Session 1 – Welcome to Greenfoot & Code Breaker

Authored by Brian Cullen (bcullen@rossettschool.co.uk)

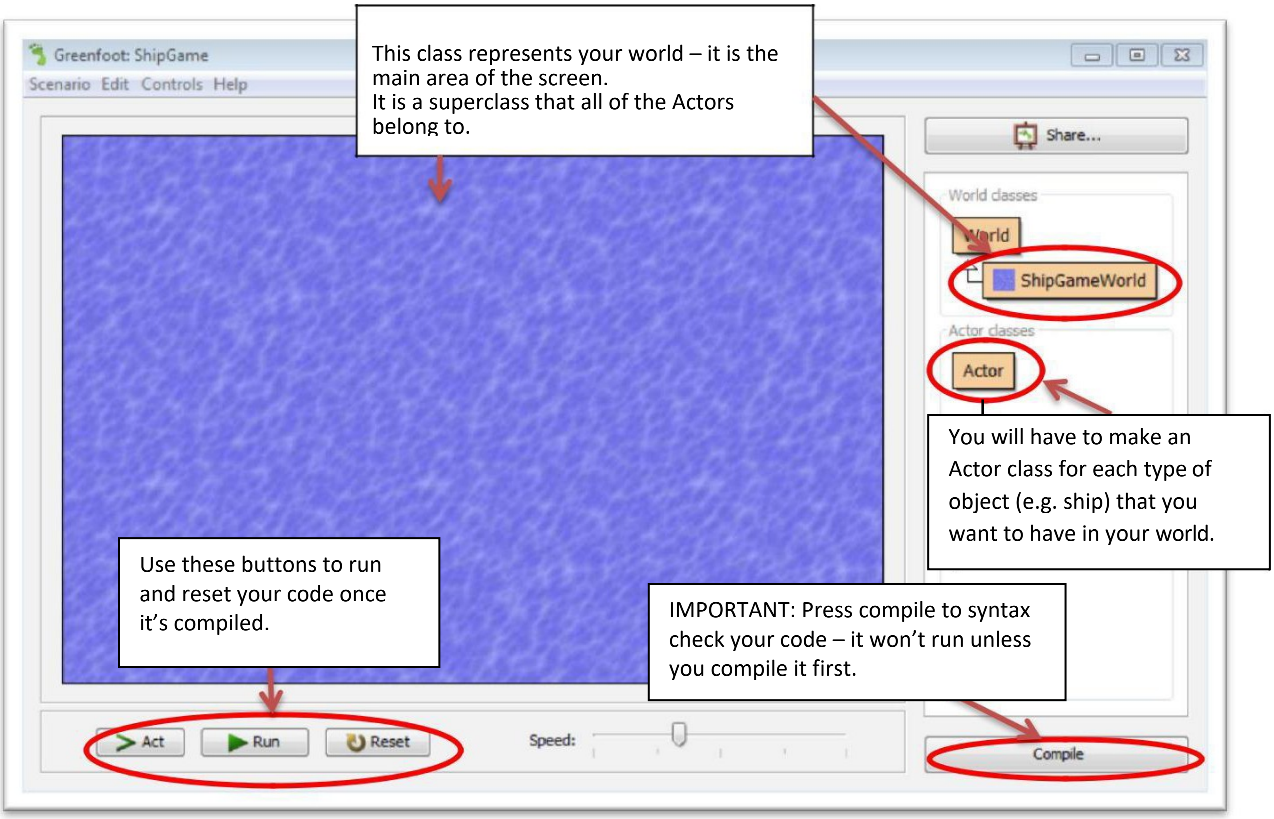
Edited by Ruth

To start:

Download the Session 1 - Ship Game folder from the Files area of Teams.

Open Greenfoot, then click on the project file in the folder you saved above.

Once you have done that your screen should look like this picture.



This class represents your world – it is the main area of the screen.
It is a superclass that all of the Actors belong to.

You will have to make an Actor class for each type of object (e.g. ship) that you want to have in your world.

Use these buttons to run and reset your code once it’s compiled.

IMPORTANT: Press compile to syntax check your code – it won’t run unless you compile it first.

Adding Actors

At the moment if you press the Run button nothing happens because we haven’t added anything to the world .

1. To create a new Actor right-click on the Actor class and pick the **New Subclass**… option

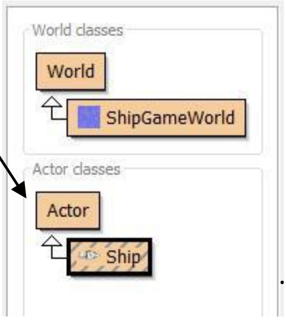
2. Name your new class Ship and choose the ship image provided (as shown on the right).

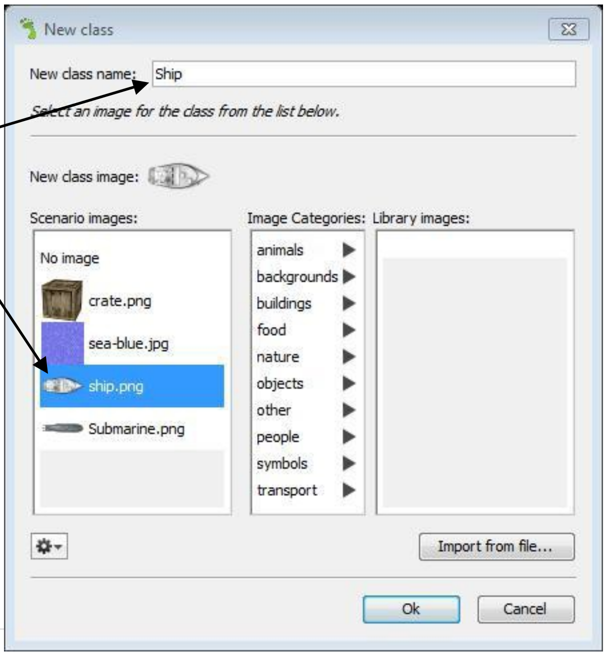
3. Press the OK button when you're done.

4. You should see your new Ship class on the Actors list here:

5. Right click on the Ship class > new Ship()

This will let you add an instance of your Ship class into the World



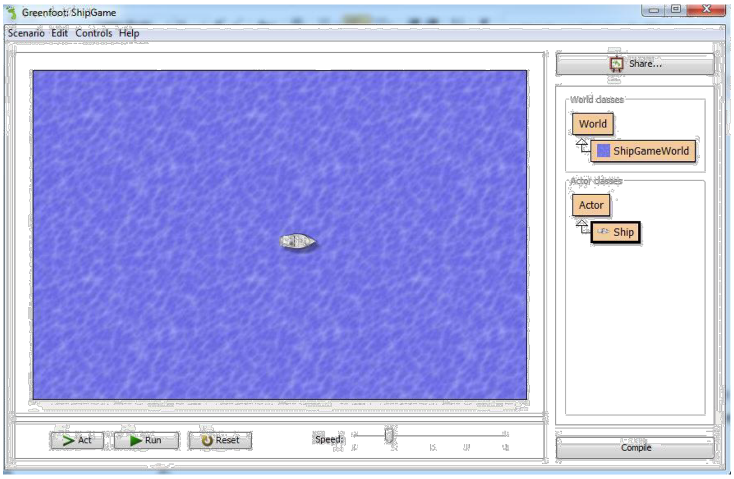


New class name: Ship

Select an image for the class from the list below.

New class image:

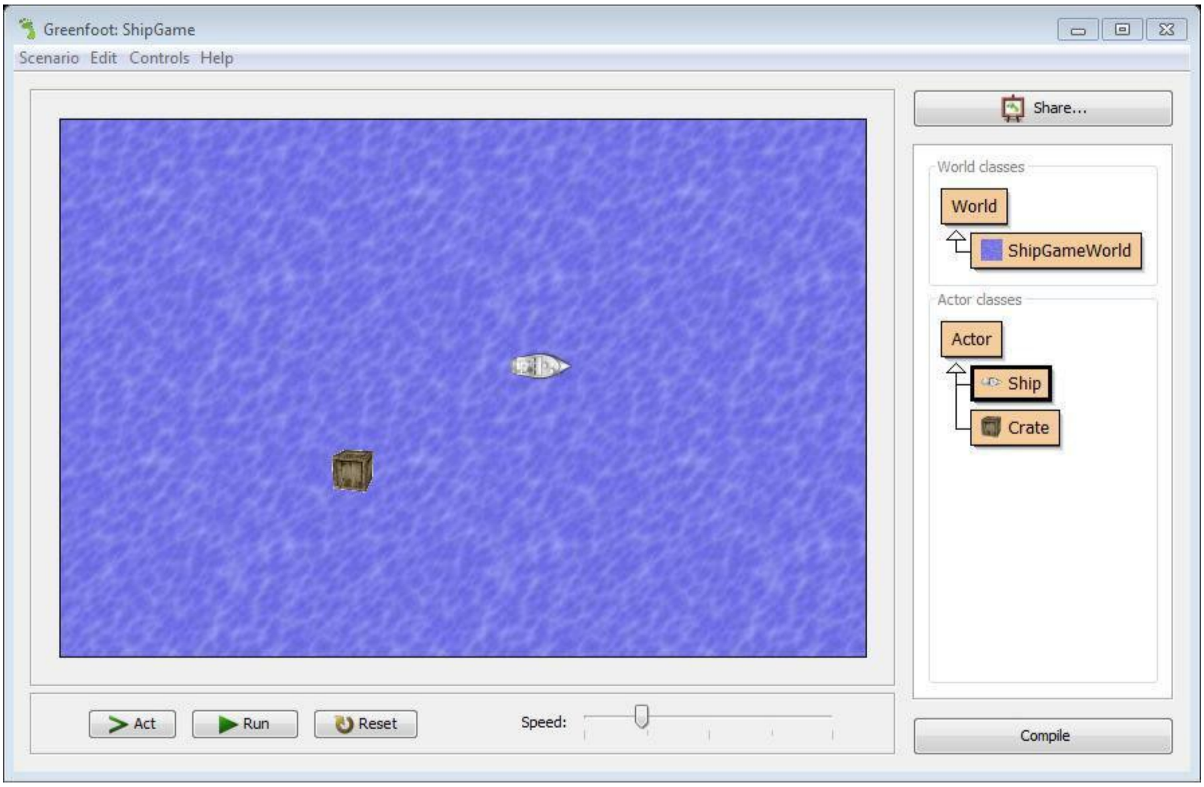
Scenario images: ship.png



So far so good but there's still not any code to do anything yet.

We're going to set up another class before we do that.

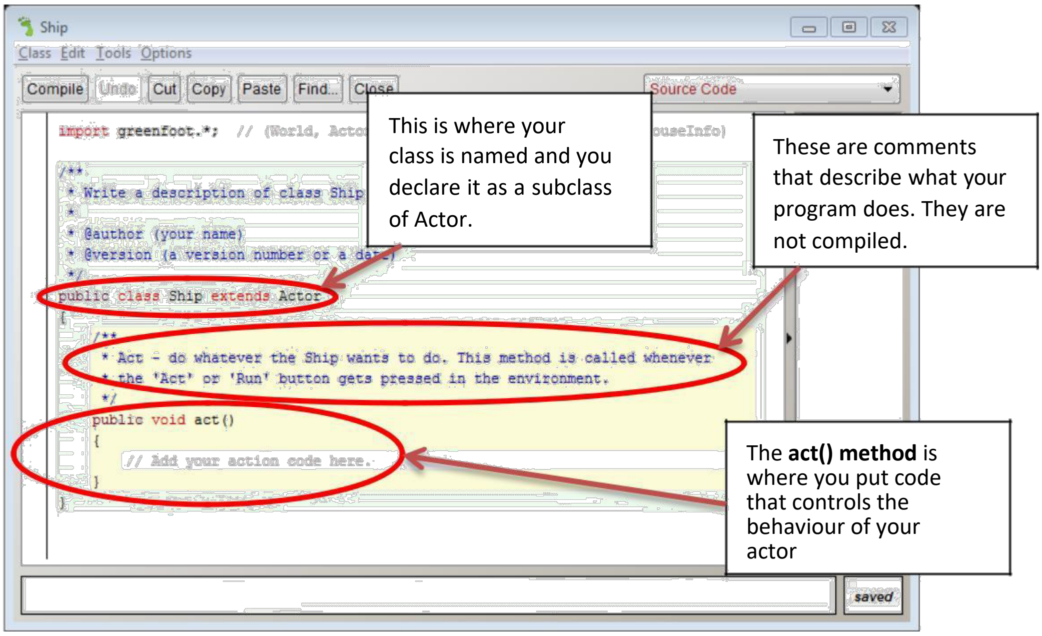
6. **Follow steps 1 to 5 above to create another Actor class called Crate**; you should end up with a scenario similar to the one shown below.



Controlling Actors

It is now time to consider what we want our actors to do. Our ultimate aim is to get the ship to move in response to the arrow keys but for now let's just get the ship to move in a straight line.

1. Double click on the Ship class on the right side of the screen and open the code editor as shown in the screenshot on the next page.



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

/**
 * Write a description of class Ship
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Ship extends Actor
{
    /**
     * Act - do whatever the Ship 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.
    }
}
```

This is where your class is named and you declare it as a subclass of Actor.

These are comments that describe what your program does. They are not compiled.

The **act() method** is where you put code that controls the behaviour of your actor

2. The act **method** (an object-oriented subroutine) is the first subroutine that is called once an **instance** of an Actor object has been created. In Greenfoot, act() is called repeatedly every millisecond or so.
   Change your code to add the following line and try running it (remember you will **need to compile it** and add a new ship to the world before running it).

```
public void act()
{
   move(10);
}
```

Your ship should now move in a straight line. You can try changing the 10 to a larger number to **speed up the ship or a lower one to slow it down**.

move() is a built in library method in Greenfoot – it's an example of **procedural abstraction** - we call the procedure without knowing the code behind it and how it works - all we need to know is what we need to pass it as a parameter (the number value) to make it work.

3. Now we're going to get the ship to turn in a circle by making use of a **turn()** method. Try the following code.

```
public void act()
{
   move (10);
   turn (10);
}
```

Reading from the Keyboard

The ship is moving nicely but we have no control. We want to check when the arrow keys on the keyboard are pressed and then move the ship in the correct way. We'll start with the ones shown in the table below.

Key	Behaviour
Up	Move 5 steps forward
Down	Move 5 steps backwards
Left	Turn 5 degrees to the left
Right	Turn 5 degrees to the right

To know if one of the arrow keys is being pressed, we can use Greenfoot's **`isKeyDown()`** method. It takes one parameter – the name of the key you are checking.
So to check if the left arrow key is pressed you would need to write the code below.

```
Greenfoot.isKeyDown(“right”);
```

We need to write "Greenfoot." in front of the method name because unlike the other methods we have used, this one belongs to a different class so we need to tell the computer where it can look to find the method

Edit and add code like this to an IF statement <u>in act()</u> so it will only move when the key is pressed e.g.:

```
if (something to check)
{
        do something
}
```

```
if (Greenfoot.isKeyDown(“right”))
{
      move (10);
}
```

1. **Work out how to move the ship backwards, using the same method** – think about the parameter values and how you can use them!

2. **Add code to the IF shown above so that your boat will move forwards and backwards** in the way shown in the table above.

3. Once you have the code working be sure to **adjust the parameters** to the methods to make the ship move in the way you want!

If you want to see something really odd then add more ships to the world and try it out. It looks like synchronised swimming! Do you know what is happening and why?

<u>Optional challenge:</u> Why not add a quick turn button? You know the feeling – there you are in a game you see the enemy coming and you can't quite turn around fast enough to get out of the way! Try adding an extra piece of code to the Ships act() method that will turn the ship 180 degrees when you press a particular key allowing you to make a quick escape.

Ship Game Part 2

So far, we've created two actors (Ship and Crate) and we got the ship moving in response to the keyboard. The next step is to make it so a ship can pick up crates and to do this we need **hit detection** to check if the ship is touching a crate.

Collision Detection

Greenfoot provides a number of methods that an actor can use to check this. We're going to use **`getOneIntersectingObject()`** It takes a single parameter which tells it what type of actor you want it to check for. To check for Crate actors we would write:

```
getOneIntersectingObject(Crate.class);
```

If this method finds that there is a crate intersecting (touching) the ship it returns which one it is so we need to store the return value. To do this we make a variable **`crate`.** The code below takes the return value of the call to the `getOneIntersectionObject()` function and stores it.

```
Actor crate = getOneIntersectingObject(Crate.class);
```

The next thing we need to do is check if the return is empty. If nothing is found, the method will return `null`. So, using an if statement, we must check that the answer we got is **`not null`.**

```
Actor crate = getOneIntersectingObject(Crate.class);
if (crate != null){
      // we have hit a crate what do we do?
}
```

Finally, we need to code what we want to do to the crate when we hit it - remove the crate from the world so that it is no longer on the screen.
The world class has a method called **`removeObject()`**. To call that method **we must first have access to the world our game is running in** and to do this we call the **`getWorld()`** method.

The final code is as follows so **add it to the bottom of the act() method for the ship** (underneath where you check for key presses).

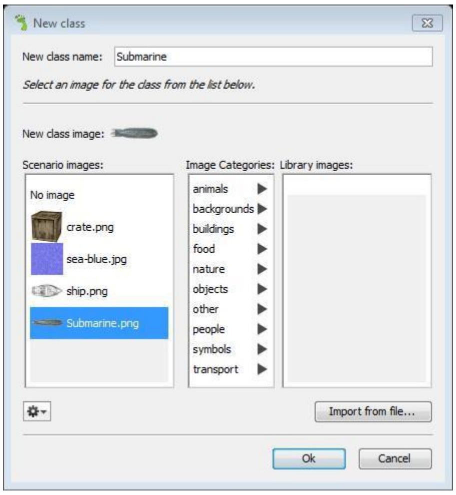
```
Actor crate = getOneIntersectingObject(Crate.class);
if (crate != null)
{
          //we have hit a crate so we get access to the
          //world using the getWorld() method and then
          //use the removeObject method to remove the
          //crate from the game.
         World shipWorld = getWorld();
         shipWorld.removeObject(crate);
}
```

TASK: Add some crates to the world and check that the ship 'collects' them when they're hit.

Adding Enemies

We also need to have a bit of challenge in getting the ship to avoid enemies - so let’s make some.

1. **Create a new class called Submarine using the image provided**. Do this in the same way that you have created the Ship and Crate class at the start of this task.



New class name: Submarine

Scenario images: Submarine.png

2. **Open up the code for the new Submarine class and add code to its act() method to get it moving in circles** like we did for the Ship.

3. If our ship is hit by one of the submarines **we need the ship (not the submarine) to explode**. We could put the code for this into either the Submarine or the Ship class – for now, we are going to put it in the Ship class, in act() underneath our other code, to keep all our collision detection code together.

   The code below is exactly same as before except for the two changes noted below.

```
    Actor sub = getOneIntersectingObject (Submarine.class);

    if (sub != null) {
            //we have hit a sub so we get access to the
            //world using the getWorld() method and then
            // use the removeObject method to remove
            //the ship from the game.
            World shipWorld = getWorld();
            shipWorld.removeObject(this);
}
```

You need to change this to the type of actor you are looking for now

This time we don’t remove the thing we hit but our instance of the ship (itself)

Test the game and make sure your code works.

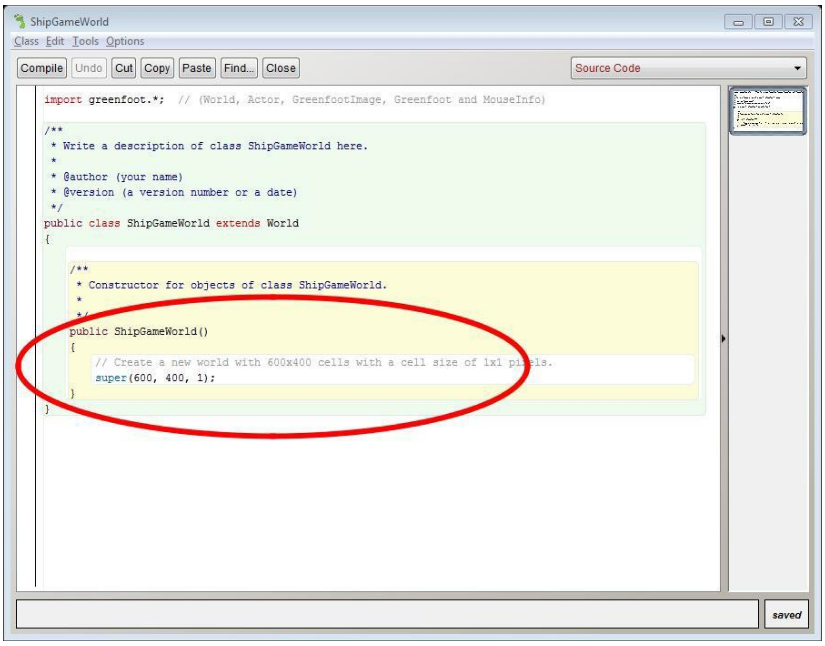
Bonus Code: If you want everything to stop once your ship has exploded then use the `Greenfoot.stop()` method.

Automatically Resetting the Game

Now we want to set the game up automatically rather than having to put the pieces on every time we reset. As this code will be setting up the world for the start of the game it needs to go in the **ShipGameWorld class.**

1. **Open the ShipGameWorld class by double clicking on it.**

   You will see a method with the same name as the class.
   This is called a **constructor** - it is the method called to setup the ShipGameWorld every time a new world is created.
   This is where we are going to put our code - underneath the line that starts with the word super.



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

/**
 * Write a description of class ShipGameWorld here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class ShipGameWorld extends World
{

    /**
     * Constructor for objects of class ShipGameWorld.
     *
     */
    public ShipGameWorld()
    {
        // Create a new world with 600x400 cells with a cell size of 1x1 pixels.
        super(600, 400, 1);
    }
}
```

2. **To add an actor to the world** we call the **`addObject()`** method.

   This needs 3 parameters for the method to work:
   1 - the new actor you want to add
   2 - how far across the screen you want to put it (x coordinate)
   3 - how far down the screen (y coordinate).

   You can see from the constructor that the world is 600 by 400 cells so the x coordinate can be between 0 and 600 and the y coordinate can be between 0 and 400 e.g. if we wanted the ship to start at the centre of the world we could write the following code:

```
addObject (new Ship(), 300, 200);
```

3. In the same way we can add submarines and crates wherever we want. An example of adding both is given below but you will want to add more.

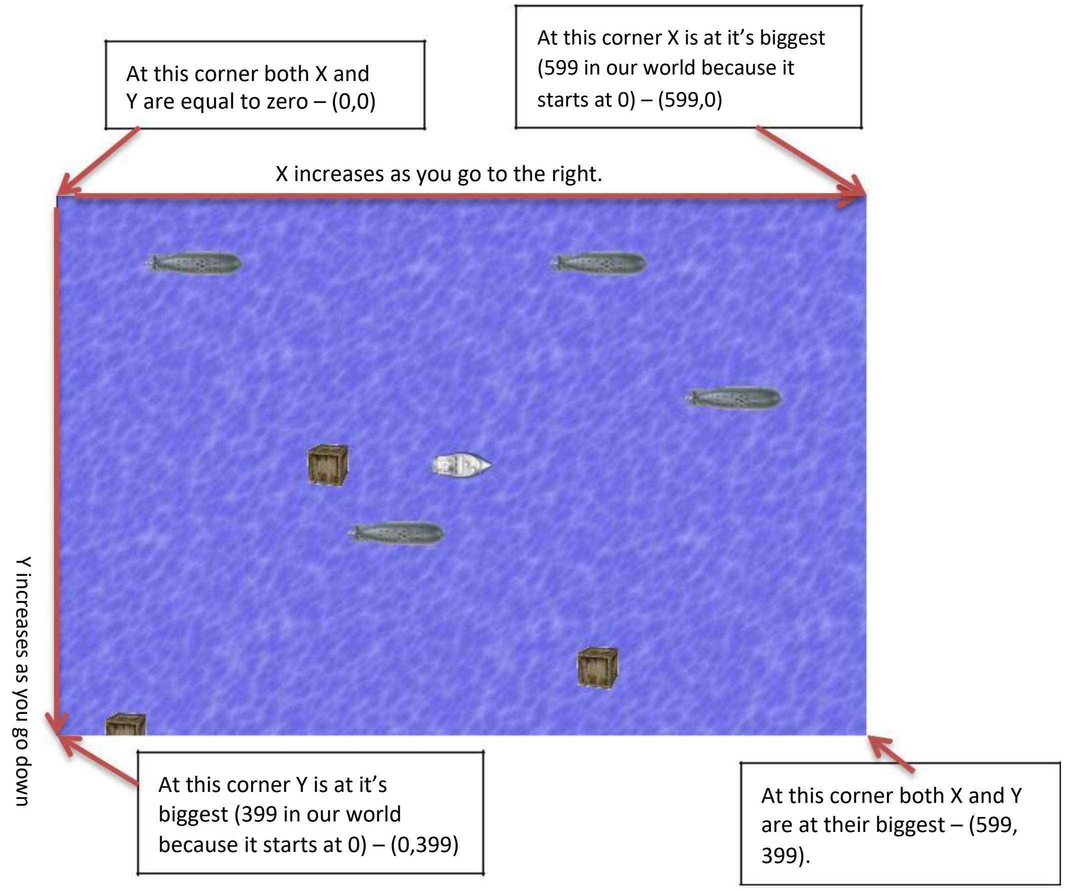
```
addObject (new Submarine(), 100, 50);
addObject(new Crate(), 400, 350);
```

4. **Add in the ship, and however many items you want. Once you are done test the game again and see what happens.**

Optional challenge: Make your game a bit harder - get the submarines to grab the crates if they get to them first. All you need to do is to add some collision detection code to the Submarine class to find when it hits a crate.

Make the Submarines move randomly

To get the submarines to turn we need to be able to tell when they reach the edge of the screen and we can only do this by looking at their X and Y coordinates. Remember we have already seen that our world is defined as being 600 cells wide by 400 cells high.


At this corner both X and Y are equal to zero – (0,0)
At this corner X is at it's biggest (599 in our world because it starts at 0) – (599,0)
X increases as you go to the right.
Y increases as you go down
At this corner Y is at it's biggest (399 in our world because it starts at 0) – (0,399)
At this corner both X and Y are at their biggest – (599, 399).

So what we want to do is tell the computer to turn the submarine 180 degrees if the X coordinate is 0 (which means it is on the left hand side) or 599 (which means it is on the right hand side). To do this we must use an if statement and the code should look like this:

```
public void act(){
        move (5);
        //gets current pos of the actor
        int xCoord = getX();
        //checks the sub hasn't gone off left or right of screen
        if (xCoord <= 0 || xCoord >=599)
        {
                turn(180);
        }
}
```

Put this code into the Submarine class and test it. If it works then your subs will be bouncing back and forth across the world.

While this is an improvement it isn't good enough because they are too easy to dodge. What we need is for the subs to turn unpredictably when they hit the edge.

How to use random numbers

There's another library method for this. If we want a random number between 0 and 9 we would write: **`Greenfoot.getRandomNumber(10);`**

So, instead of always turning the submarines 180 degrees we could turn them 160 degrees plus a random number between 0 and 40. This will mean that every time a ship hits the edge it will turn a random number of degrees between 160 and 200 degrees.

To do this **change the code where it says `turn(180);`** to the following.

```
int random = Greenfoot.getRandomNumber(41);
turn (160 + random);
```

Once you have changed the code **test it and see what happens.**

There is still one problem –the ships don't turn around when they hit the top or bottom; we never checked the Y coordinates anywhere.

The code to do this will be almost the exact same as that we have just created to check the X coordinates. So, **create the code to check if the Y coordinate is less than 0 or greater than 399 and turn the ship if it is.**
This code should go in the submarine act method just below the code that checks the X coordinate.

You should now have a fully working game - you might find the game is too easy or too hard. **If that is the case then change the parameters of the game**. Things you could try changing are the speed of your ship/the submarines or changing the number of the submarines that you start with.

Run Out of Crates?

You might quickly run out of crates to collect. It would be better to move crates to another position instead of removing them from the game completely.
To do this we must **look at the code in the Ship class that removes the crates.** We need to remove the lines in red where we use the `removeObject()` method.

```
Actor crate = getOneIntersectingObject
(Crate.class);
if (crate != null){
        World shipWorld = getWorld();
        shipWorld.removeObject(crate);
}
```

Instead of removing the crate we are going to use its **`setLocation()`** method to give it a new position in the game. We can use the random number thing we just did to make sure that the crate appears somewhere different each time it is created by generating a random x and y coordinate.
Your final code should look like this – add it in and try it:

```
Actor crate = getOneIntersectingObject
(Crate.class); if (crate != null){
      int xCoord =
      Greenfoot.getRandomNumber(600);
      int yCoord =
      Greenfoot.getRandomNumber(400);
      crate.setLocation(xCoord, yCoord);
}
```

Final challenges:

Random Startup

At the moment your ship, submarines and crates always start in the same places (the ones you set in the ShipGameWorld constructor). However now that you know how to use random numbers you should be able to go back and change the constructor so that the ship, crates etc start at different places each time you start the game. Try it on your own and see how you get on!

Bombs (super challenge)

Add code to the submarines so that they drop mines occasionally. To do this you will need to create another class (there's a bomb image in the library), add movement to the bomb and then have some hit detection for when it hits any of the objects.