



GREENFOOT IN 4 HOURS

A Quick Start Guide



Version 2.0

October 8, 2015

For questions about this document contact:

Jeff Mellinger
San Diego State MESA Center
hoovermellinger@gmail.com

CALIFORNIA MESA SCHOOLS PROGRAM
2015

Contents

| | |
|---|----|
| Introduction | 2 |
| The Greenfoot Environment | 2 |
| Creating New Subclasses | 3 |
| Developing a Game Piece by Piece: MoveAndTurn | 5 |
| Move and Turn: A better way | 7 |
| Wrapping an Object | 7 |
| Disappear and Appear | 9 |
| Random Motion | 11 |
| Shooting | 12 |
| Scoreboard | 14 |
| Putting It All Together: A Complete Game | 17 |
| Conclusion: So, Now What? | 23 |
| Appendix A: Why learn Java and how does Greenfoot help? | 24 |
| Appendix B: Setting Up Greenfoot | 25 |
| Appendix C: Dealing with Errors | 25 |
| Appendix D: How Do I Know What Methods Are Available To Use? | 26 |

Introduction

This booklet is designed to quickly get you up and running with a basic game in Greenfoot. Rather than lengthy explanations of the language, the approach we will use is to simply present you with the code you need accompanied by annotations. You might not completely understand why a segment of code works the way it does but this understanding will come with further study. We just want to get you up and running with a basic game as a foundation for further exploration of Greenfoot.

If you don't know what the Java programming language is and how Greenfoot can be used to learn Java, you can find this in [Appendix A: Why Learn Java and How Does Greenfoot Help?](#). If you have not yet loaded Greenfoot and the Java JDK8 compiler onto your computer (sounds complicated but it's not), then go to [Appendix B: Setting up Greenfoot](#). If you do have Greenfoot and Java already loaded then you are ready to get started developing your game. If you know how to open a new folder, create classes, and bring objects into the world, then skip to the section [Developing a Game Piece by piece: MoveAndTurn](#).

The Greenfoot Environment

Once you complete downloading Greenfoot and the Java JDK8 compiler, a Greenfoot icon will appear on your desktop. When you click the icon you will see the screen shown in Figure 1:

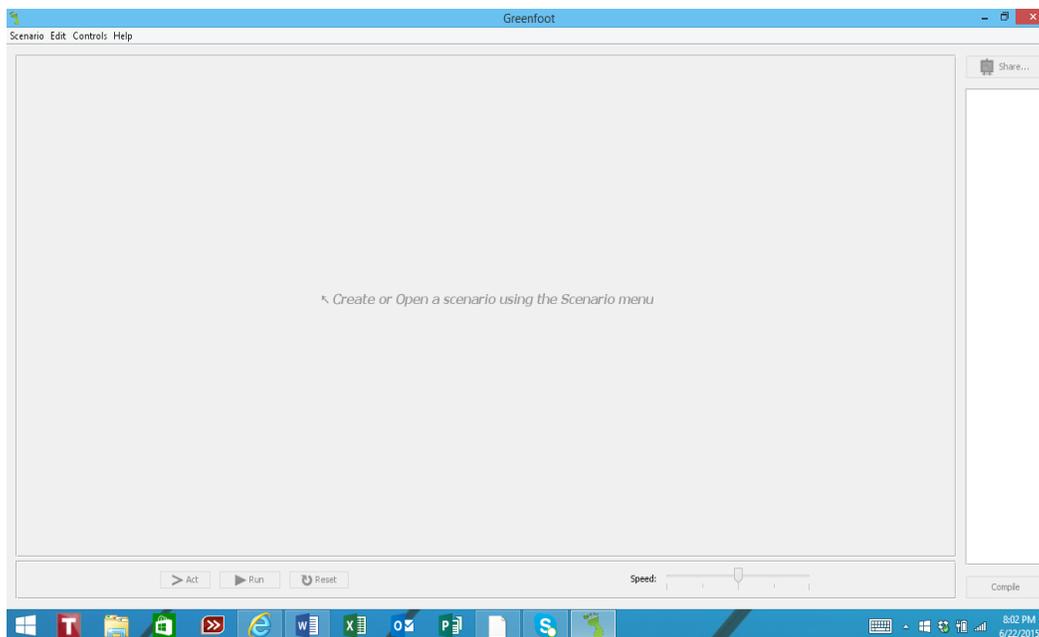


Figure 1. The Greenfoot starting screen

Click on **Scenario** and then **New**. You will see a New Scenario dialog box, as shown in Figure 2. In the Folder Name line, highlighted in blue, type the name of your new game then press the Create button. We will create a folder called **MoveAndTurn**.

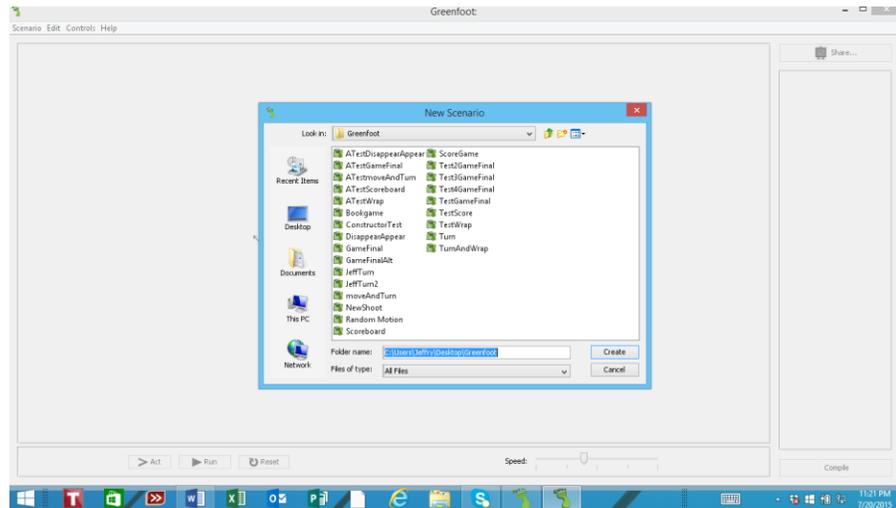


Figure 2. When you open a new folder you will give it a name

When you press create, the screen will briefly disappear then appear again with the name of your folder at the top. (Note: You do not need to save Greenfoot folders. They are automatically saved when you close the folder or exit Greenfoot. To actually close a Greenfoot folder you must go to **Scenario** and click **Close**).

Creating New Subclasses

On the right side of the screen you will see the class diagram with two rectangles; one labeled *World* and the other labeled *Actor*. The *World* class will simply serve as the background for your game though we will later add some code to create a scoreboard. The class *Actor* is where you will do all of your game programming. First, **right click on World** and a pulldown menu will appear. Select **New Subclass** and the New Class dialog box, as shown in Figure 3, will appear.

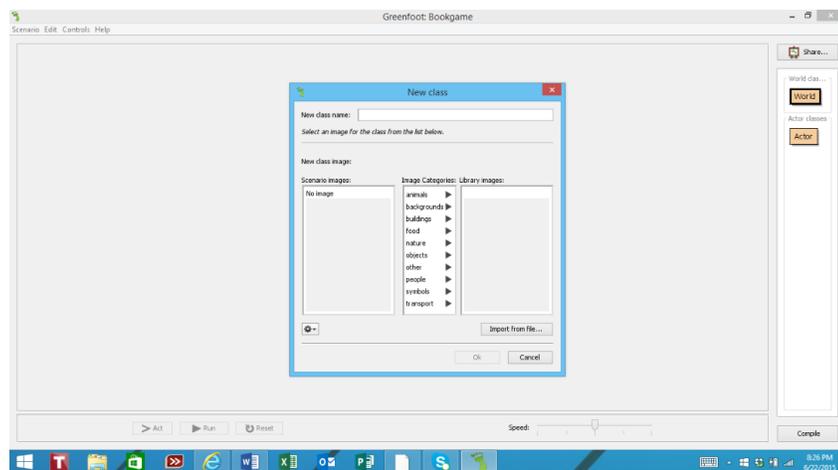


Figure 3. New Class dialog box with backgrounds and objects

For the World class choose from the **Backgrounds** in the Image Categories. A light colored background is desirable when you are new to Greenfoot so you can clearly see the movement of your actors. For this example, the background “bluerock” was

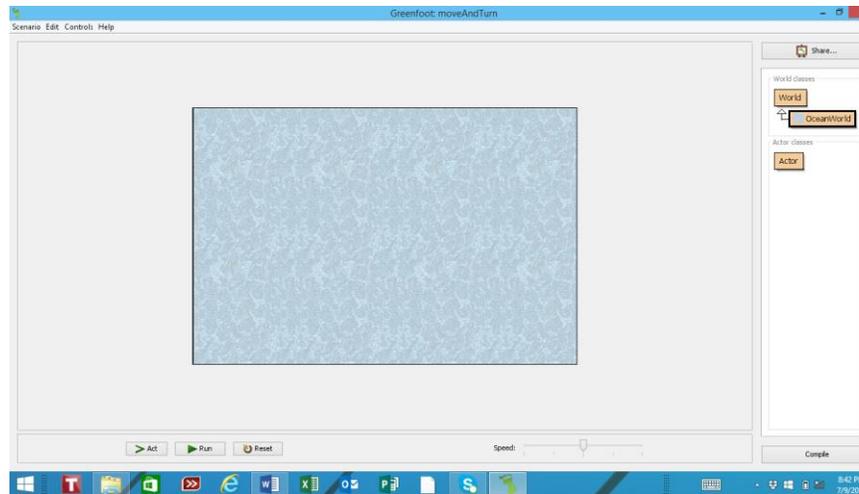


Figure 4. OceanWorld after pressing the compile button

chosen (we highly recommend that you use the same names and objects as this tutorial in order to avoid confusion). You must give the world a name on the line that reads “New class name:” Type **OceanWorld** then click **OK** and you can see that *OceanWorld* appears as a subclass of *World* in the class diagram. This means that *OceanWorld* will inherit all of the characteristics of *World* but you can then add more that are unique to it. You still don’t see anything in the game area. You must **press the Compile button** and *OceanWorld* will appear as shown in Figure 4. Anytime you see diagonal lines across any of the rectangles in the class diagram you must compile. For now, we are done dealing with *World* and will move on to the actors.

Next, **right click on Actor and left click on New Subclass**. Choose an object to be your first actor. We will use a boat in this example. **Left click on transport** in the Image Categories section. Find the object labeled **Boat02-4.png** and click on it. In the “New class name” line give it the name **Boat**, then click **OK**. You will see a new subclass of Actor called *Boat*. You again see diagonal lines so you must compile.

After you **compile, right click on the Boat rectangle** and you will see an option **New Boat()**. **Left click on New Boat()** and you will see a boat image appear that you can drag out into the world. If you press shift and continue to click you can make more boats appear. For now, just stick to one.

Developing a Game Piece by Piece: MoveAndTurn

Many games consist of about a half dozen basic elements. These include:

1. Moving and turning under the user's control
2. Wrapping around the world or bouncing off the edge of the world
3. Disappearing and Appearing
4. Random Motion
5. Shooting
6. Keeping Score

A company named Namco once made a phenomenal amount of money marketing a game called Pac-Man that contained nothing more than these functions. We will introduce these functions one by one, independent of each other. In this way you will create a library of game utilities that can be used in other games you create. When you **right click on the Boat class rectangle** you will see the option **Open Editor**. **Left click on Open editor**. When you first open the code editor of any actor you will see the following as shown in Table 1.

| Code | Comments |
|---|--|
| <pre>import greenfoot.*; /** * Write a description of class Boat here. * * @author (your name) * @version (a version number or a date) */ public class Boat extends Actor { /** * Act - do whatever the Boat wants to do. This method is called whenever * the 'Act' or 'Run' button gets pressed in the environment. */ public void act() { // Add your action code here. } }</pre> | <p>We are inside of the Boat class so we must import Greenfoot into this class.</p> <p>/* is how comments are made in Greenfoot. Comments are not a part of the program, they are just notes for you or others reading your code. These will be omitted in subsequent tables in this booklet though you will see them in your code editor.</p> <p>We declare that Boat is a subclass of Actor. Public means that it can be accessed by other actors in the game. A class contains the code for a specific type of object, such as a boat or a pirate.</p> <p>{ and } are called curly braces or just braces. Braces must always be used in pairs. They separate code into organized pieces.</p> <p>The act method is the "brain" of the class. Boat will not do anything that is not listed in the act method.</p> <p>You will replace the comment //Add your action code here with your methods for the Boat to perform.</p> |

Table 1. Code editor screen as is appears when opened in Actor or one of its subclasses

The first action we want our boat to perform is to be able to move and turn at our command. To do this we will write the code in the act method.

MoveAndTurn

The code to make our boat move about and turn on our command is shown below.

| Code | Comments |
|--------------------------------------|--|
| import greenfoot.*; | First, Greenfoot must be brought into every class in order to use predefined Greenfoot commands. |
| public class Boat extends Actor { | This line defines Boat as a subclass of Actor. This means that all of the methods that Actor has built into it, such as “turn” or “move” will be inherited by Boat. Public means that it can be accessed by other classes that we will define later such as Pirate and Bomb. |
| public void act() { | The method called “act” is the brain of the class. Only code that appears in act will run. |
| if(Greenfoot.isKeyDown("left")) { | This method uses an “if” conditional statement. The program looks to see if the left key is pressed down. The boat will turn left only if this is true. |
| turn(-5); | The “isKeyDown” method is a Greenfoot class command, not a command of the Actor or Boat classes, so the word Greenfoot must be typed before it. |
| move(1); | “Left” refers the left arrow key. In this case, “Left” is an argument (meaning input) of the isKeyDown method. |
| if(Greenfoot.isKeyDown("right")) { | What this says is that if the left key is pressed down, turn 5 degrees counterclockwise. The number “-5” is called an argument; it is an input that must be given to the method turn to determine how far the Boat should turn. |
| turn(5); | The move(1) command comes next but it is outside of the braces that contain the turn(-5) command. This means that the boat will move whether or not the left key is pressed. |
| move(1); | If the right key is pressed then the boat will turn 5 degrees clockwise. |
| } | Again, move(1) is outside of the braces that contain the turn(5) command. It will be executed whether or not the right key is pressed. So, if neither key is pressed, the boat will move straight. |

Table 2. Code and comments for the moveAndTurn method

When you have completed typing the code, press the compile button. There are two compile buttons; one in the code editor and one on the main screen. If you compile from the Code Editor and your code compiles properly you will see the message “**Class compiled-no syntax errors**”: Congratulations! You can now **right click on Boat and left click on newBoat(), then drag the Boat image out into the OceanWorld.** Now, press the “run” button to start the program and

turn the Boat using the right and left arrow keys. There is a good possibility, however, that your code did not compile properly. If this is true, see [Appendix C: Dealing with Errors](#). Knowing what methods are available for you to use can be a little confusing at first. You can read more about this in [Appendix D: How Do I Know What Methods Are Available to Use?](#)

MoveAndTurn: A better way

Now that you have gotten your program running you can learn a better way to structure your code. Writing all of your code in the act method makes adding and removing methods messy. It is better to write your code down below the act method and then “call” the method up to the act method when it is needed. As you add more and more methods to a class, you will see the advantage of organizing your code in this way. The code below, which performs the same moveAndTurn method, illustrates this.

Move and Turn (Preferred Method)

| Code | Comments |
|---|--|
| <pre>import greenfoot.*; public class Boat extends Actor { public void act() { moveAndTurn(); ← ----- } public void moveAndTurn() ← ----- { if(Greenfoot.isKeyDown("left")) { turn(-5); } move(1); if(Greenfoot.isKeyDown("right")) { turn(5); } move(1); } }</pre> | <p>In this version of the previous method, which performs exactly the same function as the original method, we gave the method the name “moveAndTurn” (the name of the folder has no bearing on the name of the methods inside).</p> <p>We now use a <i>call</i> to bring moveAndTurn up to the act method when we need it. The Java naming convention for methods is for the first letter of the first word to be lower case and the first letter of subsequent words to be uppercase—no spaces.</p> <p>“public void moveAndTurn()” is called the <i>signature line</i>. The method moveAndTurn is public so it can be accessed by other classes. It returns “void” which means it performs an action and does not give you back any information. Also, it has two parentheses with nothing inside which means that it does not require any arguments (inputs).</p> <p>This section is called the <i>body</i> of the moveAndTurn method. The same code used in the previous version is now down below the act method.</p> |

Table 3. Illustration of MoveAndTurn code using a call from the act method.

Wrapping an Object

When an object reaches the edge of your world there are two possibilities. First, it can bounce off of the edge according to some set of rules. Second, it can wrap around; that is, it goes out of one side and comes back in the other side. In this tutorial, we will use the second alternative.

Create a new folder called Wrap. In order to write the code for this routine, let’s look at the coordinate system used for the world. The default size of the world is 600 units wide by 400 units high. These dimensions can be changed in the code editor for OceanWorld but we have left them as is for this tutorial. The positive x-axis moves from left to right, just as it does in

conventional graphing. The y-axis, however, moves from top to bottom. Hence, the coordinates of the corners for the world are as shown in Figure 5. Now we can write the code to make the boat exit any point of the world and enter the opposite side while traveling in the original direction. First, create a

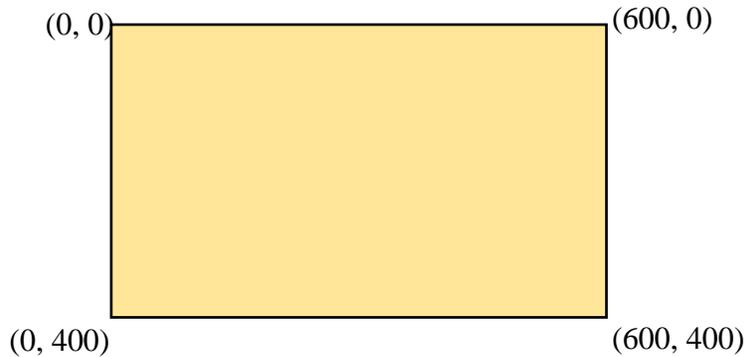


Figure 5. Coordinate system for a Greenfoot world

subclass of *World* called *OceanWorld* and a subclass of *Actors* called *Boat*. Below is the code.

Boat

| Code | Comments |
|--|--|
| <pre>import greenfoot.*; public class Boat extends Actor { public int x; public int y; public void act() { wrapAround(); } public void wrapAround() { x = getX(); y = getY(); if (x == 599) setLocation(0, y); if (y == 399) setLocation(x, 0); if (x == 0) setLocation(599, y); if (y == 0) setLocation(x, 399); move(1); } }</pre> | <p>Before we even enter the act method we must declare any variables we will use. If we don't, then Java will think these are just the letters x and y. A variable is a space in memory that will hold a number we want to use later. These variables are public, so they can be accessed by any other class. Int means they will hold an integer value data type. Other possible data types are Boolean and string.</p> <p>The only call the act method makes in this folder is wrapAround, the name of the method. The method name and the folder name do not need to be the same.</p> <p>Now, we are below the act method and must declare our new method. It is public so it can be accessed from anywhere in our game and it returns a void output meaning it does not give you back a number value, it just performs an action.</p> <p>The wrapAround method will use the getX() and getY() methods. These methods have already been defined in Greenfoot and can be accessed by any actor. These will get the x and y coordinates of the boat. Once they have done this, we need a place to put them. This is the purpose of the variables x and y. Variables are simply place holders for data.</p> <p>Now, wrapAround will ask a series of questions about the position of the boat:</p> <p>If the boat is at x-coordinate 599 (the far right edge of the world) . . . then move its location to x = 0 (the far left edge) and the current value of y. Boat will continue moving in the same direction.</p> <p>If the boat leaves the world at the y = 399 (the bottom of the world), then setLocation returns it to y = 0 (the top of the world) at the same location of x and moving in the same direction.</p> <p>If the boat leaves the world at x = 0 (the left side of the world), then setLocation returns it to x = 599 (the right side of the world) at the same location of y and moving in the same direction. The image in Figure 6 illustrates this.</p> <p>If the boat leaves the world at y = 0 (the top of the world), then setLocation returns it to y = 399 (the bottom of the world) at the same location of x and moving in the same direction. No matter which of the four options is chosen, the boat will move one unit.</p> |

Table 4. Code for wrapping an object

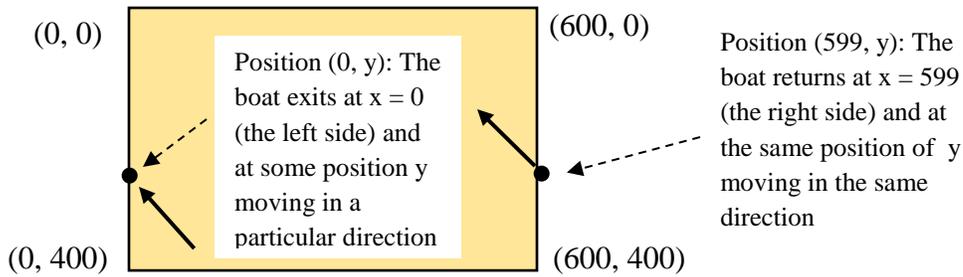


Figure 6. Object leaving the left side of the world and returning to the right side

A few points about the code in this method:

1. Line Spacing

A lot of spaces were made between lines of code in the previous folder. This was done to make the code and comments more readable. Greenfoot is picky about a lot of things but not this. You can leave as much space between lines as you would like.

2. The difference between “=” and “==”

The equals sign (=) assigns a value to a variable. For example, `x = 4` assigns the number 4 to the variable location `x`. The double equals sign (==) asks a true or false question but does not assign a value. For example, `x == 4` asks “does the value now in variable `x` equal 4?” If it does, then “true” will be returned. If it does not, then “false” will be returned. True or false returns are called *Boolean* returns.

3. How a computer counts

When the program counts to 400 it starts at 0 and ends at 399. Similarly, counting to 600 starts at 0 and ends at 599. This is why the numbers 399 and 599 are used for the bottom and right sides of the world, respectively.

The method you just wrote will cause the Boat to leave any side of the world and return to the other side. You cannot, however, use the arrow keys because the `moveAndTurn` code has not been included. **After you compile the Wrap folder drag a boat into the world and press Run.** The Boat will exit the right side and return on the left side. We will combine the `moveAndTurn` and `wrapAround` methods when we put our final game together.

Disappear and Appear

Many games involve making an object disappear and possibly reappear. **Create a new folder called Disappear.** You will **create a subclass of World called OceanWorld and an actor called Bomb** (use a ball from Objects in the New class dialog box) **and an additional actor called Pirate.** When the position of the bomb and pirate intersect the pirate will disappear then

reappear at a random location. The pirate object does not need any code modification. Copy the following code into the code editor for Bomb:

Bomb

| Code | Comments |
|--|--|
| <pre>import greenfoot.*; public class Bomb extends Actor { public int a; public int b; public void act() { move(1); pirateDisappearAppear(); } public void pirateDisappearAppear() { Actor pirate = getObjectAtOffset(0, 0, Pirate.class); if(pirate != null) { getWorld().removeObject(pirate); a = Greenfoot.getRandomNumber(599); b = Greenfoot.getRandomNumber(399); getWorld().addObject(pirate, a, b); } } }</pre> | <p>We will need two variables for this method. A variable is a holding place for data (numbers, letters, or words). These will be the x and y coordinates for a random position in the world. These variables are public so that they can be accessed from any of the objects in our game. We must also declare the type of data a and b will hold. Since they will hold an integer number we declare the data type "int."</p> <p>The act method makes the Bomb move and makes a call to pirateDisappearAppear.</p> <p>The method pirateDisappearAppear is public (can be used anywhere in the program) and void (performs an action, does not return a number). Empty parentheses mean that pirateDisappearAppear does not need an argument (input).</p> <p>The object that will disappear is the pirate. This asks if the Ball and Pirate are offset by (0, 0) meaning that they are in the same place. If this has happened then a "true" will be assigned to pirate.</p> <p>This line is read "if pirate does not equal nothing" (so, it must equal something—it's "true"). The "!=" symbol means "does not equal." This means that if pirate holds a "true" from the previous line then the following four lines will be carried out.</p> <p>We are in the Bomb class but the pirate is removed from the World, not from the Bomb. As a result, we must call in the world in order to remove an object (pirate). If all we want to do is make the pirate disappear, then we are done.</p> <p>If we wish to make the pirate reappear at a random location we get a random number between 0 and 599 for the x-coordinate and assign it to variable a. We then get another random number between 0 and 399 for the y-coordinate and assign it to variable b. getRandomNumber is a Greenfoot method, not a method of actor, so we must put Greenfoot before it. We use a and b for variables because we have already used x and y in an earlier folder which will later be combined with this one.</p> <p>We now go out and get the world because we are adding the Pirate to the world, not to the Bomb. The arguments (inputs) are the object (pirate), the x-coordinate (a), and the y-coordinate (b). This places the pirate at a random location in the world.</p> |

Table 5. Code for making an object disappear and appear

Drag a Pirate into the center of the world. Now, drag a Bomb immediately to the left of the Pirate. When you press run the Bomb will move to the right and intersect with the Pirate. The Pirate will disappear then reappear at a random location in the world.

Random Motion

Random motion can be used for an actor that you do not control but want to move freely around the board. **Create a new folder called Random. Create a world called OceanWorld and an actor called Pirate, using the skull symbol.** We will give the pirate a random motion, as follows:

Pirate

| Code | Comments |
|---|---|
| <pre>import greenfoot.*; public class Pirate extends Actor { public void act() { random(); } public void random() { if (Greenfoot.getRandomNumber(100) < 40) { turn(30); } else if (Greenfoot.getRandomNumber(100) > 60) { turn(-30); } { move(5); } } }</pre> | <p>We create a call in act to a method called random, which will be defined below.</p> <p>random can be accessed from any other class (public) and performs an action (void).</p> <p>Greenfoot contains a command for creating a random number. getRandomNumber(100) means that the number will be between 0 and 99, inclusive. If the number is less than 40, then the pirate will turn 30 degrees clockwise. . . .</p> <p>. . . or else, if the random number is greater than 60, then the pirate will turn 30 degrees counterclockwise</p> <p>. . . otherwise, if the random number is between 40 and 60, the pirate will not turn.</p> <p>No matter which of these three events occurs, the pirate will move 5 units.</p> |

Table 6. Code for creating random motion

Shooting

Shooting is a common feature of many games. Shooting consists of three actors: a shooter (the Boat), the “bullet” (the Bomb), and a target (the Pirate skull). **Create a new folder called Shooter. Create a world called OceanWorld. Create three Actor subclasses: Boat, Pirate(skull), and Bomb (represented by a ball).**

Boat

| Code | Comments |
|---|---|
| <pre>import greenfoot.*; public class Boat extends Actor { Bomb bomb = new Bomb(); public void act() { fireOnCommand(); } public void fireOnCommand() { if (Greenfoot.isKeyDown("f")) { World OceanWorld = getWorld(); OceanWorld.addObject(bomb, 0, 0); bomb.setLocation(getX(), getY()); bomb.setRotation(getRotation()); } } }</pre> | <p>The Boat creates the Bomb so a variable called bomb must be declared inside of the Boat class. This odd wording is how one class is called into another class. We won't try to explain it but you will see it in future classes.</p> <p>The only method in act is the fireOnCommand() call.</p> <p>fireOnCommand() is public to the program and performs an action (void).</p> <p>Using keyboard keys is a function of Greenfoot. Actors do not have this method available. As a result, Boat must call in Greenfoot in order to use the isKeyDown method. The argument (input) for this method in this case is the letter "f".</p> <p>Objects appear in the world, so the Boat must go out and bring in the world which is called OceanWorld.</p> <p>OceanWorld.addObject(bomb, 0, 0);</p> <p>Once we tell the Bomb to appear under the boat, we use getX() and getY() to determine the current position of the Boat. Then the starting location of the bomb is set at this location.</p> <p>This line means that getRotation (the direction the Boat is pointing) is used as the argument to set the direction (setRotation) of the Bomb. In this way, the Bomb will be pointed in the same direction as Boat and will, therefore, move in the direction the Boat is pointing.</p> |

Table 7. Code for the shooter (Boat) in the Shooter folder

Bomb

| Code | Comments |
|---|--|
| <pre>import greenfoot.*; public class Bomb extends Actor { public void act() { boom(); } public void boom() { move(3); Actor pirate = getOneIntersectingObject(Pirate.class); if(pirate != null) { World OceanWorld = getWorld(); OceanWorld.removeObject(pirate); OceanWorld.removeObject(this); } else if(isAtEdge()) { World OceanWorld = getWorld(); OceanWorld.removeObject(this); } } }</pre> | <p>Bomb acts as the “bullet” in the shooting process. It is created in the Boat actor. Once it is created, the code within the Bomb actor does three things:</p> <ol style="list-style-type: none"> 1. Makes Bomb move 2. When Bomb intersects with its target (Pirate) both the Bomb and the Pirate objects disappear. 3. Bomb disappears when it reaches the edge of the world. <p>The only item in the act method is the boom call.</p> <p>boom is public to the game and performs an action (void).</p> <p>The first thing the Bomb does once it is created is move. The only condition is that the Bomb must move faster than the Boat.</p> <p>This creates a variable pirate and places a “true” in this variable if the Bomb intersects any Pirate object.</p> <p>This line states “If pirate does not equal nothing” meaning that the Bomb is intersecting with a Pirate.</p> <p>Then the Bomb must go out and get the world to remove the Pirate and itself.</p> <p>The world removes the Pirate.</p> <p>The world also removes the Bomb (referred to as “this” since we are in the Bomb class).</p> <p>If there is no pirate at the Bomb’s current location then Bomb continues to move until it reaches the edge of the world. If this condition is true, then the program brings in OceanWorld and removes the Bomb at any edge of the world.</p> |

Table 8. Code for the Bomb in the Shooter folder

Pirate

| Code | Comments |
|---|---|
| <pre>import greenfoot.*; public class Pirate extends Actor { public void act() { } }</pre> | <p>The Pirate is the target and does not need any code.</p> |

Table 9. The Pirate class does not require any additional code

Once you successfully compile, **drag a Pirate into the world and a Boat immediately to the left of the Pirate (do not drag a bomb into the world)**. Again, you cannot steer the Boat so it must begin pointing at the Pirate. After you **press Run, press the “f” key and see the Bomb appear, move to the right, and then disappear along with the Pirate when they intersect**.

Scoreboard

Creating the scoreboard involves some advanced Greenfoot concepts. You will be given the code for creating the scoreboard but explanations will not be detailed. We will have the score increase by three points whenever a bomb strikes a pirate. **Create a**

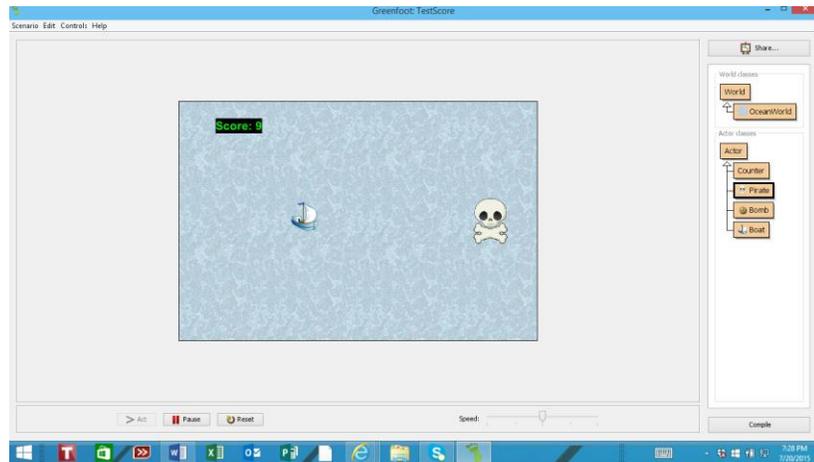


Figure 7. Class diagram for the Counter folder

new folder called Scoreboard, create OceanWorld as a subclass of World, then create a subclass of actor called Counter to serve as the scoreboard. Do not assign an object to Counter. In addition, create subclasses of Actor called Pirate, Boat, and Bomb.

OceanWorld

| Code | |
|---|---|
| <pre>import greenfoot.*; public class OceanWorld extends World { Counter counter = new Counter(); public OceanWorld() { super(600, 400, 1); Prepare(); } public Counter getCounter() { return counter; } private void Prepare() { addObject(counter, 100, 40); } }</pre> | <p>We are going to bring the counter into the world using this code.</p> <p>OceanWorld is 600 pixels wide and 400 pixels high. Motion is in increments of 1 pixel. Notice that OceanWorld does not contain an “act” method because it is not an Actor.</p> <p>The following code prepares the counter to be called by the Bomb and to function as directed in the Bomb class.</p> <p>The upper left corner of the scoreboard (counter) will be located at the coordinate (100, 40) near the upper left-hand side of OcenaWorld.</p> |

Table 10. Code for OceanWorld required to bring the scoreboard into the world

Boat

| Code | Comment |
|--|---|
| <pre>import greenfoot.*; public class Boat extends Actor { Bomb bomb = new Bomb(); public void act() { fireOnCommand(); } public void fireOnCommand() { if (Greenfoot.isKeyDown("f")) { World OceanWorld = getWorld(); OceanWorld.addObject(bomb, 0, 0); bomb.setLocation(getX(), getY()); bomb.setRotation(getRotation()); } } }</pre> | <p>This class contains the same code for Boat as found in the Shooter routine</p> |

Table 11. Code for the Boat to create the Bomb

Bomb

| Code | Comment |
|---|--|
| <pre>import greenfoot.*; public class Bomb extends Actor { public void act() { boom(); } public void boom() { move(3); Actor pirate = getOneIntersectingObject(Pirate.class); if(pirate != null) { World OceanWorld = getWorld(); OceanWorld.removeObject(pirate); OceanWorld oceanworld = (OceanWorld)OceanWorld; Counter counter = oceanworld.getCounter(); counter.addScore(); } oceanworld.removeObject(this); } }</pre> | <p>All of the code used in this folder is the same as the Bomb code used in the Shooter section with the addition of the scoreboard counter code shown here.</p> <p>In addition, the scoreboard must add the score before the Bomb is removed from the world shown here. If the bomb is removed before the scoreboard is changed then the entire program will abort.</p> |

Table 12. Code for the Bomb, which calls the counter to add to the score

Counter

| Code | Command |
|---|--|
| <pre>import greenfoot.*; import java.awt.Color;</pre> | <p>As with all classes, we import Greenfoot. In addition, we want to use a feature that Greenfoot does not contain, so we must import the color utility used for the scoreboard.</p> |
| <pre>public class Counter extends Actor {</pre> | <p>When we create the Counter class, we do not assign any object to it (such as a boat or a skull).</p> |
| <pre> int score = 0;</pre> | <p>We create an integer variable called score. We also set the beginning value of score at "0." This is called "initializing" a variable.</p> |
| <pre> public void act() { setImage(new GreenfootImage("Score: " + score, 24, Color.GREEN, Color.BLACK)); }</pre> | <p>We use the predefined Actor method "setImage" to create the image of the scoreboard. The argument for setImage is GreenfootImage. GreenfootImage, in turn, has five arguments: First, it will display the word "Score," then the numeric value held in the variable score will be displayed. The characters will be 24 pixels high. Finally, the text will be green and the background will be black. Remember, the position of the scoreboard was set in the OceanWorld class.</p> |
| <pre> public void addScore() { score = score +3; }</pre> | <p>addScore is the method we use to increase the score. Every time a bomb hits a pirate the score will increase by 3 points. Notice that the method addScore is not executed here in the Counter class, but rather, in the Bomb class.</p> |
| <pre> }</pre> | <p>This line reads, "the new score equals the old score plus three."</p> |

Table 13. Code for the Counter that creates the scoreboard

Pirate

| Code | Comments |
|---|---|
| <pre>import greenfoot.*; public class Pirate extends Actor { public void act() { } }</pre> | <p>The Pirate is the target and does not need any code.</p> |

Table 9. The Pirate class does not require any additional code

Now, place a **Pirate** into the world and a **Boat** immediately to its left. Press **Run** then press the **"f"** key. When the Bomb intersects the Pirate, it will cause the score to increase by three.

Putting It All Together: A Complete Game

We can now combine all of the folders we have created into an integrated game. **Create a new folder for your game called GameFinal. Create a subclass of World called OceanWorld and use bluerock as your background. Create the following subclasses of Actor: Boat, Pirate (skull), Anchor, Bomb (a ball) and Counter (no object).** The game's actors will perform the following functions:

- Boat:** Boat is the actor that you control using left and right arrow keys. Boat will wrap when it reaches the edge. When you press the "f" key, a bomb will be created.
- Bomb:** Bomb can only appear at the command of Boat. Once Bomb appears it immediately begins moving in the direction that Boat is pointing. If Bomb intersects with Pirate, both bomb and Pirate will disappear. In addition, when Bomb intersects with Pirate then the scoreboard counter will increase. If Bomb reaches the edge of the world before intersecting with a Pirate then Bomb will disappear. Bomb will have no effect on Boat or Anchor if it intersects with either of these.
- Anchor:** Anchor is a safe harbor for the boat and will disappear and reappear when Boat intersects with it. This can add points to the score though this was not done in this game. Anchor will wrap when it reaches the edge of the world and has no effect on Pirate or Bomb.
- Pirate:** Pirate moves randomly and can sink the Boat. In our game the boat will simply reappear at a random location and the game continues. A negative score could be assigned to this but we did not do this.
- Counter:** Counter is a class that appears and visibly keeps score. No object is assigned to the Counter class.
- OceanWorld:** OceanWorld is the background and holds the dimensions of OceanWorld as well as the location of the scoreboard.

The class diagram will appear as follows:

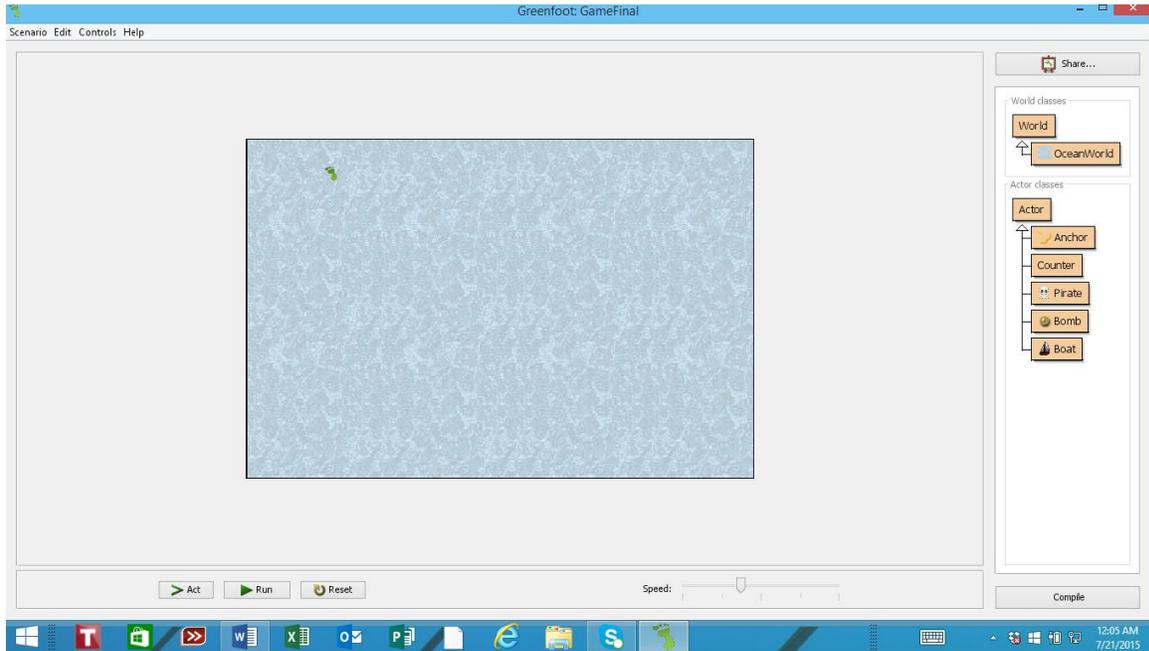


Figure 8. Class diagram for GameFinal

The code for each class follows.

OceanWorld

| Code | Comments |
|---|---|
| <pre>import greenfoot.*; public class OceanWorld extends World { Counter counter = new Counter(); public OceanWorld() { super(600, 400, 1); Prepare(); } public Counter getCounter() { return counter; } private void Prepare() { addObject(counter, 100, 40); } }</pre> | <p>The code here is the same as in the OceanWorld class used in the Scoreboard section.</p> |

Table 14. Code for OceanWorld in GameFinal

Counter

| Code | Counter |
|--|--|
| <pre>import greenfoot.*; import java.awt.Color; public class Counter extends Actor { int score = 0; public void act() { setImage(new GreenfootImage("Score: " + score, 24, Color.GREEN, Color.BLACK)); } public void addScore() { score = score +3; } }</pre> | <p>We use the same code for Counter as in the Scoreboard section</p> |

Table 15. Code for Counter in GameFinal

Bomb

| Code | Comments |
|---|--|
| <pre>import greenfoot.*; public class Bomb extends Actor { public void act() { boom(); move(10); } public void boom() { Actor pirate = getOneIntersectingObject(Pirate.class); if(pirate != null) { World OceanWorld = getWorld(); OceanWorld.removeObject(pirate); OceanWorld oceanworld = (OceanWorld)OceanWorld; Counter counter = oceanworld.getCounter(); counter.addScore(); OceanWorld.removeObject(this); } else if(isAtEdge()) { World OceanWorld = getWorld(); OceanWorld.removeObject(this); } } }</pre> | <p>The code for Bomb is the same as that found in the Scoreboard section with an addition.</p>  |

Table 16. Code for Bomb in GameFinal

Pirate

| Code | Comment |
|--|--|
| <pre> import greenfoot.*; public class Pirate extends Actor { public int e; public int f; public int m; public int n; public void act() { wrapAround(); boatDisappearAppear(); random(); } public void boatDisappearAppear() { Actor Boat = getObjectAtOffset(0, 0, Boat.class); if(Boat != null) { World OceanWorld = getWorld(); OceanWorld.removeObject(Boat); e = Greenfoot.getRandomNumber(599); f = Greenfoot.getRandomNumber(399); getWorld().addObject(Boat, e, f); } } public void random() { if (Greenfoot.getRandomNumber(100) < 40) { turn(30); } else if (Greenfoot.getRandomNumber(100) > 60) { turn(-30); } { move(5); } } public void wrapAround() { m = getX(); n = getY(); if (m == 599) setLocation(0, n); if (n == 399) setLocation(m, 0); if (m == 0) setLocation(599, n); if (n == 0) setLocation(m, 399); } } </pre> | <p>We create four variables, all integers. e and f are coordinates of a random position where Boat will reappear. m and n are the coordinates for wrapAround of the Pirate. We use different letters than in other methods because they are public and we don't want another method to change a variable.</p> <p>The code in the act method is simple and consists only of calls to code that we have already written. The only new method is boatDisappearAppear, which is the same method applied to the Pirate back in the Disappear and Appear section.</p> <p>random was defined in the Random Motion section.</p> <p>wrapAround was defined in the Wrapping an Object section.</p> |

Table 17. Code for Pirate in GameFinal

Boat

| Code | Comments |
|--|---|
| <pre>import greenfoot.*; public class Boat extends Actor { private int x; private int y; private int a; private int b; Bomb bomb = new Bomb(); public void act() { wrapAround(); anchorDisappearAppear(); moveAndTurn(); fireOnCommand(); } public void anchorDisappearAppear() { Actor Anchor = getObjectAtOffset(0, 0, Anchor.class); if(Anchor != null) { getWorld().removeObject(Anchor); a = Greenfoot.getRandomNumber(599); b = Greenfoot.getRandomNumber(399); getWorld().addObject(Anchor, a, b); } } public void moveAndTurn() { if(Greenfoot.isKeyDown("left")) { turn(-5); } move(1); if(Greenfoot.isKeyDown("right")) { turn(5); } move(1); } public void wrapAround() { x = getX(); y = getY(); if (x == 599) setLocation(0, y); if (y == 399)</pre> | <p>Boat will need four variables. x and y will provide coordinates for the Boat to wrapAround as was done in the Wrapping an Object section. a and b provide the coordinates for replacing the anchor at a random location after it intersects with Boat.</p> <p>This line creates the variable “bomb” so that a bomb can be created when the “f” button is pressed.</p> <p>The act method is kept simple. Only calls are included in the act method.</p> <p>anchorDisappearAppear is the only new method. anchorDisappearAppear simply uses the disappearAppear method and causes an anchor to disappear when the boat intersects with it and then reappear at a random location.</p> <p>The moveAndTurn method was defined in the Move and Turn section.</p> <p>The wrapAround method has been explained in the Wrapping an Object section.</p> |

| | |
|--|--|
| <pre> setLocation(x, 0); if (x == 0) setLocation(599, y); if (y == 0) setLocation(x, 399); } public void fireOnCommand() { if (Greenfoot.isKeyDown("f")) { World OceanWorld = getWorld(); OceanWorld.addObject(bomb, 0, 0); bomb.setLocation(getX(), getY()); bomb.setRotation(getRotation()); } } } </pre> | <p>The fireOnCommand method has been explained in the Shooter section.</p> |
|--|--|

Table 18. Code for Boat in GameFinal

Anchor

| Code | Comments |
|---|---|
| <pre> import greenfoot.*; public class Anchor extends Actor { private int g; private int h; public void act() { wrapAround(); move(1); } public void wrapAround() { g = getX(); h = getY(); if (g == 599) setLocation(0, h); if (h == 399) setLocation(g, 0); if (g == 0) setLocation(599, h); if (h == 0) setLocation(g, 399); } } </pre> | <p>The only method used by Anchor is wrapAround. We arbitrarily decided to declare the variables g and h as private, which means that they can only be accessed from within the Anchor class.</p> |

Table 19. Code for Anchor in GameFinal

Now, compile, correct errors as needed, drag a Boat, several Pirates, and several Anchors into OceanWorld and play!

Conclusion: So, Now What?

At this point you have a simple working game. Greenfoot, however, can create games with many more sophisticated features than have been displayed here. Here are some additional resources you might want to consider:

- A good next step is to get the book *Introduction to Programming with Greenfoot* by Michael Koellig. Michael Koellig authored the Greenfoot game development environment and his book will walk you through many additional features and more advanced games. The book is expensive (\$94 on Amazon for the 2nd edition). You might want to consider a used 1st edition for about \$40.
- Download the **book-scenarios** from www.Greenfoot.org/book. These files are intended to accompany *Introduction to Programming with Greenfoot*.
- Oracle Academy (academy.oracle.com) provides an excellent structured program for both Greenfoot and Java that is free of charge. The Oracle Academy materials are well suited for a formal course in Java.
- The Greenfoot website (www.Greenfoot.org), from which you downloaded Greenfoot, is a great user forum and you will learn a lot about Greenfoot over time by reading and viewing its content.
- YouTube, and mrstewartslessons.com contain a number of instructional Greenfoot videos.

We hope that you found this booklet helpful in getting started with Greenfoot. We believe that getting started in the MESA Computer Science Competition should not be any harder for students and advisors than making a simple glider or bridge. If you would like an online version of this booklet, just email us at **[insert email address here]**. The online version can facilitate distribution to your students. It is also possible to copy the code from the online document straight into Greenfoot, then compile and run. Good luck! We hope to see your students in next year's MESA Computer Science Competition.

Appendix A: Why learn Java and how does Greenfoot help?

Google the question, “What is the best computer language to learn?” and click on some of the top responses. Java will appear at or near the top of every list. Java was developed by James Gosling in 1994 at Sun Microsystems. The language is now owned by Oracle Corporation but don’t worry, everything needed to become a Java expert is available online for free. So, why is Java popular? Java was first developed to provide the software for household appliances, which use a variety of microprocessors. As a result, Java was developed in a way to be able to run the same program (commonly referred to as “code”) on different processors and different operating systems. By contrast, when using other languages, different versions of the same program must be written. The ability to run on different computers became even more important with the expansion of the internet.

Java is able to run on different systems by converting the Java program into an intermediate code called *bytecode* that runs on a piece of software called a *Java Virtual Machine (JVM)*. The JVM then converts the code into machine language that can be understood by each different processor. Machine language is a vast series of “0’s” and “1’s” that turn on and off electronic switches formed by transistors. The disadvantage of this intermediate step is that Java runs slower than some other languages, such as C++. Still, Java runs fast enough for most any application except high performance robotics or high speed graphics used in modern video games.

Among the other features of Java that make it popular are that it is relatively easy for new programmers to learn. It is, however, not just a teaching language but also widely used in professional applications. Also, Java is “safer” than some other languages. C++, which allows the programmer to get deeper into the machine, can also allow new programmers to create problems for themselves. By contrast, Java does not allow such complex access. Java also contains abundant libraries of predefined routines (called API) that programmers can access so they do not need to write all of their code from scratch. In sum, while learning Java will present challenges, it is an excellent first language for a new programmer.

Greenfoot is a special software package designed to introduce new student-programmers to Java programming. Greenfoot was written by Michael Koellig at the University of Kent in England and is endorsed by Oracle Corporation as a teaching tool for Java. Greenfoot is an environment designed to allow users to easily create video games. In addition to the video game development environment, Greenfoot contains specialized commands that aid in the development of a game. Examples of these are “turn” which causes the game object to turn, or “getLocation” which returns the x and y coordinates of the object’s location.

The most valuable characteristic in teaching Java is that Greenfoot requires the student-programmer to create the game as a collection of “objects” that interact with one another. This process familiarizes students with the concept of *Object Oriented Programming*; an essential

structure in Java, C++, and several other programming languages. In addition, the student-programmer creates the code for the game in Java thereby learning the structure and syntax of Java. Students become familiar with the use of semicolons, braces, and conditional statements such as “if-else.”

Appendix B: Setting Up Greenfoot

Setting up Greenfoot involves two easy steps. Both downloads are free of charge.

1. Download Greenfoot. Go to www.greenfoot.org/download and download the appropriate version of Greenfoot
2. In addition, you must download a Java compiler. The latest version is the Java Development Kit 8 (JDK8). Go to the Oracle downloads page: <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html> . Most modern computers are 64 bit and will use the Windows x64 or Mac OS X x64 versions.

Appendix C: Dealing with Errors

Your first effort at writing code will be a little frustrating. The error messages might not always be clear and, sometimes, they might even give you an explanation that is inaccurate. For example, you might receive an error that says “; expected” meaning that the compiler was expecting you to have placed a semicolon in the identified location. In fact, the problem might be that your braces don’t match up properly. The reason is that the compiler can’t always know what the programmer intended and, therefore, diagnoses the error incorrectly.

Most of your errors will occur for one of the following reasons:

1. Misspelled words
2. Improper capitalization of words
3. Mismatching braces or braces turned the wrong way (called a “parsing” error)
4. Omitting semicolons (;)
5. Not declaring variables

The more code you type, the better you will become at avoiding errors. For this reason, it is strongly recommended that you type all of your code rather than copying and pasting.

If you compile and see a highlighted yellow line on your code and an error on the bottom, such as in the image in Figure 9, you will need to correct your code. A small red mark identifies the exact location of the error. In this

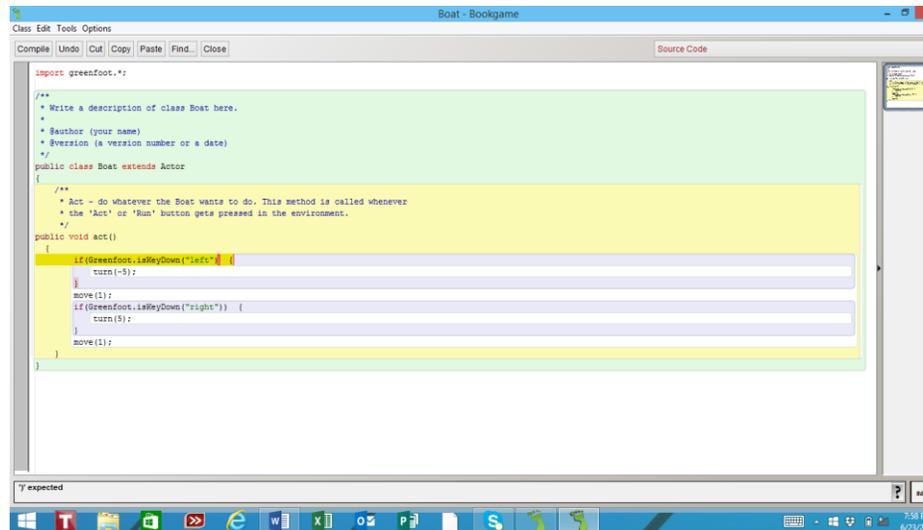


Figure 9. How an error appears in the code editor

case, a parentheses was omitted and must be added on the highlighted line. If you compile from the main screen and there is an error, you will see a Greenfoot icon flashing on your task bar (at the bottom of your screen) prompting you to enter the code editor.

If you use the code editor and see the message “**Class compiled-no syntax errors**” you’re good to go. If you compile from the main screen and there are no errors then you will see your world appear ready for you to enter actors and run.

Appendix D: How Do I Know What Methods Are Available To Use?

If you go to the “**Help**” pulldown on the upper left side of the main Greenfoot screen you will see an option “**Greenfoot Class Documentation.**” When you click the link you will see a table titled “**Class Summary**” with seven classes listed below. At this point, you are primarily interested in the World, Actor, and Greenfoot classes. If you **click on Actor and scroll down** you will see all of the methods that have been predefined in Greenfoot to be used by Actor. You will also see the return type (void, int, etc.) as well as the arguments (inputs) required for the method. These methods are also inherited by any subclasses of actor such as Boat. You will notice that you don’t see the method “isKeyDown,” which was used in the MoveAndTurn folder. If you click into the Greenfoot class over on the left panel and scroll down you will see this method. Because isKeyDown belongs to the Greenfoot class, any actor wishing to use this method must include the word Greenfoot before isKeyDown.

Another way you can access the methods available is from the code editor. Open the code editor for the Boat, place the cursor anywhere inside of the editor, then **press <Control-Space bar>** at the same time; a menu of methods available to the class will appear. You can learn to use various methods by reading the documentation and by experimenting with each of them.