SCALA

A PREPRINT

Alberto Villarreal
villarreal1@email.arizona.edu

Graysen Paul Meyers
gpmeyers@email.arizona.edu

April 21, 2020

ABSTRACT

Add abstract info here

1 Introduction

Add introduction info here

2 History

The purpose of Scala is to create functional programming using java. It can work with other programming languages since it is compiles in bytecode to transfer the information in a manner that the java virtual machine can work with it. Since Scala derives from Java it is also object-oriented as well, but it made more for functional programming. It can use operator overloading, as well as providing option parameters. The catch with Scala is that is does not check for exceptions, which is a benefit of regular java. Derives from the two words "scalable" and "language". Created in 2001 by Martin Odersky, worked like a newer version of Funnel. You can also use Scala.js (Scala Compiler) that works with JavaScript so you can make websites using Scala.

3 Control Structures

Control Structures in Scala work very similar to control structures to some of the most common of programming languages. It consists of some of the basic ones: if statements, if and else's, switch (match in scala), while loop, and for loop. However, Scala brings its own uniqueness when it comes to these control structures to make them easier and more enjoyable to use.

Starting with the one that will most likely be used is the if statement.

```
if (condition) <expression>
```

Simple and easy to use and it follows the basic if structure that most programming languages do as well, this is also true for the if and else statement.

```
if (condition) <expression>

else <expression>
```

However, the catch with using if or if and else statements in Scala is that they must return a value.

The match statement (or also known as the switch statement) is a more efficient use of utilizing various if statements. Using the case setup allows the match statement to choose a specified line of code to use, however since Scala does not contain a break it does not go through multiple cases, it instead chooses only one. Thus the match statement should be used when only one result out of many is required.

```
<expression> match {
case pattern_match => <expression>
[case ...]
}
```

3.1 Example 1

```
val day = i match {
case 1 => "Monday"
case 2 => "Tuesday"
case 3 => "Wednesday"
case 4 => "Thursday"
case 5 => "Friday"
case 6 => "Saturday"
case 7 => "Sunday"
case _ => "Invalid day"
}
```

Another nice thing about the match statement is that you can match types in Scala. Allowing for unique code to occur depending on if the value you are looking at is a specific type.

3.2 Example 2

```
def typeExample(n: Any):String = n match{
case s: String => "String"
case i: Int => "Int"
case f: Float => "Float
case l: List[_] = "List"
case d: Dog => "Dog"
case _ => "Unknown Data Type"
}
```

If statements can be used within each of the cases to lump similar cases and make your code look more simple and clean.

3.3 Example 3

```
n match {
case a: if 90 to 100 contains a => println("You got an A!")
case b: if 80 to 89 contains a => println("You got an B!")
case c: if 70 to 79 contains a => println("You got an C!")
case d: if 60 to 69 contains a => println("You got an D!")
case f: if 0 to 59 contains a => println("You got an F!")
case _ => println("Invalid grade entered!")
}
```

For loops are also similar to other languages but feature very compact iteration methods. You can easily incorporate nested iterating, filtering values, and value binding. Here are some examples of how to iterate using for loops

3.4 Example 4

```
for (arg <- args) println(arg) //prints out each arg value from a list of args
for (i <- 0 to 100) println(i) //Prints 0 to 100
for (i <- 0 to 100 by 10) println(i) //prints from 0 to 100 in multiples of 10
```

If you want only certain conditions to return values, or certain values to be used you can use if statements to create guards, they are incorporated within the for statement. Even though they may be similar to the conditional statement in most for loops seen in other languages this does not terminate the for loop.

3.5 Example 5

```
for (i <- 0 to 100 if i % 2 == 0) println(i) //prints 0 to 100 only evens
```

To do something similar to nested loops you must add another counter into the for loop

3.6 Example 6

```
for (i <- 1 to 3; j <- 1 to 3) printf("i: %d, j: %d\n", i, j)
//i: 1, j: 1
//i: 1, j: 2
//i: 1, j: 3
//i: 2, j: 1
//i: 2, j: 2
//...
```

The yield keyword is used for value binding, which means returning that specific value to insert into a list. It is a vary useful pattern to utilize list transformation in Scala since it is very simple, but can be complex depending on the use.

3.7 Example 7

```
val a = Array(1, 2, 3, 4, 5)
for (e <- a) yield e //Array(1, 2, 3, 4, 5)
for (e <- a) yield e * 2 //Array(2, 4, 6, 8, 10)
for (e <- a if e % 2 == 0) yield e //Array(2, 4, 6)
for (e <- a if e > 3) yield e //Array(4, 5)
```

While Loops are part of Scala's control structure. Once again they are similar to how other programming languages have While loops setup.

```
while (condition){
//code
}
```

The do while is also within Scala as well. Its key difference is that the loop runs at least one time before determining if the loop terminates, or otherwise repeats the loop.

```
do {//code} while (condition)
```

That is the entirety of the control structures that are within Scala, very simple but can allow for elegant code to be composed.

4 Data Types

Unfortunately, for those interested if there are any unique data types for Scala you will be disappointed in that aspect, as it contains the same data types that are regularly found in Java.

This may or may not be a good thing, for those with experience with Java or languages similar to Java, will have an easy grasp on the types of variables that exist in Scala. However, there are not interesting or new data types that are constructed in Scala, though any new Scala libraries may create them, no new types are made from Scala's construction.

Here are the following types of data types that can be found within Scala:

- Boolean -> Result in either a true or false
- Byte -> An 8-bit signed 2's complement integer (-2^7 to $2^7 - 1$)
- Short -> A 16-bit signed 2's complement integer (-2^{15} to $2^{15} - 1$)
- Int -> A 32-bit signed 2's complement integer (-2^{31} to $2^{31} - 1$)
- Long -> A 32-bit signed 2's complement integer (-2^{63} to $2^{63} - 1$)
- Float -> A 32-bit value that is used for single precision (See IEEE definition of a float online).
- Double -> A 64-bit value that is more precise than a float (See IEEE definition of a double online).
- Char -> A 16-bit unsigned Unicode character (0 to $2^16 - 1$)
- String -> A collection of Char values

5 Subprograms

Due to Scala being derived from Java, its code modularity functions similarly to Java, making it easy and clear for users to split code up via different packages, classes, files, in order to make a cohesive programming environment.

Like Java, Scala can separate files as previously mentioned via classes, packages, or other folders/files. Doing imports are easy to do, simply use the import keyword and after that you will need to choose the proper package that you wish to gain access to.

```
import mypackage._
//This denotes that we want to import everything in the mypackage package

import mypackage.SecretPackage
//This SecretPackage is a class that we want to have access to

import mypackage.{SecretPackage, ExpressPackage}
//This specifies to import only the classes that are mentioned
//in the the curly braces.

import mypackage.{SecretPackage => secret}
//This lets the user rename a class so that it can be called
//by that name for later on in the file.  Great for long file names.
```

Since scala is object oriented, we can divide various task not only in different files, but we can also delegate these various tasks to the specific classes that would best suit them. Perhaps we create a class that is good at doing FILE read/writing, or perhaps you would want to create numerous classes that parse data from a server into several classes of Fruits.

In normal Java, you could apply inheritance into two forms, the implements and extends keywords. Implements forces a class to utilize certain methods, how they are done is abstract, but the issue was that you could only do "". The extends is the basic use of inheritance where it is a super class, the class that the current class derives from.

Scala uses somewhat a combination of the both an interface (implements) and a superclass (extends) with trait. This will allow your classes to have abstract methods, or the classes themselves abstract. You can have more than one trait on a class to this use the with keyword after each new trait

```
trait Eater {
    def Eat(): String
}

trait Sleeper {
    def beginSleep(): Unit
    def beginAwake(): Unit
}

trait Driver{
    def drive() : Unit
}

//A student will be able to eat, sleep, and drive.
class Student extends Eater with Sleeper with Driver{
    //Eater part
    def Eat(): String = "mmmm Yummy!"

    //Sleeper part
    def beginSleep(): Unit = print("zzZZZ")
    def beginAwake(): Unit = print("I am awake!")

    //Driver part
    def drive(): Unit = println("I am driving.")

}
```

6 Summary

Add summary info here

References