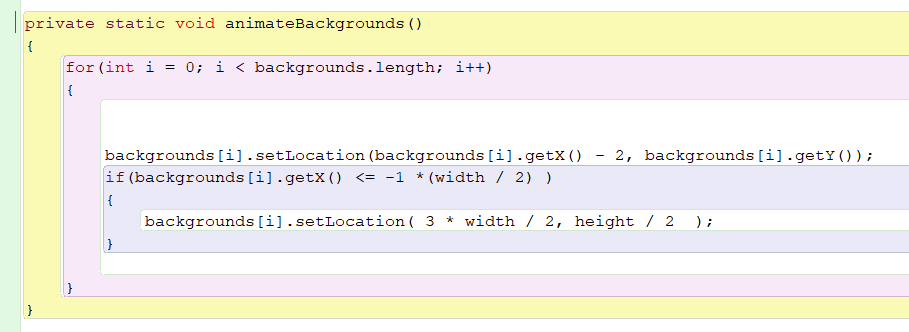
Project: Flight, Part One  
  
The goal of this project is to create a scrolling space-shooter game, in the style of old games like *R-Type* or *Gradius*.

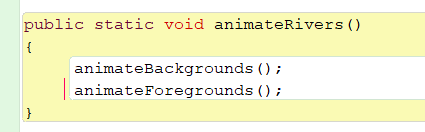


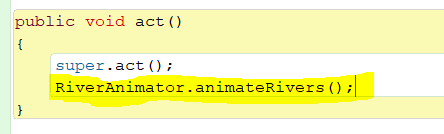
**Part I, the *static* keyword**

Before we begin this project, we need to learn a new vocabular word: **static**. A ***static*** method or variable is accessed directly from the class that contains it, instead of from an object of that class’s type. Let’s see what we mean by that.  
  
1. First go look at the code for the class *StarAnimator*. All its variables and methods are marked as *static*. They are accessed directly from the class *StarAnimator* instead of a *StarAnimator* object. What do we mean by that? Move on to step 2 to see an example.  
  
2. Now open the code for the class *StartScreen*. At the bottom of the constructor, you should see this line of code:  
  


If *addStars* was a non-static method, we would need to access it from an object, and the code would look more like this:  
  
  
  
Why would we want to do that? We’ll understand that better after we write and use some of our own static methods.

3. You’ll notice that the starting screen and the first level both have scrolling stars that run across the screen. These stars are animated using static methods inside the *StarAnimator* class. We want to do something similar, so that the river background in the second level is constantly scrolling.  
  
a. Add the below static method to the class *RiverAnimator*:  
  


b. Now add a similar static method called *animateForegrounds*(). This method should be different from *animateBackgrounds()* in two areas: 1.) it should move the elements of the array *foregrounds[]* instead of the array *backgrounds[]*, and 2.) it should move the foreground at a speed of 12 instead of a speed of 2.  
  
c. Now combine both our new static methods like this:  
  


d. And add the following call to *animateRivers* in *LevelTwo*’s *act()* method:  
  


Now check and make sure that *LevelTwo* animates correctly.

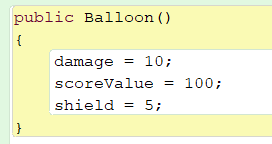
Why are all of these methods **static**? We can call them directly from the class, because the behavior of these methods will never change when an object’s **state** is changed. **State** refers to the values stored in an object’s variables. If one or more class variables changes values, then an object’s **state** is changed, and the way the object behaves might change as well. For example, two different *Enemy* objects might with different X and Y coordinates move to different locations when you call their *move()* methods, because they do not share the same state (they have different starting positions). If you change the value of one *Enemy* object’s *speed* variable, it will move farther than it did before it’s *speed* was changed.

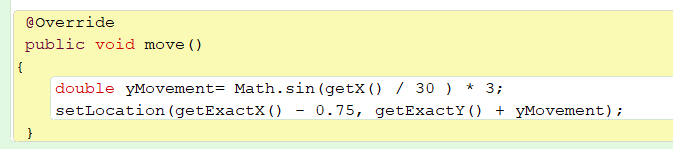
We want our *RiverAnimator* will always animate the river background in the same way. If we made a *RiverAnimator* object, the state of that object would not change the way those methods behave. There, we made those methods static, and we do not have to spend time creating new objects every time we might want to add an animated river background to one of our levels.

There is one thing to be aware of when using static methods: static methods cannot call non-static methods or access non-static variables. A static method can only call other static methods and access static variables.

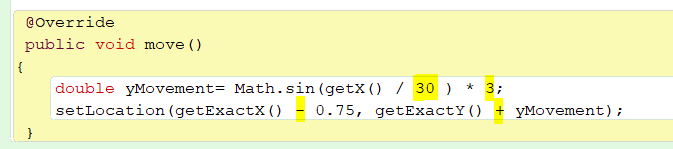
**Part II, The *Math* Class and Its *Static* Methods**

We’ve looked at lots of classes built into Java before, like *String*, *Color*, and *List*. This project, we’re going to look at a new class called *Math*. *Math* is different from those other classes, because it only contains **static** methods. We will never create a *Math* object. We will call its methods from the class directly.

1. We are going to use *Math* to help us create flight paths for our *Enemy* subclasses. First, create a new subclass of *Enemy* called *Balloon*, and set its image to the red hot air balloon.   
  
2. Our *Balloon* class won’t compile until we complete a couple more steps. First, add the following constructor to give values to the variables that *Balloon* inherits from *Enemy*.  
  
  
The *damage* int indicates how much the *Balloon* will take from the *Player*’s shield if they touch, the *shield* shows how many hits with a *Beam* it takes to be destroyed, and *scoreValue* shows how many points are added to the score when it is destroyed.  
  
3. Next, we’re going to make our *Balloon* move in a pattern called a **sine wave**. We will use *Math.sine()* to move the *Balloon* such that it’s y-coordinate is a function of the sine of its x-coordinate.

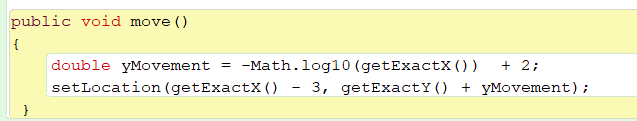


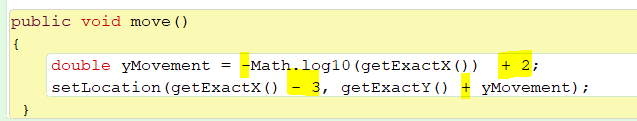
4. Add a *Balloon* to *LevelOne* and test that moves in a wave pattern. Also make sure that it takes 5 shots to destroy, adds 100 points to the score, and takes 10 shield from the *Player* if they touch.

5. Now let’s see how we can alter this sine wave. Experiment with changing the following values. Try bigger and smaller numbers, positive and negative. When you find a pattern that you like, keep it and move forward.  
  


6. Now let’s make another subclass of *Enemy* called *EyeCopter*. Give it the appropriate image, and add a constructor with whatever variable values you want.

7. Declare a new *int* variable called *stepCount*, and give *stepCount* a starting value of 0.  
  
8. We’re going to make our *EyeCopter* move in a logarithmic curve using *Math.log10*().



9. Add an *EyeCopter* to *LevelOne* and test it out.  
  
10. Just like we did with the *Balloon*, play with different values and see how it affects the *EyeCopter’s* movement.  
  


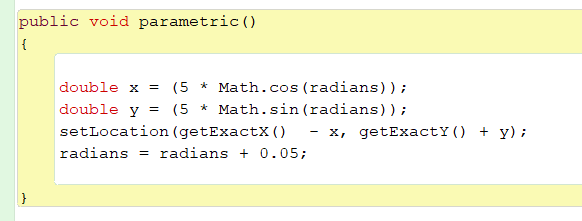
**Part III, More Advance Enemy Movement**

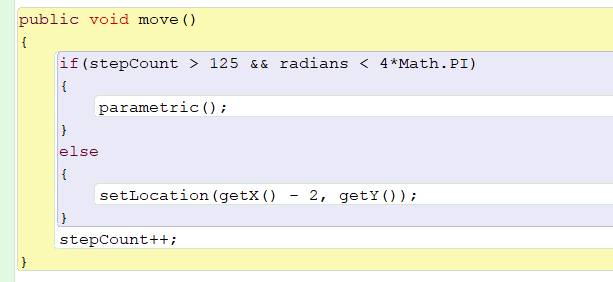
1. The next several enemies will feature more advance movement code. Create an *Enemy* subclass called *Saucer*, using the green flying saucer image, and give it an appropriate constructor.

2. For part of its movement, we’re going to make our *Saucer* fly in a circular motion, so we’ll need two new *int* variables, *stepCount* and *radians*. They should both start at 0.

3. The parametric equation for a circle looks like this:  
  

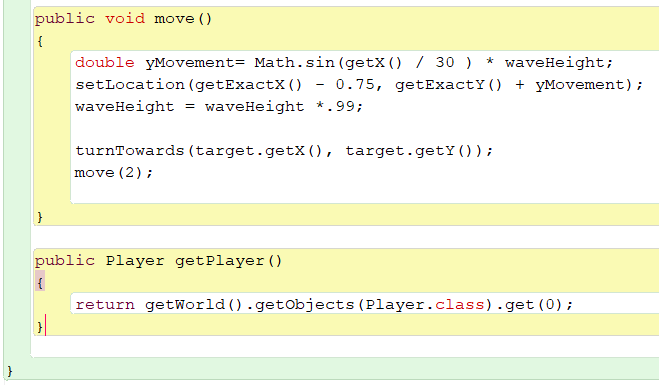

We’re going to recreate it in code with the following method:



4. Our *move* method will make use of this parametric method. For the first 125 it moves, the *Saucer* will move forward at a speed of 2. Then, it will fly in two circles (4 \* Math.PI radians, because one circle contains 2π radians), and finally move forward again:  
  


5. Test out your *Saucer*. Then, change the code so that it moves backwards instead of forwards, in a smaller circle, three times. You’ll have to figure that out yourself.

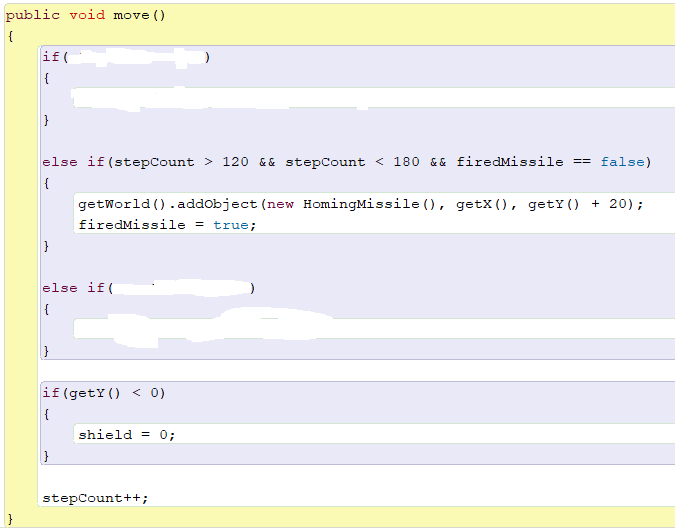
6. Next we’re going to add a *HomingMissile*. This *Enemy* requires more code than the others, because its movement is very sophisticated. A *HomingMissile* moves both in a line towards the player, and in a sine wave, but the sine wave portion of its movement gets smaller over time, so the movement becomes more and more linear as the *HomingMissile* moves towards the player. The code for *HomingMissile* should look like this:  
  

7. Test out your *HomingMissile*.

8. Now create an *Enemy* called *Plane*. The *Plane* should move in a straight line towards the *Player*. It will be similar to the *HomingMissile*, but without the diminishing sine wave movement.

9. Create a new subclass called *RedEnemy*. It needs an *int* variable named *stepCount* that starts at 0, and a *boolean* named *firedMissile* that starts at *false*.

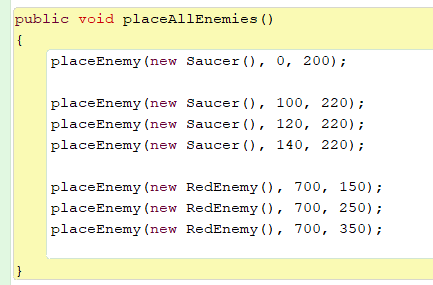
10. Our *RedEnemy*’s motion will have three steps: a.) for 100 frames it will move forward, b.) then it stay in place for 180 frames, firing a homing missile on the 20th frame of waiting, and c.) move upward until it reaches the top of the screen. Here is part of the *RedEnemy*’s move method. You will need to figure out the rest.  
  


11. Create a *PurpleEnemy*. It should have four different stages to its motion, but you can decide what happens in each stage.

**Part IV, Adding Enemies as the Levels Progress**

1. We don’t want to add all the E*nemies* directly into the *World* like we usually do in the constructor. Make sure you remove any code you currently have in either *LevelOne* or *LevelTwo* that adds any *Enemies* into the *World*. We want to have *Enemies* get added as the level progresses, to help create the illusion that the *Player* is flying to the right and the *Enemies* are flying to the left.   
  
Open the code for the classes *Triple* and *FlightLevel*. See what you can understand, and what is too confusing. Pay special attention to the methods *placeEnemy(), placeAllEnemies()* and *addlEnemies()* in *FlightLevel*.

2. We’re going to **override** the *placeAllEnemies()* method inside of *LevelOne()* (do you remember what **override** means?) Inside *placeAllEnemies()*, we are going to use the inherited method *placeEnemy()*. The first parameter that *placeEnemy()* accepts is the *Enemy* object to be added to the *World*. The second parameter represent how many frames passed the start of the level the *Enemy* gets added. The final parameter represents the *Enemy*’s y-coordinate. For instance, in the first call of *placeEnemy()*, we add a *Saucer* at y-coordinate 200 after 0 frames have passed (the level has just begun).



3. Play through *LevelOne* and see what the game is like now.

4. *LevelOne* and *LevelTwo* both last for 1500 frames. Afterwards, you automatically progress to the next level. Add more enemies to *LevelOne* so that the entire duration of 1500 frames contains *Enemies*. Then do the same thing in *LevelTwo*.

**Part V: If you’ve made it this far…**

… which seems unlikely, then I am super impressed. I haven’t finished writing the instructions yet. I should have them for you next class