






# The Darwin Game 2.0 Programming Guide

In The Darwin Game creatures compete to control maps and race through mazes. You play by programming your own species of creature in Java, which then acts autonomously during competition. Creatures can move, sense their surroundings, and attack. A successful attack replaces its target with a new instance of the attacker, allowing creatures to reproduce.



## 1 The World of Darwin

Darwin creatures exist in a rectangular world specified by a rectangular map. Different maps are available. A map square may be empty, occupied by a creature, or by an obstruction. Here are some of the common elements in Darwin maps:

<i>Object</i>	<i>Type</i>	<i>Image</i>	<i>Description</i>
<b>Wall</b>	Type.WALL		Impassable square. Attempting to move onto this square halts but does not harm a Creature.
<b>Apple</b>	Type.CREATURE		Motionless, passive Creature just waiting for you to attack it.
<b>Thorn</b>	Type.THORN		Impassable square. Attempting to move onto this square converts a Creature to an Apple as if it were attacked.
<b>Flytrap</b>	Type.CREATURE		Dangerous creature rooted in place. Continuously spins to the left and blindly attacks.
<b>Treasure</b>	Type.CREATURE		Attack this to complete a Maze map.

## **2 Development**

If you are playing the Darwin Game in a course, your instructor has probably given you a Java development environment. If you are not in a course, or just want to work on your Creature at home, you can follow the instructions in this section.

You can play the Darwin Game on any operating system, using the free Java Development Kit from Sun and any text editor, like Notepad, Emacs, Xcode, or Visual Studio. Download the JDK from:

<http://java.sun.com/javase/downloads/index.jsp>

You can use any version of Java 5.0 or later. At the time of this writing, Java 6 Update 5 SE is the latest version. If you are on OS X, Linux, or FreeBSD then Java is probably already installed on your computer.

To compile a creature, type

```
javac mycreature.java
```

at the command prompt after installing Java. You can also use the command

```
javac *.java
```

to simply compile all of the files.

There are several GUI integrated development environments for Java that are either available free or have a free trial period. These include Eclipse, IdeaJ, BlueJ, and NetBeans.

## Winning Conditions

A creature species wins a Darwin map under either of the following conditions:

- Only its species, Flytraps, and Apples remain live on the map
- The time limit is reached, it is the most populous species (ignoring Flytraps and Apples), and there are no Treasures remaining on the map.

All species lose if the time limit is exceeded and there are Treasures present or there is no majority population.

By convention, Maze maps with names beginning in “mz\_” contain only one kind of creature and Treasure (typically, one creature and one Treasure). The goal on these maps is to find and attack the treasure as fast as possible. Natural Selection maps with names beginning in “ns\_” contain multiple kinds of creatures and no Treasure. The goal on these maps is to convert all members of other species, or at least hold a majority when time runs out.

It is possible to create maps with other goals by simply combining Treasures with multiple creature spawn points. For example a head-to-head map race could contain two creatures and two treasures in disconnected mazes.

## 3 Creature Actions

In addition to regular Java commands for programming logic, creatures can perform actions within the world. Each action has a real-world time cost measured in time steps. All actions except for attack are effective at the end of the specified number of time steps. For example, when moving, the creature sits still for 3 time steps and then moves instantaneously. Attacks happen immediately, but then the creature is delayed for 4 time steps.

<i>Action</i>	<i>Cost</i>	<i>Description</i>
<b>Move Backward</b>	4	Move backward one square. If the square is blocked, the move fails but still costs time.
<b>Move Forward</b>	2	Move forward one square in the current facing direction. If that square is blocked, the move fails but still costs time.
<b>Delay</b>	1	Wait one time step without doing anything.
<b>Attack</b>	4	Attack the creature immediately in front of this one. If there is no creature present, the attack fails but still takes time. If the attack succeeds the target is replaced with new instance of this creature facing in the opposite direction.
<b>Look</b>	1	Return a description of the contents of the first non-empty square observed in the creature's facing direction.
<b>Turn Left</b>	3	Rotate 90-degrees counter-clockwise.
<b>Turn Right</b>	3	Rotate 90-degrees clockwise

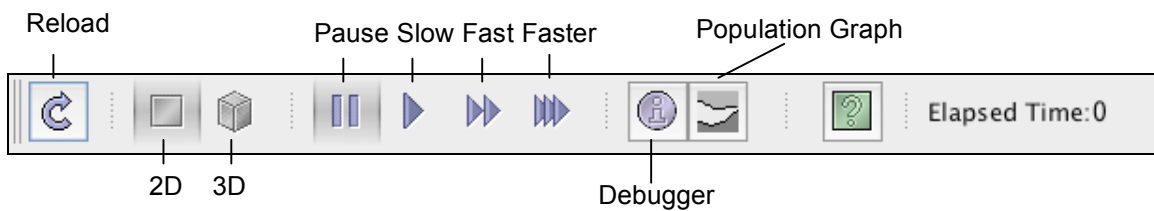
## 4 The Darwin GUI

The Darwin class runs the Simulator within a GUI. Its command line arguments are the name of the map and the Creature classes with which to populate it. An initial optional argument of -2D or -3D forces the initial view. For example, the command:

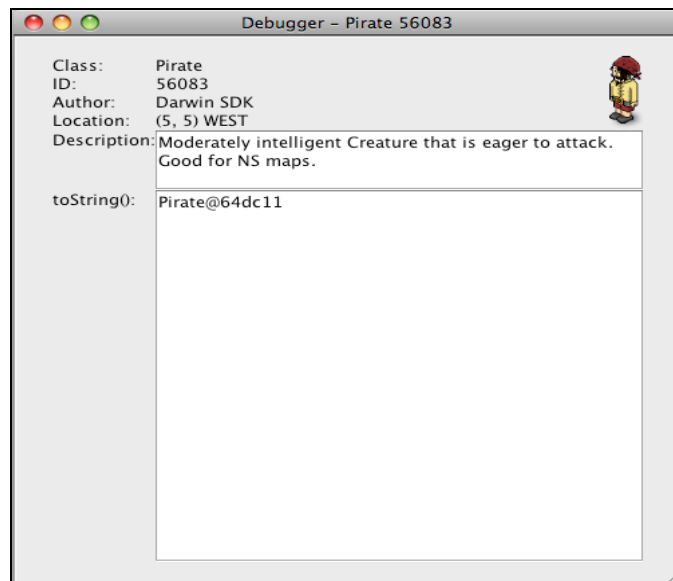
```
java Darwin -3D ns_faceoff.txt Rover Pirate
```

launches the simulator on the Faceoff! map with Rovers competing against Pirates.

The GUI always begins paused. Press one of the three play buttons to begin simulation. The speed of simulation can be changed (or paused again) during play. The view can be switched from 2D (good for debugging) to 3D (good for watching matches) using the gray square and cube icons.



In 2D view mode, click on any Creature to view it in the Darwin Debugger. This shows information about the Creature that updates in realtime, including the current value of its toString method. Override toString on your Creature and use the debugger to inspect the internal state as it moves through the map.



The Reload button not only restarts the simulation, it also reloads your Creature files from the .class files. This means that if you change your Creature and recompile it, you do not need to restart the simulator! Just press reload, as if in a web browser, and you'll get the newest version.

## 5 Creating Maps

Maps are ASCII files. The first line contains the width and height of the map and the map title, separated by spaces and terminated by a newline. The remaining lines form a picture of the map. The elements available are:

- ' ' Empty square
- 'X' Wall
- '#' Alternative color wall
- '+' Thorns
- 'f' Flytrap (which is a Creature)
- 'a' Apple (which is a Creature)
- '\*' Treasure (which is a Creature)
- 'θ'...'9' Spawn locations of Creature subclasses

At load time, the outer border of the map is forced to be all Walls regardless of what was specified in the map file.

Maze maps are named `mz_mapname.txt`. They contain a single Treasure (the goal) and a single 0 that is the start position. (It is possible to make mazes with multiple treasures; for these all Treasures must be attacked to win).

Natural Selection (“deathmatch”) maps are named `ns_mapname.txt` and may have any combination of elements.

The text file for the Faceoff! map is shown below.

```
29 29 Faceoff!
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XX X X X X X X X X X X X X
X                               X
XX XXXXX                       XX
X X 1 X                         X 1 X X
XX X 1                           XXX XX
X 1 X                             X
XX X 1X 1 1 XX
X XXXXX                           X
XX 1 XXXX                          XX
X X X X X X X X X X X X X X
XX          a a a                  XX
X          a a+a a                  X
XX          a a a +                  XX
XX XX XXXXX X+X X+XXXX+XXXXXX
XX
X X X X X X X X X X X X X X
XX          XXXX θ                  XX
X X X X X X X X X X X X X X
XX θ θ Xθ X Xθ X XX
X X X X X X X X X X X X X X
XX XXX θ X X θ X X
XX          XXXX XX
X X X X X X X X X X X X X X
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

## 6 API

Creatures all subclass `Creature`, which provides a set of protected and public methods that enable the `Creature` to interact with the world. Creatures must either provide a public constructor of no arguments or have no constructor. The constructor cannot make the creature take an action.

A `Creature`'s `run` method executes when it is inserted into the world. When the `run` method ends the creature can take no further actions (it remains in the world, however). Therefore most `Creatures` have an intentionally infinite loop in their `run` method to allow them to continue taking actions.

If a `Creature` is successfully and converted to another species, then it is removed from the world but continues executing. The `isAlive()` method for a converted creature returns `false`. If a creature that has been converted attempts to take an action, then a `ConvertedError` is thrown. Most `Creatures` catch this error and then allow their `run` method to terminate.

See <http://cs.williams.edu/~morgan/cs136/darwin2.0/doc> for the full `Creature` API.

Below is a sample of the code for a very simple `Creature` called a `Rover`. It moves until obstructed and then attacks the obstruction and turns to the left. It is surprisingly effective, but is unable to deal with `Thorns` because it never looks before moving. The gray code is boilerplate common to every `Creature`. The bold black code in the center is the logic unique to the `Rover`.

```
public class Rover extends Creature {
    public void run() {
        try {
            while (isAlive()) {
                if (! moveForward()) {
                    attack();
                    turnLeft();
                }
            }
        } catch (ConvertedError e) {}
    }
}
```

`Creature` positions are specified using `Java.awt.Point`, which you will need to import at the top of your class to perform any useful operations on positions. Note the helper methods on `Creature` and `Direction` that operate on `Points` and `Observations`.

The API uses Java enum types to specify `Directions` and creature `Types`. Enum types can generally be treated as constants, however they do provide useful utility methods as well. The following (nonsense) code shows examples of how to use the `Direction` enum.

```
Direction d = Direction.NORTH;

if (d == Direction.SOUTH) {
    ...
}

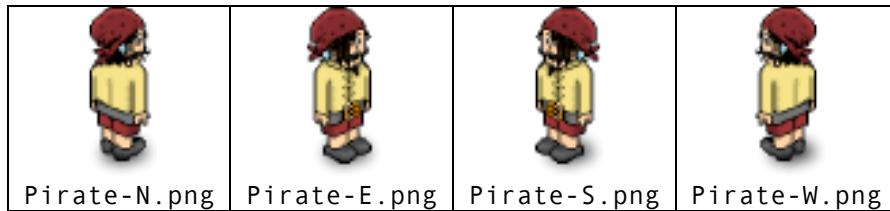
d = d.left();

Point p = new Point(3, 4);
p = Direction.forward(p);

switch (d) {
    case NORTH:
        ...
        break;
    case EAST:
        ...
}
}
```

## 7 Images

You can customize the way the Simulator renders your Creature in the 3D view by providing four images, called sprites. Each image must be in PNG format and be no larger than 40 x 60 pixels. The images must be named *Creature-D.png*, where *D* is one of N, S, E, W and *Creature* is the name of your creature's class. Below are four images for the Pirate Creature.



Images are drawn from a 45-degree isometric perspective. NS and EW lines should be diagonals with a Y:X slope of 1:2. When drawing these it often helps to look at Wall.png to get the perspective right. Since this is the perspective used for many 2D games like Age of Empires, SimCity, Diablo, and Habbo Hotel you can often use sprite images from those games (you can't publicly distribute such sprites, though, because they are copyrighted by the respective developers). A huge list of sprites ripped from 2D games can be found at <http://sdb.drshnaps.com/index.htm>.

Sprites should have transparent backgrounds. The center of the ground square is in the horizontal center of the sprite and about 8 pixels from the bottom of the sprite. Drawing a subtle drop shadow under a sprite helps makes it appear to actually be standing on the ground.



## 8 Advanced Topics

Each creature runs on its own thread. This means that Creatures trade computation time for the time that could be spent taking actions in the world. Actions take at least one millisecond to complete, so small scale efficiency will not affect most Creatures.

However, if your logic is very slow, you might find that your Creature is thinking while others are moving.

To communicate between multiple instances of your Creatures, create static fields to hold values and use the Java synchronized keyword to control access to them. If a static synchronized method has been invoked on a class, all other static synchronized method calls on that class block (wait) for it to complete. This ensures that two instances are not trying to read and write to a variable at the same time and is necessary for consistency. For example, the following code allows a Creature to track how many other members of its species are still alive:

```
class Counter extends Creature {
    static int numAlive = 0;
    static synchronized void changeNumAlive(int delta) { numAlive + delta; }
    static synchronized int getNumAlive() { return numAlive; }
    public void run() {
        changeNumAlive(1);
        try {
            // Body code here
        } catch (ConvertedError e) {
            // No longer alive
        } finally {
            changeNumAlive(-1);
        }
    }
}
```

Creatures may continue to execute after they have been converted so long as they do not take actions. This simplifies the process of performing any kind of centralized control because you can always use the first creature created to issue orders to the others.

Centralized control is not necessarily worthwhile, however—many creatures are very effective by allowing group behavior to emerge from simple individual actions.

Creatures are allowed to open network connections, so you can control them interactively from another machine if you like. However, at full simulation speed it is almost impossible for a human to give any useful input because the Creatures move so fast.

You can run the Simulator without the GUI if working at a terminal or executing offline simulations. The Simulator.toString method prints a text version of the 2D map to aid in debugging.