Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

Assignment 4

Due: Monday 27th April @ Mid-night

The Shape Drawing System

This assignment concerns the implementation of a system for drawing shapes. A *shape expression* is either a rectangle or a combination of two or more shape expressions. A shape expression evaluates to produce a *shape*. Shape expressions can be combined in three different ways:

1. Two shape expressions can be **unioned** together. *This means the resulting shape contains all the points from both shapes.*
2. Two shape expressions can be **intersected**. *This means the resulting shape contains all the points that were in both shapes.*
3. The **difference** of two shape expressions can be computed. *This means the resulting shape contains all the points that were in the left shape, but not in the right shape.*

The syntax of shape expressions is given as follows, where $\texttt{S}_1$ and $\texttt{S}_2$ are arbitrary (i.e. recursively defined) shape expressions:

Shape Expression	Interpretation
$[\texttt{x}, \texttt{y}, \texttt{width}, \texttt{height}]$	Construct rectangle at given position with the given width and height.
$\texttt{S}_1 + \texttt{S}_2$	Construct a shape expression from the union of the two shape expressions $\texttt{S}_1$ and $\texttt{S}_2$
$\texttt{S}_1 \& \texttt{S}_2$	Construct a shape expression from the intersection of the two shape exprssions $\texttt{S}_1$ and $\texttt{S}_2$
$\texttt{S}_1 - \texttt{S}_2$	Construct a shape expression from the difference of the two shape expressions $\texttt{S}_1$ and $\texttt{S}_2$
$(\texttt{S}_1)$	A bracketed shape expression to specify evaluation order.
`V`	A variable identifier which returns the shape currently assigned to the given variable. Variable identifiers must begin with an alphabetic character (either lower or upper case), but may continue with a mixture of alphabetic or numeric characters.

The shape drawing system provides two commands for drawing shapes: `draw` or `fill`. The `draw` command accepts a shape expression and a colour, and draws the outline of the shape produced on the canvas. Similarly. the `fill` command accepts a shape expression and a colour, and draws a filled version of the shape on the canvas; that is every point in the shape is drawn with the given colour.

The shape drawing system executes a sequence of one or more commands that conform to the shape drawing language. Some examples of these "shape drawing" programs, and their output, are given in Figure 1. The syntax for the three statements in our shape drawing language is given as follows:

Command	Interpretation
`draw` S_1 C_1	Draw the outline of the shape produced from the shape expression S_1 onto the canvas in colour C_1.
`fill` S_1 C_1	Draw the filled shape produced by the shape expression S_1 onto the canvas in colour C_1.
$V = S_1$	Assign the shape produced from the shape expression S_1 to the variable determined by the variable identifier `V`.

Note: Colours are specified using a 6 digit hexadecimal number of the form #`rrggbb` (i.e. identical to that used for HTML). Here, `rr` corresponds to the red component, and takes a value between `0`...`ff`. Similarly, `gg` corresponds to the green component and so on.

Note: Shape expressions can be arbitrarily complex, just like arithmetic expressions in Java. See Figure 2 for some more examples.

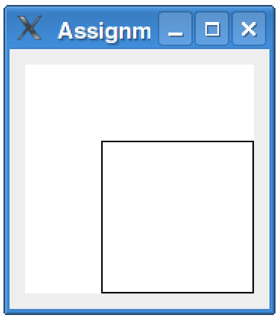
Part 1 — Basic Functionality

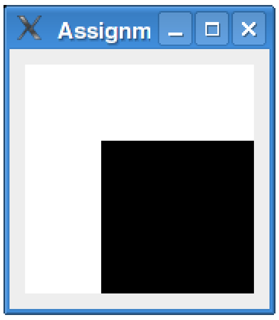
The aim here is to implement an `Interpreter` class that supports the creation, assignment and drawing/filling of shapes. At this stage, you should ignore the issue of composing shapes using *shape union*, *difference* and *intersection*.

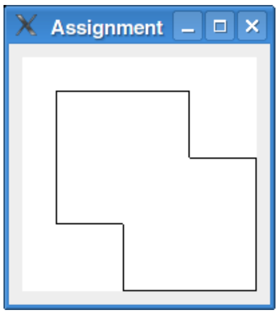
To get going, we suggest you follow these steps:

1. Download the file `shapes.jar` from the lecture schedule.
2. Create an appropriate Eclipse project and import the files from `shapes.jar`
3. Implement a class called `Rectangle` that has an `x` and `y` coordinate, as well as `width` and `height`. This should implement the `Shape` interface.
4. Implement method `fillShape` that accepts a `Shape`, a `Color` and a `Canvas`. This method should first determine the *bounding box* of the given `Shape` using method `boundingBox()`. Then, it should iterate the coordinates within that bounding box, whilst drawing those contained in the `Shape`. Test this method works by writing your own `main()` method and using `Canvas.show()`.
5. The `Interpreter` class should parse an input string and perform the required actions. To begin with, we recommend you focus on getting single line programs to work, such as these:
   - `fill [10,10,50,50] #000000`
   - `fill ([10,10,50,50]) #00ff00`

Drawing Program	**Shape Viewer**

`x = [50,50,100,100]`
`draw x #000000`
	

`v1 = [50,50,100,100]`
`fill v1 #000000`
	

`upper = [25,25,100,100]`
`lower = [75,75,100,100]`
`result = upper + lower`
`draw result #000000`
	

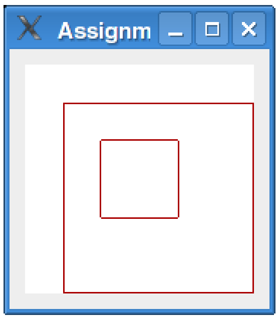
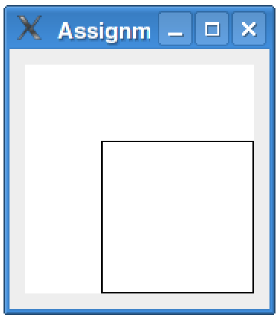
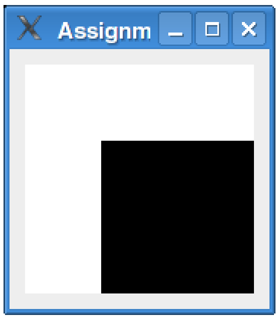
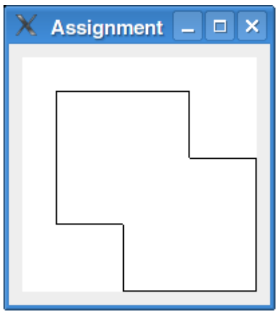
`bigBox = [25,25,125,125]`
`smallBox = [50,50,50,50]`
`bigBox = bigBox - smallBox`
`draw bigBox #aa0000`
	

Figure 1: Illustrating some simple shape drawing programs and their output (as seen using `Canvas.show()`). Observe that `+` and `-` correspond to *shape union* and *shape difference* respectively.

Drawing Program	**Shape Viewer**
`draw [50,50,100,100] #000000`	

`v1 = [50,50,100,100]`
`v2 = v1`
`fill v2 #000000`
	

`lower = [75,75,100,100]`
`result = [25,25,100,100] + lower`
`draw result #000000`
	

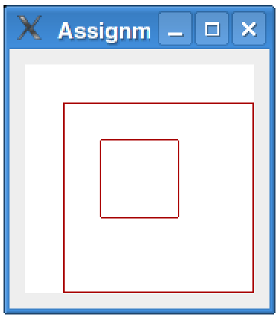
`box = [25,25,125,125]`
`box = (box + box) & box`
`draw (box - [50,50,50,50]) #aa0000`
	

Figure 2: Illustrating more shape drawing programs and their output (as seen using `Canvas.show()`). Observe that `+` and `-` correspond to *shape union* and *shape difference* respectively.

Implement methods `readShape` and `readColor` that perform the parsing of shape and colour expressions. You should find the class `Calculator` from Lab 1 helpful here. This parses commands and expressions in a similar way to what is required for the shape drawing system.

6. Extend the `Interpreter` to support drawing the outlines of shapes. The simplest way to do this is by using a horizontal and vertical *scanline* algorithm. This consists of two main loops (one for horizontal lines, the other for vertical lines). The first loop goes horizontally across each line of the shape. When it moves from being outside the `Shape` to inside, it draws a point in the specified colour. Likewise, when moving from inside to outside, it draws a point. The second loop is similar, except that it moves in a vertical direction.

7. Extend the `Interpreter` to maintain a `HashMap` that maps variable names to their current `Shape`. You will need to support assignment of `Shape`s to variables, and the occurrence of a variable within a `Shape` expression.

8. Finally, ensure that the JUnit tests `validFillTest` and `validDrawTest` now pass correctly.

HINT: You will find it helpful to construct a simple `main()` method that you can use to run shape programs and see their contents on screen.

Part 2 — Shape Composition

You should now extend the `Interpreter` class to support the `+`, `-` and `&` operators, which correspond to *shape union*, *shape difference* and *shape intersection* respectively. Having done this, all unit tests provided should now pass.

HINT: A sensible approach is to have one class for each of the different shape operators, with names such as `ShapeUnion`, etc.

HINT: An abstract class called `ShapeOperator` may be helpful in eliminating any duplicate code found in the classes for the different shape operators.

Submission

Your source files should be submitted electronically via the *online submission system*, linked from the course homepage. The minimal set of required files is:

```
assignment4/shapes/Canvas.java
assignment4/shapes/Shape.java
assignment4/shapes/Color.java
assignment4/shapes/Interpreter.java
assignment4/shapes/Rectangle.java
```

You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. See the export-to-jar tutorial linked from the course homepage for more on how to do this. *Note, the jar file does not need to be executable.*

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*

3. **All JUnit test files supplied for the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. This does not prohibit you from adding new tests, as you can still create additional JUnit test files. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in you getting zero marks for the assignment.

Assessment

This assignment will be marked as a letter grade (A+ ... E), based primarily on the following criteria:

- **Correctness of Part 1 (50%)** — does submission adhere to specification given for Part 1.
- **Correctness of Part 2 (40%)** — does submission adhere to specification given for Part 2.
- **Style (10%)** — does the submitted code follow the style guide and have appropriate comments (inc. Javadoc)

As indicated above, part of the assessment for the coding assignments in SWEN221 involves a qualitative mark for style, given by a tutor. Whilst this is worth only a small percentage of your final grade, it is worth considering that good programmers have good style.

The qualitative marks for style are given for the following points:

- **Division of Concepts into Classes**. This refers to how *coherent* your classes are. That is, whether a given class is responsible for single specific task (coherent), or for many unrelated tasks (incoherent). In particular, big classes with lots of functionality should be avoided.
- **Division of Work into Methods**. This refers to how well a given task is split across methods. That is, whether a given task is broken down into many small methods (good) or implemented as one large method (bad). The approach of dividing a task into multiple small methods is commonly referred to as *divide-and-conquer*.
- **Use of Naming**. This refers to the choice of names for the classes, fields, methods and variables in your program. Firstly, naming should be consistent and follow the recommended Java Coding Standards (see `http://g.oswego.edu/dl/html/javaCodingStd.html`). Secondly, names of items should be descriptive and reflect their purpose in the program.
- **JavaDoc Comments**. This refers to the use of JavaDoc comments on classes, fields and methods. We certainly expect all `public` and `protected` items to be properly documented. For example, when documenting a method, an appropriate description should be given, as well as for its parameters and return value. Good style also dictates that `private` items are documented as well.

- **Other Comments**. This refers to the use of commenting within a given method. Generally speaking, comments should be used to explain what is happening, rather than simply repeating what is evident from the source code.
- **Overall Consistency**. This refers to the consistent use of indentation and other conventions. Generally speaking, code must be properly indented and make consistent use of conventions for e.g. curly braces.

Finally, in addition to a mark, you should expect some written feedback highlighting the good and bad points of your solution.