Apple released their new programming language Swift in 2014. It replaced the versions of Objective-C that had been used in the macOS and iOS environments. From its earlier conception, it was built to be fast, bringing a better performance than the previous Objective-C. Using the incredibly high-performance LLVM compiler technology, Swift code is transformed into optimized native code that gets the most out of modern hardware. Swift is a successor to both the C and Objective-C languages. Not only it includes low-level primitives such as types, flow control, and operators, but it also provides object-oriented features such as classes, protocols, and generics.

I. Readability

Unlike Java, Swift has a very small set of syntax that needs to be written before execution. For example: print ("Hello, world!"). This one line of code is a complete program. There is no need to import a library, string handling, or a main function. Programmers also do not need to end each statement with a semicolon. This can result in a much simpler code, improving the readability to many programmers.

However, like most other languages, Swift is not a perfect language for everyone. Even though it cuts down on syntax and other issues, it might be doubtful for the experienced programmers not to have some lines and characters that feel necessary in the program. For example, as mentioned earlier, Swift does not need a semicolon to end a statement. This might take a little time for some programmers to analyze and study code in detail. Since line breaks are used to delimit the end of a statement instead of a semicolon, programmers who are accustomed to using semicolons might easily produce some code errors.

II. Writability

Swift can really bring out the reasoning behind writing "clean code." Objects within Swift make it easier to reuse existing code within applications. Programmers can create an object and inherit all of the code in it through attributes and methods. It saves a significant time by eliminating the need to rewrite the code every time programmers want to implement something in an application. Swift has very loose coding standards that one must adhere to, which means it could be easy to write efficiently as needed. However, on the other hand, the code that has been written badly is likely to get almost unreadable. The writability could get extremely tedious if one does not pay a close attention.

III. Reliability

Reliability is truly a strong advantage when using Swift. Since Apple has such a closed ecosystem, it is easy for them to control the written code and fix errors much more easily on the server-side. It makes things easy for developers, as they could write and maintain code for a few system specifications rather than writing a program that should fit thousands of devices. An example would be developing an application for iPhones. Since its specifications are specific, it is highly reliable and can be easily optimized. It should be much more reliable than developing an application for Android devices, which would involve tailoring to an innumerable number of devices with different hardware configurations.

IV. Keywords, Data Types, and Control Structures

Constants and variables must be declared before they're used. You declare constants with the let keyword and variables with the var keyword. Unlike Java, constant and variable names can contain almost any character, including Unicode characters like π, 你好, or 🐶🐮. Constant and variable names can't contain whitespace characters, mathematical symbols, arrows, or private-use Unicode scalar values. Comments in Swift are very similar to comments in Java. Single-line comments begin //. Multiline comments start with /* and end with */. However, unlike Java, multiline comments in Swift can be nested inside other multiline comments.

Swift offers a collection of built-in data types which are string, integer, floating-point numbers, and booleans. These data types are also found in Java. Swift is a type-safe language. It encourages you to be clear about the types of values your code is working with. In fact, it performs type checks when compiling your code and flags any mismatched types as errors. However, it doesn't mean that programmers have to specify the type of every constant and variable. Swift uses type inference to work out the appropriate type. Type inference enables a compiler to deduce the type of a particular expression by examining the values provided. Because of type inference, Swift requires far fewer type declarations than languages such as Java.

Control structures are easy to read and understand. Like Java and many other languages, to make conditionals, if and switch are used. Loops are created by for-in, for, while, and repeat-while. Optional values are represented by if and let together. If the value is missing, the nil expression is used. The use of while is to repeat a block of code until a change is read. The condition of a loop can be at the end of a statement to make sure that the entire loop is ran once.

V. Operators

An operator is a special symbol or phrase that is used to check, change, or combine values. Swift supports most standard Java operators including arithmetic, assignment, and logical ones, improving several capabilities to eliminate common coding errors. In addition, it also provides range operators that are not found in Java, such as a ..< b and a ... b, to express a range of values.

Operators are unary, binary, or ternary. Unary operators operate on a single target, either before or after the target. Binary operators operate on two targets and are infix as they appear in between the two targets. Ternary operators operate on three targets. Like Java, Swift has only one ternary operator, the ternary conditional operator.

The ternary conditional operator evaluates to one of two given values based on the value of a condition. It has the following form:

condition ? expression used if true : expression used if false

VI. Expressions

In Swift, there are four kinds of expressions: prefix expressions, binary expressions, primary expressions, and postfix expressions. Evaluating an expression returns a value, causes a side effect, or both. Prefix and binary expressions let you apply operators to smaller expressions. Primary expressions are the simplest kind of expression that provides a way to access values. Postfix expressions let you build up more complex expressions using postfixes such as function calls and member access.

Prefix expressions combine an optional prefix operator with an expression. Prefix operators take one argument, that is, the expression that follows them.

Binary expressions combine an infix binary operator with the expression that it takes as its left-hand and right-hand arguments. It has the following form:

left-hand argument   operator   right-hand argument

The assignment operator sets a new value for a given expression. It has the following form:

expression = value

The value of the expression is set to the value obtained by evaluating the value. If the expression is a tuple, the value must be a tuple with the same number of elements. Nested tuples are allowed. Assignment is performed from each part of the value to the corresponding part of the expression.

VII. Syntax and Semantics

Classes in Swift can define properties and methods, specify initializers, support inheritance, enable polymorphism, etc. Swift is also a protocol-oriented programming language, with feature-rich protocols and structs, enabling abstraction and polymorphism without using inheritance. In Swift, functions are first-class types which can be assigned to variables, passed into other functions as arguments and returned from other functions.

The biggest difference between other object-oriented languages like Java and Swift is the rich functionality offered by structs. Swift structs can define properties and methods, specify initializers, and conform to protocols. With the exception of inheritance, you can do anything that is used in a class with a struct. It brings the question of when and how to use structs and classes. This is similar to when and how to use value types and reference types. Structs are not the only value types in Swift. Enums and tuples are also value types. Likewise, classes are not the only reference types. Functions are also reference types. However, functions, enums, and tuples are more specialized in how and when they are used. The debate about value and reference types centers mostly around structs and classes.

With value semantics, a variable and data assigned to the variable are logically unified. Since variables exist on the stack, value types in Swift are said to be stack-allocated. All value type instances will not always be on the stack. Some may exist only in CPU registers while others may actually be allocated on the heap. In fact, value type instances can be thought of as being contained in the variables to which they are assigned. There is a one-to-one relationship between the variable and the data. The value held by a variable cannot be manipulated independently of the variable.

On the other hand, with reference semantics, the variable and the data are distinct. Reference type instances are allocated on the heap and the variable only contains a reference to the location in memory where the data is stored. It is quite common for multiple variables to have references to the same instance. Any of these references can be used to manipulate the instance.

Since a value type instance can have only one owner, and the copy of the instance is assigned to the new variable or passed into the function. Each copy can be amended without affecting the others. With reference types, only the reference gets copied and the new variable or the function gets the new reference to the same instance. If a reference type instance is amended using any of the references, it will affect all other owners as they hold references to the same instance.