Swift Fundamentals

1 **Quick Reference**

Swift syntax and common code snippets, perfect for quick lookups while coding.

2 **Code Examples**

Includes examples for variable declaration, control flow, and other essential Swift features.

3 **Concise Reminder**

Useful for remembering syntax nuances and best practices in Swift programming.

Reference: https://docs.swift.org/swift-book/documentation/the-swift-programming-language

Swift - Basic Syntax

Swift is a modern programming language that is used to develop iOS, macOS, watchOS, and tvOS applications. Here are some basic syntax elements:

Print Statement

The `print` statement is used to output text to the console in Swift. Here's an example:

```
print("Hello, world!")
```

Comments

You can add comments to your Swift code using `//` for single-line comments and `/* */` for multi-line comments. Here's an example:

```
// This is a single-line comment


/*
This is a multi-line comment
that spans multiple lines.
*/
```

Semicolons

Semicolons are used to separate multiple statements on the same line in Swift. However, they are not required if each statement is on a separate line. Here's an example:

```
let name = "Alice"; print("Hello, \(name)!")
```

However, it is more common in Swift to put each statement on a separate line without using semicolons, like this:

```
let name = "Alice"
print("Hello, \(name)!")
```

Variables and Constants

In Swift, you can declare variables and constants to store values. Variables are declared using the `var` keyword, followed by the variable name and its initial value (which is optional). Here's an example:

```
var myVariable = 42
```

After this line of code, `myVariable` will have the value 42. You can later change the value of `myVariable` like this:

```
myVariable = 50
```

Now `myVariable` will have the value 50. Variables can also have a type, which is specified after the variable name using a colon (:). Here's an example of declaring a variable with a type:

```
var myString: String = "Hello, world!"
```

This declares a variable called `myString` that holds a string value. Constants are declared using the `let` keyword, and their value cannot be changed once they are set. Here's an example:

```
let myConstant = 3.14
```

Now `myConstant` will always have the value 3.14.

Variables and constants are an important part of Swift programming, and understanding how they work is essential for writing effective and bug-free code.

Swift - Data Types

In Swift, there are several built-in data types that you can use to store and manipulate different kinds of values. Here are some commonly used data types:

Integers

The `Int` data type is used to represent whole numbers. It can be either positive or negative. Here's an example:

```
let age: Int = 27
```

Floats and Doubles

The `Float` and `Double` data types are used to represent decimal numbers. The main difference between them is the precision. Floats have a precision of 6 decimal places, while doubles have a precision of 15 decimal places. Here's an example:

```
let pi: Float = 3.14
let distance: Double = 10.5
```

Boolean

The `Bool` data type is used to represent boolean values. It can be either `true` or `false`. Here's an example:

```
let isSunny: Bool = true
```

Strings

The `String` data type is used to represent text. You can create a string by enclosing it in double quotes. Here's an example:

```
let name: String = "John Doe"
```

Arrays

The `Array` data type is used to represent an ordered collection of values. The values in an array must all be of the same type. Here's an example:

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

Dictionaries

The `Dictionary` data type is used to represent a collection of key-value pairs. Each value in a dictionary is associated with a unique key. Here's an example:

```
let person: [String: Any] = ["name": "John Doe", "age": 27, "isStudent": true]
```

Tuples

The `Tuple` data type is used to group multiple values into a single compound value. The values in a tuple can be of different types. Here's an example:

```
let coordinates: (Double, Double) = (3.5, 7.2)
```

Optionals

An optional type in Swift is used to represent a value that may or may not exist. It allows you to handle the absence of a value in a safe and controlled way. Here's an example:

```
let optionalValue: Int? = 42
```

These are just a few examples of the data types available in Swift. Each data type has its own set of operations and behaviors. Understanding data types, including optionals, is essential for writing effective and bug-free Swift code.

Swift - Operators

In Swift, operators are symbols or words that perform operations on values or variables. There are several different types of operators in Swift, including arithmetic, comparison, logical, assignment, and range operators.

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on numbers. The most common arithmetic operators in Swift are:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `/` (division)
- `%` (remainder)

Here's an example:

```
let x = 10
let y = 3
let z = x % y // z is 1
```

Comparison Operators

Comparison operators are used to compare two values. The result of a comparison is always a boolean value (`true` or `false`). The most common comparison operators in Swift are:

- `==` (equal to)
- `!=` (not equal to)
- `>` (greater than)
- `<` (less than)
- `>=` (greater than or equal to)
- `<=` (less than or equal to)

Here's an example:

```
let a = 5
let b = 10
let result = a < b // result is true
```

Logical Operators

Logical operators are used to combine boolean values or expressions. The most common logical operators in Swift are:

- `&&` (logical AND)
- `||` (logical OR)
- `!` (logical NOT)

Here's an example:

```
let x = 5
let y = 10
let z = 3
let result = (x < y) && (z < y) // result is true
```

Assignment Operators

Assignment operators are used to assign values to variables. The most common assignment operator in Swift is the `=` operator. There are also compound assignment operators, which combine an arithmetic operator with the assignment operator. Here's an example:

```
var x = 10
x += 5 // x is now 15
x *= 2 // x is now 30
x /= 5 // x is now 6
```

Range Operators

Range operators are used to create ranges of values. There are two types of range operators in Swift:

- `...` (closed range operator): creates a range that includes both the start and the end value.
- `..<` (half-open range operator): creates a range that includes the start value but not the end value.
- `>..<` (one-sided range operator): creates a range that includes all values greater than or equal to the start value.

Here's an example:

```
let closedRange = 1...5 // contains values 1, 2, 3, 4, 5
let halfOpenRange = 1..<5 // contains values 1, 2, 3, 4
let oneSidedRange = 5..< // contains values 5, 6, 7, 8, ...
```

Operators are used extensively in Swift programming, and understanding how they work is essential for writing effective code.

Swift - Decision Making

In Swift, decision-making is an essential part of programming. You can use conditional statements like `if`, `else if`, and `switch` to control the flow of your code based on different conditions. These statements allow you to execute specific blocks of code depending on the evaluation of certain expressions. Making informed decisions is key to writing efficient and dynamic Swift code.

Conditional Statements

The `if` statement is used to execute a block of code if a certain condition is true. It can be followed by an optional `else if` block and an optional `else` block. Here's an example:

```
let number = 10


if number > 0 {
    print("The number is positive")
} else if number < 0 {
    print("The number is negative")
} else {
    print("The number is zero")
}
```

The `switch` statement is used to compare a value against multiple possible matching patterns. It provides a concise way to write complex conditional statements. Here's an example:

```
let grade = "B"


switch grade {
case "A":
   print("Excellent!")
case "B":
   print("Good job!")
case "C":
   print("You can do better.")
default:
   print("You need to study more.")
}
```

These conditional statements help you control the flow of your program and make decisions based on different conditions.

Swift - Enumerations

Enumerations, or enums, are a powerful feature in Swift that allow you to define a group of related values. They are a way to represent a finite set of possibilities, making your code more readable and expressive. Enums can have associated values and methods, adding further flexibility to your code.

With enums, you can define custom types that represent different cases or options. For example, you can define an enum to represent the days of the week:

```
enum Day {
    case monday
    case tuesday
    case wednesday
    case thursday
    case friday
    case saturday
    case sunday
}
let today = Day.monday
print("Today is \(today)") // Output: Today is monday
```

In this example, we define an enum called `Day` with seven cases representing the days of the week. We can create an instance of the enum by assigning one of its cases to a variable or constant. In this case, `today` is assigned the value `Day.monday`.

Enums in Swift allow you to easily handle different cases and make your code more readable and maintainable.

Swift - Loops

Loops are used to repeat a block of code multiple times. In Swift, you can use different types of loops to control the flow of your program. Let's explore three common types of loops in Swift:

1. For Loop

The `for-in` loop allows you to iterate over a sequence, such as an array, range, or string. Here's an example:

```
let numbers = [1, 2, 3, 4, 5]


for number in numbers {
    print(number)
}
```

2. While Loop

The `while` loop repeats a block of code as long as a certain condition is true. Here's an example:

```
var count = 0


while count < 5 {
    print(count)
    count += 1
}
```

3. Repeat-While Loop

The repeat-while loop is similar to the while loop, but it checks the condition at the end of the loop. This guarantees that the loop will run at least once. Here's an example:

```
var count = 0


repeat {
    print(count)
    count += 1
} while count < 5
```

These loops are powerful tools for controlling the flow of your program and performing repetitive tasks. They can help you iterate over collections, validate input, and perform other operations.

Functions in Swift

In Swift, a function is a reusable block of code that performs a specific task when called. It helps in organizing code into logical pieces and promotes code reusability. Here's the basic syntax of a function:

```
func functionName(parameters) -> ReturnType {
    // code to be executed
    return value
}
```

The `functionName` is the name of the function, `parameters` are the inputs that the function can take, and `ReturnType` is the type of value that the function returns. Inside the function, you can write the code to perform a specific task.

For example, you can define a function that adds two numbers:

```
func addNumbers(number1: Int, number2: Int) -> Int {
    let sum = number1 + number2
    return sum
}
```

Swift - Closures

A closure is a self-contained block of code that can be passed around and used in your program. Closures are similar to functions, but they are defined inline and do not require a name. Here's an example:

```
// Define a closure that adds two numbers
let add = { (a: Int, b: Int) -> Int in
    return a + b
}
let result = add(2, 3)
print(result) // Output: 5
```

In this example, we define a closure called `add` that takes two integers as input and returns their sum. We then call the closure and pass in the values 2 and 3. The closure adds the numbers and returns the result, which we print to the console.

Closures are a powerful feature of Swift and are used extensively in the language and its standard library. They can be used for a variety of tasks, such as sorting collections, filtering data, and performing asynchronous operations.

Swift - Structures and Classes

In Swift, structures and classes are used to define custom data types that can be used to encapsulate related data and behavior. They are both capable of defining properties and methods, but there are some key differences between them:

Structures

A structure is a value type that is typically used to encapsulate small pieces of data. When you assign a structure instance to a new constant or variable, a copy of the instance is created. Here's an example:

```
struct Point {
    var x: Double
    var y: Double
}
var p1 = Point(x: 0, y: 0)
var p2 = p1

p2.x = 1
print(p1.x) // Output: 0
print(p2.x) // Output: 1
```

In this example, we define a `Point` structure with two properties: `x` and `y`. We create an instance of the structure and assign it to `p1`. We then assign `p1` to `p2`, which creates a copy of the instance. When we modify the `x` property of `p2`, `p1` is not affected, since it is a separate instance.

Classes

A class is a reference type that is typically used to encapsulate more complex pieces of data. When you assign a class instance to a new constant or variable, a reference to the instance is created. Here's an example:

```
class Person {
    var name: String

    init(name: String) {
        self.name = name
    }
}
var p1 = Person(name: "Alice")
var p2 = p1

p2.name = "Bob"
print(p1.name) // Output: "Bob"
print(p2.name) // Output: "Bob"
```

In this example, we define a `Person` class with a `name` property and an initializer. We create an instance of the class and assign it to `p1`. We then assign `p1` to `p2`, which creates a reference to the same instance. When we modify the `name` property of `p2`, `p1` is also affected, since they both point to the same instance.

Structures and classes each have their own use cases, and it's important to understand the differences between them to choose the right one for your needs.

Swift Methods

In Swift, methods are functions that are associated with a particular type, such as a class, structure, or enumeration. They allow you to define behavior and functionality specific to that type. Methods are an essential part of object-oriented programming and provide a way to interact with and manipulate instances of a type.

There are two types of methods in Swift:

- **Instance Methods:** Instance methods are associated with an instance of a type. They can access and modify the properties and other instance methods of the type. Instance methods are defined within the context of a class, structure, or enumeration.
- **Static Methods:** Static methods belong to the type itself rather than an instance of the type. They can be called on the type directly, without the need to create an instance. Static methods are defined using the `static` keyword.

Here's an example that demonstrates how to define and use instance and static methods:

```
class MyClass {
    var value: Int

    init(value: Int) {
        self.value = value
    }

    func instanceMethod() {
        print("Instance method called. Value: \(value)")
    }

    static func staticMethod() {
        print("Static method called.")
    }
}

let myObject = MyClass(value: 10)
myObject.instanceMethod() // Output: "Instance method called. Value: 10"
MyClass.staticMethod() // Output: "Static method called."
```

In this example, we define a class `MyClass` with an instance property `value` and two methods: `instanceMethod` and `staticMethod`. The `instanceMethod` can access and modify the `value` property of the instance, while the `staticMethod` is called on the type itself.

Methods in Swift provide a way to encapsulate behavior within types and enable code reusability. They play a crucial role in modeling real-world objects and implementing functionality specific to those objects.