Swarming Robots: Greenfoot Worksheet

The aim of this worksheet is to get you started in exploring the world of swarm robotics. You have been given a swarm robots “framework”, and this list of tasks to achieve this aim. The framework consists of some basic classes that contain necessary code that is needed to bring some of these algorithms to life. However the Greenfoot project is basically empty so it is up to you to bring these swarms to life.

Look at some of the methods in the SwarmRobot, FireflySuper, and BoidSuper classes. Think about what methods can be used from these superclass by fireflies and boids.

Don’t worry if they look complicated you do not have to understand it all. These are a great example of abstraction, the idea that complicated parts of a program can be hidden from other classes. This allows you to use a method or function without knowing exactly how it is implemented. You know how to use the move() function in Greenfoot but you don't need to know exactly how it works. This is a key concept of computer science, unnecessary complexities can be abstracted out.

If you feel unsure about what code is required or simply require a reminder of some of the Greenfoot and Java methods then look at the cheat-sheet that has been given to you. The cheat-sheet also contains tips on methods used by the swarm robots.

Now go and create your own swarming robots! We will be starting with the firefly algorithm.

Firefly Synchronisation

Page break image



The pseudocode for the algorithm is as follows:

**count** from 1 to 12 repeatedly in a cycle

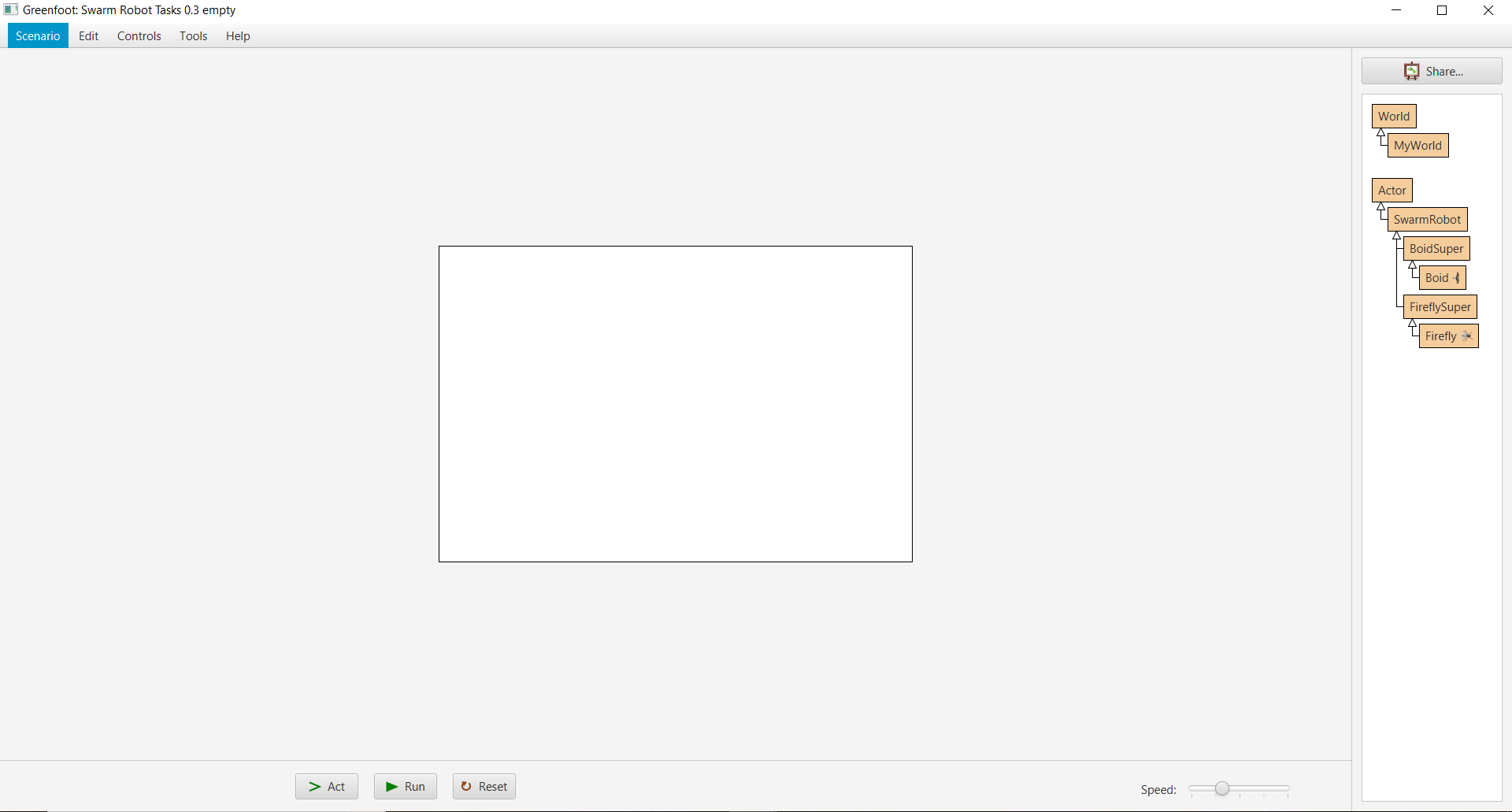
**when** the clock is greater than 12 flash and reset clock

**if** a firefly is not flashing but detects another flies flash

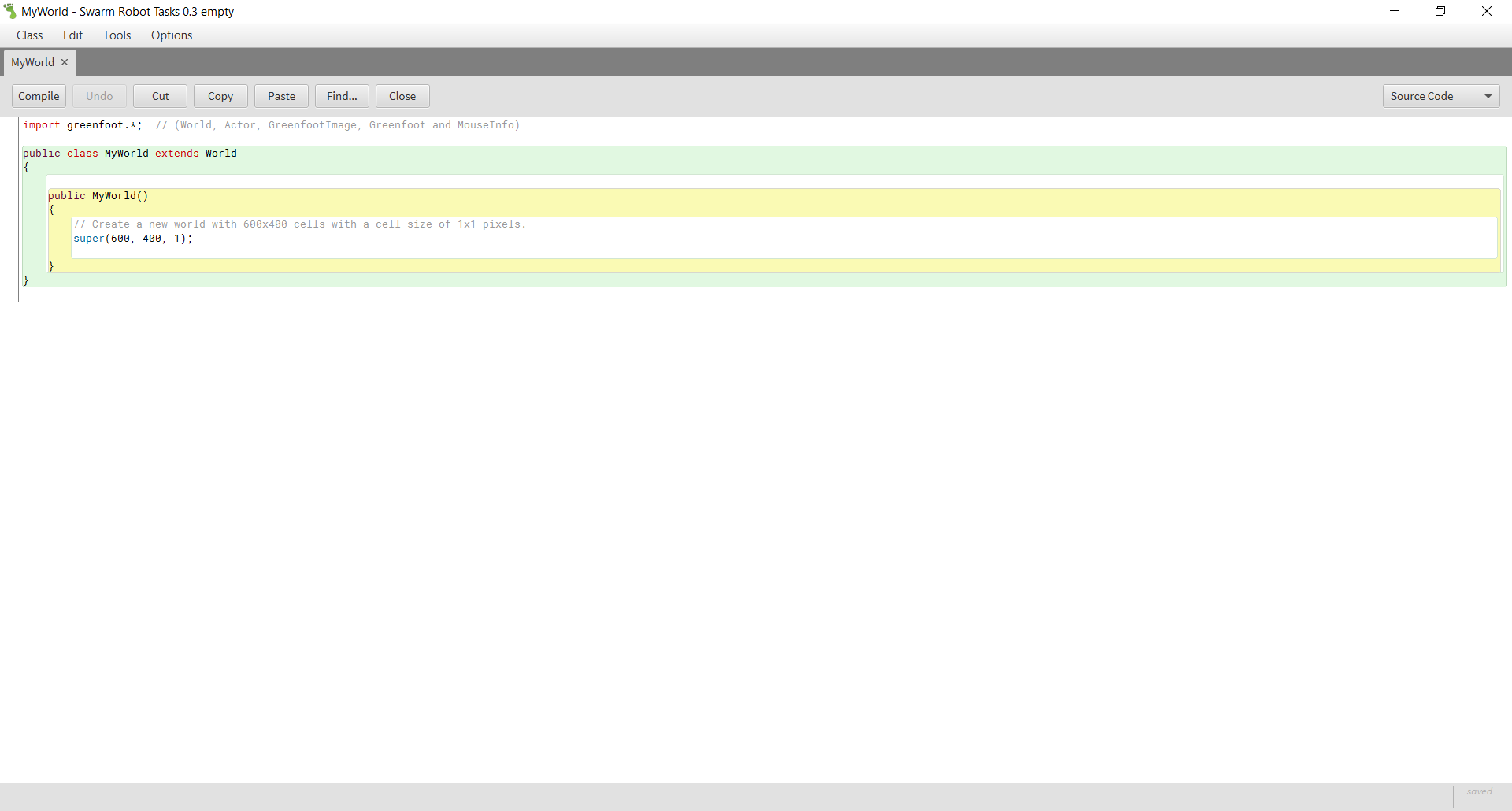
increase firefly’s clock by 1

## Task 0

First launch Greenfoot and open the Swarm Robot Tasks Greenfoot project, which should look like this.

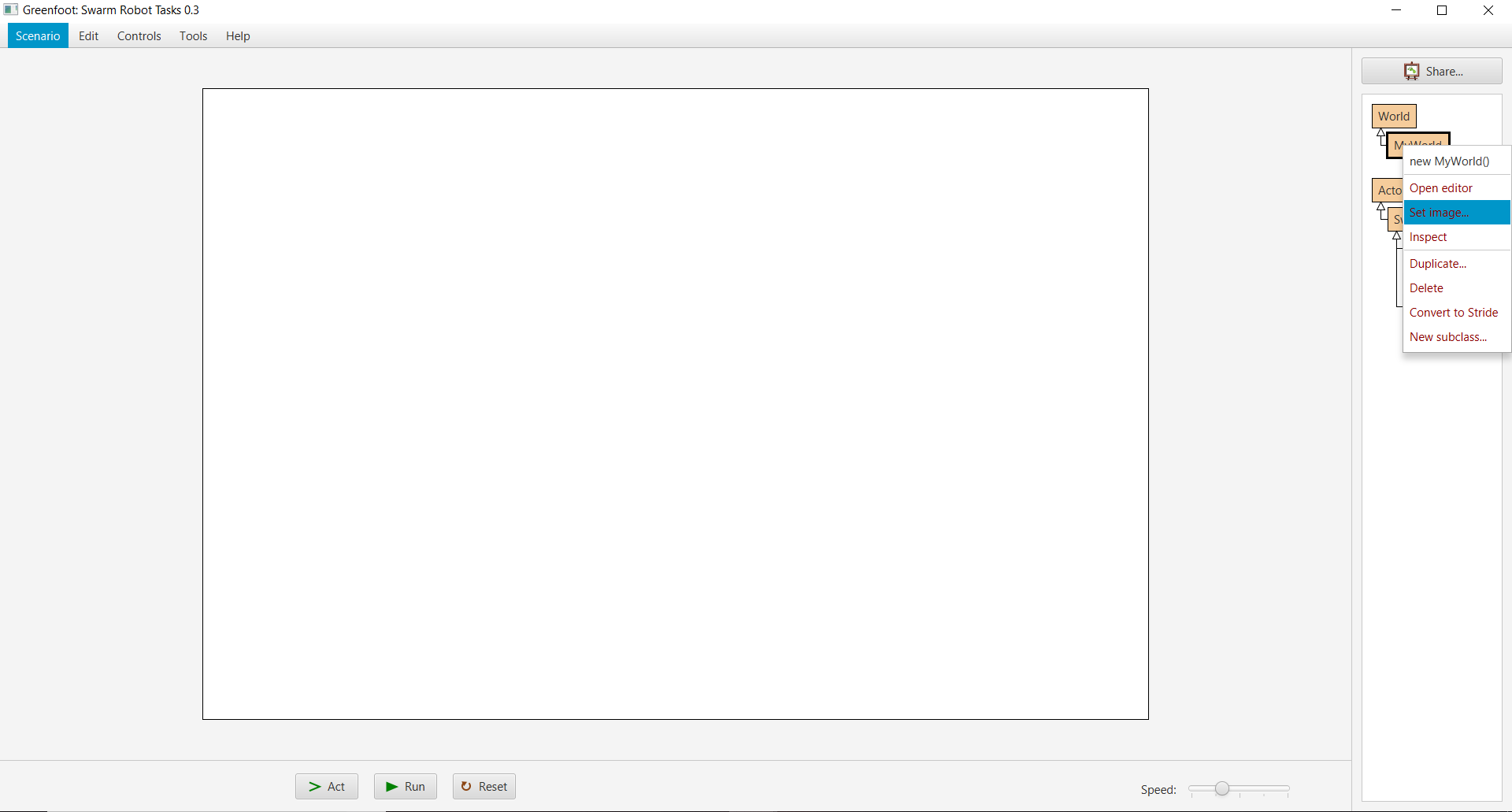


Next open the empty class MyWorld, at the moment it should be look like this.



Currently the world is 600 cells wide and 400 cells tall. This is really small and we want room for the fireflies to move around, double these values to 1200 and 800. Check that your world is now bigger.

Change the background image to an image of your choice, I would suggest the nightSky image as it is relevant to fireflies. You can change the background image by right clicking MyWorld and selecting Set image, then click the reset button to update the background.



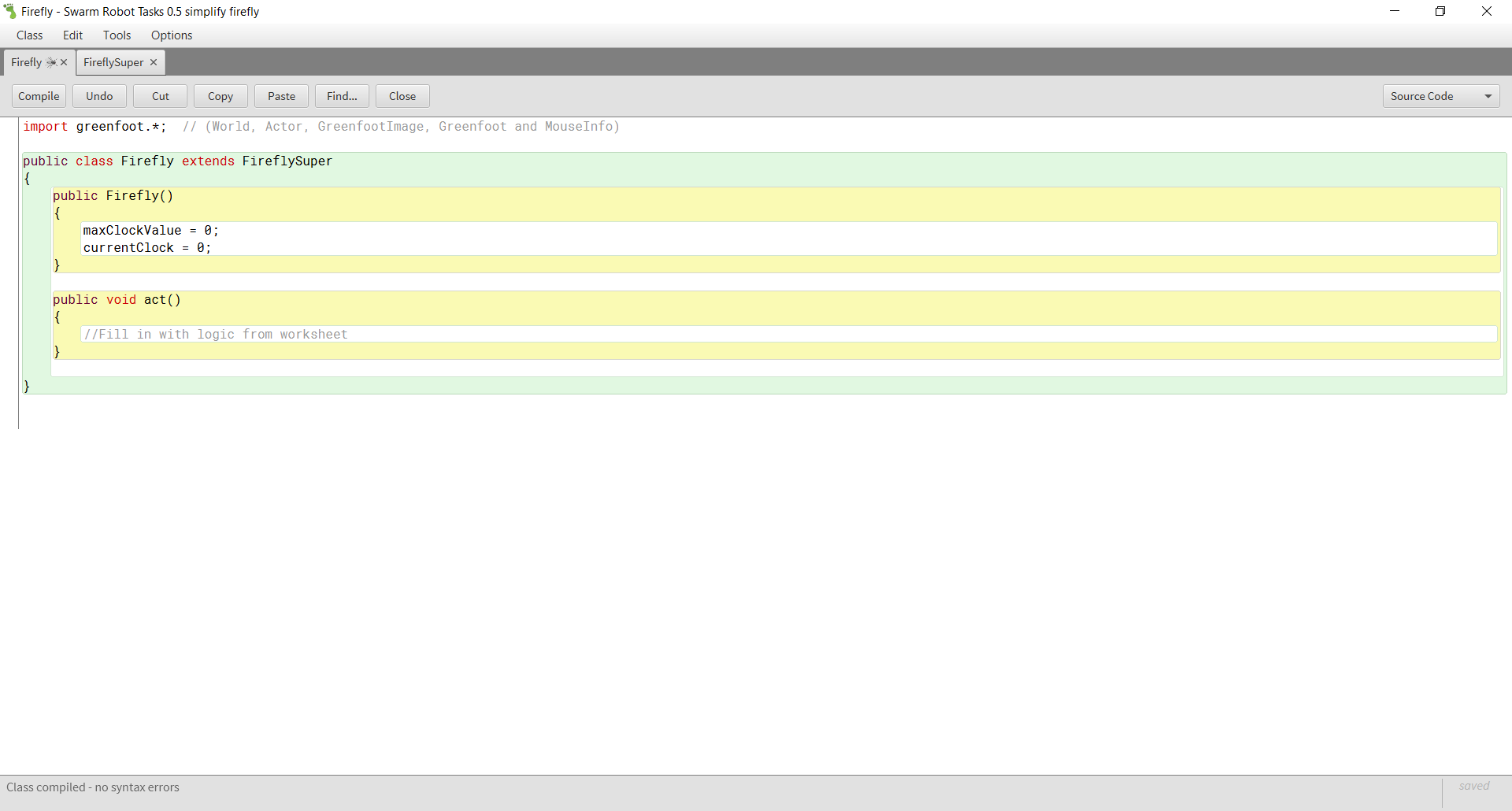
## Task 1

Now before you can do anything with the fireflies you must first populate the world with 10 Firefly actors. Go to the MyWorld class and use a for loop to populate the world with Firefly objects. The x, y locations, and the rotation of each new actor should be random. If you’re unsure about any Greenfoot methods check the cheat-sheet for a reminder.

Your world should now look like this, populated with fireflies.

## Task 2

Open the Firefly class. It should look like this.



You can see that it is mainly empty but contains two variables, maxClockValue and currentClock. These variables are not initialised here but rather in the FireflySuper class, hence why there is no int keyword before the variable name.

Add extra functionality to your swarm robotics simulation by making your firefly move like a bug. Think about how much it should move each act and what your fly should do when it reaches the edge of the world.

There exists a method in the FireflySuper superclass called crawlAround().

In the Swarm Robot superClass there are also 2 alternative methods that will control what your firefly will do when it reaches the edge of the world, these are bounceOffEdge(), and loopThroughEdge(). Read the cheatsheet to see what each method does and decide which method you would like to use.

## Task 3

Currently the maximum clock value is set to 0 with the code

maxClockValue = 0;

Decide on a more reasonable maximum clock value and think about why you have chose this value. Remember the clock analogy where a clock can get to 12 before going back to 1. You cannot have a clock with no numbers on it. How will a higher or lower maximum value effect the time needed to synchronise? Comment your code with what you think.

## Task 4

The value for currentClock is currently set to 0, however this is incorrect. The value of currentClock should be set to a value between 1 and the maxClockValue, using the cheat-sheet look at how we can assign random numbers to a variable. We want the lowest possible value to be 1 and not 0 ensure that you increase the result of the random number by 1 to achieve this. Be careful not to set the current clock value to greater than the maxClockValue.

## Task 5

Look at the pseudocode for the firefly algorithm, a firefly’s clock should increase every act cycle until it is greater than a maximum value, then it should reset back to 1. In the act() method write code to increase the variable currentClock by 1 each act cycle. If the current clock value is greater than the maximum clock value then reset the clock to 1. Add this feature to your program. In Java adding 1 to a variable is known as incrementing it and can be done with the code.

variable++;

## Task 6

Make the firefly flash when its clock has reached its max value. What's needed to do this? The default image is “firefly.png”, which is just a normal grey firefly.



to make the firefly flash the image should be set to “fireflyFlash.png”. This is the same as the normal firefly image but it is coloured yellow to appear like it is flashing.



The image should be set to “fireflyFlash.png” when the clock is equal to or greater than the maximum clock value. Otherwise the image should be set to “firefly.png”.

Run your project for a couple of seconds, you should now see that the fireflies will flash randomly based on their randomly assigned clock values. Each time you run your project this flashing should be completely unique. It should look like this.



## Task 7

The code “neighbourFlyFlashing(1000)” from the FireflySuper super class will return true if any neighbour fly is fashing within 1000 cells. Following the algorithm pseudocode make the firefly “skip a number” when it detects a neighbour firefly is flashing.

The clock should already be incremented every act cycle, if the clock is incremented twice in the same method then a fly would appear to skip a number. The firefly should only only look to see if neighbours are flashing when is not flashing itself.

Running your program at maximum speed and leaving it for a minute should result in your fireflies looking like this.



If done correctly the firefly flashing should slowly start to synchronise over time! You've now created a simulation of Swarming Robots! Sometimes the fireflies may not synchronise no matter how much time they’re given to synchronise. This can be caused by having too many fireflies or having to low of a maxClockValue. Try changing these values and run the simulation again.

## Task 8

Use the code

Greenfoot.playSound("ping.mp3");

to make the fireflies make a sound each time they flash. This will cause the fireflies to synchronise audibly as well as visually.

Currently each firefly can see other fireflies flashing within 1000 units. Change this value and experiment with it. How will a lower or higher value effect how quickly the fireflies synchronise.

Boid Flocking

Page break image



Now it is time for you to create your second swarm robotics algorithm, the boid flocking algorithm. This algorithm more complicated than the firefly algorithm, therefore more of the code has been written in the framework and given to you.

Every time the act() method is called a Boid should call the 3 flocking rules functions: separation, alignment, and cohesion. The basic simplified pseudocode for a Boid is as follows:

Boid pseudocode:

move forward

update list of neighbours

set rotation to angle average of Separation, Alignment, Cohesion

function Cohesion pseudocode:

calculate direction to centre position of neighbours

**return** direction to centre

function Alignment pseudocode:

calculate angle average heading of neighbours

**return** average heading

function Separation pseudocode:

look for neighbours in a smaller radius

calculate direction to centre position of neighbours

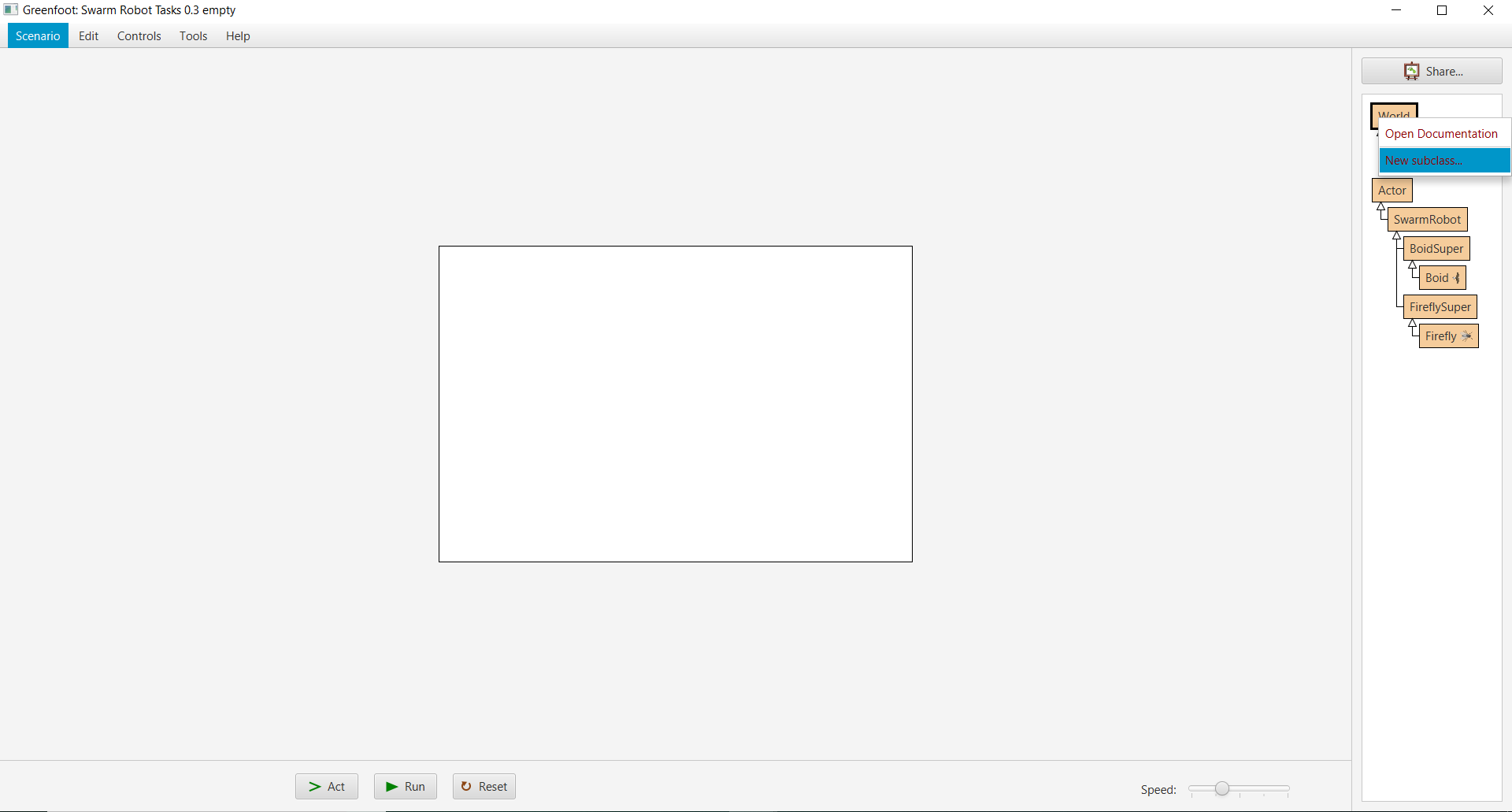
move direction to away from centre of neighbours

**Return** direction away from centre

You can view the code for the 3 rules Cohesion, Alignment, and Separation in the BoidSuper class. You do not need to know how this code works but it may be helpful to you. In these tasks you will be populating the world with boids and completing the logic for boids, not the 3 rules.

## Task 0

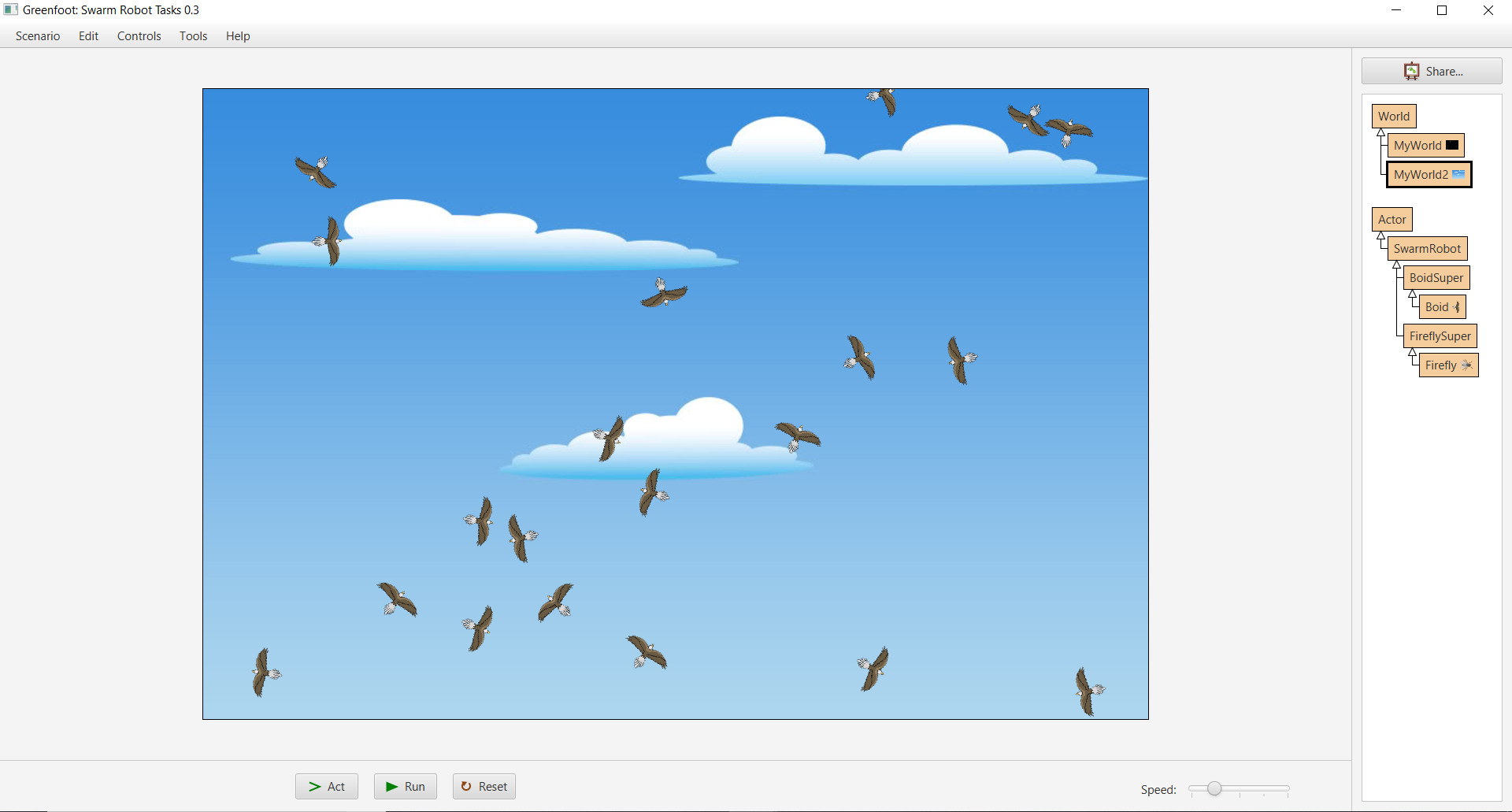
Create a new world by right-clicking on World and selecting create New subclass, name this however you like and select a background image.



Then view the world by right-clicking on the new world and selecting new world()

I would suggest the blueSky image as it is relevant to birds, which Boids are based off. Again increase the size of the world to 1200 by 800 and populate the world with Boid objects with random location and rotation like you’ve done with the fireflies in the previous task. Being able to copy and edit code for different purposes is a key skill for computer scientists. Look back at how you populated the first world with fireflies, can that code be reused and amended slightly for this task?

Your project should now look like this.



## Task 1

Now edit the Boid class to make each boid move forward 5 units each act cycle with the code

move(5);

Loop through the world using the method loopThroughEdge() so that each boid keeps the same direction of travel after hitting an edge, but will appear on the other side. The other method bounceffEdge() could be used to stop the actors from moving outside of the world, but this makes flocking harder as the direction is constantly changing.

## Task 2

Get a list of a boids neighbours within 150 cells with the code

neighbours = getBoidNeighburs(150);

neighbours is an attribute in the BoidSuper superclass so it is important to use the same name.

If a boid has no neighbours then it shouldn't turn and should continue on its current path. To do this use the if statement

if (neighbours.size() != 0)

{

//Do code only when their are neighbours.

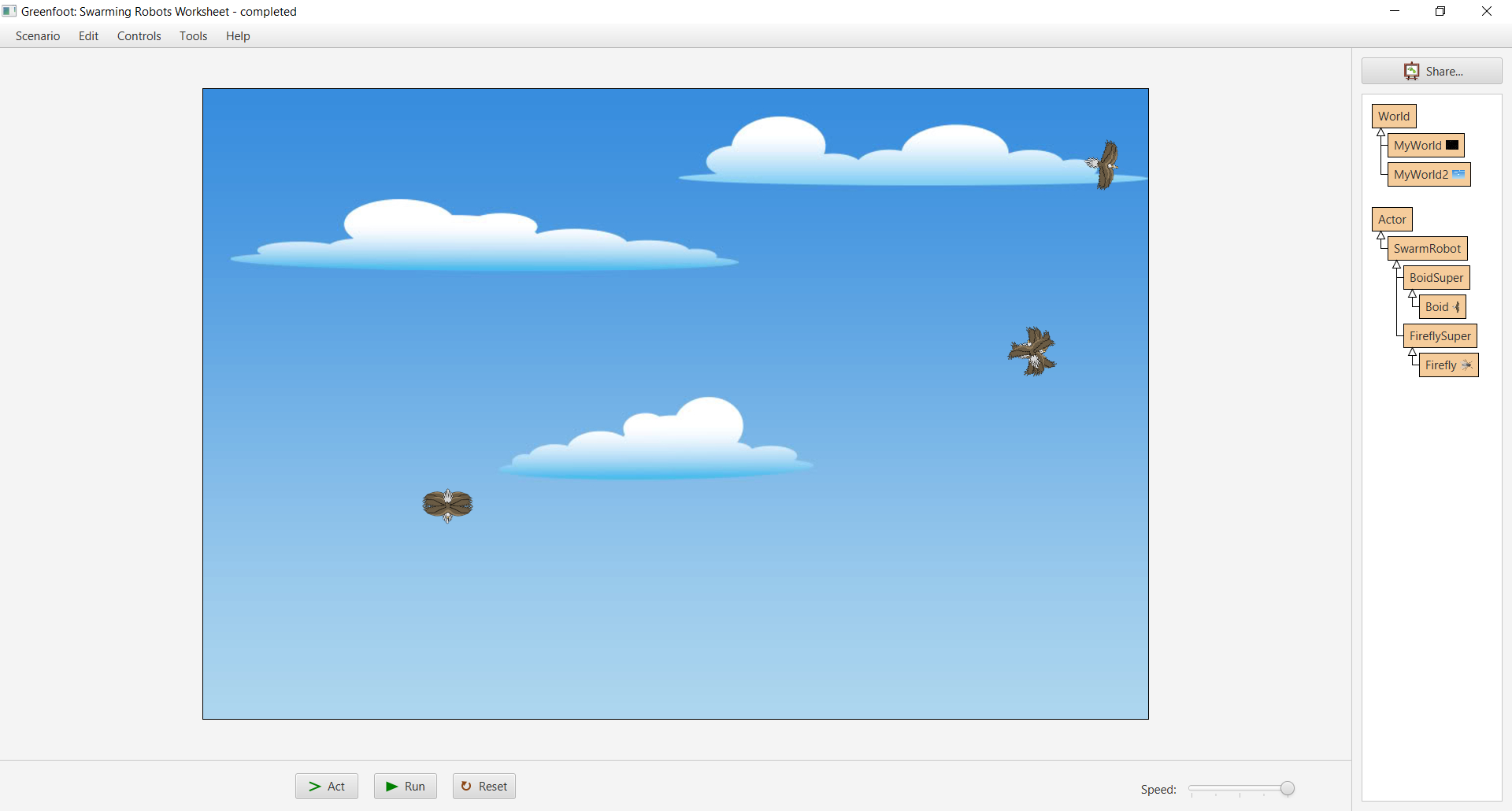
}

Next create a integer variable newHeading and set it equal to the result of the function cohesion(), then set the rotation of the boid to newHeading with the code

setRotation(newHeading);

This should only be done when a boid has neighbours.

Whats happens to the boids, is this flocking?



Your boids should quickly move together and into a few masses of boids. This is not yet flocking, although it is an important milestone to achieve it.

## Task 3

The boids have cohesion, which means that they will stay close to one another, but now they need to be able to see the direction nearby boids are flying in and match this. To achieve this the alignment rule must be used.

Beneath where you have initialised newHeading use the code

int[] intArray = {cohesion(), alignment()};

to create an array of integers called intArray that stores the result of the two functions cohesion(), and alignment().

Additionally remove the line

int newHeading = cohesion();

and calculate the average new heading from the results of the two rules with the code

int newHeading = averageOfAngles(intArray);

This should be beneath where you have defined the intArray. When dealing with circular values such as the directions in Greenfoot or compass angles calculating the mean of the angles will not give the correct average, however the BoidSuper function averageOfAngles will.

The boids should still not flock, but instead group together in multiple small groups.

## Task 4

Currently your boids instantaneously snap to a new heading whenever a new neighbour comes into range, which is unrealistic and boring. Instead, make the boids turn by only 1 degree per act() cycle either clockwise or anticlockwise. To do this you will need to replace

setRotation(newHeading);

with the method

turnSlightly(newHeading);

from the BoidSuper superclass.

Now your boids should slowly turn together rather than doing so instantaneously. This should lead to boids changing to match eachothers headings rather than instantly being attracted to each other.

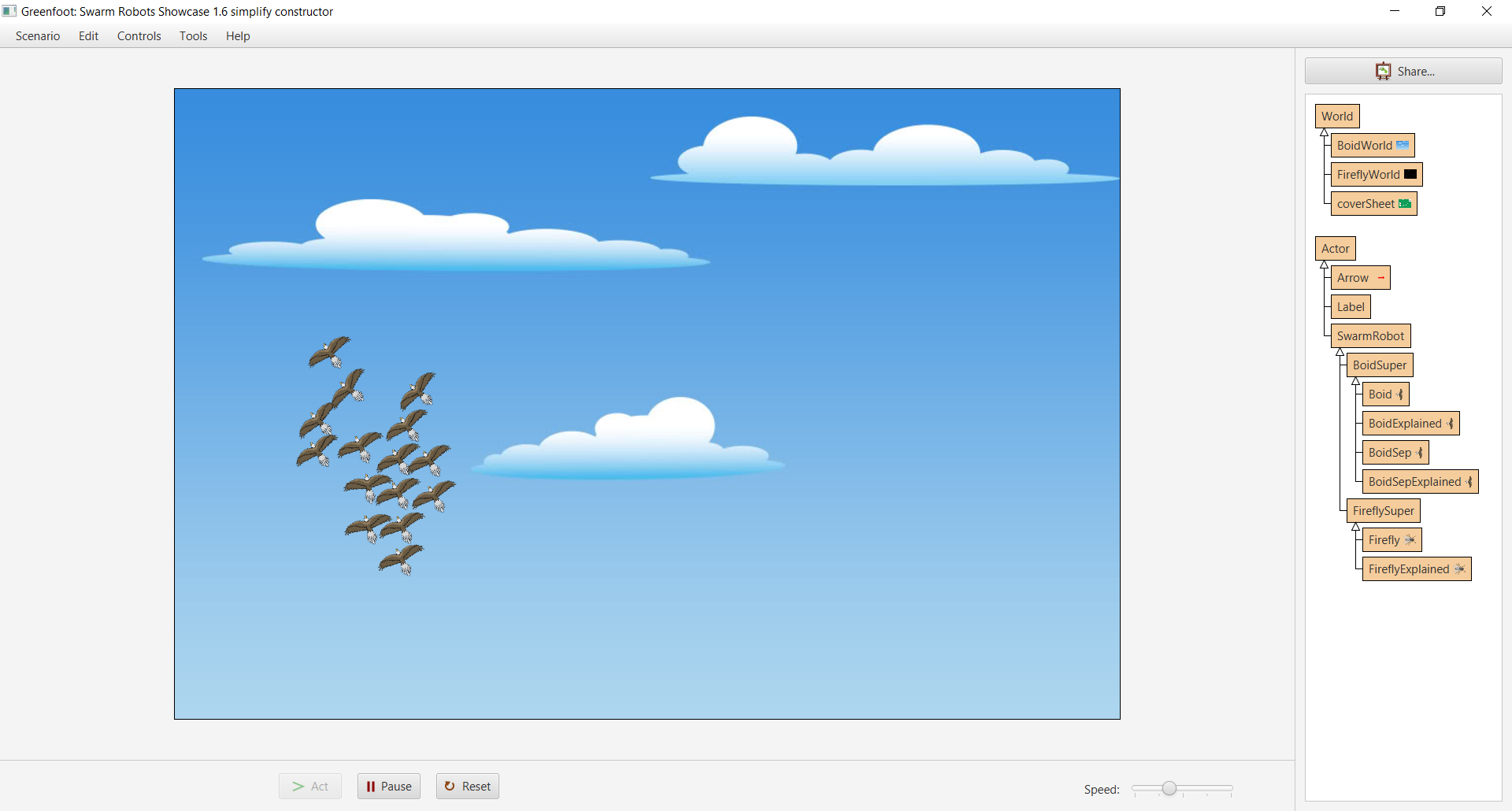
## Task 5

In order to stop the flocking boids from getting too close to one another the separation rule must be used to spread out the swarm. Unlike the other functions the separation function has a parameter separationDistance. This is the distance around the boid that it will begin to turn away from its neighbours. Choose a reasonable value for this parameter, it cannot be greater than the neighbours distance, try around half of that amount to start with. Call the function separation with a parameter of 60 with

separation(60)

and store the result as another item in intArray.

Your boids should now flock properly and mainly avoid collisions.



Your boids are now flocking! Well done!

Compare the look of your flocking simulation with this image of a murmuration of starlings flocking together. The basic pattern should be very similar!



## Task 6

Play around with distances for neighbours and separation neighbours see how the changing the values for each parameter can affect the quality of the flock, and if it will even flock at all.

When editing the separation distance use reasonable values, it must be greater than the distance for neighbours, otherwise the boids will stay well away from each other.

# Conclusion

If you have followed this worksheet and completed all tasks then congratulations! You have learning and understood plenty of swarm robotics principles, as well as actually implementing two example simulations. You have read and understood the pseudocode and by changing some of the parameters in the actor subclasses you have adjusted the settings of the simulations and discovered the values for optimum swarming.