Lista 10

1. a)

```
class UzycieWyjatkow {
    def main(args: Array[String]): Unit =
    {
      try {
      metoda1()
      }
      catch {
        case e:Exception => {
         println(e.getMessage())
         e.printStackTrace()
        }
      }
    }

    def metoda1() = metoda2()
    def metoda2() = metoda3()
    def metoda3() = throw new Exception("Wyjatek zgloszony w metoda3")
}
```

b)

Bez adnotacji:

```
public class UzycieWyjatkow implements scala.ScalaObject {
  public void main(java.lang.String[]);
  public scala.runtime.Nothing$ metoda1();
  public scala.runtime.Nothing$ metoda2();
  public scala.runtime.Nothing$ metoda3();
  public UzycieWyjatkow();
}
```

Z adnotacją:

```
public class UzycieWyjatkow implements scala.ScalaObject {
  public void main(java.lang.String[]);
  public scala.runtime.Nothing$ metoda1() throws java.lang.Exception;
  public scala.runtime.Nothing$ metoda2() throws java.lang.Exception;
  public scala.runtime.Nothing$ metoda3() throws java.lang.Exception;
  public UzycieWyjatkow();
}
```

2. a) Jako klasa inwariantna jest to oczywiste – spodziewamy się typu T i w argumencie dostajemy typ T, więc wszystko bez problemu zadziała. Jako klasa kowariantna – argument jest typu T, który jest podtypem typu klasy, np. `new GenericCellImm[Number](5.0)` lub `new GenericCellImm[Number](5)`, gdzie klasa jest typu **Number**, a argument typu **Int** lub **Double** (widać, że argument jest podtypem typu klasy). Dla **val** czyli wartości stałej jako, że argument nie zmieni na pewno swojej zawartości wszystko jest bezpieczne – wiadomo, że typ argumentu nie zmieni się nagle, np. z **Int** na **Double**. W przypadku gdy argument jest typu **var** czyli jest wartością zmienną w przypadku klasy kowariantnej w podanym przykładzie jest możliwość zmiany argumentu, np. z **Int** na **Double**, co spowoduje problem z typami. Podobnie jeśli klasa byłaby, np. generyczną tablicą, a jak wiadomo tablice są homogeniczne, więc ustawienie typu tablicy na **Any** umożliwiłoby

wrzucenie tam razem, np. **Intów** i **Stringów**, które są różnymi typami. W podobnych przypadkach Java wyrzuca wyjątek **ArrayStoreException**, Scala nie dopuszcza do takich sytuacji dając błąd kompilacji w przypadku próby skompilowania kowariantnej klasy, której argumentem konstruktora głównego jest wartość zmienna.

b) Tego błędu się można pozbyć tylko poprzez zmianę **var** na **val** lub poprzez zmianę klasy kowariantnej na inwariantną czyli stosując jeden z wcześniej podanych wariantów.

3.
```
abstract class Sequence[+A] {
  def append[B >: A](x: Sequence[B]): Sequence[A]
}
```

Teraz zachodzi kowariantność, ponieważ argument jest nadtypem A czyli operator przestrzeni funkcyjnej jest kontrawariantny względem argumentu metody oraz kowariantny względem wyniku metody.

4.
```
import scala.collection.mutable.Seq
def copy[T](dest: Seq[T], src: Seq[T]): Unit = {
    require(src.length>dest.length)
    var i = 0
    src.foreach(a => {
      dest.update(i, a)
      i += 1
    })
  }
```
5.
```
class UnderflowException(msg:String) extends Exception(msg)
class Queue[T] private(private val queue:(List[T],List[T])) {

  def normalize(xs1:List[T], xs2:List[T]) =
    (xs1,xs2) match {
    case (Nil, xs2) => new Queue((xs2.reverse, Nil))
    case queue => new Queue(queue)
  }

  def enqueue(v:T) = {
    val (xs1,xs2) = queue
    normalize(xs1,v::xs2)
  }

  def dequeue() =
    queue match {
    case (_::xs1tail,xs2) => normalize(xs1tail,xs2)
    case queue => new Queue(queue)
  }

  def first() =
    queue match {
    case (x::_, _) => x
    case _ => throw new UnderflowException("first")
  }

  def isEmpty() = queue==(Nil,Nil)
}

object Queue {
  def apply[T](xs:T*) = new Queue[T](xs.toList,Nil)
  def empty[T] = new Queue[T](Nil,Nil)
}
```