Notes on Scala for Java Programmers

H. Conrad Cunningham

25 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi

214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Note: In Spring 2016, I created these notes by adapting and expanding the Web document "A Scala Tutorial for Java Programmers" by Michel Schinz and Phillipp Haller [7] from the Scala language website:

https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html

I sought to better meet the needs of the students in my Scala-based courses: to explain several issues more thoroughly, improve the narrative, make the document accessible, and integrate it with my other Scala notes. See the Acknowledgements section for more information.

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good of April 2022 is a recent version of Firefox from Mozilla.

1 Scala for Java Programmers

1.1 Introduction

This is an introduction to Scala for programmers who have completed an introductory computer science course sequence using Java.

This document adapts and expands the Web document A Scala Tutorial for Java Programmers by Michel Schinz and Phillipp Haller [7] to better meet the needs of the students in my Scala-based courses: to explain several issues more thoroughly, improve the narrative, make the document accessible (to individuals using screen readers), and better integrate it with my other Scala and programming language notes.

The goal of this document is to provide a quick overview of Scala for programmers already familiar with Java. To learn more, use a textbook, reference book, other tutorials, and manuals.

1.2 A First Example: Hello World

A "Hello, world!" program is the obligatory first example to give when introducing a new language. We can write a program `HelloWorld` as follows in Scala:

```
object HelloWorld {
    def main(args: Array[String]): Unit = {
        println("Hello, world!")
    }
}
```

Note: The Scala source code for the `HelloWorld` program is in file `HelloWorld.scala`.

What Scala features do we use above in relation to Java?

- Keyword `object` declares a *singleton object* named `HelloWorld`. An `object` is essentially a class with a single instance. The body of the `object` is enclosed in braces following the name.
- The keyword `def` introduces a method definition.
- In a declaration, a colon (`:`) separates the name from its type.
- Method `main` takes the command line arguments as its parameter, which is an array of strings.
- The `main` method is a procedure and, hence, has return type `Unit` declared (meaning no return type). The body of the method is enclosed in braces following the method header.
- The `main` method is not declared as `static` as in Java. Static members do not exist in Scala. We can use singleton objects instead.

- The body of `main` has a single call to predefined method `println`.

1.2.1 Compiling the example

At the command line, we can use the `scalac` command (similar to the `javac` command) to invoke the Scala compiler. If the above Scala program is stored in file `HelloWorld.scala`, we can compile it from the command line as follows:

```
> scalac HelloWorld.scala
```

The above compiles the Scala source file and generates a few class files in the current directory.

File `HelloWorld.class` contains a class that can be executed.

1.2.2 Running the example

We can use the `scala` command (similar to the `java` command) to execute the `main` method. Execution of the program prints the "Hello, World" string to the console.

```
> scala -classpath . HelloWorld

Hello, world!
```

1.3 Interaction with Java

Scala code can interact with Java code. Package `java.lang` is imported by default and other packages can be imported explicitly.

Consider a program to obtain and format the current date according to the conventions used in a specific country, say France.

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
    def main(args: Array[String]): Unit = {
        val now = new Date
        val df = getDateInstance(LONG, Locale.FRANCE)
        println(df format now)
    }
}
```

Note: The Scala source code for the `FrenchDate` program is in file `FrenchDate.scala`. The source file `MoreDates.scala` expands the `FrenchDate` program to include Chinese, Portuguese, Hebrew, and Arabic date formats.

Java's class libraries such as `Date` and `DateFormat` can be called directly from Scala code.

The Scala `import` statement is more powerful than Java's.

- It can import multiple classes by enclosing names in braces.

  The first line above imports the `Data` and `Locale` classes from Java package `java.util`.

- It can import everything in a package or class by using an underscore character (`_`) instead of a Java's asterisk (`*`).

  The third line imports all members of the `DateFormat` class, making static method `getDateInstance` and static field `LONG` visible.

The `main` method in the `FrenchDate` object:

- creates an instance of Java's `Date` class, thus getting the current date
- uses the imported static method `getDateInstance` to format the date appropriately for France using `Locale.France`
- prints the current date formatted according to the localized `DateFormat` instance

The `main` method declares two local "variables" in its body: `now` and `df`.

- The "variables" are *lexically scoped*. They are only visible from the point of declaration to the end of the function body. They are not visible outside the body of the function.
- Both are declared with `val`. After initialization, Scala *does not allow* further assignments to a `val`.

  The alternative is `var`. Scala *does allow* assignments to a `var`; it is like an ordinary variable in Java.

  Although we cannot change the binding of a `val` to an object, Scala does allow the internal state of the object to be changed.

- Although Scala is statically typed like Java, neither of these variables are given explicit types. The types of the variables are *inferred* [11] by the compiler from the type of the initializing expression.

We suggest the following Programming Guideline: Declare a Scala variable with `val` by default. Only change the declaration to `var` if you explicitly decide to allow the variable binding to change.

Scala methods taking one explicit argument can be written in infix syntax such as

```
df format now
```

which is the same as the method call:

```
df.format(now)
```

This feature has important consequences, as we discuss below.

It is also possible to inherit from Java classes and implement Java interfaces directly in Scala.

1.4 Everything is an Object

In these notes, we use the concepts and terminology from the textbook *Exploring Languages with Interpreters and Functional Programming* (ELIFP) [1] to characterize programming languages and styles. In particular, we use the *object model* described in Chapter 3 [1:3.2].

Like Java, Scala's objects exhibit the three essential characteristics of objects in the object model:

a. *state*

   A Scala object has (instance) variables that have values. The mapping of an object's variables to their values forms the state of the object. The value of a variable cannot spontaneously. The value is either constant or can be changed by an operation.

b. *operations*

   A Scala object can have methods that access and possibly change the state of the object.

c. *identity*

   Scala objects are distinct from each other. In particular, each object has a unique address in memory.

Like Java, Scala's objects also exhibit the two important but nonessential characteristics of objects in the object model:

d. *encapsulation*

   The state of a Scala object can be hidden inside the object, making it inaccessible from outside. This is done by declaring the variables `private`.

e. *independent lifecycle*

   A Scala object exists independently of the program unit that created it. It

Thus, in the terminology of the object model [1, Ch. 3], Scala is a pure *object-based* language.

Everything is an object, including numbers and functions. Scala does not distinguish the Java primitive types (e.g. `Boolean` and `Int`) from reference types (e.g. objects). It also enables us to manipulate functions as *first-class* values.

1.4.1 Numbers are objects

Since numbers are objects, they also have *methods* (i.e., operations). For example, the expression

```
1 + 2 * 3 / x
```

is equivalent to the expression

```
(1).+(((2).*(3))./(x))
```

which shows the method calls explicitly.

In Scala, the "operator" symbols (e.g., `+`, `*`, `/`) are valid identifiers. The parentheses around numbers are necessary because Scala's lexical analyzer uses a longest-match rule for tokens, breaking expression

```
1.+(2)
```

into the tokens `1.`, `+`, and `2`. This results in `1.` being interpreted as a `Double`.

1.4.2 Functions are objects

Because Scala functions are objects, we can:

- store a function in a variable or data structure

  That is, a Scala function is a *first-class* function value [13].

- pass a function as an argument to or return a function as a result from another function

  That is, a Scala function may be a *higher-order* function [14].

These are key features of *functional programming* [15].

Consider the `Timer` program below. It includes a timer function named `oncePerSecond` that performs some action every second. The specific action performed is encoded as a *call-back* function passed into `oncePerSecond` as an argument.

```
object Timer {
    def oncePerSecond(callback: () => Unit): Unit = {
        while (true) { callback(); Thread sleep 1000 }
    }
    def timeFlies(): Unit = {
        println("time flies like an arrow...")
    }
    def main(args: Array[String]): Unit = {
        oncePerSecond(timeFlies)
    }
}
```

Note: The Scala source code for the `Timer` program is in file `Timer.scala`.

The type of the call-back function is `() => Unit`. This means it is a function that takes no arguments and returns nothing. The type `Unit` is similar to `void` in C/C++.

Function `oncePerSecond` calls the `Thread.sleep` method (from `java.lang`) and uses an infinite `while` loop to repeat the call-back action every second.

The `main` function calls this timer function, passing a call-back function that prints the string

```
time flies like an arrow...
```

on the console. The program endlessly prints this string every second.

The program uses the predefined Scala method `println` instead the like-named Java method from `System.out`.

1.4.3 Anonymous functions

The `Timer` programa in Section 1.4.2 (above) can be refined by replacing the function `timeFlies` by an *anonymous function* [12] as shown below:

```
object TimerAnonymous {
    def oncePerSecond(callback: () => Unit): Unit = {
        while (true) { callback(); Thread sleep 1000 }
    }
    def main(args: Array[String]): Unit = {
        oncePerSecond(() =>
            println("time flies like an arrow..."))
    }
}
```

Note: The Scala source code for the `TimerAnonymous` program is in file `TimerAnonymous.scala`.

In the anonymous function, a right arrow `=>` separates the function's argument list from its body expression.

In this example, the argument list is empty, shown by the empty pair of parentheses on the left.

1.5 Classes

In the terminology of the object model [1:3.2], both Java and Scala are *class-based languages*. A class is a template or factory for creating objects. The class concept describes a collection of related objects (i.e., instances of the class) that have a common set of operations and set of possible states. The class defines a type in the language.

Unlike Java, Scala class definitions can have parameters. Consider the following definition of class `Complex` for representing complex numbers:

```
class Complex(real: Double, imaginary: Double) {
    def re() = real
    def im() = imaginary
}
```

The `Complex` class takes two arguments, the real and imaginary parts of the complex number. In the `Complex` class, these become `val` fields of the object (i.e., class instance) that are only visible and accessible from *inside the object*. (This visibility is `private[this]`, a type of visibility that does not occur in Java.)

The default *constructor* is built into the `class` syntax. We pass values for these arguments when we create an *instance* of class `Complex`, as follows:

```
new Complex(1.5, 2.3)
```

This causes all statements in the class definition to be executed.

The `Complex` class defines function methods `re` and `im`, which provide access to the two parts of the complex number. That is, these are *accessor*, or *getter*, methods.

The methods `re` and `im` are *public* by default, as in Java. That is, they can be called from either outside or inside the class. If a method (or variable) is declared `private`, it is only accessible from inside the class, as in Java.

We declare a function with an = between the function's header (i.e., name and parameter list) and its body. The body is an expression that is evaluated when the function is called. If the body itself consists of a sequence of expressions, we enclose it in braces. The value of last expression executed is the value of the body.

In `Complex`, the compiler infers the return types for methods `re()` and `im()` by examining the right-hand sides and deducing that both return a value of type `Double`. The compiler gives an error message when it cannot infer the type of a method or variable.

Some Scala programmers suggest that the types be omitted whenever possible and only inserted when necessary.

However, the author of these notes considers it better software engineering practice to explicitly specify the types of all *public* features of a class or package such as the `re()` and `im()` methods.

But, for internal features of a class or method, it is convenient and safe to use type inference. For example, we did not give explicit types for the `now` and `df` values in the `FrenchDate` object shown in Section 1.3.

We could do that, because Scala allows method definitions in case classes just as in normal classes.

1.5.1 Methods without arguments

To call the methods `re` and `im`, we must put an empty pair of parentheses after their names:

```
object ComplexNumbers {
    def main(args: Array[String]): Unit = {
        val c = new Complex(1.2, 3.4)
        println(s"imaginary part:  ${c.im()}")
    }
}
```

In this example, we pass an *interpolated string* as the argument to `println`. The feature

```
s"imaginary part:  ${c.im()}"
```

denotes a string with the value of the expression `c.im()`, converted to a string using `toString`, and inserted in place of the term `${c.im()}`.

We can eliminate the empty parameter list by defining `re` and `im` as methods *without arguments* instead of *with zero arguments*. We simply omit the parentheses in the method definition, as shown below:

```
class Complex(real: Double, imaginary: Double) {
    def re = real
    def im = imaginary
}
```

1.5.2 Inheritance and overriding

Like Java, every Scala class inherits from some *superclass*. If no super class is given explicitly, then the superclass is `scala.AnyRef` by default.

Like Java, a Scala class inherits all methods from its superclass by default.

Like Java, a Scala class may *override* individual methods by giving new definitions.

Unlike Java, a Scala method that overrides a superclass method must be explicitly declared with the `override` modifier. This is to reduce the possibility of class overriding a superclass method by accident.

For example, the `Complex` class can redefine the `toString` method inherited from `AnyRef`.

```
class Complex(real: Double, imaginary: Double) {
    def re = real
    def im = imaginary
```

```
    override def toString() =
        s"${re}${(if (im < 0) "" else "+")}${im}i"
}
```

The new definition of `toString()` overrides the definition for `Complex` objects. However, in doing so, it uses the default definitions of `toString()` for `Double` by its references to `re` and `im` in the string concatenation.

Note: The Scala source code for the `ComplexNumbers` programs using the three different definitions of the class `Complex` are in files `ComplexNumbers.scala`, `ComplexNumbers2.scala`, and `ComplexNumbers3.scala`.

1.6 Hierarchical Data Structures and Polymorphism

In programming, we often use trees and other hierarchical data structures.

We can illustrate how to implement a tree in Scala using a small calculator program for simple arithmetic expressions composed of sums, integer constants, and variables. Examples of such expressions are `1+2` and `(x+x)+(7+y)`.

We can represent expressions naturally with a tree, where nodes represent operations (e.g., addition) and leaves represent values (e.g., constants or variables).

1.6.1 "Traditional" object-oriented program

In both Java and Scala, we can represent an expression tree using an *abstract* superclass for trees and a *concrete* subclass for each kind of node (i.e., subtree).

The abstract class defines the common interface for all tree nodes. It gives the name, parameter names and types, and return value type for each abstract method but does not give a body. It may also have concrete methods that can be fully defined at that level.

The concrete subclasses define the specific features for each kind of node. They must give concrete definitions for each abstract method.

In Scala, we can define a skeleton of this structure as shown below. Subclasses `Sum`, `Var`, and `Const` all inherit from superclass `Tree`, as denoted by the keyword `extends`. It might also be useful to override builtin functions such as `toString` in the classes of the hierarchy.

```
abstract class Tree {
 def eval(env: Environment): Double
 // ... other abstract methods
}
class Sum(l: Tree, r: Tree) extends Tree {
  def eval(env: Environment) = l.eval(env) + r.eval(env)
  // other concrete Tree method definitions
}
class Var(n: String) extends Tree {
```

```
  def eval(env: Environment) = env(n)
  // other concrete Tree method definitions
}
class Const(v: Double) extends Tree {
  def eval(env: Environment) = v
  // other concrete Tree method definitions
```

Parameters `l` and `r` of the `Sum` class constructor are *polymorphic*. That is, they can can take an argument of type `Tree` or any of its descendants (e.g., objects from subclasses `Sum`, `Var`, and `Const`).

The method `eval` in class `Sum` evaluates the left subtree `l` and right subtree `r` in some environment and then returns their sum. The call `l.eval(env)` is *dynamically bound* at runtime to the `eval` method associated with the actual concrete type of the object stored in instance variable `l`. Similarly, for `r.eval(env)`.

As we discuss in detail in Section {#sec:case-classes} (below), the `Environment` parameter `env` is an object that provides a mapping between the names of the variables and their values.

In the terminology used in the ELIFP textbook [1:5.2], the kind of polymorphism exhibited by this example is called *subtype polymorphism* (or *subtyping*, *inclusion polymorphism*, or *polymorphism by inheritance*) [10]. However, in object-oriented languages such as Java and Scala, this kind of polymorphism is usually simply called *polymorphism* without any qualifier.

A language with objects, classes, inheritance, and subtype polymorphism is called an *object-oriented* language [1:3.2]. Both Java and Scala are object-oriented. However, Scala is sometimes called a pure object-oriented language in that everything is an object.

1.6.2 Case classes and pattern matching

In functional programming languages such as Haskell, we can define an *algebraic data type* [8] for hierarchical data like the expression trees. These types often enable us to express programs concisely by using pattern matching constructs.

Scala combines the concepts of classes and algebraic data types into its *case classes*. Consider the following example:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Here, we define `Sum`, `Var` and `Const` as case classes. These differ from from standard classes in several ways:

- We can create an instance without using the keyword `new`. For example, we can write `Const(5)` instead of `new Const(5)`.
- The compiler automatically generates getter functions for the constructor parameters. That is, it is possible to get the value of the `v` constructor parameter of some instance `c` of class `Const` just by writing `c.v`.
- The compiler supplies default definitions for the methods `equals` and `hashCode`. These work on the *structure* of the instances and not on their identities.
- The compiler provides a default definition for method `toString`, which prints the value in a "source" form. For example, the tree for expression `x+1` prints as `Sum(Var(x),Const(1))`.
- Instances of these classes can be decomposed through *pattern matching* as we see below.

These features enable us to use Scala case classes much like we would use the algebraic data types in a functional language such as Haskell.

To explore the use of case classes, again consider a function to evaluate an expression in some *environment*. The purpose of an environment is to associate values with variables.

For example, the expression `x+1` evaluated in an environment that associates the value `5` with the variable `x`, written `{ x -> 5 }`, gives `6` as result.

An environment is just a function that associates a value with a (variable) name. The environment `{ x -> 5 }` given above can be written as a Scala function as follows:

```
{ case "x" => 5 }
```

This notation defines a function that, when given the string `"x"` as argument, returns the integer `5`, and, otherwise, fails and throws an exception.

An environment is a function of type `String => Int`. To simplify our evaluation program, we define the name `Environment` to be an alias for this type using the following declaration:

```
type Environment = String => Int
```

We can now define the evaluation function in Scala as follows:

```
def eval(t: Tree, env: Environment): Int = t match {
    case Sum(l, r) => eval(l, env) + eval(r, env)
    case Var(n)    => env(n)
    case Const(v)  => v
}
```

This evaluation function performs *pattern matching* on the tree `t` (i.e., using the `match` operator).

1. The pattern match first checks whether the tree `t` is a `Sum` object (i.e., an instance of the `Sum` case class).

   If the tree is a `Sum`, the pattern match binds the left subtree to a new variable `l` and the right subtree to a new variable `r`, then evaluates the expression following the arrow `=>`.

   This expression uses the values of the variables bound by the pattern match, i.e., `l` and `r` in the `Sum` case.

2. If the first check does not succeed, that is, the tree is not a `Sum` object, the pattern match then checks whether `t` is a `Var` object.

   If the tree is a `Var`, the pattern match binds the name contained in the `Var` node to a new variable `n` and then evaluates the right-hand-side expression.

3. If the second check also fails, that is, if `t` is neither a `Sum` nor a `Var`, the pattern match then checks whether the expression is a `Const` object.

   If the tree is a `Const`, the pattern match binds the value contained in the `Const` node to a new variable `v` and then evaluates the right-hand-side expression.

4. Finally, if all checks fail, the pattern match expression raises an exception to signal failure. In this version of `eval`, failure occurs only when there are additional subclasses of `Tree` that we have declared but not yet defined in the `eval` pattern match.

Pattern matching attempts to match a value to a series of patterns. As soon as a pattern matches (moving top to bottom, left to right in the source code), the program extracts and names various parts of the value and then evaluates the associated expression using the values of these named parts.

An object-oriented programmer might ask why we do not use a more `<"traditional" approach like the one in Section [@sec:traditional-oo-tree} (above). That is, we could define`eval`{.scala} as a *method* of class`Tree‘{.scala} and its subclasses.

We could do that, because Scala allows method definitions in case classes just as in normal classes.

We could do that, because Scala allows method definitions in case classes just as in normal classes.

Deciding whether to use pattern matching or methods is partly a matter of taste. But the choice also affects the extensibility the program.

- Using methods, we can easily add a new kind of node by defining a new subclass of `Tree`. But adding a new operation to manipulate the tree is tedious because it requires us to modify every subclass of `Tree`.
- Using pattern matching, the situation is reversed: adding a new kind of node requires us to modify all functions that do pattern matching on the

tree, to take the new node into account. But adding a new operation is easy, we just define it as an independent function.

To explore pattern matching further, consider another operation on arithmetic expressions: symbolic derivation. Looking back at our calculus class, we see the following rules for differentiation:

1. The derivative of a sum is the sum of the derivatives.
2. The derivative of some variable `v` is 1 if `v` is the variable relative to which the derivation takes place, and is 0 otherwise.
3. The derivative of a constant is 0.

We can directly translate these rules into a Scala function that uses the above case classes and pattern matching, as follows:

```
def derive(t: Tree, v: String): Tree = t match {
    case Sum(l, r)          => Sum(derive(l, v), derive(r, v))
    case Var(n) if (v == n) => Const(1)
    case _                  => Const(0)
}
```

Function `derive` introduces two new concepts related to pattern matching.

1. The `case` expression for variables has a *guard*, an expression following the `if` keyword. This guard prevents pattern matching from succeeding unless its expression is true.

   Here the guard ensures that the function returns the constant `1` only if the name of the variable being derived is the same as the derivation variable `v`.

2. A pattern can include a *wildcard*, written `_`, that matches any value, without giving it a name.

Consider an example with a simple `main` function that performs several operations on the expression `(x+x)+(7+y)`.

It first computes its value in the environment

```
{ x -> 5, y -> 7 }
```

and then computes its derivative relative to `x` and then to `y`.

```
def main(args: Array[String]): Unit = {
    val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
    val env: Environment = { case "x" => 5 case "y" => 7 }
    println("Expression: " + exp)
    println("Evaluation with x=5, y=7: " + eval(exp, env))
    println("Derivative relative to x:\n " + derive(exp, "x"))
    println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

Executing this program, we get the expected output:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

The result of the derivative is complex. It should be simplified before printing. Defining a basic simplification function using pattern matching is an interesting (but surprisingly tricky) problem, left as an exercise for the reader.

1.6.3 Expression tree code

The Scala source code for the case class version of the Expression Tree program is in file `ExprCase.scala`.

The traditional object-oriented version of the Expression Tree program is in file `ExprObj.scala`.

1.7 Traits

In addition to inheriting code from a superclass, a Scala class can also reuse code from one or several *traits*.

From a Java perpective, traits can be viewed as interfaces that can also contain code. In Scala, when a class inherits from a trait, it implements that trait's interface, and inherits all the code contained in the trait.

Note: In older Java versions, an interface consisted of just method signatures and constants. In Java 8+, interfaces can now include default definitions of methods, somewhat similar to Scala traits.

1.7.1 Ordered objects example

To see the usefulness of traits, consider a classic example: ordered objects. We want to compare objects of a given class among themselves. For example, we need to compare objects according to some total order to sort them.

In Java, objects that can be compared implement the `Comparable` interface.

In Scala, we can do better by defining the equivalent of `Comparable` as a trait, which we call `Ord`.

When comparing objects, six different comparison operations are useful: smaller, smaller or equal, equal, not equal, greater or equal, and greater.

But it is tedious to define all of these for every class whose instances we wish to compare.

We observe that we can define four of the six in terms of the other two. For example, given the equal and smaller comparison operators, we can define the other four comparison operators.

In Scala, we can capture this observation in the following trait declaration:

```
trait Ord {
    def < (that: Any): Boolean
    def <=(that: Any): Boolean =  (this < that) || (this == that)
    def > (that: Any): Boolean = !(this <= that)
    def >=(that: Any): Boolean = !(this < that)
}
```

This definition both creates a new type called `Ord`, which plays the same role as Java's `Comparable` interface, and generates default implementations of three comparison operators in terms of a fourth, abstract operator. All classes inherit the equality and inequality operators and, thus, those operators do not need to be defined in `Ord`.

The type `Any` used above, is the supertype of all other types in Scala. It is essentially a more general version of Java's `Object` type that is also a supertype of basic types like `Int`, `Float`, etc.

Note: See discussion of Scala's unified types [6] for more information on the type hierarchy.

To make objects of a class comparable, it is sufficient to define equality and inferiority operators and then "mix in" the `Ord` trait.

For example, consider a `Date` class representing dates in the Gregorian calendar. Such dates are composed of a day, a month, and a year, each of which we can represent with an integer. We can start the definition of the `Date` class as follows:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
    def year  = y
    def month = m
    def day   = d
    override def toString(): String = s"$year-$month-$day"
    // ... equals methods defined below
}
```

The `extends Ord` declaration specifies that the `Date` class inherits from the `Ord` trait (in addition to the default superclass).

To make the comparisons work correctly, we redefine the `equals` method, inherited from Java's `Object` class, to compare dates by comparing their individual fields. The default implementation of `equals` does not work because it compares objects physically. In Scala, the comparison becomes:

```
override def equals(that: Any): Boolean =
    that.isInstanceOf[Date] && {
```

```
        val o = that.asInstanceOf[Date]
        o.day == day && o.month == month && o.year == year
    }
```

1.7.2 Aside on equality

Scala defines method `equals` in the top-level class `Any`. The comparison `x.equals(y)` yields `true` if and only if the values of `x` and `y` are equivalent. Method `equals` can be overriden in a subclass. Thus, when we redefine `equals`, we must ensure that the new definition is an *equivalence relation*. That is, it should be:

- *reflexive* (`x.equals(x)`)
- *symmetric* (`x.equals(y)` equals `y.equals(x)`)
- *transitive* (if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`)

Scala defines method `eq` in the class `AnyRef`, which is the top-level class for *reference objects*, but a subclass of `Any`. The comparison `x.eq(y)` yields `true` if and only if `x` and `y` are identical references (pointers). That is, `x` and `y` denote the same physical object.

By default, Scala defines `equals` to be the same as `eq`.

Unlike Java, Scala defines `x == y` equivalent to

```
if (x eq null) // i.e., x.eq(null)
    y eq null
else
    x equals y // i.e., x.equals(y)
```

and defines `x != y` equivalent to `!(x == y)`.

Thus, by redefining `equals`, we redefine == and != as well.

1.7.3 Ordered objects continued

The redefined `equals` method for the `Date` class above uses the predefined methods `isInstanceOf` and `asInstanceOf`.

- `isInstanceOf` corresponds to the Java `instanceof` operator. It returns true if and only if the object to which it is applied is an instance of the given type.
- `asInstanceOf` corresponds to the Java cast operator: if the object is an instance of the given type, it is viewed as such, otherwise it throws a `ClassCastException`.

To complete our implementation, we need to define the < operator. It uses the predefined Scala method `sys.error` to throw an exception with the given error message.

```
    def <(that: Any): Boolean = {
        if (!that.isInstanceOf[Date])
            sys.error(s"Cannot compare $that and a Date")

        val o = that.asInstanceOf[Date]
        (year < o.year) ||
        (year == o.year && (month < o.month ||
                            (month == o.month && day < o.day)))
    }
```

This completes the definition of the `Date` class.

Instances of the `Date` class can be seen either as dates or as comparable objects. They define all six comparison operators:

- `equals` (i.e., ==) and < because they appear directly in the definition of the `Date` class
- `!=`, which is defined in terms of `equals`
- the other three because they are inherited from the `Ord` trait

Traits are useful in many other situations, as you will see as you program more in Scala.

Note: The Scala source code for the `OrderedDateTest` program is in file `OrderedDateTest.scala`.

1.8 Genericity

Scala supports generics. Java did not support generics until Java 5.

Genericity is the ability to write code parameterized by types. (In the terminology used in the ELIFP textbook [1:5.2], this form of polymorphism is called [*parametric polymorphism*]{https://en.wikipedia.org/wiki/Parametric_polymorphism) [9].)

For example, suppose we are writing a library for linked lists. What type do we give the elements of the list?

- If we choose a specific concrete type such as `Int`, we severely limit the usefulness of the library. It is impractical to include different implementation for every possible concrete type.
- If follow choose a default supertype like `Any`, then users of our library would need to their lace code with many type checks and type casts. (In Java, there was also the problem that basic types do not inherit from `Object`.)

Scala supports generic classes and methods to solve this problem. As an example, consider the simplest possible container class: a reference, which can either be empty or point to an object of some type.

```
class Reference[T] {
    private var contents: T = _
    def set(value: T) { contents = value }
    def get: T = contents
}
```

This example parameterizes the class `Reference` with a type `T`, which is the type of its element. The body of the class declares the `contents` variable, the argument of the `set` method, and the return type of the `get` method to have type `T`.

This is our first example to use mutable variables declare with `var`. In this example, we initialize the `contents` variable to have the value `_`, which represents the default value for the type. This default value is 0 for numeric types, `false` for the `Boolean` type, `()` for the `Unit` type, and `null` for all object types.

To use this `Reference` class, we need to specify what type to use for the type parameter `T`, that is, the type of the element contained in the cell. For example, to create and use a cell holding an integer, one could write the following:

```
object IntegerReference {
    def main(args: Array[String]): Unit = {
        val cell = new Reference[Int]
        cell.set(13)
        println(s"Reference contains the half ${cell.get * 2}")
    }
}
```

This example does not need to cast the value returned by the `get` method before using it as an integer. It is also not possible to store anything but an integer in that particular cell, because it was declared as holding an integer.

Note: The Scala source code for the `IntegerReference` program is in file `IntegerReference.scala`.

1.9 Conclusion

The goal of this document is to provide a quick overview of Scala for programmers already familiar with Java. To learn more, use a textbook, reference book, other tutorials, and manuals.

The following online overviews, cheat sheets, and tutorials might also be useful:

- Tour of Scala [5]
- Learn Scala in Y Minutes [3]
- Concise Java to Scala [2]

1.10 Source Code Recap

The Scala source code for the programs in these notes are as follows:

- `HelloWorld.scala`
- `FrenchDate.scala` and `MoreDates.scala`
- `Timer.scala` and `TimerAnonymous.scala`
- `ComplexNumbers.scala`, `ComplexNumbers2.scala`, and `ComplexNumbers3.scala`
- `ExprCase.scala` and `ExprObj.scala`
- `OrderedDateTest.scala`
- `IntegerReference.scala`

1.11 Exercises

TODO: More exercises

1. Extend the case class version of the expression-tree calculator program as described below.
   a. Modify the `Tree` algebraic data type as follows:
      - change node type `Const` to represent a floating point number instead of an integer.
      - add node types `Sub`, `Prod`, and `Div` to represent subtraction, multiplication, and division of the two values, respectively. (Assume `Sub` means that the right operand is subtracted from the left operand.)
      - add node type `Neg` to represent negating a value.
      - add node types `Sin` and `Cos` to represent the sine and cosine trigonometric functions, respectively.

   b. Extend functions `eval` and `derive` to support the above modifications of `Tree`. Test the extended functions thoroughly. (What should be done for division by zero?)
2. Further extend the expression `Tree` program developed in the previous exercise as described below.
   a. Develop a new function `simplify` that takes an extended expression `Tree`, simplifies it by evaluating *all* subexpressions involving only constants (i.e., not evaluating variables), and returns the new expression `Tree`.

      For example:

      - `simplify(Add(Const(2),Const(3)))` yields `Const(5)`

- `simplify(Add(Add(Const(1),Const(1)),Mul(Const(1),Const(3))))` yields `Const(5)`
- `simplify(Neg(Const(5)))` yields `Const(-5)`

b. Extend the `simplify` function to exploit mathematical properties such as identity elements (`x + 0 = x = 0 + x`), and zeros

c. Further extend the `simplify` function to exploit associativity (`(x + y) + z = x + (y + z)`), commutativity (`x + 1 = 1 + x`), etc.

3. Modify and extend the traditional object-oriented version of the expression-tree calculator program in the same manner as described in the previous two exercises.

1.12 Acknowledgements

In February 2016, I created these notes by adapting and expanding the Web document A Scala Tutorial for Java Programmers by Michel Schinz and Phillipp Haller [7] to better meet the needs of the students in my Scala-based courses: to explain several issues more thoroughly, improve the narrative, make the document accessible, and better integrate it with my other Scala notes. A version of the document can be found on the Scala Language website [4] at https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html.

In Spring 2017, I updated the format to be better compatible with Pandoc. I revised the document further in Spring 2018 (e.g., adding the first exercises) and Spring 2019 (e.g., correcting a few Pandoc errors, improving the markup, adding an explanation of equality, and providing links to all the Scala source code.).

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on the textbook *Exploring Languages with Interpreters and Functional Programmng* (ELIFP) [1] and other instructional materials. In January 2022, I began refining the existing ELIFP content and related documents such as this one. In general, I am integrating separately developed materials better, reformatting the documents (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc. In these notes, I added the discussion on the object model and typing terminology, including significant new text in the subsectons on "Everything is an Object" and "Traditional Object-Oriented Program".

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

1.13 Terms and Concepts

TODO: Add

1.14 References

[1] H. Conrad Cunningham. 2022. *Exploring programming languages with interpreters and functional programming (ELIFP)*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/ELIFP.pdf

[2] David Matuszek. 2022. Concise Java to Scala. Retrieved from https://www.cis.upenn.edu/~matuszek/Concise%20Guides/Concise%20Java%20to%20Scala.html

[3] George Petrov and Contributers. 2022. Learn Scala in y minutes. Retrieved from https://learnxinyminutes.com/docs/scala/

[4] Scala Language Organization. 2022. The Scala programming language. Retrieved from https://www.scala-lang.org/

[5] Scala Language Organization. 2022. Tour of Scala. Retrieved from https://docs.scala-lang.org/tour/tour-of-scala.html

[6] Scala Language Organization. 2022. Tour of Scala: Unified types. Retrieved from https://docs.scala-lang.org/tour/unified-types.html

[7] Michel Schinz and Phillipp Haller. 2016. A Scala tutorial for Java. Retrieved from https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html

[8] Wikpedia: The Free Encyclopedia. 2022. Algebraic data type. Retrieved from https://en.wikipedia.org/wiki/Algebraic_data_type

[9] Wikpedia: The Free Encyclopedia. 2022. Parametric polymophism. Retrieved from https://en.wikipedia.org/wiki/Parametric_polymorphism

[10] Wikpedia: The Free Encyclopedia. 2022. Subtyping. Retrieved from https://en.wikipedia.org/wiki/Subtyping

[11] Wikpedia: The Free Encyclopedia. 2022. Type inference. Retrieved from https://en.wikipedia.org/wiki/Type_inference

[12] Wikpedia: The Free Encyclopedia. 2022. Anonymous function. Retrieved from https://en.wikipedia.org/wiki/Anonymous_function

[13] Wikpedia: The Free Encyclopedia. 2022. First-class function. Retrieved from https://en.wikipedia.org/wiki/First-class_function

[14] Wikpedia: The Free Encyclopedia. 2022. Higher-order function. Retrieved from https://en.wikipedia.org/wiki/Higher-order_function

[15] Wikpedia: The Free Encyclopedia. 2022. Functional programming. Retrieved from https://en.wikipedia.org/wiki/Functional_programming