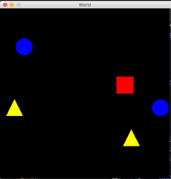
**Sprites**

This lab introduces you to the concept of ***abstract classes.*** An abstract class can contain a mixture of *abstract* (declared only) and concrete methods, along with instance variables and constructors. Abstract methods are quite useful when a parent class knows all sub-classes *should* have a method, but the implementation is not known, won't be used, or doesn't make sense for the parent.

Import the starter code and images into your project. The images must be in the "root" directory, the main project folder in BlueJ and Eclipse (NOT the *src* folder in Eclipse). The classes you will be using are summarized below:

|  |  |
| --- | --- |
| Display | The class that provides the GUI. You shouldn't need to make any changes to this class. |
| World | The World class represents the 2-dimensional stage where sprites appear. World has fields List<Sprite> sprites, a list of Sprite objects to draw, and the World's height and width. By default, a world is created 500 pixels wide by 500 pixels high.  X-coordinates increase from 0 at the left edge to 500 at the right edge, and Y-coordinates increase from 0 at the top edge to 500 at the bottom edge. |
| Sprite | This class represents a single visual entity in the world. A Sprite knows its position in the world (specifically the X-coordinate of its left edge and Y-coordinate of its top edge). A sprite also knows its width and height (in pixels), along with the file name of the image used to display it. |

You will be extending the Sprite class to make on-screen objects with various behaviors. Follow these steps:

1. Compile and run **World.java**; you should see an empty "world" pop up (a black rectangle).
2. **Sprite.java** is a base class that houses code common to multiple (future) types of sprites. In this simulation, you will never add a Sprite - you'll add more specialized *types* of sprites (e.g. a moving sprite or a bouncing sprite).

Add the abstract modifier to the Sprite class' header, and convert the existing step() method to an ***abstract method*** (a method that is *declared* only), that sub-classes will have to override:

public abstract void step(World world); //no body, declared only!

The step() method represents one step of the simulation / game (where a sprite does whatever it normally does), and must be supplied a reference to the World where the Sprite exists.

1. Make a new concrete class **StationarySprite.java** that extends Sprite.
   1. Add an appropriate constructor, given the variables inherited from Sprite that need to be initialized when making a StationarySprite object.
      1. Don't forget a super() call to pass these values to its super-class' constructor (the inherited variables should NOT be re-declared in the StationarySprite class).
   2. Override the step() method. Every step of the simulation, a stationary sprite does... nothing. This shouldn't take you very long. Nice!
   3. In the World class' two-parameter constructor, using the **"square.png"** file to represent this type of sprite, create a couple StationarySprite objects at random positions. Don't forget to add them to the sprites list.
      1. Run World's main() method – you should now see a couple red squares in the world.
2. Make a new concrete class **MobileSprite.java** that extends Sprite. The class will represent sprites that can move around, and therefore must have variables to store its *velocity*.
   1. Add two private double variables called vx and vy that represent the X- and Y-components of this sprites current velocity.
      1. For example, a vx of 2.5 and a vy of -0.5 represent a sprite that is moving to the right (increasing x) and slightly up (decreasing y). Recall that, with computer graphics, in general point (0, 0) represents the top-left corner of the screen. Bigger values of X will be further right, bigger values of Y will be further down.
   2. Add an appropriate constructor, and getters and setters (for vx and vy).
   3. Override the step() method inherited from Sprite in the following manner:
      1. A MobileSprite should bounce of the top, bottom, left, and right edges of the world. Copy / paste the following code into the step() method to accomplish this:

if (getLeft() < 0) //bounce off left edge

vx = Math.abs(vx);

if (getLeft() + getWidth() >= world.getWidth()) //bounce off right edge

vx = -Math.abs(vx);

if (getTop() < 0) //bounce off top edge

vy = Math.abs(vy);

if (getTop() + getHeight() >= world.getHeight()) //bounce off bottom edge

vy = -Math.abs(vy);

* + 1. Each step, this sprite needs to move based on its current velocity. After the above code, move this sprite the amount of its X- and Y-velocity. Note that the left and top variables are declared private in Sprite; even though this class is a sub-class of Sprite, these variables can't be accessed directly. Use the appropriate methods.
    2. Using the **"circle.png"** image, add a couple MobileSprite objects to the world at random positions, then run World's main() method to ensure everything is working.

1. Make a new concrete class **HeavySprite.java** that extends MobileSprite. A "heavy" sprite is very similar to a "mobile" sprite, except that it experiences gravity. Again override the step() method, this time utilizing a call to its super-class method, followed by the code to make it fall 0.1 pixel per step. This class' step() method should contain only two lines of code.
   1. Write the code to ensure this class works as expected. Use the **"triangle.png"** file to represent heavy sprites. Fix any mistakes you may have.

Given whatever time you have left, implement the following improvements. For those working ahead, implement as many / as few as you like (this step is quite fun and can become a decent platform for a game). Rather than simply implementing some number of unrelated behaviors one after the other, you are strongly encouraged to work toward a specific simulation or video game and implement the relevant behaviors below.

**Alive**  
Modify Sprite so that it keeps track of whether it is alive or dead.  Modify World's stepAll method to remove any dead sprites from the list.

**Lifespan**  
Create a new kind of sprite that dies at a specified time (when it reaches the edge of the screen, or after a specified number of steps).  (Completing *Alive* first will be required for this.)

**Generator**  
Create a new kind of sprite that generates other sprites (e.g. fires projectiles).  (Complete *Lifespan* if you want these projectiles to disappear eventually.)

**Overlap**  
Write a method in Sprite that takes in another Sprite and returns true if they overlap.  Specifically, the rectangular boundaries of two sprites A and B overlap if A's left is left of B's right, B's left is left of A's right, A's top is above B's bottom, and B's top is above A's bottom.  Alternatively, Java's Rectangle class has an instance method that will perform this calculation (given a second rectangle).  
  
**Killing**  
Create a new kind of sprite that kills or dies when it collides with another sprite.  (Complete *Alive* and *Overlap* first.)

**Landing**

Create a new kind of sprite that behaves exactly like a HeavySprite, except that it stops falling when it lands on another sprite.  (Hopefully it doesn't stop inside the other sprite.)  (It may help to have completed *Overlap* first, although you might not use that exact code here.)

**Contains**

Write a method in Sprite that takes in an *x* and *y* and determines if that point lies inside the boundary of the sprite.

**Clicking**

Modify World's mouseClicked method so that it creates a new sprite wherever you click, or (alternatively) interacts with (e.g. kills) whatever sprite you clicked on.  (To determine what sprite you clicked on, complete *Contains* first.)

**Keyboard**

Modify World's keyPressed method so that it interacts with a specific sprite (e.g. changes its motion) whenever you press keys with particular key codes.  (First, press keys and note what codes are printed.)  You may also choose to modify the keyReleased method.

**Jumping**

Change the velocity of a specific sprite whenever a certain key is pressed so that it appears to jump.  (Complete *Landing* and *Keyboard* first.)

**Bouncing**

Create a new kind of sprite that bounces off other sprites it comes in contact with, by changing direction (according to some rule you devise) without changing speed.

**The following code segments can help you convert between**vx**/**vy **and speed/direction.**

//given speed and direction in degrees, find vx and vy

double vx = speed \* Math.cos(Math.toRadians(direction));

double vy = speed \* Math.sin(Math.toRadians(direction));

//given vx and vy, find speed and direction

double speed = Math.sqrt(vx \* vx + vy \* vy); //pythagorean theorem

double direction = Math.toDegrees(Math.atan2(vy, vx)); //atan2 returns angle in radians

*Adapted from the Sprites lab by Dave Feinberg*

*https://sites.google.com/site/feinbergcompsci/home/hcs1/labs/spritelab*