Výjimky, výčtové typy a kolekce v Javě

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 2

A0B36PR2 – Programování 2

Obsah přednášky

Výjimky (Exceptions)

- Představují mechanismus ošetření chybových (výjimečných) stavů
- Mechanismus výjimek umožňuje metodu rozdělit na hlavní (standardní) část metody a řešení nestandardní situace

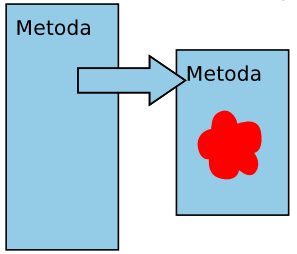
  Umožňuje zpřehlednit kód metod
- Chyba nemusí znamenat ukončení programu
  - Chybu je možné ošetřit, zotavit běh programu a pokračovat ve vykonávání dalšího kódu

`http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html`

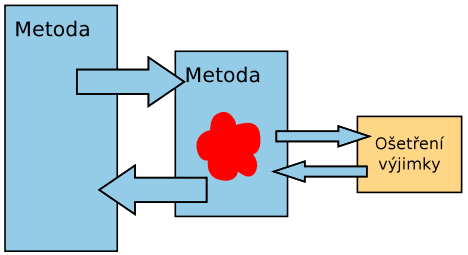
Výjimka nikoliv výjimka – výjimka označuje název děje nebo výsledku děje, je to podstatné jméno odvozené od slovesa.

Princip ošetření výjimky

- Vznik nestandardní situace může ukončit program



- Ošetřením výjimky program může pokračovat ve své „standardní" činnosti



Výjimky (Exceptions)

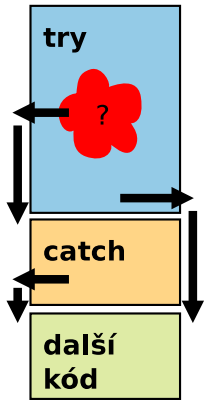
- Mechanismus výjimek umožňuje přenést řízení z místa, kde výjimka vznikla do místa, kde bude zpracována
  - Oddělení *výkonné* části od *chybové* části
- Posloupnost příkazů, ve které může vzniknou výjimka, uzavíráme do bloku klíčovým slovem **try**
- Příslušnou výjimku pak „zachytáváme" prostřednictvím **catch**
- Metodu můžeme deklarovat jako metodu, která může vyvolat výjimku – klíčovým slovem **throws**
- Java ošetření některých výjimečných situací vynucuje
  - **Reakce na očekávané chyby se vynucuje** na úrovni překladu
- Při vzniku výjimky je automaticky vytvořen **objekt**, který nese informaci o vzniklé výjimce (**Throwable**)

http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html

Základní ošetření části kódu, kde může vzniknou výjimka

try – catch

- Volání příkazů/metod výkonné části dáváme do bloků příkazu **try**
- V případě vyvolání výjimky se řízení předá konstrukci ošetření výjimky **catch**
- Při předání vyvolání výjimky se ostatní příkazy v bloku **try** nevolají




```
try {
   //prikazy kde muze vzniknout vyjimka
} catch (Exception e) {
   //osetreni vyjimky
}
// prikazy
```


Mechanismus šíření výjimek v Javě

Při vzniku výjimky hledá JVM odpovídající řešení, které je schopné výjimku ošetřit (převzít řízení):

- Pokud vzniká výjimka v bloku **try** hledá se odpovídající klauzule **catch** v tomto příkazu
- Pokud výjimka vznikne mimo příkaz **try**, předá se řízení do místa volání metody a pokračuje se podle předchozího bodu
- Pokud konstrukce pro ošetření výjimky v těle metody není skončí funkce nestandardně a výjimka se šíří na dynamicky nadřazenou úroveň
- Není-li výjimka ošetřena ani ve funkci **main**, vypíše se a program skončí

- Pro rozlišení různých typů výjimek jsou jazyku Java zavedeny třídy. Výjimky jsou instancemi těchto tříd

Základní dělení nestandardních situací (výjimek)

1. **RuntimeException** – situace, na které bychom měli reagovat a můžeme a dokážeme reagovat
   - Situace, kterým se můžeme vyvarovat programově např. kontrolou mezí pole nebo `null` hodnoty
   - Indexování mimo rozsah pole, dělení nulou, ... **ArrayIndexOutOfBoundException**, **ArithmeticException**, **NullPointerException**, ...

   http://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html
2. **Exception** – situace, na které musíme reagovat
   - Java vynucuje ošetření nestandardní situace
   - Například **IOException**, **FileNotFoundException**
3. **Error** – situace, na které obecně reagovat nemůžeme – závažné chyby
   - Chyba v JVM, HW chyba: **VirtualMachineError**, **OutOfMemoryError**, **IOError**, **UnknownError**, ...

   http://docs.oracle.com/javase/8/docs/api/java/lang/Error.html

Příklad – RuntimeException 1/3

Při spuštění sice získáme informaci o chybě, ale bez zdrojového kódu nevíme přesně co a proč program předčasně končí

- `java DemoException` → **NullPointerException**
- `java DemoException 1` → **ArrayIndexOutOfBoundException**
- `java DemoException 1 a` → **NumberFormatException**
- `java DemoException 1 1` – program vypíše hodnotu 1

```
public class DemoException {

 public int parse(String[] args) {
    return Integer.parseInt(args[1]);
 }

 public static void main(String[] args) {
    DemoException demo = new DemoException();
    int value = demo.parse(args.length == 0 ? null : args);
    System.out.println("2nd argument: " + value);
} }
```

lec02/DemoException

Příklad – RuntimeException 2/3

- Explicitní kontrola parametru

```
public class DemoExceptionTest {
 public int parse(String[] args) {
    int ret = -1;
    if (args != null && args.length > 1) {
       ret = Integer.parseInt(args[1]);
    } else {
       throw new RuntimeException("Input argument not set");
    }
    return ret;
 }

 public static void main(String[] args) {
    DemoExceptionTest demo = new DemoExceptionTest();
    int value = demo.parse(args);
    System.out.println("2nd argument: " + value);
} }
```

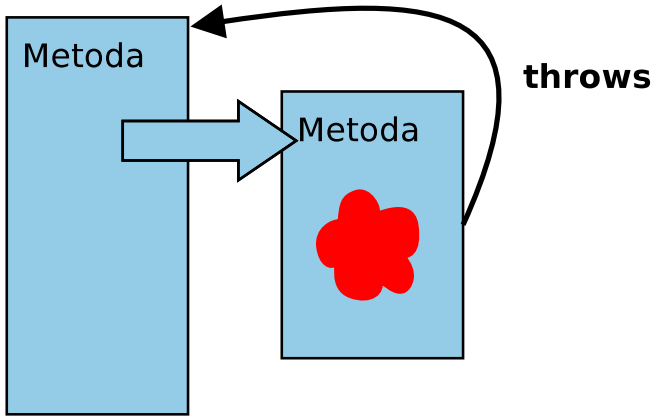
lec02/DemoExceptionTest

- Neřeší však **NumberFormatException**

Předání ošetření výjimky (Exception) výš

- Ošetření výjimky lze předat nadřazené metodě deklarací **throws**

```
public void readData(void) throws IOException {
   ...
}
```



Příklad – RuntimeException 3/3

- Výjimku **NumberFormatException** odchytíme a „nahradíme" upřesňující zprávou
- Výjimku propagujeme výše prostřednictvím **throw**

```
public int parse(String[] args) {
  try {
     if (args != null && args.length > 1) {
       return Integer.parseInt(args[1]);
     } else {
        throw new RuntimeException("Input argument not set");
     }
  } catch (NumberFormatException e) {
     throw new NumberFormatException("2nd argument must be
    int");
} }
```

lec02/DemoExceptionTestThrows

Způsoby ošetření

- Zachytíme a kompletně ošetříme
- Zachytíme, částečně ošetříme a dále předáme výše

Např. Interně v rámci knihovny logujeme výjimku

- Ošetření předáme výše, výjimku nelze nebo ji nechceme ošetřit
- *Bez ošetření výjimky* – **špatně**
  - Aspoň výpis na standardní chybový výstup

```
} catch (Exception e) {
    e.printStackTrace();
}
```

  - Případně logovat (např. do souboru) v případě grafické aplikace nebo uživatelského prostředí

system logger, log4j, ...

Příklad explicitní deklarace propagace výjimky - 1/2

- Hodnota 2. argumentu je pro nás klíčová, proto použijeme výjimku **Exception**, která vyžaduje ošetření
- Výjimku předáváme výš deklarací **throws**

```
public int parse(String[] args) throws Exception {
   try {
      if (args != null && args.length > 1) {
         return Integer.parseInt(args[1]);
      } else {
         throw new Exception("Input argument not set");
      }
   } catch (NumberFormatException e) {
      throw new Exception("2nd input argument must be
    integer");
   }
}
```


Příklad explicitní deklarace propagace výjimky - 2/2

- Kompilace třídy však selže, neboť je nutné výjimku explicitně ošetřit

```
DemoExceptionTestThrow.java:18: error: unreported
    exception Exception; must be caught or declared to
     be thrown
         int value = demo.parse(args)
```

- Proto musí být volání v bloku **try**

```
try {
   int value = demo.parse(args);
   System.out.println("2nd argument: " + value);
} catch (Exception e) {
   System.out.println("Error: " + e.getMessage());
}
```

lec02/DemoExceptionTestThrow

- Nebo **main** musí deklarovat propagaci výjimky výš

```
public static void main(String[] args) throws Exception {
```

lec02/DemoExceptionTestThrowMain

*V tomto případě je použití výjimky **Exception** nevhodné.*

Kdy předávat výjimku výš?

- Pokud je to možné výjimečnou situaci řešíme co nejblíže místa jejího vzniku
- Výjimkám typu **RuntimeException** můžeme předcházet

NullPointerException, ArrayIndexOutOfBoundException typicky indikují opominutí.

- Předávání výjimek **throws** se snažíme vyhnout

Zejména na „uživatelskou" úroveň.

- Výjimky typu **Exception** předáme výš pouze pokud nemá cenu výjimku ošetřovat, např. požadovanou hodnotu potřebujeme a bez ní nemá další činnost programu smysl
- Java při překladu kontroluje kritické části, které vyžadují ošetření nebo deklaraci předání výjimky výš

Kontrolované a nekontrolované výjimky

- **Kontrolované** výjimky musí být explicitně deklarovány v hlavičce metody
  - Jedná se o výjimky třídy **Exception**
  - Označující se také jako **synchronní výjimky**
- **Nekontrolované** výjimky se mohou šířit z většiny metod, a proto by jejich deklarování obtěžovalo
  - Jedná se o **asynchronní výjimky**
  - Rozlišujeme na výjimky, které
    - běžný uživatel není schopen ošetřit (**Error**)
    - chyby, které ošetřujeme podle potřeby; podtřídy třídy **RuntimeException**.

Třída Error

- Představuje závažné chyby na úrovni virtuálního stroje (JVM)
- Nejsme schopni je opravit
- Třída **Error** je nadtřída všech výjimek, které převážně vznikají v důsledku sw nebo hw chyb výpočetního systému, které většinou nelze v aplikaci smysluplně ošetřit

Třída RuntimeException

- Představuje třídu chyb, kterou lze úspěšně ošetřit
- Je třeba je očekávat—jsou to **asynchronní výjimky**
- Nemusíme na ně reagovat a můžeme je propagovat výše
  - Překladač ošetření této výjimky nevyžaduje
- Reagujeme na ně dle našeho odhadu jejich výskytu
  - Pokud špatně odhadneme a nastane chyba, JVM indikuje místo vzniku chyby a my můžeme ošetření výjimky, nebo ošetření vzniku výjimky

    Zpravidla situace, která „nikdy nenastane" se jednou stane. Otázkou tak spíše je, jak často to se to stane při běžném použití programu.
- Prakticky není možné (vhodné) ošetřit všechny výjimky **RuntimeException**, protože to zpravidla vede na nepřehledný kód

Vytvoření vlastní výjimky

- Pro rozlišení případných výjimečných stavů můžeme vytvořit své vlastní výjimky
- Buď odvozením od třídy **Exception** – kontrolované (synchronní) výjimky
- Nebo odvozením od třídy **RuntimeException** – asynchronní

Příklad vlastní výjimky – RuntimeException

- Vlastní výjimku MyRunTimeException vytvoříme odvozením od třídy RuntimeException
- Výjimku MyRunTimeException není nutné ošetřovat

```
class MyRuntimeException extends RuntimeException {
  public MyRuntimeException(String str) {
     super(str);
  }
}

void demo1() {
  throw new MyRuntimeException("Demo MyRuntimeException");
}
```


Vytvoření vlastní výjimky – Exception

- Vlastní výjimku MyException vytvoříme odvozením od třídy Exception
- Výjimku MyException je nutné ošetřovat, proto metodu demo2 deklarujeme s throws

```
class MyException extends Exception {
   public MyException(String str) {
      super(str);
   }
}

void demo2() throws MyException {
   throw new MyException("Demo MyException");
}
```


Ošetřování různých výjimek

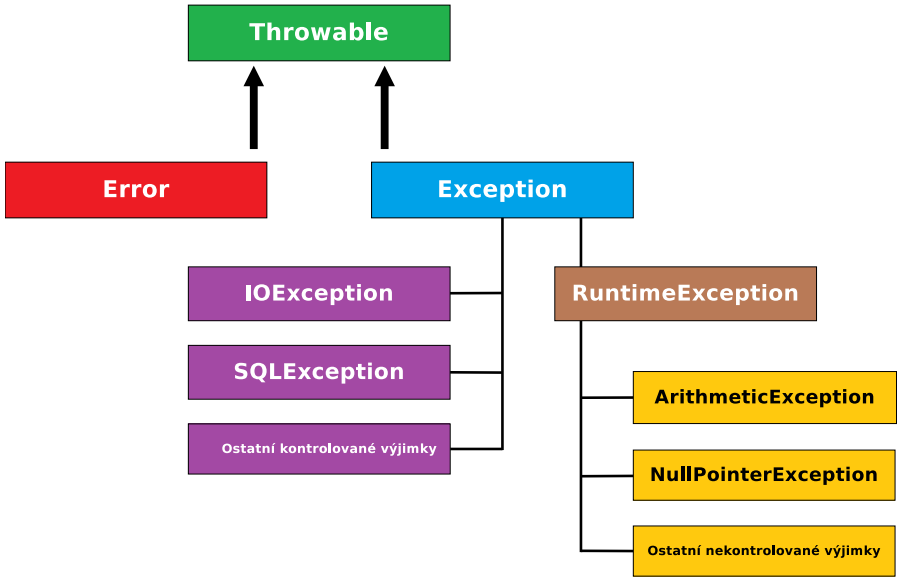
- Příslušná sekce catch ošetřuje kompatibilní výjimky
- Můžeme proto na různé chyby reagovat různě

```
public static void main(String[] args) {
   MyExceptions demo = new MyExceptions();
   try {
      if (args.length > 0) {
         demo.demo1();
      } else {
         demo.demo2();
      }
   } catch (MyRuntimeException e) {
      System.out.println("MyRuntimeException:" + e.
    getMessage());
   } catch (MyException e) {
      System.out.println("MyException:" + e.getMessage());
   }
}
```

lec02/MyExceptions

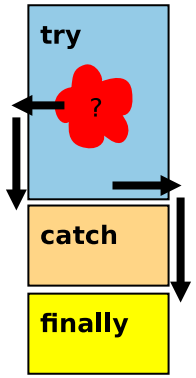
- Při ošetřování výjimek můžeme uplatnit dědické vztahy a hierarchii tříd výjimek

Struktura a hierarchie výjimek



Blok **finally**

- Při běhu programu může být nutné vykonat konkrétní akce bez ohledu na vyvolání výjimky
- Typickým příkladem je uvolnění alokovaných zdrojů, např. souborů
- Příkazy, které se mají vždy provést před opuštěním funkce je možné zapsat do bloku **finally**
- Příkazy v bloku **finally** se provedou i když blok příkazu v **try** obsahuje **return** a k vyvolání výjimečné situace nedojde



http://docs.oracle.com/javase/tutorial/essential/exceptions/finally.html

Příklad – **try** – **catch** – **finally** – 1/2

```
public class BlockFinally {

   void causeRuntimeException() {
      throw new RuntimeException("RuntimeException");
   }

   void causeException() throws MyException {
      throw new MyException("Exception");
   }

   void start(int v) {
      ...
   }

   public static void main(String[] args) {
      BlockFinally demo = new BlockFinally();
      demo.start(args.length > 0 ? Integer.parseInt(args
       [0]) : 1);
   }
}
```

lec02/BlockFinally

Příklad – **try** – **catch** – **finally** – 2/2

```
void start(int v) {
   try {
      if (v == 0) {
         System.out.println("v:0 call runtime");
         causeRuntimeException();
      } else if (v == 1) {
         System.out.println("v:1 call exception");
         causeException();
      } else if (v == 2) {
         System.out.println("v:2 call return");
         return;
      }
   } catch (MyException e) {
      System.out.println("start handle Exception");
   } finally {
      System.out.println("Leave start!");
   }
}
```

- Vyzkoušejte pro různá volání: `java BlockFinally 0`; `java BlockFinally 1`; `java BlockFinally 2`

lec02/BlockFinally

Výjimky a uvolnění zdrojů – 1/2

- Kromě explicitního uvolnění zdrojů v sekci **finally** je možné využít také konstrukce **try-with-resources** příkazu **try**
- Při volání **finally**

```
void writeInt(String filename, int w) throws
    IOException {
   FileWriter fw = null;
   try {
      fw = new FileWriter(filename);
      fw.write(w);
   } finally {
      if (fw != null) {
         fw.close();
} } }
```

totiž může dojít výjimce při zavírání souboru a tím potlačení výjimky vyvolané při čtení ze souboru.

Výjimky a uvolnění zdrojů 2/2

- Proto je výhodnější přímo využít konstrukce **try-with-resources** příkazu **try**

```
void writeInt(String filename, int w) throws
    IOException {
   try (FileWriter fw = new FileWriter(filename)) {
      fw.write(w);
   }
}
```

- **try-with-resources** lze použít pro libovolný objekt, který implementuje **java.lang.AutoCloseable**

http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Pojmenované hodnoty

- Vyjmenované hodnoty reprezentují množinu pojmenovaných hodnot
- Historicky se pojmenované hodnoty dají v Javě realizovat jako konstanty

Podobně jako v jiných jazycích

```
public static final int CLUBS = 0;
public static final int DIAMONDS = 1;
public static final int HEARTS = 2;
public static final int SPADES = 3;
```

- Mezi hlavní problémy tohoto přístupu je, že není typově bezpečný

Jak zajistit přípustné hodnoty příslušné proměnné.

- Například se jedná o hodnoty celých čísel
- Dále nemůžeme jednoduše vytisknout definované hodnoty

Výčtové typy

- Java 5 rozšiřuje jazyk o definování výčtového typu
- Výčtový typ se deklaruje podobně jako třída, ale s klíčovým slovem **enum** místo `class`

```
public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

- V základní podobně se jedná o čárkou oddělený seznam jmen reprezentující příslušné hodnoty
- Výčtové typy jsou typově bezpečné

```
public boolean checkClubs(Suit suit) {
   return suit == Suit.CLUBS;
}
```

Možné hodnoty jsou kontrolovány kompilátorem při překladu.

Vlastnosti výčtových typů

- Uložení dalších informací
- Tisk hodnoty
- Načtení všech hodnot výčtového typu
- Porovnání hodnot
- Výčtový typ je objekt
  - Může mít datové položky a metody
  - Výčtový typ má metodu **values()**
  - Může být použit v řídicí struktuře **switch()**

```
import java.awt.Color;

public enum Suit {

    CLUBS(Color.BLACK),
    DIAMONDS(Color.RED),
    HEARTS(Color.BLACK),
    SPADES(Color.RED);

    private Color color;

    Suit(Color c) {
        this.color = c;
    }

    public Color getColor() {
        return color;
    }
}
```

lec02/Suit

Příklad použití 1/2

```
public class DemoEnum {
   public boolean checkClubs(Suit suit) {
      return suit == Suit.CLUBS;
   }
   public void start() {
      Suit suit = Suit.valueOf("SPADES"); //parse string
      System.out.println("Card: " + suit);

      Suit[] suits = Suit.values();
      for (Suit s : suits) {
         System.out.println(
               "Suit: " + s + " color: " + s.getColor());
      }
   }

   public static void main(String[] args) {
      DemoEnum demo = new DemoEnum();
      demo.start();
   }
}
```

lec02/DemoEnum

Příklad použití 2/2

- Příklad výpisu:

```
java DemoEnum
Card: SPADES color: java.awt.Color[r=255,g=0,b=0]
suit: CLUBS color: java.awt.Color[r=0,g=0,b=0]
suit: DIAMONDS color: java.awt.Color[r=255,g=0,b=0]
suit: HEARTS color: java.awt.Color[r=0,g=0,b=0]
suit: SPADES color: java.awt.Color[r=255,g=0,b=0]
```

- Příklad použití v příkazu **switch**

```
Suit suit = Suit.HEARTS;

switch (suit) {
   case CLUBS:
   case HEARTS:
      // do with black
      break;
   case DIAMONDS:
   case SPADES:
      // do with red
      break;
}
```


Reference na výčet

- Výčet je jen jeden

Singleton

- Referenční proměnná výčtového typu je buď **null** nebo odkazuje na validní hodnotu z výčtu
- Důsledek: pro porovnání dvou referčních hodnot není nutné používat `equals`, ale lze využít přímo operátor `==`

Kolekce (kontejnery) v Javě

Java Collection Framework (JFC)

- Množina třídy a rozhraní implementující sadu obecných a znovupožitelných datových struktur
- Navržena a implementována převážně Joshua Blochem

J. Bloch: Effective Java (2nd Edition), Addison-Wesley, 2008

- Příklad aplikace principů objektově orientovaného programování návrhu klasických datových struktur

Dobrý příklad návrhu

- JFC poskytuje unifikovaný rámec pro reprezentaci a manipulacemi s kolekcemi

Kolekce

- Kolekce (též nazývaná kontejner) je objekt, který obsahuje množinu prvků do jediné datové struktury
- Základními datovými struktury jsou
  - Pole (statické délky) – nevýhody: konečný počet prvků, přístup přes index, implementace datových typů je neflexibilní
  - Seznamy – nevýhody: jednoúčelový program, primitivní struktura
- **Java Collection Framework** – jednotné prostředí pro manipulacemi se skupinami objektů
  - Implementační prostředí datových typů **polymorfního charakteru**
  - Typickými skupinami objektů jsou abstraktní datové typy: množiny, seznamy, fronty, mapy, tabulky, ...
  - Umožňuje nejen ukládání objektů, získávání a jejich zpracování,ale také výpočet souhrnných údajů apod.
  - Realizuje se prostřednictvím: **rozhraní** a **tříd**

Java Collection Framework (JFC)

- Rozhraní (interfaces) – hierarchie abstraktních datových typů (ADT)
  - Umožňuji kolekcí manipulovat s prvky nezávislé na konkrétní implementaci
  - `java.util.Collection`, ...
- Implementace – konkretní implementace rozhraní poskytují základní podporu pro znovupoužitelné datové struktury
  - `java.util.ArrayList`, ...
- Algoritmy – užitečné metody pro výpočty, hledání, řazení nad objekty implementující rozhraní kolekcí.
  - Algoritmy jsou polymorfní
  - `java.util.Collections`

`http://docs.oracle.com/javase/tutorial/collections`

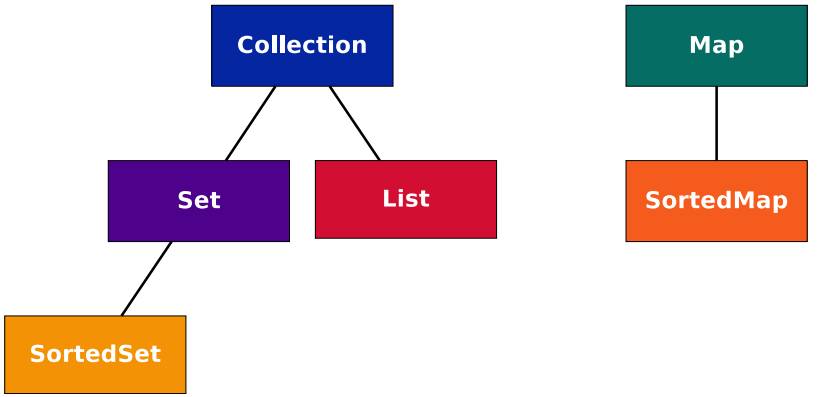
JFC – výhody

- Výkonné implementace – umožňují rychlé a kvalitní programy, možnosti přizpůsobení implementace
- Jednotné API (*Application Programming Interface*)
  - Standardizace API pro další rozvoj
  - Genericita
- Jednoduchost, konzistentnost (jednotný přístup), rychlé naučení
- Podpora rozvoje sw a jeho znovupoužitelnost

  Jednotné API podporuje interoperabilitu i částí vytvořených nezávisle.
- Odstínění od implementačních podrobností

  Kromě JFC je dobrý příklad kolekci také například knihovna STL (Standard Template Library) pro C++.
- Nevýhody
  - Rozsáhlejší kód
  - Široká nabídka možností

Struktura rozhraní kolekce



- **Collection** lze získat z **Map** prostřednictvím **Map.values()**
- Některé operace jsou navrženy jako „*optional*", proto konkretní implementace nemusí podporovat všechny operace

UnsupportedOperationException

Procházení kolekcí v Javě

- Iterátory – **iterator**
  - Objekt umožňující procházet kolekci
  - a selektivně odstraňovat prvky
- Rozšířený příkaz **for-each**
  - Zkráceny zápis, který je
  - realizován jako **o.iterator()**

```
public interface Iterator {
   boolean hasNext();
   Object next();
   void remove(); //Optional
}

Collection collection =
    getCollection();
for (Object o: collection) {
   System.out.println(o);
}
```


Iterátor

- Iterátor lze získat voláním metody **iterator** objektu kolekce
- Příklad průchodu kolekce **collection**

```
Iterator it = collection.iterator();
while(it.hasNext()) {
   System.out.println(it.next());
}
```

- Metoda **next()**:
  1. Vrací aktuální prvek iterátoru
     Po vytvoření iterátoru je to první prvek
  2. Postoupí na další prvek, který se stane aktuálním prvkem iterátoru

Iterátor – metody rozhraní

- Rozhraní **Iterator**

```
public interface Iterator {
   boolean hasNext();
   Object next();
   void remove(); //Optional
}
```

- **hasNext()** – `true` pokud iterace má ještě další prvek
- **next()** – vrací aktuální prvek a postoupí na další prvek
  - Vyvolá **NoSuchElementException** pokud již byly navštíveny všechny prvky
- **remove()** – odstraní poslední prvek vrácený **next**
  - Lze volat pouze jednou po volání **next**
  - Jinak vyvolá výjimku **IllegalStateException**
  - Jediný korektní způsob modifikace kolekce během iterování

Iterátor a způsoby implementace

- Vytvoření kopie kolekce
  - \+ vytvořením privátní kopie nemohou jiné objekty změnit kolekce během iterování
  - – náročné vytvoření $O(n)$
- Přímé využití vlastní kolekce *Běžný způsob*
  - \+ Vytvoření, **hasNext** a **next** jsou $O(1)$
  - – Jiný objekt může modifikovat strukturu kolekce, což může vést na nespecifikované chování operací

Rozhraní **Iterable**

- Umožňuje asociovat **Iterator** s objektem
- Především předepisuje metodu

```
public interface Iterable {
    ...
    Iterator iterator();
    ...
}
```

Iterator: hasNext(); next(); remove(); – jednoduché rozhraní a z toho plynoucí obecnost (genericita).

- V Java 8 rozšíření o další metody

  http://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

- Iterátory v Javě

  http://www.tutorialspoint.com/java/java_using_iterator.htm

- Iterator Design Pattern

  http://sourcemaking.com/design_patterns/Iterator/java/1

  http://java.dzone.com/articles/design-patterns-iterator

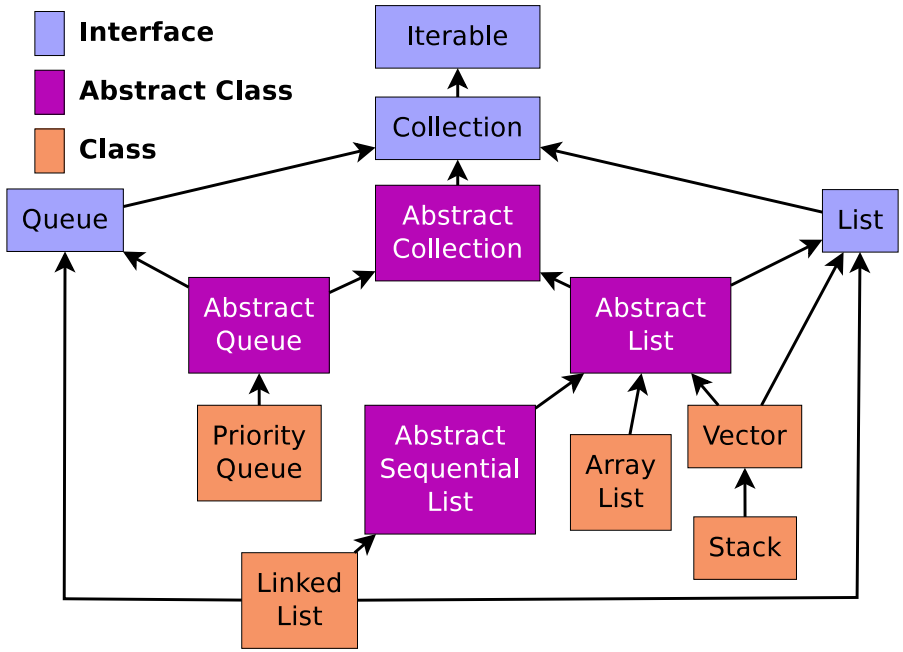
Iterátory a jejich zobecnění

- Iterátory mohou být aplikovány na libovolné kolekce
- Iterátory mohou reprezentovat posloupnost, množinu nebo mapu
- Mohou být implementovány použitím polí nebo spojových seznamů

- Příkladem rozšíření pro spojové seznamy je **ListIterator**, který umožňuje
  - Přístup k celočíselné pozici (index) prvku
  - Dopředný (forward) nebo zpětný (backward) průchod
  - Změnu a vložení prvků

    add, hasNext, hasPrevious, previous, next, nextIndex, previousIndex, set, remove

JFC overview



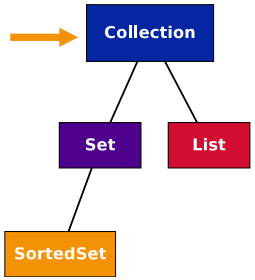
Rozhraní **Collection**

- Co možná nejobecnější rozhraní pro předávání kolekcí objektů

```
public interface Collection {
 // Basic Operations
 int size();
 boolean isEmpty();
 boolean contains(Object element);
 boolean add(Object element);    // Optional
 boolean remove(Object element); // Optional
 Iterator iterator()

 // Bulk Operations
 boolean containsAll(Collection c);
 boolean addAll(Collection c);   // Optional
 boolean removeAll(Collection c);// Optional
 boolean retainAll(Collection c);// Optional
 boolean clear();                // Optional

 // Array Operations
 Object[] toArray();
 <T> T[] toArray(T a[]);
```

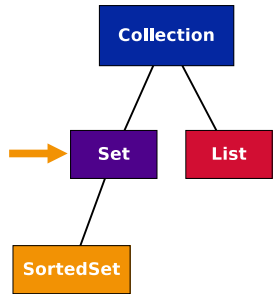


Třída AbstractCollection

- Základní implementace rozhraní **Collection**
- Pro **neměnitelnou** kolekci je nutné implementovat
  - **iterator** spolu s **hasNext** a **next**
  - **size**
- Pro **měnitelnou** kolekci je dále nutné implementovat
  - **remove** pro **iterator**
  - **add**

Rozhraní Set

- **Set** je **Collection**, ve které nejsou duplicitní prvky
- Využívá metod **equals** a **hashCode** pro identifikaci stejných prvků
- Dvě objekty **Set** jsou stejně pokud obsahují stejné prvky



- JDK implementace
  - **HashSet** – velmi dobrý výkon (využívá hašovací tabulku)
  - **TreeMap** – garantuje uspořádání, red-black strom

- **podmnožina** `s1.containsAll(s2)`
- **sjednocení** `s1.addAll(s2)`
- **průnik** `s1.retainAll(s2)`
- **rozdíl** `s1.removeAll(s2)`

Rozhraní List

- Rozšiřuje rozhraní **Collection** pro model dat jako **uspořádanou posloupnost** prvků, indexovanou celými čísly udávající pozici prvku (od 0)

```
public interface List extends Collection {
 // Positional Access
 Object get(int index);
 Object set(int index, Object element);// Optional
 void add(int index, Object element);  // Optional
 Object remove(int index);             // Optional
 abstract boolean addAll(int index,
                         Collection c);// Optional

 // Search
 int indexOf(Object o);
 int lastIndexOf(Object o);

 // Iteration
 ListIterator listIterator();
 ListIterator listIterator(int index);

 // Range-view
 List subList(int from, int to);
}
```

```
public interface ListIterator
   extends Iterator {

   boolean hasNext();
   Object next();

   boolean hasPrevious();
   Object previous();

   int nextIndex();
   int previousIndex();

   void remove();          // Optional
   void set(Object o);     // Optional
   void add(Object o);     // Optional
}
```

- Většina polymorfních algoritmů v JFC je aplikovatelná na **List** a ne **Collection**.

```
sort(List); shuffle(List); reverse(List); fill(List, Object);
copy(List dest, List src); binarySeach(List, Object);
```


Rozhraní AbstractList

- Základní implementace rozhraní **List**
- Pro **neměnitelný** list je nutné implementovat
  - **get**
  - **size**
- Pro **měnitelný** kolekci je dále nutné implementovat
  - **set**
- Pro měnitelný list **variabilní délky** je dále nutné implementovat
  - **add**
  - **remove**

Třída ArrayList

- Náhodný přístup k prvkům implementující rozhraní **List**
- Používá pole (array)
- Umožňuje automatickou změnu velikosti pole
- Přidává metody:
  - **trimToSize()**
  - **encureCapacity(n)**
  - **clone()**
  - **removeRange(int fromIndex, int toIndex)**
  - **writeObject(s)** – zápis seznamu do výstupního proudu **s**
  - **readObject(s)** – načtení seznamu ze vstupního proudu **s**
- **ArrayList** obecně poskytuje velmi dobrý výkon (využívá hašovací tabulky)
- **LinkedList** může být někdy rychlejší a
- **Vector** – synchronizovaná „varianta" **ArrayList**, ale lze též přes *synchronized wrappers*

Rozhraní Map

- **Map** je kolekce, která mapuje klíče na hodnoty
- Každý klíč může mapovat nejvýše jednu hodnotu
- Standardní JDK implementace:
  - **HashMap** – uloženy v hašovací tabulce
  - **TreeMap** – garantuje uspořádání, red-black strom
  - **Hashtable** – hašovací tabulka implementující rozhraní **Map**

*synchronizovaný přístup, neumožňuje **null** prvky a klíče*

```
public interface Map {
  // Basic Operations
  Object put(Object key, Object value);
  Object get(Object key);
  Object remove(Object key);
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  int size();
  boolean isEmpty();
  // Bulk Operations
  void putAll(Map t);
  void clear();
  // Collection Views
  public Set keySet();
  public Collection values();
  public Set entrySet();

  // Intergace for entrySet
  // elements
  public interface Entry {
   Object getKey();
   Object getValue();
   Object setValue(Object val);
  } }
```


Třída SortedSet

- **SortedSet** je **Set**, který udržuje prvky v rostoucím pořadí tříděné podle:
  - přirozené pořadí prvků, nebo dle implementace **Comparator** předaného při vytvoření
- Dále **SortedSet** nabízí operace pro
  - **Range-view** – rozsahové operace
  - **Endpoints** – vrací první a poslední prvek
  - **Comparator access** – vrací **Comparator** použití pro řazení

```
public interface SortedSet extends Set {
   // Range-view
   SortedSet subSet(Object fromElement, Object toElement);
   SortedSet headSet(Object toElement);
   SortedSet tailSet(Object fromElement);

   // Endpoints
   Object first();
   Object last();

   //Comparator access
   Comparator comparator();
}
```


Implementace kolekcí

- **Obecně použitelné implementace**
  Veřejné (**public**) třídy, které poskytují základní implementací hlavních rozhraní kolekcí, například **ArrayList**, **HashMap**
- **Komfortní implementace**
  Mini-implementace, typicky dostupné přes takzvané statické tovární metody (`static factory method`), které poskytují komfortní a efektivní implementace pro speciální kolekce, například **Collections.singletonList()**.
- **Zapouzdřující implementace**
  Implementace kombinované s jinými implementacemi (s obecně použitelnými implementacemi) a poskytují tak dodatečné vlastnosti, např. **Collections.unmodifiableCollection()**

Obecně použitelné implementace

- Pro každé rozhraní (kromě obecného rozhraní Collection) jsou poskytovány dvě implementace

		Implementace			
		Hašovací tabulky	**Variabilní pole**	**Vyvážený strom**	**Spojový seznam**
Rozhraní	**Set**	HashSet		TreeSet	
	List		ArrayList, Vector		LinkedList
	Map	HashMap		TreeMap	

Generické typy – nevýhody polymorfismu

- Flexibilita (znovupoužitelnost) tříd je tradičně v Javě řešena dědičností a polymorfismem
- Polymorfismus nám tak dovoluje vytvořit třídu (např. nějaký kontejner), která umožňuje uložit libovolný objekt (jako referenci na objekt Object)

Např. ArrayList z JFC

- Dynamická vazba polymorfismu však neposkytuje kontrolu správného (nebo očekávaného) typu během kompilace
- Případná chyba v důsledku „špatného" typu se tak projeví až za běhu programu
- Tato forma polymorfismu také vyžaduje explicitní přetypování objektu získaného z nějaké takové obecné kolekce

Příklad použití kolekce ArrayList

```
package cz.cvut.fel.pr2;

import java.util.ArrayList;

public class Simulator {

World world;
ArrayList participants;

Simulator(World world) {
 this.world = world;
 participants = new ArrayList();
}

public void nextRound() {
 for (int i = 0; i < participants.size(); ++i) {
   Participant player = (Participant) participants.get(i);
   Bet bet = world.doStep(player);
 }
}
```

- Explicitní přetypování (Participant) je nutné.

Generické typy

- Java 5 dovoluje použít Generických tříd a metod
- Generický typ umožňuje určit typ instance tříd, které lze do kolekce ukládat
- Generický typ tak poskytuje statickou typovou kontrolu během překladu.

- Generické typy představuji parametrizované definice třídy typu nějaké datové položky
- Parametr typu se zapisuje do mezi <>, například

```
List<Participant> partList = new ArrayList<Participant>();
```

http://docs.oracle.com/javase/tutorial/java/generics/index.html

Příklad použití parametrizované kolekce **ArrayList**

```
package cz.cvut.fel.pr2;

import java.util.ArrayList;

public class Simulator {

World world;
ArrayList<Participant> participants;

Simulator(World world) {
 this.world = world;
 participants = new ArrayList();
}

public void nextRound() {
 for (int i = 0; i < participants.size(); ++i) {
   Participant player = participants.get(i);
   Bet bet = world.doStep(player);
 }

}
```

- Explicitní přetypování (Participant) není nutné.

Příklad – generický a negenerický typ

```
ArrayList participants;
participants = new ArrayList();
participants.push(new PlayerRed());

// vlozit libovolny objekt je mozne
participants.push(new Bet());


ArrayList<Participant> participants2;
participants2 = new ArrayList<Participant>();
participants2.push(new PlayerRed());

// nelze prelozit
// typova kontrola na urovni prekladace
participants2.push(new Bet());
```


Příklad parametrizované třídy

```
import java.util.List;
import java.util.ArrayList;

class Library<E> {
   private List<E> resoures = new ArrayList<E>();

   public void add(E x) {
      resoures.add(x);
   }

   public E getLast() {
      int size = resoures.size();
      return size > 0 ? resoures.get(size-1) : null;
   }
}
```


Generické metody

- Generické metody mohou být členy generických tříd nebo normálních tříd

```
public class Methods {
   public <T> void print(T o) {
      System.out.println("Print Object: " + o);
   }
   public static void main(String[] args) {
      Integer i = 10;
      Double d = 5.5;

      Methods m1 = new Methods();

      m1.print(i);
      m1.print(d);

      m1.<Integer>print(i);

      /// nelze -- typova kontrola
      m1.<Integer>print(d);
   }
}
```

Shrnutí přednášky

Diskutovaná témata

- Ošetření výjimečných stavů – **exceptions**
- Výčtové typy – **enum**
- Kolekce – **Java Collection Framework** (JFC)
  - Generické typy

- Příště: GUI v Javě