Greenfoot Tutorial – Target Practice

The aim of this game is to score as many points as possible before you run out of arrows. You can score points by hitting the target, or hitting the balloons that appear. Red and blue balloons give you more points, yellow balloons give you more arrows.

Setting Up

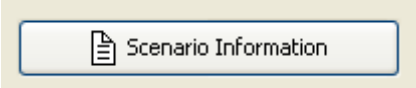
If you haven't already, download the latest version of Greenfoot at www.greenfoot.org/download and install it (at the time of writing, the latest version is 1.4.1). When it's installed, open up Greenfoot, then locate and open the invaders scenario. If you haven't been given a location for the scenario you can download it at <<URL>>. There will be a single folder inside this zip file, extract this folder and then locate and open it with Greenfoot.

When you've done the above, hit the "Compile all" button in the bottom right hand corner, and you should have a screen similar to the following:

When you see this screen you're ready to start!

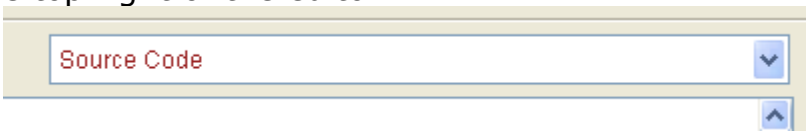
Stage 1 – Playing around

Before you dive into anything, spend a while familiarising yourself with the Greenfoot environment and the scenario. With any scenario, it's often a good idea to look at the information that's provided – so click on the "Scenario Information" button and see what comes up:



Here it should tell you the name of the project, the date it was written, the author, and it also tells you you need to follow a worksheet. That's this one!

You might also want to open up some of the classes and have a look at them. Don't be put off if you don't understand the code, but it might be helpful for later to poke around in the Documentation view of some of the classes. To switch from the source code view to the documentation view, look at the drop down box in the top right of the editor:



Next, let's take a look at what the scenario does already. To do this, hit the run button. Not a lot happening? Try dragging either side of the man. You may notice an arrow appear in the middle of him – not very useful at the moment! We need it to go somewhere.

Stage 2 – The Target class

There are a fair few classes provided with the project, but most are fairly small and / or completed for you. There are 2 main classes in the project that you need to add functionality to in order to complete this project, and the first of these is the Target class. Open it up in the editor by double clicking on it, and you should see something like the following:

```
24
25      /**
26       * Move the target up and down. Randomly create balloons in the world.
27       */
28      public void act()
29      {
30
31          /*
32           * You need to implement code here to:
33           *
34           * 1. Get the target to move up and down.
35           * 2. Randomly create 3 kinds of balloons throughout the world:
36           *       - Blue balloons
37           *       - Yellow balloons
38           *       - Red balloons
39           *
40           * The balloon classes are all provided for you complete.
41           */
42
43          //This part can be left alone. It makes sure the arrows that are st
44          List<FireArrow> arrows = getIntersectingObjects(FireArrow.class);
```

The comment in this method tells you what you need to implement, and there are 2 fields that should help you do this – one is direction, and the other is speed. Let's start with implementing the target movement up and down.

Splitting this task down further, we can think about what we want the target to actually do simply and logically. You may come up with something slightly different, but I broke it down to the following points:

- If the direction is down, move the target down
- If the direction is up, move the target up
- If the target has hit the top of the screen, change the direction to down
- If the target has hit the bottom of the screen, change the direction to up

Sounds simple enough doesn't it? The only thing we need to do now is to put this information in a form that the computer understands – in this case it involves writing it in Java.

Let's start with the first bullet point. "If the direction is down" - there's a nice way to put that in Java, and it's called an if statement! If you haven't come across them before (no pun intended) they're a simple way of executing a block of code *only* if the condition is true. An if statement is written with the condition in brackets after the if, and then with curly brackets denoting the code block. If that sounds a bit confusing, this should clear things up a bit:

```
3        if(direction.equals("down")) {
4            //move down
5        }
```

The condition is if the direction is equal to down. You may be wondering why the condition isn't just *if(direction=="down")* - after all this is what we use for numbers! The short answer is the == operator checks whether two objects are different references to the *same object*. The equals method simply checks whether the values of two different objects are equal, which is what we want here.

Now we've got our if statement sorted, how do we actually get the thing to move down? If you've followed other scenarios, you might be thinking something along the lines of "ah he's made another sneaky method for us that we just need to look for, call it, then it'll move down by itself no problem."

In fact, I must've been either feeling cruel or lazy today, since however much you may look you will not see a moveDown() or moveUp() method anywhere in sight. This time, you need to use a combination of methods provided in the Actor class to get the target to move. These 3 methods are namely setLocation, getX and getY. If you have a look at setLocation, you'll find it takes 2 parameters, the X and the Y co-ordinates of the location where the actor is to be moved. Since we want these relative to where the actor is at the moment, it makes sense to use these as the two parameters – but since we want it to move in the Y axis, we should add a value to Y depending on how

fast we want the target to travel. And oh look, that's where the speed variable comes in!

If you felt a bit lost in all that, you can see what I mean here. The X co-ordinate doesn't change, the Y does depending on what the speed variable is set to:

```
42
43          if(direction.equals("down")) {
44              setLocation(getX(), getY()+speed);
45          }
```

Now the down's been done for you (perhaps I wasn't feeling so cruel after all?) You need to implement the same thing, but for going up (that's the second bullet point on the list.) When you're done, compile and see what happens. The target should move down, since this is the default direction – but it hasn't got any code telling it to change direction at any point yet, so that's what we need to do next.

So how do we go about getting it to change direction? If you looked at the bullet point and screamed out "It's another if statement!" you'd be right. You'd also be right if you thought something along the same lines a bit more quietly in your head, but that's just no fun!

You may have figured out the easy part already, and realised that if it's at the bottom of the screen, you need to set the direction to up. You may fill in the if statement but leave it blank, and in the code to be executed set direction to be up, much like the following:

```
50          if(/*Something...*/) {
51              direction = "up";
52          }
```

If so, congratulations. You've done half, and just need to work the condition out. And before you ask – no I haven't given you a nice easy method to call for this one either. But I have given you a 3rd field that might point to the class you need to look in for appropriate methods. Open up the world class and have a poke around for the getHeight method. Here's a massive hint – it corresponds exactly in units to the getY method you looked at earlier. So rephrasing the bullet point slightly to incorporate these methods, we want to look and see if getY is greater than world.getHeight minus some offset value if we don't want the taret going all the way to the edges of the world. You can choose whatever you like for this value, and you should end up with something like the following:

```
50          if(getY()>getWorld().getHeight()-80) {
51              direction = "up";
52          }
```

You then need to implement something similar when it gets to the top of the world. This time however, you don't need the height of the world in the if statement, it's actually simpler than the one above. Have a think until you've got it, then implement it. Compile and run your program to check it, and the target should move up and down!

Let's move on to the balloons next. First, we want them to appear *randomly.* So before we go any further, let me introduce you to a method that spits out random numbers at will. You can use it by calling *Greenfoot.getRandomNumber(int limit)*. The method will then return integers between 0 and limit-1 each time it's called.

Firstly, we don't want balloons to appear each time the act method is called, so we need another if statement. But this time, we want the code to execute every so often randomly. Doesn't sound like something that can be done with an if statement? Look at this:

```
57          if(Greenfoot.getRandomNumber(500)==0) {
58
59          }
```

Essentially what this says is, generate a random number between 0 and 499, and if it's 0, execute the block of code. In practice, this means that every time the code runs there's a 1/500 chance of the block executing. You don't have to use 500, a lower number will mean more balloons and a higher number will mean less.

So what do we want inside this if statement? Well, we could just say something like *world.addObject(new RedBalloon(), 100, 100);*. Would this work? Sure it would. But it'd create red balloons in the same place every single time, we want them created all over the world randomly. Try it if you like! Those numbers 100 we want to replace with a random range first for the width of the world, then for the height. Instead of doing the whole if statement for you this time, I'm going to leave you with this line of code:

```
61          Greenfoot.getRandomNumber(getWorld().getWidth()-100)
62
```

Think about what it does and how you can integrate it into the world.addObject method to create balloons at random positions. Once you've got it sorted, test your code and then create similar if statements for the other types of balloons – blue and yellow.

When you've done this, the target class is complete!

Stage 3 – The FireArrow class

The second class that needs modifications for the program to work is the FireArrow class. This one controls the behaviour of arrows – and at the moment they don't do a lot! Open it up and look for the description of the code you need to write:

```
40      /*
41       * First and foremost, the arrow needs to go somewhere.
42       * There's a nice method provided below called move. That will move the arrow for you,
43       * as well as controlling its rotation.
44       *
45       * However, the arrow needs to stop moving sometimes, namely when it hits the target!
46       * It also needs to be removed from the world when it's gone out of bounds. The remove() method is provided for removing it from the world.
47       *
48       * Arrows also do not last forever. After their life has expired, they need to be removed as well.
49       * The life field starts at 500. It should decrease by 1 each turn, and be removed when it reaches 0.
50       */
```

This class is easier than the target class to write, so you'll be pleased to know the hard bit's out the way. However, there's still more to be done before the game is playable.

Firstly, try just running the code with the move method. What happens? It should work fairly nicely, however first let's concentrate on the fact we want the arrow to stop moving when it hits the target. So we only want the arrow to move if it is not intersecting with a target. The condition you need is as follows:

```
61          if(getOneIntersectingObject(Target.class)!=null)
```

As for what goes in the body, you need to work that out yourself.

When you've tested that (and make sure you have hit the target before you think your code's not working!) move onto the arrow's life. You can decrease it's life by one every time by calling *life--;* (or *life = life -1*, they do exactly the same thing.) As for when the life is 0, you should be able to work that out yourself, and the remove() method is provided for you.

Lastly, we need the arrow to be removed if it's out of bounds. There's no provided method that checks whether it's out of the world, so you'll have to form your own if statement from a few methods that should come in useful – world.getWidth(), world.getHeight(), getX(), and getY() would be good places to look.

When you've done this, congratulations, you've finished and you should now be able to play the game! Test it to make sure everything works as it should.

Stage 4 – Adding extra stuff

You could leave it there. The game's perfectly playable – but there's always more you can add! Here are some ideas to get you going:

- Sounds (look at the Greenfoot.playSound method, this is very easy to do)
- More obstacles that stop arrows
- Other types of balloons
- Move the target round in a circle rather than just back and forth
- Take points away for every arrow that misses
- Anything else you can think of!