

# Introduction to Programming

with Greenfoot Java [Ver. 2.0]

## Unit 1: First Program

CHAPTER 1: GETTING TO KNOW GREENFOOT

DR. ERIC CHOU

IEEE SENIOR MEMBER



Textbook



1. Brainstorm, brainstorm, brainstorm
2. Research for further inspiration
3. Refine and clarify your central idea
4. Focus on character goals and conflict
5. Weave smaller ideas into your central concept
6. Give yourself time
7. ... But don't forget to start writing!



# Overview

---

LECTURE 1



# Course Objectives

---

- There are several goals in doing this: one is to learn **programming**, another is to have **fun** along the way.
- While the examples we discuss in this book are specific to the Greenfoot environment, the concepts are general: working through this book will teach you general programming principles in a modern, object-oriented programming language.
- However, it will also show you how to make your own computer game, a biology simulation, or an on-screen piano.



# Overview

---

How to program graphical computer programs,

- such as simulations and games,
  - using the Java Programming Language and
  - the Greenfoot environment.
- 
- Topics: Language/IDE/Project

## **Unit 1: First Program**

Chapter 1: Getting to know Greenfoot  
Chapter 2: The first Program: Little Crab  
Chapter 3: Improving the Crab  
Chapter 4: Finish the Crab Game  
Interlude 1 Sharing Your Scenarios

## **Unit 2: First Program**

Chapter 5: Scoring  
Chapter 6: Making Music  
Chapter 7: Object Interaction  
Chapter 8: Interacting Objects  
Chapter 9: Collision Detection  
Interlude 2: The Greeps Competition

## **Unit 2: Additional Scenarios**

Chapter 10: Creating Images and Sound  
Chapter 11: Simulations  
Chapter 12: Greenfoot and the Kinect  
Chapter 13: Additional Scenario Ideas

## **Appendix**

Appendix A: Installing Greenfoot  
Appendix B: Greefoot API  
Appendix C: Collision Detection  
Appendix D: Some Java Details



# Project-Oriented

---

- **This book is very practically oriented.** Chapters and exercises are structured around real, hands-on development tasks.
- First, there is a problem that we need to solve, then we look at language constructs and strategies that help us solve the problem.
- This is quite different from many introductory programming textbooks that are often structured around programming language constructs.





# Project-Oriented

---

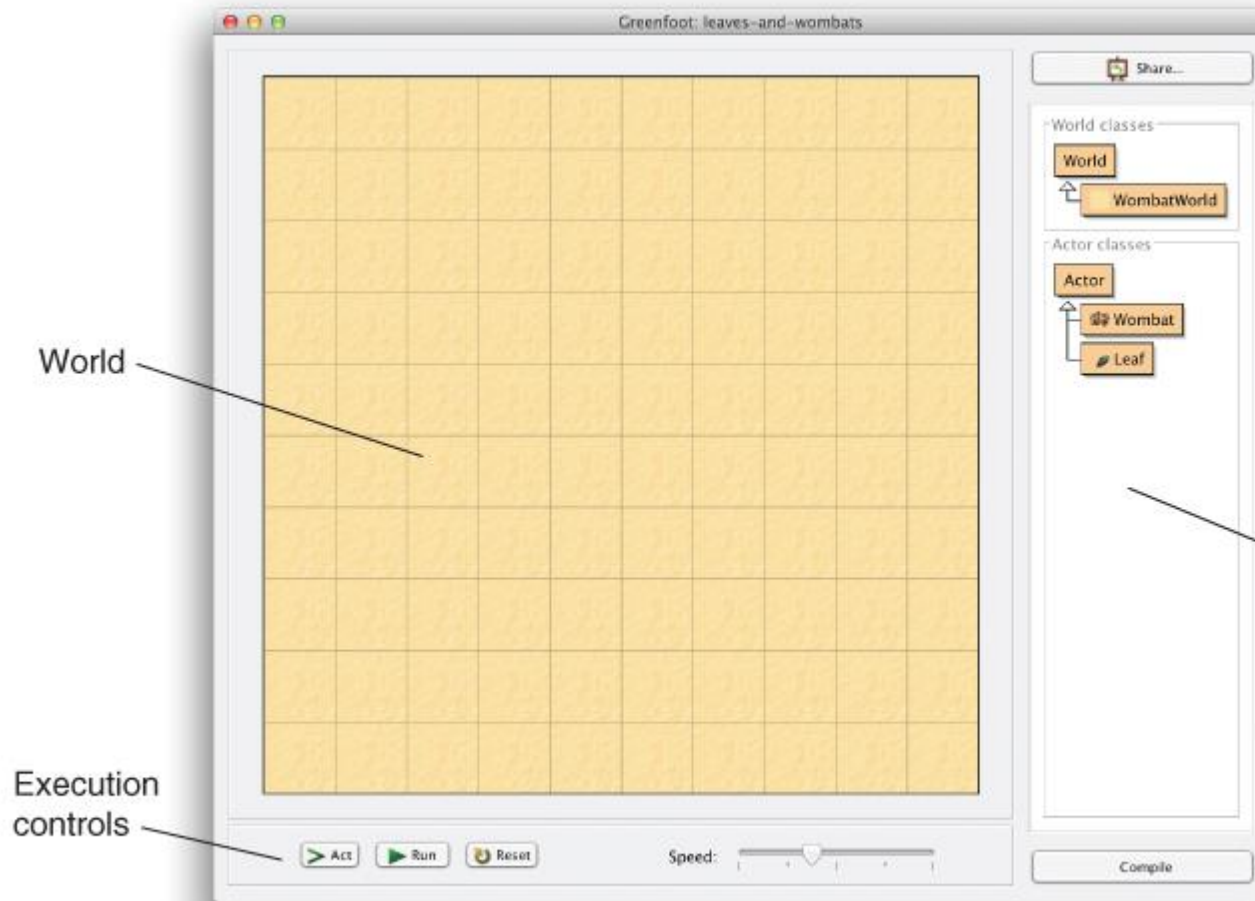
- The projects discussed in this book are easy enough that they can be managed by high school students, but they are also open and extendable enough that even seasoned programmers can find interesting and challenging aspects to do.
- While Greenfoot is an educational environment, Java is not a toy language. Since Java is our language of choice for this book, the projects discussed here (and others you may want to create in Greenfoot) can be made as complex and challenging as you like.



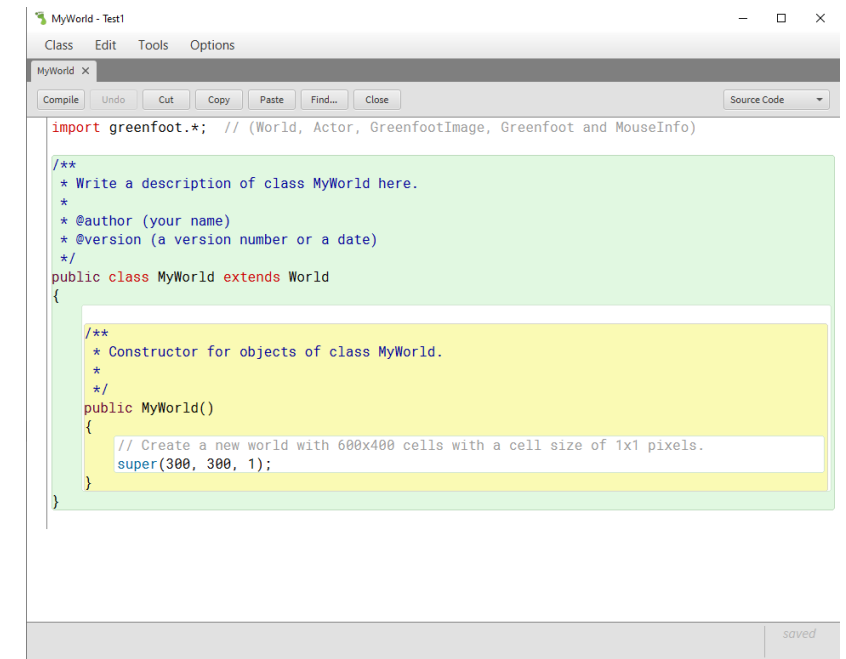
# Greenfoot IDE

---

LECTURE 2



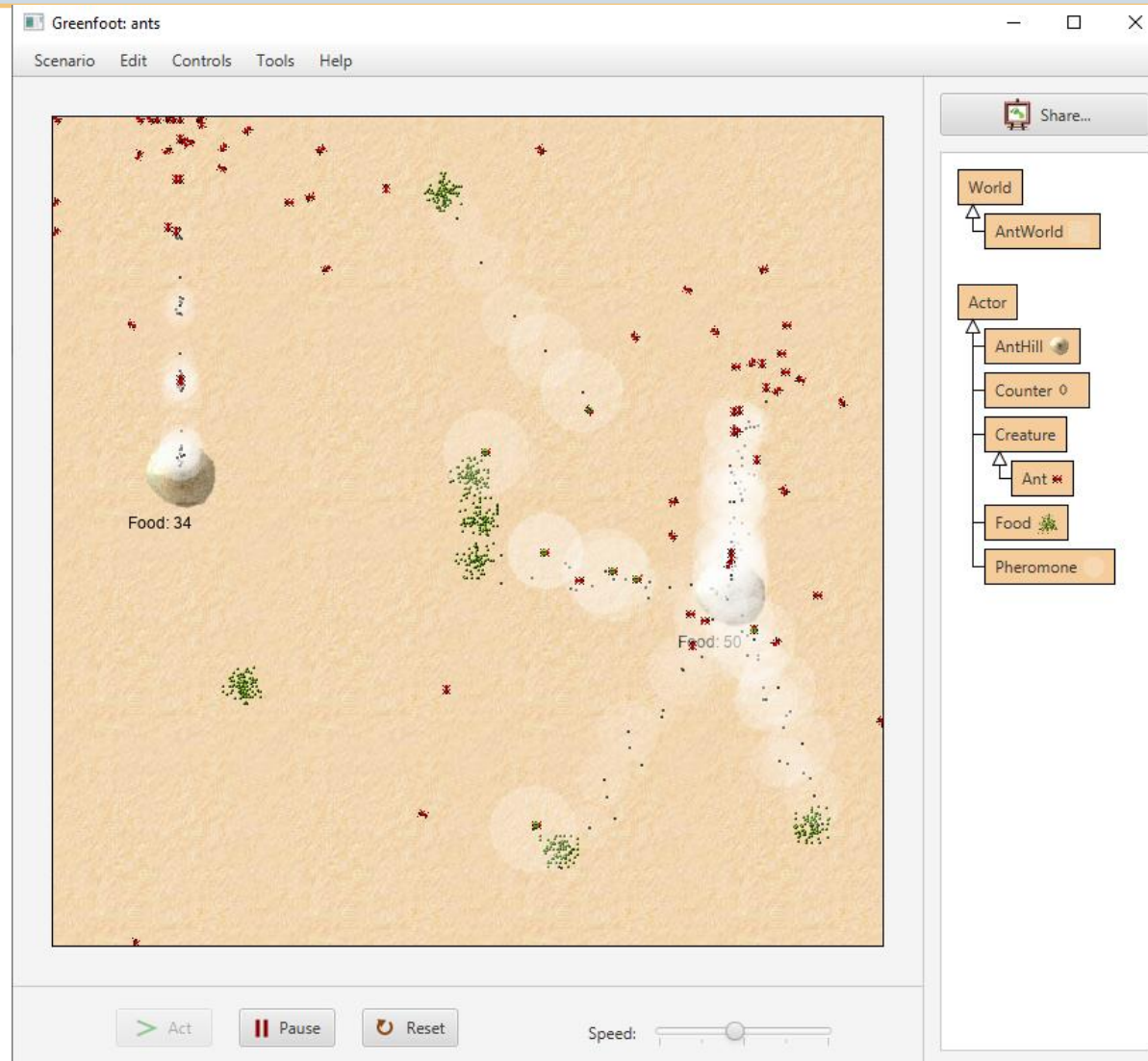
Project View



Editor View

Name	Date modified	Type	Size
images	1/20/21 19:30	File folder	
sounds	1/20/21 19:30	File folder	
MyWorld.class	1/20/21 19:34	CLASS File	1 KB
MyWorld.cbtxt	1/20/21 19:34	CTXT File	1 KB
MyWorld	1/20/21 19:34	JAVA File	1 KB
project	1/21/21 0:42	Greenfoot Project ...	1 KB
README	1/20/21 19:30	TXT File	1 KB

File Folder View



C:\Program Files\Greenfoot\scenarios\java\ants



# The Main Areas

---

## **The world:**

- The largest area covering most of the screen (a sand-colored grid in this case) is called the world. This is where the program will run and we will see things happen.

## **The class diagram:**

- The area on the right with the beige-colored boxes and arrows is the class diagram. We shall discuss this in more detail shortly.

## **The execution controls:**




- The Act, Run, and Reset buttons and the speed slider at the bottom are the execution controls. We'll come back to them in a little while, too.



# Copy Project and Scenarios Files

---

- You need to click the project file to start Greenfoot (Same as BlueJ)
- You may copy the .java file and and project.**greenfoot** file. The whole project file can be

 MyWorld	1/20/21 19:34	JAVA File	1 KB
 project	1/21/21 0:42	Greenfoot Project ...	1 KB
 README	1/20/21 19:30	TXT File	1 KB



# Objects and Classes

---

LECTURE 3



# Objects and Classes

---

- We shall discuss the class diagram first. The class diagram shows us the classes involved in this scenario. In this case, they are **World**, **WombatWorld**, **Actor**, **Wombat**, and **Leaf**.
- We shall be using the Java programming language for our projects. Java is an object-oriented language. The concepts of classes and objects are fundamental in object orientation.





A wombat, by the way, is an Australian marsupial (Figure 1.2). If you want to find out more about them, do a Web search—it should give you plenty of results.



# Experience the Project

C:\Eric Chou\Java Courses\Greenfoot\Senarios\book-scenarios\chapter01\leaves-and-wombats

---

## Open the Scenarios:

New Wombat, New Leaf

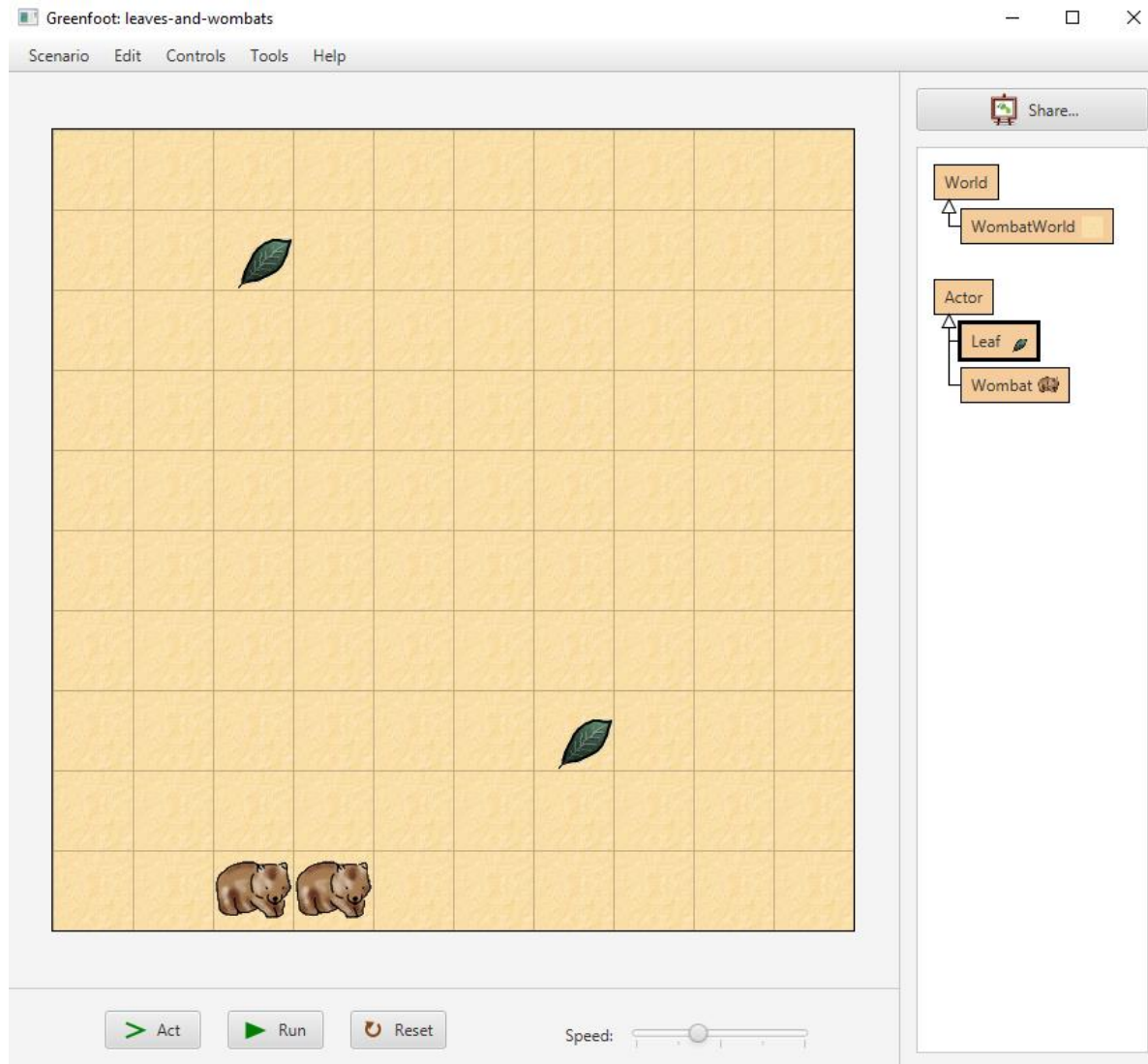
## Operations:

Act: step

Run: run in continuous mode.

Pause: pause the execution

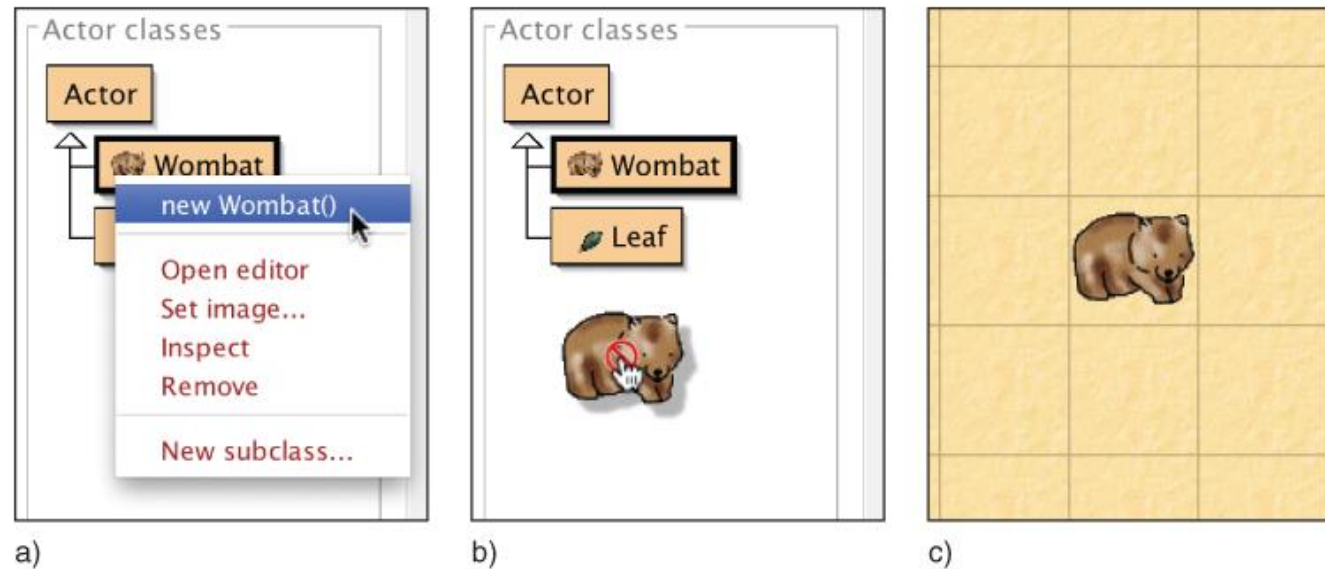
Reset: reset the simulation environment



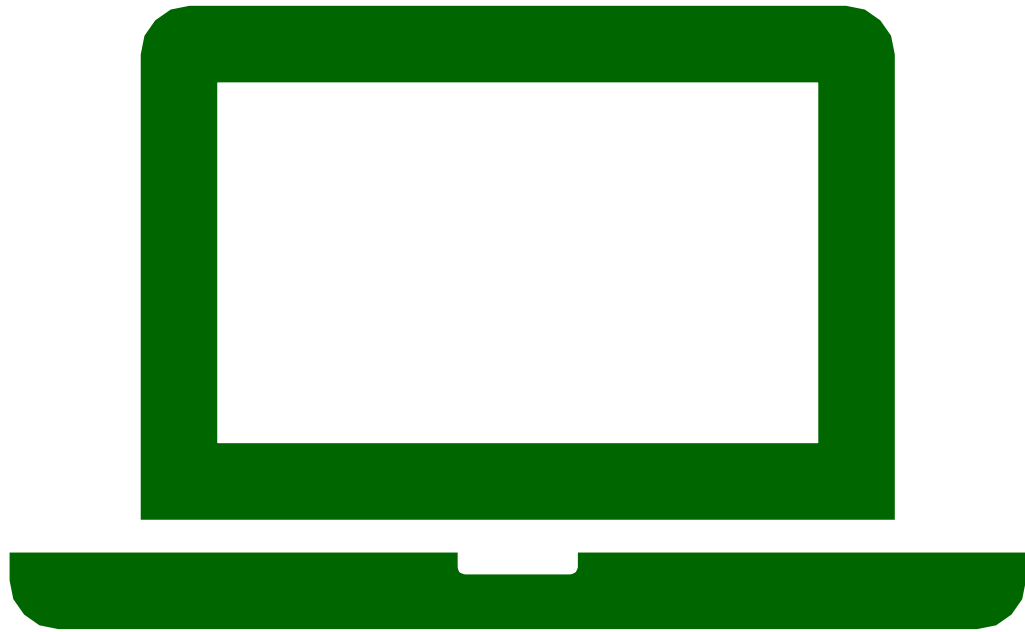


# Image source: Marco Tomasini/Fotolia

- Right-click on the Wombat class, and you will see the class menu pop up (Figure 1.3a). The first option in that menu, new Wombat(), lets us create new wombat objects. Try it out.



**Figure 1.3** a) The class menu b) Dragging a new object c) Placing the object



# Exercise 1.1

---

OBJECTS



# Exercise 1.1

---

- Create some more wombats in the world. Create some leaves.
- Currently, only the **Wombat** and **Leaf** classes are of interest to us. We shall discuss the other classes later.



# Interacting with Objects

---

LECTURE 4

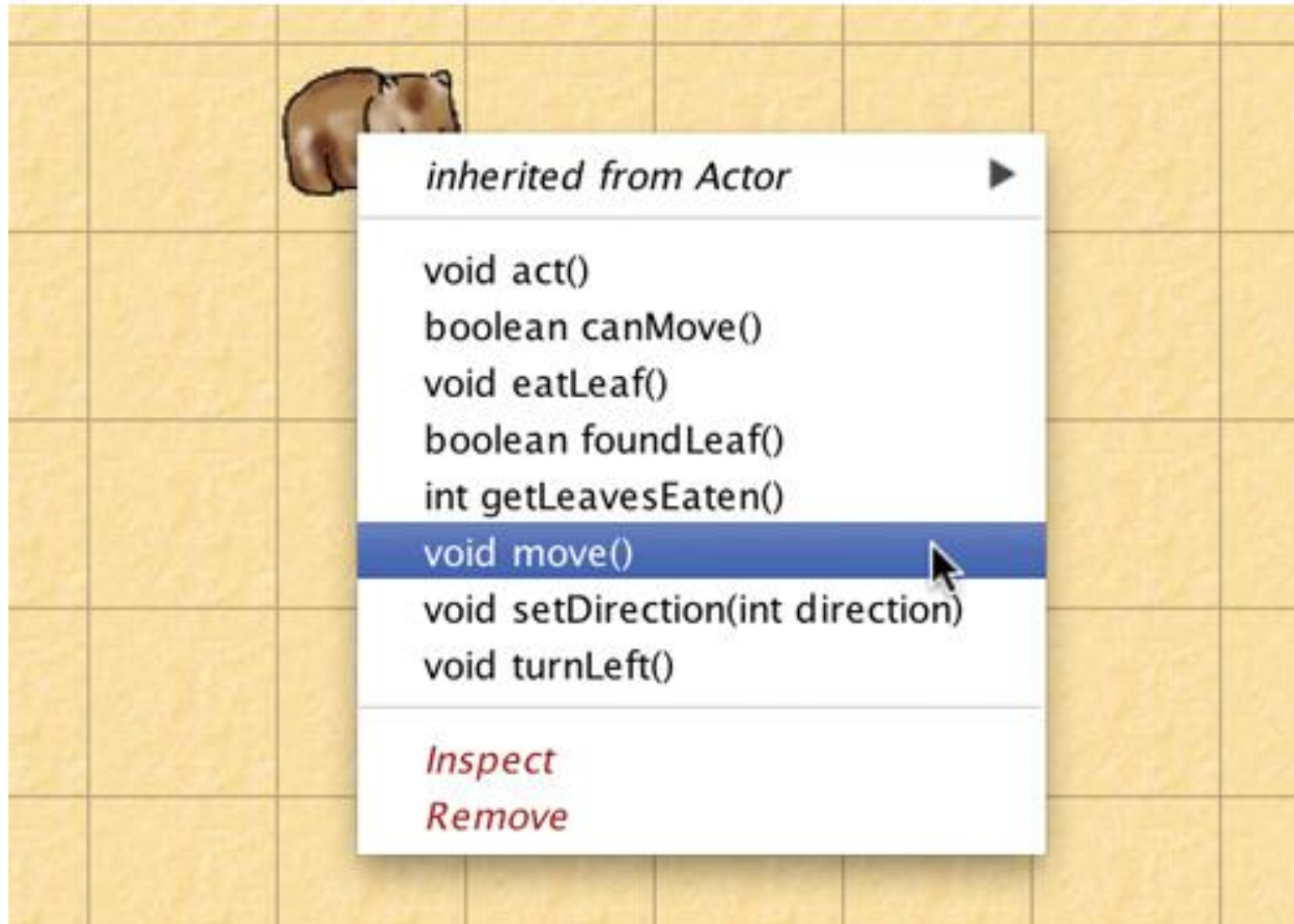


# Interacting with Objects

---

- Once we have placed some objects into the world, we can interact with these objects by right-clicking them. This will pop up the object menu (Figure 1.4). The object menu shows us all the operations this specific object can perform.
- For example, a wombat's object menu shows us what this wombat can do (plus two additional functions, Inspect and Remove, which we shall discuss later).





**Figure 1.4** The wombat's object menu



# Methods

---

## Concept

Objects have methods.

- Invoking these performs an action.
- In Java, these operations are called methods. It cannot hurt to get used to standard terminology straight away, so we shall also call them methods from now on. We can invoke a method by selecting it from the menu.



# Exercise 1.2

---

METHODS



# Exercise 1.2

---

- Invoke the **move()** method on a wombat. What does it do? Try it several times. Invoke the **turnLeft()** method. Place two wombats into your world and make them face each other. In short: we can start to make things happen by creating objects from one of the classes provided, and we can give commands to the objects by invoking their methods. Let us have a closer look at the object menu. The move and turnLeft methods are listed as:

```
void move()
```

```
void turnLeft()
```

- We can see that the method names are not the only thing shown. There is also the word void at the beginning and a pair of parentheses at the end. These two cryptic bits of information tell us what data goes into the method call, and what data comes back from it.



# Return Types

---

LECTURE 5



# Return Types

---

- The word at the beginning is called the return type. It tells us what the method returns to us when we invoke it. The word void means “nothing” in this context: methods with a void return type do not return any information. They just carry out their action, and then stop.

## **Concept**

The return type of a method specifies what a method call will return.



# Return Types

---

- Any word other than void tells us that the method returns some information when invoked, and of what type that information is. In the wombat's menu (Figure 1.4), we can also see the words int and boolean. The word int is short for “integer” and refers to whole numbers (numbers without a decimal point).
- Examples of integer numbers are 3, 42, −3, and 12000000.

## Concept

A method with a void return type does not return a value.



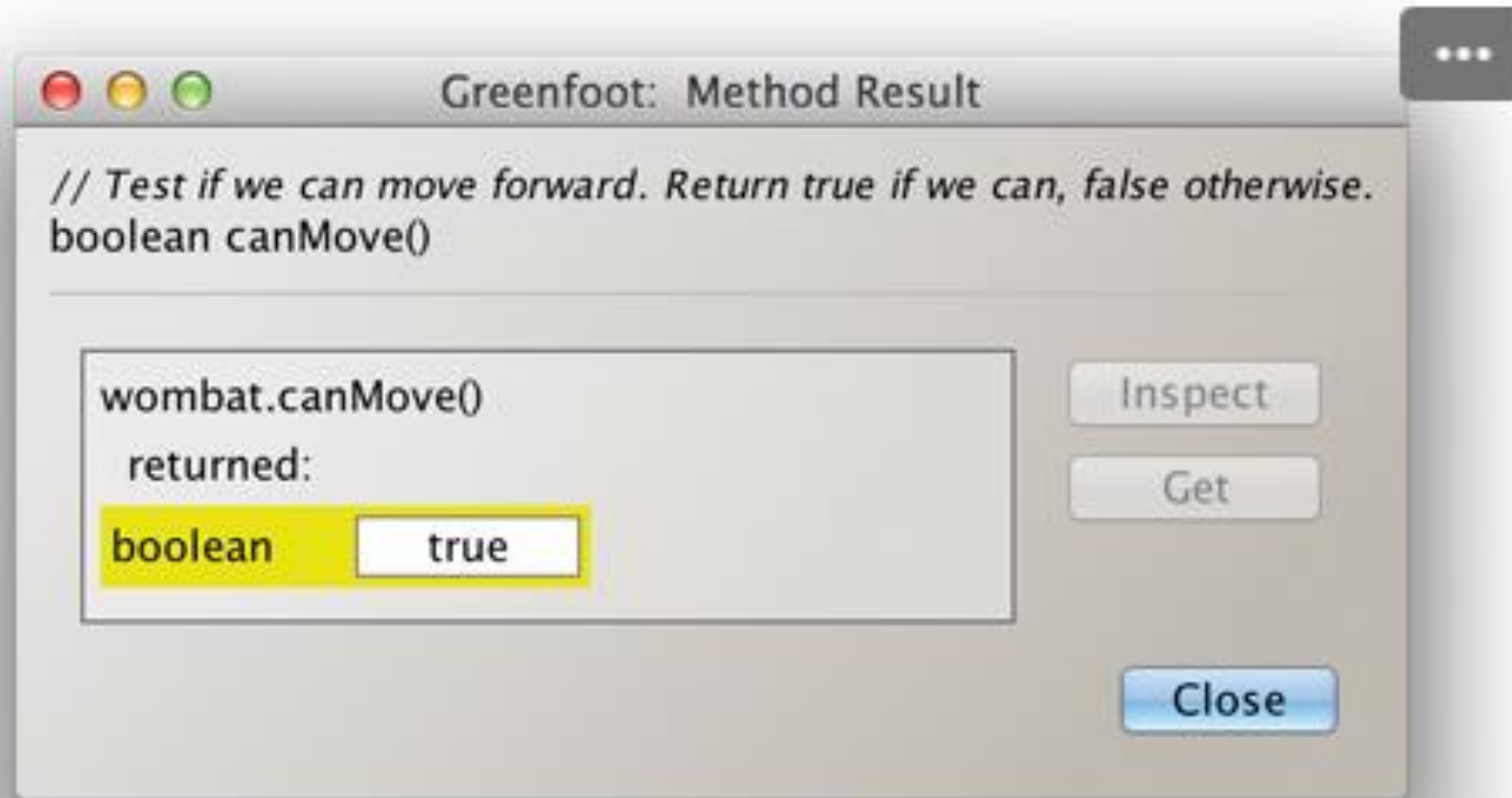
# Return Types

---

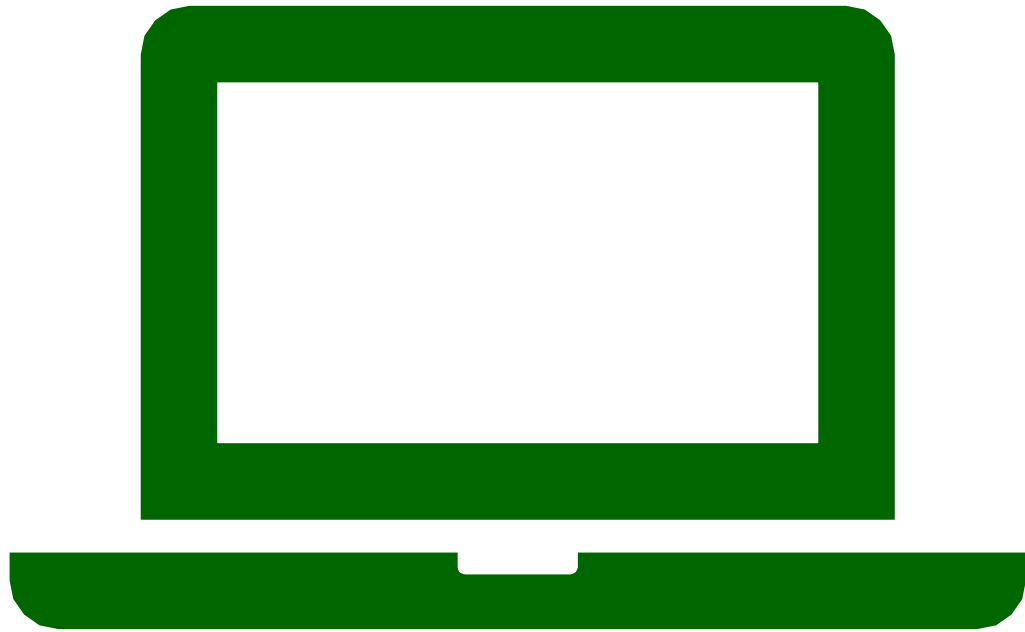
- The type **boolean** has only two possible values: **true** and **false**. A method that returns a **boolean** will return either the value **true** or the **value false** to us.
- Methods with **void** return types are like commands for our wombat. If we invoke the **turnLeft** method, the wombat obeys and turns left. Methods with non-void return types are like questions. Consider the **canMove** method:

```
boolean canMove()
```
- When we invoke this method, we see a result similar to that shown in Figure 1.5, displayed in a dialog box.
- The important information here is the word “true,” which





**Figure 1.5** A method result was returned by this method call. In effect, we have just asked the wombat “Can you move?”, and the wombat has answered by saying “Yes!” (true).



# Exercise 1.3

---

CANMOVE()



# Exercise 1.3

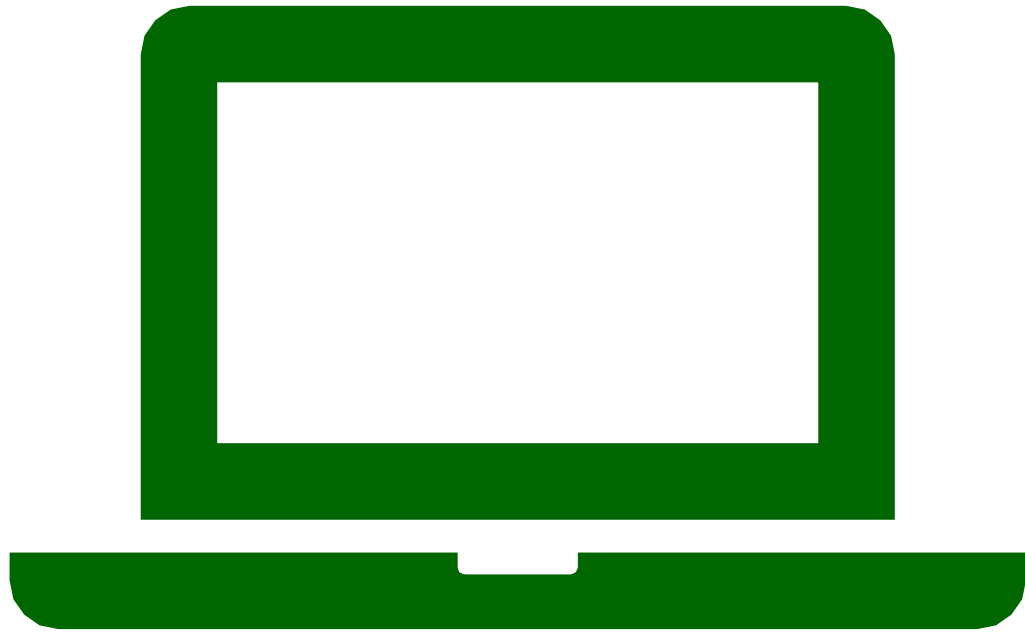
---

- Invoke the **canMove()** method on your wombat. Does it always return true? Or can you find situations in which it returns false?
- Try out another method with a return value:

```
int getLeavesEaten()
```
- Using this method, we can get the information how many leaves this wombat has eaten.

## Concept

Methods with void return types represent commands;  
methods with non-void return types represent questions.



# Exercise 1.4

---

METHODS



## Exercise 1.4

---

- Using a newly created wombat, the **getLeavesEaten()** method will always return zero. Can you create a situation in which the result of this method is not zero? (In other words: can you make your wombat eat some leaves?)
- Methods with non-void return types usually just tell us something about the object (Can it move? How many leaves has it eaten?), but do not change the object. The wombat is just as it was before we asked it about the leaves.
- Methods with void return types are usually commands to the objects that make it do something.



# Parameters

---

LECTURE 6



# Parameters

---

- The other bit in the method menu that we have not yet discussed is the parentheses after the method name.

*Return type*                      *Parameter*

↓                                      ↓

```
int getLeavesEaten()  
void setDirection(int direction)
```

## Concept

A parameter is a mechanism to pass additional data to a method.



# Parameters

---

- The parentheses after the method name hold the **parameter list**. This tells us whether the method requires any additional information to run, and if so, what kind of information.
- If we see only a pair of parentheses without anything else between it (as we have in all methods so far), then the method has an **empty parameter list**. In other words, it expects no parameters—when we invoke the method, it will just run. If there is anything between the parentheses, then the method expects one or more parameters—additional information that we need to provide.
- Let us try out the **setDirection** method. We can see that it has the words int direction written in its parameter list. When we invoke it, we see a dialog box similar to the one shown in Figure 1.6.





**Figure 1.6** A method call dialog



# Parameters

---

## Concept

Parameters and return values have **types**.  
Examples of types are **int** for numbers, **and**  
**boolean** for true/false values.

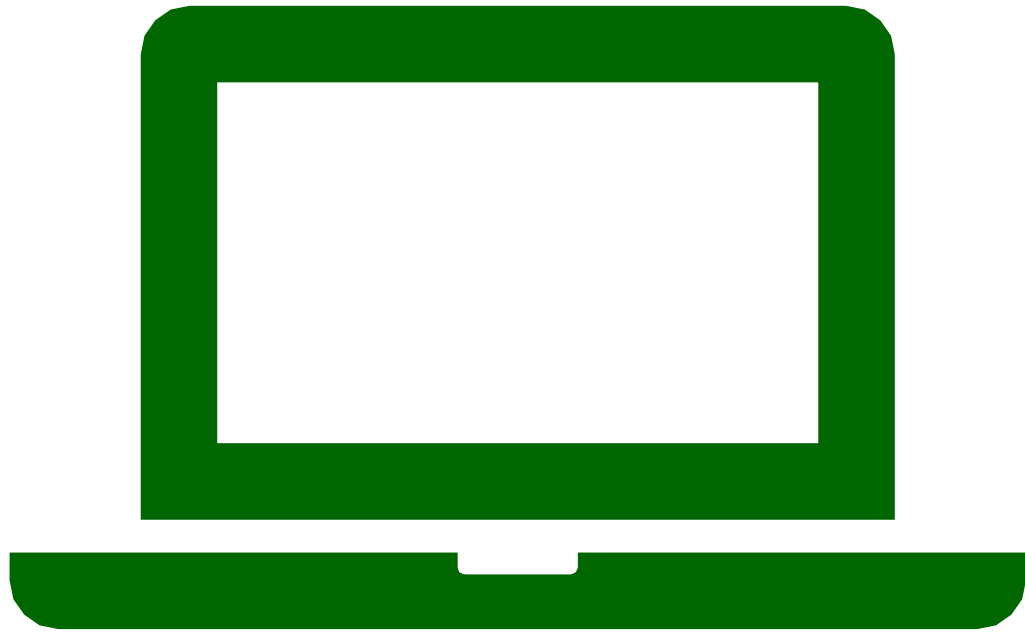
- The words **int direction** tell us that this method expects one parameter of type **int**, which specifies a direction. A parameter is an additional bit of data we must provide for this method to run. Every parameter is defined by two words: first the parameter type (here: **int**) and then a name, which gives us a hint what this parameter is used for. If a method has a parameter, then we must provide this additional information when we invoke the method.



# Parameters

---

- In this case, the type **int** tells us that we now should provide a whole number, and the name suggests that this number somehow specifies the direction to turn to.
- At the top of the dialog is a comment that tells us a little more: the direction parameter should be between 0 and 3.



# Exercise 1.5

---

SIGNATURE



## Exercise 1.5

---

- Invoke the **setDirection(int direction)** method. Provide a parameter value and see what happens. Which number corresponds to which direction? Write them down. What happens when you type in a number greater than 3? What happens if you provide input that is not a whole number, such as a decimal number (2.5) or a word (three)?
- The **setDirection** method expects only a single parameter. Later, we shall see cases where methods expect more than one parameter. In that case, the method will list all the parameters it expects between the parentheses.



# Exercise 1.5

---

## Concept

The specification of a method, which shows its return type, name, and parameters is called its signature.

- The description of each method shown in the object menu, including the return type, method name, and parameter list, is called the **method signature**.
- We have now reached a point where you can do the main interactions with Greenfoot objects. You can create objects from classes, interpret the method signatures, and invoke methods (with and without parameters).



# Greenfoot Execution

---

LECTURE 7



# Greenfoot Execution

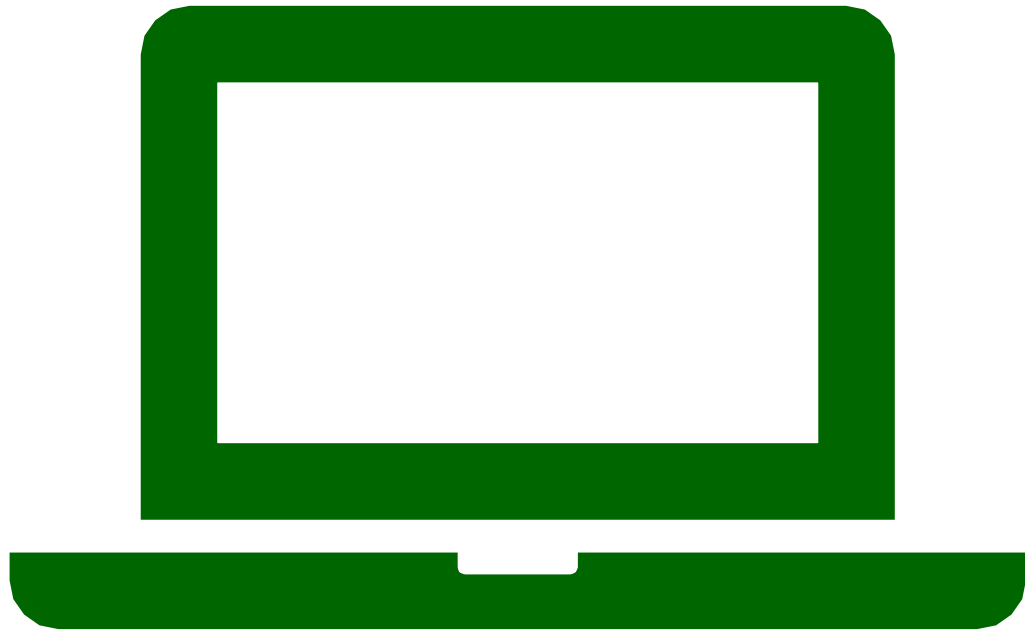
---

- There is one other way of interacting with Greenfoot objects: The execution controls.

## Tip

You can place objects into the world more quickly by selecting a class in the class diagram, and then shift-clicking in the world.





# Exercise 1.6

---

ACT()



## Exercise 1.6

---

- Place a wombat and a good number of leaves into the world, and then invoke a wombat's **act()** method several times. What does this method do?
- How does it differ from the move method? Make sure to try different situations, for example, the wombat facing the edge of the world, or sitting on a leaf.



# Exercise 1.7

---

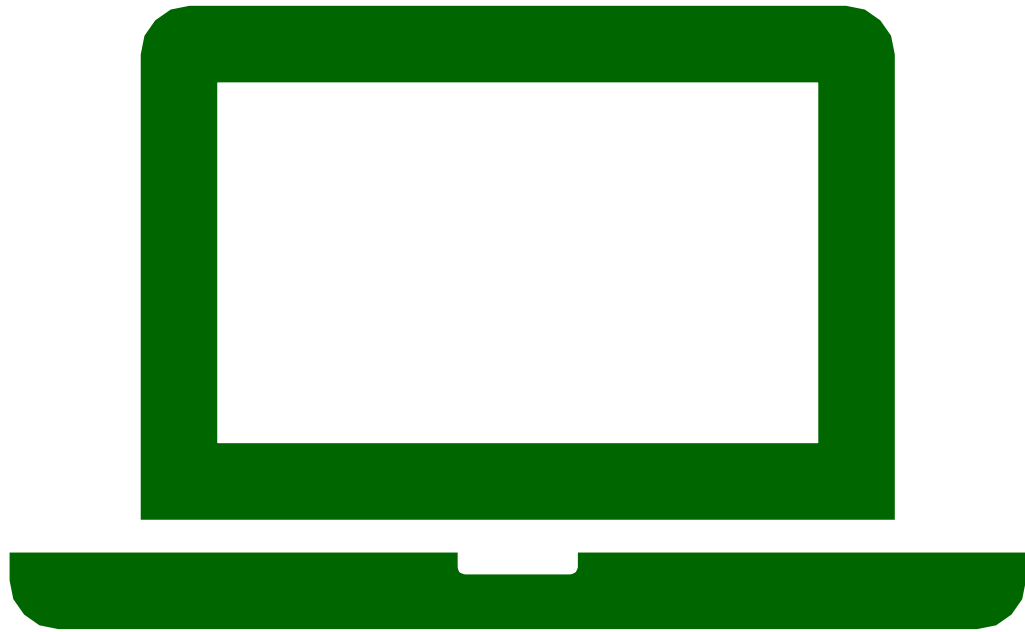
STEP OVER



## Exercise 1.7

---

- Still with a wombat and some leaves in the world, click the Act button in the execution controls near the bottom of the Greenfoot window. What does this do?



# Exercise 1.8

---

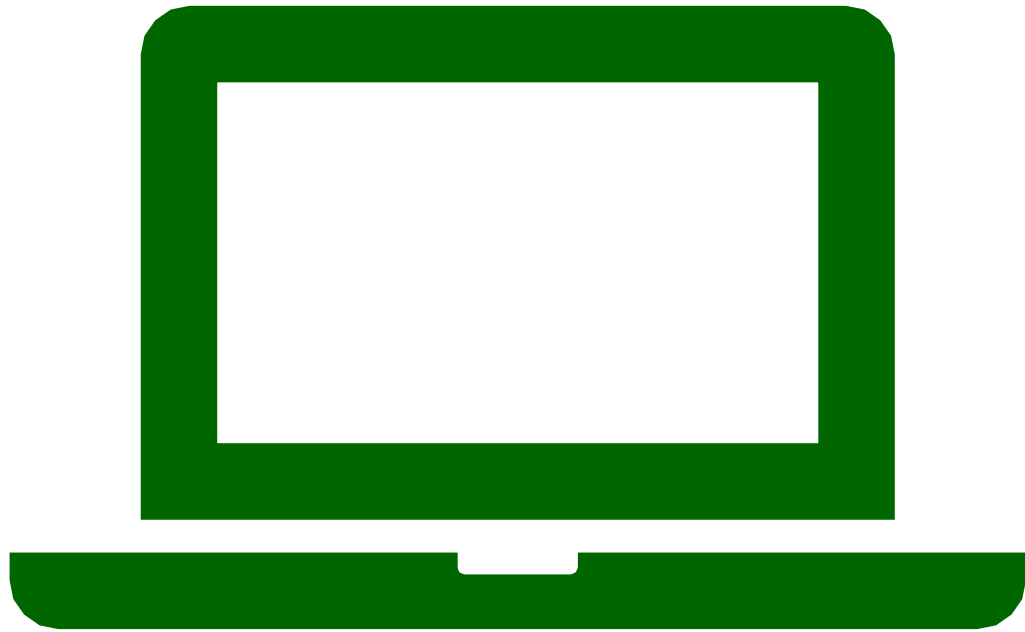
METHODS



## Exercise 1.8

---

- What is the difference between clicking the Act button and invoking the `act()` method? (Try with several wombats in the world.)



# Exercise 1.9

---

METHODS



# Exercise 1.9

---

- Click the Run button. What does it do?
- The **act** method is a very fundamental method of Greenfoot objects. We shall encounter it regularly in all the following chapters. All objects in a Greenfoot world have this **act** method. Invoking act is essentially giving the object the instruction “Do whatever you want to do now.”
- If you tried it out for our wombat, you will have seen that the wombat’s **act** does something like the following:
  - If we’re sitting on a leaf, eat the leaf.
  - Otherwise, if we can move forward, move forward.
  - Otherwise, turn left.





# Exercise 1.9

- The experiments in the exercises above should also have shown you that the Act button in the execution controls simply calls the act method of the actors in the world. The only difference to invoking the method via the object menu is that the Act button invokes the act method of all objects in the world, while using the object menu affects only the one chosen object.

## Concept

Objects that can be placed into the world are known as actors.

- The Run button just calls act over and over again for all objects, until you click Pause.
- Let us try out what we have discussed in the context of another scenario.



# A Second Example

---

LECTURE 8

# A Second Example

- Open another scenario, named **asteroids1**, from the chapter01 folder of the book scenarios. It should look similar to Figure 1.7 (except that you will not see the rocket or the asteroids on your screen yet).

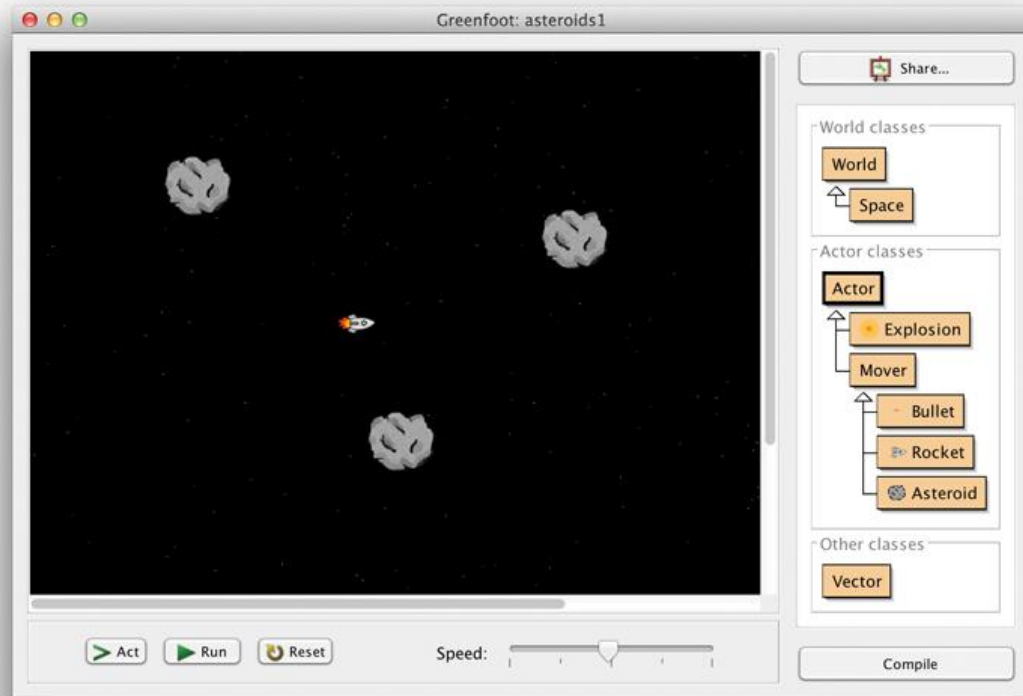


Figure 1.7 The asteroids1 scenario



# Understanding the Class Diagram

---

LECTURE 9



# Understanding the Class Diagram

- Let us first have a closer look at the class diagram (Figure 1.8). At the top, you see the two classes called **World** and **Space**, connected by an arrow.

## Concept

A subclass is a class that represents a specialization of another. In Greenfoot, this is shown with an arrow in the class diagram.

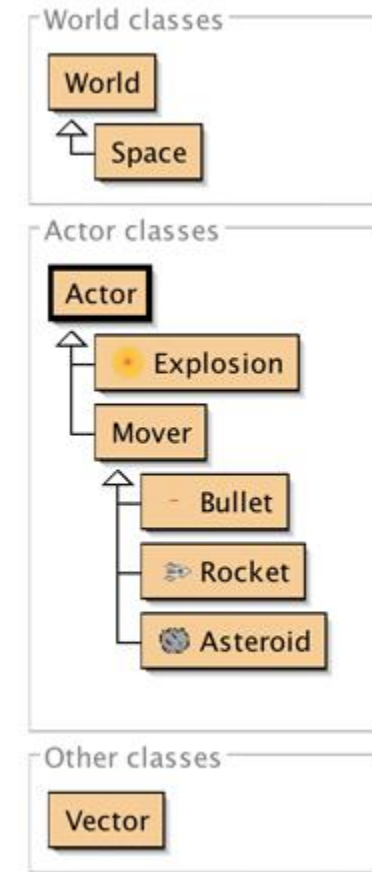


Figure 1.8 A class diagram



# Understanding the Class Diagram

---

- The **World** class is always there in all Greenfoot scenarios—it is built into Greenfoot. The class under it, **Space** in this case, represents the specific world for this particular scenario. Its name can be different in each scenario, but every scenario will have a specific world here.
- The arrow shows an is-a relationship: **Space** is a **World** (in the sense of Greenfoot worlds: **Space**, here, is a specific Greenfoot world). We also sometimes say that **Space** is a subclass of **World**.
- We do not usually need to create objects of world classes—Greenfoot does that for us. When we open a scenario, Greenfoot automatically creates an object of the world subclass. The object is then shown on the main part of the screen. (The big black image of space is an object of the **Space** class.)



# Understanding the Class Diagram

---

- Below this, we see another group of six classes, linked by arrows. Each class represents its own objects. Reading from the bottom, we see that we have asteroids, rockets, and bullets, which are all “movers,” while movers and explosions are actors.
- Again, we have subclass relationships: **Rocket**, for example, is a subclass of **Mover**, and **Mover** and **Explosion** are subclasses of **Actor**. (Conversely, we say that **Mover** is a superclass of **Rocket** and **Actor** is a superclass of **Explosion**.) Subclass relationships can go over several levels: **Rocket**, for example, is also a subclass of **Actor** (because it is a subclass of **Mover**, which is a subclass of **Actor**). We shall discuss more about the meaning of subclasses and superclasses later.



# Understanding the Class Diagram

---

- The class **Vector**, shown at the bottom of the diagram under the heading Other classes is a helper class used by the other classes. We cannot place objects of it into the world.





# Playing with Asteroids

---

LECTURE 10



# Playing with Asteroids

---

- We can start playing with this scenario by creating some actor objects (objects of subclasses of **Actor**) and placing them into the world. Here, we create objects only of the classes that have no further subclasses: **Rocket**, **Bullet**, **Asteroid**, and **Explosion**.
- Let us start by placing a rocket and two asteroids into space. (Remember: you can create objects by right-clicking on the class, or selecting the class and shift-clicking into the world.)
- When you have placed your objects, click the **Run** button. You can then control the spaceship with the arrow keys on your keyboard, and you can fire a shot by using the space bar. Try getting rid of the asteroids before you crash into them.



# Exercise 1.10

---

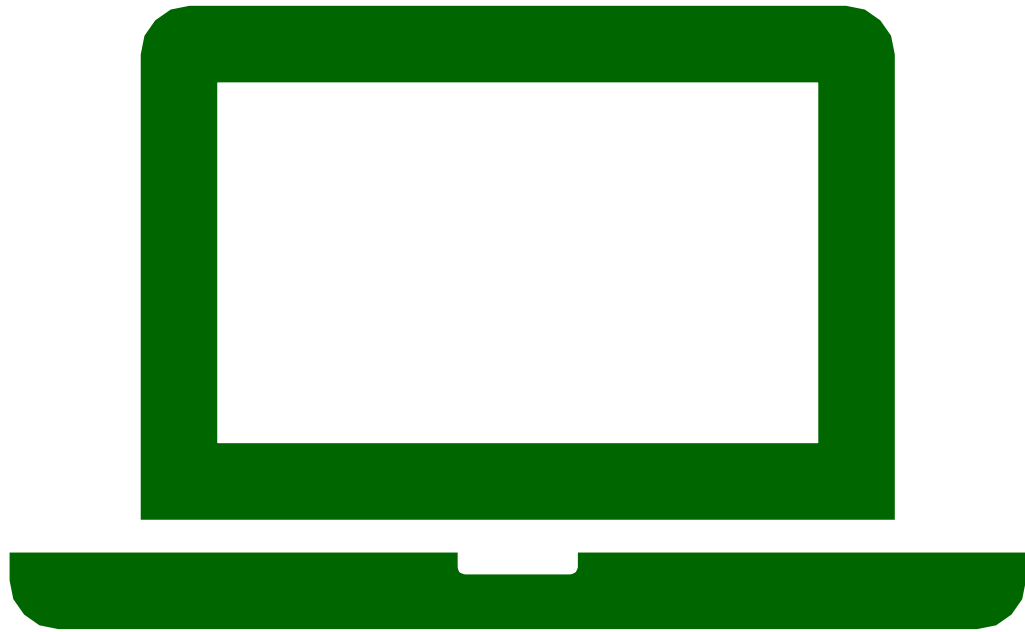
FIRING



## Exercise 1.10

---

If you have played this game for a bit, you will have noticed that you cannot fire very quickly. Let us tweak our spaceship firing software a bit so that we can shoot more quickly. (That should make getting rid of the asteroids a bit easier!) Place a rocket into the world, then invoke its **setGunReloadTime** method (through the object menu), and set the reload time to 5. Play again (with at least two asteroids) to try it out.



# Exercise 1.11

---

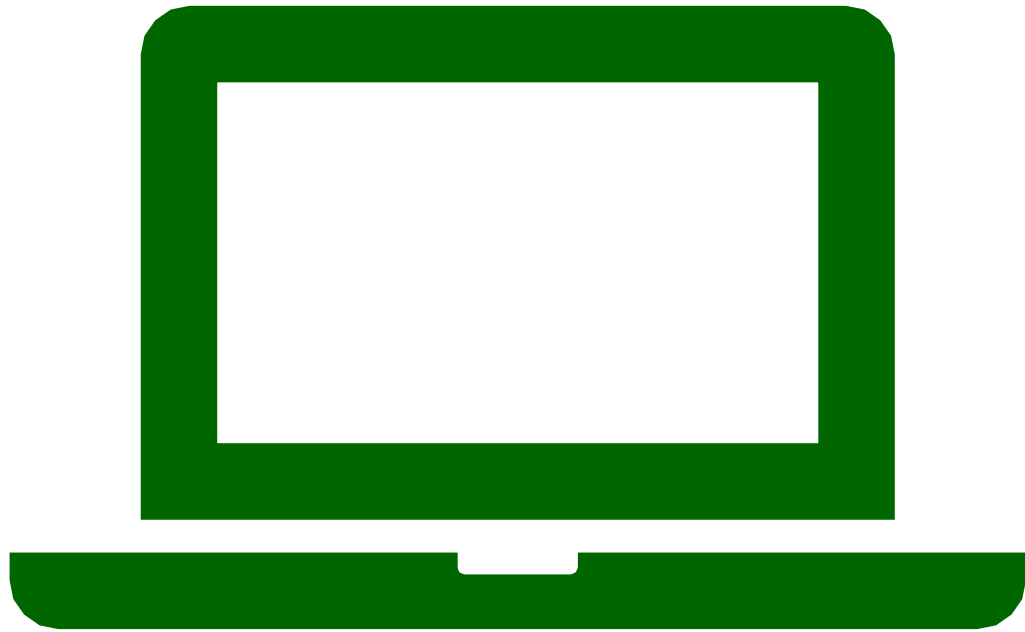
REMOVAL OF OBJECTS



## Exercise 1.11

---

- Once you have managed to remove all asteroids (or at any other point in the game) stop the execution (press Pause) and find out how many shots you have fired. You can do this using a method from the rocket's object menu. (Try destroying two asteroids with as few shots as possible.)



# Exercise 1.12

---

SPEED

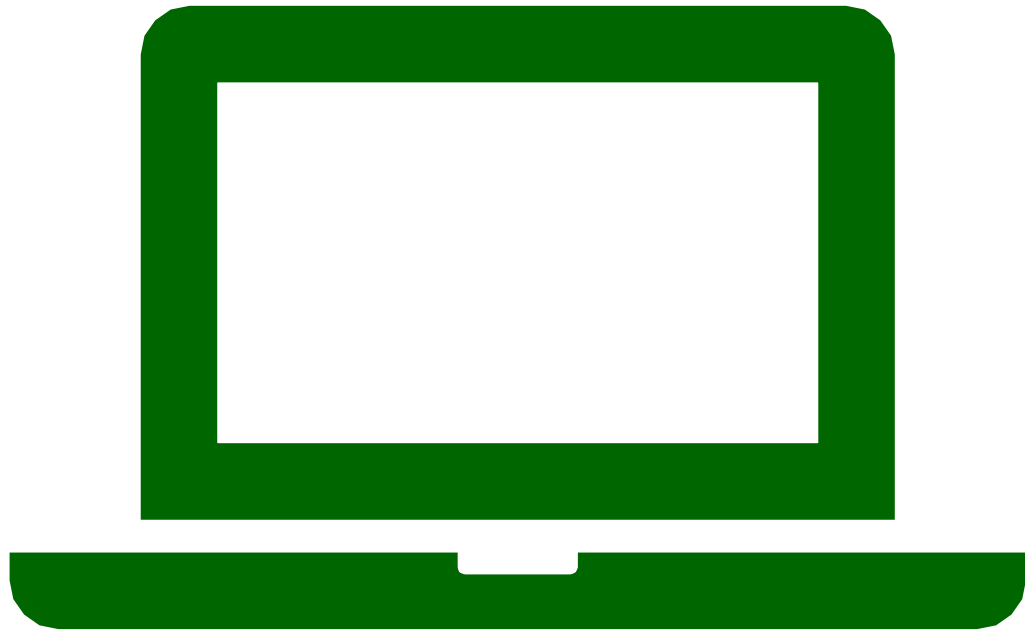


## Exercise 1.12

---

- You will have noticed that the rocket moves a bit as soon as you place it into the world. What is its initial speed?





# Exercise 1.13

---

STABILITY



## Exercise 1.13

---

- Asteroids have an inherent **stability**. Each time they get hit by a bullet, their stability decreases. When it reaches zero, they break up. What is their initial stability value after you create them? By how much does the stability decrease from a single hit by a bullet? (Hint: Just shoot an asteroid once, and then check the stability again. Another hint: To shoot the asteroid, you must run the game. To use the object menu, you must pause the game first.)



# Exercise 1.14

---

SIZE



# Exercise 1.14

---

- Make a very big asteroid.



# Source Code

---

LECTURE 11



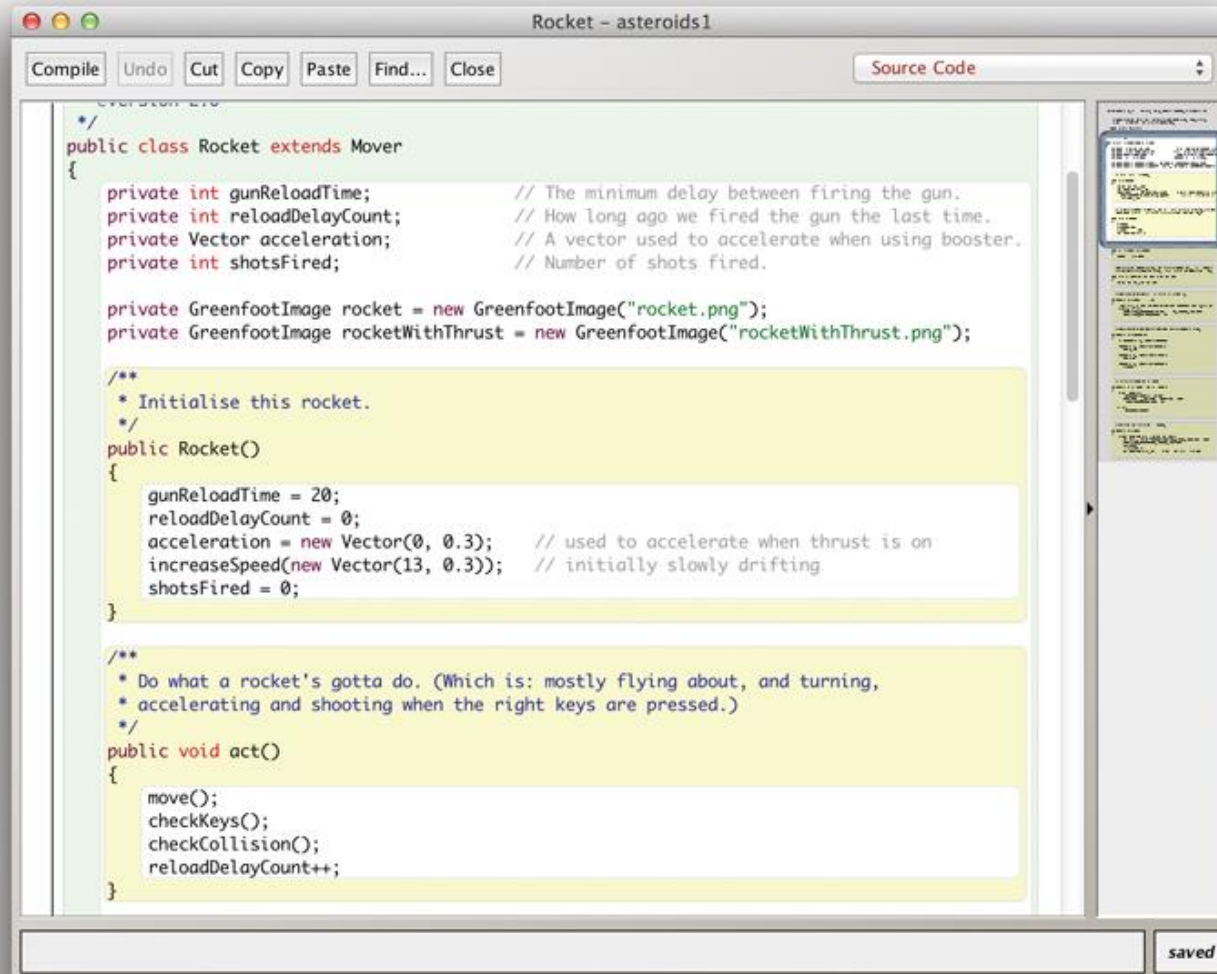
# Source Code

---

## Concept

Every class is defined by source code. This code defines what objects of this class can do. We can look at the source code by opening the class's editor.

- The behavior of each object is defined by its class. The way we can specify this behavior is by writing source code in the Java programming language. The source code of a class is the code that specifies all the details about the class and its objects. Selecting Open editor from the class's menu will show us an editor window (Figure 1.9) that contains the class's source code.



**Figure 1.9** The editor window of class Rocket



# Editor

---

- We have seen before that the default firing speed of the rocket was fairly slow. We could change this for every rocket individually by invoking a method on each new rocket, but we would have to do this over and over again, every time we start playing. Instead, we can change the code of the rocket so that its initial firing speed is changed (say, to 5), so that all rockets in the future start with this improved behavior.
- Open the editor for the Rocket class. About 25 lines from the top, you should find a line that
  - `readsgunReloadTime = 20;`
- This is where the initial gun reloading time gets set. Change this line so that it
  - `readsgunReloadTime = 5;`





# Editor

---

The source code for this class is fairly complex, and we do not need to understand it all at this stage. However, if you study the rest of this book and program your own games or simulations, you will learn over time how to write this code.

## Tip

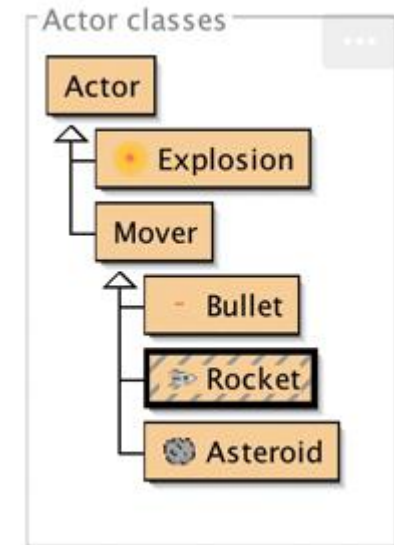
You can open an editor for a class by double-clicking the class in the class diagram.

At this point, it is only important to understand that we can change the behavior of the objects by changing the class's source code. Let us try this out.



# Class Status: uncompiled

- Be sure to change nothing else. You will notice very soon that programming systems are very fussy. A single incorrect or missing character can lead to errors. If, for example, you remove the semicolon at the end of the line, you would run into an error fairly soon.
- Close the editor window (our change is complete) and look at the class diagram again. It has changed: The Rocket class now appears striped (Figure 1.10). The striped look indicates that a class has been edited and now must be compiled. Compilation is a translation process: the class's source code is translated into a machine code that your computer can execute.



**Figure 1.10** Classes after editing



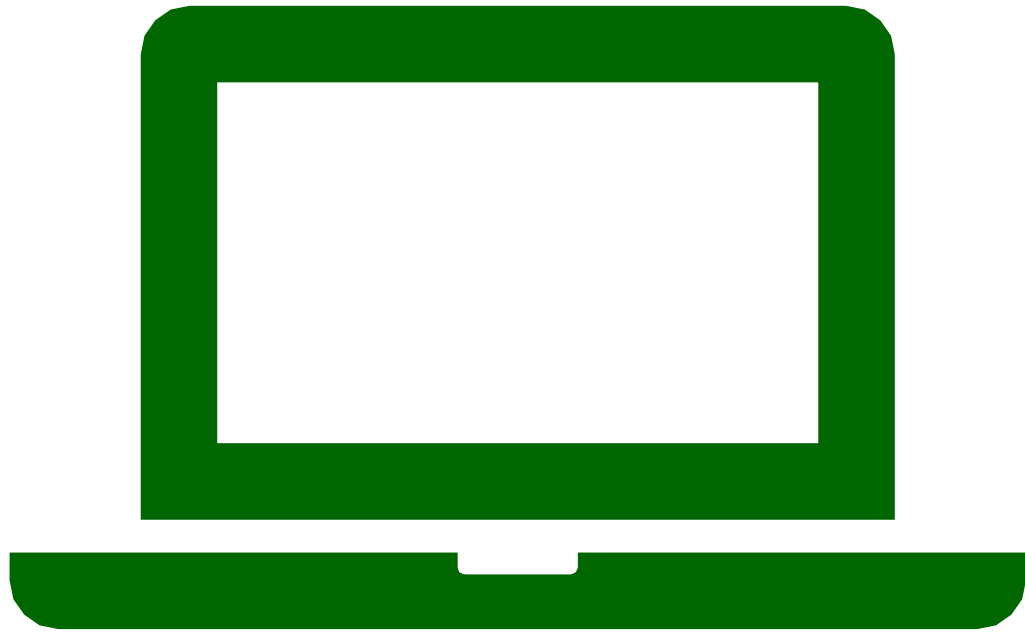
# Compilation

---

## Concept

Computers do not understand source code. It needs to be translated to machine code before it can be executed. This is called compilation.

- Classes must always be compiled after their source code has been changed, before new objects of the class can be created. (Sometimes several classes need recompilation even though we have changed only a single class. This may be the case because classes depend on each other. When one changes, several may need to be translated again.) We can compile the classes by clicking the Compile button in the bottom right corner of Greenfoot's main window. Once the classes have been compiled, the stripes disappear, and we can create objects again.



# Exercise 1.15

---

ASTEROIDS GAME



## Exercise 1.15

---

- Make the change to the Rocket class source code as described above. Close the editor, and compile the classes. Try it out: Rockets should now be able to fire quickly right from the start.
- We shall come back to the asteroids game in Chapter 7, where we will discuss how to write this game.



# Summary

---

LECTURE 12



# Summary

---

- In this chapter, we have seen what Greenfoot scenarios can look like, and how to interact with them. We have seen how to create objects, and how to communicate with these objects by invoking their methods. Some methods were commands to the object, while other methods returned information about the object. Parameters are used to provide additional information to methods, while return values pass information back to the caller.
- Objects were created from their classes, and source code controls the definition of the class (and with this, the behavior and characteristics of all the class's objects).
- We have seen that we can change the source code using an editor to make changes. After editing the source, classes need to be recompiled.
- We will spend most of the rest of the book discussing how to write Java source code to create scenarios that do interesting things.



# Summary

---

- Greenfoot scenarios consist of a set of **classes**.
- Many **objects** can be created from a **class**.
- Objects have **methods**. Invoking these performs an action.
- The **return type** of a method specifies what a method call will return.
- A method with a **void** return type does not return a value.
- Methods with void return types represent **commands**; methods with non-void return types represent **questions**.
- A **parameter** is a mechanism to pass additional data to a method.





# Summary

---

- Parameters and return values have **types**. Examples of types are int for numbers, and boolean for true/false values.
- The specification of a method, which shows its return type, name, and parameters, is called its **signature**.
- Objects that can be placed into the world are known as **actors**.
- A **subclass** is a class that represents a specialization of another. In Greenfoot, this is shown with an arrow in the class diagram.
- Every class is defined by **source code**. This code defines what objects of this class can do. We can look at the source code by opening the class's editor.
- Computers do not understand source code. It needs to be translated to machine code before it can be executed. This is called **compilation**.