeToReady;
        return false;
    }
}
```

Příklad (2)

```
OrderStatus status = OrderStatus.PREPARING;
System.out.println(status);
// may even directly modify status properties
OrderStatus.PREPARING.timeToReady = 1;
System.out.println(status);
// status is still == (though modified inside)
System.out.println(status == OrderStatus.PREPARING);
```

Výčty mají předem přesně daný počet instancí

- Toto se nezkompiluje, **duplicitní identifikátor** `ORDERED`.
- Že pokaždé jinak inicializovaný, nehraje roli.
- Parametry v závorce nezpůsobí vytvoření nových prvků jako `new`.

```
enum OrderStatus {
  ORDERED(5), ORDERED(15), PREPARING(2), READY(0);
}
```

Prvky jsou uspořádané

- Jsou uspořádané dle pořadí, v jakém jsou uvedeny, aniž bychom cokoli definovali.
- Nelze ovšem psát `OrderStatus.ORDERED < OrderStatus.PREPARING`, ale je nutno využít `compareTo`:
- `OrderStatus.ORDERED.compareTo(OrderStatus.PREPARING)`

Množina prvků výčtu `EnumSet`

- Jelikož prvků výčtu je konečný (a většinou nevelký) počet, je dobré, že pro ně máme speciální typ množiny - `java.util.EnumSet`.
- `EnumSet` je abstraktní podtřídou `AbstractSet` a má dvě neabstraktní implementace:

`RegularEnumSet`
: vnitřně implementován jako bitové pole do velikosti 64 bitů, a to pomocí jednoho `long`

`JumboEnumSet`
: jako bitový vektor libovolné, tzn. i větší velikosti

Příklad EnumSet

```
EnumSet<OrderStatus> set = EnumSet.of(OrderStatus.ORDERED, OrderStatus.READY);
// prints [ready in 5, ready in 0]:
System.out.println(set);
// prints [ready in 2]:
System.out.println(EnumSet.complementOf(set));
// prints class java.util.RegularEnumSet:
System.out.println(set.getClass());
```

- další použitelné (statické) metody EnumSet:
  - `EnumSet.noneOf()` → vytvoří prázdnou množinu ** `EnumSet.range(LOWER, UPPER)` → vytvoří množinu prvků od *LOWER* po *UPPER*

Mapa s klíči prvky výčtu EnumMap

- Z obdobného důvodu se nabízí rovněž mapa `EnumMap`.
- Klíčem jsou prvky výčtu (tzn. jsou pevně předem dány a většinou jich není moc).

```
EnumMap<OrderStatus, Integer> countOrders
  = new EnumMap<OrderStatus, Integer>(OrderStatus.class);
countOrders.put(OrderStatus.ORDERED, 4);
countOrders.put(OrderStatus.READY, 2);
System.out.println(countOrders);
```

Repl.it demo k výčtovým typům

- https://repl.it/@tpitner/PB162-Java-Lecture-03-enum

Další zdroje

- Hezký příklad najdete na The Java™ Tutorials — Enum Types

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 technicky 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!

(Ne)přetěžování

- Sémanticky totéž bez přetěžování: jiný název = ještě lepší
- Převod celého zůstatku jsme napsali jako *nepřetíženou* metodu, která přesně popisuje, co dělá.
- Z názvu metody je zřejmé, co dělá — netřeba ji komentovat!

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

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

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() {
  // calls second constructor
  this("Default name");
}
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`.

Třída `Object`

- I když v Javě vytvoříme prázdnou třídu, obsahuje 'skryté' metody.
- Je to technicky tím, že všechny třídy dědí přímo či nepřímo z třídy Object a odtud ty metody jsou.

```
public class Person { }

Person p = new Person();
p.toString(); // ???
```

- Seznam všech vestavěných metod najdete v javadocu třídy `Object`.

Některé metody třídy `Object`

- `getClass()` — vrátí název třídy
- `equals(⋯)` — porovná objekt s jiným, `hashCode()` — vrátí haš
- `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 - for non-null p:
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:

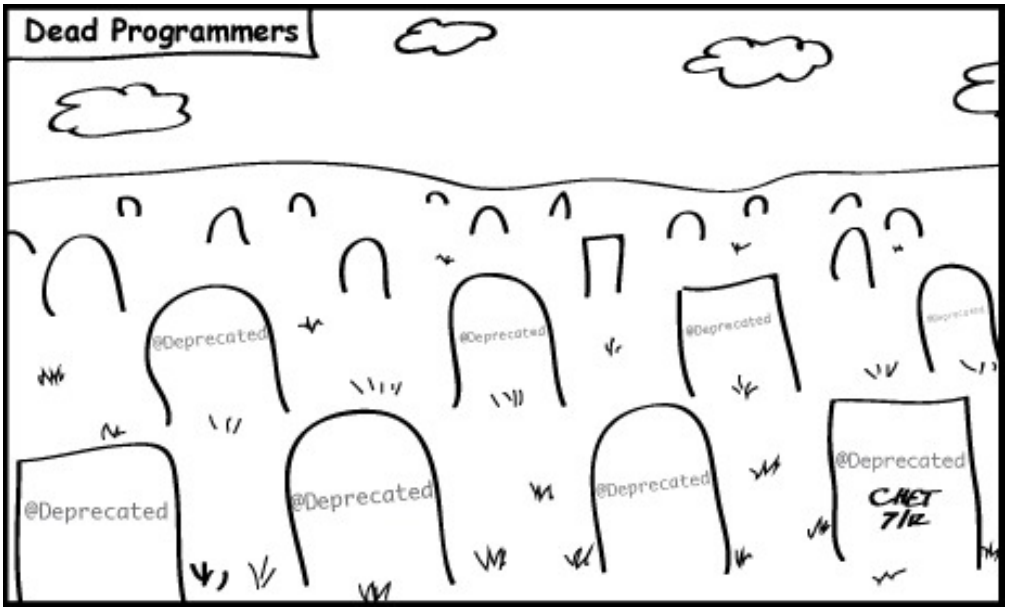
```
@Override
public String toString() {
  return "it works again";
}
```

- 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.



`Objects.toString`

- Více z předchozího, že například ve zřetězení `"Osoba " + p` se kód přeloží skoro jako `"Osoba " + p.toString()` — s výjimkou p == null, kdy vrátí `"Osoba null"`.
- Přímé volání `o.toString()` tedy vyžaduje test na nulovost `o` anebo
- bezpečně a bez testů použít `Objects.toString(o)`,
- případně `Objects.toString(o, "žádná")` — vrátí v případě `o == null` náhradní text `"žádná"`.

Co je životní cyklus?

- Doba mezi vytvořením (new), používáním, nepoužíváním a zánikem objektu (instance).
- Začíná vytvořením pomocí `new`.
- Faktické používání objektu končí ve chvíli, kdy na něj ztratíme odkaz - např. když odkaz je lokální proměnná metody, kterou opustíme.
- Fyzická existenci končí buďto koncem programu nebo zrušením objektu pomocí garbage collectoru, "uklízeče smetí".
- Chování je přibližně stejné jako v Pythonu, ale zcela jiné než v C++ či Rust.

Závislosti mezi objekty

- správě závislostí mezi objekty, tzn. jeden objekt se své činnosti potřebuje další objekt nebo objekty, se věnuje speciální sekce Dependency Injection.

Vyhnout se zbytečným objektům

- *Avoid unnecessary objects*
- Vytváření objektů je "drahá" operace - stojí výpočetní čas i paměť.
- Objekt se musí v řadě případů též dealokovat, zlikvidovat, což stojí další čas.
- V řadě případů není třeba objekt vytvářet, ukážeme si typické situace:
  - řetězce: každé zřetězení `s = s + t` znamená vytvoření nového objektu!
  - úplně stejně `s += t` vede k vytvoření nového objektu
  - `new` namísto zavolání (statické) tovární metody rovněž vždy vytvoří objekt
  - řadu složitějších objektů lze vytvořit jen jednou a znovupoužívat
  - pooling (skladiště) objektů

Metody místo `new`

- Raději `Boolean.valueOf("true")` namísto `new Boolean("true")`, neboť to vytvoří pokaždé (obsahově stejný) objekt `Boolean`
- Obdobně pro další typy - navíc statické metody mohou obecně vrátit i `null`, když se nezdaří.
- U řetezců se mohutně využívá *internalizace* (neplést s internacionalizací), tzn. uložení do poolu v rámci běžící JVM.
- Každý řetězec zadaný jako literál (do uvozovek, třeba `"abcde"`) je internalizován a opakované použití odkazuje na tutéž instanci a nezabírá další paměť.
- Internalizaci lze vyvolat i pomocí metody `s.intern()` - u dlouhých řetězců by sdílení mohlo mít smysl!
- Je to proto, že řetězcové literály - nebo obecněji řetězce, které jsou hodnotami konstantních výrazů (§15.28) - jsou "internalizovány" tak, aby sdílely jedinečné instance pomocí metody String.intern (§12.5).
- blíže viz *Java Language Specification*, 3.10.5. String literals

Předkompilace u regex

- Regulární výrazy jsou silnou a častou používanou technikou, jak nalézat nebo ověřovat vzory v řetězcích.
- Jsou použitelné ve všech běžných jazycích vč. Javy.
- Lze je buďto:
  - rychle napsat, ale pomalu používat: `"řetězec".matches("regex")`
  - zdlouhavěji zapsat a rychle - i opakovaně - vykonávat

```
Pattern pattern = Pattern.compile("regex");
// i opakovaně, bleskově provedeno (násobně rychleji):
```

```
String s = ...
if(pattern.matcher(s).matches()) ...
```

Nepoužívejme `finalize()`

- Finalizér, tj. metoda `finalize()` vlastní všem objektům, může teoreticky být překryta a tím umožněn adekvátní "likvidační" postup při zániku objektu - sestávající obvykle z uvolnění systémových zdrojů - síťové sokety, spojení na databázi atd.
- V Javě nicméně *není zaručeno*, že se finalizér skutečně zavolá - JVM jej nemusí volat, pokud nepotřebuje fyzicky uvolnit paměť obsazenou (již nepoužívaným) objektem.
- I kdyby finalizér zavolán byl, zůstává problém s určením okamžiku, *kdy* je volán a v jakém pořadí jsou finalizéry na mrtvých objektech volány.
- Celkově tedy na `finalize()` nespoléhat a nepoužívat je - zdroje uvolňovat explicitním zavoláním vhodné metody.

Tento tip je javově-specifický, jde o to, jak a kdy pracuje garbage collector při likvidaci nepřístupných ("mrtvých") objektů.

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.

Motivační příklad

- 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 :-)
- Tudíž ani není možné všechny třídy objektů, které o sobě umějí informovat, chápat jako podtřídy jedné třídy "UmímOSoběInformovat". V Javě navíc třída smí mít pouze jednu rodičovskou třídu/předka (=dědit jen z jedné rodičovské), tzn. bychom tím "spotřebovali" možného předka a žádného jiného předka už bychom nikde nemohli mít.

Rozhraní v Pythonu

- Zde máme tzv. *neformální rozhraní* (informal interfaces).
- Jedná se o dynamický jazyk, při překladu nekontroluje, zda jsou všechny metody implementované.
- Napíšeme do docstringů, co má metoda dělat a uvedeme do ní `pass`.
- V podtřídě pak metodu/y překryjeme.

```
class InformalParserInterface:
    def load_data_source(self, path: str, file_name: str) -> str:
        """Load in the file for extracting text."""
        pass

    def extract_text(self, full_file_name: str) -> dict:
        """Extract text from the currently loaded file."""
        pass
```

Příklad jednoduchého rozhraní

- V Javě musíme trochu formálněji.
- 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!

Typ rozhraní namísto třídy

```
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; je z toho pak dobře vidět, že daný objekt používáme jako instanci implementující určité rozhraní a konkrétní třída nás u použití nezajímá.

"Přetypování" obecně

- Java má podobně jako většina jiných jazyků operátor pro "přetypování" (*type cast*).
- Píše se buďto a) `(typ) hodnota` nebo b) `(typ) odkaz_na_objekt` podle toho, co "přetypováváme".
- Jeho fungování se *zásadně* liší podle toho, zda jde o a) o primitivní hodnotu nebo b) odkaz na objekt.

Typová kontrola odkazů na objekty

- U "přetypování" odkazů na objekty se ve skutečnosti nejedná o *změnu obsahu objektu*,
- nýbrž pouze o *potvrzení*, že běhový typ objektu je ten požadovaný, na nějž "přetypováváme".
- Tj. jde o *běhovou typovou kontrolu*.
- Sdělujeme překladači "Hele, já vím, že jde o osobu, tak s tím pracuj jako s osobou".
- Když běhová kontrola případně selže (tedy nikoli v době překladu, ale až při spuštění), je vyvolána výjimka a běh programu přerušen.

Příklad - typová kontrola odkazů

```
Person p = new Person();
// OK, p je třídy Person a ta implementuje Informing.
Informing i = (Informing) p;
// Rovněž OK, protože jakýkoli objekt je
// (aspoň nepřímým) potomkem Object.
Object o = (Object) i;
```

Zjištění typu

- Nejsme-li si typem jisti, otestujeme ho:

```
if(o instanceof Person p) {
  // now we know p is a Person, so we can call walk()
  p.walk();
```

```
}
```

Přetypování u primitivních typů

- U *primitivních typů* se jedná o skutečný převod hodnoty z původního typu na cílový, o *typovou konverzi*.
- Např. `long` přetypujeme na `int` a ořeže se tím rozsah.
- Korektnost a případnou ztrátu informace za nás pohlídá v některých případech překladač, který samozřejmě zná obecné vlastnosti typů. Projeví se zejména při přetypování čísel.

Proměnná deklarovaná jako rozhraní

- Můžeme taky vytvořit proměnnou *deklarovaného* typu rozhraní:

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

Proměnná deklarovaná jako rozhraní

```
Informing i = new Person();
i.retrieveInfo(); // ok
i.emptyMethod(); // cannot be done
```

- Proměnná `i` může používat pouze metody deklarované v rozhraní.
- Nevidíme ostatní metody v třídě `Person`.

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 moci používat také

Anotace `@Override`

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

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

Repl.it demo k rozhraním

- https://repl.it/@tpitner/PB162-Java-Lecture-04-interfaces-Shapes

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.*
- Je to tedy trochu "nečisté", ale je to současně jediná cesta, jak ve stávajícím kódu doplnit metodu do rozhraní, aniž bychom narušili přeložitelnost stávajícího kódu a nemuseli do VŠECH tříd implementujících určité rozhraní dopisovat implementaci této nové metody.
- Mají omezené použití a neměly by se nadužívat.

`static` a `default`

- Existují dva základní typy metod s výkonným kódem v rozhraní:

  statické
  : značíme `static`

  výchozí
  : značíme `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?
}
```

Repl.it demo k výchozím a statickým metodám rozhraní

- https://repl.it/@tpitner/PB162-Java-Lecture-13-intf-default-and-static

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`.

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`.

`WellInforming` jako rozšíření `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

Vícenásobné 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
}
```

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.
- Každopádně se jedná o case-based testování, tj. testování na příkladech.

- Nejde tedy o formální ověření korektnosti (verifikaci).
- 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, kde elementární části jsou *třídy a metody*.
- V Javě se nejčastěji používá volně dostupný balík JUnit.
- V nových produktech se používají verze JUnit 4.x nebo JUnit 5.
- Alternativně lze použít například AssertJ.
- 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.

Repl.it demo k JUnit testování

- https://repl.it/@tpitner/PB162-Java-Lecture-03-JUnit

`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()`
  - a další

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.

Vtip

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

Dědičnost

- Dědičnost je charakteristická vlastnost objektových jazyků se třídami, jako jsou Java, Python, C# a další.
- U beztřídních (klasický JavaScript) se může řešit pomocí *prototypů*.
- Objektové třídy jsou obvykle **podtřídami**, tj. speciálními případy, jiných tříd:

```
class DogKeeper extends Person {
// methods & attributes for DogKeeper
```

```
// in addition to Person
}
```

Co to znamená?

- Všechny objekty typu `DogKeeper` jsou současně typu `Person`.
- Místo jiné osoby lze použít `DogKeeper`.

```
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 vs špatné použití

Správně

Dědičnost by měla splňovat vztah **je** — každý `DogKeeper` *je* `Person`.

Špatně

Každé oddělení má osobu vedoucího; je ale špatně dědit `Department extends Manager` protože neplatí `Department` *je* `Manager`, ale `Department` *má* `Manager`.

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 *Employee* i (přeneseně) z *Person*.

Vícenásobná dědičnost

- Java vícenásobnou dědičnost ve smyslu dědění z více tříd současně **nepodporuje**!
- Důvodem je problém typu diamant:

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

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

- Vícenásobná dědičnost je možná jedině u rozhraní, kde metody nemají definovanou implementaci.

Dědičnost a vlastnosti tříd

- Dědičnost (v kontextu Javy) znamená:
  1. potomek dědí **všechny** vlastnosti nadtřídy (= metody & atributy třídy)
  2. podděděné vlastnosti potomka se **mohou změnit** (např. překrytím metody)
  3. potomek může **přidat** další vlastnosti

Dědičnost vs. rozhraní

Dědičnost

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

Použití rozhraní

- méně závislostí, více případně i opakovaného kódu

- více používané v praxi

Pravidla pro konstruktory podtříd

- Konstruktor musí volat vybraný konstruktor nadtřídy, nebo vlastní přetížený konstruktor (pomocí `this();` s případnými parametry).
- Konstruktor nadtřídy se volá pomocí `super();` s případnými parametry.
- Podobně jako u `this()`, volání `super();` musí být první příkaz a může být pouze jedno.
- V konstruktoru nezle ani zkombinovat `super()` s `this()`.
- Pokud `super()` i `this()` chybí, volá se automaticky bezparametrický konstruktor nadtřídy (tj. vloží se `super()`). Ten ale musí existovat a být neprivátní.
- Obecně platí, že volaný konstruktor musí v nadtřídě existovat.

Příklad s Account

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

Nová třída CheckedAccount

- Rozšíříme třídu `Account` tak, že bude mít minimální zůstatek.

```
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;
  }
  public CheckedAccount(Person owner) {
```

```
    this(owner, 0, 0);
  }
}
```

Volání kódu nadtřídy

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

```
public class CheckedAccount extends Account {
  private int minimalBalance;
  // construktors ...
  @Override
  public boolean debit(int amount) {
    // check min. balance
    if(getBalance() - amount >= minimalBalance) {
       return super.debit(amount); // the debit is inherited
    } else return false;
  }
}
```

- 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!
- Takto lze volat jen bezprostředního předka. Něco jako `super.super.debit(amount)` se syntaktická chyba.

Repl.it demo k dědičnosti - bankovní účty

- https://repl.it/@tpitner/PB162-Java-Lecture-05-inheritance-Account

Pozor na překrývání atributů

- Nepoužívejte chráněné (`protected`) atributy. Je to "bad practice". Používejte zásadně privátní atributy s veřejnými nebo chráněnými gettery a settery.
- **Nikdy nepřekrývejte atributy!** Tj. podtřída nesmí nikdy obsahovat atribut se jménem, jako má některá z jejích nadtříd. Přesto, že je to syntakticky možné.

```
public class CheckedAccount extends Account {
  private int minimalBalance;
  private int balance; // CHYBA - PREKRYTI ATRIBUTU Z NADRIDY
  // construktors ...
  @Override
```

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

Namísto dědění lze použít skládání (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)
}
```

Kompozice pořádně

- Kompozicí se zabývá navazující kurz *PV168 Seminář Javy*.
- Problémy s hierarchiemi dědičnosti pomocí kompozice řeší některé návrhové vzory, např.
  - Bridge: Řešení exploze podtříd rozdělením problémové domény na abstrakci a implementaci
  - Decorator: Variantnost chování je přesunuta z podtříd do spolupráce několika malých objektů za běhu
  - Více v magisterském kurzu *PA103 Object-oriented Methods for Design of Information Systems*

Repl.it demo k dědičnosti - geometrické útvary

- https://repl.it/@tpitner/PB162-Java-Lecture-05-inheritance-Shapes

Základní principy OOP

- Zapouzdření
- Dědičnost
- Polymorfismus

Dědičnost

- *Dědičnost* umožňuje vztah *X is Y* (X *je* Y) mezi objekty, `Human` is `LivingEntity`.
- Umožňuje rozšířit již existující třídu, tedy vytvořit její podtřídu (podtyp), která od svého rodiče zdědí jeho atributy a metody.

Proč?

- Znovupoužití kódu. Nemusíme v každé živé bytosti (člověku, psu…) znovu programovat, co už umí `LivingEntity`.

Polymorfismus

- *Polymorfismus* je schopnost objektu měnit vnímání počas běhu.
- `Object o = new String("String, but can be seen as general Object")`.
- Na místo, kde je očekávána instance třídy `Object`, je možné dosadit instanci jakékoli její podtřídy.

Proč?

- Kde je očekávána živá bytost `LivingEntity`, lze nasadit člověka `Human`.
- V množině živých bytostí můžeme mít současně lidi i psy.

Zapouzdření

- *Zapouzdření* je zabalení dat a metod do jedné komponenty (třídy).
- Zabalená "věc", objekt, nejenže soustřeďuje více položek (dat i metod), ale může některé z nich *navenek skrývat*.
- Princip je, že *Cokoli, co nemusí být viditelné, ani viditelné být nemá.*

Proč?

- Co není vidět, nelze zneužít. Můžeme to později bez újmy odebrat či změnit.

Viditelnost

- Co nemusí být vidět, ať vidět není.

- Daná věc (atribut, metoda) v objektu je, ale ne všichni ji vidí.
- Použití *tříd* i jejich metod a atributů lze regulovat (uvedením tzv. *modifikátoru přístupu*).
- Nastavením správné viditelnosti jsme schopni docílit skutečného 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* či *lokální v balíku* (package-private, package-local)
- `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`

Polymorfismus

Obecně máme několik typů polymorfismu (Polymorfismus (Wikipedia)):

- Ad-hoc polymorfismus
- Polymorfismus přetěžování operátorů
- Parametrický polymorfismus
- Podtypový polymorfismus

Ad-hoc polymorfismus

- objektům odvozeným z *různých tříd* volat tutéž metodu se stejným významem
- v Javě realizovatelné pomocí rozhraní
- např. třídy `Dog` i `Car` mohou implementovat rozhraní `Registered`

Polymorfismus přetěžování operátorů

- provedení rozdílné operace v závislosti na typu operandů
- například `+` se může chovat jinak na čísla, řetězce nebo matice
- v Javě není možné definovat vlastní přetěžování operátorů
- nicméně polymorfní operátory existují, typicky `+` (čísla, řetězce)
- nebo operátor podmíněného výrazu: `cond ? expr1 : expr2`
- ale například v C++ lze dodefinovat chování operátoru `+` tak, že bude sčítat matice

Parametrický polymorfismus

- V Javě realizován *parametrickými typy*.
- Například kolekce jako `List` nebo `Set` jsou realizovány jako *generické typy* (generics).
- Tyto se dají typově parametrizovat a tak mít například seznam osob `List<Person>`.

Podtypový polymorfismus

- umožněn děděním mezi objektovými třídami
- například `Employee`, `Manager` nebo `Student` jsou všechno osoby `Person` a mají tedy jméno `getName()`

Nevhodně realizovaný polymorfismus

- Typicky metoda, která si poradí se vstupem různého typu.
- Příklad `getPerimeter(Shape shape)` fungující pro různé (pod)typy `Shape`
- Nicméně má to jen nevýhody; lepší by bylo, kdyby každý typ měl svou nestatickou metodu `getPerimeter`

Příklad pseudo-polymorfismu

```
---
public static double getPerimeter(Shape shape) throws IllegalArgumentException {
    if (shape instanceof Rectangle r) {
        return 2 * r.length() + 2 * r.width();
    } else if (shape instanceof Circle c) {
        return 2 * c.radius() * Math.PI;
    } else {
        throw new IllegalArgumentException("Unrecognized shape");
    }
}
---
```

Polymorfismus pomocí vzorů ve výrazu `switch`

- Podrobněji v Pattern Matching for switch Expressions and Statements
- V Javě 17+ to polymorfní větvení lze napsat přece jen elegantněji: výrazem `switch` se vzory

```
---
public static double getPerimeter(Shape shape) throws IllegalArgumentException {
    return switch (shape) {
```

```
        case Rectangle r -> 2 * r.length() + 2 * r.width();
        case Circle c    -> 2 * c.radius() * Math.PI;
        default          -> throw new IllegalArgumentException("Unrecognized shape");
    };
}
---
```

Polymorfismus pomocí vzorů v příkazu `switch`

- Varianta téhož s pomocí *příkazu* `switch` a nikoli výrazu:

```
---
public static double getPerimeter(Shape shape) throws IllegalArgumentException {
    switch (shape) {
        case Rectangle r -> return 2 * r.length() + 2 * r.width();
        case Circle c    -> return 2 * c.radius() * Math.PI;
        default          -> throw new IllegalArgumentException("Unrecognized
shape");
    };
}
---
```

Typová bezpečnost vzorů ve `switch`

- V obou variantách - výraz i příkaz switch se vzory - musí být vzory ve `switch` *exhaustivní*,
- tzn. musí pokrývat všechny typy řídicího výrazu.
- Jednoduše řešitelné pomocí `default` na konci.

Dobré praktiky vytváření objektů

- Zvažte *statickou tovární metodu* místo konstruktorů.
- Uvažujte o "budovateli" (*builder*) namísto konstruktoru s mnoha parametry.
- Vynuťte *singleton* pomocí soukromých konstruktorů (nebo `enum`).
- Zabraňte přímé instanciaci pomocí soukromého konstruktoru.
- Upřednostněte vložení závislostí (*dependency injection*) před pevným propojením.
- Vyhněte se zbytečným objektům.
- Eliminujte zastaralé odkazy na objekty.
- Vyhněte se finalizátorům (metodám `finalize()`) a čističům.

Tovární metoda namísto konstruktoru

- *Consider static factory method instead of constructors*
- Josh Bloch: "Java efektivně, Tip 1: Používejme statickou tovární ("výrobní") metodu namísto přímého volání konstruktoru (evt. s parametry)"
- Takových metod může být *více a mohou se jmenovat různě* (výstižněji) - viz `Person createMale()`
- *Nemusí vrátit* objekt za všech okolností, při všech možných voláních s různými parametry - někdy mohou na rozdíl od konstruktoru vrátit `null`
- Metoda může vrátit již *existující instanci* - to je klíčová výhoda, např. u singletonu nebo u skladiště (poolu) objektů - skladiště objektů šetří čas a výkon, singleton je bezpečný

Tovární metoda: variabilita vracených typů

- Metoda nemusí vrátit objekt přesně stejného typu, jako je deklarován, může vrátit i objekt podtřídy, potomka - např. metoda `static Person createEmployee(…)` vrátí objekt třídy `Employee`, jež je podtřídou `Person`
- Dokonce se skutečný vrácený typ může lišit dle předaných parametrů: `static Person createEmployee(boolean manager)` může dle vrátit `Manager` nebo `Employee`
- Tovární metoda někdy může vracet objekt třídy neznámé v době překladu, tzn. objekt pomocí reflexe dynamicky zavede - tato technika umožňuje modularitu a doplňování kódu a funkcionality dokonce za běhu

U statických továrních metod se držte běžného očekávaného pojmenování, například: `newPerson`, `createPerson`, `Person.create`, pro přístup k skladišti nebo singletonu třeba `Person.getInstance()` - u *get* nikdo intuitivně nečeká, že se určitě bude tvořit nový objekt.

Namísto složitého konstruktoru použijme vzor "Výrobce"

- *Consider a builder instead of constructor with many parameters*
- Někdy máme složitější objekty s mnoha atributy
- Konstrukce pak obnáší volat konstruktor s mnoha parametry
- Příklad `new Person("Jan Novák", true, 22000)`, kde `true` značí, že jde o muže, a `22000` je plat.
- V Javě jsou parametry poziční a nepojmenované (na rozdíl od novějšího Kotlinu např.), je pak problém vidět, co kde nastavujeme -
- zejména když jsou parametry stejného typu - `new Line(0.0, 0.0, 1.0, 2.0)` znamená přesně co?
- Pro tyto případy se namísto složitého konstruktoru hodí tzv. builder ("budovatel").
- V moderních prostředích (IDEA, ale i VS Code s pluginy) vidíme, jaký parametr je na které pozici v závorce.

Příklad "budovatel"

```
public class PersonBuilder {
    private String name;
    //... male, salary
    public void setName(String name) {...}
    public void setGender(boolean male) {...}
    public void setSalary(double salary) {...}
    public Person getPerson() {
        // it can perform necessary checks
        // and refuse to create unless fulfilled
        return new Person(name, male, salary);
    }
}
```

Použití "budovatele"

```
PersonBuilder builder = new PersonBuilder();
builder.setName("Jan Novák");
builder.setGender(true);
builder.setSalary(22000.0);
// person can immediately be used
Person person = builder.getPerson();
```

Protipříklad - nepoužití "budovatele"

s - Nemáme budovatele, máme jen cílovou třídu objektů.

```
public class Person {
    private String name;
    //... male, salary
    public void setName(String name) {
        // set name here
    }
    public void setGender(boolean male) {...}
    public void setSalary(double salary) {...}
    public Person() {}
}
```

Bez použití "budovatele"

- Je to špatně, objekt může bez kontroly zůstat neúplný.

```
Person person = new Person();
// now the person is NOT complete
// it is dangerous to use, cannot be used
person.setName("Jan Novák");
person.setGender(true);
person.setSalary(22000.0);
// now it is OK
```

Privátní konstruktory

- Ve výše uvedených tipech s tovární metodou i výrobcem jsme zamlčeli podstatnou věc:
- uvedené mechanismy vytvoření objektu se daly obejít jeho přímou konstrukcí `new Person(⋯)`.
- Aby toto nebylo možné, můžeme všechny konstruktory označit jako `private`
- Pak zvenčí nelze instanci vytvořit pomocí `new Person(⋯)`, ale musí např. být statická tovární metoda `Person.newInstance()` nebo budovatel `Builder builder = Person.builder()`.

Privátní nebo chráněné konstruktory?

- Rozdíl mezi `private` a `protected`
- Kdybychom konstruktory označili jako `private`, ale některé přesto ponechali `protected`, dovolíme tím *dědění* třídy, např. z `Person` můžeme podědit do `Employee`.
- Kdyby úplně všechny byly `private`, máme smůlu a z `Person` nikdy nic nepodědíme.
- Je to proto, že každá podtřída musí mít konstruktor, který jako první příkaz volá konstruktor předka - a ten kdyby byl soukromý, nebylo by možné.

Singletony

- Jsou klasickým návrhovým vzorem popsaným jinde.
- V souvislosti s konstrukcí platí, že objekt singletonu se konstruuje buďto předem nebo až je potřeba
- Nesmí být zkonstruovatelný jinak, např. přímo přes `new` (konstruktorem)
- *Vynucení singletonu pomocí privátních konstruktorů (nebo enum)* nebo *Prevent instantiation by private constructor* - když jsou konstruktory privátní, zvenčí je nelze volat a klient musí využít singleton, třeba `MyClass.getInstance()` nebo tovární metodu.

Příklad singletonu

```
public class DataManager {
    private static DataManager manager;
    private DataManager() {
```

```
        //... initialize data manager
    }
    public static DataManager getInstance() {
        if(manager == null) manager = new DataManager();
        return manager;
    }
}
```

Vyhněte se zbytečným objektům

- Špatně je toto, s každým průchodem je alokován nový řetězec.
- Nepomohlo by ani `s += String.valueOf(i) + " ";`
- Řešitelné pomocí `StringBuilder`, tam realokace nebude.

```
String s = "";
for(int i = 0; i < 100; i++) {
    // each time a new string is created
    // ... and the old forgotten
    s = s + String.valueOf(i) + " ";
}
```

Odstranění zastaralých odkazů na objekty

statické objekty

se dealokují až zcela na konci běhu programu - patří totiž celé třídě a ta se na rozdíl od objektu "nezapomíná"

vyrovnávací paměti

tím, že objekty se v cache pamatují a je to očekávané chování, musíme rozmyslet, kdy už je potřebovat nebudeme a odkazy smazat, nastavit `null`

datové struktury

například odkazy z pole jsou pořád "živé", dokud je "živé" celé pole

zabalení/vybalení z obálky

typicky čísla `new Integer(42)` vytvoří nový objekt

Vyhněte se finalizátorům a čističům

- Finalizér, tzn. metoda `finalize()` mají všechny objekty,
- může teoreticky být překryta a tím umožněn adekvátní "likvidační" postup při zániku objektu - sestávající obvykle z uvolnění systémových zdrojů - síťové sokety, spojení na databázi atd.
- V Javě nicméně není zaručeno, že se finalizér skutečně zavolá - JVM jej nemusí volat, pokud

nepotřebuje fyzicky uvolnit paměť obsazenou již nepoužívaným objektem.

- I kdyby finalizér zavolán byl, zůstává problém s určením okamžiku, *kdy* je volán a v jakém pořadí jsou finalizéry na mrtvých objektech volány.
- Celkově tedy na `finalize()` nespoléhat a nepoužívat je - zdroje uvolňovat explicitním zavoláním vhodné metody.

Vtip

> Dědičnost. Nejlepší objektově-orientovaný způsob, jak se pohádat a naštvat ty před námi i po nás.

Přínosy dědičnosti

- Je fajn, *ušetříme psaní* kódu i jeho *objem* (kód je tam jednou), omezí se redundance.
- Někdy dokonce i *předejdeme chybám* tím, že chybu opravíme na jednom místě a OK, protože efekt se hned bez dalšího promítne do všech podtříd = je poděděn.
- Můžeme využít *polymorfismus*, kdy jsme schopni jednotně obsloužit více typů objektů
- například `Employee`, `Manager` nebo `Student` jsou všechno osoby `Person` a mají tedy jméno `getName()`

Hlavní omezení

- Dědičnost je příklad *silné (těsné, pevné) závislosti*.
- Změna v rodičovské třídě (nadtřídě) má silný vliv na funkci, případně i kompilovatelnosti podtříd.
- Metodu, která se dá překrýt v podtřídě, nesmíme volat z konstruktoru!
- Obecně: z konstruktoru pokud možno nevolat nic.

Opakování

- Pole je homogenní datová struktura - všechny prvky stejného typu.
- V objektovém pojetí je možný polymorfismus - pole `Person[]` může obsahovat prvky `Person` i `Manager`, když je to podtřída `Person`: `Manager extends Person`.
- 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]);
```

- *typ* může být primitivní 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 a
- 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/odkaz 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.
- Metoda `copyOf` bere dva parametry — původní pole a počet prvků, kolik se má nakopírovat.

```
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
```

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
```

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.

Pomocné metody pro kontrolu indexů do pole

- Třída `java.util.Objects` nabízí pro práci s poli statické metody na kontrolu, zda se index nebo rozmezí indexů "vejde" do velikosti pole

```
public static int checkIndex(int index, int length)

public static int checkFromToIndex(int fromIndex, int toIndex, int length)

public static int checkFromIndexSize(int fromIndex, int size, int length)
```

Repl.it demo k polím

- https://repl.it/@tpitner/PB162-Java-Lecture-03-arrays

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` a nové pole vrátí

`void fill(long[] a, long val)`
: naplní pole `a` hodnotami `val`

Metody třídy `Arrays` II

`boolean equals(long[] a, long[] a2)`
: vrátí `true` právě když 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 a pořadí (uspořádání)

- Obecně rozlišujeme, zda chceme zjišťovat *shodnost* (*rovnost*, *ekvivalenci*):
  - mezi dvěma *primitivními hodnotami*

    - mezi dvěma *objekty*
- U *primitivních hodnot* jsou rovnost i uspořádání 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 (celočíselných) typů funguje bez potíží.
- U čísel floating-point (`double`, `float`) je třeba porovnávat s určitou tolerancí.
- U FP navíc existují hodnoty jako `+0.0` vedle `-0.0` a tyto by měly být rovny.

```
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
```

Porovnávání metodami třídy `Objects`

- `java.util.Objects` je poměrně nová utilitní třída (obsahující jen statické metody) od Javy 8
- mimo jiné obsahuje metody `Objects.equals(o1, o2)` na porovnávání objektů
- i ve variantě *deep* - hluboké porovnání - pro struktury

```
public static boolean equals(Object a, Object b)

public static boolean deepEquals(Object a, Object b)
```

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` při obsahové ekvivalenci objektů, což musí být explicitně nadefinované

Příklad `==`

```
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 = jedná se o jeden objekt = odkazy obsahují stejnou adresu objektu.

Porovnávání objektů dle obsahu

- 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:

Metoda `equals`

```
@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`.

Příklad s `equals`: 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;
  }
}
```

Mírně odlišné `equals`

```
public class ComplexNumber {
  private int real, imag;
  public ComplexNumber(int r, int i) {
    real = r; imag = i;
  }
  @Override
  public boolean equals(Object o) {
    // instanceof type pattern, we'll see later
    if (o instanceof ComplexNumber that)
      return this.real == that.real
        && this.imag == that.imag;
    else return false;
  }
}
```

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;
  }
}
```

Technické zjednodušení `instanceof`

- Počínaje Javou 14 lze využít tzv. *instanceof pattern*.
- Odkaz na objekt se kromě běhové typové kontroly rovnou přiřadí do `person`.

```
if (o instanceof Person person) {
  return this.name.equals(person.name)
    && this.yearBorn == person.yearBorn;
} else {
  return false;
}
```

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

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

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 ani 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
```

```java
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`

Repl.it demo k porovnávání primitivních hodnot a objektů

- https://repl.it/@tpitner/PB162-Java-Lecture-06-vs-equals

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.
- Někdy se místo *Abstract* používá *Base*: `SearcherBase`, aby se zdůraznilo, že abstraktní třída je základem odvozených tříd konkrétních.
- 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(...);
```

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 though it will be implemented later
   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;
   }
}
```

Šablonové metody

- Všimněte si, že ve třídě `AbstractSearcher` volá metoda `contains` abstraktní metodu `indexOf`, která na této úrovni ještě neexistuje.
- Je to v pořádku, protože u kompletní třídy (viz `LinearSearcher` dále) již požadovaný kód musí být.
- Jedná se o návrhový vzor *Template Method (šablonová metoda)*, kdy kód třídy spoléhá na to, že kód šablonové metody dodají až podtřídy.

Repl.it demo k abstraktním třídám

- https://repl.it/@tpitner/PB162-Java-Lecture-06-abstract-classes

Struktura a zavádění javových programů

- struktura (třídy, balíky, moduly)
- zavádění (classloaders)
- další zdroje (resources)
- reflexe

Struktura a organizace programů

- Javový program sestává z alespoň jedné *třídy* s alespoň jednou *metodou*.
- Má-li být spustitelný z příkazové řádky, pak s metodou `main`.
- Ve spoustě nasazení - webové aplikace, aplikační kontejnery - není `main` třeba .
- Třídy se obvykle sdružují do *balíků* - deklarace `package xx.yy.zz` třeba `cz.muni.fi`.

Moduly

- Nově od Java 9 jsou balíky - přinejmenším v Core API - sdruženy do modulů (Modules).
- Moduly kromě tříd samotných mohou obsahovat další zdroje (*resources*) a obsahují popisovač (*module descriptor file*).
- Cílem je lepší znovupoužitelnost a potřeba u rozsáhlých programů lépe organizovat kód.
- V praxi se kromě organizace ve standardních knihovnách (Java Core API) moc nevyužívají.

- A Guide to Java 9 Modularity
- Java 9 Modules - Tutorial
- Java Modules

Účel modulů

- zabalit celou aplikaci nebo API jako samostatný modul
- umožnit sledování a zajištění závislostí mezi moduly
- napodobit "dohnat" to, co už jazyky jako JavaScript nebo Python mají
- technicky jde o soubor JAR s celým modulem, tzn. popisovačem, třídami, příp. dalšími zdroji
- dříve bylo třeba umísťovat zdroje do kořenové úrovně projektu a ručně je spravovat
- nyní lze na jemnější úrovni modulů - podle toho, který zdroj je kde potřeba

Popisovač modulu

Název

název modulu (pojmenování modulů podobně jako u balíků, tj. tečky jsou povoleny, pomlčky nikoli; časté je pojmenování ve stylu projektu (`my.module`) nebo ve stylu Reverse-DNS jako u balíků (`com.baeldung.mymodule`)

Závislosti

seznam dalších modulů, na kterých tento modul závisí

Veřejné balíky

seznam všech balíků, které chceme mít přístupné mimo modul (ve výchozím nastavení jsou všechny balíky soukromé)

Nabízené služby

můžeme poskytovat implementace služeb (*services*), které mohou být využívány jinými moduly

Využívané služby

seznam služeb využívaných tímto modulem

Povolení pro reflexi

explicitně povoluje ostatním třídám používat reflexi pro přístup k soukromým (`private`) prvkům balíku (ve výchozím nastavení jsou všechny třídy nepřístupné pro reflexi)

Hlavní deklarace v popisovači

Export z modulu

- veřejné balíky, viditelné zvnějšku

```
module com.jenkov.mymodule {
    exports com.jenkov.mymodule;
    exports com.jenkov.mymodule.util;
```

```
}
```

- je třeba exportovat skutečně každý balík, který chceme zviditelnit ven - tzn. i ten `···util`

Závislosti

- závislost na ostatních modulech

```
module com.jenkov.mymodule {
    requires com.jenkov.yourmodule;
    requires com.jenkov.yourmodule.util;
}
```

Cyklické závislosti nejsou přípustné: `module.a` závisí na `module.b`, který závisí na `module.a` NELZE.

Balíky ve více modulech

- NE, jeden balík smí být současně exportován v max jednom modulu.
- V jedné aplikaci se nesmí potkat balík exportovaný současně dvěma moduly, dvěma cestami
- a to ani v případě, že by ve skutečnosti jedna třída byla vždy jen v jednom modulu.

Nepovolený export

Ani uvedené dva exporty nejsou možné:

```
module cz.muni.fi.pb162.mod1 {
    exports cz.muni.fi.pb162.bank;
}
```

- balík `cz.muni.fi.pb162.bank` v tomto modulu obsahuje jen třídu `Bank`

```
module cz.muni.fi.pb162.mod2 {
    exports cz.muni.fi.pb162.bank;
}
```

- balík `cz.muni.fi.pb162.bank` v tomto modulu obsahuje jen třídu `Account`

Práce s moduly

- kompilace: `javac -d out --module-source-path src/main/java --module com.jenkov.mymodule`

- spuštění aplikace v modulu: `java --module-path out --module com.jenkov.mymodule/com.jenkov.mymodule.Main`
- vytvoření JAR archívu s modulem: `jar -c --file=out-jar/com-jenkov-mymodule.jar -C out/com.jenkov.mymodule .` (`-c` *create*, `--file` soubor s výsledným JAR, `-C` *change directory* změna adresáře před sbalením JARu)
- spuštění z JAR: `java --module-path out-jar -m com.jenkov.mymodule/com.jenkov.mymodule.Main`
- spuštění hlavní třídy (Main-Class) z JAR: `java -jar out-jar/com-jenkov-javafx.jar`

Balení aplikací se závislostmi

- Moduly umožňují pohodlně vytvořit spustitelný JAR se všemi závislostmi bez ručního jednotlivého "přibalování"
- Nástroj **jlink**
- `jlink --module-path "out;C:\Program Files\Java\jdk-9.0.4\jmods" --add-modules com.jenkov.mymodule --output out-standalone`

  `--module-path`

  : složka `jmods` obsahuje standardní Core API moduly a `out` je složka s našimi předkompilovanými moduly

  `--add-modules`

  : které moduly se mají do aplikace sbalit, v tomto případě `com.jenkov.mymodule`

  `--output`

  : cílová složka pro umístění JAR s aplikací

- Spuštění výsledného JAR: `java --module com.jenkov.mymodule/com.jenkov.mymodule.Main`

Kontejnery (dynamické datové struktury)

Co jsou kontejnery, kolekce (`Collection`)?

- *Dynamické datové struktury* vhodné k ukládání proměnného počtu objektů (přesněji odkazů na objekty).
- Primitivní hodnoty se do základních dynamických struktur neukládají přímo, ale prostřednictvím objektových obálek (wrappers).
- Jsou automaticky ukládané v operační paměti (ne na disku).

Proč je používat?

- pro uchování **proměnného počtu** objektů (počet prvků se může měnit — zvyšovat, snižovat)
- oproti polím nabízejí efektivnější algoritmy přístupu k prvkům
- přístupem rozumíme *vložení*, *smazání*, či *nalezení* prvku

Kvalitní seznámení s kontejnery najdete na stránkach Oracle

Připomenutí polí v Pythonu

- Pole je v Pythonu použitelné pro kompaktní ukládání primitivních hodnot (čísla, znaky...), ale ne objektů

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

Připomenutí struktur v Pythonu

Seznamy

`my_list = ["a", "b", "mpilgrim", "z", "example"]` modifikovatelné

N-tice

`my_tuple = ("a", "b", "mpilgrim", "z", "example")` nemodifikovatelné

Množiny

`my_set = {'a', False, 'b', True, 'mpilgrim', 42}` modifikovatelné

Slovníky

asociativní pole, `my_dict = {'server': 'db.diveintopython3.org', 'database': 'mysql'}` modifikovatelné

Typ prvků nás nezajímá, mohou být namíchané.

Připomenutí seznamů v Pythonu

- Datové typy v Pythonu

```
# create a 5-item list
a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
# take 3rd item
m = a_list[2] # 'mpilgrim'
# take last item (at length-1)
m = a_list[-1] # 'example'
# make a slice (výřez)
s = a_list[:2] # ['a', 'b', 'mpilgrim']
```

Jak to bude v Javě?

- Principiálně podobně, v detailu odlišně
- Seznam vytvoříme jako jiné objekty pomocí `new`
- Kapacita vytvořeného seznamu není omezená
- Pak teprve lze přidávat prvky
- Neměnné seznamy lze i `List.of("Jedna", "Dva")`
- Přístup k prvku metodou: `myList.get(2)` vrátí 3. prvek
- Nelze použít závorky pro index: `myList[2]`
- Obdobně u množin a asociativních polí, tedy slovníků, map

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

Ke všem těmti rozhraním Java nabízí hotové implementace (i několik), případně si implementujeme vlastní.

Typové parametry

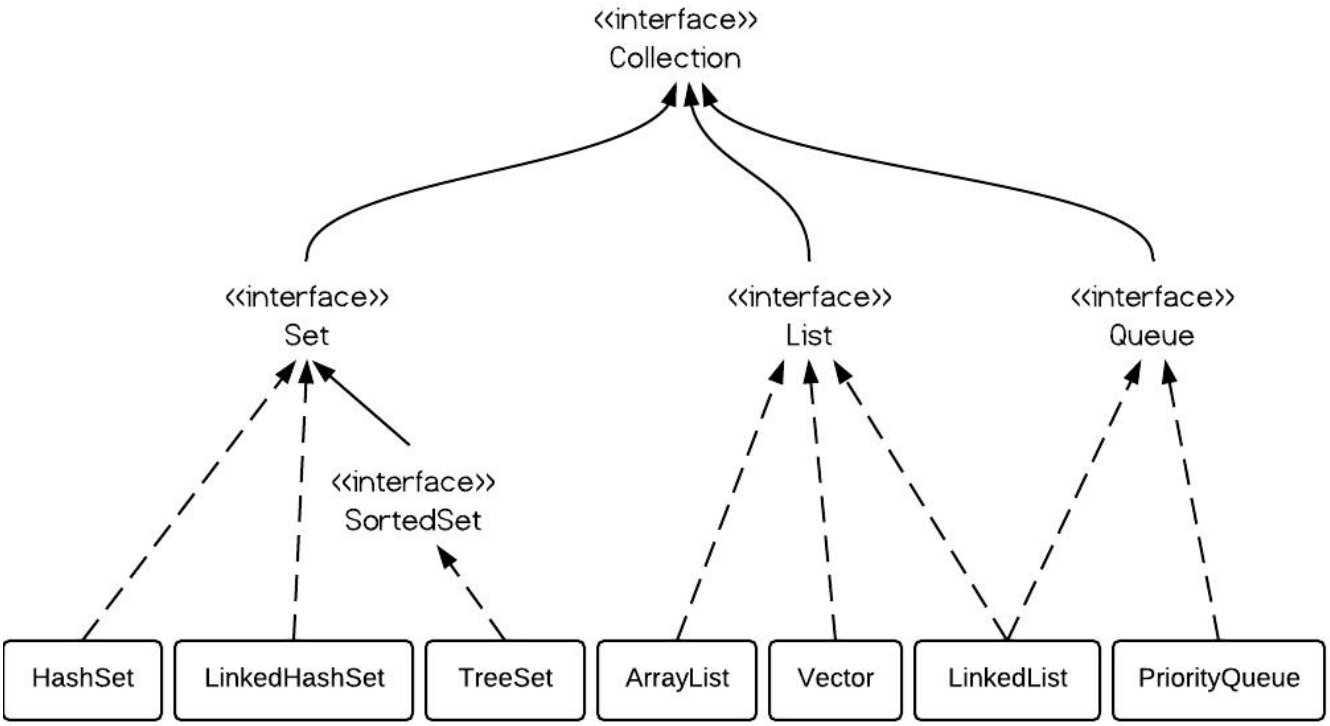
- V Javě byly dříve kontejnery koncipovány jako *beztypové*,
- do jednoho bylo možné ukládat prvky různých typů - lidi, řetězce, cokoli.
- Nyní mají *typové parametry* ve špičatých závorkách (např. `Set<Person>`), které
- určují, jaký typ položek se do kontejneru smí dostat.

```
List objects; // without type, old, BAD
List<String> strings; // with type String - OK
```

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

Typová hierarchie kolekcí

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



Rozhraní `Collection`, `List`, `Set`

`Collection`
: je nejobecnější rozhraní, zahrnuje cokoli, co shromažďuje jednotlivé prvky; dokumentace API: `Collection`

`List`, `Set`
: specifická rozšíření rozhraní `Collection`

ne však `Map` - asociativní pole není `Collection`!

Třídy implementující tato rozhraní

- `ArrayList`, `LinkedList`, `HashSet`, ...

Příklady

```
List<String> listOfStrings = new ArrayList<>();
// new empty set of strings
Collection<String> collection = new HashSet<>();
// new unmodifiable list of 2 strings
```

```
List<String> listWithValues = List.of("so", "cool");
```

Metody přidání a zjištění obsahu

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` je-li kolekce prázdná (velikost je 0)

Příklady

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

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

Metody odebrání a dotazu na přítomnost

`void clear()`
: odstraní všechny prvky z kolekce

`boolean contains(Object o)`
: `true`, právě když se `o` v kolekci nachází; na test rovnosti se použije `equals`

`boolean remove(Object o)`
: odstraní prvek z kolekce; vrátí `true`, byl-li prvek odstraněn

Příklady

```
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(); // [] empty list
```

Iterátor z kolekce

`Iterator<E> iterator()`

metoda každé kolekce; vrací něco, přes co se dá kolekce iterovat (procházet for-each cyklem)

- Jedná se o použití návrhového vzoru Iterator:
- Kolekce nenabízí přímo metody pro procházení. Místo toho vrací objekt (iterátor), který umožňuje danou kolekci procházet jednotným způsobem bez ohledu na to, jak je kolekce uvnitř implementovaná.
- Je možná i varianta, kdy procházení je umožněno jak specifickými metodami, tak iterátorem. Příkladem je `List`, který nabízí metodu `get(int i)` pro přímé procházení i obecný iterátor z důvodu kompatibility s ostatními kolekcemi.

Převod kolekce na pole

`T[] toArray(T[] a)`

vrátí z kolekce pole typu `T`, tj. je to konverze kolekce na pole

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

Metody pro hromadné manipulace

`boolean containsAll(Collection<?> c)`

`true` právě když 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 objektů jakéhokoliv typu.

Příklad metod `Collection` V

```
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 is ["A", "B"]

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

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

Repl.it demo ke kontejnerům

- https://repl.it/@tpitner/PB162-Java-Lecture-07-collections

Potomci `Collection`

Podívejme se na potomky rozhraní `Collection`, konkrétně:

Rozhraní `List`

má implementace

- `ArrayList` - seznam na bázi pole, rychlý při přímém přístupu, nejběžnější
- `LinkedList` - spojovaný seznam, méně používaný

Rozhraní `Set`

má implementace

- `HashSet` - množina na bázi hašovací tabulky, nejčastěji používaná

Seznam `List`

- něco jako dynamické pole
- každý uložený prvek má svou pozici — **číselný index**
- index je celočíselný, nezáporný, typu `int`
- umožňuje procházení seznamu dopředně i zpětně - indexem či iterátorem

- lze pracovat i s *podseznamy*, něco jako řezy (slices) v Pythonu:

```
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
- rychlý přístup k prvkům dle indexu
- pomalé operace přidávání a odebírání prvků blíže k začátku seznamu (pole, v němž je seznam, se musí realokovat)

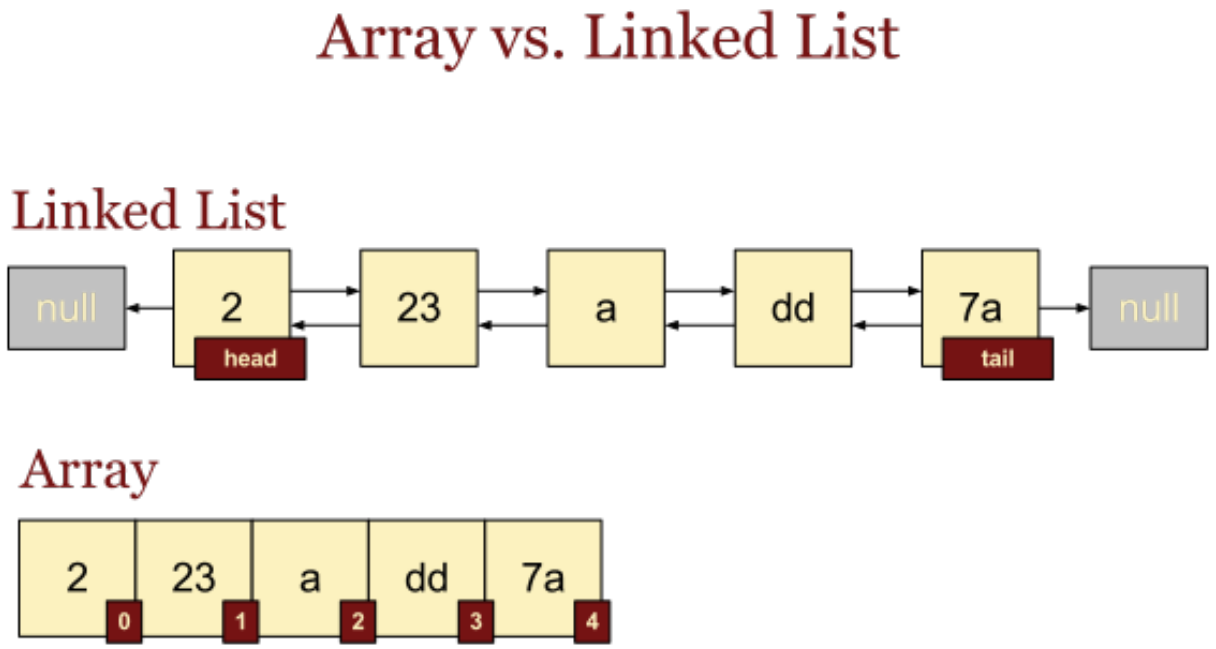
Javadoc třídy `ArrayList`

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

```
List implementation:                        | ArrayList | 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
```

Nevhodné případy užití tedy jsou `ArrayList` pro *přidávání/odebírání zkraje* seznamu a pro dlouhé seznamy `LinkedList`jakékoli přístupy *uprostřed*.

Konstruktory seznamů

`new ArrayList<>()`

vytvoří prázdný seznam (s kapacitou 10 prvků)

`new ArrayList<>(int initialCapacity)`

vytvoří prázdný seznam s danou kapacitou

`new 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 přes metody `of`

Kromě `new` a konstruktoru lze přes statické tovární metody, kolekce jsou ale pak neměnné:

`List.of(elem1, elem2, ⋯)`

- vytvoří seznam a naplní ho danými prvky
- vrátí **nemodifikovatelnou** kolekci

`Set.of`, `Map.of`

analogicky

Vytváření kolekcí přes `new`

Chceme-li kolekci modifikovatelnou, musíme vytvořit novou `new`:

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

Jelikož v množině nejsou duplicity, `Set.of` si hlídá, abychom prvek nepřidali vícekrát: proto selže například `Set.of("foo", "bar", "baz", "foo");`

Na zamyšlení

Jak udělám ze seznamu typu `List` kolekci `Collection`?

- v podstatě nedělám nic - seznam už *je* kolekcí

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

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

- musíme vytvořit nový seznam a prvky tam zkopírovat

```
// 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)`

totéž, ale vrátí 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), nejhůře 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 obsahově stejné (porovnání atributů).

`hashCode`

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

Co je haš?

jakési *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] [0,1]
2    | [1,1] [0,2] [2,0]
3    | [2,1] [1,2] [0,3] [3,0]
```

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.
- Chtěná složitost O(1), O(log n) zdegeneruje na lineární *O(n)*.

Co když mají stejné objekty různé haše?

- Porušíme kontrakt (předpis) metody `hashCode`.
- Množina `HashSet` přestane fungovat, bude obsahovat duplicity nebo vložený prvek už nenajdeme!

Speciální množina `LinkedHashSet`

- Další implementací množiny je `LinkedHashSet` = HashSet + LinkedList.
- Zachová pořadí prvků dle jejich vkládání, což jinak u `HashSet` neplatí.

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 - vkládání i odběr z obou stran

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 používame jejich objektové protějšky — `Integer, Char, Boolean, Double`...
- Java automaticky dělá *zabalení*, tzv. **autoboxing** — konverzi primitivního typu na objekt "wrapper".

- Pro zpětnou konverzi se analogicky dělá tzv. **unboxing**, *vybalení*.

```
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, ale flexibilnější
- modifikace povolena, např. mazání prvku pod iterátoem

Lambda výrazy s `forEach`

- například `list.forEach(System.out::println)`

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>`

Příklad s `while`

- 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 iterátorem 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.

Repl.it demo k iterátorům (a komparátorům z příští přednášky)

- https://repl.it/@tpitner/PB162-Java-Lecture-08-iterators-and-comparator

Kontejnery kontejnerů

- Kromě kontejnerů obsahujících již koncové hodnoty, jsou často používané i kontejnery obsahující další kontejnery:
- Například `List<Set<Integer>>` bude obsahovat seznam množin celých čísel, tedy třeba `[{1, 4, -2}, {0}, {-1, 3}]`.
- Nebo mapa, která studentům přiřazuje seznam známek `Map<Student, List<Integer>>`.

Repl.it demo ke kontejnerům kontejnerů

- https://repl.it/@tpitner/PB162-Java-Lecture-08-map-of-lists

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>`, nemá jednotlivé prvky
- 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 dvojici s klíčem `key`, vrací hodnotu (nebo `null`)

`V replace(K key, V value)`

nahradí existující klíč hodnotou

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, kde haš je zahašovaný klíč, hodnota tabulky je dvojice *(klíč, hodnota)*.
- Složitost základních operací:
  - v praxi závisí na kvalitě hašovací funkce (metody `hashCode`) na ukládáných objektech,
  - teoreticky se blíží složitosti *konstantní*, O(1).
- Klíče nejsou uspořádané, nelze iterovat v pořadí klíčů.
- Uspořádané mapy jsou TreeMap, viz dále.

Kolekce `HashSet` je implementována pomocí `HashMap` — klíč je prvek, hodnota je "dummy object".

Javadoc třídy `HashMap` = Uspořádané kolekce :course: PB112 :year: 2024 :term: jaro :javaversion: 19 :description: Lecture slides for PB112 course taught at Masaryk

University, Faculty of Informatics typically for non-IT/dev students since 2023 :Author: Tomáš Pitner, Radek Ošlejšek :copyright: © 2024 Tomáš Pitner, Radek Ošlejšek, Masaryk University — PB112 Java :slideshowlocation: Masaryk University, Brno, Czech Republic :date: jaro2024 :data-uri: :keywords: Java, object programming :email: pitner@muni.cz :source-highlighter: pygments

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!

`Comparable` jako příklad funkcionálního rozhraní

- Comparable<T> je hezký typický příklad tzv. *funkcionálního rozhraní* (functional interface); má jedinou metodu `compareTo` a lze použít například jako predikát pro filtrování objektů v proudech.

`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:

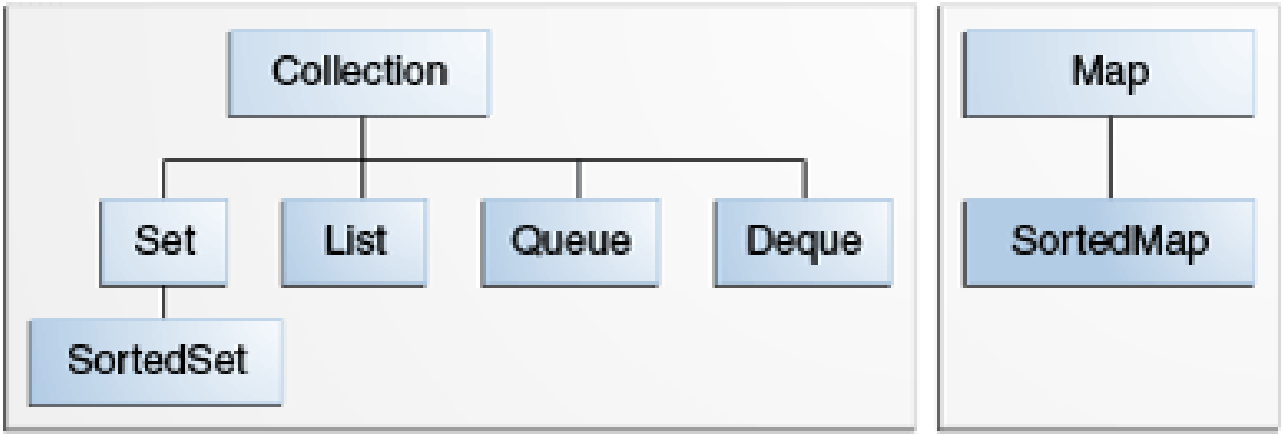
```
public class IgnoreCaseComparator implements Comparator<String> {
  public int compare(String o1, String o2) {
    return o1.toLowerCase().compareTo(o2.toLowerCase());
  }
}
...
new IgnoreCaseComparator().compare("HI", "hi"); // 0
```

Skutečné použití

- metody pro uspořádání programátor v kódu obvykle nepoužívá
- namísto toho používá **uspořádané kolekce**, kdy prvky v kolekci jsou řazeny automaticky
- je nutno definovat přirozené uspořádání nebo použít komparátor, aby kolekce věděla, podle jakých pravidel prvky seřadit
- komparátor se nastavuje při vytváření uspořádané kolekce nebo mapy (viz dále)
- Jedná se o použití návrhové vzoru Strategy:
  - Komparátor implementuje strategii třídění na daném typu objektů. Přitom můžeme mít víc strategií (víc způsobů třídění).
  - Uspořádaná kolekce nebo mapa pak představuje `Context` ve vzoru, kdy říkáme, jaká strategie třídění se má v daném případě použít.

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}
```

Repl.it demo k uspořádaným množinám a mapám

- https://repl.it/@tpitner/PB162-Java-Lecture-08-sorted-set-and-map

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<>();
// add items to list ...
List<String> syncedList = Collections.synchronizedList(list);
// now syncedList is a synchronized view of list
```

Získání nemodifikovatelných kontejnerů

- kontejnery jsou standardně modifikovatelné (read/write)
- tzn. mají *fungující* metody `add`, `addAll`, `remove`, `clear`…
- někdy však nechceme mít kontejner modifikovatelný - například když metodou `get` poskytujeme privátní atribut

Pak využijeme nemodifikovatelné kontejnery vyrobitelné statickými metodami třídy `Collections`:

- `unmodifiableSet, unmodifiableList, unmodifiableCollection, ⋯`
- metoda vrátí nemodifikovatelný pohled (view) na původní kolekci

Vytvoření přes `Collections.unmodifiableSet`

`Collections.unmodifiableSet(old)`

- vytvoří nemodifikovatelné view
- skrze view však nelze původní kontejner modifikovat (a pokus vede k výjimce)
- avšak změny přímo v původním kontejneru se promítnou i do tohoto view(!)
- původní kontejner smí obsahovat `null`

Vytvoření přes `new HashSet`

`new HashSet<>(old)`

- vytvoří kopii dat v původním kontejneru v daný okamžik
- tuto kopii lze modifikovat, což ale (samozřejmě) nemá žádný vliv na původní objekt - ten je zcela oddělený
- původní kontejner smí obsahovat `null`

Vytvoření přes `Set.copyOf`

`Set.copyOf(old)`

- vytvoří kopii dat v původním kontejneru v daný okamžik
- tuto kopii nelze modifikovat (a pokus povede k výjimce)
- nemodifikovatelnost je mj. z důvodu konzistence s chováním metod `Set.of`
- původní kontejner nemůže obsahovat `null`

Typické použití pro "getter"

```
private Set<String> presidents = new HashSet<>();
```

```
public Set<String> getPresidents() {
   return Collections.unmodifiableSet(set);
}
```

Technicky mají i nemodifikovatelné kontejnery metody `add`, `addAll`,... Ovšem tyto nefungují, nedají se ani zavolat (vyhodily by `UnsupportedOperationException`).

Návrhový vzor *Proxy*

- Při vytváření nemodifikovatelné kolekce se nikam nic nekopíruje.
- Původní kolekci se pouze předřadí objekt (kolekce), který nemodifikační metody přeposílá původní kolekci, zatímco modifikační metody selhávají s výjimkou
- Jedná se použití návrhového vzoru Proxy:
  - Původní i nový objekt jsou stejného typu (v našem případě `Collection`).
  - Nový objekt na pozadí komunikuje s původním objektem.

Prázdné nemodifikovatelné 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`)

O co jde?

- Java byla navržena jako čistě objektový jazyk.
- Lambda výrazy do tohoto jazyka přináší prvky funkcionálního programování.
- Jak mohou tyto dva různé světy koexistovat?
- Proč a jak je funkcionální syntaxe integrována do objektového API Javy?

Motivační příklad

Založeno na Java Tutorial

- Předpokládejme, že chceme vytvořit sociální síť. Administrátorům chceme umožnit provádění různých akcí jako například zasílání zpráv těm uživatelům sociální sítě, kteří splňují nějaká kritéria.
- Předpokládejme, že uživatelé jsou reprezentování následující třídou:

```
public class Person {
    public enum Sex {
        MALE, FEMALE
    }
    private String name;
    private LocalDate birthday;
    private Sex gender;
    private String emailAddress;

    public int getAge() {
        // ...
    }
    public void printPerson() {
        // ...
    }
}
```

Krok 1: Primitivní řešení

- Dále předpokládejme, že jsou uživatelé sociální sítě uložení v seznamu `List<Person>`.
- Základní implementace vyhledávání lidí podle kritéria může vypadat následovně:

```
public static void printPersonsOlderThan(List<Person> roster, int age) {
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

Otázky

Q

Co když chceme přidat další operaci, např. výpis lidí mladších než nějaký věk?

A

Musí se přidat nová metoda, nebo se existující metoda musí pojmout víc obecněji.

Krok 2: Zobecněné vyhledávání

```
public static void printPersonsWithinAgeRange(List<Person> roster, int low, int high)
{
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

Otázky

Q

Co když chceme vypisovat uživatele specifického pohlaví, nebo dokonce kombinace věk a pohlaví?

Q

Co když se rozhodneme změnit třídu `Person` a přidat do ní atributy, např. vzájemné vztahy nebo geografická lokace?

A

Přestože je tato metoda obecnější než předchozí `printPersonsOlderThan`, snaha vytvořit specifickou metodu pro každý možný vyhledávací dotaz je neudržitelná.

Řešení

Oddělit kód, který specifikuje vyhledávací kritéria, od samotného vyhledávání.

Jména metod musí být výstižná a popisná. Proto byla metoda přejmenována.

Krok 3: Použití vlastního rozhraní [1/2]

- Definujeme rozhraní pro vyhledávací a vytvoříme implementaci:

```
interface CheckPerson {
    boolean test(Person p);
}

class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
```

```
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

Ve stejnou chvíli můžeme mít definováno několik vyhledávacích kritérií (tříd implementujících rozhraní).

Krok 3: Použití vlastního rozhraní [2/2]

- Vyhledávací metoda se změní následovně:

```
public static void printPersons(List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

A volá se takto:

```
List<Person> roster = ...
printPersons(roster, new CheckPersonEligibleForSelectiveService());
```

Q

Je nutné definovat `CheckPersonEligibleForSelectiveService` ve speciální třídě?

A

Není. Můžeme použít anonymní třídy a redukovat tak kód.

Krok 4: Použití anonymní třídy

```
interface CheckPerson {
    boolean test(Person p);
}
// separate iteration and tester
public static void printPersons(List<Person> roster, CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
```

```
}
```

Provedení filtrace

```
// do the filtering
List<Person> roster = ...
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

- Všimněte si, že `CheckPerson` je funkcionální - obsahuje jedinou metodu. Proto je ale název metody nepodstatný.

Q

Mohli bychom název metody nějak vynechat?

A

Ano, pokud použijeme **lambda výraz**, který se dá chápat jako definice **anonymní metody** implementující nějaké **funkcionální rozhraní**.

Krok 5: Použití lambda výrazu

```
List<Person> roster = ...
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Vysvětlení lambda syntaxe →

- Levá část (před šipkou) obsahuje vstupní parametry, pravá strana pak kód případně výstupní hodnotu.
- Všimněte si, že rozhraní `CheckPerson` se teď v kódu objevuje pouze typ argumentu v metodě `printPerson()`.

- Ale název typu (ani metody, jak už víme) není důležitý.

Q

Mohli bychom vynechat z kódu i typ `CheckPerson`?

A

Ano. Java nabízí pro takové případy vlastní generická předdefinovaná rozhraní.

Krok 6: Použití existujících funkcionálních rozhraní [1/3]

- Zamysleme se nad původní definicí našeho rozhraní:

```
interface CheckPerson {
    boolean test(Person p);
}
```

Zobecnění pomocí generických typů

- Jeho význam můžeme zobecnit pomocí generických typů podobně, jako je tomu u rozhraní `java.util.functions.Predicate`:

```
interface Predicate<T> {
    boolean test(T t);
}
```

Krok 6: Použití existujících funkcionálních rozhraní [2/3]

- `CheckPerson` tedy již nadále nepotřebujeme. Můžeme místo něj použít `Predicate<T>`:

```
public static void printPersons(List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

Vlastní filtrace zůstává

```
List<Person> roster = ...
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Krok 6: Použití existujících funkcionálních rozhraní [3/3]

- Zamysleme se nad rozhraním `Predicate<T>` a jeho použitím ještě jednou:

```
interface Predicate<T> {
    boolean test(T t);
}
...
(Person p) -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

Typování `Predicate<T>`

- Všimněte si, že rozhraní `Predicate` používá typ `T` pouze na zjištění typu vstupního argumentu metody.
- V řadě případů lze typ argumentu odvodit z kontextu pomocí *type inference*, pak je tato informace nadbytečná, ale nevadí.

Q

Mohli bychom deklaraci typu při volání vynechat?

A

Ano. Následující fragment kódu je rovněž správně. Že `T` odpovídá `Person` zjistí Java při překladu.

```
p -> p.getGender() == Person.Sex.MALE
    && p.getAge() >= 18
    && p.getAge() <= 25
```

Jaká máme funkcionální rozhraní v Javě?

- Rozhraní v `java.util.function` package, například

  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)`.

- Další rozhraní, která sice mohou definovat více metod, ale jen jedna zni je nestatická, například
  - `Comparator<T>` z `java.util`,
  - `Iterable<T>` z `java.lang`.

Příklad použití dalších funkcionálních rozhraní

- Naše současná implementace vyhledávací metody vypisuje informace o osobách splňujících predikát:

```
public static void printPersons(List<Person> roster, Predicate<Person> tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}
```

- Co když ale s osobami, které splňují predikát daný parametrem `tester`, chceme dělat něco jiného, než jen vypisovat informace o nich?

Funkcionální rozhraní *Consumer*

- Funkcionální rozhraní `Consumer<T>` s metodou `void accept(T t)` nabízí obecnou operaci na zpracování objektu.
- Jako implementaci rozhraní `Consumer<T>` lze použít jakýkoliv kód, který na daném objektu něco vykoná, ale nic nevrací.
- Velmi často se prostě zavolá bezparametrická metoda definovaná na daném objektu, v našem případě `p.printPerson()`.

Příklad definice zpracování s `accept`

```
public static void processPersons(
        List<Person> roster,
        Predicate<Person> tester,
        Consumer<Person> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            block.accept(p);
        }
    }
}
```

Příklad zpracování

```
processPersons(
     roster,
     p -> p.getGender() == Person.Sex.MALE
         && p.getAge() >= 18
         && p.getAge() <= 25,
     p -> p.printPerson()
);
```

Funkcionální rozhraní *Function*

- Co když nám nestačí zpracovat původní objekty `Person`, ale rádi bychom z nich vytáhli nějaké informace, například e-mail, a teprve tyto informace zpracovali?
- K tomu potřebuje rozhraní, které by vracelo nějakou hodnotu.
- Funkcionální rozhraní `Function<T,R>` nabízí metodu `R apply(T t)`, která slouží k "transformaci" dat typu `T` na data typu `R`.

Příklad definice zpracování s `apply` a `accept`

```
public static void processPersons(
        List<Person> roster,
        Predicate<Person> tester,
        Function<Person, String> mapper,
        Consumer<String> block) {
    for (Person p : roster) {
        if (tester.test(p)) {
            String data = mapper.apply(p);
            block.accept(data);
        }
```

```
    }
}
```

Příklad zpracování

```
processPersons(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Extenzivní využití generických typů [1/2]

- Metoda `processPersons` sice pracuje pouze se dvěma typy (`Person` and `String`), ale její smysl se dá zobecnit takto: procházej objekty, vyber, které splňují predikát, vezmi z nich nějaká data, a tato data zpracuj.
- Tohoto zobecnění lze dosáhnout zobecněním typů za použití generik:

```
public static <X, Y> void processElements(
        Iterable<X> source,
        Predicate<X> tester,
        Function <X, Y> mapper,
        Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}
```

Extenzivní využití generických typů [2/2]

- Vypsání e-mailových adres lidí pak lze vypsat stejně jako předtím:

```
processElements(roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
```

```
    email -> System.out.println(email)
);
```

- Co se ale změnilo je to, že lze takto zpracovat i jiné třídy/objekty, než jen `Person`!

Datové proudy (streams)

- Rozhraní `java.util.stream.Stream<T>` používá nastíněné principy a nabízí jednoduché metody, které mohou být použité pro proudové zpracování dat. Náš kód lze přepsat takto:

```
roster.stream()
    .filter(
        p -> p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25)
    .map(p -> p.getEmailAddress())
    .forEach(email -> System.out.println(email));
```

Shrnutí: Principy proudového zpracování dat

- Java Core API nabízí jednoduchá funkcionální rozhraní vhodná pro proudové zpracování dat.
- Díky generickým typům jsou tato rozhraní jsou nezávislá na tom, jaká data jsou v proudu uložena.
- Rozhraní `Stream` poskytuje metody, které využívají funkcionální rozhraní pro definici jednoduchých operací nad proudovými daty.
- Operace je aplikována na všechny objekty proudu.
- Většina těchto metod vrací "výsledný" proud jako výstupní hodnotu, takže lze operace snadno řetězit.

Shrnutí/2

- Vývojáři mohou využít anonymní třídy pro snadnou definici proudových operací (implementaci požadovaných rozhraní).
- Protože se ale jedná o funkcionální rozhraní, lze navíc použít kompaktní zápis pomocí lambda výrazů.
- Resumé: Vývojáři mohou implementovat proudově-orientované zpracování dat s použitím přístupu podobného funkcionálním jazykům.

Kolekce typu `Stream`

Nyní, když známe pozadí lambda výrazů a jejich souvislost s funkcionálními rozhraními v Java Core API, zaměříme na jejich použití pro proudové zpracování dat zajišťované rozhraním `java.util.stream.Stream`.

- Stream homogenní lineární struktura prvků (např. seznam)
- Rozhraní `Stream<T>` obsahuje dva 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`)

Motivační 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 tu metodu:

`String::length`
: Zjištění délky řetězce, totéž co lambda výraz `s -> s.length()`

`System.out::println`
: Vypsání prvku, totéž co lambda výraz `s -> System.out.println(s)`:

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)`

Mapovací funkce `mapToInt, mapToLong, mapToDouble`

- vracejí speciální `IntStream, ..`
- obsahuje další funkce — např. `average()`

Terminální metody třídy `Stream` I

- Terminální anebo ukončovací metody.
- Bývají agregačního charakteru (sum, count…).

`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

Příklady terminace do kolekce s `Collectors`

```
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());
}
```

```
Set<Person> owners = cars.stream()
          .map(Car::getOwner)
          .collect(Collectors.toSet());
```

`x → x.getOwner()` se dá zkrátit na `Car::getOwner`

Cyklus s podmínkou na proud s filtrem

```
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());
```

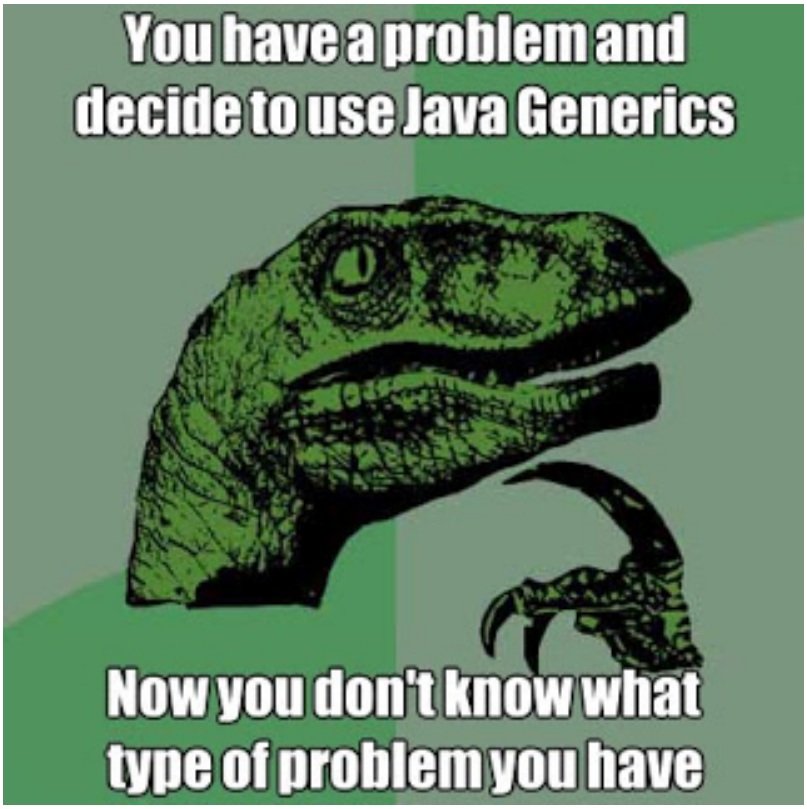
Další zdroje

- Benjamin Winterberg: Java 8 Stream Tutorial
- Amit Phaltankar: Understanding Java 8 Streams API
- Oracle Java Documentation: Lambda Expressions

Generické typy

- Generické typy = něco *obecně použitelného*, *zobecnění*
- Třídy v Javě mají společného předka, třídu `Object`.
- Potřebujeme-li pracovat s nějakými objekty, o kterých neznáme typ, můžeme využít společného předka a pracovat s ním.
- Například do jednoho seznamu prvků `Person` vložíme prvky `Employee` i `Manager` současně.
- Synonymum: generické typy = generika.

Vtip



Deklarace seznamu vs generika

```
// 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 označení typů
- písmeno vystihuje použití — `T` je **t**ype, `E` je **e**lement
- `E` nahradíme jakoukoliv třídou nebo rozhraním

Příklad seznamu lidí

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

List<Person> people = new ArrayList<>();
// all items are people:
people.add(employee1);
people.add(manager1);
```

```
people.add(person1);
```

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

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.

Repl.it demo k výjimkám - základy

- https://repl.it/@tpitner/PB162-Java-Lecture-09-exceptions

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();
}
```

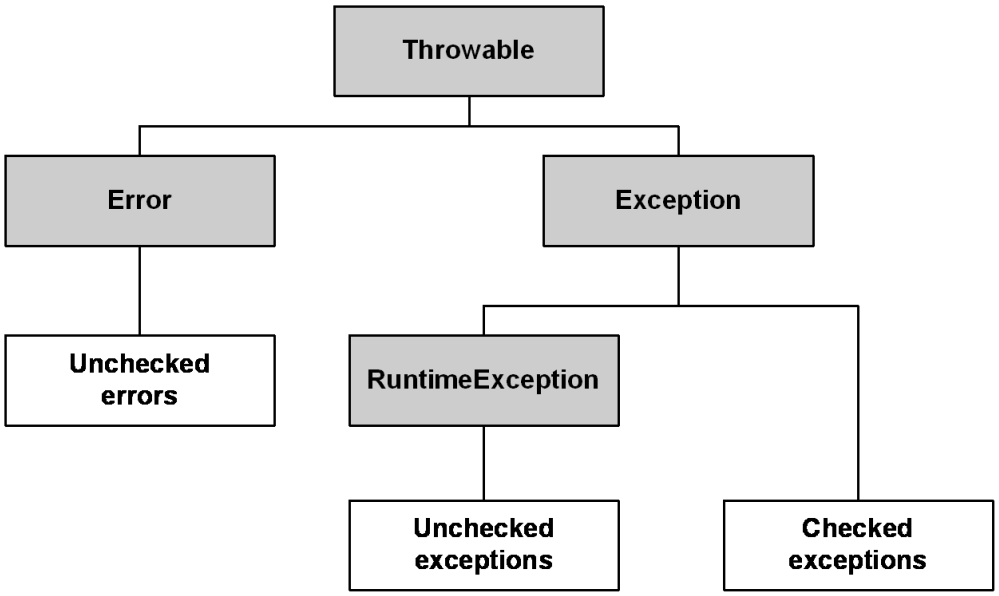
Na zamyšlení

- 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.

```java
// Will not compile
public void someMethod1() throws IOException { }
```

Metody propouštějící výjimku II

Pokud `throws` u **hlídaných výjimek** chybí, program se nezkompiluje:

```java
// 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

```java
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

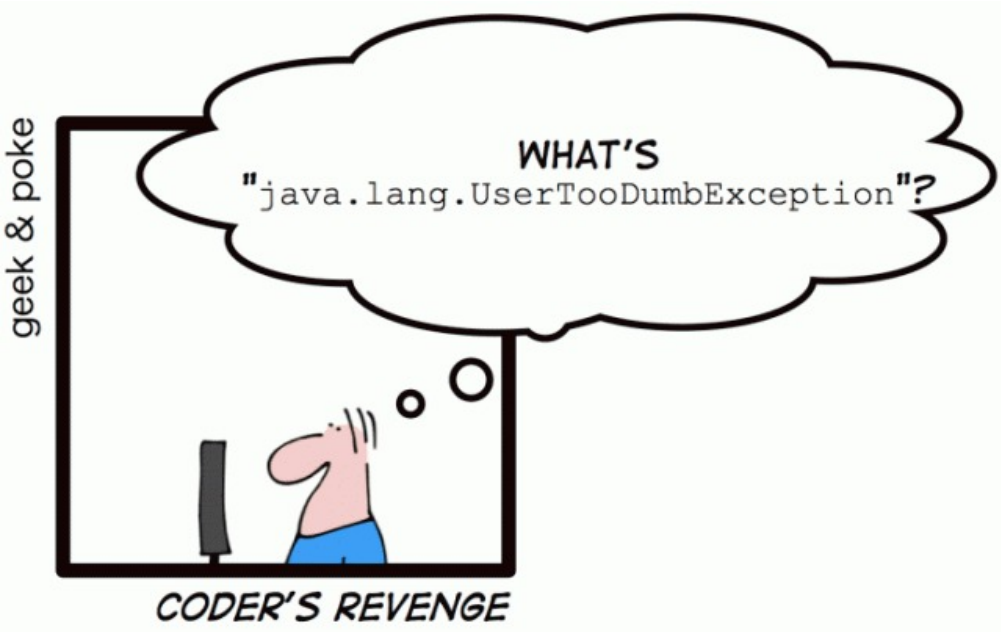
- u hlídaných výjimek **musí** být `throws`
- u nehlídaných výjimek **může** být `throws`

```java
// 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");
}
```

Repl.it demo k vlastním výjimkám

- https://repl.it/@tpitner/PB162-Java-Lecture-10-own-exceptions

Závislosti mezi objekty

Vložení závislostí

- *Dependency Injection* (DI) je postup, kdy deklarativním způsobem (přímo uvnitř javového kódu nebo vnějším popisovačem, např. XML) označíme, kde je očekáváno na začátku životního cyklu objektu vložení odkazu na objekt, na němž náš závisí - tedy *vložení závislosti*.

Příklad bez DI

- Příklad použití závislosti:
- objekt manažeru dat závisí na objektu databáze = potřebuje ho pro svou činnost (metodu `getData`);
- objekt databáze se do něj dostane už při *konstrukci* = na začátku životního cyklu manažeru;
- objekt `database` může nebo nemusí (časteji) být po dobu života manažeru vyměněn.
- Výhodné je, když je `Database` rozhraní - může být implementováno různými třídami.

```
public class DataManager {
    private Database database;
    public DataManager(Database db) { this.database = db; }
    public Data getData() { return database.getData(); }
}
```

Variantně bez DI

- Nastavení závislosti metodou `set`
- Nevýhodou je, že mezi konstrukcí a nastavením závislosti je objekt manažeru očividně nepoužitelný.

```
public class DataManager {
    private Database database;
    public DataManager() { }
    public void setDatabase(Database db) { this.database = db; }
    public Data getData() { return database.getData(); }
}
```

Životní cyklus bez DI

- Nastavení závislosti musíme provést sami.
- Nejdříve vytvoř prvý objekt - databáze, poté druhý objekt (manažer dat) a propoj závislost.

```
public static void main(String[] args) {
    // database is hardwired here - only MyDatabase is used & no other
    Database db = new MyDatabase(...);
    DataManager manager = new DataManager();
    manager.setDatabase(db);
    // now the manager is ready to give data
    Data data = manager.getData();
}
```

EJ: Tip XXX: Prefer dependency injection over hardwiring dependencies

- Existují nástroje, které zvnejšku zajistí, že do objektu je přidán vhodný objekt závislosti.
- Například do manažeru dat je přidána správná databáze = vhodný objekt implementující rozhraní `Database`.
- Vnější vložení závislosti je posláním rámců (kontejnerů) pro vkládání závislostí, *dependency injection frameworks*.
- Velmi jednoduchý rámec pro DI je Google Guice
- Populární je *Spring Framework*, ale ten je významně obsáhlejší a kvůli samotnému DI je kanónem na vrabce.

Příklad s Guice

- Příklad viz https://github.com/google/guice/wiki/Motivation
- Třída `RealBillingService` závisí na `CreditCardProcessor` a `TransactionLog`

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;

  @Inject
  public RealBillingService(CreditCardProcessor processor,
      TransactionLog transactionLog) {
    this.processor = processor;
    this.transactionLog = transactionLog;
  }

  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
     } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

Konfigurace DI v Guice

- Náš Module musí implementovat rozhraní Guice `Module`
- V konfiguraci přiřadíme každému rozhraní (vlevo) třídu (vpravo), kterou v jeho roli budeme používat.

```
public class BillingModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(TransactionLog.class).to(DatabaseTransactionLog.class);
    bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
    bind(BillingService.class).to(RealBillingService.class);
  }
}
```

Vlastní "sdrátování" DI v Guice

```
public static void main(String[] args) {
    Injector injector = Guice.createInjector(new BillingModule());
    BillingService billingService = injector.getInstance(BillingService.class);
    ... // the TransactionLog and CreditCardProcessor follow
  }
```

Avoid unnecessary objects

Eliminate obsolete object references

Avoid finalizers and cleaners

- Finalizér, tzn. metoda `finalize()` vlastní všem objektům, může teoreticky být překryta a tím umožněn adekvátní "likvidační" postup při zániku objektu - sestávající obvykle z uvolnění systémových zdrojů - síťové sokety, spojení na databázi atd.
- V Javě nicméně není zaručeno, že se finalizér skutečně zavolá - JVM jej nemusí volat, pokud nepotřebuje fyzicky uvolnit paměť obsazenou (již nepoužívaným) objektem.
- I kdyby finalizér zavolán byl, zůstává problém s určením okamžiku, *kdy* je volán a v jakém pořadí jsou finalizéry na mrtvých objektech volány.
- Celkově tedy na `finalize()` nespoléhat a nepoužívat je - zdroje uvolňovat explicitním zavoláním vhodné metody.

Vnořené, vnitřní a anonymní třídy

Vnořené

definované uvnitř jiné třídy jako její prvek - podobně jako metody nebo atributy

Lokální

definované uvnitř metody podobně jako lokální proměnná

Vnitřní

jako vnořené, ale potřebují ke své instanciaci *objekt* vnější třídy, v níž jsou definovány `Inner inner = outer.new Inner()`

Anonymní

vnitřní, ovšem *nepojmenované*, pojmenuje si je překladač sám stylem `Person$1`

Vnořená třída

- definované uvnitř jiné třídy jako její prvek - podobně jako metody nebo atributy
- je uvozené klíčovým slovem `static`, pak se jedná o *statickou vnořenou třídu*
- chybí-li static, jedná se o *nestatickou vnořenou třídu* označovanou jako *vnitřní třída* (inner class) viz dále

Statická vnořená třída

- Použití statické vnořené třídy je dobrá praktika - zapouzdření.
- Lze ji mít jako `private`, pak není vidět zvenčí.
- Přesto je použitelná v rámci atributů metod obklopující vnější třídy.
- I opačně - statická vnořená třída může používat *i privátní metody a atributy* své vnější třídy.

Příklad statické vnořené

```
public class Enclosing {
    private static int x = 1;
    public static class StaticNested {
        private void run() { ...}
    }
    public void test() {
        // StaticNested is of course visibile here -
        // even if it were private
        StaticNested nested = new StaticNested();
        nested.run();
    }
}
class Another {
    public void test() {
        // works since both Enclosing & StaticNested are visible here
        Enclosing.StaticNested nested = new Enclosing.StaticNested();
        nested.run();
    }
}
```

blíže viz Java Nested Classes

Příklad statické vnořené `Map.Entry`

```
public class MapDemo {
    public static void main(String[] args) {
```

```
        Map<String, String> map = new HashMap<>();
        for(Map.Entry<String, String> entry: map.entries()) {
            // do something for each (key, value) pair:
            // entry.getKey(), entry.getValue()
            // or entry.setValue(...)
        }
    }
}
```

blíže viz Interface Map.Entry<K,V>

Vnitřní

- *Vnitřní* = nestatická vnořená třída
- Objekt takové třídy potřebuje ke své instanciaci objekt své vnější třídy.
- `Inner inner = outer.new Inner()`

Příklad vnitřní

```
public class KarelGame {
    // will be visible from Karel, too
    private static enum Heading { LEFT, RIGHT }
    private int[] field = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    // this Karel is bound to a KarelGame
    private Karel karel = new Karel();
    // inner class
    public class Karel {
        private int position = 0;
        public void put() {
            // the field in KarelGame is accessible from Karel's method
            // even though it is private
            if(field[position] < MAX_ITEMS) field[position]++;
```

Anonymní

- vnitřní, ovšem nepojmenované
- na jednom místě se definuje a zároveň i jednou použije (instanciuje)
- většinou jako implementace rozhraní nebo abstraktní třídy

Příklad anonymní

Koncepce vstupně/výstupních operací v Javě

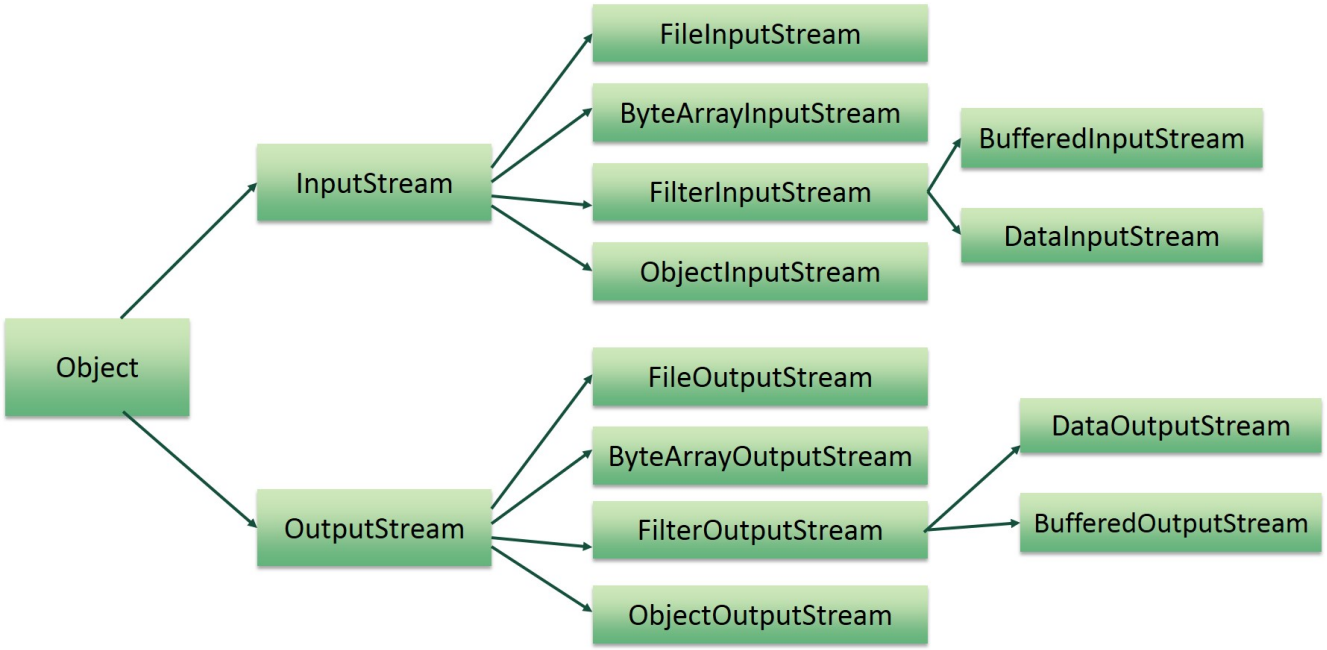
- V/V operace jsou založeny na *vstupně/výstupních proudech* (streams).
- Tím pádem je možno značnou část logiky programu psát nezávisle na tom, o který *konkrétní typ* V/V zařízení jde.
- Současně s tím jsou díky tomu V/V operace plně *platformově nezávislé*.

Table 2. Vstupně/výstupní proudy

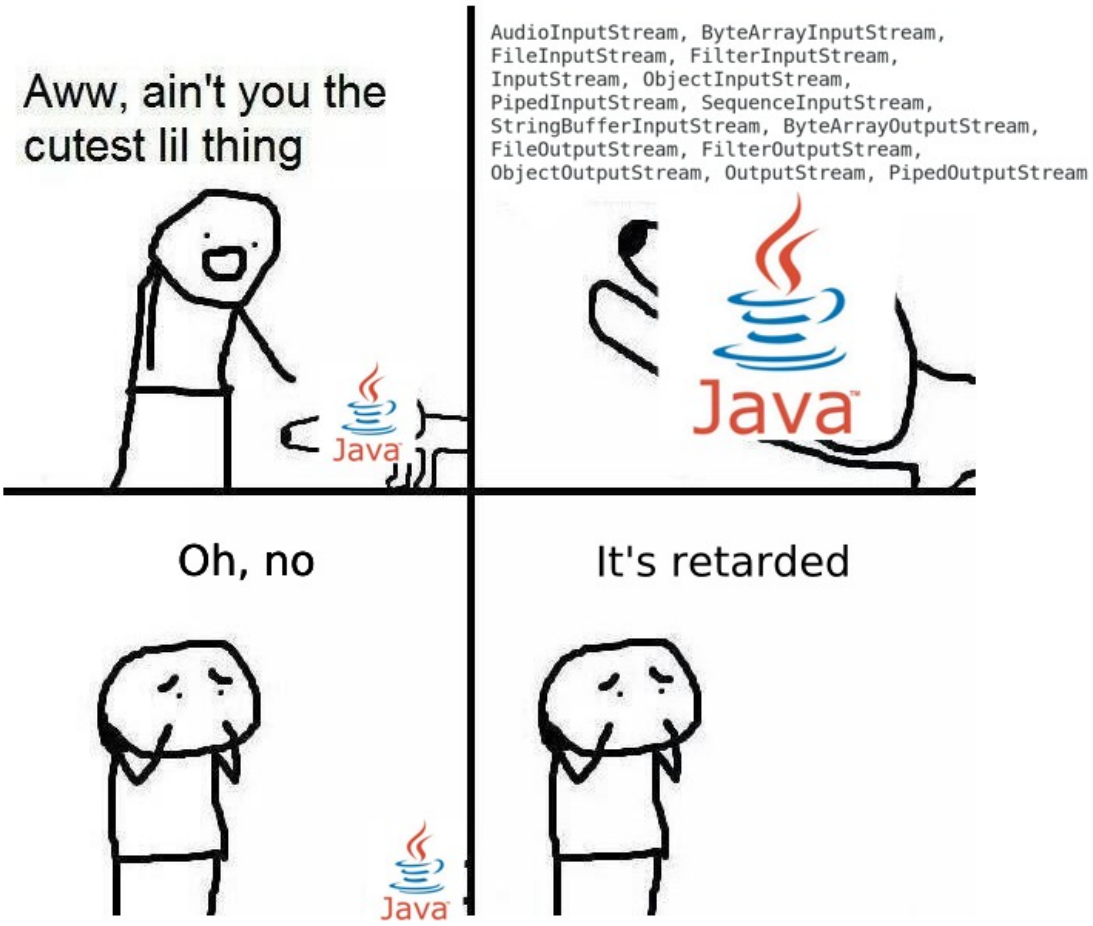
typ dat	vstupní	výstupní
binární	*InputStream*	*OutputStream*
znakové	*Reader*	*Writer*

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**!

Návrhový vzor *Decorator*

- Jedná se o použití návrhové vzoru Decorator:
  - "Nízkoúrovňové" třídy (např. `FileReader`) odpovídají třídě `ConcreteComponent` ze vzoru a poskytují základní funkcionalitu.
  - "Obalující" třídy, jejichž konstruktor vyžaduje již existující proud (např. `BuferedReader`) jsou dekorátory, které se "předsadí" před původní objekt a poskytují dodatečné metody. Na pozadí přitom komunikují s původním objektem.
  - Klientský kód může komunikovat jak s dekorátorem, tak přímo s původním objektem.

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*.
- Ale pozor. Jedná se dvě různé hierarchie. Nelze tedy například vytvořit binární proud a konvertovat ho na buffered reader:

```
FileInputStream is = new FileInputStream("file.txt");
BufferedReader reader = new BufferedReader(is); // Syntax error - incompatible type of
is
```

Konverze binárního proudu na znakový (pokr.)

Musí se použít k tomu určená třída `InputStreamReader` (obdobně pro výstupní proudy `OutputStreamWriter`).

```
// 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 .

Na zjištění, jestli je možné z čtenáře číst, se používa metoda `reader.ready()`.

- `InputStreamReader` (i `OutputStreamWriter`) implementuje návrhový vzor Bridge:
  - Hierarchie tříd `InputStream` představuje *implementaci* čtení binárních dat ze vstupních proudů.
  - `InputStreamReader` pak převádí tato binární data do *abstrakce* textových dat.

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();
```

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*.

Repl.it demo k vstupům a výstupům

- https://repl.it/@tpitner/PB162-Java-Lecture-11-input-and-output

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"));
```

Načítání vstupů (například z klávesnice)

- Třída `java.io.Scanner` - pro čtení z obecného proudu (ale i *stdin*)
- Nebo třída `java.io.Console` - přímo vždy z konzoly

Čtení z konzoly je typické pro aplikace spouštěné z příkazové řádky a není tudíž vždy možné - např. když spouštíme na serveru, v cloudu…

Repl.it demo ke třídě `Scanner`

- https://repl.it/@tpitner/PB162-Java-Lecture-11-scanner

Repl.it demo ke třídě `Console`

- https://repl.it/@tpitner/PV168-Java-Seminar-Console

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.

Návrhový vzor *Memento*

- (De)serializace souvisí návrhovým vzorem Memento:
  - Vzor umožňuje odložit si aktuální stav objektu a později ho obnovit.
  - (De)serializace v Javě tak může sloužit ke snadné implementaci tohoto vzoru.

Odkazy

Java Oracle Tutorial — essential Java I/O

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.

Rozšíření práce se soubory

- Balík `java.nio`
- Třída `java.nio.file.Path`
- Mnoho praktických tříd a metod, v nových verzích Javy:
  - pohotové čtení textů ze souboru,
  - navštěvování souborů v adresáři, ** spousta dalších možností

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

Repl.it demo ke třídě `Path`

- https://repl.it/@tpitner/PB162-Java-Lecture-12-files

Kódování znaků a národní prostředí

Locale

národní prostředí, zahrnuje abecedu, kódování, zápis času/data, měny a národní jazyk

Charset

množina dostupných znaků, které program (typicky v řetězcích) může využít

Encoding

kódování znaků na vnějších zařízeních

Volba kódování

- U mnoha (většiny) I/O operací a proudů lze při vytvoření navolit, které kódování se předpokládá nebo chce.
- Nebo ponechat kódování výchozí (system-default) převzaté ze systému, kde aplikace (JVM) běží.

Implicitní kódování

- V některých případech je výchozí kódování nastaveno napevno bez ohledu na systémová nastavení.
- Některé metody JDK neumožňují zadání znakové sady a vždy předpokládají "výchozí" znakovou sadu UTF-8 pouze pro danou metodu a bez ohledu na lokalizaci nebo konfiguraci systému.
- Většina moderních systémů má UTF-8 stejně jako výchozí.
- Např. metody utilitní třídy `Files` vytvořené kvůli jednoduchosti použití:
  - `Files.newBufferedReader(Path)`,
  - `Files.newBufferedWriter(Path, OpenOption…)`,
  - `Files.readAllLines(Path)`,
  - `Files.write(Path, Iterable, OpenOption…)` či
  - `Files.lines(Path)`.

Řešení: implicitně UTF-8

- Od Javy 18 je UTF-8 výchozí znakovou sadu standardních rozhraní API Javy:

```
Díky této změně se budou rozhraní API závislá na výchozí znakové sadě chovat
konzistentně ve všech implementacích, operačních systémech, místních jazycích a
```

```
konfiguracích.
-- https://openjdk.org/jeps/400
```

Zjištění kódování [1/2]

```
out.println("Default Locale:   " + Locale.getDefault());
out.println("Default Charset:  " + Charset.defaultCharset());
out.println("file.encoding;    " + System.getProperty("file.encoding"));
out.println("sun.jnu.encoding: " + System.getProperty("sun.jnu.encoding"));
out.println("Default Encoding: " + getEncoding());
```

Výše uvedený kód bude fungovat, deklarujeme-li na začátku import objektu `out`: `import static java.lang.System.out;`

Zjištění kódování [2/2]

Vypíše se:

```
Default Locale:   cs_CZ
Default Charset:  UTF-8
file.encoding;    UTF-8
sun.jnu.encoding: UTF-8
Default Encoding: UTF8
```

Soubory vlastností, *property files*

- Textové soubory formátu: řádek = záznam
- Každý záznam má podobu: klíč=hodnota
- Jsou to texty, je jen na nás, jak hodnotu interpretujeme.

```
iconSize=24
iconImg=myico.png
iconTitle=My icon
```

Příklad použití

```
import java.util.Properties;
...
Properties iconProps = new Properties();
iconProps.loadFromXML(new FileInputStream("iconProps.properties"));
```

```
int size = Integer.parseInt(iconProps.getProperty("iconSize"));
```

10 Java Properties Best Practices

Konfigurace aplikace pomocí `Properties` ?

- Chceme `Properties` pro nastavení/konfiguraci aplikace?
- Možná jsou lepší alternativou `java.util.prefs`:
- `java.util.prefs.Preferences`
- Podobný princip jako "systémové registry" ve Windows nebo soubory `.rc` v Linuxu
- Konfigurace specifická pro celý systém (=všechny uživatele dané aplikace)
- Konfigurace specifická pro jednoho uživatele dané aplikace

Kódování souborů vlastností

- V předchozích verzích se pro načítání balíčků prostředků vlastností používalo kódování ISO-8859-1 (=ISO Latin 1).
- Je OK pro západní abecedy.
- Již pro evropské východní nevyhovuje, pro asijské vůbec.

Soubory vlastností UTF-8 od Java 9

- V Javě SE 9 se soubory vlastností načítají v kódování UTF-8.
- UTF-8 je mnohem pohodlnější způsob reprezentace nelatinkových znaků.
- Načítač umí dokonce rozpoznat, kdy jsou v souboru .properties non-UTF-8 znaky, čili jedná se o starý formát ISO-Latin-1. Pak to načte opakovaně správně.

Kompatibilita

- Většina stávajících souborů vlastností by neměla být ovlivněna
- UTF-8 a ISO-8859-1 mají stejné kódování pro znaky ASCII.
- Lidsky čitelné ne-ASCII kódování ISO-8859-1 není platné UTF-8.
- Pokud je zjištěna neplatná posloupnost bajtů UTF-8, běhové prostředí Javy automaticky znovu načte soubor v kódování ISO-8859-1.

Pokud se vyskytne problém, zvažte následující možnosti:

- Převeďte soubor vlastností do kódování UTF-8.
- Zadejte systémovou vlastnost runtime pro kódování souboru vlastností, jako v tomto příkladu:

```
java.util.PropertyResourceBundle.encoding=ISO-8859-1
```

blíže na Java 9 Internationalization

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`

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;