

**De La Salle University • College of Computer Studies**

**Final Paper**

Name (last name first) : Fernandez, Ryan Austin

Poblete, Clarisse Felicia M.

San Pedro, Marc Dominic

Tan, Johansson E.

Section : G01

Date of Submission : April 14, 2016

Different languages could have very different designs or they may have very similar designs, but what is important is whether or not they are usable. To describe the designs of the five languages the group studied for this term, namely C++, C#, Python, Javascript, and Scala, the language’s handling of variables, namely data types, object-oriented design, type binding, scoping, addressing, and lifetime; processing, namely branching and iteration; and subprogram handling, namely parameter passing, named parameters, and optional parameters will be discussed.

C++

C#

Python

Javascript is a loosely typed and interpreted language whose control structures are based off of Java and a very static based variable system.

Its variables are simply of the type “var”. Implicitly, it has Number, Boolean, String, Array, Function, Regex, and Object types. With regards to type checking, Javascript, as much as possible, tries to convert the right hand side of an assignment statement into the left hand side’s type. Otherwise, it just changes the variable type altogether to match the right hand side’s. Javascript has no special collection types to speak of. It only has the native array and object type, the former acting as a possibly heterogeneous collection of individual values, the latter serving as an associative array or hash table that maps strings to a possibly heterogeneous set of values. As a result of not having a special collection type, Javascript has no special iterator construct. The closest is using a for…in loop which assigns the keys to the looping variable for both arrays and objects. Javascript is also object-based, not object oriented. It allows the developer to give objects attributes and methods, but there are no access modifiers, inheritance, method overriding, and method hiding. All of these disciplines will have to be simulated by the developers using certain standards, such as naming conventions for private attributes.

As for the variables themselves, they follow implicit and dynamic type binding. As previously stated, if a new type is assigned to a pre-existing variable, it may change its type. Javascript also follows static scoping. If an undeclared variable is used in a function, it attempts to resolve the variable from the static declarations, not from where the function is called. For example:

var x = 5;

function test() {

var x = 4;

test2();

}

function test2() {

console.log(x);

}

test();

test2();

When test is called, which calls test2, the value printed, x, which is undeclared, is resolved as the x in the global scope, not the x from where the function is called. Most variables have a static address. If a variable is declared inside a function, it gains a stack address unless it shares a name with a global variable, in which case it retains the same address. Static address variables last until the lifetime of the document ends. Stack variables last until the function terminates execution.

Due to its similarity with Java and C, Javascript’s constructs are readable since they very much resemble English statements sans the special characters like parenthesis and semicolon. As for special constructs, Javascript’s switch statements act similarly to C++ and Java’s. Break statements are not enforced, so special care must be taken to prevent a logical error from forgetting to break. For loops are also similar to Java and C++, but Javascript has a special for…in loop which looks like

for(x in collection) {

//do something

}

Which assigns each key to x. For an array, this is simply each index. For an object, it assigns each attribute name to x.

For subprograms’ parameter passing, Javascript follows both pass-by-value and pass-by-reference. When primitive types are passed, they are passed by value. When arrays and objects are passed, they are pass-by-reference, because any changes made in the function are reflected in the original calling environment.

Scala is a compiled language based off of Java. Its variable handling is similar to Java’s albeit less strict; it has rather unique control structures; and it supports functional programming, giving its subprograms handling some special specifications.

For Scala, everything is an Object. Anything that is declared is instantiated in the heap and the address is stored in the declared variable. For native data types, Scala supports Byte/Short/Int/Long, Float/Double, Char, String, Boolean, Unit, Any/Nothing/AnyRef. When declaring, all variables must be initialized with a value. When it comes to type checking, Scala throws errors when two types are not compatible. For special collections, Scala supports Lists, Sets, Maps, Tuples, Options, and Iterators. One point of interest here is the Option type, which is a collection containing exactly one or zero elements. It allows an optional value to be passed to a function, which then performs a check on the Option type to see if it contains a value or not. With regards to Iterators, Scala’s collections may produce an iterator object that acts like a typical iterator with hasNext() : Boolean and next() : Object methods.

As for Objects, Scala is Object Oriented. Using the “class” keyword allows you to define a class and using the “object” keyword allows you to define a singleton. Defining a “class” and an “object” with the same name makes the class a “companion class” of the singleton. Any attributes and methods in the singleton are considered like Java’s static elements. Scala supports access modifiers; private, protected, and public; which act like Java’s. One can also implement scope protection as follows.

package Pack1 {

package Pack2 {

class Sample {

private[Pack1] var1 = null

private[Pack2] var2 = null

}

class Sample2 {

}

}

class Sample3 {

}

}

The variable var1 is accessible in any class inside the package Pack1, which includes any class inside Pack2, namely Sample2 and Sample3. Regarding var2, however; it is only accessible by classes inside Pack2, which is just Sample2. Scala also supports inheritance, especially multiple inheritance. As a result, Scala supports method overriding but unlike Java, Scala requires the use of the “override” keyword like so:

override def sampleMethod : Int = {

//do something

}

To solve the diamond problem that comes with multiple inheritance, Scala follows the implementation of the last superclass declared in the class definition, so in

class Test extends X with Y

Any conflicting methods in X and Y are resolved by following Y and in

class Test2 extends Y with X

Any conflicting methods in X and Y are resolved by following X.

As for handling variables, Scala allows type binding to be Explicit or Implicit. If a type is declared such as

var x : Double = 5

x will be declared as Double, but if x is declared as

var x = 5

the compiler will attempt to resolve the right hand side, in this case, an Int, and that will be x’s implicit type. So if later on, the following assignment is attempted:

x = 4.5

The compiler rejects it because x is an Int but 4.5 is a floating point value. As such, Scala’s type binding is static. It will not attempt to convert x into a Float to fit the 4.5. It will simply throw an error. As for scoping, Scala has a static scope. Only variables declared inside the scope can be accessed, regardless of where functions and methods are called. As for addressing, since everything is an object, fields are declared on the heap. When functions are called, the parameters and local variables are declared on the stack. Object fields die when they are collected by the garbage collector; stack variables die after the function terminates.

Scala has interesting control structures, namely pattern matching and for loops. Since Scala does not support break statements, each case in a pattern matching construct has an implicit break. As such, no fall through is allowed. Multiple values can fall under one case though using either

case <val1> | <val2> => //do something

or

case x : Int if x == <val1> || x == <val2>

Also, variable binding is allowed. In the last code line, if the value being matched can be an Int, it is bound to the variable x and is then put into the if clause that follows. Scala also supports case classes, which is matching objects to particular values of their attributes. For example.

class Test(x : Int, y : Int)

def main(args : Array[String]) {

var x = new Test(1,2)

x match {

case Test(3,4) => //do something

case Test(1,2) => //do something else

}

}

x will be matched to the second case since its x and y values are 1 and 2 like the case class.

Scala’s for loops are also interesting. They do not support the C and Java construct of initialization; condition; increment, but instead allow you to set a range and an increment (default 1). Nesting can also be done in a single statement instead of two statements. Demonstrating these two concepts is this implementation of Bubble Sort.

def bubbleSort(arr:Array[Int]) {

for(i <- arr.size - 1 to 1 by -1; j <- 0 to i - 1 ) {

if( arr(j + 1) < arr(j) ) {

var temp = arr(j)

arr(j) = arr(j + 1)

arr(j + 1) = temp

}

}

}

Scala also supports the foreach construct. Let’s say you have an array. To print each element:

var arr = Array(1,2,3,4)

for(x <- arr ) {

println(x)

}

The yield of a for statement is also supported. Let’s say all even numbers of an array are to be put in another array. This is how it is done.

var even = for{x <- arr; if x % 2 == 0 } yield x

With regards to subprograms, since everything in Scala is an object and as such, all variables actually hold a value pertaining to the address of the object they are referring to, everything in Scala is technically pass-by-value since addresses are merely being passed. Scala also supports named parameters for example:

def test(x : Test,increment:Int) {

x.value += increment

}

Which uses the class

class Test(var value : Int) {

override def toString:String = value.toString

}

So the sample code

var temp = new Test(5)

println(temp)

test(temp,5)

println(temp)

Will work as well as the code

var temp = new Test(5)

println(temp)

test(increment = 5,x = temp)

println(temp)

And both calls perform the same operation. Scala also supports optional parameters. A sample function is

def test(x : Test,increment:Int = 5) {

x.value += increment

}

Which uses the class

class Test(var value : Int) {

override def toString:String = value.toString

}

So the sample code

var temp = new Test(5)

println(temp)

test(temp)

println(temp)

Results in the default value 5 being added while

var temp = new Test(5)

println(temp)

test(temp,1)

println(temp)

Results in the provided value 1 being added.

<Main comparisons and contrasts>

In conclusion, all five languages <conclusion here>.