# Programming Language Features of Scala

CMPT383 - Summer ‘21
Trevor Grabham
301281720

## Functional or Imperative

Scala is regarded as a multi-paradigm programming language. This is due to how Scala handles variables and functions. All variables in Scala are implicitly stored as objects, which leans heavily on the idea of an Object-Oriented Programming paradigm. However, functions themselves are also treated as values, and are themselves stored as objects. There is also a lot of functional support built into the language. There is support for higher order and nested functions, support for lambda or anonymous functions, and even for currying, like what we can find in Haskell.

For a comparison, we get a lot of the object oriented support that we have come to expect in a lot of the imperative programming languages that we are used to using, such as Python and C++. Scala does also give us the added benefit of a lot of functional programming options that were built into the language from the beginning, as opposed to how it is being added on in updates to other languages such as Python and C++.

## Compiled or Interpreted

Scala, like most programming languages, gives us options for how our code can be run. It does have an interpreter that can be used to run code without needing to compile it beforehand, but the most common way to run Scala code is through the Scala compiler. The Scala compiler actually compiles Scala code into Java Byte Code, which can then be run on the Java Virtual Machine. This allows the language to borrow heavily from a lot of the ground work that was put into the JVM, and all of the features that come with it.

To compare this to Python, while both languages will make use of a virtual machine, and will compile the source code to a byte code format, Scala is normally considered a compiled language, while Python is an interpreted language. This contributes to the fact that running on their native compilers, Scala code is usually much faster, for this reason, and others that we will be discussing later.

## Compiler Optimization (if compiled) or JIT/AOT compilation (if interpreted)

Even though Scala is considered to be a compiled language, it is run on the JVM. The JVM itself is a just in time compiler, which means that we get all of the benefits that would normally come with this as well. Because we are working with a statically typed language in Scala, we are able to do a lot of type checking and optimization in the compilation process, as we are creating the Java Byte Code, but once we create the byte code, we also get the benefits of the JIT compiler, such as optimized machine code for specific CPU architectures, and the collection and use of usage stats for the actual code that is being run. Because Scala was able to piggyback off of the work that was put into the JVM, most of the optimizations that are taking place in the Scala compiler are optimizations that are not already being addressed by the JVM. This doesn't leave much left to do for the Scala interpreter, but the only drawback is that it can quickly become obsolete, as improvements to the JVM happen independently from improvements to the Scala interpreter.

To compare this to Python, which with the introduction of PyPy, also uses a JIT compiler, we would expect to see relatively similar execution times. We would expect to see a bit of extra memory usage to keep track of the overhead associated with a JIT compiler, but much better optimization, especially for running on more specific architecture, with multiple cores. This could potentially have more of an increase for a language such as Scala which was meant to be a more scalable language, meant to take advantage of multi-core processors.

## Static or Dynamic Typing

Scala, like the language that it was based off of, implements strong, static typing. It does, however, have the ability to infer types, so that sometimes types can be omitted when defining variables and methods. This type inference is similar to what we see in languages such as Haskell, where certain methods will have type signatures that will allow us to infer things such as the return type, or the parameter type. Because Scala uses static typing, it allows for type checking at compile time. This can lead to much safer code. Scala has set up its type system, and the type hierarchy to allow for protection from wrong behaviour. This type system does give us the ability to implement features that are seen in other languages like Haskell, by allowing us to set up Scala's version of typeclasses. Finally, Scala does support type coercion, or type casting, but in a limited fashion. Type casting is really only meant in certain directions, like from an int to a float, and is not supported in the other direction.

To compare this to Python, which is weakly, dynamically typed, we see a lot of differences. Python allows for a lot more leeway in the declaration of types, and how bindings are handled. Python makes a lot more use of duck typing, and does all of its checking at runtime. This comes with more overhead costs, and more computational costs to look up the data type of functions and to find the corresponding function for that data type. However, the way that Scala handles generics, while solving a lot of the problems of bindings and polymorphism, can potentially introduce some problems, as it can lead to type erasure, which happens when collections, such as lists, of certain types, can lose their associated types during runtime, which can introduce non-obvious bugs to the code.

## Polymorphism

Polymorphism in Scala is handled in two main ways, generics and function overloading. Scala allows for classes to implement interfaces, which allow for a form of polymorphism, where a class can implement a function that is defined in the interface, and can from there on out, be treated as a member of the interface. It also supports generics and typeclasses, which is something that is similar to what we see in type signatures in functions from Haskell, such as map :: (a -> b) -> [a] -> [b], which says nothing about the types of a or b. Scala also allows us to overload functions, so that the same function name can have different outputs based upon the types, or number of given parameters. This does pair well with the static typing, and the type checking that is done in the compilation stage, as it is at this moment that the compiler can decide for most of the functions, which function is being referenced by each of the function calls, so that the lookup does not have to happen during runtime.

Compared to a language like Python, we see that Scala does offer limited support for polymorphism, a lot more similar to the support seen in languages like C++ and Java, than what is offered in a language like Python. Python does not require function overloading for different data types, it allows the use of the function for any data type that can perform the operations involved in the function call, and will not complain during compilation, but during runtime. In Scala, the programmer loses some of the freedom and flexibility, but is guaranteed a lot more safety in the code that is written.

## Memory Management

Since the Scala compiler compiles the source code into Java Byte Code, the memory management is handled by the JVM. The JVM implements a garbage collection system, in which all objects in memory that are no longer in use, are automatically freed and the space is reclaimed by the program. The JVM accomplishes this garbage collection in a three step process; marking, deletion, and compacting. In the marking phase, in a way similar to tracing, as covered in lecture, it will keep track of all of the objects that are reachable from threads, handlers, etc., and mark these objects as alive. Any other objects are freed during the deletion stage, and the alive objects are then shifted in memory, so that they are all adjacent, during the compacting stage.

Obviously, since this is a function of the JVM, this is identical to what happens with Java code, but this is completely different to a language like C or C++. Those languages leave memory management in the hands of the developer, choosing instead to reduce the amount of overhead and extra computational steps involved in memory management. While garbage collection does add overhead costs to the runtime, it does lead to safer code, free of memory leaks, that seems to be a big focus for the language.

## First-Class Functions

Since in Scala, every function is treated like a value, and all values are stored in memory as objects, Scala does implement first-class functions. Functions can be passed as parameters to other functions because of this aspect. Scala also allows for the creation of anonymous, or lambda functions. Lambda functions can even be created and given a name, by assigning them to variables. Scala has a lot of built in support for functions, as it was designed to be a functional programming language, and so it also supports a lot of other function operations, such as currying.

Comparing this to a language such as C, where you do not see things such as first class functions, Scala gives us a lot more flexibility in what we are able to accomplish with our function calls. This opens up a whole new way of thinking for the imperative programmer, just picking up a functional programming language like Scala for the first time.

## Concurrency

Scala was designed to be an easily scalable programming language, hence the name Scala. With this in mind, Scala has many built in tools for concurrency, one of which being from the fact that it uses the JIT compiler that allows for hardware specific optimizations. It also is able to borrow from Java because of the shared byte code, so it is also able to make use of Java thread objects, and add its own supporting objects such as futures. Finally, there are also many frameworks for concurrency using Scala, the most famous of which, probably being Apache Spark.

A lot of this concurrency support comes from Scala having a lot of emphasis on immutable objects, and from the functional programming support built into the language. This is quite different from most programming languages that I started out learning, such as C and Python. While a language like Python has frameworks that allow for better concurrency, Scala makes it quite elegant and quite easy.

## Conclusion

In conclusion, Scala is a programming language that was designed to address some issues that were seen in the Java language, while adding a lot of functional programming features. The use of the JVM allows for a lot of optimization that is well supported by the Java community, and it allows us to have all of the benefits of things like static typing, and garbage collection, from the JVM, while still having type inference, and first-class functions from the Scala compiler side. Scala scales well, with a lot of concurrency support built into the language, and frameworks that can draw upon this, and extend it.

It does come with its drawbacks though. Since Scala is a functional language that is built upon a compiler that is meant for an object oriented language, some of the functional programming features such as pattern matching are not quite as elegant as in a pure functional programming language such as Haskell. There are some finicky bugs that may also arise as the programmer has to distinguish between functions and methods, as these are handled differently by the language, but seem to be rather interchangeable upon first glance.

Scala would be a language that I would be very quick to pick up if I was coming from a Java background and was looking for support for either scaling up a previous project, or was looking to get into the work of data science, or big data. The concurrency advantages, and the use of the JVM seems quite appealing, and allows for a competitive runtime, when comparing against your more popular imperative languages such as C and Python, and a large increase when compared against your average functional language such as Haskell.