Section 1: Windows versus Macintosh Mouse

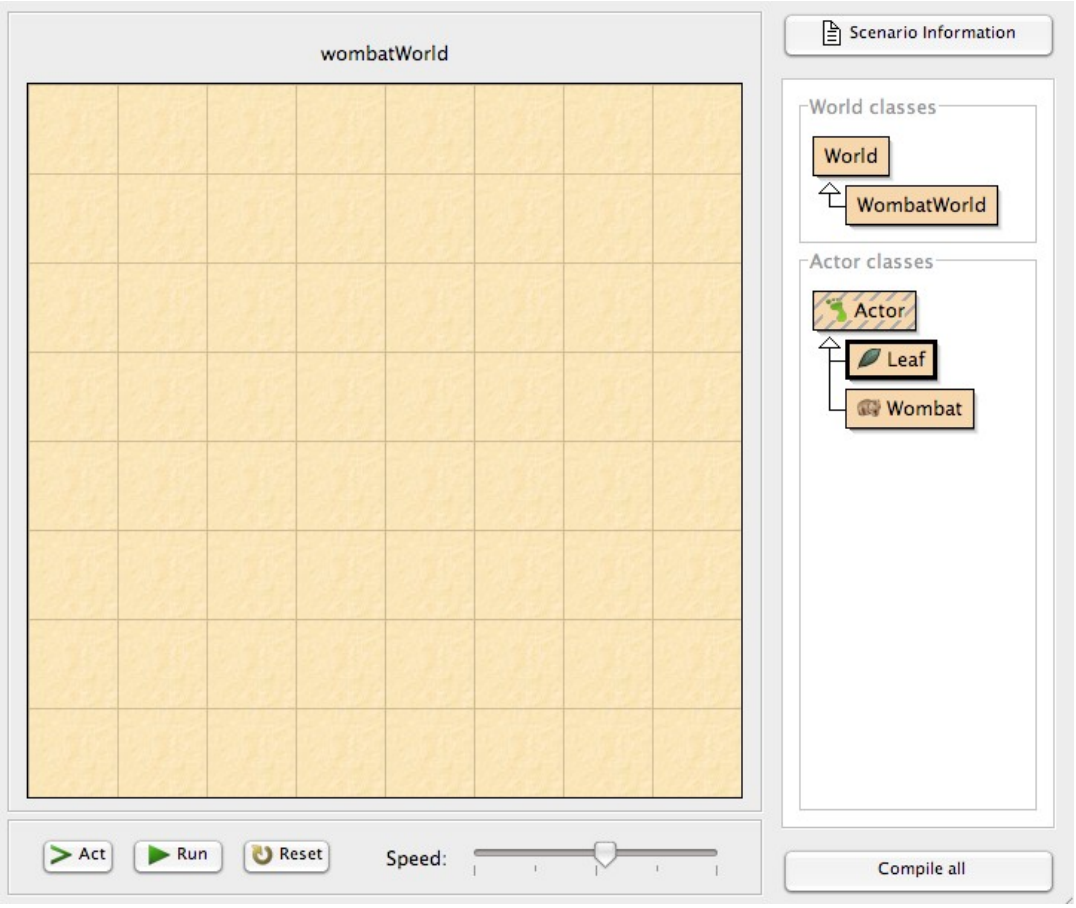
Users of these notes may be running Greenfoot on a Windows machine, a Mac, or a Linux machine. In Windows and Linux the mouse is configured to have at least a left button and a right button. Macintoshes on the other hand may have only a single button mouse. This single button is equivalent to a left button on a Windows machine. To create a "right click" with a Mac, just hold down the "ctrl" key while you click with the mouse. For the rest of this book we will just use the term "right click", if you are using a single button Mac mouse, please remember this means ctrl-click, whereas left-click just means click.

Section 2: Use Greenfoot to run "Wombats"

If you are in a school lab taking a course, Greenfoot should already be installed for you. Just start greenfoot by double clicking on it. If you intend to run this on your own computer and have not already downloaded and installed greenfoot, go to www.greenfoot.org and download and install Greenfoot. Then start greenfoot.

When you start greenfoot it will open the last scenario that was open. The first time it is run it may start up Wombats, or, it may start up nothing. If you do not get a brown grided screen with the word "wombatWorld" on the top, then from the main menu select *scenario* and then

open and then select wombats. You should see something that looks like this:



Wombats is not technically a game, but rather a simulation or a toy. It was created by the Greenfoot team to illustrate how to use Greenfoot. The “world” is the grid of brown squares. When you open Wombats there are no actors in the world. Actors are objects that exist in the world. Actors may move or have actions. In this wombat scenario there

are two types of actors: Leaf and Wombat. They are listed in the Actor classes window pane on the right.

Before we do anything else, first click on the *Compile all* button in the bottom right hand corner. This makes sure that the programming code, which we will be changing and writing later, is all up to date. Next create actors on the screen by right clicking on the “Wombat” actor class rectangle and select new from the drop down menu. Then move the mouse over and put a wombat into the world by clicking someplace in the world. Now do the same for a leaf object. You can repeat this to put many leaf or wombat objects into the world. Once you have placed a wombat in the world click on the *Run* button on the bottom. You can see the wombat moves forward and when it hits a wall it turns left and continues. Hit pause to pause the simulation. Now add a leaf object in the wombat’s path by right-clicking on the Leaf actor class rectangle and moving a leaf to a square in the world that the wombat will visit and clicking to place the leaf. Click run again. When the wombat reaches a square with a leaf the leaf disappears. Obviously the wombat is a hungry wombat. One last thing to explore in this initial example is the speed of the simulation. Along the bottom of the Greenfoot window you will find a slider bar labeled “speed”. Move the slider to the left and

click run, or move while the simulation is running. The speed button controls how fast the simulation is run.

VIDEO: To see a video of the above concepts run the website video "p4_video_wombat.mov"

Section 3: Play a Game

If your instructor has already put LittleRed on your computers for you, go ahead and open LittleRed and play the game. If not, first download LittleRedFinal.zip and play the game. You open littleRed by chosing Scenario from the main menu, open form the pull-down menu, and then navigating to the folder that contains LittleRed. LittleRed is a a simple game created based on the fable Little Red Riding Hood. Little Red, represented by the small red rectangle that starts in the upper left hand corner, must collect all the apples and then go home to grandmother's house before any wolves eat her. You control little red by using the arrow keys.

Questions a student should ask themselves:

- How is the art created for the apples, wolves, little red, and the house.
- The world is populated with apples and wolves right at the start instead of added by right clicking on the actor class rectangles. How is this done?
- How is the keyboard used to control Little Red?

- The wolves move by themselves. How is this done?
- When Little Red hits an apple, the apple disappears. How is this done?
- When a wolf hits Little Red, the game ends by showing a “You Lost” screen. How is this done?
- When Little Red gets enough apples and finds the house, the game ends with a “You Win” screen. How is this done?

We will answer all of these questions in subsequent chapters of this book.

Section 4: Creating your own simple scenario

The two scenarios you have used so far already existed. For creating your own games you will need to create a scenario from scratch. In this section we show how to create a simple Greenfoot scenario from the beginning. There are three main parts of a Greenfoot program: a scenario, one or more worlds, and one or more actors. The scenario is the entire collection of code and art assets that make up the game. The worlds are universes or spaces within the program. The actors are the *objects* which move around in the worlds. Objects are a special construct in object-oriented computer programming. Objects can have data associated with them (members) and actions they can perform (methods).

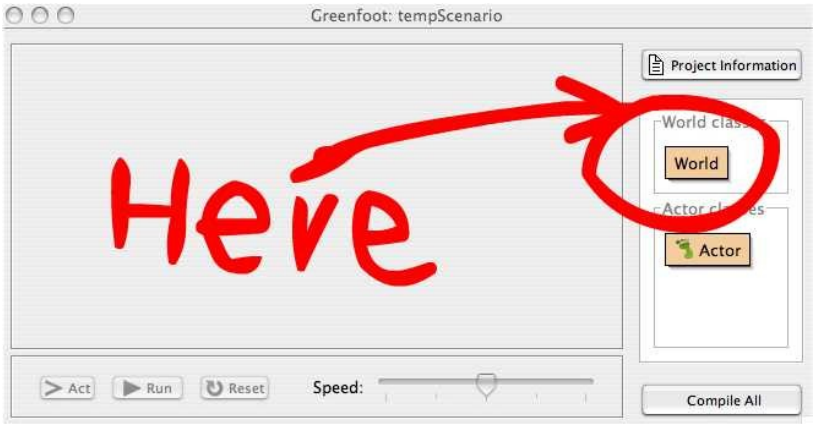
You will create a new scenario for each lab exercise, homework assignment, and project. A scenario is completely contained within one folder on your computer. This folder has the same name as the scenario itself. All scenarios folders are found inside the “scenarios” folder where Greenfoot was installed on your computer. The folder contains all of the files that comprise the scenario. All of the code that you write for a scenario is stored in files in this main scenario folder. The scenario folder contains two sub-folders: one called “images” and one called “sounds”. The images folder contains all of the artwork files for the scenario. The sounds folder contains all of the sound files for the scenario.

Exercise 1.1: Create a new scenario with non-moving sprites.
First, we need to create a new scenario. On the main menu, use the mouse to click on “Scenario” and then “New”. A new dialog window should appear. Toward the bottom of this menu is a place to type in a file name. Type in a name such as “BugScenario”. You can use any name you want and you should pick logical names such as “assignment1” or “spaceInvaders”. Now click on the **Create** button in the lower right corner of the dialog window. You should now have a new window with the words Greenfoot: BugScenario at the top of it.

In summary, the steps to create a new Greenfoot scenario are:

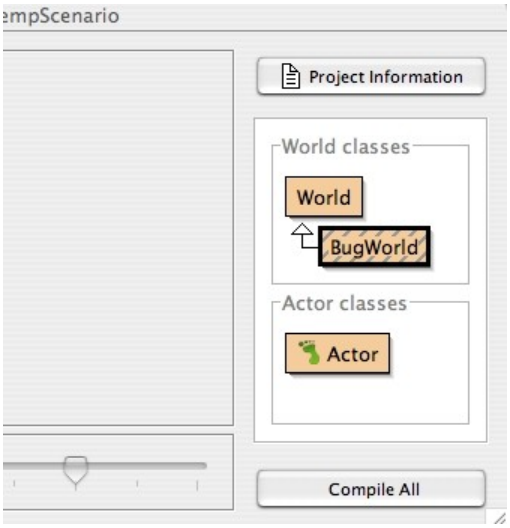
- Choose “Scenario” on the main Greenfoot menu bar at the top left of the Greenfoot window.
- Choose “New” on the “Scenario” menu
- Type a name for the scenario in the New Scenario dialog window.
- Click on the “Create” button in the lower right corner of the dialog window.

We now need to create a world in scenario “BugScenario”. This new world is the universe where the actors will exist and move. To create a new world, use the mouse to right click on World on the right side of the Greenfoot window.



A menu should appear. Choose “New subclass...”. A dialog box will appear where you can name the new type of world. Type in a name

such as “BugWorld” and either hit the return key or mouse click on the “Ok” button. Note you must make the World name all one word: “Bug-World” not “Bug World”. This is creating a Java class, and class names may not have spaces in them. It should now look like:



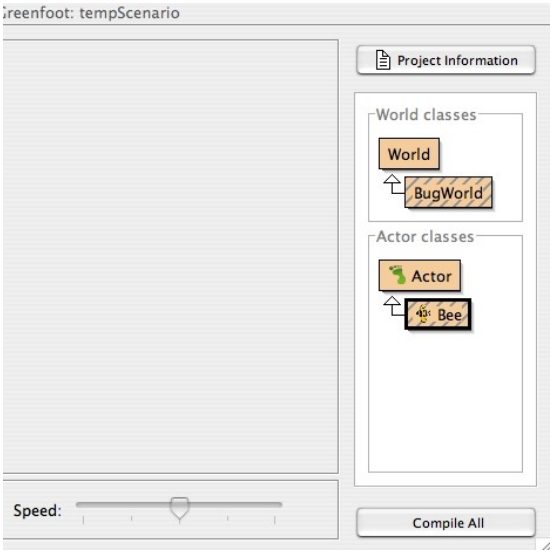
Notice that a new box has appeared in the “World classes” sections of the Greenfoot window. The arrow, drawn as a line and triangle, is there to remind us that BugWorld is a kind of World. The BugWorld box is hatched with diagonal lines to show that it has not been *compiled*. The compiling process converts the code inside the class into an executable program that can be run. To compile newly added classes, click on the “Compile All” button (in the lower right corner of the Greenfoot window). After doing this you have created your first Scenario and your own World.

In summary, the steps to create a new type of world are:

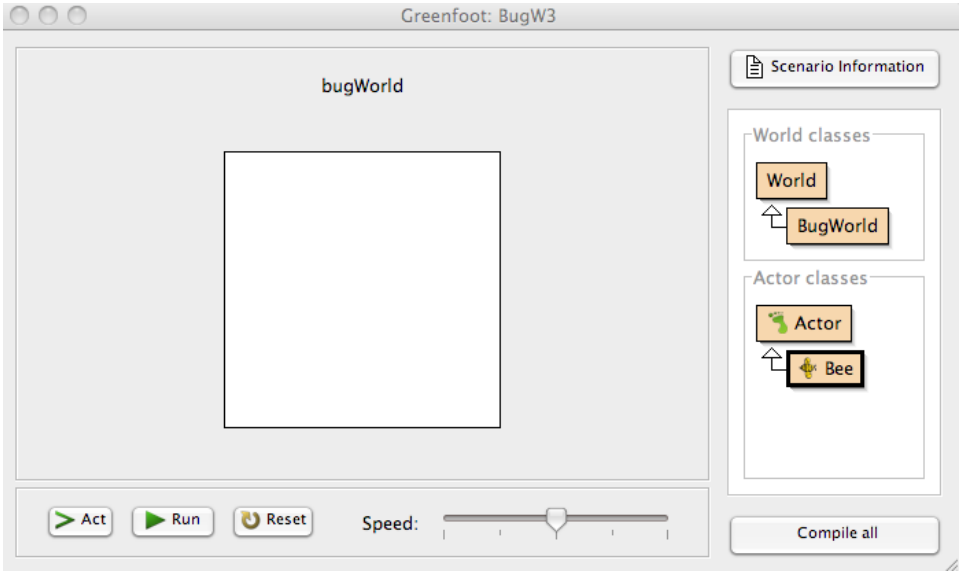
- Right click with the mouse on World on the right side of the Greenfoot window.
- Choose “New Subclass...” from the menu.
- Type in a name for the new type of world.
- Press return or click on the “Ok” button with the mouse.

We will now create bees and add to our BugWorld. We will create Bees as a kind of Actor similar to the way that BugWorld is a kind of World. First right click with the mouse on the Actor box on the right side of the Greenfoot window. Choose “New subclass...” from the drop down menu. A dialog window will appear. In the space at the top of this window type in a class name such as “Bee”. DO NOT CLICK THE “OK” BUTTON QUITE YET. We now need to choose an image for the Bee class. Click on “animals” under *Image Categories* near the middle of the window and then choose one of the bee pictures under Library images. The chosen image should appear toward the top of the window. Now click on the “Ok” button or press return.

You should now see a Bee box under Actor (under Actor classes on the right side of the Greenfoot window):

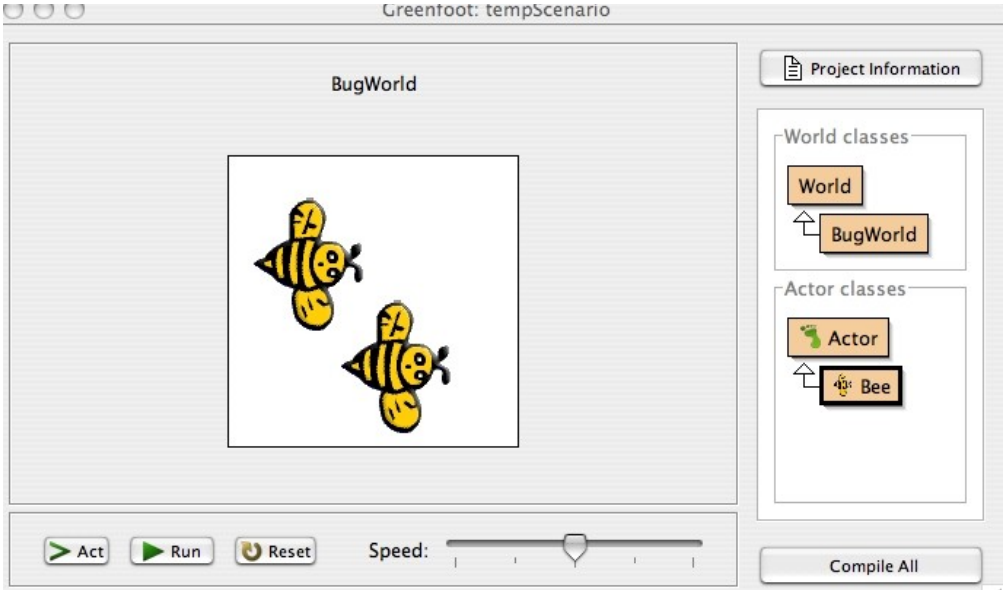


Again, notice that the new box is hatched with diagonal lines. This diagonal hatching means that we need to compile the new class. Click on the “Compile All” button to do this. Do this now. The diagonal hatching will disappear and a white box to hold the world of actors will appear in the main window. The Greenfoot screen will now look like this:



We can now add Bees to the BugWorld. Right click on Bee and choose *new Bee()* from the menu. Drag the mouse until the Bee is inside the white box and left click with the mouse to drop the Bee.

You can repeat this to create as many Bee objects as you like. Here is an example with two Bees in it:



To run your scenario click on the “Run” button at the bottom of the Greenfoot window. NOTHING HAPPENS! This is because so far we have just created the empty shell of the objects. The Bees don’t know how to do anything yet! We need to add code which controls their actions. The code is a set of instructions that tell the Bees what to do.

VIDEO: To see a video the instructions for the above section run the website video “p4_video_newScenario.mov”

Exercise 1.2: Making the Bee move

Now we will make the Bee objects move around on the screen. To do this, we will add code to the act() method for the Bee class.

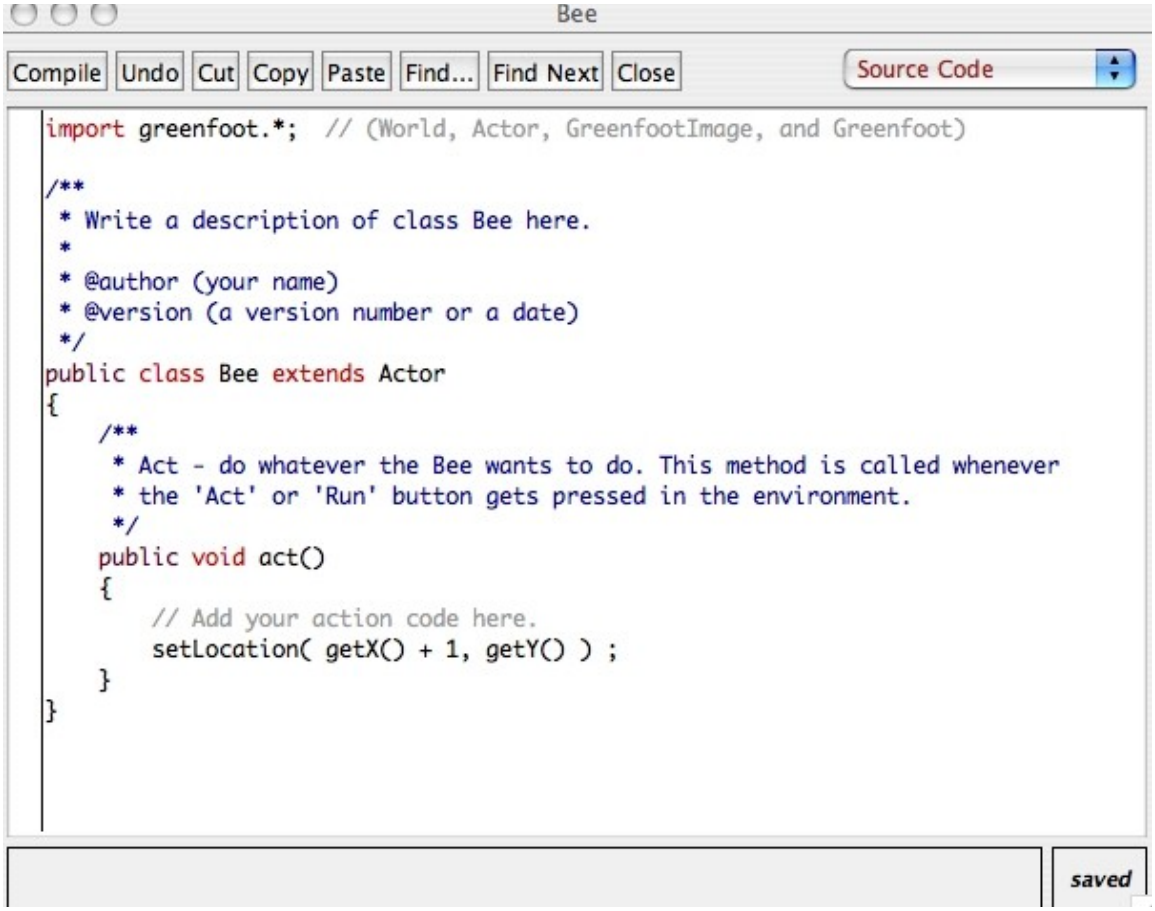
- Double click on the Bee class rectangle (under Actor classes on the right side of the Greenfoot window). This opens the code window for the Bee class.
- Look for the line which says “public void act()”. This is the beginning of the “act” method for the Bee class. Everything between the curly braces, “{“ and “}”, is what the Bee will do. Right now it is empty except for a comment. A comment is text that is there for a human to read, it is not part of the object’s instructions. Anything after a double slash, // , until the end of the line, is a comment and is ignored. So, the act() method has no code in it, i.e. no instructions for what the bee should do. Thus, the Bee objects do nothing. To make the Bee objects do something we need to add code to the act() method.

- Type in the following between the { and the } of the act() method:

←

← setLocation(getX() + 1, getY());

Make sure each (is matched with the correct), and don't forget the ; at the end of the line. Note that exact spelling is required, including capitalization. I should look like:



←

- Click
- After adding this line of code to the act() method for the Bee, compile the code again by clicking on the compile button.
- Add a Bee object to BugWorld (again, right click on the Bee box, choose "new Bee()", and left click to place it in the BugWorld.

- Click on the “Run” button.

You should see each of the Bee moving to the right until it gets to the world boundary whence it stops moving. Experiment with changing the Speed slider bar and moving the bee’s starting place around in the Bug-World.

Look closer at the code. The **method** setLocation(x, y) changes the location of the Bee sprite to whatever you put in for x and y. If I were to say setLocation(10, 20) then the location of the sprite would be changed to location 10,20. The Greenfoot world is divided into a grid of cells. By default, the grid is 20 cells wide by 20 cells high where each cell is a 10x10 pixel square. A pixel (short for picture element) is the smallest “dot” that the computer can draw on the screen.

Go ahead and change the setLocation(getX() + 1, getY()) in your code to: setLocation (10,10). Compile it, put a Bee in the world, and hit run. Regardless of the initial location of the Bee, you will see the Bee moves to location (10,10) and stays there. Change it to (19,0), then (19,19), then (0,0). Try changing it to setLocation(100, 100). The grid is only 20x20, so, Greenfoot puts the Bee sprite as far over and far down as possible.

Again look closer at the line of code: setLocation(getX() + 1, getY()). The built-in **method** getX() returns the current x-location of the actor object. (remember, a Bee is an Actor object). The **method** getY() returns the current y-location of the actor object. Thus, getX() + 1 is one more than the current x-location of the actor object, so, setLocation(getX() + 1 , getY()) will set the location of the current object to it's current x-location plus 1 and it's current y-location. All actor classes have built into them methods getX(), getY(), setLocation() and a bunch of other methods.

Section 2.1: A Bigger World for Big Bees

The world is rather small for these big bees. We need to make the world larger. To do this, double click on the BugWorld class on the right side of the Greenfoot window. In the code window that pops up, change the line:

← super(20, 20, 10);

to

← super(60, 50, 10);

and click on the "Compile All" button.

The statement super(60, 50, 10) sets the world size to be 60 cells wide by 50 cells high where each cell is 10 pixels on a side. If you change the statement to super(30,25,20) the world is the same size on the screen,

but only 30x25 cells are in the grid, where each cell is now 20x20 pixels.

VIDEO: To see a video demonstrating how to change code to make a Bee move and how to change the world size, run website video “p4_video_changeCode-MakeBeeMove.mov”

Exercise 1.3: Playing with grid sizes

First set the world to (60,40,10). Put a Bee in the world where it’s act method has one line of code: setLocation(getX() + 1 , getY()) ;

Run this code. Now change the world to (30,20,20) and run. Now change to the world to (15,10,40). What happens? Why?

Section 2.2: The Greenfoot Act and Run Buttons and Inspect

Look more closely a the Greenfoot window. You see the “Run” button and have used it, but there is also an “Act” button. When you push the “Act” button the act( ) method is called on each and every Actor object in the world. Thus, if you have three Bee objects in the world, where

Bee class has an act( ) method with the line of code setLocation(getX() + 1 , getY()), then that line of code is called once each for every Bee object in the world each time you press the act button.

Put a Bee over on the left hand side, then repeatedly press the Act button in a slow rhythmic fashion. Now put the Bee back at the left, move the speed slider all the way to the left, and press Run.

Another extremely useful part of the Greenfoot IDE (Integrated Development Environment) is the “Inspect( ) “ method. Put a Bee object in the world. Now right-click (CTRL-click for single button Mac users) on the Bee object. In the pop up menu select “Inspect”. This opens a window that show the current x and y locations of the object (plus some other stuff). Move the window over so you can see both it and the World. Click on the act button and look at how the x-location value has changed in the inspector window. This is a very useful tool for helping you both understand and debug your code.

Section 2.3: Moving in other directions

Obviously we need to be able to move in directions other then just to the right. You can use the setLocation() method to move in other directions. The y-axis starts at zero at the top and increases as you move down, thus, increasing the y-value in the setLocation() method actually moves the sprite down.

Exercise 1.4: Moving left, right, up, down, diagonally

Modify your Bee code to make the bee move in each of the following ways:

- left
- right
- up (hint, you will need to change the y-coordinate instead of the x-co-ordinate)
- down
- diagonally
- faster (further) without moving the speed bar

What you have learned:

- How to create a new scenario
- How to set the size of the world
- How to create new actor subclasses
- The coordinate system (an increase in y-coordinate is actually DOWN)
- How to move an actor in different directions

Questions students should ask:

- How can a sprite detect if it is going off the screen?
- How can I make a sprite bounce back and forth across the screen?

Section 2.4 The Edge of the World and if/else statement

Notice that when an actor tries to move beyond any of the World's boundaries it will stop Lets assume the world is 60x50 cells. When a sprite tries to move to the 61st cell the world boundary is violated and the Bee stops. This may not be what you want. Lets say you want the Bee to go to x-location 50 and then stop.

To do this we need to add code to the Bee class that says something like:

If my new x-location is will be less than 50, move over one space, if not, do nothing, i.e. don't move.

We need a condition to test if the new location will be less than 50. The code might be:

```
public void act()
   {
      if ( ( getX() + 1 ) < 50 )
         setLocation( getX() + 1, getY() ) ;
   }
```

Try this out. Put two Bees in the world at different starting locations and see what happens.

Lets say now that if the new location is going to be 50 then we want to move the Bee back to the beginning, i.e. x is changed to zero. Code for this could be:

```
public void act()
   {
      if ( ( getX() + 1 ) < 50 )
      {
         setLocation( getX() + 1, getY() ) ;
      }
      else
      {
         setLocation( 0, getY() ) ;
      }
   }
```

Here we have a full Java if statement. The general form is:

```
If (condition)
{
 statement block 1
}
else
{
 statement block 2
}
```

In other words, if the stuff in the ( ) is true, do the if statement block 1. If it is not true, then do the stuff in statement block 2. A statement block is one or more statements (or lines of code) surrounded by curly braces { }. Consider the following code, what do you expect it to do?

```
if ( ( getX() + 1 ) < 30 )
{
   setLocation( getX() + 1, getY() ) ;
   setLocation( getX() , getY() + 1) ;
}
else
{
   setLocation( 0, 0 ) ;
}
```

Not sure? Go ahead and type it , hit run, and see what happens. The if statement block has two statements in it, so both are done if getX() + 1 is less than 30. Note we could have instead done this in one statement:

```
setLocation( getX() + 1, getY() + 1) ;
```

but we wrote it as two statements so that you can understand that statement blocks can have multiple statements in them.

This is all fine and dandy, but it might make more sense to have the Bee make it to the edge of the world and then go back to zero. We know the world is 60 cells wide so we could change the code to:

```
public void act()
   {
      if ( ( getX() + 1 ) < 60 )
      {
         setLocation( getX() + 1, getY() ) ;
      }
      else
      {
         setLocation( 0, getY() ) ;
      }
   }
```

Try that code out with a couple of Bees initially at different x-locations. But what if someone comes along and changes the world size to 40x50 in the World class by saying: super(40,50,10) ? Now the Bees never make it to 60 so they never get reset at zero. We need some way to automatically now the dimensions of the world. The world class has methods getWidth( ) and getHeight( ) which return the dimensions of the world. In order to use them from inside of an Actor object we need to

call the getWorld( ) method. The following line of code will return the width of the world (in cells, not pixels):

```
getWorld().getWidth() ;
```

The getWorld() method returns the world the actor object is contained in. Then, we call the getWidth() method of that world. Thus, the following code would do what we want no matter how we change the dimensions of the world in the super(width,heigh,cellSize) method:

```
public void act()
   {
      // Add your action code here.
      if ( ( getX() + 1) < getWorld().getWidth() )
      {
         setLocation( getX() + 1, getY() ) ;
      }
      else
      {
         setLocation( 0, getY() ) ;
      }
   }
```

If the current x-location of the Actor object + 1 is going to be less then the width of the world, then move it over one, otherwise, reset it back to zero. Try this out, change the size of the world, and see that it still works.

Exercise 1.5: Repeating Movements

Create scenarios to do the following:

a) a Bee object moves down, when it gets to the bottom it is moved back to the top so it is constantly looping down the screen

b) A Bee object starts in the lower left corner of the screen and moves diagonally up until it goes of the top of the screen then goes back to the lower left corner.

c) Create a second actor object other than a Bee (anything you want). Give each object a different cyclical movement behavior. For example, maybe a bee moves in a circular loop to the right and a frog moves in a circular loop down. Put an actor object of each type in the world and let them run.