Problem Solving through Programming with Greenfoot

Mark Lewis

February 24, 2017

Preface

There are two significant goals of this text: to teach basic programming skills and to improve students' problem solving capabilities. This book is intended to provide a fairly gentle introduction to the topic of programming using examples that will relate to and be of interest to most college students. The book focuses on the Java programming language and begins in the Greenfoot environment, then moves on to the Eclipse environment later on. The teaching of programming is in many regards secondary to the broader goal of improving problem solving skills and serves as a vehicle to accomplish that goal.

This book is intended to be used in classes for introductory computer science students who don't already have a background in programming or for courses that teach programming to students who don't intend to major in computer science or the natural sciences. Using Greenfoot for the early chapters affords an easy and approachable way to understand the main concepts of object-oriented programming and allows us to build interesting, interactive programs with graphical output right from the beginning. The fact that both Greenfoot and Eclipse use standard Java means that this book does not have to cover two different languages with different syntax or semantics. In addition, Greenfoot has a wonderfully small and simple API which allows students to learn how to read and make use of a real API without throwing them into the vast expanse of the full Java API.

Early on, examples are selected that fit well with Greenfoot's visual orientation and have students writing games, doing simulations, or solving simple programs in the 2-D graphical environment. From there the examples grow into things that students can see as relevant to their lives or their course work outside of programming. The primary objective of this book is not so much to teach programming, but to instead to teach problem solving. Programming just happens to be a superb instrument for students to learn and hone these skills, and the computing environment helps them not only to formalize their thought processes, but break things down into their most fundamental pieces to ensure that all the problems they have to solve are small ones.

Late in the book students are moved away from Greenfoot as we begin to use Eclipse, a professional scale development environment. The first steps in Eclipse are taken writing applications, and students are introduced to the full Java API, having seen only parts earlier when certain standard classes were used in Greenfoot. Many of the topics in this text stand on their own and instructors

can choose to skip certain sections to fit their teaching objectives for the class as a whole.

Contents

List of Figures

List of Tables

Chapter 1

Introduction to Programming and Problem Solving

1.1 Programming and Problem Solving

Welcome to the world of programming! The fact that you are reading this likely implies that you are taking an introductory programming class. At some point, you've probably heard someone (maybe yourself!) ask you the question, "Why I should bother to learn how to program?" We want to address that, but before we do we have to take a step back and answer some other questions first.

Let's begin with the question, "What is a program?" A program is a step-by-step set of instructions that tells a computer how to do something. A more general term is "algorithm" which is a step-by-step description of how to accomplish some task, on a computer or otherwise. This is actually a concept that everyone has been familiarized with, even if you have never programmed a computer. There are many different contexts in which you interact with step-by-step instructions for completing tasks, one of the most common being recipes. A recipe is simply a set of instructions that a person needs to follow in order to complete the preparation for a certain dish. Recipes aren't the only example either. Any set of instructions given to people for them to complete something, whether it be a worksheet for an elementary school student, a Lego building guide, or a tax form from the IRS, is basically an algorithm intended for a human.

One of the first differences you have to understand between writing a set of instructions for a person and one for the computer, is that the computer, while very powerful and fast, is rather stupid. It has little capacity to interpret things as something other than what is presented directly. As a result of this, the instructions given to them have to be very precise. Computers do exactly

what you tell them to do, nothing more, nothing less. When you tell them to do something you also have to break it down into steps that they understand. The only steps the computer really understands are steps that are very small and exact. This has a significant impact on how we write programs, for example we don't use normal English to do it. It also makes programming a very valuable skill to learn even if programming computers is not part of your life plan. Understanding how computers work, will help in any field that uses them, and let's face it, they aren't going away soon.

Everything inside of a computer is actually stored as part of a vast set of numbers and the computer does nothing more than basic arithmetic operations on those numbers. The instructions that computers follow are numeric values. What numbers correspond to what instructions depends on the details of the computer you are using. Early programmers had to write everything in this numeric format that is commonly called machine language because it is the only language the computer really understands.

Writing programs in machine language is difficult, time consuming and rather unproductive. For these reasons, it wasn't long before people came up with other languages to write their programs in. To make these languages work they first had to write programs that would translate from their new languages to the machine language. These programs are called compilers, and they allow us as humans to write programs in languages that are vastly more readable and that are easier for us to think in than trying to do everything at the level of the machine. They also have the advantage that while different machines might have different machine languages, a single program from a higher level language can run across many machines as long as each one has a compiler that produces the appropriate type of machine language.

There are literally thousands of different programming languages that have been developed over the decades. Many of them are still in use today. Some languages are good for certain tasks. Some people seem naturally drawn to one language over another. In this book we will be using the Java programming language originally created by Sun Microsystems. Java is a high level programming language that is used in many different areas. It is freely available and has been implemented on many different systems. You can download Java for your computer by going to http://java.oracle.com. To write Java programs all you need is a general text editor, like Notepad on a Windows system.

Just because you can write programs with just a text editor that way doesn't make it optimal or even appealing. This book starts off using a programming environment called Greenfoot (`http://www.greenfoot.org`) that was developed for educational purposes. Greenfoot will help you to understand some of the aspects of the Java language and also provide you with a fairly simple way to make interesting programs with relatively little effort. Later, when we have developed our understanding of the main concepts of programming in general and Java specifically, we will spend some time in the Eclipse (`http://www.eclipse.org`) programming environment. Eclipse is a fully featured professional software development environment. Such environments have a great number of different tools built into them that can be very helpful to professional software developers

and that can make programming a lot easier. Of course, you have to reach a certain level of understanding before they are really of significant assistance.

Now that we have addressed what a program is, we can return to our original question of why you should bother to learn to program. There are two very different, yet equally valid answers to this. We will start with the one that is most obvious to people. You should learn how to program because it is a valuable skill. If you are considering majoring in computer science, programming is going to be at the heart of pretty much everything you do. It is how you communicate with the computer. While computer science is about a lot more than programming, the ability to write programs and understand how they work is an essential undercurrent for the rest of the field.

Even if you have no interest in majoring in computer science or ever working in computer science, learning how to program is still a valuable skill because computers are so omnipresent. Nearly every activity you do in your daily life involves a computer, whether you realize it or not. When you think about computers in your life, you might only think of your desktop or laptop. However, your smart phone, tablet, and every other electronic gadget you might own these days is really a digital computer. There are computers in nearly every car you see on the road that control various aspects of the operation of the engine as well as the internal electronics.

So far we have just listed what you might call "client-side computers", computer devices that people own and use, the ones you might interact with directly. Computers also control a lot of things in the background. They are responsible for street lights. They route packages that are sent to you and they oversee inventory control at your local stores. They even govern the the distribution of electricity and water. If all the computers in the world were to suddenly cease to function, us humans would find we have a very hard time getting anything done. Unless it could be fixed quickly, society would break down and mass chaos and death would quickly follow in developed nations.

All of those computer systems run on programs. You have to have programs telling the computers what they should be doing at every instant they are in operation. Now, this might sound like a motivation for you to switch your major to computer science. In some ways it is, but the reality is that you need people with domain specific knowledge of different fields as well who can help determine how to best put computers to use in those areas. You see, even if you don't plan on programming for a living, you will be interacting with programs for a living. If you advance far enough in whatever career you pick, at some point you will be responsible for making decisions that relate to the acquisition and use of technology. The use of computers is only increasing over time, and even if you don't want to be the person writing the programs, you will at some point be working with those people to help them create software that improves how you do your job. The more knowledge you have about what those people are doing, the better you will be able to communicate with them and the more valuable you will be at your job.

There is also a less obvious reason to learn how to program that applies equally well to anyone. Remember what a program is? A set of instructions

given to a computer to tell the computer how to complete some task. Most programs solve a problem or category of problems for us. The first step in any programming task is figuring out how to solve the problem yourself in sufficient detail that you can give the instructions to a computer. Learning how to program is a great way for anyone to improve their problem solving capabilities. As we will see through the course of this book, a lot of the challenge in programming comes from the part that you do before involving a computer at all. You start with a problem and try to determine how you can go about solving that problem. When problems are large, the first thing you typically try to do is break them down into smaller pieces. In fact, a goal that we often strive for in computer science is to never do anything hard. If we have a problem that is big and hard we break it into smaller pieces over and over again until we get down to pieces that are easy to solve.

1.2 First Steps with Greenfoot

Now that we have established what programming is and why you should learn it, we can introduce the environment that we will be using to do our programming through much of this book. For this section you should run Greenfoot on your computer and play along. As with many skills, the only way to learn how to program is to sit down and do it. Abstract discussions can be beneficial, but you won't really understand what is going on until you have gone through the steps yourself. So with that idea in mind, you should start up Greenfoot. If you don't have it on your computer, simply go to `http://www.greenfoot.org/`, download it, and install it. Greenfoot will require that you have Java installed as well. If you get a message saying that you don't have Java, you can download it from `http://java.oracle.com/`.

You should also download the zipped scenario file for this book. It contains all of the scenarios that we will be using. After downloading, you want to unzip the file in the scenarios directory where Greenfoot was installed. (If not, simply remember where you unzipped it so that you can get to it to open the different scenarios that we will use.)

When you bring up Greenfoot, go ahead and open up a scenario. To start with we will use the PSPGWombats scenario that was in the zip file. When you have loaded this, you should see a screen like Figure 1.1. A scenario is simply the term used in Greenfoot for how it organizes programs. We will work with many different scenarios over the course of the text and you will even create some of your own. Each scenario contains the text files for pieces of code, images that are used in the scenario, and various other information that Greenfoot uses.

The Greenfoot environment has several different areas in it. The largest part of the screen shows the graphical representation of the world. In the case of the Wombats scenario, this starts off with what looks like a sandy grid with one wombat in the top left and three leaves scattered around. To the right of the world display, Greenfoot lists the different CLASSES that are part of this scenario. In the Wombats scenario we see the classes broken into two areas.

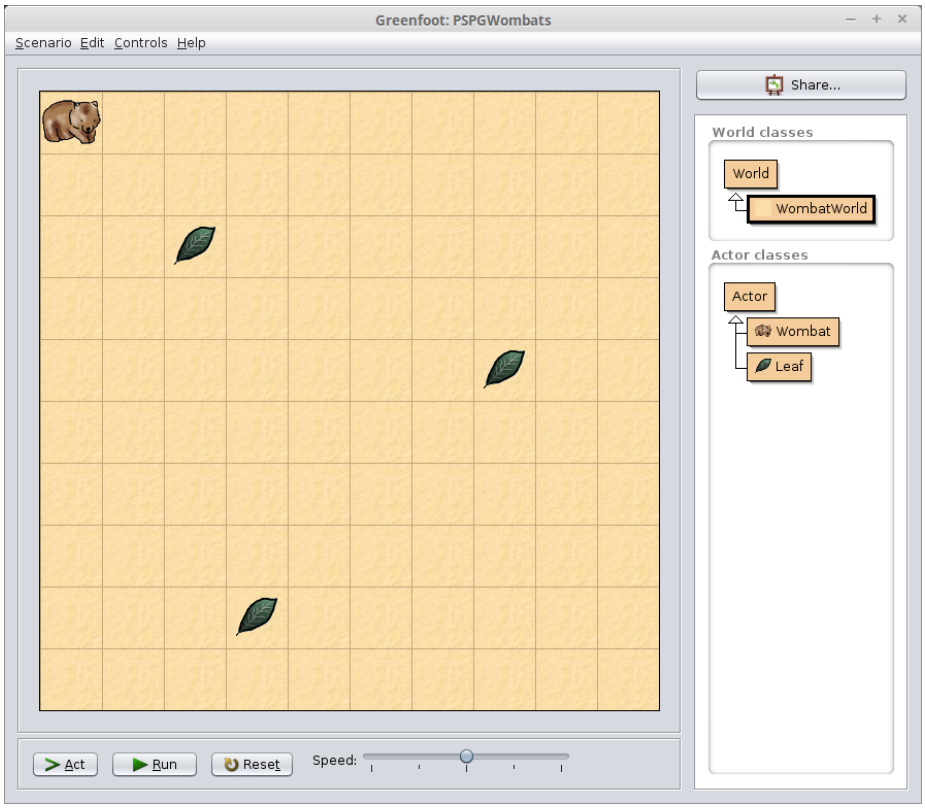

Figure 1.1: The Greenfoot GUI after opening PSPGWombats.

On top are the "World classes" and on bottom are the "Actor classes". The classes `World` and `Actor` are general and appear in all Greenfoot scenarios. The other classes below them, in this case `WombatWorld`, `Wombat`, `Rock`, and `Leaf`, are specific to this particular scenario.

In the way that Java works, classes are like blueprints and we can use these blueprints to build objects. One object of the `WombatWorld` class was automatically created and it is being drawn in the main display. The `WombatWorld` includes commands to create and add one instance of `Wombat` and the three instances of `Leaf`. We can also create more objects from the `Actor` classes and put those in the world. To see one way this is done in Greenfoot, right click on the `Wombat` class. You will see a pop-up menu with a first option of `new Wombat()`. Select this option and then move your mouse over the image of the world. You will see that there is now a wombat on your mouse pointer. Simply click someplace in the world to put your new `Wombat` object down. This process of making new objects is called *instantiation* and we often refer to the new objects that are created as *instances*.

Creating a lot of instances this way is inefficient, so Greenfoot includes a shortcut that can be handy when you want to make a lot of the same type of object. Move your mouse over the world display and press the Shift key. An instance of whatever class is selected will appear by your mouse pointer. Click to add it to the world. Try doing this with the `Leaf` class. If we do it a few times, we can create a world that looks like the image in Figure 1.2.

We now have a world with a number of `Actor` objects in it. The question is, what can we do with that? That actually depends on the `World` class as well as the classes of the various objects we have put in the world. Just like objects in the real world, objects in our programmed world have various characteristics and behaviors, which are programmed into them. The characteristics, often called PROPERTIES hold information about the objects or can represent memories the objects store. The behaviors are called METHODS and they define what an object can do or allow us to ask an object for information about it. To learn about the properties or methods of a particular object, simply right click on it. Right clicking on the background image where there aren't any `Actor`s gives you the options for the whole world (the `World` object, not all the options of the objects in the world).

Figure 1.3 shows the menu of options that you get if you right click on a `Wombat` object. Most of the menu is a list of the different methods you can call on an instance of `Wombat`. These methods are written in the notation and style of the Java language and as a result they might appear a bit odd at first. Nothing about them is very complex though. Notice that each method has at least two words with a set of parentheses. The first word tells us what type of information, if any, the method gives us back. The word `void` implies that no information is given back. The second word is the name of the method. So the method

```
void act()
```

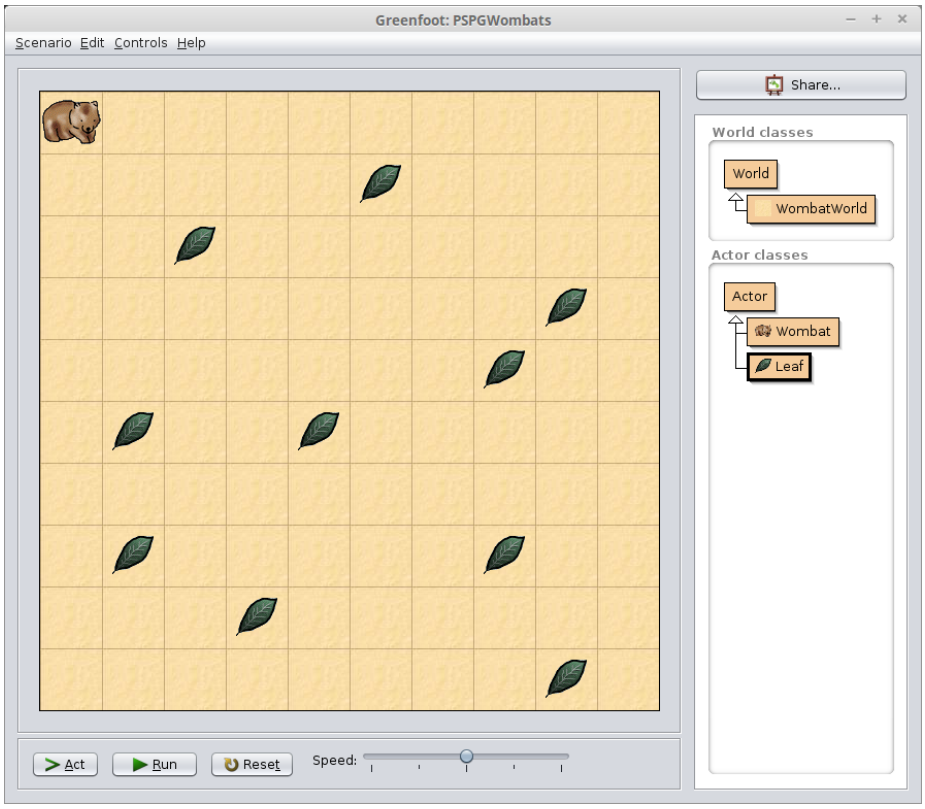

Figure 1.2: Initial setup for PSPGWombats.

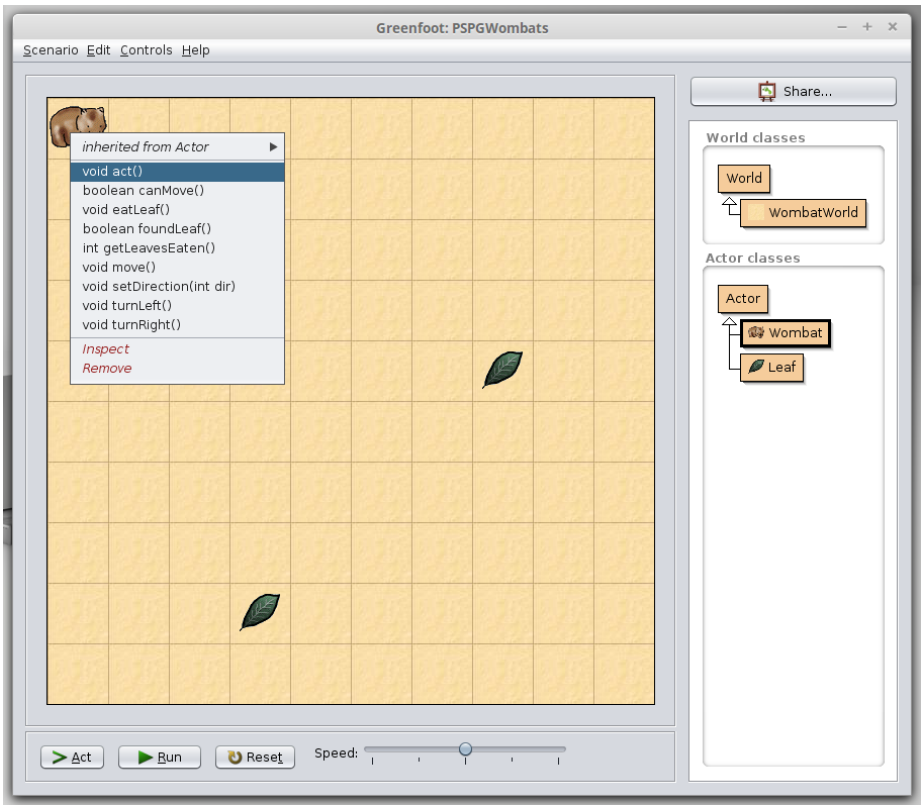

Figure 1.3: Pop-up menu for the Wombat.

tells the `Wombat` to act and doesn't give us back any information. The method

```
boolean canMove()
```

lets us ask the `Wombat` if it can move and it returns the answer to that question. Try calling some methods on the `Wombat` to see what happens.

One of the methods of the wombat,

```
void setDirection(int dir)
```

, is unlike any of the others in that it has something inside of the parentheses. As the name implies, this method lets us tell the wombat to face in a different direction. The thing inside of the parentheses, "int dir", is called a parameter and it allows us to give additional information to a method. In this case, we have to tell the wombat what direction we want it to face and we are passing in that information as an integer value. Try calling `setDirection` a few times and pass it some different values to see what happens.

At the end of the menu are two options in red that are not methods you can

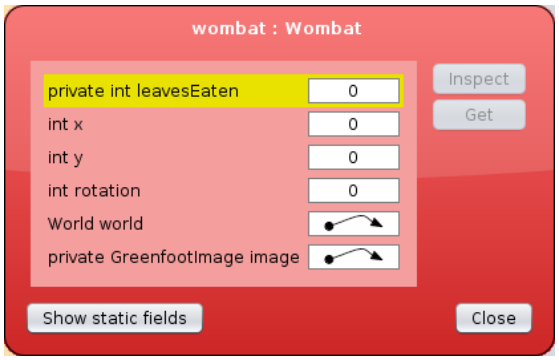

Figure 1.4: The object inspector for a Wombat.

call. These are Inspect and Remove. The "remove" option does what one would expect, it takes the object out of the world. The "inspect" option lets you see all of the properties of a particular object. Figure 1.4 below shows what we get if we select this option on our wombat. Each property of the wombat is shown on a row. It includes the type of the property, the name of the property, and the value of the property. Some properties also include the keyword `private` that we won't worry about yet. We can see from the figure that the wombat stores a rotation as an `int` and it has a value of 0. The type `int` is short for integer and can represent information that is whole numbers. The wombat stores three other `int`s as well, `leavesEaten`, `x`, and `y`. Below all of those we see two other properties called `world` and `image`. These are fairly self-explanatory and we will hold off on doing a full discussion of them until later. There are some other options in the Inspector that we will also ignore for the time being.

There are two other areas in the full Greenfoot view that we should talk about. Below the world display you find a set of controls with three buttons and a slider. The first button is labeled "Act". When this button is pressed, Greenfoot goes through every object in the world and calls each one's act method in turn. We saw above that our `Wombat` had such a method. That method is actually common to all objects of type `Actor`. Click this button and see what happens.

Next to the "Act" button is a button with the label "Run". When you click this button Greenfoot will basically do what happens with the "Act" button, but it will do it over and over again. How fast it does it depends on the setting on the "Speed" slider. Try clicking the "Run" button to see what happens. You can click it again to stop the acting.

Between the "Run" button and the slider is a button with the text "Reset". If you click this button, the world will go back to its original state. For the `WombatWorld` that means that you go back to having one `Wombat` in the top left corner and three leaves distributed around the grid.

Before we look at a specific problem, I want you to click the "Run" button and let the program run for a while. You should see the `Wombat` walk forward

and eat a `Leaf`, then go around the edge eating any leaves that are on the edge. The `Wombat` will then continue to walk around the edge until you click on "Pause". After you have watched the `Wombat` go around a few times go ahead and pause it. Now we want to call one last method on the `Wombat`. This time you should call the `getLeavesEaten()` method. Note that this method has a return type that isn't `void`. It returns an `int`. This makes sense because it should tell you the number of leaves this `Wombat` ate. What number does it tell you? Does this match with what you saw? Methods like this that have return types other than `void` are the way in which we can request information from an object.

1.3 First Problem

Now that you have some idea of how to use Greenfoot, we want to see how to solve a problem with it. As is fitting, we are going to solve a fairly simple problem. You probably noticed that the `Wombat` has a method called `eatLeaf`. If this method is called when the wombat instance is on the same square as a leaf instance, the leaf disappears and the wombat adds one to its total of leaves eaten. As it happens, the `act` method will call `eatLeaf()` after it moves. The problem I want you to solve is as follows: Given a setup like the one above, make the wombat eat all of the leaves.

If you clicked the "Run" button earlier you probably saw that our wombat is not all the intelligent. When its `act` method is called one of three things happens. It either eats a leaf, moves, or turns to the right. If it is on a leaf it will eat it. Otherwise, if it can move forward it will. If it isn't on a leaf and can't move forward, it will turn left. This simple behavior causes it to walk until it hits an edge, then go around the edge in a clockwise direction. This will get some leaves, but not all of them. What do you have to do in order to get the wombat to eat all the leaves? You should only use the "Act" and "Run" buttons or directly call methods on the wombat by right clicking on it. Do not drag the wombat or the leaves around to accomplish this.

Direct the wombat so that it eats all the leaves. You might consider writing down each of the steps you followed to make it happen. What instructions did you give to the wombat? What order were they given in? Congratulations! What you have written basically constitutes your first program, though it would likely need to be cleaned up some before it could be understood by a computer. Let's go over your solution to the problem and see how good a job you did.

How complete and accurate are your instructions? If you were to give this to another person, would they be able to follow your steps and get the wombat to eat all the leaves? What if the person in question were a three year old with point and click skills?

How flexible is your solution? Imagine if we added a few more leaves or moved some of the leaves around. Would your set of instructions still work? Technically you weren't asked to write something that would work on any set of leaves, only the set that you started with. However, one of the goals we strive

for when we are programming is to make our programs as general as possible. Even if the program as a whole is very specific, we often like the pieces we break it into to be general. That way we can reuse them later on. Writing general code can save you a whole lot of time in the long run.

Think for a minute about how you could go about making your instructions general so that they would work for any arrangement of leaves. Now remember that we want to give these instructions to a three year old. What things do you need to be able to say in your instructions? What ideas do you need to be able to express?

One last question about this problem: was the set of instructions you wrote optimal? To answer this we must first say what we mean by the word optimal. For this situation a fairly good meaning would be that we want to make as few method calls on the wombat as possible between the start of the program and when the last leaf is eaten. If you are clicking on buttons or using right clicks to call methods, this definition of optimal is actually quite practical.

Here again, you had not been told to find an optimal set of instructions, just some set. However, as we will see over time, one of the things computers can be very useful for is finding optimal solutions to problems. In this case you might have found a fairly good solution, but how could you be certain it was optimal? As a last point to consider, imagine being asked to make your solution both general and optimal. That is exactly what we will want to do with our programs most of the time.

1.4 A Second Problem

For our second sample problem we are going to need to use some scenarios that are not part of the standard Greenfoot install. You can download these from this book's website. There are three different scenarios that we are going to use: DotGame1, DotGame2, and DotGame3. Begin with DotGame. Download the file and open it. If you can, put it in the scenarios directory where Greenfoot is installed, that keeps things simple as all your scenarios will be in the same place. If you can't, just remember where you saved it so that you can load it in Greenfoot.

When you load the scenario you will be presented with a grid of colored dots like that shown in figure 1.5. One of them will have a box drawn around it. Click the run button and then "play" the game. It is a fairly standard theme where you have to line up colors. Make sure you understand what it is doing and how it works. You might consider dropping the speed some to make it easier to see what happens when you make moves.

Now that you have some idea of how this program works, load up DotGame2. This will start off looking pretty much identical. However, clicking the Run button on this version doesn't do anything. Unlike the last version, this isn't a functioning game. The `act` method, which we saw earlier and which is so important to making things happen automatically in Greenfoot, doesn't do anything in this version. Instead, with this version you have to right click on

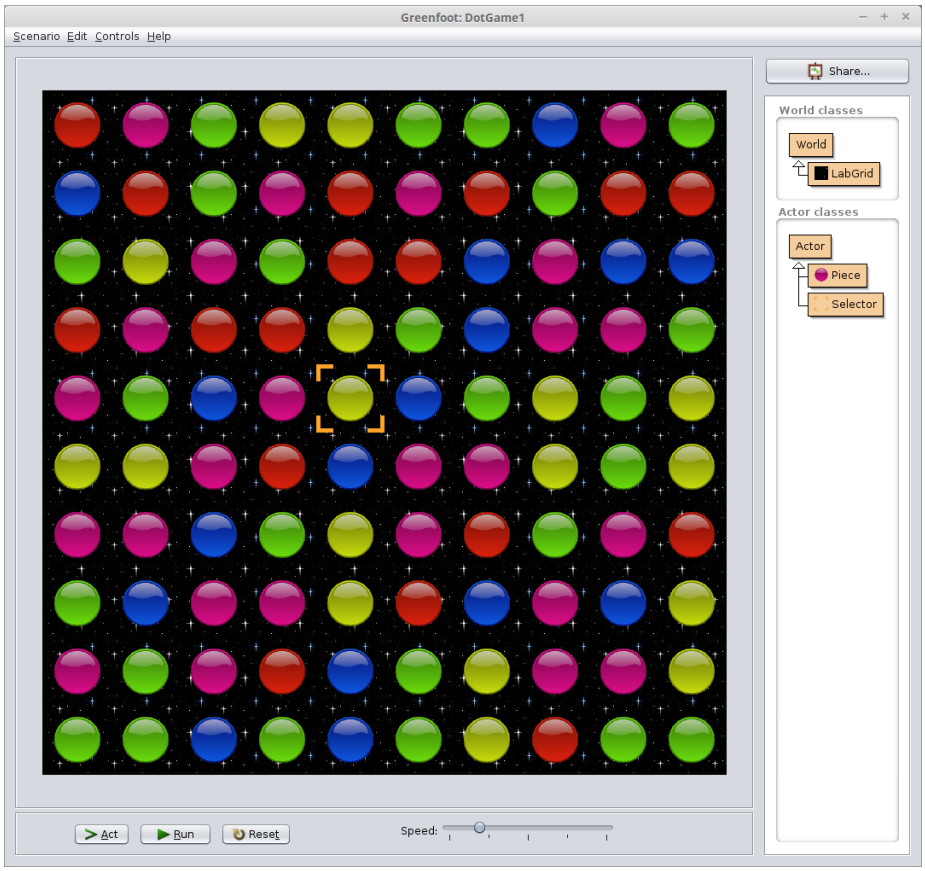

Figure 1.5: This is what might see when you run the dot game.

the world or actors in the world to make things happen. Spend a little time looking at the right click menus to see what different methods the world and the actors have. Once you feel that you have a grasp of them, try playing the game this way.

What methods do you have to call on what objects? What order do you have to call them in order to make things happen the way that they are supposed to? Take a few minutes and write down what you have to do in order to complete one "move" in the game. Pay attention to the order of what you have to call and how many times it has to be called. Some methods need extra information passed in as parameters. Which are those and what values do you need to pass in?

Now open up the third version. Do exactly what you did for the second version. Look at the methods that are available and figure out how to play the game. What has happened between the second and third versions? What does it look like now if you write down the set of instructions that you have to call in order to make the game play?

The same basic thing is happening when you go through and play the game in all three versions of this program. The difference is that in the first version all of the instructions have been put into the program. In the second version, some of those instructions were taken out. Some of the methods were basically removed. As a result of this, you have to go in and do manually what the earlier code was doing. In the third version, even more functionality was removed and you have to manually make calls on different methods had been called by the methods that was taken out.

What you are seeing here is a specific example of how problems are broken down into pieces. Even the steps in the third version are at a fairly high level. They are composed of smaller steps that deal with individual pieces in the game. As a last step, take the instructions you wrote down for version 2 and compare them to the instructions for version 3. What do you notice when you do this?

So what was the point of this exercise? The first objective was to get you to think through solving a problem in a systematic, step-by-step way. If you went through and really played the game in each scenario and wrote down what methods you had to call, with what parameters, and in what order, then you will have accomplished that objective. There is another objective as well. Unlike the wombat moving around and eating leaves, this example clearly has different levels of complexity. You can see how the problem breaks into a few high-level steps and then explore how each of those steps can be further broken down. You should have seen how a single method call that you made in version 2 gets split out into many different calls in version 3. Further, you can hopefully picture in your head how each of the calls in version 3 would be further broken down.

This approach to problem solving, where we begin with the full problem and break it into a few large pieces, then break down each of those pieces, and so on is often called top-down design. This is in contrast to bottom-up design where you start with certain basic operations that you know are going to be significant in solving a problem and put them together into successively larger pieces until you get a solution for the whole thing. In real-world problem solving we often

use a combination of these two approaches, using whichever fits better at any given time.

Another key aspect to notice about solving these problems is that the order in which you do things is very significant. If you alter the order of the moves for your wombat, say put all the turns first and the moves after them, clearing your aren't going to be able to eat all of the leaves. The same thing is true for the dot game scenarios. For example, calling `addAtTop` before you call `dropPieces` produces a a different end result.

Exercises

1. Don't know. !!!
2. There were lots of questions in the chapter that I could refer back to and make into formal exercises.
3. There were several things we discussed about at lunch at Freddies one fine day. Did you make note of them somewhere?

Chapter 2

Basics of Programming

What we did in the last chapter was helpful for getting familiar with Greenfoot and it was illustrative of the type of thinking that we do when we are programming, but it was not programming. Technically, we haven't yet seen any of the Java programming language. In this chapter we will start to learn Java and really program.

2.1 Looking at Code

Let's start Greenfoot and open up the PSPGWombats scenario again. To look at the Java code[1] that defines a particular class in Greenfoot you can either right click on the class box on the right side of the Greenfoot display and select "Open editor", or you can double click on the class. We want to look at three different classes in this scenario and we will move in order of increasing complexity beginning with the `Leaf` class.

When you bring up the editor for `Leaf` you should get a window that looks like Figure 2.1. If you recall, the `Leaf` doesn't really do much of anything, and perhaps now you can see why, there isn't much to this class. Let's go through this step-by-step to see what we have. Lines 9-19 actually contain the class itself. We know this because on line 9 it tells us that it is declaring class `Leaf`. Notice how some of the words are colored. Most programming editors do this for you because it helps you follow code and also to see when you have made an error. The words that are colored are keywords that have to be spelled and capitalized properly. If you mistype one it won't be colored in properly which

[1] The term "code" carries the connotation of being hard to read or understand as it is a term that is often used for cyrptography. In that sense, the fact that it is used for programming languages is unfortunate because programming code is not intended to be cryptic. Indeed, software developers often stress the importance of readable code that can be understood by any developer who tries to read it. Similarly, code that is hard to understand is generally looked down on and whole programming languages are derided as "write-only" if they tend to be used to create hard to read programs. Java is a very readable language, but the term "code" is still used as a general noun for text written in it.

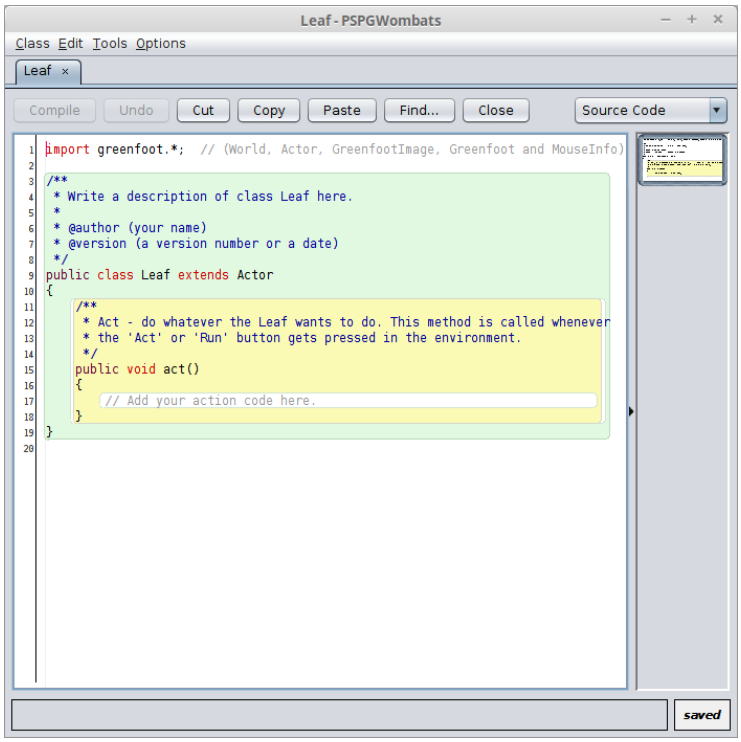

Figure 2.1: The Greenfoot editor window showing the Leaf class.

is easy for you to see. The Greenfoot editor also colors regions of the code as blocks to help you visually see the structure.

We will go through what each piece of this code means, but first we get to see our first "rule" for Java code. These will appear frequently in the text. They give you a fairly formal specification for how something should be written in Java. The text in italics refer to other rules that appear elsewhere in the chapter. Things that are in square brackets are optional. This rule is actually a simplification of the full rule for the Java language. It has been simplified to a form that better fits our needs early on. For that reason, this rule, along with some others in this chapter, will be changed later to provide more complete specifications.

Class:

`public class` *Name* `[extends` *ReferenceType*`]` {

`[`*Constructors*`]`

`[`*Methods*`]`

`[`*MemberData*`]`

}

The keywords on line 8 are `public`, `class`, and `extends`. They also appear in the first line of our rule. The keyword `public` is discussed below with its usage on line 15. The `class` keyword just tells Java that this is a class. The name of the class, `Leaf`, follows the keyword. The `extends` keyword implies something called inheritance and the name of the class we are inheriting from, in this case `Actor`, follows it. Inheritance has a number of different implications, but for now all you need to know is that this says that the `Leaf` is an `Actor`. This is what we want because anything we want to put into the Greenfoot world has to be an `Actor`. Inheritance will be discussed in detail in chapter 4.

Lines 10 and 19 have curly braces on them. Because the open curly brace on line 10 follows the class declaration, and the close curly brace on line 19 matches up with it, these two signify the range of everything that is in the class. Java uses curly braces for this purpose in many context. Recall the outline format that was discussed for instructions in the previous chapter? The curly braces are the formal way that we tell Java that certain things are grouped together in a certain *scope*. Note that lines 11-18 are indented, just like they would be in an outline. This isn't required. It turns out that Java doesn't pay attention to spacing, indentation, or even new lines, but indentation makes code much easier for humans to read. For this reason, all the code in this book will be indented such that nested lines are tabbed in one level beyond the code that it sits inside of. It is highly recommended that you do this for your own code as well. While Java will understand the code with or without indentation, you will find in time that your ability to understand it is greatly enhanced if everything is properly indented. The color coding in Greenfoot also indicates scope. In figure 2.1, the light green box is around everything that is associated with the `Leaf` class, including comments above it. The yellow box shows everything associated with the `act` method.

Lines 15-18 represent the only things in this particular class. They define a method called `act`. You saw the `act` method in the last chapter when we called it on the wombat. When you pressed the "Act" button you noticed that the leaves didn't do anything. Now you can see why. The `act` method is blank. Once again we see the curly braces as well. In this case, nothing is nested inside of these curly braces. The word `public` appears here again as well as a new keyword, `void`. At the end of the line, the word `act` is followed by parentheses. Here is a basic rule for a method. This `act` method isn't using much of it.

Method:

`public` *Type name*`(`[*Type1 name1*[`,`*Type2 name2*, ...]]`)` {
    [*Statements*]
}

We want to go through this one piece at a time and we will start with `public`. The `public` keyword tells Java that something is open and available to any code. To understand what this means, think of a television set. The main use of a television is watching, but you also have to occasionally tell it to do something. What can you tell a television? Well, turning on and off is a big

one. Changing channels and/or input sources and changing the volume are also common options. Most modern TVs also have a number of other settings. Each of these things can be done by pressing a button or a set of buttons either on the TV itself or on the remote of the TV. There is a lot of stuff hidden inside the TV as well. In theory you could change a lot of different things if you took off the case and started poking around. Of course, odds are good that if you did so you would break something and the TV would never work quite right again.

The buttons on the TV and the remote are the public elements of the TV. They are the things that anyone else is allowed to interact with. They are public in part because they are safe. The makers of the TV only exposed things to you that won't give you the power to break the TV. All the stuff hidden away inside of the TV is private. We will see shortly that `private` is also a keyword in Java and, just like the TV, is signifies things that other people should not be messing with, often because messing with it could put an object into a state in which it doesn't work well.

After public on line 15 is the word `void`. If we look at our rule for methods, this is in the place called *Type*. This is the return type of the method. It tells us what type of information the method gives back to us when it is called. The type `void` indicates that no information is returned. Methods with a `void` return type simply do something, they don't return any information to us. This makes sense for the `act` method. When we call `act`, something should happen, but in general `act` doesn't need to tell us anything when it is done. The rule for *Type* is below. It uses a new symbol for the rules. The | symbol, called a pipe, is like the word "or". It tells us that one thing or another can go in a certain place. Looking at these rules we see that for `act` *Type* is a *PrimitiveType*, and it is using the last option under *PrimitiveType* of `void`.

***Type*:**

PrimitiveType | *ReferenceType*

***PrimitiveType*:**

`boolean` | `char` | `int` | `double` | `void`

The type void can only be used as the return type of a method, not for a variable declaration.

***ReferenceType*:**

This is the name of a class. It can be one that you have written or one that is in an library. Class names typically start with a capital letter and have each new word capitalized. These are the normal rules for Java names.

Both the class and the method have referred to another rule called *Name*. When this appears it simply expects a valid Java name. The rule below presents the options you have for making Java names. While the rule is very flexible, you typically won't use all of that flexibility, nor would you want to. It is best to give things names that are easy to remember and that have meaning. That way when you come back to the same code you will be able to more easily

understand it. Java also distinguishes between upper and lower case letters so the names count, COUNT, and CoUnT are all different. It is generally not advisable to create names that only differ in their capitalization as it can get very confusing. There is something of a standard that people try to follow when it comes to naming in Java. Class names begin with a capital letter. The names for non-types, like methods and variables, which we will see later, start with a lower-case letter. In both cases, the first letter of the second, third, etc. word in the name is capitalized. This style is given the name camelcase because the capital letters on each word look like humps on a camel. All the examples in this book will adhere to that style.

***Name*:**

> Java is fairly flexible about naming. Valid Java names have to start with a letter or underscore followed by any number of letters, underscores, or numbers.

The parentheses after the word act on line 15 indicate that this is a method. From the *Method* rule we can see that it is possible to put parameters inside of the parentheses. We will see these a lot more later. One of the things you will come to learn about programming languages is that they often have a lot of punctuation [2] and they are very picky about it. So as we go through looking at various pieces of code, pay particular attention to the placement of punctuation.

What about the lines before line 8? The first line has the word `import` in red followed by `greenfoot.*;` and then some stuff in gray following two slashes. The `import` statement tells Java where to go looking for things. In this case, it is telling Java that if it ever sees something it doesn't know about, it should go look in the greenfoot library. This makes sense as this class is part of a Greenfoot program. Most of our classes that we write in Greenfoot will have a line like this at the top.

And what about the stuff in gray after the double slashes? That is a comment. The double slashes signify a single line comment in Java. Anything following them on that line is ignored by the Java compiler, so you can put whatever you want in comments. Lines 3-8 are also comments. These are multi-line comments that begin with `/*` and end with `*/`. The fact that this one begins with `/**` just means that this is a special type of comment and is a detail that we won't worry about for now. The compiler doesn't care. As soon as it sees the `/*`, it ignores everything until it gets to a `*/`. For this reason, you will never see a comment appear in any rules. They could go pretty much anywhere except in the middle of a word.

Comments can be a vital part of programs. They are typically used as documentation to help make it clear what a piece of code is doing. There are different philosophies on how much and when comments should be used though. They range from very minimal use of comments to extensive commenting that

[2] Some languages have less punctuation, but they typically have keywords in place of the punctuation. For example, the Pascal language used `begin` and `end` in basically the same way that Java uses { and }.

documents all aspects of a piece of code. The logic behind the choice of how to comment will be discussed in various parts of this book after you have a stronger foundation in programming to understand them. As a beginner though, you should probably adopt the commenting strategy that is accepted by the person giving you your grade.

Before we move on to another class, you should look at the editor window shown in Figure 2.1. In addition to the menus, there are a set of buttons that include commonly used editor tasks such as cut and paste and a button to compile this file. You might want to familiarize yourself with the options in the menus. Some of them can help to make your life much easier when we get to the stage of writing code.

So that was the `Leaf` class. It was simple. It didn't really do anything. That is what made it a good place to start. Now we move on to the `WombatWorld` class. This class is a fair bit longer. We will just look at the code and not view it as an image in the editor. It starts off much the same as `Leaf`. There is an `import` statement of `greenfoot.*` at the top, a comment, and a public class declaration. While the `Leaf` class inherited from `Actor`, the `WombatWold` class inherits from `World`. This means that `WombatWorld` is a `World`.

```
1  import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and
       MouseInfo)
2
3  /**
4   * Write a description of class MyWorld here.
5   *
6   * @author (your name)
7   * @version (a version number or a date)
8   */
9  public class WombatWorld extends World
10 {
11
12     /**
13      * Constructor for objects of class MyWorld.
14      *
15      */
16     public WombatWorld()
17     {
18         // Create a new world with 600x400 cells with a cell size of 1x1
               pixels.
19         super(10, 10, 60);
20         addObject(new Wombat(), 0, 0);
21         addObject(new Leaf(), 2, 2);
22         addObject(new Leaf(), 3, 8);
23         addObject(new Leaf(), 7, 4);
24     }
25
26      /**
27      * Populate the world with a fixed scenario of wombats and leaves.
```

```
28      */
29     public void populate()
30     {
31         removeObjects(getObjects(null));
32         addObject(new Wombat(),3,3);
33         addObject(new Leaf(),4,0);
34         addObject(new Leaf(),1,1);
35         addObject(new Leaf(),7,1);
36         addObject(new Leaf(),4,3);
37         addObject(new Leaf(),1,5);
38         addObject(new Leaf(),5,6);
39         addObject(new Leaf(),7,7);
40     }
41
42     /**
43      * Place a number of leaves into the world at random places.
44      * The number of leaves can be specified.
45      * @param howMany The number of leaves to add.
46      */
47     public void randomLeaves(int howMany)
48     {
49         for(int i=0; i<howMany; i++) {
50             Leaf leaf = new Leaf();
51             int x = Greenfoot.getRandomNumber(getWidth());
52             int y = Greenfoot.getRandomNumber(getHeight());
53             addObject(leaf, x, y);
54         }
55     }
56 }
```

The `WombatWorld` class really has stuff inside of it. To be specific, it has a constructor and two methods inside of it. The constructor is something new that we didn't see in the `Leaf` and a rule for it is presented below. In general, constructors are very much like methods. The main differences are that they don't have a return type, not even `void`, and they have to have the same name as the class they are in. So in this case the constructor has the name `WombatWorld`.

***Constructor*:**

`public` *ClassName*`(`[*Type1 name1*[,*Type2 name2*, ...]]`) {`

[*Statements*]

}

The purpose of the constructor is to set up values the way we want them when an object of this class is created. The constructor is also the first place where we have seen a statement inside of the curly braces. In this case this is a special statement using the keyword `super`. We won't get into the details of `super` at this point, but you should know that this line of code makes it so that our `WombatWorld` is an 10x10 grid with each grid being 60x60 pixels.

It should be explicitly mentioned at this point that you should not worry if everything you see doesn't make sense right now. We are going through existing code to give you a feel for what you can put into Java code and what you can do with it. Some features you aren't meant to fully understand yet. Shortly we will start writing our own new code. When we do that it will be done at a pace where each step should be explicit and completely understandable. Making everything work the way we want in PSPGWombat simply invokes a few elements that we won't get into the full details of for a few chapters.

The other two methods in `WombatWorld` are more interesting. The first one, `populate`, sets up the world in a configuration with more leaves. It has nine statements in it. The rule for a statement is given below and shows that there are several different options for statements. All the statements in `populate` happen to be method calls so that is the only specific statement rule that is presented. The others will wait for later. One thing to note about statements is that they tend to end in a semicolon. The one exception to that rule is the CodeBlock. Remember that Java doesn't pay much attention to white space, including new lines. Just like a sentence in English can be spread across multiple lines and only ends when you get to the proper punctuation mark, statements in Java can also span multiple lines and only end when you get to the semicolon. Unlike English though, we will typically put one statement on a line in Java and will only split statements across lines if they happen to get really long.

***Statement*:**

VariableDeclaration | *Assignment* | *MethodCall* | *ReturnStatement* | *CodeBlock*

***CodeBlock*:**

{ [*statements*] }

Any place a single statement is expected in Java, it can be replaced by multiple statements inside of curly braces. They don't all have to be on the same line as I have typed it here.

***MethodCall*:**

[*objectExpression*.]*methodName*([*expression1* [, *expression2*, ...]]); |

ClassName.methodName([*expression1* [, *expression2*, ...]]);

The *objectExpression* is any expression whose type is an object type. If no *objectExpression* is given, the method is called on the current object. The *methodName* must be a valid name of a method on that type. The expressions in the arguments are often just the names variables, but it can be a method call that returns an object.

The second format is used for `static` methods. For these we don't need an object, we just put the name of the class in front of the dot. The `Greenfoot` class has only static methods in it so you call them with a format like `Greenfoot.getRandomNumber(3);`.

The first statement in `populate` calls `removeObjects` and the other eight statements call `addObject`. `removeObjects` takes a single argument that is all the `Actor`s to remove from the world. In this case, we use `getObjects` with an argument of `null` to get all the `Actor`s in the world. There are three different arguments passed into `addObject` in all of these calls. The first is a new object that is an `Actor`. The second and third are numbers, integers to be more specific. The meaning of `addObject` is fairly clear. It is supposed to add an object into the world. Note how it uses the camelcase naming scheme, and because it is a method, it starts with a lower case letter. The meaning of the integer arguments that are passed in isn't quite as clear. They happen to be the location that we want the object to be placed at. The x-coordinate is first and the y-coordinate is second. How do we know that though and how can we find out what other methods we might be able to call? The solution to these problems is the API. That stands for Application Programmers Interface. What it really means is that this is the help documentation that tells you all the methods and other things that are in particular classes.

As we are programming in Greenfoot, methods like `addObject` can be found in the Greenfoot API, which you can find under "Documentation" at the Greenfoot web site (`http://www.greenfoot.org/`). There only seven classes in the Greenfoot API. This is one of the best things about Greenfoot, the API is rather simple. So which one of those seven classes is the `addObject` method in? If you read the rule for *MethodCall* and compare it to the calls to `addObject` you see that because we don't have `something.addObject` that these calls are being made on the current object, which, because we are in the class `WombatWorld`, has type `WombatWorld`. Looking at the `WombatWorld` class we see only three things, and `addObject` is not one of them. The solution is at the top of the class where it says `WombatWorld extends World`. Earlier we said that this implies that `WombatWorld` is a `World`. One implication of this is that `WombatWorld` implicitly has all of the methods that are part of `World`. Figure 2.2 shows a screen shot of the `World` API page and sure enough, there is an `addObject` method in it.

The `addObject` method is the second method in the method summary table. To the left of the method name is listed the return type of the method. This says that `addObject` returns `void`, which means that it doesn't give us back any information. In the parentheses to the right of the method name you find the argument list. This tells us that we are supposed to pass three things into `addObject`. The first is something of type `Actor` which in this scenario means either a `Wombat` or a `Leaf`. After that we see that we are supposed to pass in two integers, `x` and `y`. The API makes it clear what we are passing into the calls to `addObject` and what each one means. The API is a very helpful tool and we will be referring back to it frequently.

There is only one more thing in populate that we haven't explained. The first argument to each call of `addObject` starts with the keyword `new`, followed by a class name and parentheses. The `new` keyword tells Java that we want to make a new object. The class name is the name of the class that we want to make a new one of. The parentheses are required and can have arguments in

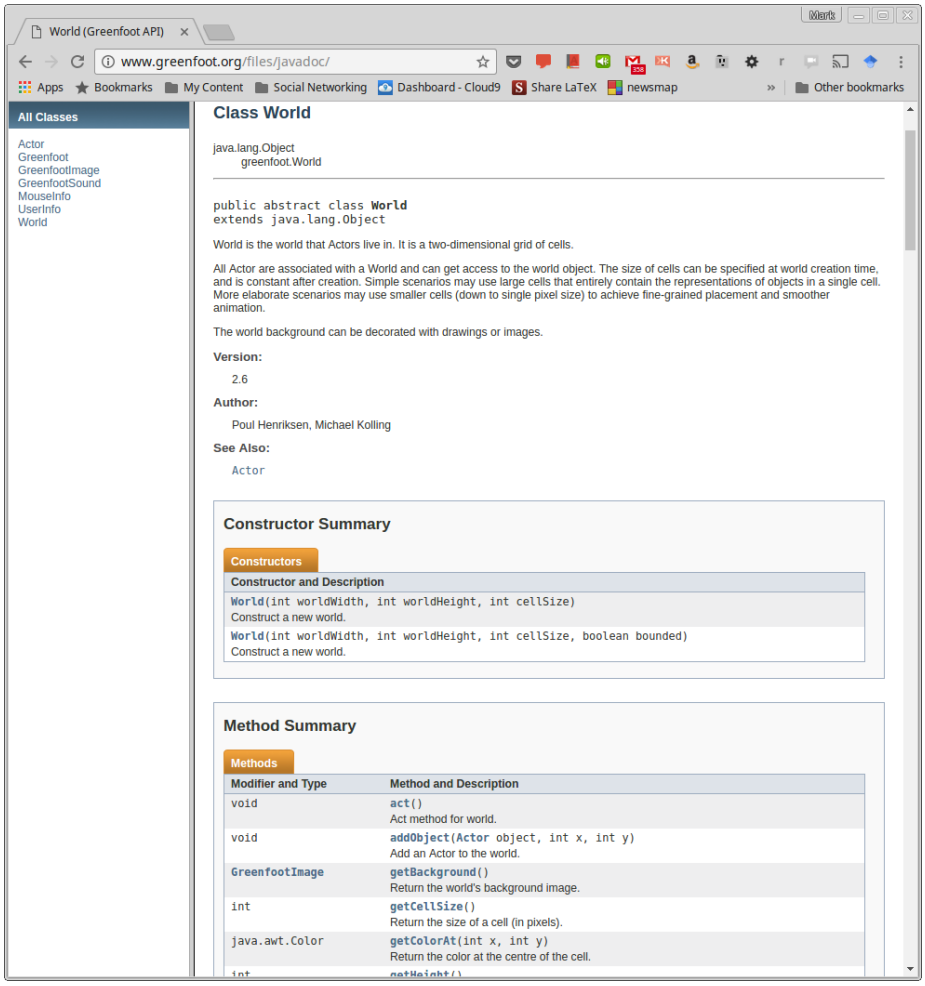

Figure 2.2: API page for the World class.

them. Using `new` is what actually calls constructors like the one we talked about earlier. If our constructor had parameters that it takes in, then we would need to provide corresponding arguments when we used `new`.

The last method in `WombatWorld`, `randomLeaves`, is a bit more complex and contains some elements we won't talk about for a few chapters. In particular, the `for` loop won't be discussed until chapter 4. At this point it is sufficient to say that it allows us to tell Java to do something multiple times. Unlike earlier methods, the `randomLeaves` method takes in a parameter. When we call `randomLeaves`, we have to provide it with an integer telling it how many leaves we want to add. This value is stored in the variable with the name `howMany`.

Inside of the `for` loop there are three other variables that are declared so at this point it is probably worth taking a minute to talk about variables and introducing two of the other statement types. Variables in programming are conceptually very similar to their counterparts in math that you learned about in algebra. They are names that we can use in an abstract way to refer to values. They are not perfectly analogous to math variables however because programming variables can change their values as a program runs. A good mental image for a variable is to think of it as a name that we put on a box that can hold a value in it. We are able to look in the box to see the value, or change the value that is in it.

***VariableDeclaration*:**

Type name[=*expression*];

This declares a variable of the given type with the specified name. You can think of variables as boxes that can hold a value. The box is labeled with the name we give it and it can only hold things of the proper type. What you can have for type is described after all the statement types are introduced.

A variable can only be used in a certain SCOPE. The scope of a variable runs from the point at which it was declared down to the close curly brace that ends the code block it is declared in. Consider the following example method.

```
public void foo() {
    int a=5;
    {
        int b=8;

        // Point 1
    }

    // Point 2
}
```

At point 1 you could use either `a` or `b`. However, the scope of `b` is only inside the inner curly braces so you can't use `b` at point 2, only `a`.

***Assignment*:**

name=expression;

This stores the value of an expression in the variable with the given name. The variable must be in scope. So it either has to be declared in the current method above this line or it should be a class level variable. In order for this to work, the types need to match.

Here are some example variable declarations and assignments that could be made from them.

```
int a;
int b;
double d; // This type will be discussed later in the chapter.
String s; // This type will be discussed later in the chapter.

a = 6+7;
b = a-4;
d = 2*3.14159;
s = ''This is a test.'';
```

Another concept that is very important to variables in programming is TYPE. We have already seen the rules for types and have had it pop up in other contexts. The first word in a variable declaration with our current rule is the type of the variable we are declaring. This is significant, because variables can only store values of the correct type. The first variable declared inside of the `for` loop has the type `Leaf` and the name `leaf` while the other two both have type `int` and names `x` and `y`. If you were to try to put an integer value in the variable `leaf` you would get an error. Similarly, if you tried to put a value of type `Leaf` in `x` or `y`, you would also get an error. We will see a lot more about how this matters over time.

At first it might seem confusing to have a type `Leaf` and a variable called `leaf`. This is one case of having two names that only differ by capitalization that is fairly common. This particular usage doesn't cause confusion as long as you follow standard naming schemes because any name that starts with a capital letter will be a type while those that start with lower case letter will be either variables or methods.

All three of the variable declarations in `randomLeaves` include the optional initialization. That is to say that the variable name is followed by an equal sign and an expression of the proper type. This puts an initial value into the variable. In the case of `leaf`, the initial value is simply a new instance of `Leaf`. For `x` and `y` the expressions are a bit more complex. The expressions for `x` and `y` use method calls to get values. Both have a call to a method named `getRandomNumber` that is in the class `Greenfoot`. Take a minute to go look in the API at the `Greenfoot` class. This class is called a utility class. All the methods in it are `static` which means that we call them on the whole class, not on some instance of the class. Basically, you will always write `Greenfoot.methodName(...)` when you call methods from the `Greenfoot` class.

The `getRandomNumber` method takes a single argument that is an `int`. In this case, we want to get numbers between 0 and the size of the world. In the constructor we saw that this world is made to be 8x8. As such, this code could have called `Greenfoot.getRandomNumber(8)`. However, it is generally considered bad form to put in values like this that appear as "magic numbers". If we put in the value 8 and later wanted to change the size of the world, we would have to find all of the 8s in the code and change them appropriately. Instead, the `World` class has methods in it called `getWidth()` and `getHeight()`. These return to us the width and height of the world. So if we alter the size in the constructor, this code will continue to work. This should remind you of the concept of generality that was discussed in the first chapter some. Using the number 8 would have made the code overly specific. By using these methods the code is made more general. That benefits us later because we can make changes to other things without having to change this code as well.

Assignment statements are also introduced in the rule above. They change the value of a variable that has already been declared. When a method is executed, the lines in that method are executed in roughly top-down order.[3] If one line declares a variable and gives it a value, it will have that value until we get to another line that does an assignment and changes the value. The variable will then contain that new value until another assignment statement is reached.

***Expression*:**

An expression in Java is anything that has a type and value. Below is a list of options.

- Variable name – The name of a variable evaluates to the value currently stored in that variable.
- Numeric literal – Numbers are expressions. You can represent integer values or decimal fractions as you normally would. Very large or very small numbers are written using scientific notation of the form 1.234e56 which is the number $1.234 * 10^{56}$, a rather large number indeed.
- String literal – String literals can be any sequence of characters enclosed in double quotes. For example, "Hi mom." is a string literal.
- null – This is the value for any object type when it doesn't refer to a valid object.
- Method call – A method call for any method that returns something other than `void` is an expression.
- Operator expression – Java supports different mathematical and logical operators. Most will join two different expressions. Some operate on one expression and one uses three expressions. This is a list that includes most of the ones we are likely to use.

[3]Some statement types, like the `for` loop, can alters this a bit and cause code to repeat statements or skip over some statements, but for normal statements, the way the program runs is to do the steps in order one at a time.

- A+B : Standard addition for numeric values. If either A or B is a `String` the plus sign will do string concatenation.
- A-B : Subtraction for numeric values.
- A*B : Multiplication for numeric values.
- A/B : Division for numeric values.
- A%B : Modulo operator that returns the remainder after division for integer values.
- A=B : This is the assignment we saw above. It is also an expression which takes on the value that is being stored. In this case, A needs to be the name of a variable while B can be any expression of the right type. The whole expression has the value that B evaluates to.
- `new` *ClassName*`(`*args*`)` : The `new` operator is used to make objects. This will make a new object of type *ClassName* using the constructor that is matched by the arguments passed in.

The last line in the `for` loop in `randomLeaves` calls the same `addObject` method we saw earlier to add the new `Leaf` into the world at the randomly generated location. You should try calling these two methods in Greenfoot to see what they do. You could also try altering the constructor to have a different sized world and see what happens. Do you notice any interesting behaviors? We talked about the fact that `randomLeaves` is general. Is populate? Why or why not?

The last class we want to look at in the PSPGWombats scenario is the `Wombat` class itself. The code for it is shown below. This class contains a lot more methods than the `WombatWorld` class did. We won't bother with going through each and every one here in the text though you should try to look through them and see if you understand what they are doing. There are a few new features that are worthy of discussion and we will go through those.

```
1  import greenfoot.*; // (World, Actor, GreenfootImage, Greenfoot and
       MouseInfo)
2  import java.util.List;
3
4  /**
5   * Write a description of class Wombat here.
6   *
7   * @author (your name)
8   * @version (a version number or a date)
9   */
10 public class Wombat extends Actor
11 {
12     private int leavesEaten = 0;
13
14     /**
15      * Act - do whatever the Wombat wants to do. This method is called
            whenever
```

```
16      * the 'Act' or 'Run' button gets pressed in the environment.
17      */
18     public void act()
19     {
20         if(foundLeaf()) {
21             eatLeaf();
22         } else if(canMove()) {
23             move();
24         } else {
25             turnRight();
26         }
27     }
28
29     public void move()
30     {
31         move(1);
32     }
33
34     public void turnLeft()
35     {
36         turn(-90);
37     }
38
39     public void turnRight()
40     {
41         turn(90);
42     }
43
44     public boolean foundLeaf() {
45         return getOneIntersectingObject(Leaf.class) != null;
46     }
47
48     public void eatLeaf()
49     {
50         Actor leaf = getOneIntersectingObject(Leaf.class);
51         if(leaf != null) {
52             getWorld().removeObject(leaf);
53             leavesEaten++;
54         }
55     }
56
57     public boolean canMove() {
58         int x = getX();
59         int y = getY();
60         move();
61         boolean ret = x != getX() || y != getY();
62         setLocation(x,y);
63         return ret;
64     }
65
```

```
66      public void setDirection(int dir) {
67          setRotation(dir*90);
68      }
69
70      public int getLeavesEaten() {
71          return leavesEaten;
72      }
73  }
```

The first line inside of the `Wombat` class provide us with something new. It look a lot like the pattern for variable declarations and indeed, that is very close to what it is. This line declares a class MEMBER VARIABLE. It is also the first place we have seen the keyword `private` which was alluded to in the discussion of `public`. The rule for member data is shown below.

***MemberData*:**

`private` *Type name*[`=`*expression*]`;`

This defines a class level variable with the specified name and type. The initial value doesn't have to be specified.

Constructors and methods have a sequence of statements inside of curly braces. As a general rule, curly braces are used in Java to group lines of code. When a method is called in Java, the statements in that method are executed one at a time in order from the top of the method to the bottom. At this point we will only consider five types of statements in Java. So any place that you can have a statement, you can have one of these five. More statement types will be introduced later on.

Here is a simple little example of a class. This one doesn't do much, but it demonstrates the three things that you can put inside of a class.

```
public class SimpleClass {
    private int value;

    public SimpleClass(int v) {
        value = v;
    }

    public int getValue() {
        return value;
    }
}
```

So what is the real difference between the variables we saw earlier, which are often called LOCAL VARIABLES, and the member variables we are seeing here? The difference is something called SCOPE. The scope of a variable determines the region of code over which it can be used and how long it is kept around. Local variables exist inside of the method that they are declared in. Technically, a local variable can only be used from the point at which it is declared down to

the end of the code block, denoted with curly braces, in which it was declared. Any attempt to use it outside of that region will be an error. What is more, the variable ceases to exist when you exit that scope and a new one is created when you get back to the declaration point. One of the big implications of this is that a local variable declared in one method can't be used in another. You can declare a variable using the same name, but it will get a separate block of memory. It is somewhat like having two people with the same name. Sharing the same name doesn't make them the same person.

Member data also has a scope inside of the curly braces it is declared in. Since member data is declared outside of any method, and it inside the curly braces of the class as a whole, that is the scope that it has. You should think of member data as the memories of objects of the given class. In this case, every instance of `Wombat` remembers how many leaves it has eaten as an integer value. Member variables are initialized by default. Because these are numbers, they are set to be zero. There is also code in the constructor for `Wombat` to explicitly initialize those values.

If you look through the class code some you will see that `leavesEaten` appears in two different methods. This fits with the fact that member variables have a scope that goes through the full class. It begs a question though. When should a variable be a member variable and when should it be local. The general rule that we will follow is to make variables have the smallest scope that they can. So while all variables could be made into member variables, doing so generally leads to problems and is considered poor style. We will only make variables into member variables when they are used across multiple methods and the value needs to be remembered from one method to another, or across multiple calls to a single method.

So what other things are new in the `Wombat` class? Obviously it is longer, and it has a bunch of different methods in it. This is an example of problem decomposition. Look at the `act` method, it is right below the constructor. It uses the `if` construct that we won't go into until next chapter, but you should be able to read it and figure out what it is doing. The `act` method doesn't really do anything directly. Instead, it has calls to five other methods that it could make, depending on the situation, and those other methods do the work. The names for those methods are straightforward and easy to read which makes the `act` method itself easy to read. It also means that when the `act` method was written, we didn't have to figure out how to do everything. Instead, we broke it into a few key steps, then went and wrote methods to accomplish each of those key steps.

The first method that `act` calls is inside of parentheses on the first `if`. It is the `foundLeaf` method. This method shows us two other new concepts that we want to look at briefly. Both new concepts stem from the fact that this is the first method we have seen that has something other than `void` as the return type. In this case, the return type is `boolean`. We will go into the details of `boolean` in the next chapter. The simple explanation is that it is a type that is either `true` or `false`. Having a return type other than `void` forces this method to include a new type of statement, the `return` statement. The rule for `return`

statements and the last statement type we haven't yet discussed appear below.

***ReturnStatement*:**

`return` [*expression*];

This statement stops the execution of a method and returns the value of the specified expression. If the method has a `void` return type, then there isn't an expression. Otherwise the there must be an expression and the expression type must match the return type of the method or be convertible to it.

Note that the rule for the return statement implies that the expression is optional. This is only the case if the method returns `void`. Otherwise, there has to be an expression and the type of that expression has to match the return type of the method. A method can contain more than one return statement, this only makes sense if one of them is conditional. Whenever a return statement is reached, the flow of control leaves that method and goes back to whatever called the method. So a return statement with no expression is an effective way to break out of a `void` method and not do anything that follows it.

None of the other aspects of the `Wombat` class are significant to us at this point. You actually have rules to understand everything here with the one exception of the `if` statements and the conditions that are in parentheses after them. At this point we have spent enough time looking at existing code. It is time to write some of our own from scratch.

2.2 Constructing New Code

We aren't going to write our code in the Wombat scenario. Instead, we will start working with a new scenario called PSPGCityScape. This is a very basic, simple scenario that we will build up over time to demonstrate a number of different aspects of programming and how we can do things in Greenfoot. When you load up the scenario and compile it, you will get a scene with two people and five houses that might look something like the image below, but might be a bit different. Look at the code in the `City` class to see how this is being set up. Why can yours be different from Figure 2.3? Do you get the same setup if you Reset the scenario? What is it in the code that causes that?

Try clicking "Act" or "Run". What happens? Nothing. If you go into the code you will see why. The `Person` and the `House` classes have nothing in them except for an empty `act` method. In fact, the code that you are seeing is the default code generated by Greenfoot when you make a new subclass of the `Actor` class. We are going to modify the `Person` class and make it so that the person moves when the `act` method is called. First you should modify the comments so that it says that you are the author and it describes the class that you are writing. To make sure that you didn't alter anything that breaks the code, press the "Compile" button on the editor or click on the world area in the main Greenfoot window. If there is an error you will be told about it.

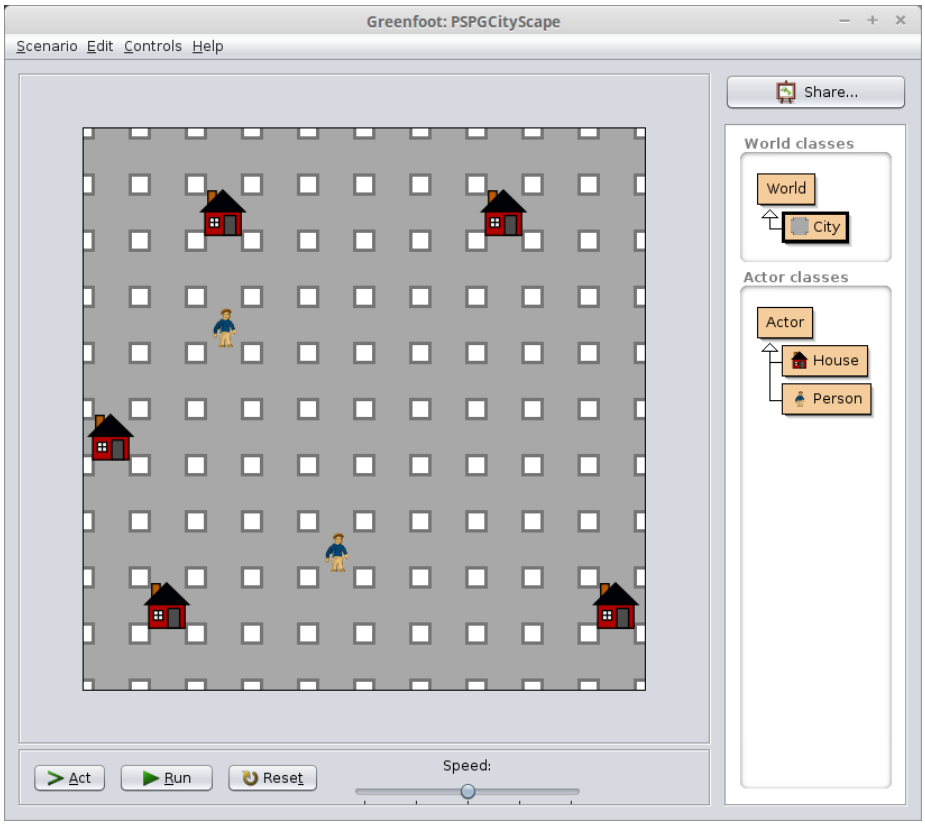

Figure 2.3: The PSPGCityScape scenario.

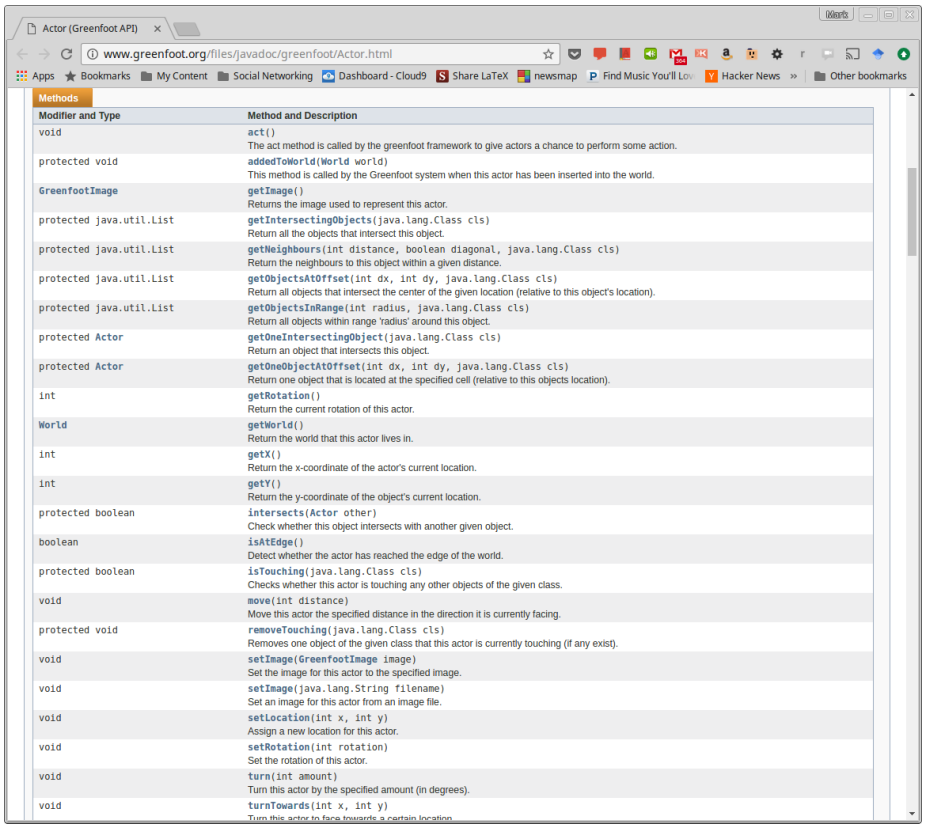

Figure 2.4: The Actor API page.

Now we want to modify the `act` method in `Person` to make it so that the person moves one square to the right each time that `act` is called. How are we going to do this? How can we make the person move? There clearly aren't any methods in the `Person` class that will do this for us. Remember in the last section where we saw that `WombatWorld` got to use the methods in `World` because it inherited from that class? `Person` inherits from `Actor`, so it can easily use the methods that are in `Actor`. Look up the `Actor` class in the API and see if there are any methods that could help with moving an `Actor`. Figure 2.4 shows the summary of methods in the Actor page of the API.

If you go through this list you notice that there are two main classes of methods. There are 11 methods that begin with the word `get` and four methods that begin with the word `set`, along with a variety of other methods. The `get` methods allow us to ask the `Actor` for information about it. The `set` methods allow us to set values in the `Actor`. There are nine methods that start with neither `get` nor `set`, which do various things. One of them is the `act` method.

Telling an actor to move is giving it an instruction. So the `get` methods are

ruled out fairly quickly as they shouldn't change the actor at all, they should just provide us with information about the actor. One of the `set` methods is named `setLocation` which seems promising for making the person move. We are supposed to pass it an x and a y coordinate and it should move the actor to that location. There is another method called `move` that should jump out as being an option.

In this case, the `move` method does exactly what we want. So if you use the following code for `act`, you will see the instances of `Person` move one square to the right each time you click the "Act" button.

```
1 public void act() {
2     move(1);
3 }
```

If you want to move to the left instead, you can use a negative value.

That was too easy as moving right and left is the default behavior for a method provided in the API. What if we want to more up or down? One way to do that is to turn, then move. We don't want to turn every time we act though, so we can put the call to `turn` in the constructor, then have the call to `move` in `act` as seen in the following code.

```
1 public Person() {
2     turn(90);
3 }
4
5 public void act() {
6     move(1);
7 }
```

This works, but it has the odd side effect that the person appears on the screen standing sideways.

To test out our logic skills and ability to use the API a bit more, we will write a version that moves up and down without turning the person. We will will do this with `setLocation`. To see what this method does, let us start by calling it and giving it constant values for x and y, perhaps something like the following.

```
1 public void act() {
2     setLocation(0,0);
3 }
```

Put this code into your `Person` class, compile all the classes, and click the "Act" button. What happens? You should see that both of the people move. They both move to the top left corner on top of each other so you only see one of them. Is this what you were expecting to have happen? Try changing the values so instead of 0, 0 you have some other numbers. What happens then?

It shouldn't take long for you to realize that the x coordinate moves the player from left to right and the y coordinate moves the player from top to

bottom. This second fact might surprise you, after all, you always learned in math that the y axis goes from bottom to top. Computer graphics tend to be oriented the opposite way for historical reasons. The earliest "displays" were printing on paper and the paper always scrolled by so that the printing was done from top to bottom. The other thing that might surprise you is that both the x and y values start at 0, not at 1 the way you normally count. We will see that it is very common in Java to start counting things at zero. This also has historical and practical roots.

So we got our people to move and we learned some things about the coordinates that are used in Greenfoot. However, we didn't get the people actually walking up or down. Before we do that, let's take one small step up in complexity for our program. The first version was boring because the people always jumped to a fixed location and it was always the same location. Why don't we make it so that the people jump to random locations? We can do this using code that is very similar to things we saw in the PSPGWombat example. We are going to call the method `Greenfoot.getRandomNumber()`. In addition to using this method, we should go ahead and introduce some variables in the `act` method. In this particular example we don't have to, but if we don't, our one line of code will get really long. Introducing variables, even when they aren't needed, can help make the code more readable. Make your act method look like this.

```
public void act() {
    int x = Greenfoot.getRandomNumber(10);
    int y = Greenfoot.getRandomNumber(10);
    setLocation(x, y);
}
```

Compile this code and click the "Act" button. What happens? What happens if you click in again? Does this behavior make sense? It is what we wanted? Is this the ideal code for doing it?

The answer to that last question is, "No." Why not? What is wrong with it? The problem is that it isn't general. If you recall the discussion in the last section we looked at the `randomLeaves` method in `WombatWorld` and discussed how the random number generation was done in a way that was general. The problem with this code is that we have hard coded in the number 10 because we know from the `City` class that the world is a 10x10 grid. What happens if someone changes the size of the world though? If they change the size of the world and don't change this `act` method in the `Person` class, our people will behave in ways that they shouldn't. How they misbehave depends on how the world size is changed, but whatever is done, we will have a bug.[4]

[4]The term bug is commonly used in reference to errors that occur in programs. This term, as applied to computers, also has historical roots. The earliest computers were vast devices that filled large rooms and were often programmed by actually changing the wiring. One of these early computers kept producing erroneous output. Eventually it was discovered that a moth had flown into the machine and was causing a short that was altering the calculated results. Unlike the moth, modern bugs aren't the results of chance. They are errors put there

We can make this code more general in the same way that `randomLeaves` was made more general. Instead of hard coding the numbers for the size of the world, we should ask for those values with a method call. In the `WombatWorld` class this was done by calling the `getWidth()` and `getHeight()` methods. This code tries to do that.

```
public void act() {
    int x = Greenfoot.getRandomNumber(getWidth()); // Error:
        // Actor doesn't have a getWidth() method, World does.
    int y = Greenfoot.getRandomNumber(getHeight()); // Error:
        // Actor doesn't have a getHeight() method, World does.
    setLocation(x,y);
}
```

Unfortunately, this code doesn't work. That is because our `act` method is in `Person` which is an `Actor`. The `getWidth()` and `getHeight()` methods are in `World`. After all, they are supposed to give the width and height of the world, not an actor. In order to call them, we have to have an object of type `World`. We don't want just any instance of `World` either, we want the world that our person is part of. Fortunately, the `Actor` class has a method that will do this for us. If you go back and look in the API for `Actor` you will see that there is a method called `getWorld()` that returns an object of type `World`. Once we have that object, we can call `getWidth()` and `getHeight()` on it. This working code is shown here.

```
public void act() {
    int x = Greenfoot.getRandomNumber(getWorld().getWidth());
    int y = Greenfoot.getRandomNumber(getWorld().getHeight());
    setLocation(x,y);
}
```

This is using the longer form of the method call where we have to explicitly provide an object to call the method on and use the dot notation to say that we are doing something with that object. Put this code in and compile the program then click the "Act" button. The behavior should be the same as what you saw before, but this code is now more general because it will work for any sized world.

Now we are ready to write the code we set out to add to our person to make it so that the person "walks" down. The question is, how do we do this? We know that locations with bigger y values are further down. The problem is that we want the player to take one step each time `act` is called. So we want the x part of the location to be one bigger than what it already was. So to do this we need to get the current position in both x and y. The y we want to make bigger and the x we want to leave the same. To get the current location we need to consult the API again. It turns out there are methods in the `Actor` class called `getX()` and `getY()` that tell us the current location of the actor. So instead

by programmers who make mistakes. Despite this, the use of the term bug is still ubiquitous.

of setting our variables `x` and `y` equal to constants or random numbers, we can set them to the current position with a little math for offsets. This gives us the following code.

```
public void act() {
    int x = getX()+1;
    int y = getY();
    setLocation(x, y);
}
```

You should put this into your `Person` class, compile it, and see what happens when you run it. Do the two people in the city walk downward? They should. Make sure you understand why they do.

2.3 Types in Java

We have talked about types a few times to this point and we have seen a few used. The first type that we saw was the `int` type. This should be used when you need an integer value, that is a number, positive or negative, with no fractional part. We have also seen the `boolean` type which can either be `true` or `false`. We will see a lot more of the `boolean` type in the next chapter when we discuss the `if` statement and Boolean logic. If you look back at the rule for types you see that both `int` and `boolean` fit under the rule *PrimitiveType*. The *PrimitiveType* rule we are using for this chapter also allows two other types: `double` and `char`.

The `double` type is used to represent numbers, like `int`, only we use the `double` type when the number has a fractional part. The `double` type is also good for representing very big numbers for which you would normally use scientific notation. You might wonder why you should ever use the `int` type if the `double` type can store more general numbers and has a larger range. There are two answers to this. First, math done on the `double` type is inexact. If you go further in Computer Science you will likely learn the details of how the `double` type works and why it is inexact, but the basic idea is that it only keeps a certain number of digits of precision and after that it rounds. So if you are doing something that needs to have exact math, you should do it with an `int` type. Probably more important to you at this point is the second reason. There are parts of the Java language that require use use of `int`s. Also, if something wants a `double` and you pass it an `int` everything works fine because every possible `int` can be represented by a `double`. However, if you try to use a `double` in a place that wants an `int`, like the `setLocation(int x,int y)` method in `Actor`, you will get an error. You can test this out by trying to call `setLocation(3.3,5.6)` and then compiling. If you have a value that is a `double` and you really need to use it as an `int` you have to do something called a type cast. To do a type cast in Java, simply put the type you want in parentheses in front of the expression you want to cast. Here is a somewhat contrived example of that.

```
double x = 6.7;
double y = 4.3;

setLocation((int)x,(int)y);
```

Note that when you cast a `double` to an `int` it doesn't round, it truncates. Any fractional value is simply thrown away and you get the whole number part. So the location in this example will be set to (6, 4), not (7, 4) which is what you would expect if the values were rounded.

The `char` type is the only other primitive type that we will deal with in this book. As the name implies, the `char` type is used to represent character data. Actually, it represents a single character. We put characters in our Java code by putting the character we want inside of single quotes as is shown in this sample code.

```
char a = 'A';
```

The *Type* rule could also become a *ReferenceType* instead of a *PrimitiveType*. We have seen this option used as well. The `randomLeaves` method declared the variable `leaf` that was of type `Leaf`. Every class defines a type in Java, whether it is one that we write, or one that someone has written. The `Leaf` type is an example of something we wrote. We will, over time, also make variables that are of some of the types defined in the Greenfoot API. We will also use a number of different classes from the standard Java libraries. One of those types is so fundamental that it is worth introducing here. It is the `String` type. The `String` type represents text and is quite general across programming languages. A `String` is made up of a bunch of characters. When you want to make a `String` in a Java program you simply put the text that you want inside of double quotes.

```
String s = "This is a test.";
```

Note that the type `String` starts with a capital letter. This is because it is the name of a class that someone else wrote and they were following the standard Java naming scheme. You can type whatever you want inside the double quotes to make a string. The one exception to this is that you can't put a newline inside of them. If you want a string with a newline, you should put `\n` inside of the string.

The `String` type also allows something that no other class type does. You can use `+` with strings and the two strings will be concatenated. This was added to the language because it is such a common operation. We will use strings in the next chapter when we deal with keyboard input.

2.4 Modulo and Integer Division

The code we wrote earlier for our person had it walk to the right. Eventually it ran into the edge of the world and didn't do anything else. This is because

Greenfoot doesn't allow actors to be placed outside of the bounds of the world. It would be nice to have the person be able to do something that isn't just random jumping around and doesn't end with it simply running into the edge of the world either. There are many options we could pick, but for now we will pick a fairly simple option. We want to have the person walk back and forth across the screen. It should take five steps to the right, then five steps to the left, and repeat that pattern.

In the next chapter we will learn how we could do something like this with conditional execution. That is likely how most programmers would choose to solve a problem like this. Since we don't know how to do that yet we will do it with basic math and some problem solving skills. Specifically, we are going to use the capabilities of integer division and the modulo operator.

Have you stopped to wonder what happens if you divide to integers? The operations of addition, subtraction, and multiplication are all closed on the integers. That is to say that if we add, subtract, or multiply integers, the result is always an integer. So if n and m are integers, we know that n+m, n-m, and n*m will also be integers. [5] This isn't normally true for division. Most of the time in math n/m will not be an integer unless n happens to be a multiple of m. If you do division in Java with `double`s you will get roughly the result that you expect. However, if you do it with `int`s the result you get will also be an integer. Specifically, it gives you the integer quotient.

To understand this, think way back to 3rd or 4th grade before you knew decimal notation. Back then you would write the answer to 13/5 as 2 remainder 3.[6] Integer division in Java gives you the 2 part of this. There is another operator that you might have noticed in the Expression rule that gives us the remainder. It is the modulo operator and it is denoted by the `%` in Java. If you write `13%5` in Java you get 3, the remainder from your elementary school math.

Combining integer division and modulo can allow us to do a number of fun things. In this case, we want to use it to make our `Person` walk back and forth across the screen. We have already seen that moving to the right can be achieved by doing something like `x = getX()+1`. Similarly, walking to the left can be accomplished with `x = getX()-1`. We can think of this as `x = getX()+dir` where `dir` is an `int` with a value of either `-1` or `1`. So all we have to do is come up with some math that allows us to have `dir` be `1` the first five times we act, `-1` the next five, and repeat.

In order to do this, we have to know what step we are on. We have to be counting our steps. This can be done in the same way that the `Wombat` counted the number of leaves it had eaten. We need to add a private member variable to the Person. We might call it `steps`.

[5] It should be noted at this point that the `int` type won't always behave the way you expect integers to. This is because the `int` type has finite bounds. If you start dealing with numbers much larger than 2 billion, the value will "overflow" and become negative. Similarly, if you have numbers smaller than -2 billion they can wrap around to be positive.

[6] The problem 13/5 might have been put on a worksheet as 13 little icons where you were told to make groups of 5. The quotient is how many complete groups you make and the remained is how many are left over.

```
private int steps = 0;
```

Inside of the `act` method we need to increment steps. We can accomplish this by putting a line like the following at the end of act.

```
steps = steps+1;
```

With these two lines, our person will now keep track of how many steps it has taken. Now all we have to do is come up with some math to go from number of steps to either 1 or -1. We wanted to do this based on whether we are in steps 0-4, 5-9, 10-14, 15-19, etc.

Look at those groupings. What is true about them, particularly in regards to division? The first group is n such that n/5 is 0. The next group would all be 1 when divided by five, etc. So steps/5 tells us something about which direction we should be heading. When steps/5 is even we want to go to the right. When it is odd, we want to go to the left. So how do we figure out if it is even or odd? That is a great use for the modulo operator. Given an integer n, n%2 is 0 when n is even and 1 when n is odd. So the expression (steps/5)%2 will be 0 when we want to go to the right and 1 when we want to go to the left.

The problem now is that we don't want 0 and 1, we want 1 and -1. That can be fixed with a fairly simple set of operation. If n is 0 or 1 we can do (n*2)-1 and we get -1 or 1. That is the opposite of what we want so we simply subtract instead of adding. This gives us final code for act that looks like the following.

```
public void act() {
    int dir = ((steps/5) % 2)*2-1;
    int x = getX()-dir;
    int y = getY();

    setLocation(x, y);
    steps = steps+1;
}
```

Put this into your `Person` class, compile the code, and click on the "Run" button. If everything is typed in correctly your person should walk back and forth across the screen, just the way we wanted it to.

!!! Passing of parameters

Chapter Rules

Class:

public class *Name* [extends *ReferenceType*] {

[*Constructors*]

[*Methods*]

[*MemberData*]

}

Method:

public *Type name*([*Type1 name1*[,*Type2 name2*, ...]]) {

[*Statements*]

}

***Type*:**

PrimitiveType | *ReferenceType*

***PrimitiveType*:**

boolean | char | int | double | void

The type void can only be used as the return type of a method, not for a variable declaration.

***ReferenceType*:**

This is the name of a class. It can be one that you have written or one that is in an library. Class names typically start with a capital letter and have each new word capitalized. They follow the normal rules for Java names.

***Name*:**

Java is fairly flexible about naming. Valid Java names have to start with a letter or underscore followed by any number of letters, underscores, or numbers.

***Constructor*:**

public *ClassName*([*Type1 name1*[,*Type2 name2*, ...]]) {

[*Statements*]

}

***Statement*:**

VariableDeclaration | *Assignment* | *MethodCall* | *ReturnStatement* | *CodeBlock*

***MethodCall*:**

[*objectExpression.*]*methodName*([*expression1* [, *expression2*, ...]]); |

ClassName.methodName([*expression1* [, *expression2*, ...]]);

The *objectExpression* is any expression whose type is an object type. If no objectExpression is given, the method is called on the current object. The methodName must be a valid name of a method on that type. The expression is often just the name of a variable, but it can itself be a method call that returns an object.

The second format is used for static methods. For these we don't need an object, we just put the name of the class in front of the dot. The

Greenfoot class has only static methods in it so you call them with a format like Greenfoot.getRandomeNumber(3);.

***VariableDeclaration*:**

Type name[=*expression*];

This declares a variable of the given type with the specified name. You can think of variables as boxes that can hold a value. The box is labeled with the name we give it and it can only hold things of the proper type. What you can have for type is described after all the statement types are introduced.

A variable can only be used in a certain "scope". The scope of a variable runs from the point at which it was declared down to the close curly brace that ends the code block it is declared in. Consider the following example method.

```
public void foo() {
    int a=5;
    {
        int b=8;
        // Point 1
    }
    // Point 2
}
```

At point 1 you could use either `a` or `b`. However, the scope of `b` is only inside the inner curly braces so you can't use `b` at point 2, only `a`.

***Assignment*:**

name=*expression*;

This stores the value of an expression in the variable with the given name. The variable must be in scope. So it either has to be declared in the current method above this line or it should be a class level variable. In order for this to work, the types need to match.

Here are some example variable declarations and assignments that could be made from them.

```
int a;
int b;
double d; // This type will be discussed later in the chapter.
String s; // This type will be discussed later in the chapter.

a = 6+7;
b = a-4;
d = 2*3.14159;
s = "This is a test.";
```

***Expression*:**

An expression in Java is anything that has a type and value. Below is a list of options.

- Variable name – The name of a variable evaluates to the value currently stored in that variable.
- Numeric literal – Numbers are expressions. You can represent integer values or decimal fractions as you normally would. Very large or very small numbers are written using scientific notation of the form 1.234e56 which is the number 1.234*1056, a rather large number indeed.
- String literal – String literals can be any sequence of characters enclosed in double quotes. For example, "Hi mom." is a string literal.
- null – This is the value for any object type when we don't have a valid object.
- Method call – A method call for any method that returns something other than void is an expression.
- Operator expression – Java supports different mathematical and logical operators. Most will join two different expressions. Some operate on one expression and one uses three expressions. This is a list that includes most of the ones we are likely to use.
  - A+B : Standard addition for numeric values. If either A or B is a String the plus sign will do string concatenation.
  - A-B : Subtraction for numeric values.
  - A*B : Multiplication for numeric values.
  - A/B : Division for numeric values.
  - A%B : Modulo operator that returns the remainder after division for integer values.
  - A=B : This is the assignment we saw above. It is also an expression which takes on the value that is being stored. In this case, A needs to be the name of a variable while B can be any expression of the right type.
  - new ClassName(args) : The new operator is used to make objects. This will make a new object of type ClassName using the constructor that is matched by the arguments passed in.

***MemberData*:**

private *Type name*[=*expression*];

This defines a class level variable with the specified name and type. The initial value doesn't have to be specified.

Constructors and methods have a sequence of statements inside of curly braces. As a general rule, curly braces are used in Java to group lines of

code. When a method is called in Java, the statements in that method are executed one at a time in order from the top of the method to the bottom. At this point we will only consider five types of statements in Java. So any place that you can have a statement, you can have one of these five. More statement types will be introduced later on. Here is a simple little example of a class. This one doesn't do much, but it demonstrates the three things that you can put inside of a class.

```
public class SimpleClass {
    private int value;

    public SimpleClass(int v) {
        value=v;
    }

    public int getValue() {
        return value;
    }
}
```

***ReturnStatement*:**

return [*expression*];

This statement stops the execution of a method and returns the value of the specified expression. If the method has a void return type, then there isn't an expression. Otherwise the there must be an expression and the expression type must match the return type of the method or be convertible to it.

***CodeBlock*:**

{ [*statements*] }

Any place a single statement is expected in Java, it can be replaced by multiple statements inside of curly braces. They don't all have to be on the same line as I have typed it here.

Exercises

1. Have it start with 20 houses instead of 5.
2. Spin the house.
3. Walk in a box shape.
4. Walk randomly.
5. Walk in a circle with sin and cos.
6. Make the person reproduce.

7. Have person remove itself at certain point.

8. Combine some of the above.

In class I did walkToward with math. This has a divide by zero error that sets us up for conditional execution.

Chapter 3

Conditionals

In the last chapter we saw how to program. We looked through the code for the `Wombat` and related classes. We even added our own functionality to the `Person` class to make it so that the person would move in various ways. We were somewhat limited in what we could do though. There was a fundamental reason for this. Given what we know, when we call a method, it executes one line after the other in order and always executes every line once. We were able to make some reasonably interesting behavior using different math functions, but at a certain level we were limited. We can add a lot more power to our programs if we have the ability to choose whether certain lines are executed or not. This power is called conditional execution and it is what we are going to learn about in this chapter.

3.1 The if Statement

The most basic conditional statement, and one that you will find in pretty much every programming language, is the `if` statement. It reads very much like plain English. If something is true, then do something. The rule for the `if` statement is shown below along with a modified definition for *Statement* that includes the `if` statement and the `switch` statement that will be introduced later in this chapter. There are two things to note about this rule. First, the `else` is optional. If you don't specify an `else`, when the condition is `false` execution will simply go to the statement after the `if`. Second, this is a common place where the *CodeBlock* rule is used. The *statement1* and *statement2* elements in this rule can be single statements, but it is very common to use a block of code in these situations. The means that you typically see the `if` statement followed by a curly brace, a set of indented statements, and a close curly brace. In fact, in this text we will always use code blocks with `if` statements, even when we only have a single statement in the curly braces and the braces aren't required.

***Statement*:**

VariableDeclaration | *Assignment* | *MethodCall* | *ReturnStatement* |

CodeBlock | *IfStatement* | *SwitchStatement*

***IfStatement*:**

`if(`*booleanExpression*`)` *statement1* [`else` *statement2*]

This is the most basic conditional statement. If the *booleanExpression* is `true`, *statement1* will execute. If it is `false` either nothing happens of *statement2* is executed if the `else` clause is used. A common usage might look like the following:

```
if(a == 5) {
    b = 25;
}
```

There is one other thing in this rule that might not make sense yet: the *boolean-Expression* in the parentheses after the `if` statement. You know what the `boolean` type is and you know what an expression is. This notation indicates that it is an expression with the type `boolean`. What types of expressions have the `boolean` type? If an expression is a method call it will be `boolean` if the method it calls returns the `boolean` type. An example of this is the `canMove()` method that was in the `Wombat` class. The keywords `true` and `false` are also valid `boolean` expressions though they aren't very interesting to use.

Java also has a set of operators that return a `boolean` type. The first set of these that we want to talk about are numerical comparisons. These include the operators `==`, `<`, `>`, `<=`, `>=`, and `!=`. The first of these operators is a check for equality. Note that there are two equal signs. A single equal sign denotes assignment in Java, as we saw in the last chapter. If you want to compare two values to see if they are the same, you put two equal signs. In most uses, if you only put one you will get an error when you compile because the type won't be `boolean` and the `if` statement requires a `boolean`. The next four operators are fairly self-explanatory. They are `true` if the first value is less than, greater than, less than or equal to, or greater than or equal to the second value. The last operator, `!=`, stands for not equal and will be `true` if the two values are different and `false` if they are the same.

!!! table of operators

These few operators, along with the `if` statement give us the ability to do some things. To start with, we will revisit a problem that we have already solved before. It is the problem of making our person walk back and forth. In the last chapter we did this with integer division and modulo arithmetic. This is a perfectly valid way of doing it, but it is an approach that most people would find somewhat confusing. We can get the same result with conditional execution using the `if` statement.

Let's think through how we want to do this. As we saw in the last solution, the trick is that we have to keep track of our steps and change what direction we are going in at the proper time. Now that we have an `if` statement, we can state what we want to do in English in such a way that it is almost directly

convertible to Java. If we have done five steps in one direction, we should turn around and go the other direction.

As with our earlier solution, we want to keep a variable, `steps`, that counts steps for us. Unlike the earlier solution, this time the variable won't keep the total number of steps we have taken. That is because we don't really care how many steps total we have taken. We only care about the number of steps that have been taken since the last time we turned around. We will also move the variable `dir` to the class level because now we need to remember what direction we are going from one call of act to the next. The relevant code from `Person` for this solution is shown below.

```
private int steps = 0;
private int dir = 1;

public void act() {
    if(steps == 5) {
        dir = -dir;
        steps = 0;
    }
    int x = getX()+dir;
    int y = getY();
    setLocation(x, y);
    steps = steps+1;
}
```

This code should be mostly self-explanatory, but you want to pay close attention to the syntax, particularly the use of parentheses and brackets. In the `act` method we first check if we have taken five steps. If we have, we do two things. We turn around, by negating the value of `dir`, and we reset our count of how many steps we have taken. The rest of the code is basically the same as what we saw before. The only minor trick here is negating `dir` in order to turn around. You don't have to do it this way. The act method could be done like this instead:

```
public void act() {
    if(steps == 5) {
        if(dir == 1) {
            dir = -1;
        } else {
            dir = 1;
        }
        steps = 0;
    }
    int x = getX()+dir;
    int y = getY();
    setLocation(x, y);
    steps = steps+1;
}
```

This solution employs an additional `if` statement with an `else` instead of doing

the negation math trick. Both are correct and they run exactly the same way in this code. The first one is a bit shorter, but if you find the second one easier to read and understand, you could certainly use it.

This version of the code also shows an interesting feature of the `if` statement: they can be nested. If you look back at the rule for the `if` statement, it simply says that it can have a statement after the parentheses and normally this is done with a code block. The code block can have statements in it. The fact that `if` is a statement allows us to nest `if` statements inside of one another. We will see this in other places as well. This generality of statements provides the language with a lot of power without making it overly complex.

Here are some additions to the expression. We won't repeat all the operators from the last chapter, we will only specify some additional operators that are relevant for our needs here. We just discussed the first six. The others will be discussed later in the chapter with the exception of the last one which appears in the next chapter.

***Expression*:**

An expression in Java is anything that has a type and value. Below is a list of options.

- Operator expression – Java supports different mathematical and logical operators. Most will join two different expressions. Some operate on one expression and one uses three expressions. This is a list that includes most of the ones we are likely to use.
  - A==B : This returns true if A is the same as B. For object types use `.equals` instead.
  - A<B : Returns true if A is less than B. Works for numeric types.
  - A>B : Returns true if A is greater than B. Works for numeric types.
  - A<=B : Returns true if A is less or equal to than B. Works for numeric types.
  - A>=B : Returns true if A is greater than or equal to B. Works for numeric types.
  - A!=B : Returns true if A not equal to B.
  - A || B : This `boolean` operator is `true` if A or B is `true`. It is inclusive or so it is `true` if both are `true`.
  - A && B : The `boolean` operator returns `true` only if both A and B are `true`.
  - !A : This `boolean` operator is the not operator. It returns `true` if A is `false` and `false` if A is `true`.
  - A ? B : C : This is the ternary operator. A should be a `boolean`. If A is `true`, the value will be B, otherwise the value will be C.
  - A `instanceof` C : Here A is an expression with an object type and C is the name of a `class`. It tell you if A is of the type C.

3.2 Boolean Logic

In the last section we introduced some operators that return `boolean` values and allow us to build some simple `boolean` expressions. There are times when we need more complex `boolean` expressions that combine multiple other `boolean` expressions. There are operators that can do this for us. The use of these operators falls under the realm of Boolean logic. It is very much like arithmetic in the two valued system of the `boolean` type.

We will look at three of these `boolean` operators: and, or, and not. To illustrate their usage, let's consider some simple examples. Say that we want to give our person an integer value for an age and have it behave differently depending on the value of age and some random chances. The behaviors will be something like combinations of things you made people do in the exercises for the last chapter. When their age is less than 24 or greater than 45 the person should move around randomly. When their age is between 25 and 30, inclusive, they should have a 20% chance of reproducing. For an age of 70 or greater the person should remove itself from the world. At all other ages the person just stays where it is.

We don't care so much about the code we would use to make the different actions happen. We care more about the `boolean` logic that goes into selecting which one we would do. Let's start with the first option. We are supposed to take that option if `age` is less that 24 or greater than 45. Putting some math notation into that we could write it as age<24 or age>45. All we need is a way to express or. In Java we express or with two pipes, ||. So our condition in Java will read as `age < 24 || age > 45`.

There are some things to note about the || operator. It is a short circuit inclusive or. The fact that it is inclusive is different from the normal usage of the word or in English, which is exclusive. When you as A or B in English it generally means that something is true if either A is true or B is true, but not if both are true. Consider a parent telling their kid that they can have cookies or cake. When we say that in English, the kid isn't allowed to grab both. The inclusive or is true is unless both operands are false. So it is true when both of the arguments are true. Using an inclusive or, the kid would be able to take both the cookies and the cake in our example.

The || operator is also a short circuit operator. In the case of or, this means that if the first operand is `true`, the second operand isn't even evaluated. That is because `true` || B is always `true`, regardless of what B is. This makes the operator more efficient and can be useful for other reasons as well.

For the second condition, we want an expression that is `true` if age is between 25 and 30, inclusive. In math class you might have written that as 25<=age<=30. That won't work in Java. To understand why, you have to understand something about what the compiler sees. You might remember from back in algebra that for operators of the same precedence, you normally do them from left to right. [1] Java does this same thing. So if you type in `25 <= age`

[1] Exponentiation is an exception to that rule as it should be done right to left, but since

`<= 30` Java sees `(25 <= age) <= 30`. To make it clear why this is a problem, pick a value for `age`, let's say 26. Because 26 is greater than 25, the expression in parentheses evaluates to `true`. So then Java sees `true <= 30`. That expression doesn't make sense. You are comparing an `int` to a `boolean` and the Java compiler will give you an error telling you that this is nonsense.

To see how we should do this, we can reword our English description of the expression a bit. Instead of saying that `age` is between 25 and 30, inclusive, we can say that `age` must be greater than or equal to 25 and be less than or equal to 30. Using the math operators that would be 25<=age and age<=30. We express the and in Java with the `&&` operator. So the condition we would want in our `if` statement is `25 <= age && age <= 30`.

The `&&` operator is a short circuit logical and. It operates on two `boolean` values and only returns `true` if both arguments are `true`. If either is `false`, or both are `false`, it will return `false`. The fact that it is short circuit means if the first argument is `false`, it doesn't bother evaluating the second argument. That is because `false &&` B is going to be `false` no matter what B is. This property of the `&&` operator comes in very handy in certain situations. As we will see later on, there are some situations where an expression could cause an error. In that case you can put that expression second and have the first expression be `true` only if the second expression is safe to evaluate.

Turning back to the code for moving a person around, for the second condition we only wanted to take the particular action 20% of the time. There are two ways that we could do this. The first is to nest another `if` statement inside of the first one and have it check a random number. That would give us an if like the following:

```
if(25 <= age && age <= 30) {
    if(Greenfoot.getRandomNumber(100) < 20) {
        // reproduce
    }
}
```

The alternative to this is to simply put the random number check in the outer `if` and combine it with the other checks using another `&&`. That would look like this:

```
if(25 <= age && age <=30 && Greenfoot.getRandomNumber(100) < 20) {
    // reproduce
}
```

This version is shorter, and assuming that there isn't something that we should do the other 80% of the time, this is probably the way that most people would write this code.

The last check is a simple check of `age` against the value 70. If we put all of this together we get the following code to represent our logic.

---

there is no exponentiation operator in Java that doesn't concern us here.

```
if(age < 24 || age > 45) {
    // walk randomly
} else if(25 <= age && age <= 30 && Greenfoot.getRandomNumber(100) < 20)
    {
    // reproduce
} else if(age >= 70) {
    // remove self from world
}
```

Because all of the conditions in this example are mutually exclusive we have used a common shortcut for nesting `if` statements in the `else` clause of other `if` statements. If the options weren't mutually exclusive, we would simply eliminate the `else` clauses all together and have each `if` start one a new line.

The expressions we used in this example all included either an || or an `&&`. None of them mixed the two. When you have expressions that use both `&&` and ||, it often matters which one is evaluated first. You are familiar with the fact that in math, we always do multiplication before we do addition. So 2+3*5 is 17, not 25. For `boolean` operators, `&&` takes precedence over || and will happen first. If you ever have a hard time remembering that though, simply use parentheses to make it clear what operations are happening first.

There is another `boolean` operator in the Java language: not. The logical not operator is represented by a exclamation point, `!`, in the Java language. It is a simple operator that flips a `true` value to `false` and a `false` value to `true`. It should precede the `boolean` expression that it is operating on. The `!` operator is higher precedence than `&&` or ||. In fact, it is higher precedence than multiplication. As such, you will often want to put the expression you are negating with `!` inside of parentheses.

3.3 Keyboard Input

Having conditional statements gives us the capability to add some interesting functionality to our `Person` class. In particular, we can make our person respond to keyboard input. If you look in the API at the `Greenfoot` class you will find that the introductory information talks about keyboard input and there are two methods that deal with getting information from the keyboard: `String getKey()`, and `boolean isKeyDown(String)`. The first method will return a `String` to you that represents the most recently pressed key. The second method takes the key you are asking about and returns `true` or `false` to tell you whether it is currently being held down. This is the method that we will work with first.

Our goal is to make it so that the `Person` moves around responding to keyboard input. For the first cut, we'll just make it so that the movement is controlled using the arrow keys. If you read the API you saw that we represent all of the keys as strings. In the case of the arrow keys, they are "up", "down", "left", and "right". So we can pass these arguments into `isKeyDown` and Greenfoot will tell us whether or not that key is pressed. We can write four `if` statements where each one has a call to `isKeyDown` with a different

argument. In the body of those `if` statements we could call `setLocations`, but we will alter variables that we declare before the `if` statements. So assuming we use the variables `x` and `y` again, and initialize them to the current location of the person, then in the `if` statement that checks if the right key is pressed we will make `x` one larger. Doing this for all four directions gives us code like the following for the act method.

```
public void act() {
    int x = getX();
    int y = getY();
    if(Greenfoot.isKeyDown("left")) {
        x = x-1;
    }
    if(Greenfoot.isKeyDown("right")) {
        x = x+1;
    }
    if(Greenfoot.isKeyDown("up")) {
        y = y-1;
    }
    if(Greenfoot.isKeyDown("down")) {
        y = y+1;
    }
    setLocation(x, y);
}
```

You should enter this code into your `Person` class, compile it, and click the "Run" button. The people will just sit there unless you press an arrow key. What happens when you press an arrow key? Why? Is this what we wanted?

What you should have seen was that when you press an arrow key, both of the people in the world go moving in the same direction. When you release the arrow key, they both stop. Technically this is what we asked for. We wanted to make it so that the `Person` class was controlled by the arrow keys and we succeeded in that. Now that we have it though, we realize that this is a bit dumb. After all, what is the point of having both people do exactly the same thing, unless perhaps we were trying to make some type of line dancing program. For most programs it might be nice to have the capability to make it so that different keys control the different people. So maybe one person is controlled with the arrow keys and the other is controlled with 'w', 'a', 's', and 'd'. If we want to make this happen, the first thing that we should do is try to make our code more general.

In the `act` method shown above, we have hard coded in string literals, the stuff inside of double quotes, to work with the arrow keys. What we should really have done is to make it so that the values we are checking are member variables. That way we could change the control keys by simply changing the values of those variables. Our first step is to make this generalization. We will change the code some without actually adding any functionality. This type of change, where you alter how a program does something without altering what

it does, is called REFACTORING and it can be a very helpful thing to do when we want to add functionality. The purpose of refactoring before we actually enhance the functionality is that we can test the refactored code to make sure it still does what it did before, then after we have verified that it still works we can add the new functionality. This is a different style of breaking a large problem down. Instead of breaking a problem down into code, we are breaking down the way we code. Our refactored code might look something like this:

```
private String leftKey = "left";
private String rightKey = "right";
private String upKey = "up";
private String downKey = "down";

public void act() {
    int x = getX();
    int y = getY();
    if(Greenfoot.isKeyDown(leftKey)) {
        x = x-1;
    }
    if(Greenfoot.isKeyDown(rightKey)) {
        x = x+1;
    }
    if(Greenfoot.isKeyDown(upKey)) {
        y = y-1;
    }
    if(Greenfoot.isKeyDown(downKey)){
        y = y+1;
    }
    setLocation(x, y);
}
```

If you compile and run this code you will see that it does exactly the same thing that the code had done before. So why did we do it? It sets us up nicely for the next step where we make it so different instances of `Person` can use different keys.

To accomplish this we need to add a constructor to the `Person` class. Actually, we will add two constructors. We'll see why in just a second. The first constructor we will add is one that sets the values of `leftKey`, `rightKey`, `upKey`, and `downKey`. Remember that constructors are like special methods that have the same name as the class and lack a return type. They are used specifically to set up values for new objects and are called when we use `new`. The constructor that we want to add looks like this:

```
public Person(String left, String right, String up, String down) {
    leftKey = left;
    rightKey = right;
    upKey = up;
    downKey = down;
}
```

If you put this code into your class and compile it, you will get an interesting error. Part of what is interesting is that the error isn't in `Person`. You will get an error in the `City` class. The error will read something like, "cannot find symbol – constructor Person()" and it will be on one of the lines calling `addObject` that adds a new person. What this error is telling us is that we need a constructor that doesn't take any arguments. You might wonder why it is that we didn't have one before and it worked, but it doesn't work now. A constructor with no arguments is often called the DEFAULT CONSTRUCTORdefault constructor. As it happens, Java will add one for you if you don't have any other constructors. However, it only adds it if you don't have other constructors. So before we added our constructor, Java provided a blank default constructor for us. Now that we have written a constructor, we have to make a constructor with no arguments explicitly. That means adding the following lines of code.

```
public Person() {
}
```

With these in you should be able to compile. If you run the program it behaves exactly as it did before. Both the the people respond to the arrow keys. This is because both people are using the default constructor and keeping the initial values for the different strings.

In order to make it so that one of the people responds to different keys we need to edit code in the `City` class where the two different people are created and added to the world. There are two lines in this class that look like the following:

```
addObject(new Person(), 2, 3);
addObject(new Person(), 4, 7);
```

We only need to alter one of these because it is fine if one of the people is controlled by the arrow keys, we just need the other one to be controlled by different keys. What we need to do is pass in four different strings in the parentheses after new `Person` so that those will be set as the controls for that player. After doing that, those two lines might look like the following.

```
addObject(new Person("a", "d", "w", "s"), 2, 3);
addObject(new Person(), 4, 7);
```

If you compile this code and run it you will find that now only one of the people moves when you use the arrow keys. The other responds to the letters we have told it to use. You could try adding yet a third person and giving it a different set of keys.

The `Greenfoot` class had another method for getting keyboard input called `getKey`. If you look at this method in the API you see that it returns a `String`. That `String` represents the most recent key pressed since the last time this method was called. If no key has been pressed it returns `null`. This `null` is something new to us. It is quite significant in Java and worth discussing. If you remember the rule for *Type* there were two options. The *PrimitiveType* we

have covered fairly completely. The *ReferenceType* we talked about a bit and we said previously that `String` was one of these types. It turns out that there are other differences between these classes of types. We said earlier that you could picture a variable as something like a box and that box can hold a value of the specified type. This is exactly the picture you should have in your head for primitive types. For class types it isn't exactly correct. When we declare a variable of type `String`, for example, we don't get a box that holds strings, we get a box that can hold references to strings. Having a reference to an object is like having a person's address. It isn't quite the same as having them, but you know where to find them when you need them. Because these variables of these types store references and not actual values, they are often called reference types.

The fact that variables of class types store references allows a new possibility that can't happen with primitive types. A variable of a reference type can have an invalid reference. An integer variable is always an integer. It could be zero, but it can't be nothing. The same is true for other primitive types. Reference types, on the other hand, are allowed to store a reference that doesn't point to anything. The name of this invalid reference is `null`. The `getKey` method does what many methods will do that return a reference type. If there isn't a good value to return, it returns `null`. Just to see what it does, let's modify the `act` method in `Person` so that it uses `getKey` instead of `isKeyDown`. A first cut at this modification might look like the following.

```
public void act() {
    int x = getX();
    int y = getY();
    if(Greenfoot.getKey()==leftKey) {
        x = x-1;
    }
    if(Greenfoot.getKey()==rightKey) {
        x = x+1;
    }
    if(Greenfoot.getKey()==upKey) {
        y = y-1;
    }
    if(Greenfoot.getKey()==downKey) {
        y = y+1;
    }
    setLocation(x, y);
}
```

This is basically the same logic we had before, but now instead of checking the value of `isKeyDown` for each key, it is checking if the return of `getKey` is the same as the keys we are looking for.

It turns out that this rather naive approach doesn't work. There are several reasons why. You should put in this code and run it to see what happens. The answer is nothing. Most likely, no matter what key you press, the people

don't move. This points us to the first thing that we have done wrong. The == operator is a great choice when working with ints. However, it typically doesn't do what you want for reference types. This is because == tells you if two references are the same, not whether the things that they reference are equal. If you think of a reference as storing the address that you can find something at, == checks to see if two things have the same address. That often isn't what you want to check for. Most of the time you want to know if the two things living at the referenced addresses are the same. To do that you should call a method named equals. Every object in Java has this method, including strings. So the first change we need to make is to replace == with calls to the equals method. That makes the code look like the following.

```
public void act() {
    int x = getX();
    int y = getY();
    if(Greenfoot.getKey().equals(leftKey)) {
        x = x-1;
    }
    if(Greenfoot.getKey().equals(rightKey)) {
        x = x+1;
    }
    if(Greenfoot.getKey().equals(upKey)) {
        y = y-1;
    }
    if(Greenfoot.getKey().equals(downKey)) {
        y = y+1;
    }
    setLocation(x, y);
}
```

If you enter this code and run it now you get a different behavior. Instead of doing nothing, this version crashes. It brings up another window with a few lines of red text that begins by telling you that you have a java.lang.NullPointerException. This means that you tried to call a method on a null reference. Remember that null is an invalid reference. Because it doesn't reference anything, you can't do anything with it. When you try, you get this exception.

So the question is, where did we try to do something with null? If you look at the second line of red text, it will tell you. It says something like the following.

```
at Person.act(Person.java:33)
```

The part in parenthesis tells you that this happened in the file Person.java, where you were writing the Person class. In this case, it was on line 33. Your line number might be different. When you go look at that line you will see that it is the first if statement in the act method. So where is null coming from? It is being returned by Greenfoot.getKey(). After all, that is what the API told us it would do if no key had been pressed since the last time this was called.

Our code crashed the first time it was called because no key had been pressed at all. We can fix that by adding checks to see if we have a `null`. However, if you read closely what `getKey` says it returns, and you run through the code that we wrote, you will see that we have another problem as well. The `getKey` method consumes key presses. If we call it multiple times in the `act` method, the first call might get a non-null value, but all the other calls will certainly get `null`. To fix that problem we need to add another variable. That variable will have the type `String` and will be set with the value of `getKey`. Then instead of calling `getKey` repeatedly, we use the variable. That gives us code that looks like this.

```
public void act() {
    int x = getX();
    int y = getY();
    String key = Greenfoot.getKey();
    if(key.equals(leftKey)) {
        x = x-1;
    }
    if(key.equals(rightKey)) {
        x = x+1;
    }
    if(key.equals(upKey)) {
        y = y-1;
    }
    if(key.equals(downKey)) {
        y = y+1;
    }
    setLocation(x, y);
}
```

This still hasn't fixed the problem with `null` when no key has been pressed though. To do that we will use a little Boolean logic. Consider the first `if` statement. The condition says that it is `true` if `key` equals the `leftKey` string for that person. We want to add the additional requirement that `key` not be `null`. So we want to move if `key` is not `null` and `key` equals `leftKey`. The following code shows that implemented in all the if statements.

```
public void act() {
    int x = getX();
    int y = getY();
    String key = Greenfoot.getKey();
    if(key!=null && key.equals(leftKey)) {
        x = x-1;
    }
    if(key!=null && key.equals(rightKey)) {
        x = x+1;
    }
    if(key!=null && key.equals(upKey)) {
        y = y-1;
```

```
    }
    if(key!=null && key.equals(downKey)) {
        y = y+1;
    }
    setLocation(x, y);
}
```

Note that the order of the two checks on either side of the `&&` matters. This is where the fact that `&&` is a short circuit operation comes into play. If `key` is `null` the first condition is `false` and the second condition is not evaluated. That is critical, because if `key` is `null`, the second condition is an error and we don't want it to happen. Feel free to put the `key!=null` checks on the right side of the `&&` and see what happens.[2]

This code should compile and run. However, it still doesn't work as well as what we had before using `isKeyDown`. When you run this code you will find that only one of the people moves. Why is that? It is because one person always has its `act` method called before the other. The one consumes the key press and the second person always has a `key` value of `null`. At this point you can probably see why we used `isKeyDown` to start with. It and `getKey` have very different behaviors and should be used in very different types of situations. At this point you can go ahead and revert to the code using `isKeyDown` as we will be coming back to that in a few sections.

3.4 The `switch` Statement

The `if` statement isn't the only conditional statement in Java. There is also the much less used `switch` statement. Technically, the `if` statement can perform any conditional execution that you want. There is no need for alternatives. However, the syntax of the `if` statement can be a bit verbose for certain types of situations where there are lots of possible cases, especially if those cases are all checking against the value of a single expression. The `switch` statement is used for exactly those situations.

The idea of the `switch` statement is that you need to perform one of many possible sets of instructions and which one you execute depends on the value of an integer or `String` expression.[3] The rule for the `switch` statement is shown below. The way `switch` works is that it evaluates the expression in parentheses, then runs through the different cases until it find the one that matches. If none match, it will do the default case.

SwitchStatement:

`switch(`*intOrStringExpression*`) {`

[2] In this example we don't have to use the `&&` to prevent the `NullPointerException`. Doing `leftKey.equals(key)` would also work as we know that `leftKey` is never `null`, and when `key` is `null`, the `equals` method will return `false`.

[3] Note that `char` is an integer type as well because characters are just numbers to the computer. What makes them characters is how they are treated.

`case` *intOrStingValue1* `:`

[*statements*]

`break;`

`case` *intOrStringValue2* `:`

[*statements*]

`break;`

`case` *intOrStringValue3* `:`

[*statements*]

`break;`

...

`default:`

[*statements*]

}

The `switch` statement allows you to branch to one of many options based on the value of an integer or `String` expression. The values don't have to start at 1 as I have shown here. They can be whatever you want. `break` statements are also optional, but they are almost always needed. `break` statements can also be used in loops to terminate the loop, but I won't ever do that.

An example use of `switch` that could occur in this book would be like the following.

```
public void moveDir() {
    int x = getX();
    int y = getY();
    switch(dir) {
    case 0:
        x = x+1;
        break;
    case 1:
        y = y+1;
        break;
    case 2:
        x = x-1;
        break;
    case 3:
        y = y-1;
        break;
    default:
    }
    setLocation(x, y);
}
```

The idea here is that our class has a member called `dir` that is an `int`. Instead of having four `if` statements to check the value of `dir`, we can use a single `switch` statement with four cases in it. The `Wombat` class in the previous chapter had this type of functionality in it, but it was not implemented with `switch` statements. For the most part the `switch` statement is mentioned here for completeness just in case you ever run across code that uses it. However, most uses of the `switch` statement can be done in better ways in Java so it will not be used extensively in this book.

3.5 Ternary Operator

One of the limitations of the `if` statement is that it is a statement. This is great when you want to do something depending on a condition. However, there are times when you would like the value of an expression to depend on some condition. The `if` statement can't do this. It isn't an expression and can't be used in places where you want a value. To see what the problem is here, consider the following code.

```
int b;
if(a < 5) {
    b = 34;
} else {
    b = 42;
}
```

All this code does is assign a different value to `b` based on the value of `a`. If the `if` statement could be used as an expression we could type in something more like this:

```
int b = if(a < 5) 34 else 42;
```

This isn't valid Java and if you try to enter it into a program you will get errors when you try to compile.[4] It is worth noting that there are other languages where this would work. Because it is so useful, there is a way of doing this in Java. It just uses more symbols instead of English words.

The way we do this is with the ternary operator. It is an operator that takes three arguments. The syntax was given in the rule for expressions earlier. The first operand is the condition, just like what you put in the `if` statement. After that is a question mark which is followed by an expression that should be the value if the condition is `true`. That is followed by a colon and then the expression if the value is `false`. So our example above could be typed into Java with the following syntax.

```
int b = (a < 5) ? 34 : 42;
```

[4]This type of syntax does work in many languages, including one called Scala that also runs on the Java platform.

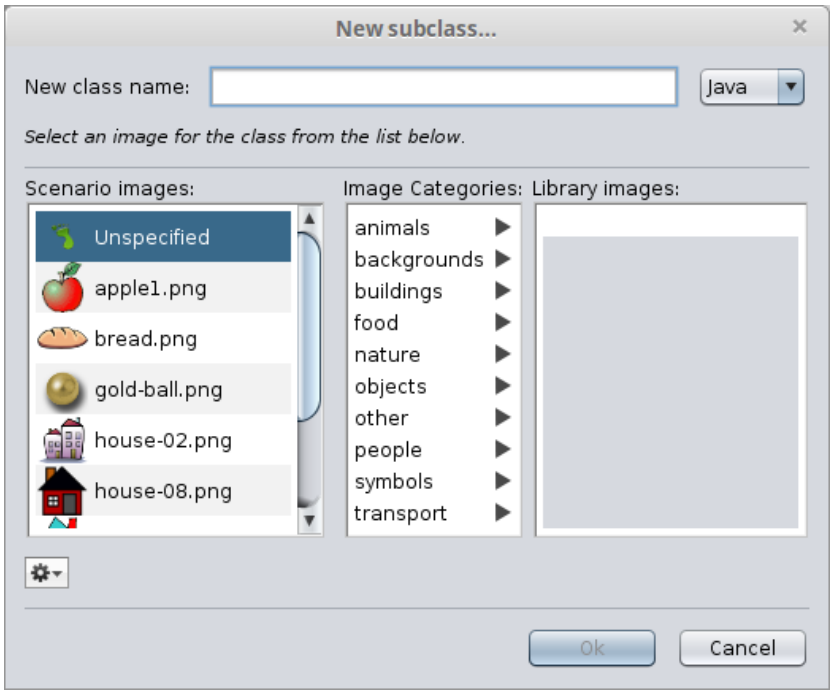

Figure 3.1: The new subclass dialog.

The problem with the ternary operator is that it can be hard to read. People commonly put in extra parentheses to make things more clear, but even then their meaning isn't always straightforward. For this reason you won't see ternary operators used all that much. They are far less common than the `if` statement, but when they are useful they can greatly reduce the length of code and so they do occur occasionally.

3.6 Our First Game (An Example)

Now it is time to extend our CityScape code so that it really has some functionality. Instead of just having the people move around, we'd like to make this into a game. The two people will compete against one another in this game. The idea is to have them race to pick up items. Whoever collects the most by the time they are all gone wins. Picking up the houses doesn't make much sense. We will use them in a different way later. That means that we need to introduce a new type of actor into our world, something that it makes sense for the people to pick up. To do this, right click on the "Actor" box that is above "House" and "Person". This will bring up a menu with two options. The second one is "New Subclass ..." When you select it, a dialog box like the one shown in figure 3.1 will come up.

We will call our new class "Food" then select the image category "food" and the "apple1.png" image. Once you have done this and clicked "Ok", a new class should appear under Actor with the name `Food`.

Like the `Leaf` in the PSPGWombats scenario, the `Food` class isn't going to do much of anything itself. The `Person` class will interact with it, much as the `Wombat` class interacted with the `Leaf` class. The class we want to edit is the `Person` class. We want to keep the code that we had earlier to do movement with `isKeyDown`, but we want to add onto that code that checks for whether we are standing on the same square as an instance of `Food`. If the person is at the same location as a food, we want to pick it up and add it to our count.

All of the code that we want to write could be put into the `act` method. Making really big methods though is generally considered bad form. What is worse, it often makes the code harder to understand. What we would like to do is decompose this problem into some major pieces and then write each of those separately. The `act` method can call each of these pieces as needed.

To do this we have make new methods. We saw the rule for methods in the last chapter when looking at code, but we haven't actually had to write new methods ourselves. We have only edited methods that were already provided for us. Proper problem decomposition requires us to create new methods so that is what we will do here. We will do this in a top down manner. First, we will edit the code in `act` and basically structure it so that calls other methods that don't yet exist. The idea being that once those other methods are written, the `act` method will work. For what we have described so far, the `act` method might look like the following.

```
public void act() {
    if(canPickUpFood()) {
        getFood();
    }
    keyboardMove();
}
```

Now all we have to do is write the three methods that this calls: `canPickUpFood`, `getFood`, and `keyboardMove`. We'll start with the last one because we have basically already written it. It should look something like this.

```
public void keyboardMove() {
    int x = getX();
    int y = getY();
    if(Greenfoot.isKeyDown(leftKey)) {
        x = x-1;
    }
    if(Greenfoot.isKeyDown(rightKey)) {
        x = x+1;
    }
    if(Greenfoot.isKeyDown(upKey)) {
        y = y-1;
    }
```

```
    if(Greenfoot.isKeyDown(downKey)) {
        y = y+1;
    }
    setLocation(x, y);
}
```

Notice that this is exactly what our act method looked like earlier. The only difference is that instead of having the method called `act`, it is now called `keyboardMove`. It is `public` since we are currently making all of our methods `public`. It has a return type of `void` because we don't need any information back from it. If you look at the `act` method, we didn't do anything with a return value so it makes sense that there shouldn't be one.

The `canPickUpFood` method does need to return something. This is because it is used in an `if` statement. Remember that the condition of an `if` statement has to be a `boolean`. So the return type of `canPickUpFood` needs to be `boolean` as well. Inside of the `canPickUpFood` method we need to answer the question of whether or not the person is standing on the same square as a food object. If you look at the API for Actor you will see that there is a method with the name, `getOneIntersectingObject`. This method takes one argument of type `java.lang.Class`. This might sound scary, but it really isn't. We can easy get a `Class` object by putting `.class` after the name of the Java class we want the `Class` of. If there isn't an intersecting object of that type, this method should return `null`. So if we pass in `Food.class`, it will give us a `Food` object if the person is on one. If the person isn't on one, it will give us `null`. So the `canPickUpFood` method, given the rules we are currently using might look like the following.

```
public boolean canPickUpFood() {
    return getOneIntersectingObject(Food.class) != null;
}
```

You might be tempted to write this method using an `if` statement like this.

```
public boolean canPickUpFood() {
    if(getOneIntersectingObject(Food.class) != null) {
        return true;
    } else {
        return false;
    }
}
```

These two version do the same thing, but the first one is a lot shorter and potentially faster as well. As a general rule, when you have an `if` where all it does is return `true` or `false` in the conditions, simply return the condition instead. After all, the condition is a `boolean` and it has the value that we want.

Last we will write the `getFood` method. This method should take the food out of the world and increment a counter that keeps track of how many food objects the person has collected. Removing an object from the world can be done

with the `removeObject` method in the `World` class. Since `Person` is an `Actor`, not a `World`, we will have to use the `getWorld` method of `Actor` as we did before when we wanted to call `addObject`. We also need to pass it the food object that we want to remove. We can get this object using `getOneIntersectingObject`, just like we did in the `canPickUpFood` method. The new code might look like the following.

```
private int foodCollected = 0;

public void getFood() {
    foodCollected = foodCollected+1;
    getWorld().removeObject(getOneIntersectingObject(Food.class));
}
```

Go ahead and put in these different methods and compile the code. You can manually add a few pieces of food to allow you to test and make sure that the code is working. Once you have added the food, click "Run" and move the people on to of some of the food. It should disappear. Once you have gotten rid of all the food, click "Pause" and then right click on one of the people and select the inspect option from the menu. This will bring up information on the person including the value of `foodCollected`. Using this, you can see which of the people had eaten more of the food. You can go into the `City` class and add a few lines to the constructor there so that every time you compile or reset, some food will be added to the world, perhaps at random locations. We will talk about a better way to do that in the next chapter.

Once you put in that code you actually have something of a game. It is a very simple game though, and we haven't used the houses at all. One way we could fix both of these problems is to make it so that a person can only carry a certain number of food at a given time and has to go to a house to drop off food so that more can be collected. The number of food instances a person can carry should be fairly small. It is interesting to note that if it were always 1, things could be coded a bit differently. We'll point out how as we edit the code.

To make this modification, we should start with the `act` method again. We start here because we want to see at a high level what pieces the problem can be broken into. There are a lot of different options for the modified act method. This is one of them.

```
public void act() {
    if(canPickUpFood()) {
        getFood();
    }
    if(canDropFood()) {
        dropFood();
    }
    keyboardMove();
}
```

The approach here is to simply add a check for whether we are able to drop the

food and if so, to drop it. To implement these methods we will have to add some member data that stores how much food the person is currently carrying. This value will also be used in modified versions of `canPickUpFood` and `getFood`. Note that if the player is allowed to hold no more than one food, this member data can be a `boolean` telling us whether or not the person is holding food. If it can be larger than one though, an `int` is needed. This code shows that member data, the two new methods, and the modified versions of the methods we had before.

```
private int foodCarried = 0;

public boolean canPickUpFood() {
    return foodCarried < 2 && getOneIntersectingObject(Food.class) !=
        null;
}

public void getFood() {
    foodCollected = foodCollected+1;
    foodCarried = foodCarried+1;
    getWorld().removeObject(getOneIntersectingObject(Food.class));
}

public boolean canDropFood() {
    return getOneIntersectingObject(House.class) != null;
}

public void dropFood() {
    foodCarried = 0;
}
```

This version of the code allows the player to carry two pieces of food at any one time. Let's run through each piece to see what is happening. First is the declaration of a new value that the `Person` remembers called `foodCarried`. After that, the `canPickUpFood` method has been altered so that the person not only has to be on food, but can't be carrying more than two food. The constant `2` here is what sets how many food objects a person can pick up before having to run to a house. Having random constants like this in your code is generally frowned upon. Because this one only appears in one place in the code, and because doing the other approaches properly requires introducing some new keywords that we won't bring in until later, we will leave it as just the number `2` for now.

The `getFood` method has one new line in it that increments the number of food items being carried. The two new methods are both quite simple. The first one returns a `boolean` to let us know if we are standing on a house and the second one sets that amount of food being carried to zero. Put in this code and run it. Make sure it works for you and that you understand what it is doing and how it works.

3.7 The Mandelbrot Set (An Example)

Before we leave this chapter on conditionals, there is one more little example that we will do. In many ways this example is much simpler than our last one. The reason for doing it is that it is a very different type of example. We are going to have Greenfoot actors form the Mandelbrot set for us. The Mandelbrot set is probably the most famous of all fractals. It is a set of points in the complex number plane and the fractal is named after the mathematician Benoit Mandelbrot.

Any time you solve a problem, whether it is on the computer or elsewhere, the first step is to figure out exactly what problem you are solving. In computer science this step is commonly called analysis. In this case, we want to make a program that will show us a graphical representation of the Mandelbrot set. This set as defined to be all points in the complex plane such that the iteration of a particular equation stays bound instead of going off to infinity. The equation is $z_{i+1} = z_i^2 + c$, where c is the point we are considering in the plane and $z_0 = 0$.

Note that the z and c values are complex numbers. For this reason it is probably worth doing a quick math refresher. Most of the time we work with real numbers. These are the numbers on the standard number line. You can do almost any type of mathematical operation you want on real numbers. There are exception though. The main exception related to complex numbers is that the square root operation is not closed over the reals. That is to say that there are some real numbers that when you take the square root, you don't get back a real number. In particular, the square root of any negative number can't be a real number, because any real number squared is a positive real number. [5] To fix this, we have imaginary numbers. The number i is defined to be the square root of -1. So the square root of -4 is $2i$ and so forth. The imaginary numbers can be laid out on a number line, just like the real numbers.

Complex numbers are simply the sum of a real number and an imaginary number. For example, $3 + 5i$ is a complex number. Complex numbers are often represented on a plane with the x-axis being the real number line and the y-axis being the imaginary number line. So our example of $3 + 5i$ would be located three units to the left and five units above the origin.

When we add complex numbers we simply add the real and imaginary parts independently. So $(3+5i)+(4-3i) = 7+2i$. More generally, $(a+bi)+(c+di) = (a+c)+(b+d)i$. When we multiply complex numbers you just use the standard FOIL rule that you learned in algebra. So $(a+bi)*(c+di) = ac+adi+bci-bd = (ac-bd)+(ad+bc)i$. The formula for the Mandelbrot set has us square one value, then add it to another so we will be doing both multiplication and addition.

To keep things simple, we will say that a point is not in the set if the magnitude of z_i ever becomes larger than 2. We will also only go for a very simple graphical representation. If you search on the Internet for the Mandelbrot set you will see lots of pretty pictures with nice color gradients and even animations

[5] This problem is more general than just square roots. Most generally, it can be a problem for any exponent that isn't a whole number. Recall that a square root is the same raising the value to the $\frac{1}{2}$ power.

that show flights through the set. We are only going for a small black and white image.

Now that we know what problem we are solving, we can think about how to solve it. This stage of problem solving is called design in computer science. Like analysis, this step doesn't involve writing any code. During design you should just think about how you can solve the problem. You are thinking about what you would put into the code, but you aren't really writing any code. The real question is, what do we know how to do in Greenfoot? We know how to put actors in the world. We know how to move them. We know how to remove them. We have the ability to do some math and do conditional logic. So what if we made an `Actor` type called `Point` and made the image for `Point` just a single dot on the screen. Then we could fill the world with a grid of these points. Each point could know its c and z values. The `act` method for the point could do the Mandelbrot iteration formula and check the magnitude of the point. If it gets too big, it can remove itself from the world.

So that is our design. Load in the PSPGMandelbrot scenario and we will implement it. Some parts have already been implemented. The `ComplexPlane` class deals with setting up the grid of `Point`s. For now we will just assume that code works. In the next chapter we will look more closely at it. What we want to do is implement the `Point` class. We know from our design that the point needs to remember the c and z values for it. Java doesn't have a type for complex numbers. Instead, we will store the real and imaginary parts independently in variables of type `double`. We use `double` because these aren't going to be whole numbers and the `double` type is as close as we get to real numbers. So our member data declarations might look something like this.

```
private double cr;
private double ci;
private double zr;
private double zi;
```

The `act` method for Point needs to do the Mandelbrot iteration on the z values, then check to see if the magnitude of the z value is greater than 2. If it is greater than 2 then we want to remove this point from the world. Here is an `act` method that does exactly that.

```
public void act() {
    double tmp = zr;
    zr = zr*zr-zi*zi+cr;
    zi = 2*tmp*zi+ci;
    double mag = zr*zr+zi*zi;
    if(mag > 4) {
        getWorld().removeObject(this);
    }
}
```

The first three lines do the math for the Mandelbrot. We have to introduce a temporary variable, called `tmp`, to store the old value of `zr` that we will need

to calculate `zi` after we have changed the value of `zr`. After those lines is a line that calculates the square of the magnitude and checks to see if it is greater than `4`. If it is, `ti` removes this point from the world.

The line that removes the current point form the world includes something new that we haven't seen before. It is the keyword `this`. The `this` keyword acts like a variable that references the object that the method you are in was called on. It is implicitly used in a lot of the code we have already written. The `act` method could actually be rewritten with this explicitly put in before every reference to member data or methods. That would make it look like the following.

```
public void act() {
    double tmp = this.zr;
    this.zr = this.zr*this.zr-this.zi*this.zi+this.cr;
    this.zi = 2*tmp*this.zi+this.ci;
    double mag = this.zr*this.zr+this.zi*this.zi;
    if(mag > 4) {
        this.getWorld().removeObject(this);
    }
}
```

Some developers like this style as it makes it explicitly clear when you are referring to something at the class level. As you can see here though, it can make things far more verbose which is why we have not adopted that style.

If you put this code inside of the `Point` class and try to compile you will actually get an error in the `ComplexPlane` class. It is on the line that tries to make a new `Point` object. That is because we haven't included a constructor that allows the `ComplexPlane` class to set the values of `cr` and `ci`. That constructor might look like the following.

```
public Point(double real_0, double img_0) {
    cr = real_0;
    ci = img_0;
}
```

Once you have included that you should be able to compile the code. Click on the "Act" button a few times and see what happens.

The first thing that you are likely to notice is that this runs really slowly, even though the image that we are generating is rather small. That is because the approach we designed isn't exactly the best way to solve this problem. It has a lot of overhead due to the fact that every little point in the picture is represented by a `Point` instance. In fact, that is why the image isn't any bigger. If we made it much bigger, you would likely get errors saying that Java had run out of memory. Once we have learned more different things that way can do in Greenfoot we can come back to this problem and write more efficient solutions.

Chapter Rules

***Statement*:**

VariableDeclaration | *Assignment* | *MethodCall* | *ReturnStatement* | *CodeBlock* | *IfStatement* | *SwitchStatement*

***IfStatement*:**

`if(`*booleanExpression*`)` *statement1* [`else` *statement2*]

This is the most basic conditional statement. If the boolean expression is true, *statement1* will execute. If it is false either nothing happens of *statement2* is executed if the else clause is used. A common usage might look like the following:

```
if(a == 5) {
    b = 25;
}
```

***Expression*:**

An expression in Java is anything that has a type and value. Below is a list of options.

- Operator expression – Java supports different mathematical and logical operators. Most will join two different expressions. Some operate on one expression and one uses three expressions. This is a list that includes most of the ones we are likely to use.
  - A==B : This returns true if A is the same as B. For object types use .equals instead.
    - A<B : Returns true if A is less than B. Works for numeric types.
    - A>B : Returns true if A is greater than B. Works for numeric types.
    - A<=B : Returns true if A is less or equal to than B. Works for numeric types.
    - A>=B : Returns true if A is greater than or equal to B. Works for numeric types.
    - A!=B : Returns true if A not equal to B.
    - A || B : This boolean operator is true if A or B is true. It is inclusive or so it is true if both are true.
    - A && B : The boolean operator returns true only if both A and B are true.
    - !A : This boolean operator is the not operator. It returns true if A is false and false if A is true.

- A?B:C : This is the ternary operator. A should be a boolean. If A is true, the value will be B, otherwise the value will be C.
- A instanceof C : Here A is an expression with an object type and C is the name of a class. It tell you if A is of the type C.

SwitchStatement:

`switch(`*intOrStringExpression*`) {`
`case` *intOrStringValue1*`:`
[*statements*]
`break;`
`case` *intOrStringValue2*:
[*statements*]
`break;`
`case` *intOrStringValue3*`:`
[*statements*]
`break;`
...
`default:`
[*statements*]
}

The switch statement allows you to branch to one of many options based on the value of an integer expression. The values don't have to start at 1 as I have shown here. They can be whatever you want. Break statements are also optional, but they are almost always needed. Break statements can also be used in loops to terminate the loop, but I won't ever do that.

Exercises

1. Walk across world and wrap
2. Enter houses
3. Have people walk randomly. They make buildings if they come together and there isn't one there.
4. Have people walk randomly, but make it so that they never go onto the same square as a house or another person.

In class we start by fixing the walkToward so that it doesn't have a divide by zero. We also then put in code for making the person disappear when it got to the destination. If done with coordinates, this is an example that needs to use &&.

Chapter 4

Loops and Greenfoot Lists

There have been a number of different times already where we have come across situations where we need to repeat a certain action multiple times. For example, in the CityScape scenario we wanted to add multiple buildings. The way we accomplished this was to take the line that added one and cut and paste it multiple times. There are many reasons why this is a bad solution. In this chapter we will look at better solutions. We will also learn how to work with the `List` type which are used in a number of places by Greenfoot.

4.1 Need for Loops

Before we look at how we should do repetitive actions in Java let's go through the reasons why copy and pasting code is so bad. There are two reasons that are easy for the novice programmer to understand. First, if you have to do something a lot, the copy and paste method isn't very efficient. In the case of the Mandelbrot program from the last chapter, it added 40,000 `Point` objects to the world. Each one of those had slightly different values. Doing that by hand would be insane. That's the type of thing computers are good at and humans aren't. Still, the copy and paste method could have been used as long as you were really patient. What about the situation where you don't know how many times you want to do something? Consider the following method signature for the `City` with our game.

```
public void addFruit(int howMany)
```

This would be for a method that is supposed to add a specified number of fruit. If we call it and pass in `5` it should add five fruit. If we call it and pass in `10` it should add ten fruit. How would you do this with cut and paste? It can be done if you put every `addObject` call inside of an `if` statement and have each on check for a larger input, but even that has limits because you probably won't cut and paste that more than a few times and in theory someone could call it with an input parameter of 1000.

So the cut and paste approach lacks flexibility and is extremely tedious. There is another fundamental problem, though that is less obvious to the novice programmer. Every line of code you write is a line of code that could have an error or that might need to be modified later. This is part of why we like solutions to things that are both general and short. Imagine you use the cut and paste method and made 100 copies of some line of code. After doing this, you find that your original line had an error in it. Now instead of having one error, you have 100. Even if you were really careful and made sure the first line was correct, you might find later on that you need to it do something slightly different. Instead of having one line to change, now you have 100.

It is something of a general rule of programming that if you find yourself copying a pasting code more than once or twice, you should probably sit back and think if there is a better way of doing whatever you are doing. This is because having multiple copies of code that do nearly the same thing almost always leads to problems down the line. For now we will use loops to help us avoid this. That only gets the simple cases though. Later in this book we will talk a little bit about a concept called abstraction and see how it can help get us around a bunch of other cases.

4.2 The `for` Loop

To start with, let's consider the simplest case, a situation where we want to do something a specified number of times. The method, `addFruit`, mentioned above is a great example to begin with. We know what statement we should execute in order to add a single `Food` object to a random location in the world. What we need is some construct that will allow us to have that statement execute a specific number of times. The most common way to do that in Java is with a `for` loop. The `for` loop is actually very general, and allows us to do many different things. However, it is most commonly used to count. In this case, that is exactly what we want do to. The first cut at writing the `addFruit` method might look like this.

```
public void addFruit(int howMany) {
    for(int i = 0; i < howMany; i++) {
        addObject(new Food(),Greenfoot.getRandomNumber(getWidth()),
            Greenfoot.getRandomNumber(getHeight()));
    }
}
```

Like the `if` statement, a `for` statement starts with the proper keyword and is then followed by stuff in parentheses. Unlike the `if` statement, the `for` statement has to have three different things in the parentheses, and they need to be separated by semicolons. When you are writing a counting loop it will often look like this. The first thing inside the `for` loop will declare a loop variable and give it an initial value. Then there is a semicolon followed by a `boolean` condition that tells it how far to count. That is followed by another

semicolon and a statement that increments the counting variable. Note that `i++` means exactly the same thing as `i = i+1`. The `++` operator is called an increment operator. There is also a decrement operator, `--`. These operators are commonly used in Java as shortcuts. We will use them in `for` loops and any place else that we need to increment or decrement values through the rest of this book. This isn't required. If it makes more sense to you, you can certainly have the last part of the `for` loop written with `i = i+1`.

!!! Consider an advanced aside here that describes the difference between pre and post increment/decrement.

So what really happens when the `addFruit` method is called? The way the `for` loop works, when it is reached, the initialization statement is executed. In this case that means that the variable `i` is declared and set to `0`. Then the condition is checked. Note that the condition is checked before the body of the loop ever executes. The `for` loop is a pre-check loop. In this type of loop, it is possible for the body to never execute. For example, if you call this method with an argument of `-1`, no food objects will be added to the world because the condition, `0 < -1` is false.

If the condition is `true`, the body of the loop is executed. After the body of the loop, the iteration statement is executed and the condition is checked again. This will repeat as long as the condition is `true`. If the condition is `false`, the loop stops and the program continues with the line below the loop. In this case, that means the method is done and it exits.

If you call this method and pass in `5` for `howMany`, note that `i` will start at `0`, then go to `1`, `2`, `3`, and `4`. After the body is executed with `i == 4`, `i` is incremented to `5` and the loop stops. It is common practice to have loops in Java start counting at zero, just like we saw that the x and y coordinates of the world start at zero. We will see later in this chapter and in the next chapter that other things in Java begin at zero as well. It would be equally valid to rewrite the `for` loop in this example like this.

```
for(int i = 1; i <= howMany; i++) {
```

This version would count from 1 to 5 instead of 0 to 4. There are some situations where you might want to do this, but most of the time we will stick with loops that begin counting at 0. After all, they require one fewer keystrokes because they don't need the equal sign in the condition.

Now let's look at a slightly more complex example. Open up the `ComplexPlane` class that was in the Mandelbrot scenario. The constructor for that class added all the `Point` objects into the world. In that case, there were a fixed number of points so technically cut and paste could have been used. In practice though, the fact that there were 40,000 points being added and that each call needed to use slightly different values make it impractical. Here is the part of that method that contains the for loops.

```
double realMin = -1.5;
double realMax = 0.5;
double imgMin = -1.0;
```

```
double imgMax = 1.0;

for(int i = 0; i < getWidth(); i++) {
    double real = realMin+i*(realMax-realMin)/getWidth();
    for(int j = 0; j < getHeight(); j++) {
        double img = imgMin+j*(imgMax-imgMin)/getHeight();
        Point p = new Point(real, img);
        addObject(p, i, j);
        p.setImage(pointImage);
    }
}
```

You should automatically see that this code has two `for` loops in it and that they are nested one inside of the other. Just like with `if` statements, the body of a `for` statement can contain any types of statements. So the body of the `for` can contain others `for`s. It could also contain `if`s, `switch`es, or any other valid Java statement. The generality of statements provides Java, and most other programming languages, with a lot of flexibility and gives you as a programmer much greater expressivity.

While it is simple to say that because a loop is a statement, it can be nested inside of other loops, it is worth taking a second to look at what really happens when loops are nested. We said above that this code adds 40,000 points to the world. How does that happen? Both width and height are only 200. So neither loop ever counts above 200. To understand what is going on, we need to trace through the code. We start with the declarations of four variables, then we hit the outer `for` loop and we set `i = 0`. Since `0 < 200` the body of the loop executes and another variables is declared and initialized. Then we hit the inner loop and it does `j = 0`. Since `0 < 200` the body of this loop executes. It executes 200 times in fact as `j` goes from 0 to 1 to 2 and on up to 199. It stops when `j` gets to 200. At that point the body of the outer loop is also done so it does `i++` and we now have `i = 1`. It then goes back through, sets `j = 0` and counts up to 200 again. So when we nest loops, every time that the outer loop goes through one iteration, the inner loop will execute all the way through. This tends to make it so that the body of the inner loop happens a multiplicative number of times. In this case, each loop counts to 200 and $200 * 200 = 40,000$. One last point about nesting loops is that the inner loops generally need to have different loop variable names. If you tried to change the `j` to an `i`, this code would not compile. It would tell you that you had declared `i` twice, which indeed you would have.

The purpose of this code was the lay out a grid of points both on the Greenfoot world and in the complex number plane. The `i` and `j` counting variables run through the grid of the Greenfoot world. The `double` variables `real` and `img` are given values that correspond to the x and y axis values in the complex plane inside of the range that is specified by the four variables at the top. You can actually change the values of those variables to see different parts of the Mandelbrot set.

The math that goes into calculating `real` and `img` is also worth taking a

few moments to look at. It does a linear scaling. In both cases, the math looks something like this: `v = min+i*(max-min)/count`. When `i = 0` in this formula, we have `v = min+0*(max-min)/count=min`. When `i = count` we have `v=min+count*(max-min)/count = min+max-min = max`. As the value of `i` goes up from `0` to `count`, the value of `v` goes from `min` to `max`. This is a fairly common formula to see as linear interpolations like this are fairly common in mathematical settings.

In it's full general form, the `for` statement follows the rule below. The first and last statements in the parentheses can actually be any single statement that you want. The middle is any expression that has the type `boolean`. You are also allowed to leave any of these elements blank. Leaving the middle one blank isn't generally a good idea because it will always evaluate to `true` and the loop will go on forever. We will come back to this general form of the `for` loop in a few sections to contrast it will one of the other loop types.

ForStatement:

`for(`*initStatement*`;` *booleanExpression*`;` *iteratorStatement*`)` *statement*

Before we move on to our next type of loop, there are a few more operators in Java that we should discuss because they are commonly used with loops. First, there is a set of assignment operators. We just saw that `++` and `--` combine doing an addition or a subtraction with an assignment. These always increment or decrement by one. Because it is common to do an operation on a variable and store the value back in the variable, Java provides shortcut operators to do this with all of their normal operators. All you have to do is follow the normal operator by an equal sign. So `+=` and `-=` increment and decrement by the value of the expression to the right of the operator. You can also use `*=`, `/=`, `%=`, or other operators that we haven't talked about at this point.

One last detail of the Java `for` loop is that you can use a comma to separate statements in the iterator statement. Consider the case where we want to increment two variables, `a` and `b`. If we were writing this normally inside of a method we could write the following as two statements.

```
a++;
b++;
```

If we need to do this as the iterator of a `for` loop (and yes, this does happen occasionally) you can't put a semicolon between them because the syntax of the `for` loop only allows two semicolons. In that case, you can separate the two with a comma instead. Due to the challenge of coming up with examples of these features in something that is Greenfoot based, the following code prints something to screen. This method will print the first ten powers of 2.

```
public void printPowersOfTwo() {
    for(int i = 0, n = 1; i < 10; i++, n*=2) {
        System.out.println("2^" + i + "="+n);
    }
}
```

Printing is something that we haven't seen previously, but it not only helps with examples, it can help a lot when you are writing code for Greenfoot and things aren't behaving the way you expect. Debugging, as we often call the process of finding a removing bugs, in a Greenfoot program can be challenging because the only feedback we normally get is in the movement of actors. Using the inspect option can show you other values, but that can get very tedious. If you include a print statement, Greenfoot will open another window and show you what you have printed out.

`System` is a standard class in Java. It has a member data called `out` that has a method in it called `println`. You can print basically any type of information that you want this way. In this case, we are printing a `String`.

Note the way this example works. We have declared two variables, `i` and `n`, that are both of type `int`.[1] Each time through the loop we want to increment `i` and double `n`. This demonstrates how we can do two things in the iterator and also demonstrates the usage of the `*=` operator. The expression `n *= 2` is equivalent to `n = n*2`. You should add this method to one of your classes in Greenfoot and call it manually to see what happens.

4.3 Lists

Earlier we looked at the Wombat code in the PSPGWombat scenario and saw that the Wombat has fairly simple logic controlling how it moves. It winds up running around the edge of the screen and missing most of the leaves. You were able to manually make it so that the wombat would walk around and eat all the leaves. As we discussed though, the solutions you came up with likely only worked for the specific configuration of leaves that we started with. If the leaves were moved or more were added, some would likely be missed. What we want to do now is fix this problem and make it so that the code in the `Wombat` class will direct a wombat to eat all the leaves in the world, no matter the configuration. We also want to make it so that the way this is done is at least reasonably efficient in how many times the wombat has to act. We aren't going to go for a truly optimal solution. That could be a problem for later, but it is a hard enough problem that we aren't going to try to solve it now. For now we simply want a good solution even if it isn't the best.

The way we are going to go about doing this is to change what happens if the wombat isn't standing on a leaf in the `act` method. When the wombat is on a leaf, it will eat it, just like before. If not, it will find the nearest leaf and, if it is walking in the right direction, it will step forward, otherwise it will turn. So the logic looks very much like what was in the Wombat class earlier. We can write this method like this.

```
public void act() {
    if(foundLeaf()) {
```

[1] One limitation of the `for` loop is that if you declare multiple variables, they all have to be of the same type.

```
        eatLeaf();
    } else {
        Leaf leaf = findClosestLeaf();
        moveToward(leaf);
    }
}
```

Now we have two smaller problems we have to solve: find the closest leaf and move toward the leaf.

We saw previously how we can check to see if we are standing on the same square as a leaf using the `getOneIntersectingObject()` method. Unfortunately, that method isn't going to help us in this case because it only tells us about objects very close to us. We need to know about objects further away. What we really need is a way to get hold of all of the leaves in the world. There isn't any method in `Actor` that will do this with us. However, even the way it was described implies that this is a method that should be found in `World` instead. Sure enough, there is a method called `getObjects()` in the `World` class. This method, like `getOneIntersectingObject()`, takes an argument of the `Class` for the type that you are interested in finding. What it returns is something we have never seen: `java.util.List`. We will call this just by the class name, `List`. The `java.util` part is a package name. Code in Java is organized into packages to help both people and the compiler find things. The `java.util` package contains a number of helpful classes, including different collections for storing things including `List`.[2]

The `List` class in Java works a lot like a list that you might make for something like groceries. Such a list stores a number of different things and allows you to interact with it in different ways. For the grocery list to be useful, you need to be able to do things like add stuff to it, remove things from it, and look through it. The following table was copied from the API page for `List`. This is part of the larger Java API that we won't look at in full until later in the text. Many of these methods include things that we don't know about yet, but fortunately they don't matter to us at this point. A few of them give us the exact functionality we were talking about for our grocery list. The `add()` methods add things to the list. The `clear()` method removes everything from the list. The `remove` methods remove specific items from the list, the `get()` method lets us look through the list, and the `size()` method tells us how many things are on the list.

[2]Technically `List` is an interface and not a class. Interfaces are similar to classes in that they define types and have methods in them. There are significant differences though that we will not worry about at this point.

Method Summary	
`boolean`	`add(E e)` Appends the specified element to the end of this list.
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this list.
`void`	`clear()` Removes all of the elements from this list (optional operation).
`boolean`	`contains(Object o)` Returns true if this list contains the specified element.
`boolean`	`containsAll(Collection<?> c)` Returns true if this list contains all of the elements of the specified collection.
`boolean`	`equals(Object o)` Compares the specified object with this list for equality.
`E`	`get(int index)` Returns the element at the specified position in this list.
`int`	`indexOf(Object o)` Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
`boolean`	`isEmpty()` Returns true if this list contains no elements.
`int`	`lastIndexOf(Object o)` Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`E`	`remove(int index)` Removes the element at the specified position in this list (optional operation).
`boolean`	`remove(Object o)` Removes the first occurrence of the specified element from this list, if it is present.
`E`	`set(int index, E element)` Replaces the element at the specified position in this list with the specified element.
`int`	`size()` Returns the number of elements in this list.
`List<E>`	`subList(int fromIndex, int toIndex)` Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
`Object[]`	`toArray()` Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`<T> T[]`	`toArray(T[] a)` Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

In a number of these methods, including the `add` methods, you see `E` used as a type. This is the called a generic type and it is the type of what the `List` can

store. When we declare a variable of type `List`, we can tell Java what type it stores by putting that type in angle braces after the name `List`. So if we want a list of leaves we would declare the variable to have the type `List<Leaf>`. If you do this, then every place you see `E` in the API it will be treated as `Leaf`. So the `findClosestLeaf()` method can call `getWorld().getObjects(Leaf.class)` and get a list of leaves. Combining this with the variable declaration we get the following line of code to put in our method.

```
List<Leaf> leaves = getWorld().getObjects(Leaf.class);
```

If you add this into the code and try to compile, you will get an error message. This is because we haven't told the compiler where to go find the `List` class. We need to add a new import statement at the top of the file to do this. The following does what we need.

```
import java.util.List;
```

Now that we have a list of leaves, the question is, how do we figure out which one is closest? To answer this question imagine you are handed a list of cities and told to find the one that is closest to you. If you had a map, you might try looking at it to see which ones are closest, but that isn't a perfect method and that visual work is very high level. We aren't going to try to program a system like that. Instead imagine that you have a way to look up the distance from where you are to any of the other cities. Now how could you use that to find the closest city? Basically, you are going to run down the list and check the distance to each city, then pick the smallest. That is still really high level. If you think about it, there are details that leaves out. As you go through the list, you are not just looking up the distance to each city in turn, you are also remembering which of the cities that you have looked at was closest, and perhaps the distance to that city so you don't have to look it up again.

With that we are just about close enough to make this into an algorithm. We start by having a memory of the closest city we have found so far and how far it is. Initially the city memory is blank because we haven't looked at any cities. Then we loop through all the cities on the list. If the city is the first one you have looked at, it is automatically the closest and you remember it and the distance to it. If it isn't the first, you compare the distance to the current city to the distance to the closest so far. If this one is closer, you forget the information on the old one and remember this one. When the loop is done, the city you are remembering is the closest one on the list. Let's convert that to code and see what it looks like.

```
public Leaf findClosestLeaf() {
    List<Leaf> leaves = getWorld().getObjects(Leaf.class);
    Leaf closest = null;
    int minDistance = 1000;

    for(int i=0; i < leaves.size(); i++) {
        Leaf leaf = leaves.get(i);
```

```
        int dist = calcDistanceTo(leaf);
        if(closest == null || dist < minDistance) {
            closest = leaf;
            minDistance = dist;
        }
    }
    return closest;
}
```

We begin by getting the list of leaves and then making variables to store the closest leaf and the distance to it. That is followed by a loop that counts from zero up to the size of the list. Note that here again we begin at zero and count up to one less than the number. We do this because that is how the items in a List are numbered. The `get()` method returns the first object in the list when we pass it 0.

Inside of the loop the first line calls the `get()` method on the list to get the next item from the list. It then calls a method, `calcDistanceTo()`, that we haven't written yet, to calculate a distance and store that in the variable `dist`. This is followed by an `if` statement that checks two things. The condition will be `true` if either this is the first object, in which case `closest` is still `null`, or the calculated distance is smaller than the minimum distance seen so far. If one of those is `true`, the current leaf and the distance to it are remembered. When the `for` loop is done, the value of closest is returned.

You should trace this method through to make certain you understand how it works. As a test of your understanding, consider what it would do if there were no leaves in the world. In that situation, the list won't have any elements in it and `leaves.size()` will return 0. What does the method return in that situation?

This method contained a call to `calcDistanceTo()` which was passed a `Leaf` object. There are many ways that this method could be written. By default you might consider using the Pythagorean theorem to return a normal Cartesian distance. This would work, but to some extent, it isn't a measure of the distance that our wombat will travel. That would be the distance if we moved on a straight line. We have so far only given our wombats the ability to move up, down, left, and right. That is something that we aren't going to change in this example. Indeed, Greenfoot locations have to be integers so making an `Actor` move on a smooth, straight line would take a bit more work. Given the style of movement, a better measure of distance is the separation in x plus the separation in y. That tells us how many times we will have to call `move()` in order to get from where to are to where the `Leaf` is. One thing to note is that the separations need to always be positive. If you just subtract one x value from the other, you will sometimes get a negative value. To fix this we need a way to take the absolute value. Java has a method that does this. It is called `abs()` and it is in the `Math` class. If we do this, the `calcDistanceTo()` method might look like the following.

```
public int calcDistanceTo(Leaf leaf) {
```

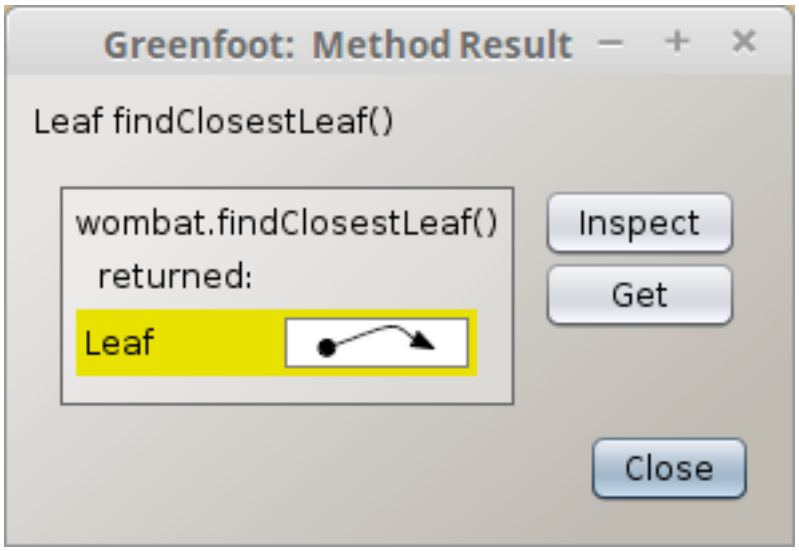

Figure 4.1: The method results dialog.

```
    return Math.abs(getX()-leaf.getX())+Math.abs(getY()-leaf.getY());
}
```

That finished off the `findClosestLeaf()` method. You should build a scenario and call it manually a few times just to make sure it doesn't crash. Greenfoot also provides two ways for you to tell if it really is returning the closest leaf. The display that pops up looks like Figure 4.1. Because `Leaf` is a reference type, all Greenfoot shows us is an arrow saying that it references something. To the right of this are two buttons: "Inspect" and "Get". If you click on "Inspect", it will bring up the Inspection window on the `Leaf` object that was returned. That will let you see the x and y coordinates which you can use to tell you if you got the right one. Alternately, if you click the "Get" button, when you move your mouse around the world, the leaf that was found will also move. Do this with a few different configurations so that you can make sure it works fairly generally.

Now all we have left to do is write the `moveToward()` method. When this was called in `act()` at the beginning of this section it was passed the `Leaf` object that we want to move toward. What we need to do in this method is make the `Wombat` step toward the leaf or turn, depending on whether it is facing a direction that will move it toward the leaf. This might be written in the following way.

```
public void moveToward(Leaf leaf) {
    if(facingGoodDirection(leaf)) {
        move();
     } else {
        turnLeft();
```

```
    }
}
```

Note that as we often do, we have included a call to a new method that doesn't yet exist, `facingGoodDirection()`. This method should return a `boolean` to us letting us know whether the direction we are currently facing will move us toward the leaf. In order to determine if the wombat is facing a direction that will take it toward the leaf, we will use the `getXOffset()` and `getYOffset()` methods that were already in the class. These method take a direction as an argument, and tell us how far we should move in x and y if we take a step. So when the direction has us heading up, `getXOffset()` returns 0 and `getYOffset()` returns -1.

The logic in `facingGoodDirection()` needs to check if the offset for our current direction moves us toward the leaf either in x or in y. As long as it is good for one axis it is appropriate to take a step that way. So how do we figure out if the offset will move us toward the leaf? There are many ways to do this and they have different complexities. The most obvious way is to write a bunch of `if` statements that do lots of checks comparing the offset values to the distance values. This way is also probably the most complex, the least efficient, and the more likely to be messed up. We will instead consider two alternate approaches.

The first one utilizes the fact that we have already written a method to find distance. Pretty much by definition a good step is one that decreases the distance. A bad step won't. So what we can do is make variables to remember the current position and distance, then move and calculate the modified distance. We then jump back and return whether the new distance was shorter than the old distance. This method might look like the following.

```
public boolean facingGoodDirection(Leaf leaf) {
    int cx = getX();
    int cy = getY();
    int cdist = calcDistanceTo(leaf);
    move();
    int ndist = calcDistanceTo(leaf);
    setLocation(cx,cy);
    return ndist < cdist;
}
```

The logic here is a little convoluted because we wind up moving and then going back to where we had been. This is because `facingGoodDirection()` isn't supposed to move us, it is only supposed to check if our direction is good. This method could be made nicer if `calcDistanceTo()` didn't always use our current position. If it allowed us to pass in a position, we could rewrite `facingGoodDirection()` in a more compact way. The following is a modified version of `calcDistanceTo()` that takes two additional arguments, an `x` and a `y`, and uses those in place of `getX()` and `getY()`.

```
public int calcDistanceTo(int x,int y,Leaf leaf) {
```

```
    return Math.abs(x-leaf.getX())+Math.abs(y-leaf.getY());
}
```

Note that you can put this method in your class without erasing the old one. That is because Java allows methods to be *overloaded*. That means you can have two methods that have the same name as long as the arguments passed to them are different. If you call `calcDistanceTo()` and only pass it a `Leaf`, Java can easily tell that it should use the first version. If it is called with two `int`s and a `Leaf` Java will use this new version.

Using this modified `calcDistanceTo()` method we can go back a rewrite `facingGoodDirection()` such that it doesn't actually move the `Wombat` and then pull it back. The modified version looks like this.

```
public boolean facingGoodDirection(Leaf leaf) {
    int cdist = calcDistanceTo(leaf);
    int ndist = calcDistanceTo(getX()+getXOffset(direction), getY()+
        getYOffset(direction),leaf);
    return ndist < cdist;
}
```

This version is a lot shorter and a lot easier to understand. Put this code into your `Wombat` class and see what happens. The wombat should walk to every leaf, always going to the closest one, and eat each one when it gets to it. It works just fine until the wombat eats the last leaf. At that point you get an error. What type of error is it? Why is it happening?

The problem is that our `moveToward()` method assumes that it actually has a leaf. When there are no leaves left in the word, the `findClosestLeaf()` method will return `null` and that leads to a `NullPointerException`. To fix this, we should make it so that the `Wombat` doesn't try to move toward the leaf if no leaf was found. This can be done by making a simple addition to our `act()` method.

```
public void act() {
    if(foundLeaf()) {
        eatLeaf();
    } else {
        Leaf leaf = findClosestLeaf();
        if(leaf!=null) {
            moveToward(leaf);
        }
    }
}
```

Let's consider one other method of writing `facingGoodDirection()`. This method isn't going to calculate distances. Instead, it is going to compare the direction to the leaf to the offset for the current direction. We are facing a good direction if the direction toward the leaf in x or y is the same as the offset along that axis. Checking if things are the same direction might seem like it

would require a fair bit of logic, but it turns out that we can do it nicely with just a bit of math. If you have two numbers and you want to check if they are either both positive or both negative, simply multiply them. If the result is positive, the two match. If it is negative they don't. Using this, we can write `facingGoodDirection()` as follows.

```
public boolean facingGoodDirection(Leaf leaf) {
    int dx = leaf.getX()-getX();
    int dy = leaf.getY()-getY();
    return dx*getXOffset(direction) > 0 || dy*getYOffset(direction) > 0;
}
```

The last line could be modified to use addition instead of the logical or as shown here.

```
return dx*getXOffset(direction)+dy*getYOffset(direction) > 0;
```

This version is basically doing a dot product on two vectors to see if the angle between them is smaller than a right angle. This is actually the more general solution. The other solution could run into problems if we extended the Wombat so that it could move along diagonals.

There is one last improvement that we could make to this code. We specifically wrote all of our methods to deal with leaves. By using the `Leaf` type for things like the argument to `facingGoodDirection`, we have produced methods that work well for this particular problem, but which aren't fully general. Nearly everything we did in this section would work more generally with the `Actor` type. Indeed, `Leaf` is a subtype of `Actor` because it inherits from it, so if we altered the code to work with `Actor`, it would continue to work with `Leaf` as well. The following code shows the complete `Wombat` class with `Leaf` specific code generalized to work with `Actor`. In most places this just required changing types. We also change variable names to make it so that things make sense. The one exception is the `findClosestLeaf()` method which we have to add a parameter to so that we can tell it what type of `Actor` we are looking for.

```
public class Wombat extends Actor {
    private int direction;
    private int leavesEaten;

    public Wombat() {
        setDirection(0);
        leavesEaten = 0;
    }

    /**
     * Do whatever the wombat likes to to just now.
     */
    public void act() {
        if(foundLeaf()) {
            eatLeaf();
```

```
        } else {
            Actor leaf = findClosestActor(Leaf.class);
            if(leaf!=null) {
                moveToward(leaf);
           }
        }
    }

    /**
     * Check whether there is a leaf in the same cell as we are.
     */
    public boolean foundLeaf() {
        Actor leaf = getOneIntersectingObject(Leaf.class);
        return leaf != null;
    }

    /**
     * Eat a leaf.
     */
    public void eatLeaf() {
        Actor leaf = getOneIntersectingObject(Leaf.class);
        if(leaf != null) { // eat the leaf...
            getWorld().removeObject(leaf);
            leavesEaten = leavesEaten + 1;
        }
    }

    /**
     * Move one cell forward in the current direction.
     */
    public void move() {
        if(canMove()) {
            setLocation(getX()+getXOffset(direction),
                getY()+getYOffset(direction));
        }
    }

    /**
     * Test if we can move forward. Return true if we can, false
         otherwise.
     */
    public boolean canMove() {
        World myWorld = getWorld();
        int x = getX()+getXOffset(direction);
        int y = getY()+getYOffset(direction);
        // test for outside border
        if (x <= myWorld.getWidth() || y <= myWorld.getHeight()) {
            return false;
        } else if (x < 0 || y < 0) {
            return false;
```

```
        }
        return true;
    }

    /**
     * Turns towards the left.
     */
    public void turnLeft() {
        direction = direction-1;
        if(direction < 0) {
            direction = 3;
        }
        setDirection(direction);
    }

    /**
     * Sets the direction we're facing.
     */
    public void setDirection(int dir) {
        direction = dir;
        setRotation(90*direction);
    }

    /**
     * Tell how many leaves we have eaten.
     */
    public int getLeavesEaten() {
        return leavesEaten;
    }

    /**
     * Returns the x offset for the specified direction.
     * @param dir The direction we want an offset for.
     * @return The x offset.
     */
    public int getXOffset(int dir) {
        if(dir==0) {
            return 1;
        } else if(dir==2) {
            return -1;
        } else {
            return 0;
        }
    }

    /**
     * Returns the y offset for the specified direction.
     * @param dir The direction we want an offset for.
     * @return The y offset.
     */
```

```
    public int getYOffset(int dir) {
        if(dir==1) {
            return 1;
        } else if(dir==3) {
            return -1;
        } else {
            return 0;
        }
    }

    public Actor findClosestActor(Class c) {
        List<Actor> list=getWorld().getObjects(c);
        Actor closest = null;
        int minDistance = 1000;
        for(int i=0; i < list.size(); i++) {
            Actor actor = list.get(i);
            int dist = calcDistanceTo(actor);
            if(closest==null || dist < minDistance) {
                closest = actor;
                minDistance = dist;
            }
        }
        return closest;
    }

    public int calcDistanceTo(Actor actor) {
        return
             Math.abs(getX()-actor.getX())+Math.abs(getY()-actor.getY());
    }

    public int calcDistanceTo(int x,int y,Actor actor) {
        return Math.abs(x-actor.getX())+Math.abs(y-actor.getY());
    }

    public void moveToward(Actor actor) {
        if(facingGoodDirection(actor)) {
            move();
        } else {
            turnLeft();
        }
    }

    public boolean facingGoodDirection(Actor actor) {
        int dx=actor.getX()-getX();
        int dy=actor.getY()-getY();
        return dx*getXOffset(direction)+dy*getYOffset(direction) > 0;
    }
}
```

4.4 The `for`-each Loop

In the `findClosestActor()` method we used a standard `for` loop to count through all the indexes in the list so that we could get out each element and do what we needed to with it. This is such a common activity that Java includes a second version of the `for` loop that does exactly this with a slightly shorter syntax. Not only is this syntax shorter, for some types of lists it can also be more efficient. This alternate style is called the for-each loop because it is used to loop through each element of a collection. The rule for the for-each loop is shown here.

ForEachStatement:

`for(`*Type name*`:`*iterableExpression*`)` *statement*

Inside of the parentheses you have what looks like a variable declaration followed by a colon and the thing you want to run through the elements of. The *Type* should be whatever type is stored in our list. The name is the variable name that we want the values to be called by inside the loop. After the colon can be any expression that evaluates to the type `Iterable`. For right now the only thing we know of that fits this description is a `List`.

Using this we could rewrite the `findClosestActor()` method as shown here.

```
public Actor findClosestActor(Class c) {
    List<Actor> list=getWorld().getObjects(c);
    Actor closest = null;
    int minDistance = 1000;
    for(Actor actor:list) {
        int dist = calcDistanceTo(actor);
        if(closest==null || dist < minDistance) {
            closest = actor;
            minDistance = dist;
        }
    }
    return closest;
}
```

This shortens the code a little and can make it easier to read. Note that you are never required to use the `for`-each loop instead of the normal `for` loop. Indeed, there are times when you really need the normal `for` loop because you need the index value. However, when the `for`-each loop can be used it is often a superior choice.

4.5 The `while` Loop

The first use of a loop in this chapter was for the `addFruit` method. Indeed, we have seen many occasions in this book where we wanted to add a number of objects at random locations in the world. The `for` loop is great for doing

this without having to copy and paste our statement and we were able to use random numbers to make it so that the objects go in random locations. There is just one problem with this. As you have probably seen, every so often the objects that are added are put at the same location. This is something that we would like to prevent. So the problem we want to solve is that we want to add a certain number of `Actor`s to the world without any of them being on the same location. How can we go about doing this?

We will begin with the same type of structure we had before. We want a `for` loop that counts up to the number of things that we are supposed to add. However, in this case, we can't just pick a random location and add the actor there. After we pick a random location we have to check to see if it is already occupied. If it is, we need to pick a different location. We should pick locations until we find one that isn't occupied. At that point we can add a new object to that location.

This repetition of picking locations is a bit different from the other repetitive actions we have done. Instead of counting and having the code happen a certain number of times, this one should occur indefinitely as long as a certain condition is `true`. That can be done with a `for` loop, but it isn't as natural. The `for` loop is the most commonly used loop statement in Java, but it isn't the only one. There are two others. The first of which is the `while` loop. The rule for the `while` statement is given below. In many ways, `while` is the simplest loop in the Java language. It isn't used as much as `for` in large part because it is too simple, and that simplicity makes it more bug prone for many uses. The `while` loop has a condition and a body. That is it, nothing else.

WhileStatement:

`while(`*booleanExpression*`)` *statement*

This simplicity is perfect for the problem we are faced with now. We want to keep picking random numbers as long as the location we have picked is already occupied. To put it a different way, while the selected location is occupied, we want to pick a different location. We can use the `Greenfoot.getRandomNumber()` method to calculate the random locations. All that we need is a way to determine if a particular location is occupied.

Earlier we did collision detection with the `getOneIntersectingObject()` method of the `Actor` class. That works fine if we want to check against an existing `Actor`. In this case though we are in the `World` and we want to find the location before we make the `Actor`. Since we are in a subclass of `World`, we should look there first for methods that might be helpful. There happens to be a `getObjectAt()` method in `World` that takes a `Class` as an argument. This method returns a List because there might be multiple `Actors` of that type at the specified location. If that location is empty, the `List` that is returned will be empty. So we want the loop to keep going as long as the list is not empty. This code shows a version of `addFruit` that does this.

```
public void addFruit(int howMany) {
    for(int i = 1; i <= howMany; i++) {
```

```
        int x = Greenfoot.getRandomNumber(getWidth());
        int y = Greenfoot.getRandomNumber(getHeight());
        while(!getObjectsAt(x, y, Food.class).isEmpty()) {
            x = Greenfoot.getRandomNumber(getWidth());
            y = Greenfoot.getRandomNumber(getHeight());
        }
        addObject(new Food(), x, y);
    }
}
```

Inside the `for` loop, this code declares variables for `x` and `y` and gives them random values in the proper range. The condition on the `while` loop tells it to keep going as long as the list of `Food` objects at (x, y) is not empty. Notice the exclamation point that indicates `boolean` not. Inside the loop is code that assigns new random values to the `x` and `y`. When the loop exits, a new `Food` object is created and added at the location that was picked.

There are some things that should be noted about this code. For example, the `while` loop can happen an indefinite number of times. If it keeps picking random positions that are already occupied, it will keep going and going. What happens if you use this method and ask it to add more houses than there are blank squares in the `City`? The answer is what is called an infinite loop. In that situation you will hit a point where the condition in the `while` loop can't become false. Then the program will try to run forever without doing anything. This will hang the program and it will have to be manually terminated. So you have to take care when using `while` loops to make sure that you don't create situations where they will become infinite loops. How could that be prevented in this code? The ease of creating infinite loops with the `while` loop is part of why the `for` loop is used more commonly. The two are actually equivalent. Consider the structure of the `for` loop:

`for(`*initStatement*`;` *booleanExpression*`;` *iteratorStatement*`)` *statement*

A `while` loop could be written using this structure that will do exactly the same thing.

initStatement `while(`*booleanExpression*`) {` *statements iteratorStatement* `}`

Most loops need to have some form of initialization and an iterator. Using a `while` loop, these things can easily be forgotten. The `for` loop contains them in its structure so they are harder to forget. In cases where you don't have one or both of these, or where they aren't easily expressed as single statements, people often use the `while` loop. However, you can empty statements in the `for` loop, basically allowing you to leave parts of it empty. In this way the `for` loop can mirror a `while` loop. Many people find that this looks odd though and will use a `while` loop instead.

Part of this functional equivalence comes from the fact that both the `for` loop and the `while` loop are pre-check loops. This means they check the condition before they execute the body of the loop. In a pre-check loop it is possible for the body of the loop to never be executed if the condition happens to be false the first time it is checked.

4.6 The do-while Loop

There is one other type of loop in Java, and it is the one you will see the least frequently. It is the `do-while` loop. The `do-while` loop is much like the `while` loop, but it is a post-check loop. This means that the body of the loop is executed before the condition is checked. As a result, the body of a `do-while` loop will always execute at least once. The rule for this loop is shown here.

DoWhileStatement:

`do` *statement* `while(`*booleanExpression*`);`

People use the `do-while` loop when the body needs to execute at least once, or when the body also does initialization. If you look back at the code for `addFruit()` using a `while` loop you notice that the values of x and y are set in two different locations. This means we duplicated the code to generate random positions on the screen. We could actually avoid doing that with a `do-while` loop. The way we had written that code, the body of the loop didn't always execute. However, the body was identical to the initialization statements so that logic did need to happen every time. Using a `do-while` loop we can rewrite this method as follows.

```
public void addFruit(int howMany) {
    for(int i = 1; i <= howMany; i++) {
        int x,y;
        do {
            x = Greenfoot.getRandomNumber(getWidth());
            y = Greenfoot.getRandomNumber(getHeight());
        } while(!getObjectsAt(x, y, Food.class).isEmpty());
        addObject(new Food(), x, y);
    }
}
```

This version is a bit shorter and a bit cleaner. Most importantly, it doesn't duplicate the code to generate random numbers. This situation where the body of the loop basically repeats the initialization code is probably the most common usage of the `do-while` loop. Because this doesn't happen all that often in practice though you aren't likely to run into this loop all that much.

4.7 Recursion

There is another way to get something to happen multiple times in our programs. !!! I have to decide if I really want to cover this here. I like it for pedagogical reasons generally, but I'm not certain I feel it belongs here in this book.

4.8 Details of Inheritance

You have

4.9 Review of Statement Types and Other Constructs

We now have a fair number of different statement types that we can put into our code. It is worth spending a little time reviewing these and some of the other constructs that we have talked about to remind ourselves of what each one looks like and also to consider the types of situations where we would want to use each one.

Chapter Rules

ForStatement:

for(*initStatement*; *booleanExpression*; *iteratorStatement*) *statement*

ForEachStatement:

for(*Type name*:*iterableExpression*) *statement*

WhileStatement:

while(*booleanExpression*) *statement*

DoWhileStatement:

do *statement* while(*booleanExpression*);

Exercises

1. dot scenario
2. maze scenario

Chapter 5

Collections of Data: Arrays and Lists

In the last chapter we saw how `List`s can be used to give us access to multiple different objects that we can easily process using a loop to run through them. There is more to `List`s than what we had discussed. `List`s aren't the only way to store multiple things in Java either. Another construct, called an array, is frequently used to store values. In this chapter we will see how to work with `List`s more generally, how to use arrays, and the differences between the two.

5.1 Arrays

`List`s are not the only way to store multiple values under a single name in Java. There are actually many types of collections that provide this type of ability. The most fundamental one is the array. It would be possible to go through this entire book without using arrays. Everything that we can do with an array can be done with a `List` and Greenfoot does not force us to use arrays the way it forces us to use `List`s. However, arrays are very fundamental in other programming languages as well, not just Java, so it is important that you be aware of them as you might run into them again in a later context. In addition, some things are simply easier to do with arrays than with `List`s. We will use them in this book to help us condense code and write code that is more flexible.

Arrays use a different syntax than `List`s. While `List`s are normal objects created from normal classes and the Java libraries, arrays have their own special syntax. This is a big part of what makes them easier to use in many situations. Array usage in Java is signified with square brackets. An array type is formed by putting square brackets after any other type in Java. The array types are themselves references types. This means we need to slightly alter one of our earlier rules to deal with array types.

ReferenceType:

ClassName | *Type*[]

Let's look at a few examples of variable declarations with array types. For these first few, we won't provide initial values.

```
int[] nums;
String[] names;
Person[] people;
```

These three declarations make variables for arrays of integer numbers, `String`s, and people. Like other reference types though, these can't be used until they are given values. There are actually two different ways that we can initialize array variables. The first is very much like how we would initialize any other reference variable using the new keyword. This usages is demonstrated here.

```
int[] nums = new int[10];
String[] names = new String[5*i];
Person[] people = new Person[numPeople];
```

Notice that we place values in the square brackets to specify the size of the array. This can be any integer valued expression. While it is common to use plain numbers like in the first example, you can use more complex expressions. These examples assume that the variables `i` and `numPeople` were previously declared to be `int`s. When created in this way, each element of the array is given a default initial value as if it had been declared as a member variable. This is to say that numeric values are set to `0`, `boolean` values are set to `false`, and reference values are set to `null`. So the arrays names and people declared here begin as all `null` values and they would have to individually initialized to real values before they are used.

There is also a special syntax that can be used to create and initialize arrays to values other than their default values. This syntax has you place the values for the array inside of curly braces. Here are our same three declarations using this alternate syntax.

```
int[] nums = {5, 4, 3, 2, 1};

String[] names = { "This", "is", "a", "test"};

Person[] people = {new Person(),new Person(),new Person()};
```

Note that this format is best for small arrays and arrays of values that have literals in Java. That is to say that it works well with primitives and `String`s. The `Person` type isn't as nice because we don't have a short syntax for expressing a `Person` object. We must use `new` to make then. What we will see in the end though is that arrays are most useful for collections that are small or when we are using primitives.

So now you know how to declare variables of array types. The syntax for passing them as arguments to methods is roughly equivalent, just place square brackets after the type you want an array of. The question now is how we access

the elements of arrays. It turns out that square brackets are used for accessing array elements as well. You place square brackets after the name of the array[1] and you need an integer expression in the brackets that tells what element you want to access. The elements are numbered from zero to one less than the number of elements in the array, just like the elements of a `List`. When used as an expression, this works like calling `get` on a `List`. When placed on the left hand side of an assignment, it works like calling `set` on a `List`. Let's look at some examples that assume the last set of declarations.

```
int a = nums[3]+5;
if(names[1].equals("is")) {
    nums[0] = 9;
}
people[0].setLocation(people[0].getX(), people[0].getY()+1);
```

This code shows the use of square brackets to access arrays in six different places. Five of those six are getting values out of the arrays and using them in expressions. These act just like calls to `get` on a `List`. The one inside of the `if` statement is doing an assignment and works like a call to `set` by altering the value at a particular location in the array.

These examples all use constants inside of the square brackets. While this makes for the simplest examples, it is not the most common usage. Typically we use arrays so that we can easily perform actions on multiple different things or access information based on variable indexes. Let's look at three different examples of methods that use arrays. The first is a method that takes an array of integers and returns the sum of all the elements of the array.

```
public int sumArray(int[] values) {
    int sum = 0;
    for(int i = 0; i < values.length; i++) {
        sum += values[i]; // This is the same as sum = sum+values[i]
    }
    return sum;
}
```

This code uses a `for` loop to run through all the elements of the array, in much the same way as we used `for` loops to run through all of the elements of `List`s in the previous chapter. There are two main differences. As we have already seen, we use square brackets to get values out of the array. In addition, the number of elements in an array is given by the member `length`. Note that there are no parentheses because `length` is not a method the way `size` is on a `List`. In this method we only get values from the array, we don't ever change them. Our second example will also work with an array of integers, but it will alter some of the value in the array.

[1] Technically you can place the square brackets after any expression whose value is of an array type. So if a method were to return an array, you could place square brackets after the parentheses of the method call. We will only use the simple form where the brackets are placed after the name of a variable or member data.

```
public void makeNonnegative(int[] numbers) {
    for(int i = 0; i < numbers.length; i++) {
        if(numbers[i] < 0) {
            numbers[i] = 0;
        }
    }
}
```

This method runs through the array and if it finds any elements that are negative, it sets them equal to zero. The assignment inside of the `if` statement alters the value of an element in the array. Imagine that this method were called with the following code.

```
int[] mixedNumbers = {2, -5, 6, 3, -7, 0, 8, -1};
makeNonnegative(mixedNumbers);
for(int i = 0; i < mixedNumbers.length; i++) {
    System.out.println(mixedNumbers[i]);
}
```

When the print statements occurred, values -5, -7, and -1 at indexes 1, 4, and 7 would be set to zero. The thing to note from this is that when you pass arrays into methods, those methods can alter the original arrays. This is true for `Lists` as well though we haven't run into code that does that yet.

The last small example we will look at will use `Strings` instead of `ints`. This method will take an array of `Strings` and a separator `String`. It will concatenate all of the `Strings` in the array together with the separator between them.

```
public String concatStrings(String[] words, String separator) {
    if(words.length > 0) {
        Stringret = words[0];
        for(int i = 1; i < words.length; i++) {
            ret += separator+words[i];
        }
        return ret;
    } else {
        return ""};
    }
}
```

A little extra code is required to handle the case where an empty array is passed in. Because we don't want an extra separator at the beginning or the end there should be one fewer separators than words. [2]

Early on we said that arrays are a bit more limited than `Lists`. The limitation is that the only things you can really do with an array are `get` and `set`

[2]This method of building long `Strings` by "adding" together a lot of little `Strings` isn't really ideal. It works and it isn't horrible for small array, but it turns out that putting `Strings` together requires building a new `String` and copying characters from the separate pieces. Doing this repeatedly is rather inefficient.

while a `List` has methods like `add`, `insert`, and `remove`. Because of this, arrays never change size after you create them. `List`s, on the other hand, can grow or shrink over time as needed. So you typically will only use arrays when you know up front how many elements you will need.

5.2 Using Arrays in Greenfoot

So now that we have seen how to make arrays and some simple examples on how to use them, it is now time to turn to a more significant example and do something useful in Greenfoot with arrays. We will start a new scenario and make it so that it plays a classic game of snake. In this game the user controls a snake with a body. Apples randomly appear in the world and the user is supposed to navigate the snake to eat the apples. Every time the snake eats an apple, the body of the snake gets one piece longer. The game ends if the snake ever runs into its own body.

In this game the snake is supposed to move around and that movement is supposed to be controlled by the user with the keyboard. At first that might sound like what we did with the Wombat having it move around with keyboard control. Unlike the Wombat though, the snake is supposed to move all the time and key presses are only used to change the direction it is heading. That means that the snake needs to remember what direction it is moving. There are many different ways that code for this could be written and the biggest variation is the way we want to store the memory of what direction we are moving. We could use a string that stores the name of the key for the direction, but we will instead use an `int`. This is in part because the `int` can be used as an index into an array.

If you open the PSPGSnake scenario you will see that it has a world and three different `Actor` classes in it. The `Actor`s are called `Head`, `Tail`, and `Apple`. The world starts off with one instance of `Head` and one of `Apple`. Our first goal is to make it so that the head moves in the proper way. We will worry about eating the apples and making the tail grow later in the chapter. As was discussed above, we need to add a memory of what direction the snake is currently moving and we will use an `int` to store that. It can be done by adding a line like the following to the code.

```
private int dir;
```

We also know that we need the code to respond to keyboard input. In an earlier chapter we did that with a series of different `if` statements that checked to see if different keys were being held down. We could do that again. We could have four different `if` statements that look something like the following.

```
if(Greenfoot.isKeyDown("up")) {
    dir = 0;
}
```

We would have one for each direction with values from 0 to 3. However, we can do that with less typing and a little different logic if we use an array. Note that we want to set `dir` to a value between 0 and 3. Each one of those different values corresponds to a direction and a `String` such as "up". To make this correspondence more concrete, we can declare an array of `String`s that holds the different directions. That declaration might look like the following.

```
private String[] directions = {"up", "right", "down", "left"};
```

Note that the `if` statement above uses one of the `String`s in the array and a numeric value that is the index of that `String`. In this case, "up" is at index 0. Similarly, "down" would set `dir` to be 2 and it is at index 2. So instead of having four different `if` statements, we can use this array with a loop and place the `if` statement inside of the loop like the following.

```
for(int i = 0; i < directions.length; i++) {
    if(Greenfoot.isKeyDown(directions[i])) {
        dir = i;
    }
}
```

This code would go inside of the `act` method. It sets the value of `dir` to be the index of the `String` in the array. In this way, it provides a mapping from the different `String`s into `int`s that we will use to control the motion.

Now the question is, how do we get from the `int` value in `dir` to motion in that direction? This again could be accomplished with multiple `if` statements. For example, we might have the following for when the direction is zero.

```
if(dir == 0) {
    setLocation(getX(), getY()-1);
}
```

We could have similar `if` statements for the other three directions. This code too can be simplified by using an array or two and a little bit of logic. To see how, let's think about what the array allows us to do. One way to view an array is something that stores multiple values in it for us to work on. Another way to view it is as a structure that lets us look up values for given integer indexes. This second view is how we want to look at it in this situation. The choice of index is fairly clear. The variable `dir` tells us the direction and works as a perfect index. The question is, what are we looking up? To answer this, perhaps it would be helpful to look at what the `if` statement would be like for a different direction. We have seen what it is for direction 0. This is what it would look like for direction 1.

```
if(dir == 1) {
    setLocation(getX()+1, getY());
}
```

What is it that varies between these two? Obviously, the direction we are

checking for in the `if` statement is different. That change will be reflected in using a different index. The other thing that is different is the way in which we change the x and y arguments to `setLocation`. In the first case we subtract one from y and in the second we add one to x. To see how we can generalize this, there are two things we should realize. First, subtraction is that same as adding a negative. So we can picture both cases as adding a value. Second, not changing something is the same as adding zero. So when the x or y value doesn't change, we can see that as adding zero to the value.

Collecting these thoughts, we have different values that we want to add to x and y and they differ based on the direction we are facing which we remember as an integer index. So to get the final logic we need two arrays. One array stores the values we will add to x for each direction and the other stores the value we will add to y. Our call to `setLocation` will add the proper value from each of these arrays into the x and y values. This is what the class look like when we put it all together.

```
public class Head extends Actor {
    private int dir;
    private String[] directions = {"up", "right", "down", "left"};
    private int[] xOffset = {0, 1, 0, -1};
    private int[] yOffset = {-1, 0, 1, 0};

    public void act() {
        for(int i = 0; i < directions.length; i++) {
            if(Greenfoot.isKeyDown(directions[i])) {
                dir = i;
            }
        }
        setLocation(getX()+xOffset[dir], getY()+yOffset[dir]);
    }
}
```

This code will give us the behavior we want when you run the scenario. You can adjust the speed to what you consider to be a reasonable level, but the head will move constantly on straight lines and the direction will be determined by the last key pressed. The thing to note is that it manages to do this with rather simple logic by effective use of arrays. The code we described that doesn't use arrays would have required a total of eight `if` statements. There would have been four that checked the key presses and another four that checked the value of `dir` and altered the location. This code is much more compact.

What is more, this code can be easily altered and extended. Consider what you have to do if you want to use a different set of key presses. For example, you might want to use W, A, S, and D instead of the arrow keys. This change could be made simply by altering the values in the `directions` array to be the `String`s "w", "d", "s", and "a". Of course, that change wouldn't have been too hard to implement with the `if` statement either. A more complex change would be to add diagonals. Going with the letter keys, you could imagine allowing the

snake to move on diagonals and having the keys Q, W, E, A, D, Z, X, and C control the eight different directions. With `if` statements that requires doubling the length of the code. With this version, it merely requires doubling the length of the arrays. We include the extra `String`s in the `directions` array and add new entries to the `xOffset` and `yOffset` arrays so they cause the head to move in the proper diagonal directions. The length of the code in lines would be unchanged.

5.3 General List Usage

We now have the head moving properly, but what about a tail and having the tail grow as we eat apples? There are a number of different ways that this could be implemented. One way is to let the head and each part of the tail know about the next one in line. To fit with the topic of this chapter, we are going to use a different method. We will, instead, have the head keep track of all of the pieces of the tail and handle them appropriately.

So how are we going to do this? Clearly we need something that can store multiple values. This should immediately bring a list or an array to mind. The question is, which one should we use? Would we be better off giving our head a `Tail[]` or a `List<Tail>` to use to remember all of the actors in the tail? We could make either one work, but if we think about the benefits and limitations of each, we will see that one is clearly superior to the other. The advantage of arrays is a briefer syntax. The is especially true when we have small arrays that we can initialize using the shortcut syntax with curly braces. The advantage of lists is that they can easily change in size as we add or remove elements. In the case of the tail for our snake, we don't have a fixed number of tail elements up front and what is more, we know that we need to add new ones every time the snake eats an apple. This clearly points to lists being the superior alternative.

Before we can do this with a `List` though, we need to learn a bit more about `List`s. The examples in the last chapter used `List`s that were constructed by Greenfoot. Any method in Greenfoot that might return multiple values to us uses a `List`. We never actually had to make our own `List`s, instead Greenfoot made a `List` for us and we simply used it. As a result, we didn't call many of the methods that are available on `List`s. In particular, we called `get` and `size`, but never used `add` or `remove`. We also never used `new` with a `List`. Let's look at how we can do that.

To make the head remember all the tail elements, we want a member of type `List<Tail`. We might be tempted to use the following line of code to declare this member and give it an initial value of an empty list.

```
private List<Tail> tail = new List<Tail>();
```

Unfortunately, this won't compile. You will get a message that the `List` type is abstract and you can't instantiate it. This concept of abstract is something we haven't discussed and won't use significantly in this book. The idea is that

you can create classes[3] that have methods that aren't implemented. It is up to the subtypes of the class, those classes that inherit from the abstract class, to provide implementations. In the case of `List` there are two main subclasses that we would consider using: `ArrayList` and `LinkedList`. The `LinkedList` is better when you are doing frequent adds and removes, but it is slower for calling `get`. If you use a `LinkedList` you should use the `for`-each loop to run through it. We will be adding, but a lot less frequently than we will be running through it and as we will stick with the standard `for` loop we will use the `ArrayList`. So the line declaring and initializing the `List` in our code will look like this:

```
private List<Tail> tail = new ArrayList<Tail>();
```

So our head now begins with an empty `List` that has the ability to contain `Tail` objects. Now that we have the `List` we need to figure out what to do with it. To begin with, when the head reaches an apple, a new instance of `Tail` should be created and added to our `List`. We could put the code for that into a method called `eatApple`. We could call this method immediately after moving the head and simply check if the head intersects with an apple.

```
public void eatApple() {
    Actor apple = getOneIntersectingObject(Apple.class);
    if(apple != null) {
        World w = getWorld();
        Tail newTail = new Tail();
        w.addObject(newTail, 0, 0);
        tail.add(newTail);
        apple.setLocation(Greenfoot.getRandomNumber(w.getWidth()),
            Greenfoot.getRandomNumber(w.getHeight()));
    }
}
```

This method basically checks if there is an overlapping apple and if it does it makes a new `Tail` actor, adds the `Tail` actor to both the world and the `tail` list, then moves the apple to a new location. You might worry a bit that the new `Tail` piece is put at (0,0). Indeed, if you put in this method and call it at the end of `act` you can play and move around and eat apples. As you do, multiple `Tail` actors will be added to the top left corner of the world. This isn't a problem for here though because it will be fixed with our next change.

After adding the `eatApple` method, our `List` will get one longer every time that the head crosses an apple. The new elements go at the end of the `List` because that is what the normal `add` method does on a `List`. We need the tail to follow the head around. The easiest way for us to do that is to loop through the `List` and move each element to the location the one before it had been at.

[3] Technically `List` is a slightly different construct called an `interface`, not an abstract class. Interfaces are very similar to classes except that they are completely abstract. Originally, none of their methods could be implemented and they weren't allowed to store any data. Java 8 added default implementations to methods in interfaces, but that is well beyond the scope of this book.

We could put this into a method called `advanceTail` that would look like this:

```
public void advanceTail(int x, int y) {
    int lastX = x;
    int lastY = y;
    for(int i = 0; i < tail.size(); i++) {
        Tail t = tail.get(i);
        x = t.getX();
        y = t.getY();
        t.setLocation(lastX, lastY);
        lastX = x;
        lastY = y;
    }
}
```

The arguments that are passed into this method give the location the head had been at before it moves so that becomes the new location of the first element of the `tail List`. Each subsequent element is then set to the location of the previous one in the loop.

If you run this you get very much the behavior that is desired. The only thing missing is that the game doesn't end if you run into your tail or the wall. It turns out that both of these can be accomplished simply by having a check at the end of `act` to determine if the head intersects with any of the `Tail` objects. If you run into the wall, the `advanceTail` method will cause the tail to move on top of the head causing the game to stop. After adding this, the final version of the `act` method looks like this:

```
public void act() {
    for(int i = 0; i < directions.length; i++) {
        if(Greenfoot.isKeyDown(directions[i])) {
            dir = i;
        }
    }
    int x = getX();
    int y = getY();
    setLocation(getX()+xOffset[dir], getY()+yOffset[dir]);
    eatApple();
    advanceTail(x, y);
    if(getOneIntersectingObject(Tail.class) != null) {
        Greenfoot.stop();
    }
}
```

We now have a fully functional game of Snake. The remarkable thing is that through proper usage of lists and arrays this can be accomplished in roughly 50 lines of code.

!!! 1.Processing data 2.Explain inheritance and polymorphism with lists.

Chapter 6

Drawing in Greenfoot

We want

!!! changing the snakes head so that it looks different depending on where it faces.

Chapter 7

Mouse Input in Greenfoot

Frequently we want our programs to be able to get input from the mouse as well as the keyboard. The mouse information includes clicking and position data. In this chapter we will explore how to get mouse input in Greenfoot and integrate that capability into some different scenarios.

7.1 Mouse Basics

The Greenfoot API includes a number of methods in the Greenfoot class and a whole separate class to support the ability to get input from the mouse. The Greenfoot class has five methods that begin with "mouse" and return a boolean to tell you if something has happened with the mouse. The

7.2 Connect Four

As an example of something we can use mouse input for we will write a game of Connect Four. This will not only test our ability to work with mouse input, it will force us to test our logic skills as well.

7.3 Spider Chase

The Connect Four game was a nice starting example, but the use of the mouse in it was fairly simple.

Chapter 8

File Processing

3.Exceptions (required for I/O)

Chapter 9

Simple Optimization

One of the most common types of problems solved on computers is the optimization problem. This problem involves determining the best solution to a particular problem from all possible solutions.

Chapter 10

Recursion

This chapter will look at some basic uses of a very powerful programming technique called recursion. The concept of recursion is fairly simple. A recursive method is a method that calls itself, or alternately, that is defined in terms of itself. This simple concept allows us to do fairly remarkable things. To help us understand how it works though, we will start with the mundane.

10.1 Start with the Math

The idea of recursion is found in mathematics with recursive functions. We'll start our exploration there by looking at an example of a recursive function and seeing how it works. The function we want to start with is this one:

$$f(n) = \begin{cases} 1 & n < 2 \\ n * f(n-1) & otherwise \end{cases}$$

. This function is defined for integers and has a value of 1 for any input value less than 1. For all other input values, its value is the input value times the value you get when you call it with a value that is one smaller. To see how this works, let's imagine calling this function with the value 5.

$$f(5) = 5 * f(4)$$

To get the real value of this we need to figure out the value of f(4). So let's take this out a bit further.

$$f(5) = 5*f(4) = 5*4*f(3) = 5*4*3*f(2) = 5*4*3*2*f(1) = 5*4*3*2*1 = 120$$

Another way to view this is the following.

$$\begin{gathered} f(5) = 5 * f(4) \\ f(4) = 4 * f(3) \\ f(3) = 3 * f(2) \\ f(2) = 2 * f(1) \\ f(1) = 1 \end{gathered}$$

That base value of 1 then gets passed back up through the operations. You can imagine we are expanding things moving down on the left then working them out to get the final solution moving back up on the right.

$$\begin{array}{cccccc} f(5) = 5 * f(4) & & & & & 5 * 24 = 120 \\ f(4) = 4 * f(3) & \downarrow & & & \uparrow & 4 * 6 = 24 \\ f(3) = 3 * f(2) & \downarrow & & & \uparrow & 3 * 2 = 6 \\ f(2) = 2 * f(1) & \downarrow & & & \uparrow & 2 * 1 = 2 \\ f(1) = 1 & \rightarrow & \rightarrow & \rightarrow & & 1 \end{array}$$

An obvious question to ask is, what function is this? Hopefully you recognize this as the factorial function. In math you would normally write factorial with an exclamation point. So n!=n*(n-1)*(n-2)*...*3*2*1. We saw previously that the exclamation point is used for something completely different in Java. In Java it is used for "not" in Boolean logic. So now that we understand this function in math and see a little bit about how recursion works with a math function, it is interesting to ask what this would look like as a method in Java. We can call our method factorial because that is what it will calculate. We start with the method signature where we state what the method returns and what it takes as input. The factorial is normally defined on integers so we want to provide an int n as the input. It needs to give us back an integer value as well so it should return an int as well. Our method might start off looking like this:

```
public~int~factorial(int~n)~\{~

//~...~

\}
```

10.2 Iteration with Recursion

The simplest application of recursion

10.3 Understanding Recursion

There are certain features that

Towers of Hanoi

Chapter 11

Path Finding

There have been many times in this book when we have had an actor walk toward a certain destination. The methods that we have used to do this have been following "direct routes". That is, the actor moves one square in each step and picks a direction such that the total distance between it and the destination decreases with each step. You can view this as a greedy algorithm for path finding because it always takes a locally optimal step and never second guesses that step. This approach is fine for many examples, but it breaks down as soon as we insert obstacles into the world.

With obstacles, such as walls in a maze, the greedy approach no longer works. There are now times when you have to walk away from your destination for a while so that you can get closer in the end. In this chapter we want to give one of of actors the ability to find an optimal route from its current location to a destination that respects obstacles. We are going to do this using recursion.

Index