





Darwin 2.0 Programming Guide

Darwin 2.0 is a game in which 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>
Wall	Type.WALL		Impassable square. Attempting to move onto this square halts but does not harm a Creature.
Apple	Type.CREATURE		Motionless, passive Creature just waiting for you to attack it.
Thorn	Type.THORN		Impassable square. Attempting to move onto this square converts a Creature to an Apple as if it were attacked.
Flytrap	Type.CREATURE		Dangerous creature rooted in place. Continuously spins to the left and blindly attacks.

2 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. The action is 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.

<i>Action</i>	<i>Cost</i>	<i>Description</i>
Move	3	Move forward one square in the current facing direction. If that square is blocked, the move fails but still costs time.
Delay	1	Wait one time step without doing anything.
Attack	2	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.
Look	1	Return a description of the contents of the first non-empty square observed in the creature's facing direction.
Turn Left	3	Rotate 90-degrees counter-clockwise.
Turn Right	3	Rotate 90-degrees clockwise

3 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. For example, the command:

```
java Darwin 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 tournaments) using the gray square and cube icons.



4 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
- '+' Thorns
- 'f' Flytrap (which is a Creature)
- 'a' Apple (which is a Creature)
- '0'...'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 Apple (the goal) and a single 0 that is the start position.

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
XX XXXXX                                     XX
X X 1 X                                     X 1 X X
XX X 1                                     XXX XX
X   1 X                                     X
XX X 11X      1      1      XX
X XXXXX                                     X
XX   1      XXXX      XX
X                                     X
XX                                     XX
XXXXXXXX+XXXXX+X X+X XXXXX XX XX
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                                     XX
X                                     X
XX          XXXX      0      XX
X                                     XXXXX X
XX          0      0      X00 X XX
X                                     X 0 X
XX          XXX      0 X XX
X   X 0 X      X 0 X X
XX          XXXXX XX
X                                     X
XX X X X X X X X X X X X X XX
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

5 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 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 (! move()) {
                    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.

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) {
    ...
}

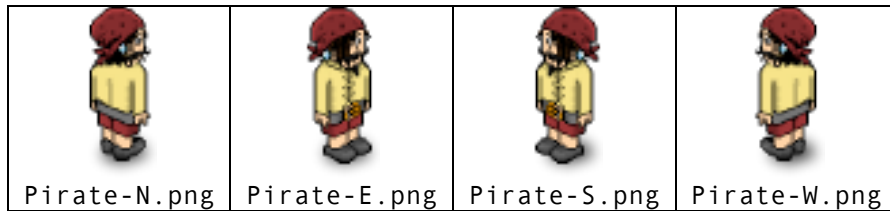
d = d.left();

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

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

6 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).

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 make it appear to actually be standing on the ground.

7 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.