# Java (zkouška)

Původní autor: kohuto

Následuje seznam důležitých věcí, které je třeba vědět, a pod tím zkouškové příklady.

## Teorie

Teorie je seřazena podle tématu.


- Java (zkouška)
  - Teorie
    - Úvod — teorie
    - Keywords — teorie
    - Referenční a hodnotové typy — teorie
    - Literály — teorie
      - Chary — teorie
      - Čísla — teorie
    - Třídy — teorie
    - Viditelnost lokálních proměnných — teorie
    - Návěští (labels) — teorie
    - Dědičnost — teorie
    - Interface — teorie
    - Access modifiers — teorie
    - Final — teorie
    - Postup zpracování programu — teorie
    - Výjimky — teorie
    - Vnitřní (inner) třídy — teorie
    - Anonymní vnitřní třídy — teorie
    - Vnořené (nested) třídy — teorie
    - Assertion — teorie
    - Generické typy — teorie
    - Enum — teorie
    - Anotace — teorie
    - Multithreading — teorie
      - High level multithreading

        - Abstraktní třídy — teorie
        - Funkcionální interface a lambda výrazy — teorie
        - StringBuilder a StringBuffer — teorie
        - Kolekce (collections) — teorie
        - Streams — teorie
            - Streams a input/output
            - Streams a kolekce
    - Zkouškové úlohy
        - Multithreading
        - Třídy
        - Interfacy
        - Lambda výrazy
        - Hodnoty proměnných
        - Triky
        - Základní znalosti
        - Výjimky
        - Jednoduché úkoly na psaní kódu


# Úvod — teorie

Zpět na Teorie.

objektově orientovaný jazyk

interpretovaný jazyk tzn. zdrojový kód (.java) se přeloží do **bytecode** (.class) a je interpertován v tzv. **virtual machine**

# Keywords — teorie

Zpět na Teorie

běžné **keywordy** jsou následující: `abstract` , `assert` , `boolean` , `break` , `byte` , `case` , `catch` , `char` , `class` , `continue` , `default` , `double` , `else` , `enum` , `extends` , `finally` , `float` , `for` , `if` , `implements` , `import` , `int` , `interface` , `long` , `new` , `package` , `private` , `protected` , `public` , `return` , `short` , `static` , `switch` , `this` , `throw` , `throws` , `try` , `void` , `while`

zatímco ty nečekané jsou:

- `const` a `goto` : zatím nedělají nic, jsou pouze rezervované
- `do` : do/while loop

- `final` : viz níže
- `instanceof` : check třídy
- `native` : metoda je implementovaná v Java Native Interface, kde Java spolupracuje s C++, C, Assemblerem apod.
- `strictfp` : třída/interface/metoda, zaručí, že floating-point aritmentika vyjde na všech platformách stejně
- `super` : přístup k proměnným/metodám z přímého předka
- `synchronized` : viz níže v multithreadingu
- `transient` : při ukládání objektů pomocí Seriazable interface se takto označená proměnná neuloží
- `volatile` : viz níže v multithreadingu

# Referenční a hodnotové typy — teorie

Zpět na Teorie

- referenční: jsou někde v paměti a máme na ně pouze pointer
  - v Javě se píší s velkým písmenem
  - jedná se o všechny třídy a objekty ( `String` , `ArrayList` atd.)
  - je možné do nich nastavit `null` jako nulový pointer
  - když je posíláme jako jako argumenty do nějaké metody, posíláme tedy jen *pointer* na tyto objekty, a pokud metoda nějakým způsobem bude své argumenty měnit, projeví se to i u nás
- hodnotové: jsou malé, pracujeme přímo s jejich hodnotou
  - v Javě se píší s malým písmenem
  - např. `boolean` , `int` , `byte` , `float` atd.
  - když je posíláme jako argumenty, pošleme pouze jejich hodnotu (zkopírují se), takže i pokud je metoda bude měnit, ty naše zůstanou nezměněny

od Java 5 automatická konverze mezi primitivními a wrapper typy

```
int a = 5;
Integer b = a; // autoboxing
int c = b; // autounboxing
```

Porovnání `a == b` porovnává hodnoty uložené v `a` a `b` . U referečních typů jsou v `a` a `b` uložen pouze pointery na nějaké objekty, ne ty objekty samotné, proto například neplatí

```
String a = new String("a");
String b = new String("a");
System.out.println(a == b) // false
```

`a` a `b` jsou v tomto případě pointery na dvě různá místa v paměti, tedy se nerovnají. To, jestli jsou na těchto *dvou různých místech* uloženy stringy se stejnou hodnotou zjistíme až pomocí `a.equals(b)`.

string interning - virtual machine ukládá pouze jednu kopii od každého String literálu, proto:

```
String a = "hello";
String b = "hello";
System.out.println(a == b); // true
```

# Literály — teorie

Zpět na Teorie

- neboli způsoby zapsání dat *doslova* v rámci kódu (běžné příklady: `10`, `"string"`, `3.3`, `false`)
- `null` vyjadřuje prázdný pointer, proto jej lze nastavit **pouze u referenčních typů**
- do `boolean` lze nastavit **pouze `false` a `true`**

## Chary — teorie

- **jeden** znak v jednoduchých uvozovkách: `'c'`
- číselný literál: `99` (`0 - 2^16-1`)
- případně i speciální znaky jako `'\n'` nebo unicodové znaky

## Čísla — teorie

- pouze se znaménkem
- `byte` má rozsah -128 až 127
- integery lze zapisovat v různých soustavách:
  - osmičková: začínají na 0, např. `07`
    - pozor, `08` už je špatně, 8 v osmičkové neexistuje
  - binární: `0b01`
  - šesnáctková: `0xAB`
- double lze psát i s exponenty: `1e10`
- v rámci všech čísel jde pro přehlednost použít podtržítka: `1234_5678`

# Třídy — teorie

Zpět na Teorie.

Atributy mají implicitní hodnoty

- boolean: false
- ostatní primitivní typy: 0
- reference: null

`var` – rezervované jméno typu, pouze u lokálních proměnných, musí být inicializace

```
var s = "hello";
var list = new ArrayList<String>();
```

Více konstruktorů rozlišeno různými parametry

```
class MyClass {
  int value;
  public MyClass() { value = 10; }
  public MyClass(int v) { value = v; }
}
```

inicializátor - ekvivalent konstruktoru bez parametru. nezbytné pro inicializaci anonymních vnitřních tříd

```
class MyClass {
 int a;
 int b;
 {
    a = 5;
    b = 10;
 }
 ...
}
```

Metody

- U metod je předávání parametrů pouze **hodnotou**
- overloading (přetížení) pomocí různých parametrů, nelze přetížit jen pomocí změny návratového typu

- Všechny metody jsou virtuální
- static atributy a metody (nejsou virtuální) nejsou svázány s konkrétní instancí (objektem), volají se na třídě

```
// toto je správný příklad jak volat statickou metodu (voláme na třídě)
class A {
  public static void foo() { ... }
}
A.foo();
```

```
// toto je špatný příklad jak volat statickou metodu (voláme ji na instanci)
// odstrašující příklad - typ objektu je A, static se volají na třídě, proto vypíše A
//už kompilátor rozhodne, která metoda se zavolá
public class A {
  public static void foo() {
    System.out.print(“A”);
  }
}
public class B extends A {
  public static void foo() {
    System.out.print(“B”);
  }
}

A a = new B(); // polymorphism
a.foo(); // vypíše A
```

Proměnný počet parametrů

- zápis `...` (tři tečky)
- pouze jako poslední parametr
- lze předat pole nebo seznam parametrů
- v metodě dostupné jako pole

```
void argtest(String firstArg, Object... args) { // tri tecky = promenlivy pocet parametru
  System.out.println(firstArg);
  for (int i=0;i <args.length; i++) {
    System.out.println(args[i]); // v metodě dostupné jako pole
  }
}
argtest("Ahoj", "jak", "se", "vede"); // lze předat pole nebo seznam parametrů
argtest("Ahoj", new Object[] {"jak", "se", "vede"});
```

# Viditelnost lokálních proměnných — teorie

Zpět na Teorie

```
{
  int x=10;
  // dosazitelne je x
  {
    int y=11;
    // dosazitelne je x i y
  }
  // dosazitelne je pouze x
}
{
  int x = 1;
  {
    int x = 2;
    // chyba pri prekladu
    }
}
```

# Návěští (labels) — teorie

Zpět na Teorie

```
label: vnejsi-cyklus {
  vnitrni-cyklus {
    continue label;
    break label;
    }
  }
```

# Dědičnost — teorie

Zpět na Teorie

Dědičnost je jednoduchá (pouze jeden předek). Lze vícenásobná dědičnost pomocí inteface

- subclass (child, podtřída) - třída, která dědí
- superclass (parent, nadtřída) - třída, ze které se dědí

konstruktor

- konstruktor předka: `super()`
  - bez uvedení se stejně automatický vloží, implicitně ale zavolá bezparametrický konstruktor (když v předkovi chybí, kompilátor bude řvát)
- jiný konstruktor stejného objektu: `this()`

Viditelnost členů - platí v rámci jednoho modulu (více v Acces modifiers).

# Interface — teorie

Zpět na Teorie

- pouze definice rozhraní
- může obsahovat
  - hlavičky metod
  - atributy
    - musejí být inicializovány
  - vnitřní interface
- třída musí implementovat všechny metody interface s výjimkou default metod
  - při implementaci dvou interfaců se stejnou default metodou nutno implementovat metodu ve třídě

```
public interface Iterator {
  int a = 5;
  boolean hasNext();
  Object next();
  void remove();
  default void foo() { ... }
}
```

Vícenásobná dědičnost

```
interface Iface1 { ... }
interface Iface2 { ... }
interface Iface3 extends Iface1, Iface2 { ... }
```

# Access modifiers — teorie

Zpět na Teorie

- `private` : přístupné pouze z dané třídy
  - třídy a interfacy nemohou být `private` (pokud to nejsou vnitřní třídy)
- žádný: přístupné všem třídám v současném balíku
- `protected` : přístupné všem podtřídám dané třídy (i v jiném balíku) a všem třídám v současném balíku
  - třídy, interfacy a pole a metody v interfacu nemohou být `protected`
- `public` : přístupné všemu, i z jiného balíku

# Final — teorie

Zpět na Teorie

- `final` proměnná je konstantní (pokud se jedná o referenční proměnnou, její samotný stav se měnit může, pouze reference na ni ne)
- `final` metoda se nedá overloadovat (předefinovat)
- `final` třída nejde subclassovat

Používá se i při tvoření anonymních vnitřních tříd v metodách, objekty, které ve vnitřní třídě použijeme, musí být `final` , nebo alespoň efektivně `final` (tj. nemají explicitně `final` , ale nemění se).

```
public interface MyIface {
   int value();
 }

public class MyClass {
        public MyIface add(final int val) {
                return new MyIface() {
                        private int i = val;
                        public int value() { return i; }
                };
  }
}
```

# Postup zpracování programu — teorie

Zpět na Teorie

1. překlad unicode escape sekvencí (a celého programu) do posloupnosti unicode znaků
2. posloupnost z bodu 1 do posloupnosti znaků a ukončovačů řádků
3. posloupnost z bodu 2 do posloupnosti vstupních elementů (bez "bílých znaků" a komentářů)

# Výjimky — teorie

Zpět na Teorie

- všechny výjimky jsou instance `Throwable`. Ta má dvě podtřídy:
  - `Error`: nikdy by se neměly odchytávat, signalizují velký problém
  - `Exception`: někdy je možná chcete odchytit
- `Exception` se dále dělí
  - *unchecked* (pouze podtřídy `RuntimeException`): kompilátoru nevadí, že je neřešíte
  - *checked* (všechny ostatní): musí být odchyceny nebo vyhozeny výše

Ošetřují se pomocí try/catch/finally bloku.

```
try {
  // zde je blok kodu, kde muze nastat chyba a chceme ji osetrit
} catch (Exception1 e | Exception2 e) {
  // osetreni vyjimky typu Exception1 nebo Exception2
} catch (Exception3 e) {
  // osetreni vyjimky typu Exception3
}
finally {
  // provede se vzdy
}
```

- pokud výjimku neodchytí blok, kde nastala, šíří se do následujícího vyššího bloku
- lze vynechat catch nebo finally (ne obojí)
- v `catch` lze odchytit více různých exception pomocí oddělení `|`, nebo lze za `try` dát více catchů
  - v druhém případě se to chová podobně jako if/else, jakmile se matchne jeden catch, další už se nezkoušejí
- finally proběhne nehledě na to, jak dopadly věci v try/catch
- rozšířený try - v `try` lze použít i objekty, které jsou `AutoClosable`, poté lze vynechat `catch` i `finally` a objekty se samy zavřou

```
class Foo implements AutoClosable {
  public void close() {  ...  }
}

try ( Foo f1 = new Foo(); Foo f2 = new Foo() ) {
  ...
} catch (...) {
  ...
} finally {
  ...
}
```

Pokud metoda může způsobit výjimku, tak musíme výjimku odchytit a specifikovat typ výjimky pomocí `throws`.

- `Error` a `RuntimeException` nemusíme explicitně do `throws` dávat, je možné je vyhazovat vždy

```
class MyException extends java.lang.Exception {}

public class A {
  public void foo(int i) throws MyException {
        if (i < 0) {
      throw new MyException();
    } else if (i == 0) {
      throw new Error(); // V pořádku, i když není ve throws
    } else {
      throw new RuntimeException(); // V pořádku, i když není ve throws
    }
  }
}
```

Řetězení výjimek - př. reakce na systémovou výjimku vlastní výjimkou

```
try {
  ...
}
catch (Exception1 e) {
  throw new Exception2(e);
}
```

# Vnitřní (inner) třídy — teorie

Zpět na Teorie

definice třídy v těle třídy

```
public class MyClass {
  class InnerClass {
    int i = 0;
    public int value() { return i; }
  }
  public void add() { // metoda vytvori instanci InnerClass
    InnerClass a = new InnerClass();
  }
}
```

vnější třída může vracet reference na vnitřní

```
public class MyClass {
  class InnerClass {
    int i = 0;
    public int value() { return i; }
  }
  public InnerClass add() { // metoda vraci instanci InnerClass
    return new InnerClass();
  }
  public static void main(String[] args){
    MyClass p = new MyClass();
    MyClass.InnerClass a = p.add();
  }
}
```

vytvoření objektu vnitřní třídy vně třídy s definicí vnitřní třídy

- nelze vytvořit objekt vnitřní třídy bez objektu vnější třídy, objekt vnitřní třídy má vždy (skrytou) referenci na objekt vnější třídy

```
public class MyClass {
  class InnerClass { ... }

  public static void main(String[] args) {
    MyClass p = new MyClass();
    MyClass.InnerClass i = p.new InnerClass();
  }
}
```

- lze provádět i vícenásobné vnoření tříd (vytváření objektů pak funguje stejně)
- vnitřní třída může být `private` i `protected`
  - přístup k ní přes interface
- vnitřní třídy lze definovat i v metodách (nebo bloku)
  - platnost jen v dané metodě (bloku)

```
public class MyClass {
  public MyIface add() {
    class InnerClass implements MyIface {
      private i = 0;
      public int value() {return i;}
    }
    return new InnerClass();
  }

  public static void main(String[] args) {
    MyClass p = new MyClass();
    MyIface a = p.add(); // nelze - MyClass.InnerClass a = p.add();
  }
}
```

- instance vnitřní třídy může přistupovat ke **všem** členům nadřazené třídy

# Anonymní vnitřní třídy — teorie

Zpět na Teorie

```
public interface MyIface {
   int value();
 }

public class MyClass {
  public MyIface add() {
    return new MyIface() {
      private i = 0;
      public int value() {return i;}
    };
  }

  public static void main(String[] args) {
    MyClass p = new MyClass();
    MyIface a = p.add();
  }
}
```

anonymní vnitřní třídy nemůžou mít konstruktor

# Vnořené (nested) třídy — teorie

Zpět na Teorie

- definovány s `static`
- lze je definovat uvnitř interfacu (vnitřní třídy nelze)
- nemají referenci na objekt vnější třídy
  - pro vytváření instancí nepotřebují objekt vnější třídy

```
public class MyClass {
  public static class NestedClass { ... }

  public static void main(String[] args) {
    MyClass.NestedClass nc = new MyClass.NestedClass();
  }
}
```

# Assertion — teorie

Zpět na Teorie

- příkaz obsahující výraz typu `boolean`
- používá se pro ladění
  - **nesmí mít žádné vedlejší efekty**
- assertions lze zapnout nebo vypnout

příklady použití

- invarianty

```
if (i%3 == 0) {
  ...
} else if (i%3 == 1) {
  ...
} else {
  assert i%3 == 2;
  ...
}
```

- nedosažitelná místa v programu

```
class Directions {
  public static final int RIGHT = 1;
  public static final int LEFT = 2;
}
switch(direction) {
  case Directions.LEFT:
  ...
  case Directions.RIGHT:
  ...
  default:
    assert false;
}
```

preconditions – testování parametrů private metod

```
private void setInterval(int i) {
  assert i>0 && i<=MAX_INTERVAL;
  ...
}
```

# Generické typy — teorie

Zpět na Teorie

- Překlad generických typů (zjednodušeně) – při překladu se vymažou všechny informace o generických typech
- nelze parametrizovat primitivními typy
  vytváření objektů

```
List<Integer> list = new List<Integer>();
List<List<Integer>> list2 =new List<List<Integer>>();

// od Java 7 (operátor „diamant“)
List<Integer> list = new List<>();
List<List<Integer>> list2 =new List<>();
```

nejsou povoleny žádné změny v typových parametrech

```
List<String> a = new List<String>();
List<Object> b = a; // nelze

// protoze bych pak mohl udelat toto
b.add(new Object());
String s = a.get(0); //chyba – přiřazení Object do String
```

`Collection<?>` je nadtyp všech kolekcí

- kolekce neznámého typu (collection of unknown)
- lze přiřadit kolekci jakéhokoliv typu
- do `Collection<?>` nelze přidávat

```
Collection<?> a = new List<String>(); // lze
List<?> b = new List<String>(); // lze
a.add("text"); // nelze
b.add("text"); // nelze
```

použití `extends` :

```
// List něčeho, co extenduje MyClass
List<? extends MyClass>
```

př.:

```
// List něčeho, co extenduje Object
List<? extends Object> = Arrays.asList(1,"Collection",3); //lze

// List něčeho, co extenduje String (tzn. pouze Stringy, String je final)
List<? extends String> names = Arrays.asList(1,"Collection",3); //nelze
```

# Enum — teorie

Zpět na Teorie

```
public enum Color { BLUE, RED, GREEN }
...
public Color clr = Color.BLUE;
```

- nelze dědit
- "normální" třída
    - atributy, konstruktor, metody, i metoda main

```
public enum Planet {
  MERCURY (3.303e+23, 2.4397e6),
  VENUS (4.869e+24, 6.0518e6),
  EARTH (5.976e+24, 6.37814e6)

  private final double mass; // atributy
  private final double radius;

  Planet(double mass, double radius){ // konstruktor
    this.mass = mass;
    this.radius = radius;
  }

  double surfaceGravity() { // metoda
    return G * mass / (radius * radius);
  }
}
```

# Anotace — teorie

Zpět na Teorie

- umožňují přidat informace k elementům v programu
- zapisují se `@JmenoAnotace` (př. `@Deprecated` `@Override`)

# Multithreading — teorie

Zpět na Teorie

- spouštění více *vláken* najednou
- "hlavní" vlákno aplikace metoda `main()`
- JVM skončí až skončí všechna vlákna (která nejsou nastavena jako daemon)
- každé vlákno potřebuje vědět, co na něm poběží (implementace vlákna)
    - v konstruktoru může dostat objekt, který implementuje `Runnable` interface (konkrétně metodu `run`)
    - samotná třída `Thread` je runnable, takže jí můžeme subclassovat a implementovat `run` sami (nedoporučuje se)

- vlákno se po konstrukci musí spustit metodou `.start()`

```
// nedoporučený způsob
public class SimpleThread extends Thread {
  public SimpleThread() {
    start();
  }
  public void run() {
    for (int i=0; i<5; i++)
      System.out.println(getName() + " : " + i);
  }
  public static void main(String[] args) {
    for (int i=0; i<5; i++) {
      new SimpleThread();
    }
  }
}
```

```
// doporučený způsob
public class SimpleThread implements Runnable {
  public void run() { // potřeba implementovat metodu run z interface
     for (int i=0; i<5; i++)
      System.out.println(i);
  }

  public static void main(String[] args) {
    SimpleThread obj = new SimpleThread();
    Thread thread = new Thread(obj); // jako parametr je předán objekt, který implementuje Runna
    thread.start();
  }
}
```

`yield` metoda třídy Thread - dočasné pozastavení vlákna, aby mohlo běžet jiné vlákno

```
// uprava predchoziho prikladu
public void run() {
  for (int i=0; i<5; i++) {
    System.out.println(getName() + " : "+i);
    yield();
  }
}
```

- `sleep(int milis)` uspí vlákno na požadovanou dobu
- `.interrupt()` přeruší **čekání vlákna**, vlákno musí na to být ale připraveno (odchytnout `InterruptedException` nebo kontrolovat `Thread.interrupted` )
- `t.join()` pozastaví současné vlákno do doby, než se dokončí vlákno *t*
- `t.join(int milis)` čeká na dokončení vlákna *t*, ale maximálně zadanou dobu

Daemon vlákna

- "servisní" vlákna (př. vlákno pro garbage collector)

Kdyby dvě vlákna najednou upravovala jeden objekt, mohlo by dojít k chybám; proto má každý objekt *zámek*, který určuje, které vlákno s daným objektem zrovna pracuje.

- vlákno si zámek daného objektu vezme, poté s objektem může pracovat a když je hotové, zámek uvolní
- `synchronized (object) { ... }` zařídí, že kód v bloku bude spouštěn v jednu chvíli pouze jedním vláknem, kterému poskytne zámek objekt `object` (může jít o jakýkoli objekt, nemusí se poté v bloku vůbec vyskytnout)
  - vhodné pro stavy, kdy je hodně writerů i hodně readerů
- `synchronized` může být i metoda

```
class C {
  synchronized void method() {
    /* ... */
  }
}

// chová se stejně jako

class C {
  void method() {
    synchronized (this) {
      /* ... */
    }
  }
}
```

Všechny `synchronized` ne-statické metody jednoho objektu se tedy blokují navzájem (mohou být najednou používany pouze jedním vláknem). Statické `synchronized` metody používají jako objekt k získání zámku samotnou třídu.

Když ale nepotřebujeme udržovat *posloupnost* úprav, jako to dělá `synchronized` , stačí nám modifikátor **atributů** `volatile` . `volatile` je rychlejší (ale slabší) než `synchronized` .

- `volatile` garantuje, že pokud nějaké vlákno do této proměnné zrovna zapisuje, jakékoli čtení této proměnné proběhne až poté, co zapisovací vlákno dokončí svou práci
- každé vlákno tedy vždy vidí nejnovější verzi `volatile` proměnné
- vhodné pro vztahy jeden writer, mnoho readerů

Pokud potřebujeme, aby jedno vlákno čekalo na znamení, že se má spustit, od jiného vlákna, můžeme použí `wait` a `notify` ( `notifyAll` ).

- `wait` pustí zámek současného objektu a suspeduje současné vlákno
- `notifyAll` probudí všechna čekající vlákna, která poté mohou zkontrolovat, jestli už mají běžet (a buďto se spustí, nebo se zase suspendují přes `wait` )

## High level multithreading

- jednodušší než se ručně starat o vlákna a mít jeden task (Runnable objekt) = jedno vlákno
- používají se množiny dlouho existujících vláken (**thread pool**), z nichž každé dělá >1 task (nemusí se tak často rušit a zase vyrábět)

- `FixedThreadPool` operuje s konstantním počtem vláken, zatímco `ForkJoinPool` se hodí pro rekurzivní problémy (buďto vyřeším problém na svém vlákně, nebo ho rozpůlíme mezi dvě)
- počet dostupných jader je možno získat pomocí `Runtime.getRuntime().availableProcessors()`

# Abstraktní třídy — teorie

Zpět na Teorie

- deklarovány pomocí `abstract class ...`
- nelze z nich tvořit objekty, ale lze z nich dědit (jsou vlastně podobné interfacům)
- mohou, ale nemusí, obsahovat abstraktní metody
- mohou, ale nemusí obsahovat neabstraktní (běžné) metody

Co jsou abstraktní metody?

- abstraktní metody nemají implementaci (podobně jako např. metody v interfacu, tam ale nejsou označeny `abstract`)
- jsou také označeny `abstract`
- vždy musí být v abstraktní třídě

Jak se tedy abstraktní třídy od interfaců liší?

- mohou mít atributy, které nejsou `static` a `final`
  - v interfacech jsou implicitně obojí
- mohou mít `public`, `protected`, i `private` konkrétní (neabstraktní) metody
  - v interfacech jsou běžně všechny metody `public`, i když se jedná o `default` implementace
    - `private` metody byly do interfaců přidány v Javě 9

# Funkcionální interface a lambda výrazy — teorie

Zpět na Teorie

- funkcionální interface = interface s právě jednou abstraktní metodou (SAM, single abstract method)
  - právě jedna SAM se dá ověřit s `@FunctionalInterface`
  - pokud má právě jednu abstraktní a spoustu defaultních, taky to projde
  - nesmí ani dědit další abstraktní z jiného interfacu
- používá se jako *typ* pro lambda výrazy (anonymní funkce)
  - argumenty lambda výrazu musí odpovídat argumentům SAM
  - výsledný typ těla lambda výrazu musí odpovídat návratové hodnotě SAM

příklady

```
(int x, int y) -> x + y

(x, y) -> x - y

() -> 42

(String s) -> System.out.println(s)

x -> 2 * x

c -> { int s = c.size(); c.clear(); return s; }
```

lambda výrazy jsou objekty

```
Runnable r = () -> {};
Object o = r;
```

ale lambda výrazy nelze (přímo) přiřadit do typu `Object` ( `Object` není funkcionální interface)

```
Object r = () -> {}; // NELZE
```

reference na objekty

```
String::valueOf
//ekvivalent
x -> String.valueOf(x)

x::toString
// ekvivalent
() -> x.toString()
```

```java
@FunctionalInterface
interface Predicate<T> {
    boolean isTrue(T a); // SAM
}

public class Main {
    public static void main(String[] args) {
        // input = čísla 0 až 9
        List<Integer> input = IntStream.range(0, 10).boxed().collect(Collectors.toList());
        System.out.println(filter(input, n -> n > 5));
              // výsledek: [6, 7, 8, 9]
        // n -> n > 5 má typ Predicate, je automaticky zjištěný
    }

    // vrací prvky seznamu, pro které je predikát pravdivý
    static <T> List<T> filter(List<T> list, Predicate<T> pred) {
        ArrayList<T> result = new ArrayList<>();
        for (T a: list) {
            if (pred.isTrue(a)) {
                result.add(a);
            }
        }
        return result;
    }
}
```

# StringBuilder a StringBuffer — teorie

Zpět na Teorie

- "měnitelný" řetězec (instance třídy `String` jsou neměnitelné)
- nejsou potomky `String` ( `String` , `StringBuffer` , `StringBuilder` jsou final)
- mají stejné metody (vše pro `StringBuffer` platí i pro `StringBuilder` )
  - typické metody `append` , `insert` , `delete` , `length`
  - `StringBuffer` **je** bezpečný vůči vláknům
  - `StringBuilder` **není** bezpečný vůči vláknům

```java
StringBuilder sb = new StringBuilder();
sb.append("text");
```

# Kolekce (collections) — teorie

Zpět na Teorie

objekt obsahující jiné objekty

dva základní druhy - interface Collection a Map. Kolekce neimplementují přímo daný interface, ale třídy `AbstractSet`, `AbstractList`, `AbstractMap`, `AbstractQueue`, `AbstractDeque`

- `Collection<E>` - skupina jednotlivých prvků
  - `List<E>` - drží prvky v nějakém daném pořadí
  - `Set<E>` - každý prvek obsahuje právě jednou
  - `Queue<E>` - fronta prvků
  - `Deque<E>` - oboustranná fronta prvků
- `Map<K,V>` – skupina dvojic klíč–hodnota

Typické metody na `Collection<E>`: `add`, `addAll`, `clear`, `contains`, `remove`, `size`, `toArray`

kolekce mají metodu `Iterator<E> iterator()`, která vrací objekt typu `Iterator<E>`, který umožní projít všechny prvky v kolekci

- `E next()` - vrací další prvek kolekce–boolean
- `hasNext()` - true, pokud jsou další prvky
- `void remove()` - odstraní poslední vrácený prvek z kolekce

```
List c = new ...
Iterator e = c.iterator();
while (e.hasNext()) {
  System.out.println(e.next());
}

// cyklus for na kolekce s iteratorem
for (x:c) {
  System.out.println(x);
}
```

# Streams — teorie

Zpět na Teorie

## Streams a input/output

- Vstupní streamy: `ByteArrayInputStream` , `FileInputStream`
- Výstupní streamy: `ByteArrayOutputStream` , `FileOutputStream`
- Filtry: `FilterInputStream` , `FilterOutputStream` , `PrintStream`

Čtení ze souboru:

```
FileInputStream fileInputStream = new FileInputStream(“file.txt”);
ch = fileInputStream.read();

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

Čtení z konzole:

```
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
String name = reader.readLine();

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

## Streams a kolekce

```
List<String> names = Arrays.asList("Reflection","Collection","Stream"); // list jmen
List<Integer> number = Arrays.asList(2,3,4,5); // list cisel

// pocet slov delsich nez 7 znaku
long count = names.stream().filter(w -> w.length() > 7).count();

// nech pouze jmena zacinajici na "S"
List<String> result = names.stream().filter(s->s.startsWith("S")).
                        collect(Collectors.toList());

// setrizena jmena
 List<String> show = names.stream().sorted().collect(Collectors.toList());

// list druhych mocnin
List<Integer> square = number.stream().map(x -> x*x).
                         collect(Collectors.toList());

// vytiskni prvky
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

operace na streamu nemění zdroj, ale vytvářejí nový stream

lze je snadno paralelizovat

```
long count = words.parallelStream().filter(w -> w.length() > 10).count();
```

# Zkouškové úlohy

Jsou seřazeny podle tématu.

1. Multithreading
2. Třídy
3. Interfacy
4. Lambda výrazy
5. Hodnoty proměnných
6. Triky
7. Základní znalosti
8. Výjimky
9. Jednoduché úkoly na psaní kódu

## Multithreading

Zpět na Zkouškové úlohy.

**Uvažujme následující třídu a předpokládejme, že nějaké vlákno získalo přístup a je uvnitř metody `setX(int, value)`. Pak jiným vláknem:**

```
public class A {
    private int x;

    public synchronized void setX(int value) {
        x = value;
    }

    public synchronized int getX() {
        return x;
    }
}
```

1. nelze pristupovat ani k `getX()` ani k `setX(int value)`

2. lze pristupovat k `getX()` , ale ne k `setX(int value)`
3. kod nelze prelozit, u metod nejsou deklarovany `throws` parametry

*Odpověď:* [1]

**Mějme následující třídu, co platí?**

```
class Test {
   public synchronized int foo() {...}
   public static synchronized void bar() {...}
}
```

1. `foo()` a `bar()` jsou pro přístup více vlákny vyloučeny každá sama se sebou i mezi sebou navzájem
2. `foo()` a `bar()` jsou pro přístup více vlákny vyloučeny každá sama se sebou, ale nikoliv mezi sebou navzájem
3. chyba překladu, nedeklaruje se výjimka `IllegalMonitorStateException`

*Odpověď:* [2]

Jedna metoda je static (používá zámek na třídě jako takové) a druhá není (používá zámek na instanci), proto nejsou navzájem vyloučené.

**Co vypíše následující kód (pokud něco):**

```
public class Main {
    synchronized public void foo() {
        synchronized (this) {
            System.out.print("A");
            synchronized (this) {
                System.out.print("B");
            }
        }
    }
    public static void main(String[] args) {
        new Main().foo();
    }
}
```

1. nepůjde přeložit, synchronized nelze napsat uvnitř metody
2. nic, vlákno se zablokuje

3. AB

*Odpověď:* [3]

Když metoda jednou získá zámek na objektu, tak už ho další bloky kódu v té metodě taky dostanou.

**Která definice `s` je možná, aby se dal kód přeložit?**

```
synchronized (s) {
        /* ... */
}
```

1. `synchronized` se takto nedá použít
2. `Thread s = new Thread();`
3. `Object s = new Object();`
4. `String s = "Hello";`
5. `int s = 100;`
6. `Runnable s = () -> {};`

*Odpověď*: [2, 3, 4, 6] Za `s` je možno dosadit jakýkoli objekt.

**Doplňte deklaraci hashTable tak, aby obsahovala základní sémantiku hash tabulky — metody `V get (K key)` a `void put(K key, V value)`. Navíc k objektu musí bezpečně přistupovat více vláken najednou (tedy volání metod více vlákny najendou je vyloučeno). Můžete si definovat libovolné další třídy nebo použít cokoliv ze standardní knihovny.**

```
import java.util.HashMap;

class HashTable<K, V> {
  private HashMap<K, V> map;

  public HashTable() {
    map = new HashMap<>();
  }

  public synchronized V get(K key) {
    return map.get(key);
  }

  public synchronized void put(K key, V value) {
    map.put(key, value);
  }
}
```

Metody `get` a `put` by obě měly být `synchronized`, zbytek je jednoduchý.

# Třídy

Zpět na Zkouškové úlohy.

**Upravte následující kód tak, aby se zkompiloval:**

```
public class Class {
    abstract void foo();
}
```

*Řešení*:

```
public abstract class Class {
    abstract void foo();
}
```

Abstraktní metody musí být v abstraktní třídě.

**Mějme abstraktní třídu, pak:**

1. od ní nelze vytvářet instance

2. lze od ni dědit, ale nelze předefinovat žádnou její metodu
3. nelze od ni dědit
4. všechny jeji metody jsou také abstraktní

*Odpověď:* [1]

**Jaký modifier může mít vnitřní (inner) třída?**

1. public
2. private
3. static
4. friendly
5. volatile

*Odpověď:* [1, 2, 3]

**Co je pravda?**

1. vnitřní třídy musí implementovat alespoň jeden interface
2. vnitřní třídy mají přístup ke všem (i private) elementům třídy, která je obsahuje
3. vnitřní třídy nedědí od třídy `Object`
4. vnitřní třídy dědí od té vnější

*Odpověď*: [2]

**O jakékoliv třídě Enum platí:**

1. nemůže mít konstruktor
2. není potomkem žádné třídy
3. dědí od `java.lang.Enum`
4. nemůže obsahovat žádné metody
5. může obsahovat `public static void main(String[] args)`

*Odpověď:* [3, 5]

**Co vypíše (pokud něco) kód:**

```
class A {
   static int x = 1;
}

public class Main {
   static A a = null;
   public static void main(String[] args) {
      System.out.println(a.x);
   }
}
```

1. 0
2. 1
3. pokažé něco jiného (závisí na okolnostech)
4. spadne na NullPointerException
5. nelze přeložit

*Odpověď:* [2]

U statického atributu *null* nevadí. Navíc statické atributy lze volat i na instancích třídy, nejen na třídě samé.

**Co vypíše (pokud něco) kód:**

```
class A {
    int x = 1;
}

class B extends A {
    int x = 2;
    public void foo() {
        System.out.println(this.x);
        System.out.println(super.x);
        System.out.println(((A)this).x);
    }
    public static void main(String[] args) {
        B b = new B();
        b.foo();
   }
}
```

1. 2 2 2
2. 2 2 1
3. 2 1 2
4. 2 1 1
5. chyba překladu, this nejde přetypovat
6. chyba překladu, super není na proměnné

*Odpověď:* [4] Viz také otázka na stack overflow.

**Co se vypíše?**

```
public class A {
  public static void foo() { System.out.println("A"); }
}

public class B extends A {
  public static void foo() { System.out.println("B"); }
}

public class Main {
        public static void main(String[] args){
                A a = new B();
                a.foo();
        }
}
```

1. A
2. B
3. nelze určit

*Odpověď*: [1]

Protože voláme **statickou** funkci `foo`, ta se dívá pouze na typ `a` a to je `A` (jinými slovy, není *virtualizovaná*) Zavolá se tedy `A.foo()`. Kdyby se nejednalo o statickou funkci, zavolalo by se `foo()` od objektu `a`, který je reálně ze třídy `B`.

**Rozhodněte, co bude na standardním výstupu po spuštění programu:**

```java
class A {
    int x = 1;
}

class B extends A {
    int x = 2;

    public void foo() {
        System.out.println(this.x);
        System.out.println(super.x);
    }

    public static void main(String[] args) {
        B b = new B();
        b.foo();
    }
}
```

1. 2 2
2. 1 1
3. 2 1
4. nelze aplikovat klicove slovo ***super*** na atributy
5. nelze prepisovat atributy tridy, od ktere se dedi

*Odpověď:* [3] Podobné jako otázka výše. Tomuto se říká *field hiding*.

**Co se vypíše?**

```
class MyClass{
        public static int i = 0;

        public MyClass() {
                i++;
        }

  public static void main(String[] args) {
                MyClass[] my = new MyClass[5];
                for(int i = 0; i < 5; i++){
                        my[i] = new MyClass();
                }
                System.out.println(i);
        }
}
```

1. 0
2. 1
3. 4
4. 5
5. Nelze určit

*Odpověď*: [4]

Vypíše se 5, protože při každém zavolání `new MyClass()` se ke statickému atributu `i` přičte jednička. Protože je atribut statický, jedná se vždy o stejné `i` (netvoří se pro každý `new MyClass` objekt zvlášť). 4 by se vypsalo v případě, že bychom vypisovali `i` z for loopu, to ale mimo loop neexistuje.

**Co vypíše následující kód (pokud něco):**

```
class A {
    public A() {
        super();
        System.out.print("A");
    }
}
class B extends A {
    public B() {
        super();
        System.out.print("B");
    }
}
class C extends B {
    public C() {
        System.out.print("C");
    }
}
public class Main {
    public static void main(String[] args) {
        C c = new C();
    }
}
```

1. nepůjde přeložit, konstruktor třídy A volá super() ale přitom nemá explicitně definovaného předka
2. C
3. ABC

*Odpověď:* [3]

`super()` se totiž z konstruktoru volá implicitně, a každá třída implicitně extenduje `Object` (nebo explicitně nějakou jeho podtřídu)

# Interfacy

Zpět na Zkouškové úlohy.

**Co platí o rozhraních (*interface*):**

1. třída může implementovat nejvýše jeden interface
2. třída může implementovat žádný, jeden nebo i více interfaces
3. interface může implementovat nejvýše jedna třída
4. interface může dědit od nejvýše jednoho interface

5. interface může dědit od žádného, jednoho nebo i více interfaces

*Odpověď:* [2, 5]

**Co se stane s následujícím kódem?**

```
interface A {
    default void hello() { System.out.println("Hello A"); }
}

interface B {
    default void hello() { System.out.println("Hello B"); }
}

class C implements A, B { }
```

1. nepřeloží se nic
2. přeloží se A, B nepřeloží se C
3. nepřeloží se A ani B, ale C ano

*Odpověď:* [2]

C má metodu s dvěma implementacemi a neví, kterou si vybrat. Správně by mělo `hello()` být v C overridnuté a implementované znovu.

**Která tvrzení jsou správná?**

```java
/* soubor A.java */
public interface A {
  void foo(int i);
  default void bar() {
    foo(0);
  }
}

/* soubor B.java */
public class B implements A {
        public void foo(int i) {
                System.out.println(i);
        }
  public void bar() {
    System.out.println("Hello");
        }
}
```

1. Obojí se přeloží bez chyb
2. A se nepřeloží, z defaultní metody se nedá volat nedefaultní
3. A se přeloží bez chyby, ale v B je chyba: defaultní metody z interface se nedají předefinovat
4. A se přeloží bez chyby, ale v B je chyba: před přepsáním metody `bar` chybí klíčové slovo `default`

*Odpověď*: [1]

Defaultní implementace je pouze "záloha" v případě, že metoda není implementována ve třídě.

**Co se vypíše?**

```
interface Iface {
    default void foo() { System.out.println("Interface"); }
}

class A {
    public void foo() { System.out.println("Class"); }
}

class B extends A implements Iface {
    public static void main(String[] args) {
        B b = new B();
        b.foo();
    }
}
```

1. Interface
2. Class
3. Nevypíše se nic
4. Nepůjde přeložit

*Odpověď*: [2]

# Lambda výrazy

Zpět na Zkouškové úlohy.

**Máme definované `Callable`, `Supplier` a `Predicate`. Která z následujících přiřazení lamda výrazu jsou správná (kompilátor je přeloží):**

```
interface Callable<T> {
        T call();
}

interface Supplier<T> {
        T get();
}

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

1. `Supplier<Boolean> su = () -> { return true; };`
2. `Callable<Boolean> sa = () -> { return true; };`
3. `Predicate<Boolean> pr = () -> { return true; };`

*Odpověď*: [1, 2] `Predicate` potřebuje jeden argument.

**Do následujícího kódu doplňte do parametru metody map lambda výraz tak, aby výstupní stream obsahoval délky řetězců ve vstupním streamu.**

```
List<String> list = ...;
Stream<Integer> st = list.stream().map(<DOPLNIT>);
```

*Řešení:*

```
Stream<Integer> st = list.stream().map(s -> s.length);
```

**Metoda `static <T> void sort( T[] array, Comparator<T> c)` setřídí pole `array` a použije k tomu komparátor `c`. Doplňte do následujícího kódu implementeaci komparátoru lambda vyrázem tak, aby třídil řetězce podle délky.**

```
interface Comparator<T> {
        int compare(T o1, T o2);
}

String[] strings = new String[1000];
/* setup kód */

Comparator<String> comparator = <DOPLNIT>;
sort(strings, comparator);
```

*Řešení:*

```
Comparator<String> comparator = (s1, s2) -> s1.length - s2.length;
```

**Máme následující kód. Doplňte do `forEach` výraz tak, aby se vypsaly všechny prvky seznamu.**

```
interface Consumer<T> {
        void accept(T t);
}

void forEach(Consumer<? super T> action) { ... } // hlavička

List<String> list = ...;
list.stream.forEach(<DOPLNIT KÓD>);
```

*Řešení*:

```
list.stream.forEach(a -> System.out.println(a));
```

# Hodnoty proměnných

Zpět na Zkouškové úlohy. Viz Literály.

**Atribut (= field, pozn. Evžena) typu `int` bez explicitní inicializace:**

1. je inicializován hodnotou 0
2. má nedefinovanou hodnotu a při čtení je vráceno předem nestanovitelné číslo
3. má nedefinovanou hodnotu a při čtení je vyvolána výjimka typu UndefinedValueException
4. má nedefinovanou hodnotu a překladač nedovolí použití, dokud není jisté, že se napřed nějaká hodnota nastaví.
5. je inicializován maximální hodnotou, která se do typu int vejde

*Odpověď:* [1]

Atribut je defaultně 0 (případně *false* nebo *null*, podle typu), ale kdyby se jednalo o proměnnou v metodě, neprojde to přes kompilaci.

**Co se nepřeloží?**

1. `byte x = 1024;`
2. `int x = 08;`
3. `long x = null;`
4. `char x = "a";`
5. `int x = 0b01;`

*Odpověď*: [1, 2, 3, 4] Viz část o literálech výše.

**Lokální proměnná typu `int` vyskytující se uvnitř metody:**

1. je od mista deklarace inicializovana hodnotou 0
2. ma nedefinovanou hodnotu a pri cteni je vraceno predem nestanovitelne cislo
3. ma nedefinovanou hodnotu a prekladac nedovoli pouziti
4. je inicializovana maximalni hodnotou, ktera se do typu int vejde

*Odpověď:* [3] viz otázka výše, ale naopak.

**Co se nezkompiluje?**

1. `System.out.println(1+1);`
2. `int x = 42 + '25';`
3. `String s = "foo" + 'bar';`
4. `byte x = 255;`

*Odpověď:* [2, 3, 4]

Mezi `''` musí být `char` (tj. jeden znak) a `byte` má rozsah -128 až 127.

**Co jde přiřadit do proměnné typu boolean?**

*Odpověď:* pouze `true` a `false`

**Co se nepřeloží?**

1. `int i = 1234_5678;`
2. `double d = 3.14_15;`
3. `int j = 0x12_ab_12;`
4. `int k = null;`
5. `boolean b = 0;`
6. 
```
char c = '
';
```

*Odpověď*: [4, 5, 6] Viz literály.

**Co se vypíše?**

```
Integer i1 = 5;
int i2 = i1;
i1 += 1;
System.out.println(i1);
System.out.println(i2);
```

1. 5 5
2. 5 6
3. 6 5
4. 6 6
5. nic

*Odpověď*: [3]

Do `i2` se uloží (zkopíruje) pouze unwrapovaná hodnota z `i1`; změna `i1` tedy `i2` neovlivní.

# Triky

Zpět na Zkouškové úlohy.

**Lze napsat deklaraci proměnné `i` tak, aby následující cyklus byl nekonečný?**

```
while (i != i ) {} // 1.
while (i != i + 0) {} // 2.
while (i <= j && j <= i && i != j) {} // 3.
```

*Odpovědi:*

1. `i = Double.NaN`, protože `NaN` se nerovná ničemu.
2. `i = ""`, stringy lze sčítat s čísly, číslo se převede na string
3. `i = new Integer(1)` a `j = new Integer(1)`, jejich porovnání poté porovnává reference

**Příkazem `import static ...` lze naimportovat do lokálního jmenného prostoru:**

1. všechny atributy a metody třídy
2. pouze statické atributy třídy
3. pouze statické metody třídy
4. pouze statické metody a statické atributy třídy
5. pouze atributy a metody označené anotací `@exportStatic`

*Odpověď:* [4]

Analogicky k běžnému `import`, který importuje třídy z balíků, `import static` importuje statické věci ze tříd.

**Výstupem následujícího úseku kódu?**

```
public class A {
    public int x = 0;

        // [pozn. Evžena] toto je ta zajímavá část
    {
        x += 1;
    }
        // konec zajímavé části

    public A() {
        x += 1;
    }

    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.x);
    }
}
```

1. nelze přelozit
2. 0
3. 1
4. 2
5. hodnota se může lišit při opakovaném spuštění programu

*Odpověď:* [4]

Jedná se o tzv. *inicializační blok*. Je to blok kódu, který proběhne při každém vytvoření objektu od dané třídy, a to *před* konstruktorem. Inicializačních bloků může mít třída více, poté jsou spuštěny odshora jeden po druhém (a po nich teprve konstruktor).

**Napište deklaraci proměnné `x` tak, aby po provedení `x = x + 0` neměla původní hodnotu. Pokud to nejde, zdůvodněte.**

*Řešení:*

```
String x = "";
x = x + 0; // x je "0"
```

**Máme kolekci `ArrayList<T>`. Které přiřazení se přeloží bez chyby?**

1. `Collection<Object> co = new ArrayList<Object>();`
2. `Collection<Object> co = new ArrayList<String>();`
3. `Collection<String> co = new ArrayList<Object>();`
4. `Collection<Object> co = new ArrayList<>();`
5. `Collection<?> co = new ArrayList<Object>();`

*Odpověď*: [1, 4, 5]

Přiřazení vlastně tvrdím, že věc napravo má stejný typ jako věc nalevo. Proto můžu vždy přiřazovat do stejného anebo obecnějšího typu, například `Object a = "a"` je v pohodě, protože každý `String` je i `Object`.

Stejně tak platí, že každý `ArrayList<T>` je i `Collection<T>` (odpovědi 1, 4, 5 — `?` znamená "jakýkoli typ"). Ovšem, o vztahu `Collection<A>` a `ArrayList<B>` nevíme nic (odpovědi 2, 3).

Pro zajímavost, druhý a třetí bod by šly opravit přidáním otazníku:

```
// <nějaká subclassa Objectu> ~ String
Collection<? extends Object> co = new ArrayList<String>();
// <nějaká superclassa Stringu> ~ Object
Collection<? super String> co = new ArrayList<Object>();
```

**Napište metodu `void copy(seznamA, seznamB)` (hlavička je pouze přibližná), která zkopíruje prvky seznamu A do seznamu B pomocí metod `T get(int i)` a `void add(T element)` (kde T je typová proměnná). Pozor, seznam A může obsahovat jiné typy než seznam B, vždy ale takové, aby kopírování bylo možná (např. seznam A bude `List<String>` a seznam B `List<Object>`).**

*Řešení:*

```
public static <T> void copy(List<? extends T> a, List<T> b) {
    for (T item: a) {
        b.add(a);
    }
}
```

Nejdůležitější je hlavička, kde se používá `<T>` (deklarování toho, že hodláme použít typovou proměnnou) a poté `?`. Otazník značí "jakýkoli typ" a `? extends T` znamená "jakýkoli typ, který je podtřídou T", kde `T` je obsah druhého seznamu. Akceptovalo se i (o trochu horší) řešení se `super`, které funguje analogicky:

```
public static <T> void copy(List<T> a, List<? super T> b) { ... }
```

# Základní znalosti

Zpět na Zkouškové úlohy. Většinou jsou pokryty v kapitolách výše, nebo se jedná o jednoduché věci jako "equals u objektů".

**Kam lze napsat `abstract`?**

*Odpověď:* Jen před třídu a metodu.

**Co je obsahem pole `args` v metodě `main`?**

*Odpověď:* Pouze argumenty, které byly programu předány (tedy `args[0]` není jméno samotného programu, jako tomu je např. v shellu nebo C#).

**Není-li u prvku třídy (metoda, atribut, ...) uveden žádný modifikátor viditelnosti (public, private, ...), je tento prvek viditelný:**

1. pouze z této třídy
2. pouze z této třídy a potomků této třídy
3. pouze ze stejného balíku
4. pouze ze stejného balíku a potomků této třídy
5. odkudkoliv

*Odpověď:* [3]

**Která slova jsou klíčová?**

1. throw
2. throws
3. class
4. array
5. namespace
6. method

*Odpověď*: [1, 2, 3] Viz klíčová slova.

**Co se vypíše?**

```
int i = 9;
switch (i) {
  default: System.out.println("default");
  case 0: System.out.println("nula");
      break;
  case 1: System.out.println("jedna");
      break;
  case 2: System.out.println("dva");
      break;
}
```

*Odpověď: default nula*

Za defaultem není `break`, takže poté, co matchne, se pokračuje v tělech casů dokud nepřijde break.

**Co může v následujícím kódu být místo `/* modifier */` ?**

```
public class MyClass {
/* modifier */ void foo() {}
}
```

1. `public`
2. `final`
3. `static`
4. `friendly`
5. `volatile`
6. `override`

*Odpověď*: [1, 2, 3] `volatile` je pouze pro atributy, `friendly` a `override` nejsou klíčová slova.

**Co se stane při překládání?**

```
class MyClass {
    public static void main(String[] args) {
        amethod(args);
    }
    public void amethod(String arguments[]) {
        System.out.println(arguments[1]);
        System.out.println(arguments);
    }
}
```

1. nelze přeložit - metoda `amethod` není deklarována před voláním
2. nelze přeložit - statická metoda `main` volá instanční metodu `amethod`
3. nelze přeložit - špatná deklarace pole v `main`
4. nelze přeložit - `println` nepřijímá jako parametr pole

*Odpověď*: [2]

**Co se přeloží?**

```
if (3 == 2) System.out.println("Hi!"); // 1
if (3 = 2) System.out.println("Hi!"); // 2
if (true) System.out.println("Hi!"); // 3
if (3 != 2) System.out.println("Hi!"); // 4
if ("abcd".equals("abcd")) System.out.println("Hi!"); // 5
```

*Odpověď*: [1, 3, 4, 5]

# Výjimky

Zpět na Zkouškové úlohy. Viz Výjimky — teorie.

**Které výjimky je nutné odchytit nebo deklarovat?**

1. všechny
2. potomky `java.lang.Error`
3. potomky `java.lang.Exception`
4. potomky `java.lang.RuntimeException`
5. žádné

*Odpověď:* [3]

Samozřejmě kromě potomků `RuntimeException`, kteří jsou technicky také potomci `Exception`.

**Co je správná deklarace?**

1. `void foo(void) throws MyException { }`
2. `void foo() throws MyException { }`
3. `void foo() throws { } MyException`
4. `foo() throw MyException { }`

*Odpověď:* [2]

**Co může být v následujícím kódu místo `/* type of exception */` ?**

```
class MyException extends java.lang.Exception {}

public class A {
        public void foo(int i) throws MyException {
                if (i < 0)
                        throw new /* type of exception */();
  }
}
```

1. `MyException`
2. `java.lang.Exception`
3. `java.lang.Error`
4. `java.lang.RuntimeException`
5. `java.lang.Throwable`
6. `java.io.IOException`

*Odpověď*: [1, 3, 4]

`Error` a `RuntimeException` je možno vyhodnit kdykoli i bez předchozí deklarace v `throws`.

**Napište program, který překopíruje jeden soubor do druhého. Ošetřete všechny výjimky.**

Následující kód nemá vše potřebné, ale základ lze poznat.

```
try (InputFileStream ifs = ... ; OutputFileStream ofs = ...;) {
        char c;
        while((c = ifs.read()) != -1) {
                ofs.write(c);
        }
} catch (IOException ex) {
        System.out.println(ex.getMessage()); // ifs a ofs se zavřou samy díky try/with
}
```

## Jednoduché úkoly na psaní kódu

První dvě řešení jsou převzata z Matfiz: Java. Zbytek se týká většinou nějakého jednoduchého chytáku typu `equals` u stringů. Zpět na Zkouškové úlohy.

**Napište metodu, která má dva parametry typu `int`, *hrubou mzdu* a *daň v procentech*, a vrací hodnotu typu `double` udávající daň k zaplacení. Ověřte, že daň je v rozmezí 0–100 a mzda je nezáporná, pokud parametry nejsou v pořádku vyhoďte výjimku `MyException`, která je přímým potomkem `java.lang.Exception` (předpokládá se, že je deklarovaná a importovaná).**

*Řešení:*

```
public class TaxCalculator{
        /* MyException je přímý potomek java.lang.Exception => je třeba deklarovat: */
  public double calculateTax(int gross, int taxPercentage) throws MyException{
    if( gross < 0 || taxPercentage < 0 || taxPercentage > 100 )
      throw new MyException();
    /* Pro FP aritmetiku je třeba buď přetypovat nebo použít FP konstantu (tady 100.0): */
    return gross * (taxPercentage/100.0);
  }
}
```

**Napište třídu pro dynamické pole hodnot typu `int`. Implementujte jen metody pro přidání prvku na konec pole `void add(int x)` a získání hodnoty prvku `int get(int i)` (v případě chybného indexu by měla vyvolat výjimku). Pro implementaci použijte skutečné pole hodnot typu `int`, které se podle potřeby dynamicky realokuje.**

*Řešení:*

```java
public class DynamicArray{
  private int[] intArray = new int[0]; // Nejsnažší řešení (netřeba ošetřovat hodnotu null a/neb
  public  int   get(int i){
    /*
     * Následující přístup může vyhodit ArrayIndexOutOfBoundsException,
     * kterou nemá smysl zachytávat jen, abychom ji opět vyhodili.
     * Jedná se o runtime exception, takže se ani nedeklaruje v hlavičce.
     */
    return intArray[i];
  }
  public  void  add(int x){
    /*
     * Místo ručního kopírování lze použít:
     * - java.lang.System.arraycopy(),
     * nebo ještě lépe:
     * - java.util.Arrays.copyOf(),
     * ale to bych asi u zkoušky nedělal, neb si nepamatuju pořadí parametrů.
     */
    int[]  intArrayX  = new int[intArray.length + 1];  // (zvýšení jen o 1 je na implementaci ne
    int    i          = 0;
    for(int a : intArray)
      intArrayX[i++]  = a;
    intArrayX[i]      = x;
    intArray          = intArrayX;
  }
}
```

**Máme `class Pair { String key; int value; }`, napište metodu `Pair[] find(String key, Pair[] items)`, která vrátí všechny páry z `items`, které mají stejný klíč jako ten daný.**

```
class Pair {
  String key;
  int value;
}

class PairUtil {
  public static Pair[] find(String key, Pair[] items) {
    int count = 0;
    for (Pair item : items) {
      if (item.key.equals(key)) {
        count++;
      }
    }

    Pair[] result = new Pair[count];
    int index = 0;
    for (Pair item : items) {
      if (item.key.equals(key)) {
        result[index++] = item;
      }
    }

    return result;
  }
}
```

**Napište metodu, která v daném seznamu spočítá osoby alespoň tak staré, jak je zadáno v argumentu parametrem `age`. Osoby s neplatným jménem (`null` nebo `""`) do hledání nezahrnujte.**

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

*Řešení:*

```
public static int countOlder(Person[] list, int age) {
    return
      (int) Arrays.stream(list)
        .filter(p -> p.name != null)
        .filter(p -> !p.name.equals(""))
        .filter(p -> p.age >= age)
        .count(); // .count() vrací long, proto na začátku ten cast do (int)
}
```

Dá se samozřejmě řešit i for-loopem. Důležité je nejprve odstranit lidi s `null` jménem, protože jinak bychom kontrolovali `null.age` nebo `null.name.equals("")` a spadlo by nám to. Stringy porovnáváme s `.equals()`. Dobré je také vědět, že `filter` z původního streamu vybere prvky, které *splňují* uvedenou podmínku (má tedy trochu neintuitivní jméno).

**Napište metodu, která vrátí řetězec obsahující n-krát řetězec, který bere v parametru. Např. pro `3` a `"Hello"` vrátí `"HelloHelloHello"`:**

```
public static String repeat(String s, int n) {
  StringBuilder sb = new StringBuilder(s);
  for (int i = 0; i < n; i++) {
    sb.append(s);
  }
  return sb.toString();
}
```

Použití `StringBuilder` místo přičítání stringů.

**Napište metodu, která vytiskne obsah jakékoliv kolekce (jako parametr bere instanci typu `Collection` nebo jakéhokoli jejího podtypu a objekty v ní jsou definovány jakýmkoliv typem mezi špičatými závorkami).**

*Řešení:*

```
<T> void print(Collection<T> items) {
  items.stream().forEach(i -> System.out.println(i));
}
```

**Doplňte metodu tak, aby vracela počet osob s daným křestním jménem narozených v daném roce. Nezapomeňte otestovat, zda předané parametry nejsou `null` atd.**

```
class Person { String firstname; String surname; int yearOfBirth; }
```

*Řešení:*

```
public static int count(Person[] array, int year, String name) {
  if (array == null) { return 0; }
  return
    (int) Arrays.stream(array)
    .filter(p -> p.firstname.equals(name) && p.yearOfBirth == year)
    .count()
}
```

**Definujte korektní `equals()` v následující třídě:**

```
class Complex {
  int re;
  int im;
}
```

*Řešení*:

```
class Complex {
  int re;
  int im;

  @Override
  public boolean equals(Object other) {
    if (other instanceof Complex) {
      Complex that = (Complex) other;
      return this.re == that.re && this.im == that.im;
    }
    return false;
  }
}
```

Musíme zkontrolovat, že je `other` stejná třída pomocí `instanceof` a potom teprve můžeme porvnávat.