Tuan Nguen

Question 1:

One of the possibilities is to have a different class for each of the four shapes, but then we would have a lot of repetitive code. Squares are rectangles, but with the same height as width. Analogically for circles – they are ellipses with two equal axes. Another possibility is to have a single class for all the shapes, but then we would have to have a field which categorizes whether the shape is rectangle of ellipse. So it is best to have a single class for the rectangles, and a class for ellipses. That way we can do not need to write repeated code, but still able to distinguish between ellipse and rectangle.

```
class Rectangle{
  private var width: Double = 0.0
  private var height: Double = 0.0

  def change(w: Double, h: Double): Unit = {
    width = w; height = h
  }

  def allSquares(s: Array[Rectangle]): Unit = {
    var array = new Array[Rectangle](s.size)
    var index = 0; var i = 0
    // Invariant: array[0..index) contains all the squares in s[0..i)
    //         && 0 <= index <= s.length && 0 <= i <= s.length
    while(i < s.length){
      if(s(i).width == s(i).height){ array(index) = s(i); index += 1}
      i += 1
    }
    i = 0
    // Invariant: array[0..i) has been printed && 0 <= i <= index
    while(i < index){
      array(i).printWH()
      i += 1
    }
  }

  def printWH(): Unit = {
    println("The width is: " + width + ", and the height is: " + height)
  }
}

class Ellipse{
  private var major: Double = 0
  private var minor: Double = 0

  def change(maj: Double, min: Double): Unit = {
    major = maj; minor = min
  }

  def allCircles(s: Array[Ellipse]): Unit = {
    var array = new Array[Ellipse](s.size)
    var index = 0; var i = 0
    // Invariant: array[0..index) contains all the squares in s[0..i)
    //         && 0 <= index <= s.length && 0 <= i <= s.length
    while(i < s.length){
      if(s(i).major == s(i).minor){ array(index) = s(i); index += 1}
      i += 1
    }
    i = 0
    // Invariant: array[0..i) has been printed && 0 <= i <= index
    while(i < index){
      array(i).printMajMin()
      i += 1
    }
  }

  def printMajMin(): Unit = {
    println("The major is: " + major + ", and the minor is: " + minor)
  }
}
```

Question 2:
It would be better if the fields of the class are encapsulated, so if the fields are private we would need a function to get and set them. This way they values of the fields cannot be changed by mistake. Another design error is that if we change the value of any of the fields, the area would not be updated. To avoid that, whenever a function to change one of the fields is called, we can update the area.

Question 3:
Even though the argument of the class is val, the fields of that argument can still be changed. So if one of fields is changed, _area will not be updated and the invariant breaks. To fix that, we can have a function construct that changes the _dimension field of the class. That way the field is encapsulated, and the width and the height cannot be changed, unless construct is called.

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

class Slab{
  private var _dimension: Rectangle = null
  private var _area: Int = 0

  // The problem is that the fields of _dimension can be changed even
  // though that it is a val
  def construct(r: Rectangle): Unit = {
    _dimension = r
    _area = _dimension.width * _dimension.height
  }

  def printRec(): Unit = {
    println("The width is: " + _dimension.width + ",
            and the height is " + _dimension.height)
    println(_area)
  }
}
```

Question 4:
The result of the original program is going to be one time “Accepted for rendering.” and one time “Accepted for ray-trace rendering.” This is because they are different functions. The function is not overridden, it is overloaded, so each of the objects r1 and r2 are going to use the method from its static class. In order to change the output form r1.accept(a), we need to override the accept method in RayTracingRenderer.

```
class Triangle
class OpaqueTriangle extends Triangle
class Renderer{
  def accept(a: Triangle) = println("Accept for rendering.")
}
class RayTracingRenderer extends Renderer{
  def  accept(a:  OpaqueTriangle)  =  println("Accepted  for  ray-trace
rendering.")

  /** In this case there will be two methods with the same name
    * and same argument list. In this case the dymanic binding rule
    * applies. */
  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) = {_a = a}

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

  /** If swap is defined this way, then we would need
    * to override it in the LoggedEllipse class,
    * because _a and _b would not swap in the subclass */
  def swap = {
    val x = _a
    _a = _b
    _b = x
  }

  /** This function does not need to be overriden in the subclass */
  def swapCorrect = {
    val x = a
    a_=(b)
    b_=(x)
  }
}

class LoggedEllipse(private var major: Int, private var minor: Int)
extends Ellipse(major, minor){
  private val area = scala.math.Pi * major * minor
  private var change: Double = 0.0

  override def a_=(a: Int) = {
    major = a
    val newArea = scala.math.Pi * major * minor
    change = newArea / area
  }

  override def b_=(b: Int) = {
    minor = b
    val newArea = scala.math.Pi * major * minor
    change = newArea / area
  }

  def getIncrease = change

  def printArea = println("The area is: " + area)
}
```

Question 6:

a) Instead of making PlaneText a subclass of Text, we can create a new Text object in the class. For each of the functions in PlaneText that override the ones in Text, we can just call the functions on the object. This way the two objects are more loosely coupled.

b)

```
trait TextTrait{
  /** Clears the buffer */
  def clear()
  /** Insert char in a position */
  def insert(pos: Int, ch: Char)

  /** Insert string in a position */
  def insert(pos: Int, s: String)

  /** Insert a whole text in a position */
  def insertRange(pos: Int, t: Text, start: Int, nchars: Int)

  /** Insert a file */
  def insertFile(pos: Int, in: java.io.Reader)

  /** Delete a char in a position */
  def deleteChar(pos: Int)

  /** Delete n number of characters
    * from a certain position */
  def deleteRange(start: Int, len: Int)

  def transmit(text: Any): Unit = {
    text match{
      case that: Text => transmit(that: Text)
      case that: PlaneText => transmit(that: PlaneText)
      case _ => new Error
    }
  }
}
```

c) With the proposed changes there are two benefits: the two classes are more loosely couple, and the delegated member text can be managed in the PlaneText class. This fixes the fragile base class problem, because in PlaneText we do not rely on how the methods are defined in Text. Although, there are two problem with this change: first, we will have to write a lot of repeated code, and second, no polymorphism can be used.

Question 7:

```
object Test{
  def main(args: Array[String]) = {
    val r1 = new Rectangle(20, 30)
    val r2 = new Rectangle(20, 30)
    val r3 = new Rectangle(50, 60)
    assert(r1 == r2)
    assert(!(r2 == r3))

    val set = new scala.collection.mutable.HashSet[Rectangle]
    set += r1
    set += r2
    set += r3

    val testRect1 = new Rectangle(20, 30)
    assert(set.contain(testRect1)) // This will throw error
  }
}
```

In this set of test, the last assertion will throw error. At first glance, this equals methods works, but when we put the objects into a set, it does not behave as desired. When we call the method contains, the compiler thinks we are calling equals with type Any. Hence, we are not overriding anything, we are just overloading it. We also need to change the hashCode function.

```
class Rectangle(val width: Int,val height: Int){
  /** This is the version that overloades equals */
  //def ==(other: Rectangle): Boolean = {
  //  this.width == other.width && this.height == other.height
  //}

  /** If we want to use hashSet we need to override hashCode */
  override def hashCode = (width, height).##

  override def equals(other: Any) =
    other match{
      case that: Rectangle =>
        (that canEqual this) &&
        this.width == that.width && this.height == that.height
      case _ => false
  }

  def canEqual(other: Any) = other.isInstanceOf[Rectangle]
}

class ColouredRectangles(width: Int, height: Int, val colour: String)
    extends Rectangle(width, height){
  override def hashCode = (super.hashCode, colour).##
  override def equals(other: Any) = other match {
    case that: ColouredRectangles =>
      (this.colour == that.colour) && super.equals(that)
    case _ => false
  }
  override def canEqual(other: Any) =
    other.isInstanceOf[ColouredRectangles]
}

object Test{
  def main(args: Array[String]) = {
    val r1 = new Rectangle(20, 30)
    val r2 = new Rectangle(20, 30)
    val r3 = new Rectangle(50, 60)

    //println(r1 == r2, r2 == r3)
    //r3.width = 20; r3.height = 30
    //println(r1 equals r2, r2 equals r3)

    val set = new scala.collection.mutable.HashSet[Rectangle]
    set += r1
    set += r2
    set += r3
    val testRect1 = new Rectangle(20, 30)
    val testRect2 = new Rectangle(50, 60)
    val testRect3 = new Rectangle(0, 0)
    val testRect4 = new Rectangle(90, 100)

    val testShape1 = new ColouredRectangles(20, 30, "blue")
    val testShape2 = new ColouredRectangles(50, 100, "red")
    val testShape3 = new ColouredRectangles(20, 30, "black")
    val testShape4 = new ColouredRectangles(50, 100, "red")
    val testShape5 = new ColouredRectangles(50, 100, "red")

    /** Tests for mutable collection */
    assert(set.contains(testRect1))
    assert(set.contains(testRect2))
    assert(!set.contains(testRect3))
    assert(!set.contains(testRect4))

    /** Tests for equivalence relation */
    assert(!testShape1.equals(testRect1))
    assert(!testShape1.equals(testShape3))
    assert(testShape2.equals(testShape4)) // symmetric
    assert(testShape4.equals(testShape2)) // symmetric
    assert(testShape3.equals(testShape3)) // reflexive
    assert(!testShape3.equals(null))
    assert(testShape2.equals(testShape4)) // transitivity
    assert(testShape4.equals(testShape5)) // transitivity
    assert(testShape2.equals(testShape5)) // transitivity
  }
}
```

I have made the fields of the Rectangle class vals, because if they were mutable, when we put the objects in a hashSet, when we change one of the fields of a particular object, then the original hash code would not correspond to the actual has code.