Introduction to Scala

Scala was designed with Java in mind and is best understood in relation and contrast to it. Publicly released in 2004 on the Java platform, Scala was updated to v2.0 in March of 2006 and has a currently stable v3.2.2, released in January of 2023. Its key features include type inference, strong static typing, static binding (with some exceptions), higher-order functions, traits, concurrency support, and the ability to integrate seamlessly with Java. Scala is generally compiled before execution but also includes an interactive shell. Importantly, Scala has found enough traction to be adopted in several industry sectors, such as finance, data engineering, and distributed systems.

Java is a wildly popular language, but it comes with its criticisms: its forced use of object-oriented programming, lack of immutability, and the omission of lazy evaluation among them [1]. Scala seeks to address these (and other) faults labelled at Java, culminating in a multi-paradigmatic programming language that leverages most of what Java has to offer, including the portability of the Java virtual machine (Scala source code can be compiled to bytecode and run on the JVM), and the thousands of existing Java libraries, while also incorporating many of the functional programming elements seen in languages like Haskell, including pattern matching, currying, and more. Interestingly, the name Scala is a portmanteau of the words “scalable” and “language,” indicating that it is a standalone programming language designed to scale with the needs of its users.

Programming Paradigm

Scala is called multi-paradigmatic since it is fundamentally a combination of object-oriented and functional programming. Indeed, programs in Scala can be written using a nearly solely functional programming style while allowing a user to blend the two as needed [1]. Allowing its users to switch between each, depending on need, is one of the main attractions Scala offers.

In terms of specific object-oriented programming concepts, Scala provides notable features (all of which are also found in Java), such as classes, objects, inheritance, encapsulation, and polymorphism. Scala allows you to define classes and create objects based on those classes. Classes can be considered “blueprints” for creating objects, while objects are actual instances of classes. The syntax for defining classes, creating objects, and accessing methods are like other OOP languages in this regard. We can define a class and create an instance of that class, for example:

```
Class Worker( val name: String, val job: String):
  def job(): Unit =
    println(s”Hello, my name is $name and my job is $job.”)

val worker = Worker(“Alice”, “hostess”)
worker.job() // prints “Hello, my name is Alice and my job is hostess.”
```

Regarding specific functional programming concepts, Scala utilizes immutability by default - when a value is assigned, it cannot be modified. For example:

```
val x = 5
val x = 3 // will not compile.
```

This focus on immutability encourages a functional programming mindset by default. Of course, since Scala blends OOP with FP, it defines objects variable objects for mutable data:

```
var x = 5
x = 3 // compiles.
```

Scala also treats functions as first-class citizens, allowing them to be assigned to variables, passed as arguments to other functions, and returned as results. Consider the following higher-order function:

```
val ffx(f: Int => Int, x: Int): Int =
  f(f(x))
```

For Haskell users, the type signature should look familiar: ffx is a function that takes a function (that takes an Int and returns an Int), an Int value x, and returns an Int. Note also the cleanliness of the syntax: while you can use **return** f(f(x)), it is understood that the expression on the last line will be the value returned.

As a final introduction to the functional programming aspects of Scala, let's look at pattern matching. Recall that pattern matching is a way to check a value against various patterns, where the first pattern to match produces a particular action. In Scala, pattern matching is implemented using the `match` keyword, allowing you to list multiple cases (patterns) and the corresponding code to execute upon a match. For example:

```
def matchDay(day: String): String = day match
  case "Monday" => "Feeling refreshed, I hope. Let’s get started."
  case "Tuesday" | "Wednesday" | "Thursday" => "Keep up the great work."
  case "Friday" => "One more day, bud."
  case "Saturday" | "Sunday" => "You’re still… studying?"
  case _ => "I’ve never heard of this day."
```

Again, those with experience in Haskell (or similar FP languages) will find familiarity here. matchDay is a function that takes a single String argument, which is matched against five possible patterns, each corresponding to code to execute upon a match. Note how easily we implement an exhaustive check with the last case, utilizing the _, which catches all strings that do not conform as expected.

The flexibility accrued by mixing paradigms is one of the things about Scala that shines. It makes allows for imperative programming where OOP makes sense but allows for the conciseness of declarative programming when FP is called for. Coupled with its integration with

Java's ecosystem, especially Java's vast, well-established libraries, and the portability of the Java runtime environment, it should be no surprise that Scala has its advocates.

Data Manipulation

Expressions

In general, in Scala, expressions are computable statements. For example:

```
println(1+1) // prints 2.
```

The results of expressions can be named via the val keyword. Recall that values cannot be reassigned. To reassign, we use variables via the var keyword.

```
val x = 1 + 1
var y = x
y += 5 // y == 7, x == 2
```

Blocks

You can group multiple expressions by encapsulating them in curly braces {}. This is coined a "block." For example:

```
println({
  val x = 5
  x*5
}) // 25
```

Notice that the expression on the second line of the block computes a new value, in this case, 25. It does not modify x.

Functions

There are anonymous functions in Scala which can quickly and easily be assigned to values. For example:

```
val modulo = (x: Int, y: Int) => x % y
```

We can then quickly and easily use those functions to perform computations:

```
scala> modulo(25, 4)
val res0: Int = 1
```

Methods

Methods are very similar to functions, with a few notable differences. Syntactically, they require the def (define) keyword, followed by the parameter list, a colon, and the return value. Consider again the modulo function, written as a method:

```
def moduloMethod(x: Int, y: Int): Int = x % y
```

Operationally, moduloMethod works much the same as our original function. And as is typical for OOP, methods are generally declared as a class member.

Singleton objects

Scala also uses a particular construct known as an object, which is a singleton instance of a class. Singleton objects are helpful when you only need one instance of that object in the entire program. They are generally used in situations where one globally accessible instance is required. A singleton instance that implements our modulo method would look like this:

```
object moduloUtil {
  def moduloMethod(x: Int, y: Int): Int = x % y
}

scala> val moduloObject = moduloUtil
scala> moduloObject.moduloMethod(5,4)
val res0: Int = 1
```

Nested Methods

Scala also allows for the use of nested methods. That is, a method defined within another method. For example:

```
def outerMethod(): Unit =
  println("Outer method")

  def nestedMethod(): Unit =
    println("Nested method")

  nestedMethod()

scala> outerMethod()
Outer method
Nested method
```

For Comprehensions

For comprehensions in Scala provide an easy way to work with collections, performing iterations, filtering, and transformations, and can be used with various collections like lists and arrays. For example:

```
val numbers = List(1,2,3,4,5)

val numbersDoubled =
  for num <- numbers
  yield num * 2

scala> numbersDoubled
val res0: List[Int] = List(2, 4, 6, 8, 10)
```

Of note: the <- symbol binds each element to the num variable. The yield keyword is used to collect the transformed values into a new collection. In addition, we can add filters to the process to remove elements as desired:

```
val numbersDoubled =
  for num <- numbers if num % 2 == 0
  yield num * 2

scala> numbersDoubled
val res0: List[Int] = List(4, 8)
```

Classes

Scala uses classes as blueprints for creating objects in step with Java and OOP. Classes can be abstract, defining common traits and methods for all classes derived from that abstract class. Abstract classes, like in Java, cannot be instantiated. The following is an example of an abstract class:

```
abstract class MagicalCreature {
  val typeOf: String
  val attack: Double
  val travel: String
  def displayInfo(): Unit = {
    println(s"Hello, I am of type $typeOf.")
    println(s"I do damage of " + attack)
    println(s"$typeOf's travel by " + travel + "\n")
  }
}
```

In contrast to abstract classes are concrete classes, which represent classes that can be instantiated. A class derived from an abstract class is said to extend that class and must define all of the attributes and methods stated by that class. For example:

```
class Dragon() extends MagicalCreature {
  val typeOf = "Dragon"
  val attack = 100
  val travel = "soaring through the sky with mighty wings."
}
```

Traits, Polymorphism

Scala also provides several other OOP concepts that can be found in Java. Traits in Scala are similar to interfaces in other languages like Java. They define abstract methods that must be defined by classes implementing those traits. While classes can only be derived from a single superclass, they may extend multiple traits.

In step with the idea of inheritance, Scala also allows for polymorphism. Polymorphism allows different objects to respond differently to the same method. Variables defined as a type of superclass can then be assigned as a derivative of that class later in the program. While Scala is a compiled language and much of the binding is done at compile time, Scala must also incorporate dynamic binding to allow for polymorphic behaviour.

First Class Functions

Scala treats functions as first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, and returned as results. It should come as no surprise then that Scala has higher-order functions, i.e., functions that take functions as arguments. One example of this can be seen in the List.tabulate(size)(function) function, which can be used to quickly populate a List by supplying a size and a function, which receives the index of that list as an argument, to produce objects for that list. For example, we can create a list of the squares from 0-19 via the following (where the second argument is an anonymous function):

```
val numbers = List.tabulate(20)(i => i*i)
```

Pattern Matching and Recursion

As was discussed in the overture, Scala uses pattern matching, an elegant way of breaking down lists and collections. Let's consider one more example, the following quicksort function, which shows off several nice features of the language:

```
def quicksort[A](xs: List[A])(implicit ord: Ordering[A]): List[A] = xs match
  case Nil => Nil
  case x :: xs =>
    val smaller = xs.filter(elem => elem <= x)
    val larger = xs.filter(elem => elem > x)
    quicksort(smaller) ++ List(x) ++ quicksort(larger)
```

We see the use of pattern matching against two separate cases: either an empty list or a non-empty list. The first two lines of the non-empty case are akin to the "let/where" syntax in Haskell, followed by a recursively returned expression. It should also be noted that Scala supports tail recursion for directly recursive calls (meaning, unfortunately, that the quicksort implementation above will not be optimized).

Laziness

Leveraging again from the FP paradigm, Scala incorporates lazy evaluation in the form of LazyList (formerly streams), meaning it is possible to produce and cull from infinite sequences as desired. The syntax blends ideas and elements from the world of OOP and Haskell. Creating and printing 15 elements from an infinite list of non-negative integers, for example:

```
val nonNegatives = LazyList.from(0)
nonNegatives.take(15).foreach(println) // prints values 0 - 15
```

Scala's Integration with Spark

Scala is a popular choice for big data and data processing due to its integration with Apache Spark, one of the most widely used frameworks. One of the challenges in working with Big data involves such massive amounts of data that the processing capacity of a single machine is exceeded. To deal with this, Spark enables distributed processing, whereby data is divided across multiple machines and processed in parallel. In addition to quantity, Big data also poses the problem of variety, in the sense that data can be structured, semi-structured, or unstructured. Spark provides libraries that offer support for working with each. Finally, a fundamental problem with Big data is the ability to scale as necessary. As the size of the data set grows, the number of machines required to process the data may also need to scale. Spark handles scaling efficiently by automatically handling the tasks of scheduling and data partitioning. Apache Spark is written in Scala, making Scala an obvious choice when deciding on a language that utilizes Spark's API [2].

Summary

We have delved into the various aspects of Scala programming, drawing comparisons to the language it was designed in response to, Java. Scala is a powerful multi-paradigm language that combines the ideas of object-oriented and functional programming into a highly flexible programming experience. Offering first-class functions, recursion, concurrency support, and seamless integration with the Java ecosystem, there is something for everyone. This leads to extreme flexibility in its approach to data manipulation. We have seen the use of traits, abstraction, polymorphism, and the accompanying data encapsulation of objects via classes, as well as a variety of functional programming concepts like for comprehensions, lazy evaluation, pattern matching and recursion. With the additional integration of Apache Spark for big data processing, it's not hugely surprising that Scala has found adoption across various industries - a testament to its versatility and scalability.

[1] "Scala (Programming Language)." *Wikipedia*, Wikimedia Foundation, 22 Jun. 2023, en.wikipedia.org/wiki/Scala_(programming_language). Accessed 4 Jul. 2023.

[2] "Apache Spark." *Wikipedia*, Wikimedia Foundation, 28 Jun. 2023, en.wikipedia.org/wiki/Apache_Spark. Accessed 5 Jul. 2023.