Air Raid Game

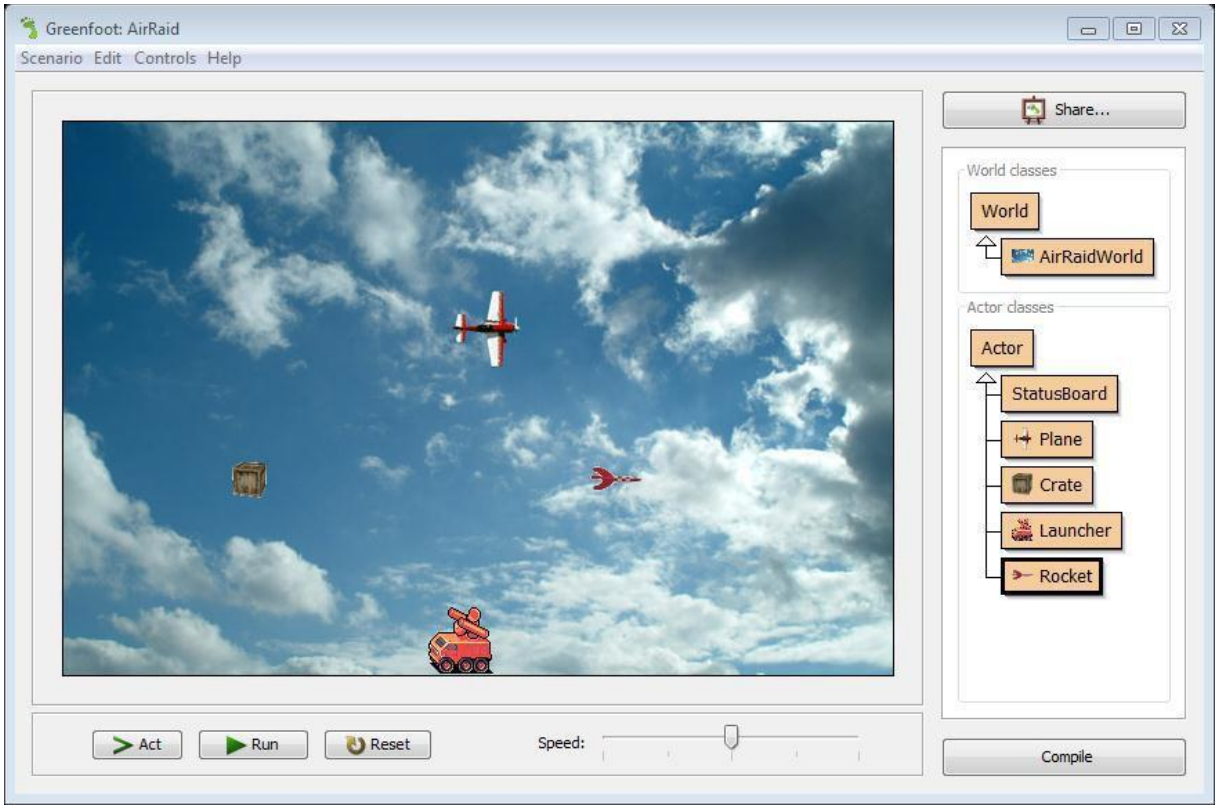
In this game airplanes will fly across the top of the screen dropping supplies to enemy troops. You are in charge of a rocket launcher and it is your responsibility to destroy the crates before they hit the ground.

Save your own version of the **Session 2 - Air Raid** folder from Files on your class Team.

Step 1

Create classes for each type of Actor object that we are going to have in the game. We will need at least the 4 classes here: Plane, Launcher, Rocket, Crate.

Create a new subclass for each of these objects. The images have already been provided for each of them.



Moving the Launcher

In this game we only want the launcher to **move left and right along the bottom of the screen**.

Add the code to the act method of the Launcher class.

1. Write an if statement to check if the right arrow on the keyboard is being pressed exactly like the example here. →

```
if ( Greenfoot.isKeyDown("right") )
{
        // Add your code to do something
        // if the right arrow is pressed here.
}
```

2. If the right arrow is being pressed we want to move the launcher 5 cells – note that unless you turn an actor first the move method will move the actor towards the right of the screen.

3. **Write an if statement to check if the left arrow on the keyboard is being pressed** - if it is, we want to move the launcher 5 cells backwards (a negative number).

4. **Test that the code works and that you can move the launcher.**

Moving the Rocket

When we launch a rocket we want it to go **straight up until it hits the top of the screen**.

In the **Rocket class' act method write a single line of code telling it to move** e.g. `move(10);`.

When you test this code you will notice the rocket goes straight to the right edge of the screen rather than to the top. We need to turn the actor first.

Create a constructor for when we create a new object from the Rocket class – a constructor is called when we instantiate a new object. It is a procedure that builds it – setting any **attributes** (variables) with their data.

Constructors are usually given the same name as their class (sometimes with a lowercase first letter instead).
Add the following code before the `act()` method and test it works.

```
public Rocket ()
{
    setRotation (270);
}
```

Removing Rockets

We already know we can get the world we are in by calling **`getWorld()`**, then using **`removeObject()`** to remove something from the game.
Add this code to the `act()` method of Rocket.

```
int yCoord = getY();
if (yCoord <= 0)
{
        World airWorld = getWorld();
        airWorld.removeObject(this);
}
```

Firing from the Launcher

Every time we press space a new rocket should be added to the game where the launcher is at that time.

In the Launcher class `act()` method….

1. **Write an if statement that checks if the space key is being pressed** - if it is create a new variable called airWorld (like the 3rd line above shows) and store the return value from calling the getWorld() method into it.
2. **Use the addObject() method as shown on the right to add the rocket to the game.** Notice how we use getX and getY to find the current position of the launcher and use that as the starting position for the rocket.

```
airWorld.addObject(new Rocket(), getX(), getY());
```

3. Test it – it will work but not perfectly.

What is happening is that the act method is called so quickly that it is almost impossible to press and let go of the space key without more than one rocket being fired. The simplest solution to this is to **introduce a time delay** between firing rockets.

Add a class variable to hold a delay counter - declare it as shown below just after the start of your Launcher class. It will be an integer because we just want it to hold numbers between 0 and 25. We say it **is private** because we don't want any other classes using this information. **Finally we give it a starting value of 0**.

```
public class Launcher extends Actor
{
   private int delay = 0;

... rest of your class
```

2. **In the `act()` method check if the delay counter is greater than zero - if it is reduce the value of the counter by one** (see example below).

3. The final step is to **set the delay to 25 once you fire a rocket**. By changing this number, you will change the delay between rockets. The final code is shown below including the code for firing a rocket that you already made (**this should replace that code** otherwise you will be firing twice).

```
if (delay > 0)
{
        delay = delay - 1;
}
else
{
        if (Greenfoot.isKeyDown("space"))
        {
                World airWorld = getWorld();
                airWorld.addObject(new Rocket(), getX(), getY());


                delay = 25;
        }
}
```

Falling Crates

Having got the launcher and rockets working we now need to **change the Crate class so that crates fall to the bottom of the screen and disappear when they hit the bottom**. The code will be almost exactly the same as the Rocket class, the main differences are that you want to set the rotation to 90 degrees and you will want to remove the crate when the Y coordinate is greater than or equal to 399 (as opposed to less than or equal to zero for the rocket).

Using the Rocket class as a guide try and edit this code yourself and test that it works.

Moving the Plane and Dropping Crates

Edit the Plane class. We want the **plane to go back and forth across the top of the screen**. The final code in the planes act method should look like this:

```
move (5);

int xCoord = getX();
if (xCoord <= 0 || xCoord >= 599)
{
        turn (180);
}
```

We now need to decide how often we want the plane to drop a crate. We could use a delay but a better solution would be to drop them randomly. We can do this by generating random numbers and only dropping a crate if the number is a 1. So if we generated a random number between 0 and 100 then there is a 1% chance that we will drop a crate each time the act method is called. The if statement for this would be:

```
int rand = Greenfoot.getRandomNumber(100);

if (rand == 1)
{
        // Get the world and add a new crate
        // the same way you added a new
        // rocket for the launcher.
}
```

Using the code for creating rockets in the Launcher class as a guide you should be able to **add the code to make a new crate to this if statement and add the completed code to the act method of the plane**. Once you have done it, test it works.

Collision Detection

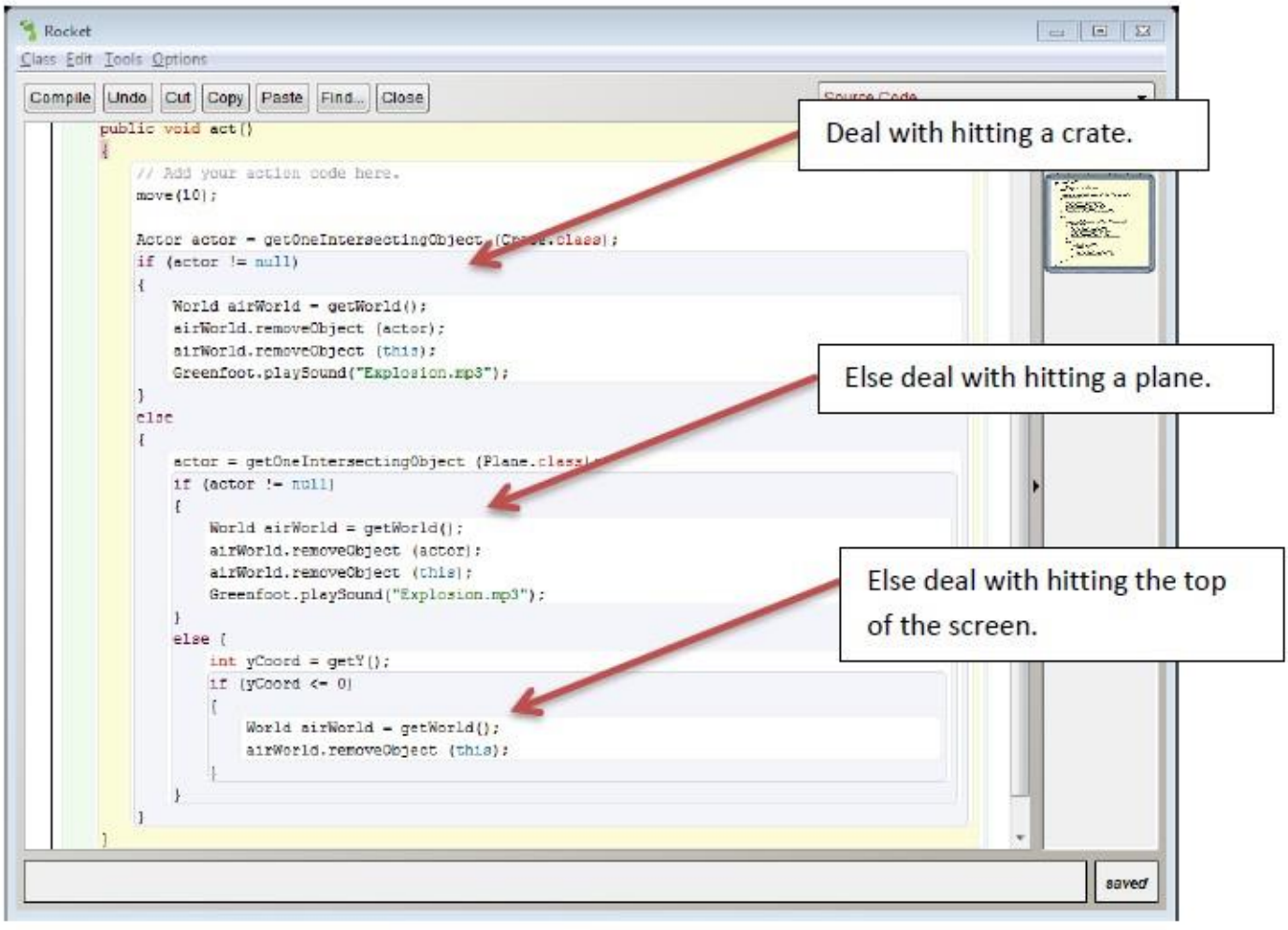
Checking if the rocket is hitting a crate or plane should be something you are familiar with. However, because we now have more than one condition that can cause a rocket to be removed from the game we have to make sure we don't try and remove it twice as this will cause our game to crash.

In general, the code to check if the rocket has hit another object and remove them both from the game will look like this (for this example I have used the Crate class). To **make sure we don't remove the rocket twice** the rest of the code will be put in an else block so it is only executed *if you haven't hit a crate.*

```
Actor actor = getOneIntersectingObject (Crate.class);
if (actor != null)
{
        World airWorld = getWorld();
        airWorld.removeObject (actor); // remove the crate
        airWorld.removeObject (this);   // remove the rocket

}
else
{
        // Code to check if you have hit the plane or the top of
        // the screen goes here.
}
```

The final version of the act method in my version of the Rocket class is shown below – finish off your code so that it works like this too:



Automatic Setup

Double click on the **AirRaidWorld class** and **add the following two lines of code to the constructor** so that the items are put into the world automatically:

```
addObject (new Launcher(), 300, 375);
addObject (new Plane (), 100, 30);
```

Keeping Score (PASSING DATA BETWEEN CLASSES)

The best place to store score information is in the AirRaidWorld class as the score is something that applies to the entire game. To do scores properly **we will need to provide a variable to store the score in, a way to increase the score and a way to check what the current score is**.

1. **In the AirRaidWorld class add a new private property called score**. It is private because we want other classes to have to ask us to change the score rather than then changing it themselves – it is **ENCAPSULATED** (hidden inside).

```
public class AirRaidWorld extends World
{
   private int score = 0;

... rest of your class
```

2. We are now going to **write a method that can be used to increase the score**. This method must **be public** because we want it to be used by other classes.

```
public void increaseScore ()
{
        score = score + 1; // could just write score++; which does the same thing.
}
```

3. The final thing we need to do is **provide other classes with a way of seeing what the current score is**. To do this we are going to **add another public method**. this time an **int function** as the method is going to return the score value.

```
public int getScore ()
{
        return score;
}
```

Now that we have a way of keeping score we need to **increase it when we hit a crate or plane**. To do this we must **go back into the Rocket class** and change our collision detection code a little.

What we need to do is call the **`increaseScore()`** method that we have just made when we are removing the objects from the game.

The problem is that we have said that airWorld is made from the World class and the increaseScore method is only in the AirRaidWorld class. **So what we need to do is tell the computer that the world we are in is actually made from the AirRaidWorld class.** **We do that by putting (AirRaidWorld) in front of the getWorld method.**

An example of the complete code is shown in the box below. You will need to **change this in two places** in the Rocket class – once for when you are checking for crates and once for when you are checking for planes.

```
Actor act = getOneIntersectingObject (Crate.class);
if (actor != null)
{
        AirRaidWorld airWorld = (AirRaidWorld)getWorld();
        airWorld.removeObject (actor); // remove the crate
        airWorld.removeObject (this);   // remove the rocket
        airWorld.increaseScore();
        // play sounds or anything else
}
```

As we haven't hooked up our score board yet the easiest way to check the score is working is to play the game for a while and then pause it. You can then right click on the background and choose **getScore()** to see what your current score is.

Tracking Lives

Tracking lives is very like keeping score although instead of counting up you are counting down.

As with the score, the number of lives you have left should be stored in the World and **we will need methods to both decrease the number of lives and get how many you have left**. The code is almost exactly the same as for the score. **Add the code shown here to the AirRaidWorld class:**

```
private int lives = 5; // number of lives to start with.

public void loseLife ()
{
        lives = lives – 1; //could write lives --;
}

public int getLives()
{
        return lives;
}
```

Open the Crate class and find where you check if a crate has hit the bottom of the screen. We need to change this code in a similar way to the way we had to change the collision detection to keep the score.

We must first tell the computer that we want an AirRaidWorld and then add a line calling the loseLife() method. The altered code should look like this:

```
if (yCoord >= 399)
{
    AirRaidWorld World airWorld = (AirRaidWorld)getWorld();
    airWorld.removeObject (this);
    airWorld.loseLife();
}
```

At the moment the number of lives you have can go into negative figures. This is because we have never told the game to stop when we reach zero.

The easiest place for us to do this **is in the `loseLife()` method**. All we need to **add is a simple if statement that checks if the number of lives left is zero and calls the Greenfoot.stop method** if it is. The following lines of code are all you need to add. If you test this code the game should stop after 5 crates have hit the bottom of the screen.

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

Using the Scoreboard

Now that we are keeping track of the score and how many lives are left we want to display it on screen.

The StatusBoard class has one method that we are interested in called **`updateStatus()`** which simply takes the score and number of lives to display on the screen. We will need to call this method every time we change the score or the number of lives we have left.

We need to **create the score board** from the class **and put it into the world** class.

1. **Create a new private class property at the top of the AirRaidWorld class** to hold the score board in memory.

```
private StatusBoard scoreBoard;
```

2. **In the AirRaidWorld constructor make the new score board and put it into the world** by calling the `addObject()` method. The code is shown here - you may need to edit the addObject values to **35,** 370.

```
scoreBoard = new StatusBoard ();
addObject (scoreBoard, 0, 370);
scoreBoard.updateStatus (score, lives);
```

3. **Add calls to the `updateStatus()` method into both the `increaseScore()` and `loseLife()` methods**. The line you need to add to ***both*** methods is shown below.

```
scoreBoard.updateStatus(score, lives);
```

4. Copy the StatusBoard.java text from the file in the Session 2 folder (where this PDF is) to replace the empty class in your code. Test it and make sure that it works.

Challenge Tasks

Experiment with:

- the speed of the rocket launcher
- how frequently crates are dropped
- the speed of the plane

You could also go back to the Ship Game from last lesson and see if you can implement some of the lives and scoring features to it, and add a scoreboard.