Scalatron Tutorial

About Scalatron

Scalatron is an educational resource for groups of programmers that want to learn more about the Scala programming language or want to hone their Scala programming skills. It is based on Scalatron BotWar, a competitive multi-player programming game in which coders pit bot programs (written in Scala) against each other.

The documentation, tutorial and source code are intended as a community resource and are in the public domain. Feel free to use, copy, and improve them!

Table of Contents

Overview

The behavior of each bot is determined by an associated computer program fragment that implements a control function mapping inputs (what the bot sees) to outputs (how the bot responds). This tutorial explains (very briefly) what Scala is and (in slightly more detail) how to write a bot program in Scala. It is based on the following premises:

- you already know how to program, probably in Java or C++
- the quickest way for a programmer to understand almost anything is to look at sample code and to play around with working programs, modifying and incrementally improving them.

So this tutorial simply presents and analyzes the code to increasingly sophisticated bots, which you can immediately try out and run in the game server. With each version, additional Scala syntax is explained and the most important language constructs introduced. The idea is to let you get from zero to a running bot that you can play with very quickly, and to let you zip through the tutorial at your own pace, picking up useful tools as you go.

Have fun!

Scala Resources

Learning Scala

- Twitter Scala school: http://twitter.github.com/scala_school/ - this is awesome. Consider starting here.
- Book by the inventor of Scala: "Programming in Scala", 2nd Edition, by Odersky, Spoon, Venners. 879 pages. Nice for offline reading. Available in PDF format online: http://www.cs.uwaterloo.ca/~brecht/courses/702/Possible-Readings/scala/ProgrammingInScala.pdf

Other Resources

- The main Scala web site: http://www.scala-lang.org
- Scala style guide: http://www.codecommit.com/scala-style-guide.pdf

About Scala

What is Scala?

Scala is a programming language invented by Martin Odersky in 2000. It is statically typed (like Java) and combines functional (FP) and object oriented (OO) features. Just like Java, Scala compiles into Java byte code and can run on any Java Virtual Machine (JVM). Scala interoperates well with Java and can serve as a by-file or by-module replacement for Java. In many, many respects it fixes the things that are awkward or broken in Java.

Comparison to Java

The syntax of Scala overall is quite similar to Java, but in general it omits elements that are not strictly necessary and place a burden on the programmer (such as semicolons or type declarations that are inferable for the compiler) and generally required less boilerplate code.

The result is that Scala code is often only a third or a quarter of the size of equivalent Java code.

Scala Performance

The performance of Scala depends on how you code. Code written in a functional style can be slower to execute in certain circumstances; code written in an imperative, procedural will be as fast as in Java.

However, functional code is

- much more concise,
- much faster to write,
- much easier to compose
- much easier to parallelize

So generally, writing functional code is preferable for the vast majority of code. Nevertheless, Scala lets you fall back to imperative code for performance critical sections.

How to get your bot into the game

Throughout the tutorial, you will create increasingly complex bot versions in Scala. To see your bot appear in the game, you will need to compile and publish it. There are two approaches to doing this, which in the Scalatron documentation are referred to as the "serious" and "casual" paths, respectively.

The "Casual" Path

The "casual" path is intended for less experienced programmers or for shorter bot coding sessions (2-3 hours). On this path, players write, build, debug and publish their bots in a web browser. The browser-based development environment is provided by an embedded web server which is hosted inside the game server and requires no setup by the user.

If this is the path you want to follow, here is how to do it:

1. open a web browser and point it to the game server's address, which you can get from the workshop organizer. It will generally be something like http://scalatron:8080
2. log into the account associated with your name. If there is no account for you displayed on the log-in screen, ask the workshop organizer to create one for you.
3. enter your code in the editor that appears. You can also copy and paste code from the tutorial and example bot sources.
4. click the Build button. This will upload your code to the game server, compile it there, and display a list of errors (if there were any). Do this until your code compiles.
5. click the Build and Run in Sandbox button. This will upload and compile your code and then start a private, "sandboxed" game for your bot on the server. You can single-step through the simulation and observe the view and state of your bot. Tune your bot until you are happy with it.
6. click the Build and Publish into Tournament button. This will upload and compile your code and then publish it into the tournament loop, where it will be picked up automatically when the next game rounds starts.

The "Serious" Path

The "serious" path is intended for experienced programmers planning for a longer bot coding session (5-7 hours). On this path, bots are built locally by each player, using an IDE or command line tools. The bots are then published into the tournament by copying them into the plug-in directory on the central computer from which the game server loads them at the start of each round.

If this is the path you want to follow, here is how to do it:

Compiling

To compile your bot, you need to feed your Scala source file through the Scala compiler. How you do this depends on the build environment you are using. If you are using the IntelliJ IDEA setup described in the document Player Setup, all you will need to do is select Build > Make Project from the main menu. This should perform the following steps:

- incrementally compile your source file into `.class` files using the Fast Scala Compiler (FSC)
- package the resulting .class files into a Java Archive `.jar` artifact file
- copy the `.jar` file into your bot directory on the game server

Once your plug-in `.jar` file was copied to the game server, it should be picked up as soon as the next game round starts. You can see how far the current round has progressed and estimate how long you'll have to wait in the bottom left corner of the screen, where it will say something like:

```
round 42: 240 steps (of 1000), 4.0s, 20ms/step
```

Publishing

The publishing process basically consist of copying the `.jar` plug-in file you built into your plug-in directory on the server. This directory will be a sub-directory with your name below a network-shared directory available to all users, which the workshop organizer should have made available to everyone.

Example:

- the organizer published a network-shared directory that is visible on your computer as `/Volumes/Scalatron/bots/`
- within this directory, if your name is Tina, you will at some point create a sub-directory called `/Tina`
- into this directory you will publish (i.e., copy) your .jar file, which must be called `ScalatronBot.jar`
- the complete path where your plug-in resides is therefore `/Volumes/Scalatron/bots/Tina/ScalatronBot.jar`

For more details, see the Scalatron Server Setup guide and the Scalatron Protocol documentation.

Updating a bot plug-in

The details of how you publish a newly built bot plug-in depend on how the system running the game server is configured.

In some configurations, new plug-in versions can simply overwrite old ones, even as part of the build process. This is obviously the most convenient setup, since you can configure your IDE to build the .jar artifact directly into the plug-in directory.

In other configurations, the existing `.jar` file may be locked, in which case you must first move that file to another location or delete it.

Bot #1: Hello World

Objective

Create a minimal bot that compiles and appears in the game.

Source Code

Create a source code file called `Bot.scala` and type the following code into it:

```
class ControlFunctionFactory {
    def create = new Bot().respond _
}

class ControlFunction {
    def respond(input: String) = "Status(text=Hello World)"
}
```

Tip: the source code for all bots contained in this tutorial is also available in the appropriate sub-directories of the `/samples` directory of the Scalatron installation. You can also access the samples directly through the browser user interface exposed by the server.

Caution: when you copy source code from a PDF file and paste it into your IDE, it is possible that some characters will be modified and that newlines will be missing, leading to compiler errors. The best way to avoid this is to either type in the code (sorry) or to copy the code from from the samples on disk.

What Is Going On?

The code defines two classes, `ControlFunctionFactory` and `ControlFunction`. In each class, it defines one method. The method `ControlFunctionFactory.create()` constructs a new instance of the class `ControlFunction` and returns a reference to its `respond()` method. This is the control function that the game server will invoke to interact with the bot.

The method `ControlFunction.respond()` receives an String parameter (which it ignores) and returns a constant string that contains a command for the game server, in this case the command `Status()` which asks the server to set the status text of the bot to "Hello World".

What does `class` do?

The `class` keyword begins the definition of a new class.

Examples in our bot:

- `class ControlFunctionFactory { ... }`
- `class ControlFunction { ... }`

What does `def` do?

The `def` keyword defines a function or method. Here are the two examples from our bot:

Example 1:

```
def create = new ControlFunction().respond _
```

- defines a method `create()` in the class `ControlFunctionFactory`

- that takes no parameters
- that returns a function of type `String => String`
- the return type is inferred by the compiler from the method body
- the body uses `new` to create an instance of class `ControlFunction`
- it then returns a reference to the `respond()` method of class `ControlFunction` whose execution context is the instance we just created

Example 2:

```
def respond(input: String) = "Status(text=Hello World)"
```

- defines a method `respond()` in the class `ControlFunction`
- that takes one parameter of type `String` called `input`
- that returns a `String`
- the return type is inferred by the compiler from the method body
- the body consists of the `String` constant `"Status(text=Hello World)"`

Other examples:

```
def foo: Int = 1                    // function without side-effects
def foo = 1                         // Scala infers return type

def bar(s: String): String = "Hello"
def bar(s: String) = "Hello"        // Scala infers return type

def bar(): Unit = { println(..)     // function with side-effects
def bar() { ... }                   // Scala infers return type
```

How are types specified?

In Scala, names precede types:

```
def foo(input: String) : String = ""
val bar: Int = 1
val l: List[Int] = List(1, 2, 3)
```

Types can be omitted if they can be inferred by the compiler:

```
def foo(input: String) = ""
val bar = 1
val l = List(1, 2, 3)
```

What types are available in Scala?

Scala has the well-known types of Java, including

- `String`
- `Int`
- `Double`
- `Char`
- `Array[T]`

Scala also has functions as values and has the appropriate types. Example: a function that takes a `String` and returns a `String`:

```
val foo: String => String = input => input + "."
```

If we want to omit the type on the left-hand side, we need to tell the compiler about it on the right-hand side:

```
val foo = (input: String) => input + "."
```

Why is there no `return` statement?

In Scala, and in functional programming in general, you should think of methods and most code elements as expressions that yield the result of the expression as their value. Instead of writing, for example, code like:

```
int n = 0
if(x.isEmpty) n=1 else n=2
```

we can write

```
val n = if(x.isEmpty) 1 else 2
```

Likewise, we can view entire methods as expressions that yield a value. Instead of writing

```
def respond(input: String) = { return "" }
```

we can write

```
def respond(input: String) = ""
```

Scala does have a `return` statement, but using it is frowned upon as poor functional programming practice and occurrences of `return` are considered by many as a "code smell". If you do want to use `return` (for example to break out of a loop), note that you must explicitly specify the return type of the method from which you want to return.

What does the underscore after `respond` mean?

From the `create()` method, we want to return a function value, namely the method `respond()` of our newly created `ControlFunction`. The underscore, by appearing instead of the parameter list of `respond()`, tells the compiler that we do not wish to invoke the function but to turn it into a value. If you are not familiar with functional programming, this may be a bit confusing but don't be put off by it - you do not yet need to understand the details of this right now and it'll appear quite trivial later on.

Bot #2: Counting Cycles

Objective

Make the bot display on the screen the number of times its control function was invoked.

Source Code

```
// ControlFunctionFactory definition is done as it stands
// from now on, we'll omit it

class ControlFunction {
    var n = 0
    def respond(input: String) = {
        val output = "Status(text=" + n + ")"   // temp value
        n += 1                                  // increments after use
        output                                  // yield
    }
}
```

What Is Going On?

We're from now on omitting the definition of the ControlFunctionFactory class, since it will never change. Just leave it at the top of your source file for now.

In the code, we add a mutable field `n` to the definition of our `ControlFunction` class. From the assignment of the zero value, the compiler infers that this field will have type `Int`.

In our `respond()` method, instead of returning a constant string as the status message, we now construct a new string every time by concatenating multiple sub-strings and storing the result in a temporary, immutable value called `output`. One of these sub-strings is a String rendering of our counter field. The compiler conveniently converts the Int into a String for us.

We then increment the counter by one and return the string we constructed earlier. We indicate that we want to return the constructed string by simply making a reference to the `output` value the last expression in the method.

This "yielding" behavior of functional languages is also the reason we create the temporary value in the first place: if `n += 1` was the last expression in our method, that would also be the return value of the method.

What does `var` do?

The `var` keyword defines a new mutable value, which can be a local variable in a method or a mutable field of a class. Note that mutability is generally frowned upon by functional programmers since it makes it more difficult to reason about the state and behavior of programs.

What does `val` do?

The `val` keyword defines a new immutable value, which can be a local constant in a method or an immutable field of a class. Because it is immutable, its value can never change.

Where do I need curly braces?

Generally speaking you need curly braces to define the body of a class and to group multiple statements into a block. That's why this method definition does not need curly braces:

```
def respond(input: String) = "Status(text=" + n + ")"
```

but this one does:

```
def respond(input: String) = {
   val output = "Status(text=" + n + ")"
   n += 1
   output
}
```

Where do I need a semicolon?

Unlike in Java, you do not need a semicolon at the end of every statement. For the moment, the only reason you might want to use a semicolon is to separate multiple expressions on the same line, like this:

```
def respond(input: String) = { n += 1; "Status(text=Hello)" }
```

Why do we store the output in a temporary constant?

Scala yields the value of the last expression in a method as the result value of that method. To see what this means, consider the following variants of the code above:

- **Step 1**: mutable field, but no incrementation

  ```
  class ControlFunction {
     var n = 0 // declare mutable field
     def respond(input: String) = "Status(text=" + n + ")"
  }
  ```

  This code obviously always returns the same string.

- **Step 2**: field incrementation before String concatenation

  ```
  class ControlFunction {
     var n = 0
     def respond(input: String) = { // curly braces for scope
        n += 1 // increment counter
        "Status(text=" + n + ")"
     }
  }
  ```

  This code returns a string containing an incrementing counter, but the counter does not start at zero. Because it is incremented before the string is constructed, the first value returned is one.

- **Step 3**:

  ```
  class ControlFunction {
     var n = 0
     def respond(input: String) = {
        "Status(text=" + n + ")"
        n += 1 // BUG: yields Int (the updated n), not string
     }
  }
  ```

This now constructs the correct text, but instead of returning the text it returns the incremented value. Not only that, but the compiler (correctly) infers the type of the function to be `String => Int`, leading to a compile time error because it does not match the signature expected by the factory.

- **Step 4**:

```
class ControlFunction {
   var n = 0
   def respond(input: String) = {
     val output = "Status(text=" + n + ")" // temp value
     n += 1 // now increments after use
     output // yield
   }
}
```

This creates a temporary constant value of type `String`, then increments the value, and finally references the constant value again to make it the return value of the function. Now everything works as expected.

Bot #3: Checking the Opcode

Objective

We now want to let the bot distinguish between the different commands that the game server may pass as input parameters into the control function, such as "`React`" versus "`Welcome`". To do that, we need to parse the input string provided by the server. Here is a simplified input string (for details, check the `Scalatron Protocol` documentation):

```
React(generation=0,time=0,view=__W_W_W__,energy=100)
```

We will modify the control function to now only respond with a command if the server sends an input containing a React() opcode. To make the bot behavior a little more interesting, we will have it return a Move() command, prompting the bot to run towards the right.

Source Code

The ControlFunctionFactory and the ControlFunction boilerplate code remains unchanged from the previous example; we replace only the `respond()` method:

```
def respond(input: String) = {
    val tokens = input.split('(')   // split at '(', returns Array[String]
    if(tokens(0)=="React") {        // token(0): 0th element of array
        "Move(direction=1:0)"       // response if true
    } else {
        ""                          // response if false
    }
}
```

What Is Going On?

First, we need to extract the opcode from the command string. A simple way to achieve this is to use one of the library methods that Scala offers for values of type String. Namely, we use the `split()` method, which takes a value of type `Char` as a parameter, to break the string into a collection of sub-strings at all occurrences of that character value (here, occurrences of the opening parenthesis that immediately follows an opcode).

`String.split()` returns a result of type `Array[String]`. We extract the first array element, which should now contain the string representing the opcode, and compare it to the expected opcode. Unlike in Java, in Scala the comparison with == does what you expect.

If the opcode is `React`, we return a `Move()` command. Otherwise we return an empty string.

What does `split()` do?

`String.split()` breaks a string into a collection of sub-strings at occurrences of a given separator character. It is defined in the Scala library and has the following signature:

```
def split(separator: Char) : Array[String]
```

So it takes a value of type `Char` as a parameter (the separator) and returns a result of type `Array [String]` (the sub-strings).

Example: `"a(b)".split('(')` returns `Array("a", "b)")`

What are some other methods available on `String`?

A good way to learn about the methods available at any point in your code is to use your IDE's auto-complete method. This is highly recommended, since e.g. the Scala plug-in for IntelliJ IDEA will also display methods available through implicit conversions, something that is extremely convenient but initially somewhat hard to fathom for people new to Scala.

Here are a few more methods available on String:

```
val a = "Hello" + " " + "World"
a.length            // = 11
a.drop(6)           // = "World"
a.dropRight(6)      // = "Hello"
a.split(' ')        // = Array("Hello", "World")
```

How does the `if` statement work?

Remember that in Scala, you should think of your code components as expressions. Here, the if block is an expression whose result value is either the value of the true branch or the false branch:

```
if(condition) { true-block } else { false-block }
```

This is actually a bit similar to the '?:' operator, which is not necessary in Scala.

Examples:

```
val b = if(true) true else false
val a = if(b) false else true
val not : Boolean => Boolean = in => if(in) false else true
```

What does `tokens(0)` do?

In Scala, you retrieve a particular element of an array by specifying its (zero-based) index in simple parentheses. Since tokens is an Array[String] that was returned by String.split(), the expression tokens(0) returns the first element (at index zero) of that array.

Assignment to array elements works in the same manner. Here are some additional examples of working with arrays:

```
val nums = Array(1, 2, 3)
val strings = Array("Hello", "World")
val empty = Array.empty[Int]

val s0 = strings(0)     // read
nums(1) = 9             // write
```

Bot #4: Expanded Input Parser

Objective

Instead of a counter, we want to display the bot's current energy level next to it on the screen. To do that, we first have to extract the energy from the input string and then set it as the bot's status string using the `Status()` command. The server informs the bot about its energy level via the energy parameter of the input string. Here is again an example input string:

```
React(generation=0,view=__W_W_W__,energy=100)
```

Since we'll soon need to extract all of the other parameters as well, the most sensible way to proceed is to parse the entire parameter list.

Source Code

```
def respond(input: String) = {
    val tokens = input.split('(')
    val opcode = tokens(0)
    if(opcode=="React") {
        val rest = tokens(1).dropRight(1)
        val params = rest.split(',')
        val strPairs = params.map(s => s.split('='))
        val kvPairs = strPairs.map(a => (a(0),a(1)))
        val paramMap = kvPairs.toMap

        val energy = paramMap("energy").toInt
        "Status(text=Energy:" + energy + ")"
    } else {
        ""
    }
}
```

What Is Going On?

The first two line uses `split()` to break the input into two parts: the opcode and the remaining line:

```
val tokens = input.split('(')
```

These two parts are returned in an array, from which we extract the first element (index zero) and store it into a local immutable value:

```
val opcode = tokens(0)
```

We then test the just-extracted opcode against the `"React"` constant:

```
if(opcode=="React") {
```

The second element of the tokens array contains the parameters of the command, plus a trailing closing parenthesis (the opening parenthesis was swallowed by `split()`). To obtain a string containing just the comma-separated parameters, we need to get rid of this trailing character. We do this using the `String` method `dropRight()`, a method available on many collections which takes the number of elements to drop as a parameter:

```
val rest = tokens(1).dropRight(1)
```

The rest value now contains a string consisting just of comma-separated key/value pairs, as shown in the comment on the line above. So we use `split()` again to break it apart, this time at occurrences of the comma character:

```
val params = rest.split(',')
```

The result is again an array of `String` values, each of which has the format `"key=value"`. So now we want to break apart each one of these strings using split() at the equals sign. We achieve this by using `map()` to process every element of the array, obtaining a new collection containing the processed elements:

```
val strPairs = params.map(s => s.split('='))
```

So `strPairs` now is an array of arrays of `String`, with overall type `Array[Array[String]]`. For easy lookups later on, we'd really like to have a map from keys to values. Conveniently, there is the `toMap()` method that converts a collection of key/value pairs into a map. Unfortunately, we don't have a collection of pairs but a collection of arrays. So we process `strPairs` again to obtain a collection of key/value pairs:

```
val kvPairs = strPairs.map(a => (a(0),a(1)))
```

which we then convert into a map:

```
val paramMap = kvPairs.toMap
```

The value map now contains a reference to an instance of the Scala `Map` collection, with type `Map[String,String]`. Now it is easy to extract the value associated with the key `"energy"` and to convert it from a string into an integer value:

```
val energy = paramMap("energy").toInt
```

And finally we assemble the command string that updates the bot's status message:

```
"Status(text=Energy:" + energy + ")"
```

A Word on Scala Collections

Scala has a very extensive and very well-designed library that contains a number of useful collection types, including:

- `List`
- `Set`
- `Map`
- `Queue`

In many cases, there are two implementations for these collection types: a mutable and an immutable one. In the mutable variants, operations modify the collection's contents. In the immutable variants operations return a new, modified collection without disturbing the original collection.

Scala defaults to immutable collections and for the bot code we will also exclusively select these immutable collections. Immutability makes it much easier to reason about our code and to parallelize operations, since we never need to worry about synchronizing concurrent accesses. In certain performance-critical code sections it may occasionally be advantageous to internally use a mutable collection with in-place updating, but this is rarely the case and should be done only if profiling indicates a need for it. As a rule of thumb, to get the benefits of functional programming we'll embrace immutability as the default. The Scalatron server and the Scalatron bot examples exclusively use immutable collections.

If you are interested in the ideas behind immutable collections and other data structures and want to know more about their time and space characteristics, check out the book "Purely Functional Data Structures" by Chris Okasaki.

What does `dropRight()` do?

Given a collection of elements, we often want to do the same kinds of things with them: extract elements, drop elements, split the collection, etc. The Scala library provides implementations for many such operations. If an operation is not present in exactly the form you need it, in many cases it is still possible to compose existing operations to achieve your goal.

`dropRight()` is one example of such a collection method. It drops the last N elements of a collection and returns the resulting new collection. In our bot example:

```
val rest = tokens(1).dropRight(1)
```

we apply dropRight() to an instance of String. This works because there is an implicit conversion from String into a collection type that presents its contents as an indexed sequence (IndexedSeq) of characters.

In excruciating detail, our code

```
val rest = tokens(1).dropRight(1)
```

does this:

- defines an immutable value `rest`
- takes the element with index `1` in the array of Strings in `tokens`
- this will be the right-hand side of the split, like "key=value,key=value)"
- invokes its method `dropRight()`
- passing the parameter '`1`' for the number of elements to drop
- which returns a value of type `String` without the closing parenthesis
- we can then split the key/value pairs at commas

Useful collection methods

Here are some other frequently used collection methods:

- **`head`**: returns the first element of a collection
- **`last`**: returns the last element of a collection
- **`tail`**: drops the first element of a collection and returns the resulting new collection
- **`drop(n)`**: drops the first N elements of the collection and returns the resulting new collection

A few examples:

```
List(1,2,3).drop(2)     // returns List(3)
List(1,2,3).head        // returns 1
List(1,2,3).tail        // returns List(2,3)
List(1,2,3).last        // returns 3
```

As mentioned, many collection operations also apply to `String`, treating it as a collection of `Char`:

```
"abc".drop(1)           // returns "bc"
"abc".dropRight(1)      // returns "ab"
```

What does `(a(0),a(1))` do?

This expression constructs a pair (a tuple of arity two) containing the first and second elements of an array value `a`. We'll talk about where the array comes from in the next section, and for now focus only on the tuple creation.

Generally speaking, when writing methods, it is often necessary for a function to return more than one value. Traditionally, for example in Java or C++, this requires the creation of a parameter object, which involves the definition of a new named type (a `class` or `struct`) and quite a bit of boilerplate code.

Scala already offers a built-in type `Tuple` with elements whose types are parameterized. There are implementations for tuples for arity up to 22. Here are some examples:

```
val pair = Tuple2[Int,String](1, "A")
val triple = Tuple3[Int,String,Char](1, "A", 'B')
```

So instead of defining a new class or struct for a parameter object, we can use a generic `Tuple` to return more than one value from a function. Here is an example for a function declared to return a pair of `String` values:

```
def foo : Tuple2[String,String]   // function that returns a tuple
```

and here is an example implementation:

```
def foo = new Tuple2[String,String]("Hello", "World")
```

for which the compiler can infer the types, so we can omit those:

```
def foo = new Tuple2("Hello", "World")
```

and because `Tuple2` offers a static construction method (more on those later), we can omit the `new` keyword:

```
def foo = Tuple2("Hello", "World")
```

But Scala can do even better than that. Because using `Tuple` to return values is so useful and hence used so frequently, Scala has special syntactic sugar to create tuples and describe tuple types: you simple use parentheses around a comma-separated list of values or types. Let's do that for our example:

```
def foo = ("Hello", "World")
```

Here are some other examples for the use of tuples:

```
val pair = (1, "A")                      // = Tuple2[Int,String]
val triple = (1, 2, 3)                   // = Tuple3[Int,Int,String]
```

Scala's syntactic sugar not just helps with creating tuples, it also extends to declaring tuple types. In the original declaration of our example function `foo`, we had:

```
def foo : Tuple2[String,String]     // function that returns a tuple
```

Using the tuple syntax we can now write the exact same declaration like this:

```
def foo : (String,String)           // function that returns a tuple
```

We can use this wherever a type would appear in Scala:

```
def foo(pair: (Int, String)) // function accepting Tuple2[Int,String]
def bar: (Int, String)       // function returning Tuple2[Int,String]
val f = () => (Int, String)  // fct. val. returning Tuple2[Int,String]
```

How do you access the contents of a tuple? The tuple implementations in Scala all contain fields with names like `_1`, `_2`, `_3`, etc. Here is an example:

```
val a = ("A","B")._1       // extracts the first field of the tuple
val b = ("A","B")._2       // extracts the second field of the tuple
```

Some more examples for the use of tuples:

```
val tuple = ("A",2)            // = Tuple2[String,Int] = (String,Int)
val array = Array("A","B")     // = Array[String]
val tuple = (array(0),array(1)) // = Tuple2[String,String]
```

What does `map()` do?

The `map()` method appears in our example twice:

```
val strPairs = params.map(s => s.split('='))
val kvPairs = strPairs.map(a => (a(0),a(1)))
```

It is a method available on collections (like `List`, `Set`, `Map`) that, given a transformation function, transforms every element of the collection into a new element and returns a new collection containing the transformed elements.

It also works for `String` because `String` can act as a collection of `Char`. And, even though `Array` in Scala is really just a Java `Array` which does not extend any of the neat Scala collection traits, it also works for `Array` because there is an implicit conversion that makes it available.

Let's pick apart the two lines that appear in our example in excruciating detail.

Example 1:

```
val strPairs = params.map(s => s.split('='))
```

We start by using `val` to declare an immutable value that will be available in the scope of our method. We tell the compiler that we will want to access that value by the name `strPairs`. We then reference the collection we want to work on, here params, and use the dot-notation familiar from Java or C++ to invoke the `map()` method on it.

The `map()` method takes one parameter, the transformation function. This has to be a function that takes one parameter (an element of the collection) as its input and that returns a new value. The type of the parameter must naturally be the type of the element. The return value can and often will be of a different type. In the example, the transformation function operates an a collection of `String` elements and therefore will take an input parameter of type `String`. We invoke `split()` on each of these strings and therefore convert each element into an `Array[String]`. Here, the return type of `split()` is also the return type of our transformation function. It therefore has the following type:

```
String => Array[String]
```

i.e. it takes a `String` parameter and returns an `Array[String]` value.

The Scala collection library is very clever in how it implements `map()`. This has the benefit that you will, whenever possible, get back a collection of the same type as the one you operated on. Invoking `map()` on a `List` will result in a new `List`, invoking it on a `Set` will result in a new `Set` and invoking it on a `Map` will result in a new `Map`. This is not trivial. The downside is that the type signature of `map()` looks scary as hell to people new to Scala. So we won't show it here. But it does what you expect it to do and you can for now should use it without worrying about the details. You will have absolutely not problem working out the details once you're more familiar Scala.

So all that is left for us to do is to somehow tell the compiler that we want the argument to map() to be an invocation of split() on every element of the collection. That is what the code in the parentheses after map does:

```
s => s.split('=')
```

Let's first re-write this code using a slightly more verbose but semantically identical syntax, which we will then explain and show how to get to the simplified version. Here is the verbose version:

```
(s: String) => { s.split('=') }
```

This defines a function value by specifying, from left to right:

- the function's input parameter list: `(s: String)`
- a symbol that tells the compiler "this is a function": =>
- the body of the function: `{ s.split('=') }`

The input parameter list here consist of a single `String` parameter, to which we assign the name `s`. Since the compiler knows the type of the collection on which we invoke `map()` and it knows the signature of `map()`, it can infer the type of the input of our transform function. It already knows that `s` will be a `String`. It therefore allows us to drop the explicit mention of the type from our function definition. This gives us:

```
(s) => { s.split('=') }
```

Since there is only a single parameter, we can also omit the parentheses. Like this:

```
s => { s.split('=') }
```

Now on the right-hand side of the => we have the function body. It invokes `split()` with the argument `s`, which obviously contains whatever was passed as the input parameter to the function; here, consecutively each element of the collection. Since the function body consists only of a single expression, we can omit the curly braces. This gives us:

```
s => s.split('=')
```

Which is exactly what we have in the example.

Scala allows us to express this even more concisely, however. We did not use this in the example above to keep things simple to start with, but you can now try this out right away. Essentially, it is very common for functions like the one we're passing to `map()` to reference their parameters exactly once, for example to invoke a method on them. In instances where this is the case, there is really no need to assign a name to the parameter - better to keep it short. Scala allows us to make such an "anonymous" reference to a parameter by using an underscore. So we can actually rewrite the example code like this:

```
val strPairs = params.map(_.split('='))
```

When the `map()` method is executed at run time, it will iterate over the collection elements of params and invoke `split('=')` on each one. This generates a new collection containing elements of type `Array[String]`, which is then available through the value strPairs. So when we start, params might contain the following value:

```
Array("a=b", "c=d", "e=f")
```

Afterwards, strPairs would contain the following value:

```
Array(Array("a", "b"), Array("c", "d"), Array("e", "f"))
```

Example 2:

```
val kvPairs = strPairs.map(a => (a(0),a(1)))
```

The intent is to convert the array of arrays of strings into an array of pairs of strings, so that we can then turn it into a map using `toMap()`. This is achieved by invoking the `map()` method on `strPairs`, which has type `Array[Array[String]]`, and using a transform function that turns each element of the collection (which will be an `Array[String]` that was returned by `split()`, see above) into a pair of strings. Here is the transform function in isolation:

```
a => (a(0),a(1))
```

This is the concise equivalent of a more verbose version, like what we had in the previous example:

```
(a: Array[String]) => { (a(0),a(1)) }
```

Which would be even more verbose if we did not have the syntactic sugar for tuple creation and type inference for the tuple's parameter types:

```
(a: Array[String]) => { new Tuple2[String,String](a(0), a(1)) }
```

Just think for a moment about the amount of boilerplate code you'd have to write to do this in whatever language you are using now, the time it takes to type all that stuff in and the opportunity for introducing errors that generates.

Wrapping up

The example uses a lot of named values to clarify the meaning of each computational step. Like this:

```
val rest = tokens(1).dropRight(1)
val params = rest.split(',')
val strPairs = params.map(s => s.split('='))
val kvPairs = strPairs.map(a => (a(0),a(1)))
val paramMap = kvPairs.toMap
```

We could of course draw all this together into a single invocation chain. The result would look like this:

```
val paramMap =
    tokens(1)
    .dropRight(1)
    .split(',')
    .map(_.split('='))
    .map(a => (a(0),a(1)))
    .toMap
```

and the entire control function would look like this:

```
def respond(input: String) = {
    val tokens = input.split('(')
    val opcode = tokens(0)
    if(opcode=="React") {
        val paramMap =
            tokens(1).dropRight(1).split(',').map(_.split('='))
            .map(a => (a(0), a(1))).toMap
        val energy = paramMap("energy").toInt
        "Status(text=Energy:" + energy + ")"
    } else {
        ""
    }
}
```

More on Scala Maps

Like for most collection types, there are mutable and immutable versions of `Map`. The appendix of this tutorial contains a cheat sheet with useful methods of `Map`.

Bot #5: Creating a Command Parser Function

Objective

Having the somewhat complicated command parsing code in the middle of our control function will be distracting. So we will pull it out into a separate function, `parse()`. While we're at it, we'll also add some validation code.

Source Code

```
class ControlFunction {
    def respond(input: String) = {
        val parseResult = parse(input)
        val opcode = parseResult._1
        val paramMap = parseResult._2
        if(opcode=="React") {
            "Status(text=Energy:" + paramMap("energy") + ")"
        } else {
            ""
        }
    }

    def parse(command: String) = {
        def splitParam(param: String) = {
            val segments = param.split('=')
            if( segments.length != 2 )
                throw new IllegalStateException(
                  "invalid key/value pair: " + param)
            (segments(0),segments(1))
        }

        val segments = command.split('(')
        if( segments.length != 2 )
            throw new IllegalStateException(
              "invalid command: " + command)

        val params = segments(1).dropRight(1).split(‘,’)
        val keyValuePairs = params.map(splitParam).toMap
        (segments(0), keyValuePairs)
    }
}
```

What Is Going On?

The code in the `respond()` method should by now be obvious. The primary change is that instead of parsing the input string locally, we invoke a newly defined function `parse()`.

The parse function is defined within the body of our `ControlFunction` class, which makes it a method with access to the `ControlFunction` instance (via `this`) - something we do not really need here. It could just as well be a static function. In the next example we will make this change, but for now let's look at the method as it stands.

What is the return type of `parse()`?

We might note that in the first line of the method definition

```
def parse(command: String) = {
```

no return type is specified. The = indicates to the compiler that a value will be returned, but we leave it up to the compiler to figure out what the return type is. After looking at the method definition (which we'll also do in just a minute) the compiler will infer it to be:

```
(String, Map[String, String])
```

i.e. a tuple (pair) containing a `String` as well as a `Map` from `String` to `String`. In the first element of the tuple `parse()` will return the opcode of the command. In the second element it will return the parameter map. Its keys are the parameter names and its values are the parameter values.

Here is an example of what the `parse()` method expects as its input:

```
"React(generation=0,energy=100)"
```

and what it will return:

```
("React", Map( "generation" -> "0", "energy" -> "100") )
```

Why is there a `def` in the body of `parse()`?

To structure our parsing code, we use a local helper function `splitParam()` to break apart the individual key/value pairs. Since this helper function is not used or useful anywhere else in our code, we can keep it private to the method that needs it. Scala allows us to nest functions as deeply as we want, so we simply defined `splitParam()` within `parse()`, keeping the symbol local to the function.

Can my Bot just throw an exception?

Yes. The Scalatron game server will catch and process all exceptions thrown within the bots. It also won't send you any invalid commands, but knowing how to validate your inputs and how to throw an exception is useful.

When an exception is thrown, just as in Java, control escalates up the call stack to the nearest exception handler. This is an instance where no value is returned from a function.

Actually, just for reference, here is how you catch an exception in Scala:

```
try {
    ... code that might throw an exception...
} catch {
    case e: Exception => ... code that handles the exception ...
}
```

The code within the `catch` block uses pattern matching, a powerful feature that we will look at later on.

Note that the entire code block above is also an expression. If no exception is thrown, it yields the value of the code within the `try` block. If an exception is thrown, it yields the value of the exception handler.

What does `map(splitParam)` do?

It is another example of syntactic sugar offered by Scala to eliminate boilerplate and make your code more concise. The full line in the example reads:

```
val keyValuePairs = params.map(splitParam).toMap
```

We already looked at how `map()` works: it expects a transformation function as its only parameter. Here, the transformation function is obviously `splitParams`, the local function we defined to take apart the key/value pairs. But the syntax we use here is new: it contains neither a named parameter nor an underscore as an anonymous placeholder. Instead, it exploits the fact that - in the frequently occurring case of a function whose body is simply another function that takes the input parameter of the outer function as its parameter - we can omit the parameter altogether. The verbose variant of the line above would be:

```
val keyValuePairs = params.map(s => splitParam(s)).toMap
```

Bot #6: Extracting a Command Parser

Objective

In the preceding bot version we noted that the `parse()` method could really be a static method. So let's turn it into one.

Source Code

```
class ControlFunction {
    def respond(input: String) = {
        val (opcode, paramMap) = CommandParser(input)
        if(opcode=="React") {
            "Status(text=Energy:" + paramMap("energy") + ")"
        } else {
            ""
        }
    }
}

object CommandParser {
    def apply(command: String) = {
        def splitParam(param: String) = {
            val segments = param.split('=')
            if( segments.length != 2 )
                throw new IllegalStateException(
                  "invalid key/value pair: " + param)
            (segments(0),segments(1))
        }

        val segments = command.split('(')
        if( segments.length != 2 )
            throw new IllegalStateException(
                "invalid command: " + command)

        val params = segments(1).dropRight(1).split(‘,’)
        val keyValuePairs = params.map( splitParam ).toMap
        (segments(0), keyValuePairs)
    }
}
```

What Is Going On?

The code above uses the keyword `object` to define a container for our static parser function (more on `object` below). The container gets the name `CommandParser` and it contains a static method `apply()`, which is simply a renamed copy of our former `parse()` function. And the method body of `respond()` invokes the static parsing function via a reference to the `CommandParser` object. Here you should note the absence of an explicit mention of `apply()` in the invocation and the way the parsed result is broken up into two values in parentheses - both of which we'll explain below.

What does `object` do?

The `object` keyword in Scala defines a container for static values and methods. Unlike Java, where static fields and methods are mixed into the regular methods of a `class`, Scala isolates static methods into such containers.

Here is an example of how you might use `object`:

```
object BotConstants {
    val energyThreshold = 100
    val spawnDelay = 10
}
```

and here is how code elsewhere could access these static values:

```
class FooBar {
    def foo() {
        bar(BotConstants.energyThreshold)
        blop(BotConstants.spawnDelay)
    }
}
```

An important concept in this context is that of a companion object for a `class`. A companion object is a container for the static methods of a class that has the same name as the class and that is defined in the same `.scala` file.

Why is there no explicit call to `apply()`?

In the bot example code, we have the following line:

```
val (opcode, paramMap) = CommandParser(input)
```

We now know that `CommandParser` refers to a container for static functions and values. And evidently the code is intended to parse the input. But the parser function has the name `apply()` and there is no mention of `apply()` here. What is going on?

In Scala, methods called `apply()` have special meaning. The most immediate practical consequence is that for methods and functions whose name is `apply` you can omit the method name and use just parentheses and the parameters. So the verbose variant of the invocation above would be:

```
val (opcode, paramMap) = CommandParser.apply(input)
```

We simply omitted the dot and the `apply` method name, because we do not need it. You'll see various uses of this hand syntax later on, which will in many scenarios make your code look much more readable and elegant (once you've figured out what is going on, of course).

What does `paramMap("energy")` do?

The complete line of code in our bot example is:

```
"Status(text=Energy:" + paramMap("energy") + ")"
```

Overall this is a string concatenation in which the symbol `paramMap` refers to the parameter map returned by the command parser, i.e. it is a reference to an instance of a `Map[String,String]` whose keys are the parameter names and whose values are the parameter values.

So we can already anticipate that this code will retrieve the value associated with the key "energy". But why is this a valid method invocation, if we have an instance but no dot or method?

The solution is the same as for the code that we used to parse the command string: it is an invocation of a method of `Map` called `apply()`, in which Scala allows us to omit the method name. The verbose variant would be this:

```
"Status(text=Energy:" + paramMap.apply("energy") + ")"
```

What does `(opcode, paramMap)` do?

So we know that `parse()` returns a tuple that Scala generates for us on the fly. In the preceding example, we have used the following code to take apart the tuple and obtain the individual element values:

```
val parseResult = parse(input)
val opcode = parseResult._1
val paramMap = parseResult._2
```

Now, the code just reads

```
val (opcode, paramMap) = CommandParser(input)
```

How does this work?

We already noted that the need for functions with multiple return values is common and prompted the Scala designers to offer some syntactic sugar for constructing and typing `Tuple` values. To complete the cycle, Scala also offers syntactic sugar for taking the results apart again at the call site, some of which is specific to `Tuple` and some is not.

The details are beyond what is required here, but the gist of the code above is this: Scala breaks the returned tuple up into its elements and assigns the element values to the local values `opcode` and `paramMap`.

If you want to read up on the concepts related to this extremely powerful idea, look up "extractors" and "unapply" in the "Programming in Scala" book.

Bot #7: Brownian Motion

Objective

Create a bot that will meander around the arena using random motions.

Source Code

We will from now on omit the code for the `CommandParser` object, since we will never again change it. Just leave it in your source file, just like we did with `ControlFunctionFactory`.

```
import util.Random

class ControlFunction {
    val rnd = new Random()
    def respond(input: String): String = {
        val (opcode, paramMap) = CommandParser(input)
        if( opcode == "React" ) {
            val dx = rnd.nextInt(3) - 1
            val dy = rnd.nextInt(3) - 1
            "Move(direction=" + dx + ":" + dy + ")"
        } else {
            ""
        }
    }
}
```

What Is Going On?

In the first line, we add an `import` statement that will pull in the definition of the `scala.util.Random` class, a pseudo-random number generator (actually a Scala wrapper for `java.util.Random`):

```
import util.Random
```

This makes the symbol `Random` available to the code in our source file.

We then add a new field called `rnd` to our `ControlFunction` class which will hold a reference to an instance of such a pseudo-random number generator, which we simply create with `new`:

```
val rnd = new Random()
```

Further down, we use the generator to produce random X and Y values for the movement direction of our bot:

```
val dx = rnd.nextInt(3) - 1
val dy = rnd.nextInt(3) - 1
```

This code uses the member function `nextInt(n: Int)` of `Random`, which returns a value in the range 0 to `(n-1)`.

We then combine the generated coordinate values with the Scalatron command opcode `Move` (see the Scalatron Protocol documentation) to assemble the bot's response:

```
"Move(direction=" + dx + ":" + dy + ")"
```

The result is a bot that, every time it is asked by the game server what it wants to do, requests the be moved once cell in some random direction.

Bot #8: Missile Launcher

Objective

Create a control function that will let our (master) bot launch missiles (mini-bots) at random intervals in random directions and will guide the missile mini-bots along those directions.

Source Code

```
class ControlFunction {
    val rnd = new Random()

    def respond(input: String): String = {
        val (opcode, paramMap) = CommandParser(input)
        if( opcode == "React" ) {
            val generation = paramMap("generation").toInt
            if( generation == 0 ) {
                if( paramMap("energy").toInt >= 100 &&
                    rnd.nextDouble() < 0.05 ) {
                    val dx = rnd.nextInt(3)-1
                    val dy = rnd.nextInt(3)-1
                    val direction = dx + ":" + dy // e.g. "-1:1"
                    "Spawn(direction=" + direction + ",energy=100," +
                        "heading=" + direction + ")"
                } else ""
            } else {
                val heading = paramMap("heading")
                "Move(direction=" + heading + ")"
            }
        } else ""
    }
}
```

What Is Going On?

Much of the code is identical to the previous version: we define a field for a random number generator and we parse the input command.

But then the code differentiates between the `generation` of the entity for which the control function is invoked. To do that, the code extracts the value of the generation parameter from the parameter map. Master bots have `generation 0` (zero), while mini-bots have `generation 1` (one) or higher. The game server provides this parameter to indicate to our control function for which of the controlled entities it wants a response.

```
val generation = paramMap("generation").toInt
```

The master bot will always see this value as `0` (zero), but mini-bots that we spawned will see a `1` (one). So we can follow this with an `if` expression:

```
if( generation == 0 ) {
```

In the true-block, we have the code for our master bot. It checks whether the bot has enough energy to spawn a mini-bot by checking the "energy" parameter passed by the game server:

```
if( paramMap("energy").toInt >= 100 /* ... */ ) {
```

And we use the random number generator to express that we want to launch a new missile each cycle we're invoked with a probability of 5%:

```
if( /* ... */ rnd.nextDouble() < 0.05 ) {
```

We then assemble a random launch direction for the missile:

```
val dx = rnd.nextInt(3) - 1
val dy = rnd.nextInt(3) - 1
```

and use the X and Y direction values to construct a string `direction`, which we'll use both as the spawn direction and as a custom state parameter called `heading` for our mini-bot:

```
val direction = dx + ":" + dy // e.g. "-1:1"
```

We will use this string as we concatenate and return a command string that tells the game server to spawn a new mini-bot at the given offset (relative to the master bot):

```
"Spawn(direction=" + direction + ",energy=100,heading=" + direction + ")"
```

But in addition to specifying the `direction` and `energy` parameters that are defined for the `Spawn` opcode in the server/plug-in protocol, we're passing a custom state parameter called `heading`. In doing so we exploit the fact that the server lets us use the `Spawn` to initialize the state property of the new mini-bot with arbitrary additional values before it is brought into existence in the arena.

In the false-block, we have the code for the case where the control function is invoked for any of the mini-bots we spawned. The mini-bot needs to know in which direction to move. That is where the custom state property `heading` comes into play which we initialized with `Spawn`. The mini-bot fetches this property's value and uses it as its movement direction:

```
val heading = paramMap("heading")
"Move(direction=" + heading + ")"
```

Since we already stored the `heading` in the format appropriate for x/y coordinate values (`"x:y"`), we can simply pass the string value of the property as the direction of the `Move` command.

Bot #9: Missile Launcher v2

Objective

The missile launcher in the preceding section was the second bot variant in which we had to work with X/Y coordinates. Obviously, for a bot navigating a two-dimensional environment, there'll be a lot more of this. So we will write a little class that makes dealing with such X/Y coordinates more convenient.

Departing from the style of the preceding examples for a moment we will not present the entire source code up front but rather develop it in several steps.

Step 1: using 'Tuple'

Since we already know about `Tuple` and constructing tuples with Scala is so convenient, we might start out without introducing our own type and simply work with `Tuple2`. The only code we have so far for working with X/Y coordinates simply generates a random coordinate using an instance of a random number generator:

```
val dx = rnd.nextInt(3)-1
val dy = rnd.nextInt(3)-1
val direction = dx + ":" + dy // e.g. "-1:1"
"Spawn(direction=" + direction + ",heading=" + direction + ")"
```

Note that we're omitting the `energy` parameter; the server will then simply use the minimum spawn-energy value (100 EU) as a default.

Let's define a method that constructs a tuple instead of two separate values:

```
def random() = (rnd.nextInt(3)-1, rnd.nextInt(3)-1)
```

We could then rewrite our code as follows:

```
def random() = (rnd.nextInt(3)-1, rnd.nextInt(3)-1)
val xy = random()
val direction = xy._1 + ":" + xy._2 // e.g. "-1:1"
"Spawn(direction=" + direction + ",heading=" + direction + ")"
```

Detour: parentheses on methods with no arguments

Just a very brief detour to talk about parentheses on invocations of methods that take no parameters, like `random()` above: the convention in Scala is to omit the parentheses if the method has no side effects, i.e. does not change any state (in most cases this means it does nothing beyond computing and returning a value). If the method has side effects, we'll signal that by declaring and invoking it with empty parentheses, as we do above.

Why does the method have side effects, even though it looks like it just computes and returns a value? Well, invoking `nextInt(n)` changes the internal state of the randomizer, and this is a side effect of invoking our `random()` method. That is why we declared it with parentheses.

Step 2: moving construction to an `object`

If the construction of pairs holding random values would be accessed from several call sites in our code, we'd have to supply the random number generator as a parameter instead of silently accessing the one in the surrounding object instance. Like this

```
def random(rnd: Random) = (rnd.nextInt(3)-1, rnd.nextInt(3)-1)
```

But now we have no reference to the `this` context at all, and the method could exist as a static factory function. To get a static function, we need an `object`, so let's define one:

```
object XY {
    def random(rnd: Random) = (rnd.nextInt(3)-1, rnd.nextInt(3)-1)
}
```

We can then update the code fragment within the ◊ as follows:

```
val xy = XY.random(rnd)
val direction = xy._1 + ":" + xy._2 // e.g. "-1:1"
"Spawn(direction=" + direction + ",heading=" + direction + ")"
```

Step 3: creating our own class

Now what if we wanted to transform an X/Y coordinate instance, for example by adding another X/Y value as an offset? We could extend `Tuple2` and add methods to it, but that is inadvisable for both technical reasons (`Tuple2` is a `case class`, more on those soon) and because we prefer composition over inheritance. We could also do some fancy footwork and provide new methods for the existing `Tuple2` type (using Scala's "pimp my library" approach), but that's not really called for if we expect to add a lot of methods.

No, the natural path here is to define our own `class`, just like we would in Java:

```
class XY(val x: Int, val y: Int)
```

which is roughly equivalent to the following Java code:

```
class XY {
    final int x;
    final int y;

    XY(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

We can now instantiate values of this class and like this:

```
val xy = new XY(1, 1)
```

Step 4: adding a member function

Let's add a method called `add` that will add the x and y fields of a given `XY` instance to those of the current one:

```
class XY(val x: Int, val y: Int) {
    def add(other: XY) = new XY(x+other.x, y+other.y)
}
```

We can use this method as follows:

```
val xy1 = new XY(1, 1)
val xy2 = new XY(2, 2)
val xy3 = xy1.add(xy2)
```

Keep in mind that we're striving for immutability, i.e. methods like `add()` will always return a new instance rather than modifying the existing instance.

In Scala, we can call our method almost whatever we want, including the naturally attractive +, so we'll use that:

```
class XY(val x: Int, val y: Int) {
    def +(other: XY) = new XY(x+other.x, y+other.y)
}
```

and use it as follows:

```
val xy1 = new XY(1, 1)
val xy2 = new XY(2, 2)
val xy3 = xy1 + xy2
```

As a side note, the ability to create methods that look like operators leads a lot of programmers to define methods that look like operators, resulting in libraries that define lots of methods that look like operators and are consequently completely incomprehensible to anybody who does not know the operator symbols. Don't be put off by this, though: soon enough you'll know all the operator symbols you frequently use and will know how to look up the others. And you'll probably join the crowd of programmers defining lots of methods that look like operators, although an argument could be made that code with short symbols is not necessarily better code.

Step 5: creating a factory function

Let's assemble the code we have so far:

```
class XY(val x: Int, val y: Int) {
    def +(other: XY) = new XY(x+other.x, y+other.y)
}
object XY {
    def random(rnd: Random) =
        new XY(rnd.nextInt(3)-1, rnd.nextInt(3)-1)
}
```

We have a `class` definition with a + method and a companion `object` with a `random()` factory function. Using the `new` keyword, we can construct class instances and using the factory methods we can generate random instances. Now, we know of the syntactic sugar we get with methods called `apply()`, so why don't we add one:

```
...
object XY {
    def apply(x: Int, y: Int) = new XY(x,y)
    def random(rnd: Random) = XY(rnd.nextInt(3)-1, rnd.nextInt(3)-1)
}
```

Now, instead of littering our code with `new XY(x,y)` we can say just `XY(x,y)` to construct `XY` instances:

```
val xy3 = XY(1,1) + XY(2,2)
```

Step 6: upgrading to a `case class`

Next, we'll "upgrade" our class definition to a `case class`:

```
case class XY(x: Int, y: Int)
```

The modifier keyword `case` before the `class` keyword basically instructs the compiler: "add all the stuff to my class that a sane person will usually want". This includes:

- an immutable field for each parameter of the `class` definition

- an `equals()` method that tests equality by checking each field
- a `hasCode` method that takes into account each field
- a `toString` method that takes into account each field

as well as a few others. Upgrading a `class` to a `case class` also automatically generates a companion object of the same name with an `apply()` function as a factory function that takes the parameters of the `class` as its arguments. The new code in its entirety looks like this:

```
case class XY(x: Int, y: Int) {
    def +(other: XY) = XY(x+other.x, y+other.y)
}
object XY {
    def random(rnd: Random) = XY(rnd.nextInt(3)-1, rnd.nextInt(3)-1)
}
```

Where our hand-coded `apply()` was removed from the definition of `object XY` because the compiler now generates it automatically for us at compile time.

We can now do stuff like this:

```
assert(XY(1,1) == XY(1,1))
assert(XY(1,1) != XY(2,2))
assert(XY(1,1).hashCode == XY(1,1).hashCode)
assert(XY(1,1).toString == "XY(1,1)")
assert(XY(1,1) + XY(2,2) = XY(3,3))
```

Step 7: adding a few other methods

We can add methods like the following to our class:

```
case class XY(x: Int, y: Int) {
    // ...
    def isNonZero = x!=0 || y!=0
    def isZero = x==0 && y==0
    def isNonNegative = x>=0 && y>=0

    def updateX(newX: Int) = XY(newX, y)
    def updateY(newY: Int) = XY(x, newY)

    def addToX(dx: Int) = XY(x+dx, y)
    def addToY(dy: Int) = XY(x, y+dy)

    def +(pos: XY) = XY(x+pos.x, y+pos.y)
    def -(pos: XY) = XY(x-pos.x, y-pos.y)
    def *(factor: Double) = XY((x*factor).intValue, (y*factor).intValue)

    def distanceTo(pos: XY) : Double = (this-pos).length
    def length : Double = math.sqrt(x*x + y*y)

    def signum = XY(x.signum, y.signum)

    def negate = XY(-x, -y)
    def negateX = XY(-x, y)
    def negateY = XY(x, -y)
}
```

And we can expand out static code in `object` as well:

```
object XY {
    val Zero = XY(0,0)
    val One =  XY(1,1)

    val Right      = XY( 1,  0)
    val RightUp    = XY( 1, -1)
    val Up         = XY( 0, -1)
    val UpLeft     = XY(-1, -1)
    val Left       = XY(-1,  0)
    val LeftDown   = XY(-1,  1)
    val Down       = XY( 0,  1)
    val DownRight  = XY( 1,  1)
}
```

Step 8: the modified bot

Using the `XY` class we just defined, we can update our bot's `respond()` method as follows:

```
def respond(input: String): String = {
    val (opcode, paramMap) = CommandParser(input)
    if( opcode == "React" ) {
        val generation = paramMap("generation").toInt
        if( generation == 0 ) {
            if( paramMap("energy").toInt >= 100 &&
                rnd.nextDouble() < 0.05 ) {
                val heading = XY.random(rnd)
                val headingStr = heading.x + ":" + heading.y // e.g. "-1:1"
                "Spawn(direction=" + headingStr + ", " +
                  "heading=" + headingStr + ")"
            } else ""
        } else {
            val headingStr = paramMap("heading")
            val directions = headingStr.split(':').map(_.toInt)
            "Move(direction=" + directions(0) + ":" + directions(1) + ")"
        }
    } else ""
}
```

We notice that we could also shift the encoding and decoding of the `heading` state parameter into the `XY` class:

```
case class XY(x: Int, y: Int) {
    // ...
    override def toString = x + ":" + y
}

object XY {
    // ...
    def fromString(s: String) = {
        val xy = s.split(':').map(_.toInt)
        XY(xy(0), xy(1))
    }
}
```

Note the `override` modifier on the definition of `toString`. In Scala, overriding a method that is already defined in a base class or trait requires this modifier in order to keep you from inadvertently overriding a parent method you may not even have noticed was there, thus changing the behavior of the code in unexpected ways.

We then obtain the final variant of our `respond()` method:

```
def respond(input: String): String = {
    val (opcode, paramMap) = CommandParser(input)
    if( opcode == "React" ) {
        val generation = paramMap("generation").toInt
        if( generation == 0 ) {
            if( paramMap("energy").toInt >= 100 &&
                rnd.nextDouble() < 0.05 ) {
                val heading = XY.random(rnd)
                "Spawn(direction=" + heading + "," +
                  "heading=" + heading + ")"
            } else ""
        } else {
            val headingStr = paramMap("heading")
            val heading = XY.fromString(headingStr)
            "Move(direction=" + heading + ")"
        }
    } else ""
}
```

Note that in the construction of the `Spawn` and `Move` response strings, we are not explicitly calling `XY.toString`. The compiler detects that, in order to assemble the concatenated values into a string, it needs to convert our `direction` values from type `XY` into type `String` and automatically inserts calls to `XY.toString` to accomplish that.

Step 9: changing `fromString` to `apply`

The static method `XY.fromString()` is essentially a factory method that, instead of `Int` x and y coordinates, takes a specially formatted `String` value as a parameter. So it makes sense to also define an `apply()` function on `object XY`:

```
object XY {
    def apply(s: String) : XY = {
        val xy = s.split(':').map(_.toInt)
        XY(xy(0), xy(1))
    }
}
```

As before, because the function is overloaded, we need to declare the return value explicitly to be of type `XY`.

After making this change, we can change the earlier code:

```
val heading = XY.fromString(headingStr)
```

into this slightly more concise format:

```
val heading = XY(headingStr)
```

Bot #10: Food Finder

Objective

The bot for the first time examines its surroundings, searches for the nearest food item (edible plant), runs towards it and eats it. To realize this, we need to examine the string value the server passes to the control function in the "view" parameter. The string looks something like this:

```
WWWWWWWW_____WW_____WW__M__WW_____WW____PWWWWWWWWW
```

Note that this is a simplified version; the actual string sent by the game server will contain a larger view and will be be much longer. The size of the "view" string is described in detail in the document Scalatron Game Rules.

The string has a very simple structure: it is a cell-by-cell rendering of the surroundings of the bot into an ASCII string. Every letter in the string corresponds to a cell on the game board. The view is always square and its edges span an odd number of cells.

The example string above contains 49 characters, leading us to deduce that it is a rendering of a 7x7 square of cells. Each letter corresponds to a game entity. If we break the view into multiple lines by fragmenting it at every 7th letter, we obtain:

```
WWWWWWW
W_____W
W_____W
W__M__W
W_____W
W____PW
WWWWWWW
```

The game entity character codes are described in detail in the document Scalatron Protocol. Here is an excerpt:

• **_**	empty cell
• **W**	wall
• **M**	Bot (=master; yours, always in the center unless seen by a slave)
• **P**	Zugar (=good plant, food)

So what the bot observes in the example is:

- the view is completely surrounded by wall blocks (`W`)
- the bot itself resides exactly at the center (`M`)
- at the bottom right there is an edible plant (`P`)

Given this string, the bot now needs to find the nearest food item and move towards it. We'll again develop the code to achieve this in several steps.

Step #1: a view parser class

Obviously, in a programming game in which bots must react to what they see in the game, there'll be a lot of view parsing and analyzing. So we anticipate that we'll benefit from having a smarter representation of the view string than a sequence of ASCII characters. Since the view won't change while we work on it each cycle, a simple solution would be to just wrap the string and add whatever fields and methods we may want:

```
case class View(cells: Array[Char])
```

To access the cell at a particular index, we can provide an `apply()` method, like this:

```
case class View(cells: Array[Char]) {
    def apply(index: Int) = cells.charAt(index)
}
```

We could use this code in our control function as follows:

```
val viewString = paramMap("view")
val view = View(viewString)
val cell = view(10)
```

Step #2: `XY` cell access

However, we'll want to mostly access cells using `XY` coordinates, the `2D` integer vector class we just defined in the preceding chapter. To do that, we need a method that translates `XY` coordinates into indices, like so:

```
def indexFromAbsPos(absPos: XY) = absPos.x + absPos.y * size
```

We refer to this addressing scheme as using "absolute positions" since the `XY` value is relative to the top-left corner of the view. We'll look at positioning relative to the bot's location shortly.

However, the absolute position to index translation code relies on an as-yet-undefined value `size`, which contains the number of cells per line. We could compute this on-the-fly of course by defining a method `size`:

```
def size = math.sqrt(cells.length).intValue
```

But since this method would be called a lot we probably want to cache the value. So we could convert the definition from a method into an immutable field, which would be initialized at construction time:

```
val size = math.sqrt(cells.length).intValue
```

The `View` class now looks like this:

```
case class View(cells: String) {
    val size = math.sqrt(cells.length).intValue
    def indexFromAbsPos(absPos: XY) = absPos.x + absPos.y * size
    def apply(absPos: XY) = cells.charAt(indexFromAbsPos(absPos))
}
```

We can now construct and use `View` instances as follows:

```
val viewString = paramMap("view")
val view = View(viewString)
val cell = view(XY(1,1))
```

Detour: class parameter versus field

As a quick detour, let's examine an alternative implementation. Why not shift the `size` field into the `case class` parameter list at the top and provide a factory method that computes this value up front? Like so:

```
object View {
    def apply(view: String): View =
         View(math.sqrt(view.length).toInt, view)
}
case class View(size: Int, cells: String) {
    def indexFromAbsPos(absPos: XY) = absPos.x + absPos.y * size
    def apply(absPos: XY) = cells.charAt(indexFromAbsPos(absPos))
}
```

Note that here, the Scala compiler requires that we specify the return type of the `apply()` method because the method is overloaded. How is it overloaded even though there is only one variant here? Well, we defined `View` as a `case class`, so the compiler automatically generates an `apply()` method for the parameters of the class at compile time, which is not visible for us in the source code. For overloaded functions, Scala requires an explicitly specified return type, which we therefore specify here. The reason we add another `apply()` function is that it allows users of the `View` class to omit the `size` parameter now expected by the auto-generated `apply()`.

But would this even be a good alternative? It would certainly work, but recalling that using a `case class` also generates implementations of `toString`, `equals` and `hashCode` for us that are based on the class parameter list, we'd now burden each one with the `size` parameter which really is redundant: for equality or hashing we only need the string. So while this solution is possible, it is here better to leave `size` as a field that is defined within the class body, more like a class-local constant value.

Step #3: relative addressing

Absolute addressing (relative to top-left of the view) is useful, but eventually we'll always want to guide a bot to some location, and the game server expects all movement commands to contain offsets relative to the bot position. So a very common use case is relative addressing, where we supply and receive coordinates relative to the center of the view (where the bot resides).

So we'll expand the `View` class with relative addressing, modify the `apply()` method to expect relative `XY` addresses and add a few more utility methods. Here is the enhanced `View` class:

```
case class View(cells: String) {
    val size = math.sqrt(cells.length).toInt
    val center = XY(size/2, size/2)

    def apply(relPos: XY) = cellAtRelPos(relPos)

    def indexFromAbsPos(absPos: XY) = absPos.x + absPos.y * size
    def absPosFromIndex(index: Int) = XY(index % size, index / size)
    def absPosFromRelPos(relPos: XY) = relPos + center
    def cellAtAbsPos(absPos: XY) =
         cells.charAt(indexFromAbsPos(absPos))

    def indexFromRelPos(relPos: XY) =
         indexFromAbsPos(absPosFromRelPos(relPos))
    def relPosFromAbsPos(absPos: XY) = absPos - center
    def relPosFromIndex(index: Int) =
         relPosFromAbsPos(absPosFromIndex(index))
    def cellAtRelPos(relPos: XY) =
         cells.charAt(indexFromRelPos(relPos))
}
```

Step #4: finding the nearest food item

Let's now add a method that the bot can use to find the nearest food item visible in its view. In Java style, we might declare such a method with the signature

```
def offsetToNearest(c: Char) : XY
```

where we pass the ASCII letter that we're looking for (here, `P`) as a parameter and expect the relative `XY` coordinate as a result.

What if no cell of the desired type is presently visible within the view, however? In Java, we might return `null` to indicate this special case, or, somewhat better, we might define some instance of `XY` to act as a sentinel value. We might also throw an exception (not that this would be a good choice here, since it is not really an unexpected occurrence).

But in none of these cases, what the function returns is obvious to a caller without additional documentation. We might forget to handle a `null` value and get a `NullPointerException`, or we might screw up the comparison to our sentinel value, which in addition clutters our namespace.

Fortunately, Scala offers a much better approach, which is to return an `Option` value. An `Option` is a generic abstract class whose instances are either `None` (a singleton) or a `Some` value that wraps the actual result value. Using `Option`, we can change the signature of our method to:

```
def offsetToNearest(c: Char) : Option[XY]
```

All potential users of this method immediately know that the result may be either `None` if no such character occurrence is found or `Some(xy)` if one is found. Using `Option` is absolutely the preferred way to handle such cases in Scala and for this reason most Scala programmers haven't seen a `NullPointerException` in a long time.

Adopting this signature, we can implement the method in a variety of ways. Here are three options, just to give you an idea of the options available to you.

Variant A: procedural style with `for`

Here is a version that is closest to the classic procedural style known from Java or C++ (note that this function should reside inside the View class definition):

```
def offsetToNearest(c: Char) = {
    var nearestPosOpt : Option[XY] = None
    var nearestDistance = Double.MaxValue
    for(i <- 0 until cells.length) {
        if(c == cells(i)) {
            val pos = absPosFromIndex(i)
            val distanceToCenter = pos.distanceTo(center)
            if(distanceToCenter < nearestDistance) {
                nearestDistance = distanceToCenter
                nearestPosOpt = Some(pos - center)
            }
        }
    }
    nearestPosOpt
}
```

The method uses a `for()` expression to scans all cell indices in a loop. The `for()` expression works by iterating over elements drawn from a range (of integers) generated by the expression `0 until cells.length`. It uses two mutable values to store the nearest cell found so far and its distance to the bot.

Using constructs like `for()`, it is generally straight-forward to translate Java code into Scala code. In fact, in IntelliJ IDEA, you can copy Java code from a Java source file and paste it directly as Scala code into a Scala source file, with automatic translation.

Variant B: functional style

However, for any such procedural implementation there is generally a more compact and more elegant purely functional implementation. Having mutable values, doing manual range comparisons and generally writing code that is longer than necessary introduces opportunities for errors to sneak in. So here is an example of a functional implementation that gets by without any mutable state:

```
def offsetToNearest(c: Char) = {
    val relativePositions =
        cells
        .view
        .zipWithIndex
        .filter(_._1 == c)
        .map(p => relPosFromIndex(p._2))
    if(relativePositions.isEmpty)
        None
    else
        Some(relativePositions.minBy(_.length))
}
```

Let's take this apart:

- `cells.view` indicates that the subsequent call to `zipWithIndex` should not generate a new collection instance but rather a transient view onto the existing collection. This ensures that no new copy of the collection will be made.
- `.zipWithIndex` generates a collection of pairs in which each element of the original collection (the `Char` elements of our `cells` string) is paired with its index in the collection.
- `.filter(_._1 == c)` generates a collection that contains only those elements which match the filter predicate. The predicate we use is `_._1 == c`, i.e. whether the first element of the (just-zipped) pair equals the desired character value stored in `c`. Note there that the first underscore is an anonymous reference to the parameter passed to the predicate function. More verbosely, this could be written as `pair => { pair._1 == c}`.
- `.map(p => relPosFromIndex(p._2))` transforms the collection of pairs into a collection of `XY` instances representing the relative cell coordinates of the matching cells. This is achieved by supplying a transform function to match that receives a pair as its parameter (accessible via the parameter name `p`) and maps the second element of the pair (the index) to a relative position via an invocation of `relPosFromIndex()`.
- `if(relativePositions.isEmpty)` then tests whether the resulting collection is empty (i.e. there were no matches) and if so, returns `None`. If there were matches, it uses...
- `.minBy(_.length)` to find the one nearest to the center. This works by supplying a "scoring" function to `minBy` that, for each element in the collection, returns a value for which an ordering exists (here, an `Int`). The expression `_.length` is the usual short-hand employing an underscore as a reference to the (anonymous) call parameter. The verbose variant would be `relPos => { relPos.length }`.
- `Some(...)` finally wraps the result into a `Some` instance, matching expected the `Option[XY]` return type.

Variant C: procedural style with `while`

So the functional style is very concise and for many reasons preferable for the vast majority of code one might write. However, you may have noticed that a lot is going on here, with several traversals and potential interim copies of the collection being generated in calls like `zipWithIndex`. Some of this overhead can be eliminated or minimized, for example by using `view` to avoid a copy.

But what about the 1% of your code where you really, absolutely require optimal efficiency? The answer is that in those rare cases, you may want to write procedural code with mutable state. And the good news is that you can do that. Here is a more performance-conscious version of the `offsetToNearest` method that uses a `while` loop to do an efficient traversal:

```
def offsetToNearest(c: Char) = {
    var nearestPosOpt : Option[XY] = None
    var nearestDistance = Double.MaxValue
    for(i <- 0 until cells.length) {
        if(c == cells(i)) {
            val pos = absPosFromIndex(i)
            val distanceToCenter = pos.distanceTo(center)
            if(distanceToCenter < nearestDistance) {
                nearestDistance = distanceToCenter
                nearestPosOpt = Some(pos - center)
            }
        }
    }
    nearestPosOpt
}
```

The traversal with `while` will often be slightly faster than with `for` because `for` will be translated by the compiler into functional code using closures, which carry a slight overhead.

The takeaway is that initially, Scala allows you to start coding like you would in Java, using familiar constructs like `for` loops. As you learn about functional programming, Scala lets you write concise, error-minimizing functional code for the 99% of your code where productivity trumps performance. In the other 1% of your code where performance trumps productivity, Scala also lets you write procedural code that is as fast as anything you might write in Java.

Variant D: but ... there is also parallelism

For sequential code, that is. If you expect to be able to exploit multiple CPU cores, the picture changes again dramatically. Now, purely functional code that embraces immutability gains an upper hand because it is so much more readily parallelizable.

Let's imagine for a moment that we were not processing a few hundred characters in a bot view, as we expect to be doing here, but rather millions. What would it take to parallelize this code to exploit multiple cores? For the procedural code, use your own experience to judge the amount of work this would take.

For the functional version, here is the only change we'd have to make: we'd replace the reference to `cells` at the top with `cells.par`, turning it into a parallel collection. That's it.

Here is a real-life example from the Scalatron game server. The primary bottleneck for the server is the drawing code, which uses unaccelerated Java drawing code. On an 8-core system, the single-threaded variant of the server generates close to 100% CPU load, i.e. it loads only a single CPU, as would be expected. After spending an hour or so to add some hand-crafted threading code to introduce double buffering and parallelize drawing and state updating, the CPU load rises to 180%. But then, after spending about ten seconds to add a call to `.par` in the bot response generation code to parallelize all bot control function invocations, the load rises to 300%.

Step #5: exploiting the nearest food item

Now that we have the `offsetToNearest()` method available, we can use it in our bot's control function like this:

```
def respond(input: String): String = {
    val (opcode, paramMap) = CommandParser(input)
```

```
        if( opcode == "React" ) {
            val viewString = paramMap("view")
            val view = View(viewString)
            view.offsetToNearest('P') match {
                case Some(offset) =>
                    val unitOffset = offset.signum
                    "Move(direction=" + unitOffset + ")"
                case None =>
                    ""
            }
        } else ""
    }
```

Let's look at what this code does in more detail.

What do `match` and `case` do?

In Java, we can handle situations in which a value can take on one of several possible values with a `switch` statement. Scala provides pattern matching via `match` and `case`, which is more general. To see how this works, let's isolate the associated example code:

```
view.offsetToNearest('P') match {
    case Some(offset) => /* handler code */
    case None => /* handler code */
}
```

The value to be examined precedes the `match` keyword. Here, we wish to examine the value returned by `view.offsetToNearest('P')`, i.e. an `Option` value that may or may not contain an `XY` instance. Within the curly braces that follow after the `match` keyword, we define the cases to be handled. We do this by using the `case` keyword, followed by the pattern we want to match against, followed by a => symbol, followed by the handler code for that case.

So in the code above, since we know the result type of `offsetToNearest` to be an `Option[XY]`, we have two handlers: one for the case where `None` is returned and one for the case where `Some (offset)` is returned.

Like `if` or `try`, a `match` expression yields a value. It is the value associated with the first matching handler. Here, the handler for `None` simply yields the empty string; the handler for `Some(offset)` assembles a `Move` command that instructs the bot to move in the direction of the nearest visible plant.

Some other notes on `match` and `case`:

- In Scala, unlike Java, there is no fall-through from one `case` to the next.
- The last pattern can be an underscore ('_'), which matches any pattern:
  ```
  foo match {
      // handlers for specific cases
      case _ => // handler for everything else
  }
  ```
- Patterns can be constants, like so:
  ```
  foo match {
      case "Frank" => // handler for "Frank"
      case "Joe" => // handler for "Joe"
      case "Tina" => // handler for "Tina"
  }
  ```

- Patterns can be types, with a symbol name to bind the value to if the type matches, like so:
  ```
  foo match {
      case n: Int => println("an Int: " + b)
      case s: String => println("a String: " + s)
      case c: Char => println("a Char: " + c)
  }
  ```

Caveat: in Scala, since it is based on and constrained by the Java Virtual Machine and its generic type system, generic types are subject to type erasure just like they are in Java. Due to this limitation, `case` patterns cannot distinguish between variants of a generic type by subtype. Code like the following will therefore not work:

```
anOption match {
    case a: Some[A] => // ...
    case b: Some[B] => // ...
}
```

The only exception to this rule is `Array`, which is not subject to type erasure.

What does case `Some(offset)` do?

The `match` expression we just examined contains one `case` handler that is neither a constant nor (just) a type:

```
... match {
    case Some(offset) => // handler code
}
```

Here, Scala extracts the value contained in the `Some` instance for us and binds it to a symbol `offset`, which is of type `XY` (since the matched value was an `Option[XY]`).

This parameter extraction is available for classes whose companion object implements a (static) method `unapply()`, which the compiler uses to extract the class parameters.

Complete Listing

Here is a complete listing for the bot, which you can cut and paste into a single source file. Some unused functions were omitted.

```
class ControlFunctionFactory {
    def create = new ControlFunction().respond _
}

class ControlFunction {
    def respond(input: String): String = {
        val (opcode, paramMap) = CommandParser(input)

        if( opcode == "React" ) {
            val viewString = paramMap("view")
            val view = View(viewString)
            view.offsetToNearest('P') match {
                case Some(offset) =>
                    val unitOffset = offset.signum
                    "Move(direction=" + unitOffset + ")"
                case None =>
                    ""
            }
        } else ""
```

```
    }
}


case class View(cells: String) {
    val size = math.sqrt(cells.length).toInt
    val center = XY(size/2, size/2)

    def indexFromAbsPos(absPos: XY) = absPos.x + absPos.y * size
    def absPosFromIndex(index: Int) = XY(index % size, index / size)
    def absPosFromRelPos(relPos: XY) = relPos + center
    def cellAtAbsPos(absPos: XY) =
           cells.charAt(indexFromAbsPos(absPos))

    def indexFromRelPos(relPos: XY) =
           indexFromAbsPos(absPosFromRelPos(relPos))
    def relPosFromAbsPos(absPos: XY) = absPos - center
    def relPosFromIndex(index: Int) =
           relPosFromAbsPos(absPosFromIndex(index))
    def cellAtRelPos(relPos: XY) =
           cells.charAt(indexFromRelPos(relPos))

    def offsetToNearest(c: Char) = {
        val relativePositions =
           cells.view.zipWithIndex
           .filter(_._1 == c)
           .map(p => relPosFromIndex(p._2))
        if(relativePositions.isEmpty) None
        else Some(relativePositions.minBy(_.length))
    }
}


case class XY(x: Int, y: Int) {
    override def toString = x + ":" + y
    def +(pos: XY) = XY(x+pos.x, y+pos.y)
    def -(pos: XY) = XY(x-pos.x, y-pos.y)
    def distanceTo(pos: XY) : Double = (this-pos).length
    def length : Double = math.sqrt(x*x + y*y)
    def signum = XY(x.signum, y.signum)
    def toEntityName = x + "_" + y
}

object CommandParser {
    def apply(command: String) = {
        def splitParam(param: String) = {
            val segments = param.split('=')
            if( segments.length != 2 )
                throw new IllegalStateException(
                  "invalid key/value pair: " + param)
            (segments(0),segments(1))
        }

        val segments = command.split('(')
        if( segments.length != 2 )
```

```
            throw new IllegalStateException(
                "invalid command: " + command)

        val params = segments(1).dropRight(1).split(‘,’)
        val keyValuePairs = params.map( splitParam ).toMap
        (segments(0), keyValuePairs)
    }
}
```

Debugging Your Bot

Approaches

One important area was omitted from the tutorial so far: how do you debug your bot code? There are multiple options. Which one is best depends on your goals and your development environment. Here are the three main approaches:

- Using the Scalatron IDE for browser-based development: since compilation and execution occur on the server computer (to which you may not have any other access) and the IDE is fairly basic, there is no way to debug the code by setting break-points ot single-stepping through it. You can, however, single-step through the simulation and inspect the input and state of your entities (bots and mini-bots), as well as have your bot output debug information.
- Using IntelliJ IDEA with a local test server: if you are using IDEA to build your bots (the same may also be true for the Scala IDE), you can install and run a local test server, have it load your plug-in and then debug with breakpoints and single-stepping within the IDE. This approach is probably the most powerful. It is recommended for serious enthusiasts that want to build really clever and complex bots. For more information, see the chapter on debugging with IDEA in the Player Setup guide.
- Using local builds with a central tournament server: the only option for debugging in this scenario is to use server-side logging into a user-specific directory.

Some of these options will be elaborated on in the next few bot examples of the tutorial.

Making State Visible

The debugging approach with longest history is also the simplest: use outputs generated by your program to trace its behavior at points of interest and to narrow the location of aberrations. The Scalatron protocol provides several mechanisms for doing this:

- Status() -- using the `Status` opcode, a bot or mini-bot can display a short message above its location on the tournament screen. This can be used to indicate some aspects of the state of the bot in a way that is visible while it is running in a multi-player tournament round. You can, for example, display messages like "Searching..." while the bot is scouting for food, "Approaching..." while it is closing in on a food item, etc.
- Say() -- using the `Say` opcode, a bot or mini-bot can leave a textual "bread-crumb" at its current location on the screen. The bread-crumb will stay in its initial location even when the bot moves away and will gradually fade away over the course of a few seconds. Say is thus a good way to generate records of events or state changes (as opposed to states themselves). They remain visible even as the simulation moves on. Examples include messages like "Spotted Food", "Proximity Fuse" or "Fighting->Harvesting".
- Log() -- using the `Log` opcode, a bot or mini-bot can record a multi-line debug message in its entity state. This debug message can be inspected in the browser-based Scalatron IDE while running in a sandbox. You can essentially record whatever you want, with the unfortunate limitation that the server's relatively dumb parser must not be confused by it. So certain characters are not permitted: comma (,), equals sign (=), parentheses ((,)), pipe (|).
- Set() -- using the `Set` opcode, a bot or mini-bot can record key/value pairs into the state parameters which the server maintains for it. These state parameters can be inspected in the browser-based Scalatron IDE while single-stepping through a sandboxed simulation.
- Displaying a window -- open a window (e.g. using an AWT `Frame`) and render some debug information in real-time. This makes sense only if your code uses complex data structures that

would be too cumbersome to analyze in a log file. Note also that while you can, of course, do this on a locally running test game server at any time, on the shared game server whose display is projected for everyone to see this is pretty intrusive.

- Logging to disk – in the `Welcome` message sent by the server, record the path to the directory in which your bot plug-in resides as seen by the server. Then add debug logging instructions to your code that make the state of your program visible at critical points by writing into a log file in that directory. This technique is explained in detail soon via a bot example.

Example Bot: Debug Logger

Objective

This section describes how to log debug information to disk while your bot is running inside a multi-player tournament on a central game server. One reason this crude debugging mechanism is required is that your code is executed remotely on another compute inside a Java Virtual Machine that you have no access to. There is therefore no way to set breakpoints, to single-step through your code or to examine the values of variables at run-time, as you would with a local debugger.

Note that this debugging approach is a measure of last resort. If you are working in the browser-based Scalatron IDE or if you are using a local IDE and debugging your bot in a local server is sufficient, you can skim this example and focus only on the discussion of the Scala code for its own sake.

Debug Logging

Since your code is running alongside multiple other bots on the server, simply sending output to the console via `println` is not a viable option, except for extremely rare messages. Log output would scroll past way too quickly and no-one would be able to see what is going on.

The recommended approach is therefore to log your debug output into a plug-in specific log file. But where to place that file? Your plug-in will need to use a path that is valid in the context where it is being executed (i.e., on the computer where the game server is running). To keep things simple and to avoid having you hard-code paths into your plug-in, the server provides the path of the directory from which your plug-in was loaded as a parameter to the "Welcome" command:

```
Welcome(name=string,path=string,round=int)
```

You can cache the `path` and `round` values for later in fields of your bot class.

A second issue is logging overhead. If every plug-in were to log lots of data every step of every round for every entity (master bot and mini-bots), there is a risk that the game server would experience significant slowdown and the experience would be spoiled for everyone. It is therefore recommended that you log information only during certain steps of the simulation (say, at step zero or every 100th step). It is up to you whether you append multiple data points to a single log file or whether you generate separate log files for each such log-relevant event.

The example code below illustrates how one might do this.

Source Code

```
import java.io.FileWriter

// omitted: class ControlFunctionFactory, object CommandParser

class ControlFunction() {
    var pathAndRoundOpt : Option[(String,Int)] = None

    def respond(input: String): String = {
        val (opcode, paramMap) = CommandParser.apply(input)
        opcode match {
            case "Welcome" =>
                // first call made by the server.
                // We record the plug-in path and round index.
                val path = paramMap("path")
                val round = paramMap("round").toInt
                pathAndRoundOpt = Some((path,round))

            case "React" =>
                // called once per entity per simulation step.
                // We check the step index; if it is a multiple of
                // 100, we log our input into a file.
                val stepIndex = paramMap("time").toInt
                if((stepIndex % 100) == 0) {
                    val name = paramMap("name")
                    pathAndRoundOpt.foreach(pathAndRound => {
                        val (dirPath,roundIndex) = pathAndRound
                        val filePath = dirPath + "/" + name + "_" +
                                    roundIndex + "_" + stepIndex + ".log"
                        val logFile = new FileWriter(filePath)

                        logFile.append(input)   // log the input

                        // if we logged more stuff, an occasional newline:
                        logFile.append('\n')

                        // close the log file to flush what was written
                        logFile.close()
                    })
                }

            case "Goodbye" =>
                // last call made by the server. Nothing to do for us.
            case _ =>
                // plug-ins should simply ignore unknown command opcodes
        }
        ""      // return an empty string
    }
}
```

What is going on?

We added a field to the `ControlFunction` class that will hold the directory path and the round index as an `Option[Tuple2]`:

```
var pathAndRoundOpt : Option[(String,Int)] = None
```

Initially, the field will hold the value `None`. Once the server tells us what the actual values should be, the field will hold a `Some` value storing the plug-in directory path and the round index as a pair. We'll explain the reason for the explicit type specification later.

Instead of the earlier `if` clause that tested for just the "React" opcode, we now want to distinguish multiple possible values (and potentially complain about unknown opcodes). So a `match` expression is better suited. In it, we distinguish between the three opcodes "Welcome", "React" and "Goodbye" (see the Scalatron Protocol documentation for details).

In the opcode handler for "Welcome", we extract the values the server gives us for the directory path and the round index and update the field:

```
val path = paramMap("path")
val round = paramMap("round").toInt
pathAndRoundOpt = Some((path,round))
```

In the opcode handler for "React", we extract the index of the current simulation step:

```
val stepIndex = paramMap("time").toInt
```

and test whether it is evenly divisible by 100:

```
if((stepIndex % 100) == 0) {
```

The conditional code will therefore be executed only every 100th simulation step. In it, we extract the name of the entity for which the server is invoking the control function (for a master bot this will be the name of the plug-in; for mini-bots it will be whatever the plug-in told the server their name should be in `Spawn`):

```
val name = paramMap("name")
```

we then test whether the optional value `pathAndRoundOpt` was set (using a `foreach` instead of `match` or `if`, a technique we'll explain in just a second) and, if so, we extract the plugin directory path and the round index into two local values `dirPath` and `roundIndex` (using the extraction syntax for `Tuple2` we already saw earlier)

```
val (dirPath,roundIndex) = pathAndRound
```

and then construct a log file name that incorporates all of the distinguishing elements of the debug "event": the entity name, the round index and the step index. Since the file resides in the plug-in's directory, there is no need to incorporate the plug-in name.

```
val filePath = dirPath + "/" +
               name + "_" +
               roundIndex + "_" +
               stepIndex + ".log"
```

We then create a `FileWriter` instance for the log file

```
val logFile = new FileWriter(filePath)
```

Now we can log into the file whatever we want. The example just logs the command the server sent, plus a newline (just so you know how):

```
logFile.append(input)   // log the input
logFile.append('\n')
```

and then closes the log file to flush any buffered contents to disk:

```
logFile.close()
```

That's it. In "Goodbye" we don't need to do anything for the example. If we had kept a log file open to log information across multiple steps, however, we'd put the `logFile.close()` into that handler.

Why is the type of `pathAndRoundOpt` explicitly specified?

Note that in the field declaration

```
var pathAndRoundOpt : Option[(String,Int)] = None
```

we explicitly specify the type as `Option[(String,Int)]` rather than letting the compiler infer it for us. The reason this is necessary is that on its own, the compiler would (correctly) infer the type by looking at the initialization value and would conclude it should be the type of the singleton object `None` (which could be written down as `None.type`).

This would later lead to a compile time error when we attempt to assign the updated `Some` value, since that type does not match what is expected.

To make this work as intended, we need to tell the compiler that what we really want is for the field to have the parent type of `None` and `Some`, which - taking the polymorphic type of the wrapped value into account - is `Option[(String,Int)]`. Hence the explicit specification of the field's type.

What does `pathAndRoundOpt.foreach()` do?

The objective of that code is to test whether the `Option` value is a `None` or a `Some` and, if it is a `Some`, to extract the wrapped value so we can work with it.

Let's first look at a "brute force", more procedural implementation of the same code:

```
if(pathAndRoundOpt.isDefined) {
    val pathAndRound = pathAndRoundOpt.get
    /* ... */
}
```

We can do the same with a `match` expression, which many Scala programmers consider preferable compared to `if`:

```
pathAndRoundOpt match {
    case Some(pathAndRound) => /* ... */
    case None => // do nothing
}
```

But the code in the example illustrates another possibility: use `foreach`. As the name implies, `foreach` is a method available on collection types that executes a function for each element of the collection. The reason we can use this on `Option` is that an `Option` value mimics a collection that is either empty (if the `Option` value is `None`) or that contains a single element consisting of the wrapped value (if the `Option` is `Some`). So the code we used:

```
pathAndRoundOpt.foreach(pathAndRound => {
    val (dirPath,roundIndex) = pathAndRound
    /* ... */
})
```

works because the closure `pathAndRound => {/*...*/}` is only executed if the `Option` is a `Some`, i.e. if indeed the "Welcome" command was already received and the directory path and round index have become available.

For the example shown here, the solution feels a bit contrived since there are really no savings in terms of code size. However, the approach demonstrates a more general principle that is extremely useful and in many situations much more elegant than `if` or `match`, not least because it also extends to other collection methods like `map`, where the benefits may be more evident. In `map`, we can provide a function that transforms the wrapped value into a new value, resulting in an `Option` wrapping that new value without the need to handle the `None` case at all.

Appendix A: Cheat Sheet for `String`

Constructing a String

```
val a = "A"
val ab = "A" + "B"
val a1b = "A" + 1 + "B"
```

Taking a String apart

To break a string into segments at all occurrences of a character:

```
val segments = string.split('=')
```

To retrieve the character at a specific index within the string:

```
val c = string.charAt(5)
val c = string(5)
```

Comparing Strings

```
string.compareTo("other")
string.compareToIgnoreCase("other")
```

Miscellaneous

```
string.isEmpty
string.head     // first character
string.last     // last character
string.filter(c => c == 'A')
string.foreach(c => println(c))
string.map(c => c+1)
"abc".zipWithIndex // => ('a',0), ('b',1), ('c',2)
```

Appendix B: Cheat Sheet for `Map`

Note that this cheat sheet covers only the immutable variant of `Map`.

Constructing a Map

Constructing a `Map` from key/value pairs:

```
val map = Map("A" -> 1, "B" -> 2, "C" -> 3)
```

which uses syntactic sugar for tuples and is identical to:

```
val map = Map(("A",1), ("B",2), ("C",3))
```

Converting a `List` of `Tuple2` into a `Map`:

```
val map = List(("A",1), ("B",2), ("C",3)).toMap
```

Constructing an empty map:

```
val map = Map.empty[String,Int]
```

Updating a Map

```
val newMap = map.updated(key, value)
```

Lookups

If the key must exist (exception thrown if not):

```
val value = map.apply(key)
```

which is identical to:

```
val value = map(key)
```

If the key may or may not exist:

```
val valueOpt = map.get(key) // returns Option[]
```

Keys and Values

To obtain an iterable collection of the map's values:

```
val values = map.values
```

To obtain an iterable collection of the map's keys:

```
val keys = map.keys
```

Transformations

To apply a transformation function to every element, obtaining a new, transformed `Map`:

```
val newMap = map.map(entry => (entry._1, entry._2))
```

To apply a transformation function to all values, obtaining an `Iterable` collection:

```
val newMap = map.values.map(value => { (*...*/ })
```