CMPT 383 Assignment 2
Jiadi Luo
301354107

The language I choose for Assignment 2 is Scala, and I will use the Types-First outline.

## 1. Introduction

Scala, whose name is “scalable language”, is a strong statically typed high-level programming language combining object-oriented programming and functional programming paradigms into a concise and efficient language (Potvin & Bonja, 2015). It's not a simple language, and it may take some time for programmers to master it because of its many advanced features (described later in this article), but familiarity with Scala will lead to a more readable, efficient, error-free, and well-organized coding experience.

Scala was designed to develop better language support, especially to create a “better Java” by getting rid of aspects of Java that were considered restrictive, overly cumbersome, or frustrating (Hicks, 2012). Its source code can be compiled into Java bytecode, enabling it to run on the Java Virtual machine (JVM) and interact directly and seamlessly with Java libraries (Seka, 2020). Unlike Java, Scala has many features of functional programming languages, such as currying, pattern matching, immutability, lazy evaluation, tail recursion, etc (Laufer, Lewis, & Thiruvathukal, 2014). In addition to these features, Scala also has an advanced type system that encompasses algebraic data types, higher-order types, and more (Data Fair, 2018).

Furthermore, as a statically typed language, Scala’s variable types are known at the compile time, allowing its code to be executed quicker in less memory (Data Fair, 2018). It has a series of built-in error controls that enable Scala to do quick bug and compile error detection, which makes it safer for programmers’ use (Data Fair, 2018).

In general, Scala is a powerful and popular programming language that combines the benefits of object-oriented and functional paradigms. Its concise syntax and interoperability with Java make programmers much more productive. Scala's emphasis on error control largely increases the security and performance of coding, which further enhances its appeal to developers.

## 2. Type System

### 2a. Built-in types

The built-in types in Scala can be represented in a hierarchy diagram as shown in Figure 1. At the top of all types — *Any*, is the supertype (a.k.a top type) which serves as the root class and defines some standard methods (i.e. *equals, hashCode,* and *toString*) that are accessible to all the Scala objects (Baeldung, 2020). Any has two direct subtypes: *AnyVal* and *AnyRef*.

*AnyVal* represents value types and has 9 value classes: *Double, Float, Long, Int, Short, Byte, Unit, Boolean, and Char* (Baeldung, 2020). These value types cannot be null and have specific characteristics and ranges (Bishabosha, et al.). For example, *Int* represents an integer value,

*Boolean* represents a true/false value, and *Unit* represents a valueless type (Bishabosha, et al.). *Unit* is especially useful as a return type for functions that do not return any meaningful information (Baeldung, 2020).

*AnyRef*, on the other hand, represents reference types in Scala, thus non-value types in Scala are subtypes of *AnyRef* (Bishabosha, et al.). In a Java runtime environment, *AnyRef* corresponds to *java.lang.Object*. Reference types contain objects created from classes and have various methods and fields (Bishabosha, et al.).

*Null* is at the bottom of all reference types and can be assigned to reference types (*AngRef*) (Bishabosha, et al.). While at the bottom of all types — *Nothing*, is a special type that means nothing and is a child of all types, including *Null* (Bishabosha, et al.). *Nothing* cannot be assigned to variables and is typically used to indicate cases where a function or expression returns nothing (Bishabosha, et al.).

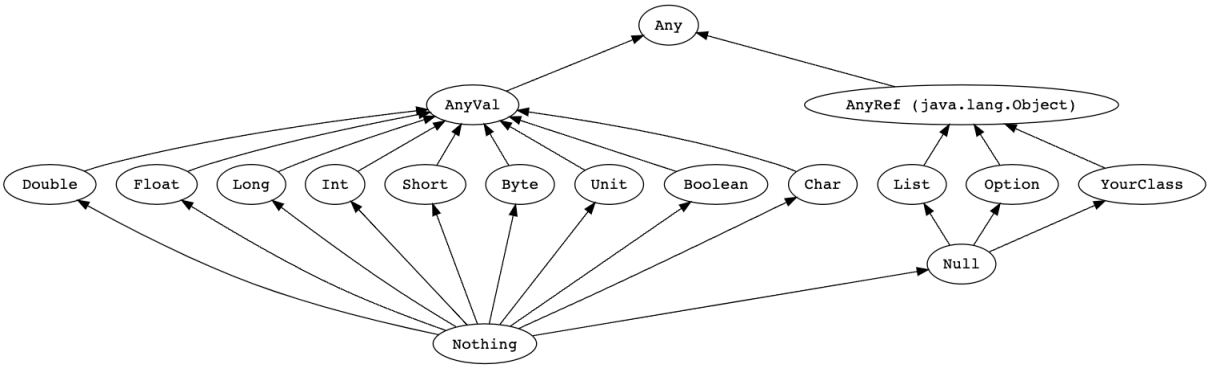


**Figure 1. Scala Type Hierarchy**

## 2b. Objects

As an object-oriented language, Scala has a similar basic object-oriented programming concept as we saw in C and Java. In this section, I will give some Scala class examples in order to better introduce the Scala object-oriented programming.

The basic class definition for Scala can be represented as:

```
class Student {
...
}
```

Now we can add more information to the class to make it look like a real class. The first thing we can add is *Field*, variables that can be accessed throughout the object:

```
class Student {
        var studentID : Int = 0
}
```

We might also want to have a constructor which can be defined using the "*this*" keyword. A constructor initializes the object when it is created and it can have multiple constructors in Scala, each with a different set of parameters (Jenkow, J, 2015). Below is an example:

```
class Student {
        var studentID : Int = 0;

        def  this(value : Int) = {
                this()
                this.studentID = value
        }
}
```

In this example, a constructor is defined with a single parameter that assigns the parameter value to the *studentID* field.

Method is a useful way to encapsulate behaviour and can manipulate an object's state, which can be defined using the "*def*" keyword.

```
class Student {
        var studentID : Int = 0;

        def getStudentID() : Int = {
                return this.studentID
        }

        def printStudentStatus() : Int = {
                println("Current student is enrolled")
        }
}
```

When we want to derive a subclass from the parent class and have a different value, we can use "*extends*" keyword and *Method overriding* by using "*override*" keyword (Prwatch, 2021):

```
class GradStudent extends Student{
        var gradStudentID : Int = 0;

        def getGradStudentID() : Int = {
                return this. gradStudentID
        }

        def printGradStudentStatus() : Int = {
                println("Current student is graduated")
        }
}
```

Overall, classes play an essential role in Scala object-oriented programming and are blueprints for creating objects. They enable the encapsulation of data and behaviour and promote code reuse (Sharma, 2019). Overall, they are foundational to object-oriented programming in Scala, enabling the creation of robust and scalable software systems (Sharma, 2019).

### 2c. Type system features

Scala has a static type system that performs type checking at compile time, so potential errors can be detected early in the development process (Bondar, 2023). Type inference is also supported, allowing the compiler to infer the types of expressions and variables which reduces the programmer's job to declare them explicitly (Flomebul, et al).

Moreover, Scala is a strongly typed language which enforces type safety and prevents many common type-related errors. As we saw in the **2a. Built-in types**, Scala has a type hierarchy (supertype *Any*, subtypes *AnyVal* and *AnyRef*, etc.) which provides lots of flexibility and ensures type safety.

Overall, Scala's type system features help programmers write safer and more powerful code, greatly increasing flexibility and decreasing cumbersome.

## 3. Memory Management

As discussed in **1. Introduction**, Scala runs on the Java Virtual Machine (JVM), and memory management is one of the benefits that Scala gets from running on JVM. The JVM uses garbage collection to automatically reclaim memory that is no longer in use, and Scala objects are subject to garbage collection, similar to Java objects (Julienrf, et al). JVM can be divided into two parts: 1) the memory structure inside the JVM and 2) the working of the garbage collector (Gamage, 2018).

Inside the JVM, there are separate memory spaces for storing runtime data and compiled code: Heap, Non-Heap, and Cache. The main memory space, Heap (Figure 2), is divided into two parts: Young Generation and Old Generation. The Young Generation consists of the Eden Memory and two Survivor Memory spaces. Newly-allocated objects are placed in the Eden space, and when it becomes full, a Minor GC (Young Collection) is triggered.

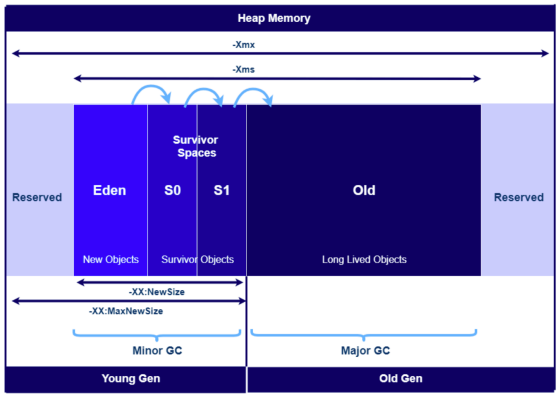


**Figure 2. JVM Heap Memory**

Surviving objects are moved to one of the Survivor spaces. After several cycles of GC, objects that persist are promoted to the Old Generation. The Old Generation is for long-lived objects, and when it becomes full, a Major GC (Old Collection) is performed. The Non-Heap memory

(Figure 3) in the JVM consists of the Permanent Generation that stores per-class structures such as the code for methods and constructors. The Cache Memory (Figure 3) includes the Code Cache, which stores compiled code generated by the JIT compiler and other related information, and got flushed when exceeds a threshold (Gamage, 2018).

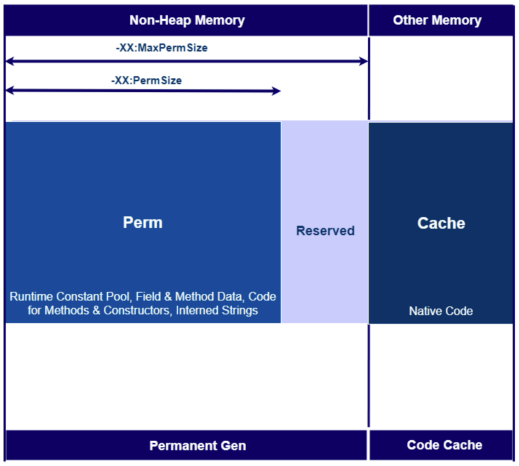


**Figure 2. JVM Non-Heap & Cache Memory**

Working of the garbage collector (a.k.a garbage collection) is crucial for efficient memory management in JVM. As we saw in the lecture, the garbage collector can automatically free up memory by identifying and deleting unreferenced objects. This avoids manual memory allocation and the memory-related problems it causes. By doing this, programmers can save time and the code address memory problems can be improved (Gamage, 2018).

Except for JVM basic features, Scala also provides additional features like value types and lazy evaluation that can affect memory usage and performance.

## 4. Other Interesting Features

### 4a. Singleton Objects and Companion Objects

I've discussed the Scala class in **2b. Objects**, it's worth noting that Scala classes cannot have static variables or methods. Instead, Scala provides a feature called singleton objects or companion objects as an alternative. Similarly, I will give some examples in this section to better clarify the concept.

Singleton objects can be defined using the "*object*" keyword and can contain methods, variables, and other members. The example below defines a singleton object named *objectPrint*:

```
object ObjectPrint {
        def printMessage() {
                println("Hello World!")
        }
}
```

It's also easy if we want to call the method:

```
ObjectPrint.printMessage()
```

A companion object is a singleton object that has the same name defined as a class, for example:

```
class Print {
        def classPrintMessage() {
                println("Hello World from class!")
```

```
        }
}
object Print {
        def objectPrintMessage() {
                println("Hello World from object!")
        }
}
```

They can access private members of the class and provide a way to define static-like behaviour associated with the class. In the example provided, the Main class has a companion object with the same name. We can create an instance of the Main class and call its methods, or call the *objectPrintMessage()* method directly on the companion object:

```
var classPrint: Print = new Print()
classPrint.classPrintMessage()
Print.objectPrintMessage()
```

### 4b. Concurrency and Parallelism

Scala provides powerful features and abstractions to work with concurrent and parallel programming.

Concurrency refers to the ability of an application to execute multiple tasks or processes simultaneously (Nagarajan, 2019). In Scala, concurrency can be achieved through various ways, for example, *Future* (Rockpunk, et al). The *Future* abstraction, just like its name, represents a computation that may complete in the future and programmers can use the Future API to execute tasks concurrently and retrieve the results asynchronously (Rockpunk, et al).

Parallelism, on the other hand, refers to breaking down a bigger task into smaller tasks that will be executed simultaneously (Nagarajan, 2019). Scala's parallelism shares similarities with what we learn in Haskell, for example, the *par* method in Scala can be used on collections like *List* or *Array* to convert them into parallel collections and then apply further operations (Kazanzhy, et al).

### 4c. Lazy Evaluation

Like what we learned in Haskell, Scala also has lazy evaluation, which can be defined using "*lazy*" keyword. The expression won't be evaluated until it's being used first time, for example:

```
val lazyList = List(1, 2, 3, 4, 5)
lazy val result = lazyList.map(x => x + 1)
println(result)
```

In this example, since we are using "*lazy*", the value of *result* won't be evaluated until we print it using *println(result)*.

### 4d. Traits

Traits in Scala are similar to interfaces in other languages, allowing for code reuse and composition by sharing interfaces and fields between classes (Sharma, 2019). For exmaple:

```
trait TraitDef {
        def traitDef()
}

class ClassTraitDef extends TraitDef{
        def traitDef() {
                println("Hello world from trait!")
        }
}

object Main{
        def main(args : Array[String]){
                var a = new ClassTraitDef()
                a.traitDef()
        }
}
```

### 4e. Type Inference

Scala's type inference feature automatically inferences the types of expressions and variables at compile time, reducing the need for explicit type comments and making code cleaner and easier to read. For example:

```
val doubleObj = 10.01   // infers double type
val charObj = 'a'       // infers char type
val stringObj = "hello" // infers java.lang.String type
val booleanObj = true   // infers boolean type
```

## 5. Comparison with Other Languages

### 5a. Scala and Haskell

When comparing Scala with Haskell, they have some similarities in their functional programming features such as pattern matching, lazy evaluation, immutability, higher-order functions, etc. However, unlike Haskell, which is a purely functional language that focuses more on immutability and laziness, Scala is also object-oriented programming which makes Scala more reusability, modularity, encapsulation, and inheritance. Scala also benefits a lot from its compatibility with Java, its concise syntax and interoperability with Java make programmers much more productive. However, Scala is more complex than Haskell due to its object-oriented programming nature and has a steeper learning curve for programmers who are not familiar with object-oriented programming.

### 5b. Scala and Java

When comparing Scala and Java, the JVM-based language has some similarities in nature, allowing you to call Scala from Java and Java from Scala, which seems to offer less integration, and existing application code and open-source Java libraries can be used in Scala. In addition, both languages support object-oriented programming, but Scala goes further by treating

everything as an object, including functions. Compared to Java, Scala provides a more concise and expressive syntax than Java, which greatly reduces the hassle that exists in Java. Scala's compiler can infer types and provides some advanced features that Java does not, such as pattern matching, lazy evaluation, immutability, higher-order functions, etc., enhancing the modularity and expressiveness of code. However, Scala has a steeper learning curve than Java due to its more advanced features and concepts.

## 6. Conclusion

All in all, Scala is a powerful and popular programming language that combines the benefits of object-oriented and functional paradigms. Its powerful features allow developers to write expressive and modular code. Its concise syntax and interoperability with Java make programmers much more productive. As discussed earlier, it's not a simple language and may take some time for programmers to master it because of its many advanced features, but familiarity with Scala will lead to a more readable, efficient, error-free, and well-organized coding experience, which makes it a compelling choice for programmers looking for a modern, powerful language. Overall, Scala offers unique features and capabilities and is a valuable software development tool.

## References


Bondar, V. (2023). *Exploring Scala's Statically Typed Features: Building Safer Code.* MarketSplash. Retrieved July 2, 2023 from https://marketsplash.com/tutorials/scala/scala-statically-typed/

Bishabosha, et al. *Tour of Scala: Unified Types*. Scala. Retrieved July 2, 2023 from https://docs.scala-lang.org/tour/unified-types.html#:~:text=Scala%20Type%20Hierarchy&text=AnyVal%20represents%20value%20types.,Char%20%2C%20Unit%20%2C%20and%20Boolean%20.

Data Fair. (2018). *What is Scala? | A Comprehensive Scala Tutorial*. Data Fair. Retrieved July 2, 2023 from https://data-flair.training/blogs/scala-tutorial/

Baeldung. (2020). *Type Hierarchies in Scala.* Baeldung. Retrieved July 2, 2023 from https://www.baeldung.com/scala/type-hierarchies

Flomebul, et al. *Tour of Scala: Type Inference*. Scala. Retrieved July 2, 2023 from https://docs.scala-lang.org/tour/type-inference.html

Gamage., T. (2018). *Understanding Java Memory Model*. Medium. Retrieved July 2, 2023 from https://medium.com/platform-engineer/understanding-java-memory-model-1d0863f6d973

Hicks, M. (2012). *Scala vs. Java: Why Should I Learn Scala?* Technology. Retrieved July 2, 2023 from https://www.toptal.com/scala/why-should-i-learn-scala

Jenkov, J. (2015). Scala Classes. Jenkov.com. Retrieved July 2, 2023 from https://jenkov.com/tutorials/scala/classes.html

Julienrf, et al. *Scala features*. Scala. Retrieved July 2, 2023 from https://docs.scala-lang.org/scala3/book/scala-features.html#client--server

Kazanzhy, et al. *Scala parallel collections*. Scala. Retrieved July 3, 2023 from https://docs.scala-lang.org/overviews/parallel-collections/overview.html

Laufer, K., Lewis, M., & Thiruvathukal, G. (2014). *Scala*. Loyola University Chicago. Retrieved July 2, 2023 from https://scalaworkshop.cs.luc.edu/epub/intro.html

Nagarajan, M. (2019). *Concurrency vs. Parallelism* — A brief view. Medium. Retrieved July 3, 2023 from https://medium.com/@itIsMadhavan/concurrency-vs-parallelism-a-brief-review-b337c8dac350

Potvin, P., Bonja, M. (24 September 2015). An IMS DSL Developed at Ericsson. *Lecture Notes in Computer Science.*
Vol. 7916. arXiv:1509.07326. doi:10.1007/978-3-642-38911-5. ISBN 978-3-642-38910-8. S2CID 1214469.

Prwatech. (2021). *Scala - Method Overriding*. Prwatech. Retrieved July 2, 2023 from https://prwatech.in/blog/scala/scala-method-overriding/

Rockpunk, et al. *Scala concurrency*. Scala. Retrieved July 3, 2023 from https://docs.scala-lang.org/scala3/book/concurrency.html

Sekar, T. (2020). *Scala in 2 minutes.* Medium. Retrieved July 2, 2023 from https://medium.com/analytics-vidhya/scala-76a9954f7602

Sharma, A. (2019). *Scala Classes and Objects.* ClassTraitDef. Retrieved July 2, 2023 from https://www.ClassTraitDef.com/tutorial/scala-classes-objects