SIGCSE Workshop on Scala

Release 1.0

Lewis, Läufer, and Thiruvathukal

27-October-2014 16:15:35

1 Table of Contents **3**
1.1 About Scala . . . 3
1.2 Installing Scala . . . 6
1.3 Downloading these Materials . . . 7
1.4 Scala in CS1 . . . 8
1.5 CS2 . . . 35
1.6 Build Tools for Scala . . . 45
1.7 Web Application and Services . . . 46
1.8 Mobile Application Development with Android . . . 49
1.9 Basic Parallelism using Par . . . 50
1.10 Parallelism using Actors . . . 57
1.11 Programming Language Topics . . . 66

2 Indices and tables **69**

Bibliography **71**

Index **73**

Mark Lewis
Trinity University
Department of Computer Science

Konstantin Läufer and George K. Thiruvathukal
Loyola University Chicago
Department of Computer Science

CHAPTER

ONE

TABLE OF CONTENTS

1.1 About Scala

"Scala is an object-functional programming and scripting language for general software applications, statically typed, designed to concisely express solutions in an elegant, type-safe and lightweight (low ceremonial) manner. Scala has full support for functional programming (including currying, pattern matching, algebraic data types, lazy evaluation, tail recursion, immutability, etc.). It cleans up what are often considered poor design decisions in Java (such as type erasure, checked exceptions, the non-unified type system) and adds a number of other features designed to allow cleaner, more concise and more expressive code to be written." [ScalaWikipedia]

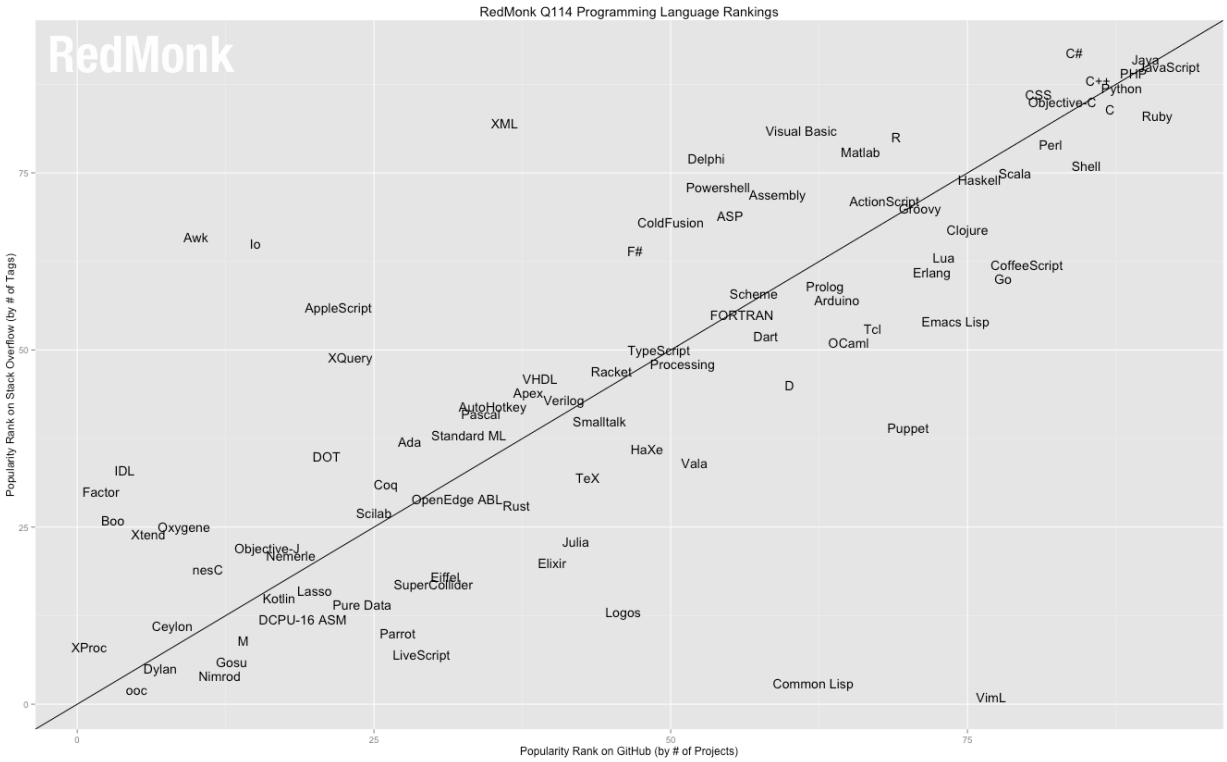
Scala has no connection to *La Scala* or *Teatro all Scala* (the world renowned opera house in Milan), but we (especially Dr. Thiruvathukal) love the name because it addresses the traditions of programming languages and scalable computing, while *La Scala* is the title of one of his favorite jazz piano albums by Keith Jarrett, who often names his concerts by the venues where he performs. [LaScalaConcert]

We'd like to think that the Scala language represents the design ideal of being "small and beautiful" but also great for computer science and practical problem solving. We hope you agree!

Source is Wikimedia Commons [LaScala]

1.1.1 Why Scala?

Redmonk Language Rankings



Scala is ranked 13 on this year's listing. This comes courtesy of [RedmonkPL]

Technology Radar

Thoughtworks maintains a resource known as the Technology Radar [TWTechRadar]. Four classifications:

- Adopt
- Trial
- Assess
- Hold

Scala is listed under *Adopt*.

Language Complexity

See presentation for now, http://goo.gl/Q68fA.

1.2 Installing Scala

1.2.1 Prerequisite: JDK

As a prerequisite to Scala development, we recommend that you install the Oracle Java Development Kit (Java 7 SE 7u51 or later). While you can work with OpenJDK and other VM implementations to run Scala, our initial testing is that the best experience and performance comes from the latest stable release of the Java 7 Platform.

- Oracle Java Platform Downloads

1.2.2 Command-Line Tools

We do most of our CS1 instruction using command line tools. If you want to go this route, you can perform a standard Scala 2.10.x standalone install using your system's package manager or manually from

- http://www.scala-lang.org/download/

This is generally a good choice for projects without external dependencies.

For projects with external dependencies (such as unit testing), we also recommend using sbt (Simple Build Tool for Scala). You can install version 0.13.x or newer of sbt using your system's package manager or manually from

- http://www.scala-sbt.org/release/docs/Getting-Started/Setup.html

You may also need to configure the paths on your machine to make these tools easy to work with.

1.2.3 Text Editors

To go along with the command-line tools, you will need a text editor that you like to work with.

- On Linux, we recommend Vim or Emacs. OS X Terminal also supports both via the command line or via the MacVim and AquaMacs projects.
- The Sublime Text Editor is also wildly popular among agile developers and works on all major platforms. While not free/open source, it can be used for an indefinite period for free (with only occasional nagging suggesting you upgrade to the paid version). George uses this editor not only for writing Scala but also for editing reStructuredText (the source code for these notes).

1.2.4 IDE Option: JetBrains IntelliJ IDEA

Many faculty teaching introductory CS courses prefer an Integrated Development Environment (IDE). We recommend IntelliJ IDEA, which is growing in popularity over Eclipse and preferred by many of us. You can get the Community edition for free from the following URL and then install the Scala plugin through the plugin manager.

- http://www.jetbrains.com/idea/download/

The IntelliJ IDEA Scala plugin undergoing active development, and there is a tradeoff between stability and features/bug fixes. For advanced Scala development, you may find yourself wanting to be more bleeding edge. To this end, we recommend the current early access version:

- http://confluence.jetbrains.com/display/IDEADEV/IDEA+13.1+EAP

When you install the Scala plugin through the plugin manager, you will automatically get the version that matches that of IDEA. There are still a few glitches, but it has gotten a lot better since January 2014. In particular, compilation (and execution of Scala worksheets) has become much faster.

To work around false compilation errors in Scala worksheets, we also recommend a standalone installation of Scala (sufficient for projects without external dependencies) or sbt.

1.2.5 IDE Option: Eclipse Scala IDE

The official Scala IDE is provided as an Eclipse bundle that has Scala already installed. It will work on all platforms with very minor differences. The following link will take you there.

- http://scala-ide.org/download/sdk.html

1.3 Downloading these Materials

1.3.1 scalaworkshop.com (Primary)

Table 1.1: Available Tutorial Formats

Format	URL
Web Pages	http://scalaworkshop.com/
Web Pages (offline ZIP)	http://scalaworkshop.com/dist/html.zip
PDF	http://scalaworkshop.com/latex/sigcse-scala.pdf
ePub (Experimental)	http://scalaworkshop.com/epub/sigcse-scala.epub

1.3.2 scalaworkshop.cs.luc.edu (Mirror)

Table 1.2: Available Tutorial Formats

Format	URL
Web Pages	http://scalaworkshop.cs.luc.edu/
Web Pages (offline ZIP)	http://scalaworkshop.cs.luc.edu/dist/html.zip
PDF	http://scalaworkshop.cs.luc.edu/latex/sigcse-scala.pdf
ePub (Experimental)	http://scalaworkshop.cs.luc.edu/epub/sigcse-scala.epub

1.3.3 Source Code Repositories

The following table contains links to direct download the source code for all of the long examples in our tutorial. We note that shorter, interactive sessions shown in the tutorial (especially the ones meant to be done simultaneously by the reader) are not in any repository. You must copy/paste to try them out on your own!

Table 1.3: ZIP Format

Description	URL
integration-scala	https://bitbucket.org/loyolachicagocs_plsystems/integration-scala/get/default.zip
lcs-systolicarray	https://bitbucket.org/loyolachicagocs_plsystems/lcs-systolicarray-scala/get/default.zip
hpjpc	https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java/get/default.zip
introcs-scala-examples	https://bitbucket.org/loyolachicagocs_books/introcs-scala-examples/get/default.zip
numerical-explorations-scala	https://bitbucket.org/loyolachicagocs_plsystems/numerical-explorations-scala/get/default.zip

The following table contains links to the source code repository (usually Bitbucket but sometimes GitHub). You will need to be fluent with version control clients (e.g. `hg` or `git`) to check out the code from these repositories.

Note: This option is recommended for experts only!

Table 1.4: Repositories

Description	URL
integration-scala	https://bitbucket.org/loyolachicagocs_plsystems/integration-scala
lcs-systolicarray	https://bitbucket.org/loyolachicagocs_plsystems/lcs-systolicarray-scala
hpjpc	https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java
introcs-scala-examples	https://bitbucket.org/loyolachicagocs_books/introcs-scala-examples
numerical-explorations-scala	https://bitbucket.org/loyolachicagocs_plsystems/numerical-explorations-scala

Alternately, you can clone https://bitbucket.org/sigcse2013scala/notes and run the `fullmake.sh` script, which will pull all examples (assuming `hg` is installed on your computer) into the `examples` subdirectory.

1.3.4 Comments?

Please don't hesitate to contact the authors.

Table 1.5: Repositories

Name	E-mail
Mark Lewis	mlewis@trinity.edu
Konstantin Läufer	laufer@cs.luc.edu
George K. Thiruvathukal	gkt@cs.luc.edu

1.4 Scala in CS1

Warning: This section is still in draft form but is nearly complete in terms of examples, subject to editing. There might still be a few rough spots. Comments welcome to gkt@cs.luc.edu.

This is an elaboration of our Google presentation slides: http://goo.gl/Q68fA.

1.4.1 Motivating Scala in CS1

- Programming in the small
- REPL - Read-Evaluate-Print-Loop
- Scripting Environment
- Libraries allow for interesting code (JVM)
- Static type checking
- Uniform syntax
- Everything is a method call
- Powerful collections

Our motivation is to have the best of both worlds:

- Concise script-ability of languages like Python and Ruby

- Static type checking in the small makes it (generally) easy to fix compilation errors

The emerging *worksheet* model makes it possible to give students many examples that just work and can be adapted to solve new problems.

1.4.2 Getting Started

Like many *agile* languages, Scala embraces the notion of being able to get started using a REPL (Read-Evaluate-Print-Loop), which allows for interactive execution and spontaneous feedback.

We're assuming the reader can set up Scala and Java (needed to run Scala, a language that targets the JVM primarily). Once you have Scala and Java installed, you can open up an interactive session using the *command line*. (For those who don't prefer the command line, especially on Windows, we recommend installing IntelliJ Community Edition and the Scala plug-in. This will allow you to get an interactive session as well.)

Hello, World

```
$ scala-2.10
Welcome to Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_45).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello, World")
Hello, World
```

Types - "Lack" of Primitives

```
scala> 1
res1: Int = 1
```

In the following, the user types `1.` and then hits the `<tab>` key–known as tab completion–to show the available operations and methods available for an `Int`.

```
scala> 1.<tab>
%               &               *               +               -
/               >               >=              >>              >>>
^               asInstanceOf    isInstanceOf    toByte          toChar
toDouble        toFloat         toInt           toLong          toShort
toString        unary_+         unary_-         unary_^         |
```

Literals are mostly what you expect

Integers

```
scala> val a = 95
a: Int = 95

scala> val b = -95
b: Int = -95
```

Octal and Hexadecimal

```
scala> val c = 039
<console>:1: error: malformed integer number
       val c = 039
               ^

scala> val c = 037
warning: there were 1 deprecation warning(s); re-run with -deprecation for details
c: Int = 31

scala> val d = 0xff
d: Int = 255

scala> val d = 0xffac
d: Int = 65452
```

Double and Float

```
scala> val e = 1.34e-7
e: Double = 1.34E-7

scala> val f = 1.34e-7f
f: Float = 1.34E-7
```

Statements vs. Expressions

Scala eschews statements as a general rule in favor of expressions (which produce values), owing to its functional heritage, which emphasizes the composition of functions to effect computation.

In practice, however, many expressions, however, behave like statements as they produce `Unit` as a result.

A notable consequence of this language decision is that some constructs are not provided in Scala, including the familiar break and continue (found in loops).

Everything is an Object

Scala is not designed in a vacuum. Its heritage is one shared with Java, even if its ideas have value above and beyond Java. Everything is achieved in Scala with *objects*, although you can do a tremendous amount of programming before ever writing a *class* per se.

There are two consequences to this decision:

- There are no primitive types per se. Every one of the literals discussed previously is an object. This eliminates the need to distinguish `int` from `Integer` (a major pain point for Java prior to boxing/unboxing support).
- There are very few keywords in Scala. Even when you encounter something that you think is a keyword, it is often just a method call. A really good example is the concept of mapping (map) from functional programming, which is a method common to just about every Scala collection.

Let's look at a couple of examples of methods. Here is an example of how to see the methods available to an `Int` instance:

```
scala> a.<tab>
%               &               *               +               -
/               >               >=              >>              >>>
^               asInstanceOf    isInstanceOf    toByte          toChar
toDouble        toFloat         toInt           toLong          toShort
toString        unary_+         unary_-         unary_^         |
```

Where you see <tab>, you can use a feature known as *tab completion* in the REPL to see the options available to value a. You'll notice one thing immediately, especially if you are familiar with Java: operator overloading! That's right, every operator is a method.

Let's use the + method:

```
scala> a.+(3)
res5: Int = 98

scala> a + 3
res6: Int = 98
```

Let's convert an Int to a Float:

```
scala> val b = a.toFloat
b: Float = 95.0
```

You can also invoke methods like toFloat (which take no parameters) without using dots. We will take advantage of this syntax as part of good Scala style (in many of our examples).

```
scala> val b = a.toFloat
b: Float = 95.0
```

We're going to look at more *advanced* objects later (Scala collections) but this should give you a taste of what is possible.

Tuples

A language feature that was popularized (but not invented) by the Python language are tuples. Tuples eliminate the need for unwanted uses of a class for grouping multiple values. Let's take a quick look.

```
scala> val t = (3, 4)
t: (Int, Int) = (3,4)

scala> val u = (3.0, 4.0)
u: (Double, Double) = (3.0,4.0)

scala> val v = (3.0, 4)
v: (Double, Int) = (3.0,4)
```

Notice that Scala infers the type of each one of these value definitions.

When working with a tuple, you're really working with an object. You can inspect the methods that are available using the tab completion method shown previously.

```
scala> t.
_1                _2                asInstanceOf      canEqual
copy              isInstanceOf      productArity      productElement
productIterator   productPrefix     swap              toString

scala> u.
_1                _2                asInstanceOf      canEqual
copy              isInstanceOf      productArity      productElement
productIterator   productPrefix     swap              toString

scala> v.
_1                _2                asInstanceOf      canEqual
copy              isInstanceOf      productArity      productElement
productIterator   productPrefix     swap              toString
```

You can inspect the components of a tuple by using the _1, _2, ... methods.

```
scala> t._1
res9: Int = 3

scala> t._2
res10: Int = 4
```

You'll obviously not want to use these names to refer to the components of a tuple. Using pattern matching, you can extract the components of a tuple and *bind* them to proper names. For example, if your tuple represents an (x, y) pair, you are likely to use a match expression like this:

```
scala> t match { case (x, y) => println(s"($x, $y)") }
(3, 4)
```

Semicolon Inferencing

Python programmers already know and love not having to deal with semicolons. Scala follows this excellent practice as well.

You can do this:

```
scala> val x = 30;
x: Int = 30
```

But this works just as well and is the preferred way to write Scala code:

```
scala> val y = 30
y: Int = 30
```

Simple Input (and Output)

Much like Python and Java, import can be used to experiment with library objects and functions, even before you know how to create classes:

```
scala> import scala.tools.jline.console.ConsoleReader

scala> val input = new ConsoleReader
input: scala.tools.jline.console.ConsoleReader = scala.tools.jline.console.ConsoleReader@3ec642e5
```

Then you can inspect the capabilities of the ConsoleReader by using tab completion (as shown before)

```
scala> input.r<tab>
readCharacter     readLine          readVirtualKey    redrawLine
removeCompleter   replace           resetPromptLine   restoreLine
```

Now we look up more details on the `readLine()` methods, which is what we want to do basic, line-oriented input (a common need in introductory teaching).

```
scala> input.readLine<tab>

def readLine(): String
def readLine(Character): String
def readLine(String): String
def readLine(String, Character): String

scala> val data = input.readLine("Please enter some text: ")
```

```
Please enter some text: Hello, World
data: String = Hello, World
```

You'll probably find it necessary to read through the Scala documentation, but in a number of cases, the behavior is similar to what you've seen in other language APIs. `readLine()` is pretty well known in Java circles. As you can see above, `readLine(String)` gives us what we want: the ability to read input with a prompt of sorts.

val vs. var

- Values (keyboard `val`) are used for immutable storage.
- Variables (keyword `var`) are used for mutable storage.
- You can think of this as the reemergence of `const` but it takes on a more powerful and predictable form in Scala than other languages that preceded it.
- Scala thinking prefers val to var. So do we.
- Interesting

Scripts and Worksheets

Similar to modern scripting languages (e.g. Python and Ruby) and the original shell, you can create a Scala script in a file, e.g. `myscript.scala` and run the script using the `scala` command.

```
$ scala myscript.scala
```

You can also *load* the script within the Scala REPL:

```
$ scala
scala> :load myscript.scala
```

We'll be taking advantage of this a bit more in our discussion about `sbt`, the Scala Build Tool.

Conditional and Functions

Functional → expressions

if expression

```
scala> val a = 25
a: Int = 25

scala> val b = 30
b: Int = 30

scala> val max = if (a > b) a else b
max: Int = 30
```

Contrast with:

```
scala> var max = 0
max: Int = 0

scala> if (a > b)
     |   max = a

scala> if (a > b) {
```

```
     |     max = a
     | } else {
     |     max = b
     | }

scala> max
res4: Int = 30
```

Note: Similar to other agile languages, you can enter compound statements and blocks of code in the REPL. The | is a continuation character to indicate that more input is expected. It's often best to use a text editor once you start entering more complex fragments of code (especially more complicated than what you see here).

Functions are front and center when it comes to Scala programming. Although object-functional, a pure function can be written without the boilerplate associated with OOP. We're proponents of OOP but prefer to introduce functional thinking and *use* of objects prior to creating classes.

```
scala> def square(x : Int) = x * x
square: (x: Int)Int


scala> square(25)
res6: Int = 625

scala> square(25.0)
<console>:12: error: type mismatch;
 found   : Double(25.0)
 required: Int
              square(25.0)
                     ^
```

As expected, the second invocation of `square()` results in an error. Scala performs static type checking in real time. That is, this is *not* a run-time check.

Function literals

Let's build on the `square()` example to see how easy it is to generate the first `n` squares. We'll use this to show how you can use functions as parameters and to sensitize the use of Scala *function literals*, which are used rather idiomatically in Scala programming.

We'll start by generating the first 25 values using a Scala range.

```
scala> val n = 10
n: Int = 10

scala> val first_n = 1 to n
first_n: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

This shows how to map the `square()` function to the range of values.

```
scala> first_n.map(square)
res16: scala.collection.immutable.IndexedSeq[Int] =
   Vector(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

This shows how to map a *function literal* to the range of values.

```
scala> first_n map (n => n * n)
res17: scala.collection.immutable.IndexedSeq[Int] =
   Vector(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

This shows how you can combine a function literal with a previously defined function:

```
scala> first_n map (n => square(n))
res18: scala.collection.immutable.IndexedSeq[Int] =
   Vector(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

You can avoid having to name arguments in function literals using the _ parameter. This syntax is a bit awkward to new programmers (and therefore should be introduced gently in CS1 courses) but allows for concise (and sometimes clearer) expression, especially when used in a disciplined way.

Consider this code that creates the first n even numbers:

```
scala> 1 to 10 map (_ * 2)
res26: scala.collection.immutable.IndexedSeq[Int] =
   Vector(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

You might be tempted to try this by doing the following:

```
scala> 1 to 10 map (_ + _)
<console>:11: error: wrong number of parameters; expected = 1
              1 to 10 map (_ + _)
```

Alas, it doesn't work. Why? Because each occurrence of _ corresponds to an expected parameter. In this case, there would have to be pairs of values. Unfortunately, in the range 1 to 10, each value is an Int.

Default parameters and named parameters

Similar to other agile languages, Scala allows you to specify default parameter values. This is particularly useful, especially when diving into object-oriented programming, but has uses even before then.

Consider this version of `square()`:

```
scala> def square( x : Int = 0) = x * x
square: (x: Int)Int

scala> square()
res28: Int = 0

scala> square(5)
res29: Int = 25
```

Here we are creating a version that has a default value of zero, if the caller doesn't specify `x`. (This is not necessarily intended to be pedagogically interesting but is effective, considering we spent most of our time in this section looking at the `square()` function!)

1.4.3 Early Collection, Arrays and Lists

Scala has some very advanced and comprehensive collections libraries. For CS1 you can focus only on the simplest of these, beginning with Arrays and Lists. In actually makes sense to introduce these before loops (and iteration in general) in Scala for two reasons. First, the collections have many methods that allow you to do standard tasks that would often go in loops. As such, Scala code often doesn't include all that many loops. Second, the for loop in Scala is really a foreach type loop that works with collections so it makes more sense to introduce it after the basic collections.

Making Arrays and Lists

The simplest way to make an Array or a List is with the following syntax.

```
scala> Array(1,2,3,4)
res0: Array[Int] = Array(1, 2, 3, 4)

scala> List(1.3,4.2,3.14)
res1: List[Double] = List(1.3, 4.2, 3.14)
```

Note that the Array and List types take type parameters. These appear in square brackets. It is standard across the Scala language that value arguments appear in parentheses, type arguments appear in square brackets, and curly braces are used to denote code blocks for complex expressions or statements. Also note that as before, Scala's type inference figures out the types for these expressions.

Note: Technically these are making calls to the apply methods of the Array and List companion objects. The details of how that works don't really matter in CS1, but you can find them in the section on object-orientation under CS2. We cover this topic later in our discussion of classes and objects.

You can also made longer arrays using the new syntax that one uses in Java.

```
scala> new Array[Int](100)
res2: Array[Int] = Array(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

Here we need to specify the type and say how big the Array is. Array objects in Scala are implemented as Java arrays and they have the same general properties. Like Java Arrays, they are mutable and they can't dynamically change their size. Note that this approach to making Arrays is not the one typically used in Scala because if the type is something like String you get the same behavior you have in Java and all the values are initialized to null. Later in this section we will see the fill and tabulate methods, which are much less likely to produce a NullPointerException and are generally favored over this style of creating a large Array.

Lists can be built using the `::` operator. We read `::` as `cons`, following the tradition of Scheme (Lisp and ML. `::` is a right associative operator so it takes a value on the left and a List on the right. For example:

```
scala> 1::2::3::4::Nil
res4: List[Int] = List(1, 2, 3, 4)
```

The `Nil` object is a short name for an empty List. You can also use `List()` (the empty list) in place of `Nil`. Note that the List type is a singly-linked, *immutable* list. So prepending as shown above is an efficient operation. Also note that you can't change values without making a new List because it is immutable.

This form of building a List with `::` is probably most beneficial with recursive functions and works very well in situations where you don't know how many values you will be working with.

```
scala> def readNumbers(n:Int):List[Int] = if(n<1) Nil else
     | readInt :: readNumbers(n-1)
readNumbers: (n: Int)List[Int]

scala> readNumbers(5)
1
2
3
4
5
res6: List[Int] = List(1, 2, 3, 4, 5)

scala> def readUntilQuit():List[Int] = {
     | val data = readLine
```

```
     | if(data!="quit") data.toInt :: readUntilQuit() else Nil
     | }
readUntilQuit: ()List[Int]

scala> readUntilQuit()
1
2
3
4
5
quit
res7: List[Int] = List(1, 2, 3, 4, 5)
```

Indexing/Traversing Arrays and Lists

You can get to values in an Array or a List by indexing with parentheses.

```
scala> val arr = Array(6,8,2,4,1)
arr: Array[Int] = Array(6, 8, 2, 4, 1)

scala> val lst = List('g','e','o','r','g','e')
lst: List[Char] = List(g, e, o, r, g, e)

scala> arr(2)
res1: Int = 2

scala> lst(2)
res2: Char = o
```

The use of parentheses instead of square brackets keeps the syntax consistent. (Having parentheses is the next best thing to square brackets as well. It is certainly preferable to having an explicit method call as found in Java to access elements in the array. We also note that the well-known math/science focused programming language, FORTRAN, uses parentheses for subscripting!)

Neither the Array type, nor the List type have any special place in the Scala syntax. They are both handled in the programming language just like any other library code. The compiler does optimize Arrays to Java bytecode arrays, but that is an implementation detail beyond the scope of our workshop.

You can also use the syntax you would expect to set the values of elements in an Array. You can't do this for Lists because they are immutable.

```
scala> arr(2) = 99

scala> arr
res4: Array[Int] = Array(6, 8, 99, 4, 1)
```

The common way to run through a List is using the head and tail methods. The head method returns the first element of the List while tail returns the last remaining after the first element. (Note that these can be called on an Array as well, but tail will be very inefficient as it builds a new, shorter Array.) The use of head and tail is particularly well suited for recursion.

```
scala> def sumList(lst:List[Int]):Int = if(lst.isEmpty) 0 else
     | lst.head+sumList(lst.tail)
sumList: (lst: List[Int])Int

scala> sumList(List(1,2,3,4,5))
res5: Int = 15
```

This same function can be defined using a match and patterns in the following way.

```
scala> def sumList(lst:List[Int]):Int = lst match {
     | case Nil => 0
     | case h::t => h + sumList(t)
     | }
sumList: (lst: List[Int])Int
```

Standard Methods

The collection types in Scala have a large number of methods defined on them. We won't list them here. Instead, you can look in the [ScalaAPI] for the names and details. This is what you get when you use tab completion in the REPL on a List.

```
scala> lst.<tab>
++                    ++:                   +:
/:                    /:\                   :+
::                    :::                   :\
addString             aggregate             andThen
apply                 applyOrElse           asInstanceOf
canEqual              collect               collectFirst
combinations          companion             compose
contains              containsSlice         copyToArray
copyToBuffer          corresponds           count
diff                  distinct              drop
dropRight             dropWhile             endsWith
exists                filter                filterNot
find                  flatMap               flatten
fold                  foldLeft              foldRight
forall                foreach               genericBuilder
groupBy               grouped               hasDefiniteSize
head                  headOption            indexOf
indexOfSlice          indexWhere            indices
init                  inits                 intersect
isDefinedAt           isEmpty               isInstanceOf
isTraversableAgain    iterator              last
lastIndexOf           lastIndexOfSlice      lastIndexWhere
lastOption            length                lengthCompare
lift                  map                   mapConserve
max                   maxBy                 min
minBy                 mkString              nonEmpty
orElse                padTo                 par
partition             patch                 permutations
prefixLength          product               productArity
productElement        productIterator       productPrefix
reduce                reduceLeft            reduceLeftOption
reduceOption          reduceRight           reduceRightOption
removeDuplicates      repr                  reverse
reverseIterator       reverseMap            reverse_:::
runWith               sameElements          scan
scanLeft              scanRight             segmentLength
seq                   size                  slice
sliding               sortBy                sortWith
sorted                span                  splitAt
startsWith            stringPrefix          sum
tail                  tails                 take
takeRight             takeWhile             to
toArray               toBuffer              toIndexedSeq
```

```
toIterable          toIterator          toList
toMap               toSeq               toSet
toStream            toString            toTraversable
toVector            transpose           union
unzip               unzip3              updated
view                withFilter          zip
zipAll              zipWithIndex
```

You might notice that there is a sum method, so the listSum written above is something you never need to write. There are also methods for converting between types such at toList and toArray.

Higher Order Methods

Some of the methods in that list are worth describing instead of leaving it to the reader to explore the API. In particular, the most commonly used higher- order methods, like `map` and `filter`, are used to a significant extent in Scala programming. The `foreach` method is also helpful in a number of situations.

Higher-order methods are methods that take or return functions. In this situation, we are considering methods that take a function as an argument. The `filter` method, for example, takes a function that takes one argument of the type of the collection and returns a Boolean. That function is called on every element of the collection. A new collection is produced that contains only the elements for which the function returned true. To see how this is used, consider the following code where we filter a List of Ints in two different ways.

```
scala> val lst = List(6,4,9,1,2,8,3,7)
lst: List[Int] = List(6, 4, 9, 1, 2, 8, 3, 7)

scala> lst.filter(_ < 6)
res8: List[Int] = List(4, 1, 2, 3)

scala> lst.filter(_ % 2 == 0)
res9: List[Int] = List(6, 4, 2, 8)
```

Both examples use the shorthand underscore syntax for writing the lambda expressions/function literals. They could have been written using the longer syntax as `n => n<6` and `n => n%2 == 0` respectively.

Another higher-order method that is used extensively is the `map` method (one of our favorites). The argument passed into `map` is a function that takes the type of the collection and returns something. The return type could be the same as the input type, but it could be different. The map method returns a new collection of the return type of the function. A first example will use the list of integers from above and make a collection that contains values that are twice the original values.

```
scala> lst.map(_*2)
res10: List[Int] = List(12, 8, 18, 2, 4, 16, 6, 14)
```

To show that the types can be different, this second example shows mapping Strings to their length.

```
scala> val words = "This is a sentence with words".split(" ")
words: Array[String] = Array(This, is, a, sentence, with, words)

scala> words.map(_.length)
res0: Array[Int] = Array(4, 2, 1, 8, 4, 5)
```

The `foreach` method takes a function and simply executes the function on all the elements of the collection. While map and filter return new collections, `foreach` is used for its side effects and returns `Unit`.

```
scala> words.foreach(println)
This
is
```

```
a
sentence
with
words
```

There are other higher-order methods that are very accessible for CS1 students such as `count`, `exists`, and `forall`, all of which take predicate functions on a single argument. There are more complex methods like the `reduce` methods and `fold` methods that can be very helpful, but which are more challenging for CS1 students to understand. We use these occasionally in this workshop but recommmend that they be used sparingly and with an emphasis on clarity (as opposed to conciseness).

Currying and By-Name Arguments

The best ways to create large Arrays and Lists in Scala are with the `fill` and `tabulate` methods. These use some syntax that needs to be introduced. The first element of new syntax is that these methods are *curried*. This means that the arguments are broken across multiple argument lists. So instead of having `f(x,y)` you would have `f(x)(y)`. This is a common feature of functional programming languages and it is done to allow functions to be partially applied. (A partial application means that you apply the function to a subset of the arguments, which results in a new function of the remaining arguments.) The syntax for defining a curried function in Scala is fairly straightforward.

```
scala> def currySum(x:Int)(y:Int) = x+y
currySum: (x: Int)(y: Int)Int

scala> currySum(3)(5)
res2: Int = 8

scala> val plus3 = currySum(3)_
plus3: Int => Int = <function1>

scala> plus3(5)
res3: Int = 8
```

The need for the underscore in the partially applied call removes ambiguity in the Scala syntax. Often functions are curried in Scala to help with type inference or to simplify the syntax for calling them.

The second new element of the syntax is pass-by-name arguments. By default, variables in Scala are passed in a manner similar to Java. The variables are references and the references are passed by-value. With pass-by-name semantics, the code is passed more like a function instead of by-value. Instead of being evaluated at the call point, the code is wrapped in a "thunk" (a term that originated with Algol 60). Every time the variable is used, the code is executed. We specify that a parameter is pass-by-name using a rocket with no arguments.

```
scala> def multiplyThrice(x: => Double) = x*x*x
multiplyThrice: (x: => Double)Double

scala> var i = 5
i: Int = 5

scala> multiplyThrice(i)
res5: Double = 125.0

scala> multiplyThrice({i += 1; i})
res6: Double = 336.0
```

The first invocation of `multiplyThrice` does what one would expect, even though the argument is passed by-name.

We use call-by-name for timing blocks of code in the Monte Carlo Pi example later in this section. See *Monte Carlo Calculation*.

Fill and Tabulate

The fill method, which is defined on the Array object, is...

1.4.4 Recursion for Iteration

```
scala> def sum(n : Int) : Int = if (n <= 0) 0 else n + sum(n-1)
sum: (n: Int)Int

scala> sum(0)
res0: Int = 0

scala> sum(1)
res1: Int = 1

scala> sum(2)
res2: Int = 3

scala> sum(3)
res3: Int = 6

scala> sum(100)
res4: Int = 5050
```

Recursion does take awhile to master, but much of the trouble students have with it in practice is a consequence of side-effect-full thinking. Nevertheless, you can replace this with a more imperative (von Neumann) style. See our loops section.

1.4.5 Some CS1 Friendly Examples

GCD

This is based on a classic algorithm by Euclid and one that is covered in many introductory computer science courses. (See [GCD] for a more detailed discussion of this method and the ways it has been coded over the years, especially in imperative programming languages.)

```
def gcd(a : Int, b : Int) = if (b == 0) a else gcd(b, a % b)
```

The Scala version looks a lot like Euclid's definition!

Factorial

This is the basic example that comes from [MiscExplorationsScala], which also show show to use Scala and functional programming to rework explicitly recursive computations into non-recursive versions using higher-order functional thinking.

This shows how to explicitly define the `fac(n : Int) : Int` function.

```
def fac(n : Int) : Int = if (n <= 0) 1 else n * fac(n - 1)
```

This shows how to define a function *object*. It makes use of the literal syntax we introduced earlier, You read this as "fac is a function that maps Int into Int and binds the name `n` to the `Int` parameter.

```
val fac: Int => Int = n => if (n <= 0) 1 else n * fac(n - 1)
```

In many cases, you can write these functions without explcit recursion, simply by taking advantage of Scala's innate collections:

```
def fac(n : Int) : Int = (1 to n).foldLeft(1)((l,r) => l * r)
```

Or even more concisely (and cryptically?)

```
def fac(n : Int) : Int = (1 to n).foldLeft(1)(_ * _)
```

We'll look more at higher-order thinking shortly. Many who think of functional programming (especially the 1960's and 1970's style) tend to think Lisp, which very much required you to think *recursively* most of the time in practice. Today, the recursive element is still there, but you'll find that the need to use it explicitly is not required and can be replaced with higher-order abstractions. In the end, students still need to know about this technique, if only to understand that certain methods (e.g. folds) often have a recursive nature to them.

In addition to rather outdated notions many tend to have about functional programming when it comes to recursion, the optimization of recursive calls is achieved through a version of tail-recursion elimination (a.k.a. tail call optimization). See [TailCalls] for more information.

Monte Carlo π Calculation

An example that often resonates well with students is the Monte Carlo method, which uses randomness to perform the π computation. In the interests of showing how Scala's approach to functional programming follows the textual description, we will write the steps and show the code (in Scala) to perform each step in an integrated fashion.

In the Monte Carlo method for calculating π, we will randomly generate a given number of darts using a Scala stream. We *fire* the darts into a unit circle, which is bounded by a square, whose dimensions are 2×2 units. The darts that fall within the unit circle satisfy the constraint $x^2 + y^2 \leq 1$.

Let's start by establishing the needed functions. First, here is the now- familiar `square(x:Double)` function, again for reference.

```
def sqr(x: Double) = x * x
```

We use this function to create another function, which tests whether a given dart, specified as an (x, y) coordinate pair (a Scala tuple), falls within the unit circle:

```
val inCircle: ((Double, Double)) => Boolean = { case (x, y) => sqr(x) + sqr(y) <= 1.0 }
```

Let's to a quick sanity check here:

```
scala> inCircle((0, 0))
res1: Boolean = true

scala> inCircle((1, 1))
res2: Boolean = false

scala> inCircle((0.7, 0.7))
res3: Boolean = true

scala> inCircle((0.7, -0.7))
res4: Boolean = true

scala> inCircle((1, -1))
res5: Boolean = false
```

So how do we generate a set of darts and test whether they fall within the circle? We start by using a Scala Stream (more on this in collections).

```
scala> val randomPairs = Stream continually (math.random, math.random)
randomPairs: scala.collection.immutable.Stream[(Double, Double)] =
   Stream((0.45422625790687077,0.1916739269602844), ?)
```

Note: The examples we are showing here, much like typical introductory examples, trade good pedagogy (not bogging down students with too many details) for performance and scalability. If you need to operate on a larger number of darts, you'll want to take advantage of `StreamIterator`, not `Stream`. A StreamIterator can be obtained from a Stream by reworking the code above as follows:

```
scala> val randomPairs = Stream continually (math.random, math.random) iterator
randomPairs: scala.collection.immutable.Stream[(Double, Double)] =
   Stream((0.45422625790687077,0.1916739269602844), ?)
```

Notice that the `iterator()` method is being invoked. This can be taught after students have learned more about Scala collections.

What you see here is the first item of the stream being displayed. The "?" indicates that there are more values (an infinite number, in theory) which will be obtained on demand. This principle is an advanced one but one that is teachable and can be understood in greater detail later.

So how do we get a finite number of darts? This is where a number of famous methods from functional programming come to play. One is known as `take(n)`, which can take however many we want. Let's use this to show how to print the first 5 random coordinate pairs.

```
scala> randomPairs.take(5)
res7: scala.collection.immutable.Stream[(Double, Double)] =
   Stream((0.45422625790687077,0.1916739269602844), ?)

scala> randomPairs.take(5).foreach(println)
(0.45422625790687077,0.1916739269602844)
(0.9252028282996272,0.5638265909110913)
(0.5588908846857542,0.21516929857230815)
(0.5842149396390998,0.7226255374753748)
(0.8454163994561401,0.6805038035803781)
```

So what is going on here? We're taking the first 5 coordinate pairs in the stream and applying the `println()` function to each item in the stream (that is, each coordinate pair). While there are some aspects of this that are advanced, there are some aspects that are *vastly simpler* than their equivalent in tradition imperative object-oriented languages (e.g. C++, Java, C#). For example, we rely on the notion of a tuple (a pair of Double values), which usually requires the premature exploration of a Pair class in other languages. Before long, you need to learn a lot of arcane type theory to understand Pair, whereas in Scala, the type information is *inferred*.

Looking more closely:

```
scala> (math.random, math.random)
res10: (Double, Double) = (0.20679803333001656,0.91233235776938)
```

This generates a random pair, and it even *looks* like a random pair from mathematics. Of course, it's also type-safe!

So we're now near the point where we can put all of the pieces together. We have a function to determine whether a randomly generated coordinate pair falls within the unit circle. Let's compute π.

```
scala> val n = 1000000
n: Int = 1000000

scala> val darts = randomPairs.take(n)
darts: scala.collection.immutable.Stream[(Double, Double)] =
   Stream((0.45422625790687077,0.1916739269602844), ?)
```

```
scala> val dartsInCircle = darts.count(inCircle)
dartsInCircle: Int = 784894

scala> val totalDarts = darts.length
totalDarts: Int = 1000000

scala> val area = 4.0 * dartsInCircle / totalDarts
area: Double = 3.139576
```

This is a good time to introduce the *dot-less* syntax, which is often associated with object-oriented programming but actually precedes these languages (C *struct* et al).

You can also write the above code (where you see dots) as follows:

```
scala> val darts = randomPairs take n
darts: scala.collection.immutable.Stream[(Double, Double)] =
   Stream((0.45422625790687077,0.1916739269602844), ?)

scala> val dartsInCircle = darts count inCircle
dartsInCircle: Int = 784894

scala> val totalDarts = darts length
totalDarts: Int = 1000000

scala> val area = 4.0 * dartsInCircle / totalDarts
area: Double = 3.139576
```

At some point, you realize that you want to enter the code into a text editor that can be loaded into the Scala interpreter (as opposed to being entered interactively).

The actual code for this can be found at https://bitbucket.org/loyolachicagocs_plsystems/numerical-explorations-scala. You can pull up the Scala worksheet from `MonteCarloPiStreamIteratorChunkFree.sc` (by drilling into Source).

Here's the final version of our function to calculate π.

We also provide a simple *timing* function, that allows us to time a *block* of Scala statements. We'll not present it in detail now, but this function could be given to your students with the hope of sensitizing them to the notion of *performance*. It also shows the tremendous power available in Scala to work with a block of Scala code as an object (which can produce a value).

We're going to use this to show how to measure the execution time of our π calculation by varying the problem size. Here is the fragment of code that tries different problem sizes up to $n = 10^k$, where $k = \lfloor log_{10}(Int.MaxValue) \rfloor$ ($k = \lfloor log_{10}(2147483647) \rfloor = 9$)

Note: You might wonder why we are limited to `Int` in this version as opposed to a 64-bit `Int`. It is a perfectly valid idea to ponder, especially in the modern era. Scala collections do not support operations like `take()` and `drop()` with 64-bit Int values. The explanation for this is a bit beyond the scope here, but we have worked out 64-bit versions where we perform the Monte Carlo π calculation in chunks. You can visit our repository for these versions, which are a bit complex compared to the Int version. We hope future versions of Scala will evolve beyond 32-bit thinking but don't see this as a show stopper for introductory teaching. (We also hope the friendly competition between F# and Scala, where F# supports Int64, will eventually make its way to Scala.

Here is the output (some output has been deleted for conciseness).

```
scala> :load MonteCarloPiStreamIteratorChunkFree.sc
Loading MonteCarloPiStreamIteratorChunkFree.sc...
sqr: (x: Double)Double
```

```
inCircle: ((Double, Double)) => Boolean = <function1>
randomPairs: Iterator[(Double, Double)] = non-empty iterator
n: Int = 1000000
darts: Iterator[(Double, Double)] = non-empty iterator
dartsInCircle: Int = 785719
totalDarts: Int = 0
area: Double = Infinity
longDartsInCircle: (numDarts: Int)Long
monteCarloCircleArea: (numDarts: Int)Double
time: [R](block: => R)R
powers: scala.collection.immutable.Range.Inclusive =
   Range(1, 2, 3, 4, 5, 6, 7, 8, 9)
sizes: scala.collection.immutable.IndexedSeq[Int] =
   Vector(10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000, 1000000000)
problemSizes: scala.collection.immutable.IndexedSeq[Int] =
   Vector(1000000, 10000000, 100000000, 1000000000)
Trying these probem sizes
1000000
10000000
100000000
1000000000
numDarts: 1000000
Elapsed time: 0.149393s
The area is 3.141644
numDarts: 10000000
Elapsed time: 1.110151s
The area is 3.1406688
numDarts: 100000000
Elapsed time: 10.744608s
The area is 3.141555
numDarts: 1000000000
```

We will discuss performance and timing issues again when speaking to parallel computing in *Basic Parallelism using Par*.

1.4.6 More Scala

While Loop

- While loop
- Not an expression

In Scala, you can do imperative style loops and interactive loops:

Consider this interactive session to compute well-known example for sum

```
scala> def sum(n : Int) : Int = {
     |   var i = 0
     |   var sum = 0
     |   while (i <= n) {
     |     sum = sum + n
     |     i = i + 1
     |   }
     |   sum
     | }
sum: (n: Int)Int
```

```
scala> sum(100)
res5: Int = 10100

scala> sum(0)
res6: Int = 0
```

Much is possible without iteration with the added advantage of being recursive and side-effect free underneath the hood.

> scala> def sum(n : Int) = (1 to n).sum sum: (n: Int)Int
>
> scala> sum(0) res8: Int = 0
>
> scala> sum(100) res9: Int = 5050

What about interactive loops?

```
scala> val reader = new ConsoleReader
reader: scala.tools.jline.console.ConsoleReader =
   scala.tools.jline.console.ConsoleReader@26075b18

scala> var response = 0
response: Int = 0

scala> while (response < 0 || response > 100) {
     |    response = reader.readLine("Enter a number >= 0  and <= 100? ").toInt
     | }

scala> var response = -1
response: Int = -1

scala> while (response < 0 || response > 100) {
     |    response = reader.readLine("Enter a number >= 0  and <= 100? ").toInt
     | }
Enter a number >= 0  and <= 100? -5
Enter a number >= 0  and <= 100? 105
Enter a number >= 0  and <= 100? 100
scala> ...
```

It is interesting to think about whether we can turn an interactive while loop into one without side effects. There are so many bad things that happen to us as CS1 educators when we work with interactive loops:

- Improper initialization of the `response` variable.
- Improper setting of `response` upon termination.
- etc.

This is my early attempt to figure out how to have a side-effect free interactive while loop. Basic idea:

- Use `Stream.continually` to get a continuous stream of input lines read.
- Because `Stream` is lazy only in the tail, assume a blank line as the first value of the stream (which is safely ignored). I'd like to figure out the right way to eliminate this without introducing complexity.
- Use `takeWhile` to accept input until an appropriate terminating condition (the word "no" is entered)
- Convert each line to integer, if possible. We simplify exception management by applying a `Try` wrapper to each attempted conversion. `Try` is an important Scala idiom for representing a computation that either succeeds with a result value or fails with an exception.

- Use `flatMap` to get only the input values where the conversion succeeded. In the process of applying `flatMap`, however, we need to convert each `Try` value to an `Option` value. `Option` is very similar to `Try` but simpler in that it represents all failures as `None`, which `flatMap` discards from the resulting list.

```
scala> val s1 = "" #:: Stream.continually(reader.readLine("Prompt: "))
s1: scala.collection.immutable.Stream[String] = Stream(, ?)

scala> val s = s1.takeWhile(_ != "no")
s: scala.collection.immutable.Stream[String] = Stream(, ?)

scala> val l = s.toList
Prompt: 25
Prompt: 35
Prompt: 55
Prompt:
Prompt: s
Prompt: no
l: List[String] = List("", 25, 35, 55, "", s)

scala> import scala.util.Try
import scala.util.Try

scala> def toInteger(s: String) = Try(s.toInt)
toInteger: (s: String)scala.util.Try[Int]

scala> l.map(toInteger)
res3: List[scala.util.Try[Int]] = List(Failure(java.lang.NumberFormatException: For input string: ""

scala> l.map(t => toInteger(t).toOption)
res6: List[Option[Int]] = List(None, Some(25), Some(35), Some(55), None, None)

scala> l.flatMap(t => toInteger(t).toOption)
res7: List[Int] = List(25, 35, 55)

scala> l.flatMap(t => toInteger(t).toOption).sum
res8: Int = 115
```

for loop

Similar to the while loop, a for loop exists for *imperative* style programming, often when there is a need to do something where a side-effect is needed.

```
scala> var sum = 0
sum: Int = 0

scala> var n = 100
n: Int = 100

scala> for (i <- 1 to n)
     | {
     |     sum = sum + i
     | }

scala> println(s"The sum is $sum")
The sum is 5050
```

for comprehension

A for comprehension is designed for where you want a more functional style. That is, there is no intention of having side effects, and it is likely that you want to use the result of the comprehension in another computation.

Let's look at something a bit more interesting: Getting the first `n` squares:

```
scala> val n = 10
n: Int = 10

scala> val first_n_squares = for (i <- 1 to n) yield { i * i }
first_n_squares: scala.collection.immutable.IndexedSeq[Int] =
   Vector(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

You could wrap this up nicely in a Scala function as follows:

```
scala> def squares(n : Int) = for (i <- 1 to n) yield { i * i }
squares: (n: Int)scala.collection.immutable.IndexedSeq[Int]

scala> squares(10)
res7: scala.collection.immutable.IndexedSeq[Int] =
   Vector(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

options and failure sans exceptions

Per the Scala documentation: An option represents optional values. Instances of Option are either an instance of scala.Some or the object None.

The Scala Option type is useful for dealing with the notion of success and failure (but is not limited to supporting this concept). In the interests of keeping this discussion focused on being CS1-friendly, an Option is always connected to an underlying type. For example Option[Int] means that you either get the Int, or nothing (None).

```
scala> val i = Some(3)
i: Some[Int] = Some(3)

scala> val j = None
j: None.type = None
```

`Some` and `None` are both *case classes* that extend `Option[A]` (a generic class that allows you to use any time). While knowing all of the details requires a mastery of object-oriented programming (and perhaps more), the introduction of this idea is remarkably straightforward and one that can help students to write better code from the beginning. In CS1 courses, we tend to spend a lot of time handling special cases, and options give us a framework for dealing with missing information, etc.

As you can see from the above, `i` is defined as some value, in this case 3. It's a wrapped value, of course, that must be unbundled later. For example, here is an attempt to use the value `i` (incorrectly):

```
scala> i + 5
<console>:9: error: type mismatch;
 found   : Int(5)
 required: String
              i + 5
                ^
```

So how do we make use of `i` and `j`? A method associated with options, `getOrElse` allows us to get the *option*, if set to *some* value. Rather delightfully, we can set a value to be used, if the option was not set previously.

```
scala> i getOrElse(-1)
res13: Int = 3

scala> j getOrElse(-1)
res14: Int = -1
```

So now if we want to use this value in a computation, we can just do this:

```
scala> i.getOrElse(-1) + j.getOrElse(-1)
res17: Int = 2
```

Options have their uses throughout Scala, notably its libraries. For example, maps (a.k.a. associative structures/arrays) use option to return the result of getting a key from the map:

```
scala> val map = scala.collection.mutable.HashMap.empty[String, Int]
map: scala.collection.mutable.HashMap[String,Int] = Map()

scala> map += ("scala" -> 10)
res4: map.type = Map(scala -> 10)

scala> map += ("java" -> 20)
res5: map.type = Map(scala -> 10, java -> 20)

scala> map += ("C#" -> 15)
res6: map.type = Map(scala -> 10, java -> 20, C# -> 15)

scala> map += ("F#" -> 25)
res7: map.type = Map(scala -> 10, F# -> 25, java -> 20, C# -> 15)

scala> map += ("scala" -> 22)
res8: map.type = Map(scala -> 22, F# -> 25, java -> 20, C# -> 15)
```

Here's an attempt to get the case-sensitive and case-insensitive versions of key, F#, from the map:

```
scala> map.get("F#")
res9: Option[Int] = Some(25)

scala> map.get("f#")
res10: Option[Int] = None
```

Seasoned Java programmers know that an entry not found in the map will result in the *null* value being returned. This differs from Scala, because the value gotten from the map must be *tested* before attempting to use it in any way.

In Scala, because `None` and `Some(25)` are both options, you can use `getOrElse` to obtain the options value (irrespective of whether the value is null, or not set) without writing an (unwanted) if-then-else statement, which results in bloat (in most programming languages).

```
scala> val entry = map.get("F#").getOrElse(-1)
entry: Int = 25

scala> println(s"The entry for F# is $entry")
The entry for F# is 25
```

It's often the case that we have *default* values associated with failure to accomplish a certain task. The Option idiom is an attempt to standardize this for core data structures and (as we'll see) other situations (e.g. working with complex for comprehensions). In the case of maps, an entry not found would usually default to 0 or -1 (a convention that dates back to the earliest days of C), which is preferable to throwing exceptions for no good reason (not to mention our general dislike of prematurely covering exceptions as a programming technique in CS1 in particular.)

yield

Yield is used to take a collection and produce a new one with mapped values.

Let's produce a List[Int] of squares from a List[Int] of the first three integers:

```
scala> val l = List(1, 2, 3)
l: List[Int] = List(1, 2, 3)

scala> l
res10: List[Int] = List(1, 2, 3)

scala> for (i <- l) yield { i * i }
res11: List[Int] = List(1, 4, 9)
```

Let's try this with an Array[Int]:

```
scala> val a = Array(1, 2, 3)
a: Array[Int] = Array(1, 2, 3)

scala> for (i <- a) yield i * i
res16: Array[Int] = Array(1, 4, 9)
```

For the most part, when iterating over values and using yield, you will always get back the same type, or another type that makes sense.

In these basic examples, the above can also be written as follows (without the for):

```
scala> l map (i => i * i)
res17: List[Int] = List(1, 4, 9)

scala> a map (i => i * i)
res18: Array[Int] = Array(1, 4, 9)
```

Ranges

Ranges should be familiar to you if you've worked with other agile scripting languages, e.g. Python.

```
scala> Range(1, 5)
res20: scala.collection.immutable.Range = Range(1, 2, 3, 4)
```

This gives a range of values from 1 to 5 but stopping at the last value before 5. The increment is +1.

```
scala> Range(1, 9, 2)
res22: scala.collection.immutable.Range = Range(1, 3, 5, 7)
```

You can also work backwards:

```
scala> Range(9, 0, -2)
res24: scala.collection.immutable.Range = Range(9, 7, 5, 3, 1)
```

Multiple generators

If guards

Variables

Patterns

1.4.7 Files

- Can use Scanner
- scala.io.Source
- Scala Iterator[Char]
- getLines : Iterator[String]
- Use with higher-order methods
- Write with PrintWriter
- Introduce APIs?

1.4.8 Classes and Objects

Classes

Here is the familiar Point class. It's often shown where the (x, y) coordinate pair are Int (even in the Scala documentation) but is even more interesting with Double. This is an elaborated version that includes elements appropriate mostly to CS1 and some that are best covered in CS2 and beyond.

```
class Point(initial_x: Double, initial_y: Double) {
   var x: Double = initial_x
   var y: Double = initial_y

   def move(dx: Double, dy: Double) {
      x = x + dx
      y = y + dy
   }

   def distanceToOrigin() : Double = {
      math.sqrt(x * x + y * y)
   }

   def distanceTo(p : Point) = {
      math.sqrt( (p.x - x) * (p.x - x) + (p.y - y) * (p.y - y) )
   }

   def add(p : Point) = {
      x = x + p.x
      y = y + p.y
      this
   }

   def +(p : Point) = {
      add(p)
   }
```

```
27
28     override def toString(): String = s"($x, $y)";
29 }
```

What does this class Point show?

- how to create a simple Scala class. Notice the complete lack of keyboards and public/private/static that tend to confuse students!
- Scala brings back *disciplined* operator overloading. We'd probably not use this in CS1, but it is entirely appropriate for CS2.
- Shows how to refer to the object (familiarly) with `this`. Again, the methods relying on this might be more appropriate for CS2 or even later courses that dive into OOP's complexity.
- Constructors? The Scala class definition itself makes it clear how one constructs an instance. Just like a function definition in general, there can be default values. It is liberating not to have constructors (especially too many of them), especially when trying to introduce a topic. (This will become readily apparent when we look at *case classes*, which provide a mechanism for more data-centric OO abstraction.)
- Convert an instance of class Point to a String representation using `ToString(). ToString()` can be a valuable pedagogical tool, done right (as observed in languages like Python). Scala 2.10 gives us the ability to do type-safe string interpolation by substituting the value of variables (their String representation) into a String template. In Scala, prefixing a string literal with `s` will give you a string where any variables (beginning with `$`) are substituted. Here, it allows us to get a beautiful representation of a point as an (x, y) pair with virtually no effort or complexity!

Let's take a look at how the `Point` class is used:

```
1  val p = new Point(2, 3)
2  val q = new Point(-2, 3)
3
4  println(s"Two points $p and $q")
5
6  val distanceToOrigin = p.distanceToOrigin()
7
8  println(s"distance from p to origin = $distanceToOrigin")
9
10 val dpq = p.distanceTo(q)
11 val dqp = q.distanceTo(p)
12
13 println(s"d(p,q) = $dpq")
14 println(s"d(q,p) = $dqp")
15
16 val pointSum = p.add(q)
17 val pointSumOp = p + q
18
19 println(s"p.add(q) = $pointSum; p + q = $pointSumOp")
```

This results in the following output.

```
scala> :load point.sc
:load point.sc
Loading point.sc...
defined class Point
p: Point = (2.0, 3.0)
q: Point = (-2.0, 3.0)
Two points (2.0, 3.0) and (-2.0, 3.0)
distanceToOrigin: Double = 3.605551275463989
distance from p to origin = 3.605551275463989
```

```
dpq: Double = 4.0
dqp: Double = 4.0
d(p,q) = 4.0
d(q,p) = 4.0
pointSum: Point = (0.0, 6.0)
pointSumOp: Point = (-2.0, 9.0)
p.add(q) = (-2.0, 9.0); p + q = (-2.0, 9.0)
```

A Look at Singleton Objects

At some point, relying on the interpreter for trying out the Point class (as we have been doing until now) grows a bit tedious. Furthermore, sometimes you want to have a complete, functioning program that includes not just the class Point but also a driver program–often found in a `main()` method–that allows you to run it from the command line.

Scala's answer to `main()` (largely a vestige of C-based languages) is to support *singleton objects*, which we rely upon in some more advanced examples. While we consider the singleton pattern to be a bit overrated and overused (e.g. `Runtime.getRuntime()` and many others like it in Java's API), the singleton object as found here is completely decoupled from any class and allows you to create an *environment* so to speak with its own namespace but without the burden of a full class definition.

In this example, we create a singleton object to act as our `main()` driver.

```
object PointDemo {
   def apply() {
      val p = new Point(2, 3)
      val q = new Point(-2, 3)

      println(s"Two points $p and $q")

      val distanceToOrigin = p.distanceToOrigin()

      println(s"distance from p to origin = $distanceToOrigin")

      val dpq = p.distanceTo(q)
      val dqp = q.distanceTo(p)

      println(s"d(p,q) = $dpq")
      println(s"d(q,p) = $dqp")

      val pointSum = p.add(q)
      val pointSumOp = p + q

      println(s"p.add(q) = $pointSum; p + q = $pointSumOp")
   }
}

PointDemo()
```

Similar to a class definition, you *can* have methods. Notably absent, however, are any parameters to the object name. To create an entry point that allows the object to be used like any other Scala function, we define an `apply()` method, which may or may not have parameters. In our case, we just want to be able to run the same demo code we produced previously and then invoke the PointDemo as a function, e.g. `PointDemo()`. This could be further wrapped with some logic to handle command-line arguments, etc. More on that towards the end.

1.4.9 Useful REPL functionality

The Scala REPL supports a number of commands that can be greatly helpful for working interactively. We've relied on many of these in the preparation of this tutorial but will focus on the highlights, especially for use in CS1 teaching.

```
scala> :help
All commands can be abbreviated, e.g. :he instead of :help.
Those marked with a * have more detailed help, e.g. :help imports.

:cp <path>                  add a jar or directory to the classpath
:help [command]             print this summary or command-specific help
:history [num]              show the history (optional num is commands to show)
:h? <string>                search the history
:imports [name name ...]    show import history, identifying sources of names
:implicits [-v]             show the implicits in scope
:javap <path|class>         disassemble a file or class name
:load <path>                load and interpret a Scala file
:paste                      enter paste mode: all input up to ctrl-D compiled together
:power                      enable power user mode
:quit                       exit the interpreter
:replay                     reset execution and replay all previous commands
:reset                      reset the repl to its initial state, forgetting all session entries
:sh <command line>          run a shell command (result is implicitly => List[String])
:silent                     disable/enable automatic printing of results
:type [-v] <expr>           display the type of an expression without evaluating it
:warnings                   show the suppressed warnings from the most recent line which had any
```

paste

When writing longer definitions in the REPL, it can be tricky. Having paste mode allows you to take some code you have (perhaps from an editor where you are typing a Scala program) and copy/paste into the Scala session.

This shows an example of entering a slightly more verbose than needed definition of the `square()` function (presented earlier in this section):

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

def square(x : Int) : Int = {
   x * x
}

// Exiting paste mode, now interpreting.

square: (x: Int)Int
```

Notice that you don't see the continuation characters when entering multiple lines of text.

load

Many programmers coming to Scala find it a bit frustrating at first that some things (like interactive scripts, found in languages like Python) don't work quite the same way. More often than not, the real issue is whether there is an easy way to load a script into the REPL–as opposed to having to run it on the command line (which is also possible but not the focus of this section). It is a matter of loading the filename, which may be an absolute or relative path.

```
scala> :load myscript.scala
```

history

History should be familiar to anyone who has used modern Unix shells. Even if you haven't, you've probably used the history buffer, which allows you to use the up/down arrows or emacs/vi commands (^p, ^n, j, k) to access previous and next commands in the history buffer.

Scala also allows you to do ^r to perform a regex search for text within a previous command. We rely on this heavily in our work, especially in this section, where it was necessary to look up previous attempts within the REPL so they could be pasted into the notes!

In this example, I used ^r to search for the substring "val" so I could find a previous value definition in my REPL session:

```
(reverse-i-search)`val': val entry = map.get("F#").getOrElse(-1)
```

When you type ^r, you'll be given the "(reverse-i-search)" prompt to perform a search. While the full functionality of regex is provided, the nominal use is to type a few characters of something you probably remember (at least partially). More often than not, I am looking for previous "val" or "def" (functions).

1.4.10 Additional Resources

1.5 CS2

> **Warning:** This section is not ready yet. Working on it!

1.5.1 Collections

- Just for Scala
- Doesn't make sense before loops in most languages.
- One mutable, one immutable
- Many standard methods
- Many higher-order methods
- Syntax
- Use () for indexing
- List also have ML style operations
- Creation, pass-by-name

1.5.2 Case Classes

- Immutable struct in simplest usage
- Simple syntax for grouping data
- Works as a pattern

- Copy method

1.5.3 GUIs

- scala.swing wraps javax.swing
- Cleaner beginner syntax
- No explicit inheritance
- Reactions use partial functions
- Drawbacks: Currently no JTree, Tables complex, Button syntax uses companion object
- Full Java2D
- Really using Java
- Override paint method
- Events for animations
- Keyboard, Mouse, Timer

Simple GUI Example

This example shows a simple GUI where there is no inheritance. Clicking the button increments the number in the text field. If the user has changed it so that it is not an integer, it is set back to 0.

Those familiar with Java will notice a lot of similarities. This is because the scala.swing library is a wrapper around Java's Swing library. The ideas are similar, but the way in which you interact with them has been changed to match the Scala style.

```
import swing._

val frame = new MainFrame
val field = new TextField("0")
val button = Button("Increment") {
    try {
        field.text = (field.text.toInt+1).toString
    } catch {
    case ex:
        NumberFormatException => field.text = "0"
    }
}

val bp = new BorderPanel
import BorderPanel.Position._
bp.layout += field -> North
bp.layout += button -> Center
frame.contents = bp
frame.centerOnScreen
frame.open
```

Simple GUI Example using scala.util.Try

Scala supports simplified exception handling through its `scala.util.Try` wrapper type. This is an important Scala idiom for representing a computation that either succeeds with a result value or fails with an exception.

For example, say you want to validate and convert a text field in your UI from string to integer. You could write this simple conversion function to do so:

```
scala> def toInteger(s: String) = scala.util.Try(s.toInt)
toInteger: (s: String)scala.util.Try[Int]

res0: scala.util.Try[Int] = Failure(java.lang.NumberFormatException: For input string: "blah")

scala> toInteger("35")
res1: scala.util.Try[Int] = Success(35)
```

Then you can use `getOrElse` to process the enclosed value and, if the `Try` value represents failure, return the given default value (as you see above when we tried to validate the string "blah").

```
scala> toInteger("35").getOrElse(-1)
res2: Int = 35

scala> toInteger("blah").getOrElse(-1)
res3: Int = -1
```

It's clear that being able to validate input efficiently is something that excites us. It certainly makes UI development more reliable and resilient to failures. (We've had more than our share of fun chasing down validation bugs in web and mobile app development. Most of the time it is caused by unnecessarily complex validation logic.)

You can see how this plays out in a slightly reworked version of the code:

```
import swing._
import scala.util.Try

val frame = new MainFrame
val field = new TextField("0")

val button = Button("Increment") {
  val attempt = Try(field.text.toInt)
  field.text = (attempt.getOrElse(-1)+1).toString
}

val bp = new BorderPanel
import BorderPanel.Position._
bp.layout += field -> North
bp.layout += button -> Center
frame.contents = bp
frame.centerOnScreen
frame.open
```

Simple Paint Example

This example shows how you can override the paint method to make a custom drawing. It also shows interactions with the mouse.

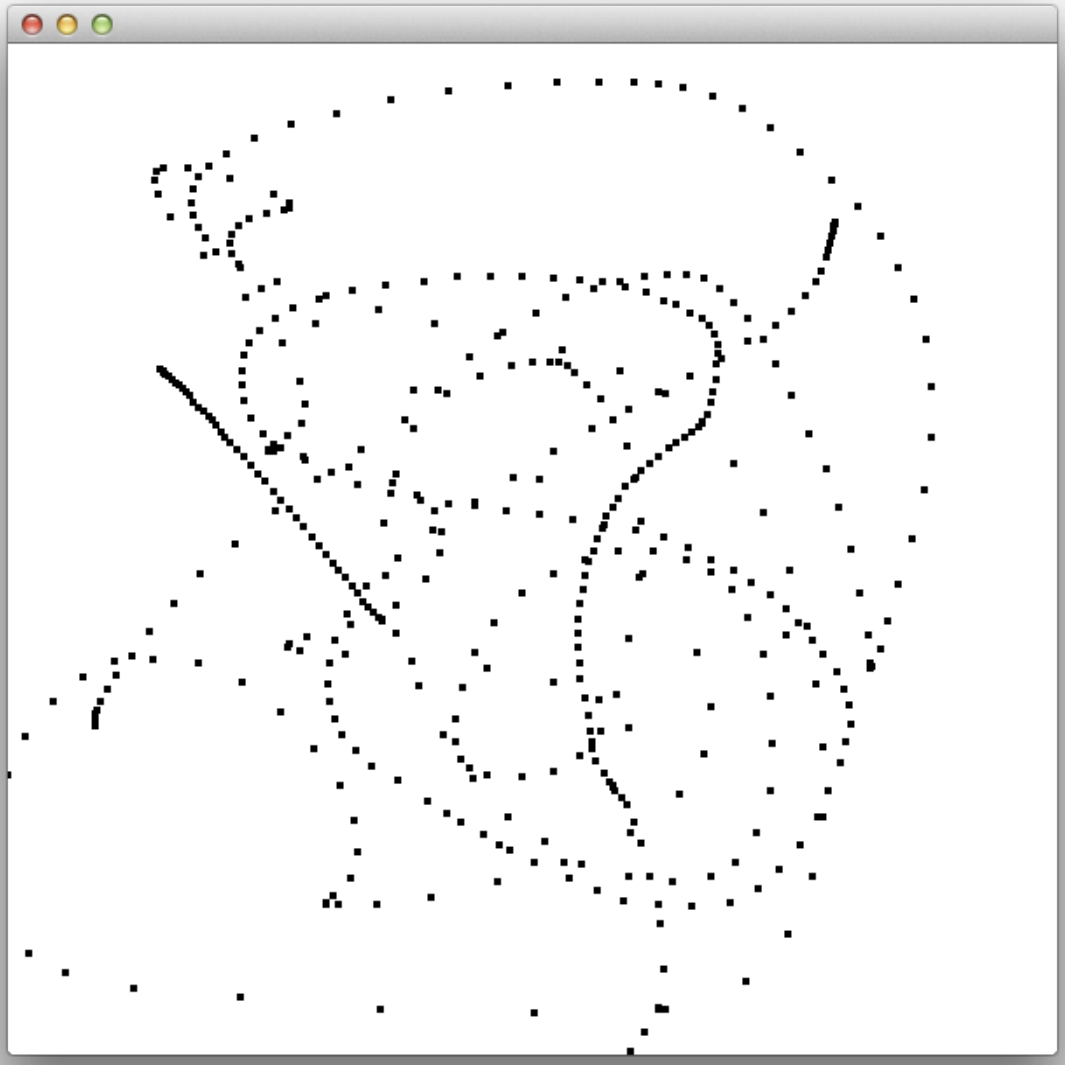
```
import swing._
import event._
import java.awt.{Color,Shape}
import java.awt.geom._

var dots = List.empty[Shape]

val panel = new Panel {
```

```
9     override def paint(g:Graphics2D) {
10      g.setPaint(Color.white)
11      g.fillRect(0,0,size.width,size.height)
12      g.setPaint(Color.black)
13      for(s <- dots) g.fill(s)
14    }
15    listenTo(mouse.clicks,mouse.moves)
16    reactions += {
17    case mc: MouseClicked =>
18      dots ::= new Ellipse2D.Double(mc.point.x-2,mc.point.y-2,5,5)
19      repaint
20    case mc: MouseDragged =>
21      dots ::= new Ellipse2D.Double(mc.point.x-2,mc.point.y-2,5,5)
22      repaint
23    }
24  }
25
26  val frame = new MainFrame {
27    contents = panel
28    size = new Dimension(600,600)
29    centerOnScreen
30  }
31
32  frame.open
```

Here's what the output looks like when you drag the mouse quasi-randomly on the blank canvas that first comes appears. (Your output may vary!)



A More Complex GUI Example

This is a large GUI example. There are two lists with a text fields and some buttons. The first list is populated by the text field and the buttons move things between lists or remove them from the second list.

The populating from the text field demonstrates how you listen to GUI elements and react to them. The behavior of the lists shows how collection methods can play a role in GUIs.

```
1 import swing._
2 import event._
3
4 val list1 = new ListView[String]()
5 val list2 = new ListView[String]()
6 val buttons = new FlowPanel {
```

```
    contents += Button("<-") {
        list1.listData ++= list2.selection.items
        list2.listData = list2.listData.diff(list2.selection.items)
    }
    contents += Button("->") {
        list2.listData ++= list1.selection.items
        list1.listData = list1.listData.diff(list1.selection.items)
    }
    contents += Button("Remove") {
        list2.listData = list2.listData.diff(list2.selection.items)
    }
}
val field = new TextField() {
    listenTo(this)
    reactions += {
    case ed:
        EditDone =>
list1.listData :
        += text
        text = ""
    }
}

val frame = new MainFrame {
    contents = new BorderPanel {
        import BorderPanel.Position._
        layout += field -> North
        layout += new ScrollPane(list1) -> West
        layout += new ScrollPane(list2) -> East
        layout += buttons -> Center
    }
    size = new Dimension(600,500)
    centerOnScreen
}

frame.open
```

Asteroids

This program is a little implementation of asteroids. It shows keyboard events and the use of case classes to group data together.

We start by importing varius dependencies. This shows how you can take advantage of existing Java libraries.

Case classes are used to maintain information about key elements of the game, notably the asteroids and bullets. Although you see the word class here, we're primarily using class to aggregate the data (think about C *struct* but even nicer). These are used to maintain two typesafe lists of asteroids and bullets, respectively, with type `List[Asteroid]` and `List[Bullet]`.

```
import swing._
import event._
import java.awt.{Color,Shape}
import java.awt.geom._
import javax.swing.Timer

case class Asteroid(x:Double,y:Double,vx:Double,vy:Double,size:Double)
case class Bullet(x:Double,y:Double,vx:Double,vy:Double,age:Int)
```

```
9
10 val windowSize = 600
11 val shipSize = 6
12
13 var asteroids = List.fill(5){
14     val theta = math.random*math.Pi*2
15     Asteroid(windowSize/2+math.cos(theta)*windowSize/4,
16                     windowSize/2+math.sin(theta)*windowSize/4,
17                       math.random-0.5,math.random-0.5,50)
18 }
19 var bullets = List[Bullet]()
20 var shipX = windowSize/2.0
21 var shipY = windowSize/2.0
22 var heading = 0.0
23 var shipVx = 0.0
24 var shipVy = 0.0
25 var leftDown = false
26 var rightDown = false
```

The `wrap()` method does what you might be thinking it does. It takes an x or y coordinate (even though you only see `x` in the parameter name) of a given asteroid and ensures it falls within the bounds of the window. Note that this references an external "global" *value* `windowSize`. Because this value is immutable, there is no risk of a side effect. (This could be called the revival of the `const` from C/C++ and Pascal but in a more modern formulation.)

```
1 def wrap(x:Double):Double = {
2     var nx = x
3     while(nx < 0) nx += windowSize
4     while(nx > windowSize) nx -= windowSize
5     nx
6 }
```

The definition of the panel is where the actual drawing (and redrawing) of the game takes place. It also shows how to clearly separate the drawing from the reactions to events of interest (keyboard and mouse). Notably, we can handle these events without having to use *classes*. This allows us to stay focused on design principles instead of the vagaries of event objects and interfaces (even though these details are still present, being able to match the event's *type* allows us to avoid premature complexity from a student perspective.)

```
1 val panel = new Panel {
2     override def paint(g:Graphics2D) {
3         g.setPaint(Color.black)
4         g.fillRect(0,0,size.width,size.height)
5         g.setPaint(Color.lightGray)
6         for(a <- asteroids) g.fill(new Ellipse2D.Double(a.x-a.size/2,a.y-a.size/2,a.size,a.size))
7         g.setPaint(Color.red)
8         for(b <- bullets) g.fill(new Rectangle2D.Double(b.x,b.y,2,2))
9         g.setPaint(Color.blue)
10        g.fill(new Ellipse2D.Double(shipX-shipSize,shipY-shipSize,shipSize*2,shipSize*2))
11        g.setPaint(Color.green)
12        g.draw(new Ellipse2D.Double(shipX-shipSize,shipY-shipSize,shipSize*2,shipSize*2))
13        g.fill(new Ellipse2D.Double(shipX+(shipSize+2)*math.cos(heading)-2,shipY+(shipSize+2)*math.si
14    }
15    listenTo(keys,mouse.clicks)
16    reactions += {
17        case kp:KeyPressed =>
18            if(kp.key == Key.Left) leftDown = true
19            else if(kp.key == Key.Right) rightDown = true
20            else if(kp.key == Key.Up) {
21                shipVx += math.cos(heading)*0.2
```

```
                shipVy += math.sin(heading)*0.2
            } else if(kp.key == Key.Down) {
                shipVx -= math.cos(heading)*0.2
                shipVy -= math.sin(heading)*0.2
            } else if(kp.key == Key.Space) {
                bullets ::= Bullet(shipX+(shipSize+2)*math.cos(heading),shipY+(shipSize+2)*math.sin(h
            }
        case kp:KeyReleased =>
            if(kp.key == Key.Left) leftDown = false
            else if(kp.key == Key.Right) rightDown = false
        case me:MouseEntered => requestFocus
    }
    preferredSize = new Dimension(windowSize,windowSize)
}
```

A *timer* is particularly useful in game design, where you want to have self-updating without user interaction. In the case of this game, whether or not the user is doing anything, asteroids continue moving, subject to their velocities. Same for bullets. There is also logic to determine collisions and whether the ship is destroyed (which ends the game).

```
val timer:Timer = new Timer(10,Swing.ActionListener(e => {
    if(leftDown) heading -= math.Pi/40
    if(rightDown) heading += math.Pi/40
    asteroids = asteroids.map(a => {
        a.copy(x = wrap(a.x+a.vx), y = wrap(a.y+a.vy))
    })
    shipX = wrap(shipX+shipVx)
    shipY = wrap(shipY+shipVy)
    var hit = List[Asteroid]()
    bullets = bullets.map(b => {
        b.copy(x = wrap(b.x+b.vx), y = wrap(b.y+b.vy), age = b.age+1)
    }).filter(b => {
        b.age<100 && asteroids.forall(a => {
            val dx = b.x-a.x
            val dy = b.y-a.y
            val dsqr = dx*dx+dy*dy
            val isHit = dsqr < a.size*a.size/4
            if(isHit) hit ::= a
            !isHit
        })
    })
    asteroids = asteroids.flatMap(a => {
        if(hit.contains(a)) {
            if(a.size <=10) List()
            else List.fill(4)(Asteroid(a.x+(math.random-0.5)*a.size,a.y+(math.random-0.5)*a.size,a.vx
        } else List(a)
    })
    if(asteroids.exists(a => {
            val dx = shipX-a.x
            val dy = shipY-a.y
            val dsqr = dx*dx+dy*dy
            dsqr < (shipSize+a.size/2)*(shipSize+a.size/2)
    })) timer.stop()
    panel.repaint
}))
```

Setup of the game is fairly concise. Create the main (Swing) frame and set the desired properties. In particular, we embed the created panel (a method that is familiar to anyone who has taught Java based UIs using AWT or Swing) and disallow frame resizing. We also center the frame on the screen (if supported). Then we display the frame and start

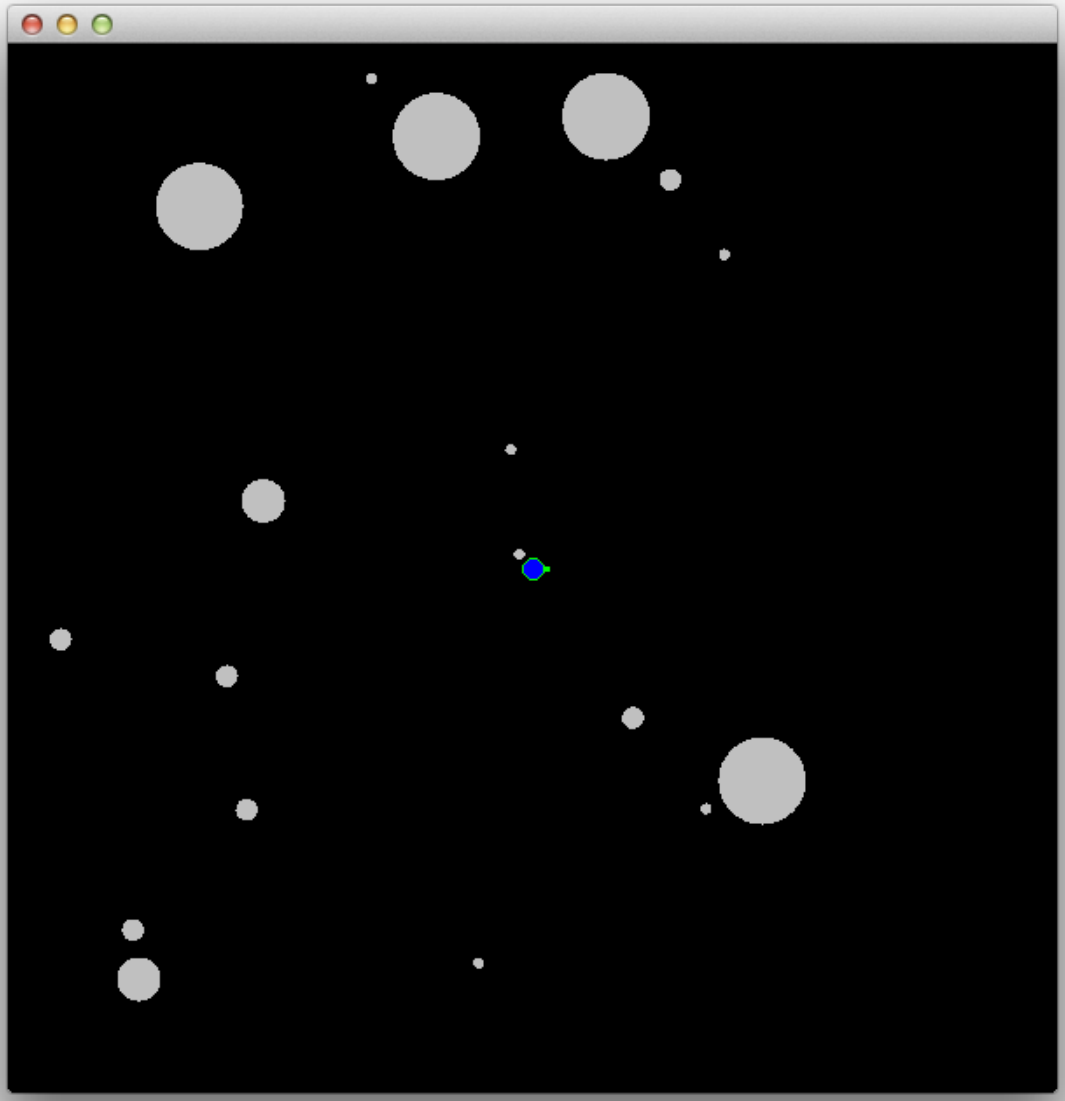
the timer.

While there are a few details about Swing to know here, much of this code is common to all Swing application development, so it is eminently teachable–and you can always point students to the basic documentation for Java to learn the details.

```
val frame = new MainFrame {
    contents = panel
    resizable = false
    centerOnScreen
}

frame.open
panel.requestFocus
timer.start
```

Here's a screenshot of the game:



1.5.4 CS2

- Pure OO
- Fewer quirks than Java
- Powerful type system
- Traits
- Rich collections
- Libraries again
- Can make things interesting/relevant

- Multithreading and networking
- Eclipse (maybe) and IntelliJ (our favorite)
- Scalable language
- Libraries as language
- Special methods

1.6 Build Tools for Scala

In this section, we provide a brief overview of build tools for Scala. In general, build tools support the build process in several ways:

- structured representation of the project dependency graph
- management of the build lifecycle (compile, test, run)
- management of external dependencies

1.6.1 Why Use a Build Tool?

When using the Java or Scala command-line tools, the developer is responsible for setting the dreaded classpath. This can quickly become unwieldy when dependencies even as simple as JUnit are involved, so this is not something you would usually want to do manually.

1.6.2 Brief History of Build Tools

Unix make, Apache ant These tools manage the build lifecycle but not external dependencies.

Apache maven This tool also manages external dependencies but requires a lot of XML-based configuraration.

```
<dependency>
        <groupId>org.restlet</groupId>
        <artifactId>org.restlet.ext.spring</artifactId>
        <version>${restlet.version}</version>
</dependency>
```

Apache ivy, Gradle, Scala's Simple Build Tool (sbt), etc. These tools emphasize convention over configuration in support of agile development processes. sbt is compatible with ivy and designed primarily for Scala development. For example, ivy uses a structured but lighter-weight format:

```
<dependency org="junit" name="junit" rev="4.11"/>
```

1.6.3 sbt

In the simplest case, sbt does not require any configuration and will use reasonable defaults. The project layout is similar to that used by Maven:

- Production code goes in `src/main/scala`.
- Test code goes in `src/test/scala`.

sbt supports two configuration styles, one based on a simple Scala-based domain-specific language, and one based on the full Scala language for configuring all aspects of a project.

build.sbt format

A minimal sbt `build.sbt` file would look like this. The empty lines are required, and the file must be placed in the project root folder.

```
name := "integration-scala"

version := "0.0.2"
```

Additional dependencies can be specified either one at a time

```
libraryDependencies += "com.novocode" % "junit-interface" % "0.10" % "test"
```

or as a group

```
libraryDependencies ++= Seq(
  "org.scala-lang" % "scala-actors" % "2.10.1",
  "com.novocode" % "junit-interface" % "0.10" % "test"
)
```

Build.scala format

Examples of more complex Scala-based project configurations can be found in these examples:

- Android click counter app
- Prime checker web service

1.6.4 Plugin Ecosystem

sbt includes a growing plugin ecosystem. Key examples include

sbteclipse automatically generates an Eclipse project configuration from an sbt one.

sbt-start-script generates a start script for running a Scala application outside of sbt.

The IntelliJ IDEA Scala plugin also integrates directly with sbt.

1.7 Web Application and Services

1.7.1 Professional Context

Disclaimer: I am not affiliated in any form with Typesafe.

In this section, we discuss how Scala can enhance the pedagogy in certain foundational and applied advanced courses. Given that the majority of students coming out of our undergraduate and graduate programs find jobs in local and global industry, we place considerable emphasis on professional practice.

To this end, we draw on several (local and global) resources:

- Uncle Bob
- ThoughtWorks Technology Radar
- IEEE Software's Software Engineering Radio

We attempt to strike a balance among solid foundational knowledge, state-of-the-art technology, and job market demands. In particular, we have adopted the following rule for our advanced courses and research projects: *new code in Java (the language, as opposed to the platform)*.

We also strive to choose technologies that scale properly. Here, the typical startup story goes like this: Implement in RoR, oops, it doesn't scale, then redo in guess what language? There are an increasing number of good language and platform choices here, and in our experience, Scala is one of them.

The job market is also increasingly interested in Scala. Here is an interesting local internship ad.

1.7.2 Curricular Context

Starting in 2010, we have been incorporating Scala into several of these courses, taught mostly by Konstantin and experienced professionals serving as adjuncts.

- COMP 373/473: Advanced Object-Oriented Development, using Scala since spring 2010, planning to add Android in spring 2014
- COMP 372/471: Theory (and Practice) of Programming Languages, using Haskell (and some F#) since fall 2010
- COMP 338/442: Server-Side Software Development (focusing on web applications), using Scala with the Play! framework since fall 2010
- COMP 388/433: Web Services Development, using Scala with the spray toolkit since spring 2011
- COMP 313/413: Intermediate Object-Oriented Development (focusing on software design and architecture), using Java with Android since fall 2012, considering the addition of Scala down the road

1.7.3 Why Scala?

In our experience, Scala can help with the pedagogy of advanced applied courses in multiple ways:

- less boilerplate
- focus on deep concepts
- focus on good practices

In the remainder of this section, we will give an overview of web application and web service development in Scala and contrast the experience with the corresponding Java-based techniques.

1.7.4 Web Applications

The predominant web application frameworks targeting Scala are Lift and Play!. Typesafe has adopted the latter as part of its official stack, and we had actually started to use Play! in our course before Typesafe's decision was known.

The primary concerns in the development of web application for human users—as opposed to a web service for programmatic consumption—are

- **views**
  - presentation
  - visual styles
  - layout
  - navigation

  - i18n
- **controllers**
  - validation
  - authentication
  - dynamic behavior/interaction/state
- **models**
  - services
  - domain model
- **persistence**
  - database technologies
  - mapping domain object to persistent storage
- **testing**
  - unit testing
  - integration testing
  - functional testing
  - acceptance testing
  - performance/load testing

When using a Java-based stack, we typically address these concerns with a stack tied together by a dependency-injection framework such as Spring and an object-relational mapper (ORM) such as Hibernate, along with a MVC framework for the upper layers.

When using a Scala-based stack, we an express an equivalent architecture much more concisely and using language mechanisms instead of requiring a DI framework.

Examples

- Linear regression in Java with Spring, Stripes, and maven
- To-do list in Scala with Play!
- Live version of the to-do list deployed to the cloud

1.7.5 Web Services

As opposed to the Simple Object Access Protocol (SOAP), we will focus on representational state transfer (REST), which has emerged as the preferred approach of the broader agile community.

The implementing-rest community has put together a helpful language matrix of REST libraries, toolkits, and frameworks. Typesafe's stack does not yet include strong support for RESTful web services in the sense of a high-level DSL for request routing, which is where some of the choices in the language matrix come into the picture.

We have picked spray not only because the author is from Konstantin's home town but also because it is:

- concise
- flexible
- type-safe

- focus on HTTP and request routing
- more and more widely used and supported

Scala and Spray are supported by Heroku and several other newer APaaS cloud providers. Deploying a service to the cloud requires a simple Git commit; this makes it possible to achieve continuous delivery.

Examples

- Social bookmarking example based on Java with the Restlet framework
- Prime number checker based on Scala with spray

1.8 Mobile Application Development with Android

Mobile applications backed by cloud-based RESTful services have emerged as the primary face of computing in terms of massive consumer participation. Jason Christensen described this system architecture in his OOPSLA 2009 presentation. Therefore, not only do we find it important to cover this system architecture in the curriculum, but we also see this architecture as a very effective context for teaching various important computer science topics:

- **(real-world) software architecture**
  - dependency inversion principle (DIP)
  - model-view-adapter architectural pattern (MVA)
  - testability
  - etc.
- **concurrent, parallel, and distributed computing topics (PDC/TCPP/EduPar)**
  - events
  - timers (one-shot and recurring)
  - background threads
  - offloading tasks to the cloud
- **embedded/resource-conscious computing**
  - limitations of the device
  - capabilities of the device (numerous sensors)

Konstantin drew the inspiration to use Android instead of Swing as a context for teaching these topics from the mobile computing session at SIGCSE 2012 in Raleigh.

Furthermore, we have found the cost of switching from, say, Java Swing to Android minimal. Besides, Android matters in the real world: it is a widely used technology, and mobile app development skills are in increasing demand.

While our overall goals are similar to those of the Sofia framework project, we discuss here a language-based approach but are planning to enhance the practice of Android development in Scala through additional support classes.

As mentioned above, current and future focus has been on these courses:

- COMP 313/413: Intermediate Object-Oriented Development (focusing on software design and architecture), using Java with Android since fall 2012, considering the addition of Scala down the road
- COMP 373/473: Advanced Object-Oriented Development, using Scala since spring 2010, planning to add Android in spring 2014

1.8.1 Tools

There are two sbt plugins for developing Android:

- https://github.com/pfn/android-sdk-plugin
- http://fxthomas.github.io/android-plugin

The rapidly evolving topic of developing Android apps in Scala is the subject of this discussion forum:

- https://groups.google.com/forum/#!topic/scala-on-android

After some experimentation, we have found pfn's plugin to be easier to use and have adopted it for our Scala-based Android development.

1.8.2 Examples

The learning objectives of each example are stated in the example's readme.

- Clickcounter app
- Prime checker app
- Prime checker web service

1.8.3 Lab Assignment

Format Pair project

Time 10 minutes

Deliverable An enhancement of this clickcounter app that addresses at least one following additional functional requirements:

- New user story: a max (^) button as the analogous dual to the reset (0) button.
- Retaining application state during rotation (see here to find out how to rotate the emulator).

Nonfunctional requirements

- You should update the tests and the rest of the existing code accordingly.
- You should implement the `onSaveInstanceState` and `onRestoreInstanceState` application lifecycle methods (see for details. The system passes this method a Bundle in which you can save state information about the activity as name-value pairs, using methods such as `putString()` and `putInt()`.

1.9 Basic Parallelism using Par

Using parallelism solves problems more quickly than using a single-processor machine, as is the case where groups of people solve problems larger than one person can solve. But just as with groups of people, there are additional costs and problems involved in coordinating parallel processors:

- We need to have more than one processor work on the problem at the same time. Our machine must have more than one processor, and the operating system must be able to give more than one processor to our program at the same time. Kernel threads allow this in Java. An alternative approach is to have several networked computers work on parts of the problem; this is discussed in Chapters 11 and 12, "Networking" and "Coordination." of the HPJPC book by Christopher and Thiruvathukal.

- We need to assign parts of the problem to threads. This at least requires rewriting a sequential program. It usually requires rethinking the algorithm as well.
- We need to coordinate the threads so they perform their operations in the proper order, as well as avoid race conditions and deadlocks. A number of useful facilities are not provided by the standard Java language package. We provide a good collection for your use in our thread package.
- We need to maintain a reasonable grain size. Grain size refers to the amount of work a thread does between communications or synchronizations. Fine grain uses very few instructions between synchronizations; coarse grain uses a large amount of work. Too fine a grain wastes too much overhead creating and synchronizing threads. Too coarse a grain results in load imbalance and the underutilization of processors.

Two easy, practical approaches to dividing the work among several processors are executing functions in parallel and executing iterations of loops in parallel. Parallelizing loops will be presented in the next chapter. In this chapter we will discuss running subroutines in parallel.

Executing functions in parallel is an easy way to speed up computation. The chunks of code are already packaged for you in methods; you merely need to wrap runnable classes around them. Of course, there are certain requirements:

- The function must be able to run in parallel with some other computation. This usually means that there are several function calls that can run independently.
- The function must have a reasonable grain size. It costs a lot to get a thread running, and it doesn't pay off for only a few instructions.

Two kinds of algorithms particularly adaptable to parallel execution of functions are the divide-and-conquer and branch-and-bound algorithms. Divide-and-conquer algorithms break large problems into parts and solve the parts independently. Parts that are small enough are solved simply as special cases. You must know how to break a large problem into parts that can be solved independently and whose solutions can be reassembled into a solution of the overall problem. The algorithm may undergo some cost in breaking the problem into subparts or in assembling the solutions.

1.9.1 Example: Trapezoidal Numeric Integration

Sometimes, a program needs to integrate a function (i.e., calculate the area under a curve). It might be able to use a formula for the integral, but doing so isn't always convenient, or even possible. An easy alternative approach is to approximate the curve with straight line segments and calculate an estimate of the area from them.

This visual (courtesy of HPJPC) shows the trapezoidal method:

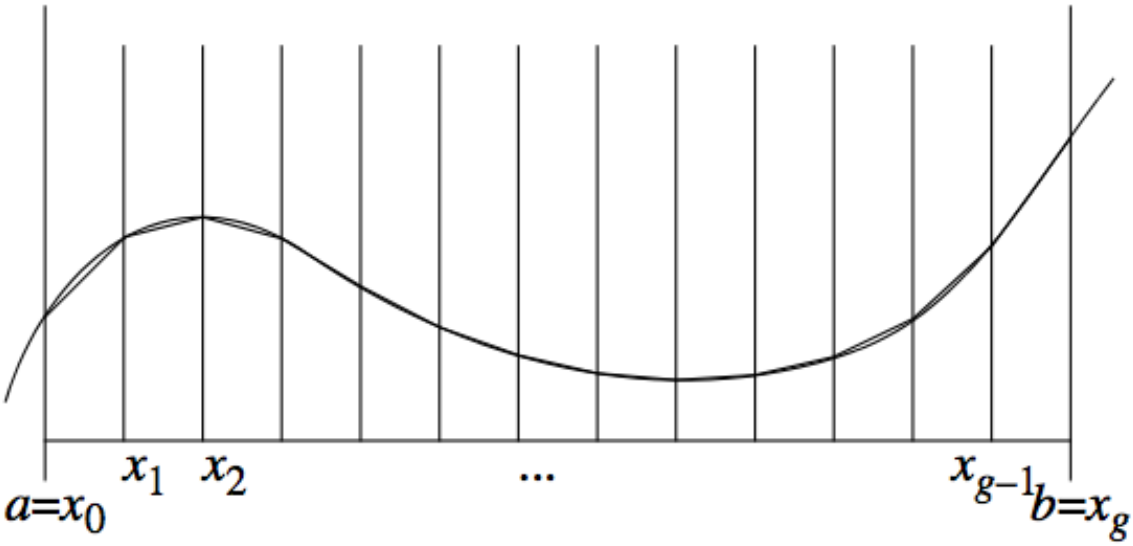


Figure 5–1 Approximating an integral with trapezoids.

This equation shows how to calculate the area.

We wish to find the area under the curve from a to b. We approximate the function by dividing the domain from a to b into g equally sized segments. Each segment is $(b-a)/g$ long. Let the boundaries of these segments be $x_0 = a$, x_1, x_1, ... , $x_g = b$. The polyline approximating the curve will have coordinates $(x_0, f(x_0))$, $(x_1, f(x_1))$, $(x_2, f(x_2))$, ..., $(x_g, f(x_g))$.

The area is then given by this formula, which sums the area of all trapezoids:

$$A = \sum_{i=1}^{g} \frac{1}{2} \cdot \frac{(b-a)}{g} \cdot (f(x_{i-1}) + f(x_i))$$

If we apply that formula unthinkingly, we will evaluate the function twice for each value of x, except the first and the last values. While the correct result would be obtained, it is inefficient and kind of defeats the purpose of going parallel. A little manipulation gives us the following:

$$A = \frac{b-a}{g} \cdot \left(\frac{f(a)}{2} + \frac{f(b)}{2} + \sum_{i=1}^{g-1} f(x_i) \right)$$

We've now reduced the problem to computing the parallel sum term, which is represented nicely in Scala.

We present three solutions in Scala:

- `integrateSequential()`: The sequential (i.e. not parallel) solution is aimed at showcasing one of the key benefits of functional programming in general. In many cases, the code closely follows the mathematics you write. Because the code is clear in terms of its intentions, we can adapt the solution for parallel execution.
- `integrateParallel()`: The parallel version shows how to get parallel speedup in Scala from the sequential one, simply by adding `par`.
- **`integrateParallelGranular()`: This version shows how to combine parallel and serial execution. By** allowing the user to specify the grain size (number of retangles to do sequentially at a time), it is possible to determine how Scala itself might be chunking the work in the `integrateParallel()` solution.

1.9.2 Download the Code

ZIP File https://bitbucket.org/loyolachicagocs_plsystems/integration-scala/get/default.zip

via Mercurial hg clone https://bitbucket.org/loyolachicagocs_plsystems/integration-scala

Build and Run Instructions https://bitbucket.org/loyolachicagocs_plsystems/integration-scala/overview

1.9.3 Going Scala!

Let's start by looking at `integrateSequential()`.

```
1   def integrateSequential(a: Double, b: Double, rectangles: Int, f: Fx): Double = {
2     val interval = (b - a) / rectangles
3     val fxValues = (1 until rectangles).view map { n => f(a + n * interval) }
4     interval * (f(a) / 2 + f(b) / 2 + fxValues.sum)
5   }
```

As would be expected, we should be write this the way we think of the problem mathematically:

- `a` and `b`: The endpoints of the integration interval
- `rectangles`: The number of rectangles to use to approximate the integral
- `f`: The function to be integrated.

In the last case, this is where Scala makes our work particularly easy by allowing us to define a proper function type as shown below:

```
1   type Fx = Double => Double
```

In our previous Java work (in the book), we had to use Java *interfaces* for this same task. While seemingly just *syntactic sugar*, Java's boilerplate is offputting to computational scientists, who would rather use C and FORTRAN where function parameters are possible. Unlike those choices, however, Scala gives us the compelling aspect of *full type checking*, which means that we can be assured of excellent performance without the complexity–and sometimes unsafe behavior–that is found in other languages.

Because it is essential to understand the sequential version (the core algorithm) before proceeding, we offer a brief explanation of each line of code, even when obvious, and how we are taking advantage of Scala (when less obvious!)

- In line 2, we calculate `interval` using the formula that was presented earlier. Scala, being pragmatic, is able to do the right thing and treat the entire expression as Double, resulting in a Double value. Scala really shines here by not requiring us to declare every val type, owing to its innate type inferencing mechanism. This results in code that is much easier to comprehend (at least we like to think so).
- In line 3, this requires a bit more explanation. Working from our equation, recall that our summation term goes from 1 to the number of rectangles minus 0. We use Scala's until to get the indices. When `rectangles` is small, this is fine. What happens when it is large? The answer depends on whether we are using eager or lazy evaluation. In mathematical/scientific computing, we often need to do a large number of iterations to get a better answer. This is where the `view` comes in. It gives us a lazy sequence that can then be mapped (also lazily) using the user-supplied function, `f`.
- In line 4, we are able to plug everything into the derived formular for calculating the trapezoidal integration. Technically, we could have put the `fxValues` val definition in the same line of code, but having it separate makes it easier to understand for new Scala users (one of the goals for our SIGCSE workshop). More importantly, you should be able to write the code this way and not have to worry about losing performance. By setting up this lazy computation, we're able to compute the sum *on demand*. Aside from this split, the code here exactly matches the formula we derived for performing trapezoidal integration.

The sequential version of integration presented here is completely side-effect free. That is, all of the work is being done without mutating state. This means that it can immmediately be turned into something parallel in Scala, provided we know where the actual work is being done. Let's continue this exploration!

1.9.4 Going Parallel

The immediately preceding discussion was presented with great care, because what we are about to demonstrate illustrates how one needs to do very little work to take what is sometimes known as an *embarrassingly parallel* algorithm and make it run in parallel. The term *embarrassing* is a tad misleading. As we'll see, the results don't always follow your intuition. Furthermore, while the results can be gotten quickly, it doesn't always mean that you are getting the best results possible. For example, as well as Scala does, it still doesn't do nearly as well as our hand-coded multithreaded Java example from HPJPC. We'll say more about this later.

Let's look at the parallel version.

```
1   def integrateParallel(a: Double, b: Double, rectangles: Int, f: Fx): Double = {
2     val interval = (b - a) / rectangles
3     val fxValues = (1 until rectangles).par.view map { n => f(a + n * interval) }
4     interval * (f(a) / 2 + f(b) / 2 + fxValues.sum)
5   }
```

In this version, observe that we have added the `par` method call just before generating the lazy `view`. This is the only sensible place to add `par`, because in mathematical/scientific computing, we know that most of the parallel potential is found *where the loops are*. The `1 until rectangles` is where the actual workload is being generated, so it is a natural place to suggest `par`.

1.9.5 Testing

The following code shows the unit tests for our various *integration* examples.

```
package edu.luc.etl.sigcse13.scala.integration

import org.junit.Test
import org.junit.Assert._
import Integration._
import Fixtures._

/**
 * Simple JUnit-based tests.
 */
class Tests {

  @Test def testSequential() {
    assertEquals(333.3, integrateSequential(0, 10, 1000, sqr), 0.1)
  }

  @Test def testParallel() {
    assertEquals(333.3, integrateParallel(0, 10, 1000, sqr), 0.1)
  }

  @Test def testParallelGranular() {
    assertEquals(333.3, integrateParallelGranular(10)(0, 10, 1000, sqr), 0.1)
  }
}
```

For the purpose of testing, we set up $f(x) = x^2$ and integrated it from 0 to 10. The value of this integral should be 333.333333333....

We can't stress enough the importance of unit tests, especially when working a sequential algorithm into a parallel one. While you are less likely to make mistakes in Scala, it is very easy when trying certain strategies to get the wrong answer. (In fact, one of them, the third, gave an incorrect answer during testing, simply because of a division error I made when computing the number of workers!)

Using the notion of a test fixture, it is possible to specify what function we wish to test without contaminating the general-purpose code we wrote with a specific function to be integrated. See below.

```
package edu.luc.etl.sigcse13.scala.integration

// begin-object-Fixtures
object Fixtures {
  def sqr(x: Double): Double = x * x
}
// end-object-Fixtures
```

1.9.6 Running

The tradition of scientific computing is one where users *want* and *need* to be able to run it from the command-line, often in an unattended fashion (say, on a computing cluster or network of workstations or cloud resources). The following is

the main program we put together to run the integration of $f(x) = x^2$ manually. You can specify the number of rectangles, the number of times to run each of the experiments, and a grain size for testing the combined parallel/sequential version, `integrateParallelGranular()`. We'll say more about this function in our performance discussion.

```
package edu.luc.etl.sigcse13.scala.integration

import Integration._
import Fixtures.sqr

object Main extends {

  def main(args: Array[String]) {
    try {
      require { 2 <= args.length }
      val rectangles = math.max(args(0).toInt, 1000)
      val n = math.max(args(1).toInt, 1)
      val grainSize = if (args.length == 3) math.min(args(2).toInt, rectangles) else rectangles

      timedRun(rectangles, n, "sequentially", integrateSequential)
      timedRun(rectangles, n, "in parallel", integrateParallel)
      timedRun(rectangles, n, "in parallel with " + grainSize +
        " rectangles per serial worker", integrateParallelGranular(grainSize))

    } catch {
      case _: NumberFormatException    => usage()
      case _: IllegalArgumentException => usage()
    }
  }

  def usage() {
    Console.err.println("usage: rectangles (>= 1000) " +
      "numberOfRuns (>= 1) [ grainSize (rectangles % grainSize == 0) ]")
  }

  def timeThis[A](s: String)(block: => A): A = {
    val time0 = System.currentTimeMillis
    val b = block
    val time1 = System.currentTimeMillis - time0
    println("Timing " + s + " = " + time1)
    b
  }

  // begin-timedRun
  def timedRun(rectangles: Int, n: Int, how: String,
    integrationStrategy: (Double, Double, Int, Fx) => Double) {
    timeThis(how) {
      print("Computing area " + how + "; now timing " + n + " iterations")
      val area: Double = (1 to n).map { _ => integrationStrategy(0, 10, rectangles, sqr) }.head
      println("; area = " + area)
    }
  }
  // end-timedRun
}
```

1.9.7 Initial Experiments with Performance

This is still being written up but will be demonstrated live.

```
1   def integrateParallelGranular(grainSize: Int)(a: Double, b: Double, rectangles: Int, f: Fx): Double
2     require { rectangles % grainSize == 0 } // can relax this later
3     val workers = rectangles / grainSize
4     val interval = (b - a) / workers
5     val fullIntegration = (0 until workers).par.view map { n =>
6       val c = a + n * interval
7       integrateSequential(c, c + interval, grainSize, f)
8     }
9     fullIntegration.sum
10  }
```

1.9.8 Previous Work

This example was developed as part of High-Performance Java Platform Computing by Thomas W. Christopher and George K. Thiruvathukal.

PDF of Book https://hpjpc.googlecode.com/files/HPJPC%20Christopher%20and%20Thiruvathukal.pdf

bitbucket **ZIP File** https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java/get/default.zip

bitbucket **via Mercurial** hg clone https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java

Build and Run Instructions https://bitbucket.org/loyolachicagocs_plsystems/integration-scala/overview

1.10 Parallelism using Actors

In this section, we're going to take a look at the Java vs. Scala way of doing things. We'll look at a guiding example that is focused on concurrent/parallel computing. This example appeared in *High Performance Java Platform Computing* by Thomas W. Christopher and George K. Thiruvathukal. We'll show how to organize a previously worked out solution that uses more explicit concurrency mechanisms from Java and how it can be reworked into a side-effect free Scala version by taking advantage of Scala's innate support for basic actor-style parallelism.

1.10.1 Guiding Example: Longest Common Subsequence

A longest common subsequence (LCS) of two strings is a longest sequence of characters that occurs in order in the two strings. It differs from the *longest common* substring in that the characters in the longest common subsequence need not be contiguous. There may, of course, be more than one LCS, since there may be several subsequences with the same length.

There is a folk algorithm to find the *length* of the LCS of two strings. The algorithm uses a form of dynamic programming. In divide-and-conquer algorithms, recall that the overall problem is broken into parts, the parts are solved individually, and the solutions are assembled into a solution to the overall problem. Dynamic programming is similar, except that the best way to divide the overall problem into parts is not known before the subproblems are solved, so dynamic programming solves all subproblems and then finds the best way to assemble them.

The algorithm works as follows: Let the two strings be `c0` and `c1`. Create a two-dimensional array `a`:

```
int [][] a=new int[c0.length()+1] [c1.length()+1]
```

Initialize `a[i][0]` to 0 for all `i` and `a[0][j]` to 0 for all `j`, since there are no characters in an empty substring. The other elements, `a[i][j]` , are filled in as follows:

```
for (int i=0; i <= c0.length(); i++)
   a[i][0] = 0;

for (int j=0; j <= c1.length(); j++)
   a[0][j] = 0;
```

We will fill in the array so that `a[i][j]` is the length of the LCS of `c0.substring(0,i)` and `c1.substring(0,j)`. Recall that `s.substring(m,n)` is the substring of `s` from position `m` up to, but not including, position `n`.:

```
for (i=1; i <= c0.length(); i++)
   for (j=1; j <= c1.length(); j++)
      if (c0.charAt(i-1) == c1.charAt(j-1))
         a[i][j]=a[i-1][j-1]+1;
      else
         a[i][j]=Math.max(a[i][j-1],a[i-1][j]);
```

The above shows a *traditional* imperative solution that constructs a result matrix comprising the results of the LCS.

So how exactly does this method work?

Element `a[i-1][j-1]` has the length of the LCS of string `c0.substring(0,i-1)` and `c1.substring(0,j-1)`. If elements c0.charAt(i-1) and c1.charAt(j-1) are found to be equal, then the LCS can be extended by one to length `a[i-1] [j-1]+1`. If these characters don't match, then what? In that case, we ignore the last character in one or the other of the strings. The LCS is either `a[i][j-1]` or `a[i-1][j]`, representing the maximum length of the LCS for all but the last character of `c1.substring(0,j-1)` or `c0.substring(0,i-1)`, respectively.

The chore graph from [HPJPC] for calculation of the LCS is shown in the following figure.

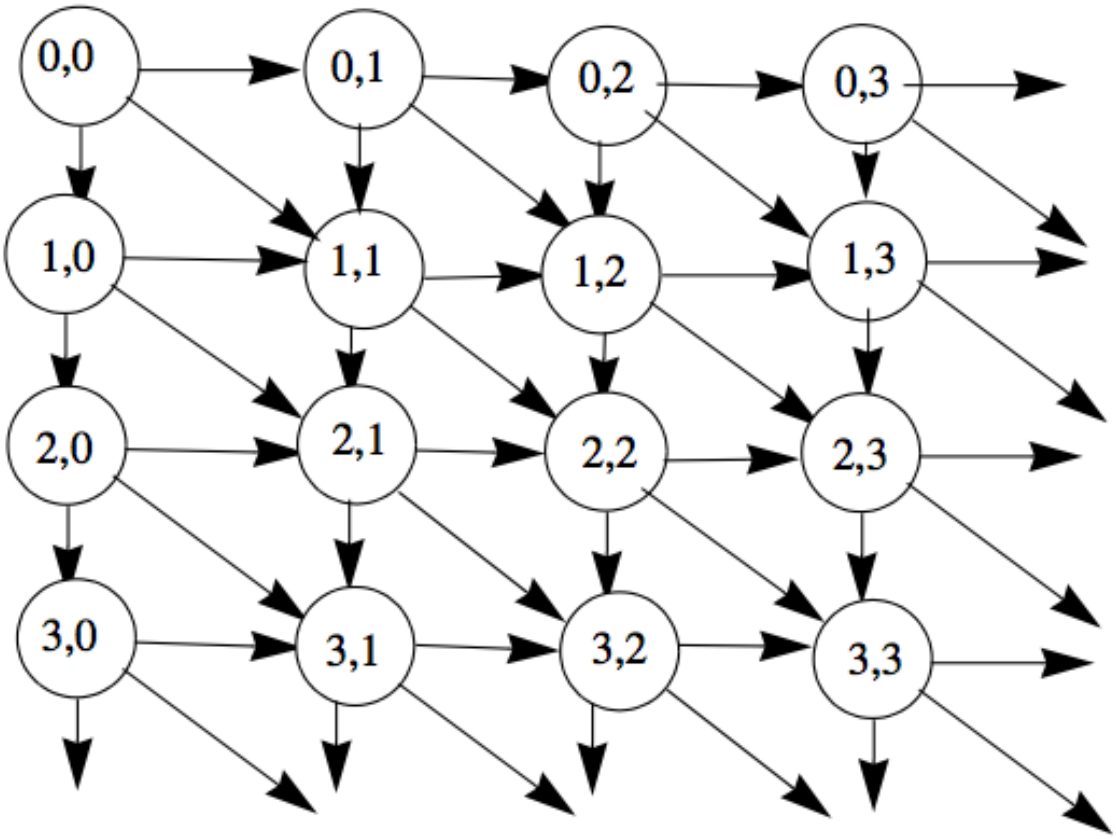


Figure 6–8 Chore graph for LCS.

Any order of calculation that is consistent with the dependencies is permissible. Two are fairly obvious: (1) by rows, top to bottom, and (2) by columns, left to right.

Another possibility is along diagonals. All `a[i][j]`, where `i+j==m` can be calculated at the same time, for `m` stepping from 2 to `c0.length()+c1.length()`. . Visualizing waves of computation passing across arrays is a good technique for designing parallel array algorithms. It has been researched under the names systolic arrays and wavefront arrays [Wavefront].

The following figure shows how a wavefront computation progresses.

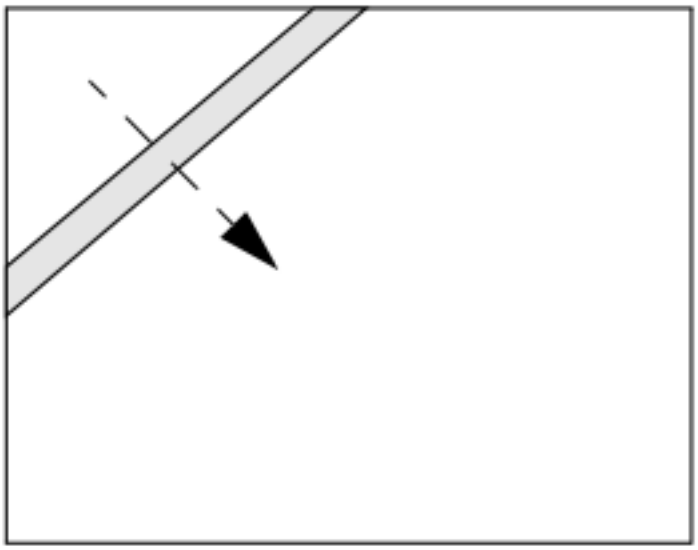


Figure 6–9 Wavefront calculation of LCS. Shading represents elements being calculated.

1.10.2 Java Threads Implementation

ZIP File https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java/get/default.zip

via Mercurial hg clone https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java

Build and Run Instructions https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java

Our Java implementation (see LCS.java) of the LCS algorithm divides the array into vertical bands and is pictured in Each band is filled in row by row from top to bottom. Each band (except the leftmost) must wait for the band to its left to fill in the last element of a row before it can start can start filling in that row. This is an instance of the producer-consumer releationship.

The following figure shows how our Java solution organizes the work in bands:

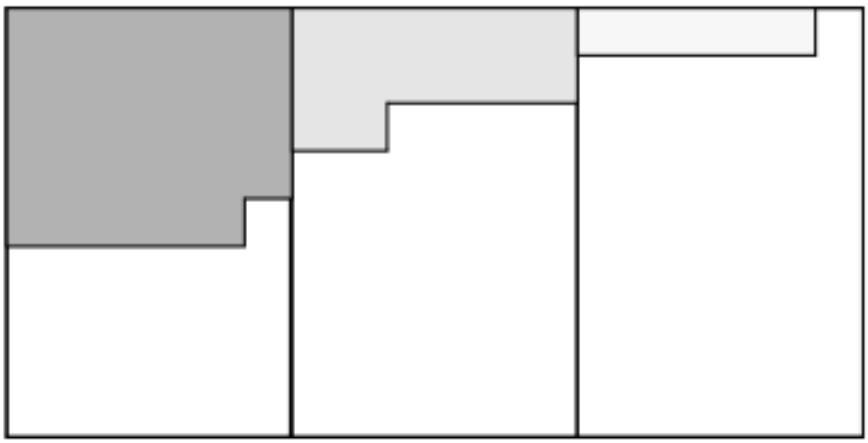


Figure 6–10 LCS calculated in vertical bands. Shading represents elements that have been calculated.

LCS class

```
    int numThreads;
    char[] c0;
    char[] c1;
    int[][] a;
    Accumulator done;
```

```
    public LCS(char[] c0, char[] c1, int numThreads) {
       this.numThreads = numThreads;
       this.c0 = c0;
       this.c1 = c1;
       int i;
       done = new Accumulator(numThreads);

       a = new int[c0.length + 1][c1.length + 1];

       Semaphore left = new Semaphore(c0.length), right;
       for (i = 0; i < numThreads; i++) {
          right = new Semaphore();
          new Band(startOfBand(i, numThreads, c1.length), startOfBand(i + 1,
                numThreads, c1.length) - 1, left, right).start();
          left = right;
       }
    }
```

```
    public LCS(String s0, String s1, int numThreads) {
       this(s0.toCharArray(), s1.toCharArray(), numThreads);
    }
```

```
    int startOfBand(int i, int nb, int N) {
       return 1 + i * (N / nb) + Math.min(i, N % nb);
    }
```

```
   public int getLength() {
      try {
         done.getFuture().getValue();
      } catch (InterruptedException ex) {
      }
      return a[c0.length][c1.length];
   }
```

Band internal class (does the work)

```
      int low;
      int high;
      Semaphore left, right;

      Band(int low, int high, Semaphore left, Semaphore right) {
         this.low = low;
         this.high = high;
         this.left = left;
         this.right = right;
      }
```

Actual Runnable body

```
      public void run() {
         try {
            int i, j;
            for (i = 1; i < a.length; i++) {
               left.down();
               for (j = low; j <= high; j++) {
                  if (c0[i - 1] == c1[j - 1])
                     a[i][j] = a[i - 1][j - 1] + 1;
                  else
                     a[i][j] = Math.max(a[i - 1][j], a[i][j - 1]);
               }
               right.up();
            }
            done.signal();
         } catch (InterruptedException ex) {
         }
      }
```

Main

```
      public static void main(String[] args) {
         if (args.length < 2) {
            System.out.println("Usage: java LCS$Test1 string0 string1");
            System.exit(0);
         }
         int nt = 3;
         String s0 = args[0];
         String s1 = args[1];
         System.out.println(s0);
         System.out.println(s1);
         long t0 = System.currentTimeMillis();
         LCS w = new LCS(s0, s1, nt);
         long t1 = System.currentTimeMillis() - t0;
         System.out.println(w.getLength());
         System.out.println("Elapsed time " + t1 + " milliseconds");
      }
```

1.10.3 Scala Actors Implementation

ZIP File https://bitbucket.org/loyolachicagocs_plsystems/lcs-systolicarray-scala/get/default.zip

via Mercurial hg clone https://bitbucket.org/loyolachicagocs_plsystems/lcs-systolicarray-scala

Build and Run Instructions https://bitbucket.org/loyolachicagocs_plsystems/lcs-systolicarray-scala

Trait

```
trait SystolicArray[T] {
  def start(): Unit
  def put(v: T): Unit
  def take(): T
  def stop(): Unit
}
```

The entire SystolicArray implementation is here:

```
trait SystolicArray[T] {
  def start(): Unit
  def put(v: T): Unit
  def take(): T
  def stop(): Unit
}
```

Logging

// begin-object-logger

```
  private object logger {
    private val DEBUG = false
    // use call-by-name to ensure the argument is evaluated on demand only
    def debug(msg: => String) { if (DEBUG) println("debug: " + msg) }
    // add other log levels as needed
  }
```

Apply

```
  def apply[T](rows: Int, cols: Int, f: Acc[T]): SystolicArray[T] = {
    require { 0 < rows }
    require { 0 < cols }
    val result = new SyncVar[T]
    lazy val a: LazyArray[T] = Stream.tabulate(rows, cols) {
      (i, j) => new Cell(i, j, rows, cols, a, f, result)
    }
    val root = a(0)(0)
    new SystolicArray[T] {
      override def start() = root.start()
```

```
      override def put(v: T) { root ! ((-1, -1) -> v) }
      override def take() = result.take()
      override def stop() { root ! Stop }
    }
  }
```

The internal Cell class, used to represent the cells of the Systolic Array (generally).

```
  protected class Cell[T](row: Int, col: Int, rows: Int, cols: Int, a: => LazyArray[T],
      f: Acc[T], result: SyncVar[T]) extends Actor { self =>

    require { 0 <= row && row < rows }
    require { 0 <= col && col < cols }

    logger.debug("creating (" + row + ", " + col + ")")

    override def act() {
      logger.debug("starting (" + row + ", " + col + ")")
      var start = true
      loop {
        logger.debug("waiting  (" + row + ", " + col + ")")
        barrier(if (row == 0 || col == 0) 1 else 3) { ms =>
          if (start) { startNeighbors() ; start = false }
          propagate(ms)
        }
        // one-way message: anything below here is skipped!
      }
    }

    protected def barrier(n: Int)(f: Map[Pos, T] => Unit): Unit =
      barrier1(n)(f)(Map.empty)

    protected def barrier1(n: Int)(f: Map[Pos, T] => Unit)(ms: Map[Pos, T]): Unit = {
      if (n <= 0)
        f(ms)
      else
        react {
          case Stop => stopNeighbors() ; exit()
          case (p: Pos, v: T) => barrier1(n - 1)(f)(ms + (p -> v))
        }
      // one-way message: anything after react is skipped!
    }

    protected def applyToNeighbors(f: Cell[T] => Unit) {
      if (row < rows - 1)                   f(a(row + 1)(col    ))
      if (col < cols - 1)                   f(a(row    )(col + 1))
      if (row < rows - 1 && col < cols - 1) f(a(row + 1)(col + 1))
    }

    protected def startNeighbors() { applyToNeighbors { _.start() } }

    protected def propagate(ms: Map[Pos, T]) {
      val r = f((row, col), ms)
      val m = (row, col) -> r
      logger.debug("firing   " + m)
      if (row < rows - 1)                    a(row + 1)(col    ) ! m
      if (col < cols - 1)                    a(row    )(col + 1) ! m
      if (row < rows - 1 && col < cols - 1)  a(row + 1)(col + 1) ! m
```

```
51        if (row >= rows - 1 && col >= cols - 1) result.put(r)
52      }
53
54      protected def stopNeighbors() { applyToNeighbors { _ ! Stop } }
55    }
```

This is used for autowiring the quadrant from where messages are being fired (from). It is an example of how Scala can help us avoid making mistakes. In scientific computations, subscript problems are common.

```
1   implicit class Helper[T](ms: Map[Pos, T]) {
2     def north    (implicit current: (Pos, T)): T = ms.get((current._1._1 - 1, current._1._2    )).get
3     def west     (implicit current: (Pos, T)): T = ms.get((current._1._1    , current._1._2 - 1)).get
4     def northwest(implicit current: (Pos, T)): T = ms.get((current._1._1 - 1, current._1._2 - 1)).get
5   }
```

This is used to autowire the left and top edges of the array. Although easy enough to check, it can be difficult to remember which subscript is row or column. Scala again makes this very easy for us. As we'll see, it also helps to make the user function self-documenting (literate).

```
1   implicit class PosHelper(p: Pos) {
2     def north = p._1 - 1
3     def west = p._2 - 1
4     def isOnEdge = p._1 == 0 || p._2 == 0
5   }
```

Wrapping it up with object lcs...

```
1  object lcs {
2    import SystolicArray._
3
4    def f(c0: String, c1: String)(p: Pos, ms: Map[Pos, Int]) = {
5      implicit val currentPosAndDefaultValue = (p, 0)
6      if (p.isOnEdge)
7        0
8      else if (c0(p.north) == c1(p.west))
9        ms.northwest + 1
10     else
11       math.max(ms.west, ms.north)
12   }
13
14   def apply(c0: String, c1: String): Int = {
15     val root = SystolicArray(c0.length + 1, c1.length + 1, f(c0, c1))
16     root.start()
17     root.put(1)
18     root.take
19   }
20 }
```

Setting up the text fixtures...

```
1  object Fixtures {
2    import SystolicArray._
3
4    val c0 = "Now is the time for all great women to come to the aid of their country"
5
6    val c1 = "Now all great women will come to the aid of their country"
7
8    val f1 = (p: Pos, ms: Map[Pos, Int]) => ms.values.sum
9
```

```
  val f2 = (p: Pos, ms: Map[Pos, Int]) => {
    implicit val currentPosAndDefaultValue = (p, 0)
    ms.north + ms.northwest + ms.west
  }

  val f3 = lcs.f(c0, c1) _

  // "bare-metal" version of lcs, does not run significantly faster
  val f4 = (p: Pos, ms: Map[Pos, Int]) => {
    if (p._1 == 0 || p._2 == 0)
      0
    else if (c0(p._1 - 1) == c1(p._2 - 1))
      ms.get((p._1 - 1, p._2 - 1)).getOrElse(0) + 1
    else
      math.max(
        ms.get((p._1 - 1, p._2)).getOrElse(0),
        ms.get((p._1, p._2 - 1)).getOrElse(0))
  }
}
```

Testing...

```
class Tests {

  @Test def testSum() {
    val root = SystolicArray(3, 3, f1)
    root.start()
    root.put(1)
    assertEquals(13, root.take())
  }

  @Test def testSample() {
    assertEquals(53, lcs(c0, c1))
  }
}
```

1.11 Programming Language Topics

1.11.1 Abstraction

Topics

- higher-order methods
- traits
- genericity

1.11.2 Representation

Topics

- algebraic data types
- lexing and parsing
- interpreters

- domain-specific languages

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *search*

[RedmonkPL] Redmonk Programming Language Rankings, http://redmonk.com/sogrady/2014/01/22/language-rankings-1-14/

[ScalaWikipedia] Scala Programming Language, Wikipedia, http://en.wikipedia.org/wiki/Scala_(programming_language)

[LaScala] La Scala, Wikimedia Commons, http://en.wikipedia.org/wiki/File:Milano-scalanotte_e.jpg

[LaScalaConcert] Keith Jarrett, http://en.wikipedia.org/wiki/La_Scala_(album)‘

[TailCalls] Tail Call, http://en.wikipedia.org/wiki/Tail_call

[GCD] Greatest Common Divisor, http://introcs.cs.luc.edu/html/gcdexamples.html

[MiscExplorationsScala] Miscellaneous Scala Explorations, https://bitbucket.org/lucproglangcourse/misc-explorations-scala

[ScalaAPI] Scala API Documentation, http://www.scala-lang.org/api/current/#package

[Wavefront] 8. (a) Kung, C. E. Leiserson: Algorithms for VLSI processor arrays; in: C. Mead, L. Conway (eds.): Introduction to VLSI Systems; Addison-Wesley, 1979

[HPJPC] Thomas W. Christopher and George K. Thiruvathukal, *High Performance Java Platform Computing*, Prentice Hall PTR and Sun Microsystems Press, 2000.

A

Actors, 62
actors
    parallelism, 57
algorithm
    Longest Common Subsequence, 57

C

concurrency
    explicit, 60

D

dataflow, 62
download, 7

E

explicit
    concurrency, 60
    parallelism, 60

I

implicit
    parallelism, 62

L

logging
    technique, 63
Longest Common Subsequence
    algorithm, 57

P

parallelism
    actors, 57
    explicit, 60
    implicit, 62

S

systolic arrays, 62

T

technique
    logging, 63