Designing Methods

3.1 Designing Methods for Classes

Remember the `Rocket` class you defined in the beginning of the last lab? Find the class definition and copy it into a new file for this lab. Rename your `Posn` class to `Loc`, so it doesn't clash with later exercises.

Design the following methods for the `Rocket` class (not the union, yet):

1. the method `belowRadar` that determines whether the height of this `Rocket` is below the given radar limit.

2. the method `sameName` that tells whether this `Rocket` has the same `name` as a given `rocket`. *Hint*: you'll need to use the **`boolean`** `equals(String)` method defined in the `String` class to compare the rocket `name`s.

3. the method `moveUp` that produces a new `Rocket` with it's location moved up by 10 pixels. *Note*: this method also involves the design of a `Loc` method which returns a new `Loc` moved up by 10 pixels.

4. the method `nowLanding` that produces a new `Rocket` with its `landing` as **`true`**.

3.2 Designing Methods for Unions of Classes

Last lab we also designed an interface (`IRocket`) and classes (`SelfFlying`, `DualEngine`, and `Toy`) to represent different kinds of Rockets.

1. Design the method `nasaApproved` that determines whether this `IRocket` could be used in a future NASA mission. Obviously, only `Toy` rockets are not approved.

2. Design the method `moveUp` that returns a new `IRocket` with its location moved up by 10 pixels. `Toy` rockets don't have locations, so they can be returned unchanged.

3.3 Methods for Lists-of-Strings

Note: The following method defined for the class `String` will be useful:

```
/* How does this String compare to the given
 *   String lexicographically?
 *  - result  < 0 : if this String comes before
 *  - result == 0 : if this String is the same
 *  - result  > 0 : if this String comes after */
int compareTo(String that)
```

1. Design an interface and classes for representing a List of Strings. It's a Union right? Call the interface `ILoString` and the two classes `MtLoString` and `ConsLoString` (Guess which is which?).
2. Design the method `length` that determines the number of elements in this `ILoString`.
3. Design the method `isSorted` that determines whether or not this `ILoString` is sorted in lexicographical order (see the above `compareTo` method).

   Hint: You will need a helper method, and will probably need something like an accumulator to remember the previous list element.
4. Design the method `insert` that inserts a given `String` into this `ILoString` in the correct order, assuming this list is sorted.

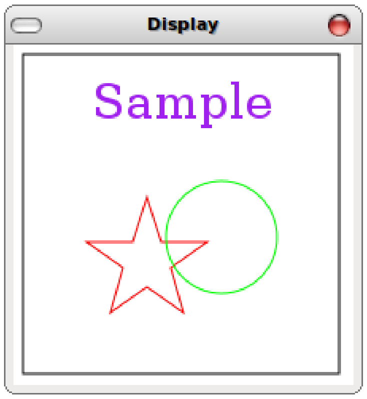
   Hint: focus on the examples and tests first, then work from the simple to the more complex cases.

Make sure you keep updating your *Template* as you go.

3.4 Using the JavaWorld Library

Now we get to draw some pretty pictures! Grab the JavaWorld Library (`JavaWorld-2.jar`) from the main Lab 3 page and save it in your `EclipseJars` directory. Add the Jar to your Project's build path (right-click on your Project, select
`Build Path > Add External Archives...`).

Download the `SceneExamples.java` file and save it in your project's `src` directory (drag and drop, or just save it and refresh your project). Setup a new run configuration for `FunJava` that has the new file (`src/SceneExamples.java`) as an argument. Run the file... it should popup a window like the following:

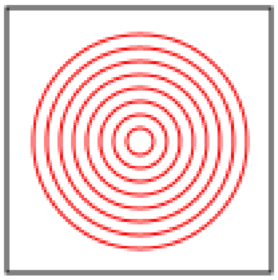


Read through and understand the code. Modify some of the parameters to see how everything works. After creating `Scenes` and `Images` in Racket last semester, the classes available should make sense. For more information review the tutorial (also used for Assignment 3) at:

```
http://www.ccs.neu.edu/course/cs2510/Assignments/
   Assignment3/WorldTutor/tutorial.html
```

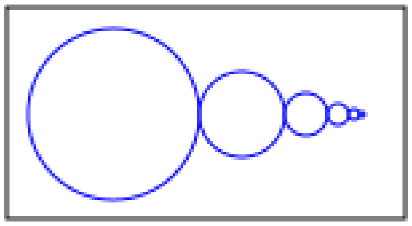
Remember your excursion into natural number recursion? Think really hard... then complete the following (fun) exercises, and try to match the images as closely as possible, they might look familiar to some of you...

1. Design the recursive method `bullsEye`, that takes an **`int`** `size`, and a `Scene`, and produces a `Scene` that contains a *"bulls-eye"* with concentric red circles. My circles are placed in a 100 x 100 `EmptyScene`, with a base-case when `size <= 0`.

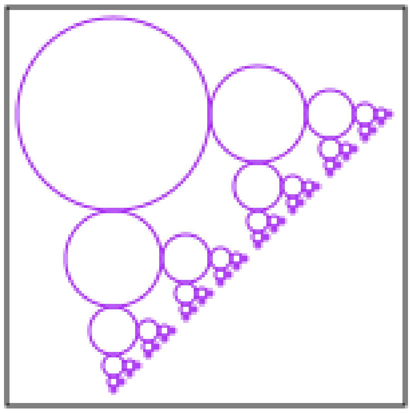
   Hints: You'll need to use `placeImage` and **`if`**, and the `size` becomes an accumulator (or, decumulator, depending on your base case.

2. Design the method `circles`, that takes two **`int`**s (`size` and `x`) and a `Scene`, and creates a `Scene` with connected circles, decreasing from left to right, in a 150 x 80 `EmptyScene`.

   Here `size` and `x` are accumulators... `size` gets cut in half, and `x` increases by `size*3/2` for each recursive call.

3. Design the method `circles2d`, which is like `circles`, but draws decreasing circles in both the `x` and `y` directions. You can accomplish this by recurring with an updated `x`, and composing the resulting `Scene` with a recursive call that updates `y`.

   The result looks something like your typical British crop circle.

Experiment with other interesting designs if you have extra time.

Save your work for later...