Introduction to Programming

with Greenfoot Java [Ver. 2.0]

Unit 1: First Program

CHAPTER 4: FINISH THE CRAB GAME

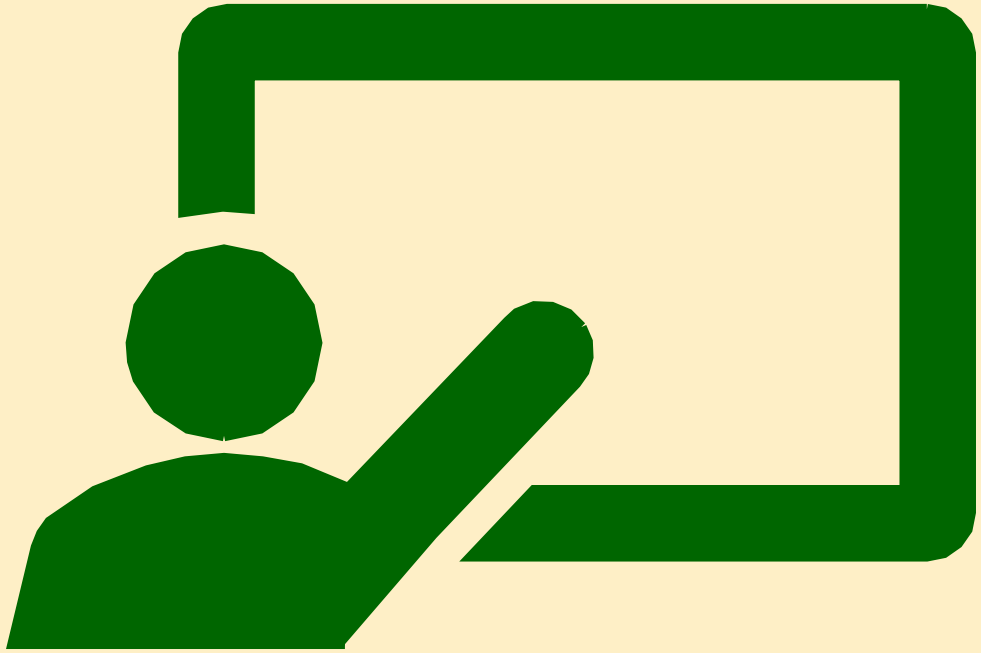
DR. ERIC CHOU IEEE SENIOR MEMBER

Finishing the Crab Game

Topics: world initialization, setting images, animating images

Concepts: Constructors, state, variables (instance variables and local variables), assignment, new (creating objects programmatically)

- In this chapter, we will finish the crab game. “Finish” here means that this is where we stop discussing this project in this book. Of course, a game is never finished—you can always think of more improvements that you can add. We will suggest some ideas at the end of this chapter. First, however, we will discuss a number of improvements in detail.



Adding Objects Automatically

LECTURE 1

Adding Objects Automatically

- We are now getting close to having a playable little game. However, a few more things need to be done. The first problem that should be addressed is the fact that we always have to place the actors (the crab, lobsters, and worms) manually into the world. It would be better if that happened automatically.

Adding Objects Automatically

- There is one thing that happens automatically every time we successfully compile: the world itself is created. The world object, as we see it on screen (the sand-colored square area), is an instance of the **CrabWorld** class. World instances are treated in a special way in Greenfoot: while we have to create instances of our actors ourselves, the Greenfoot system always automatically creates one instance of our world class and displays that instance on screen.
- Let us have a look at the **CrabWorld**'s source code (Code 4.1). (If you do not have your own crab game at this stage, use little-crab-4 for this chapter.)

```
import greenfoot.*;  // (Actor, World, Greenfoot, GreenfootImage)

public class CrabWorld extends World
{
    /**
     * Create the crab world (the beach). Our world has a size
     * of 560x560 cells, where every cell is just 1 pixel.
     */
    public CrabWorld()
    {
        super(560, 560, 1);
    }
}
```

Code 4.1 Source code of the CrabWorld class

Adding Objects Automatically

- In this class, we see the usual **import** statement in the first line. (We will discuss this statement in detail later—for now it is enough to know that this line will always appear at the top of our Greenfoot classes.)
- Then follows the class header, and a comment (the block of lines in a blueish color starting with asterisks—we have encountered them already in the last chapter). Comments usually start with a /** symbol and end with */.

Adding Objects Automatically

Next comes the interesting part:

```
public CrabWorld(){
    super(560, 560, 1);
}
```

Concept

A constructor of a class is a special kind of method that is executed automatically whenever a new instance is created.

Adding Objects Automatically

- This is called the constructor of this class. A constructor looks quite similar to a method, but there are some differences:
    1. A constructor has no return type specified between the keyword “public” and the name.
    2. The name of a constructor is always the same as the name of the class.
- A constructor is a special kind of method that is always automatically executed whenever an instance of this class is created. It can then do what it wants to do to set up this new instance into a starting state.

Adding Objects Automatically

- In our case, the constructor sets the world to the size we want (560 by 560 cells) and a resolution (1 pixel per cell). We will discuss world resolution in more detail later in the book.
- Since this constructor is executed every time a world is created, we can use it to automatically create our actors. If we insert code into the constructor to create an actor, that code will be executed as well. For example:

```
public CrabWorld() {
  super(560, 560, 1);
  Crab myCrab = new Crab();
  addObject(myCrab, 250, 200);
}
```

Adding Objects Automatically

- This code will automatically create a new crab, and place it at location x=250, y=200 into the world. The location 250,200 is 250 cells from the left edge of the world, and 200 cells from the top. The origin—the 0,0 point—of our coordinate system is at the top left of the world (Figure 4.1).
- We are using four new things here: a variable, an assignment, the new statement to create the new crab, and the **addObject** method. Let us discuss these elements one by one.

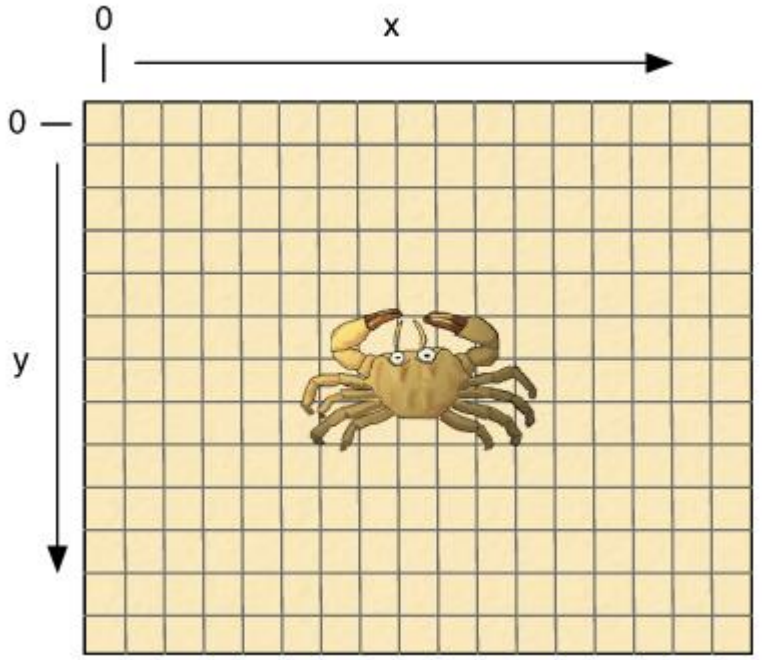
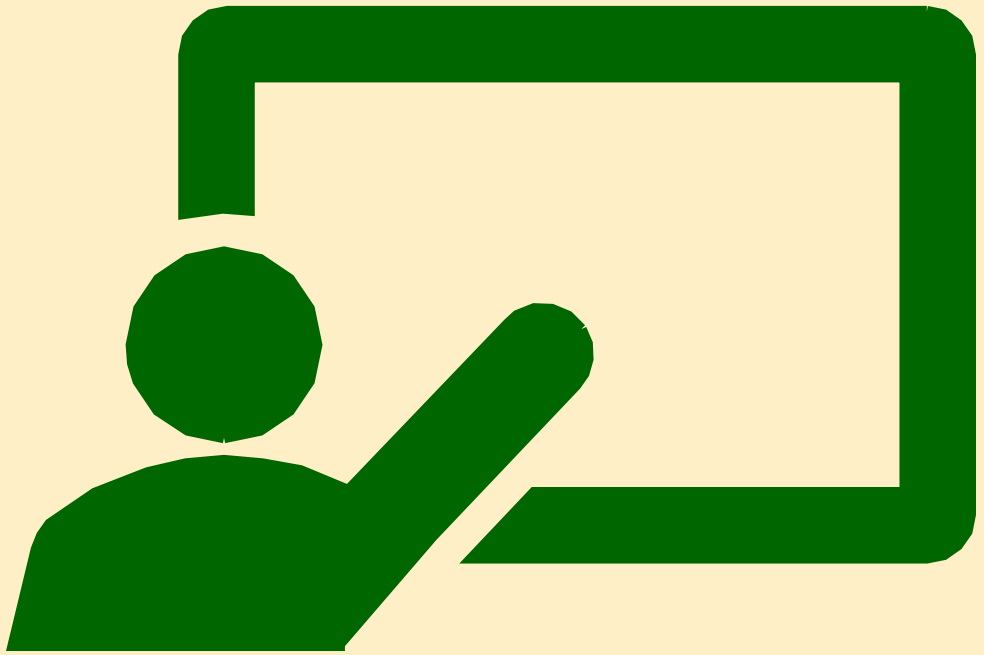


Figure 4.1 The coordinate system of the world



Creating New Objects

LECTURE 2

Creating New Objects

- If we want to add a crab into the world, the first thing we need is a crab. The crab in our case is an object of the Crab class. Previously, we have created crab objects interactively, by right-clicking the Crab class and selecting “new Crab()” from the pop-up menu.
- Now, we want our constructor code to create the new crab object for us automatically.
- The Java keyword new allows us to create new objects of any of the existing classes. For example, the expression

Creating New Objects

Concept

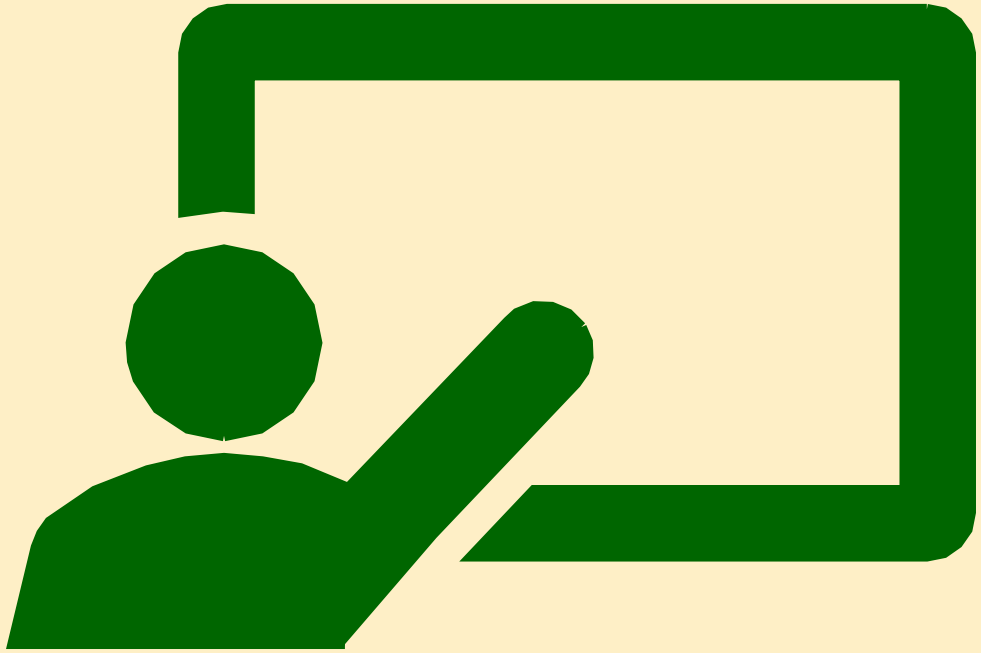
Java objects can be created programmatically (from within your code) by using the new keyword.

new Crab()

creates a new instance of class **Crab**. The expression to create new objects always starts with the keyword **new**, followed by the name of the class we wish to create and a parameter list (which is empty in our example). The parameter list allows us to pass parameters to the new object's constructor. Since we did not specify a constructor for our **Crab** class, the default parameter list is empty. (You might have noticed that the instruction that we select from the class's pop-up menu to create objects is exactly this statement.)

Creating New Objects

- In our constructor code above, you find the **new Crab()** instruction as the right half of the first line we inserted.
- When we create a new object, we have to do something with it. In our case, we **assign it** to a variable.



Variables

LECTURE 3

Variables

- In programming, we often need to store some information to remember and use it later. This is done by using variables.

Concept

Variables can be used to store information (objects or values) for later use.

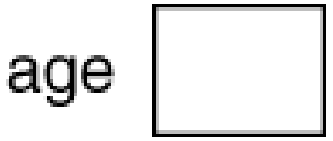


Figure 4.2 An (empty) variable

- A variable is a bit of storage space. It always has a name to refer to it. When we draw diagrams of our objects or code fragments, we usually draw variables as white boxes with their name to the left side.
- Figure 4.2, for example, shows a variable called age. (We might want to store the age of the crab.)

Variables

- Variables also have a type. The type of a variable states what kind of data can be stored in it. For example, a variable of type **int** can store whole numbers, a variable of type **boolean** can store true/false values, and a variable of type **Crab** can store crab objects.

Concept

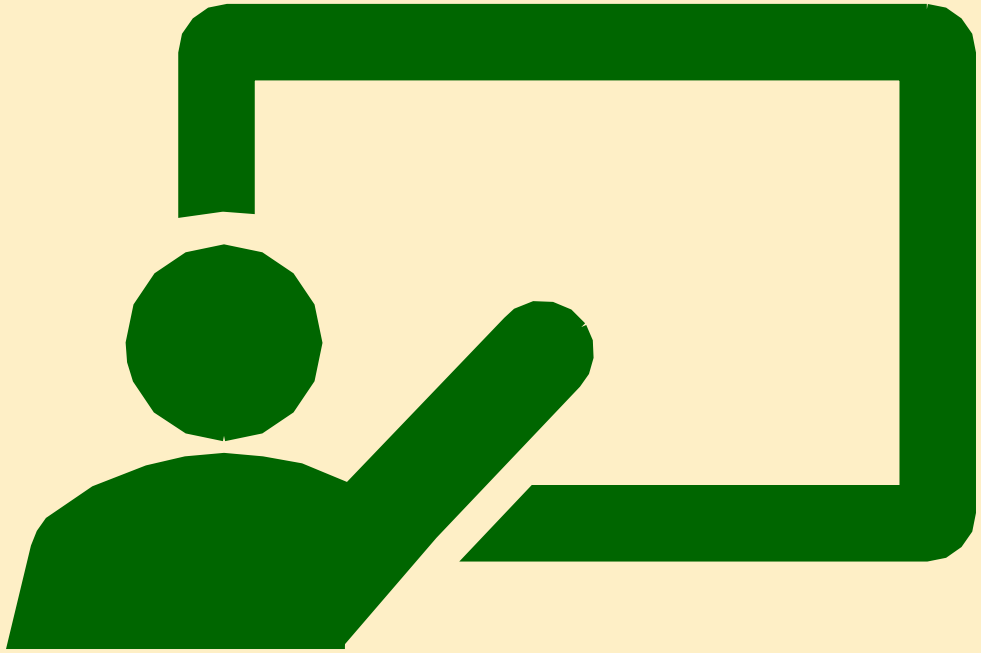
Variables can be created by writing a variable declaration.

Variables

- In our source code, when we need a variable, we can create one by writing a variable declaration. A variable declaration is very simple: we just write the type and the name of the variable we want, followed by a semicolon. For example, if we want an **age** variable as shown in Figure 4.2 to store whole numbers, we can write

```
int age;
```

- This will create our **age** variable, ready to store **int** values.



Assignment

LECTURE 4

Assignment

- Once we have a variable, we are ready to store something in it. This is done using an assignment statement.

Concept

We can store values into variables by using an assignment statement (=).

- An assignment is a Java instruction written as an equal sign: =.

Assignment

- For example, to store the number 12 into our age variable, we can write

```
age = 12;
```

- It is best to read assignment statements from right to left: The value 12 is stored into the variable age. After this assignment statement is executed, our variable will hold the value 12. In our diagrams, we show this by writing the value into the white box (Figure 4.3).

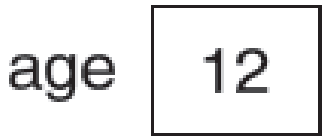


Figure 4.3 A variable storing an integer value

Assignment

- The general form of an assignment statement

```
isvariable = expression;
```

- That is, on the left hand side is always the name of a variable, and on the right hand side is an expression that is evaluated, and its value is stored into the variable.Often in our programs, we want to declare a variable and store a value in it. So often we will find the variable declaration and assignment statements together:

Assignment

```
int age;
age = 12;
```

- Because this is so common, Java allows us to write these two statements together in one line:

```
int age = 12;
```

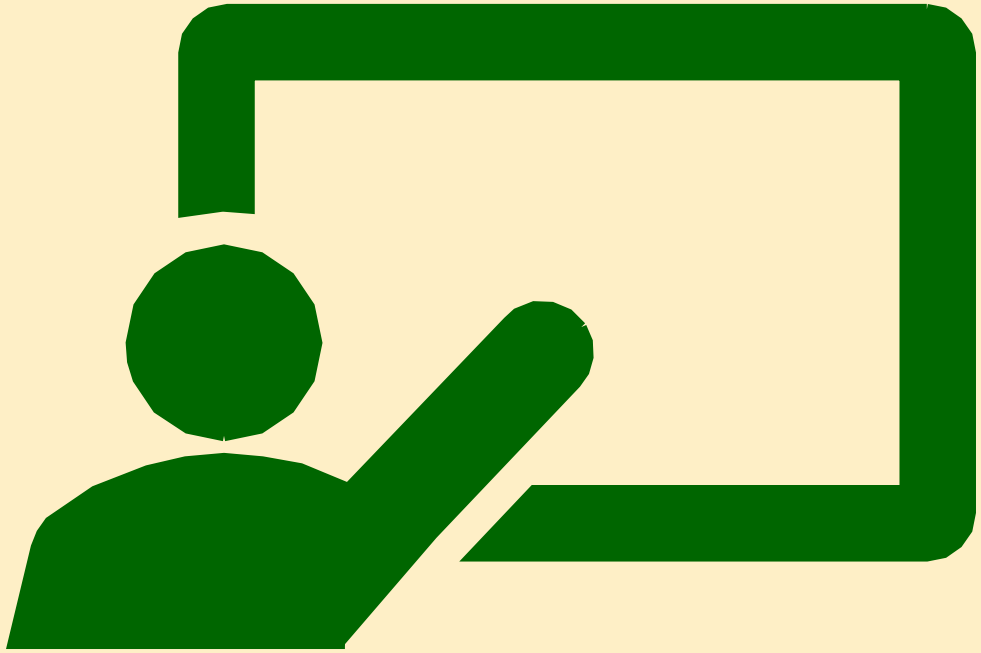
- This single line creates the integer variable **age** and assigns the value 12 to it. It does exactly the same as the two-line version above.

Assignment

- Assignments overwrite any value previously stored in it. Thus, if we have our age variable now storing the value 12, and then we write

```
age = 42;
```

- the **age** variable will now store the value 42. The 12 is overwritten, and we cannot get it back.



Object Variables

LECTURE 5

Object Variables

- We mentioned above that variables cannot only store numbers, they can also store objects.
- Java distinguishes primitive types and object types. Primitive types are a limited set of often used data types, such as int, boolean and char. There are not many of them—Appendix D lists all primitive types in Java.

Concept

Variables of primitive types store numbers, booleans and characters; variables of object types store objects.

Object Variables

- Every class in Java also defines a type, and these are called **object types**. So with our class **Crab**, for example, we get a type **Crab**, our **Lobster** class defines a type **Lobster**, and so on. We can declare variables of these types:

```
Crab myCrab;
```

- Note that again, as before, we write the type at the front (**Crab**), followed by the name which we can make up (**myCrab**), and a semicolon.

Object Variables

- Once we have an object variable, we can store objects into it. We can now put this together with our instruction to create a crab object, which we saw in Section 4.2 .

```
Crab myCrab;
myCrab = new Crab();
```

- As before, we can also write this in a single line:

```
Crab myCrab = new Crab();
```

Object Variables

This single line of code does three things:

- It creates a variable called myCrab of type Crab.
- It creates a crab object (an object of type Crab).
- It assigns the crab object to the myCrab variable.

Concept

Objects are stored in variables by storing a reference to the object.

- When we have an assignment statement, the right hand side of the assignment is always executed first (the crab object is created), and then the assignment to the variable on the left takes place.

Object Variables

- In our diagrams, we draw object variables storing objects using an arrow (Figure 4.4). Here, the variable **myCrab** stores a reference to the crab object. The fact that object variables always store references to the objects (and not objects directly) will become important later, so we will be careful to always accurately draw it like this.

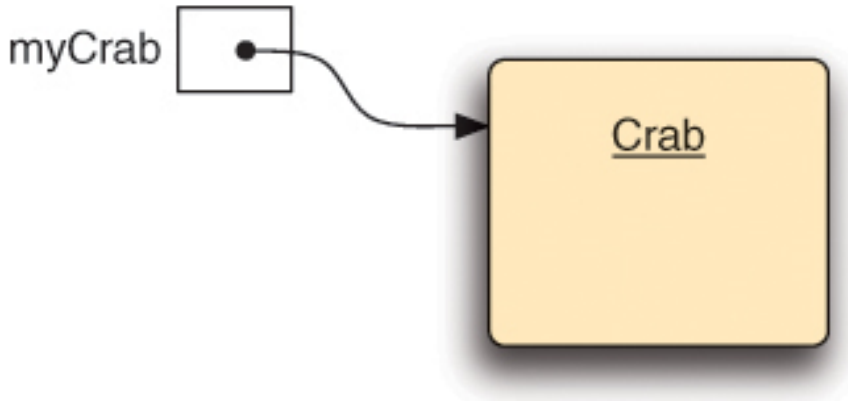
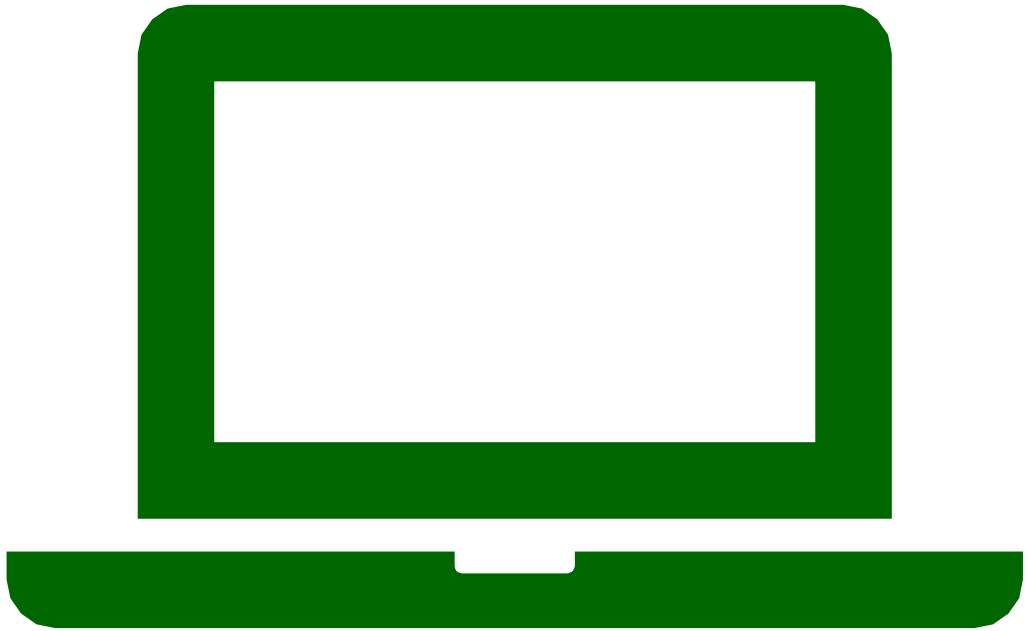


Figure 4.4 An object variable storing a reference to an object

Object Variables

- The type of the variable and the type of the value assigned to it must always match. You can assign an **int** value to an **int** variable, and you can assign a **Crab** object to a **Crab** variable. But you cannot assign a **Crab** object to an **int** variable (or any other non-matching combination).
- **Note:**

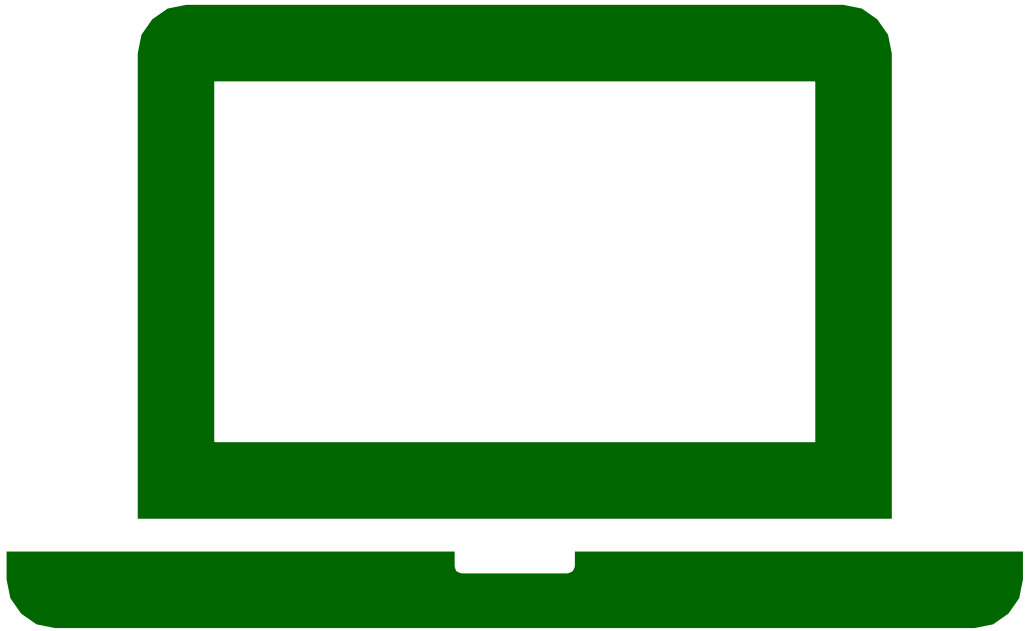
  When we say “the types must match,” this does not actually mean that they must be the same. There are situations where types match that are not the same. For example, we can assign a subclass to a superclass type, such as assigning a Crab to an Actor variable (because a crab is an actor). These are subtleties that we shall discuss later.



Exercise 4.1

Exercise 4.1

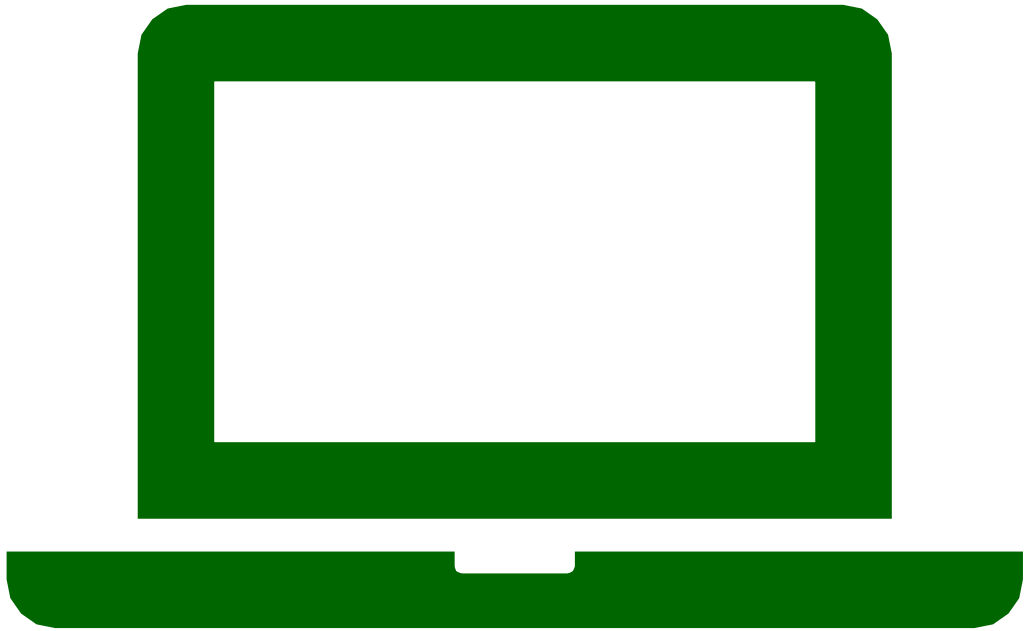
- Write a variable declaration for a variable of type **int** where the variable has the name “score.”



Exercise 4.2

Exercise 4.2

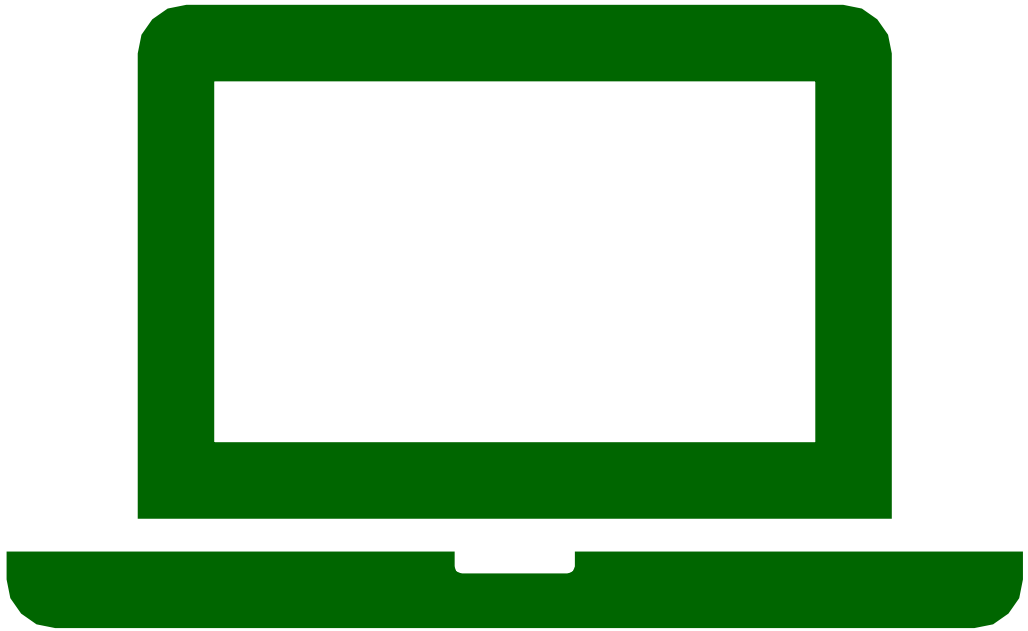
- Declare a variable named “isHungry” of type **boolean**, and assign the value “true” to it.



Exercise 4.3

Exercise 4.3

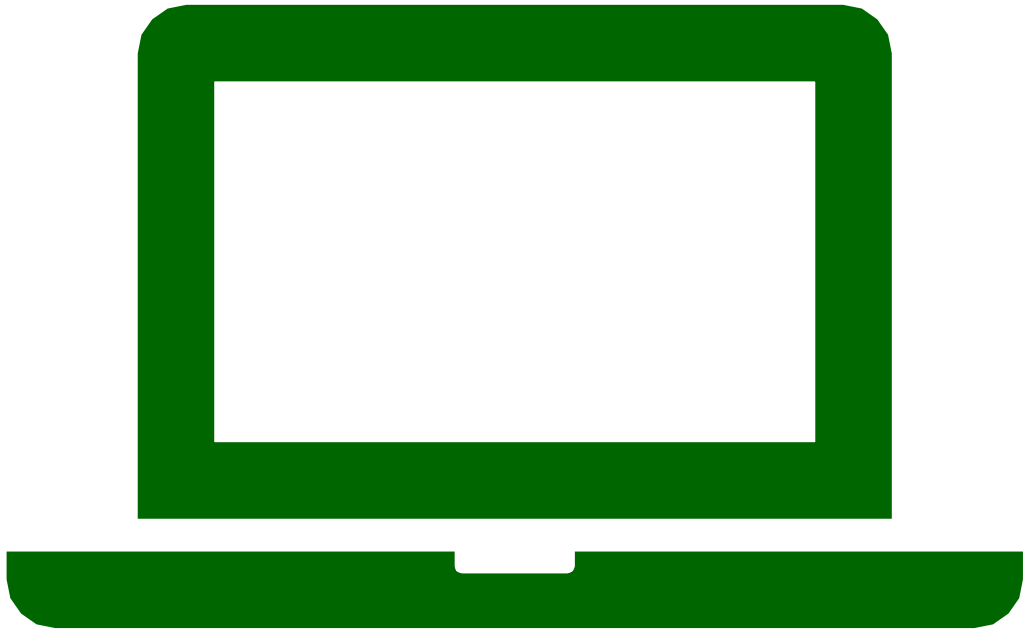
- Declare a variable named “year” and assign the value 2014 to it. Then assign the value 2015.



Exercise 4.4

Exercise 4.4

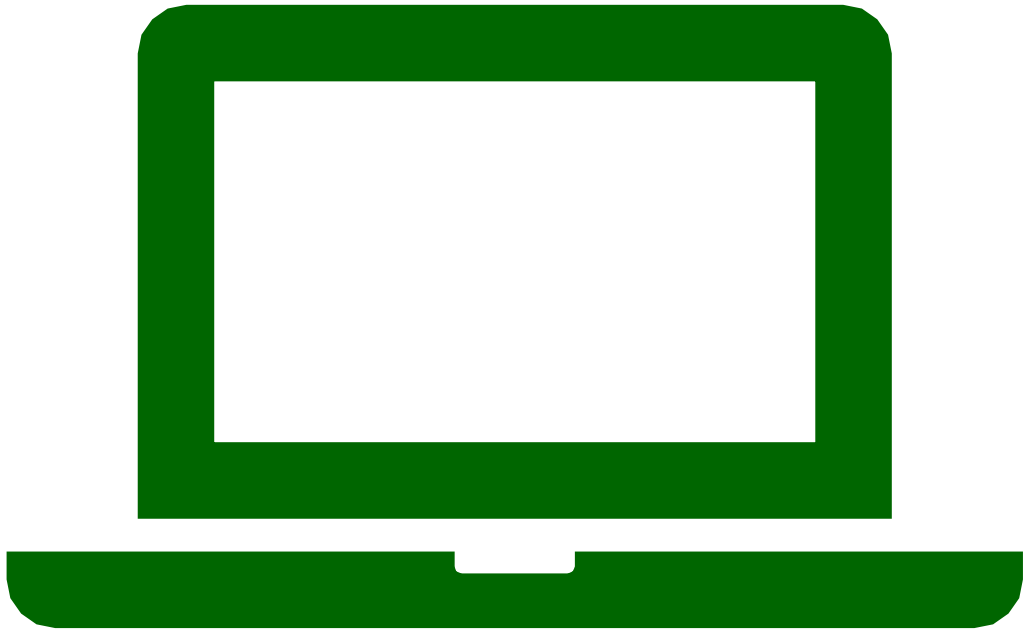
- Declare a variable of type **Crab**, named “littleCrab,” and assign a new crab object to it.



Exercise 4.5

Exercise 4.5

- Declare a variable of type **Control**, named “inputButton,” and create and assign an object of type **Button**.

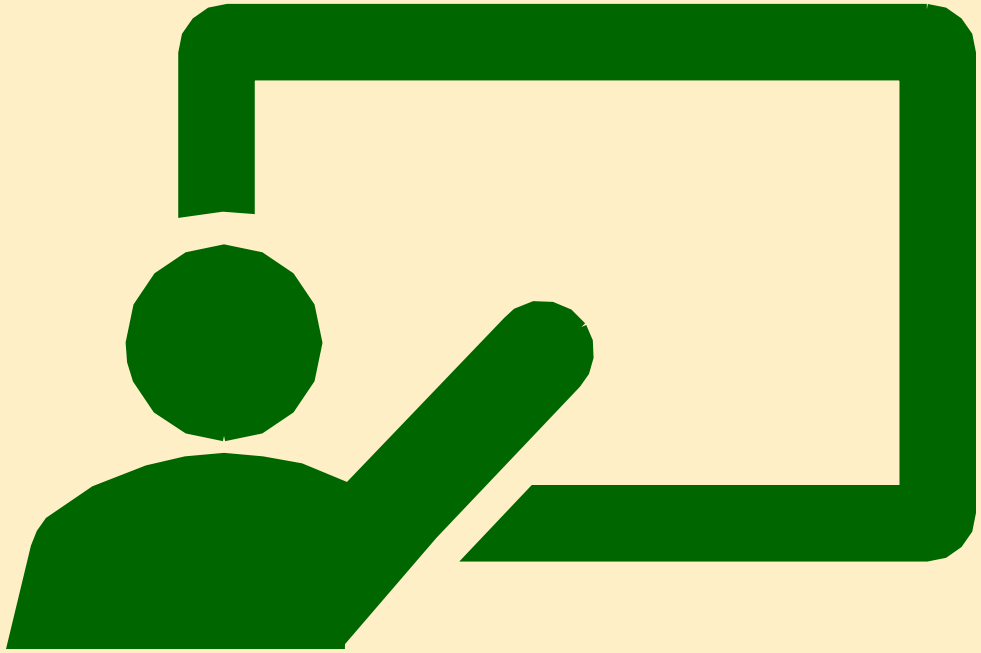


Exercise 4.6

Exercise 4.6

- What is wrong with the following statement:

```
int myCrab = new Crab();
```



Using Variables

LECTURE 6

Using Variables

- Once we have declared a variable and assigned a value, we can use it by just using the name of the variable.
- For example, the following code declares and assigns two integer variables:

```
int n1 = 7;
int n2 = 13;
```

- We can then use them on the right hand side of another assignment:

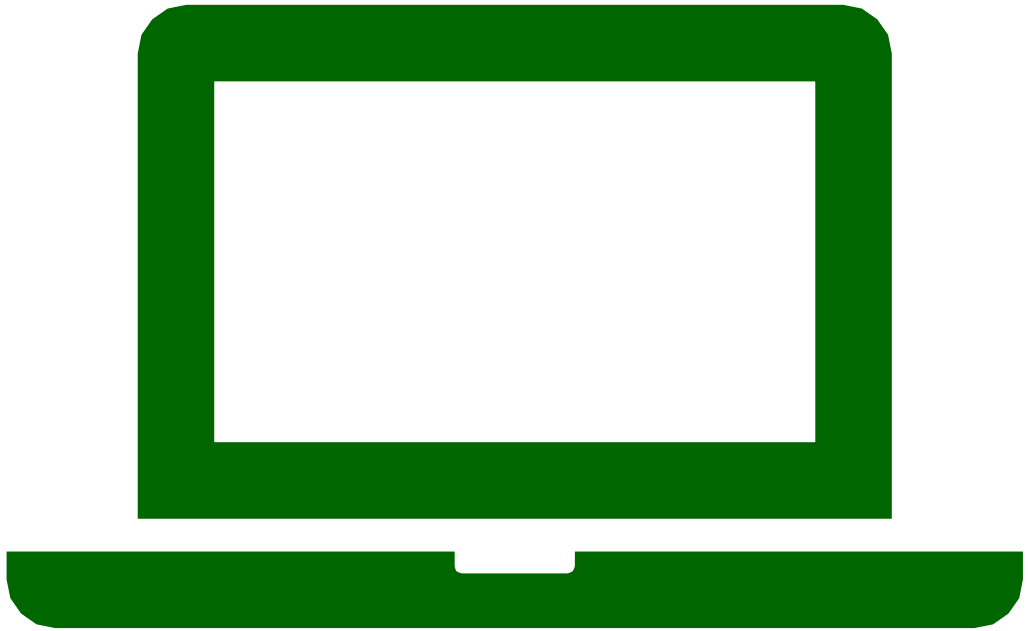
```
int sum = n1 + n2;
```

Using Variables

- After this statement, sum contains the sum of n1 and n2. If we write

```
n3 = n1;
```

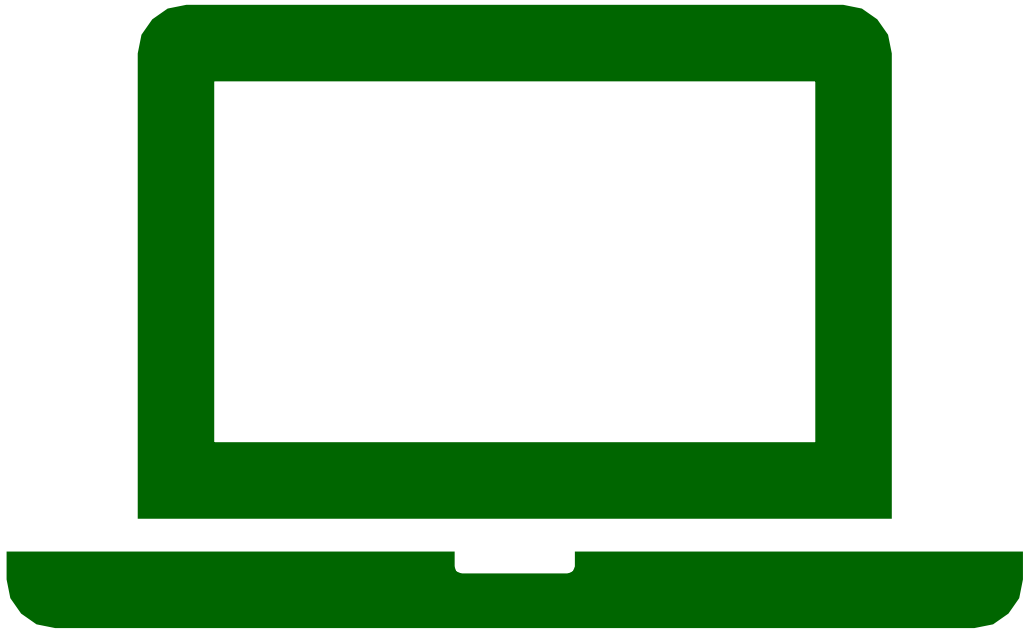
- then the value of **n1** (7, in this case) will be copied into **n3** (assuming a variable **n3** has been declared previously). **n1** and **n3** now both contain the value 7.



Exercise 4.7

Exercise 4.7

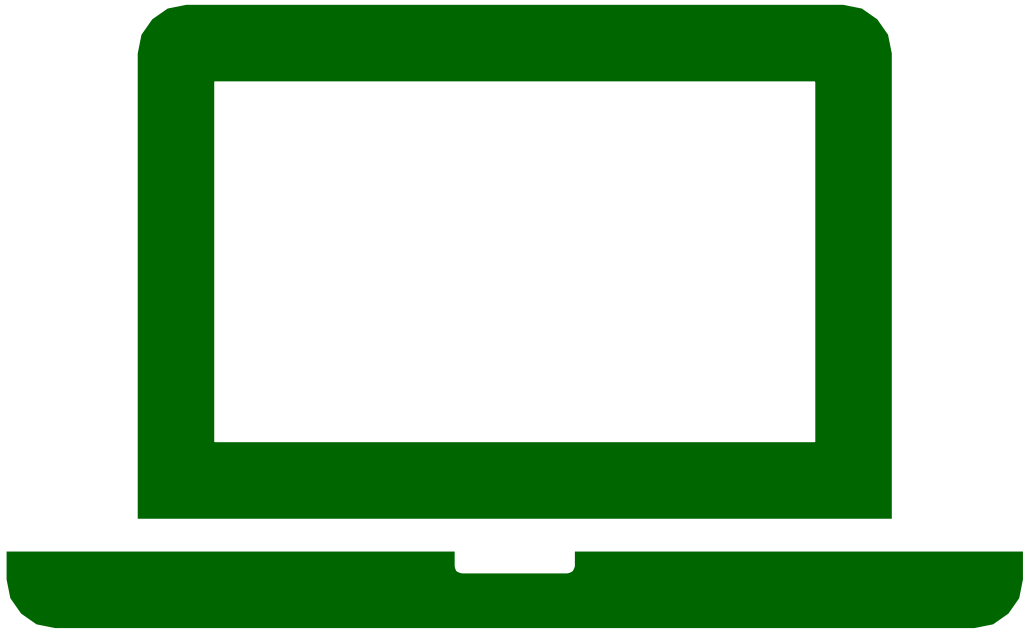
- Declare a variable called **children** (of type **int**). Then write an assignment statement that assigns to this variable the sum of two other variables named **daughters** and **sons**.



Exercise 4.8

Exercise 4.8

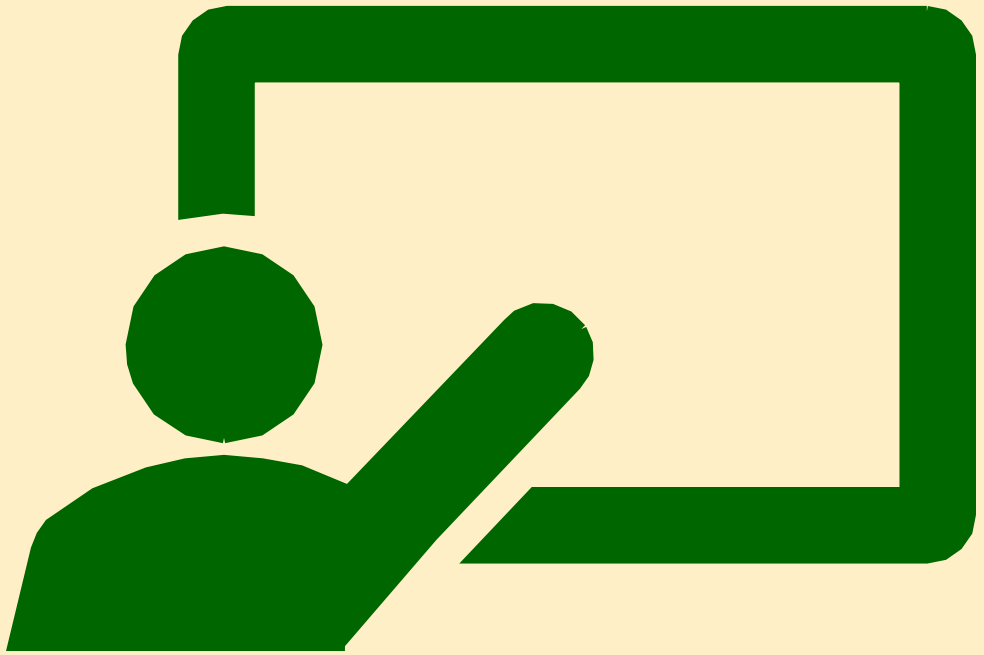
- Declare a variable named **area** of type **int**. Then write an assignment statement that assigns to **area** the product of two variables called **width** and **length**.



Exercise 4.9

Exercise 4.9

- Declare two variables **x** and **y** of type **int**. Assign the values 23 to **x** and 17 to **y**. Then write some code to swap those values (so that afterwards **x** contains 17, and **y** contains 23).



Adding Objects to the World

LECTURE 7

Adding Objects to the World

- We have now seen how we can create a new crab and store it in a variable. The last thing to do is to add this new crab into our world.
- In the code fragment in the constructor code shown in Section 4.1, we have seen that we can use the following line:

```
addObject(myCrab, 250, 200);
```

Adding Objects to the World

- The **addObject** method is a method of the **World** class, and it allows us to add an actor object to the world. We can look it up by looking at the class documentation for class **World**. There we see that it has the following signature:

```
void addObject(Actor object, int x, int y)
```

Adding Objects to the World

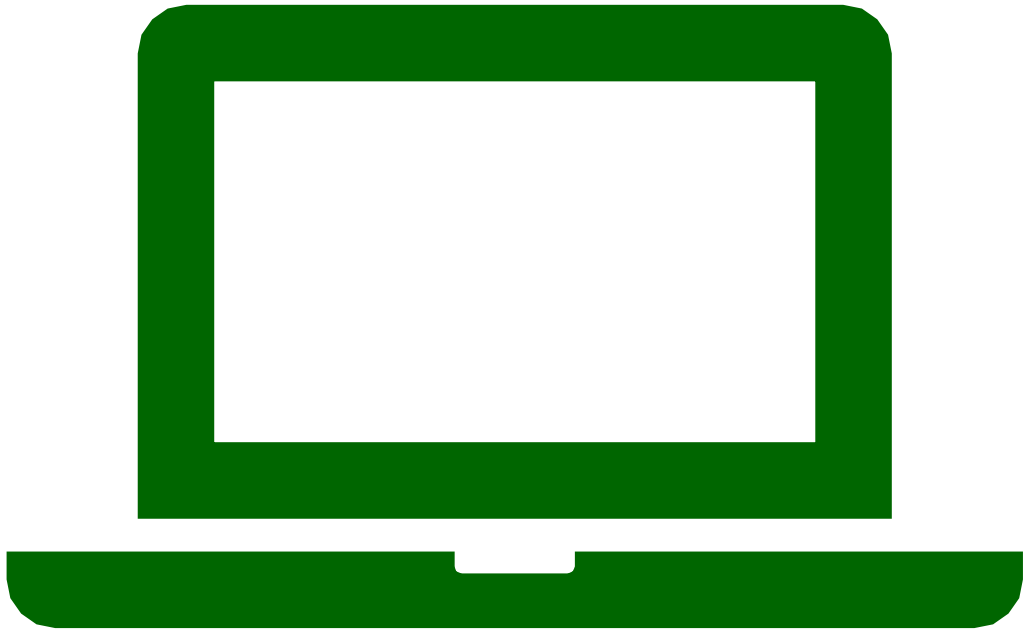
- Reading the signature from start to finish, this tells us the following:
  - The method does not return a result (void return type).
  - The name of the method is addObject.
  - The method has three parameters, named object, x, and y.
  - The type of the first parameter is Actor, the type of the other two is int.
- This method can be used to add a new actor into the world. Since the method belongs to the **World** class and **CrabWorld** is a World (it inherits from the **World** class), this method is available in our **CrabWorld** class, and we can just call it.

Adding Objects to the World

- We have just created a new crab and stored it in our **myCrab** variable. Now we can use this crab (by using the variable it is stored in) as the first parameter to the **addObject** method call. The remaining two parameters specify the x and y coordinate of the position where we wish to add the object.
- All the constructs together (variable declaration, object creation, assignment, and adding the object to the world) look like this:

```
Crab myCrab = new Crab();
addObject(myCrab, 250, 200);
```

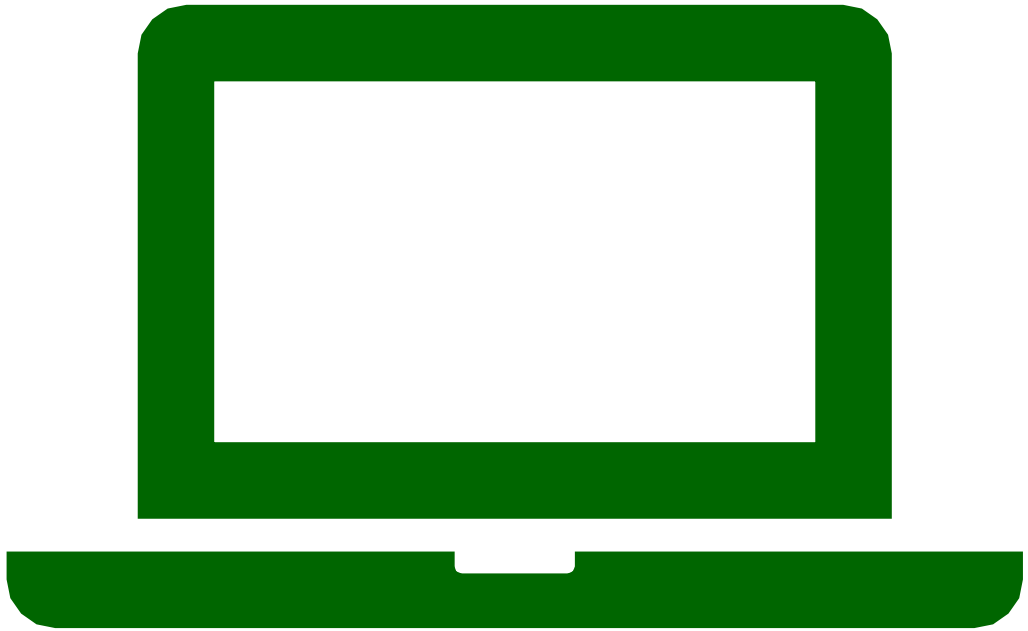
- We can use an object of type **Crab** for the **Actor** parameter, because a crab is an actor (class **Crab** is a subclass of class **Actor**).



Exercise 4.10

Exercise 4.10

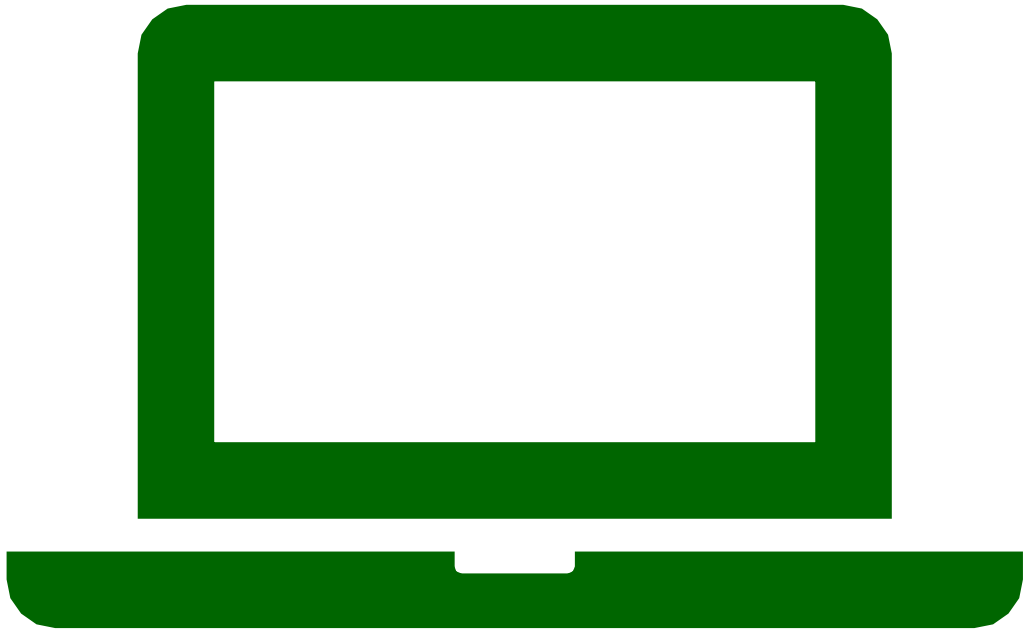
- Add code to the **CrabWorld** constructor of your own project to create a crab automatically, as discussed above.



Exercise 4.11

Exercise 4.11

- Add code to automatically create three lobsters in the **CrabWorld**. You can choose arbitrary locations for them in the world.



Exercise 4.12

Exercise 4.12

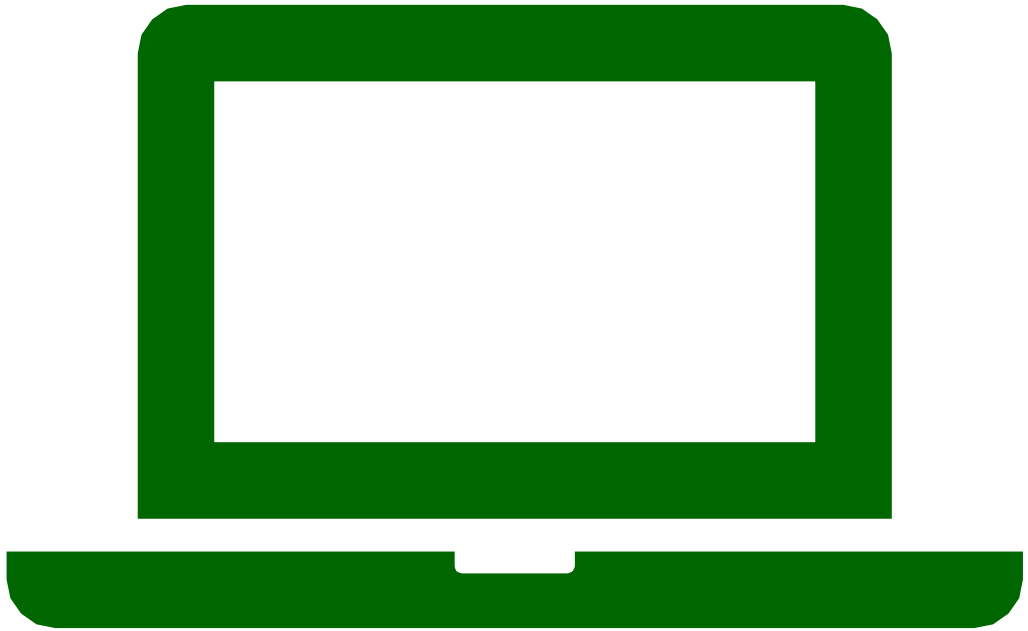
- Add code to create two worms at arbitrary locations in the **CrabWorld**.

Save the World

LECTURE 8

Save the World

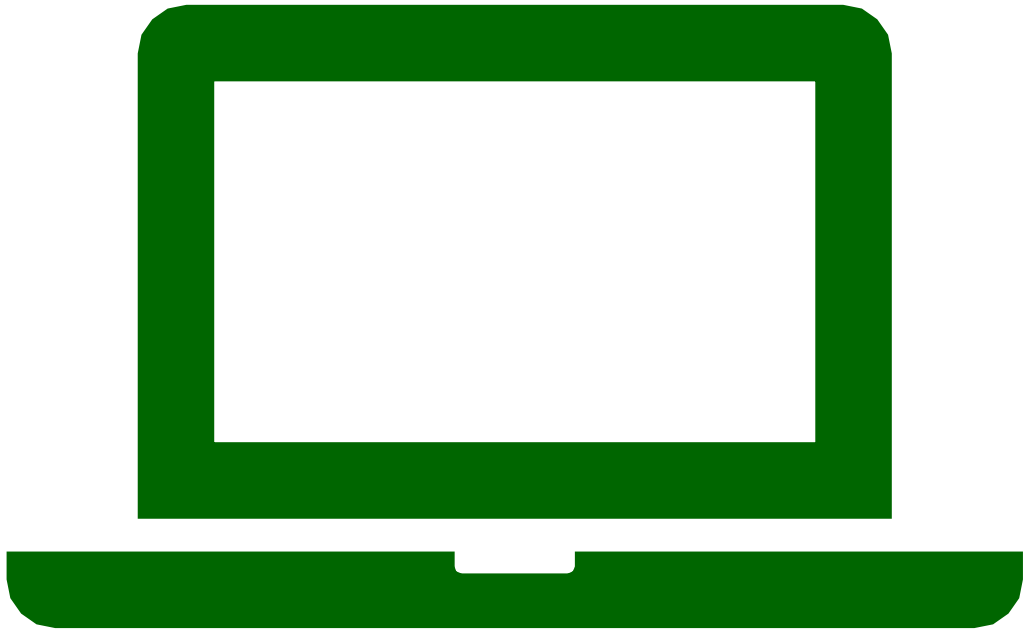
- We will now introduce an easier method to achieve the same thing.
- First, remove the code again that you introduced in the last set of exercises, so that the objects are not created automatically. When you compile your scenario again, the world should be empty. Then do the following exercises.



Exercise 4.13

Exercise 4.13

- Compile your scenario. Then place the following actors into your world (interactively): one crab, three lobsters, and ten worms.



Exercise 4.14

Exercise 4.14

Right-click on the world background. The world's context menu will pop up. From this menu, select Save the World.

- When you place some objects into your world and then select the Save the World function (Figure 4.5), you will notice that your **CrabWorld** source code opens, and some new code has been inserted in this class. Study this code carefully.

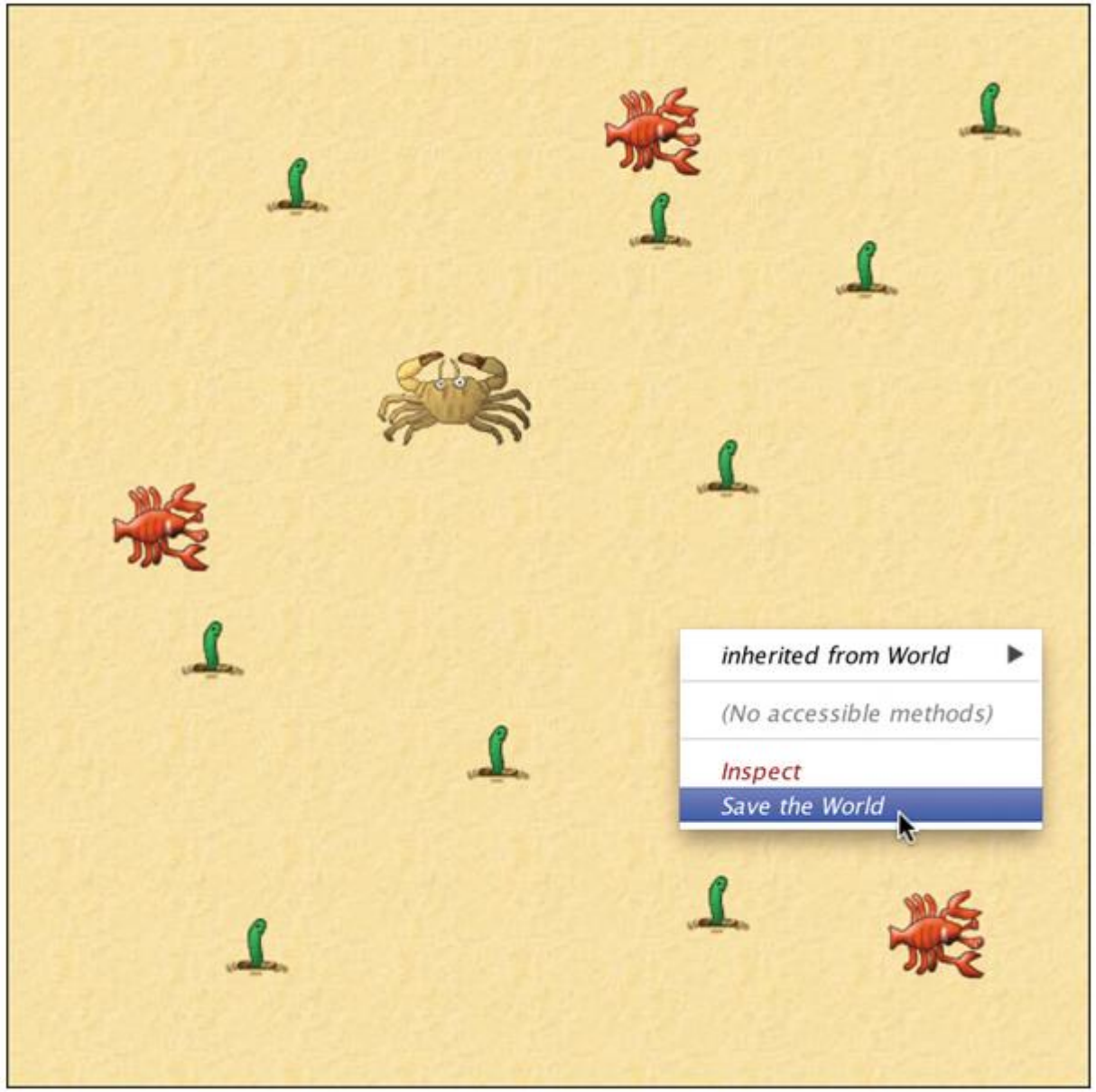


Figure 4.5 The "Save the World" function

Exercise 4.14

You will see that this code does the following:

- The constructor now includes a call to a new method named **prepare()**.
- A method definition for this method has been added.
- The **prepare()** method contains code that creates and adds all the actors that we have just created interactively.

So what is happening here?

Exercise 4.14

- When we create objects interactively, and then select Save the World, Greenfoot writes code into our world class to recreate the situation just as we set it up by hand. It does this by creating and calling a method called **prepare()**.
- The effect is that now, every time we click Compile or Reset, the actors are immediately created again

Exercise 4.14

- Our previous exercises to write the code manually to create and place the actors help us understand how this method works. In many cases, we do not need to write this code manually—and can use Save the World instead—but is important to understand it in detail. There are other occasions where we want a more sophisticated setup, where we will still write the initialization code by hand.

Animating Images

LECTURE 9

Animating Images

- Now that we have managed to start our game off with a good setup automatically, we can spend a bit of time improving some details.
- We will next work on animating the image of the crab. To make the movement of the crab look a little better, we plan to change the crab so that it moves its legs while it is walking.
- Animation is achieved with a simple trick: we have two different images of the crab (in our scenario, they are called crab.png and crab2.png), and we simply switch the crab's image between these two versions fairly quickly. The position of the crab's legs in these images is slightly different (Figure 4.6).

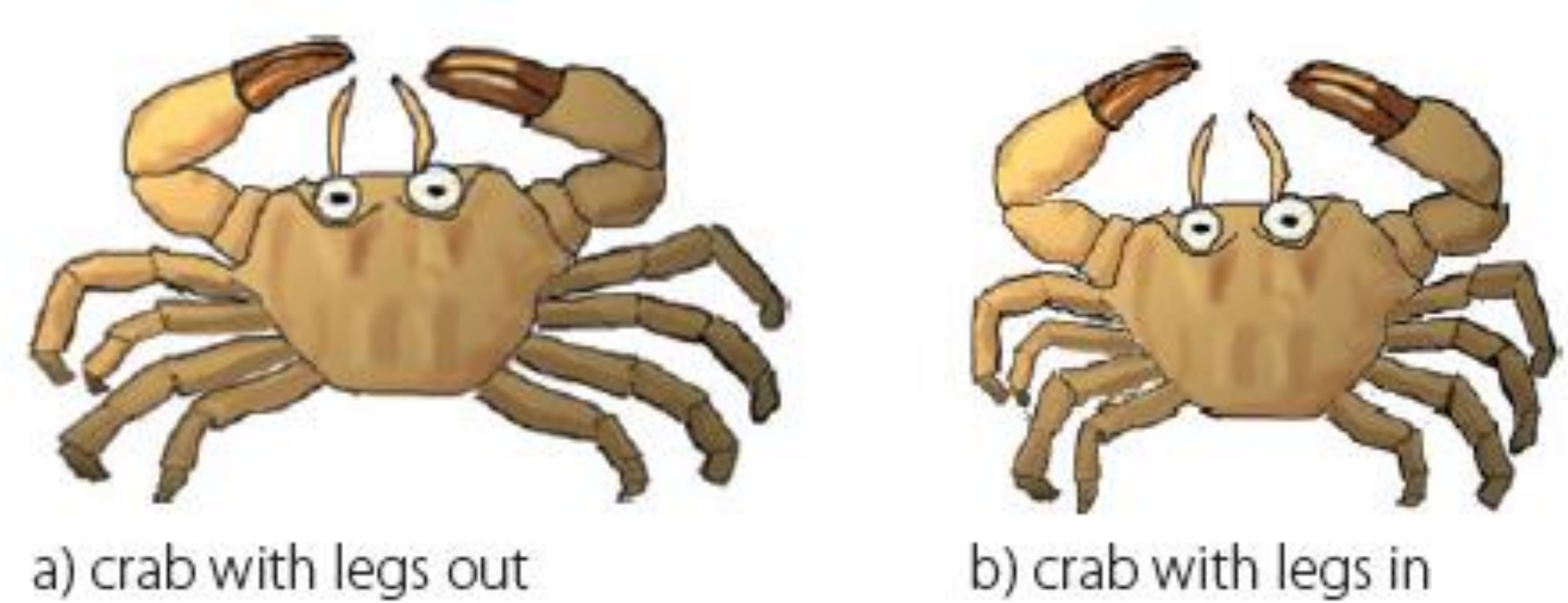


Figure 4.6 Two slightly different images of the crab

Animating Images

- The effect of this (switching back and forth between these images) will be that the crab looks as if it is moving its legs.
- In order to do this, we have to use some more variables and also discuss how to work with **Greenfoot** images.

Greenfoot Images

LECTURE 10

Greenfoot Images

- Greenfoot provides a class called **GreenfootImage** that helps in using and manipulating images. We can obtain an image by constructing a new **GreenfootImage** object—using Java's **new** keyword—with the file name of the image file as a parameter to the constructor. For example, to get access to the crab2.png image, we can write

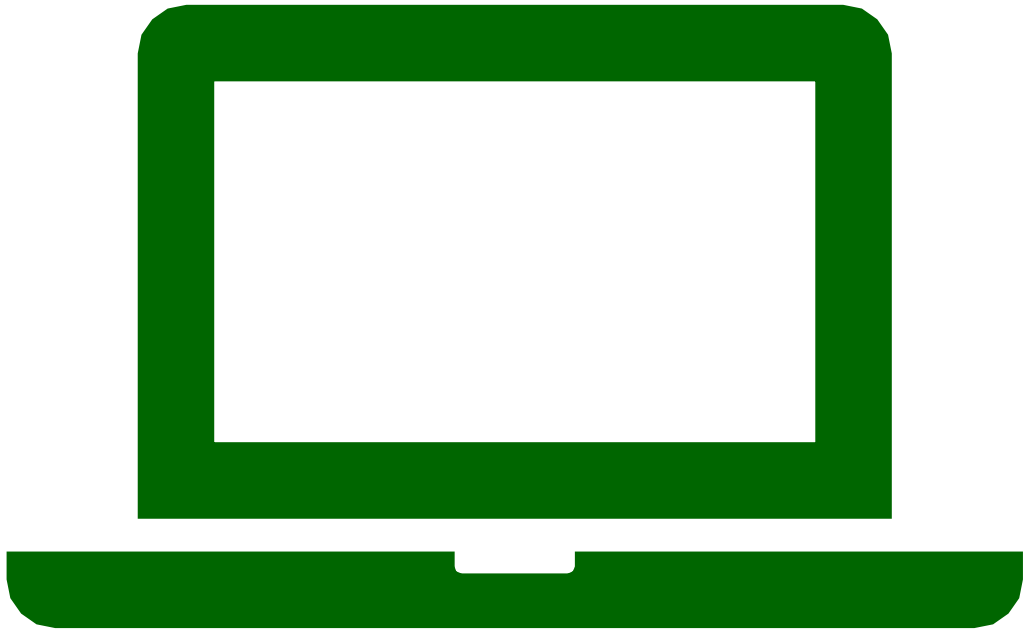
```
new GreenfootImage("crab2.png")
```

> **Concept**
>
> Greenfoot actors maintain their visible image by holding an object of type GreenfootImage.

The file we name here must exist in the scenario's images folder.

Greenfoot Images

- All Greenfoot actors have images. By default, actors get their image from their class. We assign an image to the class when we create it, and every object created from that class will receive, upon creation, a copy of that same image. That does not mean, however, that all objects of the same class must always keep the same image. Every individual actor can decide to change its image at any time.



Exercise 4.15

Exercise 4.15

Check the documentation of the Actor class. There are two methods that allow us to change an actor’s image. What are they called, and what are their parameters? What do they return?If you did the exercise above, you will have seen that one method to set an actor’s image expects a parameter of type GreenfootImage. This is the method we shall use. We can create a GreenfootImage object from an image file as described above and assign it to a variable of type GreenfootImage. Then we use the actor’s setImage method to use it for the actor. Here is a code snippet to do this:GreenfootImage image2 = new GreenfootImage(“crab2.png”);

setImage(image2);

Exercise 4.15

To set the image back to the original image, we write:GreenfootImage image1 = new GreenfootImage(“crab.png”);

setImage(image1);This creates the image objects from the named image files (crab.png and crab2.png) and assigns them to the image1 and image2 variables. Then we use these variables to set our new image as the actor’s image. To create the animation effect, we just have to set it up somehow so that these two code fragments are executed in alternating sequence: first one, then the other, back and forth.

Exercise 4.15

We could go ahead now and add code similar to this to our act method. However, before doing this, we shall discuss one improvement: we want to separate the creation of the image objects from the setting of the image.The reason is efficiency. When our program runs with the image animation, we will change the image many times, several times per second. With the code as we have written it, we would also read the image from the image file and create the image objects many times. This is not necessary, and it is wasteful. It is enough to create the image objects once and then just set them back and forth many times. In other words, we want to separate the code fragments like this:

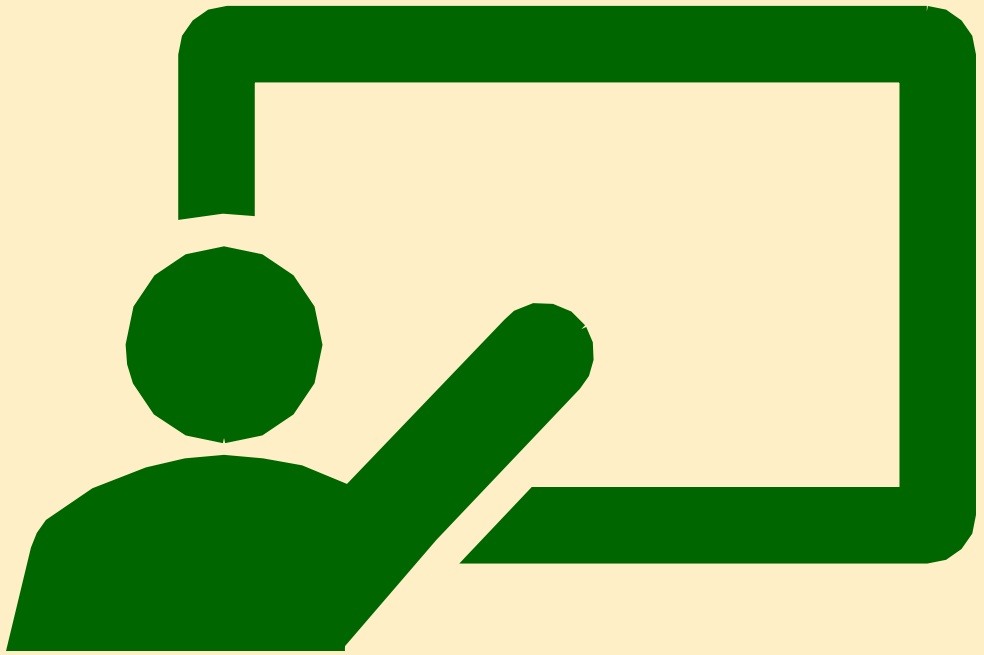
Exercise 4.15

Do this only once at the beginning: GreenfootImage image1 = new GreenfootImage(“crab.png”);

GreenfootImage image2 = new GreenfootImage(“crab2.png”);Do this many times over and over: setImage(image1);orsetImage(image2);

Exercise 4.15

Thus, we shall first create the images and store them, and later we shall use the stored images (without creating them again) over and over to alternate our displayed image.To achieve this, we need a new construct that we have not used before: an instance variable.



Instance Variables (fields)

LECTURE 11

Instance Variables (fields)

- Java provides different kinds of variables. The ones we have seen before are called **local variables**, and the ones we shall discuss now are **instance variables**. (Instance variables are also sometimes called fields.)
- The first difference is the place where they are declared in our source code (Code 4.2): local variables are declared inside a method, while instance variables are declared inside the class, but before any methods.

```
public class Crab extends Actor
{
    private GreenfootImage image1;
    private GreenfootImage image2;
    private int age;

    /**
     * Make out crab act.
     */
    public void act()
    {
        boolean isAlive;
        int n;

        // ... acting code omitted
    }
}
```

Instance variables

Local variables

Code 4.2 Instance variables and local variables in a class

Instance Variables (fields)

Concept

Instance variables (also called fields) are variables that belong to an object (rather than a method).

Concept

Lifetime of instance variables: Instance variables persist as long as the object exists that holds them.

- The next easily visible difference is that instance variables have the keyword **private** in front of them (see Code 4.2).
- More important, however, is the difference in behavior: local variables and instance variables behave differently, especially regarding their **lifetime**.

Instance Variables (fields)

- Local variables belong to the method they are declared in, and disappear as soon as the method finishes executing. Every time we call the method, the variables are created again, and can be used while the method executes, but they do not survive between method calls. Values or objects stored in them are lost when the method ends.

Note: To be exact: local variables belong to the scope they are declared in and exist only to the end of that scope. Often this is a method, but if the variable is declared, for example, inside an if-statement, it will disappear at the end of that if-statement.

Concept

Lifetime of local variables: Local variables persist only during a single method execution.

Instance Variables (fields)

- Instance variables, on the other hand, belong to the object they are declared in, and survive as long as the objects exist. They can be used over and over again, over multiple method calls and by multiple methods. Thus, if we want an object to store information for a longer time, an instance variable is what we need.Instance variables are defined at the top of the class, following the class header, using the keyword private followed by the type of the variable and the variable name:

Note: Java does not enforce instance variables being at the top of the class, but we will always do this as it is good practice and helps us find the variable declarations easily when we need to see them.

```
private variable-type variable-name;
```

- In our case, since we want to store objects of type GreenfootImage, the variable type is GreenfootImage and we use the names image1 and image2 as in our code snippets before (Code 4.3).

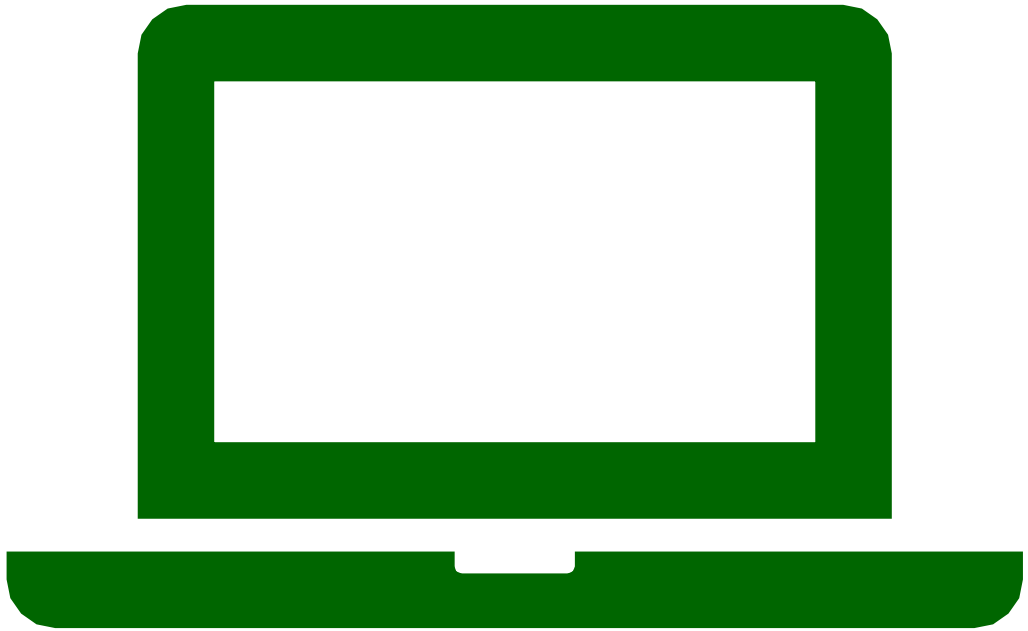
```
import greenfoot.*;  // (World, Actor, GreenfootImage, and Greenfoot)

 // comment omitted

public class Crab extends Actor
{
    private GreenfootImage image1;
    private GreenfootImage image2;

    // methods omitted
}
```

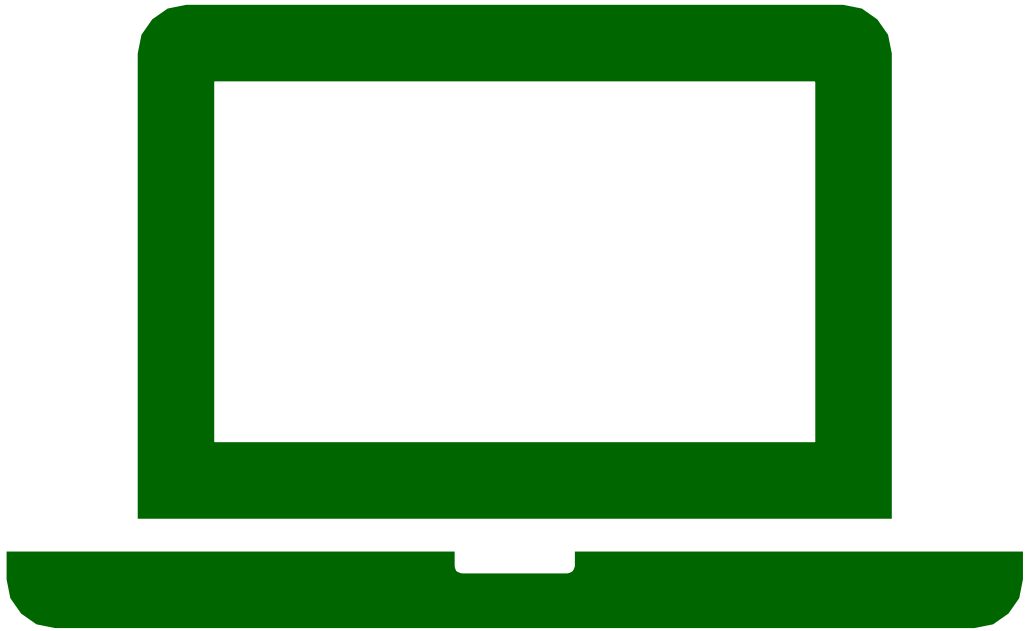
Code 4.3 The Crab class with two instance variables



Exercise 4.16

Exercise 4.16

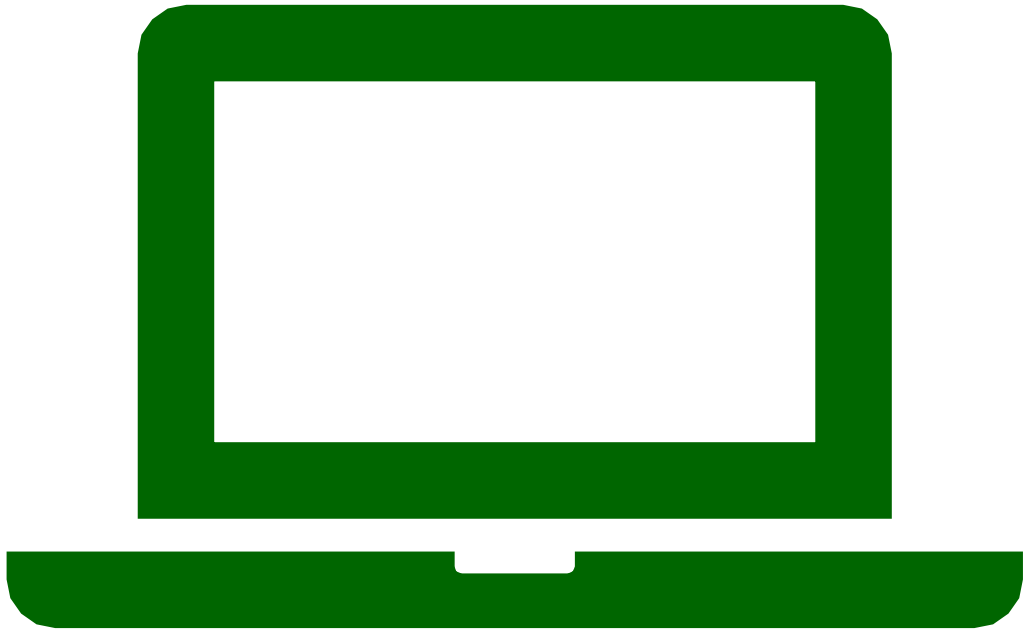
- Before adding this code, right-click a crab object in your world and select **Inspect** from the crab's pop-up menu. Make a note of all the variables that are shown in the crab object.



Exercise 4.17

Exercise 4.17

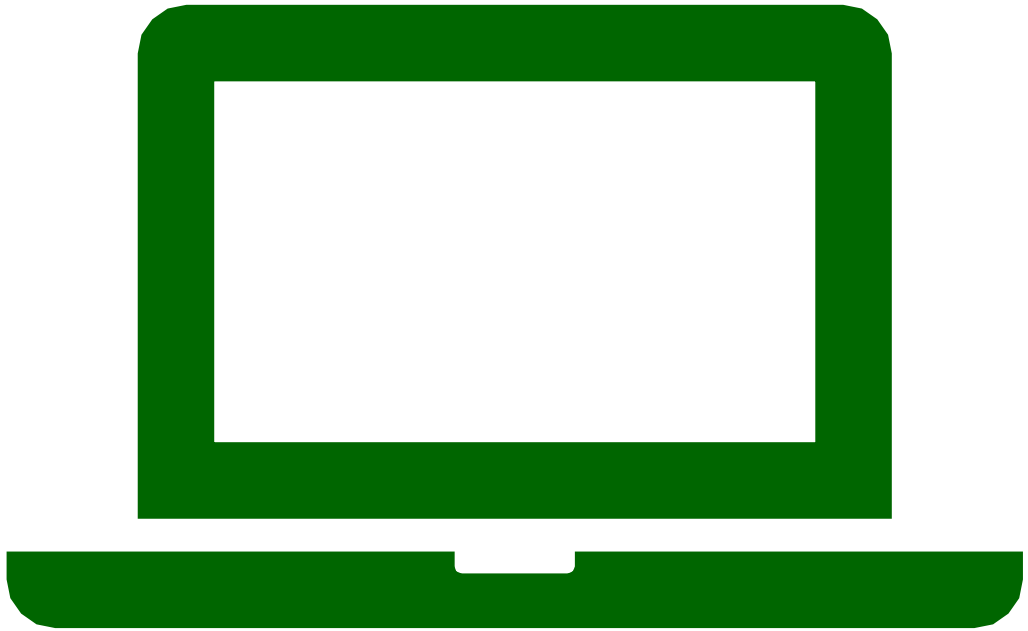
- Why do you think the crab has any variables at all, even though we have not declared any in our crab class?



Exercise 4.18

Exercise 4.18

- Add the variable declarations shown in Code 4.3 above to your version of the **Crab** class. Make sure that the class compiles.



Exercise 4.19

Exercise 4.19

- After adding the variables, inspect your crab object again. Take a note of the variables and their values (shown in the white boxes).
- In our diagrams, we show objects as colored boxes with rounded corners, and instance variables as white boxes inside an object (Figure 4.7). Note that the declaration of these two **GreenfootImage** variables does not give us two **GreenfootImage** objects. It just gives us some empty space to store two objects.

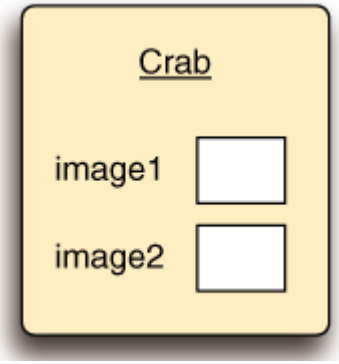


Figure 4.7 A crab object with two empty instance variables

Exercise 4.19

- Next we have to create the two image objects and store them into the instance variables. The statement for the creation of the objects has already been shown above. It was achieved with the code snippet

```
new GreenfootImage("crab2.png")
```

- Now we just need to create both image objects and assign them to our instance variables:

```
image1 = new GreenfootImage("crab.png");
image2 = new GreenfootImage("crab2.png");
```

- Following these statements, we have three objects (one crab and two images), and the crab's variables contain references to the images. This is shown in Figure 4.8.

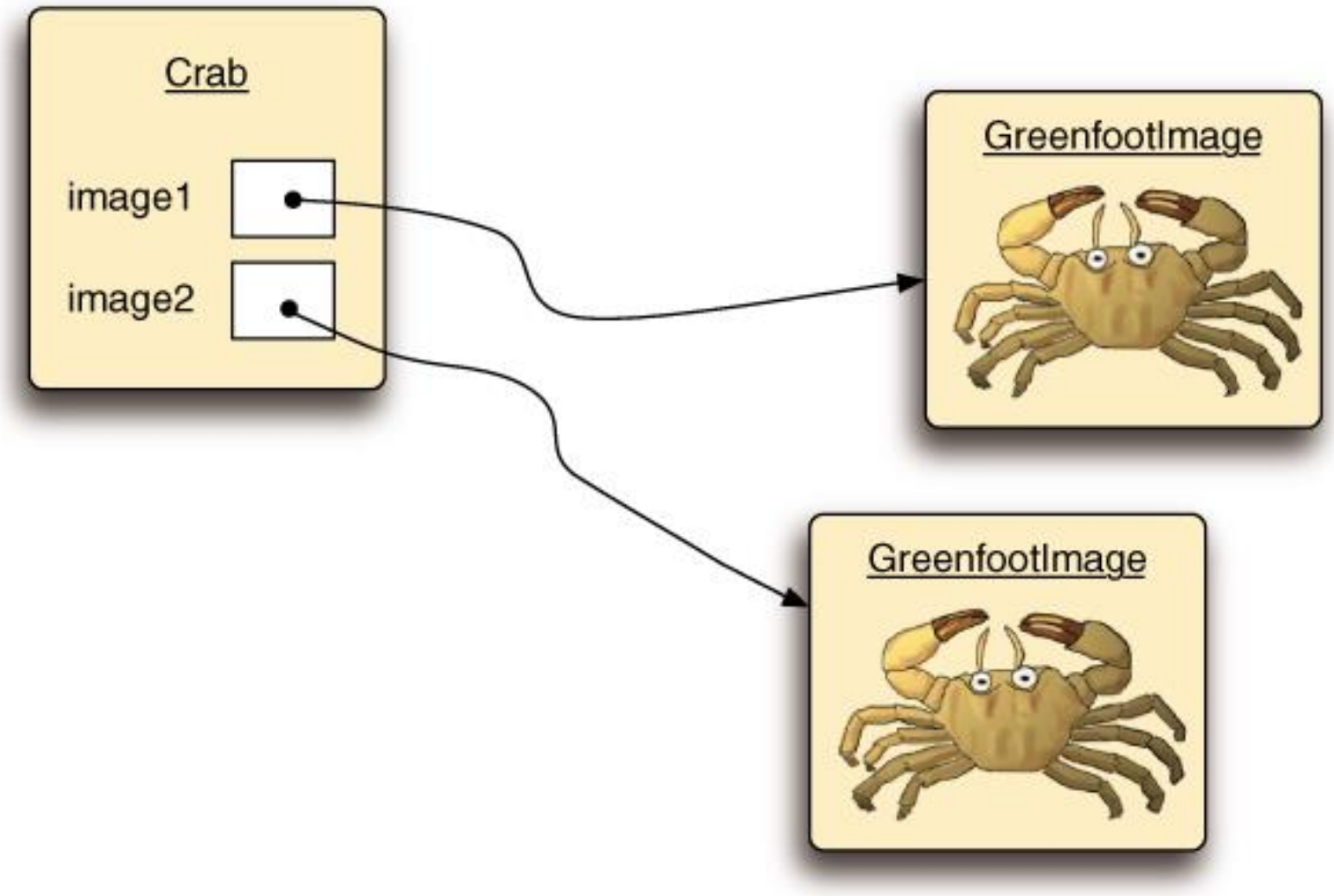
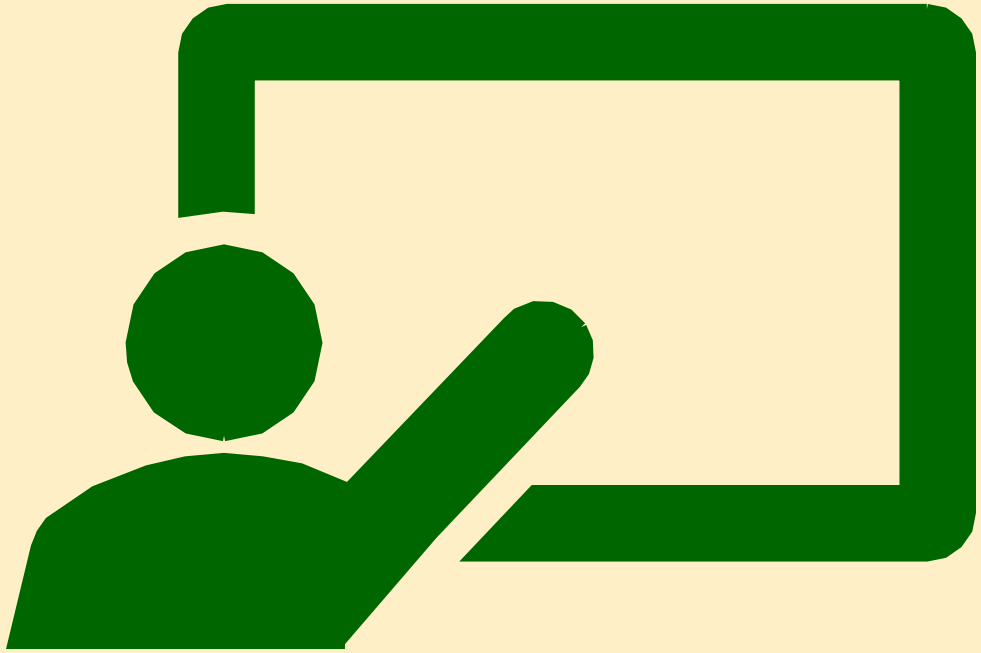


Figure 4.8 A crab object with two variables, pointing to image objects

Exercise 4.19

- The last remaining question is where to put the code that creates the images and stores them into the variables. Since this should be done only once when the crab object is created, and not every time we act, we cannot put it into the **act** method. Instead, we put this code into a constructor.



Using Actor Constructors

LECTURE 12

Using Actor Constructors

- At the beginning of this chapter we have seen how to use the constructor of the world class to initialize the world. In a similar manner, we can use a constructor of an actor class to initialize the actor. The code in the constructor is executed once when the actor is created. Code 4.4 shows a constructor for the Crab class that initializes the two instance variables by creating images and assigning them to the variables.

```
import greenfoot.*;  // (Actor, World, Greenfoot, GreenfootImage)

// comment omitted

public class Crab extends Actor
{
    private GreenfootImage image1;
    private GreenfootImage image2;

    /**
     * Create a crab and initialize its two images.
     */
    public Crab()
    {
        image1 = new GreenfootImage("crab.png");
        image2 = new GreenfootImage("crab2.png");
        setImage(image1);
    }

    // methods omitted
}
```

Code 4.4 Initializing the variables in the constructor

Using Actor Constructors

- The same rules described for the **World** constructor apply to the **Crab** constructor:
  - The signature of a constructor does not include a return type.
  - The name of the constructor is the same as the name of the class.
  - The constructor is automatically executed when a crab object is created.
- The last rule—that the constructor is automatically executed—ensures that the image objects are automatically created and assigned when we create a crab. Thus, after creating the crab, the situation will be as depicted in Figure 4.8.

Pitfall

- Note carefully that there is no type before the variable name in the assignment in the constructor. The variable is defined before the constructor using the statement

```
private GreenfootImage image1;
```

and assigned in the constructor using the line

```
image1 = new GreenfootImage("crab.png");
```

- If instead we write in the constructor

```
GreenfootImage image1 = new GreenfootImage("crab.png");
```

Pitfall

then something entirely different happens: we would declare an additional local variable called image1 in the constructor (we then have two variables called **image1**: one local, one instance) and assign our image to the local one. It would then be lost as soon as the constructor ends, and our instance variable is still empty.

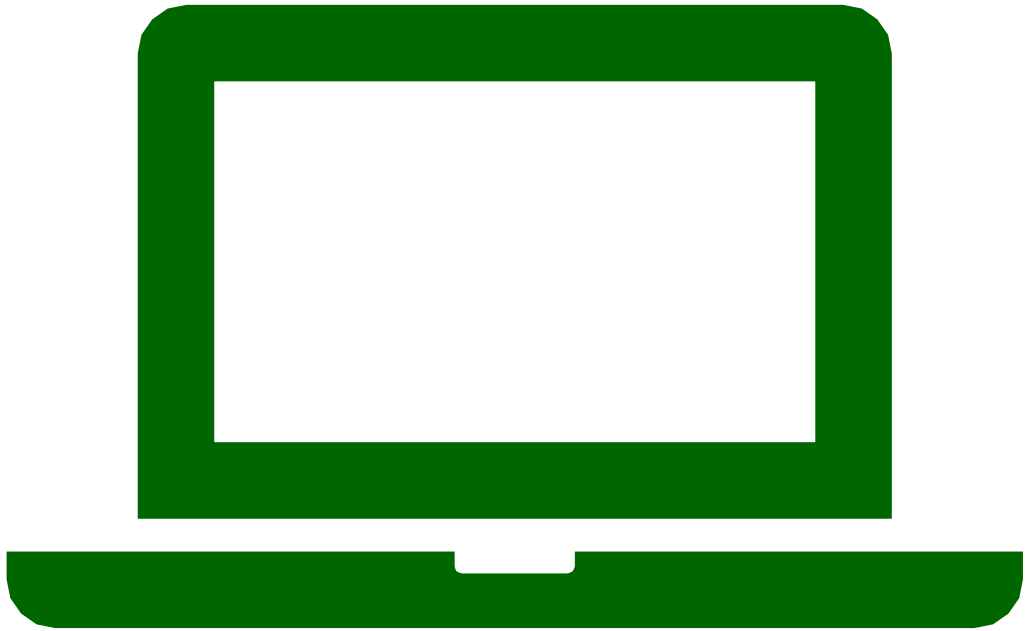
This is a very subtle error, easy to make and difficult to find. So make sure you have your variable declaration at the top, and only an assignment without the declaration in the constructor.

Using Actor Constructors

- The last line of the constructor sets the first of the two created images as the crab's current image:

```
setImage(image1);
```

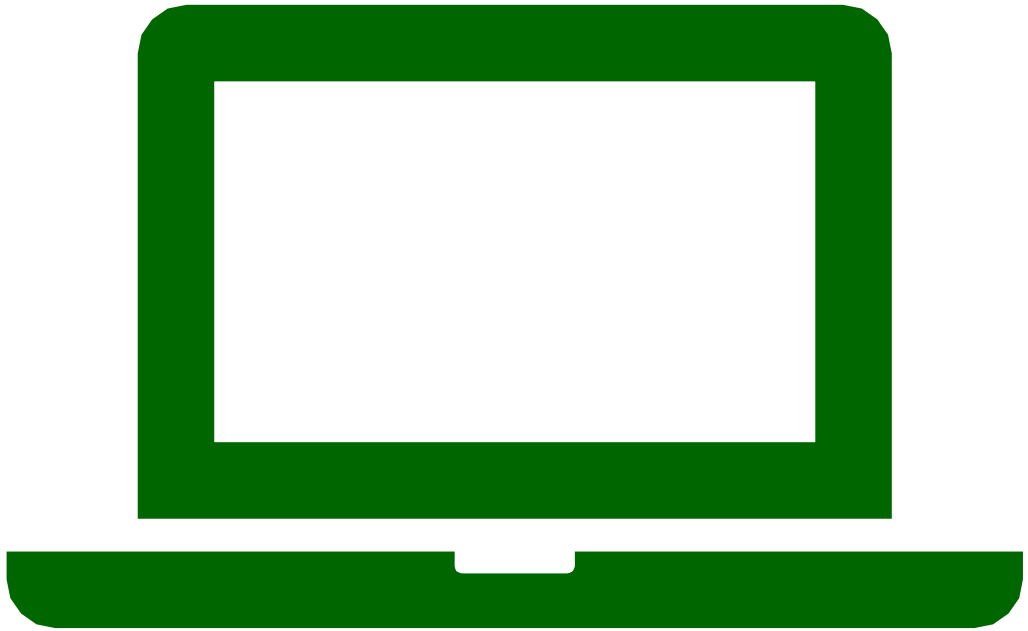
- We can later use a similar method call to swap the images in the act method.



Exercise 4.20

Exercise 4.20

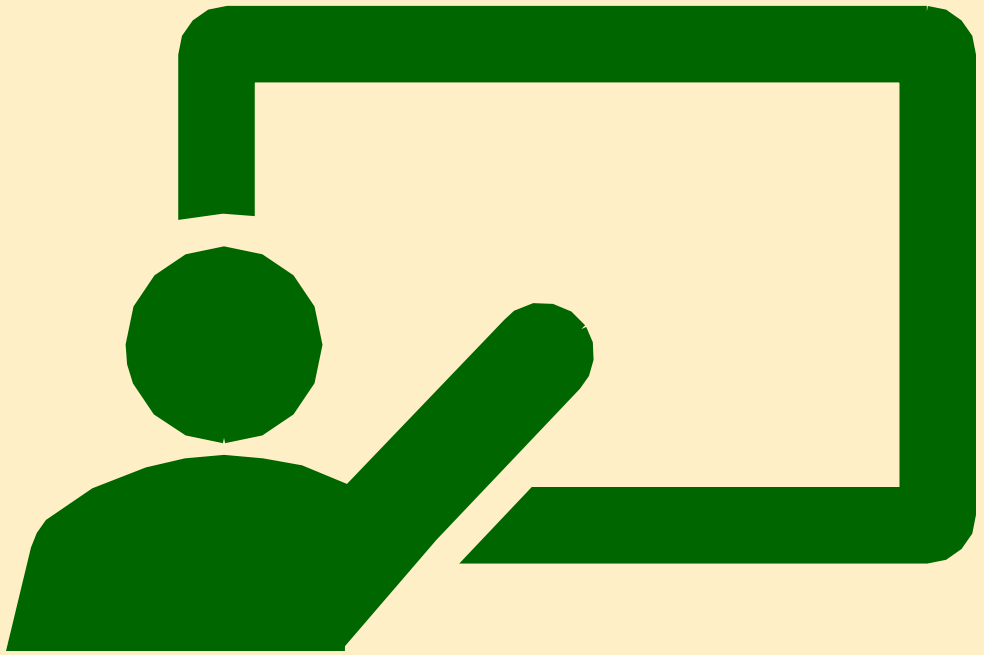
- Add this constructor to your **Crab** class. You will not yet see any change in the behavior of the crab, but the class should compile, and you should be able to create crabs.



Exercise 4.21

Exercise 4.21

- Inspect your crab object again. Take a note again of the variables and their values. Compare those to the notes you took previously.



Alternating the Images

LECTURE 13

Alternating the Images

- We have now reached a stage where the crab has two images available to do the animation. But we have not done the animation itself yet. This is now relatively simple.To do the animation, we need to alternate between our two images. In other words, at every step, if we are currently showing **image1**, we now want to show **image2**, and vice versa. Here is some pseudo-code to express this:

```
if (our current image is image1) then
    use image2 now
else
    use image1 now
```

- Pseudo-code, as used here, is a technique expressing a task in a structure that is partly like real Java code, and partly plain English. It often helps in working out how to write our real code. We can now show the same in real Java code (Code 4.5).

```
if ( getImage() == image1 )
{
    setImage(image2);
}
else
{
    setImage(image1);
}
```

Code 4.5 Alternating between two images

Alternating the Images

In this code segment, we notice several new elements:

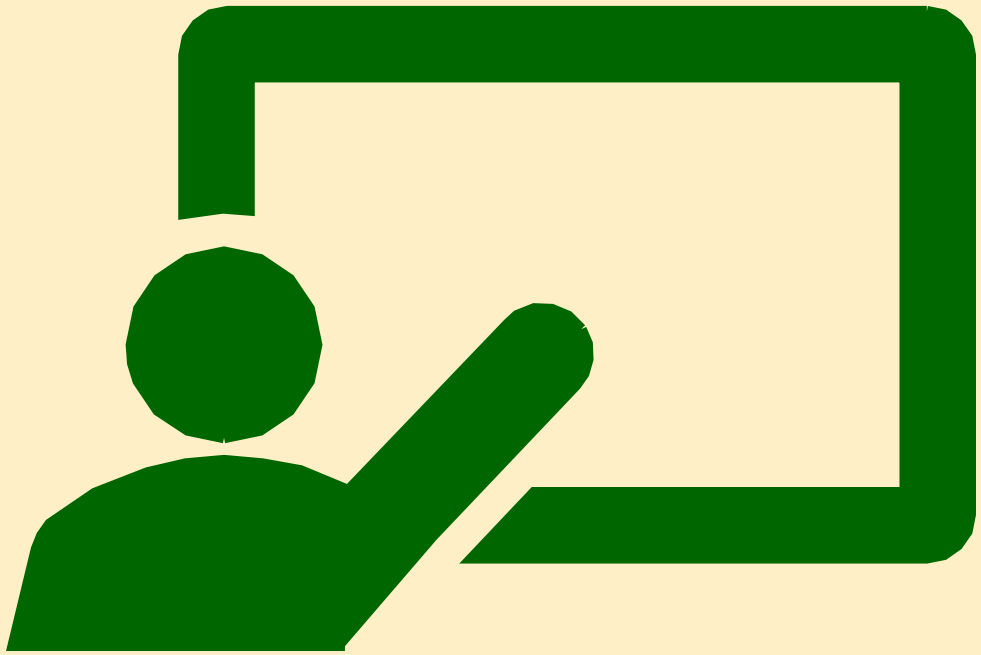
Concept

We can test whether two things are equal by using a double equals symbol: ==.

- The method **getImage** can be used to receive the actor's current image.
- The operator == (two equal signs) can be used to compare one value with another. The result is either true or false.
- The if-statement has an extended form that we have not seen before. This form has an **else** keyword after the first body of the if-statement, followed by another block of statements. We investigate this new form of the if-statement in the next section.

Pitfall

- It is a common mistake to get the assignment operator (=) and the operator to check equality (==) mixed up. If you want to check whether two values or variables are equal, you must write two equal symbols.



The if/else Statement

LECTURE 14

The if/else Statement

- Before moving on, let us investigate the if-statement again in some more detail. As we have just seen, an if-statement can be written in the form

```
if ( condition ){
    statements;
}
else{
    statements;
}
```

The if/else Statement

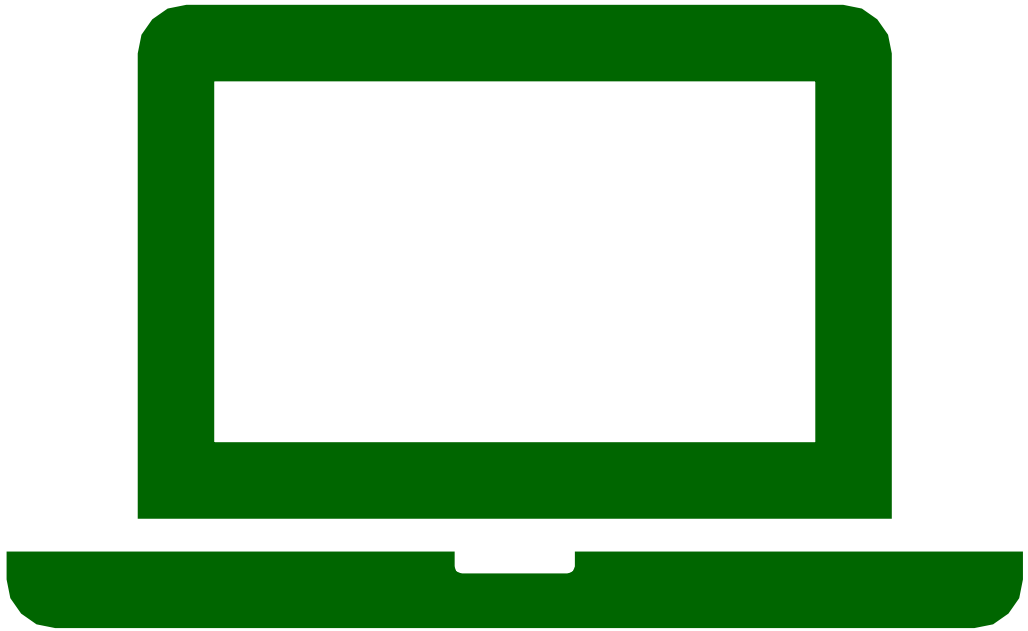
Concept

The if/else statement executes a segment of code when a given condition is true, and a different segment of code when it is false.

- This if-statement contains two blocks (pairs of curly brackets surrounding a list of statements): the if-clause and the else-clause (in this order).When this if-statement is executed, first the condition is evaluated. If the condition is true, the if-clause is executed, and then execution continues below the else-clause. If the condition is false, the if-clause is not executed; instead we execute the else-clause. Thus, one of the two statement blocks is always executed, but never both.

The if/else Statement

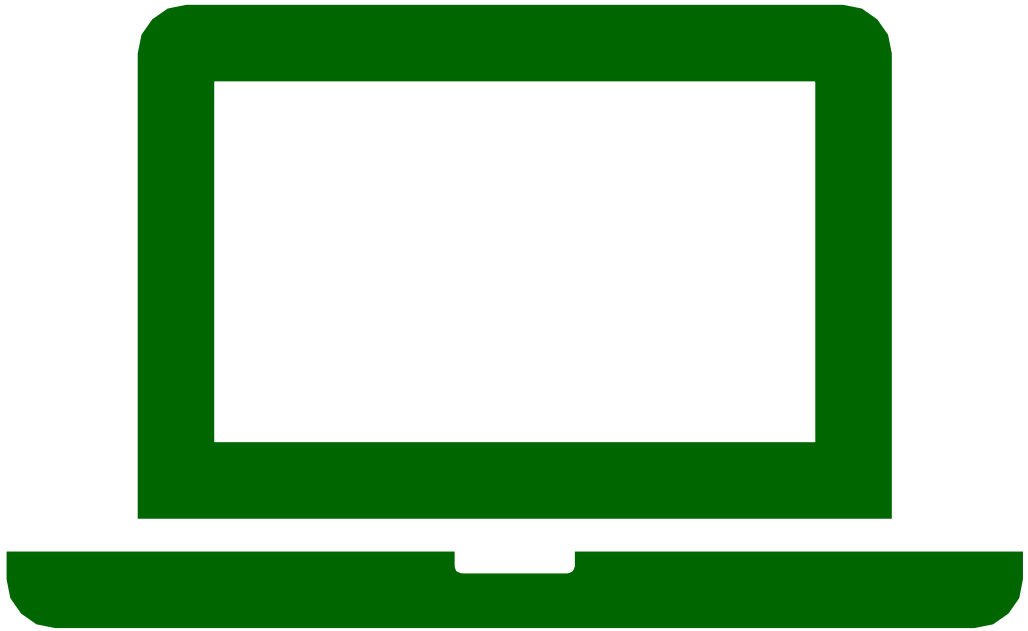
- The else part with the second block is optional—leaving it off leads to the shorter version of the if-statement we have seen earlier.
- We have now seen everything we need to finalize this task. It is time to get our hands on the keyboard again to try it out.



Exercise 4.22

Exercise 4.22

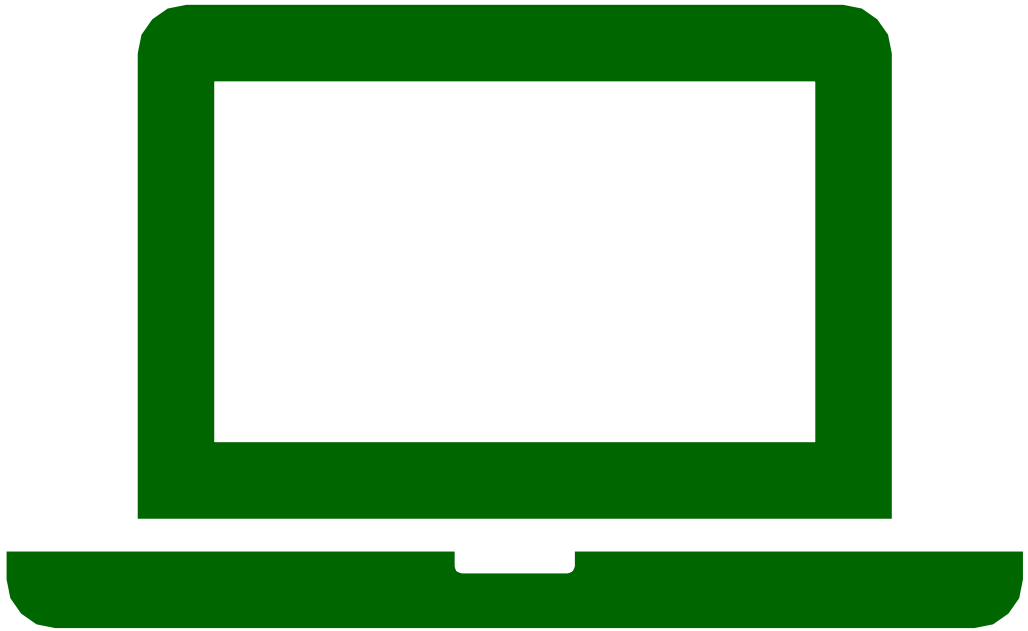
- Add the image switching code, as shown in Code 4.5, to the **act** method of your own **Crab** class. Try it out! (If you get an error, fix it. This should work.) Also try clicking the **Act** button instead of the **Run** button in Greenfoot—this allows us to observe the behavior more clearly.



Exercise 4.23

Exercise 4.23

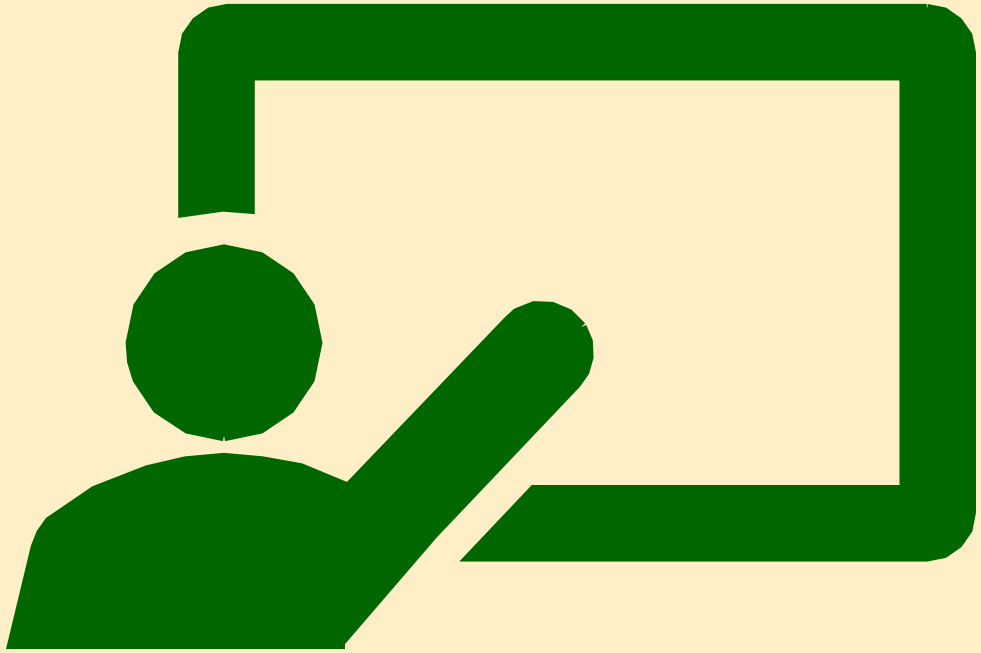
- In Chapter 3, we discussed using separate methods for subtasks, rather than writing more code directly into the **act** method. Do this with the image switching code: Create a new method called **switchImage**, move your image switching code to it, and call this method from within your **act** method.



Exercise 4.24

Exercise 4.24

- Call the **switchImage** method interactively from the crab's pop-up menu. Does it work?



Counting Worms

LECTURE 15

Counting Worms

- The final thing we want to achieve is to add functionality so that the crab counts how many worms it has eaten. If it has eaten eight worms, we win the game. We also want to play a short “winning sound” when this happens.
- To make this happen, we will need a number of additions to our crab code. We need
  - an instance variable to store the current count of worms eaten;
  - an assignment that initializes this variable to zero at the beginning;
  - code to increment our count each time we eat a worm; and
  - code that checks whether we have eaten eight worms, and stops the game and plays the sound if we have.

Counting Worms

- Let us do the tasks in the order in which we have listed them here.
- We can define a new instance variable by following the pattern introduced above. Below our two existing instance variable definitions, we add the line

```
private int wormsEaten;
```

- Here the type int indicates that we want to store integers (whole numbers), and the name wormsEaten indicates what we intend to use it for.
- Next we add the following line to the end of our constructor:

```
wormsEaten = 0;
```

Counting Worms

- This initializes the **wormsEaten** variable to zero when the crab is created. Strictly speaking, this is redundant, since instance variables of type **int** are initialized to zero automatically. However, sometimes we want the initial value to be something other than zero, so writing our own initialization statement is good practice.
- The last bit is to count the worms and check whether we have reached eight. We need to do this every time we eat a worm, so we find our **lookForWorm** method, where we have our code that does the eating of the worms. Here, we add a line of code to increment the worm count:

```
wormsEaten = wormsEaten + 1;
```

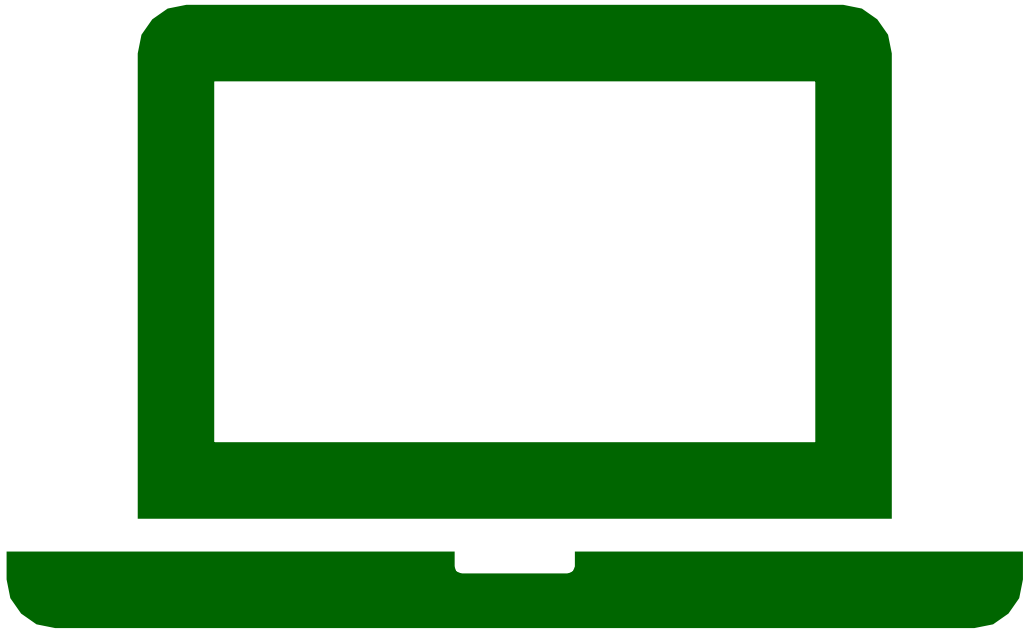
Counting Worms

- As always in an assignment, the right hand side of the assignment symbol is evaluated first (**wormsEaten + 1**). Thus, we read the current value of **wormsEaten** and add **1** to it. Then we assign the result back to the **wormsEaten** variable. As a result, the variable will be incremented by 1.
- Following this, we need an if-statement that checks whether we have eaten eight worms yet, and plays the sound and stops execution if we have. Code 4.6 shows the complete **lookForWorm** method with this code. The sound file used here (fanfare.wav) is included in the sounds folder in your scenario, so it can just be played.

```
/**
 * Check whether we have stumbled upon a worm.
 * If we have, eat it. If not, do nothing. If we have
 * eaten eight worms, we win.
 */
public void lookForWorm()
{
    if ( isTouching(Worm.class) )
    {
        removeTouching(Worm.class);
        Greenfoot.playSound("slurp.wav");

        wormsEaten = wormsEaten + 1;
        if (wormsEaten == 8)
        {
            Greenfoot.playSound("fanfare.wav");
            Greenfoot.stop();
        }
    }
}
```

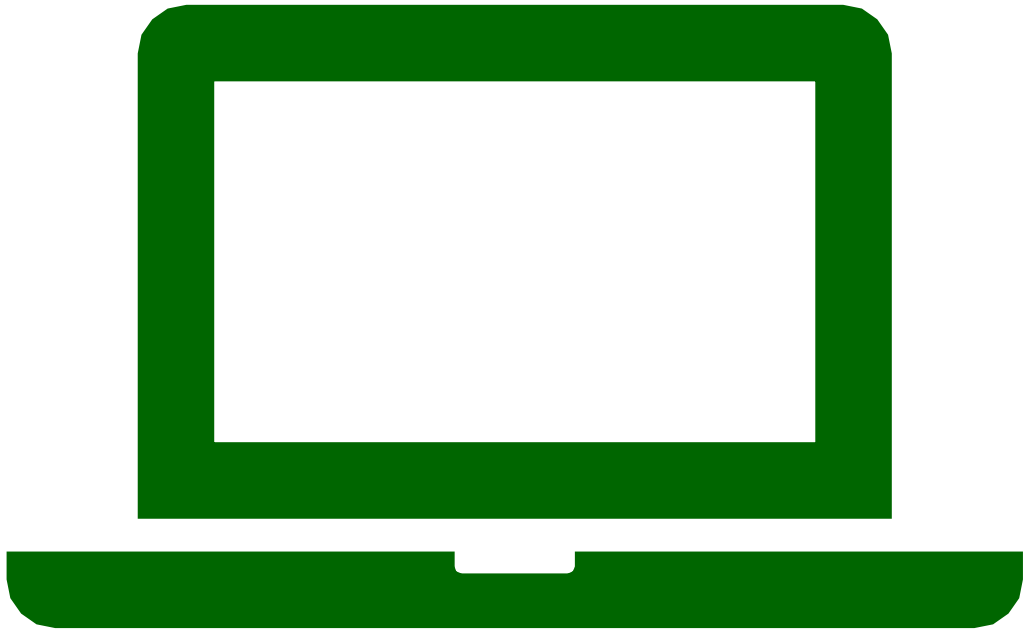
Code 4.6 Counting worms and checking whether we win



Exercise 4.25

Exercise 4.25

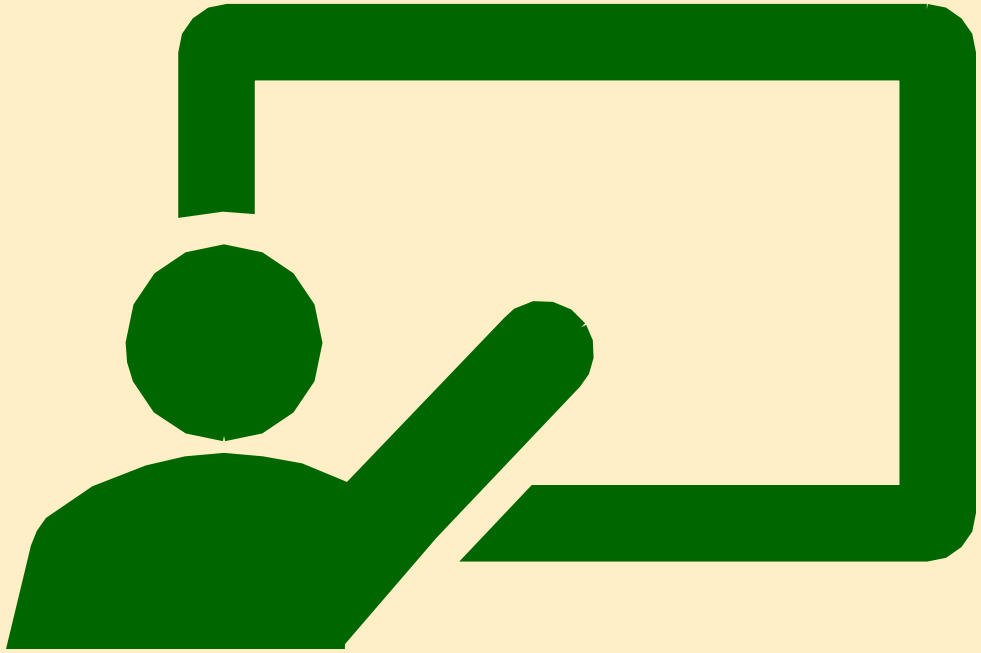
- Add the code discussed above into your own scenario. Test it, and make sure that it works.



Exercise 4.26

Exercise 4.26

- As a further test, open an object inspector for your crab object (by selecting **Inspect** from the crab's pop-up menu) before you start playing the game. Leave the inspector open and keep an eye on the **wormsEaten** variable while you play.



More Ideas

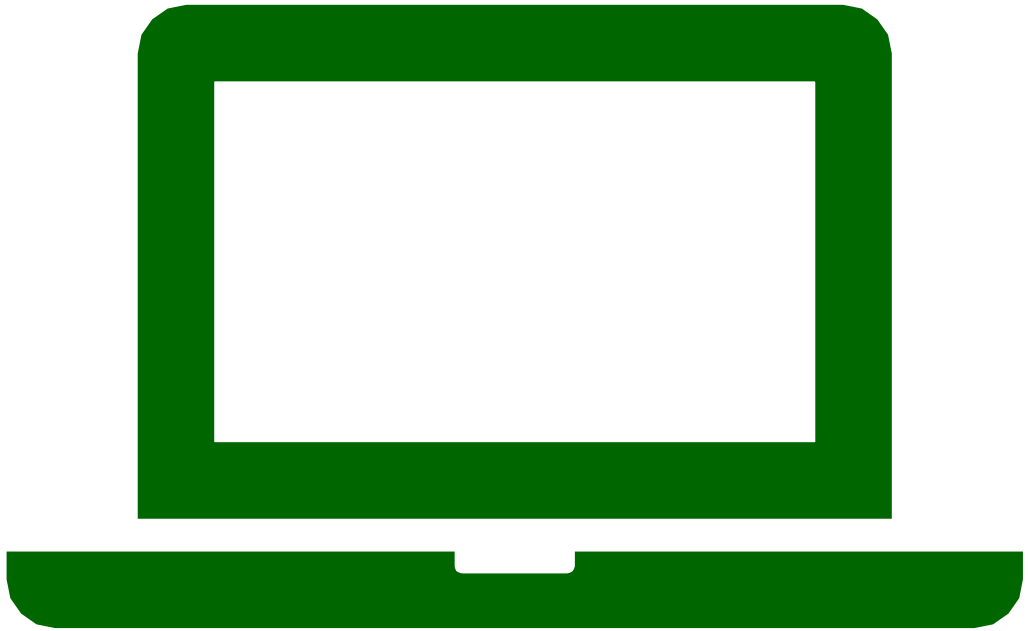
LECTURE 16

More Ideas

- The scenario little-crab-5, in the book scenarios folder, shows a version of the project that includes all the extensions discussed here.
- We will leave this scenario behind now and move on to a different example, although there are many obvious things (and probably many more less obvious things) you can do with this project. Ideas include

More Ideas

- using different images for the background and the actors;
- using different kinds of actors;
- not moving forward automatically, but only when the up-arrow key is pressed;
- building a two-player game by introducing a second keyboard-controlled class that listens to different keys;
- making new worms pop up when one is eaten (or at random times); and
- many more that you can come up with yourselves.



Exercise 4.27

Exercise 4.27

- The crab image changes fairly quickly while the crab runs, which makes our crab look a little hyperactive. Maybe it would look nicer if the crab image changed only on every second or third act cycle. Try to implement this. To do this, you could add a counter that is incremented in the act method. Every time it reaches two (or three), the image changes, and the counter is reset to zero.

Summary of Programming Techniques

LECTURE 17

Summary of Programming Techniques

- In this chapter, we have seen a number of important new programming concepts. We have seen how constructors can be used to initialize objects—constructors are always executed when a new object is created.
- We have seen instance variables and local variables. Instance variables, also called fields, are used—together with assignment statements—to store information in objects, which can be accessed later. Local variables are used to store information for a short period of time—within a single method execution—and are discarded at the method end.

Summary of Programming Techniques

- We have used the **new** statement to programmatically create new objects, and finally, we have seen the full version of the if-statement, which includes an else part that is executed when the condition is not true.
- With all these techniques together we can now write quite a good amount of code already.

Concept Summary

- A **constructor** of a class is a special kind of method that is executed automatically whenever a new instance is created.
- Java objects can be created programmatically (from within your code) by using the **new** keyword.
- **Variables** can be used to store information (objects or values) for later use.
- Variables can be created by writing a **variable declaration**.
- We can store values into variables by using an assignment **statement (=)**.
- Variables of **primitive types** store numbers, booleans, and characters; variables of **object types** store objects.

Concept Summary

- Objects are stored in variables by storing a **reference** to the object.
- Greenfoot actors maintain their visible image by holding an object of type **GreenfootImage**. These are stored in an instance variable inherited from class Actor.
- **Instance variables** (also called **fields**) are variables that belong to an object (rather than a method).
- **Lifetime of instance variables**: Instance variables persist as long as the object exists that holds them.
- **Lifetime of local variables**: Local variables persist only during a single method execution.

Concept Summary

- We can test whether two things are **equal** by using a double equals symbol: ==.
- The **if/else statement** executes a segment of code when a given condition is true, and a different segment of code when it is false.

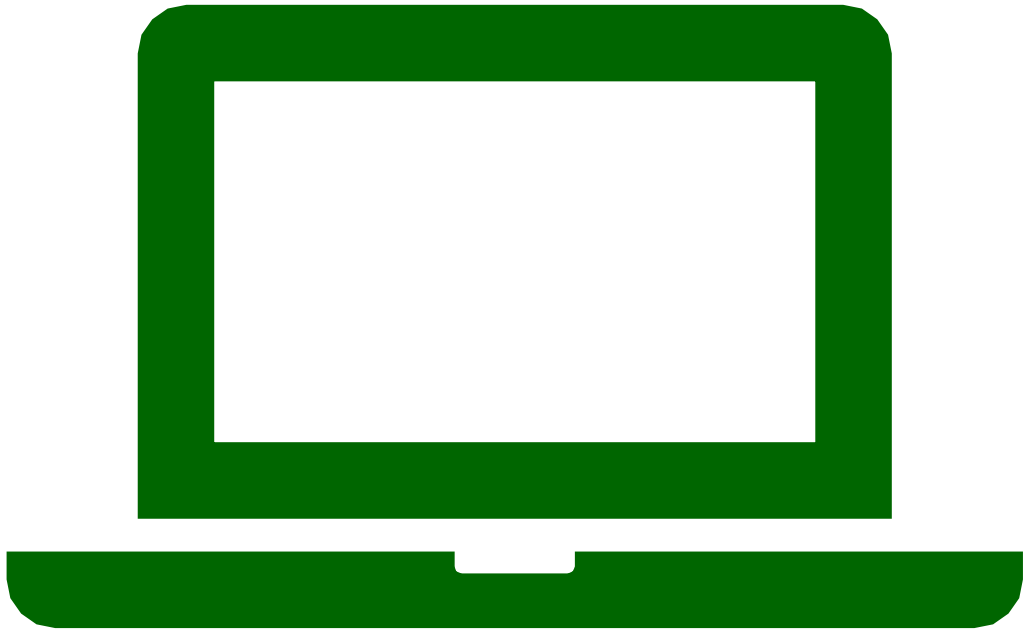
Drill and Practice

LECTURE 18

Drill and Practice

- This time, we do some more exercises with calling methods, including a new inherited Actor method, and practice more use of variables.

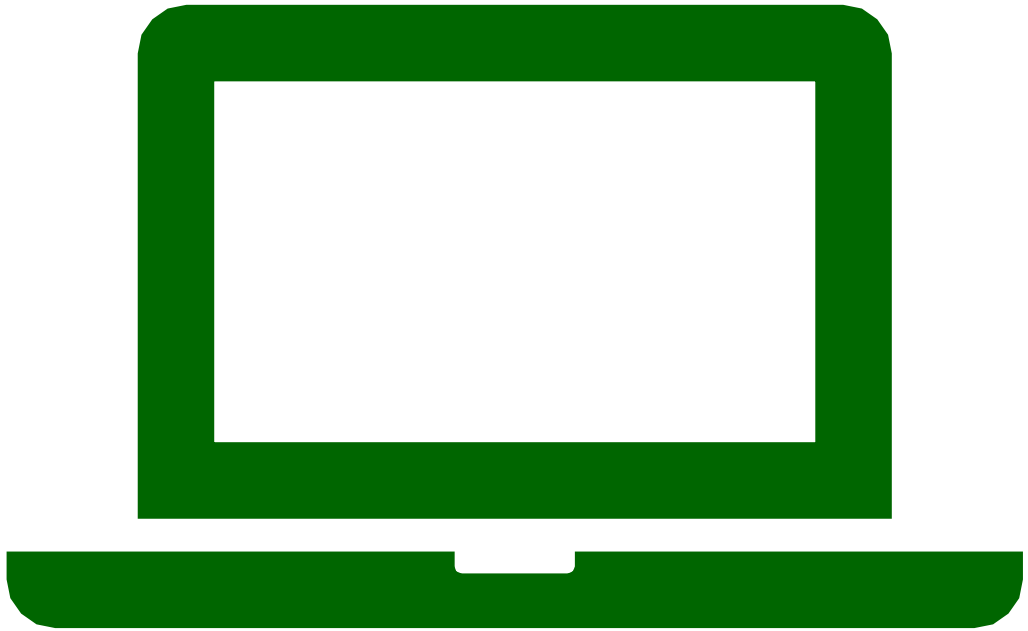
More Crab Work



Exercise 4.28

Exercise 4.28

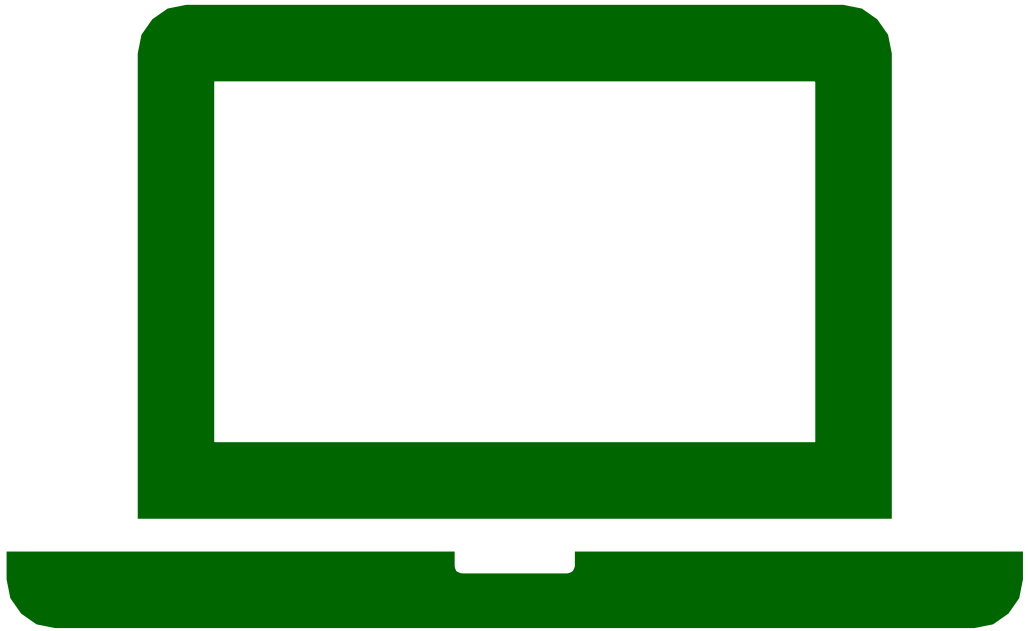
- Make the lobsters a bit more dangerous. The **Actor** class has a method called **turnTowards**. Use this method to make the lobsters turn toward the center of the screen, occasionally. Experiment with the frequency of doing this, and also with different walking speeds for lobsters and the crab.



Exercise 4.29

Exercise 4.29

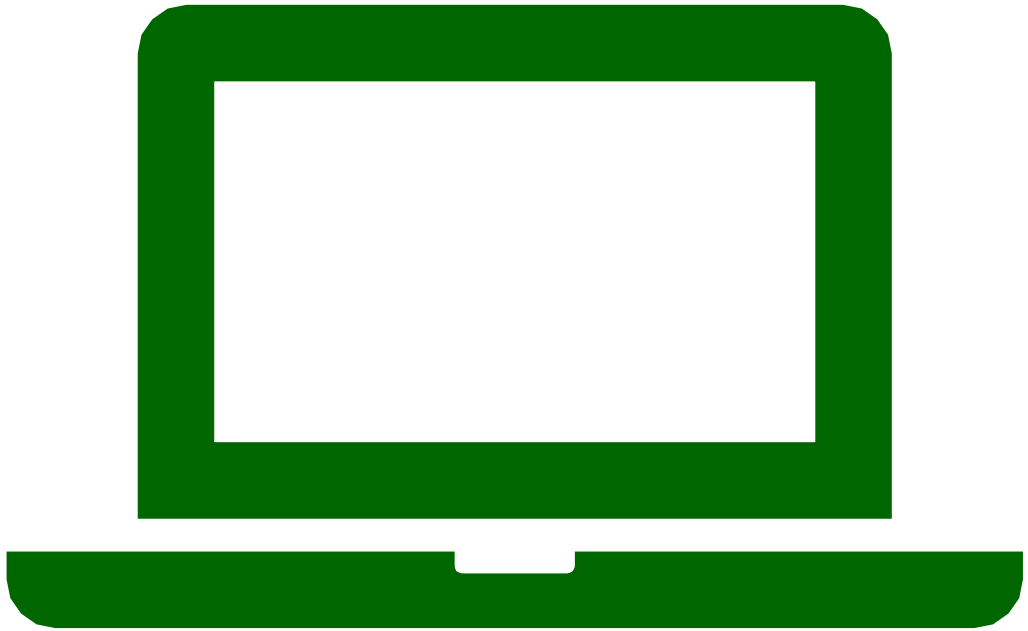
- Add a time counter to the crab. You can do this by adding an **int** variable that is incremented each time the crab acts. (You are, in effect, counting act cycles.) Should this be a local variable or an instance variable? Why?



Exercise 4.30

Exercise 4.30

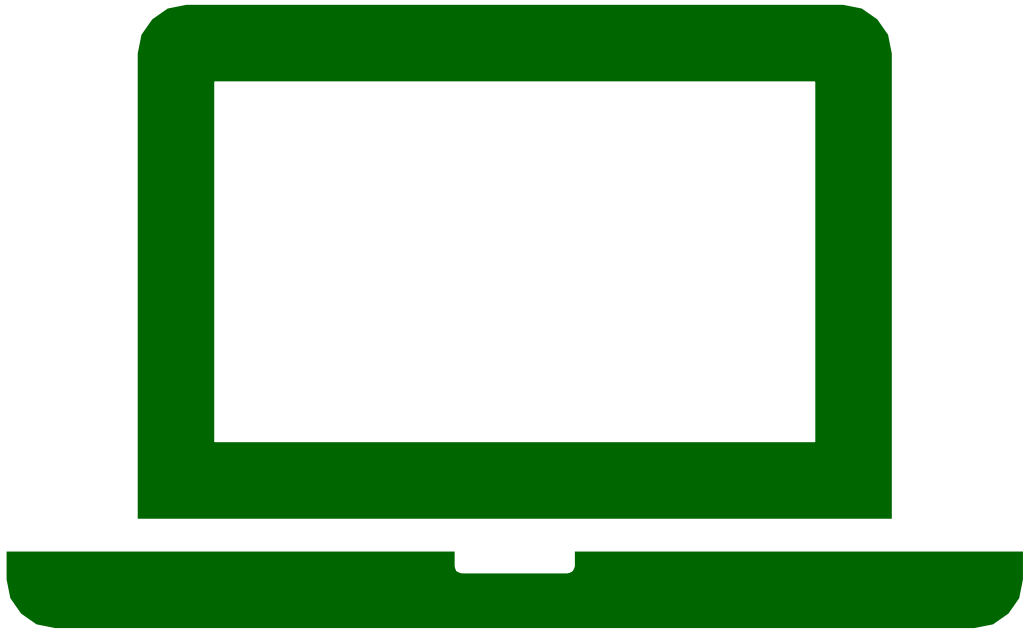
- Play your game. Once you manage to win (eat eight worms), inspect the crab object and check how long you took. How many act cycles did it take?



Exercise 4.31

Exercise 4.31

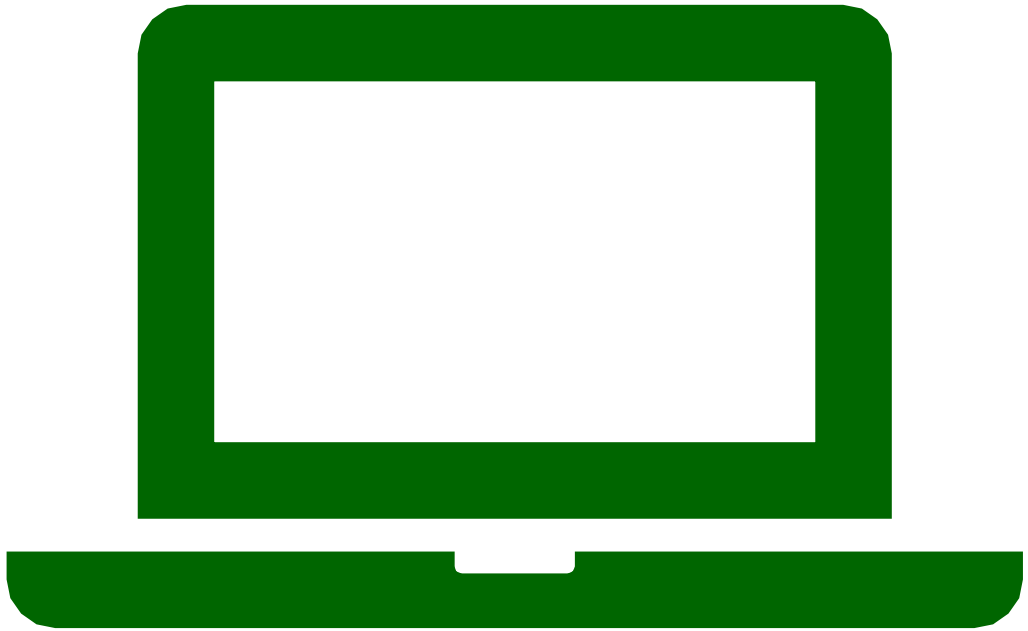
- Move your time counter from the **Crab** class to the **CrabWorld** class. (It makes more sense for the world to manage time, than an individual crab.) The variable is easy to move. To move the statement that increments the time, you need to define an **act** method in the **CrabWorld** class. World subclasses can have **act** methods just like **Actor** subclasses. Just copy the signature of the crab's **act** method to create a new act method in **CrabWorld** and place your time counting statement here.



Exercise 4.32

Exercise 4.32

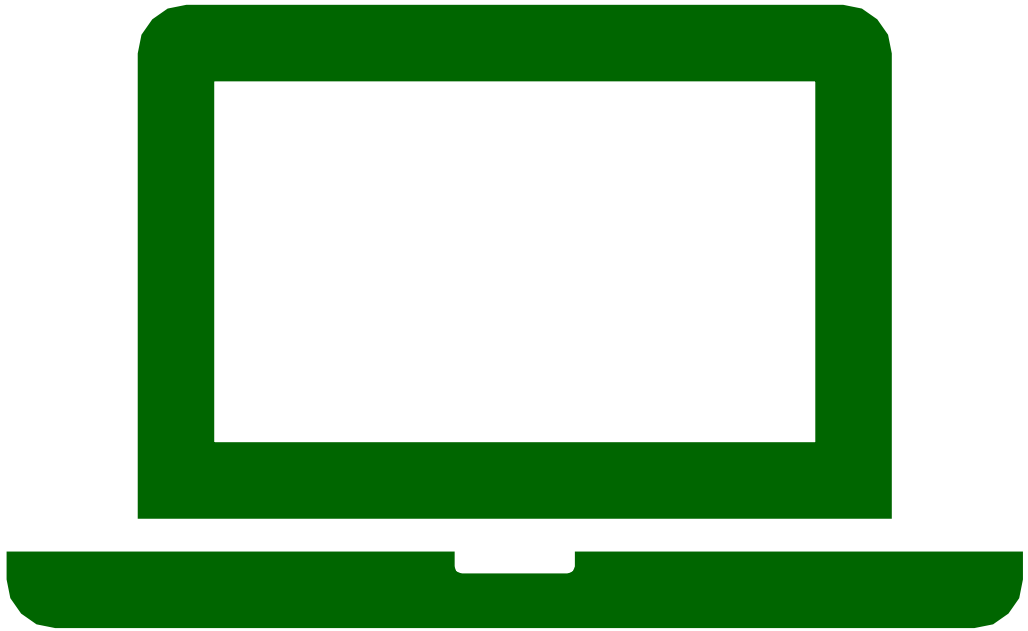
- Modify your game's time counter to be a game timer. That is: Initialize the time variable to some value (for example, 500), and count down (decrement the variable by one) at every act step. If the timer reaches zero, make the game end with a “time is up” sound. Experiment with different values for the game time.



Exercise 4.33

Exercise 4.33

- Investigate the **showText** method of the World class. How many parameters does it have? What are they? What does it return? What does it do?



Exercise 4.34

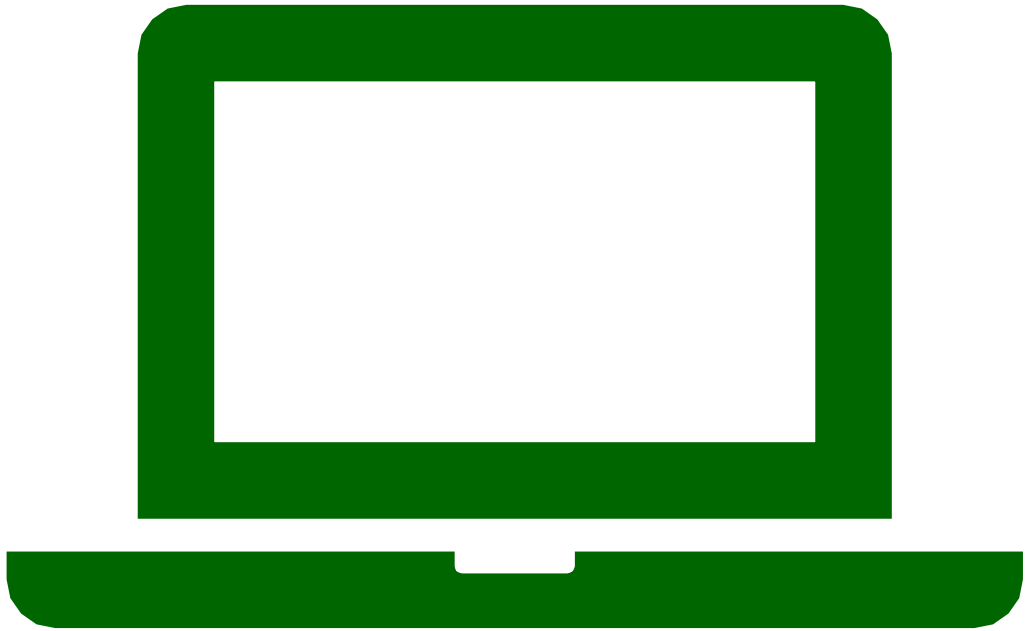
Exercise 4.34

- Display the game timer on screen using the showText method. You can do this in the **CrabWorld**'s act method. You need a statement similar to this:

```
showText("Time left: "+ time, 100, 40);
```

- where **time** is the name of your timer variable. (Note: this statement uses the plus operator and a text string, which we will explain in Chapter 5.)

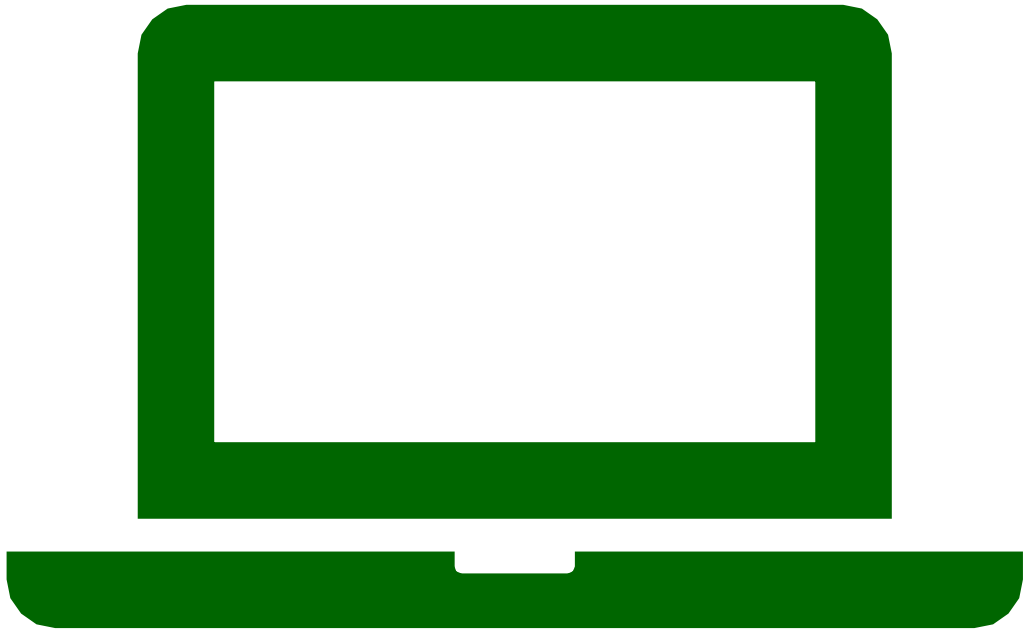
Bouncing Ball Practice



Exercise 4.35

Exercise 4.35

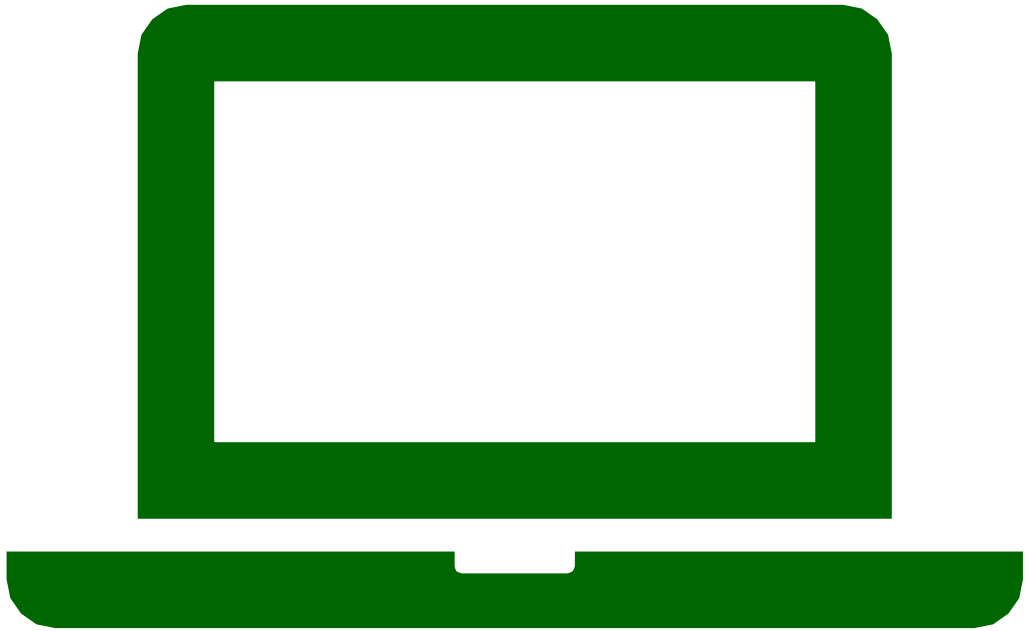
- Create a new scenario. In it, create a World and an Actor class called Ball. (Give it a ball-like image.) Program the ball so that it moves at constant speed, and bounces off the edges of the world.



Exercise 4.36

Exercise 4.36

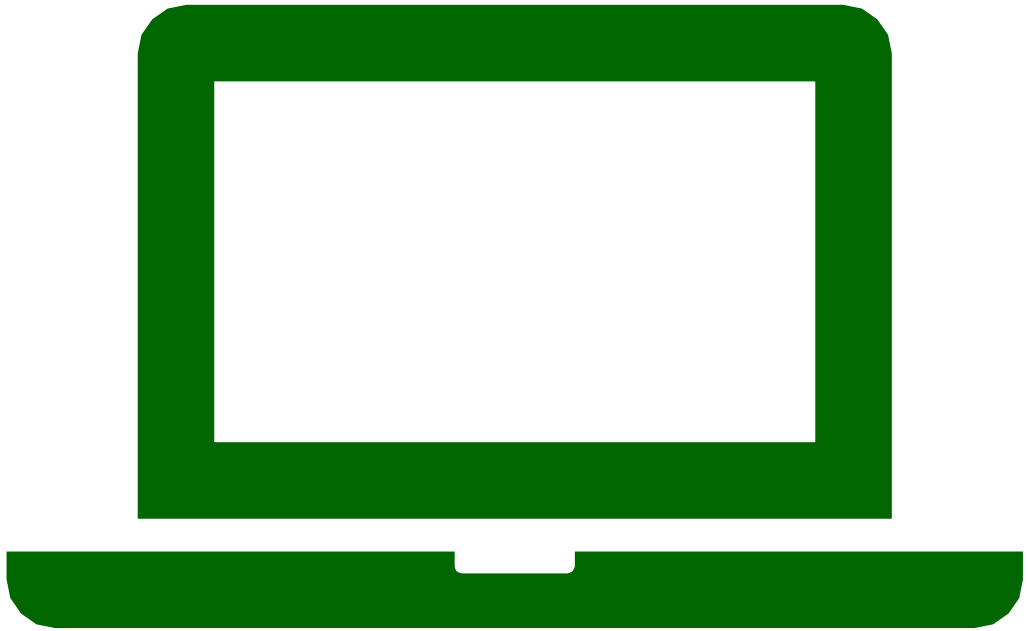
- Program your ball so that it counts how often it has bounced off the edge. Run your scenario with the ball's object inspector open to test.



Exercise 4.37

Exercise 4.37

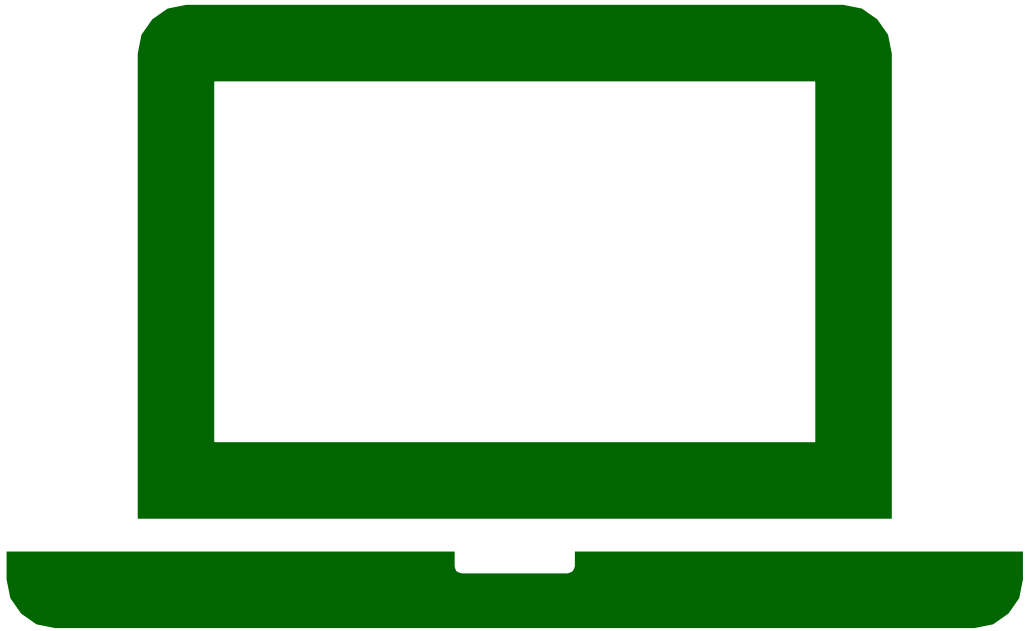
- Program your scenario so that three balls are automatically present at the start.



Exercise 4.38

Exercise 4.38

- Change your setup code, so that the three balls appear at random locations.



Exercise 4.39

Exercise 4.39

- Change the bouncing-off-the-edge code so that the balls bounce off the edge at somewhat random angles.