technocamps

Inspiring | Creative | Fun

Ysbrydoledig | Creadigol | Hwyl

Greenfoot Workbook

i.t.wales www.itwales.com

Swansea University Prifysgol Abertawe

Prifysgol Aberystwyth University

Prifysgol Bangor University

University of South Wales Prifysgol De Cymru

ESF Llywodraeth Cymru Welsh Government

Ewrop & Chymru: Buddsoddi yn eich dyfodol Cronfa Gymdeithasol Ewrop

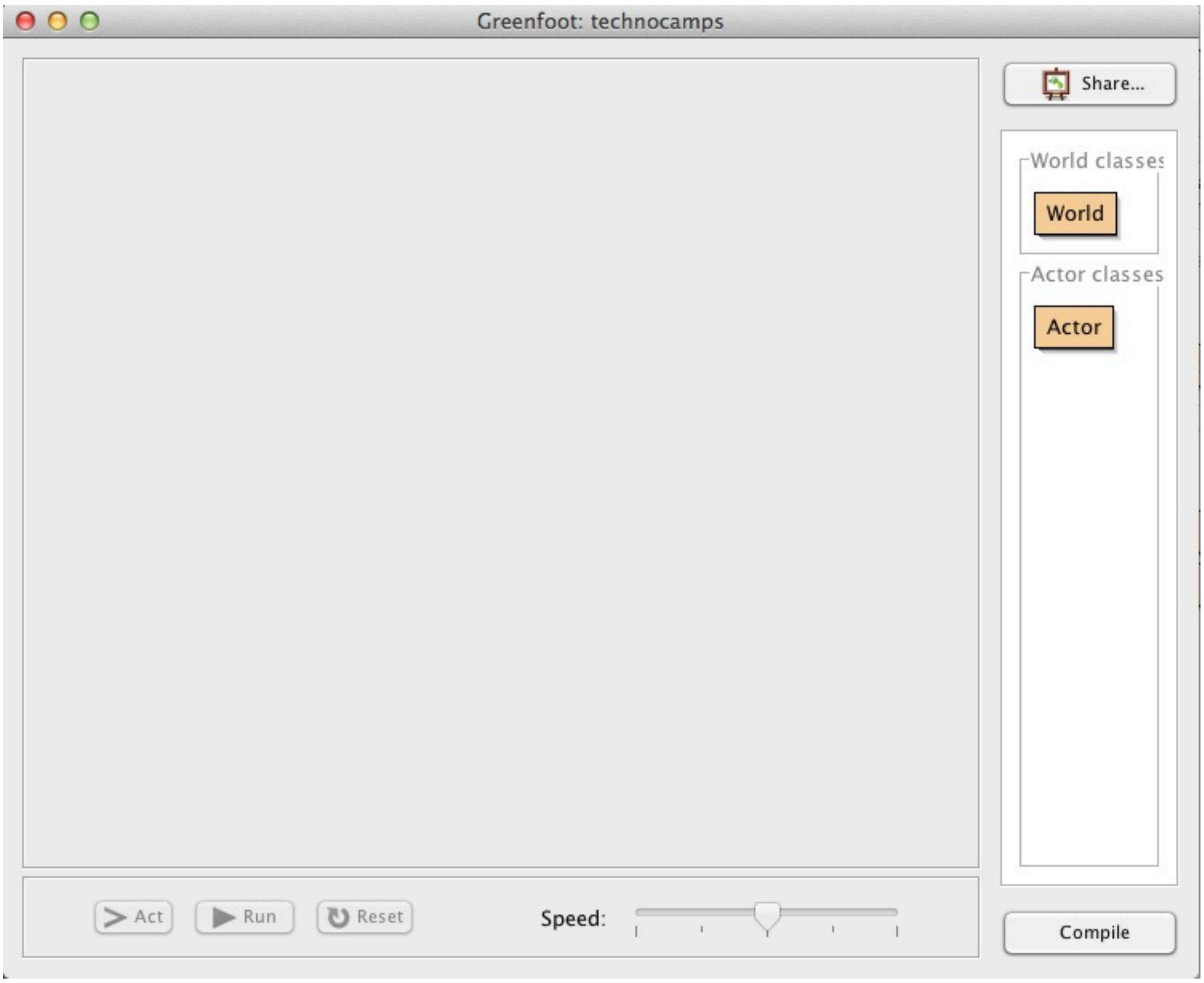
Europe & Wales: Investing in your future European Social Fund

Greenfoot

Greenfoot is an excellent introduction into advanced programming. You should already be familiar with what programming is and have some background experience, such as our Scratch or GameSalad workshops or Visual Basic and similar within your schools.

One of the benefits of using Greenfoot for an introduction into a real programming environment is not only the helpful guides and tutorials available through the website, but also the colourful user-friendly design of the interface. Java as a programming language is under high demand throughout the Industry. It is a great language to learn and incorporates a textual experience of programming, providing users with great initial programming knowledge and understanding of basic principles.

The interface can be broken down as follows. In the centre is where the World is, once you've applied a background and begin adding actors you will build up your game here. Beneath the World, you can see the 'Execution' tools, these include the buttons 'Act', 'Run', 'Reset' and a 'Speed' slider bar. On the right of the interface you can the World and Actor classes, these are required when using Greenfoot and some classes can differ depending on the code you apply and the how they are used.



Greenfoot

On the bottom right hand corner of the image on Page 1, there is a button called 'Compile'. This button (once pressed) changes the code into a way the computer understands. The way we write code in Java is similar to the way we use natural language, but the computer doesn't understand this, so it needs to transform the Java code into Binary so it knows how to output your game onto the World. When you begin adding actors and new subclasses to your game there will be lines over the boxes, these disappear once the code is compiled for the first time.

1) Adding a background

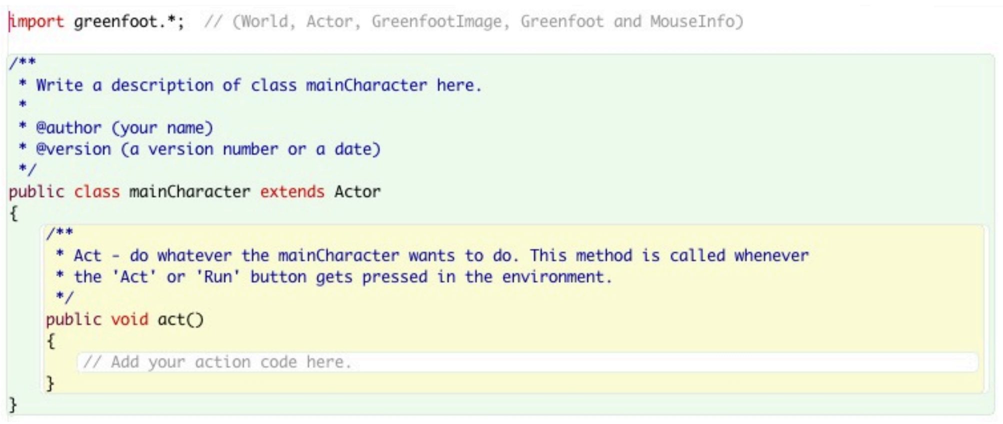
To begin, open up a new scenario and give it a name. A small, blank window should open with a "World" and "Actor" class. To give the world a background, right-click on "World" and select "New subclass". Here you create a world subclass, where you give it an appropriate name and apply a chosen or imported image. There is already a "backgrounds" folder within the image categories to choose from.

Note: There cannot be any spaces in the class name, so if you wish to use several words to name your class use a variation of uppercase and lowercase letters. Such as "newWorld".

Once you have pressed ok, you will see a new subclass appear in the class structure on the right hand side of the screen but the World has not yet been populated with the new background image. To do this, use the "Compile" button and you will see the World update.

2) Adding actors

Similarly to task 1, you can add new actor subclasses by right-clicking on "Actor" and selecting "New subclass". Add a new image subclass and assign it an image of your choosing, just do one for now for the purpose of demonstration. Don't forget to give your Actor subclass an appropriate name.



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

/**
 * Write a description of class mainCharacter here.
 * 
 * @author (your name) 
 * @version (a version number or a date)
 */
public class mainCharacter extends Actor
{
    /**
     * Act - do whatever the mainCharacter wants to do. This method is called whenever
     * the 'Act' or 'Run' button gets pressed in the environment.
     */
    public void act() 
    {
        // Add your action code here.
    }    
}
```

Adding functionality to your game is the next element of this workbook, where you can begin adding code to the game. Double click on your actor and you will see it opens up a new window where there is some code already available.

Greenfoot

At the top you can see import - importing Actor, World, Greenfoot and GreenfootImage. Underneath you can see "`/* ... */`" and lines that begin with "`//`", these are comments. The compiler knows not to compile any text that begin in such a way, they are purely notes to the author or whoever is reading the code. Even the best of programmers comment their code and start each class with a description and information about authors and/or versions.

Beginning the Actor class you can see "`public class ***** extends Actor`", inheriting for the Actor class. Each class must begin and end with "{" and "}" and all content must be between the 2 curly brackets otherwise errors will occur in your code because the game won't know where the class ends.

On the bottom of the window is a white bar, once you've added code and clicked compile to see whether your code is correct so far, error messages appear here to alert the user of any mistakes made in the code "`e.g. ';' expected`".

Inside the class there is a method with no code added to it, the code in here is called when the "Act" button is pressed on the main game screen. Pressing "Act" will only call the method once, but by clicking the button next to it labelled "Run", this calls the method over and over.

3) Act and Run buttons

Try adding the following code to the "Act" method:

```
move(4);
```

What happens when you press "Act" on the main screen? What happens when you press "Run"? Try changing the value in the (); what do you think this value stands for?

Instead of getting the character to move in a straight line, we can also add numerous lines of instructions in the "Act" method that will tell the character to do more movements and actions. For instance, you could tell it to move and turn when the "Act"/"Run" buttons are pressed.

4) Turning

Try adding the following code to the "Act" method:

```
turn(3);
```

What does the value in the brackets stand for? How does the action differ if a negative value is entered inside the brackets instead? What happens when you press "Run"?

You will have noticed at this stage that semi-colons **;** are used at the end of some lines of code, this tells the computer where the end of the instruction is and to read the next line.

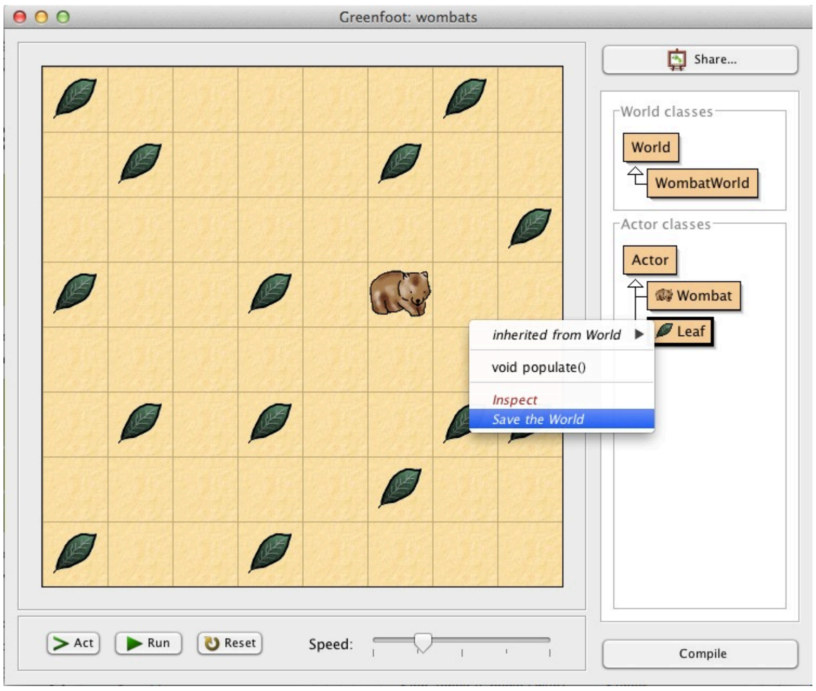
5) Wombat scenario

Open up an existing example scenario on Greenfoot, in the scenarios folder called “Wombats”. You will see the list of classes appear on the right hand side of the Greenfoot main screen. You will also notice that there are no characters in the World yet.

Add some instances of the actors to the World by right clicking on the actor subclass “Wombat” and selecting “New Wombat()”, you can then place one onto the World. Press run and see what happens.

If some of your classes have lines across them on the main Greenfoot window, it means you need to compile your code so the computer knows how to display and run your game.

Populate the World with several leaves and several wombats, press “Run” and “Act” - what do you notice about the way they move and interact? What do you notice about the actors when “Compile” is pressed again? To solve this problem of re-populating the World each time you can save the world or alternatively you could program the actors to populate in specific locations or even randomly! To begin by saving the World in the same format, once you have populated the World how you want it to look, right-click on the World and select “Save the World”.



6) Obstacles

Add a new Actor subclass called “Rock”, with an image of a rock assigned to it. No code is required for the Rock as we want it to be a motionless obstacle within our game.

7) Randomly populating

To place a number of rocks randomly within the World, open up the code for the "WombatWorld" class. Add the following code to this class, make sure it is not inserted within other methods:

```
public void randomRocks(int howMany)
{
     for(int i=0; i<howMany; i++)
     {
          Rock rock = new Rock();
          int x = Greenfoot.getRandomNumber(getWidth());
          int y = Greenfoot.getRandomNumber(getHeight());
          addObject(rock, x, y);
     }
}
```

See if you can also make the leaves randomly appear in the World. Now that the rocks are randomly populating the World, the next task is to make them become static objects so the Wombats cannot move through the rocks.

8) Rocks as obstacles

In the wombat class, make the new "noRocks()"method as below:

```
public boolean noRocks(int x, int y)
{
     World myWorld = getWorld();
     List rocks = myWorld.getObjectsAt(x, y, Rock.class);
     if(rocks.isEmpty())
     {
          return true;
     }
     else
     {
          return false;
     }
}
```

Find the "canMove" method in the Wombat class and change "`return true`" to:

```
if(noRocks(x, y)){
     return true;
}else{
     return false;
}
```

Alternatively, could also instruct the Wombats to move randomly around the World by implementing the following code:

```
public void turnRandom()
{
     int turns = Greenfoot.getRandomNumber(4);
     for(int i =0; i<turns; i++)
     {
          turnLeft();
     }
}
```

9) Twirling Leaves

Following the "turn" instruction implemented earlier in task 4, see if you can command your leaves to turn on the spot when the "Act" method is called.

Sound can also be implemented into your game, improving the quality of the game by using "`Greenfoot.playSound(filename.filetype);`" within the code. Perhaps it could be embedded within a conditional statement for when the Wombat touches the leaves a chosen sound can be played.

Score is also a fun part of designing, developing and testing a game, but this requires a bit of code. To begin, create a new actor called Counter, it does not need an image, add the code below to the WombatWorld under "`public class WombatWorld`":

```
theCounter = new Counter("Leaves eaten", 0);
addObject(theCounter, 1, 1);
```

Add a new method in "WombatWorld" class as below:

```
public Counter getCounter()
{
     return theCounter;
}
```

10) Score

In the "Act" method in the Counter subclass, enter the following code:

```
String text = "";
public Counter(String label, int value)
{
     text = label;
     setValue(value);
}
```

Add a new method in the Counter subclass as below:

```
public void setValue(int newVal)
{
     GreenfootImage img = new GreenfootImage(100,50);
     img.drawString(text + ":" + newVal, 2, 20);
     setImage(img);
}
```

In the Wombat class, underneath the line where the Wombats eat the leaves, add the following line of code:

```
WombatWorld theWorld = (WombatWorld) getWorld();
Counter counter = theWorld.getCounter();
counter.setValue(leavesEaten);
```

11) Winning

To win the game, create a new actor called "Win". You want this to appear once the Wombat is in the winning position of 7 , 7 on the 8x8 World. To begin, add the following code to the "Leaves" Actor subclass:

```
if(getX()==7 & getY()==7)
{
     ((WombatWorld) getWorld().gameOver();
}
```

Create a new gameOver() method to the WombatWorld and add the following content:

```
addObject(new Win(), 3, 3);
Greenfoot.stop();
```

11) Winning continued...

Add the following constructor to Win:

```
public Win()
{
     GreenfootImage image = new GreenfootImage(100,100);
     image.drawString("You Win", 0, 100);
     setImage(image);
}
```

12) Have a go

At this stage you should have a good overall understanding of the Java syntax and enough knowledge to try and make your own edits to the Wombats scenario. Try achieving the following tasks:

- Changing the images for actors and backgrounds.
- Change the size of the world (currently 8x8).
- Change the method of scoring, perhaps deduct points when collide with an obstacle.
- Adding various sounds during the game.

You can find a list of other commands at the following link:
www.greenfoot.org/files/javadoc/index-all.html

To try out another tutorial, go to the following webpage:

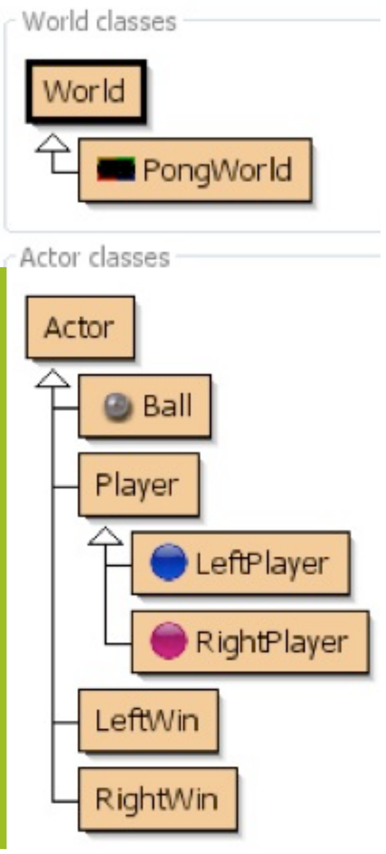
www.greenfoot.org/doc/tut-2

This is a tutorial that will explain how to do movement in Greenfoot and how to control actors using a connected keyboard. Download the .zip file on the page and follow this demonstration, some of the code will be familiar. Once this is completed, move onto the next tasks to apply this knowledge is a slightly different way creating your own "Pong!" game.

Don't forget to save your scenario.

Create a new scenario and save it as "Pong". Here you are going to develop a brand new game, applying your knowledge of Java so far as well as learning more about the language and the different instructions to implement.

The game is to have 2 players, one on the left and one on the right. Each use up and down controls on the connected keyboard to move their actors up and down the screen, bouncing a ball across the World. The overall aim of the game is to knock the ball in the other oppositions goal that they are protecting to win or to accumulate points - depending on how you decide to code it.



13) Pong classes

On the right is a break-down of the class structure for the pong scenario. It also shows what images to apply to each of the classes/subclasses.

Replicate the image on the right, naming the Actors appropriately. Then insert an instance of the "LeftPlayer" to the left hand side of the World and an instance of the "RightPlayer" to the right hand side of the World. Add an instance of the "Ball" Actor to the middle of the world. Click on "Save the World".

From the class structure above, you can see that LeftPlayer and RightPlayer are both subclasses from Player, this means you could add instructions to the Player class and both the LeftPlayer and RightPlayer classes will inherit this information.

14) LeftPlayer and RightPlayer

Insert the following to "Player" so we can use the methods in the LeftPlayer and RightPlayer:

```
public void moveUp()
{
     setLocation(getX(), getY()-2);
}
public void moveDown()
{
     setLocation(getX(), getY()+2);
}
```

15) Conditional player control

Now that the information about moveUp() and moveDown() is ready to reference, add the following code to Act() for the RightPlayer telling the game when to tell the RightPlayer instance to move up (when the “up” key is pressed) and when to move down (when the “down” key is pressed):

```
if(Greenfoot.isKeyDown(“up”))
{
    moveUp();
}
if(Greenfoot.isKeyDown(“down”))
{
    moveDown();
}
```

LeftPlayer will be using the keys “W”, “A”, “S” and “D” instead of the arrow keys on a keyboard, how can the above code be altered for this? Add the code to the “LeftPlayer” class in the Act() method.

16) Winning the game

Firstly, you need to tell the game where the goal areas are on the World. To do this, open up PongWorld and enter the following code, telling the game where “leftWin” and “rightWin” are:

```
public void leftWin()
{
    addObject(new LeftWin(), 200, 300);
    Greenfoot.stop();
}
public void rightWin()
{
    addObject(new RightWin(), 200, 300);
    Greenfoot.stop();
}
```

Next, add the following code for “LeftWin”, copy this into “RightWin” and edit accordingly:

```
public leftWin()
{
    GreenfootImage image = new GreenfootImage(200, 200);
    image.setColor(new Color (255, 255, 255));
    image.drawString(“Left Player Wins”, 50, 100);
    setImage(image);
}
```

Add the following code to the “Ball” class, telling the game when to call the Win method:

```
public void win()
{
     if(getX()== 0)
     {
          ((PongWorld) getWorld().rightWin();
     }
     if(getY()== 599)
     {
          ((PongWorld) getWorld().leftWin();
     }
}
```

You are almost complete, but the last thing you need to implement is movement for the ball. The ball needs to continuously move and react as it comes in contact with “LeftPlayer” or “RightPlayer” or the goals being protected.

17) Ball movement

Add the following code to the ball class:

```
public boolean hit()
{
     Player player = (player) getOneIntersectingObject(Player.
               class);
     if(player != null)
     {
          ySpeed = (player.getY()-this.getY())/10;
          xSpeed = xSpeed*-1;
          return true;
     }
     return false;
}
public boolean ifOnEdge()
{
     int x = getX();
     int y = getY();
     if(y<50)
     {
          ySpeed = ySpeed*-1;
          return true;
     }
```

17) Ball movement continued...

```
    if(y>350)
    {
        ySpeed = ySpeed*-1;
        return true;
    }
    if(y < 100 || y > 200)
    {
        if(x < 50 || x > 550)
        {
            xSpeed = xSpeed*-1;
            return true;
        }
    }
    return false;
}
```

You also need to add 2 variables to make them available to all methods:

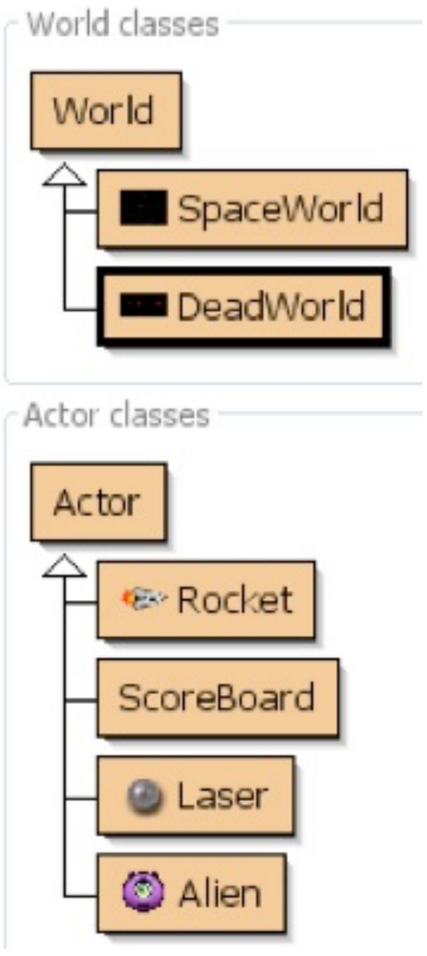
```
int xSpeed = 4;
int ySpeed = 0;
public void act()
{
    hit();
    ifOnEdge();
    setLocation(this.getX() + xSpeed, this.getY() + ySpeed);
    win();
}
```

Don't forget to save your scenario.

Next, create a new scenario and save it as "Shooting". Here you are going to develop a brand new game, applying your knowledge of Java as well as learning more about the language and the different instructions to implement.

The game is to have 1 player (the rocket), that shoots lasers at randomly appearing aliens to accumulate a score. You will notice on this particular scenario that there are 2 worlds, one "SpaceWorld" and one "DeadWorld". The "DeadWorld" is used as a Game Over screen!

Make sure you draw the "DeadWorld" background image and import it, you could draw this on "Paint" or similar software with the words "You are dead" or "You have lost" in it.



18) Shooting classes

On the right is a break-down of the class structure for the Shooting scenario. It also shows what images to apply to each of the classes/subclasses.

Replicate the image on the right, naming the Actors appropriately.

To being, resize the laser image by going into the laser class and entering the following code:

```
public laser()
{
     this.getImage().scale(10,10);
}
```

19) Laser

You need to tell the laser how to act. Remember to call the following method from the "act" method in laser - as it is an Actor subclass:

```
public void move()
{
     if(this.getX() == getWorld().getWidth()-1)
     {
```

19) Laser continued...

```
            getWorld().removeObject(this);
      }
      else
      {
            super.setLocation(this.getX()+5, this.getY());
      }
}
```

20) Adding Rocket to the world

To add an initialised rocket to the spaceWorld, add the following code:

```
Rocket rocket = new Rocket();
addObject(rocket, 50, 200);
```

21) Rocket

You need to implement a manner of controlling the Rockets movement using the arrow keys on a connected keyboard. Using your knowledge acquired from previous tasks and scenarios, add a method called “keyboard” to the Rocket class and give commands for when up, down, left and right buttons have been pressed. Note that the y axis is upside down so to move up you need to deduct values from the y co-ordinate, but the x axis remains as usual. Make sure you call the keyboard() method in the Act() method to make sure the Rocket movement controls work when you press “Run”.

22) Shooting the laser

To shoot the laser from the rocket, add the following code:

```
public void shoot()
{
      if(Greenfoot.isKeyDown(“space”))
      {
            getWorld().addObject(new Laser(), getX()+48, getY()-2);
      }
}
```

23) Aliens

To tell the aliens how to act, enter the following code in the alien actor subclass and don't forget to add the "move()" method to the "Act()" method:

```
public void move()
{
     if(this.getX() == 0)
     {
          getWorld().removeObject(this);
     }
     else
     {
          super.setLocation(this.getX()-5, this.getY());
     }
```

Previously you have applied code to other scenarios that populate the World with that character randomly, repeat this for the "Aliens" by entering the desired code in the "SpaceWorld" act method.

24) Hit

Add the following hit() method in Laser, Aliens and in Rocket so they have instructions on how to react if a collision occurs:

```
public void hit()
{
     getWorld().removeObject(this);
}
```

Add the following isHit() method to Laser and Rocket to see if it has hit the Alien. If the Alien has been hit then remove the Laser and the Alien from the World.

Don't forget to add isHit() to the "Act" method:

```
public void isHit()
{
     Alien alien = (Alien) getOneIntersectingObject(Alien.class);
     if(alien != null)
     {
          alien.hit();
          hit();
     }
}
```

25) isHit edited

In the Rocket actor subclass, edit the isHit() method just added so that you can have a game over, the code should look like below:

```
public void isHit()
{
     Alien alien = (Alien) getOneIntersectingObject(Alien.class);
     if(alien != null)
     {
          gameOver();
          hit();
     }
}
```

26) GameOver

Add the following code for a gameOver method to go in the Rocket actor subclass:

```
public void gameOver()
{
     DeadWorld deadWorld = new DeadWorld();
     Greenfoot.setWorld(deadWorld);
}
```

Don’t forget to save your scenario.

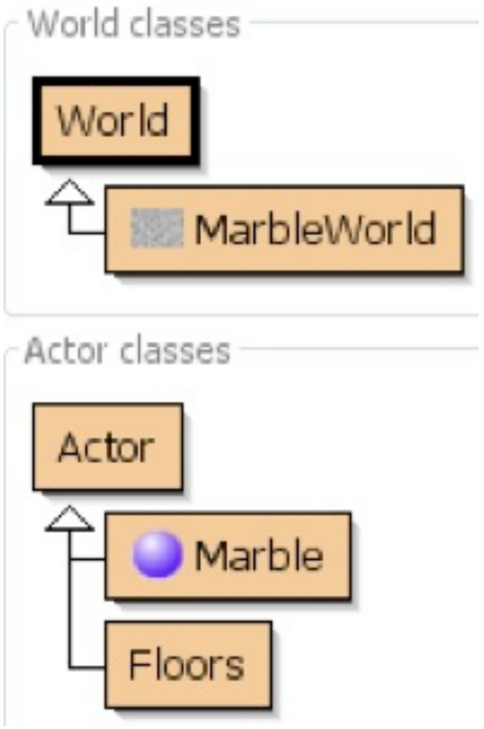
Create a new scenario and save it as “MarbleDrop”. Here you are going to develop a brand new game, applying your knowledge of Java so far as well as learning more about the language and the different instructions to implement.

The game is to guide a marble that follows the mouse pointer left and right across the screen to try and fall through the floors with gaps that are moving up the screen. The gaps in the floor are randomly produced increasing how challenging the game is.

27) MarbleWorld Classes

On the right is a break-down of the class structure for the MarbleDrop scenario. It also shows what images to apply to each of the classes/subclasses.

Replicate the image on the right, naming the Actors appropriately



The floors actor subclass does not require any image, these will be created at random a little later on. To begin, the marble needs to be set to the middle of the World, then should begin slowly falling down the screen, following the direction of the mouse pointer to the left or right. The floors will move up the screen at random intervals with randomly produced gaps in them for the marble to fall through. The floors are a red line across the screen that will disappear at the top of the screen.

28) Setting the screen size

To begin, the size of the World needs to be set to 700 x 500 and the marble should be added to the center of the World by entering the following code in the MarbleWorld class:

```
public MarbleWorld()
{
     super(700, 500, 1);
     Marble marble = new Marble();
     addObject(marble, 350, 250);
}
```

29) Marble movement

The marble will be dropping down the screen slowly and will move left or right following the mouse pointer. To implement this, enter the following method into the Marble actor subclass:

```
public void fallDownScreenAndFollowMouse()
{
     setLocation(this.getX(), this.getY()+5);
     MouseInfo mouse = Greenfoot.getMouseInfo();
     if (mouse != null)
     {
          setLocation(mouse.getX(), this.getY());
     }
}
```

New floors need to be created at random intervals, this improves the fun element of a game that you can play over and over and it will be different each time. Add the code in the following task to the Marble actor subclass.

30) Random interval floors

```
public long last = 0;
public void addNewFloors()
{
     if(Greenfoot.getRandomNumber(100) < 10)
     {
          if(System.currentTimeMillis() - last > 1000)
          {
               last = System.currentTimeMillis();
               getWorld().addObject(newFloors(), 350, 500);
          }
     }
}
```

The next element of the implementation stage is stating what would happen if the Marble collides with one of the floors moving up the screen, instead of falling through the gap.

31) Marble collision with floor

Add the following code to the Marble actor subclass:

```
import java.awt.Color;
public void floorCollisions()
{
     Actor floor = getOneIntersectingObject(Floors.class);
     if(floor != null)
     {
          if(floor.getImage().getColorAt(this.getX(),10.equals(new
               Color(255, 0, 0)))
          {
               setLocation(this.getX(),this.getY()-floorSpeed-5);
          }
     }
}
```

32) Marble collision with floor

Next you need to get the game to draw the red floor lines with gaps in random positions, to do this enter the code below in the “Floors” actor subclass:

```
import java.awt.Color;
public int gapWidth = 100;
public int gapPos = 0;
pubic Floors()
{
     gapPos = Greenfoot.getRandomNumber(600);
     GreenfootImage image = new GreenfootImage(700, 20);
     image.setColor(new Color(255, 0, 0));
     image.drawRect(0, 0, gapPos, 20);
     image.fillRect(0, 0, gapPos, 20);
     image.drawRect(gapPos + gapWidth, 0, 700 - gapPos, 20);
     image.fillRect(gapPos + gapWidth, 0, 700 - gapPos, 20);
     setImage(image);
}
```

The final element to the code is ensuring the floors move up the screen and disappear when they reach the top of the World, to do this follow the next ask.

33) Floor movement

The following code needs to be entered into "Floors()" actor subclass:

```
public int floorSpeed = 3;
public void moveUpScreenAndDisappearAtTop()
{
     setLocation(this.getX(), this.getY()-floorSpeed);
     if(this.getY() < 10)
     {
          getWorld().removeObject(this);
     }
}
```

What improvements do you think you could make to improve the quality of the code and its efficiency? Try some of the following changes to your code:

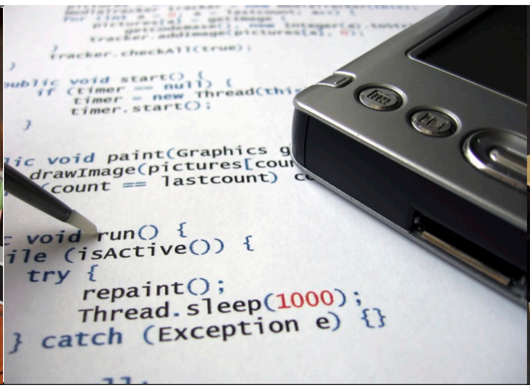
- Use variables to change the speed and gap width of the floors as they are created. Update these after a certain amount of time has passed. If the marble reaches the top of the screen, stop the game.
- Add a new constructor to the "Floors" class to update the floor speed and gap width using the inputs "`public Floors(int speed, int gapWidth)`".
- Try adding your own comments throughout the code to make sure it is readable and other developers understand the code implemented.

33) Have a go

Try making your own scenario. Try achieving some of the following tasks:

- Changing the images for actors and backgrounds.
- Change the size of the world.
- Have a method of scoring.
- Have more than one type of enemy.
- Add sounds.

You can find a list of other commands at the following link:
www.greenfoot.org/files/javadoc/index-all.html



```
tracker.checkAll(true);
}
public void start() {
  if (timer == null) {
    timer = new Thread(this
    timer.start();
  }

lic void paint(Graphics g
 drawImage(pictures[coun
 (count == lastcount) co

 void run() {
ile (isActive()) {
  try {
      repaint();
      Thread.sleep(1000);
  } catch (Exception e) {}
```

technocamps

www.technocamps.com

@Technocamps

Find us on Facebook