EUROPEAN UNIVERSITY OF LEFKE
Faculty of Engineering
Department of Software Engineering

COMP 364
PRINCIPLES OF PROGRAMMING LANGUAGES

SWIFT

Prepared by Mustafa Can Yücel (21121322)

Introduction

When analyzing the Swift language, being an open source language has made it sustainable by a developer community as well as Apple. This has allowed it to constantly evolve with new features and improvements. Thus, the Swift language, which was initially limited to Apple, has gained cross-platform capabilities with server-side frameworks and community efforts.

In this project, the Swift Programming language will be analyzed in depth according to its fundamental aspects. In the analysis phase, we aim to understand the Swift language not only as a language used in the industry, but also from a theoretically focused perspective.

1.Programming Paradigms

Swift supports multiple programming paradigms for developers to choose the best style for their specific problems. This flexibility allows them to develop more readable, maintainable and robust software. These paradigms are:

1.1 Imperative Programming

Swift allows developers to create a series of commands that change the state of the program, such as if, while, for, and control structures, and the ability to change variables. For example:

```
var count = 0
for i in 1...5 {
    count += i
}
print(count) // output: 15
```

In this example, the count variable is a mutable variable, which is a feature of the impreative programming of the swift language. Although it is intuitive for small programs, different problems may arise when programming large, so the use of constants by default is encouraged in large programs. However, the use of variables is not completely prohibited.

1.2Object Oriented Programming

As we mentioned in the history of Swift, encapsulation supports object-oriented principles such as inheritance and polymorphism. Developers can define both data and behavior by creating classes, reduce code repetition through inheritance and change functionality in subclasses. For example:

```
class Vehicle {
    var speed: Int = 0
    func accelerate() {
        speed += 10
    }
}
```

Here, the Vehicle class is a reusable template that includes an internal state (speed) and a behavior that changes it (accelerate). In Swift, classes are reference types, meaning objects are transmitted as references in memory, and the same object can be accessed from multiple locations. However, reference-based data sharing can cause errors when not used carefully. That's why Swift encourages to prefer protocols and value types (1.4struct and enum) in most cases.

1.3 Functional Programmıng

The Swift language also includes the concepts of the functional programming language, such as invariance, pure functions, and the fact that the functions are first-class citizens. It supports this paradigm by defining them with map, reduce, filter, closures and let and constant. For example:

```
let numbers = [1, 2, 3, 4, 5]
let squares = numbers.map { $0 * $0 }
print(squares) // output: [1, 4, 9, 16, 25]
```

In this diagram, the original list and the value of the variable do not change, the map function creates a new list by acting on each element. Since this diagram we mentioned earlier is free of side effects, the inputs always produce the same output. This situation increases predictability and testability. Therefore, this feature of the diagram uses to prevent data confusion in larger programs.

1.4 Protocol Oriented Programming

Swift also includes Protocol-oriented programming. In this approach, behaviors are complemented by protocols instead of inheritance. By the protocols such as struct and enum, these features can gain functionality without depending on the class hierarchy. For example:

```
protocol Drawable {
    func draw()
}

struct Circle: Drawable {
    func draw() {
        print("Drawing a circle")
    }
}
```

In this diagram, composition is encouraged, it breaks down behaviors into small and reusable pieces and eliminates fragile upper class problems. Another feature is; Shared mutable state is blocked. Since struct types in Swift are copied as a value type during assignment. This also increases modularity and reduces dependencies.

In shortly, the Swift language covers most diagrams, each diagram has its use and advantages and disadvantages. While the developers we mentioned at the beginning find problems with their specific problems, they are asked to be hurt by this much variety.

2.Names

In Swift, nouns can be used to represent variables, constants, functions, classes, structures, enumerations, typealias and parameters. If these nomenclatures are used correctly, they increase the ease of maintenance, understandability, and ease of maintenance of the code.

2.1 Case Sensitivity

Case sensitivity differs in each language. Swift language has case sensitivity. For example;

```
let UserName = "Alice"
let username = "Bob"
print(UserName)  // output: Alice
print(username)  // output: Bob
```

2.2 Unicode Support

The Swift language allows variable names to be written with letters or symbols other than the Latin alphabet. The reason why it allows it is that it offers Unicode support. Providing Unicode support is beneficial in areas such as internationalization, scientific calculations or educational applications. For example;

```
let \u03C0 = 3.14159    // Greek letter pi
let привет = "Hello"   // Cyrillic
let 年 = 2025           // Chinese character for 'year'
let 🚀 = "Rocket"       // Emoji
print(π, привет, 年, 🚀)
```

2.3 Rules for Defining Valid Identifiers

In order for definitions to be valid in Swift, the descriptor must begin with a letter or underscore. The descriptor can contain a number, but according to the rule, it cannot start with the number. Again, descriptors cannot contain special characters and you cannot use keywords as descriptive. It is possible to use key descriptors with reverse quotes. Its externally valid tim Unicode characters can be used. For example;

```
let 1name = "Invalid" //  Negative Starts with digit
let first-name = "No"  // Negative Hyphen is not allowed
let `class` = "keyword used as variable"
print(`class`)
```

2.4 Naming Conventions(kaynakça belirt)

The Swift language has a naming convention according to the Swift guide. Although these are not used, the program works, but it should be preferred to do more professional work. Type names should be made according to PascalCase by enlarging the letter at the beginning of each word. For variable functions, it should be made according to camelCase. The first letter of the first word begins with a lowercase letter and the first letter of the other words is enlarged. In abbreviations, all letters are capitalized, and names are chosen in a way that are meaningful and not too long.

3.Binding

What it covers and the meaning varies according to the Binding times. These times are divided into five times as Programming language design time, Compiler application time, Compilation time, Load/Start Time and Run Time. I have reported below what was determined according to these times and at what time;

3.1 Programming Language Design Time

At this time, the syntax of the language is determined. Keywords and operators are connected in certain operations. For example: The + operator is associated with addition in Swift.

3.2 Compiler Application Time

In Swift, representations of data types are also determined. For example, in Swift, the Int type is usually represented as 64-bit.

3.3 Compilation Time

Statically linked variables and types are determined at this stage. For example, the Number constant is matched with the Int type and value at compile time let number = 10 // Static binding of type Int and value 10

The number constant is statically connected with an Int value of 10. This binding is done by the compiler and does not change during the running time of the program.

3.4 Upload / Start Time

In the Swift language, this binding time covers the allocation of the location of Global variables in memory, which is made just before the program is started, binding.

3.5 Run Time

While the program is working, some bindings are made, dynamic binding takes place at this stage, especially in polymorphism and inheritance cases, which method to call is determined at the working time.

In generally, Swift Language is a language that mostly determines types statically. However, runtime dispatch is also possible thanks to structures such as class inheritance and protocol conformance. Binding is an important concept used at every stage of a program, such as writing and running. Modern programming languages such as Swift connect statically and dynamically, providing a compromise between flexibility and performance. Knowing when to use all types of bindings helps programmers generate code efficiently and without errors.

4.Variables

Variables in Swift are symbols or names used to store and reach a value that represents a position in memory. The Swift language supports both variable (var) and constant (let) definition as it has been in our report since the beginning. For example;

```
var age = 25        // changeable
let name = "Atiye" // constant
```

4.1 Properties of Variables in Swift Language

Let's see how to evaluate the six features of variables in Swift language:

4.1.1 Name

The variable is the descriptive name given by the Programmer. According to the guide, there are conditions that do not need to be followed. Case sensitivity is one of the features that should not be forgotten when naming.

4.1.2 Stroge

Some variables may represent registers of the CPU. Swift is among the languages that automatically manage system memory. It provides this with ARC: Automatic Reference Counting.

4.1.3 Type

Swift is a statically typed language, in which the types are known during compile-time.Type specification is optional; the compiler has the ability to infer the type automatically (type inference).

```
var score: Int = 90   // The type is explicitly specified as Int
var grade = "A"       // The type is automatically inferred as String
```

4.1.4 Value

Value is the logical location where the variable is stored. Values in Swift are stored in memory depending on its type.

```
var x = 42            // The value of the variable x is 42
```

4.1.5 Lifetime

How long the variable will be kept in the memory is determined according to its scope in the block where it is defined. For example;

```
func example() {
    var message = "Hello"    // message lives here
}
// message no longer exists
```

The value defined in this function lives by that function process.

4.1.6 Scope

Scopler specifies in which blocks of code the variables are accessible in a language. In the Swift language, it is determined by scope {} blocks. For example;

```
var globalVar = 10
func demo() {
    let localVar = 5
    print(globalVar)  // accessible
    print(localVar)   // accessible
}// print(localVar)  // Error: localVar is undefined here
```

4.2 Aliasing

In Swift language, aliasing is shared with the same memory region. Data in the same memory can be aliased with references types. The same object can be accessed by more than one reference. Swift language does not have strict aliasing rules. Because memory security is automatically ensured.

4.3 Representation of Data Types in Memory

In Swift language , basic types such as Int, Float, Bool are represented in memory in a specific low-level manner:
-Integer: Usually 64-bit, little endian order, 2's complement format.
-Float: Represented according to the IEEE-754 standard.
-Bool: Represented by a single bit or byte (implementation dependent).

In general, the features of variables in Swift, such as name, type, value, scope and life time, are designed as basic structures that allow Swift to communicate with memory by combining with powerful type system and automatic memory management. This can be included among the main reasons for secure and high-performance software development with Swift language.

5.Type Declarations

The Swift language primarily uses static type determination and supports explicit and hidden type declarations. However, it does not support dynamic type determination in the traditional sense.

5.1 Static Explicit Type Declarations

The type of the variable is specified as soon as it is defined. Type does not change throughout the life of the variable. Initial identification and modification of the type of the variable improves readability and type security, and helps to

identify errors related to the type at the time of compilation of problems. It provides a clear understanding of these codes in complex codes. However, in large projects, if it is desired to change the return type of a function, all codes must be updated.

5.2 Statik Implicit Type Declarations

Swift uses the type inferction of the language intensively. The compiler predicts according to the initial value assigned to the variable. Again, because it is a Static language, it has the advantages and disadvantages of not being able to change its type. It causes us to avoid unnecessary type declarations by reducing the extra code repetition. However, in rare cases, the predicted type can be something unexpected.

5.3 Dynamic Type Declarations

Swift programming language does not include this type of notification model. In Swift, the type of a variable is clearly determined or automatically subtracted from the first pulse (type inference). In both cases, once the type is determined, it cannot be changed. If we take this feature into account, Swift is defined as a statically derived language.

The reason is to maximize security in the software process and to be able to identify possible type error as soon as possible before the code is compiled. In this way, developers can predict and prevent their errors when it is time to work. Thus, this is an approach that makes the code more predictable, performant and sustainable.

As a result, Swift is more than the flexibility provided by dynamic type notification, such as security and compile time accuracy expectations.

6.Type Checking

In Swift, type checking is the process of checking whether a variable, function parameter or operator operand is compatible with the expected type. Swift does this at compile time and does not allow type incompatibilities. This check at compile time allows you to notice typos and logical problems before the program is running..

6.1 Compatible Types and Coercion

In order for a type to be compatible with another type in Swift, it is possible with a variable of the same type and a convertible type with an explicit cast. However, it does not support swift coercion, for example:

```
let x: Int = 10
let y: Float = Float(x) // open conversion required
```

A variable of type Integer needs an open transformation to be used as float. The compiler does not support implicit type conversion. This situation provides a great advantage in terms of security.

6.2 Tür Hatalarının Önlenmesi

In Swift language, most type errors are caught at compile time. This is due to the fact that the swift language has a powerful static type system. For example;

```
let decimal: Double = 42.5
let wholeNumber: Int = decimal  // Compile error
```

As seen here, the Swift language does not allow implicit coercion and therefore gives a compilation error. Swift does not allow reinterpreted casta except for special methods such as unsafeBitCast. This greatly prevents complex and difficult to extract errors.

Swift's strong and secure static type system, not supporting implicit type conversions such as Coercion, clearly specifying all type conversions, making the Swift language more secure by noticing errors in advance.This type of control structure of Swift makes it one of the most powerful languages in terms of type security.

7.Lİfe time

In fact, in the report we wrote, although it is short and not descriptive, we saw what life expectancy means for variables. In Swift, it offers automatic and developer-controlled approaches to variable life time. This process is supported by the ARC (Automatic Reference Counting) mechanism.

7.1 Static Life Time

Class members defined by the global and static keyword defined out of the class in Swift language are kept in memory as long as the application is running. This situation causes the Swift language to be open to side effects, creating difficulties in parallel programming recursion support; It allows quick access and global data to be used in many places.

7.2 Stack Dynamic

In Swift, the intra-function let and var definitions are kept on the stack. These variables are automatically removed from memory when they end in the scope in which they are defined. This makes memory usage efficient. Unlike stack dynamic life time static life time, it is suitable for recursion and multi-threading. Since the memory is automatically cleaned, it is not error-prone. The disadvantage is that it is not possible to move the reference of a local variable outside the scope.

7.3 Explicit Heap Dynamic

Swift does not have functions such as direct malloc or free. However, the objects created with the class type are stored on the heap and their life time is determined according to the reference count (ARC). For example;

```
class Person {
    var name: String
    init(name: String) {
        self.name = name
    }
}
var p1: Person? = Person(name: "Alice")
p1 = nil /// Object memory is automatically freed thanks to ARC
```

Although managed with ARC, it can cause memory leaks. This risk should be managed with additional structures such as weak and unowned. Of course, this is a convenience in terms of creating complex data structures and automatic management.

7.4 Implicit Heap Dynamic

Effective types like Array, String, Dictionary actually store references in the stack; the actual data is stored in the heap. When these structures exit scope, the reference count is reset and the data is automatically deleted. Easy to use The risk of memory leakage is low. However, performance management in large data sets requires attention

7.5 Deallocation

In Swift language, belek management is done with ARC (Automatic Reference Counting). This mechanism holds a reference counter for each object and when the number is reset, the object is automatically released. Some of the deallocation systems in Swift language have a counterpart while others do not. As we mentioned before, when using the Automatic Reference Counting system. There is no manual (free/delete) use. It does not support the Garbage collection either.

In Swift, the life time of the variables is managed depending on the scope, structure type (class, struct, let, var) and ARC mechanism. Stack and heap-based life strategies are automatically implemented safely and efficiently. Thanks to these strategies of the Swift language, the user does not need to perform complex memory cleaning operations, as we say every time, the software offers high security and speed because it is a friendly language for developers.

8.Scope

While explaining the variables in the swift language on the subject of Scope, we touched on it in a small way. Like many languages in the Swift language, it is a programming language that uses static scoping, which means determining the accessibility of a variable according to which scope it is defined in. Again, like many languages, a variable or constant, lexical (static) scoping is used, which allows it to be visible in the block in which it is defined and in the sub-blocks. When a variable is searched in the scope where it is defined in a classical way, if it is not found, a higher scope is searched. Control structures such as if, while, for also form their own scope. This facilitates the readability of the code and debugging.Swift has established global scope, function scope, and block scope.Swift static keyword is used to define class-level shared members, unlike C/C++ to restrict at file level.

Swift does not have dynamic scoping, so variables are not accessed based on the scope of the invoking function.

9.Statement

Statements are the basic units of execution in Swift programs. They define actions such as variable declarations, flow control, and function calls.

```
// Variable Declarations
let a = 10                  // 'a' is a constant with value 10
var name = "Alice"          // 'name' is a variable holding a string
// Flow Control
if a > 5 {
    print("Greater than 5") // This will be printed since a = 10 }
// Loops
for i in 1...5 {
    print(i)                // Prints numbers from 1 to 5 }
```

```
// Function Call
func greet(name: String) {
    print("Hello, \(name)!") }
greet(name: "Alice")        // Function call with argument "Alice"
// Assignments
var x = 5
x = 10                      // Simple assignment is allowed
// Chained assignment is NOT allowed in Swift
var b = 10
var c = 0
c = b                       // ✅ Allowed
// a = b = 10 ❌ Not allowed in Swift
```

Unlike C/C++/Java, Swift treats assignment as a statement, not an expression. Therefore, you cannot use assignment as a subexpression in Swift.

Swift promotes clean and understandable syntax. Chain assignments like a = b = 10 are not provided. Function calls, control, and loops have strict and clean structural rules. Constants (let) and variables (var) are properly separated for security and predictability.

10.Expressions

Expressions are the fundamental way to compute values in Swift. They can be composed of literals such as numbers or strings, arithmetic operations, function calls, property accesses, and even conditional expressions. For example, simple literals like 5 or "Hello" can be used, arithmetic like 3 + 4 performed, functions such as max(10, 20) called, properties accessed with object.property, or conditional expressions like a > b ? a : b written.

Swift enforces strict rules regarding how operators behave within expressions. Every operator has a clearly defined precedence, so operations like multiplication are evaluated before addition. Operators can be unary or binary, and their associativity determines how expressions with multiple operators of the same precedence are grouped—typically from left to right. The evaluation order usually follows this left-to-right rule as well.

Side effects are allowed in Swift—for example, with statements like x += 1—but pure functions without side effects are strongly encouraged, especially in SwiftUI or functional programming styles. This leads to safer and more predictable code.

Mixed-type expressions require careful handling because Swift is strongly typed and does not perform implicit type conversions. For instance, adding an integer and a floating-point number directly causes an error. Explicit conversion is required, such as:

```
let a = 10
let b = 3.14
// let result = a + b // This causes an error
let result = Double(a) + b // This works because I convert 'a' to Double
```

I also appreciate that Swift allows me to overload operators. This means I can define exactly how operators like + work for my custom types. For example, I can create a Vector struct and specify how adding two vectors should behave:

```
struct Vector {
    var x: Int, y: Int
    static func + (lhs: Vector, rhs: Vector) -> Vector {
        return Vector(x: lhs.x + rhs.x, y: lhs.y + rhs.y)
    }
}
```

When it comes to side effects again, I try to keep my functions pure whenever possible, but sometimes it's necessary to modify values, like this:

```
func increment(_ x: inout Int) {
    x += 1
}
```

In summary, Swift expressions are designed to balance clarity, strict type safety, and flexibility. This enables writing clean code that is both safe and practical. Also, as mentioned earlier, while side effects are permitted, Swift encourages avoiding them to keep code predictable and easier to maintain.

11.Flow Control Instructions

Flow control instructions are used to modify the execution flow of a program. Swift provides structured and expressive constructs for conditional execution, loops, and early exits, with a strong emphasis on readability and safety.

11.1 Conditional Instructions

Swift language does not use special keywords when using if, else if and else structures. Instead, it uses else if structure. By paying attention to the scope rules, it allows the code structure to continue in a sequential way without breaking its meaning.

Switch expression is much stronger compared to C or C++. Pattern mapping ranges can run any type of tuple and where conditions.

All if and switch conditions must return a Bool type value. This increases type security by preventing errors such as "everything except 0 is considered true" encountered in languages such as C.

11.2 Looping Constructs

In Swift, the working logic of the while loop is in the logic that the loop runs as the condition is provided. The repeat {} while structure makes the condition work at least once in swift. Because the condition is checked later. For and for-in loops are used for certain interval repetition. The For-in structure is similar to the for-each expression in other languages.

12.SubPrograms

Swift dilinde fonksiyonlar func anahtar kelimesi ile tanımlanır.Tanımlanan fonksiyonda daha önceki konularda değindiğimiz isim, parametre listesi ve isteğe bağlı dönüş türü ile tanımlanır.Örneğin;

```
func greet(name: String) -> String {
    return "Hello, \(name)!"
}
```

When the function is called, the executed code is stopped, a new stack frame is created, actual parameters are assigned to formal parameters, if there is a rotation value, a place is reserved for this function body is executed. Then for Return in Function operations; If there is a rotation value, it is assigned to the relevant place, the stack memory is free and the execution of the called code continues again.

13.Parameters

With Swift, being a type safe language, it is mandatory to specify the types of the parameters when functions are defined. Parameters in functions are classified as formal and actual ones. Formal parameters would be the names and types specified during function definitions; actual parameters are the real values passed when the function is called.Default values are assigned to parameters within Swift functions.

```
func add(x: Int, y: Int) -> Int {
    return x + y}
let result = add(x: 5, y: 3) // x = 5, y = 3
```

This feature allows some parameters to be omitted during function call. When a parameter is given a default value, the compiler will not mark an error if the calling function failed to pass the parameter. Instead, the parameter with the default value will be used.

```
func greet(name: String = "Guest") {
    print("Hello, \(name)!”)}
greet()                 // "Hello, Guest!"
greet(name: "Alice") // "Hello, Alice!"
```

Swift follows by-name mapping by default for parameters. This makes function calls more readable. In certain cases, the parameter names are omitted in calls; that is, the parameters are mapped by position. When this is the case, in function definitions those parameters are identified by _ (underscore), which guarantees that the parameter name will be hidden in the function calls.The language is strict in enforcing the exactness of the number and types of parameters in respective function calls.

```
func greet(_ name: String) {
    print("Hello, \(name)!")}
greet("Alice") // call without name
```

Fast is also its general capacity. Generics allow properties to work with different types, but type safety is provided. That is, functions written with generics can be used with different data types, and the type check is still collected at run time. This both cuts and increases security.

```
func swapValues<T>(_ a: inout T, _ b: inout T) {
    let temp = a
    a = b
    b = temp}
var num1 = 10
var num2 = 20
swapValues(&num1, &num2)// num1 = 20, num2 = 10
```

13.1 Pass by value

In the Swift language, a copy of the parameters sent to the function by default is taken. That is, pass by value is the default. When the basic types are changed, the original entered first does not change.

```
func increment(number: Int) {
    var num = number
    num += 1}
var x = 5
increment(number: x)
print(x) // x = 5
```

13.2 Pass by reference

Referenced transition in Swift language is done a little indirectly compared to other languages. The inout structure is used to make a reference transition. Inout is added to the beginning of the parameter, and it is passed with & when calling. Thanks to this method, changes in the function can be made.

```
func increment(number: inout Int) {
    number += 1}
var y = 5
increment(number: &y)
print(y) // 6
13
```

13.3Pass by Name

The pass by name approach is not supported in Swift. In Swift, all parameters must be clearly defined by the variable name and type.

13.4 Overloading

Like other languages, the Swift language has overloading support. Thanks to overloading, the functions with the same name can be defined by different parameter types or numbers.However, it cannot be overloaded only by the type of return.For example;

```
func printValue(_ value: Int) {
    print("Integer: \(value)")}
func printValue(_ value: String) {
    print("String: \(value)”)}
```

13.5 Return

Swift functions can return the value. For example;

```
func square(x: Int) -> Int {
    return x * x }
```

When the function is called, the return value can be assigned or not:

```
_ = square(x: 4) // Can be called without assignment
```

If the return type is Void or (), it acts like a procedure (i.e. it does not return a value).

13.5 Generic Functions

Swift has strong and flexible support for generic programming, which is an essential feature of modern programming languages. With generic structures, functions and data types can work generically with different types and avoid duplication of code and rewriting the same algorithm or operation when it comes to different data types.

With respect to generics, a primary distinction of Swift's generics structure is that it retains its type information entirely at compile time. However, in some languages (e.g. Java), generics make use of a technique called type erasure, and thus erase type information prior to runtime. This technique can increase runtime costs and doesn't allow for direct access to certain type information.

Swift makes use of generic functions and types that completely retain their type information at compile time; thus providing high performance and full type safety. Swift uses a similar approach as C++ template programming, while providing safer and easier-to-use functionality compared to the complexities of C++. While C++ templates provide some powerful and flexible programming paradigms, they can provide complexities and errors that make C++ programming difficult. Swift provides a fully powerful generic system that is accessible and easy to understand.

For example, in Swift you can write a generic function like this:

```
func swapValues<T>(_ a: inout T, _ b: inout T) {
    let temp = a
    a = b
    b = temp}
```

14.Arrays

Swift offers a built-in collection type that makes it easy and powerful to work with series. Arrays can hold one-dimensional, multi-dimensional or more complex data sets. In Swift language, arrays are stored consecutively in memory. Thanks to this

storage method, which is different from others, fast access to strings is provided from other languages. Now let's see how arrays are defined.

14.1 One Dimensional Arrays

```
var numbers: [Int] = [1, 2, 3, 4, 5]
```

14.2 Multidimensional Arrays

```
var matrix: [[Int]] = [
   [1, 2, 3],
   [4, 5, 6],
   [7, 8, 9]
]
```

It can be achieved by keeping an array in an array for multidimensional arrays in Swift language.

15.Structures

Swift dili, C ve C# gibi yapısal programlama dillerinden farklı olarak yapıları çok güçlü ve esnek bir şekilde destekler.Bu desteklemeyi Swift dilinde class ve struct yapısı farklarına bakarak görebiliriz.

15.1 Struct Vs Class in Swift

Unlike a struct class, it is a value type and is discarded by copying. Class is a reference type, a single copy is shared in memory.These features of the Struct structure make it ideal for light and fast data structures according to the class.

Structures in the Swift language also have many advantages. It can hold multiple variables together, memory representation is simpler, it has an Memory alignment and the Swift language can easily adapt to C structures. The structs in the C libraries you import can be used naturally in Swift.

16.Errors and Exceptions

Apple'ın dünya çapında kendini öne çıkarttığı önemli konulardan biri olan güvenlik, elbette kullandığı yazılım dilinde de ön planda tutulmalıydı.Swift dilinde, alışılmış try/catch istisna yöntemini kullanmakla birlikte kendi güçlü modelini sunar.

16.1 Error Protocol ve Throwing Fonksiyonlar:

Swift dilinde, hata durumlarını belirtmek için Error protokolü kullanılır. Fonksiyonlar throws anahtar kelimesiyle işaretlenirken hata fırlatabilir şekilde belirlenir. Hata fırlatıldığında çağıran fonksiyonun hata yakalaması gerekir.

16.2 do-catch Blocks

Hataların kontrollü şekilde ele alınmasını ve programın beklenmedik biçimde çökmesini engeller.Swift'te hatalar do bloğu içinde try ifadesi ile çağrılır, hatalar ise catch bloklarında yakalanır.

16.3 Resource Management With Defer

Swift guarantees that any dynamic variables on the stack are forced to clean up appropriately during an error. Moreover, the defer block gives you the ability to write cleanup codes that must be executed at the end of the function.

16.4Preventing Memory Leaks

Swift automatically eliminates leaks in error and exception circumstances with automatic reference counting (ARC) for resources managed or residing on the heap. Resources that aren't being utilized are cleaned up as needed automatically. ARC provides security like smart pointers, such as shared_ptr and unique_ptr, in C++.

```
enum FileError: Error {
    case fileNotFound
    case unreadable}
func readFile(name: String) throws -> String {
    guard name == "valid.txt" else {
        throw FileError.fileNotFound
    }
    // File reading operation
    return "File content"
}
do {
    let content = try readFile(name: "invalid.txt")
    print(content)
} catch FileError.fileNotFound {
    print("File not found!")
} catch {
    print("An unexpected error occurred: \(error)")}
```