Lecture 1

Course Description: Development of individual skills necessary for designing, implementing, testing and modifying larger programs, including: use of integrated design environments, design strategies and patterns, testing, working with large code bases and libraries, code refactoring, and use of debuggers and tools for version control.

Introduction

COMPSCI220 Programming Methodology introduces you to all the concepts above in the context of a modern programming language: Scala. You could use Scala to write exactly the same kind of object-oriented code that you've seen in Java. In fact, Scala code and Java code can seamlessly co-exist and interoperate in the same program; we'll leverage this feature later in the course. In fact, many of the design patterns that you will learn in this course will be applicable to Java and Scala.

However, a key reason we're using Scala is to expose you to programming techniques and language features that are beyond the scope of Java. Most modern software systems are written in a plethora of languages. In fact, large systems tend to use several programming languages. Therefore, to succeed in your computing career, you have to be familiar with several languages and be able to learn new languages on your own. Programming languages are constantly invented and abandoned[1] and it is impossible to predict the next big language that everyone will use or the language you'll need to learn for your first job.

Scala is a big language with many unique features and we are not going to learn to use them all. Instead, we are going to focus on ideas that Scala shares with other modern programming languages. Here are some of the key ideas that we will cover in this course that go beyond Java:

- *First-class functions* are the cornerstone of *functional programming*. They are pervasive in JavaScript, Ruby, Python, Swift, and almost all modern languages.

  In fact, even Java and C++ recently added support for first-class functions.

- *Algebraic data types* are available in Apple Swift, Mozilla Rust, Microsoft F#, and several other programming languages. Programming with algebraic data types is very different from programming in an object-oriented style. We'll cover both styles of programming in this course and develop a deep understanding of the tradeoffs.

- *Type inference* is available in modern typed programming languages, such as C# and Swift, and even in a limited form in Java. As the name suggests, in a language with type inference, the compiler can often "infer" or fill-in types that you omit. So, your programs become shorter, but retain all the advantages of type checking.

The main themes of the course are not language-specific. We will emphasize the following broad ideas that are applicable to all software development:

- *Testing* is critical for building reliable software. You will learn how to test complex functions and make effective use of testing tools and frameworks. Every programming problem you solve in this course will have to be tested. We expect you to write good tests yourself. The quality of your tests will be a significant portion of your grade on every assignment.

- *Design patterns* are recipes for solving typical programming problems. This course will emphasize object-oriented and functional design patterns. We will focus on design patterns that are applicable to a variety of programming languages, and not Scala-specific design patterns.

- *Refactoring* is a key concept that we emphasize throughout the course. As we introduce new ideas, we will systematically refactor our old code to exploit them.

[1] This poster is a very incomplete history of the birth and death of programming languages.

- *Debugging* is a necessary skill because even small programs often have bugs.
- *Command-line tools* such as compilers and build tools lie under the hood of sophisticated IDEs such as Eclipse. Learning to use the command-line will make you a better IDE user. Moreover, since many new languages lack good IDEs, we will emphasize the use of command-line tools in this course.
- *Version control* software is critical for collaborative software development and used by all professional programmers. Although you will be programming alone in the course, version control will still help you organize your programming and save you a lot of time if you accidentally delete or break your code.
- *Using libraries* is critical for writing software that gets real work done. Initially, you'll use libraries that were developed specifically for this course, but you will eventually learn to discover and use libraries from the Web.

The overarching goal of this course is to make you a better programmer, and an important part of that is to get familiar with programming terminology and culture. Unfortunately, there is a lot of misinformation on the Web about programming, but we will try to point you to sources that are reliable. Here are two good places to start:

- Paul Graham's Essays. The earlier essays are particularly pertinent, E.g., Beating the Averages and Being Popular.
- Joel Spolsky's blog. E.g., Advice for Computer Science College Students and Getting Your Resume Read.

Finally, XKCD comics often make obscure programming references and this course will help you decipher some of them.

The Command-Line

The UNIX *command-line*[2] is a critical part of this course:

- If you're using the course virtual-machine, you should use the program **LXTerminal** to start the command-line.
- If you're using macOS, you you get to command line by running the launching the **Terminal** application, which is in the **Applications ▶ Utilities** folder.
- If you're using Windows 10, you can use **Bash for Windows**.
- If you're using Windows, you can install and run **Cygwin**.

Unless you're already familiar with the command-line, you must read Zed Shaw's Command Line Crash Course, up to and including the chapter "Removing a File (rm)". Zed likes to swear at his own readers, so we'd like to apologize in advance on his behalf. The rest of these lecture notes assume that you are familiar with the command-line.

`sbt` and the Scala REPL

`sbt` is the Swiss Army Knife of Scala programming. It is a command-line tool that can be use to run Scala programs, compile them, test them, package them deployment, publish them to the Web, and more. Like many modern programming languages, `sbt` has a *REPL* (read-eval-print loop), which you can use to type in and run one-line programs immediately, without the bother of creating files, etc.

To start the Scala REPL, open a terminal, type in `sbt console` and press enter. Your screen will look like this:

```
student@vm:~$ sbt console
[info] Loading global plugins from /Users/arjun/.sbt/0.13/plugins
[info] Set current project to del (in build file:/Users/arjun/scratch/del/)
[info] Updating {file:/Users/arjun/}arjun...
[info] Resolving org.fusesource.jansi#jansi;1.4 ...
[info] Done updating.
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

The `scala>` prompt indicates that you can type in Scala expressions.

[2] The command-line is also known as a *terminal* or *shell*.

Scala Basics

Simple Expressions and Names

Arithmetic in Scala is very similar to arithmetic in Java:

```
scala> 19 * 17
res0: Int = 323
```

Strings in Scala will also look familiar:

```
scala> "Hello, " + "world"
res1: String = Hello, world
```

Boolean expressions will be familiar too:

```
scala> true && false
res2: Boolean = false
```

Let's examine the last interaction more closely. When you type `true && false`, Scala prints three things:

- An automatically-generated *name* (`res2`),
- The *type* of the expression (`Boolean`), and
- The *value* of the expression (`false`).

On the Scala REPL, you can use the generated name as a variable. But, you're better off picking meaningful names yourself using `val`:

```
scala> val mersenne = 524287
mersenne : Int = 524287
scala> val courseName = "Programming Methodology"
courseName : String = Programming Methodology
```

Type Inference

You'll find that Scala programs are significantly shorter than their Java counterparts. A key feature of Scala that lets you write less code is *type inference*. Notice in the variable definitions above, you did not have to write any types. Instead, Scala *inferred* them for you. This feature is very helpful in large programs, where types can become long and complicated.

Type inference is not magic; later in the course, you'll learn more about how it works and when it doesn't. For now, here's a rule of thumb: Scala can infer the type of variable named with `val`. But, Scala *cannot* infer the type of function parameters.

Functions

Here is a very simple Scala function:

```
scala> def twice(n: Int): Int = n + n
double: (n: Int)Int
```

This code defines a function called `twice`, which takes an argument called `n` of type `Int` and returns a value of type `Int`. We can apply the function as follows:

```
scala> twice(10)
res3: Int = 20
```

The following function takes two arguments, `x` and `y` and calculates the distance from the point `(x,y)` to the origin:

```
scala> def dist(x: Double, y: Double): Double = math.sqrt(x * x + y * y)
dist: (x: Double, y: Double)Double

scala> dist(3.0, 4.0)
res4: Double = 5.0
```

Notice that unlike variable definitions, we need *type annotations* on function parameters and result types.

If your function actually fits on a line (without scrolling off your window), you can define them very tersely as shown above. But, many interesting functions span several lines and need local variables.

```
object Lecture1 {

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

}
```

Figure 1.1: A Scala module

Blocks and Local Variables

You can define local variables within a *block*. A block is code delimited by curly-braces. For example:

```
scala> def dist2(x: Double, y: Double): Double = {
  val xSq = x * x
  val ySq = y * y
  math.sqrt(xSq + ySq)
}
```

`sbt` Project Structure

In principle, you can write a full-fleged program line-by-line in the Scala console. But, it makes a lot more sense to save large programs to files for a particular *project*. To do so, we will being by creating a new directory for your project. Every time you create a new Scala program (e.g., for a homework assignment, you should create a new project.)

First, exit the Scala console by typing `:quit` and then exit `sbt` by typing `exit`. You should return to the command-line:

```
scala> :quit

[success] Total time: 3 s, completed Jan 13, 2016 8:47:38 PM

sbt> exit
student@vm:~$
```

At the terminal, let's create a directory for the project:

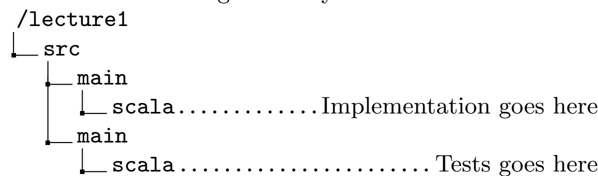
```
student@vm:~$ mkdir lecture1
```

Then, let's enter the directory:

```
student@vm:~$ cd lecture1
student@vm:~/lecture1$
```

Notice that the name of the directory is displayed on the command-line.

Your Scala projects will have two kinds of files: test cases and your implementation. `sbt` requires you to organize your files into the following directory structure:



You can create these directories by running the following commands:

```
student@vm:~/lecture1$ mkdir src
student@vm:~/lecture1$ mkdir src/main
student@vm:~/lecture1$ mkdir src/main/scala
student@vm:~/lecture1$ mkdir src/test
student@vm:~/lecture1$ mkdir src/test/scala
```

We will now see how to save Scala code to files. Using a text editor (e.g., Sublime Text), create a file called `Lecture1.scala` in the `src/main/scala` directory, with the contents shown in fig. 1.1. The code creates an object with a single function to calculate factorials.

When you write functions in a Scala file, you *have* to place it in an object. You cannot just write `def fac ...` without an enclosing object. This is a peculiarity of Scala that we will explain later in the course. In this example, the name of the object is "Lecture1", but it can be anything you like.

Now that we've saved this function to a file, we can use it from the console:

```
// src/main/scala/Lecture1Tests.scala
import Lecture1._

class Lecture1Tests extends org.scalatest.FunSuite {

  test("fac -- base case") {
    assert(fac(0) == 1)
  }

  test("fac -- inductive case") {
    assert(fac(5) == 120)
  }

}
```

Figure 1.2: Unit tests for the code in fig. 1.1

```
sbt> console
scala> import Lecture1._
scala> fac(10)
```

Testing

The `sbt` console is a convenient way to experiment with new code or write a "one off' functionn. However, you must write *unit tests* to test any actual code you write. Figure 1.2 shows an example of unit tests that use the *ScalaTest* library. The code is quite self-explanatory: each test suite is a class that extends `org.scalatest.FunSuite`. The body of the class has several test blocks, as shown in the figure.

Building and Pattern-Matching on Lists

In this section, we will show you how to write simple list-processing functions. We will cover basic *functional programming* and introduce *pattern matching*. You should try out all the code yourself using the `sbt` console.

Constructing Lists

The simplest list is the empty list, which is written in Scala as

```
Nil
```

Given the empty list, we can construct larger lists using the `::` operator (which is pronounced *cons*). Here is a simple example that constructs a one-element list:

```
50 :: Nil
```

Given a one-element list, we can build a two-element list by using the `::` operator again:

```
100 :: (50 :: Nil)
```

We can use `::` again to build a three-element list:

```
200 :: (100 :: (50 :: Nil))
```

In an expression `x :: y`, `x` is known as the *head* of the list and `y` is known as the *tail*. Note that the tail of a list is always a list itself, (though it may be the empty list `Nil`).

For example, consider the list below:

```
val letters = "a" :: ("b" :: :: Nil)
```

- The head of `letters` is `"a"`.
- The tail of `letters` is `"b" :: Nil`.
- The head of the tail of `letters` is `"b"`.
- The tail of the tail of `letters` is `Nil`.

```
def countDown(n: Int): List[Int] = {
  if (n == 0) {
    Nil
  }
  else {
    n :: countDown(n - 1)
  }
}
```

(a)

```
def fromTo(lo: Int, hi: Int): List[Int] = {
  if (lo == hi) {
    lo :: Nil
  }
  else {
    lo :: fromTo(lo + 1, hi)
  }
}
```

(b)

Figure 1.3: Functions that produce lists.

- `Nil` does not have a head or a tail.

In our examples so far, we've used parenthesis to make the head and tail clear. However, you can simply write `"a" :: "b" :: "c" :: Nil`. Intuitively, everything to the right of a `::` is the tail. If you get confused, write the parenthesis explicitly.

It is usually easier to write lists in the following way:

- `List("a", "b", "c")` is equivalent to `"a" :: ("b" :: ("c" :: Nil))`.
- `List()` is equivalent to `Nil`.

However, it is important to understand that this is just a convenient notation. Under the hood, Scala transforms these expressions to use `::` and `Nil`, as we described above.

Lists and Type Inference Consider the following interaction, which you should try yourself:

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

As you can see, Scala prints lists using the shorthand notation, even if you explicitly use `::` and `Nil`. More significantly, Scala has inferred that the type of the list is `List[Int]`. There was no need to explicitly state that is is the case.

Here is another example, where Scala infers that the type of a list is `List[String]`:

```
scala> val lst = List("a", "b", "c")
lst: List[String] = List("a", "b", "c")
```

Type inference is very convenient and spares you from having to explicitly specify the type of the element. However, type inference is not magic and can behave in unexpected ways. For example, in the interaction below, Scala infers that the type of the list is `List[Any]`:

```
scala> val lst = List("a", 10, "c")
lst: List[Any] = List("a", 10, "c")
```

Although this is technically true, if you write this code, it is more likely that you made a mistake and intended to actually create a list of strings. If you're ever unsure, you can write the type explicitly, which would signal a type error in this case:

```
scala> val lst = List[String]("a", 10, "c")
<console>:10: error: type mismatch;
 found   : Int(10)
 required: String
       val lst = List[String]("a", 10, "c")
                                   ^
```

Functions that produce lists Now that we've seen how to construct lists explicitly, it is straightforward to write functions that produce lists. Figure 1.3 shows some simple recursive functions that construct new lists.

Pattern Matching

Now that we've seen how to write functions that produce lists, we'll learn how to write functions that consume lists as arguments. We'll start by writing a simple function to calculate the sum of a list of numbers. Here are some examples of of `sum` being used:

```
def product(alist: List[Int]): Int = {
  alist match {
    case Nil => 1
    case n :: tail => n * product(tail)
  }
}
```

(a) Calculate the product of a list of numbers.

```
def repeatTwice(alist: List[Int]): List[Int] = {
  alist match {
    case Nil => Nil
    case n :: tail => n :: n :: repeatTwice(tail)
  }
}
```

(b) Repeats every element of a list twice.

Figure 1.4: Two simple functions that consume lists.

```
def countOnes(alist: List[Int]): Int = {
  alist match {
    case Nil => 0
    case n :: tail => {
      if (n == 1) {
        1 + countOnes(tail)
      }
      else {
        countOnes(tail)
      }
    }
  }
}
```

(a) Counting ones using an if-expression.

```
def countOnes(alist: List[Int]): Int = alist match {
  case Nil => 0
  case 1 :: tail => 1 + countOnes(tail)
  case n :: tail => countOnes(tail)
}
```

(b) Counting ones using a composite pattern.

Figure 1.5: Pattern matching can make complex conditionals clearer.

```
assert(sum(20 :: 30 :: Nil) == 50)
assert(sum(1 :: 2 :: 3 :: Nil) == 6)
```

Intuitively, to calculate `sum(1 :: 2 :: 3 :: Nil)`, we can can recursively calculate the sum of the tail and add that value to the head:

```
   sum(1 :: 2 :: 3 :: Nil)
== 1 + sum(2 :: 3 :: Nil)
== 1 + (2 + sum(3 :: Nil))
== 1 + (2 + (3 + sum(Nil)))
```

The last line shows an important special case. Since the empty list doesn't have a head or a tail, we need to treat it differently. We'll say that `sum(Nil)` is `0`.

We can write `sum` by using a powerful feature of Scala called *pattern matching*.

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

This code makes it clear that the function is inspecting `alist` and considering two *cases*. When `alist` is `Nil`, it produces `0` and when alist is constructed with the `::` operator, the function recurs on the tail and ands that result to the head. In this code, `Nil` and `h :: t` are called *patterns*.

Figure 1.4 shows two more list-consuming functions that use pattern-matching. Notice that in `::` cases, these functions use different variable names for the head and the tail. A pattern can use any variable to refer to the value of the head or the tail.

Complex Patterns Pattern-matching is extremely powerful and can be used to express complex conditionals. For example, fig. 1.5a is a function that counts the number of `1`s that occur in a list: it uses pattern matching as introduced above and then an if-expression to check if the head of the list is `1`.

Figure 1.5b is the same function, rewritten to use pattern matching. This version is shorter and makes it clear that there are three cases. To do so, we exploit the fact that patterns can match *almost* any value, including numbers, strings, lists, and user-defined data structures too (which we will see in a later class).

Exhaustivity and Reachability Checking Here is another function that uses pattern matching to count the number of tens in a list:

```
def countTens(alist: List[Int]): Int = alist match {
  case 10 :: tail => 1 + countTens(tail)
  case n :: tail => countTens(tail)
}
```

However, this function had a bug. Do you see it? If you type this into the console, Scala prints the following:

```
<console>:10: warning: match may not be exhaustive.
It would fail on the following input: Nil
       def countTens(alist: List[Int]): Int = alist match {
                                                ^
```

Scala has detected that we forgot to write a case for `Nil`. Scala ensures that your patterns are *exhaustive*. Here is another buggy version of the function:

```
def countTens(alist: List[Int]): Int = alist match {
  case n :: tail => 1 + countTens(tail)
  case n :: tail => countTens(tail)
  case Nil => 0
}
```

In this version, we wrote the patterns incorrectly, so the first and second patterns are identical. Scala reports the following error:

```
<console>:13: warning: unreachable code
         case n :: rest => countTens(rest)
```

Scala ensures that all cases are *reachable*.

This automatic exhaustivity and reachability checking makes programs that use pattern-matching much more robust than programs that use complicated, nested if-statements. Pattern matching is a very powerful tool that you can exploit to make your programs more robust. We will emphasize pattern-matching over conditionals in this course.

Immutability

We want to emphasize that all that functions that we've written to produce lists *produces a new list* and leaves the original list unchanged.

You can observe this behavior in the Scala console:

```
scala> val original = List(10, 20, 30)
original: List[Int] = List(10, 20, 30)

scala> val result = repeatTwice(original)
result: List[Int] = List(10, 10, 20, 20, 30, 30)

scala> original
res0: List[Int] = List(10, 20, 30)

scala> result
res1: List[Int] = List(10, 10, 20, 20, 30, 30)
```

At no point did we change or *mutate* the original list. In fact, lists in Scala are *immutable* data structures. There is no way to update their head or tail. Programming with immutable data structures is a key part of *functional programming*, which we will emphasize in the first half of this course.