Introduction To Programming
With Greenfoot

Object-Oriented Programming in Java
With Games and Simulations
Ch 4
Pixel

From Wikipedia, the free encyclopedia

A pixel is generally thought of as the smallest single component of a digital image.

This example shows an image with a portion greatly enlarged, in which the individual pixels are rendered as little squares and can easily be seen.
4. Finishing the crab game

Here we will discuss a number of improvements.

4.1 Adding objects automatically

We need to address the fact that every time we compile, we always have to place the actors (the crab, lobsters, and worms) manually into the world. It would be better if that happened 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.

```java
import greenfoot.*;
public class CrabWorld extends World
{
    /*
     */
    /*
    */
    public CrabWorld()
    {
        super(560, 460, 1);
        setBackground("sand.jpg");
    }
}
```

This method 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.

3. A constructor is automatically executed when an instance of this class is created.
(Remember, the origin – the 0/0 point – of our co-ordinate system is at the top left of the world.)
In our case, the constructor sets the world to the size we want (560 by 460 cells) and a *resolution* (1 pixel per cell).

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.

```java
public CrabWorld()
{
    super(560, 460, 1);
    setBackground("sand.jpg");
    // add objects here
}
```
Look at the class documentation of the **World** class for a method that would add an object to the world

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>act()</strong></td>
</tr>
<tr>
<td>Act method for world.</td>
</tr>
<tr>
<td>Add an Actor to the world.</td>
</tr>
<tr>
<td><strong>getBackground()</strong></td>
</tr>
<tr>
<td>Return the world's background image.</td>
</tr>
<tr>
<td><strong>getCellSize()</strong></td>
</tr>
<tr>
<td>Return the size of a cell (in pixels).</td>
</tr>
<tr>
<td><strong>getColorAt(int x, int y)</strong></td>
</tr>
<tr>
<td>Return the color at the center of the cell.</td>
</tr>
<tr>
<td><strong>getHeight()</strong></td>
</tr>
<tr>
<td>Return the height of the world (in number of cells).</td>
</tr>
<tr>
<td><strong>getObjects(java.lang.Class&lt;?&gt; cls)</strong></td>
</tr>
<tr>
<td>Get all the objects in the world, or all the objects of a particular class.</td>
</tr>
<tr>
<td><strong>getObjectsAt(int x, int y, java.lang.Class cls)</strong></td>
</tr>
<tr>
<td>Return all objects at a given cell</td>
</tr>
<tr>
<td><strong>getWidth()</strong></td>
</tr>
<tr>
<td>Return the width of the world (in number of cells).</td>
</tr>
<tr>
<td><strong>isTiled()</strong></td>
</tr>
<tr>
<td>Returns true if the world is tiled.</td>
</tr>
<tr>
<td><strong>numberOfObjects()</strong></td>
</tr>
<tr>
<td>Get the number of actors currently in the world.</td>
</tr>
</tbody>
</table>
The `addObject` method is a method of the `World` class. There, we see that it has the following signature:

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

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`. 
The method `addObject` belongs to the `World` class, and since `CrabWorld` is itself a `World` (it inherits all methods from its superclass).

Therefore `addObject` is available in our `CrabWorld` class, and we can just call it without the `World` prefix.
4.2 Creating new objects

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

In order to add an object, we must first have an object to add.

The Java keyword `new` allows us to create new objects of any of the existing classes. For example, the expression

```
new Crab()
```

creates a new instance of class Crab.

We can now use it in place of the actor parameter of the `addObject` method to add this object to the world.

```
addObject( new Crab(), 150, 100 );
```

The remaining two parameters specify the x and y coordinate of the position where we wish to add the object.
**Exercise 4.1** Add code to the *CrabWorld* constructor of your own project to create a crab automatically, as discussed above.

**Exercise 4.2** Add code to automatically create three lobsters in the *CrabWorld*. You can choose arbitrary locations for them in the world.

**Exercise 4.3** Add code to create ten worms in the *CrabWorld*.

**Exercise 4.4** Move all the code that creates the objects into a separate method, called *populateWorld*, in the *CrabWorld* class. You need to declare the *populateWorld* method yourself (it takes no parameters and returns nothing) and call it from the constructor. Try it out.

**In the *populateWorld* method, can you make the locations of the crab, lobsters and worms random?**
4.3 Animating images

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 called *crab.png* and *crab2.png*, and we shall simply switch the crab's image between these two versions fairly quickly, to make the crab look as if it is moving its legs.
4.4 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.

```
GreenfootImage(java.lang.String filename)
    Create an image from an image file.
```

For example, to get access to the *crab2.png* image, we can write

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

The file we name here must exist in the scenario's "images" folder.
**Exercise 4.5** 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?

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getOneObjectAtOffset(int dx, int dy, java.lang.Class cls)</code></td>
<td>Return one object that is located at the specified cell (relative to this object's location).</td>
<td></td>
</tr>
<tr>
<td><code>getRotation()</code></td>
<td></td>
<td>Return the current rotation of the object.</td>
</tr>
<tr>
<td><code>getWidth()</code></td>
<td></td>
<td>Return the width of the object.</td>
</tr>
<tr>
<td><code>getWorld()</code></td>
<td></td>
<td>Return the world that this object lives in.</td>
</tr>
<tr>
<td><code>getX()</code></td>
<td></td>
<td>Return the x-coordinate of the object's current location.</td>
</tr>
<tr>
<td><code>getY()</code></td>
<td></td>
<td>Return the y-coordinate of the object's current location.</td>
</tr>
<tr>
<td><code>intersects(Actor other)</code></td>
<td></td>
<td>Check whether this object intersects with another given object.</td>
</tr>
<tr>
<td><code>setImage(GreenfootImage image)</code></td>
<td></td>
<td>Set the image for this object to the specified image.</td>
</tr>
<tr>
<td><code>setImage(java.lang.String filename)</code></td>
<td></td>
<td>Set an image for this object from an image file.</td>
</tr>
<tr>
<td><code>setLocation(int x, int y)</code></td>
<td></td>
<td>Assign a new location for this object.</td>
</tr>
<tr>
<td><code>setRotation(int rotation)</code></td>
<td></td>
<td>Set the rotation of the object.</td>
</tr>
</tbody>
</table>
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.

```java
void setImage(GreenfootImage image)
    Set the image for this object to the specified image.
```
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 store the two images in our crab object, we need a new construct that we have not used before: **variables**.

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

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

```java
setImage( image1 );

setImage( image2 );
```
4.5 Instance variables (fields)

Often, our actors need to remember some information. In programming languages, this is achieved by storing the information in a variable.

Java supports different kinds of variables. The first one we shall look at here are called instance variables, or fields. (These two terms are synonymous.)

An instance variable is a bit of memory that belongs to the object (the instance of the class, hence the name). Anything stored in it will be remembered as long as the object exists, and can be accessed later.
An instance variable is declared in a class by writing the keyword `private` followed by the type of the variable and the variable name:

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

The type of the variable defines what we want to store in it. In our case, since we want to store objects of type `GreenfootImage` in our variable, the type should be `GreenfootImage`. The variable name gives us a chance to give a name to the variable that we can use later to refer to it. It should describe what this variable is used for.

Let us look at our Crab class as an example.
public class Crab extends Animal
{
    private GreenfootImage image1;
    private GreenfootImage image2;

    // constructors and methods omitted
}
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. In the figure, the instance variables are depicted as two white boxes.

```java
private GreenfootImage image1;
private GreenfootImage image2;
```

Next, we have to create the two image objects and store them into the variables. The creation of the objects we have already seen above. It was achieved with the code snippet

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

To store the object into the variable, we need a Java construct known as an assignment.

*assignment*
4.6 Assignment

An assignment is an instruction that enables us to store something into a variable. It is written with an equals symbol:

```
variable = expression;
```

On the left of the equals symbol is the name of the variable we want to store into, on the right is the thing that we want to store.
In our crab example, we write:

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

These two lines of code will create the two images we wish to use and store them into our two variables `image1` and `image2`. 
The next question regarding the creation of these images 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.
4.7 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.

```java
public CrabWorld()
{
    super(560, 460, 1);
    setBackgroundImage("sand.jpg");
    // add objects here
}
```
4.7 Using actor constructors

In a similar manner, we can use a constructor of an actor class to initialize the actor.

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

// comment omitted

class Crab extends Animal
{
    private GreenfootImage image1;
    private GreenfootImage image2;
}

// Constructor for the Crab class
public Crab() {
}
```
/**
 * Create the crab's two images.
 */
public Crab()
{
    image1 = new GreenfootImage("crab.png");
    image2 = new GreenfootImage("crab2.png");
    setImage(image1);
}

The same rules described for the world constructor apply to the Crab constructor:

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.

3. A constructor is automatically executed when an instance of this class 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.
The last line of the constructor sets the first of the two created images as the crab's current image:

```java
setImage(image1);
```

This demonstrates how the name of the variable (image1) can be used now to refer to the image object stored in it.
Exercise 4.10 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 4.11 Inspect your crab object again. Take a note again of the variables and their values.
The code in the constructor will be executed once when the actor is created.

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

// comment omitted

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

    /**
     * Create the crab's two images.
     */
    public Crab()
    {
        image1 = new GreenfootImage("crab.png");
        image2 = new GreenfootImage("crab2.png");
        setImage(image1);
    }

    // other methods omitted
}
```

31
4.8 Alternating the images

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:

```pseudo
if (our current image is image1) then
    switch to image2
else
    switch to image1
```

**Pseudo-code**, as used here, is a technique expressing a task in a structure that is partly like real Java code, and partly natural language. It often helps in working out how to write our real code.
In this code segment, we notice several new elements:

- 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 to another. The result is either `true` or `false`.

- This form has an `else` keyword after the first body of the if statement, followed by another block of statements.

We can test whether two things are equal by using a double equals symbol: `==`. 
4.9 The if/else statement

As we have just seen, an if statement can be written in the form

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

When this if statement is executed, first the condition is evaluated. If the condition is true, the first block of statements is executed, and then execution continues below the complete if statement.

If the condition is false, the first block of statements is not executed, instead we execute the second statement block (after the `else` keyword).

Thus, one of the two statement blocks is always executed, but never both.
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.

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

```
if ( condition )
{
  statements;
}
```
Exercise 4.12 Add the image switching code, as shown, 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 4.13 In Chapter 4, 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 4.14 Call the switchImage method interactively from the crab's popup menu. Does it work?
4.10 Counting worms

We want to add functionality so that the crab counts how many worms it has eaten, and 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 shall need a number of additions to our crab code. We need:

- an instance variable to store the current count of worms eaten;
  ```java
  private int wormsEaten;
  ```
- an assignment that initializes this variable to zero at the beginning;
  ```java
  wormsEaten = 0;
  ```
- code to increment our count each time we eat a worm; and
  ```java
  wormsEaten = wormsEaten + 1;
  ```
- code that checks whether we have eaten eight worms, and stops the game and plays the sound if we have.
  ```java
  if (wormsEaten == 8)
  {
  }
  ```
Below our two exiting instance variable definitions, we add the line

```java
private int wormsEaten;
```

The type `int` indicates that we want to store integers (whole numbers) in this variable, and the name `wormsEaten` indicates what we intend to use it for.

Next, we add the following line to the end of our constructor:

```java
wormsEaten = 0;
```

This initializes the `wormsEaten` variable to zero when the crab is created.
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:

```java
wormsEaten = wormsEaten + 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.

```java
if (wormsEaten == 8) {
    // plays the sound
    // stop the execution
}
```

Check the documentation of the `???` Class for a method to play a sound and a method to stop the simulation.
The complete **lookForWorm** method. The sound file used here ("fanfare.wav") is included in the sounds folder in your scenario, so it can just be played.

**Exercise 4.15** Add the code discussed above into your own scenario. Test it, and make sure that it works.

**Exercise 4.16** As a further test, open an object inspector for your crab object (by selecting ‘Inspect’ from the crab’s popup menu) before you start playing the game. Leave the inspector open and keep an eye on the ‘wormsEaten’ variable while you play.

**Exercise 4.17** 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 gets incremented in the ‘act’ method. Every time it reaches 2 (or 3) the image changes, and the counter is reset to zero.
4.11 More ideas

There are many obvious things (and probably many more less obvious things) you can do with this project. Ideas include

- using different images for the background and the actors;
- using more different kinds of actors;
- not moving forward automatically, but only when the up-arrow key is pressed;
- making new worms pop up when one is eaten (or at random times);
- and many more that you can come up with yourselves...

step 1: the world shall have no "borders" - i.e. when you leave on the right side you re-appear on the left side.

step 2: there should be "walls" - here we used corall to be with the topic "Under-Sea-World"

step 3: the Lobster should become intelligent, i.e. every 10 moves he should rotate in direction of the crab which he is hunting...
4.12 Summary of programming techniques

In this chapter, we have seen a number of 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 how to use instance variables – also called fields – and assignment statements to store information, and how to access that information later.

We have used a new instruction to programmatically create new objects.

We have seen the full version of the if statement, in which it includes and else part that is executed when the condition is not true.
## 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.

- Greenfoot actors maintain their visible image by holding an object of type `GreenfootImage`.

- **Instance variables** (also called **fields**) can be used to store information (objects or values) for later use.

- An **assignment statement** assigns an object or a value to a variable.

- When an object is assigned to a variable, the variable contains a **reference** to that object.

- 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.
Doing more

To find out what other methods are available to do things in Greenfoot, use the 'Greenfoot Class Documentation' item from the Help menu.