ICT Pro
Pokročilé techniky programování (JAV2)

Petr Gregor

Náplň kurzu – den 1.

- Generika
  - Princip vytváření generických tříd a metod
  - `Java Collections API`
- Standardní kolekce `List`, `Map`, `Set`, `Collection`
  - Implementace různých druhů standardních kolekcí, výhody a nevýhody
- Vnořené a vnitřní třídy
  - Statické vnořené, vnitřní, lokální a anonymní třídy

Náplň kurzu – den 2.

- Lambda výrazy, funkcionální programování
  - Parametrizace chování, funkcionální rozhraní
  - Lambda výrazy a jejich použití
  - Výchozí metody rozhraní, třída `Optional`
- Streamy
  - Třída `Stream`, zpracování kolekcí, práce se streamy, rozdíly oproti kolekcím
  - Filtrování, mapování, vyhledávání, redukce a collectory
  - Paralelní streamy
- Vlákna, asynchronní operace, synchronizace
  - Třída `Thread`, rozhraní `Runnable`, stavy a metody vláken

Náplň kurzu – den 3.

- `Swing` GUI
  - Tvorba jednoduché okenní aplikace v Javě
- Práce se soubory
  - Princip `InputStream`, `OutputStream`, `Reader`, `Writer`
  - Neblokující I/O, NIO2 Paths, `Path`, `Files`, `Charsets`
- Java API pro datum a čas
  - `LocalDate`, `LocalDateTime`, `Instant`, `Duration`, `Period`
  - Práce s datem a časem, parsování a formátování

Generika

- Generika jsou také známá jako šablony.
- Umožňují parametrizaci tříd/metod/rozhraní, která se provádí předáváním typů argumentů, v okamžiku, kdy jsou skutečně použity v kódu.
- Pomocí takového mechanismu můžeme znovu použít stejné fragmenty kódu, protože nejsou striktně spojeny s žádnou konkrétní implementací.
- Díky generikům se také můžeme zbavit zbytečné projekce.

Generika

```
1  public class Box {
2
3      private Object item;
4
5      public Object getItem() {
6          return item;
7      }
8
9      public void setItem(Object item) {
10         this.item = item;
11     }
12 }
```

Generika

V případě výše uvedeného fragmentu kódu můžeme předat libovolný objekt objektu třídy `Box` (díky použití typu `Object`). Na úrovni kompilace však nejsme nijak chráněni před chybným předáním různých typů objektů stejnému objektu `Box`, což může vést k chybám za běhu. Díky možnosti použití generických typů je kompilátor schopen validovat přenášené typy.

Generika

Název generického typu definujeme v závorkách <, >, hned za názvem třídy.

```
1  public class Box<T> {
2      private T item;
3
4      public T getItem() {
5          return item;
6      }
7
8      public void setItem(T item) {
9          this.item = item;
10     }
11 }
```

Konvence pojmenování

- E – Prvek (`Element`) (například použitý pro `Java Collection API`)
- K – Klíč (`Key`)
- N – Číslo (`Number`)
- T – Typ (`Type`)
- V – Hodnota (`Value`)
- S (U, V atd.) – druhý, třetí, čtvrtý typ

Vytváření instancí generických tříd

Abychom mohli vytvořit objekt, kterému potřebujeme předat hodnotu generického typu, musíme zadat specifický typ, který nahradí parametr, např. `T`:

```
1 new Box<Integer>();
```

Takto vytvořený objekt lze přiřadit k příslušné referenci, která by měla také obsahovat informace o zadaném generickém typu, tj.:

```
1 Box<Integer> numberBox = new Box<Integer>(); // T replaced
      with Integer
```

Vytváření instancí generických tříd

V Javě, pokud reference, ke které přiřazujeme vytvořený generický objekt, obsahuje informace o typu, nemusíme tento typ při vytváření objektu opakovat, ale použití <> je stále povinné:

```
1 Box<Integer> numberBox = new Box<>(); // <> required
```

Tento typ je zase nutné specifikovat při vytváření objektu, pokud použijeme klíčové slovo `var`, např.:

```
1 var intList = new ArrayList<Integer>();
```

Počet generik

V rámci jedné třídy můžete deklarovat více generik, které budou součástí třídy. Každý parametr typu by měl být dle konvence deklarován jako jedinečný znak.
Následující příklad definuje dvojici generických objektů. Pole klíče i pole hodnoty mohou být libovolného typu.

Počet generik

```
1  public class Pair<K, V> {
2      private K key;
3      private V value;
4
5      public K getKey() {
6          return key;
7      }
8
9      public void setKey(K key) {
10         this.key = key;
11     }
12
13     public V getValue() {
14         return value;
15     }
16
17     public void setValue(V value) {
18         this.value = value;
19     }
}
```

Počet generik

Vytvoření instance pro výše uvedenou třídu bude vypadat takto:

```
1 Pair<String, Float> pair = new Pair<String, Float>();
```

Rozšíření generické třídy

Generickou třídu můžeme snadno rozšířit. Třída, která dědí generickou třídu, musí specifikovat generický typ, jinak zůstane generická. Následující definice tříd demonstrují tyto možnosti:

```
1  public class BaseClass<T, V> {
2  }
3
4  public class NoLongerGenericClass
5    extends BaseClass<String, Integer> {
6    // generics are specified
7  }
8
9  public class StillGenericClass<T>
10   extends BaseClass<T, Integer> {
11   // one generic type was specified - V.
12   // The class is still generic and requires
13   // that the parameter T be specified.
14 }
```

Generické metody

Nejen třídy mohou být generické. Metody také umožňují deklarovat vlastní parametrické typy. Viditelnost těchto typů je omezena na konkrétní metodu (signatura a tělo). Povoleny jsou statické i nestatické generické metody. Syntaxe generických metod rozšiřuje deklaraci metody o parametrické typy, které se umisťují před návratový typ. Při použití takových metod to nemusíme dělat, ale můžeme specifikovat generika. Uděláme to pomocí závorek <>, ve kterých uvedeme hodnoty generik.

Generické metody

```
1  public class PairGenerator {
2    public static <K, V> Pair<K, V> generatePair(K key, V value
         ) {
3      Pair<K, V> pair = new Pair<K, V>();
4      pair.setKey(key);
5      pair.setValue(value);
6      return pair;
7    }
8
9    public static void main(String[] args) {
10     final Pair<Integer, String> firstPair = PairGenerator.
           generatePair(1, "value1");
11     final Pair<Long, String> secondPair = PairGenerator.<Long
           , String>generatePair(2L, "value2");
12   }
13 }
```

Omezení typů parametrů

Obvykle chceme, aby generické třídy, které vytváříme, mohly používat pouze určité hodnoty generik, například ty, které dědí z určité třídy nebo implementují specifické rozhraní. Pro tento účel používáme tzv. *parametry omezených typů*. Deklarace se provádí zadáním parametrického typu, následným použitím klíčového slova `extends` a definováním omezení.

Omezení typů parametrů

```
1  public class NumberBox<T extends Number> {
2
3    private T value;
4
5    public T getValue() {
6      return value;
7    }
8
9    public void setValue(T value) {
10     this.value = value;
11   }
12
13   public static void main(String[] args) {
14     NumberBox<Double> doubleBox = new NumberBox<>();
15     doubleBox.setValue(3.3);
16     NumberBox<Integer> intBox = new NumberBox<>();
17     intBox.setValue(10);
18     System.out.println(intBox.getValue() + " " + doubleBox.
           getValue());
   }
```

Omezení typů parametrů

Pro parametry s omezenými typy používáme klíčové slovo `extends` jak pro třídy, tak pro rozhraní. Navíc je možné vynutit použití generického typu k implementaci mnoha rozhraní, např.:

```
1 public class NumberBox<T extends Number & Cloneable &
      Comparable<T>>
```

Podtypy

U generik je chybou zacházet s parametry typu stejně jako s generickými třídami, např. pokud je `Integer` podtypem `Number`, neznamená to, že `Box<Integer>` je podtypem `Box<Number>`.
Abyste dosáhli očekávaného vztahu, měli byste použít tzv. zástupné znaky.

Zástupné znaky

U generik představuje znak ? neznámý typ. Zástupný znak může představovat:

- typ proměnné
- typ pole třídy
- volitelný návratový typ.

Nemůže však představovat:

- argument generické metody
- argument generické třídy.

Horní limit

Předpokládejme, že chceme vytvořit metodu, která bude fungovat pro seznamy obsahující libovolný číselný typ (tj. objekty, které dědí z třídy `Number`). Pro tento účel můžeme použít tzv. zástupný znak s horní hranicí, který je reprezentován znakem `?` a klíčovým slovem `extends`.

Horní limit

```
1  public class UpperBoundedWildcards{
2
3    public static double sum(final List<? extends Number>
         numbers) { // the method accepts only types extending
         the Number class
4      double sum = 0;
5      for (Number number : numbers) {
6        sum += number.doubleValue();
7      }
8      return sum;
9    }
10
11   public static void main(String[] args) {
12     List<Integer> values = List.of(1, 2, 3);
13     System.out.println(sum(values));
14   }
15 }
```

Dolní limit

Předpokládejme, že potřebujeme napsat metodu, která jako argument bere seznam objektů typu `Integer` nebo ze kterých tato třída dědí (např. `Number` nebo `Object`). Pro tento účel můžeme použít tzv. zástupný znak s dolní hranicí, který je reprezentován znakem `?` a klíčovým slovem `super`, jež odpovídá zadané třídě a všem nadřazeným třídám.

Dolní limit

```
1  public class LowerBoundedWildcards {
2
3    public static void main(String[] args) {
4      addNumbers(List.of(1, 2, 3));
5      addNumbers(List.of(new Object(), new Object(), new Object
           ()));
6    }
7
8    public static void addNumbers(List<? super Integer> list) {
9      for (int i = 1; i <= 10; i++) {
10       list.add(i);
11     }
12   }
13 }
```

Cvičení

1. Úkol 1
   Vytvořte třídu `Pair`, která na základě generických typů umožní ukládat libovolnou dvojici objektů.
2. Úkol 2
   Navrhněte generickou metodu `countIf`, která na základě pole prvků libovolného typu počítá počet prvků splňujících podmínku pomocí funkčního rozhraní. Jakékoli rozhraní implementované anonymně může být funkcí.
3. Úkol 3
   Navrhněte generickou metodu `swap`, která bude zodpovědná za výměnu pozice vybraných prvků pole.

Cvičení

4. Úkol 4
   Vytvořte třídu, která se bude chovat jako knihovna pro následující typy médií:
   - knihy
   - noviny
   - filmy

   Prosím, uveďte řešení pro generické typy. Pro sběr dat použijte libovolnou třídu pole nebo `Collection API`.
5. Úkol 5
   Vytvořte třídu, která se bude chovat jako bouda pro domácí mazlíčky pro následující zvířata:
   - kočka
   - pes

   Prosím, uveďte řešení pro generické typy. Pro sběr dat použijte libovolnou třídu pole nebo `Collection API`.

Kolekce

Kolekce, často označované také jako kontejnery, tj. objekty, které agregují položky. Kolekce se používají k ukládání objektů, získávání uložených dat nebo manipulaci s daty.
V Javě jsou kolekce zabudované do jazyka založeny na následujících mechanismech:

- Rozhraní: abstraktní datový typ reprezentující kolekce. Rozhraní umožňují manipulaci s kolekcemi nezávisle na detailech implementace.
- Implementace: konkrétní implementace rozhraní kolekcí. Nejčastěji se jedná o opakovaně použitelné datové struktury.
- Algoritmus: užitečné operace s datovými strukturami, které nejčastěji mají polymorfní strukturu.

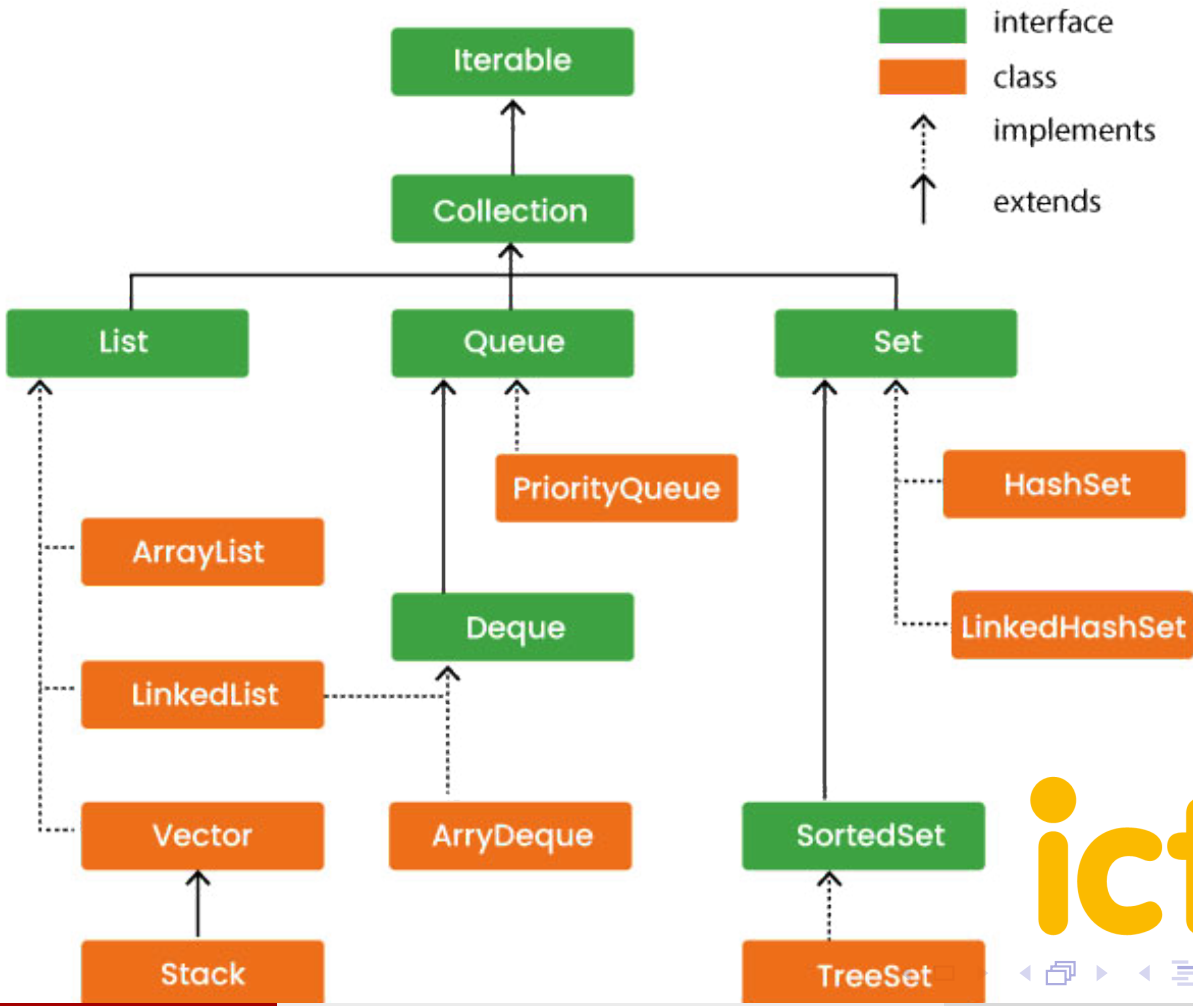
Kolekce

Kolekce v Javě poskytují vývojářům následující výhody:

- zbavují programátora povinnosti implementovat datové struktury od nuly, což zkracuje čas potřebný k implementaci konkrétní funkcionality,
- implementované datové struktury používají nejefektivnější mechanismy a jejich implementace je optimální,
- při návrhu vlastních datových struktur není třeba „znovu vynalézat kolo“. Můžete znovu použít stávající.

Rozhraní (`interfaces`)

Rozhraní kolekcí představují různé typy kontejnerů. Tato rozhraní umožňují manipulaci s kolekcemi bez zacházení s detaily implementace.



Java Collection API

Jádrem všech kolekcí je generické rozhraní:
`veřejné rozhraní Kolekce <E> ...`
Níže jsou uvedena všechna základní API kolekcí:

- Množina (`Set`) – kolekce, která nemůže ukládat duplikáty.
- Seznam (`List`) – uspořádaná kolekce, může obsahovat duplikáty.
- Fronta (`Queue`) – kolekce, která implementuje mechanismus FIFO (*first in, first out*) nebo LIFO (*last in, first out*).
- `Deque` – druh fronty, kde můžete přidávat a odebírat položky, a to jak od začátku, tak od konce.
- `Map` – kolekce pro ukládání párů klíč-hodnota.

Množina (Set)

`Java.util.Set` je generická datová struktura zodpovědná za ukládání **unikátních** prvků. `Set` používá pouze metody rozhraní `Collection`. Navíc zavádí silné propojení mezi metodami: `equals` a `hashCode`. Existují tři základní implementace rozhraní `Set`: `HashSet`, `TreeSet`, `LinkedHashSet`.

- `HashSet`
  - Pořadí položek se nezachovává
  - Ukládá informace do hašovací tabulky
- `TreeSet`
  - Pořadí prvků se zachovává podle tzv. přirozeného pořadí nebo podle nějakého komparátoru
  - Ukládá data do červeno-černého stromu
- `LinkedHashSet`
  - Ukládá informace o pořadí přidávání jednotlivých prvků
  - Implementace je založena na hašovací tabulce s podporou propojených seznamů

HashSet

```java
// creating instance of HashSet
final Set<Integer> numbersSet = new HashSet<>();

// true, Set does not contain elements
System.out.println(numbersSet.isEmpty());

numbersSet.add(1);
numbersSet.add(17);
numbersSet.add(3);
numbersSet.add(2);

// Adding an item with a value that already exists
// the item is NOT added again
numbersSet.add(1);

numbersSet.forEach(System.out::println);

/* example order in which items can be listed:
 1 17 2 3
*/
```

TreeSet

```java
final Set<Integer> numbersSet = new TreeSet<>();

numbersSet.add(1);
numbersSet.add(3);
numbersSet.add(2);

// Adding an item with a value that already exists
// the item is NOT added again
numbersSet.add(1);

numbersSet.forEach (System.out::println);
/* The order of the items will ALWAYS be the same
(sorted in natural order):
 1 2 3
*/
```

LinkedHashSet

```
1  final Set<Integer> numbersSet = new LinkedHashSet<>();
2
3  numbersSet.add(1);
4  numbersSet.add(3);
5  numbersSet.add(2);
6
7  // Adding an item with a value that already exists
8  // the item is NOT added again
9  numbersSet.add(1);
10
11 numbersSet.forEach (System.out::println);
12 /* The order of items will ALWAYS be the same
13 (in the order of adding items)
14  1 3 2
15 */
```

Seznam (List)

`java.util.List` je rozhraní, které reprezentuje uspořádané kolekce.
Vyznačuje se tím, že:

- může obsahovat duplicitní prvky (tj. se stejnou hodnotou)
- položka může být stažena na základě její pozice v seznamu (na základě indexu)
- položka může být vyhledána.

Nejčastěji používané implementace rozhraní `List` jsou:

- `ArrayList`, který je založen na struktuře pole
- `LinkedList`, který je implementován jako uzly. Každý uzel označuje pozici dalšího uzlu

ArrayList

```
1 final List<String> names = new ArrayList<>();
2
3 names.add("Andrew");   // adding an item to the end of the
      list
4 names.add("Gregory"); // adding an item to the end of the
      list
5
6 for (final String name: names) {
7   System.out.println(name); // Andrew, Gregory will be
        displayed on the screen, keeping the order
8 }
```

LinkedList

```
1 final List<String> names = new LinkedList<>();
2
3 names.add(0, "Andrew");  // adding an item to the top of the
      list
4 names.add(0, "Gregory"); // adding an item to the top of the
      list
5
6 for (final String name: names) {
7   System.out.println(name); // Gregory, Andrew will be
        printed on the screen in order
8 }
```

ArrayList a LinkedList – rozdíly

Kdy bychom měli použít `ArrayList` a kdy `LinkedList`? Abychom na tuto otázku odpověděli, musíme si uvědomit, že:

- získání položky na základě jejího indexu z `ArrayList` je rychlejší (`O(1)`) než z `LinkedList` (`O(n)`)
- přidání prvku pomocí metody `add(E someElement)` má stejnou výpočetní složitost pro obě implementace, ale když interní pole v `ArrayList` přeteče, je tato operace pomalejší (`O(n)`)
- přidání prvku do specifického indexu, tj. pomocí metody `add(int index, E someElement)`, je s `LinkedList` rychlejší.

Fronta (Queue)

`java.util.Queue` je generické rozhraní reprezentující datovou strukturu pro implementaci front FIFO a LIFO. Tyto fronty fungují stejným způsobem jako fronty u pokladny v obchodě, tj.:

- přidané položky jdou na konec fronty
- osobu můžeme nejprve „obsloužit“ od:
  - začátku fronty (*FIFO*)
  - konce fronty (*LIFO*).

Fronta (Queue)

Fronta kromě základních operací z rozhraní `Collection` provádí i další operace manipulace s daty:

```
1 public interface Queue<E> extends Collection<E> {
2     E element();
3     boolean offer(E e);
4     E peek();
5     E poll();
6     E remove();
7 }
```

Fronta (Queue)

Různé metody se používají k:

- `element()` – vrací prvek ze „začátku“ fronty (ale **neodstraňuje** ho ze struktury) nebo vyvolá výjimku `NoSuchElementException` v případě prázdné kolekce
- `peek()` – funguje stejně jako metoda `element()`, ale nevyvolává výjimku u prázdné fronty
- `offer()` – přidá položku do fronty a vrátí hodnotu, pokud byla operace úspěšná
- `remove()` – odstraní prvek ze „začátku“ fronty a vrátí jeho hodnotu nebo vyvolá výjimku `NoSuchElementException` v případě prázdné kolekce
- `poll()` – funguje stejně jako metoda `remove()`, ale nevyvolává výjimku u prázdné fronty

Fronta (Queue)

Primární implementací fronty je `LinkedList`.

```
1  final Queue<String> events = new LinkedList<>();
2  events.offer("ButtonClicked");
3  events.offer("MouseMoved");
4
5  // displaying the first element
6  System.out.println(events.peek());
7
8  // removing the first item from the queue and returning a
       value
9  System.out.println(events.poll());
10
11 // removing the first item from the queue again and returning
       the value
12 System.out.println(events.poll());
13
14 // at this point the queue is empty
15 System.out.println(events.isEmpty());
```

Deque

`Deque`, neboli oboustranná fronta, je rozhraní, které popisuje datovou strukturu, což je druh fronty, která umožňuje přidávání a odebírání položek, a to jak na začátku, tak na konci.
Rozhraní `java.util.Deque` rozšiřuje rozhraní `java.util.Queue`.
Nejčastěji používané implementace `Deque` jsou `ArrayDeque` a `LinkedList`.

Deque

Deque nám navíc poskytuje přístup k následujícím metodám:

- `addFirst(e)` a `offerFirst(e)`, které přidávají prvek na začátek fronty
- `addLast(e)` a `offerLast(e)`, které přidávají položku na konec fronty
- `removeFirst()` a `pollFirst()`, které odstraňují prvek ze začátku fronty
- `removeLast()` a `pollLast()`, které odstraňují položku z konce fronty
- `getFirst()` a `peekFirst()`, které načítají položku ze začátku fronty
- `getLast()` a `peekLast()`, které načítají položku z konce fronty.

Deque

Je třeba si uvědomit, že výše uvedené prefixové metody:

- `add`, `remove` a `get` vyvolají v případě selhání výjimku.
- `offer`, `poll` a `peek` vrátí speciální hodnotu v případě selhání operace (`null` pro objekty a `false` pro booleovské hodnoty).

Deque

```java
// create a Deque object
final Deque<Integer>deque = new ArrayDeque<>();

// add elements to deque
deque.offerLast(2);
deque.offerFirst(1);

// remove elements from deque along
// with removing from structure -> 2
System.out.println (deque.pollLast());

// remove elements from deque
// without removing them from structure -> 1
System.out.println (deque.peekLast());
```

Map

Rozhraní `java.util.Map` je datová struktura, která umožňuje manipulaci s daty typu klíč-hodnota. Každý klíč v takovém objektu musí být jedinečný, tj. jeden klíč může obsahovat právě jednu hodnotu.
Mezi metody mapování, které se používají k provádění základních operací, patří:

- `put` – používá se k přidání vhodného páru do kolekce nebo k nahrazení staré hodnoty novou pro konkrétní klíč
- `get` – používá se k získání hodnoty na základě klíče
- `remove` – odebere prvek na základě klíče (nebo další hodnoty)
- `containsKey` – vrátí `true`, pokud v mapování existuje hodnota pro daný klíč
- `containsValue` – vrátí `true`, pokud v mapování existuje klíč pro danou hodnotu
- `size` – vrátí počet párů (tzv. `Entry`) v kolekci
- `isEmpty` – vrátí `true`, pokud je mapa prázdná

Map

POZNÁMKA:

Hodnota `null` může být klíčem v mapě.

Kromě standardních operací obsahuje mapa sadu operací zodpovědných za vracení položek ve formě jiné kolekce:

- `keySet` – vrací sadu klíčů jako `Set`
- `values` – vrací všechny hodnoty jako `Collection`
- `entrySet` – vrací sadu objektů klíč-hodnota. Jeden pár je reprezentován třídou `innerMap.Entry`.

Map

Tři běžné implementace rozhraní `Map` jsou `HashMap`, `TreeMap` a `LinkedHashMap`. Názvy a vlastnosti implementací jsou velmi podobné odpovídajícím implementacím `Set`, tj.:

- `HashMap`:
  - pořadí párů se nezachovává
  - ukládá informace do hašovací tabulky.
- `TreeMap`:
  - pořadí párů se zachovává podle tzv. přirozeného pořadí klíčů nebo podle určitého komparátoru klíčů
  - ukládá data do červeno-černého stromu.
- `LinkedHashMap`:
  - ukládá informace o pořadí přidávání jednotlivých párů
  - implementace je založena na hašovací tabulce s podporou propojených seznamů.

Map

```java
// creating of HashMap
final Map<Integer, String> ageToNames = new HashMap<>();

// adding items
ageToNames.put(11, "Andrew");

// adding another pair
ageToNames.put(22, "Michael");

// adding a third pair to the map
ageToNames.put(33, "John");

// removing an item based on the key
ageToNames.remove(22);

// displaying the value based on the key 11 -> Andrew
System.out.println(ageToNames.get(11));
```

Map

```java
// creating LinkedHashMap
final Map<Integer, String> ageToNames = new LinkedHashMap<>();
ageToNames.put(20, "Maggie");
ageToNames.put(40, "Kate");
ageToNames.put(30, "Anne");

for (final Integer key : ageToNames.keySet()) { // key iteration using keySet()
  System.out.println("Key is map: " + key);     // the order of the keys is always the same -> 20, 40, 30
}

for (final String value : ageToNames.values()) {   // iteration over values using values()
  System.out.println("Value in map is: " + value); // the order of the values is always the same -> Maggie, Anne, Kate
}
```

Map

```
1 for (final Map.Entry<Integer, String> ageToName : ageToNames.
    entrySet()) { // iteration over pairs with entrySet()
2   System.out.println("Value for key  " + ageToName.getKey() +
        " is " + ageToName.getValue());
3   /* the result will always be the following 3 lines, in this
        exact order (results from the use of LinkedHashMap)
4     Value for key 20 is Maggie
5     Value for key 40 is Kate
6     Value for key 30 is Anne
7    */
8 }
```

Map

```java
final Map<Integer, Integer> numberToDigitsSum = new TreeMap<>();
numberToDigitsSum.put(33, 6);
numberToDigitsSum.put(19, 10);
numberToDigitsSum.put(24, 6);
numberToDigitsSum.forEach((key, value) -> System.out.println(key + " " + value));

/* Items will always be listed in the same order:
   19 10
   24 6
   33 6
*/
```

Cvičení

1. Úkol 1
   Implementujte třídu `IctProArrayList<T>`, která bude implementovat logiku `ArrayList<T>`. Pro tento účel implementujte následující metody:
   - `add`
   - `remove`
   - `get`
   - `display`

Cvičení

2. Úkol 2
   Třída `Author`
   Implementujte třídu `Author`, která bude obsahovat pole: `name` (jméno), `surname` (příjmení), `gender` (pohlaví). Zvažte prosím všechny dostupné metody a parametry konstruktoru. Připravte prosím implementaci `hashCode` a `equals`.
   Třída `Book`
   Implementujte třídu `Book`, která bude obsahovat: `title` (název), `price` (cenu), `year` (rok vydání), `authors` (seznam autorů), `genre` (žánr) (reprezentovaný jako třída `enum`). Zvažte prosím všechny potřebné metody a parametry konstruktoru. Připravte prosím implementaci `hashCode` a `equals`.

Cvičení

Třída `BookService`

Implementujte třídu `BookService`, která bude obsahovat seznamy knih a musí zahrnovat následující metody:

- přidávání knih do seznamu
- odebírání knih ze seznamu
- vrácení seznamu všech knih
- vrácení seznamu knih podle typu fantasy
- vrácení seznamu knih vydaných před rokem 1999
- vrácení nejdražší knihy
- vrácení nejlevnější knihy
- vrácení knihy napsané 3 autory
- vrácení seznamu knih seřazeného podle parametru: vzestupně/sestupně

Cvičení

3. Úkol 3
   Na základě pole o 100 prvcích s náhodně vybranými prvky z rozsahu 0-50 implementujte prosím následující funkce:
   - vrácení seznamu unikátních prvků
   - vrácení seznamu prvků, které se v generovaném poli opakovaly alespoň jednou

Cvičení

4. Úkol 4
   Na základě úkolu 2 implementujte metodu, která bude zodpovědná za vrácení unikátních párů klíč-hodnota. Klíč by měl být reprezentován jako žánr knihy, hodnota musí obsahovat název.
5. Úkol 5
   Na základě úkolu 2 implementujte metodu, která bude zodpovědná za vytvoření hromady knih seřazených od nejvyšší po nejnižší cenu.

Vnitřní třídy

V Javě je možné deklarovat třídy uvnitř jiných tříd. Tyto třídy nazýváme vnořenými třídami. Mohou být deklarovány jako:

- statické třídy (tzv. statické vnořené) pomocí klíčového slova `static`
- nestatické třídy (tzv. nestatické nebo vnitřní)

Následující příklad ukazuje, jak nejjednodušeji definovat takové třídy:

```
1  public class Outer {
2
3      static class NestedStatic {
4          // class body
5      }
6
7      class Inner {
8          // class body
9      }
10 }
```

Viditelnost

Nestatické vnořené třídy mají přímý přístup k nadřazené třídě, tj. z nadřazené třídy mohou používat:

- pole (také statická a privátní)
- metody (také statické a privátní)

Viditelnost

```java
public class OuterClass {

  private static int outerClassStaticField;
  private int outerClassField;

  void outerClassMethod() {
    System.out.println("I am outer class method");
  }

  public class InnerClass {
    void useOuterClassField() {
      System.out.println(outerClassStaticField); // use of a
          static field
      outerClassMethod();                        // use of
          method
      System.out.println(outerClassField);       // use of
          private field
    }
  }
}
```

Viditelnost

POZOR:

Na rozdíl od lokálních tříd mohou vnitřní třídy používat modifikátory přístupu.

POZNÁMKA:

Nestatické lokální třídy nemohou definovat statická pole a metody.

Viditelnost

Vnořené statické třídy z nadřazené třídy:

- mohou používat statická pole a metody nadřazené třídy
- nemohou používat nestatická pole a metody

Viditelnost

```
1  public class OuterClass {
2
3    private static int outerClassStaticField;
4    private int outerClassField;
5
6    void outerClassMethod() {
7      System.out.println("I am outer class method");
8    }
9
10   protected static void outClassStaticMethod() {
11     System.out.println("I am out class static method");
12   }
13
14   static class InnerStaticClass {
15
16     void useOuterClassField() {
17       System.out.println(outerClassStaticField);
18       outClassStaticMethod();
19       //outerClassMethod(); -> compile error, we must NOT use
             non-static methods
```

Vytvoření interních tříd

Další příklady budeme zakládat na následujících třídách:

```
1 public class OuterClass {
2
3   class InnerClass {
4   }
5
6   static class InnerStaticClass {
7   }
8 }
```

Pro vytvoření instance vnitřní nestatické třídy musíme nejprve vytvořit instanci vnější třídy, např.:

```
1 OuterClass outerClass = new OuterClass();
2 final OuterClass.InnerClass innerClass = outerClass.new
    InnerClass();
```

Vytvoření interních tříd

Naopak, pro vytvoření instance vnitřní statické třídy je nutné k vnitřní třídě přistupovat pomocí `.`, např.:

```
1 OuterClass.InnerStaticClass innerStaticClass = new OuterClass
      .InnerStaticClass();
```

Anonymní třídy

Anonymní třídy fungují stejně jako lokální třídy. Jediný rozdíl je v tom, že anonymní třídy:

- nemají název
- měly by být deklarovány, pokud je potřebujeme pouze jednou

Syntaxe anonymní třídy

Vytvoření objektu anonymní třídy je téměř identické s vytvořením běžného objektu. Rozdíl je v tom, že při vytváření objektu jsou implementovány všechny požadované metody. Výraz se skládá z:

- operátoru `new` nebo použití `lambda` v případě rozhraní pro funkci
- názvu rozhraní, které implementujeme, nebo abstraktní třídy, kterou dědíme
- parametrů konstruktoru (v případě rozhraní používáme prázdný konstruktor)
- těla třídy/rozhraní

Syntaxe anonymní třídy

V anonymní třídě můžeme deklarovat:

- pole (včetně statických polí)
- metody (včetně statických metod)
- konstanty

Nelze však deklarovat:

- konstruktory
- rozhraní
- statické inicializační bloky

Syntaxe anonymní třídy

Následující příklad ukazuje, jak implementovat anonymní třídu pomocí klíčového slova `new` a následně pomocí `lambda` funkce:

```
1  public interface ClickListener {
2    void onClick();
3  }
4
5  public class UIComponents {
6      void showComponents() {
7          // an anonymous class implementation using the new
               keyword
8          ClickListener buttonClick = new ClickListener() {
9              @Override
10             public void onClick() {
11                 System.out.println("On Button click!");
12             }
13         };
14         // end of anonymous class implementation
15
16         buttonClick.onClick();
```

Přístup k nadřazeným třídám

Anonymní třídy mohou odkazovat na pole nadřazené třídy a lokální proměnné, pokud jsou `final`. Stejně jako u vnořených tříd mají proměnné se stejnými názvy jako pole nadřazené třídy přednost před jejich vlastnostmi.

Lokální třídy

Lokální třídy jsou třídy deklarované v bloku kódu, například v metodě, smyčce `for` nebo příkazu `if`. Při deklaraci lokální třídy vynecháváme informace o modifikátoru přístupu. Objekty lokálních tříd mají přístup k polím metod nadřazených tříd. Následující příklad ukazuje použití lokální třídy přímo v metodě `main`.

POZNÁMKA:

Rozhraní nelze deklarovat v bloku kódu.

Lokální třídy však nejsou tak flexibilní jako „běžné“ třídy:

- nemohou definovat statické metody
- nemohou obsahovat statická pole
- ale mohou obsahovat statické konstanty, tj. ty se statickým modifikátorem `final`

Cvičení

1. Úkol 1
   Vytvořte třídu `UserValidator`, která bude s metodou `validateEmails` zodpovědná za ověřování uživatelských dat, jako například: e-mail, alternativní e-mail. V rámci metody `validateEmails` vytvořte lokální třídu `Email`, která bude zodpovědná za formátování zadaného e-mailu. Ověření by mělo zahrnovat následující scénáře:
   - pokud je zadaná e-mailová adresa prázdná nebo má hodnotu `null`, nastavte hodnotu na neznámá
   - pokud zadaná e-mailová adresa nesplňuje kritéria e-mailu, nastavte hodnotu na neznámá (použijte regulární výrazy)

Cvičení

2. Úkol 2

   Třída `Film`

   Vytvořte třídu `Film`, která bude zahrnovat pole: název, režisér, rok vydání, žánr, distributor. Tato třída by měla obsahovat výchozí konstruktor a metody `getter` a `setter`. Zvažte prosím vytvoření metody `toString`, která bude zodpovědná za vrácení informací o konkrétním objektu.

   Třída `MovieCreator`

   Vytvořte statickou vnořenou třídu `MovieCreateor`. Měla by obsahovat:

   - pole třídy podobná třídě `Film`
   - metody, které budou zodpovědné za nastavení hodnot filmu. Každá metoda by měla vracet instanci objektu, pro který je metoda volána
   - metoda `createMovie` vytvoří instanci třídy `Film`. Vrátí ji jako výsledek metody

Cvičení

3. Úkol 3
   Třída `Car`
   Vytvořte třídu `Car`, která bude ukládat informace o značce a typu automobilu. Měla by obsahovat metody `getter` a `setter`.
   Třída `Engine`
   Implementujte třídu `Engine`, která bude vnořenou nestatickou třídou pod třídou `Car`. Tato třída by měla obsahovat pole: `engine Type` a metodu `setEngine`, která nastaví typ na základě typu automobilu. Pokud je typ automobilu ekonomický, pak by měl být typ motoru nastaven na dieselový. Pokud je typ automobilu luxusní, pak by měl být typ motoru definován jako elektrický. V opačném případě by měl být typ motoru definován jako benzínový.

Cvičení

4. Úkol 4

   Rozhraní Validátoru

   Vytvořte rozhraní Validátoru, které bude obsahovat booleovskou metodu `validate(T input)`.

   Třída `User`

   Vytvořte třídu `User`, která bude obsahovat pole: jméno, příjmení, věk, přihlašovací jméno, heslo, výchozí konstruktor, metody `setter` a `getter`. Metody `setter` by měly jako parametry metody akceptovat hodnotu pro pole a instanci rozhraní Validátoru. Metody `setter` by měly spustit metodu `validate` na základě instance přenášeného objektu. Parametr předaný metodě validate by měl být hodnotou argumentu

   Anonymní třída

   Instance třídy Validátor by měly být implementovány jako anonymní třída.

Cvičení

Implementace by měla splňovat následující kritéria:

- validace jména: jméno nesmí být prázdné ani `null`, mělo by začínat velkým písmenem
- validace příjmení: příjmení nesmí být prázdné ani `null`, mělo by začínat velkým písmenem
- validace věku: hodnota by měla být mezi 0 a 150
- validace přihlašovacího jména: hodnota pole by měla obsahovat 10 znaků
- validace hesla: měla by obsahovat znak `!`

Prosím, uveďte výše popsané řešení na příkladu.

Funkcionální programování

Java je objektově orientovaný jazyk, který se spoléhá na vytváření objektů a komunikaci mezi nimi. V této části se seznámíme s paradigmatem funkcionálního programování. Funkcionální programování se spoléhá pouze na funkce. Hlavní program je funkce, které dáváme argumenty a na oplátku dostáváme výsledek. Hlavní funkce programu se skládá pouze z dalších funkcí, které zase agregují ostatní.

Rozhraní SAM

Mnoho rozhraní v Javě se skládá pouze z jedné abstraktní metody, například `Runnable`, `Callable` nebo `Comparator`. Taková rozhraní se nazývají *Single Abstract Method* (Jedna abstraktní metoda). Java 8 zavádí pro tento typ komponenty název `FunctionalInterface` (Funkcionální rozhraní). Rozhraní `Action`, definované níže, je rozhraní SAM:

```
1 public interface Action {
2     void execute(int x, int y);
3 }
```

Níže uvedené rozhraní nelze kvalifikovat jako rozhraní typu SAM, protože má dvě abstraktní metody.

```
1 public interface Presenter {
2     void present(String text);
3     void present(String text, int size);
4 }
```

@FunctionalInterface

Aby se usnadnila identifikace funkčních rozhraní, byla v Javě zavedena anotace `@FunctionalInterface`, která informuje programátora, že uvedené rozhraní má být funkčním rozhraním. Tato anotace by měla být umístěna nad definicí rozhraní. Pokus o umístění této informace nad rozhraní, které má například nula nebo dvě abstraktní metody, povede k chybě při kompilaci.
Níže vidíme správné použití anotace `FunctionalInterface`:

```
1 @FunctionalInterface
2 public interface Executor {
3     void executor(int x);
4 }
```

POZNÁMKA:

Funkční rozhraní NEMUSÍ být označeno anotací `FunctionalInterface`.

default

Java 8 zavádí mechanismus defaultní metody, specifikovaný klíčovým slovem `default`. Defaultní metoda je metoda rozhraní, která má defaultní implementaci (tj. tato metoda má tělo). Díky této nové syntaxi můžeme v rámci funkčního rozhraní deklarovat více než jednu metodu podle principu jedné abstraktní metody, např.:

```
1 @FunctionalInterface
2 public interface Executor {
3     void executor(int x);
4
5     default void executor(int x, int y) {
6       // I have a body I am a default method
7     }
8 }
```

Lambda výrazy

Velkým problémem anonymních tříd je, že i když implementujeme jednoduché rozhraní s jednou metodou, zápis může být nečitelný. Příkladem toho je předání funkce jiné metodě, například pro zpracování kliknutí na tlačítko.

```
1 Thread thread = new Thread(new Runnable() {
2   @Override
3   public void run() {
4     System.out.println("Implementation of the Runnable
          interface as an implementation of an anonymous class!"
          );
5   }
6 });
7 thread.start();
```

Lambda výrazy

V Javě 8 byly zavedeny lambda výrazy, které umožňují zacházet s anonymní třídou jako s normální funkcí, čímž se výrazně zkracuje syntax samotné notace.

```
1 Thread thread = new Thread(() -> {
2   System.out.println("Runnable example using lambda!");
3 });
4 thread.start();
```

Jak vidíte, lambda výraz nás osvobozuje od zápisu klíčového slova `new` a použití anotace `Override`.

Syntaxe lambda výrazu

Lambda výraz se skládá ze tří částí:

- seznam argumentů, které:
  - musí být v závorkách, pokud je počet argumentů jiný než 1
  - může, ale nemusí uvádět typy argumentů
- operátor `->`, který se používá za seznamem argumentů
- tělo implementované metody, které následuje za operátorem `->`
  - pokud tělo metody sestává z jednoho výrazu, pak:
    - toto tělo nemusí být uvnitř složených závorek, tj. `{` a `}`
    - v případě, že tento výraz vrací nějaký objekt, můžeme klíčové slovo `return` vynechat
  - pokud tělo metody sestává z mnoha výrazů, pak:
    - toto tělo metody musí být uvnitř složených závorek, tj. `{` a `}`.

Syntaxe lambda výrazu

POZNÁMKA:

Pokud je úkolem lambdy vyvolat výjimku, musí se tak stát uvnitř složených závorek, i když se jedná o jediný výraz.

POZNÁMKA:

Názvy argumentů se mohou lišit od názvů definovaných v rozhraní.

POZNÁMKA:

Při definování lambdy nás vůbec **ne**zajímá název abstraktní metody v rozhraní.

Příklad 1

Pro následující rozhraní:

```
1 public interface Action {
2     String execute(int x, int y);
3 }
```

Lambda výraz vypadá takto:

- seznam argumentů: `(int x, int y)`
- operátor `->`
- tělo implementované metody: `{ return x + "-"+ y; }`

Celá věc vypadá takto:

```
1 Action action = (int x, int y) -> {
2     return x + "-" + y;
3 };
```

Příklad 1

```
1 Action action = (int x, int y) -> {
2     return x + "-" + y;
3 };
```

Všimněte si, že ve výše uvedeném příkladu jsme se rozhodli přidat volitelné typy argumentů, deklarovat tělo lambdy uvnitř složených závorek a přidat slovo `return`. Všechny tyto prvky můžeme vynechat a zápis zkrátit na jeden řádek:

```
1 Action action = (x, y) -> x + "-" + y;
```

Příklad 2

Uvažujme rozhraní `Runnable`:

```
1 @FunctionalInterface
2 public interface Runnable {
3   public abstract void run();
4 }
```

Příklad implementace by mohl vypadat takto:

```
1 Runnable runnableExample = () -> {
2   System.out.println("Hello from runnable");
3   System.out.println("{ and } cannot be omitted");
4 };
```

Všimněte si, že pokud nemáme argumenty, musíme použít `()`. Tělo výše uvedené lambdy se skládá ze dvou příkazů, takže musí být definováno uvnitř `{ }`.

Příklad 3

Dalším příkladem by bylo použití funkčního rozhraní `FruitEater` s následující definicí:

```
1 @FunctionalInterface
2 public interface FruitEater<T> {
3     void consume(T t);
4 }
```

Následující implementace má jeden vstupní argument, takže můžeme vynechat závorky a uvést seznam argumentů:

```
1 FruitEater<String> fruitEater = fruit -> System.out.println(
      String.format("eating %s ... omnomnom", fruit));
```

Odkazy na metody

Několik jednoduchých lambda výrazů, které můžeme napsat s odkazem na metodu. Místo psaní volání metody můžeme uvést pouze její název. V tomto případě musí být název třídy a metoda odděleny znaky `::`.
Odkaz na metodu můžeme použít, když má lambda **jeden** argument a je splněna jedna z následujících podmínek:

- vstupní argument lambda je argument metody z nějaké třídy
- na vstupním argumentu je volána metoda bez argumentů.

Odkazy na metody

Následující příklady ukazují možná použití odkazu na metodu:

```
1 // use of lambda
2 Consumer<String> consumerExample = someString -> System.out.
      println(someString);
3
4 // identical notation as in the line above, use of references
5 Consumer<String> consumerExampleReference = System.out::
      println;
```

```
1 // using lambda in the map method
2 List.of("someString").stream().map(str -> str.toUpperCase());
3
4 // using a reference to a method, equivalent notation in the
      line of code above
5 List.of("someString").stream().map(String::toUpperCase);
```

Funkční rozhraní v Javě 8

Java 8 představuje mnoho užitečných funkčních rozhraní, která lze použít mimo jiné ve Stream API. Všechna rozhraní, která budou probrána, jsou v balíčku `java.util.function`. Nejčastěji používaná jsou:

- `Supply<T>`
- `Function<T, R>`
- `Consumer<T>`
- `UnaryOperator<T, T>`
- `Predicate<T>`

Function<T, R>

`Function` je generické funkční rozhraní, které přijímá objekt libovolného typu (`T`) a vrací objekt libovolného typu (`R`). Metoda `apply` je zodpovědná za volání implementované akce. Následující příklad ukazuje použití tohoto rozhraní:

```
1 Function<Employee, String> employeeToString = (employee) ->
      employee.getName();
```

Ve výše uvedeném příkladu vytváříme implementaci rozhraní `Function` pomocí unární lambdy, která vrací hodnotu pole `name` v objektu `Employee`. Protože tato lambda se skládá z jednoho výrazu, nemusíme psát klíčové slovo `return`.

Supplier

`Supplier` je generické funkční rozhraní, jehož úkolem je poskytnout hodnotu objektu typu `T`. Metoda `get` vrací hodnotu implementovanou v rozhraní.

Následující příklad používá lambda výraz k implementaci rozhraní `Supplier`:

```
1 public class SupplierExample {
2   public static void main(String[] args) {
3     getValue(() -> "supplier test!");
4   }
5
6   static void getValue(Supplier<String> supplier){
7     System.out.println(supplier.get());
8   }
9 }
```

Consumer

`Consumer` (spotřebitel) je další generické funkční rozhraní. Představuje operaci, která přijímá jeden vstupní argument a nevrací žádný výsledek. Generický typ `T` je argument metody. Rozhraní se používá, když je potřeba objekt „spotřebovat“. Metoda `accept` (akceptovat) je zodpovědná za volání implementované metody, např.:

```
1 Consumer<String> stringTrim = (s) -> {
2     s = s.trim();
3     System.out.println(s);
4 };
```

Predicate

Funkční rozhraní `Predicate` představuje operaci, která přijímá jeden argument a vrací logickou hodnotu na základě předaného parametru. Jedná se tedy o specializaci funkčního rozhraní. Generický typ `T` je typ vstupního argumentu metody.
Testovací metoda je zodpovědná za vrácení logické hodnoty testu, např.:

```
1 Predicate<Integer> predicate = (value) -> {
2     return value >= 0;
3 };
```

Optional

V Javě může odkaz na objekt buď ukazovat na skutečný objekt, nebo může být prázdný, tj. `null`. Z pohledu programátora je nepohodlné pokaždé ověřovat platnost odkazu. Proto často provádíme takové ověření, např. když:

- dokumentace nám říká o možných hodnotách `null`
- relevantní části metody jsou označeny anotací, např. `Null` nebo `@Nullable`

Místo toho můžeme použít mechanismus zavedený v Javě 8. `Optional` v balíčku `java.util` je objekt, který obaluje cíl, tj. je to nějaký druh rámečku, který takový objekt může nebo nemusí obsahovat.

Optional

Třída `Optional` nám poskytuje přístup k mnoha metodám, včetně:

- `of`
- `ofNullable`
- `isPresent`
- `ifPresent`
- `orElse`
- `orElseGet`

Vytvoření objektu typu Optional

Statická metoda `of` nebo `ofNullable` je zodpovědná za vytváření objektů. Rozdíl mezi první a druhou metodou spočívá v tom, že první metoda neumožňuje nabývat hodnoty typu `null`.
Příklad ukazuje oba výše popsané způsoby vytváření:

```
1 // creating a box with an object of the String type
2 final Optional<String> stringOptional = Optional.of("Hello,
      World!");
3
4 // use ofNullable since value can be null.
5 // In this case, Optional will be an empty box
6 final Optional<String> optionalThatCanBeEmpty = Optional.
      ofNullable(value);
```

Kontrola hodnot

Třída `Optional` poskytuje metody, které umožňují provést operaci podmíněně, pokud skutečně obsahuje odkaz na objekt.

- Metoda `isPresent` vrací hodnotu `true`/`false` v závislosti na tom, zda třída `Optional` obsahuje nějaký objekt.
- Metoda `isEmpty` vrací hodnotu opačnou k hodnotě vrácené metodou `isPresent`, tj. poskytuje informaci, zda třída `Optional` neobsahuje žádný objekt.
- Metoda `ifPresent` bere jako argument rozhraní funkce `Consumer`. Tato třída `Consumer` bude volána pouze tehdy, když třída `Optional` obsahuje odkaz na objekt.

Kontrola hodnot

```
1  public class OptionalsPresenceExample {
2    public static void main(String[] args) {
3      final Optional<String> optional = getStringForEvenNumber
           (3);
4      if (optional.isPresent()) {
5        System.out.println("I am optional with a value, I am
             non empty box");
6      } else if (optional.isEmpty()) {
7        System.out.println("I am an empty optional");
8      }
9      // writing values to a box on screen, only if available
10     optional.ifPresent(System.out::println);
11   }
12   private static Optional<String> getStringForEvenNumber(
       final int number) {
13     if (number % 2 == 0) {
14       return Optional.of("even");
15     }
16     return Optional.empty();
17   }
```

Získání hodnoty

Třída `Optional` nabízí několik metod pro získání hodnoty:

- Metoda `get` vrací hodnotu uloženou v objektu, nebo v případě hodnoty `null` vyvolá výjimku: `NoSuchElementException`
- Metoda `orElse` vrací hodnotu uloženou v `Optional` nebo hodnotu uvedenou jako argument metody v případě prázdného `Optional`
- Metoda `orElseGet` vrací hodnotu uloženou v `Optional` nebo hodnotu uvedenou v `Supplier`, což je vstupní argument.

```
1 public class OptionalOrElseExample {
2   public static void main(String[] args) {
3     String object = null;
4     String name = Optional.ofNullable(object).orElse("john");
5     System.out.println(name); // the value john will be
          printed on the screen
6   }
7 }
```

```
1 public class OptionalOrElseGetExample {
```

Získání hodnoty

POZNÁMKA:

Pokud chceme použít metodu `get`, měli bychom nejprve ověřit dostupnost objektu pomocí metody `isPresent`.

POZNÁMKA:

Metoda `orElseGet` vypočítá náhradní hodnotu, na rozdíl od metody `orElse`, pouze pokud je proměnná `Optional` prázdná. To znamená, že výkon metody `orElseGet` je lepší než u metody `orElse`.

Cvičení

Úkol 1

Použijte mechanismy funkcionálního programování založené na dané struktuře a uveďte:

- seznam všech epizod
- seznam všech videí
- seznam názvů všech sérií
- seznam všech čísel sérií
- seznam názvů všech epizod
- seznam všech čísel epizod
- seznam všech názvů videí
- seznam všech adres URL pro každé video
- pouze epizody ze sudých sérií
- pouze videa ze sudých sérií
- pouze videa ze sudých epizod a sérií
- pouze klipová videa ze sudých a lichých sérií
- pouze náhledová videa z lichých epizod a sudých sérií

Streamy

Třídy, které přistupují k rozhraní `Stream` v balíčku `java.util.stream`, umožňují funkční způsob zpracování dat. Proudy (tzv. *streams*) představují sekvenční sadu prvků a umožňují provádět s těmito prvky různé operace.

Streamy a kolekce

Streamy jsou druhem reprezentace kolekce, ale existují mezi nimi určité rozdíly:

- Stream není datová struktura, která ukládá prvky. Pouze přenáší reference prvků ze zdroje, kterým může být například:
  - datová struktura
  - deska
  - kolekce
  - generátorová metoda.

Streamy a kolekce

Streamy jsou druhem reprezentace kolekce, ale existují mezi nimi určité rozdíly:

- Streamové operace vracejí výsledek, ale nemodifikují zdroj Streamu, např. filtrování streamu získaného z kolekce vytvoří nový stream bez filtrovaných položek a žádná položka se neodstraní z původní kolekce.
- Kolekce MAJÍ konečnou velikost, streamy nemusí mít konečnou velikost. Operace jako `limit(n)` nebo `findFirst()` umožňují dokončení výpočtu v streamech v konečném čase.
- Položky v streamech jsou během streamu navštíveny pouze jednou. Stejně jako u Iterátoru je nutné vygenerovat nový stream, aby se znovu navštívily stejné prvky zdroje, např. kolekce.
- Streamy jsou zpracovávány líně, to znamená, že žádná z nepřímých metod není volána, dokud není volán jeden z terminátorů.

Způsoby, jak získat streamy

V Javě existuje mnoho způsobů, jak přijímat streamy na základě vybraných kolekcí, polí a objektů:

- Metoda `stream()` vrací stream pro třídy dostupné v Collection API
- Metoda `Arrays.stream(Object[])` umožňuje vytvářet streamy z polí
- Statická metoda `Stream.of(T ... values)` umožňuje vytvářet streamy na základě polí a objektů
- Statická metoda `Stream.generate()` umožňuje vytvořit stream prvků na základě vstupu `Supplier`.
- Souborové streamy lze vrátit na základě třídy `Files`.

Způsoby, jak získat streamy

```
1 Stream<Integer> streamOfInts = Arrays.asList(1, 2, 3).stream
      ();
2 Stream<String> streamOfStrings = Set.of("one", "two", "three"
      ).stream();
3 Stream<Map.Entry<String, Integer>> stream = Map.of("someKeyA"
      , 1, "someKeyB", 2).entrySet().stream();
4 IntStream arraysStream = Arrays.stream(new int[]{1, 2, 3});
5 Stream<Double> ofStream = Stream.of(1.1, 2.2, 3.3);
6 Stream<Integer> generateStream = Stream.generate(() -> new
      Random().nextInt());
7 Stream<String> fileLinesStream = Files.lines(Path.of("/tmp/1.
      txt"));
```

Operace s proudy

Operace s proudy se dělí na dva typy – mezilehlé a koncové.
Nepřímé operace vždy vracejí nový proud a jsou zpracovávány líně. Patří mezi ně mimo jiné:

- `filter`
- `map`
- `flatMap`
- `peek`
- `distinct`
- `sorted`
- `limit`

Operace s proudy

Post-operace jsou operace, které vracejí konečný výsledek. Jejich volání způsobí spuštění všech předchozích mezilehlých funkcí. Mezi ukončující funkce patří:

- `toArray`
- `collect`
- `count`
- `reduce`
- `forEach`
- `forEachOrdered`
- `min`
- `max`
- `anyMatch`
- `allMatch`
- `noneMatch`
- `findAny`
- `findFirst`

Nepřímé operace – map

Metoda `map` čeká na vstup z objektu `Function <T, R>`. Jejím úkolem je převést element stream na nový element, který může být navíc jiného typu.

```
1 // creating a stream and processing the input elements of the
      Integer type to a value three times greater of the Double
      type
2 List.of(1, 2, 3).stream()
3     .map(streamElem -> streamElem * 3.0);
```

Nepřímé operace – flatMap

Metoda `flatMap` umožňuje zploštit vnořenou datovou strukturu. To znamená, že pokud každý zpracovaný prvek obsahuje prvek, ze kterého můžeme vytvořit nový `Stream`, pak výsledkem operace `flatMap` bude nový jediný `Stream`, který byl vytvořen jejich sloučením do jednoho. Operace `flatMap` přebírá funkci `interfaceFunction <T,? extends Stream <? extends R>>`. Příklad ukazuje, jak vytvořit jeden stream z objektů `Statistics` a polí `values`.

```
1 public class FlatMapDemo {
2   public static void main(String[] args) {
3     final Statistics statisticsA = new Statistics(2.0, List.
          of(1, 2, 3));
4     final Statistics statisticsB = new Statistics(2.5, List.
          of(2, 3, 2, 3));
5     Stream.of(statisticsA, statisticsB)
6         .flatMap(statistics -> statistics.getValues().stream
              ());
7     // We get a stream of values 1, 2, 3, 2, 3, 2, 3
8   }
```

Nepřímé operace – flatMap

```
1  class Statistics {
2    private double average;
3    private List<Integer> values;
4
5    public Statistics(final double average, final List<Integer>
         values) {
6      this.average = average;
7      this.values = values;
8    }
9
10   public double getAverage() {
11     return average;
12   }
13
14   public List<Integer> getValues() {
15     return values;
16   }
17 }
```

Nepřímé operace – filter

Operace `filter` umožňuje odstranit z proudu ty prvky, které nesplňují určitý predikát, který je vstupním argumentem metody. Následující příklad ukazuje, jak odstranit lichá čísla z proudu čísel.

```
1 final int[] idx = { 0 };
2 Stream.generate(() -> idx[0]++)
3     .limit(10)
4     .filter(elem -> elem % 2 == 0);
5     //the following values remain in the stream: 0, 2, 4, 6,
          8
```

Nepřímé operace – sorted

Metoda `sorted` seřadí položky v proudu. K dispozici je verze bez argumentů, která třídí prvky *přirozeně*. Pokud chceme třídit prvky podle jiného pravidla, měli bychom použít přetížení, které využívá rozhraní funkce `Comparator <T>`.

```
1 Arrays.asList(6, 3, 6, 21, 20, 1).stream()
2     .sorted(Comparator.reverseOrder());
3     // in the stream you will find: 21, 20, 6, 6, 3, 1 – in
          this order
```

Nepřímé operace – distinct

Operace distinct umožňuje vytvořit stream, ve kterém jsou všechny prvky jedinečné, tj. zbavíme se opakování, např.:

```
1 Arrays.asList(3, 6, 6, 20, 21, 21).stream()
2     .distinct();
3     // there will be items in the stream: 3, 6, 20, 21
```

Terminální operace – forEach

Operace `forEach` představuje funkcionální verzi cyklu `for`. Tato funkce volá jakoukoli operaci implementovanou s funkčním rozhraním `Consumer` na každém prvku streamu.

```
1 List.of(1, 2, 3, 4, 5).stream()
2     .forEach(System.out::println);
```

Terminální operace – collect

Metoda `collect` umožňuje shromažďovat položky streamu do určitého cíle. Pro shromažďování položek potřebujeme rozhraní `Collector`, které **není** funkcionálním rozhraním. Třída `Collectors` se hodí, protože obsahuje statické metody zodpovědné za akumulaci prvků streamu do uvedené struktury, např. `List` nebo `Set`.

```
1 final List<Integer> listCreatedFromCollectMethod = Stream.
      generate(() -> new Random().nextInt())
2     .limit(10)
3     .distinct()
4     .filter(elem -> Math.abs(elem) < 1000)
5     .collect(Collectors.toUnmodifiableList());
```

Terminální operace – groupingBy

Kolektor získaný pomocí metody `groupingBy` nám umožňuje vytvořit objekt `Map` ze streamu. Metoda `groupingBy` očekává funkci `Function`, která se stane nějakou vlastností streamu. Výsledkem je mapa, jejíž klíče jsou hodnoty výše zmíněné vlastnosti a hodnota je seznam prvků, které tuto vlastnost splňují. Nejlépe to ilustruje příklad, např. níže uvedený kód vytvoří mapu, která seskupuje slova podle jejich délky v streamu:

```
1  Stream.of("This", "is", "ICT", "the", "best", "academy", "in"
       , "the", "universe")
2       .collect(Collectors.groupingBy(String::length))
3       .forEach((key, value) -> System.out.println(key + " " +
            value));
4
5  /* the result is:
6  2 [is, in]
7  3 [ICT, the, the]
8  4 [This, best]
9  7 [academy]
10 8 [universe]*/
```

Terminální operace – `findFirst`

Metoda `findFirst` dokončí zpracování streamu a načte první dostupnou položku. Protože tuto metodu lze volat i na prázdný stream, návratový typ je `Optional`, např.:

```
1 List.of("who", "will", "be", "first").stream()
2     .sorted()
3     .findFirst() // returns Optional
4     .ifPresent(System.out::println); // will display "be"
```

Terminální operace – `findAny`

Funkce `findAny`, stejně jako `findFirst`, vrací jeden element streamu (jako objekt zabalený v `Optional`), ale v tomto případě si nejsme jisti, který element streamu bude vrácen, pokud je v něm více elementů.

```
1 List.of(7, 21, 13, 4, 8).stream()
2     .filter(x -> x % 2 == 0)
3     .findAny()
4     .ifPresent(System.out::println);
```

Terminální operace – reduce

Operace `reduce` umožňuje získat jeden výsledek ze všech prvků proudu. Metoda `reduce` přijímá dva argumenty:

- počáteční hodnotu
- transformační metodu, tj. informaci o tom, jak změnit aktuální výsledek další zpracovanou položkou. Tyto informace ukládáme pomocí funkčního rozhraní `BiFunction <T, U, R>`.

Příklad ukazuje, jak sečíst prvky proudu:

```
1 final Integer sum = List.of(2, 5, 9, 19, 14).stream()
2     .reduce(0, (currentSum, streamElement) -> currentSum +
          streamElement);
3     // or Integer::sum
4 System.out.println(sum); // the result is a sum - 49
```

Paralelní zpracování

Řídicí příkazy typu `for` jsou sekvenční povahy. Streamy naopak umožňují snadnější paralelizační operace. K tomu potřebujeme vytvořit speciální stream. Můžeme ho vytvořit například:

- voláním metody `parallelStream()` na kolekci
- voláním statické metody `StreamSupport.stream()` se zadáním hodnoty `true` pro argument `parallel`
- voláním metody `parallel()` na existujícím streamu.

Následující příklad ukazuje stream, který paralelně zpracovává prvky:

```
1 final List<String> result = Arrays.asList("Alice has a cat
    named Catson".split(" ")).parallelStream()
2     .sorted()
3     .map(String::toUpperCase)
4     .collect(Collectors.toList());
```

Paralelní zpracování

POZNÁMKA:

Mezi paralelním zpracováním streamu a jeho sekvenční formou není žádný rozdíl, není žádný rozdíl ani na úrovni volání mezilehlé a terminační metody.

POZNÁMKA:

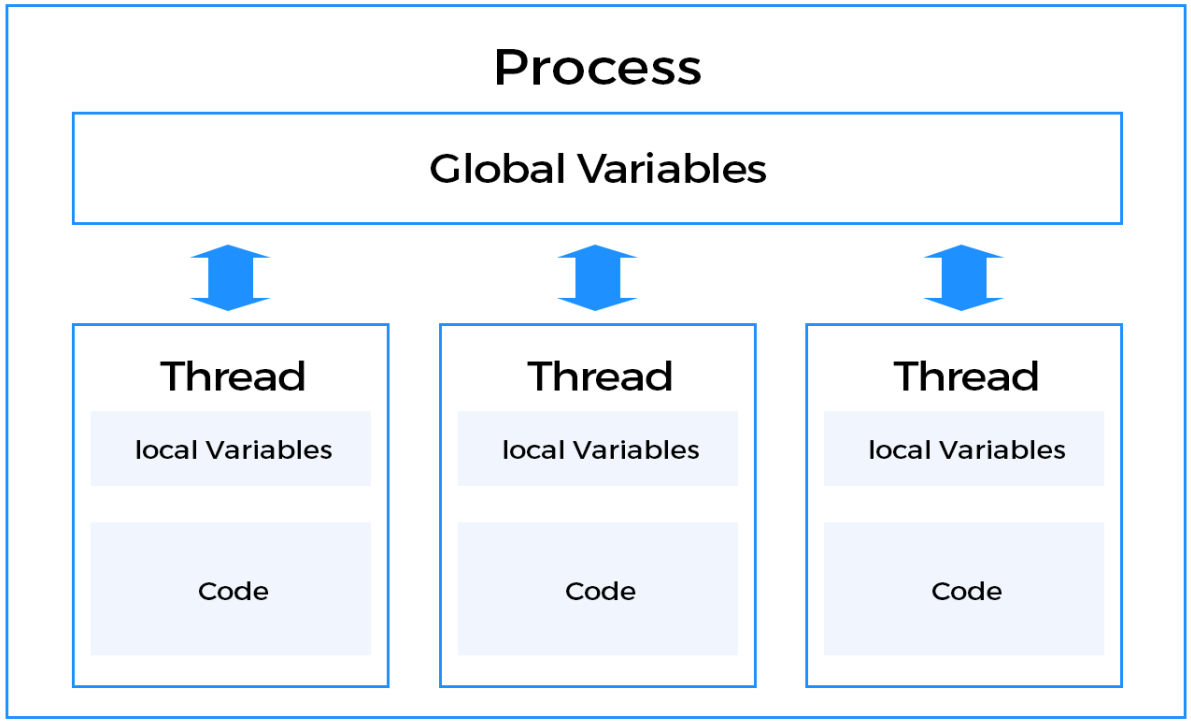
Z paralelního streamu na sekvenční stream můžete „přejít“ pomocí metody `sequencial()`.

Souběžné a paralelní programování

Souběžné programování zahrnuje návrh a tvorbu programů, které se ve fázi provádění skládají z alespoň dvou souběžných jednotek (každá z nich je sekvenčním procesem) a zároveň zajišťuje synchronizaci mezi nimi. Těmito entitami mohou být *vlákna (threads)* nebo *procesy (processes)*.

Vlákno a proces

Když spouštíme naše aplikace, spouštíme nový proces na úrovni operačního systému. Seznam takových procesů můžeme získat pomocí příkazu `ps`. V rámci jednoho procesu může existovat více vláken. Vlákna mají *společný* adresní prostor a otevřené systémové struktury (například otevřené soubory), procesy zase mají *nezávislé* adresní prostory.



Vlákno (Thread)

Vlákno je spouštěcí vlákno v programu. Virtuální stroj Java umožňuje aplikaci spouštět více vláken současně. Každé vlákno má prioritu. Vlákna s vyšší prioritou běží před vlákny s nižší prioritou.
Když se JVM spustí, obvykle se spustí hlavní vlákno, které volá metodu `main`. Poté můžeme z hlavního vlákna spustit další vlákna. Tato vlákna běží, dokud nenastane jedna z následujících situací:

- vlákna dokončí svou práci
- vlákno vyvolá výjimku
- bude volána metoda `System.exit()` (z libovolného vlákna).

Vlákno Java

Vlákno Java je reprezentováno třídou `Thread`. Vlákna můžeme vytvářet mnoha způsoby, např.:

- rozšířením třídy `Thread` a přepsáním metody `run`
- implementací funkčního rozhraní `Runnable`.

Dědění z třídy Thread

Pokud chcete definovat nové vlákno a rozhodnete se rozšířit třídu `Thread`, veškerý kód, který by se měl spustit v samostatném vlákně, musí jít do metody `run`. Příklad ukazuje vytvoření nového vlákna rozšířením třídy `Thread`. Vlákno je navíc spuštěno pomocí metody `start()`. Hlavní vlákno a námi vytvořené vlákno na konci své operace vypíší svůj identifikátor pomocí `Thread.currentThread().GetId()`:

```
1  public class ThreadsExample {
2    public static void main(String[] args) {
3      new HelloWorldThread().start();
4      System.out.println(Thread.currentThread().getId());
5    }
6  }
7
8  class HelloWorldThread extends Thread {
9    @Override
10   public void run() {
11     System.out.println("Hello World from another Thread");
12     System.out.println(Thread.currentThread().getId());
```

Runnable

Dalším, lepším způsobem, jak vytvořit vlákno, je deklarovat třídu, která implementuje rozhraní `Runnable` s jednou abstraktní metodou `run` bez argumentů. Instanci třídy `Runnable` lze předat jako argument konstruktoru třídy `Thread`. Vytvořené vlákno spouštíme pomocí metody `start`.

Runnable

Následující příklad spouští dvě samostatná vlákna pomocí rozhraní `Runnable`, jedno definováním samostatné třídy a podruhé pomocí lambda:

```
1  public class ThreadsExample {
2    public static void main(String[] args) {
3      new Thread(new HelloWorldRunnableThread()).start();
4      new Thread(() -> System.out.println("Hello from another
           thread implemented with lambda")).start();
5    }
6  }
7
8  class HelloWorldRunnableThread implements Runnable {
9    @Override
10   public void run() {
11     System.out.println("Hello World from another Thread");
12   }
13 }
```

Přerušení vlákna

V závislosti na okolnostech a stavu aplikace můžeme občas chtít přerušit práci vykonávanou určitým vláknem. Vlákno však není možné přerušit triviálním způsobem. Třída `Thread` poskytuje metodu `stop`, ale **neměli bychom ji používat**. Maximálně můžeme odeslat požadavek na zastavení takového vlákna. Programátor v kódu samostatného vlákna rozhodne, co s touto skutečností udělá. Takový signál odešleme voláním metody `interrupt`, která je k dispozici na instanci třídy `Thread`. V závislosti na stavu vlákna existují dvě možné situace:

- vlákno na základě signálu vyvolá výjimku `InterruptedException`
- vlákno může zkontrolovat, zda přijalo signál (požadavek na zastavení), metodou:
  - `isInterrupted`, kde se při volání smaže informace o požadavku na zastavení, nikoli informace
  - `interrupted`, která kromě informace o tom, zda byl odeslán signál na zastavení, také resetuje stav.

Přerušení vlákna

```
1  public class ThreadsExample {
2    public static void main(String[] args) {
3      final Thread sleepingThread = new Thread(new
           SleepingThread());
4      sleepingThread.start();
5      sleepingThread.interrupt(); // sending a stop request
6    }
7  }
8  class SleepingThread implements Runnable {
9    @Override
10   public void run() {
11     System.out.println("I will go to sleep");
12     try {
13       Thread.sleep(3000L);
14     } catch (InterruptedException e) {
15       System.out.println("I was interrupted during sleep");
16     }
17     System.out.println("I am exiting");
18   }
```

Přerušení vlákna

Následující příklad vytvoří samostatné vlákno, které během svého provádění kontroluje, zda byl mezitím odeslán požadavek na zastavení:

```
1  public class ThreadsExample {
2    public static void main(String[] args) {
3      final Thread sleepingThread = new Thread(new
           SleepingThread());
4      sleepingThread.start();
5      sleepingThread.interrupt();
6    }
7  }
8
9  class SleepingThread implements Runnable {
10
11   @Override
12   public void run() {
13     final List<Integer> ints = new ArrayList<>();
14     for (int idx = 0; idx < 1000; idx++) {
15       ints.add(new Random().nextInt());
16     }
```

Synchronizace

Při vytváření vícevláknové aplikace musíme mít na paměti, že v takové aplikaci:

- existuje jedna halda bez ohledu na počet vláken
- každé běžící vlákno vytváří samostatný zásobník (stack)

Proto musíme ve vícevláknové aplikaci vzít v úvahu skutečnost, že objekt na haldě může být v daném okamžiku změněn mnoha vlákny. Abychom se tomu vyhnuli, tj. k objektu lze přistupovat pouze v jednom vlákně současně, můžeme použít mechanismus synchronizace.
Java zavádí dva základní způsoby synchronizace:

- synchronizace metod
- synchronizace bloků kódu

Obě výše uvedené metody jsou implementovány pomocí klíčového slova `synchronized`.

Synchronizace

Problém se synchronizací je vidět v následujícím úryvku kódu. Vytvoříme v něm dvě vlákna, která modifikují **tutéž** instanci třídy `Pair`. Na konci každého vlákna vypíšeme na obrazovku konečné hodnoty levého a pravého pole. I když jsme program spustili s hodnotami 0 a 0 a obě vlákna je 100krát inkrementovala, s největší pravděpodobností pro tato pole nedostaneme hodnotu 200.

Synchronizace

Problém popsaný v příkladu spočívá v tom, že inkrementace je v kontextu kódu, který píšeme, jedna instrukce, ale pro procesor jsou to ve skutečnosti *tři* instrukce. Jsou to:

1. načtení aktuální hodnoty z paměti
2. přičtení jedné k načtené hodnotě
3. uložení zvýšené hodnoty do paměti.

Pokud jedno vlákno provede operaci (1), ale ještě neprovede operaci (3), a během této doby se druhému vláknu podaří provést operaci (1), dva inkrementy povedou ke zvýšení hodnoty o 1, nikoli o 2.

Synchronizace metody

Abychom mohli synchronizovat metodu, přidáme do její deklarace klíčové slovo `synchronized`. Když je metoda synchronizovaná, volající vlákno k ní má exkluzivní přístup, dokud se nedokončí. Abychom problém v předchozím příkladu vyřešili, musíme synchronizovat následující metody:

```
1 public synchronized void incrementLeft() {
2   left++;
3 }
4
5 public synchronized void incrementRight() {
6   right++;
7 }
```

Díky tomuto řešení je programátor schopen řídit přístup k objektu, což jej činí předvídatelnějším a sekvenčnějším napříč více vlákny.

Synchronizace bloků

Synchronizace bloků provádí přesně stejný mechanismus jako synchronizace metod, ale může omezit rozsah synchronizace dat pouze na jednotlivé instrukce související např. s polem třídy. Pro implementaci synchronizace bloků se také používá klíčové slovo `synchronized`, ale navíc pomocí funkce `between()` vkládáme objekt, ke kterému chceme přistupovat souběžně, sekvenčním způsobem.

Synchronizace bloků

Pokud bychom se rozhodli použít synchronizaci bloků kódu v synchronizovaných metodách z předchozího příkladu, mohli bychom změnit jejich implementaci, např. na:

```
1  public void incrementLeft() {
2      System.out.println("Out of synchronized block");
3      synchronized (this) {
4          left++;
5          System.out.println("In synchronized block");
6      }
7      System.out.println("Out of synchronized block");
8  }
9  public void incrementRight() {
10     System.out.println("Out of synchronized block");
11     synchronized (this) {
12         right++;
13         System.out.println("In synchronized block");
14     }
15     System.out.println("Out of synchronized block");
16 }
```

Spojení (Join)

V aplikacích často používáme další vlákna k výpočtu určitých dat, která pak zpracováváme, např. v hlavním vlákně. Než můžeme začít se zpracováním, jsme nuceni čekat na dokončení všech vláken, která zpracovávají data. Abychom počkali na ukončení vlákna, musíme použít metodu spojení (`join`). K dispozici jsou přetížení:

- bez argumentu, čekání na ukončení vlákna
- verze s argumenty, kde můžeme zadat počet milisekund (a volitelně nanosekund), což znamená maximální dobu čekání na ukončení vlákna.

Zablokování (*Deadlock*)

V situaci, kdy se několik vláken navzájem neomezeně blokuje, se to nazývá zablokování (*deadlock*). Program není schopen dokončit indikovanou operaci kvůli trvalému vzájemnému zablokování zdrojů. Lze to popsat následovně:

- A čeká na B, protože:
- B čeká na A.

V uvedeném případě:

- Vlákno `t1` blokuje přístup ke zdroji `r1` a poté se pokouší vynutit přístup ke zdroji `r2`.
- Vlákno `t2` blokuje přístup ke zdroji `r2` a poté se pokouší o přístup k zdroji `r1`.

Jelikož ani vlákno `t1` neuvolní přístup ke zdroji `r1`, ani vlákno `t2` neuvolní přístup k `r2`, způsobí to tzv. `deadlock`.

Koordinace vláken

Klíčové slovo `synchronized` se používá k zabránění nežádoucí interakci vláken. Není to však dostatečné opatření k zajištění toho, aby vlákna spolupracovala. Často může být nutné neprovádět určitou operaci, dokud není splněna určitá podmínka.

```
1 Queue<Runnable> runnableQueue = new LinkedList<>();
2 while (consumerQueue.isEmpty()) {
3   // waiting, waiting and still waiting for something to
        appear
4 }
5 actionQueue.poll().run();
```

Výše uvedený fragment kódu řeší uvedený problém, ale je také velmi neefektivní, protože se během čekání provádí nepřetržitě. Stejný problém lze vyřešit metodami `wait` a `notify`/`notifyAll`.

wait

Vlákno volá metodu `wait` na daném objektu, když očekává, že se něco stane (obvykle v kontextu tohoto objektu), např. změnu stavu objektu, kterou provede jiné vlákno a která je implementována např. změnou hodnoty nějaké proměnné - pole objektu). Volání metody `wait` blokuje vlákno a metoda, na které je operace volána, **musí** být synchronizována. Jiné vlákno může změnit stav objektu a upozornit na to čekající vlákno (pomocí metody `notify` nebo `notifyAll`).

```
1 Queue<Runnable> runnableQueue = new LinkedList<>();
2 while (runnableQueue.isEmpty()) {
3   try {
4     wait();
5   } catch (InterruptedException e) {
6     System.err.println("Oops");
7   }
8 }
9 runnableQueue.poll().run();
```

notify a notifyAll

Objekt je odblokován, když jiné vlákno zavolá metodu `notify` nebo `notifyAll` pro stejný objekt, kde vlákno čeká:

- Volání `notify` odblokuje jedno z čekajících vláken, kterým může být **libovolné** z nich.
- Metoda `notifyAll` odblokuje všechna vlákna čekající na objekt.
- Volání metody `notify` nebo `notifyAll` musí být v synchronizovaném bloku / metodě.

Ve výše uvedeném příkladu je vlákno `WithdrawThread` pozastaveno (`wait`), dokud nebudete mít na účtu dostatek finančních prostředků. Každá platba spustí vlákno (`notify`). Vlákno `WithdrawThread` nedokončí svůj chod, dokud daný `Customer` nebude mít požadované finanční prostředky.

Callable a Future

Obecné funkční rozhraní reprezentující úlohu, která může vrátit buď výsledek, nebo výjimku pomocí metody `call()` bez argumentů. Rozhraní `Callable` je podobné rozhraní `Runnable`, až na to, že metoda `run` nemůže vrátit žádný výsledek. Obě rozhraní jsou si navzájem podobná kvůli jejich potenciálnímu využití ve vícevláknových službách.

```
1 public class GetRequest implements Callable<String> {
2
3     @Override
4     public String call() throws Exception {
5         return "Dummy http response";
6     }
7 }
```

Callable a Future

`Future` je rozhraní, které představuje *budoucí* výsledek metody `async`, který bude nakonec vrácen v budoucnosti po dokončení zpracování operace. Hodnotu operace `operation` lze načíst pomocí metody `get()`, která funguje podobně jako metoda `join` ve třídě `Thread`, tj. blokuje aktuální vlákno a čeká na očekávaný výsledek.

ExecutorService

Při vytváření vícevláknových aplikací zřídka používáme nízkoúrovňové API a vlákna spravujeme ručně. Kdykoli je to možné, měli bychom používat tzv. pool vláken (`thread pool`), což je skupina vláken spravovaných externí entitou. Jedním z takových mechanismů v Javě je rozhraní `ExecutorService`, které zjednodušuje provádění úloh v asynchronním režimu a k tomu využívá pool vláken. Pro vytvoření instance `ExecutorService` můžeme použít továrnu, třídu `Executors`, která má několik užitečných statických metod. Základní jsou:

- `newSingleThreadExecutor()` – vrací `ExecutorService` běžící na jednom vlákně
- `newFixedThreadPool(int nThreads)` – vrací `ExecutorService` běžící na poolu vláken dané velikosti.

ExecutorService

Kromě toho máme také:

- `newCachedThreadPool()` – vytvoří `ExecutorService`, která by v případě absence vlákna mohla zpracovat novou úlohu, přidá do poolu nové vlákno. Vlákna jsou navíc z poolu odebrána, pokud po dobu jedné minuty nedostane novou úlohu k provedení.
- `newScheduledThreadPool(int corePoolSize)` – vytvoří `ExecutorService`, která spustí úlohu po určitém čase nebo v zadaných intervalech.

ExecutorService – příklad

Následující kód ukazuje různé způsoby, jak vytvořit různé instance ExecutorService:

```
1 public class ExecutorsCreationExample {
2   public static void main(String[] args) throws
        InterruptedException {
3     final int cpus = Runtime.getRuntime().availableProcessors
          ();
4     final ExecutorService singleThreadES = Executors.
          newSingleThreadExecutor(); // single thread pool
5     final ExecutorService executorService = Executors.
          newFixedThreadPool(cpus); // pool with threads equal
          to cpu
6     final ExecutorService cachedES = Executors.
          newCachedThreadPool();              // cached thread pool
7     final ScheduledExecutorService scheduledExecutorService =
           Executors.newScheduledThreadPool(cpus); // scheduled
          thread pool with cpu equal number of threads
8   }
9 }
```

Zavření služby ExecutorService

Při vytváření služby `ExecutorService` si musíme pamatovat na její ruční zavření. K tomu se používají následující metody:

- `shutdown()` – fond vláken přestane přijímat nové úlohy, ty spuštěné budou dokončeny a poté bude fond uzavřen
- `shutdownNow()` – podobně jako `shutdown`, `ExecutorService` přestane přijímat nové úlohy, navíc se pokusí zastavit všechny aktivní úlohy, zastaví zpracování čekajících úloh a vrátí seznam úloh čekajících na provedení.

Provádění úloh

Pro provedení úlohy na vlákně z fondu můžeme použít následující metody:

- `submit()` – provede úlohu typu `Callable` nebo `Runnable`,
- `invokeAny()` – `ExecutorService` ve svém fondu vláken spustí provádění seznamu vstupních úloh. Vrátí výsledek úloh, které byly spuštěny a úspěšně dokončeny, když byla první úloha úspěšně dokončena. Zbývající nedokončené úlohy budou zrušeny.
- `invokeAll()` – provede všechny volatelné úlohy a vrátí seznam výsledků typu typeList `<Future <T>>`.

Provádění úloh

POZNÁMKA: `invokeAny()`

Všimněte si, že na obrazovce se nezobrazilo „Room cleaned“, protože výsledek první dokončené úlohy byl vrácen rychleji. Prvním výsledkem je „washed dishes“. Tato skutečnost je způsobena tím, že náš pool má méně vláken, než je počet úloh, které chceme spustit. Proto i přes to, že třetí volatelná funkce spí nejkratší dobu, začne se provádět pouze tehdy, když je jedno z vláken z poolu volné, tj. dokončí provádění aktivní úlohy.

POZNÁMKA: `invokeAll()`

Všimněte si, že všechny úkoly budou vždy dokončeny.

ScheduledExecutorService

Tato implementace `ExecutorService` umožňuje naplánovat spuštění operace po určitém čase nebo intervalu. V rámci metod této implementace `ExecutorService` můžeme rozlišit následující metody:

- `scheduleAtFixedRate` – tato metoda umožňuje provést danou akci se zpožděním a poté cyklicky po určitém časovém intervalu.
- `scheduleWithFixedDelay` – tato metoda umožňuje provést danou akci se zpožděním a poté cyklicky v určitém časovém úseku. Jediný rozdíl je v tom, že časový interval se počítá od konce předchozí úlohy, nikoli od začátku.

Každá z výše uvedených metod vrací speciální objekt: `ScheduledFuture`, který dědí z chování třídy `Future` a kromě možnosti zrušení úlohy umožňuje vrátit čas zbývající do provedení další operace.

Atomické proměnné

Operace jako inkrement z pohledu Javy je jeden výraz, ale pro procesor se jedná o několik operací. Operaci nazýváme atomickou, pokud během jejího provádění nemůže jiné vlákno číst ani měnit hodnoty měněných proměnných. Balíček `java.util.concurrent.atomic` definuje třídy, které zpracovávají atomické operace s jednotlivými proměnnými. Třídy skupiny `Atomics` poskytují sadu synchronizovaných operací a samotné objekty lze bezpečně sdílet mezi více vlákny. Často používané typy jsou například:

- `AtomicInteger`
- `AtomicLong`
- `AtomicBoolean`

volatile

Jednoduše řečeno, vytvořením proměnné v programu lze tuto proměnnou uložit do hlavní paměti programu nebo pro optimalizaci do paměti procesoru (tzv. L2 Cache). U vícevláknových aplikací je možné, že hodnota proměnné uložené v paměti procesoru se liší od hodnoty uložené v hlavní paměti. Hodnota v hlavní paměti může být zastaralá a aktuální hodnota v paměti procesoru není pro některá vlákna k dispozici, takže naše aplikace nemusí fungovat podle očekávání.

Výše popsaný problém lze vyřešit označením takové proměnné klíčovým slovem `volatile`, což znamená, že hodnota bude vždy uložena pouze v hlavní paměti aplikace. Důležité je, že `volatile` nezaručuje atomicitu operace, tj. jedná se o způsob synchronizace, který produkuje ne méně než atomické proměnné nebo klíčové slovo `synchronized`.

Cvičení – úkol 1

Napište program, který bude paralelně vyhledávat sudá čísla ve dvou intervalech: 1000-2000 a 14300-17800.

Cvičení – úkol 2

Napište program, který vyřeší níže uvedený problém.
Na silnici mezi městy `A` a `B` je most, na kterém může být pouze jedno auto.
Implementujte mechanismus, který umožní synchronizovaný přístup objektu typu auto k objektu třídy `Bridge`.
Třída `Car` by měla obsahovat následující data:

- název auta
- typ auta

Třída `Bridge` může obsahovat následující metodu:

- `driveThrough`, která bude jako parametr akceptovat objekt třídy `Car`. Cesta by měla trvat 5 sekund.

Cvičení – úkol 3

Napište program, který spustí dva nezávislé třídicí algoritmy ve dvou samostatných vláknech. Hlavním cílem implementace je vrátit informace o algoritmu, který byl dokončen rychleji.

Cvičení – úkol 4

Napište program, který bude synchronizovat přístup k bankovnímu účtu. Pokud chce jakákoli cyklická internetová služba naúčtovat na účet vyšší částku, než je aktuálně k dispozici, pak by mělo být vlákno pozastaveno. Jakmile budou na účet převedeny další peníze, mělo by být vlákno znovu spuštěno.

Cvičení – úkol 5

Napište datovou strukturu, která vám umožní procházet pole dvěma směry:

- vpřed (`next()`)
- zpět (`prev()`)

Datová struktura by měla uchovávat aktuálně prohledávaný index.
Postarejte se prosím o jeho dodatečnou synchronizaci.

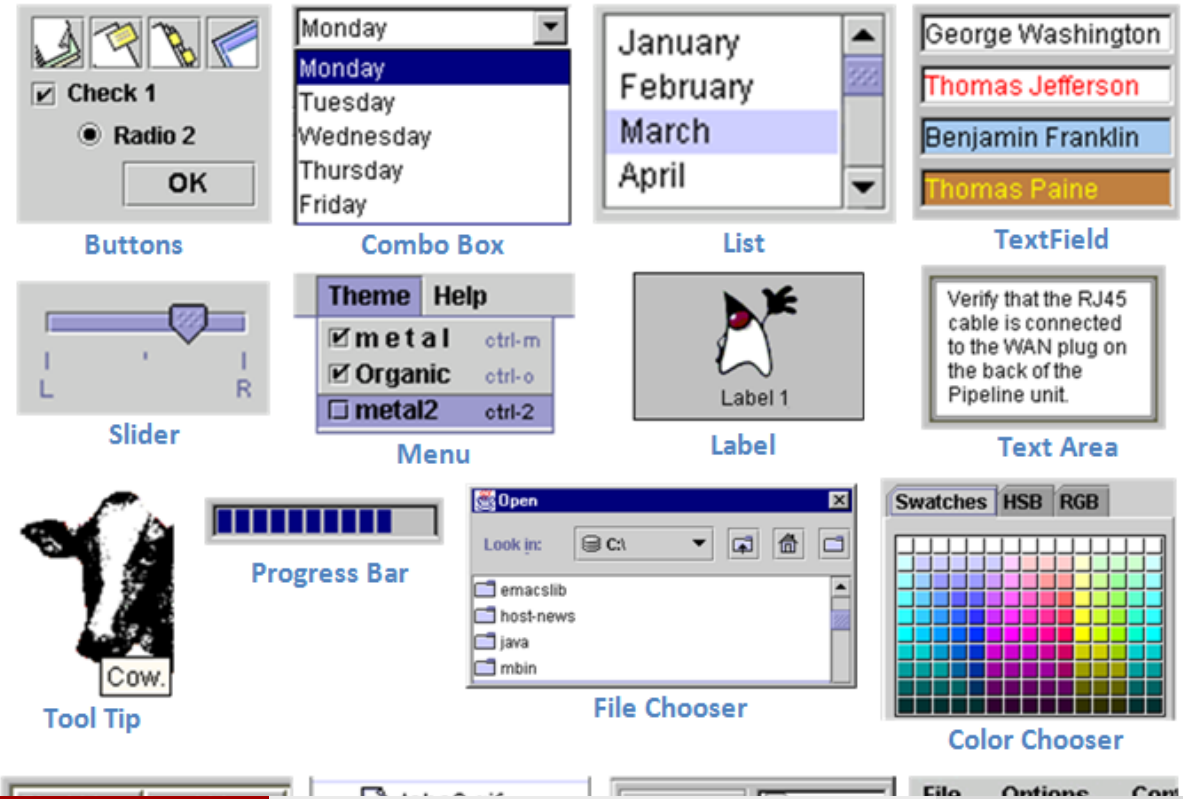
Swing GUI

Cílem programování grafického uživatelského rozhraní v Javě je umožnit programátorovi vytvářet grafické uživatelské rozhraní, které vypadá dobře na VŠECH platformách. `AWT` v JDK 1.0 byl neohrabaný a neobjektově orientovaný (používal mnoho funkcí `event.getSource()`). `AWT` v JDK 1.1 zavedl model delegace událostí (řízený událostmi), který byl mnohem přehlednější a objektově orientovaný. JDK 1.1 také zavedl vnitřní třídy a `JavaBeans` – model programování komponent pro vizuální programovací prostředí (podobný Visual Basic).

`Swing` se objevil po JDK 1.1. Do JDK 1.1 byl zaveden jako součást doplňku JFC (*Java Foundation Classes*). Swing je bohatá sada snadno použitelných a srozumitelných komponent grafického uživatelského rozhraní JavaBean, které lze přetahovat jako „stavitele grafického uživatelského rozhraní“ ve vizuálním programovacím prostředí. Swing je nyní nedílnou součástí Javy od JDK 1.2.

Vlastnosti Swingu

Swing je obrovský (skládá se z 18 balíčků se 737 třídami, stejně jako v JDK 1.8) a má velkou hloubku. Ve srovnání s AWT nabízí Swing obrovskou a komplexní kolekci opakovaně použitelných komponent grafického uživatelského rozhraní, jak je znázorněno na obrázku níže (převzato ze Swing Tutorial).



Vlastnosti Swingu

Hlavní vlastnosti Swingu jsou (převzato z webových stránek Swingu):

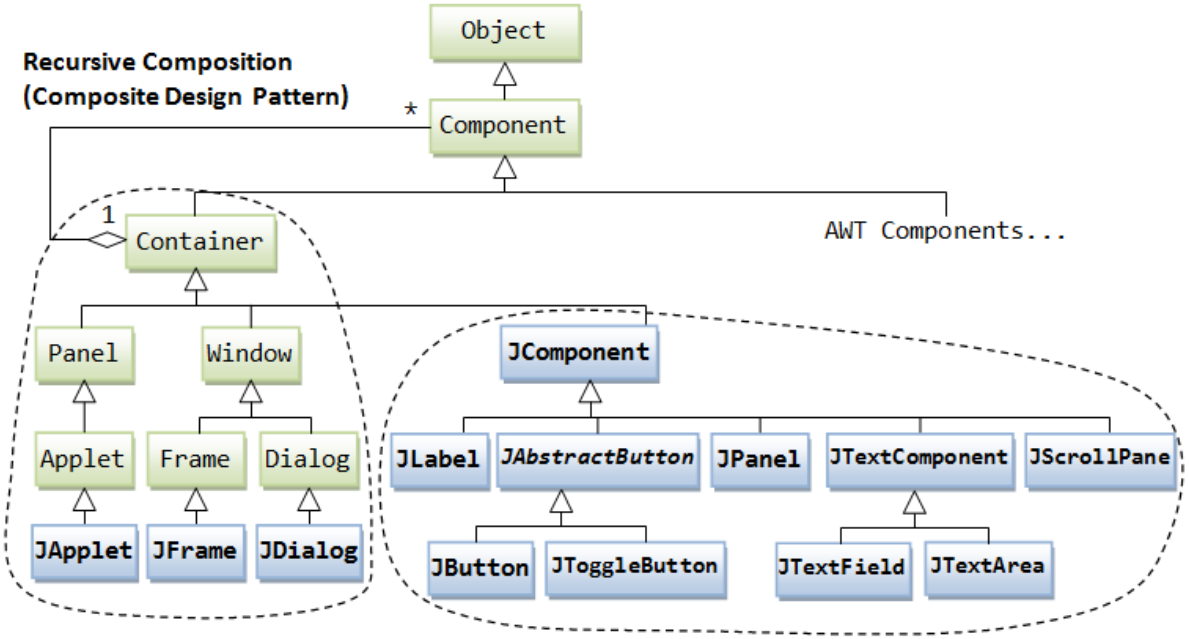
- `Swing` je napsán v čisté Javě (s výjimkou několika tříd), a proto je 100% přenositelný.
- Komponenty `Swingu` jsou lehké. Komponenty `AWT` jsou těžké (z hlediska využití systémových zdrojů). Každá komponenta `AWT` má svůj vlastní neprůhledný nativní displej a vždy se zobrazuje nad lehkými komponentami. Komponenty `AWT` se silně spoléhají na podkladový okenní subsystém nativního operačního systému. Například tlačítko `AWT` se váže na skutečné tlačítko v podkladovém nativním okenním subsystému a spoléhá se na nativní okenní subsystém pro své vykreslování a zpracování. Komponenty Swingu (`JComponents`) jsou napsány v Javě. Obecně nejsou „zatíženy“ složitými požadavky grafického uživatelského rozhraní, které podkladový okenní subsystém klade.

Vlastnosti Swingu

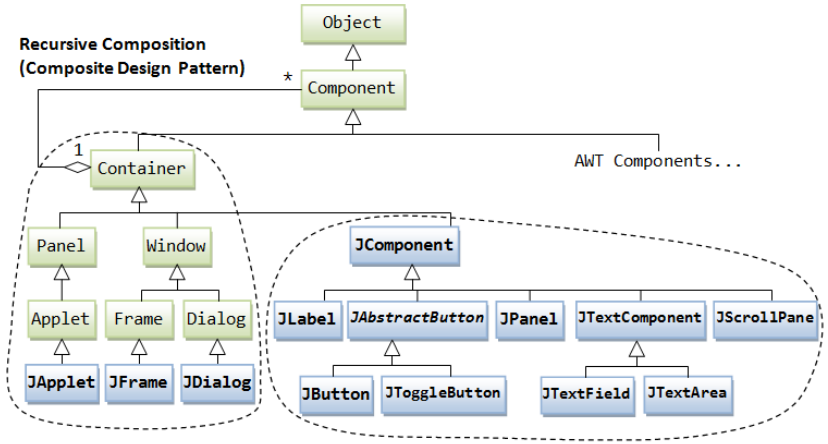
- Komponenty `Swingu` podporují připojitelný vzhled a chování. Můžete si vybrat mezi vzhledem a chováním Javy a vzhledem a chováním podkladového operačního systému (např. Windows, UNIX nebo macOS). Pokud je vybrána druhá možnost, tlačítko Swing běží na Windows, vypadá jako tlačítko Windows a chová se jako tlačítko Windows. Podobně tlačítko Swing běží na UNIXu, vypadá jako tlačítko UNIXu a chová se jako tlačítko UNIXu.
- a další...

Komponenty Swingu

V porovnání s třídami komponent `AWT` (v balíčku `java.awt`) začínají třídy komponent Swingu (v balíčku `javax.swing`) předponou `J`, např. `JButton`, `JTextField`, `JLabel`, `JPanel`, `JFrame` nebo `JApplet`.



Komponenty Swingu



Výše uvedený obrázek ukazuje hierarchii tříd grafického uživatelského rozhraní `Swing`. Podobně jako v `AWT` existují dvě skupiny tříd: *kontejnery* a *komponenty*. Kontejner se používá k uchovávání komponent. Kontejner může také obsahovat kontejnery, protože je (podtřídou) komponenty.

POZOR:

Nemíchejte těžké komponenty `AWT` a lehké komponenty `Swing` ve stejném programu, protože těžké komponenty budou vždy vykresleny přes lehké komponenty.

Kontejnery nejvyšší úrovně a sekundární kontejnery ve Swingu

Stejně jako aplikace v AWT vyžaduje i aplikace Swing kontejner nejvyšší úrovně. Ve Swingu existují tři kontejnery nejvyšší úrovně:

- JFrame: používá se pro hlavní okno aplikace (s ikonou, názvem, tlačítky pro minimalizaci/maximalizaci/zavírání, volitelným panelem nabídek a obsahovým panelem),
- JDialog: používá se pro sekundární vyskakovací okno (s názvem, tlačítkem pro zavření a obsahovým panelem).
- JApplet: používá se pro zobrazovací oblast appletu (obsahový panel) uvnitř okna prohlížeče.

Podobně jako v AWT existují sekundární kontejnery (například JPanel), které lze použít k seskupení a rozvržení relevantních komponent.

Obsah kontejneru nejvyšší úrovně ve Swingu

Na rozdíl od `AWT` se však komponenty `JComponents` nepřidávají přímo do kontejneru nejvyšší úrovně (např. `JFrame`, `JApplet`), protože se jedná o lehké komponenty. Komponenty `JComponents` musí být přidány do tzv. obsahu kontejneru nejvyšší úrovně. Obsah je ve skutečnosti objekt `java.awt.Container`, který lze použít k seskupení a uspořádání komponent.
Můžete:

- získat obsah pomocí `getContentPane()` z kontejneru nejvyšší úrovně a přidat do něj komponenty.
- nastavit obsah na `JPanel` (hlavní panel vytvořený ve vaší aplikaci, který obsahuje všechny komponenty grafického rozhraní) pomocí `setContentPane()` v `JFrame`.

Obsah kontejneru nejvyšší úrovně ve Swingu

```
1  public class SwingDemo extends JFrame {
2     // Constructor
3     public SwingDemo() {
4        // Get the content-pane of this JFrame, which is a java
             .awt.Container
5        // All operations, such as setLayout() and add()
             operate on the content-pane
6        Container cp = getContentPane();
7        cp.setLayout(new FlowLayout());
8        cp.add(new JLabel("Hello, world!"));
9        cp.add(new JButton("Button"));
10       ......
11    }
12    .......
13 }
```

Obsah kontejneru nejvyšší úrovně ve Swingu

```
1  public class SwingDemo extends JFrame {
2      // Constructor
3      public SwingDemo() {
4          // The "main" JPanel holds all the GUI components
5          JPanel mainPanel = new JPanel(new FlowLayout());
6          mainPanel.add(new JLabel("Hello, world!"));
7          mainPanel.add(new JButton("Button"));
8
9          // Set the content-pane of this JFrame to the main
               JPanel
10         setContentPane(mainPanel);
11         ......
12     }
13     .......
14 }
```

Zpracování událostí ve Swingu

Swing používá třídy pro zpracování událostí AWT (v balíčku `java.awt.event`). Swing zavádí několik nových tříd pro zpracování událostí (v balíčku `javax.swing.event`), ale ty se nepoužívají často.

Psaní Swing aplikací

Stručně řečeno, pro napsání Swing aplikace musíte:

1. Použití `Swing` komponent s prefixem `J` v balíčku `javax.swing`, např. `JFrame`, `JButton`, `JTextField`, `JLabel` atd.
2. Je potřeba kontejner nejvyšší úrovně (obvykle `JFrame`). `JComponents` by neměly být přidávány přímo do kontejneru nejvyšší úrovně. Měly by být přidány do obsahového panelu kontejneru nejvyšší úrovně. Odkaz na obsahový panel můžete načíst voláním metody `getContentPane()` z kontejneru nejvyšší úrovně.
3. Swing aplikace používají třídy pro zpracování událostí `AWT`, např. `ActionEvent`/`ActionListener`, `MouseEvent`/`MouseListener` atd.
4. Spusťte konstruktor ve vlákně `Event Dispatcher` (místo hlavního vlákna) pro bezpečnost vláken, jak je znázorněno v šabloně programu.

Proudy bajtů

Programy používají pro vstup a výstup 8bitových bajtů proudy bajtů. Všechny třídy související s proudy bajtů jsou odvozeny od tříd `InputStream` a `OutputStream`, např. `FileInputStream`, `FileOutputStream`.

Proudy bajtů

```
1  public class ByteStream {
2      public static void main(String[] args) throws IOException
           {
3          FileInputStream in = null;
4          FileOutputStream out = null;
5          try {
6              in = new FileInputStream("user.txt");
7              out = new FileOutputStream("user_output.txt");
8              int c;
9              while ((c = in.read()) != -1) {
10                 out.write(c);
11             }
12         } finally {
13             if (in != null) {
14                 in.close();
15             }
16             if (out != null) {
17                 out.close();
18             }
```

Znakové proudy

Platforma Java ukládá znakové hodnoty pomocí konvence Unicode. Pomocí znakových proudů se data z bajtového proudu překládají do lokální znakové sady.
Všechny třídy znakových proudů jsou odvozeny od tříd `Reader` a `Writer`.
Stejně jako u bajtových proudů existují třídy znakových proudů, které se specializují na I/O soubory. Jsou to `FileReader` a `FileWriter`.

Znakové proudy

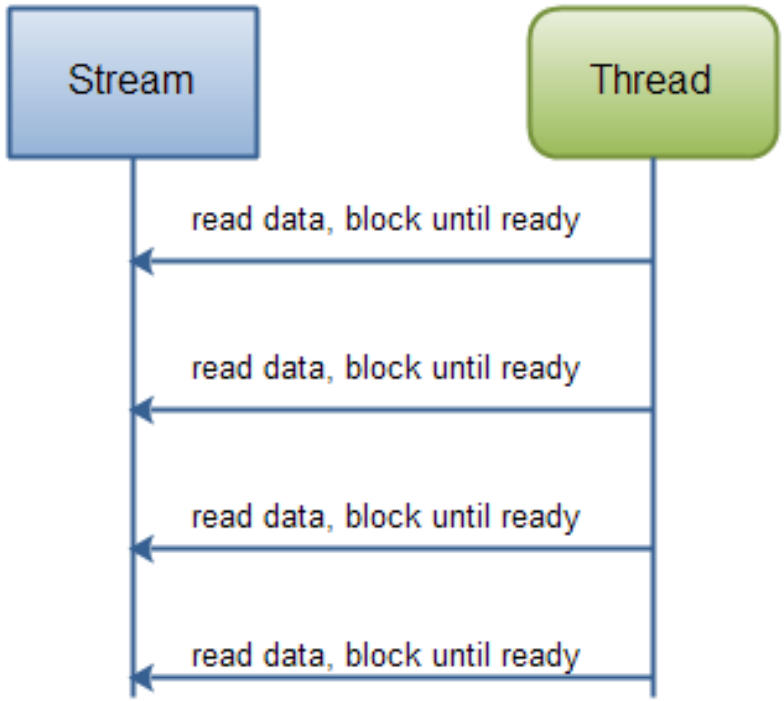
```
1  public class CharacterStream {
2      public static void main(String[] args) throws IOException
           {
3          FileReader in = null;
4          FileWriter out = null;
5          try {
6              in = new FileReader("user.txt");
7              out = new FileWriter("user_output.txt");
8              int c;
9              int nextChar;
10             while ((nextChar = in.read()) != -1) {
11                 out.append((char) nextChar);
12             }
13         } finally {
14             if (in != null) {
15                 in.close();
16             }
17             if (out != null) {
18                 out.close();
```

Ukládání dat do mezipaměti

Třídy založené na `InputStream` a `OutputStream` používají nebufferovaný vstup/výstup. To znamená, že jakýkoli požadavek na čtení nebo zápis je zpracováván přímo podkladovým operačním systémem. To může program výrazně snížit efektivitu, protože každý takový požadavek často umožňuje přístup k disku, síťovou aktivitu nebo jinou nákladnou operaci.
Aby se tento druh režie snížil, implementuje platforma Java bufferované I/O streamy. Bufferované vstupní streamy čtou data z oblasti paměti známé jako *buffer*.
Program může převést nebufferovaný stream na bufferovaný stream pomocí tříd, jako jsou: `BufferedReader`, `BufferedWriter`, které umožňují ukládání znakových streamů do mezipaměti. Třídy, jako jsou `BufferedInputStream` a `BufferedOutputStream`, mohou streamovat bajty.

Zpracování dat

V Java IO se data čtou bajt po bajtu pomocí `InputStream` nebo `Reader`. Podle diagramu program pokračuje pouze tehdy, když jsou k dispozici nová data ke čtení.



Java NIO — Buffery

Buffer je datový kontejner specifického primitivního typu. Buffer je lineární, konečná posloupnost prvků určitého primitivního typu. Kromě obsahu jsou základní vlastnosti bufferu:

- `capacity` – počet prvků, které obsahuje. Kapacita bufferu není nikdy záporná a nikdy se nemění.
- `limit` – je index prvního prvku, který by neměl být čten ani zapisován. Limit bufferu není nikdy záporný a nikdy není větší než jeho kapacita.
- `position` – je index další položky pro čtení nebo zápis. Pozice bufferu není nikdy záporná a nikdy nepřekročí svůj limit.

Java NIO — Buffery

Třída `Buffer` provádí následující operace:

- `clear()` – připraví buffer na novou posloupnost operací čtení kanálu nebo relativních operací vkládání.
- `flip()` – připraví buffer na novou posloupnost operací zápisu do kanálu nebo relativního načítání.
- `rewind()` – připraví buffer na opětovné čtení dat, která jsou v něm již obsažena.

Kanály

Kanál (z balíčku `java.nio.channels.Channel`) představuje otevřené připojení k entitě, jako je hardwarové zařízení, soubor, síťový socket nebo programová komponenta, která je schopna provádět jednu nebo více různých I/O operací, jako je čtení nebo zápis.
Kanál může být otevřený nebo uzavřený. Jakmile je kanál vytvořen, je otevřený a po uzavření zůstává uzavřený. Po uzavření kanálu jakýkoli pokus o provedení I/O operace na něm vyvolá výjimku `ClosedChannelException`.
V podstatě jsou kanály určeny k bezpečnému přístupu s více vlákny, jak je popsáno ve specifikacích rozhraní a tříd, které toto rozhraní rozšiřují a implementují.

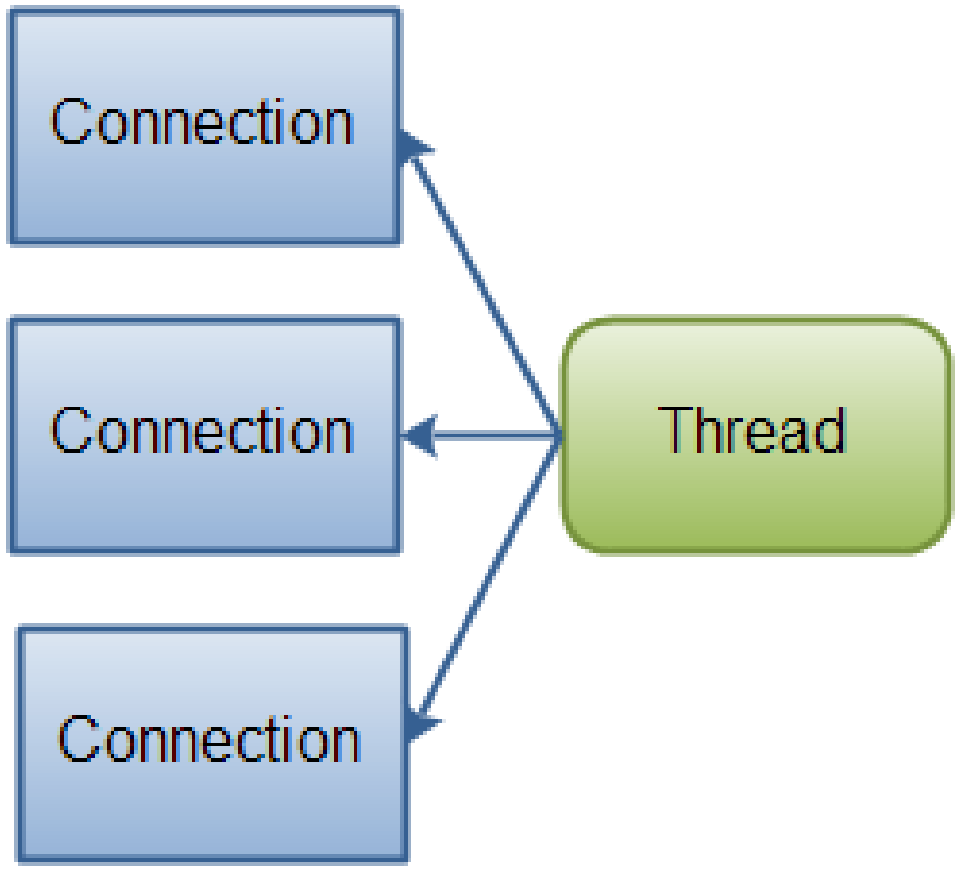
Kanály

Mezi nejčastěji používané implementace patří:

- `FileChannel` – umožňuje zapisovat a číst data ze souboru
- `DatagramChannel` – umožňuje ukládat a číst data prostřednictvím protokolu UDP
- `SocketChannel` – umožňuje zapisovat a číst data prostřednictvím protokolu TCP
- `ServerSocketChannel` – umožňuje naslouchat příchozím TCP připojením

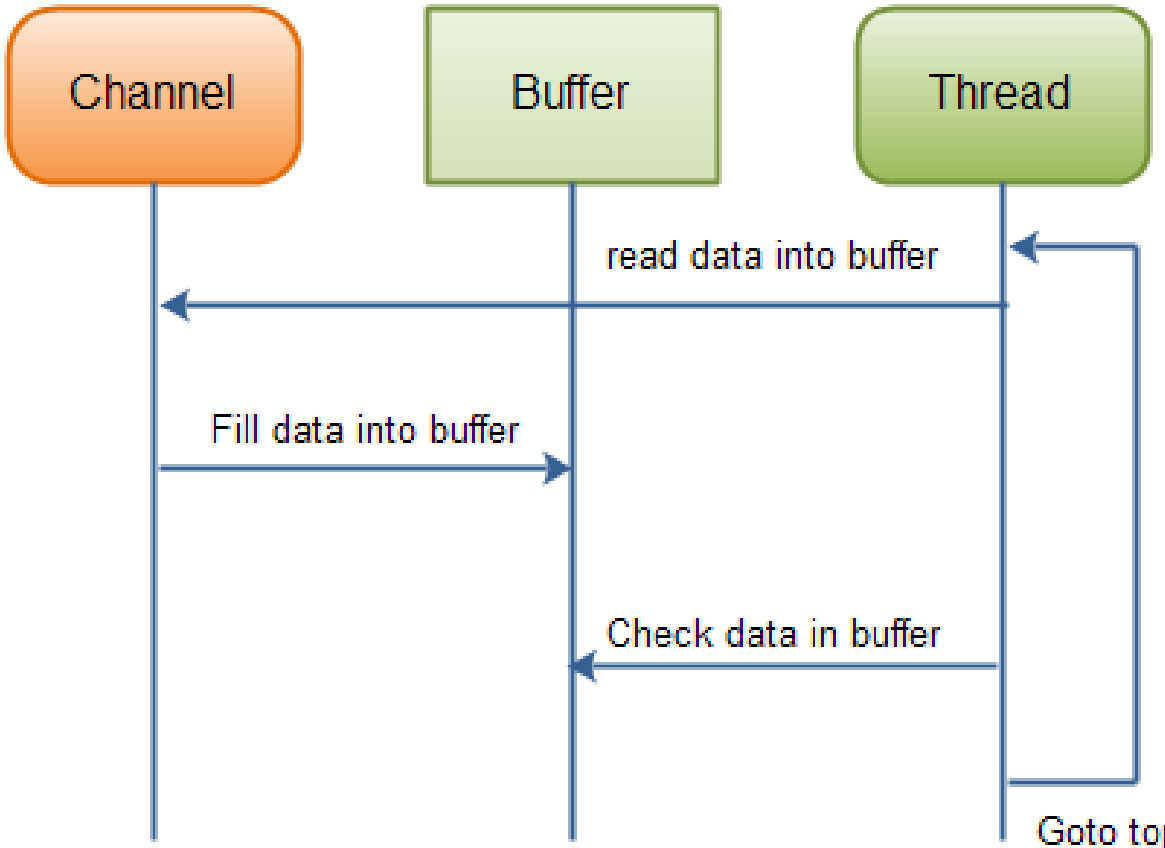
Zpracování dat

NIO umožňuje spravovat více kanálů (např. síťová připojení nebo soubory) pomocí určitého počtu vláken. Parsování dat však může být o něco složitější než čtení dat z blokovacího proudu.



Zpracování dat

Aby bylo možné postupně načítat data z vyrovnávací paměti, je nutné ji často dotazovat, aby se shromáždily informace o dostupnosti dat, což může vést ke snížení výkonu při čtení připojení.



Java NIO - operace se soubory

Java NIO zavedla mnoho zjednodušených mechanismů pro práci se soubory. Tyto mechanismy byly umístěny v balíčku `java.nio.file` a výchozím bodem pro jejich implementaci je třída `Files`.

Cesta (*Path*)

Rozhraní `Path` představuje cestu k souboru. Obsahuje metody, které lze použít mimo jiné k:

- získání informací o cestě
- přístupu k položkám cesty
- extrakci prvků cesty

```
1 Path p1 = Paths.get("/nio/example/path");
2 Path p2 = Paths.get(URI.create("file:///nio/example/FileTest.
      java"));
3 Path p3 = Path.of("/tmp", "dir_a", "dir_b");
```

Kontrola souborů nebo adresářů

Třída `Files` obsahuje mechanismy, které umožňují ověřování souborů, včetně:

- `Files.isExecutable(Path)` – kontrola, zda je soubor spustitelný
- `Files.isReadable(Path)` – kontrola, zda je soubor čitelný
- `Files.isWritable(Path)` – kontrola, zda je soubor zapisovatelný.

Mazání souborů

V rámci třídy `Files` je možné mazat soubory pomocí následujících metod:

- `Files.delete(Path)` – metoda smaže soubor, pokud existuje, nebo vyvolá výjimku
- `Files.deleteIfExists(Path)` – metoda smaže soubor, pokud existuje, jinak neprovede žádnou operaci

Kopírování souborů

Třída `Files` poskytuje mechanismus pro kopírování souborů nebo adresářů pomocí metody:

- `Files.copy(Path, Path, CopyOption ...)`

`CopyOption` může být jedna z následujících:

- `REPLACE_EXISTING` – vynutí kopírování souboru, i když soubor již v zadaném umístění existuje
- `COPY_ATTRIBUTES` – zahrnuje kopírování atributů souboru
- `NOFOLLOW_LINKS` – redukuje kopírování na symbolické odkazy

Přesouvání souborů

Metoda `Files.move(Path, Path, CopyOption...)` umožňuje přesouvat soubory, což bude mít za následek výjimku, dokud nebude nastaven příznak `REPLACE_EXISTING`.

Vytváření souborů

Metoda `Files.createFile(Path, FileAttribute <?>)` je zodpovědná za vytváření souborů s různými oprávněními. Pokud soubor, který chceme vytvořit, již existuje, bude vyvolána výjimka.

Čtení ze souboru

Čtení ze souboru se provádí pomocí:

- `Files.readAllBytes(Path)`
- `Files.readAllLines(Path, Charset)`

Zápis do souboru

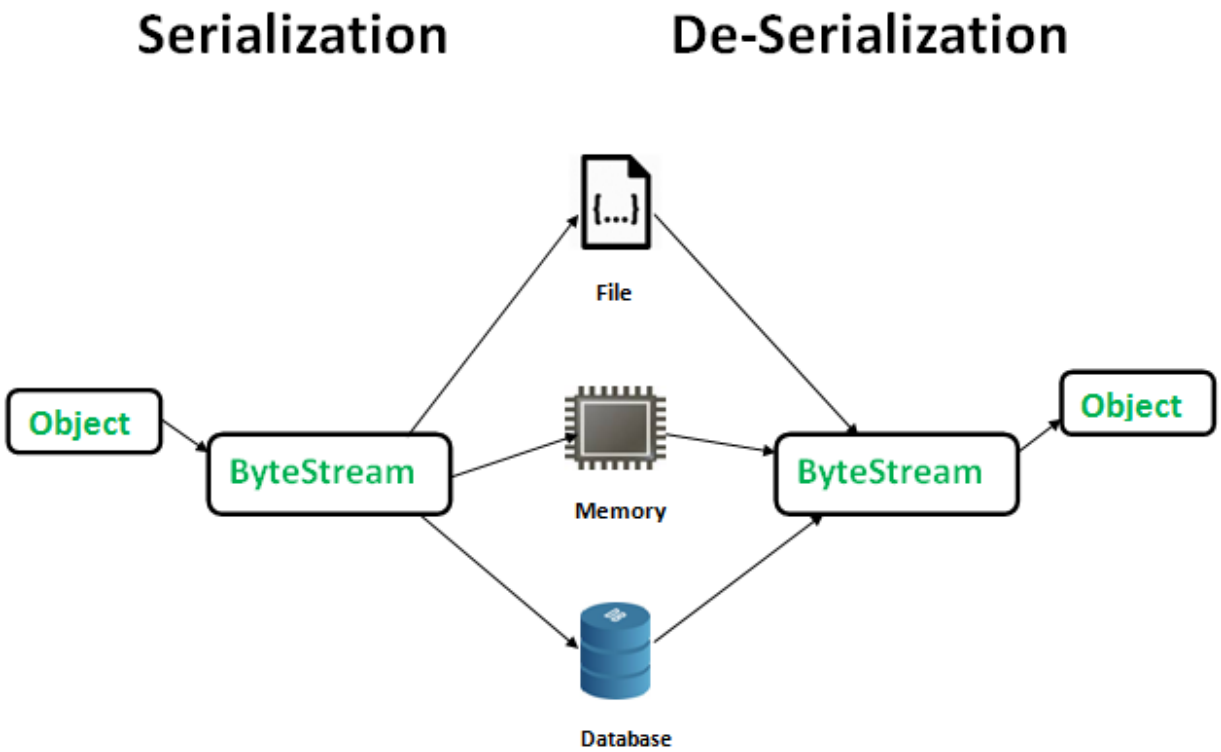
Do souboru je možné zapisovat pomocí následujících metod:

- `write(Path, byte[], OpenOption...)` – zápis bajtů
- `write(Path, Iterable <extends CharSequence>, Charset, OpenOption...)` – zápis více řádků, např. všech hodnot z `List<String>`

Při zadávání `OpenOption` používáme implementaci jako `StandardOpenOption`. Dostupné možnosti umožňují ovládat chování při ukládání, např. umožňují definovat, zda se před uložením dat do souboru nejprve smaže jeho obsah, nebo se doplní stávající.

Serializace a deserializace

Serializace je mechanismus, který převádí stav objektu do proudu bajtů. Deserializace je naopak inverzní proces, který umožňuje obnovit stav objektu v paměti z proudu bajtů.
Výsledný proud bajtů je nezávislý na platformě, takže objekt serializovaný na jedné platformě lze úspěšně deserializovat na jiné platformě.



Serializable

Rozhraní `Serializable` umožňuje serializaci a deserializaci objektů pomocí, mimo jiné, tříd `ObjectInputStream` a `ObjectOutputStream`, které obsahují API na vysoké úrovni pro serializaci a deserializaci objektů.

```
1  import java.io.Serializable;
2
3  public class Movie implements Serializable {
4      private int id;
5      private String title;
6      private String type;
7
8      public Movie(int id, String title, String type) {
9          this.id = id;
10         this.title = title;
11         this.type = type;
12     }
13
14     //getters and setters
15 }
```

Serializable

Bez implementace rozhraní `java.io.Serializable` nebudou výše uvedené třídy schopny správně serializovat a deserializovat objekt třídy `Movie`, což povede k výjimce `java.io.NotSerializableException`. Všimněte si také, že rozhraní `Serializable` nevyžaduje implementaci žádné metody.

Cvičení

1. Vytvořte řešení, které zobrazí všechny soubory/adresáře obsažené v zadaném adresáři.
2. Připravte řešení, které bude číst a zobrazovat soubor řádek po řádku.
3. Připravte řešení, které přidá řetězec do zadaného souboru.
4. Připravte řešení, které vrátí nejdelší slovo z poskytnutého souboru.
5. Vytvořte analyzátor CSV:

```
1 John,Smith,23
2 Sam,Johnson,40
3 Andrew,Manly,43
```

S výše uvedeným souborem by měl program vrátit tříprvkový seznam objektů typu `user` s poli: `name`, `surname`, `age`.

Cvičení

6. Vytvořte program, který bude na základě objektů třídy `Movie` poskytovat následující funkce:
   - přidávání objektů
   - vracení seznamů objektů

   Třída `Movie` by měla obsahovat pole: `title`, `genre`, `director`, `year_of_release`. Přidávání objektů by mělo odeslat jejich serializovanou formu do souboru. Zobrazení seznamu objektů by mělo umožnit deserializaci textového souboru pro převod jednotlivých řádků na objekty `Movie`.

Datum a čas

Jednou z nových funkcí zavedených v Javě 8 je nové API pro datum a čas, známé také jako JSR-310, které je snadno srozumitelné, logické a do značné míry podobné knihovně `Joda` (dostupné i před verzí Java 8). Hlavní třídy pro práci s datem/časem počínaje lokálním časem budou popsány níže.

Datum a čas

Podpora místního času (bez časového pásma)
Hlavní třídy podporující místní datum a čas (bez časových pásem) jsou:

- `java.time.LocalTime`
- `java.time.LocalDate`
- `java.time.LocalDateTime`
- `java.time.Instant`

LocalTime

Třída `LocalTime` reprezentuje čas bez vazby na konkrétní časové pásmo nebo dokonce datum. Jedná se o čas bez časového pásma v kalendářním systému ISO-8601. Má několik velmi užitečných metod:

- `now()` – statická metoda, která vrací aktuální čas. Výchozí formát je `HH:mm:ss.mmm` (hodiny:minuty:sekundy.milisekundy). Příklad:

```
1 LocalTime localTime = LocalTime.now();
2 System.out.println("Current time: " + localTime); // Current
    time: 22:34:27.106
```

LocalTime

- `withHour()`, `withMinute()`, `withSecond()`, `withNano()` – nastaví čas, minuty, sekundy a nanosekundy v aktuálním objektu `LocalTime`, např.:

```
1 LocalTime localTime = LocalTime.now()
2         .withSecond(0) // set the seconds to 0
3         .withNano(0); // set nanoseconds to 0
4 System.out.println("Current time: " + localTime); // Current
    time: 22:41
```

LocalTime

- `plusNanos(x)`, `plusSeconds(x)`, `plusMinutes(x)`, `plusHours(x)`, `minusNanos(x)`, `minusSeconds(x)`, `minusMinutes(x)`, `minusHours(x)` – přičítání (odečítání) nanosekund, sekund, minut, hodin k (od) nastaveného času, např.:

```
1 LocalTime now = LocalTime.now();
2 System.out.println("Current time: " + now); // Current time:
      22:49:01.241
3 now = now.plusMinutes(10).plusHours(1);
4 System.out.println("Current time after addition: " + now); //
      Current time after addition: 23:59:01.241
```

LocalTime

- `getHour()` – vrací hodinu
- `getMinute()` – vrací minutu
- `getSecond()` – vrací sekundu

Níže uvedený příklad ukazuje, jak libovolně formátovat hodiny, minuty a sekundy z aktuálního času:

```
1 LocalTime now = LocalTime.now();
2 String formattedTime = now.getHour() + ":" + now.getMinute()
    + ":" + now.getSecond();
3 System.out.println(formattedTime); // 22:55:26
```

LocalDate

Podstatou existence třídy `LocalDate` je reprezentace data (rok, měsíc, den). Tento objekt nebere v úvahu a neukládá čas (např. aktuální čas) ani časové pásmo. Datum je standardně uloženo ve formátu ISO-8601. Hlavní metody pracující s objekty lokálního data jsou uvedeny níže:

- `now()` – statická metoda, která vrací aktuální datum. Výchozí formát je `YYYY-mm-dd`, například:

```
1 LocalDate now = LocalDate.now();
2 System.out.println(now); // 2025-10-10
```

LocalDate

- `of(year, month, dayOfMonth)` – vytvoří objekt reprezentující datum (rok, měsíc, den). Měsíc může být reprezentován výčtovou hodnotou `java.time.Month` nebo indexem měsíce, např.:

```
1 LocalDate localDate = LocalDate.of(2025, Month.MARCH, 28);
2 System.out.println(localDate); // 2025-03-28
```

LocalDate

- `getYear()` – vrací celé číslo představující rok
- `getMonth()` – vrací měsíc pomocí objektu `java.time.Month`
- `getDayOfYear()` – vrací celé číslo informující o dni v roce
- `getDayOfMonth()` – vrací celé číslo představující den v měsíci
- `getDayOfWeek()` – vrací den v týdnu pomocí výčtu `java.time.DayOfWeek`

LocalDateTime

Podstatou existence třídy `LocalDateTime` je reprezentace data (rok, měsíc, den) a času (hodina, minuta, sekunda, milisekunda). Jedná se o formát bez časového pásma v kalendářním systému ISO-8601. Zde jsou některé z nejčastěji používaných metod z této třídy:

- `now()` – statická metoda, která vrací aktuální datum a čas. Výchozí formát je `YYYY-MM-ddThh:mm:ss.mmm`, např.

```
1 LocalDateTime localDateTime = LocalDateTime.now();
2 System.out.println(localDateTime); // 2025-10-10T14:25:16.124
```

LocalDateTime

- `of(year, month, dayOfMonth, hour, minutes, seconds, milliseconds)` – statická metoda, která vrací lokální datum a čas podle zadaných parametrů (rok, měsíc, den v měsíci, hodina, minuty, sekundy, milisekundy). Existují také přetížené ekvivalenty této metody s proměnným počtem parametrů. Příklad níže:

```
1 LocalDateTime localDateTime = LocalDateTime.of(2020, Month.
    MARCH, 28, 20, 0, 10, 0);
2 System.out.println(localDateTime); // 2020-10-10T15:00:10
```

V zobrazené hodnotě si všimněme, že znaménko `T` je konvenční oddělovač oddělující hodnotu data od času.

Instant

Tato třída se od ostatních odlišuje tím, že představuje specifický a jasně definovaný bod v čase (s přesností na jednu sekundu). Druhou důležitou vlastností je, že se nevztahuje na koncept dnů nebo let, ale pouze na univerzální čas, tzv. „čas v roce“. UTC. Stručně řečeno, interně ukládá počet sekund (s přesností na nanosekundu) od určitého pevného bodu v čase (1. ledna 1970 – tzv. epocha). Nejlépe se hodí k reprezentaci času způsobem, který bude zpracován systémem a nebude zobrazen koncovým uživatelům. Instanci třídy `Instant` lze použít k vytvoření instancí tříd, jako jsou `LocalTime`, `LocalDate` nebo `LocalDateTime`, např.:

Instant

```java
Instant instant = Instant.now();

LocalDateTime localDateTime = LocalDateTime.ofInstant(instant, ZoneId.systemDefault());
System.out.println(localDateTime); // 2025-10-10T18:33:29.116691800

LocalTime localTime = LocalTime.ofInstant(instant, ZoneId.of("CET"));
System.out.println(localTime); // 18:33:29.116691800

LocalDate localDate = LocalDate.ofInstant(instant, ZoneId.ofOffset("UTC", ZoneOffset.ofHours(2)));
System.out.println(localDate); // 2025-10-10
```

Třídy, které reprezentují intervaly

JSR-310 (JSR -Java Specification Request) zavádí koncept intervalu jako času, který uplynul mezi okamžiky A a B. Pro to existují dvě třídy:

- `java.time.Duration`
- `java.time.Period`

Liší se pouze v jednotkách (umožňují reprezentovat: časové jednotky (`Duration`) a např. měsíce nebo roky (`Period`)).

Třídy, které reprezentují intervaly

```
1 System.out.println(Duration.ofHours(10).toMinutes()); // 10
    hours expressed in minutes: 600
2
3 // In the example below, the time difference in minutes
    between the current time and the time 2 days later was
    calculated
4 System.out.println(Duration.between(LocalDateTime.now(),
    LocalDateTime.now().plusDays(2)).toMinutes()); // 2880
5
6 // The number of months between the two dates is calculated
    below.
7 System.out.println(Period.between(LocalDate.now(), LocalDate.
    now().plusDays(100)).getMonths()); // 3
```

Třídy, které reprezentují intervaly

Rozdíl mezi `Duration` a `Period` je založen na jednoduchém faktu – všechny délky vyjádřené pomocí `Duration` jsou reprezentovány v základních jednotkách času (tj. např. v sekundách), zatímco ty vyjádřené pomocí `Period` (měsíc, rok, století, tisíciletí) mohou mít různé reálné délky (různým počtem dnů v měsících, přestupných letech atd.).

Formát zobrazení data

Pro objekty typu formátování: Metoda `format(formatter)` používá `LocalDate`, `LocalTime` a `LocalDateeTime`. Příklad pro místní čas je uveden níže:

```
1 LocalTime localTime= LocalTime.now();
2 String formattedLocalTime = localTime.format(
      DateTimeFormatter.ISO_LOCAL_TIME);
3 System.out.println(formattedLocalTime); // 21:11:00.024
```

Formát zobrazení data

Formatter	Description	Example
ofLocalizedDate(dateStyle)	Formatter with date style from the locale	'2011-12-03'
ofLocalizedTime(timeStyle)	Formatter with time style from the locale	'10:15:30'
ofLocalizedDateTime(dateTimeStyle)	Formatter with a style for date and time from the locale	'3 Jun 2008 11:05:30'
ofLocalizedDateTime(dateStyle,timeStyle)	Formatter with date and time styles from the locale	'3 Jun 2008 11:05'
BASIC_ISO_DATE	Basic ISO date	'20111203'
ISO_LOCAL_DATE	ISO Local Date	'2011-12-03'
ISO_OFFSET_DATE	ISO Date with offset	'2011-12-03+01:00'
ISO_DATE	ISO Date with or without offset	'2011-12-03+01:00'; '2011-12-03'
ISO_LOCAL_TIME	Time without offset	'10:15:30'
ISO_OFFSET_TIME	Time with offset	'10:15:30+01:00'
ISO_TIME	Time with or without offset	'10:15:30+01:00'; '10:15:30'
ISO_LOCAL_DATE_TIME	ISO Local Date and Time	'2011-12-03T10:15:30'
ISO_OFFSET_DATE_TIME	Date Time with Offset	2011-12-03T10:15:30+01:00'
ISO_ZONED_DATE_TIME	Zoned Date Time	'2011-12-03T10:15:30+01:00[Europe/Paris]'
ISO_DATE_TIME	Date and time with ZoneId	'2011-12-03T10:15:30+01:00[Europe/Paris]'
ISO_ORDINAL_DATE	Year and day of year	'2012-337'
ISO_WEEK_DATE	Year and Week	2012-W48-6'
ISO_INSTANT	Date and Time of an Instant	'2011-12-03T10:15:30Z'
RFC_1123_DATE_TIME	RFC 1123 / RFC 822	'Tue, 3 Jun 2008 11:05:30 GMT'

Formát zobrazení data

Kromě hotových formátů si můžeme připravit i vlastní implementace. Pro tento účel musíme použít speciální symboly, které mají specifický význam a reprezentaci, jako například:

```
1 String date = LocalDate.now().format(DateTimeFormatter.
    ofPattern("MM:yyyy:dd"));
2 System.out.println(date); // 04:2020:19
```

Seznam a popis všech dostupných symbolů pro formátování objektů `LocalDate`, `LocalTime` a `LocalDateTime` naleznete zde.

Cvičení

Implementujte metodu, která přijímá jeden parametr typu `LocalDate` a vrací informaci, pokud je toto datum starší než aktuální. Implementujte prosím také testovací metodu `main()`.