Swift Summery

Data Types

**Int**: Represents integer values.

// let age: Int = 30

**Double**: Represents floating-point numbers (double-precision)

//let age: Int = 30

**Float**: Represents floating-point numbers(Single - precision) .

// let age: Int = 30

**Bool**: Represents a Boolean value (true or false).

// let isSwiftFun: Bool = true

**String**: Represents a sequence of characters (text).

// let greeting: String = "Hello, Swift!"

**let**: Used to declare a constant. Once a constant is assigned a value, it cannot be changed

**var**: Used to declare a variable. A variable can be modified after its initial assignment

**Collections**

**Array**: An ordered collection of values.

// let fruits: [String] = ["Apple", "Banana", "Cherry"]

**Dictionary**: A collection of key-value pairs.

let person: [String: String] = ["name": "Alice", "city": "Wonderland"] **Set**: An unordered collection of unique values.

let uniqueNumbers: Set<Int> = [1, 2, 3, 3, 2]

// Only 1, 2, 3 will be stored

Optional Types: allows variable to have value or to be nil

var name: String? = nil or name = "Bob"

**Type inference** : determine the type of a variable based on its initial value

let number = 42 // Swift infers that `number` is of type Int

var name = "Bob" // Swift infers that `name` is of type String

**Types of Operators in Swift**

**Arithmetic Operators**: Used for basic mathematical operations (+, - , / , \* , % ).

// let remainder =number % 3 // remainder is 1

**Comparison Operators**: Used to compare two values. (> , < , == , != , )

**Logical Operators**: Used for logical operations.( && , ||, ! )

**Assignment Operators**: Used to assign values (=).

**Compound Assignment Operators**: Combine an operation with assignment.(+=)

== -> called value equality is used to compare between values of two variables

=== -> called reference equality is used to compare between two instances of classes

Operator overloading -> allows you to define custom behavior for existing operators

struct Vector {

var x: Double var y: Double

static func + (left: Vector, right: Vector) -> Vector { return Vector(x: left.x + right.x, y: left.y + right.y) }

}

let result = vector1 + vector2 // Uses overloaded + operator

**Function ->** is a self-contained block of code that performs a specific task.

func multiply(\_ a: Int, by b: Int = 2) -> Int { return a \* b } // \_ for no need to pass parameter label

**Closures ->** self-contained block of code capture values by storing references to variables and constants from their surrounding context. This means that a closure can use those values even after they have gone out of scope.

func makeCounter() -> () -> Int { var count = 0 return { count += 1 return count } }

let counter = makeCounter()

print(counter()) // Prints: 1 print(counter()) // Prints: 2

**Self** -> used to refer to the current instance of a class or struct. It helps differentiate between instance local properties or parameters when they have the same name

**Abstraction** focuses on hiding the complex implementation details of a system and exposing only the necessary features. This allows users to interact with objects at a high level without needing to understand the underlying complexities.

**Inheritance**

Allows a class to copy properties, methods, and other characteristics from another class. This promotes code reusability and establishes a hierarchical relationship between classes.

The class being inherited from is called the (superclass).

The class that inherits is called the subclass (or derived).

Deinit special method allows you to clean up resources or perform final actions before the object is removed from memory dellocated.

Init special method used to setup object and can take parameters

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Class** | **Struct** | **Protocol** | **Enum** |
| **Purpose** | A blueprint for creating objects with state and behavior. | Represents complex data types that encapsulate multiple related values and behaviors for modeling data. | A blueprint that defines a set of methods, properties, and other requirements that a class, struct, or enum can adopt. |  |
| **Inheritance** | Classes support inheritance, allowing one class to inherit properties, methods, and behaviors from another class | Not support inheritance  a struct that inherits from another struct. | Not support inheritance, but can require conforming types to implement specific methods and properties. | Not support inheritance  they can adopt protocols to share functionality |
| **Type** | **Reference type** | **Value type** | **Not value or reference** | **Value type** |
| **Initialization** | **Have init & dinit** | **Have init** | **no** | **no** |
| **Implementation** | provide concrete implementations for their properties and methods | provide concrete implementations for their properties and methods | can have extensions that provide default implementations,  but those are not part of the protocol definition itself. | those are not part of the enum definition itself. |

Extensions:

allows you to add new functionality to an existing class, structure, enumeration, or protocol

cannot override existing methods in an extension

conform to a protocol by implementing the required methods and properties within the extension

protocol Describable {

var description: String { get }

}

extension Rectangle: Describable {

var description: String {

return "Rectangle with width \(width) and height \(height)"

}

}

Looping

1. For-In Loop: Iterates over sequences collections like arrays or ranges (known).
2. While Loop: Repeats a block of code while a condition is true ().
3. Repeat-While Loop: Similar to a while loop, but guarantees at least one execution to the block of code regardless the condition
4. Break: exit a loop immediately, terminating any further iterations
5. Continue: skip the current iteration of the loop and proceeds to the next iteration avoid code execution

Switch

control flow structure that allows you to execute different blocks of code based on the value of a variable.

It differs from if-else in that it is typically cleaner and more readable for handling multiple discrete values or cases

What are some best practices to follow when using switch statements in Swift?

1. Use default Cases: Always include a default case to handle unexpected values.
2. Group Cases: Combine multiple cases when they execute the same code to keep the switch concise.
3. Use Enums: When applicable, use enums for switch cases to improve code clarity and safety.
4. Document Complex Cases: If a switch statement contains complex patterns or logic, comment on its purpose to enhance readability.

Generic

enable you to define functions, types, and data structures that can work with any data type

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

struct Stack<Element> { private var items: [Element] = []

when to use -> write code that is flexible and reusable without sacrificing type safety

Key Concepts of Type Safety (static - dynamic) (Strongly Typed vs. Weakly Typed) that improve Improved Readability

Key Concepts of Error Handling in Swift

1. Error Types: In Swift, errors are represented by types that conform to the Error protocol. You can define your own error types using enums or structs.
2. Throwing Functions: Functions that can throw errors are marked with the throws keyword. When an error is thrown, it must be caught by the caller.
3. Handling Errors: You can handle errors using do-catch statements. You can also use try? for optional error handling and try! for forced error handling.

enum NetworkError: Error {

case notFound

case unauthorized

case timeout

}

func handleError(error: NetworkError) {

switch error {

case .notFound:

print("Error: Resource not found.")

case .unauthorized:

print("Error: Unauthorized access.")

case .timeout:

print("Error: Request timed out.")

}

}

// Example usage

handleError(error: .timeout)

func readFile(at path: String) throws -> String {

// Simulate file reading

if path.isEmpty {

throw FileError.notFound

}

// Simulate other logic...

return "File contents"

}

do {

let content = try readFile(at: "file.txt")

print(content)

} catch FileError.notFound {

print("Error: File not found.")

} catch {

print("An unexpected error occurred: \(error).")

}