Greenfoot is an **Integrated Development Environment** (IDE) geared at beginners, which can be used to teach **object-oriented programming**. In object-oriented programming (OOP) each piece of data, along with the functions that work with that data, is packaged as an "**object**". Objects can be re-used, making code more modular, and to work with an object a programmer only needs to know how to interact with it (what instructions it will understand) rather than all the details of the object's underlying structure.

This flexibility makes OOP good for complex programming tasks. For this reason, many of today's programming languages are wholly or partially object-oriented, for instance Python, C++, Java, C# and VB.NET. So, learning a bit about OOP is a good grounding for the future.

Greenfoot is highly visual and interactive, making it easy to get started creating all sorts of scenarios and games. Greenfoot uses Java as its programming language, one of the most popular languages used in industry – so concepts that you learn with Greenfoot will directly translate to any future programming you may encounter!

If you get stuck with anything in this tutorial, then feel free to head over to the Greenfoot forums (***http://www.greenfoot.org/topics***), where you can ask for help.

Notes:

Tip...

An Integrated Development Environment (IDE) is a suite of software that provides the programmer with all the tools they need to work with a certain programming language.

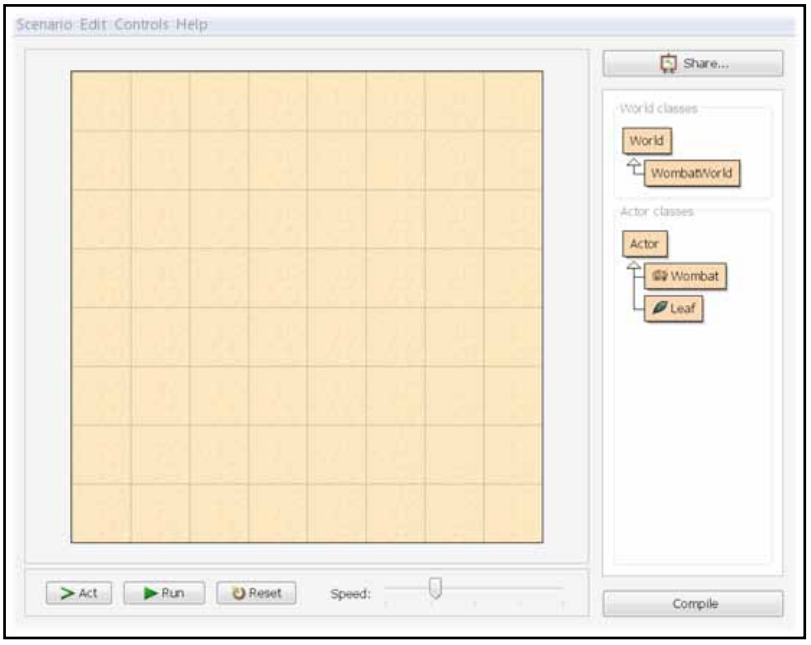
Lesson 2.1: Interacting with Greenfoot

Notes:

This tutorial will explain the basics of the Greenfoot interface, and interacting with Greenfoot. It uses a scenario called "Wombats", which is distributed with Greenfoot.

Open Greenfoot and launch the "Wombats" scenario; you should then see this:

The Greenfoot interface.



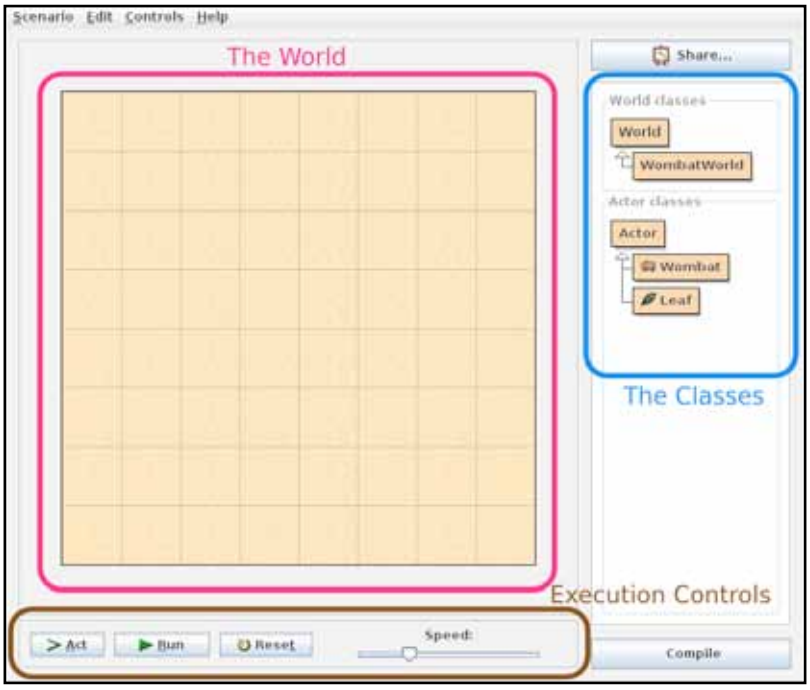
If you don't see the world, and the classes on the right have diagonal slashes over them, this is because the code is uncompiled. Click on the "**Compile**" button in the bottom-right and that should fix the problem.

The large grid area that covers the majority of the window is called the "**world**". Since we have a scenario here that has to do with wombats, we see a wombat world. Towards the right side of the window is the class display. Here you can see all Java classes that are involved in the project. The classes "World" and "Actor" will always be there – they come with the Greenfoot system. The other classes belong to the Wombat scenario, and will be different if you use different scenarios.

Below the world are the Execution Controls: the "**Act**" and "**Run**" buttons and the slider.

Let's label all these things on our interface:

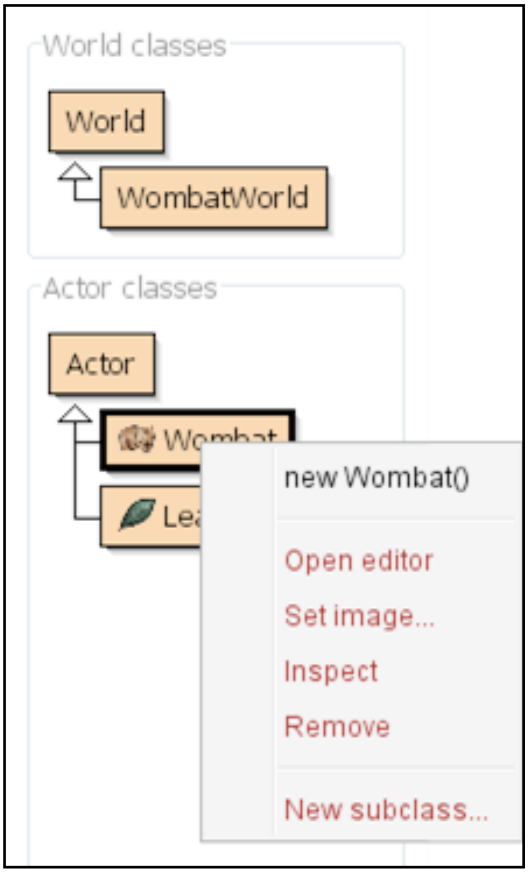
Notes:



Greenfoot's world, classes and execution controls.

Place objects into the world

We will now place some objects into the world. Right-click the Wombat class in the class display. You will see a pop-up menu like this:



The Wombat class.

Choose “new Wombat()” from the menu. Then click anywhere in the world. You have just created a wombat (in Java terms: an **object**) and placed it into the world.

Wombats eat leaves, so let’s put some leaves into the world as well. Right-click the “Leaf” class, select “new Leaf()” and place the leaf into the world.

There is a shortcut to place several objects a bit faster: shift-clicking into the world. Make sure the “Leaf” class is selected (left click on it in the classes panel, and it will get a thicker black border), then hold down the Shift key and left-click in the world several times. You will place one object of the selected class at every click. Much faster!

Notes:

Make objects act

Click the “**Act**” button in the execution controls. Each object now acts – that is: each object does whatever it is programmed to do. In our example, leaves are programmed to do nothing, while wombats are programmed to move forward. Try placing two wombats into the world and press “**Act**” again. Both will move.

Wombats also like to eat leaves. If they happen to come across a leaf in their path, they will eat it. Try placing some leaves in front of the wombats, then click “**Act**” – the wombats will move forward and eat the leaves.

Run a scenario

Click the “**Run**” button. This is equivalent to clicking the “**Act**” button over and over again, very quickly. You will notice that the “**Run**” button changes to a “**Pause**” button; Clicking “**Pause**” stops everything acting.

The slider next to the “**Act**” and “**Run**” buttons sets the speed.

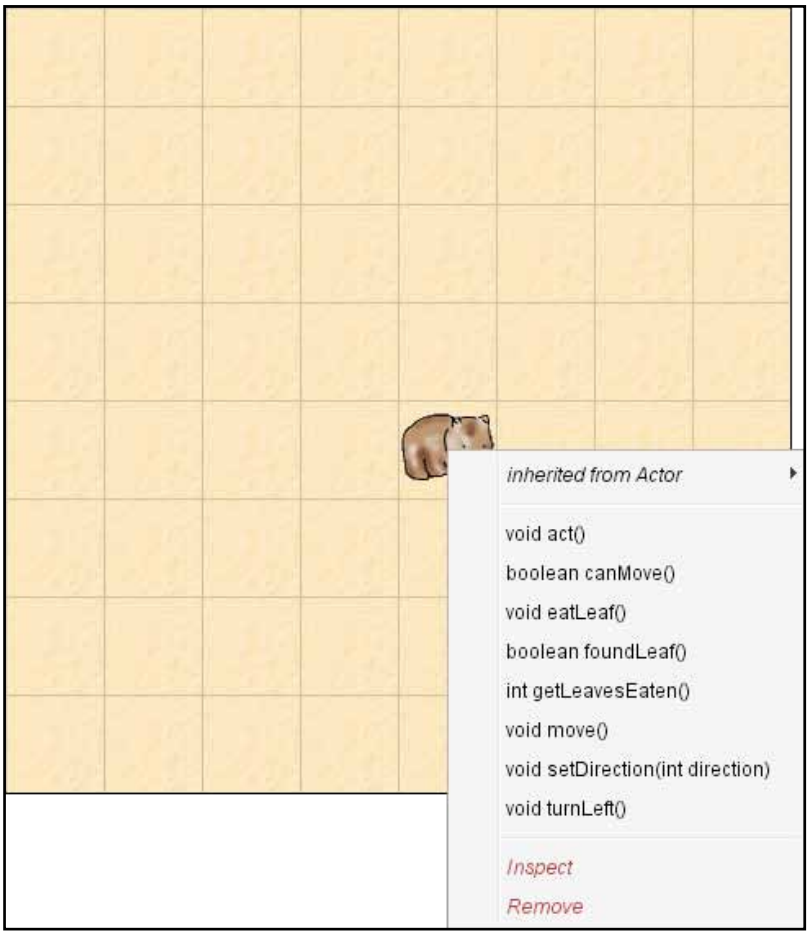
Click “**Run**” and then change the slider, and you’ll see the difference.

Invoke methods directly

Instead of just running the whole scenario, you can also invoke single **methods**. A method is a single action that an object can perform.

Make sure you have a wombat in the world, and the scenario is not running. Then right-click on the wombat (the one in the world, not the “Wombat” class), and you will see that objects in the world also have a pop-up menu (shown on the next page).

The pop-up menu for the Wombat object.



Notes:

You can select any of the methods shown here to ask the wombat to do something. Try, for example, "turnLeft()". Selecting this from the menu tells the wombat to turn to its left. Try "move()", as well.

Some methods give you an answer. "getLeavesEaten()", for example, will tell you how many leaves this wombat has eaten so far. Try it. Then get the wombat to eat another leaf, and try calling that method again.

You will also notice a method called "act()". This method is called every time you click the "**Act**" button. If you want just one object to act instead of all the objects in the world, you can do this by invoking the object's "act()" method directly.

Create a new world

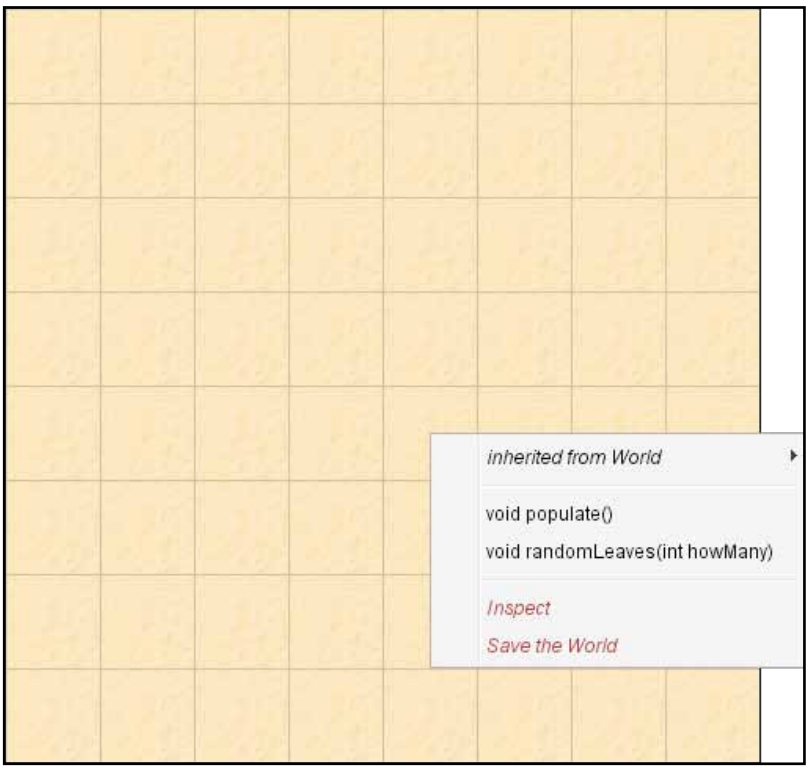
If you have many objects in the world that you do not want anymore, and you want to start all over, there is one easy option: throw away the world and create a new one. This is usually done by clicking the "**Reset**" button in the execution controls. You will get a new, empty world. The old world is discarded (and with it all the objects that were in it) – you can only have one world at a time.

Invoke a world method

Notes:

We have seen that objects in the world have methods that you can invoke via a pop-up menu. The world itself is also an object with methods that you can invoke. Right-click on any empty space in the world, or in the grey area immediately next to the world, and you will see the world's menu:

The pop-up menu for the world



One of the methods in this menu is "populate()". Try it out. It is a method that creates several leaves and wombats and places them into the world. You can then run the scenario.

Another world method is "randomLeaves(int howMany)". This method places some leaves in the world at random locations. Note that this method has some words between the parentheses after its name: "int howMany".

This is called a **parameter**. It means that you must specify some additional bit of information when you invoke this method. The term "int" tells you that a whole number is expected (an **integer**), and the name "howMany" suggests that you should specify how many leaves you want. Invoke this method. A dialogue will pop up that lets you enter a value for this parameter. Enter a number (say, 12) and click OK.

(You may notice, if you count, that it sometimes appears as if fewer than the specified number of leaves were created. This is because some leaves may be at the same location, and are lying on top of each other.)

Okay, that's enough of wombats running around in circles, endlessly – let's move on to the really interesting stuff: programming!

Lesson 2.2: Movement and key control

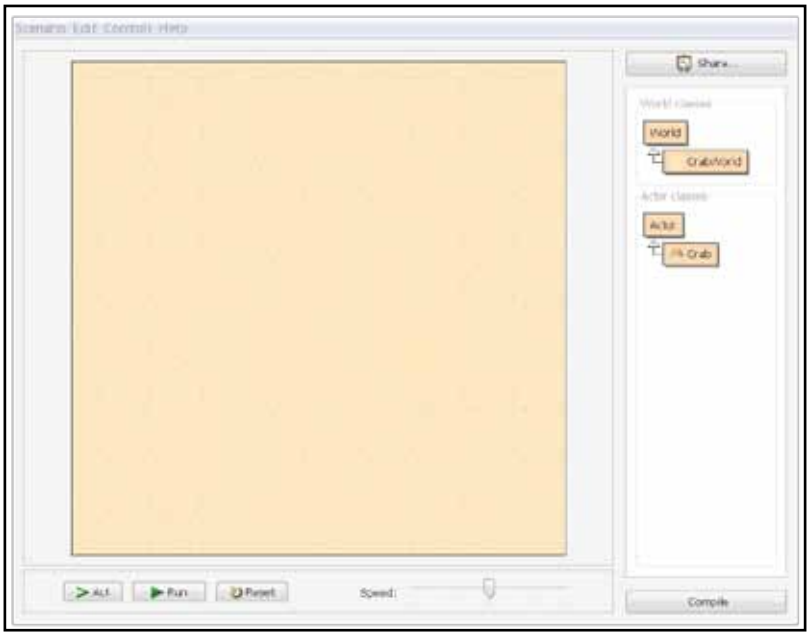
Notes:

This tutorial will explain how to do movement in Greenfoot, and how to control actors with the keyboard. For this section you will need the "Crabs" scenario, which you can download from here:
http://www.greenfoot.org/tutorial-files/modern-crab.zip.

The Crabs scenario

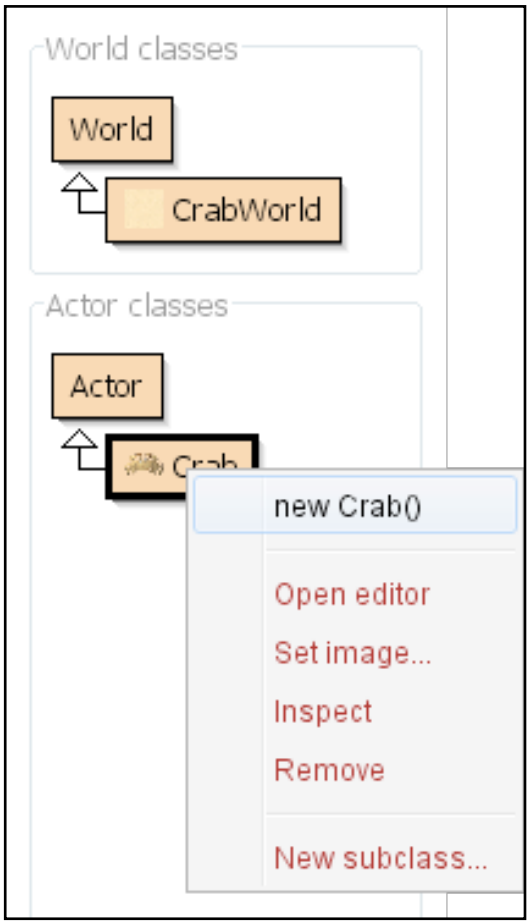
Open the "Crabs" scenario in Greenfoot. You should see the standard Greenfoot interface, with an empty sandy world:

Crab World.



Right-click on the "crab" class and select "new Crab()", then left click on the world to place the crab (shown on the next page).

The Crab class.

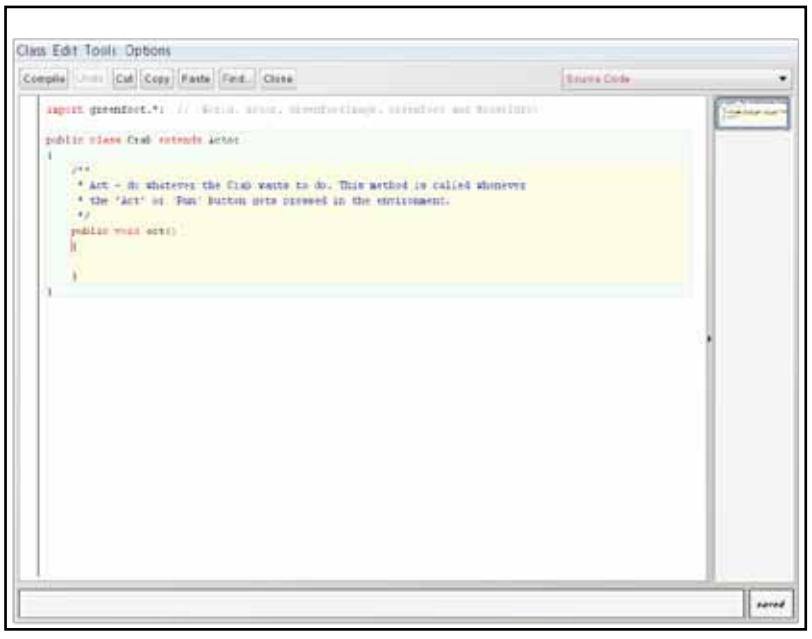


Notes:

After that, click "**Run**". You might be hoping you could watch the crab do an amazing dance around the screen. Unfortunately, it seems we have a lazy crab! Let's open up the code and have a look; you can either double-click on the "crab" class in the class browser, or you can right-click and select "**Open Editor**".

What you'll see is the Java code for the crab. You don't need to understand all of it just yet, but the important bit is the code between the curly brackets below the "public void act()" line. There's nothing there at the moment:

The code editor for the Crab class.



Bracket spotting

There are three main bracket types used in Java. Let's see some blown-up images of them:

() ***These are round brackets, known in the USA as "parentheses", but in the UK simply as "brackets". In Java, they are used in mathematical expressions, and to surround the parameter lists for method calls (we'll get to those shortly).***

[] ***These are square brackets, known in the USA simply as "brackets" (see where confusion can arise?). In Java, they are used for arrays.***

{} ***These are curly brackets, also known as "braces" or "squiggly brackets". In Java, these are used to surround blocks of code, such as methods or the contents of classes.***

If we want our crab to do anything, we're going to have to fill that in. Let's get it moving, by adding a move instruction to the code:

This code will move our crab.

```
/**
 * Act - do whatever the Crab wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act()
{
    move(4);
}
```

You need to precisely match what is written here. It's the word "move", followed by round brackets containing the number "4", followed by a semi-colon. Common mistakes include capitalising the "m" (capitalisation matters in Java!), missing the semi-colon, using the wrong brackets (or thinking that empty round brackets are a zero), or accidentally deleting the curly brackets. If you get an error during this tutorial, look for one of these errors that you might have made when copying the code.

Once you've written this, hit the "**Compile**" button in the main Greenfoot interface (or at the top of the editor window) then place a crab in the world again and click "**Run**". Now the crab should glide sideways across the screen. Then it should hit the edge of the world and abruptly stop. If you like, you can pause the scenario, drag the crab over to the left, hit "**Run**" and watch it again. Why not place a few more crabs in the world and watch them all do that?

The crabs aren't actually stopping, as such; they are still trying to move, but Greenfoot doesn't let them move out of the world (if they did, how would you drag them back in again?). You can vary the speed of the crab by changing the number "4" in the code to a different number. Higher will be faster, lower will be slower – see if you can guess what happens with a negative number.

Notes:

Let's make the crab do a bit more than moving in a straight line. Go back to the code, and after the move line, add another line (but still inside the curly brackets for the act method) that says "turn(3)", like this:

Notes:

The turn command makes the crab run in circles.

```
/**
 * Act - do whatever the Crab wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act()
{
    move(4);
    turn(3);
}
```

You'll see that the crab runs in a circle. Experiment with the turning amount to get the circle tighter or larger.

The good thing about the crab turning all the time is that, even if it does hit the edge of the world, eventually it will turn enough that it moves back off the edge of the world and into the middle again. What would be even better is if we can control the crab's turning – let's get some interaction in our scenario! We can make the crab turn when we press the left or right cursor (arrow) keys. Here's the code:

The first "if" statement controls turns to the left; the second controls turns to the right.

```
/**
 * Act - do whatever the Crab wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act()
{
    move(4);

    if (Greenfoot.isKeyDown("left"))
    {
        turn(-3);
    }
    if (Greenfoot.isKeyDown("right"))
    {
        turn(3);
    }
}
```

We use the Greenfoot built-in methods for checking if a key is down. Between the quotes is the name of the key, "left" is the left cursor key, "right" is right. If you want something like "a" and "d", just use those instead! Our code is saying: if those keys are down, turn a certain number of degrees. Enter that code, compile it and try it out for yourself. You can alter how fast the crab turns by increasing those numbers.

If you put multiple crabs in the world, you'll find that pressing the keys controls all of them at once in glorious synchronisation, making you into some sort of crab overlord: all of the crabs are executing the same code, so if you press the left key, all of them will see that the left key is down, and they will all turn accordingly.

What happens if you hold down left and right at the same time? Try it, and find out – then look at the code and see if you can work out why that is.

Lesson 2.3: Detecting and removing actors, and making methods

Notes:

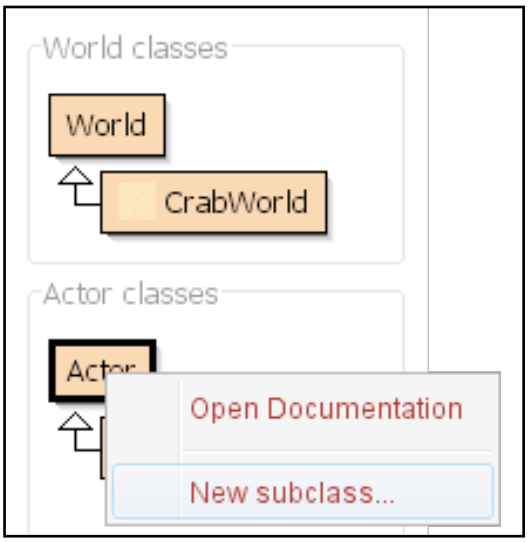
This section will explain how to find out if you are touching another actor, and subsequently remove that actor from the world. It will also explain how to keep your code readable using methods.

Eating worms

We're going to continue on directly from Lesson 2.2, in which we had our crabs moving around the screen, under our control. In this part, we're going to introduce some worms to eat.

We already have a class for the crab – we're now going to add another for the worm. Worm is also going to be an actor, so to make the "Worm" class, right-click on the "Actor" class and select "new subclass...":

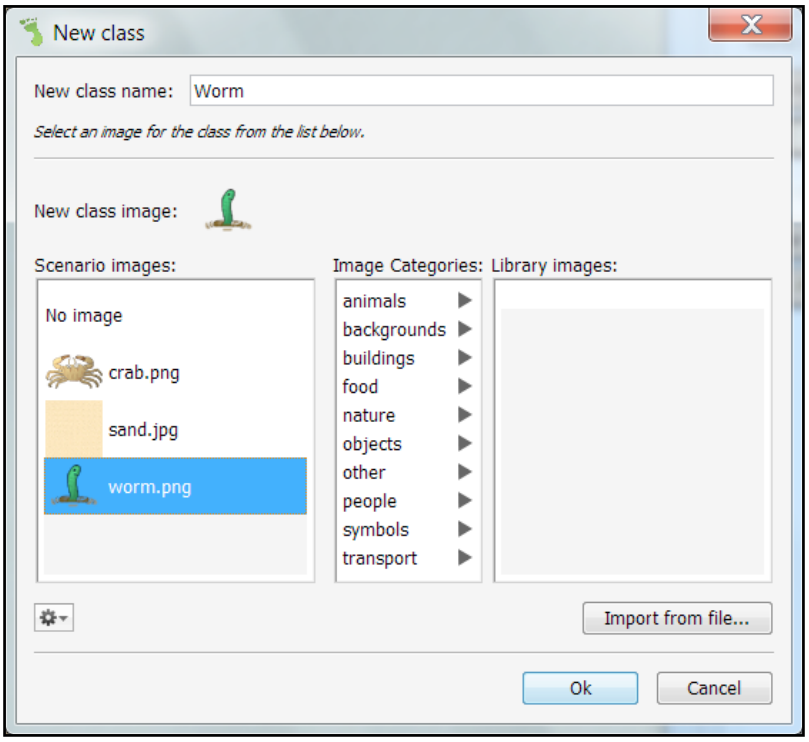
Select "New subclass..." from the drop-down menu.



As the new class name, enter: "Worm" – note that we are using a capital "W". It's the convention in Java that we begin class names with a capital letter. If you accidentally use a small "w" now, you'll get an error later on when copying my code that uses a capital "W". From the left-hand list of images, select "*worm.png*" as the image and press okay.

Notes:

Creating a new subclass.



Our "Worm" class will have no real code in it to begin with – just like our crab class when we started. We're going to leave it that way; our worms are dumb and just sit in one place ready to be eaten. Easy prey! We're going to modify our "Crab" class so that when our crabs pass over a worm, they eat the worm. To do this, we go back to our crab's source code, which currently looks like this:

The crab's source code.

```
/**
 * Act - do whatever the Crab wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act()
{
    move(4);

    if (Greenfoot.isKeyDown("left"))
    {
        turn(-3);
    }
    if (Greenfoot.isKeyDown("right"))
    {
        turn(3);
    }
}
```

At the end of the "act()" method, we're going to insert some code to check whether the crab is currently touching a worm. We'll use the method "getOneObjectAtOffset". This method takes three parameters. The first two are the X and Y offset (difference) from our current position. We want to know about what's directly underneath us, so we'll pass zero for both of these. The third parameter allows us to indicate which class we're interested in; that's the "Worm" class. Take a look at the code on the next page.

The additional code at the end checks to see if the crab is currently touching a worm.

```
/**
 * Act - do whatever the Crab wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act()
{
    move(4);

    if (Greenfoot.isKeyDown("left"))
    {
        turn(-3);
    }
    if (Greenfoot.isKeyDown("right"))
    {
        turn(3);
    }

    Actor worm;
    worm = getOneObjectAtOffset(0, 0, Worm.class);
}
```

Note that we're not only calling the method, we're doing something on the left-hand side, too. This method returns something (the object underneath us, if there is one), so we need to store this return value ready to use it again. To do this we declare a **variable** (something for holding values), which we've called "worm", on the previous line. Then we use the **assignment operator** (an equals sign) to indicate that the value of worm should be set equal to the return value of the method.

Now we have the return value of the method in the variable "worm". If there was no worm touching us, this variable will contain the special value "**null**". We can only remove the worm when there is a worm, so we need a check that the worm is not the special null value before removing:

The new "if" statement confirms there is a worm (worm NOT EQUAL null) and then removes it from the world.

```
/**
 * Act - do whatever the Crab wants to do. This method is called whenever
 * the 'Act' or 'Run' button gets pressed in the environment.
 */
public void act()
{
    move(4);

    if (Greenfoot.isKeyDown("left"))
    {
        turn(-3);
    }
    if (Greenfoot.isKeyDown("right"))
    {
        turn(3);
    }

    Actor worm;
    worm = getOneObjectAtOffset(0, 0, Worm.class);
    if (worm != null)
    {
        World world;
        world = getWorld();
        world.removeObject(worm);
    }
}
```

The "**!=**" is the "**does not equal**" operator in Java. This means we will only execute the code when "worm" is not null. The code in question is to get a reference to the world in the variable imaginatively named "world" and tell it to remove the worm using the "removeObject" method.

So, let's give it a test. Compile it, create some worms and place them in to your world, then add a crab, hit run and use your left and right keys to guide the crab to the worms, which should then get eaten. Tasty!

Notes:

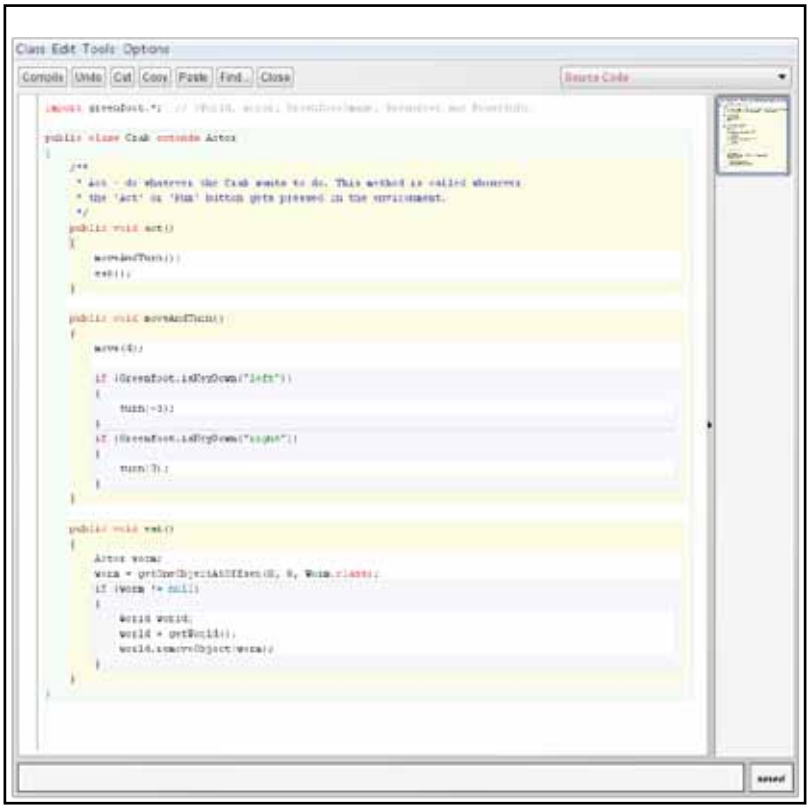
Refactoring

Notes:

Before we move on, we are going to make a change to the code. This is something known as **refactoring**: changing the code without changing its behaviour. Currently the "run()" method for the crab has two distinct behaviours in it: the top half deals with moving around, and the bottom half deals with eating worms. It would be clearer to name these and split them up, to avoid confusing ourselves later on. We can do this by creating a separate method for each behaviour. We've actually already seen how methods are written: "act()" is a method, after all.

Both of our new methods require no parameters and return no value, so they can be declared just the same as "act()". The adjusted code is below:

The refactored worm code.



If you compare this code to the previous version, you'll notice that we haven't actually changed or removed any of the existing code. We moved the top half into a new method, called "moveAndTurn()", and we moved the bottom half into a new method, called "eat()". We then replaced the contents of the "act()" method with calls to these new methods. This will make the code easier to follow when we have to focus on them in future parts of the tutorial.

Lesson 2.4: Saving the world and playing with sound

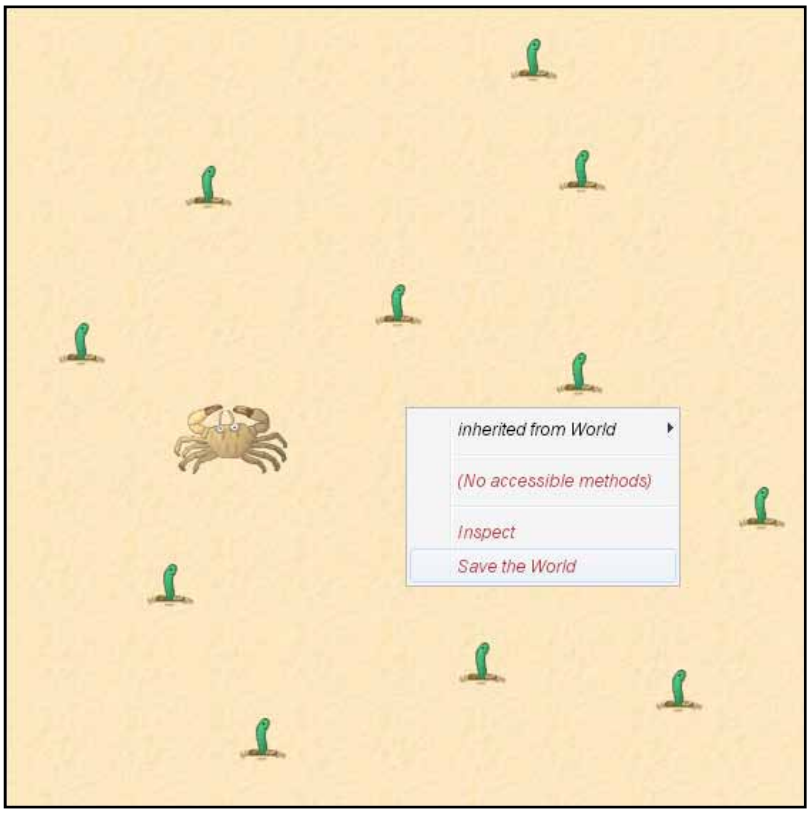
Notes:

This tutorial will explain how to initialise the world with actors, as well as how to play and record sounds.

Saving the world

By now you're probably tired of always having to add new objects to the world every time we compile the code. It's possible to add some code to automatically create some worms and a crab for you – what's more, it's possible to get Greenfoot to write that code for you! Hit "**Reset**", and then add some worms and a crab into the world. Before you press "**Run**", right-click on the world and select the "**Save the World**" option:

Save the World!



This adds some code to our CrabWorld class that will create the worms and crab and add them to the world at the same position next time you reset or compile. We're not going to explore the code in detail just now, but if you're curious then look in the source code for the "CrabWorld" class that pops up. Once you're done, you can close the window for the "CrabWorld" source code.

Playing and recording sounds

Notes:

We can add some sound to our scenario. The scenario comes with a sound ready for you to use, named "*eating.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" after we remove a worm from the world:

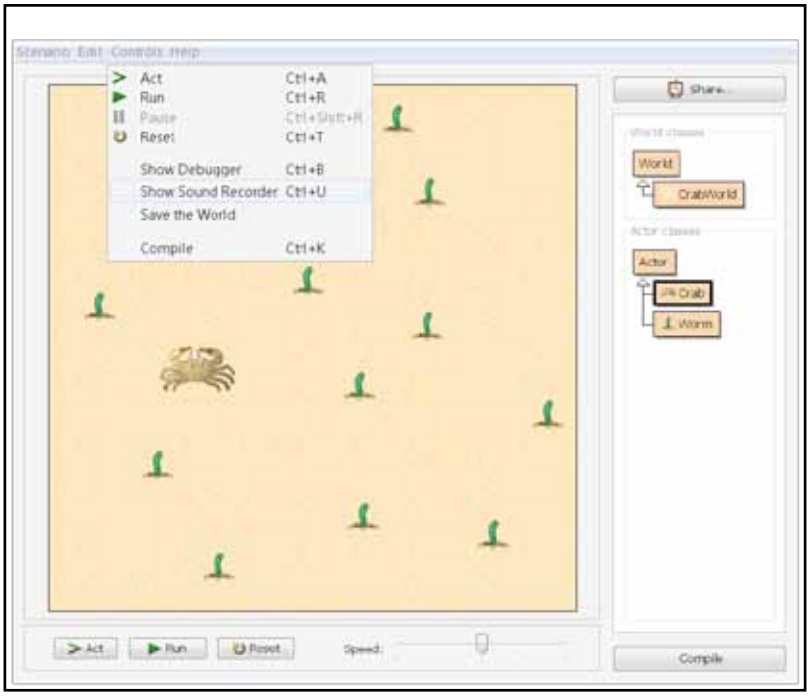
Adding sound when a worm is eaten.

```
public void eat()
{
    Actor worm;
    worm = getOneObjectAtOffset(0, 0, Worm.class);
    if (worm != null)
    {
        World world;
        world = getWorld();
        world.removeObject(worm);
        Greenfoot.playSound("eating.wav");
    }
}
```

Give that a try. 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! To do this, follow these steps. In the **Controls** menu there is an option to show the sound recorder:

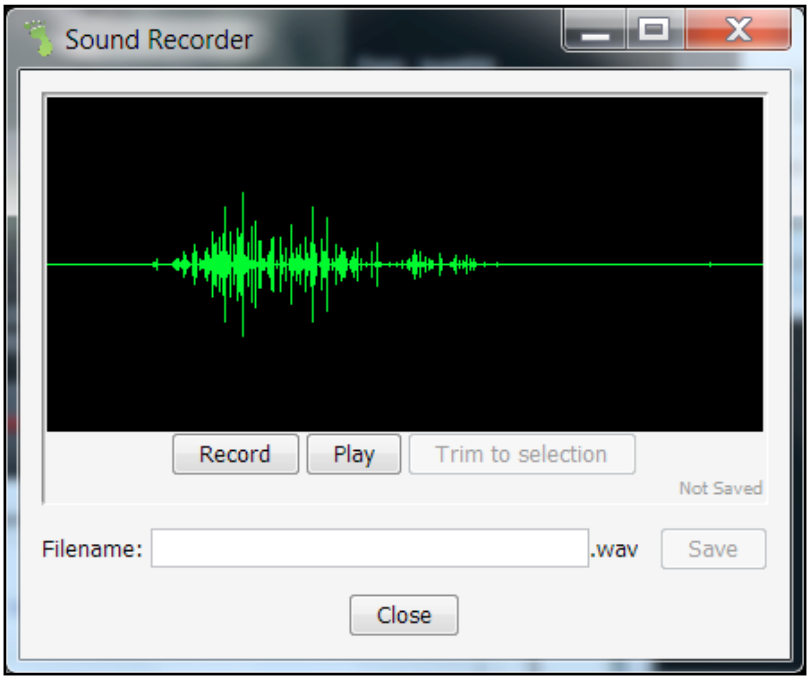
Locating the sound recorder in the Controls menu.



Select that, and you'll get the sound recorder, shown on the next page.

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 using Google 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:

Notes:



Your recorded sound will probably look a bit like this, with the flat line at either end indicating an annoying period of silence.

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.

We're close to finishing our game now, but before we're done, we need to add an enemy!

Notes:

After the last section, we now have a scenario with a crab that we can control, that runs around eating worms. The game is quite easy though – even though the crab is (deliberately) a little tricky to control, there's no tension. What we need is an enemy that eats crabs: a lobster!

To begin with, let's add a lobster that moves in a straight line and eats crabs. We can do that using code that we've already seen how to write. First, we add a "Lobster" class – remember that we do that by right-clicking on the "Actor" class and selecting "new subclass". You'll find the lobster picture in the list of images on the left.

Once you've made your "Lobster" class, you can fill it in by making it move in a straight line and eating a crab if it finds one. You've already seen how to do each of those things in previous tutorials, so I'm not going to paste the code in here. Have a go yourself, but if you get stuck, you can see the answer at this URL:
http://www.greenfoot.org/images/tutorials/crab-4/edit-Lobster-eat.png?1314205634

You can test that the lobster is working by placing one to the left of a crab, then clicking "**Run**" and letting them both run to the right-hand side of the world where the crab will be eaten. (If you want to, make your own sound for when the lobster eats the crab.)

That's great, but our lobster is pretty dumb; it's easy to get away from it by moving sideways, and when it reaches the right-hand edge of the world it stays there forever (just like our original crab did).

Let's make our lobster more difficult to avoid by introducing some randomness. Greenfoot provides a "Greenfoot.getRandomNumber" method that will give you a random number. Here's a first attempt, where we turn a random amount every frame:

Simple code to make the lobster turn randomly.

```
public void moveAround()
{
    move(4);
    turn(Greenfoot.getRandomNumber(90));
}
```

That code means we will turn a random amount each frame, between 0 degrees (inclusive) and 90 degrees (exclusive). Try it out, and see how it works. You'll see that it doesn't create a very threatening enemy: the lobster seems to mostly spin on the spot (turning too often, in effect), and it always turns to the right. Let's fix those problems one by one, starting with the spinning on the spot.

At the moment, our lobster turns every frame, which makes it spin rather than wander around. There's a couple of different ways that we could make it turn less often. One would be to have a counter variable that keeps track of how long it was since we last turned, and turns, say, every 10 frames. Another way is to use the random number generator to turn randomly, with a certain average (e.g. every 10 frames). We'll go with another use of the random generator, as it makes for a less predictable lobster.

Notes:

Let's say that a lobster has a 10% chance of turning each frame. We can code this by comparing "Greenfoot.getRandomNumber(100)" to a given percentage:

Now the lobster will randomly determine whether to turn or not.

```
public void moveAround()
{
    move(4);
    if (Greenfoot.getRandomNumber(100) < 10)
    {
        turn(Greenfoot.getRandomNumber(90));
    }
}
```

If you are interested, think carefully about how this works – why do we use "**<**" (less than) rather than "**<=**" (less than or equal to)? Could we have coded this differently? For example, using "Greenfoot.getRandomNumber(50)" or "Greenfoot.getRandomNumber(10)"? What about "Greenfoot.getRandomNumber(5)"?

That will make our lobster turn (on average) every 10 frames. Compile and run it, and see what happens. The lobster should mostly wander along in a straight line, occasionally turning right by a varying amount. That's great, and it brings us back to our other problem: the lobster always turns right.

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

Simply by subtracting 45, our lobster now turns left as well as right.

```
public void moveAround()
{
    move(4);
    if (Greenfoot.getRandomNumber(100) < 10)
    {
        turn(Greenfoot.getRandomNumber(90) - 45);
    }
}
```

Is our range of turning perfectly symmetric at the moment? If not, how could you fix this?

Compile and run that, and we should have a somewhat effective predator that can turn towards you at any moment. Put a crab, several lobsters and lots of worms into the world (then save the world!), and try to eat all the worms before the lobsters catch you.

You might notice, however, that there is one remaining flaw: the lobsters can get stuck for a time at the edges of the world. This is because once they hit the edge of the world, they'll only move away from it once they've done a few random turns.

We can speed up the process of getting the lobsters off the walls again by making them turn 180 degrees as soon as they reach the edge of the world. We can check for being at the edge of the world by seeing if their X coordinate is close to zero, or close to the width of the world – and a similar logic for the Y coordinate (and the height of the world). The code is on the next page.

The second and third "if" statements check to see if the lobster has reached the edge of the world. If it has, it turns 180 degrees - a U-turn.

```
public void moveAround()
{
    move(4);
    if (Greenfoot.getRandomNumber(100) < 10)
    {
        turn(Greenfoot.getRandomNumber(90) - 45);
    }
    if (getX() <= 5 || getX() >= getWorld().getWidth() - 5)
    {
        turn(180);
    }
    if (getY() <= 5 || getY() >= getWorld().getHeight() - 5)
    {
        turn(180);
    }
}
```

That finishes the crab tutorial. You can continue to experiment with it and add new features (such as a second player), or use your knowledge to make your own new scenario. Have fun!

Useful links

http://www.greenfoot.org/
Greenfoot's homepage. Here, you can upload your completed games or scenarios for others to comment on and play, or if you get stuck with something then you can ask for help on the forums.

http://blogs.kent.ac.uk/mik/category/joy-of-code/
A series of videos teaching programming with Greenfoot from the ground up. Great for if you'd like to learn more about programming or about Greenfoot in general.

Notes: