

**CSC617M**

Final Paper

|  |  |
| --- | --- |
| **Section** | G01 |
| **Team Members** | Fernandez, Ryan Austin  Poblete, Clarisse Felicia M.  San Pedro, Marc Dominic  Tan, Johansson E. |
| **Date Submitted** | April 14, 2016 |

**Table of Contents**

I. Introduction / 1

II. C++ / 2

III. C# / 5

IV. Python / 11

V. Javascript / 21

VI. Scala / 24

VII. Comparisons / 32

VIII. Conclusion / 34

**Introduction**

Different languages may be designed differently or similarly, 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++ is a general-purpose programming language developed Bjarne Stroustrup at Bell Labs as an extension to C. C++ has object oriented features, just like in Java, but it also provides facilities for low-level memory manipulation.

**Variables**

Aside from the usual primitive types (char, int, float, double), C++ has added a boolean type, bool, to represent *true* or *false*. C++ supports both explicit and implicit variable declaration. C++ has static scoping. Aliases for data types still exist in C++ by using *typedef*. C++ has also added a string class to handle strings. However, this is not considered part of the fundamental types which are data types directly implemented in the language.

The memory of any program in C++ is divided into different areas called segments including the heap and the call stack.

The heap is where dynamically allocated variables are allocated from while the call stack is where function parameters, local variables, and other function-related information are stored. Function scope variables die when the function dies. Global variables last until the lifetime of the document. There exists static variables (declared with the keyword *static*) that lasts like a global variable but is limited to be used to the block scope it was declared. These variables are also only declared once.

**Control Structures**

The typical *if* statements, *while*, *do-while* and *for* loops, and *switch* are still handled the same in C++. Switch statements in C++ supports fall-through. Variables can be declared in cases in *switch* can be used in succeeding cases, unless the case block was enclosed in {}. Switch is limited to compare its evaluated expression to a constant expression, so no strings and no labels or ranges. *Goto* is supported in C++. *Break* in switch still works the same as before. *Break* for *for* loops will make the program exit the loop. *Continue* for *for* loops will make the program jump back to the top of the loop, disregarding code after the *continue* statement, existing or not. *For-each* loop is also supported but only since C++11.

Iterators in C++ are classified into 5: Input, Output, Forward, Bidirectional and Random-Access.

Input iterators can only advance one at a time to the next element, is used for reading and can be compared to another iterator.

Output iterators can only advance one at a time to the next element and is used for storing.

Forward iterators are input and output iterators combined although they are can only move forward.

Bidirectional iterators are like forward iterators but can move backward.

Random-access iterators are like bidirectional iterators but they can do more. They can access ranges non-sequentially given an offset value. They also support inequality comparisons to other iterators. They also support += and -= as well as dereference using [].

**Subprograms**

Function parameters in C++ can either be pass by value or pass by reference. Passing a variable as is and receiving it as is, or passing the address of a variable and receiving it as a pointer to the variable are considered as pass by value. Passing the variable as is and receiving it as a reference to the variable (done using & symbol in function parameters) is considered as pass by reference.

**Object Oriented Features**

Structures and classes exists in C++. Their main difference is that members in a structure are by default in public while members in a class are by default in private. There exists a special type called union wherein its member variables will only pertain to one location in memory. As such, different data types as member variables in union is subject to change value whenever one of the variables’ value is changed.

Classes in C++ can contain member variables and functions. Function ordering in classes are not important unlike in normal functions in the program. Access modifiers for members include public, private and protected. Encapsulation is implemented via these access modifiers. Classes have default constructors but can have user-defined constructors. Once user-defined constructors are made, default constructors will need to be defined if needed to be used. Single and multiple inheritance is supported in C++. Polymorphism is also present in C++. A virtual keyword is used for member functions in base classes which are redefined in derived classes. Those virtual functions must be defined. There exists abstract classes in C++, which is like a combination of an abstract class and interface of Java, where its virtual member functions can be without definition.

A friend keyword exists that can be used for functions and/or classes. When a friend keyword is used for a function, this function can access the members of a class. This function can be declared in a class but it is not a member of that class. It is a “friend”. As such, this function can access public, private or protected variables of the class it is “friends” with without having to be a member. Same things will apply for friend classes declared in a class.

A const keyword exists in C++. When const is used in parameter declaration, it ensures that the parameter will not change value in the function. If const is used before a function’s name, then it makes the return type as constant. If const is after the () of a function, then the function itself is a const. These case is only used for member functions of a class. Using const for this makes the function not able to change any class members. Using const for variables makes these variables not changeable. They would also need to be initialized.

C++ supports optional parameters. Optional parameters must be the rightmost parameters. If you have more than one optional parameters, you cannot supply your own value for 2nd, and so forth, optional parameters if you do not put a value for the preceding optional parameters.

C++ supports method overriding. This can be done by simply rewriting the method of the base class(superclass) in the derived class (subclass). Hiding also exists in C++. Hiding occurs when at least one method from overloaded methods (2 or more) in a class is overwritten in a derived class. The override causes those other overload methods to be hidden.

**C#**

C# is a general-purpose object-oriented programming language, developed by a team led by Anders Hejlsberg, and initially released in 2002. Hejlsberg himself stated that flaws in most major programming languages of his time (C++, Java, Delphi, and Smalltalk) drove the fundamentals of the Common Language Runtime, Microsoft's equivalent of Oracle's JVM, and in turn, drove the core fundamentals of C# itself.

**Typing**

C# supports both implicit and explicit strongly typed variable declarations. The former is made possible by the standard *int, long, char*, identifiers, to name a few, while the former is made possible by the var keyword. C# also supports a strict boolean datatype (*bool*). It is enforced in conditional statements, such as if() and while(), to have a boolean result as the condition, unlike, say, C and C++, which allow the idea of integers representing true/false. This is to prevent mistakes like *if(a=b)* instead of *if(a==b)* (provided a and b are not booleans), as the C# compiler will not allow this.

**Strings**

C# strings are treated the same way as Java strings. C# strings are immutable, meaning that once they are created, they cannot change. String concatenation creates a new string reference, and does not modify the original one. Once a particular string has no more variables referencing it, it is ready for garbage collection. C# also supports *verbatim strings*, which improve readability by eliminating the need to escape certain characters. Accessing individual characters of a string is done by using the [] notation, unlike Java which uses the charAt() method.

To modify characters within a string, C# provides the StringBuilder class. One can modify the individual characters of a StringBuilder, and there are also various methods available for more complex operations. StringBuilders can be converted back into strings with the toString() method.

**Control Structures**

**Selection**

C# supports the standard selection structures of most modern programming languages today, namely *if*, *if-else*, and *switch* statements. However, C#'s switch statement acts slightly different from that of other languages. The switch statement in C# is interesting in the sense that it does not allow cascading cases if the first case has code in it. This design choice was taken as accidental fall-through in switch statements was a common source of hard-to-find bugs in the early days of C and Java, so they decided to make breaks in non-empty cases mandatory.

**Iteration**

C# also supports the standard iteration structures of most modern programming languages, namely the *count-controlled* loop (for), and the *event-driven* loop (while, do-while). However, C# also supports a special *foreach* loop in order to be able for developers to easily iterate through elements in a collection. This *foreach* loop is commonly used with special datatypes called iterators. Iterators in C# are methods with return type IEnumerable. They make use of the *yield return* statement so return individual values of the collection. The first time the iterator is invoked, the method starts executing until the first *yield return* statement is encountered. This value is then returned. Subsequent calls to the iterator continue execution from the previous yield return statement, and continues until another one is encountered.

**Variable Lifetimes**

The lifetime of a variable depends on where it is declared. A variable declared within an if-else statement or a loop will only last for the lifetime of that statement/loop. A variable declared in a method will only last for the lifetime of that method. A variable declared within a class (field) will last for the lifetime of an object instantiated from that class. However, there are also *static* variables. Static variables last for the entire life of the entire program, and do not require an instantiation of a class to be used.

**Methods**

Methods in C# are mostly identical to methods in Java, but C# has additional features to make the lives of developers easier, in the form of named and optional parameters. In C#, one can use the names of the parameters if he/she does not remember the order of these parameters. For example, if one has a function CalculateBMI(double weight, double height), then one might get the parameters mixed up occasionally, especially since they are both of datatype *double*. To avoid this, either of the following can be done:

CalculateBMI(weight: 123, height: 64);

CalculateBMI(height: 64, weight: 123);

Both these function calls will produce the same result.

C# also supports optional parameters. In C#, a function parameter p1 of a certain function f can be specified as optional to mean that a call to f may or may not necessarily supply a value to p1. Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used.

Optional parameters must be defined at the end of the parameter list, after any required parameters. If the caller provides an argument for any one of a succession of optional parameters, it must provide arguments for all preceding required parameters.

Sample:

public void ExampleMethod(int required, string optionalstr = "default string",int optionalint = 10)

Given this method signature, the following call

ExampleMethod(3, ,4);

will cause an error, since an argument was provided for the third parameter but not the second. Instead, if you want to override the default value of optionalInt, what should be done is:

ExampleMethod(3, optionalint: 4);

**Exceptions**

C# and Java have a major difference when it comes to how they handle exceptions - C# does not have checked exceptions. Java is more restrictive with this since it requires methods that have the possibility to encounter exceptions to declare a *throws* keyword in their method signature. Furthermore, Java also requires other methods that call these previously mentioned throwing methods to also handle these exceptions, either by throwing them yet again, or via a try-catch block.

The minds behind C#, on the other hand, believe that checked exceptions are more trouble than they are worth. Hejlsberg describes two reasons why he chose to remove checked exceptions from C# - versioning and scalability. In versioning, he argues that when a method *foo* that may initially throw exceptions A, B, and C is rewritten to throw, say, an additional method D, then all methods that call *foo* will then also have to be rewritten to handle D in some way. Next, he argues scalability. He argues that while the numbers are few, checked exceptions look great. But as the numbers get higher, from four, to ten, to forty, to eighty exceptions in the throws clause, they just become such an irritation, and the programmers themselves do not handle them individually anyway. They would usually do one of two silly things - either say *throws Exception* in the calling method, or have multitudes of empty catch() statements, both of which defeat the whole purpose of checked exceptions anyway, and result in dirty code. He further argues that exception handling is not as important as being able to simply protect yourself against them. He would much rather use statements like try-finally, with the finally block making sure that the method can "clean up after itself", by say deallocating any resources, and ensuring that the system is always in a consistent state. Generally speaking, he feels that checked exceptions do little to actually make the code more readable. In fact, he believes that checked exceptions often do the exact opposite, that they clutter up the code so much and that they are much more trouble than what they are worth.

**LINQ**

One of the defining features that sets C# apart from Java is the **L**anguage **IN**tegrated **Q**uery (**LINQ**) framework. LINQ adds data querying capabilities to C#, and the ability to write SQL-like statements natively, which results in overall more readable code. LINQ provides almost identical functionality to SQL queries, with *Select, Where, Sum, Min, Max, Average, Join,* and *GroupBy* being just some of the keywords available to LINQ. There even exist extensions like *LINQ to Objects* and *LINQ to SQL*, which build upon LINQ and add Object-Relational Mapping (ORM) features to allow the application layer to easily communicate with the database layer for large enterprise systems.

**Object-Oriented Programming**

**Classes**

Being a language that supports object-oriented programming, C# allows developers to define their own classes and create objects from them. A class in C# is essentially a collection of data (fields, properties) and actions performed on them (methods) that are organized in a logical way. Objects in C# are simply instantiations of classes. C# objects are allocated on the heap, with only a reference to the object location maintained on the stack. As such, objects are *passed by reference* when they are used as method parameters, unlike primitive datatypes, which are passed by value. Once there are no longer any references to a certain object being maintained, that object becomes ready for garbage collection.

**Abstraction and Encapsulation**

C# supports abstraction and encapsulation slightly differently from Java. Both C# and Java support fields for their classes, but the way these fields are accessed is different. Unlike Java, which makes use of explicit getter and setter methods to access fields, C# makes use of *properties*. Properties are special datatypes in C#, which contain a get{} and set{} function. A class should usually declare a property as public, and any calling code will use it as any other variable, reading data via *ClassName.PropertyName*, and writing data via *ClassName.PropertyName = value;*. However, performing a read actually calls the get{} of the property, and performing write calls the set{} of the property. By implementing properties, C# is able to improve code readability, while at the same time still maintaining the principles of abstraction and encapsulation.

**Inheritance and Polymorphism**

C# supports the implementation of multiple interfaces, but not the extension of multiple base classes. To support polymorphism, C# makes use of the virtual and override keywords. Specifying a method or property as virtual in a base concrete class means that extending classes can (but do not necessarily have to) override these properties/methods. In order for an extending class to override a base class’ virtual property/method (eg. public virtual void drawShape()), the override keyword is added (eg. public override void drawShape()). The virtual modifier cannot be used with the static, abstract, private, or override modifiers. By default, properties/methods are non-virtual and cannot be overriden.

Abstract classes in C# work in almost the same way they do in Java. To create an abstract class, the abstract keyword is added to the class signature. Any methods that the abstract class wants to force extending classes to implement will also be marked with this keyword. Abstract methods are implicitly virtual methods. Extending classes then need to mark their implemented methods with the override keyword.

C# interfaces are identical to Java interfaces in functionality. All interface methods are intrinsically abstract. The methods of implementing classes no longer need to be supplied with the abstract keyword.

C# also supports both method hiding and method overriding. Method overriding and method hiding have almost identical uses, both of which allow a deriving class to change the method implementation of a base class without changing the method signature. However, the main difference between the two is that if method hiding is used, the actual method called is based on the class type defined at compile-time, while in method overriding, it is based on the class type defined at run-time.

**Python**

Python is a high-level, general-purpose, object-oriented programming language. It is interpreted and dynamic, and places importance in code readability. Unlike most other programming languages, Python makes use of spacing and new lines to distinguish different segments of code rather than semi-colons and brackets.

**Variables**

**Data Types**

Python has many of the usual data types, including int, float, long, boolean and string. It also has some additional types such as complex, for complex numbers, as well as a few options for collection types, such as lists, dictionaries and tuples. However, unlike some other programming languages, it does not include a data type for individual characters such as char.

Python performs type checking on these variables during run-time rather than compile-time. While performing certain operations on certain data types isn’t allowed and will produce an error, assignment a value with a new data type to an existing variable is allowed. For example:

x = 10

y = “Hello”

z = x + y

Attempting to perform addition/concatenation on an integer variable with a string variable this way will produce an error.

x = 10

x = “Hello”

However, reassigning a string value to a variable that was originally an integer will not produce an error.

Python also allows programmers to make use of iterators. An iterator in Python can be derived from an iterable object through the iter function. Python’s collection data types are iterable, but any class that implements the methods \_\_iter\_\_ and \_\_next\_\_ is also iterable. Using the iterator, one can move through the elements of that object using the next function. For example, this program:

x = [1, 2, 3, 4, 5]

xIter = iter(x)

print next(xIter)

print next(xIter)

print next(xIter)

Will produce the output:

1

2

3

Python makes use of iterators in its implementation of a for loop, which will be addressed later on in this paper.

**Classes**

Python allows programmers to create custom data types through classes. These classes may have their own methods and attributes.

However, access to these methods and attributes is not controlled, since Python has no implementation for access modifiers. All classes, methods, attributes, etc. are treated as public. While the Python programming community has developed naming conventions to make it easier to determine the intended access, there is no way of directly ensuring control of access.

An example of a class definition in Python is as follows:

class MyClass:

def \_\_init\_\_(self):

self.attribute1 = 10

self.attribute2 = “My String”

def function(self, value):

self.attribute1 = self.attribute1 + value

The methods within the class are defined similarly to regular functions in Python, except for one mandatory parameter – “self”. This represents the specific instance of the class that calls the function.

The “self” keyword is also used in defining and manipulating attributes that are specific to the class object, and are defined within the methods.

The \_\_init\_\_ function represents the constructor of the class, and is called whenever a new instance of that class is created. For example:

x = MyClass()

This creates a new MyClass object which the variable x will now refer to.

Though “self” is included among the parameters of the function, it does not need to be among the arguments passed by the programmer when calling the methods. For example:

x = MyClass()

x.function(20)

Python also supports inheritance. This is done in the class definition. For example:

class Parent:

def hello(self):

print "Hello"

def hi(self):

print "Hi"

class Child(Parent):

def hello(self):

print "Hello World"

Here the class Child is defined as a child of the class Parent through the passing of the parent class to the child class.

In doing this, Child inherits the methods and attributes of Parent. However, this example also shows a rewriting of the hello method from the parent. This can be done because Python allows child classes to overwrite methods of their parent classes.

If the following lines were written after the class declaration:

x = Parent()

y = Child()

x.hello()

y.hello()

x.hi()

y.hi()

The program will output the following:

Hello

Hello World

Hi

Hi

The first two output lines are different from each other because of the overridden hello method, but the last two are the same because of the implicit inheritance of the hi method.

**Variable Type Binding, Address, Scope, and Lifetime**

All variables in Python are treated as objects, and are accessed through their addresses in memory.

Type binding for these values are done implicitly, since it doesn’t require the programmer to explicitly declare each variable’s data type, and is dynamic, allowing variables to change data types throughout the program. For example:

x = 10

print x

x = 10.5

print x

x = "My String"

print x

This will produce no errors, and will output:

10

10.5

My String

Python’s variables are statically scoped, so the objects being referred to by each variable can be determined before run time. However, Python does not make use of any explicit variable declaration statements, such as data type declaration or the “var” keyword in Javascript, which makes it more difficult to determine statements in Python refer to existing variables or to new local variables. Python addresses this through the “global” keyword. For example:

myValue = 10

def function1(value):

global myValue

myValue = value

print "function1:"

print myValue

def function2(value):

myValue = value

print "function2:"

print myValue

print "global:"

print myValue

function1(20)

function2(30)

print "global:"

print myValue

This will output:

global:

10

function1:

20

function2:

30

global:

20

This shows us that function1 was able to update the global value of myValue to 20 since the “global” keyword was used to indicate that it was accessing the global variable of myValue. On the other hand, function2 created a new local variable instead of changing the global myValue, since it did not indicate in the beginning that it intended to access the global value instead.

Python stores all values used in the heap, but accesses these values through references stored in the stack.

Variables local to a function die when the function exits, but global variables last for the whole lifetime of the document.

**Control Structures**

Like most other programming languages, Python supports if, else if and else statements, though “else if” is instead written as “elif”. Python also does not make use of brackets and parentheses in the writing of these statements. For example:

if x > 5:

print x

elif x > 2:

x = x + 3

print x

else:

x = x – 5

print x

Python does not support the switch statement.

It supports the standard while loop, but does not support do…while. Parentheses and brackets are also omitted in the implementation of the while loop. For all loops, Python supports break statements.

Python supports the for loop, which makes use of iteration. It is because of this that the for loop may be used for the usual counting, as well as for iterating through iterable objects. For example:

x = ["One", "Two", "Three", "Four"]

for i in x:

print i

This for loop iterates through the iterable object x, and outputs all the values in x.

As previously mentioned, a for loop may also be used for as a usual counting loop. For example:

for i in range(10):

print i

for i in range(3, 10):

print i

for i in range(3, 10, 2):

print i

The first for loop outputs all numbers from 0 to 9. While it may appear to do this like a regular for loop that starts at 0, executes the statements, increments by one and executes statements until it no longer satisfies the condition of the value being less than 10, it actually does something different. The function range here actually creates an iterable object containing the integer values from 0 to 9, and then the for loop simply iterates through that object after it is created, so it is able to act similarly to the previous example.

The second loop does something similar, except the starting value of 3 was identified, so the output will include all numbers integers from 3 to 9.

For the third loop, the increment of 2 was indicated, so the output will be all odd numbered integers from 3 to 9.

**Subprograms**

Python supports functions. An example of a Python function definition is as follows:

def function(value1, value2, value3):

value1["name"] = "The Name"

value2 = 10

value3 = [0, 1, 2, 3]

val1 = {'name' : "Untitled"}

val2 = 0

val3 = [10, 20, 30]

function(val1, val2, val3)

print val1[“name”]

print val2

print val3

Which will output:

The Name

0

[10, 20, 30]

Some data types in Python, such as int and string, are immutable types. These values, once written to memory, do not change. Objects like classes and collections on the other hand are mutable types, and can change in the memory when they are modified. It’s because of this that immutable types, when modified in functions, do not reflect these modifications in their original variables, while mutable types do.

Because of the way mutable and immutable types behave when passed to function, Python appears to pass mutable types by reference and immutable types by value. This is seen in the example, where val1 appears to have been modified by the function while val2 was not. However, that is not actually what occurs in the background.

Python is pass-by-object-reference, meaning that it passes object references by value.

When an immutable object is modified, Python does not modify the actual value. It instead creates a new value, and then changes the address that the variable is referencing to this new value. The same thing occurs when an immutable value is passed to a function – the reference is still passed to the local variable, but when a change is made, the reference of the local variable is changed, making it reflect a different value than the original variable.

Mutable objects appear to be passed by reference because when they are modified the actual value is changed instead of the reference. Since the local variable as well as the original variable remain pointing to the same value, they end up reflecting the same values. However, if a mutable object is assigned a completely new object in the function, the local variable will point to the new object that was created rather than the object referenced by the original variable, causing them to reflect different values. This is seen in the example, since val3 still reflected the same value after the function.

Python also supports named parameters. For example, given the function:

def function(one, two, three):

# Function code here

This function can be called like this:

function(three = 3, two = “Two”, one = 1.0)

Python also supports optional parameters. Here is an example of how they are written:

def func(required, optional1 = 10, optional2 = “String”):

# Function code here

In case a value can be assigned to the second optional parameter but not the first, Python still allows you to since it allows you to specify arguments (even required ones) in any order through named parameters. For example:

func(optional2 = “Hello”, required = 10)

**Javascript**

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

**Variables**

**Types**

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.

**Variable Type Binding, Address, Scope, and Lifetime**

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.

**Control Structures**

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.

**Subprograms**

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**

Scala is a compiled language based off of Java that follows both the procedural and the functional programming paradigm. Its variable handling is similar to Java’s albeit less strict; it has rather unique control structures; and, as previously mentioned, it supports functional programming, giving its subprogram handling some special specifications.

**Variables**

**Types**

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.

**Object-Oriented Features**

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.

**Variable Type Binding, Address, Scope, and Lifetime**

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.

**Control Structures**

Scala has interesting control structures, namely pattern matching and for loops.

**Pattern Matching**

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.

**For-loops**

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

**Subprograms**

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.

**Functional Programming**

Scala also supports functional programming. Functions may be passed to other functions and be returned by other functions. One interesting example is partially applied functions. Take for example

def sayMessage(message : String, recipient : String) {

println(message + “, “ + recipient)

}

One could partially apply this in two ways. First by providing a default message.

var sayHello = sayMessage(“Hello”,\_ : String)

So calling sayHello(“Raphael”), sayHello(“Michelangelo”), sayHello(“Donatello”) and sayHello(“Leonardo”) would result in Hello, <name> being printed based on the parameter passed in the new function.

The other way is to provide a recipient.

var tellBob = sayMessage(\_ : String, “Bob”)

So calling tellBob(“Hi”) and tellBob(“Bye”) would result in the message being printed as <message>, Bob.

If a function is declared as follows:

def assignFirst(func : (String,String) => Unit, value : String) : String => Unit = return func(value,\_:String)

This function assigns the passed value as the first parameter in an partially applied function and it returns that function. Going back to a previous example, if a sayHello function is desired, the following statement will produce the same result.

var sayHello = assignFirst(sayMessage, “Hello)

And sayHello can be used like the sayHello in the previous example.

Functional programming can also be applied to generate specific functions. Consider this function:

def generateMultiplier(mult : Double) : Double => Double = (x : Double) => x \* mult

This function returns a function that multiplies its parameter by the amount passed to it. So if a doubling function is desired:

var doubleNum = generateMultiplier(2)

So calling doubleNum(Double) on any number will return twice the number.

**Comparisons**

Now let us look at the similarities and differences between the languages in terms of variables, control structures, and subprograms.

**Variables**

**Data Types**

All the languages deal with most of the primitives. Python, however, does not have a char data type. Javascript has no specific Byte, Int, Long, Float, or Double type, encapsulating all these into a generic Number type. Javascript also has no Char type. Most of the languages throw errors when incompatible type assignments are performed except for Javascript, which either converts the right hand side to the left hand’s type or changes the variable type altogether. For Python and Scala, all variables are objects.

**Object-Oriented Programming**

With regards to object oriented programming, all the languages except Javascript support the essential object-oriented programming features such as access modifiers, inheritance, and polymorphism. C++ has a special keyword “friend” that allows a function or class to access attributes and methods of a friend class. C#’s specialty is its aim for readability as getting and setting is implicitly done in the language, even though the access syntax is as if the developer was accessing a public property. Python does not support access modifiers and the passing of a “self” keyword to any method is required for it to work. C++ and Scala support multiple inheritance.

**Variable Type Binding, Address, and Scope**

When it comes to type binding, C++ and C# have explicit type binding; Python and Javascript have implicit type binding; Scala supports both explicit and implicit type binding. All of the languages except for Javascript have static type binding. All five languages have static scoping. C++ and C# make use of all static, stack, and heap addressing. Javascript mainly uses static, only using stack for functions. For Python and Scala, since everything is an object, these languages mainly use heap memory but the pointers are either stored in static or stack addressing mode.

**Control Structures**

For control structures, each languages, at the very least, supports the if-else, while, and for loop constructs. All except Python have switch statements. C++ and Javascript’s switch statements are rather similar. C# only allows empty fall-through, which is if a case’s code is empty if it falls through. Scala has no break statements so each case is isolated. Variable binding can be done with Scala’s pattern matching though, as illustrated earlier. C++, C#, and Javascript all support the classic C-style for-loop and some form of iteration through a collection foreach loop. Python and Scala support mainly counting and iteration for-loop constructs.

**Subprograms**

For subprograms, all of the languages support the basic format of multiple parameters returning one type. C++ provides the const keyword, which restricts the changes performed to either the passed actual parameters or the returned value. C++ also supports either pass-by value or pass-by-reference, the latter using the special & character in function declaration, albeit restricting such that null values cannot be passed. C#, Python, and Scala support named and optional parameters.

**Conclusion**

In conclusion, all five languages essentially support all basic requirements of a programming language, albeit with certain design choices that may improve usability in certain cases. C++ is ideal for more experienced programmers since they can make full use of C++’s features, especially with their understanding if the more confusing language constructs such as pass by reference, multiple inheritance, the const modifier, and classic C pointers. C# is better for large scale application development, especially for programmers that are looking for better usability, since C# provides plenty of language considerations such as allowing strings to be compared using == or allowing object attributes to be accessed without explicitly calling getters and setters, to improve readability and writability. For quick scripting, Javascript’s design allows it to be used rather quickly, but it doesn’t exactly ensure good code. Python can be used as scripting but with better code practices enforced, since it has more well-defined language constructs than Javascript, as well as a very readable syntax. Scala can be used for more functional and object oriented problems. Each language has its own use, strengths, and weaknesses and it is up to a wise developer to understand these different language design choices and choose the most appropriate language for the job.