1. Domácí úloha 11

Základní informace:

- **Účel:** Lambda výrazy a funkční rozhraní
- **Kostra:** `11_LambdaVyrazy.zip`
- **Odevzdávaný soubor/JAR:** `11_Lambda.jar`

Warning

Toto je velmi těžká úloha, nikoliv však objemem požadovaného kódu, ale nutností pochopení všech souvislostí. Ty lze dohledat v přednáškách. V tomto zadání nehledejte podrobný popis kroků řešení.

Záměrně zde nejsou Javadoc komentáře. Je to proto, abyste si uvědomili jejich důležitost a to i v případě, že v kódu je snaha o maximální srozumitelnost identifikátorů. Takže z pohledu autora je "všechno na první pohled jasné". Sami se přesvědčíte, že tomu tak zřejmě nebude.

Zadání:

- prostudujte připravený výčtový typ `Fakulty` a třídu `VysledekZkousky` - jejich dokonalé pochopení je klíčové pro řešení této DÚ
  - zde nebudete nic měnit
- třída `SeznamVysledku` spuštěná ve třídě `Hlavni` pouze připraví seznam instancí třídy `VysledekZkousky` - jejich detailní pochopení není nezbytné
  - ve třídě `SeznamVysledku` se používá formální parametr `start` jako počáteční hodnota generátoru náhodných čísel
    - její změnou lze dostat zcela jiné výsledky
  - zde nebudete nic měnit
- třídy `AgregovaneOperaceTest` a `FunkcniRozhraniTest` jsou kompletní testy
  - testy využívají možnosti parametrizovaných testů, které není nezbytné chápat do detailů (význam detailů viz na konci tohoto zadání)
  - testy se spouštějí standardním způsobem
    - v Eclipse - Run / Run As / JUnit Test
    - v IntelliJ IDEA - přes zelenou šipku vlevo od názvu třídy nebo od názvu testu
  - zde nebudete nic měnit
- třídu `AgregovaneOperace` budete doplňovat - počítá různé statistiky nad seznamem výsledků zkoušek
  - je třeba, abyste využili agregovaných operací nad `stream()` a využívali lambda výrazy
  - první metoda (`getPocetVsech()`) je pro ukázku a je kompletní - její testy projdou

- další metody budete postupně doplňovat
  - ♦ co mají vracet, je zřejmé z jejich názvu
  - ♦ jaký má být konkrétní výsledek, lze vidět z parametrů testu, případně z jeho chybového výpisu
- třída má pro jednoduchost všechny metody statické

■ třídu `FunkcniRozhrani` budete doplňovat - zpracovává jednotlivé části osobního čísla

- metoda `castOsobnihoCisla()` využívá funkčního rozhraní `Function`, které je jejím parametrem
  - ♦ to při používání umožní snadnou změnu funkčnosti celé metody
  - ♦ v této metodě nebudete nic měnit
- je třeba, abyste dopsali kód následujících funkčních rozhraní
  - ♦ je potřeba využívat lambda výrazů
- první konstanta funkčního rozhraní (`PORADI`) je pro ukázku a je kompletní - její testy projdou
- další funkční rozhraní budete postupně doplňovat
  - ♦ co mají vracet, je zřejmé z jejich názvu
  - ♦ jaký má být konkrétní výsledek, lze vidět z parametrů testu, případně z jeho chybového výpisu
- třída má pro jednoduchost všechny části statické
  - ♦ to mimo jiné například umožní statický import ve `FunkcniRozhraniTest`

■ do Portálu odevzdáte JAR soubor celého projektu

Postup řešení:

■ stáhněte si soubor `11_LambdaVyrazy.zip`

■ rozbalte tento soubor, založte v Eclipse / IntelliJ nový projekt a přetáhněte Java soubory do tohoto projektu

- měly by být v balíku `lambda`
- jsou v kódování UTF-8, což v Eclipse nastavíte v *Properties* projektu

■ připojte do projektu knihovnu JUnit 4.12 (nikoliv Junit 5)

- Eclipse - Project / Properties / Java Build Path / Add Library / JUnit
- IntelliJ - v importech `AgregovaneOperaceTest` klikněte na červeně zbarvený `junit` a stiskněte Alt + Enter
  - ♦ následně vyberte JUnit 4.12

■ podle instrukcí výše doplňte dodanou třídu `AgregovaneOperace`

- správnost doplnění vždy zkontrolujte pomocí `AgregovaneOperaceTest`

- pokračujte tak dlouho, dokud neprojdou všechny testy

```
package lambda;

import java.util.List;
import java.util.stream.Collectors;

public class AgregovaneOperace {
  public static int getPocetVsech(List<VysledekZkousky> seznam) {
    return (int) seznam
        .stream()
        .count();
  }

  public static int getPocetNaFakulte(List<VysledekZkousky> seznam, Fakulty ▶
fakulta) {
    // TODO
    return -1;
  }

  public static int getPocetZnamekNaFakulte(List<VysledekZkousky> seznam, ▶
Fakulty fakulta,
                                            int konkretniZnamka) {
    // TODO
    return -1;
  }

  public static double getPrumernaZnamka(List<VysledekZkousky> seznam) {
    // TODO
    return -1.0;
  }

  public static String getSerazenaOsobniCislaNaFakulte(List<VysledekZkousky> ▶
seznam,
                                                       Fakulty fakulta) {
    // TODO
    return null;
  }
}
```

- podle instrukcí výše doplňte dodanou třídu `FunkcniRozhrani`
  - správnost doplnění vždy zkontrolujte pomocí `FunkcniRozhraniTest`
    - pokračujte tak dlouho, dokud neprojdou všechny testy

```
package lambda;

import java.util.function.Function;

public class FunkcniRozhrani {

  public static String castOsobnihoCisla(String osobniCislo, Function<String, ▶
```

```
String> f) {
    return f.apply(osobniCislo);
  }

  public static final Function<String, String> PORADI = (oc) -> {return ▶
oc.substring(4, 8);};

  public static final Function<String, String> PISMENO_FAKULTY = null; // TODO

  public static final Function<String, String> ROK_NASTUPU = null; // TODO

  public static final Function<String, String> TYP_STUDIA = null; // TODO

}
```

- Java soubory ze `src` adresáře zabalte do JAR souboru `11_Lambda.jar`, který budete odevzdávat
  - zkopírujte z adresáře Eclipse / IntelliJ podadresář `src` do nějakého pomocného adresáře, např.: `d:\zzz`
  - pak v adresáři `d:\zzz` použijte příkaz:

    ```
    jar cMf 11_Lambda.jar src
    ```

  - odevzdejte na Portál soubor `11_Lambda.jar`

    Note

    Pokud budete zkoušet připravovat JAR soubor jakýmkoliv jiným způsobem (generováním z Eclipse, pomocí Ant apod.), zkontrolujte, zda má výsledný JAR stejnou adresářovou strukturu, jako když použijete příkazy z příkazové řádky. Pokud bude struktura jiná nastanou problémy s validací.

1.1. Stručné vysvětlení konstrukce parametrizovaného JUnit testu

- je možné připravit sadu parametrů, nad kterými bude opakovaně spouštěn jeden (nebo více) testovací případ
- využívá se anotace `@RunWith(Parameterized.class)`, která je doplněna anotací `@Parameters`
- příklad bude ukazován na triviální testované třídě

```
public class NasobilkaPeti {
  public static int vypocti(int cislo) {
    return cislo * 5;
  }
}
```

- předání parametrů testu je provedeno injekcí přes atributy

- atributy `vstup` a `ocekavano` jsou anotovány pomocí `@Parameter(value = 0)`, kde `value` znamená pořadí ve dvojici dat (první hodnota je vždy `0`)
- každá instance `NasobilkaPetiInjekceTest` je vytvořena pomocí defaultního bezparametrického konstruktoru, který nastaví atributy anotované pomocí `@Parameter`
  - Pozor: atributy musí být `public`

```
@RunWith(Parameterized.class)
public class NasobilkaPetiInjekceTest {
    @Parameters
    public static Collection<Object[]> dvojiceDat() {
        return Arrays.asList(new Object[][] {
          { 0, 0 }, { 2, 10 }, { 4, 21 }, { 6, 30 }, { 8, 40 }, { 10, 50 },
        });
    }

    @Parameter(value = 0)
    public int vstup;
    @Parameter(value = 1)
    public int ocekavano;

    @Test
    public void parametrizovanyTest() {
        assertEquals(ocekavano, NasobilkaPeti.vypocti(vstup));
    }
}
```

- po spuštění vypíše:

```
getTestHeader: parametrizovanyTest[2](parameterized.NasobilkaPetiInjekceTest)
getMessage: expected:<21> but was:<20>
```

- pro zajištění přehlednějšího chybového výpisu je možné jméno testu libovolně rozšířit a zahrnout do něj i hodnoty testu
  - používá se parametr anotace `@Parameters(name = "{index}: 5 * {0} = {1}")`
    - kde se v `{}` zadají index `{index}` a pak první parametr `{0}`, druhý parametr `{1}` atd. to vše doplněné libovolným vhodným textem – zde operátor `*` a rovnítko `=`
  - vše ostatní zůstává zcela stejné

```
public class NasobilkaPetiInjekceJmenoTest {
    @Parameters(name = "{index}: 5 * {0} = {1}")
    public static Collection<Object[]> dvojiceDat() {
        return Arrays.asList(new Object[][] {
          { 0, 0 }, { 2, 10 }, { 4, 21 }, { 6, 30 }, { 8, 40 }, { 10, 50 },
        });
    }
…
```

- po spuštění vypíše:

```
getTestHeader: parametrizovanyTest[2: 5 * 4 = 21]
          (parameterized.NasobilkaPetiInjekceJmenoTest)
getMessage: expected:<21> but was:<20>
```

  - kde výpis:

    ```
    parametrizovanyTest[2: 5 * 4 = 21]
    ```

    je mnohem přehlednější a víc vypovídající o charakteru chyby