SWIFT PROGRAMMING LANGUAGE

By: Shreeya Sharda, Nina Ebensperger, Julia Smith, Sumairah Rahman, Ngoc Phan

TABLE OF CONTENTS

We have split the presentation into 2 Parts:

01

Details about different aspects of the Swift language

(e.g., binding, data types, support to OO programming, concurrency, exception handling)

02

Our project proposal

INTRODUCTION

Swift is a high-level, open-source, dynamically-typed, compiled programming language specifically designed for creating safe, rich, and highly performant iOS and MacOS apps.

NAMES

```swift
project_example.swift > ...
1   let `class`: String = "This is a variable named 'class'"
2
3   let π: Double = 3.14159
4   let こんにちは: String = "Hello"
5
6
7   var numberOfApples: Int = 10
8
9   func calculateSum() -> Int {
10      return numberOfApples + 5
11  }
12
13  let result: Int = calculateSum()
14  print(result) // Output: 15
15
16
17  class FruitBasket {
18      // class definition
19  }
20
```

Case-sensitive

- **VariableName** distinct from **variablename**

Reserved words vs Keywords

- Variable or function names
- Ex. **class**, **func**, **let**

Allows the use of a wide range of characters in identifiers

- Multiple languages
- Mathematical equations

Naming conventions

- Variables and Functions:
  - lowerCamelCase
- Types (Classes, Structs, Enums):
  - UpperCamelCase

BINDING

Association between an identifier (variable name) and its value or type

Static Binding (Fixed at Compile Time)

- Ex: type of “count” is known before the program runs.
- Explicit

Dynamic Binding (Happens at Runtime)

- Ex: type or value of “data” can change while the program runs.
- Implicit

project_example.swift > ...

```swift
var count: Int = 5

func add(a: Int, b: Int) -> Int {
    return a + b
}

var data: Any = "Hello"
data = 42

protocol Vehicle {
    func move()
}

class Car: Vehicle {
    func move() { print("Car moves") }
}

var transport: Vehicle = Car()
transport.move()
```

BINDING CONT.

```
project_example.swift > ...
21  var count: Int = 5
22
23  func add(a: Int, b: Int) -> Int {
24      return a + b
25  }
26
27  var data: Any = "Hello"
28  data = 42
29
30  protocol Vehicle {
31      func move()
32  }
33
34  class Car: Vehicle {
35      func move() { print("Car moves") }
36  }
37
38  var transport: Vehicle = Car()
39  transport.move()
40
```

```
@dynamicMemberLookup
class DynamicObject {
    private var storage: [String: Any] = [:]

    subscript(dynamicMember key: String) -> Any? {
        get { return storage[key] }
        set { storage[key] = newValue }
    }
}

var obj: DynamicObject = DynamicObject()
obj.name = "Swift"
print(obj.name!)
```

Late Binding (Used in Polymorphism & Protocols)

- Function executed depends on the object's actual type at runtime. (ex: "transport)
- Implicit

Dynamic Member Lookup (Like JavaScript/PHP Objects)

- Allows accessing unknown properties dynamically. (ex: "DynamicObject"
- Implicit

SCOPE

```swift
var globalVar: Int = 100 // Global scope (Accessible everywhere)

func printGlobalVar() {
    print(globalVar)
}

printGlobalVar()

func exampleFunction() {
    var localVar: Int = 50
    print(localVar)
}

exampleFunction()
print(localVar)
```

```swift
if true {
    let blockVar: String = "Inside block"
    print(blockVar) // Works inside block
}

print(blockVar) // Not accessible outside the block

var name: String = "Outer"

func example() {
    var name: String = "Inner" // Shadows the global 'name'
    print(name) // Output: Inner
}

example()
print(name) // Output: Outer
```

- **Scope** determines where a variable is accessible in a program
- A variable's lifetime depends on its scope

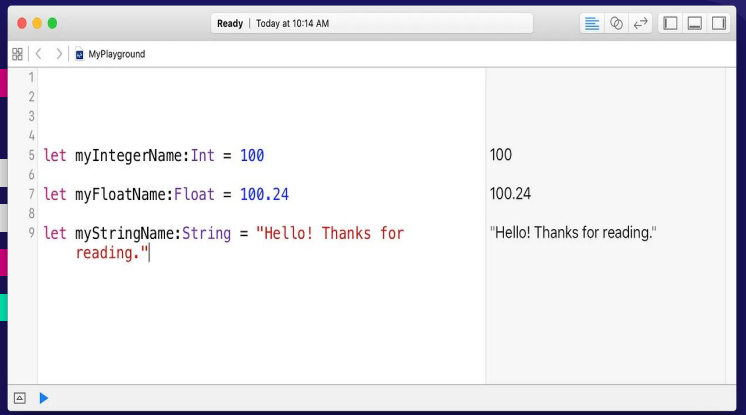
- Global Scope
  - A variable declared outside any function is accessible everywhere
- Local Scope
  - Variables declared inside a function exist only in that function
- Block Scope
  - Variables inside {} exist only within that block
- Nested Scopes & Shadowing
  - A variable in an inner scope can hide a variable in an outer scope.

SCOPE VS LIFETIME IN SWIFT

- **Scope**: Determines where a variable can be accessed.
- **Lifetime**: Determines how long a variable exists in memory.

```swift
var globalVar = 100 // Global scope (Lifetime = entire program)

func exampleFunction() {
    var localVar: Int = 50
    print(localVar)
} // localVar is destroyed here

exampleFunction()
print(globalVar) // Works
print(localVar)  // Error! localVar no longer exists
```

DATA TYPES



- **Basic Data Types**: Int, Double, Float, String, Bool
- **Optionals:** These values can either hold a value or be nil (aka have no value). They use a question mark
  - Syntax Ex: String?
- **Dictionaries**: key-value pairs
  - Syntax Ex: let person = [“name”: Alice]
- **Class Types** – creating objects and giving them properties
- **Tuples** – allows you to group multiple values together

DATA TYPES – 2 FEATURES

```
let someTuple: (Double, Double) = (3.14159, 2.71828)
func someFunction(a: Int) { /* ... */ }
```

Additional Example (2)

- **Type Inference** – You can declare a variable and assign a value without explicitly giving a type
  - Example (1): let age = 25 //Implicity an int
- Type Safety – This ensures that an already established type cannot be assigned a value of another type.

BASIC ASSIGNMENT STATEMENTS

```
let b = 10
var a = 5
a = b
// a is now equal to 10
```

```
var red, green, blue: Double
```

- Simple assignment done using = statements
- Keywords for variable assignments are let (immutable) and var (mutable)
- Single and multiple assignments supported
- Untyped and typed assignment supported
- Assignment operator does not return anything, so cannot be used as an expression

More About Assignments

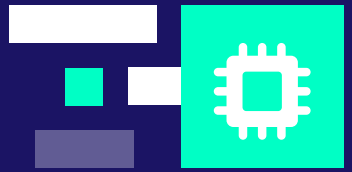
Conditional Assignment Operators

- Conditional assignments supported using **ternary**
  Example: (bool1 ? true : false)
- Null coalescing assignment also supported using ??
  Example: someOptional ?? “ “

Compound Assignment Operators

- Compound assignments to string and number supported using +=
- Example: newNumber+=5
- Numbers can be combined with strings

EXPRESSIONS

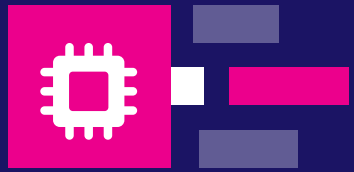


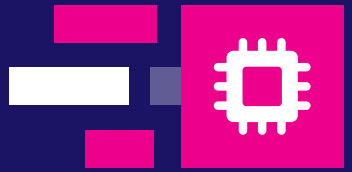
Prefix

Combines optional prefix operator with expression
Ex: +(x)

Infix

Combines infix operator with left and right-hand arguments
Ex: (x)+(y)



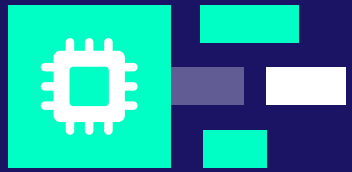


Primary

Basic expression that can be combined with other tokens to make the other three types

Postfix

Combines postfix operator/syntax to an expression
Ex: (x)+



SO MANY EXPRESSIONS

Math

Addition, subtraction, division, multiplication modulo, unary, range

Boolean

true/false, if/else, AND, NOT, OR, comparison operators, optional chaining

Class

super, self, explicit and implicit value/getter/setter accessors, typecasting

String

Wildcard patterns, print comma-separated list

Function

Function call, lambda function

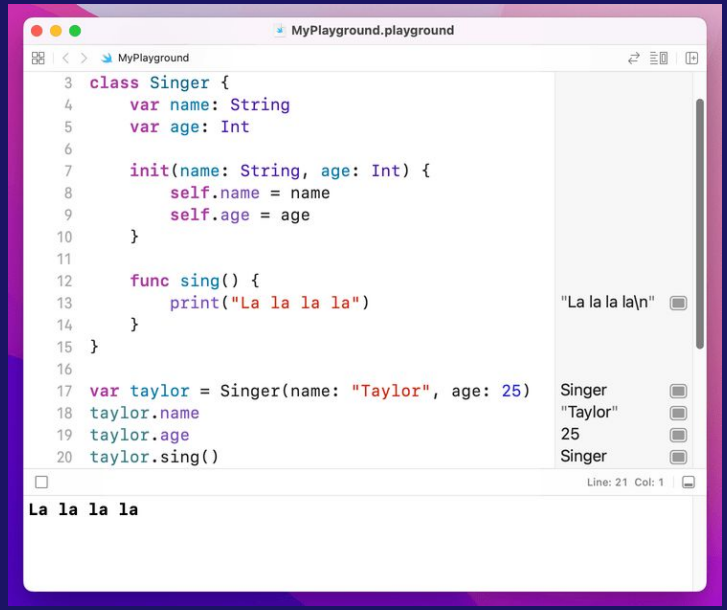
Control

try, await

AND YOU CAN OVERLOAD

OO Support

- **Classes and objects**: classes and structures
- **Inheritance**: single inheritance
- **Encapsulation**: private (in class), fileprivate (in file), internal (in module), public (anywhere), open (anywhere, subclassing outside module)
- **Polymorphism**: Subclass method override
- **Abstraction**: “Protocols” define abstract behavior for one or more classes



Concurrency

```
func listPhotos(inGallery name: String) async -> [String] {
    let result = // ... some asynchronous networking code ...
    return result
}
```

- Mix of parallel and asynchronous code is known as concurrency
- Allows simultaneous execution of asynchronous tasks
- Writing parallel and asynchronous code is supported by Swift by default
- The **async** keyword is added to a function or method's declaration after its parameters to indicate that it is asynchronous.

Concurrency

```
let firstPhoto = await downloadPhoto(named: photoNames[0])
let secondPhoto = await downloadPhoto(named: photoNames[1])
let thirdPhoto = await downloadPhoto(named: photoNames[2])


let photos = [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```

- Multiple bits of code running simultaneously is referred to as parallel coding.
- Parallel or asynchronous programming offers more scheduling freedom, but at the expense of greater complexity.
- You can identify issues at compile time by utilising Swift's language-level support for concurrency in code that must be concurrent.

Exception Handling vs. Error Handling

Before we see how Swift handles these two concepts, let's see how they differ

Exception

Detects errors that can halt the execution of the program.

Error

Detects errors for recoverable conditions (e.g., errors that will likely not stop the program and will not ruin the user's experience.

Exception Handling

- **The most common way of dealing with exceptions in Swift is to** explicitly mark functions with the throws keyword and/or try-catch blocks

```
func checkPassword(_ password: String) throws -> Bool {
    if password == "password" {
        throw PasswordError.obvious
    }

    return true
}
```

Error Handling

3 different ways of using “try” (most common mechanism) to handle recoverable errors.

1. try (requires do-catch)
2. try? (returns nil on failure)
3. try! (forces execution if an error occurs)

```
let result = try! someFunctionThatThrows()
```

```
let result = try? someFunctionThatThrows()
```

Another Example for Error Handling

- **Do-catch** is used within functions that use the **throws** keyword
- do-block is first executed, and if an error is found within the do-block, then code inside catch block is executed

```
do {
    // Create audio player object
    audioPlayer = try AVAudioPlayer(contentsOf: soundURL)

    // Play the sound
    audioPlayer?.play()
}
catch
{
    // Couldn't create audio player object, log the error
    print("Couldn't create the audio player for file\(soundFilename)")
}
```

What happens after you fix the Error?

```
func writeLog() {
    let file = openFile()
    defer { closeFile(file) }

    let hardwareStatus = fetchHardwareStatus()
    guard hardwareStatus != "disaster" else { return }
    file.write(hardwareStatus)

    let softwareStatus = fetchSoftwareStatus()
    guard softwareStatus != "disaster" else { return }
    file.write(softwareStatus)

    let networkStatus = fetchNetworkStatus()
    guard neworkStatus != "disaster" else { return }
    file.write(networkStatus)
}
```

- **Problem**: We need to clean up the stack after the error has been fixed.
- **Solution**: Implement the Keyword **defer**.
  - Cleans up the resources (a.k.a stack unwinding) used during exception/error handling
- NOTE : Defer statements run no matter what else happens

Functional Programming Support

- Functions are first-class citizens in Swift: can be assigned, passed as args, and returned
- Pure functions, lambdas, and closures are supported
- Function composition is supported
- **Protocol** programming is supported (behavior-based instead of class-based)
- Swift offers many built-in functional programming functions like .filter, .map, .flatMap, .sorted

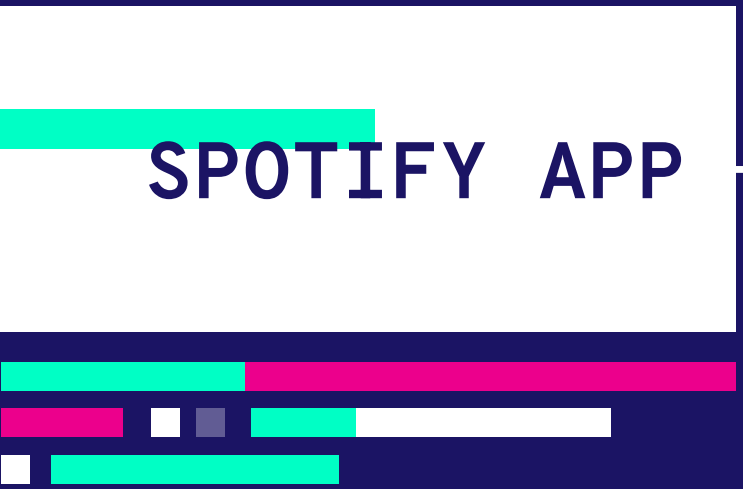
```
let add: (Int, Int) -> Int = { $0 + $1 }
let result = add(2, 3) // result is 5

func performOperation(_ a: Int, _ b: Int, operation: (Int, Int) -> Int) -> Int {
    return operation(a, b)
}

let sum = performOperation(4, 5, operation: add) // sum is 9
```


Part 2: Project Proposal

OUR PROJECT OBJECTIVE

A Spotify-like music app that leverages Swift's interoperability with iOS media capabilities to create a simple and zen experience

Features



BROWSE

Fetch songs and display their titles, album covers, and artists using the Spotify API

PLAY

Leverage iOS's media capabilities to add music controls (e.g., volume)

INTERACT

Like songs and add them to playlists

FIND

Use a cosine recommendation system to find more songs!

Technology

UI

SwiftUI
Kingfisher
Xcode (IDE for app)

Audio Handling

AVFoundation and AVPlayer

Media Controls

MediaPlayer

Data Management

CoreData, UserDefaults, FileManager

State Management

@State annotations, Combine

APIs

URLSession, JSON mapper Codable, and Spotify API/Apple Music API

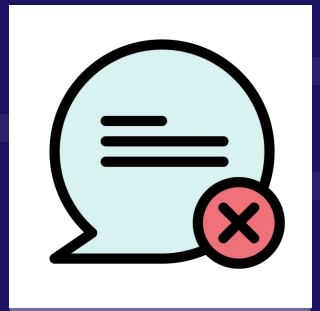
Constraints

No inter-user interaction

Users will not be able to interact with one another

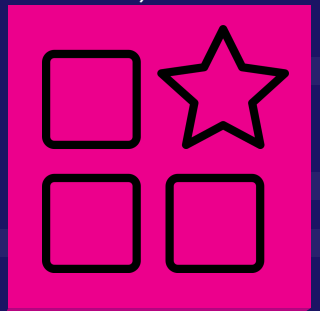
API Constraints

Music APIs may limit info, streaming, request numbers

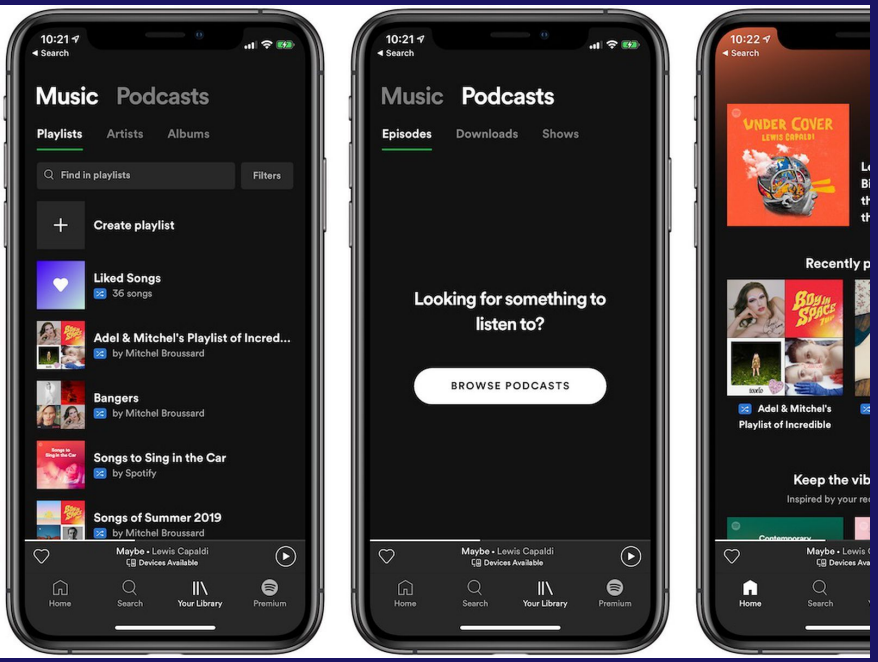


Fewer Features

No Spotify QOL features like daily mix, etc.



PROJECT APPLICATION



- **RELEVANCE**: Students are always looking for affordable ways of listening to music which is VERY helpful during times of stress and/or productivity
- **APPLICATION**: STUDENTS can use our app to go to the gym, unwind, or study
- **APPLICATION**: STUDENTS can use our application as a development tool or starting point for their own projects

THANK YOU

We have Swiftly reached the end

CITATIONS

“Apple Developer Documentation.” *Developer.apple.com*, Apple, developer.apple.com/documentation/swift/.

Bakshioye, Medium. “Static vs. Dynamic Dispatch in Swift - A Decisive Choice.” Medium, 2018, https://medium.com/@bakshioye/static-vs-dynamic-dispatch-in-swift-a-decisive-choice-cece1e872d. Accessed 19 Feb. 2025.

C, Chris. “CodeWithChris.” *CodeWithChris*, 3 Sept. 2019, codewithchris.com/swift-try-catch/. Accessed 19 Feb. 2025.

denoni. “GitHub - Denoni/SpotifyClone: An IOS App That Visually Clones Spotify’s App and Consumes the Official Spotify’s Web API to Show(and Play) Songs, Podcasts, Artists and More.” *GitHub*, 2021, github.com/denoni/SpotifyClone. Accessed 19 Feb. 2025.

Dheeraj M. “Stack vs. Heap Memory Management in iOS.” Medium, 2020, https://dheerajm.medium.com/stack-vs-heap-memory-management-in-ios-9d5a50724b97. Accessed 19 Feb. 2025.

“Documentation.” *Docs.swift.org*, docs.swift.org/swift-book/documentation/the-swift-programming-language/.

“Pros and Cons of Swift Programming Language.” *AltexSoft*, 12 Sept. 2021, www.altexsoft.com/blog/the-good-and-the-bad-of-swift-programming-language/.

Sedlacek, Magnus. “Functional Programming in Swift: A Comprehensive Guide.” *Ada Beat*, 9 Nov. 2023, adabeat.com/fp/functional-programming-in-swift-a-comprehensive-guide/. Accessed 19 Feb. 2025.

sreesairaghava. “GitHub - Sreesairaghava/Spotify: Spotify Clone, Using Swift, UIKit with Spotify WebAPI. Network Manager, Haptics, Authentication Manager, Programmatic UI, Views, TableView, Compositional Layout. Used First Party Official API from Spotify.” *GitHub*, 2021, github.com/sreesairaghava/Spotify. Accessed 19 Feb. 2025.

twostraws, Paul Hudson. “Parsing JSON Using the Codable Protocol - a Free Hacking with Swift Tutorial.” *Hacking with Swift*, 2019, www.hackingwithswift.com/read/7/3/parsing-json-using-the-codable-protoco. Accessed 19 Feb. 2025.

“Writing Throwing Functions - a Free Hacking with Swift Tutorial.” *Hacking with Swift*, 2020, www.hackingwithswift.com/sixty/5/8/writing-throwing-functions. Accessed 19 Feb. 2025.

“The Basics - Swift Language Guide.” Docs.swift.org, Apple, https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html. Accessed 19 Feb. 2025.

“Automatic Reference Counting (ARC).” Docs.swift.org, Apple, https://docs.swift.org/swift-book/documentation/the-swift-programming-language/automaticreferencecounting/. Accessed 19 Feb.

“Swift Type Inference.” Developer.apple.com, Apple, https://developer.apple.com/documentation/swift/type_inference. Accessed 19 Feb. 2025.

2025

01

SWIFT LANGUAGE

The TLDR of Swift

2025

INTRODUCTION

Swift is a high-level, open-source, dynamically-typed, compiled programming language specifically designed for iOS and Mac apps. It’s specifically designed to create rich and performant apps within the Apple ecosystem.

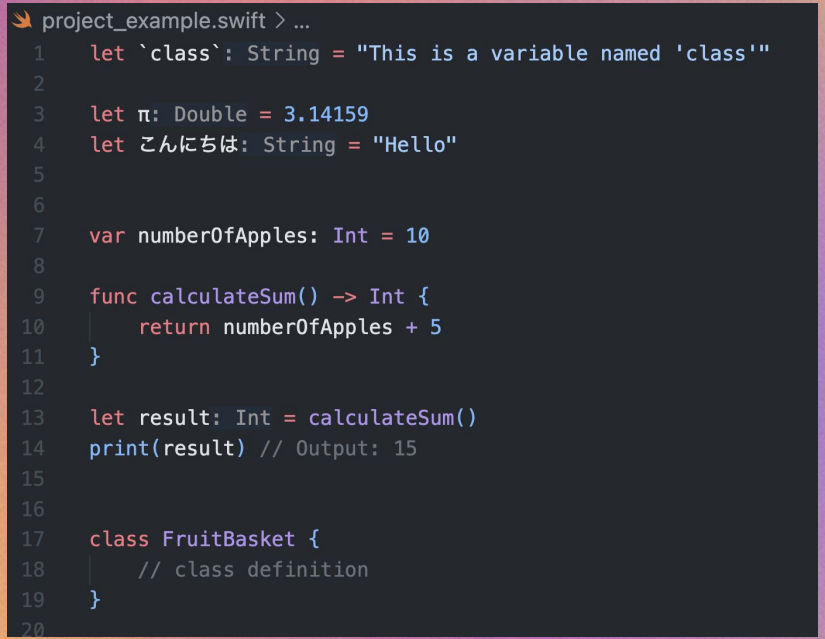
Names

Case-sensitive
- **VariableName** is distinct from **variablename**

Reserved words vs Keywords
- Variable or function names
- Ex. **class**, **func**, **let**

Swift allows the use of a wide range of characters in identifiers
- Multiple languages
- Mathematical equations

Naming conventions
- Variables and Functions:
  - lowerCamelCase
- Types (Classes, Structs, Enums):
  - UpperCamelCase



Binding

project_example.swift > ...

```swift
var count: Int = 5

func add(a: Int, b: Int) -> Int {
    return a + b
}

var data: Any = "Hello"
data = 42

protocol Vehicle {
    func move()
}

class Car: Vehicle {
    func move() { print("Car moves") }
}

var transport: Vehicle = Car()
transport.move()
```

```swift
@dynamicMemberLookup
class DynamicObject {
    private var storage: [String: Any] = [:]

    subscript(dynamicMember key: String) -> Any? {
        get { return storage[key] }
        set { storage[key] = newValue }
    }
}

var obj: DynamicObject = DynamicObject()
obj.name = "Swift"
print(obj.name!)
```

The association between an identifier (variable name) and its value or type.

Static Binding (Fixed at Compile Time)

- The type of count is known before the program runs.
- Explicit

Dynamic Binding (Happens at Runtime)

- The type or value can change while the program runs.
- Implicit

Late Binding (Used in Polymorphism & Protocols)

- The function executed depends on the object's actual type at runtime.
- Implicit

Dynamic Member Lookup (Like JavaScript/PHP Objects)

- Allows accessing unknown properties dynamically.
- Implicit

```swift
var globalVar: Int = 100 // Global scope (Accessible everywhere)

func printGlobalVar() {
    print(globalVar)
}

printGlobalVar()

func exampleFunction() {
    var localVar: Int = 50
    print(localVar)
}

exampleFunction()
print(localVar)
```

```swift
if true {
    let blockVar: String = "Inside block"
    print(blockVar) // Works inside block
}

print(blockVar) // Not accessible outside the block

var name: String = "Outer"

func example() {
    var name: String = "Inner" // Shadows the global 'name'
    print(name) // Output: Inner
}

example()
print(name) // Output: Outer
```

Scope

Scope determines where a variable is accessible in a program.

A variable’s lifetime depends on its scope.

Global Scope

- A variable declared outside any function is accessible everywhere

Local Scope

- Variables declared inside a function exist only in that function

Block Scope

- Variables inside {} exist only within that block

Nested Scopes & Shadowing

- A variable in an inner scope can hide a variable in an outer scope.

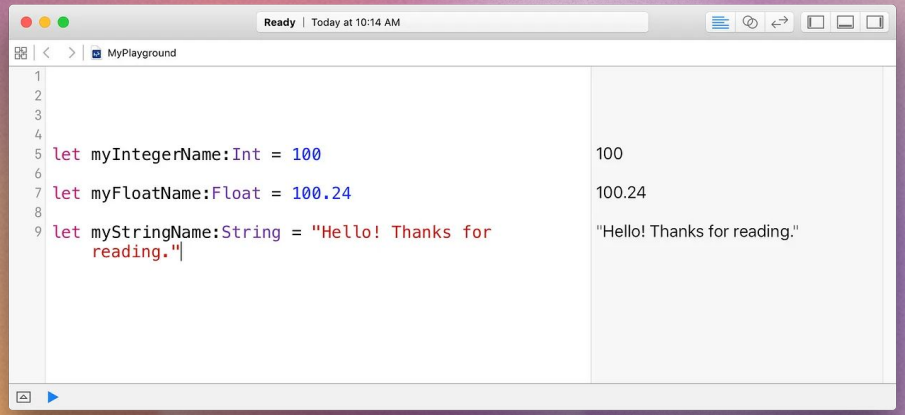
Scope vs Lifetime in Swift

- **Scope**: Determines where a variable can be accessed.
- **Lifetime**: Determines how long a variable exists in memory.

```swift
var globalVar = 100 // Global scope (Lifetime = entire program)

func exampleFunction() {
    var localVar: Int = 50
    print(localVar)
} // localVar is destroyed here

exampleFunction()
print(globalVar) // Works
print(localVar)  // Error! localVar no longer exists
```

Data Types

- Swift is by default a dynamically typed language, but supports mandatory type assignments and type hints
- Number types: integer, double, float
- Booleans supported
- Both strings and characters (chars) supported as types
- Collection types include: Arrays (heap dynamic/flexible with slice/functional utilities), Set (unique collection), and Dictionary (object)



Data Types

20
XX

- There are several different data types in the Swift programming language. They include the following:
  - Constants and Variables - these must be declared before they are used in the program (similar to the way variables work in Python). Examples of syntax include:
    - A main difference between constants and variables is that the value of constants cannot be changed once you have declared it. However, the value of variables can be changed through the program.
    -

Basic Assignment Statements

- Simple assignment done using = statements
- Keywords for variable assignments are let (immutable) and var (mutable)
- Single and multiple assignments supported
- Assignment operator does not return anything, so cannot be used as an expression

The *assignment operator* (a = b) initializes or updates the value of a with the value of b:

```
let b = 10
var a = 5
a = b
// a is now equal to 10
```

If the right side of the assignment is a tuple with multiple values, its elements can be decomposed into multiple constants or variables at once:

```
let (x, y) = (1, 2)
// x is equal to 1, and y is equal to 2
```

Conditional Assignments

- Conditional assignments supported using ternary
  Example: (bool1 ? true : false)
- Null coalescing assignment also supported using ??
  Example: someOptional ?? “ “

Compound Assignment Operators

- Compound assignments to string and number supported using +=
- Example: newNumber+=5
- Numbers can be combined with strings

Basic Expressions

- Math: Supports PEMDAS order addition, subtraction, division, multiplication for all number types, modulo on integers
- Base arithmetic operations are left-associative
- Supports bracketed if/else, logical OR (||), logical AND (&&), logical NOT (!), and relational operators (including shallow/deep object equality)
- Supports C++-like bitwise operators

```
let name = "world"
if name == "world" {
    print("hello, world")
} else {
    print("I'm sorry \(name), but I don't recognize you")
}
// Prints "hello, world", because name is indeed equal to "world".
```

Comparison Operators

Swift supports the following comparison operators:

- Equal to (a == b)
- Not equal to (a != b)
- Greater than (a > b)
- Less than (a < b)
- Greater than or equal to (a >= b)
- Less than or equal to (a <= b)

> **Note**
> Swift also provides two *identity operators* (=== and !==), which you use to test whether two object references both refer to the same object instance. For more information, see Identity Operators.

Expressions: The Nitty-Gritty

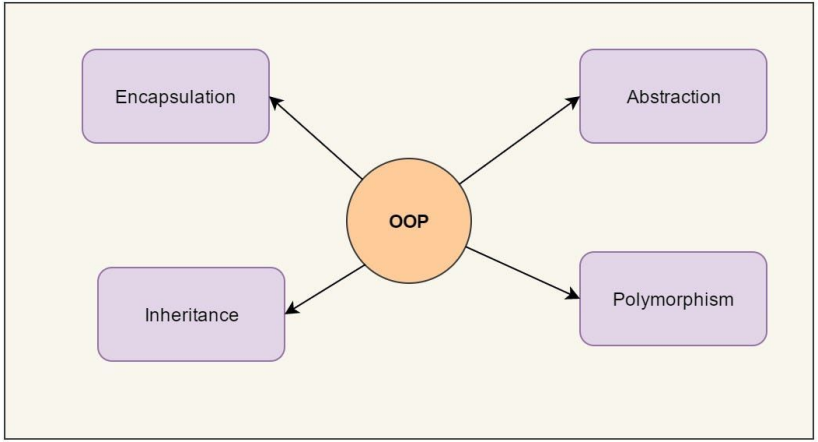
- Operator overloading: Swift allows custom overloads to be applied to base operators, defining fix, associativity, etc.
- Typecasting: Parent classes can be downcast to subclasses, can use Any/AnyObject generic type, number->string and string->number classes
- Short circuiting: used to evaluate boolean expressions

```swift
prefix operator +++

extension Int {
    static prefix func +++ (number: inout Int) -> Int {
        number += 2
        return number
    }
}

var myNumber = 1
let incrementedNumber = +++myNumber // `myNumber` is now 3
```

OO Programming Support

Swift fully supports **Object-Oriented Programming (OOP)** with:

- Classes and objects
- Encapsulation (private, public, internal)
- Inheritance
- Polymorphism
- Abstraction (protocols)



Four Pillars of Object Oriented Programming

Classes and Objects

```swift
class Person {
    var name: String
    var age: Int

    // Initializer (Constructor)
    init(name: String, age: Int) {
        self.name = name
        self.age = age
    }

    func greet() {
        print("Hello, my name is \(name) and I am \(age) years old.")
    }
}

// Creating an instance of the class
let person1 = Person(name: "Alice", age: 25)
person1.greet()  // Output: Hello, my name is Alice and I am 25 years old.
```

Inheritance

- Swift allows one class to inherit from another
- Only single inheritance–class can only inherit from one superclass

```swift
class Employee: Person {
    var jobTitle: String

    init(name: String, age: Int, jobTitle: String) {
        self.jobTitle = jobTitle
        super.init(name: name, age: age)
    }

    override func greet() {
        print("Hello, my name is \(name), I am \(age) years old, and I work as a \(jobTitle).")
    }
}

// Creating an instance of the subclass
let employee1 = Employee(name: "Bob", age: 30, jobTitle: "Software Engineer")
employee1.greet()
// Output: Hello, my name is Bob, I am 30 years old, and I work as a Software Engineer.
```

Encapsulation

Encapsulation is achieved through **access control modifiers**:

- private: Accessible only within class
- fileprivate: Accessible within same file
- internal: Default, accessible within module
- public: accessible anywhere
- open: Like public, but also allows subclassing outside the module

```
class BankAccount {
    private var balance: Double

    init(balance: Double) {
        self.balance = balance
    }

    func deposit(amount: Double) {
        balance += amount
        print("Deposited \(amount). New balance: \(balance)")
    }

    func getBalance() -> Double {
        return balance
    }
}

let account = BankAccount(balance: 1000)
account.deposit(amount: 500)
// account.balance = 2000  ❌ ERROR: Cannot access private property
print("Current balance: \(account.getBalance())")
```

Polymorphism

- Swift supports method overriding in subclasses.

```swift
class Animal {
    func makeSound() {
        print("Some generic animal sound")
    }
}

class Dog: Animal {
    override func makeSound() {
        print("Woof! Woof!")
    }
}

let myDog = Dog()
myDog.makeSound()  // Output: Woof! Woof!
```

Abstraction

- Swift uses **protocols** to define abstract behavior that can be adopted by multiple classes.

```swift
class Vehicle {
    var name: String
    var speed: Double

    init(name: String, speed: Double) {
        self.name = name
        self.speed = speed
    }

    func move() {
        print("\(name) is moving at \(speed) km/h.")
    }
}

// Car is a type of Vehicle
class Car: Vehicle {
    var fuelType: String

    init(name: String, speed: Double, fuelType: String) {
        self.fuelType = fuelType
        super.init(name: name, speed: speed)
    }

    override func move() {
        print("\(name) (Car) is driving at \(speed) km/h using \(fuelType).")
    }
}

// Bicycle is also a type of Vehicle
class Bicycle: Vehicle {
    override func move() {
        print("\(name) (Bicycle) is pedaling at \(speed) km/h.")
    }
}

let myCar = Car(name: "Tesla", speed: 120, fuelType: "Electric")
let myBike = Bicycle(name: "Mountain Bike", speed: 20)

myCar.move()  // Output: Tesla (Car) is driving at 120 km/h using Electric.
myBike.move() // Output: Mountain Bike (Bicycle) is pedaling at 20 km/h.
```

Concurrency

```
func listPhotos(inGallery name: String) async -> [String] {
    let result = // ... some asynchronous networking code ...
    return result
}
```

- Mix of parallel and asynchronous code is known as concurrency
- Allows simultaneous execution of asynchronous tasks
- Writing parallel and asynchronous code is supported by Swift by default
- The async keyword is added to a function or method's declaration after its parameters to indicate that it is asynchronous.

Concurrency

```
let firstPhoto = await downloadPhoto(named: photoNames[0])
let secondPhoto = await downloadPhoto(named: photoNames[1])
let thirdPhoto = await downloadPhoto(named: photoNames[2])


let photos = [firstPhoto, secondPhoto, thirdPhoto]
show(photos)
```

- Multiple bits of code running simultaneously is referred to as parallel coding.
- Parallel or asynchronous programming offers more scheduling freedom, but at the expense of greater complexity.
- You can identify issues at compile time by utilising Swift's language-level support for concurrency in code that must be concurrent.

Exception Handling

- Errors represented by values of types that conform to the abstract Error protocol
- Errors thrown using keyword “throw”
- Thrown errors can be caught using do/catch with patterns, try
- try? converts error to optional that is nil if error exists
- Error propagation can be disabled entirely using try!

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
} catch {
    print("Unexpected error: \(error).")
}
// Prints "Insufficient funds. Please insert an additional 2 coins."
```

Exception Handling

1. **What it is?** This refers to a mechanism within a program that detects and responds to errors (aka “exceptions”) that occur during execution

2. **The benefit?** The program will have a structured way of handling unexpected errors which prevents the program from terminating and ruining the user’s experience

Exception Handling: How it works?

- **STEP 1 - Exception thrown:** At this stage, an error has occurred and the program has been alerted
- **STEP 2 - Exception caught**
- **STEP 3 - Error handling logic**

3 Mini Examples of Exception Handling

- Dividing by zero
  - Swift uses the `throw` keyword
- File not found error
- Invalid user input

```swift
func divideNumbers(_ numerator: Double, _ denominator: Double) throws -> Double {
    if denominator == 0 {
        throw DivisionError.divisionByZero
    }
    return numerator / denominator
}
```

Error Handling

1. **What is it?** This is a mechanism used to manage issues in the execution of a program.
2. **Difference from exception handling:**
    a. Errors are usually a result of failed network calls or resource issues whereas in exception handling, the errors are a result of logical constraints (e.g., division by zero)
    b. Exception handling usually deals with errors that can halt the execution of the program whereas error handling deals with errors

3. **How does Swift conduct Error Handling?**
    a. It uses error codes, throws, do-catch blocks, and nil-checks. There are other ways that Swift conducts Error Handling, however those are the most common methods

Error Handling Example

```
func checkPassword(_ password: String) throws -> String {
    if password.count < 5 { throw PasswordError.short }
    if password == "12345" { throw PasswordError.obvious }
```

Functional Programming Support

- Functions are first-class citizens in Swift: can be assigned, passed as args, and returned
- Pure functions and closures are supported
- Function composition is supported
- Protocol programming is supported (behavior-based instead of class-based, see right)
- Swift offers many built-in functional programming functions like .filter, .map, .flatMap, .sorted

```
let add: (Int, Int) -> Int = { $0 + $1 }
let result = add(2, 3) // result is 5

func performOperation(_ a: Int, _ b: Int, operation: (Int, Int) -> Int) -> Int {
    return operation(a, b)
}

let sum = performOperation(4, 5, operation: add) // sum is 9
```

```
protocol Drawable {
    func draw() -> String
}

struct Circle: Drawable {
    func draw() -> String {
        return "Drawing a circle"
    }
}

struct Square: Drawable {
    func draw() -> String {
        return "Drawing a square"
    }
}

let shapes: [Drawable] = [Circle(), Square()]
for shape in shapes {
    print(shape.draw())
}

// Outputs:
// Drawing a circle
// Drawing a square
```

SWIFT SUMMARY

STRENGTHS

- Memory-safe
- Super-fast on Apple devices
- Flexible and familiar dynamic language paradigms
- Robust object-oriented and functional programming support

WEAKNESSES

- Advanced features like generics and protocols can be challenging and verbose
- Copious operator overloading
- Lacks robust library and development support compared to other languages
- Poor cross-platform support

20
XX

PART 2: OUR PROJECT

20
25

TECHNOLOGY USED

Dev env : Xcode
Code quality enforcement : XCTest (testing), SwiftLint (linter)
UI components/interface : SwiftUI
Primary Framework : AVFoundation and AVPlayer for audio; MediaPlayer for system media control
Data management : CoreData, UserDefaults, FileManager (built-in)
State management: @State, Combine (built-in)
Music API : Spotify SDK, URLSession (built in network handling), Codable (JSON mapper)
Cloud : iCloud integration for syncing user preferences, playlists etc.

Constraints

- No comprehensive recommendation system based on user preference or genre like Spotify has
- No Spotify-wrapped-like system
- No sharing playlists between users
- No Spotify “friends”
- Simple, less feature rich UI

Features

- Display songs with album art
- Like songs
- Make playlists
- Play songs to user

PROJECT APPLICATION

Available for everyone with a mobile device (iOS) to stream and discover new music.