**1.1.5 Images and Sound in Greenfoot**

**Introduction**

We finish our tour of Greenfoot by working with the GreenfootImage class and finalizing our Crab scenario. To do this we will learn about variable **scope** and employ **instance variables** and **local variables**. We will also create a **constructor** for our classes.

Concepts covered are:

* Scope
* Local variable
* Instance variable
* Constructor
* Bug
* Debug
* Comparison operators
* Keywords: **if-else**

**Materials**

* Computer with Greenfoot
* Source files - little-crab.zip in the 1.1.4 folder with all edits from 1.1.4 and 1.1.5

**Activity**

**Part I: Scope**

1. Open the little-crab scenario we’ve been working on.
2. Right-click the crab that was generated in your world and choose Inspect to see the variables. Some variables exist even though we did not declare them - where do you think they came from?

|  |  |
| --- | --- |
|  | You can inspect further by clicking on the arrow next to a variable. For example, take a look at the world variable. Complex types (objects) can contain many primitive pieces of data. |

1. Our goal in this exercise is to animate our actors. To do this in Greenfoot, we store several images and then flip rapidly between them to give the appearance of motion.
   * To do this we need to create variables that will store the image objects (declaration).
   * Next we need to use the GreenfootImage constructor to create a new image and assign it to the variable (assignment).

We could create image variables inside of the act() method, but this would be wasteful as the image objects would be created constantly. It saves memory and space to create the image variables *outside* of the act method. The variables we create could then be employed inside any of the methods in the Crab class.

This brings us to the idea of variable scope. In Java, a variable ‘lives’ in the code block where it was declared. For example, variables can be created inside of methods. These variables have a local **scope**, which means they are created by the method or block of code, used, and when the block of code (method, if, loop, etc) completes the variables are destroyed.

Other variables can be created inside the class before any of the methods are defined. These variables are called instance variables and have a larger **scope**, meaning they can be used by ANY of the methods in the class and are only destroyed when the *object* is destroyed. Instance variables are usually declared using the **private** keyword at the top of the class before method definitions. This is because we rarely share variables between classes. Here’s some sample code:



Notice that the Greenfoot compiler places a colored box behind code blocks. From the highest to lowest scope: green is for classes, yellow is for methods, purple is for code blocks within methods, and white is for instructions. Each block should be nested inside of the higher scope. For example, a green box should completely surround your yellow boxes and yellow boxes should completely surround purple etc.

|  |
| --- |
| At the top of your crab class, declare the image1 variable, which will contain a GreenfootImage object. On the next line declare the image2 variable, which will also contain a GreenfootImage object. |

1. Place a new Crab instance in the world (or press *Reset*) and inspect it. What is the value of image1? The value inside is what variables contain when they are empty.

**Part II: Constructors**

1. Next we need to create a new **constructor**.
   * The constructor is always the first method we create inside of a class
   * It always has the same name as the class
   * Constructors have no return type
   * Constructors are automatically executed when a new object is **instantiated** (created) from the class

|  |
| --- |
| Create a constructor for the Crab class. |

|  |
| --- |
| 1. We will now assign image1 and image2 respectively to **new** GreenfootImage objects containing “crab.png” and “crab2.png”. |

1. Look at the Javadoc for the Actor class in Greenfoot. There are two methods that are used to manipulate images in Greenfoot. You may have used one or both in the last assignment. What are the two methods, what are their parameters, and what do they return?

|  |
| --- |
| Still inside the constructor, use the correct method to change the crab’s starting image to image1. |

1. To complete the animation, we need to place an **if-else structure** into a new method called switchImage(). This structure is similar to what we already know. Remember: the **if** keyword is followed by parentheses around a boolean or boolean statement and an open curly brace. Lines of code come next and a closed curly brace to end the block. If the boolean in parentheses is true the **if** code executes. An **else** keyword is next but with no condition - just code between curly braces. The **else** executes when the boolean in the parentheses is false.

We also need to know how to use comparison operators (**==**). Remember that one **=** is for assigning a value to a variable. The **==** on the other hand seeks to compare two sides of an equation. The comparison operators are:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| is equal to | is not equal | greater than | less than | greater than or equal to | less than or equal to |
| **==** | **!=** | **>** | **<** | **>=** | **<=** |

|  |
| --- |
| Write code into the switchImage() method. Use this pseudocode:  If (our current image is image1) then use image2 now.  Else use image1 now.  -->Don’t forget your Javadoc comment for this new method! |

**Part III: Winning the Game**

1. Our last task is to refine our win condition to make it more like a video game. We will pseudocode our condition like this:

|  |
| --- |
| Inside the Crab class, create a new instance variable called wormsEaten  Initialize wormsEaten to zero at the start of the game within the constructor.  Increment wormsEaten each time one of the worms is destroyed within the lookForWorms method. This means add one to it each time a worm is eaten.  If (wormsEaten is greater than or equal to 10) then stop the game and play “fanfare.wav”.  Add a lose condition - play “au.wav” when the lobster kills the crab then make the code stop. Use the API to find the method if you need help. |

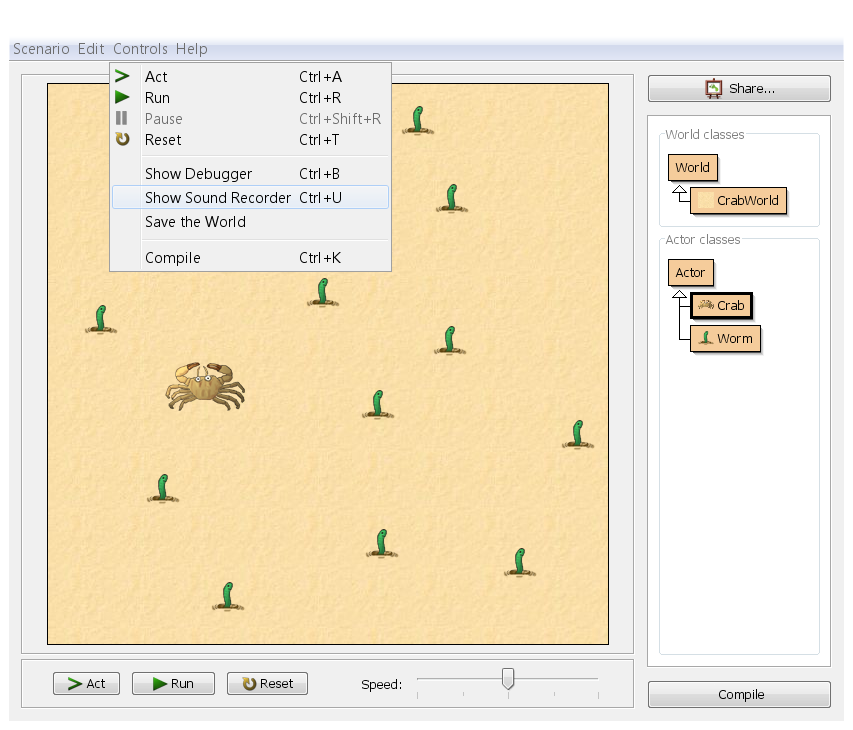
Hints: Debug your code by inspecting variables during the game. You can pause the action and inspect.

1. Make one of these modifications to the completed game

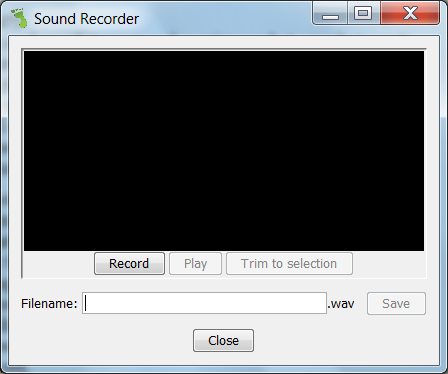
|  |
| --- |
| * + Make this a two-player game by introducing a new player-controlled class with different controls   + Make new worms pop up at random times or when one is eaten |

**Part IV: Adding Sounds**

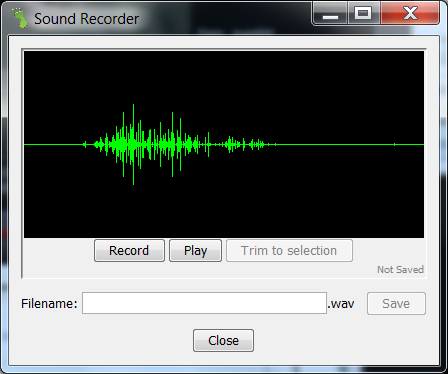
1. We can add some sound to our scenario. The scenario comes with a sound ready for you to use, named "slurp.wav". We can make that sound play every time the crab eats a worm by adding a single line to our crab class that calls Greenfoot.playSound(“slurp.wav”);
2. Don't forget to turn your speakers on (or plug in your headphones). One last thing -- if you have a microphone for your computer, you can record your own sounds. In the Controls menu there is an option to show the sound recorder:



1. Select that, and you'll get the sound recorder:



1. Press the record button and speak (or scrunch an empty food packet, or whatever!), then press stop. You should see a green wave, and when you press play you should hear your noise played back to you. If not, there is a problem with your microphone -- try googling to get help with that. Assuming it does have some sound, you'll almost invariably have a bit of silence at the beginning and end of the sound -- you can see this in the green display, as it will have a flat horizontal line at the beginning and end before the shape:

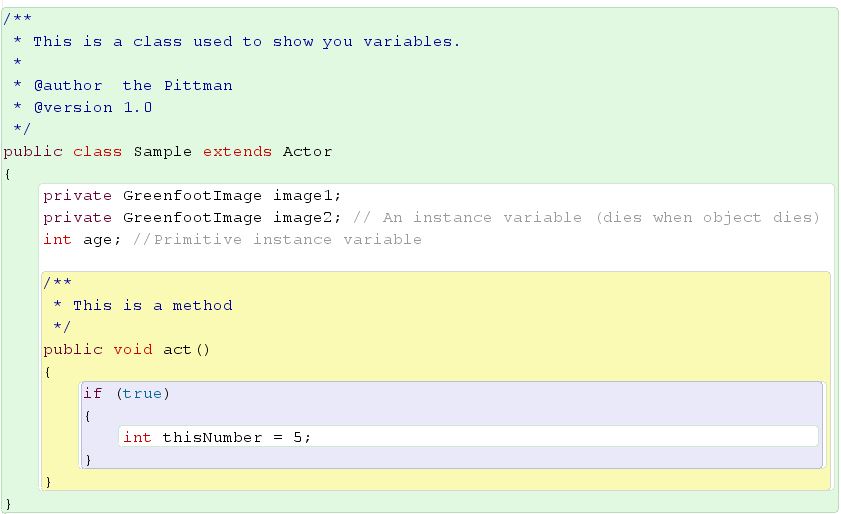


1. Silence at the end isn't much of a problem, but silence at the beginning is irritating -- it means that when you tell the sound to play when a worm is eaten, there will appear to be a short delay before the sound starts playing, as if your game is lagging. You can clean up the silence by selecting the bit in the middle (the bit you want to keep) by clicking at the beginning (after the initial silence) and dragging to the end (before the final silence) -- the selection should be shown in grey. Then press "Trim to selection". The silence should be removed.  
     
   Save the sound by entering something in the filename box (e.g. "myeating") then pressing Save. Close the sound recorder and go back to your code. Find the line with "eating.wav" and change it to "myeating.wav" (or whatever name you used, plus the .wav extension). Then when you play your game, you should hear your own sound playing.

**Conclusion**

Answer the following questions:

1. What is variable scope? What is the main difference between an instance variable and a local scope variable?
2. Look at the following code:



What are the names of the declared variables? Which of them are instance variables and which are local variables?

For the local variable(s), at what point will the program destroy the variable(s)?

What are the values assigned to each of the variables?

1. What is the name of the constructor for the Lobster class? What does it mean to instantiate an object from a class? What keyword is used before the constructor to instantiate an object?
2. Write the line of code that would call the constructor for class Tortoise to instantiate a new tortoise into the variable mutantNinja.