The Greenfoot Programmers' Manual

Version 1.1

for Greenfoot 2.0

Michael Kölling, Martin Pain
University of Kent

Contents

1 Introduction

Greenfoot is a software tool designed to let beginners get experience with object-oriented

programming. It supports development of graphical applications in the Java™ Programming Language. For tutorials, videos and a quick-start guide for Greenfoot, see the '*Getting Started*' section on http://greenfoot.org/programming/

This manual is an introduction to programming in Greenfoot. It starts from the start: We first discuss how to create a new scenario, then how to create worlds and actors, and so on.

This may not be the order in which you approach your own personal Greenfoot programming experience. In fact, probably the most common way for people to start programming in Greenfoot is by modifying an existing scenario. In that case, you already have a scenario, a world, and one or more actor classes.

Feel free to jump right into the middle of this manual and start reading there. You may, for instance, be interested in generating actor images in a certain way, or in dealing with collisions. The secitons in this manual were written with the goal that they can be read independently - there is no strong need to read everything in order.

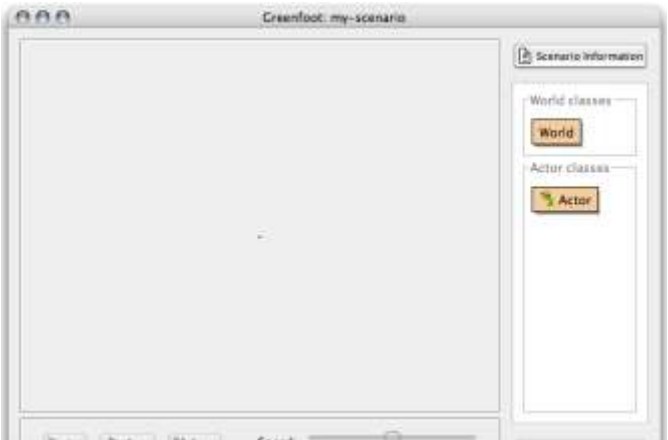
Every now and then, when it is useful to refer to examples, we will use the 'wombats', 'ants', 'balloons' and 'lunarlander' scenarios as examples. All these scenarios are included in the standard Greenfoot distribution. You can find them in the 'scenarios' folder.

Have fun!

2 Creating a new scenario

The first thing to do when you want to create your own program is to create your own scenario.

Doing this is easy: choose 'New' from the 'Scenario' menu, and select a location and name for your scenario. Greenfoot will create a folder with that name that contains all files associated with your scenario.



see a screen similar to the one on the left. That is: you have the 'World' and 'Actor' classes in the class display, and not much else.

Both 'World' and 'Actor' are abstract classes - that is: you cannot create any objects of them. Especially, there is no world object at the moment, because we have no finished world class to make a world object from.

As a result, even if we had an actor object now, we could not place it anywhere, because we cannot create a world.

In order to progress from here, we need to create subclasses (that is: special cases) of World and of Actor. In other words: we have to define our own world, and then we have to define one or more of our own actors.

That's what we'll start in the next section.

On the Grenfoot website there is a tutorial video available that shows how to create and set up a new scenario.

3 Using the API

When programming with Greenfoot, it is essential to know about the methods available in the standard Greenfoot classes. The available methods are known as the **Greenfoot API** (for "Application Programming Interface"), and they are available from the Greenfoot web site.

You can select the 'Greenfoot Class Documentation' option from Greenfoot's Help menu (right) to open the API documentation in a web browser, or double-click on the *World* or *Actor* classes. The API is distributed with Greenfoot, so you do not need to be connected to the Internet to view it.

Greenfoot provides five classes that you should know about. They are World, Actor, Greenfoot, GreenfootImage and MouseInfo.

The World and Actor classes serve as superclasses for our own world and actor classes - we have seen them in the class display.

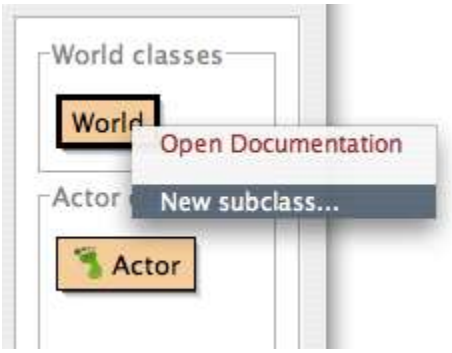
'GreenfootImage' is a class that provides images and image drawing methods for us to use with our worlds and actors.

'Greenfoot' is a class that gives us access to the Greenfoot framework itself, such as pausing the execution or setting the speed.

'MouseInfo' is a class that provides information on mouse input, such as the co-ordinates of where the mouse was clicked and what actor was clicked on.

All of these classes will be mentioned more below. While you are programming in Greenfoot, it is often a good idea to have the API available to you, either printed out or in a web browser window.

4 Creating a world



To create a new world, select 'New subclass...' from the World class's popup menu (see right).

After choosing a name for your new world and clicking Ok, you should see the new world in the class display. Your new world class will have a skeleton that is valid and compiles.

You can now compile your scenario, and you will notice that the new world is automatically visible in the world view. This is one of the built-in features of Greenfoot: whenever a valid world class exists, Greenfoot will, after every compilation, create one world object and show it in the world display.

In theory, you could have multiple world subclasses in a single Greenfoot scenario. It is non-deterministic, though, which of those world will then be automatically created. You could manually create a new world using the world's constructor, but in practice we usually have only a single world subclass in every scenario.

World size and resolution

Let us have a closer look at the code inside the new world class. The default skeleton looks something like this:

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

public class MyWorld  extends World
{

    /**
     * Constructor for objects of class MyWorld.
     *
```

```
     */
    public MyWorld()
    {
        // Create a new world with 20x20 cells with a cell size of 10x10
pixels.
        super(20, 20, 10);
    }
}
```

The first things we have to decide are the size of our world and the size of its cells. There are essentially two kinds of worlds: those with large cells, where every actor object occupies only one single cell (that is: the cell is large enough to hold the complete actor), and those with small cells, where actors span multiple cells.

You have probably seen examples of both. *'wombats'*, for example, is one of the first kind, whereas *'ants'* is of the second kind. (The *wombats* and *ants* scenarios are distributed as examples together with Greenfoot. If you have not looked at them, and are not sure what we are discussing here, it would be a good idea to look at them now.)

We can specify the world size and cell size in the *super* call in the constructor. Assume we change it to

```
    super(10, 8, 60);
```

In this case, we will get a world that is 10 cells wide, 8 cells high, and where every cell is 60x60 pixels in size. The signature of the World class constructor (which we are calling here) is

```
    public World(int worldWidth, int worldHeight, int cellSize)
```

All positioning of actors in the world is in terms of cells. You cannot place actors between cells (although an actor's image can be larger than a cell, and thus overlap many cells).

Large cell / small cell trade-off

The trade-off to be made when choosing a world's cell size is between smooth motion and ease of collision detection.

One result of the fact that actors can only be placed in cells is that worlds with large cells give you fairly course-grained motion. The wombats scenario, for instance uses a 60-pixel sized cell, and thus, when wombats move one step forward, their image on screen moves by 60 pixels. On the other hand, in worlds with large cells, where the actors are completely contained inside a cell, detecting other actors at one actor's position is a bit simpler - we do not need to check for overlapping images, but simply for the presence of other actors in the same cell. Finding actors in neighbouring cells is also easy.

We will discuss this in more detail in the section 'Detecting other objects (collisions)', below.

If you want smoother motion, you should use a smaller cell size. the 'ants' scenario, for example, uses 1-pixel cells, giving the ants the ability to move in small steps. This can also be combined with large actor objects. The 'lunarlander' scenario, for instance, uses a large actor (the rocket) on a 1-pixel cell world. This gives the actor very smooth motion, since it's location can be changed in one-pixel intervalls.

World background images

Most of the time, we want our world to have a background image. This relatively easy to do.

First, you have to make or find a suitable background image. There are a number of background images distributed with Greenfoot. You can set one of these as the world's background either when you create the class, or by selecting 'Set image...' from the class's popup menu. Select the 'backgrounds' category from the categories box, select an image and click Ok.

There are more background images available in the Greenfoot Image Collection on the Greenfoot web site.To use a custom image, such as one from the Greenfoot Image Collection or one you have created yourself, place the image file into the 'images' folder inside your scenario folder. Once the image file is there, it is available to your Greenfoot scenario, and you can select it from the 'Scenario images' box in the 'Set image' or 'New subclass' dialog.

You can also set the background image in the Java code with the line:

```
setBackground("myImage.jpg");
```

where "myImage.jpg" should be replaced with the correct image file name. For example, assume we have placed the image "sand.jpg" into our scenario's 'images' folder, then our world's constructor might look like this:

```
public MyWorld()
{
    super(20, 20, 10);
    setBackground("sand.jpg");
}
```

The background will, by default, be filled with the image by tiling the image across the world. To get smooth looking background, you should use an image whose right edge fits seamlessly to the left, and the bottom to the top. Alternatively, you can use a single image that is large enough to cover the whole world.

If you want to paint the background programmatically you can easily do so instead of using an image file. The world always has a background image. By default (as long as we do not specify anything else) it is an image that has the same size as the world and is completely transparent. We can retrieve the image object for the world's background image and perform drawing operations on it. For example:

```
GreenfootImage background = getBackground();
background.setColor(Color.BLUE);
background.fill();
```

These instructions will fill the entire background image with blue.

A third option is to combine the two: You can load an image file, then draw onto it, and use the modified file as a background for the world:

```
GreenfootImage background = new GreenfootImage("water.png");
background.drawString("WaterWorld", 20, 20);
setBackground(background);
```

Background images are typically set only once in the constructor of the world, although there is nothing that stops you from changing the world's background dynamically at other times while your scenario is running.

Showing tiles

Sometimes, when you have worlds with large grid sizes, you want to make the grid visible. The 'wombats' scenario does this - you can see the grid painted on the background of the world.

There is no special function in Greenfoot to do this. This is done simply by having the grid painted on the image that is used for the world background. In the case of 'wombats', the world has 60-pixel cells, and the background image is a 60x60 pixel image (to match the cell size) that has a one pixel line at the left and top edges darkened a bit. The effect of this is a visible grid when the tiles are used to fill the world background.

The grid can also be drawn programatically onto the background image in the world constructor. Here is an example:

```
    private static final Color OCEAN_BLUE = new Color(75, 75, 255);
    private static final int ENV_SIZE = 12;  // environment size in numer
of cells
    private static final int CELL_SIZE = 40; // cell size in pixels

    ...

    private void drawBackground()
    {
        GreenfootImage bg = getBackground();
        bg.setColor(OCEAN_BLUE);
        bg.fill();
        bg.setColor(Color.BLACK);
        for(int i = 0; i < ENV_SIZE; i++) {
            bg.drawLine(i * CELL_SIZE, 0, i * CELL_SIZE, ENV_SIZE *
CELL_SIZE);
            bg.drawLine(0, i * CELL_SIZE, ENV_SIZE * CELL_SIZE, i *
CELL_SIZE);
        }
   }
```

5 Creating new actors

This section discusses some of the characteristics of actor classes, and what to consider when writing them.

All classes that we want to act as part of our scenario are subclasses of the built-in 'Actor' class. We can create new actor classes by selecting 'New subclass...' from the Actor class's popup menu.

The following dialogue lets us specify a name and an image for our new class. The name must be a valid Java class name (that is: it can contain only letters and numbers). The class image will be the default image for all objects of that class.

Class images

Every class has an associated image that serves as the default image for all objects of that class. Every object can later alter its own image, so that individual objects of the class may look different. This is further discussed in the section 'Dealing with images', below. If objects do not set images explicitly, they will receive the class image.

The image selector in the dialogue shows two groups of images: project images and library images. The library images are included in the Greenfoot distribution, the project images are stored in the 'images' folder inside your scenario (project) folder. If you have your own images that you like to use for your class, you have two options:

- You can copy you image file into the scenario's 'images' folder. It will then be available for selection in this dialogue; or
- You can use the 'Browse for more images' button in this dialogue to select your image from anywhere in your file system. The image will then automatically be copied into the images folder of the scenario.

Initialisation

As with most Java objects, the normal initialisation of the object happens in the object's constructor. However, there are some initialisation tasks that cannot be finished here. The reason is that, at the time the object is constructed, it has not been entered into the world yet. The order of events is:

1. The object is constructed.
2. The object is entered into the world.

During step 1, the object's constructor is executed. Since the object is not in the world at this time, methods such as getWorld(), getX() and getY() cannot be called in the constructor (when you're not in the world, you do not have a location).

So, if we want to do anything as part of our initialisation that needs access to the world (such as create other objects in the world, or set our image depending on neighbouring objects), it cannot be done in the constructor. Instead, we have a second initialisation method, called 'addedToWorld'. Every actor class inherits this method from class 'Actor'.

The signature of this method is

```
public void addedToWorld(World world)
```

This method is automatically called when the actor has been added to the world. So, if we have work to do at the time the object has been added, all we have to do is to define an 'addedToWorld' method in our class with this signature and place our code there. For example:

```
public class Rabbit extends Actor
{
    private GreenfootImage normalImage;
    private GreenfootImage scaredImage;

    public Rabbit()
    {
```

```
        normalImage = new GreenfootImage("rabbit-normal.png");
        scaredImage = new GreenfootImage("rabbit-scared.png");
    }

    public void addedToWorld(World world)
    {
        if(isNextToFox()) {
            setImage(scaredImage);
        }
         else {
            setImage(normalImage);
        }
    }

    private boolean isNextToFox()
    {
        ... // calls getWorld() to check neighbours in world
    }
}
```

In this example, the intention is that a rabbit, when placed into the world, looks 'normal' if it is not next to a fox, but looks scared when placed next to a fox. To do this, we have two image files ("rabbit-normal.png" and "rabbit-scared.png"). We can load the images in the Rabbit's constructor, but we cannot select the image to show, since this involves checking the world, which is not accessible at the time the constructor executes.

When a user places an object into the world, things happen in this order:

1. The object is created (and the constructor is executed).
2. The object is placed into the world.
3. The *setLocation* method of the object is called with its new location.
4. The *addedToWorld* method of the object is called.

In our example above, by the time the *addedToWorld* method is called, the object is in the world and has a location. So we can now call our own *isNextToFox()* method (which presumably makes use of the world and our location).

The *setLocation* method will be called every time the location of the object changes. The *addedToWorld* method is called only once.

The 'Lander' class in the 'lunarlander' scenario (from the Greenfoot sample scenarios) shows another example of using this method.

6 Making things move

Every time a user clicks the 'Act' button on screen, the 'act' method of each object in the world will be called. The 'act' method has the following signature:

```
public void act()
```

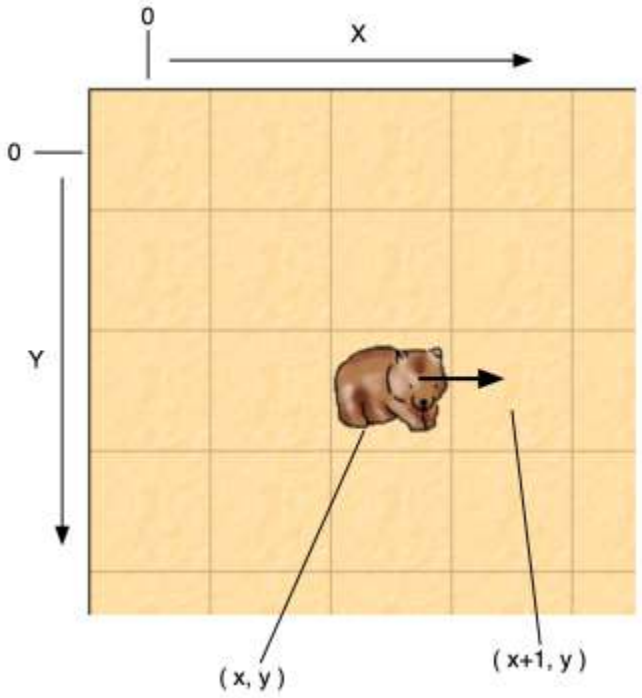
Every object that is active (i.e. is expected to do something) should implement this method.

The effect of a user clicking the 'Run' button is nothing more than a repeated (very fast) click on the 'Act' button. In other words, our 'act' method will be called over and over again, as long as the scenario runs.

To make object move on screen, it is enough to modify the object's location. Three attributes of each actor become automatically and immediately visible on screen when you change them. They are:

- the location (given as x and y coordinates)
- the rotation
- the image

If we change any of these attributes, the appearance of the actor on screen will change. The Actor class has methods to get and set any of these.



Changing the location

The first thing to look at is location changes. Consider, for example, the following code:

```
public void act()
{
    int x = getX();
    int y = getY();
    setLocation(x + 1, y);
}
```

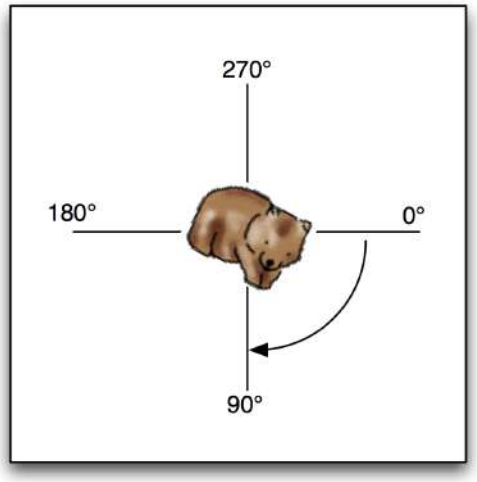
The effect of this code fragment is to move the actor one cell to the right. It does this by getting the actor's current x and y coordinates, and then setting a new location for the actor with the x-coordinate increased by one.

We can write the same code a little shorter as

```
public void act()
{
    setLocation(getX() + 1, getY());
}
```

Once our actor gets to the end of the world it keeps trying to move outside it, but Greenfoot will not let it. To fix this, we need to add code that checks whether we can move in that direction before actually moving. The 'wombats' scenario shows an eample of this.

The location coordinates are the indices of the cells in the world. They should not exceed the world size. The (0,0) location is in the top left of the world, and coordinates increase right (x) and down (y).



Changing the rotation

In a similar manner to the location, we can change the rotation of the object's image. Here is an example:

```
public void act()
{
    int rot = getRotation() + 1;
    if(rot == 360) {
        rot = 0;
    }
    setRotation(rot);
}
```

This method gets the object's current rotation, and then increases it by one. The effect is that the object will slowly rotate clockwise.

The valid range for the rotation is [0..359], and the angle increases clockwise. Thus, in the code example above, we check whether we have reached 360 (that is: left the valid range) and then reset the value to 0.

Changing the image

The last of the actor attributes that is automatically visualised is the actor's image. Changing the image will become immediately visible on screen. Consider this:

```
public void act()
{
    if(hasEaten()) {
        setImage("happy.jpg");
    }
    else {
        setImage("sad.jpg");
    }
}
```

This code assumes that we have a hasEaten() method in our code, and then sets the image accordingly. There are two versions of the setImage method: one expects a file name of an image file as a parameter, the other one expects an object of type GreenfootImage.

In general, it is a good idea to load image file only once into a GreenfootImage object, and then to reuse the same image object if you need to set the image multiple times. For instance, instead of calling

```
setImage("happy.jpg");
```

repeatedly in the act method, you could initialise an instance field with the image:

```
private GreenfootImage happyImage = new GreenfootImage("happy.jpg");
```

and then set the image using this object:

```
setImage(happyImage);
```

More information about dealing with images is in the section 'Dealing with images' (below).

There is a tutorial video about making actors move available on the Greenfoot website.

7 Random behaviour

Random behavior in Greenfoot scenarions is based on random numbers.

Generating random numbers in Greenfoot is fairly easy. Greenfoot has a built-in class called 'Greenfoot' that is part of the framework (see The Greenfoot class, below). This class has a method called *getRandomNumber*. Its signature is

```
public int getRandomNumber(int limit)
```

In our own code, whenever we need a random number, we can call this method:

```
int myNumber = Greenfoot.getRandomNumber(10);
```

This example will give us a number in the range [0..9]. That is: the number is always between zero (inclusive) and the limit you specify (exclusive). For details, see the description of the Greenfoot class in the Greenfoot API.

Once you have random numbers, using these for random behaviour is only a small step. For example:

```
if(Greenfoot.getRandomNumber(2) == 0) {    // 50% chance
    turnLeft();
}
else {
    turnRight();
}
```

For more examples, see the 'Wombat' class in the 'wombats2' scenario, or the 'Ant' class in 'ants'.

8 Dealing with images

Greenfoot supports various different ways that objects can acquire images.

Objects can get an image by using one of these three ways, or a combination of them:

1. using default images from their class;
2. loading image files from disk;
3. containing code to paint an image.

All three method are used by the various objects in the *ants* scenario that is included in the Greenfoot distribution. It is useful to study this example if you want to learn about images. We will refer to this scenario repeatedly in this section.

We discuss all three methods in turn. There are also tutorial videos available on the Greenfoot website showing how to make background images using these methods.

Using default class images

Every class has an associated image. This image is usually assigned when creating the class, but may be changed later using the 'Set Image...' function from the class's popup menu.

For an object to use the class's image, there is nothing we need to do. If we write no special image handling code, this is the image that will be used to display objects of this class.

In the *ants* project, the *AntHill* objects use this technique. A fixed image is assigned to the class, and all AntHill objects look the same.

Using image files

We can easily alter the image of an individual object by using the Actor's

'setImage(..)' method. In the *ants* scenario for example, the Ant class uses this method. When an ant finds some food, its takeFood() method is executed, which includes the line

```
setImage("ant-with-food.gif");
```

When the ant drops the food (in the dropFood() method) it uses this line:

```
setImage("ant.gif");
```

This way, the image of each individual ant object can dynamically change.

When this 'setImage' method is used, an image file is loaded from disk. The parameter specifies the file name, and the file should be located in the project's 'images' folder.

If images of objects change frequently, or have to change quickly, this can be optimised by loading the image from disk only once, and storing them in an image object (of type GreenfootImage). Here is a code snippet to illustrate this:

```
public class Ant extends Actor
{
    private GreenfootImage foodImage;
    private GreenfootImage noFoodImage;

    public Ant()
    {
        foodImage = new GreenfootImage("ant-with-food.gif");
        noFoodImage = new GreenfootImage("ant.gif");
    }

    private boolean takeFood()
    {
        ...
        setImage(foodImage);
    }

    private boolean dropFood()
    {
        ...
        setImage(noFoodImage);
    }
}
```

This example illustrates a second version of the setImage method: setImage can also be called with a GreenfootImage as a parameter, instead of a file name. The GreenfootImage can be constructed using the image file name, and the resulting object can then be reused.

This version saves resources and executes quicker. It is preferrable whenever the images change frequently.

The file names of image files are case-sensitive in Greenfoot. For example, "image.png", "Image.png" and "image.PNG" are all different. You have to use the correct case or Greenfoot will not be able to find your image.

Using generated images

Images can be generated at run-time by code included in your class. This approach can be useful when there will be many variations of the image, and they can be drawn simply. In the *ants* scenario the *Food* class uses this approach so it can show how many pieces of food remain in the pile and the *Pheromone* class uses it to show its intensity decreasing.

Creating an image object

To start generating an image, you need a GrenfootImage object to work with. This can be a blank image, an image from a file, or the default class image. To create a blank image, you need to pass the width and height of the desired image to the GreenfootImage constructor. The image will be completely transparent.

```
GreenfootImage image = new GreenfootImage(60, 50);
    //Creates an image 60 pixels wide and 50 pixels high
```

If you are using a GreenfootImage which already exists you may want to create a copy of the image to draw on, so that the original is preserved.

```
GreenfootImage image_copy = new GreenfootImage(image);
```

Drawing on an image

Once you have your image to draw on, whether it be blank or already contain a base image, you can use some of the GreenfootImage class's methods to add to the image. The GreenfootImage class provides methods that let you:

- Draw straight lines, rectangles, ovals (including circles) and polygons
- Draw filled rectangles, ovals and polygons
- Set the colour of individual pixels
- Fill the entire image with a single colour
- Draw a string of text on the image
- Copy another GreenfootImage onto the image
- Scale, rotate and mirror the image
- Set the transparency of the image

Setting the colour

Before using any of the drawing methods, you need to tell the image what colour you want to draw in. To do this, call the GreenfootImage object's setColor() method. This takes a Color object from Java's library. The easiest way to get a Color object is to use one of the pre-defined constants in the Color class, such as *Color.BLACK*. Other colours available include white, gray, red, green, blue, yellow and orange. See the Java API documentation of the Color class for a full list of pre-defined colours and other ways of getting Color objects.

As we are using a class from the Java library we have to say where to find it. To do that, we add an 'import' statement to the top of the class file. There is already one there which says where to find the Greenfoot classes. We add another line importing the Color class, whose fully-qualified name is *java.awt.Color*.

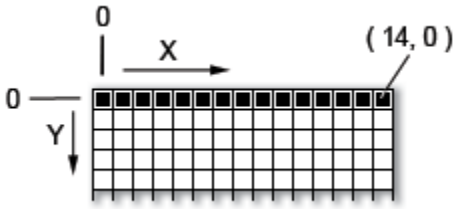
```
    import greenfoot.*;  // (World, Actor, GreenfootImage, Greenfoot and
MouseInfo)
    import java.awt.Color;
```

Then we can go ahead and set the colour we want to draw with, in whichever method we are generating the image:

```
    image.setColor(Color.BLACK);
```

Drawing a line

To draw a line on the image, use the drawLine() method. It takes four parameters, the first two represent the x and y co-ordinates for the start of the line, and the last two are the x and y co-ordinates of the end of the line. The co-ordinates start at (0,0) at the top-left of an image, and are measured in pixels. This is similar to how actors are placed in the world, except that the world uses cells which can be larger than one pixel, and GreenfootImage always measures in pixels.



The following code will draw a line 15 pixels long across the top of an image:

```
    image.drawLine(0,0, 14,0);
```

The line starts at (0,0) which is the top-left of the image, and ends at (14,0). The x co-ordinate of the 15th pixel is 14, as the first pixel is numbered 0. If you were drawing a line across the entire top of an image, the code would be:

```
    image.drawLine(0,0, image.getWidth()-1,0);
```

This code will draw a line from the top-left of an image (0,0) to halfway down the right-hand side (59,25) (of a 60x50 pixel image):

```
    image.drawLine(0,0,59,25);
```

Drawing circles, rectangles and other shapes

Rectangles can be drawn using the drawRect method. The method takes four parameters: the x and y co-ordinates of the top-left corner of the rectangle, its width and its height. The following code would draw a rectangle 60 pixels wide by 30 pixels high in the top-left corner of the image:

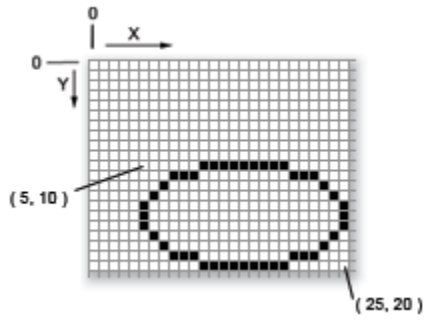
```
    image.drawRect(0,0,60,30);
```

The rectangle covers an area that is *width + 1* pixels wide and *height + 1* pixels tall. This means if you want to draw a rectangle around the edge of an image, use the following code:

```
    image.drawRect(0,0,image.getWidth()-1,image.getHeight()-1);
```

Circles and ovals can be drawn by calling the drawOval method, specifying the co-ordinates of the

top-left corner of the area that the oval is to be drawn in, and the width and height of the oval. Unlike with rectangles, the co-ordinates given will not be on the line which is drawn: the left of the circle will be at the x co-ordinate, and the top of the circle will be at the y co-ordinate.



To draw an oval 20 pixels wide and 10 pixels tall at position (5,10) you would use the following parameters:

```
image.drawOval(5,10,20,10);
```

Like rectangles, ovals cover an area that is *width + 1* pixels wide and *height + 1* pixels tall.

You can draw more complicated polygons using the drawPolygon method, which takes an array of x co-ordinates, an array of y co-ordinates and the number of points the shape has. Lines are drawn between each set of co-ordinates in turn, and then from the final point to the first point (the polygon is closed).

For each of these three methods there is also 'fill' version, which draws a filled-in shape, rather than just the shape's outline. The fill uses the same colour as the outline, which is the colour most recently passed to the image's setColor method. To draw a shape with different fill & outline colours, call the fill version of the method with one colour, then the outline method using a different color:

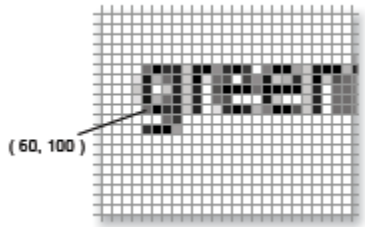
```
image.setColor(Color.BLACK);
image.fillRect(5,10,40,20);
image.setColor(Color.RED);
image.drawRect(5,10,40,20);
```

Writing in the images

You can use the *drawString* method to put text into an image. The method takes the string to be drawn, and the x and y position of the baseline of the first character. This means that the point specified will almost be the bottom-left corner of the area where the text is drawn, except that the bottom of letters such as *p* and *q* will fall below that point.

You can set the font using the *setFont* method, passing it a *Font* object. The easiest way to get a font object with a specific text size is to use the *deriveFont* method on another font object. You can get the current font that the image is using from the *getFont* method. To store a Font object in a variable, we first need to add an import statement to the top of the class to say where the Font object is:

```
import java.awt.Font;
```



The code below (taken from the *ScoreBoard* class in the *balloons*

example) can then be used to get a font whose size is 48 points:

```
Font font = image.getFont();
font = font.deriveFont(48);
image.setFont(font);
```

That font will then be used for any following *drawString* method calls:

```
image.drawString(title, 60, 101);
```

To get a font with a certain style, such as serif/sans-serif or bold/italic/underlined, see the Font class's constructors in the Java Library Documentation for the Font class.

There is a tutorial video about this available on the Greenfoot website.

Copying an image onto other images

GreenfootImage provides a method to draw one image onto another image. This can be useful if you want to add an icon to an actor's image to show something special about that actor, such as if it's carrying something. To draw one image onto another, simply call the drawImage method on the image to draw to, passing the image to copy from and the co-ordinates that the image should be placed at.

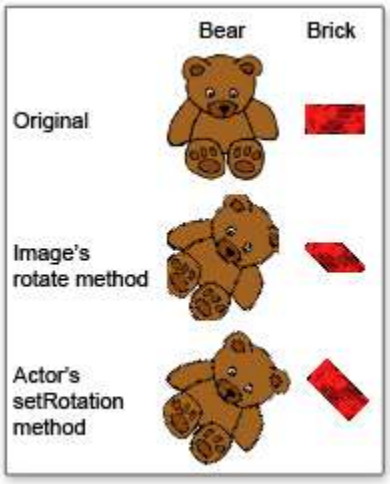
```
image.drawImage(new GreenfootImage("smaller-image.png"), 10, 10);
```

Scaling, mirroring & rotating images

Images can be scaled (stretched or compressed), mirrored vertically or horizontally, and rotated.

The *scale* method takes two integers as parameters, which represent the width and height that you want the image to be. The image will then be stretched (or made smaller) to fit that size.

To mirror an image, use one of the *mirrorVertically* or *mirrorHorizontally* methods. They take no parameters and flip the image along the appropriate axis.

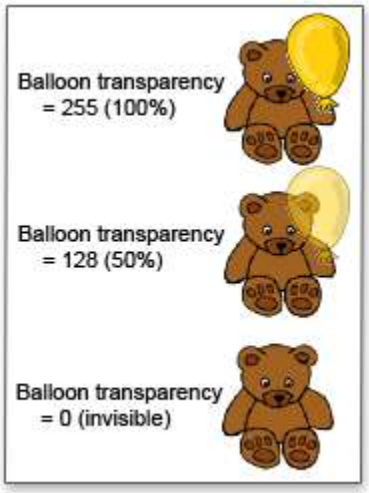


Images can be rotated using the *rotate* method, which takes the number of degrees to rotate the image by. Note that if you rotate the image by anything which is

not a multiple of 90 degrees then what was in the corners of the image will be cut off, as the image will still have horizontal and vertical edges.

If the image is to be used as an actor's image, then usually rotation is better achieved by using the Actor object's *setRotation* method, as it just displays the image at a different rotation, rather than changing the image itself. Also, the Actor's *setRotation* method remembers its rotation, which can be used to work out which direction the actor should move in, whereas the GreenfootImage objects' *rotate* method just rotates the image the specified number of degrees from whatever rotation it is at the moment.

Transparency



The image can be made partially transparent by using the *setTransparancy* method. This will allow you to see other objects and the world's background through the image. The method takes one parameter, which should be a number between 0 and 255, where 0 means completely invisible and 255 means completely visible.

The image will not be able to be seen by the user at low numbers; with a patterned background it can get difficult to see anything with a transparency up to about 40. However, the actors will still be found by the collision detection methods. If there is an actor which can be collided with, which the user is suposed to avoid, and which is fading out using transparency, it is a good idea to remove it from the world when the transparency gets low, but before it reaches zero.

Dealing with individual pixels

You can set the colour of any specific pixel by using the *setColorAt* method, passing the x and y co-ordinates of the pixel and the colour to set it to.

The '*Food*' class in the '*ants*' example scenario uses this method. It loops through all the pieces of food left in the pile, and for each one draws four pixels onto the image next to each other in a random position. The colours of the pixels were chosen to give a 3D effect on the pieces of food.

```
GreenfootImage image = new GreenfootImage(SIZE, SIZE);
for (int i = 0; i < crumbs; i++) {
    int x = randomCoord();
    int y = randomCoord();
    image.setColorAt(x, y, color1);
```

```
            image.setColorAt(x + 1, y, color2);
            image.setColorAt(x, y + 1, color2);
            image.setColorAt(x + 1, y + 1, color3);
        }
        setImage(image);
```

9 Detecting other objects (collisions)

One of the really nice features in the Greenfoot API is the ability to find other objects in the world. As soon as you want objects to interact with each other, you need to be able to "see" these other objects. Greenfoot gives you many different ways to find other objects to suit many different kinds of scenarios. We have divided the methods into two different categories: one that is strictly based on the location of objects, and one that is using the image representation of the objects. The methods discussed here are all available when sub-classing the Actor class.

Cell based

In some scenarios, like Wombats, objects are always entirely contained within a cell, and you are only interested in the location of the object in the grid. For these scenarios we have methods that works strictly on the location of the objects. We call these methods for *cell based*.

When wombats are looking for leaves to eat, they look at where they are right now, to see if there is a leaf. In the foundLeaf() method in the Wombat the code to do this is:

```
Actor leaf = getOneObjectAtOffset(0, 0, Leaf.class);
```

This method returns one object at a relative location to where the wombat is currently. The first two parameters specify the offset from the current location, which in this case is (0,0), since wombats can only eat leaves where they are right now. The third parameter specifies which types of objects we are looking for. The method will then only return objects of the given class or sub-class. If several objects exist at the location, it is undefined which one will be returned.

If you have a really big wombat that can eat several leaves at once you can use another method which will return a list of leaves:

```
List leaves = getObjectsAtOffset(0, 0, Leaf.class);
```

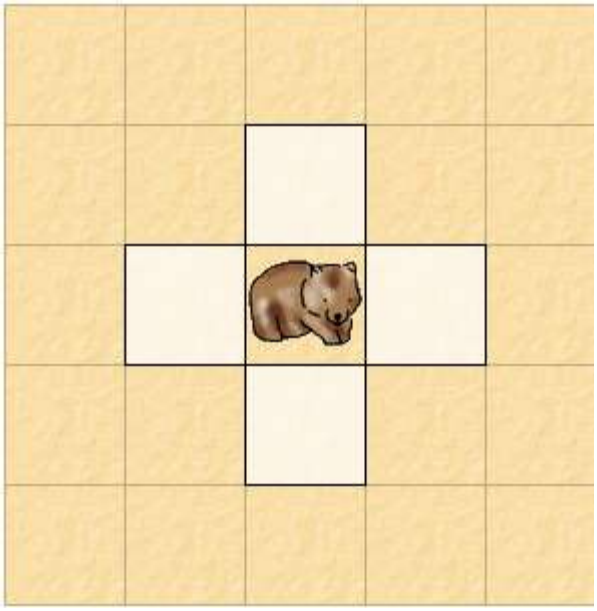
The wombats, as they are implemented in the wombats scenario, are pretty dumb and do not make use of any of the senses that real wombat has. If we want the wombat to look around for leaves before moving we have several methods to choose from.

If the wombat should only be able to look at the immediate neighbours to the north, south, east and west we can use the following methods

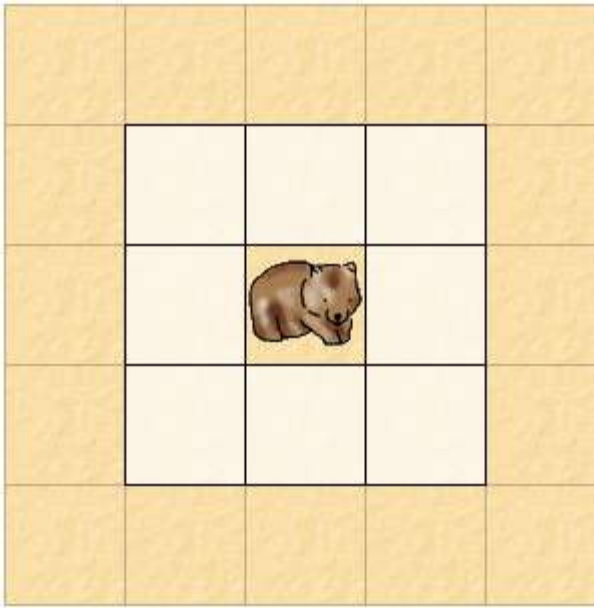
```
List leaves = getNeighbours(1, false, Leaf.class);
```

That call will find all objects within a walking distance of 1 cell, where you are not allow to go diagonally. If you wanted the diagonals included you should replace *false* with *true*. If the wombat should be able to look farther you can increase the distance from 1 to something larger.

Below we have made a few pictures to illustrate what cells will be considered when looking for neighbours with different parameters for the method call.



Without the diagonal:
getNeighbours(1, false, Leaf.class);

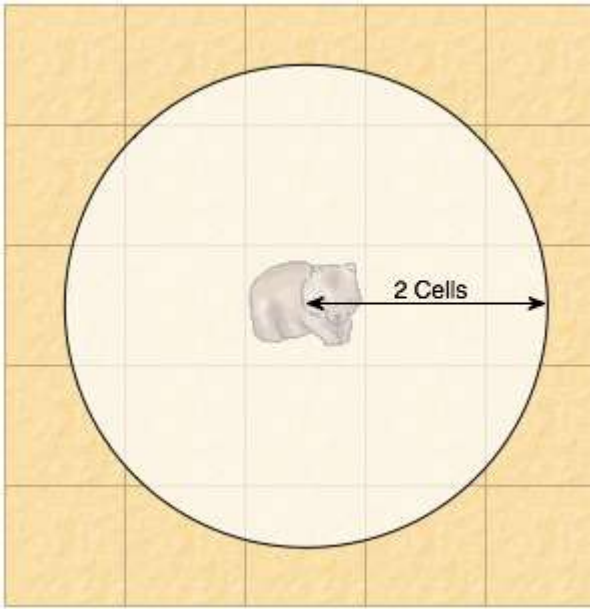


With the diagonal:
getNeighbours(1, true, Leaf.class);

A method related in functionality to the getNeighbours methods:

```
List leaves = getObjectsInRange(2, Leaf.class);
```

This method call will return all objects (of the class Leaf and subclasses) that have a location that is within 2 cells. If the distance between two actors is exactly 2, it is considered to be in range. See the picture below for an illustration of this



```
getObjectsInRange(2, Leaf.class);
```

Representation based

Sometimes it is not precise enough to use the cell location to determine collisions. Greenfoot has a few methods that allow you to to check whether the graphical representations of actors overlap.

```
Actor leaf = getOneIntersectingObject(Leaf.class);
List leaves = getIntersectingObjects(Leaf.class);
```

Be aware that these method calls require more computation than the cell based methods and might slow down your program if it contains many actors.

10 Keyboard input

There are two ways that the keyboard can be used in Greenfoot scenarios: holding a key down for a continuous action, or pressing a key for a discrete action.

Keys on the keyboard, when being passed to or returned from a method, are referred to by their names. The possible names are:

- "a", "b", .., "z" (alphabetical keys), "0".."9" (digits), most punctuation marks
- "up", "down", "left", "right" (the cursor keys)
- "enter", "space", "tab", "escape", "backspace"
- "F1", "F2", .., "F12" (the function keys)

Continuous actions: holding a key down

The *lunarlander* example scenario uses this method: while the 'down' cursor key is pressed the lander's thruster is fired. This is achieved using the isKeyDown() method of the Greenfoot class. In an actor's act method you can call this method with a key's name as the parameter, and it will return *true* if that key is currently being held down. You can use this in the condition of an if-statement to make the actor act differently or look different, depending on whether the key is down or not:

```
if(Greenfoot.isKeyDown("down")) {
    speed+=thrust;
    setImage(rocketWithThrust);
} else {
    setImage(rocket);
}
```

In this method, when referring the the alphabetical keys either the uppercase or lowercase letters can be used as the parameter to the method.

Single actions: hitting a key

The getKey() method of the Greenfoot class can be used to get the name of the most recently pressed key since the previous time the method was called. This can be used for single actions

where the key is to be hit once, rather than held down, such as a 'fire bullet' action:

```
if(Greenfoot.getKey().equals("space")){
    fireGun();
}
```

When comparing strings, unlike comparing numbers, it is much better to use the string's *equals* method than to use the == equality operator, as strings are objects (like actors are objects) and the == equality operator, when used on objects, tests to see if they are the same object, not if they have the same contents.

Note that as the method returns the last key pressed down since the last time it was called, if the method has not been called for a while then the key returned might not have been pressed recently - if no other key has been pressed since. Also, if two keys are pressed at once, or close together between two calls of *getKey*, then only the last one pressed will be returned.

11 Mouse input

You can test if the user has done anything with the mouse using these methods on the Greenfoot class: *mouseClicked, mouseDragged, mouseDragEnded, mouseMoved* and *mousePressed*. Each of these methods take an object as a parameter. Usually this will be an Actor to find out if the mouse action (click, drag over, move over, etc) was on or over that actor. The parameter can also be the world object, which finds out if the mouse action was over an area where there were no actors, or *null* which finds out if that mouse action happened at all irrespective of what actor it was over.

If the appropriate mouse action has been performed over the specified actor or world then the method returns true, otherwise it will return false. If the mouse action was over a number of actors, only the top one will return true. These methods will only return true for the world if the mouse action was not over any actors at all.

If the methods are called from an actor's *act* method then usually the parameter will be that actor object, using the *this* keyword. For example, an actor could use the following code to change its image when it is clicked. *clickedImage* would be a field containing a GreenfootImage object, but the *this* keyword is not a variable and so does not need to be defined anywhere.

```
if(Greenfoot.mouseClicked(this)){
    setImage(clickedImage);
}
```

The *mousePressed* method is used to find out if the mouse button was pressed down, whether or not it has been released again yet. *mouseClicked* is used to find out if the mouse has been released after it has been pressed. That is, it finds out if the mouse has been pressed and released. *mouseMoved* is used to find out if the mouse has been moved over the specified object (while the button is not pressed down). *mouseDragged* is used to find out if the mouse has been moved over the object with the mouse button pressed down. *mouseDragEnded* is used to find out if the mouse has been released after having been dragged.

Additional information can be acquired from a *MouseInfo* object returned from the *getMouseInfo* method. Using this object you can find out the actor that the action was on or over (if any), the button that was clicked (i.e. left or right mouse button), the number of clicks (single or double), and the x and y co-ordinates of the mouse.

This information is used by the *balloons* example scenario. The *mouseMoved* method (using *null*

as the parameter as the move does not have to be over a specific actor) and the *getX* and *getY* methods of the *MouseInfo* object are used to move the dart around:

```
if(Greenfoot.mouseMoved(null)) {
    MouseInfo mouse = Greenfoot.getMouseInfo();
    setLocation(mouse.getX(), mouse.getY());
}
```

The *mouseClicked* method is used to pop the balloons. The *getActor* method of the *MouseInfo* class is not used in this case, as it will always return the dart, and the point at which we want to test for a balloon is not where the mouse is, it is at the tip of the dart.

A tutorial video about handling mouse input is available on the Greenfoot website.

12 Playing audio

Playing a sound in Greenfoot is extremely simple. Copy the sound file into your scenario's *sounds* folder.

In your code create a GreenfootSound object using the GreenfootSound constructor with the name of the sound file

```
GreenfootSound sound=new GreenfootSound ("explosion.wav");
```

and call the sound's *play* method in order to play the sound

```
sound.play();
```

For example, an explosion sound is played by the following line of code:

```
GreenfootSound sound=new GreenfootSound("explosion.wav");
sound.play();
```

The sound can be stopped and paused by using the stop() and pause() methods respectively

```
sound.stop();
sound.pause();
```

The sound can be played in a loop using the playLoop() method

```
sound.playLoop();
```

Finally, the user can check whether a sound is playing via the isPlaying() method which returns true if the sound is playing and false if not.

```
sound.isPlaying();
```

The audio file must be a wav, aiff, au, mp3 or midi file. If a wav file does not play, it should be converted to a "Signed 16 bit PCM" wav file. This can be done with many audio programs, for example the excellent and free "Audacity" has an *Export* function that will do the conversion.

Sound file names are case-sensitive (i.e. "sound.wav", "Sound.wav" and "sound.WAV" are all different); if you use the wrong case you will get an *IllegalArgumentException* as the sound will

not be able to be played.

13 Controlling the scenario

The *Greenfoot* class provides methods to control the execution of the scenario, and the *World* class provides methods to control how the scenario looks, as well as to react to the scenario stopping or starting. The *Greenfoot* class allows you to stop, start, pause (delay) and set the speed of the action (as well as providing the mouse, keyboard & sound methods mentioned earlier). The *World* class allows you to set the order in which the actors are painted on the screen, set the order in which the actors have their *act* method called, respond to the scenario being started and respond the the scenario being stopped.

Stopping the scenario

The *Greenfoot.stop()* method allows you to stop the scenario, which is usually done when everything has finished.

```
if(getY() == 0){
    Greenfoot.stop();
}
```

Calling the *stop* method is equivalent to the user pressing the *Pause* button. The user can then press *Act* or *Run* again, but in the example code above as long as the if-statement's condition returns true every time it is called after the scenario should stop then pressing the *Run* button again will have no effect.

Setting the speed

You can set the speed of execution using the *Greenfoot.setSpeed()* method. It is good practice to set the suggested speed in the World's constructor. This following code sets the speed at 50%:

```
Greenfoot.setSpeed(50);
```

Delay

The *Greenfoot.delay()* method allows you to suspend the scenario for a certain number of steps, which is specified in the parameter. Note that the length of time that the scenario will be delayed for will vary a great deal, based on the speed of execution.

```
Greenfoot.delay(10); //Pause for 10 steps
```

All actors will be delayed when the *delay* method is called; if you want some actors to continue moving, or may in the future want to add an actor that continues moving during that time, then it would be better to have a boolean field or method that is *false* when you want to delay and *true* the rest of the time. Then all the actors that should stop can check at the beginning of their *act* methods:

```
public void act(){
    if(!isPaused()){
        ...
    }
}
```

Starting the scenario

The *Greenfoot.start()* method is very rarely used, as most of the code that is written in scenarios is only executed after the scenario has been started. However, while the scenario is stopped the user can call methods of the actors in the world form their popup menus. In this situation, calling *Greenfoot.start()* from within a method will make sure that the scenario starts running if the method is called from the popup menu.

For example, in the *balloons* scenario you can start the scenario, allow some balloons to appear, pause the scenario and then call the *pop* method on each balloon. If you added the following code to the *pop* method then the scenario would re-start as soon as a user tried to pop a balloon through the popup menu.

```
public void pop()
{
    Greenfoot.start();
    ...
}
```

Setting the order actors are painted in

The *setPaintOrder* method of the *World* class sets the order in which the actors are painted onto the screen. Paint order is specified by class: objects of one class will always be painted on top of objects of some other class. The order of objects of the same class cannot be specified. Objects of classes listed first in the parameter list will appear on top of all objects of classes listed later. Objects of a class not explicitly specified effectively inherit the paint order from their superclass.Objects of classes not listed will appear below the objects whose classes have been specified.

This method is usually only called from the constructor of your subclass of *World*, such as in the *ants* scenaro:

```
    setPaintOrder(Ant.class, Counter.class, Food.class, AntHill.class,
Pheromone.class);
```

where objects of the *Ant* class would always appear above (that is, completely visible when they overlap) actors of other classes.

To call the method from within an actor class, you would first have to get the current world object using the *getWorld()* method:

```
   getWorld().setPaintOrder(MyActor.class, MyOtherActor.class);
```

Setting the order actors act in

The *setActOrder* method of the *World* class sets the order in which the actors *act()* methods are called. Similar to *setPaintOrder*, the act order is set by class and the order of objects of the same class cannot be specified. When some classes are not specified in a call to *setActOrder* the same rules apply as for *setPaintOrder*.

In each turn (such as one click of the *Act* button) the *act* method will be called on every object of the class which was listed first in the call to *setActOrder*, then on every object of the class which was listed second, and so on.

Responding to the scenario being started or stopped

When the scenario is started Greenfoot will call the *started* method of the current world object, and when the scenario is stopped it will call the *stopped* method. If you want to run some code when these events happen, override these methods in your subclass of *World*. To 'override' a method in a subclass, you simply declare a method with the same name, such as how you override the *act* method from *Actor* in every subclass of it that you create.

```
public void started(){
    // ... Do something when the scenario has started
}

public void stopped(){
    // ... Do something when the scenario has stopped
}
```

14 Using support classes

There are a number of reusable classes available to use in your projects. These can help you achieve effects and functionality in your scenarios that would take a long time to program on your own, or that you do not have the programming experience to achieve. Some of the support classes are actors themselves that you can put straight into your scenaros (such as *Explosion*), some are abstract subclasses of Actor which you have to subclass with your own Actors to provide some extra functionality to those Actors (such as *SmoothMover*), and some are not actors at all, but utilities which actors can use (such as *Vector*, as used by *SmoothMover*).

There is a list of official Greenfoot support classes at http://greenfoot.org/programming/classes.html. There is also a collection of support classes on the Greenfoot Gallery at http://greenfootgallery.org/collections/4. To use any of the support classes, you first have to download their source code. On the list of official useful classes, clicking the name of the class will show you its source code. You can also right-click the link to save the *.java* source file. On the gallery, click on the name of a scenario to go to its page. There you can try out the scenario, or click the *Download the source for this scenario* link above it.

There are two ways to copy a support class into your own scenario. The simplest is to create a class of the right name in your scenario, then copy the entire contents of the support class's source and paste it over the entire contents of the class you just created in your scenario. The other way is to save the files for that class into your scenario folder. If you are using a support class from the official list, then save the *.java* file into your scenario folder. If you are using a class from a gallery scenario, copy across the file with the same name as the class with a *.java* extention from the gallery scenario's folder to your scenario's folder. (You can also copy across the files with the same name as the class and the *.class* and *.ctxt* extentions, but you do not have to.)

Note that some support classes use other support classes, images or sounds, so you may have to copy them in to your scenario to use the class. (For example, the *SmoothMover* abstract actor requires the *Vector* support class, and the *Explosion* actor requires an image and a sound to be copied into the *images* and *sounds* folders respectively.)

Using reusable actors

Once you have copied a reusable actor into your class, it should be ready to use like any other actors in your scenario - you can create instances of it, create subclasses of it, set its image and

edit its source code.

Using abstract actors

Some support classes are *abstract*. This means that you cannot create instances of them - you cannot place them directly on the world - similar to how the *Actor* class itself works. To use them, create actors which are subclasses of the abstract classes - either by clicking *New subclass* from the popup menu, or by changing the class after the *extends* keyword in an actor's source to the abstract class. For example, to make an existing actor called *MyActor* a subclass of the *SmoothMover* class, change the appropriate line in the *MyActor* class source code from:

```
class MyActor extends Actor
```

to:

```
class MyActor extends SmoothMover
```

MyActor will still be extending *Actor* as *SmoothMover* extends *Actor* and *MyActor* extends *SmoothMover*.

Using utility classes

To use the support classes that are not actors or abstract actors, you copy them into your scenario the same way as the other support classes. As they are not actors you cannot place them in the world - they are made to be used by the source code of the actors. For example, the *Vector* support class is used to store a motion vector (a direction and velocity) which is used by the *SmoothMover* class to store its movement. These classes can be created, stored in variables and have their methods called like any other objects.

Creating your own support classes

If you create a class which you would like to share, create a scenario demonstrating the class and upload it to the Greenfoot Gallery with the *demo* tag, making sure you include the source code. Other users of the gallery will be able to view it, try it out, leave feedback and download it for their own use, and it may be added to the *Support classes* collection.

15 Exporting a scenario

Greenfoot provides you with three ways to export your scenario, all available from the *Export...* option under the *Scenario* menu. The three options available to you are:

- Publish the scenario to the Greenfoot gallery
- Export the scenario as a webpage
- Export the scenario as an application

Publishing a scenario to the Greenfoot gallery

The greenfoot gallery is at http://www.greenfootgallery.org and provides a way for Greenfoot users to share their scenarios with other people, try out other people's scenarios and both give and receive feedback.

By publishing your scenario to the greenfoot gallery you are making it publicly available to anyone who visits the site. If you only wish to share your scenario with specific people, export it as an application and distribute it by email.

To publish a scenario to the gallery you will need a gallery account. You can create an account by following the link from the gallery homepage or clicking the *Create account* link at the bottom of the export dialog in Greenfoot. You will need to create an account once, then each time you wish to publish a scenario you enter your username and password in the export dialog.

The *Publish* page of the export dialog box provides you with fields to enter information to be published with your scenario.

.

Each scenario on the gallery has an icon, which is part of a sceenshot of the scenario. You can select what that icon should show the *Scenario icon* box. It shows a picture of the scenario as it is at the moment. You can use the slider next to the box to zoom in and out, and you can drag the image around in the box (after you have zoomed in) to select the exact area you want the icon to show. If you run the scenario and pause it in the middle, then your icon can show what the scenario looks like while it is running, which is useful if your main actors are not visible in the world at the beginning of the scenario.

Enter a title for your scenario, a short description and a full description. The short description will be shown next to your scenario when it is displayed in a list, such as a search results page. The full description will be displayed above your scenario on the gallery, and can include an explaination and instructions for using the scenario. If you have your own webpage about this scenario you can enter its URL to provide a link to it on the gallery.

The Greenfoot gallery organises scenarios by means of tags. Select some of the commonly used tags if they are relevant to your scenario, and add any tags of your own. Tags should usually be one word long, or a small number of words connected by hyphens, and each tag should be typed

on a new line in the textbox. Have a look at the tag cloud on the homepage of the gallery for ideas about what tags might be appropriate. It is always a good idea to re-use tags that other authors have already used so that similar scenarios can be grouped together. The popular tags provided for you in the export dialog are:

game
: For games

demo
: For demonstrations of Greenfoot features or new ways to do things in Greenfoot

simulation
: For simulatons of real-world situations, such as behaviour of swarms or the workings of a machine

mouse
: For scenarios which use mouse input

GUI
: For scenarios which use some sort of Graphical User Interface (such as buttons, dialogs, sliders and other application-like input)

There are two additional options which you can select for your scenario. If you check the *Publish source code* check box then other users of the gallery will be able to download your scenario to see its source code, and play around with it on their computers (however that will not affect the version on the gallery). If you select this option your scenario will have the *with-source* tag added to it when it is published.

The *Lock scenario* option allows you to prevent users of your scenario from changing the speed of execution and dragging objects around the world before the scenario is run. If you uncheck this box then users will be able to move your actors around and change the speed, as is possible in Greenfoot itself.

Once you have filled in all the details, make sure you have entered your username and password and click the *Export* button to publish your scenario to the gallery. If you click *Cancel* then the details you have entered will be saved ready for when you do want to export it.

Exporting the scenario as a webpage

To export your scenario as a webpage, select the *Webpage* section in the export dialog box. Select a folder to save the webpage to, choose whether you want to lock the scenario or not, and click *Export*.

Checking the *Lock scenario* check box prevents users of your scenario from changing the speed of execution and dragging objects around the world before the scenario is run. If you uncheck this box then users will be able to move your actors around and change the speed, as is possible in Greenfoot itself.

Note that exporting the scenario as a webpage will not publish your scenario on the web, it will only allow you to view the scenario in a web browser. To publish the scenario to the web (if you do not have your own website) use the *Publish* section of the export dialog to publish the scenario to the Greenfoot gallery. To publish the scenario to your own website, export it as a webpage then upload the *.html* and *.jar* files to your web server.

Exporting the scenario as an application

To export your scenario as an application, select the *Application* section in the export dialog box.

Select a folder to save the application to, choose whether you want to lock the scenario or not, and click *Export*.

Checking the *Lock scenario* check box prevents users of your scenario from changing the speed of execution and dragging objects around the world before the scenario is run. If you uncheck this box then users will be able to move your actors around and change the speed, as is possible in Greenfoot itself.

The application created will be an *executable jar file*. Any computer with the right version of Java installed should be able to run this application. On most computers you will be able to double-click on the jar file to run it. If the computer is not set up to run jar files like that, you will have to use the following command on a command-line to run the scenario:

```
java -jar Scenario.jar
```

While you are in the same folder as the jar file, replacing the name *Scenario.jar* with the name of your file.