Laboratorium 3.

1. **Wyjątki** (`try/catch` , `try/finally` i `try/catch/finally`)

    1. Tworzymy plik `main.scala` i wpisujemy w nim:

```
object Appl {
  def main(args: Array[String]) = {
    println("Opening the file...")
    val inFile = scala.io.Source.fromFile("logins.txt")
    for (line <- inFile.getLines) println(line)
    println("Closing the file...")
    inFile.close
  }
}
```

    2. Kompilujemy plik (`scalac main.scala`)

    3. Uruchamiamy Appl (`scala Appl`) i analizujemy wynik

    4. Modyfikujemy zawartość `main.scala` :

```
object Appl {
  def main(args: Array[String]) = {
    try {
      println("Opening the file...")
      val inFile = scala.io.Source.fromFile("logins.txt")
      for (line <- inFile.getLines) println(line)
      println("Closing the file...")
      inFile.close
    } catch {
      case ex: java.io.FileNotFoundException =>
        println(ex.getMessage)
    }
  }
}
```

    5. Kompilujemy plik (`scalac main.scala`)

    6. Uruchamiamy Appl (`scala Appl`) i analizujemy wynik

    7. W katalogu roboczym (w którym znajduje się plik `main.scala`), dodajemy pusty plik `logins.txt` (np. komendą `touch logins.txt`)

    8. Modyfikujemy zawartość main.scala:

```
object Appl {
  def main(args: Array[String]) = {
    try {
      println("Opening the file...")
      val inFile = scala.io.Source.fromFile("logins.txt")
      for (line <- inFile.getLines) println(line)
      val x = 100 / inFile.getLines.length
      println("Closing the file...")
      inFile.close
    } catch {
      case ex: java.io.FileNotFoundException =>
        println(ex.getMessage)
    }
  }
}
```

    9. Kompilujemy plik (scalac main.scala)

    10. Uruchamiamy Appl (`scala Appl`) i analizujemy wynik. Czy plik `logins.txt` został zamknięty?

    11. Modyfikujemy zawartość pliku `main.scala` :

```
object Appl {
  def readFile(fileName: String) = try {
    println("Opening the file...")
    val inFile = scala.io.Source.fromFile(fileName)
    try {
      for (line <- inFile.getLines) println(line)
      val x = 100 / inFile.getLines.length
    } finally {
      println("Closing the file...")
      inFile.close
    }
  } catch {
    case ex: java.io.FileNotFoundException =>
      println(ex.getMessage)
    case ex: Throwable =>
      println("Default exception handler: "+ ex.getMessage)
  }

  def main(args: Array[String]) {
    readFile("logins.txt")
  }
}
```

Uwaga: plik `logins.txt` musi być pusty

12. Kompilujemy plik (`scalac main.scala`)
13. Uruchamiamy Appl (`scala Appl`) i analizujemy wynik. Czy plik `logins.txt` został zamknięty?

2. **Pakiety**

    1. W pliku `main.scala` wpisujemy kolejno następujące definicje, kompilujemy kod, analizujemy wyniki kompilacji (w tym kod bajtowy) i na koniec (po każdej analizie) usuwamy katalog `p1` (np. `rm -r p1`)

```
package p1.p2.p3
class A3
```

```
package p1
package p2
package p3
class A3
```

```
package p1.p2.p3 {
  class A3
}
```

```
package p1 {
  package p2 {
    package p3 {
      class A3
    }
  }
}
```

    2. W pliku `main.scala` wpisujemy następujące definicje, kompilujemy kod, analizujemy wyniki kompilacji (w tym kod bajtowy), uruchamiamy Appl i na koniec usuwamy katalog `p1` (np. `rm -r p1`)

```
package p1 {
  class A1T { new A1B }
  package p2 {
    class A2T { new A2B; new A1T; new A1B }
    package p3 {
      class A3 { new A2T; /* new A2B; */ new A1T; /* new A1B */
}
    }
    class A2B { /* new A2T; new A1T; new A1B */ }
  }
  class A1B { /* new A1T */ }
}

object Appl {
  def main(args: Array[String]) {
    new p1.A1T
    new p1.A1B
    new p1.p2.A2T
    new p1.p2.A2B
    new p1.p2.p3.A3
  }
}
```

Zadanie: zmieniamy linię `class A1B { /* new A1T */ }` na `class A1B { new A1T }` (“odkomentowujemy” kod), kompilujemy plik, uruchamiamy Appl i analizujemy wynik.

3. Zmieniamy zawartość `main.scala` na podaną poniżej, kompilujemy kod, analizujemy wyniki kompilacji:

```
package p1 {
  class A1
  package p2 {
    class A21
  }
}

package p1.p2 {
  class A22 { new A1 }
}
```

Usuwamy katalog p1

4. Zmieniamy zawartość `main.scala` na podaną poniżej, kompilujemy kod, analizujemy wyniki kompilacji:

```
package p1 {
  class A1
  package p2 {
    class A21
  }
}

package p1 {
  package p2 {
```

```
    class A22 { new A1 }
  }
}
```

Usuwamy katalog p1

5. Tworzymy plik `file1.scala`, wpisujemy w nim poniżej podane definicje:

```
package p1 {
  class A1 { println("p1.A1") }
  package p2 {
    class A21 { println("p1.p2.A21")}
  }
}
`
```

6. W pliku main.scala wpisujemy:

```
package p1 {
  package p2 {
    class A1 { println("p1.p2.A1") }
    class A22 {
      println("Calling new p1.p2.A1")
      new A1
      println("Calling new _root_.p1.A1")
      new _root_.p1.A1
      println("p1.p2.A22")
    }
  }
}

object Appl {
  def main(args: Array[String]) {
    new p1.A1
    new p1.p2.A21
    new p1.p2.A22
  }
}
```

7. Kompilujemy oba pliki (`scalac main.scala file1.scala`), uruchamiamy Appl (`scala Appl`) i analizujemy wyniki

8. Usuwamy katalog `p1`

3. **Importy**

1. W pliku `file1.scala` wpisujemy:

```
package p1 {
  class C1 {
    println("C1")
    def m1C1() { println("m1C1()") }
  }
  package p2 {
    class C21 { println("C21") }
    class C22 { println("C22") }
    class C23 { println("C23") }
    object C2 {
      def m1OC2() { println("m1OC2") }
      def m2OC2(c1: C1) {
        import c1._
        println("m2OC2")
        m1C1()
      }
    }
  }
}
```

2. W pliku `main.scala` wpisujemy:

```
import p1.p2.C2._
object Appl {
  import p1._
  new C1
  import p2.{C21, C22=>MyC22, C23=>_}
  new C21
  new C22
  new MyC22
  new C23
  def main(args: Array[String]) {
    m1OC2()
    m2OC2(new C1)
  }
}
```

3. Kompilujemy oba pliki (`scalac main.scala file1.scala`) i analizujemy wyniki (dlaczego wywołania `new C22` i `new C23` są niepoprawne)?

4. Usuwamy wywołania `new C22` i `new C23` , kompilujemy ponownie oba pliki, uruchamiamy `Appl` i analizujemy wyniki
5. Usuwamy katalog `p1`

4. **Obiekt pakietu**

1. Tworzymy katalog `p1` (`mkdir p1`) i przechodzimy do niego (`cd p1`)
2. Tworzymy plik `package.scala` (np. `touch package.scala`)
3. W pliku `package.scala` wpisujemy:

```
package object p1 {
  def m1(i: Int) = 2 * i
  def m2(d: Double) = 1.5 * d
}
```

4. Przechodzimy do katalogu nadrzędnego (`cd ..`), w którym jest plik `main.scala`
5. W pliku `main.scala` wpisujemy:

```
object Appl {
  def main(args: Array[String]) {
    import p1._
    println(m1(2))
    println(m2(5.5))
  }
}
```

6. Kompilujemy oba pliki (`scalac main.scala p1/package.scala`), analizujemy wyniki kompilacji (w tym kod bajtowy), a następnie uruchamiamy Appl (`scala Appl`)

5. **Generator haseł** (µ-projekt)

Zakładamy, że:

1. w pliku `logins.txt` zapisane są loginy kolejnych studentów:

```
jkowal
dnowak
itd.
```

2. hasła będą generowane w następujący sposób:

```
def randomPasswd(passwdLen: Int) = {
   val passwd = new StringBuilder(passwdLen)
   val allowedChars = (('0' to '9') ++
                       ('A' to 'Z') ++
                       ('a' to 'z')).toArray ++
                       Array('-', '.', '_')) // more special cha
rs?
   for (i <- 0 to passwdLen)
     passwd += allowedChars(
       util.Random.nextInt(allowedChars.length))

   passwd.toString
}
```

3. wygenerowane hasła będą zapisywane w pliku:

```
val outFile = new java.io.PrintWriter("login-passwds.txt")
...
for (login <- ...) {
  ...
  outFile.println(login + ":" + passwd) // przykładowa linia jko
wal:D4_xdf5$gR
}
...
outFile.close
```

Zadanie:

1. sprawdzić działanie metody `randomPasswd`
2. (w pliku `file1.scala`) zdefiniować obiekt `PasswdGen` i umieścić go w pakiecie `utils`
3. metodę generującą hasła nazwać `nextPasswd`
4. zdefiniować prywatne pola przechowujące odpowiednio `allowedChars` i `allowedChars.length`
5. zdefiniować prywatną metodę `nextChar` (wykorzystywaną przez `nextPasswd`)
6. przetestować `PasswdGen.nextPasswd`
7. w pliku `main.scala` dodać `import utils._` oraz napisać kod, który:
   - czyta kolejne loginy z pliku `logins.txt`
   - dla każdego loginu generuje losowe hasło
   - zapisuje parę login:hasło w pliku login-passwds.txt
     wskazówka: pomocne mogą okazać się ćw.1. (`try/catch/finally`) oraz założenie 3. (zapis do pliku)

Zadanie (opcjonalne): rozszerzyć możliwości metody `nextPasswd` w wybranym przez siebie kierunku (np. kontrola liczby znaków specjalnych w wygenerowanym haśle)

Zadanie (opcjonalne): dodać obsługę argumentów uruchomienia programu, określających pliki wejściowy i wyjściowy

6. **Enumerations**

Zadanie: uzupełnić poniżej wpisany kod

```
object ThreeColors extends __ {
   ___ ThreeColors = ___
   ___ Blue, White, Red = ___
}

import ___

object utils {
   def leadingActor(part: ThreeColors) = part match {
     ___ Blue ___ "Juliette Binoche"
     ___ White ___ "Zbigniew Zamachowski"
     ___ Red ___ "Irene Jacob"
   }
}

object Appl {
   def main(args: Array[String]) {
     for (part <- ThreeColors.___)
       println(part.___ + ": " + utils.leadingActor(___))
   }
}
```

tak, aby uruchomiony program (`scala Appl`) wypisał na ekranie:

```
Blue: Juliette Binoche
White: Zbigniew Zamachowski
Red: Irene Jacob
```

Zadanie: przeanalizować kod bajtowy obiektu `ThreeColors`

7. **Równość obiektów**

   1. Wpisujemy w pliku main.scala poniższy kod:

```
class Int2DVec(val x: Int, val y: Int)

object Appl {
  def checkPredicate(pred: Boolean, predAsString: String) {
    if (pred) println(predAsString + ": OK")
    else println(predAsString + ": Failed")
  }

  def main(args: Array[String]) {
    val v1 = new Int2DVec(4, 5)
    val v2 = new Int2DVec(4, 5)
    checkPredicate(v1 equals v2, "v1 equals v2")
    checkPredicate(!(v1 eq v2), "!(v1 eq v2)")
    checkPredicate(v1 == v2, "v1 == v2")
  }
}
```

   2. Kompilujemy, uruchamiamy i analizujemy wyniki

   3. Modyfikujemy definicję klasy `Int2DVec` :

```
class Int2DVec(val x: Int, val y: Int) {
  def equals(other: Int2DVec): Boolean =
    this.x == other.x && this.y == other.y
}
```

   4. Ponownie kompilujemy, uruchamiamy i analizujemy wyniki (dlaczego warunek `v1 == v2` jest nadal niespełniony?)

   5. Dodajemy override przed definicją metody `equals` :

```
class Int2DVec(val x: Int, val y: Int) {
  override def equals(other: Int2DVec): Boolean =
    this.x == other.x && this.y == other.y
}
```

   6. Kompilujemy kod i analizujemy komunikaty kompilatora

   7. Zmieniamy jeszcze raz definicję klasy `Int2DVec` :

```
class Int2DVec(val x: Int, val y: Int) {
  override def equals(other: Any): Boolean = {
    other match {
      case that: Int2DVec =>
        (this.x == that.x) && (this.y == that.y)
      case _ => false
    }
  }
  override def hashCode = 41 * (41 + x) + y
}
```

   8. Ponownie kompilujemy, uruchamiamy i analizujemy wyniki

Zadanie (opcjonalne): uruchomić i przeanalizować poniższy kod

```
class Int2DVec(val x: Int, val y: Int) {
  override def equals(other: Any): Boolean = {
    //print("[Int2DVec.equals(other: Any) called] ")
```

```
    other match {
      case that: Int2DVec =>
        (that canEqual this) && (this.x == that.x) && (this.y == tha
t.y)
        case _ => false
    }
  }
  override def hashCode = {
    //print("[Int2DVec.hashCode called] ")
    41 * (41 + x) + y
  }
  def canEqual(other: Any) = other.isInstanceOf[Int2DVec]
}

class MutableInt2DVec(var x: Int, var y: Int) {
  override def equals(other: Any): Boolean = {
    //print("[MutableInt2DVec.equals(other: Any) called] ")
    other match {
      case that: MutableInt2DVec =>
        (that canEqual this) &&
        (this.x == that.x) &&
        (this.y == that.y)
      case _ => false
    }
  }
  override def hashCode = {
   //print("[MutableInt2DVec.hashCode called] ")
   41 * (41 + x) + y
  }
  def canEqual(other: Any) = other.isInstanceOf[MutableInt2DVec]
}

class Int2DArrow(x: Int, y: Int, val width: Int)
extends Int2DVec(x, y) {
  override def equals(other: Any): Boolean = {
    //print("[Int2DArrow.equals(other: Any) called] ")
    other match {
      case that: Int2DArrow =>
        (that canEqual this) &&
        (this.width == that.width) &&
        super.equals(that)
      case _ => false
    }
  }
  override def hashCode = {
    //print("[Int2DArrow.hashCode called] ")
    41 * super.hashCode + width
  }
  override def canEqual(other: Any) = other.isInstanceOf[Int2DArrow]
}

object testSuite {
  private var fail = false
  private def reset() { fail = false }
  private def signallFail() { fail = true }
  private def printTestResult() {
    if (fail) println("******* TESTS FAILED *******")
    else println("******* TESTS PASSED *******")
  }

  import scala.collection.mutable._

  def checkPredicate(pred: Boolean, predAsString: String) {
    if (pred) {
      println(predAsString + ": OK")
    } else {
      println(predAsString + ": Failed")
      signallFail()
    }
  }
  def tc1() {
    val v1, v2 = new Int2DVec(4, 5)
    checkPredicate(v1 equals v2, "v1 equals v2")
    checkPredicate(!(v1 eq v2), "!(v1 eq v2)")
    checkPredicate(v1 == v2, "v1 == v2")
  }
  def tc2() {
    val v1, v2 = new Int2DVec(4, 5)
    val vectors = HashSet(v1)
    checkPredicate(vectors.contains(v1), "vectors.contains(v1)")
    checkPredicate(vectors.contains(v2), "vectors.contains(v2)")
  }
  def tc3() {
    val v1, v2 = new Int2DVec(4, 5)
    val v2AsAny: Any = v2
    checkPredicate(v1 equals v2AsAny, "v1 equals v2AsAny")
    checkPredicate(v1 == v2AsAny, "v1 == v2AsAny")
  }
  def tc4() {
    var mutV1 = new MutableInt2DVec(4, 5)
    val mutVectors = HashSet(mutV1)
    checkPredicate(mutVectors.contains(mutV1),
```

```
                  "[Before change] mutVectors.contains(mutV1)")
    mutV1.x *= 2
    checkPredicate((!mutVectors.contains(mutV1)),
                  "[After change] (!mutVectors.contains(mutV1))")
  }
  def tc5() {
    val v1 = new Int2DVec(4, 5)
    val a1 = new Int2DArrow(4, 5, 1)
    checkPredicate(!(v1 equals a1), "!(v1 equals a1)")
    checkPredicate(!(v1 == a1), "!(v1 == a1)")
    checkPredicate(!(a1 equals v1), "!(a1 equals v1)")
    checkPredicate(!(a1 == v1), "!(a1 == v1)")
    checkPredicate(!(HashSet[Int2DVec](v1) contains a1),
                  "!    (HashSet[Int2DVec](v1) contains a1)")
    checkPredicate(!(HashSet[Int2DVec](a1) contains v1),
                  "!(HashSet[Int2DVec](a1) contains v1)")
  }
  def run() {
    reset()
    try {
      tc1(); tc2(); tc3(); tc4(); tc5()
    } finally { printTestResult() }
  }
}

object Appl {
  def main(args: Array[String]) {
    testSuite.run()
  }
}
```

8. **Specyfikatory dostępu** (uzupełnienie)

    1. Wpisujemy w pliku `main.scala` :

```
package p1
package p11 {
  private[p1] class C1 {
    protected[p1] def protP1MC1() {
      print("[protP1MC1]: "); privP11MC1()
    }
    private[p1] def privP1MC1() {
      print("[privP1MC1]: ");  privP11MC1()
    }
    private[p11] def privP11MC1() {
      print("[privP11MC1]: "); println((new C11).a13)
    }
    protected[p11] def protP11MC1() {
      print("[protP11MC1]: "); privP11MC1()
    }
    private[C1] var a11 = 11
    private[this] var a12 = 12
    class C11 {
      private[C1] val a13 = 13
    }
    override def equals(other: Any): Boolean = other match {
      case that: C1 => (this.a11 == that.a11)
      // && (this.a12 == that.a12) // (1)
      case _ => false
    }
  }
}
package p12 {
  import p11._
  object O1 {
    private[p12] val ao1 = new C1
  }
  class C12 extends C1 {
    def mC121() { print("[mC121]: "); super.protP11MC1() }
    // def mC122() { print("[mC122]: "); super.privP11MC1() } //
 (2)
  }
}
object Appl {
  def main(args: Array[String]) {
    val c1 = new p11.C1
    c1.protP1MC1()
    c1.privP1MC1()
    val c12 = new p12.C12
    c12.mC121()
  }
}
```

    2. Kompilujemy kod (`scalac main.scala`) i analizujemy wynik kompilacji. Gdzie znajduje się plik `Appl.class` ?

    3. Przechodzimy do katalogu `p1` , wpisujemy `scala Appl` i analizujemy wynik
    4. Analizujemy kod bajtowy `Appl.class` . Jaka jest pełna nazwa klasy?
    5. Wpisujemy `scala p1.Appl` i analizujemy wynik
    6. Przechodzimy do katalogu nadrzędnego (`cd ..`), wpisujemy `scala p1.Appl` i analizujemy wynik
    7. Usuwamy komentarz (“odkomentowujemy”) fragment `(1)` , kompilujemy i

analizujemy komunikaty kompilatora
8. “Zakomentowujemy” ponownie fragment `(1)`, “odkomentowujemy” frament `(2)`, kompilujemy i analizujemy komunikaty kompilatora

Zadanie: znaleźć w definicji klasy `C1` metody, do których dostęp jest określony odpowiednikami specyfikatorów `package` i `protected` z Javy

9. **Case classes/objects, sealed classes**

    1. Wpisujemy w pliku `main.scala` poniższy kod:

    ```
    class CC(x: Int)
    ```

    2. Kompilujemy plik i analizujemy kod bajtowy (dodajemy opcję `-p`) pliku wynikowego
    3. Zmieniamy definicję klasy `CC`:

    ```
    case class CC(x: Int)
    ```

    4. Kompilujemy i analizujemy wyniki kompilacji (ile plików zostało utworzonych?)
    5. Analizujemy kod bajtowy obu plików (z opcją `-p`)
    6. Definiujemy (w pliku `main.scala`) obiekt `O1` (usuwamy poprzednią zawartość pliku):

    ```
    object O1 { val x = 2 }
    ```

    7. Kompilujemy i analizujemy wyniki kompilacji (ile plików zostało utworzonych?)
    8. Analizujemy kod bajtowy obu plików (z opcją `-p`)
    9. Zmieniamy definicję obiektu `O1` na:

    ```
    case object O1 { val x = 2 }
    ```

    10. Kompilujemy i analizujemy wyniki kompilacji (ile plików zostało utworzonych?)
    11. Analizujemy kod bajtowy obu plików (z opcją `-p`)
    12. W pliku `main.scala` wpisujemy kolejno poniższe definicje klas, kompilujemy i analizujemy wyniki kompilacji:

    ```
    class C1(val x: Int)
    class SubC1(x: Int) extends C1(x)
    ```

    ```
    class C1(val x: Int)
    case class SubC1(x: Int) extends C1(x)
    ```

    ```
    class C1(val x: Int)
    case class SubC1(y: Int) extends C1(y)
    ```

    ```
    case class C1(x: Int)
    class SubC1(x: Int) extends C1(x)
    ```

    ```
    case class C1(x: Int)
    case class SubC1(x: Int) extends C1(x)
    ```

    13. W pliku file1.scala definiujemy `T1`:

    ```
    trait T1 {
      def f(i: Int): Unit
    }
    ```

    Uwaga: pliki `file1.scala` i `main.scala` powinny być w tym samym katalogu

    14. W pliku `main.scala` wpisujemy

    ```
    class C1 extends T1 {
      def f(i: Int) = { println(i) }
    }
    object Appl {
      def main(args: Array[String]) {
        (new C1).f(3)
      }
    }
    ```

    15. Kompilujemy pliki (`scalac main.scala file1.scala`) i uruchamiamy Appl (`scala Appl`)
    16. Modyfikujemy definicję `T1`:

    ```
    sealed trait T1 {
    def f(i: Int): Unit
    }
    ```

    17. Kompilujemy ponownie oba pliki (`scalac main.scala file1.scala`) i analizujemy wyniki kompilacji
    18. Przenosimy definicję klasy `C1` do pliku `file1.scala`:

```
sealed trait T1 {
  def f(i: Int): Unit
}
class C1 extends T1 {
  def f(i: Int) = { println(i) }
}
```

19. Kompilujemy ponownie oba pliki (`scalac main.scala file1.scala`)
20. Uruchamiamy Appl (`scala Appl`)

10. **Klasy wewnętrzne** (ćwiczenie opcjonalne)
    1. Wpisujemy w pliku main.scala poniższy kod:

```
class Outer {
  class Inner {
    def m1In(arg: Outer) = arg
  }
  def m1Out(inArg : this.Inner) = inArg
  def m2Out(outArg : Outer#Inner) = outArg
}
```

    a następnie kompilujemy go (`scalac main.scala`)

    2. Analizujemy wyniki kompilacji (ile plików zostało utworzonych?, jakie są ich nazwy? jakie są typy parametrów i wyników metod?)
    3. Uruchamiamy konsolę (`scala`)
    4. Wpisujemy (wklejamy) poniższy kod

```
class Outer {
  class Inner {
    def m1In(arg: Outer) = arg
  }
  def m1Out(inArg : this.Inner) = inArg
  def m2Out(outArg : Outer#Inner) = outArg
}
```

    5. Wpisujemy kolejno:

```
scala> :javap Outer
scala> :javap Outer$Inner
```

    i analizujemy kod bajtowy.

    6. Wpisujemy kolejno:

```
scala> :javap -c Outer
scala> :javap -c Outer$Inner
scala> :javap -v Outer
scala> :javap -v Outer$Inner
```

    i analizujemy kod bajtowy.

    7. Wpisujemy kolejno poniższe linie i analizujemy wyniki:

```
scala> val out1 = new Outer
scala> val in1 = new Outer.Inner
scala> val in1 = new Outer#Inner
scala> val out2 = new Outer
scala> val in1 = new out1.Inner
scala> val in2 = new out2.Inner
scala> in1.getClass
scala> in2.getClass
scala> in1.getClass == in2.getClass
scala> in1.m1In(out1)
scala> in1.m1In(out2)
scala> out1.m1Out(in1)
scala> out1.m1Out(in2)
scala> out1.m2Out(in1)
scala> out1.m2Out(in2)
```

    8. Definiujemy (w konsoli) klasę `Outer` :

```
class Outer {
  def m1Out(inArg : Outer.Inner) = inArg
  def m2Out(outArg : Outer.Inner) = outArg
}
```

    9. Definiujemy (w konsoli) obiekt `Outer` :

```
object Outer {
  class Inner {
    def m1In(arg: Outer) = arg
  }
}
```

    i analizujemy odpowiedź (komunikat REPL)

    10. Przechodzimy do trybu `:paste` , wklejamy poniższy kod:

```
object Outer {
  class Inner {
    def m1In(arg: Outer) = arg
  }
}
class Outer {
  def m1Out(inArg : Outer.Inner) = inArg
  def m2Out(outArg : Outer.Inner) = outArg
}
```

a następnie wychodzimy z trybu `:paste` (`[Ctrl+d]`).

11. Wpisujemy kolejno poniższe linie i analizujemy wyniki:

```
scala> val out1 = new Outer
scala> val in1 = new Outer.Inner
scala> val out2 = new Outer
scala> val in2 = new Outer.Inner
scala> in1.getClass
scala> in2.getClass
scala> in1.getClass == in2.getClass
scala> in1.m1In(out1)
scala> in1.m1In(out2)
scala> out1.m1Out(in1)
scala> out1.m1Out(in2)
scala> out1.m2Out(in1)
scala> out1.m2Out(in2)
```

11. **Elementy programowania generycznego** (ćwiczenie opcjonalne)

1. W pliku `main.scala` wpisujemy:

```
class SuperA1
class A1 extends SuperA1
class SubA1 extends A1

class GenInVar[T]
class GenCoVar[+T]
class GenContrVar[-T]

object Appl {
  def mInVarA1(inVar: GenInVar[A1]) {}
  def mCoVarA1(coVar: GenCoVar[A1]) {}
  def mContrVarA1(contrVar: GenContrVar[A1]) {}
  def main(args: Array[String]) {
    //mInVarA1(new GenInVar[SuperA1]) // (1)
    mInVarA1(new GenInVar[A1])
    //mInVarA1(new GenInVar[SubA1]) // (2)

    //mCoVarA1(new GenCoVar[SuperA1]) // (3)
    mCoVarA1(new GenCoVar[A1])

    // GenCoVar[SubA1] <: GenCoVar[A1]
    mCoVarA1(new GenCoVar[SubA1])

    // GenContrVar[SuperA1] <: GenContrVar[A1]
    mContrVarA1(new GenContrVar[SuperA1])
    mContrVarA1(new GenContrVar[A1])
    //mContrVarA1(new GenContrVar[SubA1]) // (4)
  }
}
```

2. Kompilujemy kod i uruchamiamy (`scala Appl`)
3. “Odkomentowujemy” kolejno linie `(1)` - `(4)` , za każdym razem kompilujemy i analizujemy komunikaty błędu
4. W pliku main.scala wpisujemy:

```
class SuperA1
class A1 extends SuperA1
class SubA1 extends A1

class GenInVar[T](private var arg: T) {
  def in(a: T) {}
  def inOut(a: T): T = a
  def out: T = arg
}

class GenCoVar[+T](private val arg: T) {
  //def in(a: T) {} // (1)
  def inOut[A >: T](a: A): A = a
  def out: T = arg
}

class GenContrVar[-T] /*(private val arg: T)*/ { (2)
  def in(a: T) {}
  //def inOut(a: T): T = a (3)
  //def out: T = // ?
}

object Appl {
  def main(args: Array[String]) {
```

```
    val genInVarA1 = new GenInVar[A1](new A1)
    val genCoVarA1 = new GenCoVar[A1](new A1)
    val genContrVarA1 = new GenContrVar[A1]

    //genInVarA1.in(new SuperA1) // (4)
    genInVarA1.in(new A1)
    genInVarA1.in(new SubA1)

    //genInVarA1.inOut(new SuperA1) // (5)
    genInVarA1.inOut(new A1)
    genInVarA1.inOut(new SubA1)

    val r1: SuperA1 = genInVarA1.out
    val r2: A1 = genInVarA1.out
    //val r3: SubA1 = genInVarA1.out // (6)

    val r4: SuperA1 = genCoVarA1.inOut(new SuperA1)
    //val r5: A1 = genCoVarA1.inOut(new SuperA1) // (7)
    //val r6: SubA1 = genCoVarA1.inOut(new SuperA1) // (8)
    val r7: SuperA1 = genCoVarA1.inOut(new A1)
    val r8: A1 = genCoVarA1.inOut(new A1)
    //val r9: SubA1 = genCoVarA1.inOut(new A1) // (9)
    val r10: SuperA1 = genCoVarA1.inOut(new SubA1)
    val r11: A1 = genCoVarA1.inOut(new SubA1)
    //val r12: SubA1 = genCoVarA1.inOut(new SubA1) // (10)
    val r12: SuperA1 = genCoVarA1.out
    val r13: A1 = genCoVarA1.out
    //val r14: SubA1 = genCoVarA1.out // (11)

    //genContrVarA1.in(new SuperA1) // (12)
    genContrVarA1.in(new A1)
    genContrVarA1.in(new SubA1)
  }
}
```

5. Kompilujemy i uruchamiamy `Appl`
6. “Odkomentowujemy” kolejno linie `(1)` - `(12)`, za każdym razem kompilujemy i analizujemy komunikaty błędu