Objectives

1. Create a small game using the Mayflower 2.0 framework.

Setup

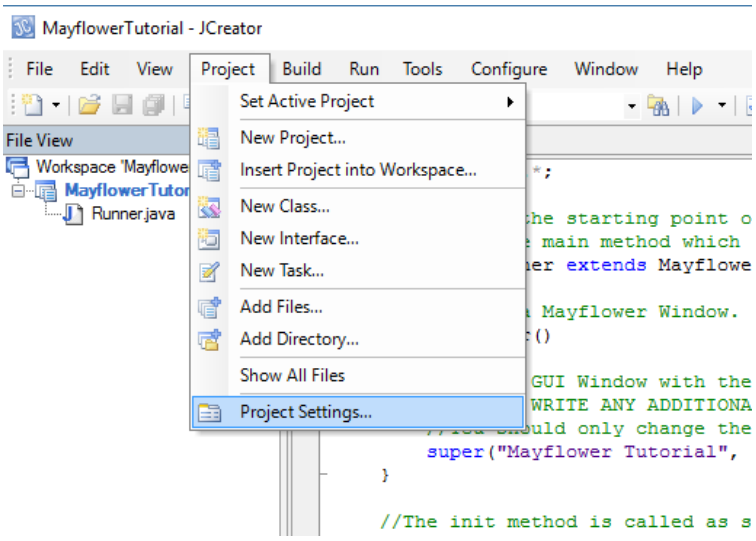
1. Create the folder `H:/APCS/MayflowerTutorial`
2. Download the starter code from eLearn
3. Download the `mayflower2.1.jar` file from eLearn
    a. Copy the `.jar` file into your `H:/APCS` folder.
4. Extract the files from the `.zip` file into your `MayflowerTutorial` folder

Part 0: Adding the Mayflower 2.0 Library

Open the JCreator project by double clicking on the `MayflowerTutorial.jcw` file or by using the **File→Open Workspace** menu option in JCreator.

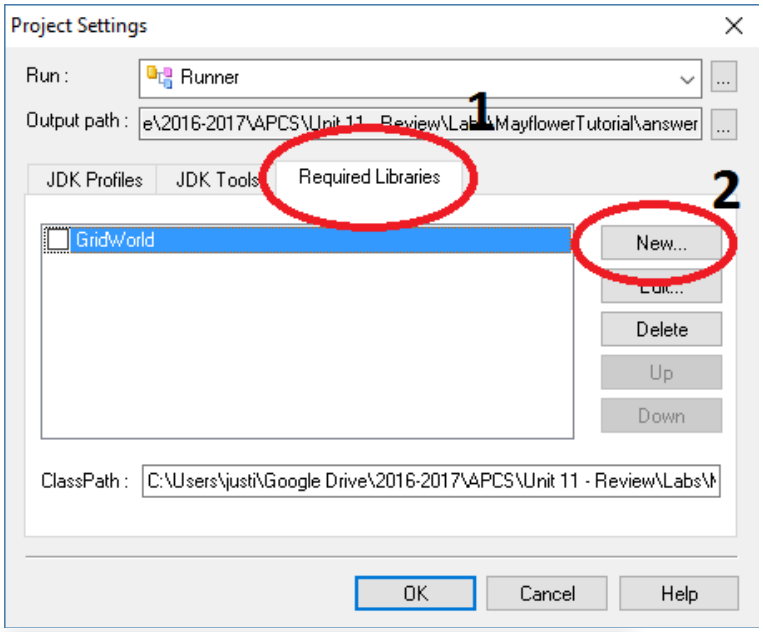
Before the project will compile, you must include the `Mayflower2.1.jar` library as a **Required Library**.

Choose the **Project→ Project Settings** menu option.



Click on the **Required Libraries** tab.

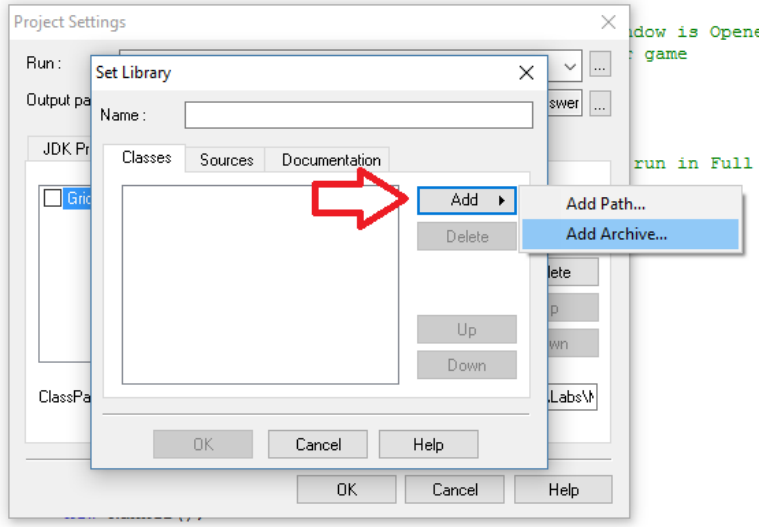
Then, click th **New** button



Click the **Add** button, then choose the **Add Archive** option.

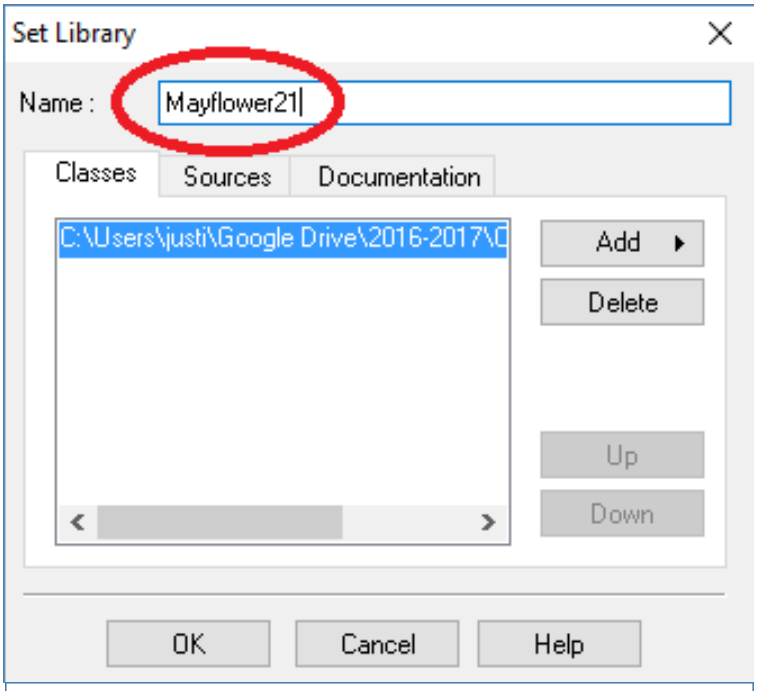
Browse to the location you downloaded the `Mayflower2.1.jar` file and choose that.

It *should* be in your `H:/APCS` folder.



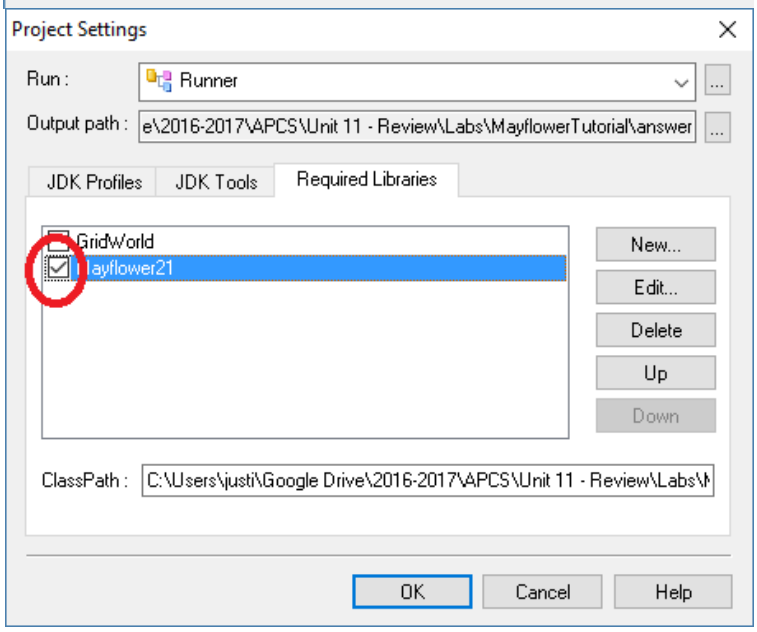
You *must* name the Library. I recommend naming it `Mayflower21`.

Then click the **OK** button.



You must **check the box** next to the library you just added!

Then click the **OK** button.



Now you can compile your Mayflower project.

Note: Mayflower projects require several `.dll` files which are included in the starter code. These `.dll` files must be in the same folder as the `.class` files that are created when you build your project.

Part 1: Mayflower 2.0: The Big Idea

Implementation

The Mayflower Library uses the Greenfoot API and wraps it around the Slick2D game engine. All of the classes and methods in Greenfoot are replicated in the Mayflower library so you can use the Greenfoot API Documentation to identify the classes and methods provided by the Mayflower library (but you should replace the word Greenfoot with Mayflower).

Because Mayflower uses the Slick2D game engine, it is able to provide some additional features, such as full screen mode.

Greenfoot Documentation	https://www.greenfoot.org/files/javadoc/
Slick2D Website	http://slick.ninjacave.com/

The Big Idea

A Mayflower program is a collection of `World` objects. Each world is composed of several `Actor` objects which can be controlled by the keyboard or mouse and interact with each other.

Only one world can be active at a time, and it manages the actors that live in it.

The Mayflower framework *updates* the active world 60 times per second. Each time the world is updated, its `act` method is called then the `act` method of all of its actors is called.

When writing a Mayflower application, you will create your own `SpecificWorld` and `SpecificActor` objects by creating classes that *extend* the `World` and `Actor` classes that are built-in to the Mayflower library. You will *override* the `act` method in these classes to specify how your actors will interact with each other, and how your worlds will look and feel.

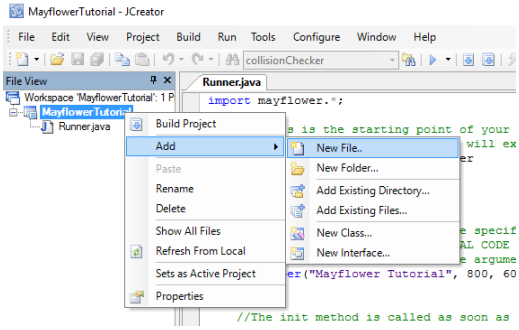
Part 2: Creating a World

Creating a new Class File

To create a world you must create a new class file.

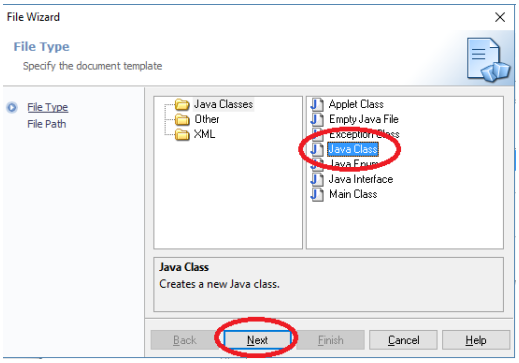
Right click on the Project name (`MayflowerTutorial`) in the **File View** pane.

Choose the **Add→New File** option.



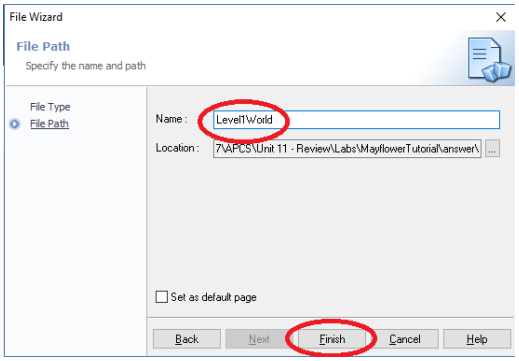
Choose **Java Class** from the right option pane.

Then click the **Next** button.

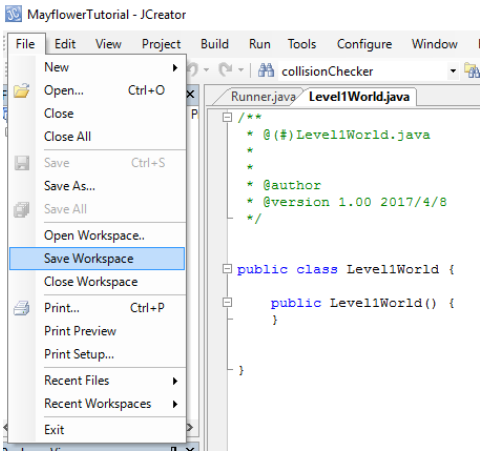


Name the class `Level1World`

Then click the **Finish** button.



Whenever you add a new file to your project remember to save your workspace by choosing **File→Save Workspace**



The first thing you must do whenever you create a new class in a Mayflower project is make sure it imports the Mayflower library by adding the following line of code to the top of the file:

```
import mayflower.*;
```

This will give you access to all of the classes in the Mayflower library.

Next, since you are creating a world, you must make it extend the World class by changing the class header.

FROM	TO
`public class Level1World`	`public class Level1World` **`extends World`**

The World class is abstract, so anything that extends it must implement the `act` method. Add the following method to your Level1World. This method doesn't do anything yet.

```
@Override
public void act()
{
}
```

At this point your `Level1World.java` file should look like this:

```
import mayflower.*;

public class Level1World extends World
{
  public Level1World()
  {
  }

  @Override
  public void act()
  {
  }
}
```

The two method you will write code in are the constructor and the `act` method. The type of code you write in each of these methods is important to distinguish.

The Constructor

Code that should only run *once* should be put in the constructor. This code will only execute at the moment the world is instantiated. This code should setup the world by setting the image that it will display and setting up actors in their starting positions.

The Act Method

Code that should run over and over again should be put in the act method. This code will execute every 60 seconds and is responsible for dynamically adding and removing actors from the world. If you want to "spawn" new actors into the world mid-game, this is where you would write that code.

Setting the World Image

You will find several images in the `/img` folder that was included in the starter code. Among them is the image `bg_space.png`. You will use this image as the background of your `Level1World`.

In the constructor add the line:

```
setBackground("img/bg_space.png");
```

This tells the world what image to use as its background.

Setting the Active World

Your `Level1World` is ready to go, but if you run your program it won't display the space picture! That is because you haven't told Mayflower that you want the `Space1World` to be active.

Look at the code in `Runner.java`. There are three methods in this class: The constructor, `init`, and `main`. You will not write any code in the constructor or the `main` method. These methods are setup to get the Mayflower framework running. Once the framework is running (the GUI window is open) the `init` method is called. This is where you will write code to setup your game.

Notice there are two `TODO` comments. You will `DO` those now.

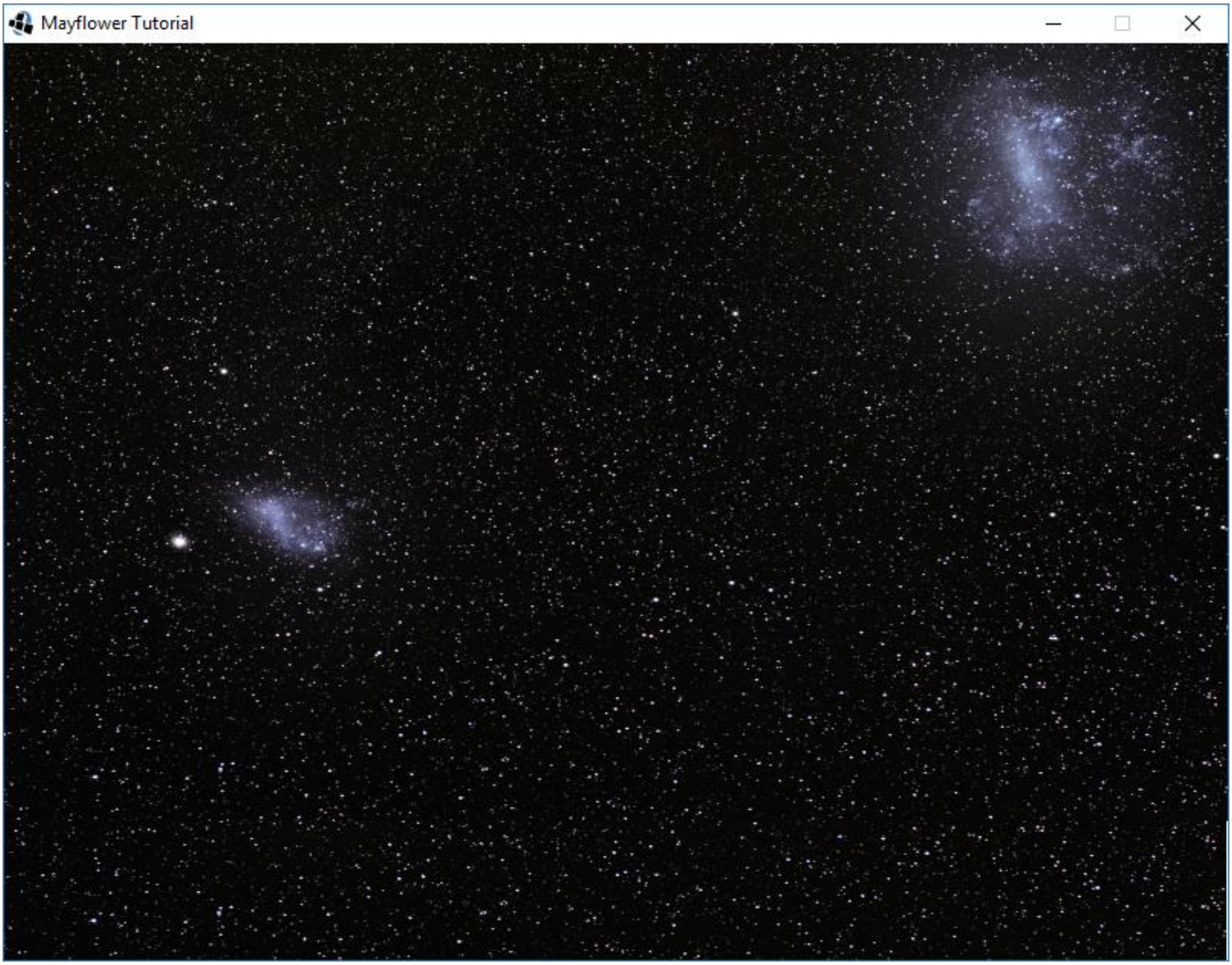
Uncomment the line of code under the first `TODO`, and change `NAME_OF_YOUR_WORLD()` to `Level1World()`

FROM	`//TODO: create a new World` `//World startingWorld = new NAME_OF_YOUR_WORLD();`
TO	`// create a new World` `World startingWorld = new Level1World();`

Then, uncomment the code under the second `TODO`.

FROM	`//TODO: load your world into the Mayflower Window` `//Mayflower.setWorld(startingWorld);`
TO	`// load your world into the Mayflower Window` `Mayflower.setWorld(startingWorld);`

Now you can build and run your project. You should see a window into space!



This window is 800x600 and the title bar says “Mayflower Tutorial”. These settings are all arguments to the `super` method in the constructor of `Runner.java`. You can adjust these arguments to change the size of the window or the name in the title bar.

```
super("Mayflower Tutorial", 800, 600);
```

Another option you have is to make the window open in full screen mode! In the `init` method, there is a call to the method `Mayflower.setFullScreen(false)`. If you change that argument to `true`, then your program will open in full screen mode! (Note: this only works if your width and height matches a supported resolution of your monitor so you must stick to common resolution like 800x600 and 1024x768)

When you are in full screen mode you can always return to windowed mode by pressing the ESC key. This will give you access to the [X] button so you can quit your game.

Part 3: Creating an Actor

Now that you have a world, you can create an `Actor` to add to it. Create a new class called `RobotActor` the same way your created your `Level1World` class above. (Remember to save your workspace, too!).

Add the `import mayflower.*` statement to the top of the class (you will have to do this in *every* class you create!)

Change the class header so that it *extends* the Actor class from the Mayflower library.

FROM	TO
`public class RobotActor`	`public class RobotActor` **`extends Actor`**

Like the `World` class, the `Actor` class is also abstract and any class that extends it must *override* the `act` method. Add the following method to your `RobotActor` class.

```
@Override
public void act()
{
}
```

At this point, your `RobotActor` class should look like this:

```
import mayflower.*;

public class RobotActor extends Actor
{
  public RobotActor()
  {
  }

  @Override
  public void act()
  {
  }
}
```

If you want your actor to appear in the world, you need to tell it what image to display. Add the following line to the constructor to tell it what image to use.

```
setImage("img/robot.png");
```

Now your actor is ready to be added to a world. The next bit of code will be added to the `Level1World` class.

In the constructor of the `Level1World` class, add the following code to instantiate a `RobotActor` object and then add it to the world at the (x, y) coordinate (400, 300). Note: the actors top-left corner will be at this coordinate.

```
RobotActor robot = new RobotActor();
addObject(robot, 400, 300);
```

Now you have an actor in your world. **Build and run your project to see what it looks like**.

Nothing much is happening yet, because your actor doesn't actually do anything…yet!

Part 4: Moving the Actor

The code that makes an Actor *do* things goes in its `act` method. You will add code to the `act` method of the `RobotActor` class that makes it move when the arrow keys on the keyboard are pressed.

Keyboard Methods

There are several methods you can use to identify how and what keys are being pressed on the keyboard. All of these methods are static methods in the Mayflower class.

`Mayflower.isKeyDown(KEYNAME)`	This method will return true if the specified key is currently being pressed.
`Mayflower.wasKeyDown(KEYNAME)`	This method will return true if the specified key *was* pressed the last time the `act` method was called. This is useful to check if the key was just pressed, or if it is being held down.
`Mayflower.isKeyPressed(KEYNAME)`	This method will return true if the specified key is currently being pressed and it was **not** pressed the last time the `act` method was called. ie, `isKeyDown` is `true` and `wasKeyDown` is `false`.

Key names are constants that are stored in the Keyboard class. If you want to check if the H key is being pressed you would use the following if statement:

```
if( Mayflower.isKeyPressed( Keyboard.KEY_H ) )
{
  //do something
}
```

Most keys can be checked by using `Keyboard.KEY_*` where `*` is the letter of the key you are checking. Here is a chart of other commonly used keys:

KEY_UP	KEY_SPACE	KEY_HOME	KEY_1	KEY_A
KEY_DOWN	KEY_TAB	KEY_END	KEY_2	KEY_B
KEY_LEFT	KEY_ENTER	KEY_DELETE	KEY_3	KEY_C
KEY_RIGHT	KEY_LSHIFT	KEY_INSERT	KEY_4	KEY_D

Actor Methods

There are several ways to move an actor in a world.

`setLocation(x, y)` This method teleports the actor to the specified (x, y) location.

`move(distance)` This method moves the actor the specified number of pixels in the direction it is facing.

By default, actors are facing east.

You can pass an `int` or a `double` to this method. If you pass a `double`, the actor will only visibly move a whole number of pixels, but it will remember the decimal amount.

If you call `move(0.5)`, the actor will not move. But if you call `move(0.5)` a second time, the actor will move 1 pixel.

`turn(degrees)` How many degrees, clock wise, should this actor turn. It *teleports* to the new angle.

`setRotation(angle)` This method sets the angle this actor is currently facing.

Degrees	Direction
0	East
90	South
180	West
270	North

`turnTowards(x, y)` This method will set the angle of the actor so that it is facing towards the point (x, y)

`turnTowards(Actor)` This method will set the angle of the actor so that it is facing towards the middle of the specified actor.

Directions

When using the `turn` and `setRotation` methods, you can use the helpful constants that are located in the `Direction` class to specify named directions.

```
Direction.NORTH  Direction.NORTHEAST  Direction.LEFT        Direction.AHEAD
Direction.SOUTH  Direction.NORTHWEST  Direction.HALF_LEFT   Direction.HALF_CIRCLE
Direction.EAST   Direction.SOUTHEAST  Direction.RIGHT       Direction.FULL_CIRCLE
Direction.WEST   Direction.SOUTHWEST  Direcetion.HALF_RIGHT
```

Cartesian Movement

You will add code to the `act` method of the `RobotActor` class that will allow you to move the robot up, down, left, and right using the arrow keys on the keyboard.

Add the following code to the act method:

```
if(Mayflower.isKeyDown(Keyboard.KEY_RIGHT))
{
  move(1);
}
```

Build and run your program. What happens when you press the right arrow key?

Experiments

1. Change the move method's argument from 1 to 10. How does that change the way the robot moves?
2. Change the `isKeyDown` method call to `isKeyPressed`. How does that change the way the robot moves?
3. Hold the right arrow key down until the robot reaches the edge of the screen. What happens?

Exercises

1. Change the isKeyPressed method call back to isKeyDown, and add 3 more if statements that check if the KEY_LEFT, KEY_UP, and KEY_DOWN keys are pressed.

   Build and run your code. What happens when you press the up, down, and left arrow keys?

2. Before each call to the move method (inside each if statement) add the following method call to change the direction the robot is facing.

   ```
   setRotation(Direction.NORTH);
   ```

   Be sure to change `Direction.NORTH` to the appropriate direction for each key (NORTH, SOUTH, EAST, WEST)

   Build and run your code. What happens when you press the up, down, left, and right arrow keys?

Cartesian Movement sans Rotation

You can make your robot move without spinning around by using the `setLocation` method in conjunction with the `getX` and `getY` methods.

Find the if statement you wrote that checks for the UP arrow to be clicked, and change its body like this:

FROM	TO

`setRotation(Direction.NORTH);`
`move(10);`
	`int x = getX();`
	`int y = getY();`
	`setLocation(x, y - 10);`

Build and run your code. What happens when you press the UP key? Try using the left, right, and down arrows to change the robot's rotation, then use the up key.

Exercises

1. Change the body of the other three if statements so that the robot's rotation never changes as you move it around using the arrow keys.

   You will have to change which argument you add or subtract 10 from, depending on the direction you want the robot to move.

   Remember, the top-left corner of the screen is (0, 0). As you go south, the y coordinate gets bigger. As you go east, the x coordinate gets bigger.

Experiments

1. What happens when you hold down the LEFT and UP arrows at the same time?
2. What happens why you hold down the UP and DOWN arrows at the same time?
3. If you used a chain of `else-if` statements instead of four separate `if` statements how would that change what happens when you hold down two arrow keys at the same time?

Rotational Movement

Another way you can make your robot move is by using the `turn` and `move` methods. The left and right keys will make the robot turn left or right, and the up and down arrows will make the robot move forward or backward.

Remove the code from the body of all four `if` statements. You will be completely rewriting them!

```
if(Mayflower.isKeyDown(Keyboard.KEY_RIGHT))
{
}

if(Mayflower.isKeyDown(Keyboard.KEY_LEFT))
{
}

if(Mayflower.isKeyDown(Keyboard.KEY_UP))
{
}

if(Mayflower.isKeyDown(Keyboard.KEY_DOWN))
{
}
```

The up and down keys should move the robot forward and backward using the move method. Use the following code to accomplish this.

Forward	Backward
`move( 10 );`	`move( -10 );`

The left and right keys should turn the robot some amount of degrees, lets say 5. Use the following code to accomplish this.

Right	Left
`turn( 5 );`	`turn( -5 );`

Build and run your code. Now your robot will move around differently than before.

Experiments

1. Press UP and LEFT at the same time. What happens?
2. Press UP and DOWN at the same time. What happens?
3. Press LEFT and RIGHT at the same time. What happens?

Part 5: Adding Other Actors

Now that you have full control over your robot, lets add another actor into the world.

Create the Actor

1. Create a new class called `CookieActor` (see above for step by step instructions)
2. Make sure this class imports the Mayflower Library
3. Change the class header to extend the `Actor` class
4. Override the `act` method
5. Set this actor's image to "img/cookie.png"

Add the Actor to the World

1. In `Level1World.java`, create three instances of the `CookieActor` class. Name them `cookie1`, `cookie2`, and `cookie3`.
2. Add the `CookieActor` objects to the world using the `addObject` method. Add them to (25, 50), (600, 300), and (300, 500)

Build and run your code. Witness the *Space Cookies*.

Experiment

1. What happens when you move your robot on top of a cookie? Which actor is *on top* of the other actor?

You can control the order in which the actors are drawn to the world. Actors that are drawn early will appear *below* actors that are drawn later.

In the constructor of `Level1World`, you can use the `setPaintOrder` method to tell the world what order to draw particular `Actor` classes.

Add the following code to the `Level1World` constructor (it can go anywhere in the constructor!)

```
setPaintOrder(CookieActor.class, RobotActor.class);
```

Build and run your code. Now which actor is on top?

The `setPaintOrder` method uses a special feature of Java that allows it to take an *unlimited* number of parameters. You can call it with any number of arguments!

Part 6: Collision Detection

Your robot is hungry. You will add code that allows your robot to pick up a cookie when it collides with it.

There are two ways you can implement collision detection between two actors. In your case, you will be checking if `RobotActor` collides with `CookieActor` by adding code to the `CookieActor`'s `act` method. But you could just as well add the code to the `RobotActor`'s `act` method.

The Actor class has several methods that can be used to deal for collisions between actors.

`List<E> getIntersectingObjects(Class<E>)`	This method returns a list of Actors that are intersecting `this` actor. You must specify a specific class of Actor that you are checking for as the argument. `List<RobotActor> collisions = getIntersectingObjects(RobotActor.class);`
`E getOneIntersectingObject(Class<E>)`	This method works similarly to the getIntersectingObjects method above, except it only returns the object at index 0.
`boolean intersects(Actor)`	This method returns `true` if `this` actor is intersecting the specified actor.
`boolean isTouching(Class<E>)`	This method returns `true` if `this` actor is touching any actors of the specified class. `if( isTouching( RobotActor.class ) )`
`void removeTouching(Class<E>)`	This method removes all objects of the specified class from the world *if* they are touching `this` actor. `removeTouching( RobotActor.class )`

You will be writing code in the `CookieActor` class that checks for when a `CookieActor` object is touching a `RobotActor` object. When that happens, you will remove the `CookieActor` from the world.

You can use the `World` class's `removeObject` method to remove an actor from the world. In order to call this from within an `Actor` class you will need to use the `Actor`'s `getWorld` method to get a reference to the world the actor is in.

If you pass `this` as the argument to the `removeObject` method, then the object that is executing the `act` method will be removed from the world. Since there are three `CookieActor` objects in the world, they will all be checking if they are touching the `RobotActor`. Whichever CookieActor detects a collision which remove itself from its world.

Add the following code to the `act` method of the `CookieActor` class.

```
if( isTouching( RobotActor.class ) )
{
      getWorld().removeObject(this);
}
```

Build and run your code. What happens when your robot touches a cookie?

Part 7: Keeping Score

Displaying Text

You have to jump through a few hoops to display text in Mayflower.

You can create a MayflowerImage object that contains text like this

```
MayflowerImage img = new MayflowerImage("Hello Robot", 24, Color.RED);
```

The three arguments are the text you want to display, the font size you want, and the color it should be. There are several colors to pick from in the Color class

BLACK	DARK_GRAY	LIGHT_GRAY	PINK
BLUE	GRAY	MEGENTA	RED
CYAN	GREEN	ORANGE	WHITE
YELLOW			

Or you can create your own color by mixing red, blue, and green like this:

```
Color crayon = new Color(123, 53, 0);
```

Where the arguments are red, blue, and green values between 0 and 255.

Unfortunately, you cannot add an Image directly to a world. You can only add Actors to a world, so you have to create an `Actor` object and set its image to the `MayflowerImage` that contains your text.

Exercise

1. Create a class called `ScoreLabel`
2. Import the Mayflower Library
3. Extend the `Actor` class
4. Override the `act` method
5. In the constructor, create a `MayflowerImage` object that says "Hello Robot" at size 24, and in the color `WHITE`
6. In the `Level1World` constructor, create an instance of the `ScoreLabel` object called `score`
7. Add `score` to the world at (0, 0) using the `addObject` method.

Build and run your code. You should see some text in the upper left hand corner.

Scoring Points

Your robot will score 1 point for each cookie it eats. It will have to *remember* how many points it has. Whenever you need to *remember* something, you must use an *instance variable.*

Exercises

1. Create a `private int` instance variable called `score` in the `RobotActor` class
2. Initialize the `score` instance variable to 0 in the constructor
3. Create a getter method for the `score` instance variable called `getScore`
4. Create a mutator method called `scorePoints(int amnt)` that increases the instance variable `score` by the value in the `amnt` parameter. (this method mutates/changes the value of an instance variable)

Build and run your code. Make sure you didn't introduce any errors in the code you just wrote. So far, nothing should have changed in how your game plays.

You have just added a few new abilities to your `RobotActor`. You can *get* the robot's current score and you can *increase* its score. You will use the `getScore` method to display the robot's score and the `scorePoints` method to increase its score every time it eats a cookie.

Since the code that detects when a cookie is eaten (when the `RobotActor` collides with a `CookieActor`) is in the `CookieActor` class, that is where you will write the code that calls the `scorePoints` method.

Inside the if-statement that checks if the CookieActor is touching the RobotActor, you will add some code that does the following:

1. Get a reference to *one* RobotActor that is touching the cookie
2. Increase that RobotActor's score by 1

You can use the getOneIntersectingObject method to get a reference to the RobotActor that collided with the CookieActor. Add the following code to the body of the if statement:

```
RobotActor robot = getOneIntersectingObject( RobotActor.class );
```

Now, the robot variable contains a reference to your Robot (the one that is eating the cookie). You can use that variable to call the robot's `scorePoints` method like this:

```
robot.scorePoints(1);
```

Now, the act method of the `CookieActor` should look like this:

```
@Override
public void act()
{
  if( isTouching( RobotActor.class ) )
  {
    RobotActor robot = getOneIntersectingObject( RobotActor.class );
    robot.scorePoints(1);
    getWorld().removeObject(this);
  }
}
```

Build and run your code. There still shouldn't be any difference in how your game plays. You can test if you are updating your score correctly by adding the following code to the `act` method of the `RobotActor`:

```
System.out.println(score);
```

This will spam the console with the robot's score (60 times per second!). It should just be showing 0 over and over again until you collide with a cookie, then it will start spewing out 1's!

You probably want to remove that line of debug code once you have confirmed that your score is updating correctly.

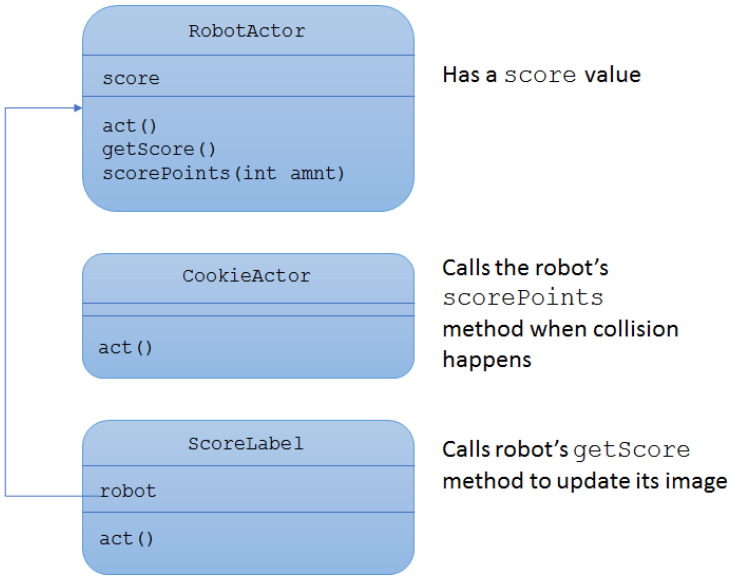
Displaying the Score

Now it is time to display how many cookie eating points your robot has scored.

To do this you will update the `ScoreLabel` class to keep a reference to the `RobotActor` whose score it should be displaying.

1. Create an `private RobotActor` instance variable called `robot` in the `ScoreLabel` class
2. Update the `ScoreLabel` constructor to take a `RobotActor` parameter called `r`
3. Initialize the `robot` instance variable to the value in the parameter `r`.

Try to build your code. You should get a compiler error.



You changed the method header of the constructor, so everywhere that you created an instance of a `ScoreLabel` object is broken now. The error messages will show you exactly what line numbers you need to go to so that you can update your code to pass the appropriate arguments to the constructor.

Exercise

Use the error message to find the line of code where you created a `ScoreLabel` object. Pass the constructor a reference to the `robot` (look around in that method for a `RobotActor` object that you can use as your argument)

Build your code, it should compile successfully.

Now you will make the `ScoreLabel` class update its image to reflect the robot's current score. Add the following code to the `act` method of the `ScoreLabel` class.

```
setImage(new MayflowerImage("Score " + robot.getScore(), 24, Color.WHITE));
```

Build and run your code. Now, when you collide with a cookie you should see your score update! However, the performance of your game is probably terrible now…

Creating new images isn't cheap. Your code is creating a new image 60 times every second! That is expensive (in a processor usage sort of way) It would be much more efficient to only create a new image if the robot's score has changed.

Exercises

1. Add a `private int` instance variable called `score` to the `ScoreLabel` class. This will keep track of the score that this label is currently displaying.
2. In the constructor, initialize the `score` instance variable to 0
3. In the `act` method, use the robot's `getScore` method to *get* the robots current score. If the robot's score is different (not equal to) than the `score` instance variable in `ScoreLabel`, then do the following:
    a. Change the `score` instance variable to be the same as the robot's current score
    b. Create a new `MayflowerImage` that displays the robot's current score
    c. Set the `ScoreLabel`'s image to be the `MayflowerImage` you just created

There is still a bit of a stutter when the score changes, but that's a bug in Mayflower. Someday it will get fixed!

Part 8: Game Over: Gotta Catch 'em All!

After you eat all of the cookies, something should happen. You will create a "You Win!" screen that will be displayed after your robot eats all of the cookies in space.

A Whole New World

Every different screen in your game is represented by a world. You will create a `YouWinWorld` and make that the active world once all of the cookies have been eaten.

Exercises

1. Create a class called `YouWinWorld`
2. Import the Mayflower library
3. Extend `World`
4. Override the `act` method
5. Set the background to "img/winner.png"

Transitioning Between Worlds

Only one world can be active at the same time. You can use the `Mayflower.setWorld` method to set the active world.

You will write code in the `Level1World` class that checks if there are any `CookieActors` left in the world. If there are no space cookies left, then it will change the active world to `YouWinWorld`.

Counting Actors

You can use the World method getActors to get a list of all the actors in the world that match a specified class. For example:

```
List<CookieActor> cookies = getObjects( CookieActor.class );
```

Will return a List of CookieActor objects that are in the world. If you want to answer the question "are there any cookies left?" You can check the size of the list; if the size is 0 then there are no cookies left!

Exercises

1. In the act method of the Level1World class, add code that will get a list of all CookieActor objects in the world.
    a. Note: you will have to `import java.util.List` to use the `List` class.
2. Write an if statement that will check if there are **no more cookies** left in the world
    a. If there are no more cookies, then create a new YouWinWorld object and use the Mayflower.setWorld method to change it to the active world like this:

```
YouWinWorld nextWorld = new YouWinWorld();
Mayflower.setWorld(nextWorld);
```

Build and run your code. What happens after you collect all three cookies?

Show the Final Score

It is nice to show the final score of a game on the game over screen. You can do this by adding a `ScoreLabel` object to the `YouWinWorld`. Remember that the `ScoreLabel` constructor requires a reference to the `RobotActor` object (specifically the instance that knows what the score is)

You should **not** create a new `RobotActor` object in the `YouWinWorld`. That object would have a score of 0! You want to get a reference to the original `RobotActor` object from `Level1World`.

Exercises

1. In `Level1World`, create a `private RobotActor` instance variable named `robot`
2. Change the line of code in the constructor that creates a `RobotActor` object so that it does not redefine the instance variable `robot` (don't declare the *type* of the variable `robot` in the constructor)

FROM	TO
`RobotActor robot = new RobotActor();`	`robot = new RobotActor();`

3. Change the header of the `YouWinWorld` constructor so that it takes a `RobotActor` parameter called `ra`

FROM	TO
`public YouWinWorld()`	`public YouWinWorld(RobotActor ra)`

4. In the act method of `Level1World`, change the line of code that creates a new `YouWinWorld` object so the constructor is passed the `robot` instance variable as an argument

FROM	TO
`… = new YouWinWorld();`	`… = new YouWinWorld(robot);`

5. In the constructor of `YouWinWorld`, create a `ScoreLabel` object and use `ra` as the argument
6. Add that `ScoreLabel` object to the world at (0, 0)

Build and run your code. Look at the upper left corner of the screen after you beat the game.

Of Mice and Buttons

This game is so fun, playing it once isn't enough! After you beat the game, it would be nice if there was an easy way to start playing it again! You will create a "Play Again" button.

In Mayflower, buttons are just Actor objects that respond to mouse clicks. The Mayflower class has a static method that will tell you if some Actor is being clicked.

```
boolean mouseClicked(Object obj)
```

You can call this method from inside an Actor object to check if it is being clicked like this:

```
Mayflower.mouseClicked(this)
```

Exercises

1. Create a class called `PlayAgainButton`
2. Import the Mayflower Library
3. Extend `Actor`
4. Override the `act` method
5. Set the image to "img/playagain.png"
6. Add a `PlayAgainButton` object to the `YouWinWorld` at (304, 400)

Now, add the following code to the act method of the PlayAgainButton. This code will check if the button is clicked, and if it is then it will create a new Level1World (which will create a new RobotActor with a score of 0) and set that as the active world.

```
if( Mayflower.mouseClicked(this) )
{
  Mayflower.setWorld( new Level1World() );
}
```

Build and run your code. Beat the game, then click on the Play Again button.

Part 9: Title Screen

When you first start your game, it plops you right into the thick of things. You will write code that starts the game on a Title Screen and waits for the player to click a button before starting the game.

Another World

1. Create a class called `TitleScreenWorld`
2. Import the Mayflower Library
3. Extend `World`
4. Override the `act` method
5. Set the background to "img/title.png"
6. Create a `StartGameButton` class
    a. Import Mayflower
    b. Extend `Actor`
    c. Override `act`
    d. Set image to "img/start.png"
    e. When this button is clicked, set the active world to a new `Level1World`
7. Add a `StartGameButton` to the `TitleScreenWorld` at (304, 400)
8. Change the line of code in `Runner.java` that sets the initial active world to `Level1World` so that it starts the game with `TitleScreenWorld`

FROM	`World startingWorld = new Level1World();`
TO	`World startingWorld = new TitleScreenWorld();`

Part 10: Fading Away

Now it's time to make the game more interesting. Cookies don't last forever in space. They eventually fade away into nothingness! You will add code to the `CookieActor` class that makes the cookies fade away. Once the cookie has become completely transparent it will remove itself from the world, never to be eaten by a robot.

The `MayflowerImage` class has several methods that are useful for manipulating the image.

`int getTransparency()`	This method returns the current transparency of the image, between 0 and 100. 0 being completely opaque, and 100 being completely transparent.
`void setTransparency(int)`	This method sets the transparency of the image. 0 being completely opaque, and 100 being completely transparent.
`void scale(width, height)`	This method resizes the image to the specified width and height.
`void scale(factor)`	This method resizes the image by the specified factor. A 1.0 factor will not change the image at all A 0.5 factor will shrink the image by 50% A 1.5 factor will grow the image by 50% A 2.0 factor will double the size of the image
`Color getColorAt(x, y)`	This method will return a Color object representing the color at the specified (x, y) coordinate.
`void setColorAt(x, y, Color)`	This method will change the color at the specified (x, y) coordinate to the specified color.
`int getRotation()`	This method will return the number of degrees this image has been rotated.
`void setRotation(degrees)`	This method will rotate the image.
`void mirrorHorizontaly()`	This method will flip the image over the vertical axis.
`void mirrorVertically()`	This method will flip the image over the horizontal axis.

You will use the `setTransparency` and `getTransparency` methods to make the cookies fade out and disappear. First you will need to *get* the actor's image so that you can call these `MayflowerImage` methods. You can do that using the `Actor` method `getImage`.

Exercises

1. Add an `else` statement to the if statement in the `act` method of the `CookieActor`.
2. Add code to the `else` statement that gets the actor's image and stores it into a `MayflowerImage` variable named `img`

```
MayflowerImage img = getImage();
```

3. Call the `getTransparency` method on the `img` variable and store the result in a variable called `alpha`

```
int alpha = img.getTransparency();
```

4. Call the `setTransparency` method on `img` and set the transparency to `alpha - 1`.

```
img.setTransparency(alpha - 1);
```

5. Write an if statement that checks if the current transparency is less than or equal to 0
    a. If it is, then remove this `CookieActor` from its world (look for an example of how to do this earlier in the `act` method)

Build and run your code. What happens?

If your robot is still hungry then you didn't win the game, did you?

Exercise

1. Create a new class called `YouLoseWorld`
2. Import Mayflower
3. Extend `World`
4. Override `act`
5. Set background to "img/loser.png"
6. Add a `PlayAgainButton` to (304, 400)
7. Add code to the if statement in the `act` method of the `Level1World` class that checks if there are no more cookies. Add a nested if statement that checks if the `robot` instance variable's score is less than or equal to 0.
    a. If it is, then set the active world to a new instance of `YouLoseWorld`
    b. else, set the active world to a new instance of `YouWinWorld`

Part 11: Randomizing Cookies

This game is a little repetitive. You can spice it up a bit by making the cookies spawn at different locations every time you play.

You will use the `Math.random` method to generate random (x, y) coordinates for each cookie.

Exercise

Fill in the blanks of the following code so that the x variable is a random number between 50 and 750 and the y variable is a random number between 50 and 550.

```
int x = (int)(Math.random() * __________ ) + __________;

int y = (int)(Math.random() * __________) + __________;
```

Because you are creating several cookies, it would be worth your while to *abstract* the cookie creation into a helper method so that later you don't have to worry about how a cookie is created or added to the world, you can just call the helper method and a cookie will be added like magic.

Exercise

1. Create a method `public void addCookie()` in the `Level1World` class
2. This method should generate 2 random `int` variables, `x` and `y`. The `x` variable should be an `int` between 50 and 750 and the `y` variable should be an `int` between 50 and 550.
3. Create a new `CookieActor` object.
4. Add that `CookieActor` object to the world at the randomized coordinate (x, y)
5. Replace the code in the constructor that creates and adds `CookieActors` to the world with three calls to the `addCookie` method.

FROM	TO

`CookieActor cookie1 = new CookieActor();`
`CookieActor cookie2 = new CookieActor();`
`CookieActor cookie3 = new CookieActor();`
 	
`addObject(cookie1, 25, 50);`
`addObject(cookie2, 600, 300);`
`addObject(cookie3, 300, 500);`
	`addCookie();`
	`addCookie();`
	`addCookie();`

Build and run your code. Notice that the cookies appear in different locations each time you play!

Part 12: Spawning Cookies

The Mayflower Library has a Timer class that you can use to trigger events after a specified number of milliseconds has passed (there are 1000 milliseconds in 1 second)

When you create a new Timer object you tell it how long it should go.

```
Timer t = new Timer(500);
```

This timer will "go off" after half a second. There are several useful methods in the Timer class.

`boolean isDone()`	This method returns `true` if the specified amount of time has passed.
`void reset()`	This method resets the timer so it starts counting again.
`long getTimeLeft()`	This method returns the number of milliseconds that are left before the timer "goes off"
`void set(int)`	This method changes the number of milliseconds the timer will wait before it "goes off"

Usually when you are using a `Timer` object you will create an instance variable for it and you will check if it has "gone off" using the `isDone` method in the `act` method of an `Actor` or `World`.

Inside that if statement you will write whatever code you want to execute after the timer rings. If you want that code to happen over and over again (every X milliseconds) then you should also call the `reset` method in the body of the if statement.

```
if( myTimer.isDone() )
{
  //do something

  myTimer.reset();
}
```

You will use a `Timer` object to spawn more cookies into the world every couple of seconds. This code will go in the `Level1World` class.

Exercises

1. Create a `private Timer` instance variable called `spawnTimer` in the Level1World class
2. Initialize the `spawnTimer` instance variable to a `new Timer(2000)`
3. In the `act` method, check if the timer is done (using the `isDone` method)
    a. If it is, add 3 cookies to the world at random locations (use your helper method!)
    b. *Reset* the `spawnTimer`.

Build and run your code. Can you catch 'em all?

Part 13: Bad Guys and AI

Your robot isn't the only hungry thing in space!

Exercises

1. Create a new class called `MonsterActor`
2. Import Mayflower
3. Extend `Actor`
4. Override act
5. Set image to "img/monster.png"
6. In the constructor of `Level1World`, create a `MonsterActor` object and add it to the world at (50, 50)
7. In the `act` method of `CookieActor` add another if statement that checks if the `CookieActor` is touching a `MonsterActor`.
    a. If it is, remove the cookie from its world
8. In the `act` method of `MonsterActor`…
    a. If the `MonsterActor` is touches a `RobotActor` then…
        i. Set the active world to a new `YouLoseWorld`

Build and run your code. What happens when you crash your robot into the monster?

You can make it even more interesting if the monster actively hunts cookies!

Add the following code to the `act` method of the `MonsterActor` class. Don't forget to `import java.util.List` at the beginning of the file!

```
List<CookieActor> cookies = getWorld().getObjects(CookieActor.class);
if(cookies.size() > 0)
{
  CookieActor cookie = cookies.get(0);
  turnTowards(cookie);
  move(2);
}
```

Build and run your code. What does the monster do now?

Part 14: Sound Effects

You can use the `Mayflower.playSound(filename)` method to play sound effects in your game.

Find some (short) `.wav` files and save them into the /snd folder of your project. Add calls to the `playSound` method in critical places (when a cookie is eaten, when the `YouWinWorld` is created, when the `YouLoseWorld` is created, etc…)

Part 15: More Levels

1. Create a `Level2World` class that extends `World`.
    a. Find a new, appropriate image to use as your background from google image search
    b. Use photoshop to resize the image to 800x600
    c. The constructor of this world should take a `RobotActor` as a parameter (it should **not** create a new `RobotActor`!)
    d. This world should go to the `YouWinWorld` when there are no more cookies left (just like the `Level1World`)
2. Create another "collectable" item (like the cookie)
    a. Use any appropriate image you want (resize it to around 64x64)
    b. Use photoshop to save it as a PNG with transparency
    c. This object should award 5 points to the robot when it gets eaten
3. This world should spawn 2 `CookieActors`, 2 `MonsterActors`, and 1 of your new objects
4. Every 200 milliseconds your world should spawn 3 new items, but it should be random how many of those items are cookies. (it could be 3 cookies and 0 newItems, or 2 cookies and 1 newItem, or 1 cookie and 2 newItems, etc...)
5. Change the code in `Level1World` that goes to the `YouWinWorld` so that it goes to the `Level2World` instead (and still passes the robot instance variable as an argument)