IMPERATIVE PROGRAMMING 3 TT2019

SHEET 1

GABRIEL MOISE

Question 1

First, I defined the Shape class, with a private value id, which will be used for us to identify the type of each Shape subclass we will have:

```
class Shape (private val id : String){
  def getType : String = id
}
```

Then, each shape we will create will extend Shape with a different id and will have the "retrieve" and "change" functions needed to obtain the sizes and modify them by the user of the interface:

```
class Rectangle (private var width : Double, private var height : Double) extends Shape("rectangle") {
  def retrieve : (Double,Double) = (width,height)
  def change (newWidth : Double, newHeight : Double) : Unit = {
    width = newWidth
    height = newHeight
  }
}


class Square (private var size : Double) extends Shape("square") {
  def retrieve : Double = size
  def change (newSize : Double) : Unit = size = newSize
}


class Ellipse (private var major : Double, private var minor : Double) extends Shape("ellipse") {
  def retrieve : (Double,Double) = (major,minor)
  def change (newMajor : Double, newMinor : Double) : Unit = {
    major = newMajor
    minor = newMinor
  }
}
```

```
class Circle (private var radius : Double) extends Shape("circle") {
 def retrieve : Double = radius
 def change (newRadius : Double) : Unit = radius = newRadius
}
```

In order for us to identify all the squares, we check for each shape if its id is either "square", which means it's definitely a square, or "rectangle", and we additionally verify if the 2 dimensions of it are equal. If they are, then we have a square, if not, we don't. The same reasoning is applied for the ellipse and circle case.

My other idea for this was without the id value I gave to each class, but instead to create 2 functions "identifySquares" and "identifyCircles" that would test each Shape-object from the list we have with a match instruction. They look like this: (they return an array consisting only of squares/circles objects)

```
def identifySquares (set : Array[Shape]) : Array[Shape] = {
  var N = set.size
  var squares = new Array[Shape](N)
  var k = 0
  for (i <- 0 until N)
    set(i) match {
      case Rectangle(w,h) => if (w==h) {squares(k) = set(i) ; k += 1}
      case Square(s) => {squares(k) = set(i) ; k += 1}
      case _ => {}
    }
  squares
 }


 def identifyCircles (set : Array[Shape]) : Array[Shape] = {
  var N = set.size
  var circles = new Array[Shape](N)
  var k = 0
  for (i <- 0 until N)
    set(i) match {
      case Ellipse(major,minor) => if (major==minor) {circles(k) = set(i) ; k += 1}
      case Circle(s) => {circles(k) = set(i) ; k += 1}
      case _ => {}
    }
  circles
 }
```

Both my ideas rely on inheritance, although my first implementation actually has a function which all the subclasses use, whereas the second one doesn't have any functions from the Shape class to use. The advantage of my implementations is that it is very easy to add new shapes, for example if we want to add a Triangle, we could simply create a new subclass for Shape, with the id =”triangle” (for the first method), and the functions that we have for checking squares and circles don't have to be modified. If I defined each class to be independent of each other, I couldn't have used polymorphism and I would have had more code to write.

Question 2

```
class Rectangle (var width : Int, var height : Int) {

  var area = width * height

  ...

}
```

The problems with this piece of code are:

- the width and height variables are not private and therefore can be modified, thus breaking the principle of encapsulation
- the width and height are of a wrong type, they should be Double variables because the dimensions might not be integer values (not a big problem if they are intended to be integers, but it's worth mentioning this)
- there should be a datatype invariant, which would incorporate the obvious relationship area = width* height. Thus, the invariant can be violated, since the width and height are declared as a variables and not as private ones
- we might need this class to be a case class in case we want to use equality or pattern-matching on it (not too important aspect)
- the area should be obtained by the user with a separate function, getArea, which should just return the value, which is ought to be updated each time one of the dimensions is modified in any context (efficiency)

My version would look like this:

```
case class Rectangle (private var width : Double, private var height : Double) {

  // DTI : area = width * height && width > 0 && height > 0

  private var area = width * area

  // This should be returned immediately, without needing a multiplication

  def getArea : Double = area

  ... (any function that modifies the dimensions will also modify the area)

}
```

Question 3

```
case class Rectangle (var width : Int, var height : Int)


class Slab (private val __dimension : Rectangle) {

  private val _dimension = new Rectangle (__dimension.width, __dimension.height)
  private val _area = _dimension.width * _dimension.height


  def dimension = new Rectangle (_dimension.width, _dimension.height)
}


/**
class Slab (private val _dimension : Rectangle) {
  private val _area = _dimension.width * _dimension.height


  def dimension = _dimension
}
*/
object Question3 {
  def main (args : Array[String]) = {
    var rectangle = new Rectangle(1,2)
    var slab = new Slab(rectangle)
    println(slab.dimension) // Both versions print Rectangle(1,2)
    slab.dimension.width = 10
    println(slab.dimension) // Original version prints Rectangle(10,2), my version prints Rectangle(1,2)
  }
}
```

The problem that is illustrated in main is that the parameters of the dimension of the Slab object can be modified, although they are private to the user and this happens because when calling the dimension function from Slab, we get the address of the Rectangle object and we can therefore modify it, which should not be allowed.

My solution would be to create new Rectangle objects in the Slab class, so when we modify the data from the dimension parameter, we just modify the object we created in the Slab class. Therefore, every time we ask for the dimension of the Slab object, we get the original version from the Rectangle object, and this is the effect we wanted (we shouldn't be able to modify it as a user due to its private type).

Question 4

The output for this input is going to be

```
Accepted for rendering.

Accepted for ray-trace rendering.
```

This is due to the fact that r1 is considered to be of class OpaqueTriangle, which is a subclass of the Triangle class, and using the "is a" rule, we can say that any OpagueTriangle object “is a” Triangle, but not vice-versa. This means that, because r1 is of type Renderer, the method which will be used is the one which returns "Accepted for rendering.", because in this class there is not the other “accept” function (the RayTracingRenderer is a subclass of Renderer) and the accept function is not overridden in the dynamic class.

The second output will be "Accepted for ray-trace rendering.", since r2 is defined to be a RayTracingRenderer object, so the compiler has to decide which “accept” function to use, since it now has two overloaded functions, and it will choose the one which receives an OpaqueTriangle type argument, rather than a Triangle one, because of pattern-matching.

To change the first output to also be "Accepted for ray-trace rendering.” we simply need to change the type of the argument that accept receives to Triangle, and to override the accept function:

```
class RayTracingRenderer extends Renderer {

  override def accept(a: Triangle) = println("Accepted for ray-trace rendering.")

}
```

Question 5

```
class Ellipse (private var _a: Int, private var _b: Int) {

  def a = _a

  def a_= (a: Int) : Unit = {_a = a}

  def b = _b

  def b_= (b: Int) : Unit = {_b = b}

  def swap : Unit = {

    var t = _a

    _a = _b

    _b = t

  }

}

class LoggedEllipse (private var _a : Int, private var _b : Int) extends Ellipse (_a,_b) {

  private var incr : Int = 0

  // DTI : incr = # increases in the area of the Ellipse (when either _a or _b get increased)

  override def a_= (a: Int) : Unit = {if (a>_a) incr += 1 ; _a = a}

  override def b_= (b: Int) : Unit = {if (b>_b) incr += 1 ; _b = b}

  def getIncrease : Int = incr

}

object Question5 {

  def main (args : Array[String]) = {

     var ellipse = new LoggedEllipse(3,5)

     ellipse.a_=(4) //(4,5)
     ellipse.b_=(2) //(4,2)

     ellipse.b_=(6) //(4,6)

     ellipse.a_=(1) //(1,6)

     ellipse.swap   //(6,1)

     ellipse.a_=(5) //(5,1)

     println(ellipse.getIncrease) // we get 3 as a result, although we should only have 2 increases, but the swap part doesn't actually
reverse the parameters because it is not overridden in the LoggedEllipse subclass

  }

}
```

The problem here is that when we use the function swap from the Ellipse class, it doesn't modify the current values of the LoggedEllipse object we have. Same goes for the "a" and "b" functions, which are not overridden in the subclass, and for instance if we print ellipse.a, we would get 5, because we only applied swap to the parameters of the Ellipse object. Therefore, we would need to override every function from Ellipse to make it work. Instead, my solution looks like this:

```
class LoggedEllipse (private var __a : Int, private var __b : Int) extends Ellipse (__a,__b) {

  private var incr : Int = 0

  // DTI : incr = # increases in the area of the Ellipse (when either _a or _b get increased)

  override def a_= (aa: Int) : Unit = {if (aa>a) incr += 1 ; super.a_=(aa)}

  override def b_= (bb: Int) : Unit = {if (bb>b) incr += 1 ; super.b_=(bb)}

  def getIncrease : Int = incr

}
```

The difference it that now I use the functions "a" and "b" that are inherited from Ellipse instead of its values, which are not the ones I want, and we also use the functions from Ellipse ("super.a_=" and "super.b_=") to modify the parameters, thus modifying them in the Ellipse object, too.

Now, we will have in main:

```
ellipse.a_=(4) //(4,5)

ellipse.b_=(2) //(4,2)

ellipse.b_=(6) //(4,6)

ellipse.a_=(1) //(1,6)

ellipse.swap   //(6,1)

ellipse.a_=(5) //(5,1)

println(ellipse.a) // 5

println(ellipse.b) // 1

println(ellipse.getIncrease) // 2
```

Question 6

The trait I use for the second part of the question:

```
trait Transmittable {
  def transmit
}


class Text(init: Int) extends Transmittable {
  // text = buffer[0..gap) ++ buffer[max-len+gap..max)


  private var buffer = new Array[Char](init)
  private var len = 0
  private var gap = 0
  private def max = buffer.length
  def length = len
  def charAt(pos: Int) : Char = {..}
  def insert(pos: Int, ch: Char) = {..}
  def transmit = {..}
}
```

PlaneText has a Text as an instance variable, so the functions have to use the corresponding ones from the Text class

```
class PlaneText extends Transmittable {
  private val _text : Text = new Text()
  // _text = _text.buffer[0.._text.gap) ++ _text.buffer[_text.max-_text.len+_text.gap.._text.max)


  def length = _text.length
  def charAt (pos: Int) : Char = _text.charAt(pos)
  var lstart = new Array[Int](MAXLINES)
  var nlines = 0
  // Invariant: 0 ≤ nlines ≤ MAXLINES
  // Abs: lstart = lstart[0..nlines)
  def insert(pos: Int, ch: Char) {
    _text.insert(pos,ch)
    // update lstart and nlines
  }
  def transmit = {..}

}
```

The function we want to use for the second part is:

```
def transmit (text : Transmittable) = text.transmit
```

This method encourages loose coupling as these 2 classes do not depend closely of the details of one another, high cohesion because of the close relationship between the variables of each class, and also avoids inheritance so the Fragile Base Class problem is solved. Two consequences of this is that we lose polymorphism and every method from the PlaneText class has to call the corresponding method from the Text class and this is not time-efficient.

Question 7

```
class Rectangle (private val width:Int, private val height:Int) {

/** The version of the unexperienced programmer:

  def == (other : Rectangle) : Boolean =

    this.width == other.width && this.height == other.height

*/
```

My function that allows Rectangles to be used with the scala collections library as the equals function is overridden properly and also the HashCode function.

```
  override def equals (other: Any) : Boolean =

    other match {

      case oth : Rectangle => this.width == oth.width && this.height == oth.height

      case _ => false

    }

  override def hashCode = (width,height).##

}

object Question7 {

  def main (args : Array[String]) = {

    // Tests for equality method

    var r1 = new Rectangle(20,30)

    var r2 = new Rectangle(20,30)

    var r3 = new Rectangle(50,60)

    println(r1 == r2, r2 == r3) // Correctly returns "true, false"

    println(r1 equals r2, r2 equals r3) // Incorrectly returns "false, false" in the first case, but in the second it returns “true,false”

    val set = scala.collection.mutable.HashSet(r1)

    println(set.contains(r2)) // Before overriding the HashCode function this returned “false”, but after that it returns “true”

  }
```

```
}
```

So, we had to solve 4 problems:

1. Defining equals with the wrong signature.

We solved it after we saw that the “equals” relation doesn't work in the main and we used the good version (the one that is not commented)

2. Changing equals without also changing hashCode.

We solved it by overriding the HashCode function in Rectangle so that it works for primitive values, reference types, and null.

3. Defining equals in terms of mutable fields.

We solved it by changing the parameters of Rectangle from var to private val so that they cannot be modified.

4. Failing to define equals as an equivalence relation.

We need the following properties for equals:

- to be reflexive
- to be symmetric
- to be transitive
- to be consistent (which it is due to the fixed dimensions of the Rectangle)
- x == null to be false for each non-null x and true otherwise

```
object Question7 {
 def main (args : Array[String]) = {
  // Tests for equality method
  var r1 = new Rectangle(20,30)
  var r2 = new Rectangle(20,30)
  var r3 = new Rectangle(20,30)
  println(r1 == r1) // reflexivity
  println(r1 == r2, r2 == r1) // symmetry
  println(r1 == r2, r2 == r3, r3 == r1) // transitivity
  var r = null : Rectangle
  println(r == null) // True
  println(r1 == null) // False
 }
}
```

This is the new collection of tests that ensure the fourth point of the proof.