Laboratorium 2.

1. **Przygotowanie klasy referencyjnej w Javie**

    1. Wpisujemy poniższy kod w pliku `Person.java`

```
public class Person {
  private final String id; // np. PESEL
  private final String givenName;
  private String surname;

  public String getGivenName() { return givenName; }
  public String getSurname() { return surname; }
  public void setSurname(String surname) { this.surname = surnam
e; }
  public String getName() { return givenName + " " + surname; }
  protected String getId() { return id; }

  public Person(String givenName, String surname, String id) {
    this.id = id;
    this.givenName = givenName;
    this.surname = surname;
  }
}
```

    2. W pliku `JavaAppl.java` wpisujemy poniższy kod:

```
public class JavaAppl {
  public static void main(String[] args) {
    Person janek = new Person("Jan", "Kowalski", "1234567890");
    System.out.println(janek.getName());
  }
}
```

    3. Kompilujemy oba pliki (`javac`), potem uruchamiamy JavaAppl (`java JavaAppl`)

    4. *(Zadanie opcjonalne)* Analizujemy (`javap`) kod bajtowy klasy `Person` .

2. **Definicja klasy w Scali**

    Zadanie: Przepisać definicję klasy `Person` z Javy do Scali (do pliku `lab2.scala`)

    1. Kompilujemy (`scalac`) plik `lab2.scala`
    2. (*Zadanie opcjonalne*) Analizujemy kod bajtowy utworzonej klasy. Porównujemy go z wynikiem `javac Person.java`

3. **Definicja (obiektu) aplikacji w Scali**

    1. W pliku `lab2.scala` wpisujemy poniższy kod:

```
object Appl {
  def main(agrs: Array[String]) {
    val p = new Person("Jan", "Kowalski", "1234567890")
    println(p.name)
  }
}
```

    2. Kompilujemy plik lab2.scala (`scalac lab2.scala`) i uruchamiamy kod (`scala Appl`)

    3. W metodzie `main` dodajemy (na końcu) linię

```
println(p.id)
```

    4. Kompilujemy kod (`scalac lab2.scala`) i analizujemy komunikaty kompilatora

    5. Usuwamy linię `println(p.id)` , potem modyfikujemy obiekt `Appl` :

```
object Appl extends App {
  val p = new Person("Jan", "Kowalski", "1234567890")
  println(p.name)
}
```

    6. Kompilujemy plik `lab2.scala` i uruchamiamy kod (`scala Appl`)

    Zadanie (opcjonalne):

    - zmodyfikować plik `JavaAppl.java`

```
public class JavaAppl {
  public static void main(String[] args) {
    Person janek = new Person("Jan", "Kowalski", "123456789
0");
    // dostęp do id() jest chroniony (protected)!
    System.out.println(janek.id());
  }
}
```

    - skompilować kod (`javac`), potem uruchomić `JavaAppl`

4. `@BeanProperty`

    Zadanie:

- zmodyfikować definicję klasy `Person` (wersję napisaną w Scali) tak, aby składowa `surname` stała się "właściwością" (get+set)
- sprawdzić (modyfikując odpowiednio metodę `main`) możliwość wywołania `getSurname`, `setSurname`
- przeanalizować kod bajtowy klasy `Person` (ile jest w sumie metod dostępowych?)

5. **Definicja "operatorów" w Scali**

   1. Wpisujemy poniższy kod (np. w pliku lab2.scala):

```
class Int2DVec(val x: Int, val y: Int) {
  def +(other: Int2DVec) = new Int2DVec(x + other.x, y + other.y
)
  def unary_- = new Int2DVec(-x, -y)
}

object Appl {
  def main(agrs: Array[String]) {
    val v1 = new Int2DVec(1, 2)
    val v2 = new Int2DVec(10, 20)
    val v3 = v1 + v2
    val v4 = -v2
    println(v3)
    println(v4)
  }
}
```

   2. Kompilujemy kod (`scalac`) i uruchamiamy Appl (`scala Appl`)

   3. W klasie `Int2DVec` dodajemy metodę `toString`

```
override def toString() = "(" + x.toString + "," + y.toString +
")"
```

   4. Ponownie kompilujemy kod i uruchamiamy Appl

      Zadanie: dodać do klasy `Int2DVec` metody pozwalające odejmować i mnożyć (skalarnie) dwa wektory

6. **Konstruktor pomocniczy**

   1. W definicji klasy `Int2DVec` dodajemy dwa konstruktory pomocnicze:

```
class Int2DVec(val x: Int, val y: Int) {
  //...
  //Auxiliary constructors
  def this() = {
    this(0, 0)
    println("Creating Int2DVec(0,0)")
  }

  def this(prototype: Int2DVec) = {
    this(prototype.x, prototype.y)
    println("Copy-constructor working")
  }
}
```

   2. Modyfikujemy kod obiektu `Appl` (dodajemy wywołania obu konstruktorów pomocniczych):

```
object Appl {
  def main(agrs: Array[String]) {
    //...
    println(v4)
    println(new Int2DVec())
    println(new Int2DVec(v1 + v1 + v2))
  }
}
```

      Zadanie: zamienić kolejność linii (`this(...)` <-> `println(...)`) w obu konstruktorach pomocniczych; następnie skompilować kod i przeanalizować komunikaty kompilatora.

7. **Konstruktor prywatny**

   1. Modyfikujemy klasę Int2DVec w następujący sposób:

```
class Int2DVec private (val x: Int, val y: Int) {
  //...
}
```

   2. Kompilujemy kod i analizujemy komunikaty kompilatora

8. **Obiekt towarzyszący i metoda `apply()`**

   1. Dodajemy definicję obiektu `Int2DVec` ("towarzyszącego"):

```
object Int2DVec {
  def apply(x: Int, y: Int) = new Int2DVec(x, y)
  def apply() = new Int2DVec(0, 0)
  def apply(prototype: Int2DVec) = new Int2DVec(prototype.x, pro
totype.y)
```

```
}
```

2. Usuwamy konstruktory pomocnicze z definicji klasy `Int2DVec`
3. Modyfikujemy kod metody `main`:

```
object Appl {
  def main(agrs: Array[String]) {
    val v1 = Int2DVec(1, 2)
    val v2 = Int2DVec(10, 20)
    val v3 = v1 + v2
    val v4 = -v2
    println(v3)
    println(v4)
    println(Int2DVec())
    println(Int2DVec(v1 + v1 + v2))
  }
}
```

4. Kompilujemy kod i uruchamiamy `Appl`. Dlaczego metoda `apply` może sięgać do prywatnego konstruktora klasy `Int2DVec`?
5. Sprawdzamy, ile i jakie pliki `.class` zostały utworzone w procesie kompilacji

Zadanie: dodać w obiekcie towarzyszącym (`Int2DVec`) metodę `unitVectorOf`

Zadanie (opcjonalne): analizujemy kod bajtowy plików wynikowych powyższej kompilacji

Zadanie (opcjonalne): rozszerzyć klasę Int2DVec w taki sposób, aby można było sprawdzać prostopadłość wektorów w następujący sposób

```
if (v1 |-? v2) { /* "orthogonal case block" */ }
else { /*...*/ }
```

uwaga: `|-?` znaczy “czy ortogonalne/prostopadłe?” (np. `v1 |-? v2` to “czy `v2` jest prostopadłe do `v1`?)”

9. **Cechy (traits), domieszki (mix-ins)**

Wpisujemy, kompilujemy poniższy kod:

```
trait Fraction {
  val num: Int
  val denom: Int
  def *(other: Fraction): Fraction
}
```

```
trait Loggable {
  def log(timeStamp: Long, msg: String) =
    println("[" + timeStamp.toString + "]: " + msg)
}

object Fraction {
  // one of the "creational patterns/idioms"
  def apply(num: Int, denom: Int, loggable: Boolean = false): Fractio
n =
    if (loggable) new LoggableImpl(num, denom)
    else new DefaultImpl(num, denom)

  private class DefaultImpl(val num: Int, val denom: Int) extends Fra
ction {
    override def *(other: Fraction): Fraction =
      Fraction(this.num * other.num, this.denom * other.denom)
    override def toString() = num.toString + "/" + denom.toString
  }

  private class LoggableImpl(num: Int, denom: Int)
  extends DefaultImpl(num, denom) with Loggable {
    def timeStamp = System.nanoTime // to keep the example short...
    override def *(other: Fraction): Fraction = {
      log(timeStamp, "multiplying " + this.toString + " by " + other.
toString)
      // super.*(other) is not loggable
      Fraction(this.num * other.num, this.denom * other.denom, true)
    }
  }
}

object Appl {
  def main(agrs: Array[String]) {
    val f1 = Fraction(3, 7)
    val f2 = Fraction(4, 9)
    val f3 = Fraction(1, 19, true)
    val f1f2 = f1 * f2
    println(f1.toString + " * " + f2.toString + " = " + f1f2)
    println(f3.toString + " * " + f2.toString + " * " +
            f1.toString + " = " + f3 * f2 * f1)
  }
}
```

Zadanie: zaimplementować metody “+”, “-“ i “/“

Zadanie (opcjonalne): dodać trait Simplifiable (por. Loggable) pozwalający na

sprowadzanie ułamków do najprostszej postaci (np. 1/6 + 1/3 = 1/2)

Zadanie (opcjonalne): wykorzystując trait Simplifiable, dodać (w obiekcie towarzyszącym) trzecią implementację “cechy” Fraction

Zadanie (opcjonalne): napisać testy sprawdzające poprawność przygotowanej implementacji (arytmetyki liczb wymiernych)