

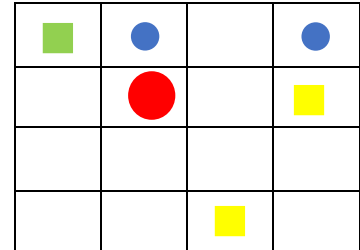
Semester 3 Workshop on Java

You need to develop a basic system that allows geometrical shapes to be contained, first in a row, and then in a board. Imagine something like this:

Row



Board



You are concerned with how to code the shapes and store them in a first a row and then a board, not how to display them. The code you are given does allow the shapes and board to be displayed, but that is extra to the point of this workshop and to allow you to literally visualize your solution. You are not required to develop the display part of the system but of course, if you wish to review the display code and experiment with it, it will help your Java skills.

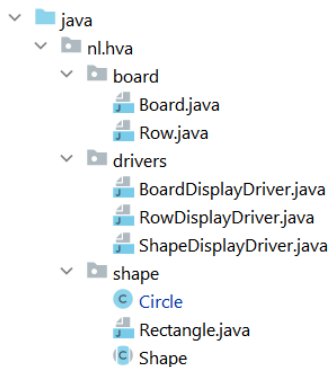
Getting started

You are given a starter project that has these classes:

- **In package (directory) shape**
- Shape – the parent class of Rectangle and Circle – **you must not change this class**
- Rectangle – you need to write the code for this class
- Circle – you need to write the code for this class
- HelloWorld – use this class to print the traditional “HelloWorld”
- ArrayExercise – use this class to experiment with arrays in Java.
- IteratorExercise – use this class to experiment with iterators.
- **In package board:**
- Row – you will develop this class to so that it can store shapes and be iterable.
- Board – you can work on this class to extend your skills and knowledge.
- **In package ui:**
- BoardDisplay – display a Board
- RowDisplay – display a Row
- ShapeDisplay – display a single Shape

- In package drivers:
- Classes to start displayers
- Main – use this class if you have the Board class working.
-
- TestClasses – uncomment the code if necessary and run these to test the code that you have written. Apart from uncommenting, **you should not change the code.**
- A diagram showing the classes and their relationships is on the next page.

A note about class names and packages: classes have a given name: eg Circle. However, each class has full name that depends on the directory that it is in. The Circle class that we are working with has a full name of `nl.hva.shape.Circle`. Look in the project and see that there is a directory structure (shown in the screen grab here) :



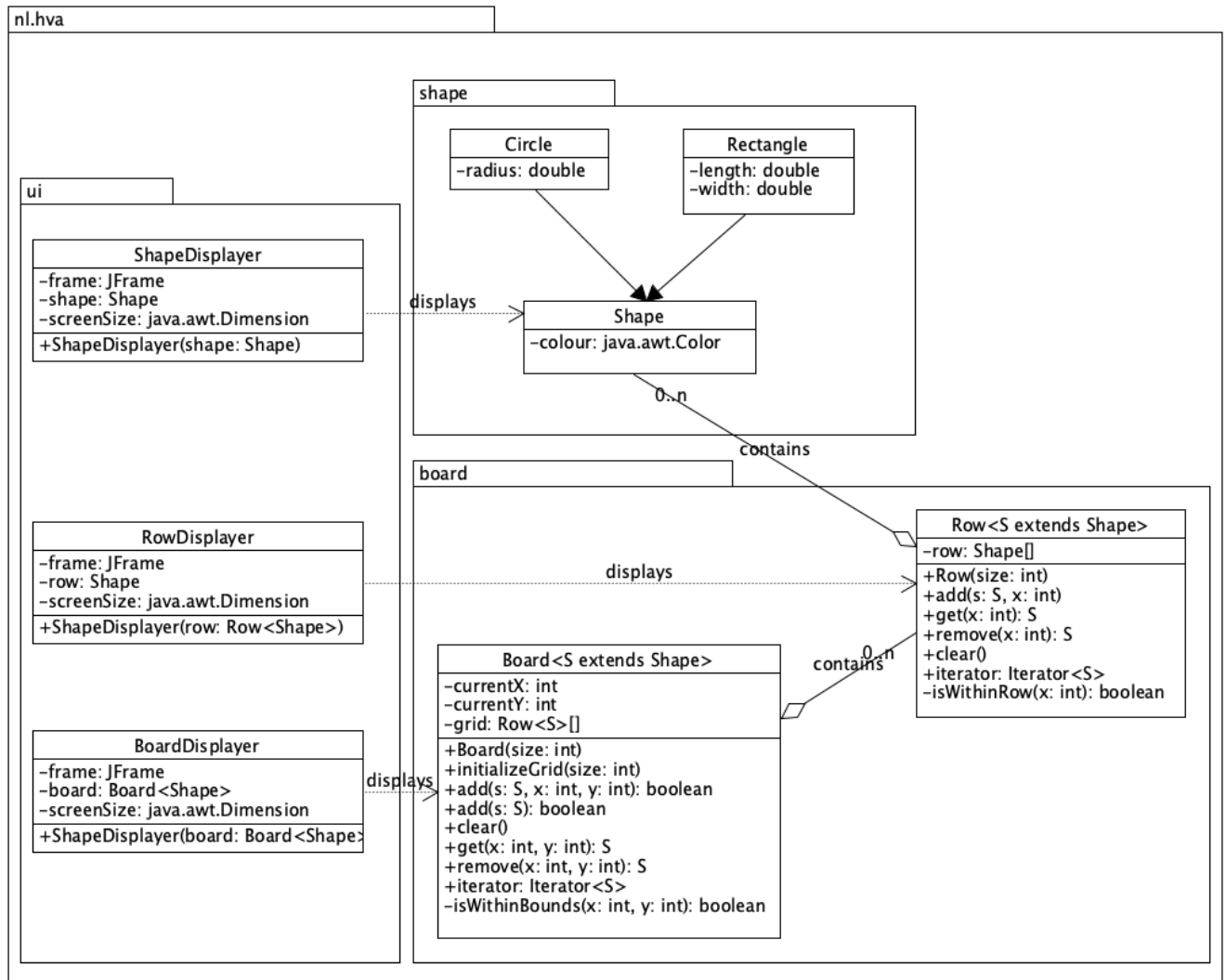
Our Shape class is also in the package `nl.hva.shape` so its full name is: `nl.hva.shape.Shape`. In the Java API, there is also a Shape class. Its full name is: `java.awt.Shape`. In each class file, the package that the class belongs to is declared at the very line (see the first line in the Circle class in starter project).

After the package declaration comes the import statements. Look in the source code for the Circle class. There is a line:

```
import java.awt.*;
```

This is because the class is going to use `java.awt.Color` and the system needs to import this class. Note that Circle uses Shape, but Shape does not get imported because Circle and Shape are in the same package.

Class diagram of the Shape system. Note the packages (directories) called `board`, `shape` and `ui`. Review the import statements in the classes to see these and other packages being imported.



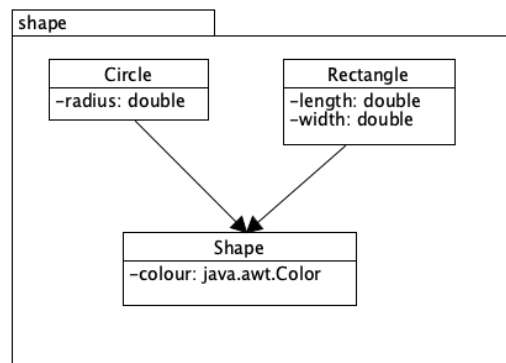
Step 0:

It is traditional to write a program that prints Hello World as the first step to learning a programming language. In Java we can use `System.out.println()` to print to the screen.

`System` and `out` are both objects and `println()` is a method call. It takes a string as a parameter. In the starter project there is a class called `HelloWorld`. Use the information above to write Hello World to the screen.

Step 1: Shape and the class hierarchy

Begin with the classes that represent shapes. These are `Shape`, `Circle` and `Rectangle` classes. Below is a diagram of the relationship between `Shape`, `Circle`, `Rectangle`.



Start by considering the `Shape` class. It is abstract. Find out what that means. See the Java syntax for declaring a class. Understand:

Fields, methods (abstract and concrete), visibility modifier (`public`, `private`), constructor, the distinction between a class and an object.

Notice that in the `Shape` class the `toString`, `equals`, `hashCode` methods have an annotation: `@Override`. They are overriding methods declared in the `Object` superclass. `Shape` is a sub-class of `Object`. All classes in Java are a sub-class of `Object`, either directly or indirectly.

Notice that the `toString` method calls the `getArea` method even though, in `Shape`, `getArea` is not yet defined (it is abstract).

You should not change the `Shape` class in any way.

Step 2: Circle and Rectangle

In the starter project most of the code has commented out. This means that it is not read by the compiler. Java has commenting as follows:

`//` this is a single line comment – it prevents a single line from being read by the compiler

`/*`

This is a multi-line or block comment. Anything between `/*` and `*/` is not read by the compiler.

`*/`

`/**`

```
* This is a Javadoc comment. Anything between /** and */ will be ignored by the compiler
* but used by the Javadoc tool to generate the documentation of classes.
*/
```

Find out more about Javadoc on the Oracle official website or other sites.

The class CircleTest is not commented out and so it is showing many errors. This will disappear as you start creating the Circle class. You might want to comment out parts of the CircleTest class so that it will compile as you begin your Circle class.

Create a Circle class.

The Circle class will be a child of Shape. It needs to be concrete (not abstract). Find out what this means and how it is done. The Circle class will inherit some properties from Shape. Find out what exactly this means.

This is a general definition of the Circle class:

Circle:

- subclass of Shape
- the colour will be inherited from Shape
- has a radius of type double
- the radius can be viewed by means of an accessor method
- the radius can be changed
- it can calculate its area
- equals: two circles are equal if their areas and colours are equal
- hashCode hint: use the superclass hashCode and

`Double.valueOf(radius).hashCode()` – the full line of code will be:

```
return Double.valueOf(radius).hashCode() +
super.hashCode();
```

As you are developing your Circle class the tests in the CircleTest class should be begin passing. When all tests pass your Circle class is working. You can use the ShapeDisplay to view an instance of a Circle. To do this go to the DisplayDriver class and follow the instructions for displaying a Circle.

Note that the ShapeDisplay uses Java graphical user interface (GUI) components to display its shapes. You need to close the display window to terminate the program.

Question: what would happen if a Circle was constructed with a negative value for a radius? What steps can be taken to prevent this? Is it necessary to prevent it?

Build the Rectangle class

A Rectangle is:

- A sub-class of Shape
- It conforms to the general mathematical definition of a rectangle (a quadrilateral with four right angles)
- Because it is a sub-class of Shape it must have a colour, and the area should be calculated and returned
- It can be constructed with two parameters (width and length) or one parameter. If it is one parameter, then the parameter will give the size of both sides – the constructed Rectangle will be square. The correct way of doing this is to call one constructor from the other (the one-parameter constructor calls the two-parameter constructor and passes its value as both length and width).
- The width and length can be changed
- The width and length can be obtained with accessor methods
- Two rectangles are equal if their sides are equal (not the areas). Obviously if the sides are equal, the areas will be equal, but the areas can be equal without the sides being equal.

Start by uncommenting some or all of the RectangleTest class. As you build your Rectangle class the rectangle tests should begin passing. Once you have built your Rectangle class you can use the DisplayDriver to display it.

Follow the instructions in the file.

Once you have both Circle and Rectangle you can start thinking about the how we might display the shapes, first in a Row and then in a Board. A row can be visualized as this:



Then, rows can be used to build a board (as at the beginning of the document). Somehow shapes need to be stored in a group. An array can be used to do this and will form the underlying storage for the Row class that will be built later.

Step 3: Arrays in Java

In the Java project for this workshop, study the file called ArrayExercise. In this file you will find information and some exercises on working with arrays.

Step 4: Interfaces and Iterator in Java

As well as having a mechanism for holding the shapes in a row, we also need to access and use the shapes in the row. A typical requirement is to print out all the shapes in a given row. How can this be done? It might be useful to provide a mechanism that allows this without having to know the inner workings of the Row class. This is a common problem for all storage data types. One way of solving this problem is to provide an iterator – a class

that has the job of iterating across all the items in a given data structure, in this case, the shapes in a row. Let's find out more about an iterator.

In the Java project for this workshop, study the file called `IteratorExercise`.

Step 5: Row

Now both concrete Shape classes are built, and you have some knowledge about arrays, interfaces and the Iterator, you can start thinking about how to display them in a row using a class called Row. A Row has a specification as follows:

Specification Part 1:

- It can hold only any kind of (subclass of Shape). This should be ensured with a generic type. The generic type declaration would be:
`<S extends Shape>`
- A row is created with a size (the number of Shape objects it can hold – also referred to as the number of elements it can hold). This is immutable (find out what that means).
- Like arrays in Java, the row is “zero based”. That means that the first position of the row will be position 0.
- A Row has a minimum size of 1. This should be enforced by the constructor. If a value of less than 1 is passed to the constructor, a row of size 1 should be created.
- Initially all the values in the Row are set to null.
 - A Shape can be added to a Row by adding it to a specific position.
 - if a Shape is added a value of true is returned. If it is not added (an illegal position is supplied) then false is returned.
 - Adding a Shape to a row position that already contains a Shape replaces the Shape already in the Row.
- A Shape can be retrieved from the Row by specifying the position of the Shape in the Row. If there is no Shape at the position supplied (there is no Shape at the position or the parameter is out of bounds), then null will be returned.
- A Shape can be removed from the Row by specifying the position of the Shape in the Row. If there is no Shape at the position supplied (there is no Shape at the position or the parameter is out of bounds), then null will be returned. When a Shape is removed then the Row element is set to null.
- A Row can be cleared of all elements. All shapes in the Row are set to null.

Specification Part 2:

- A row is iterable – it implements the Iterable interface (`java.util.Iterator`).

In the Java project, follow the TODOs in the code to build the Row class. At first, build the class without the Iterator (Specification Part 1) . After that is working add the Iterator (Specification Part 2). As you build the class the tests should start passing. As soon as you can add a Shape to the Row, you can display the Row using the ShapeDisplay.

Once you have a Row class built then think about how to design an Iterator. Iterator (java.util.Iterator) is an interface that defines 4 methods. Two of these methods need to be implemented to create a concrete Iterator. The other two methods are “default” – they have a default implementation and, for the purposes of this exercise, do not need to be considered.

The two methods that need to be implemented are:

`hasNext()` and `next()`.

See the TODOs and hints in the Row class about implementing these.

Run the RowIterator test to test your implementation.

This is the end of the required workshop. The next part is optional, if you want to challenge yourself and learn some more complex Java.....

Optional Workshop – Challenge 1

Step 6: Board

We have enough knowledge now to attempt coding the 2D board to display shapes. We are going to build on classes already developed (good practice).

A Board is an array of Rows.

	y				
Row 0					
Row 1		●			
Row 2					
Row 3					

The origin (y=0, x= 0) is at the top left.

	y				
x	●				

Shape added to position 0,0

The general specification of Board is as follows:

- A board is created with a size.
- The size cannot be changed.
- The board is square (the column and row sizes are the same)
- The board must have a minimum size of 1 and this should be enforced by the constructor.
- A Shape can be added to the board at a given position.
- A board can have shapes added sequentially. The method keeps track of the last added shape, and when the next shape is added in the next empty position until the board is full. This will only be reset when the board is cleared, not when a Shape is removed. Sequential adding will overwrite a shape already in the board.
- A successfully added Shape returns true
- An unsuccessful add (out of bounds) should not crash but just return false.
- Get and remove behave the same way as they do in Row (see specification for Row).
- A board can be cleared (all positions set to null).
- There is a method to return the row size
- There is a method to return the board size (row size * row size)

Iteration

The Board class should implement the `Iterable` interface. You should ignore the `spliterator()` and `forEach` methods. It is probably a good idea to create an `Iterator` class (implements `Iterator`) that is an inner class of `Board`. An instance of this class will be returned when the method `iterator()` (as specified by `Iterable`) is called. You will see that you need to maintain the state between calls to `hasNext()` and `next()`. Please view the `TODO` sections in the Board class.

Optional Workshop: Challenge 2

Add a new Shape class – child class of Shape. This could be something like triangle, pentagon, hexagon. This is actually quite straightforward – any descendant of Shape should work in the Row and Board. However, they will not be displayed. The ShapeDisplayer needs extending to be able to draw any new Shape. The ShapeDisplayer class could almost certainly be better designed to eliminate some code duplication, although it is not straightforward. If you do create a new shape, there are several places where ShapeDisplayer needs to be modified in order for it to work. It should be noted that the Shape and its sub-classes were not designed specifically to work with the ShapeDisplayer. The ShapeDisplayer was an afterthought.