**1.1.3 Source Code in Greenfoot**

**Introduction**

In the last assignment we learned to use the Greenfoot interface and context menus. We explored the basics of OO programming. In this assignment we will delve into the **source code** for the classes and begin to actually program in Java. This assignment focuses mainly on basic Java syntax and structure.

We will cover:

* source code
* syntax
* access modifier
* compile-time error (syntax error)
* parentheses, curly braces, square braces
* class signature
* method signature
* control structure
* branching
* if-statement
* conditional
* boolean
* dot notation
* Keywords: **import**, **public**, **private**, **protected,**  **class, extends, if, true, false**

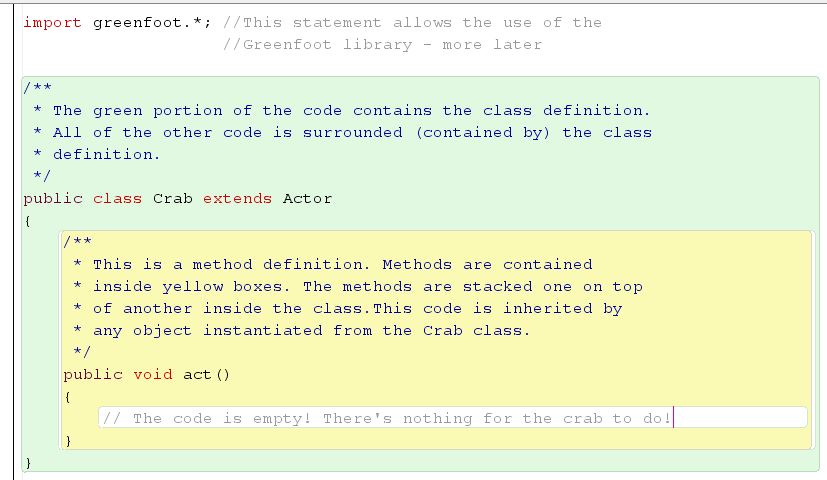
**Materials**

1. Computer with Greenfoot
2. Source files: crab-world.zip

**Activity**

**Part I: Source Code & Basic Syntax**

1. Download, unzip, and open the little-crab scenario in Greenfoot. Compile any classes that you need to. (Very important reminder- don’t open files from a zip folder - they will seem to open but won’t work correctly.)
2. Place a Crab into the world and run it. What happens?
3. Double-click the Crab class in the class diagram and take a look. What you'll see is the Java **source code** for the Crab - it’s mostly empty! When an object is instantiated from the class it inherits this code. In other words, the crab you created has this empty code inside.



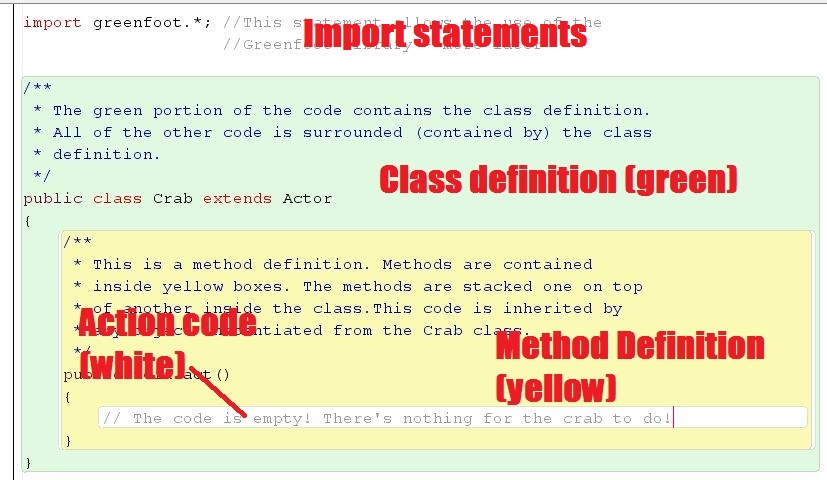
Source code contains Java, which is a language with its own rules about using periods, capitalization, punctuation, and grammar. In other words, it has its own **syntax** and it is time for us to start learning it. The syntax in Java is very unforgiving - if you make a small mistake, such as capitalizing where you shouldn’t - your code will not work. These errors are called **syntax (or compile-time) errors**. Some syntax rules in Java:

|  |
| --- |
| 1. Always start with a **class definition** 2. **Method definitions** go inside class definitions 3. All definitions and **control structures** (like if statements and loops) begin with an open **curly brace** { and end with a close curly brace } 4. Action code goes inside definitions and each line must end with a semicolon (;) |

A quick note about brackets: there are three main bracket types used in Java:

* 1. ( ) --- These are **parentheses** or **round braces**. In Java, they are used in mathematical expressions, and to surround the parameter lists for method calls.
  2. [ ] --- These are **square braces** or brackets. In Java, they are used for arrays.
  3. { } --- These are **curly braces**, also known as squiggly brackets. In Java, these are used to surround blocks of code, such as methods or the contents of classes.

1. Let’s break down the source code. Each part of the code is color-coded.



* The first part of the source code is the **import** statements. These statements bring methods from other files into your source code for you to use. Every Greenfoot class file has this line at the beginning:

**import** greenfoot.\*;

This allows the program to use all of the Greenfoot **built-in methods**.

* The next part is the **class signature**
  + It begins with an access modifier keyword: either **public**, **private**, or **protected**.
    - Public means that any other class in the world can access and use this class or method.
    - Private means that this class can access the class or method.
    - Protected means that only subclasses can access or use this class or method.

We will be using the **public** keyword to define new classes.

* + Second is the name of the class. We use the keyword **class** followed by the name we want to give our class. In Java, class names are always capitalized
  + The third part is for optional keywords. In this case we are using the keyword **extends**, which creates a subclass that has all of the superclass’ methods (except the private ones). The class listed after **extends** is the superclass from which we want to take methods.

|  |  |  |  |
| --- | --- | --- | --- |
| **Class Signature** | | | |
| **Access modifier** | **Class name** | **Optional keyword** | **Keyword modifier** |
| **public** | **class** Crab | **extends** | Actor |

* The next part of the source code is the **method signature**. We covered methods last assignment so you should be familiar with these. They are in yellow
* Inside each method is action code to describe what the method does.

|  |  |  |
| --- | --- | --- |
| **Method Signature** | | |
| **return type** | **method name** | **parameter(s)** |
| **void** | setDirection | (**int** direction) |

1. Anyway, back in the code, you can see that there's nothing in the method definition. If we want our crab to do anything, we're going to have to fill that in.

|  |
| --- |
| Fill in the following code (be precise - one mistake will cause a **compile-time error** and you will not be able to compile your code!)  C:\Users\anthony pittman\Desktop\edit-Crab-act-narrow-ef97407c552c3472ca8d079545f372bc.png  Once you've written this, hit the *Compile* button. Go to the main Greefoot window, place a crab, and click *Run*. What happens? |

1. Let's make the crab do a bit more than moving in a straight line.

|  |
| --- |
| Go back to the code, and after the move line, add another line (but still inside the curly brackets for the act method) that says turn(3), like this:  C:\Users\anthony pittman\Desktop\edit-Crab-act-narrow-7b87b8a20a2f4609f3afb8e2f610c33d.png  You'll see that the crab runs in a circle. Experiment with the turning amount to get the circle tighter or larger. What does the argument for turn() represent?  Place multiple crabs in the world and run it - what happens?  Make the crabs turn left by editing the code. |

1. This program doesn’t have much interactivity. In order to add some, we need to understand the concept of branching. When you have a choice in a program we say the program can branch. We accomplish this with **if** structures. These structures go inside of method definitions. Here’s an if structure:

|  |
| --- |
| **if** (conditional)  {  action code;  } |

The **conditional**, if true, will execute the code inside the braces. If it is false, the code in the braces is skipped. The conditional must be a boolean. A **boolean** is a **true** or **false** variable, expression, or a method that returns **true** or **false**.

|  |
| --- |
| Let’s create user controls using an if statement. Copy the following code into the act() method:  C:\Users\anthony pittman\Desktop\edit-Crab-act-narrow-3c807d157a9140178a0ee7d2d76fc638.png  Compile and run.  Notice that the Greenfoot.isKeyDown(“left”) method is where the boolean should be. This is because the method returns **true** or **false**. This method is a built-in method that we can use thanks to our import at the beginning of the program.  Change the controls to use ‘a’ and ‘d’.  What happens if you put in multiple crabs?  Hold down *a* and *d* at the same time. What happens? Why?  Make it so that the crab does not move unless ‘w’ is pressed and moves backward when ‘s’ is pressed. How does the placement of the code cause this behavior? |

**Part II: Dot Notation Syntax**

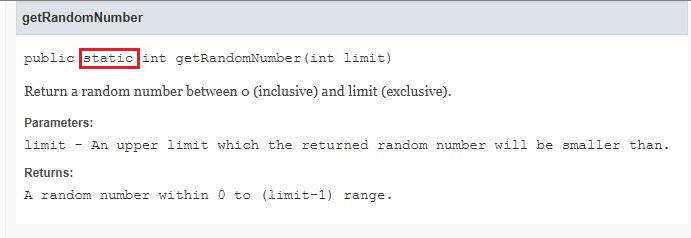
1. What we need is an enemy that eats crabs: a lobster! Let's add a lobster that moves in a straight line and eats crabs. We can do that using code that we've already seen how to write.

|  |
| --- |
| * 1. Right-click on the Actor class and select "new subclass….”   2. As the new class name, enter: Lobster -- note that we are using a capital L. Remember that in Java we begin class names with a capital letter.   3. From the left-hand list of images, select "lobster.png" as the image and press okay.   4. Once you've made your Lobster class, create a new method called moveAround() that has no return value. Fill it in by making the lobster move in a straight line.   5. Use the new moveAround() method inside of the Lobster class’ act() method.   6. Compile and run. You now have a lobster that moves in a straight line - but he gets stuck when he hits a wall.   7. Go back to the Crab class code. Create a moveAndTurn() method for the crab and transfer all the code from act() to moveAndTurn(). Place moveAndTurn() inside of act(). This is called refactoring and should not change the behavior of the program.   8. Tip - *ctrl + shift + i* will fix your indentation in Greenfoot. This can help you find errors or make your code look cleaner. |

1. Let's make our lobster more difficult to avoid by introducing some randomness. We can achieve this by using the getRandomNumber() method in the Greenfoot class. We want to use the method within the Lobster class, but the it only knows methods that it inherited from the Actor class. We need to pull this method in from an entirely different class and we’ll do it with **dot notation**.

|  |  |  |
| --- | --- | --- |
| **Dot notation** | | |
| **Method belongs to a class** | | |
| class-name | .method-name | (parameters) |
| **Method belongs to an object** | | |
| object | .method-name | (parameters) |

If a method belongs to a class instead of an object, it will be marked with the keyword **static** within the source code of the class. Here’s a snippet of the Greenfoot documentation (more about where to get this next assignment):



To use the getRandomNumber method and retrieve a number from 1-100 we would type Greenfoot.getRandomNumber(90);

The code below means we will turn a random amount each frame, between 0 degrees (inclusive) and 90 degrees (exclusive).

|  |
| --- |
| Type this code into the lobster. |

1. Currently the lobster will spin quite a lot. Let’s refine the code:  
     
   Let's say that a lobster has a 10% chance of turning each frame. We can code this by comparing Greenfoot.getRandomNumber(100) to a given percentage.

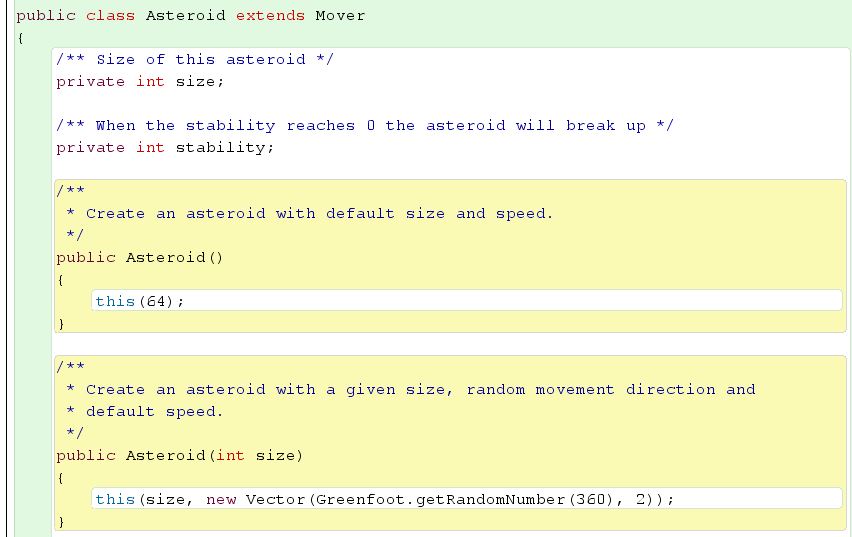
|  |
| --- |
| Add the following code to your lobster class:    Think carefully about how this works -- why do we use **<** rather than **<=**. Could we have coded this differently, for example using Greenfoot.getRandomNumber(50) or Greenfoot.getRandomNumber(10)? What about Greenfoot.getRandomNumber(5)? |

1. We know from our crab that the way to turn left is to use a negative number for the angle. If we could change from turning our lobster in the range 0 to +90 to turning in the range -45 to +45, that would fix our problem. There's a few different ways to achieve this, but here's the simplest: notice that if we subtract 45 from our number, we end up with a number in the right range.

|  |
| --- |
| So let's adjust our code accordingly:    Is our range of turning perfectly symmetric at the moment? Fix it!  Compile and run that, and we should have a somewhat effective predator that can turn towards you at any moment. We will improve this more in the next assignment. |

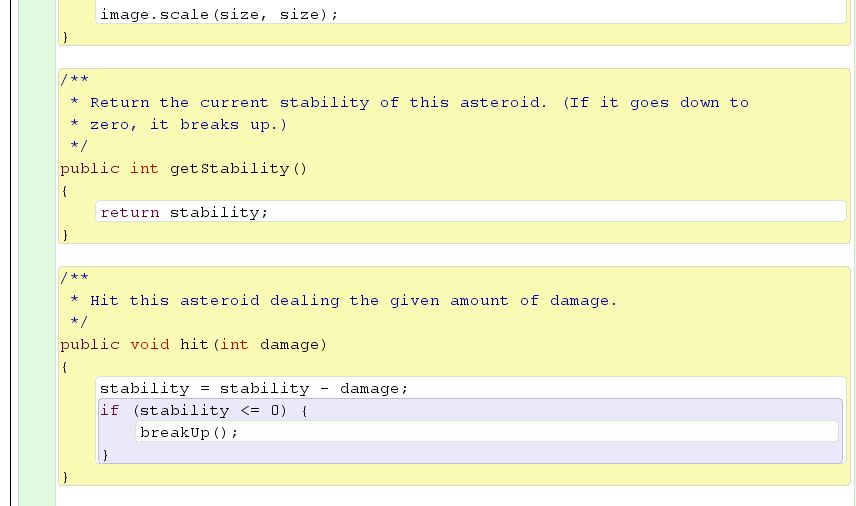
**Conclusion**

1. Look at the following source code:



* 1. What is the name of the class? What is its superclass?
  2. What does the keyword **extends** do for this class?
  3. How many methods are pictured? What are their names and parameters?
  4. What is the access modifier for the classes and methods pictured? What does this access modifier mean?

1. Look at the following source code:



* 1. What is the conditional for the if statement? What is a value of stability that would cause the if structure to be skipped?
  2. What is the parameter of the hit() method? What does it return?

1. If I want to call the Greenfoot class method isKeyDown() into my Crab class, how would I do it? What if I wanted to call moveAround() from an instance of the Lobster class called lobster?
2. What keyword is used in a method signature to enable access from outside objects? What keyword is used to show that a method belongs to a class and not an instance of the class (prevent inheritance of the method by an instance)?