Swift Programming

Part 1: Building Blocks

What We'll Cover

1. Variables and Constants
2. Type Safety
3. Basic Data Types
4. Type Inference

Variables & Constants

```swift
// Variables can change
var score = 0
score = 100

// Constants cannot change
let maxScore = 1000
// maxScore = 2000 // This will cause an error
```

Storing Data in Swift

Type Safety

1. Each variable has a specific type
2. Types cannot be mixed accidentally
3. Prevents common programming errors

```swift
let age: Int = 25
// age = "25" // Error: Cannot assign String to Int
```

Swift's Guardrails

Basic Data Types

```swift
//Int

let age: Int = 42
let count = 30      // Type inference
let max = Int.max   // Maximum value
let min = Int.min   // Minimum value
```

```swift
//String

let name: String = "Swift"
let message = "Hello" // Type inference
let multiline = """
    Multiple
    lines
    """
```

```swift
// Double

let pi: Double = 3.14159
let height = 1.78    // Type inference
let scientific = 1.23e-2
```

```swift
//Bool

let isEnabled: Bool = true
let isVisible = false    // Type inference
let toggle = !isEnabled  // Negation
```

Type Inference

```swift
// Type inference examples
let name = "Swift"          // Type: String
let version = 5.0           // Type: Double
let isAwesome = true        // Type: Bool

// Explicit type annotation
let score: Int = 100
let message: String = "Hello"

// Type inference with operations
let total = 10 + 20         // Int
let average = 95.5 + 87.5 // Double
let greeting = "Hello " + "Swift" // String
```

Swift is Smart!

Benefits of Type Inference

1. **Cleaner Syntax & Less Verbose Code**

   Makes code shorter and easier to read by reducing repetitive type annotations

2. **Type Safety Maintained**

   Swift’s strong typing ensures variables maintain a consistent type

   Any attempt to assign a value of a different type will cause an error.

3. **Compiler Optimization**

   Swift’s compiler uses type inference to optimize code execution

   This leads to better performance

When to Use Type Annotation

1. **Ambiguous Types**

   When the compiler may have difficulty inferring the exact type, such as with nil values or numeric literals.

2. **Required Protocols**

   If a variable must conform to a specific protocol, explicitly declaring its type helps enforce that protocol’s requirements.

3. **API Clarity**

   Adding type annotations improves readability, especially for complex or custom types, and clarifies the intent when used in an API.

Type Conversion

```swift
// Numeric type conversion
let integer = 42
let double = Double(integer)    // Int to Double
let float = Float(double)       // Double to Float
let string = String(integer)    // Int to String

// String to number conversion
let numberString = "123"
if let number = Int(numberString) {
    print(number) // Safe conversion
}

// Type conversion in operations
let price = 9.99            // Double
let quantity: Int = 2
let total = price * Double(quantity)

// Type safety
let x = 10
let y = 3.14
let result = Double(x) + y  // Must convert explicitly
```

Swift Tuples

Group Multiple Values

Introduction

A tuple is a group of multiple values combined into a single compound value.

```
let httpResponse = (200, "OK")
```

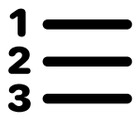
1. Group related values together
2. Return multiple values from functions
3. Create lightweight, ad-hoc data structures

Basic Tuple

```swift
// Basic tuple
let coordinates = (2, 3)
let x = coordinates.0    // Access by index
let y = coordinates.1

// Named tuple elements
let person = (name: "John", age: 25)
print(person.name)       // Access by name
print(person.age)
```

1. Index-based access (zero-based)
2. Named elements for clarity
3. Type inference

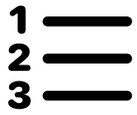
Tuple Decomposition



```swift
// Decomposing into variables
let http404Error = (404, "Not Found")
let (statusCode, message) = http404Error
print("Status: \(statusCode)")

// Ignoring parts with underscore
let (justCode, _) = http404Error
```

1. De-structure into separate variables
2. Use underscore to ignore values
3. Pattern matching support

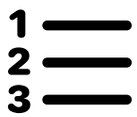
Functions & Tuples



```swift
func getUserInfo() -> (name: String, age: Int) {
    return ("Alice", 30)
}

let user = getUserInfo()
print("\(user.name) is \(user.age)")
```

1. Return multiple values
2. Type-safe return values
3. Named return values for clarity

Best Practices



✓ Use for temporary grouping of related values

✓ Return multiple values from functions

✓ Always use named elements for clarity

✗ Don't use tuples for complex data structures

✗ Avoid tuples with more than 4 elements

✗ Don't use tuples for database records

Part 2: Control Flow & Collections

Control Flow & Collections

1. Conditional Statements (if/else)
2. Switch Statements
3. Loops (for-in, while)
4. Arrays
5. Sets
6. Dictionaries

Conditional Statements

```swift
// Basic if statement
if score > 100 {
    print("High score!")
}

// if-else
if age >= 18 {
    print("Welcome!")
} else {
    print("Come back later!")
}

// else-if chains
if grade >= 90 {
    print("A")
} else if grade >= 80 {
    print("B")
} else {
    print("Keep trying!")
}
```

Features:

1. No parentheses required
2. Curly braces mandatory
3. Boolean conditions
4. Multiple conditions

Comparison Operators:

1. **>**, **<**, **>=**, **<=**
2. **==** , **!=**
3. **===**, **!==**
4. **&&** (**AND**), **||** (**OR**)

Making Decisions in Swift

Switch Statements

```
// Switch statement
let weather = "sunny"
let score = 90

switch weather {
case "sunny":
    print("Wear sunscreen")
case "rainy":
    print("Bring umbrella")
case "snowy":
    print("Wear boots")
default:
    print("Check forecast")
}

// Switch with ranges
switch score {
case 90...100:
    print("A")
case 80..<90:
    print("B")
case 70..<80:
    print("C")
default:
    print("Study more")
}
```

Key Features:

1. No implicit fall-through
2. Must be exhaustive
3. Multiple patterns
4. Value binding

Patterns:

1. Value matching
2. Range matching
3. Tuple matching
4. Enum matching

Elegant Multiple Choice

Loops - For and Stride

```swift
for i in 1...5 {
    print("Count: \\(i)")
}

// for-in with stride
for i in stride(from: 0, to: 10, by: 2) {
    print(i)  // prints 0, 2, 4, 6, 8
}

// for-in with arrays
let fruits = ["apple", "banana", "orange"]
for fruit in fruits {
    print("I like \\(fruit)")
}

// for-in with enumerated()
for (index, value) in fruits.enumerated() {
    print("\\(index + 1). \\(value)")
}
```

Range Types:

1. Closed range (...)
2. Half-open range (..<)
3. One-sided ranges
4. Stride ranges

Collection Iteration:

1. Arrays
2. Dictionaries
3. Sets
4. Sequences

Repeating Tasks in Swift

Loops - While and Repeat-While

```swift
// while loop
var lives = 3
while lives > 0 {
    print("Playing game...")
    lives -= 1
}

// repeat-while loop
var attempts = 1
repeat {
    print("Attempt \(attempts)")
    attempts += 1
} while attempts <= 3

// Control transfer statements
for num in 1...10 {
    if num % 2 == 0 {
        continue    // Skip even numbers
    }
    if num > 7 {
        break       // Exit loop
    }
    print(num)
}
```

Control Statements:

1. break
2. continue
3. fall-through
4. return

Watch Out For:

1. Infinite loops
2. Off-by-one errors
3. Unintended fall-through
4. Missing loop conditions

Repeating Tasks in Swift

Arrays

Arrays are ordered collections of values of the same type

Key Features:

1. Ordered collection
2. Zero-based indexing
3. Type-safe
4. Dynamic size

Common Uses:

1. Lists of items
2. Sequential data
3. Batch operations
4. Data transformation

Ordered Collections

Creating Arrays

Initialization Methods:

1. Array literal
2. Empty array with type
3. Repeating values
4. Type inference

```swift
// Literal syntax
var fruits = ["apple", "banana", "orange"]

// Type annotation
var numbers: [Int] = []

// Array with repeating value
var scores = Array(repeating: 0, count: 3)

// Array with type inference
var strings = Array<String>()
```

Accessing & Modifying Arrays

```swift
// Modifying arrays
fruits.append("mango")          // Add to end
fruits.insert("kiwi", at: 1)    // Insert at index
fruits.remove(at: 0)            // Remove at index
fruits[0] = "grape"             // Update element

// Accessing elements
let first = fruits[0]           // Index access
let last = fruits.last          // Last element
let range = fruits[1...3]       // Slice
```

Modification Methods

1. append(_:)
2. insert(_:at:)
3. remove(at:)
4. subscript syntax

Array Operations

```
// Common operations
let count = fruits.count          // Number of items
let first = fruits.first         // First element
let isEmpty = fruits.isEmpty    // Check if empty
let contains = fruits.contains("banana")

// Combining arrays
let combined = fruits + ["pear"]
fruits += ["melon"]              // Append array
```

Modification Methods

1. count: Number of elements
2. first/last: Optional first/last elements
3. isEmpty: Boolean check
4. contains: has that element

Array Transformations

```swift
// Transforming arrays
let uppercase = fruits.map { $0.uppercased() }
let filtered = fruits.filter { $0.count > 5 }
let sorted = fruits.sorted()
let reduced = numbers.reduce(0, +)

// Iterating
for fruit in fruits {
    print("I like \(fruit)")
}

// Enumerated access
for (index, fruit) in fruits.enumerated() {
    print("\(index + 1). \(fruit)")
}
```

Higher-Order Functions:

1. map: Transform elements
2. filter: Select elements
3. reduce: Combine elements
4. sorted: Order elements

Iteration Methods:

1. for-in loop
2. enumerated()
3. forEach closure
4. indices iteration

Sets

Sets are unordered collections of unique values

Key Features:

1. Unique elements only
2. Unordered collection
3. Type-safe
4. Hash-based storage

Best Used For:

1. Removing duplicates
2. Membership testing
3. Set operations
4. Unique constraints

Unordered Collections

Creating and Modifying Sets

```swift
// Creating sets
var colors: Set = ["red", "blue", "green"]
var numbers: Set<Int> = []

// Modifying sets
colors.insert("yellow")     // Added
colors.insert("red")        // Ignored (duplicate)
colors.remove("blue")       // Removed
colors.removeAll()          // Empty set
```

Key Operations:

1. insert(_:) - Adds an element
2. remove(_:) - Removes an element
3. contains(_:) - Checks membership
4. count - Number of elements

Unordered Collections

Set Operations

```
let set1: Set = [1, 2, 3]
let set2: Set = [2, 3, 4]

// Set operations
let union = set1.union(set2)                    // [1,2,3,4]
let intersection = set1.intersection(set2)    // [2,3]
let difference = set1.symmetricDifference(set2) // [1,4]
let subtraction = set1.subtracting(set2)     // [1]
```

Mathematical Operations:

1. union - All elements
2. intersection - Common elements
3. symmetricDifference - Non-common
4. count - Number of elements

Result Properties:

1. Creates new Set
2. Original Sets unchanged
3. Maintains uniqueness
4. Order not guaranteed

Unordered Collections

Dictionaries

Dictionaries store key-value associations where each key is unique

Key Features:

1. Unique keys
2. Unordered collection
3. Type-safe keys and values
4. Hash-based lookup

Best Used For:

1. Lookup tables
2. Caching
3. Grouping data
4. Counting occurrences

Key-value Collection

Creating Dictionaries

```swift
// Dictionary literal
var scores = ["Rajesh": 100, "Manoj": 95]

// Type annotation
var menu: [String: Double] = [:]

// With type inference
var ages = Dictionary<String, Int>()

// With initial capacity
var cache = Dictionary<String, Data>(minimumCapacity: 100)
```

Initialization Methods:

1. Dictionary literal
2. Empty dictionary with type
3. Type inference
4. With capacity hint

Key-value Collection

Accessing & Modifying

```swift
// Adding and updating
scores["Suresh"] = 98       // Add new entry
scores["Manoj"] = 99        // Update existing
scores.updateValue(97, forKey: "Suresh")

// Removing
scores.removeValue(forKey: "Rajesh")
scores["Rajesh"] = nil      // Alternative removal

// Safe access
let johnScore = scores["John"] ?? 0  // Default value
if let score = scores["Manoj"] {     // Optional binding
    print("Score: \(score)")
}
```

Modification Methods:

1. Subscript syntax
2. updateValue(_:forKey:)
3. removeValue(forKey:)
4. removeAll()

Key-value Collection

Dictionary Operations

```swift
// Common operations
let count = scores.count
let isEmpty = scores.isEmpty

// Collections of keys and values
let allKeys = scores.keys
let allValues = scores.values

// Checking for keys
let hasKey = scores.keys.contains("Suresh")

// Transform to array
let keyArray = Array(scores.keys)
let valueArray = Array(scores.values)
```

Properties and Methods:

1. count: Number of entries
2. isEmpty: Check if empty
3. keys: Collection of keys
4. values: Collection of values

Key-value Collection

Iterating Dictionaries

```swift
// Iterate key-value pairs
for (name, score) in scores {
    print("\(name): \(score)")
}

// Iterate keys only
for name in scores.keys {
    print(name)
}

// Iterate values only
for score in scores.values {
    print(score)
}

// Using forEach
scores.forEach { (key, value) in
    print("\(key) scored \(value)")
}
```

Iteration Methods:

1. for-in loop
2. forEach closure
3. keys iteration
4. values iteration

Key-value Collection

Swift Functions

Building Blocks of Code

Basic Functions

```swift
// Basic function
func sayHello() {
    print("Hello!")
}

// Function with parameter
func greeting(name: String) {
    print("Hello, \(name)!")
}

// Function with return value
func add(a: Int, b: Int) -> Int {
    return a + b
}
```

Anatomy:

1. func keyword
2. Function name
3. Parameter list
4. Return type
5. Function body

Key Points:

1. Clear naming
2. Type safety
3. Return arrow ->
4. Explicit returns

Defining and Calling Functions

Parameter Labels

```swift
// External and internal parameter names
func greet(to name: String) {
    print("Hello, \(name)!")
}

// Calling with external name
greet(to: "Swift")

// Omitting parameter label with _
func multiply(_ x: Int, by y: Int) -> Int {
    return x * y
}

// Calling without parameter name
let product = multiply(4, by: 2)
```

Label Features:

1. External labels for callers
2. Internal names for implementation
3. More readable function calls

In-Out Parameters

```swift
// In-out parameter example
func swapValues(_ a: inout Int, _ b: inout Int) {
    let temp = a
    a = b
    b = temp
}

// Using in-out parameters
var x = 5
var y = 10
swapValues(&x, &y)
// Now x is 10 and y is 5

// Real-world example
func applyDiscount(_ price: inout Double, discount: Double) {
    price = price * (1 - discount)
}
```

Modifying Parameters

Key Points:

1. inout keyword
2. & prefix when calling
3. Modifies original value
4. Pass by reference

Caution:

1. Use sparingly
2. Can make code complex
3. Consider alternatives
4. Document clearly

Default Parameters

```swift
// Function with default parameter
func greet(name: String, greeting: String = "Hello") {
    print("\(greeting), \(name)!")
}

// Different ways to call
greet(name: "Swift")              // Uses default
greet(name: "Swift", greeting: "Hi")  // Custom greeting

// Multiple default parameters
func createUser(
    name: String,
    age: Int = 0,
    isAdmin: Bool = false
) {
    print("Created user \(name)")
}
```

Benefits:

1. Flexible function calls
2. Sensible defaults
3. Backward compatibility
4. Cleaner call sites

Optional Parameter Values

Variadic Parameters

```swift
// Variadic parameter
func sum(_ numbers: Int...) -> Int {
    return numbers.reduce(0, +)
}

// Call with any number of parameters
let total1 = sum(1, 2, 3)
let total2 = sum(1, 2, 3, 4, 5)

// Mixing regular and variadic parameters
func printTeam(captain: String, _ members: String...) {
    print("Captain: \(captain)")
    for member in members {
        print("Member: \(member)")
    }
}
```

Features:

1. Variable argument count
2. Type safety maintained
3. Treated as array inside
4. Zero or more arguments

Variable Number of Parameters

Swift Enumerations

Type-Safe Constants and More

Basic Enums

```swift
// Basic enum declaration
enum CompassDirection {
    case north
    case south
    case east
    case west
}

// Shorter syntax
enum Season {
    case spring, summer, autumn, winter
}
```

Defining Simple Options

Key Features:

1. Type-safe options
2. Multiple declaration styles
3. First-class types
4. No raw values needed

Best Practices:

1. Use PascalCase for enum names
2. Use camelCase for cases
3. Group related options
4. Keep cases simple

Switch with Enums

```
enum Weather {
    case sunny, rainy, cloudy, snowy
}

let currentWeather = Weather.sunny

switch currentWeather {
    case .sunny:
        print("Wear sunscreen")
    case .rainy:
        print("Bring umbrella")
    case .cloudy:
        print("Could go either way")
    case .snowy:
        print("Wear boots")
}
```

Advantages:

1. Exhaustive matching
2. No default needed
3. Compiler verification
4. Type inference

Pattern Matching:

1. Case pattern matching
2. Value binding
3. Multiple patterns
4. Where clauses

Perfect Pattern Matching

Enums with Associated Values

```swift
// Enum with associated values
enum NetworkResponse {
    case success(data: Data)
    case failure(error: Error)
    case progress(percent: Double)
}

// Using associated values
let response = NetworkResponse.progress(percent: 85.5)

switch response {
    case .success(let data):
        print("Received \(data.count) bytes")
    case .failure(let error):
        print("Error: \(error.localizedDescription)")
    case .progress(let percent):
        print("Loading: \(percent)%")
}
```

Key Points:

1. Can store additional data
2. Different types per case
3. Pattern matching in switch
4. Value binding with let/var

Adding Dynamic Values to Enum Cases

Enums with Raw Values

```swift
// Enum with raw values
enum HttpStatus: Int {
    case ok = 200
    case notFound = 404
    case serverError = 500

    var message: String {
        switch self {
        case .ok: return "Success"
        case .notFound: return "Not Found"
        case .serverError: return "Server Error"
        }
    }
}

// Using raw values
let status = HttpStatus.notFound
print(status.rawValue)      // 404
print(status.message)       // "Not Found"

// Creating from raw value
if let status = HttpStatus(rawValue: 404) {
    print(status.message)   // "Not Found"
}
```

Adding Underlying Values to Enum Cases

Optionals

Safe Handling of Nil Values

Understanding Optionals

```swift
// Optional declaration
var name: String?  // Can be String or nil
var age: Int?      // Can be Int or nil

// Optional assignment
name = "Swift"
age = nil

// Forced unwrapping (dangerous!)
let forcedName = name!  // Crashes if nil!

// Safe unwrapping
if name != nil {
    print("Name is \(name!)")
}

// Safer: Optional binding
if let unwrappedName = name {
    print("Name is \(unwrappedName)")
}
```

Key Concepts:

1. Optional type syntax (?)
2. nil value possible
3. Type safety enforced
4. Explicit unwrapping

Handling Missing Values

Optional Binding

```swift
let userName: String? = "Steve Jobs"

// if let binding
if let name = userName {
    print("Hello, \(name)")
} else {
    print("No name provided")
}

let firstName: String? = "Steve"
let lastName: String? = "Jobs"

// Multiple optional binding
if let firstName = firstName,
   let lastName = lastName {
    print("Name: \(firstName) \(lastName)")
}

// while let
var numbers = [Int?]([1, 2, nil, 4])
while let num = numbers.first {
    numbers.removeFirst()
    print(num ?? 0)
}
```

Binding Patterns:

1. if let binding
2. guard let binding
3. while let binding
4. Multiple bindings

Guard Statements

```swift
func processUser(name: String?, age: Int?) {
    // Early returns with guard
    guard let name = name else {
        print("Name required")
        return
    }

    guard let age = age else {
        print("Age required")
        return
    }

    // name and age are available here
    print("Processing user \(name), age \(age)")
}
```

```swift
func processUser(name: String?, age: Int?) {

    // Guard with multiple conditions
    guard let name = name,
          let age = age else {
        return
    }

    // name and age are available here
    print("Processing user \(name), age \(age)")
}
```

Advantages:

1. Early returns
2. Cleaner code flow
3. Reduced nesting
4. Keep else blocks simple
5. Used for preconditions

Early Returns and Clean Code

If let vs guard

Feature/Aspect	If let	guard
Purpose	Checks and unwraps optionals within a local scope	Checks and unwraps optionals, ensures certain conditions are met for further execution
Code Flow	Continues execution inside the block if the unwrapping is successful.	Exits the current scope (e.g., function, loop) if the condition fails.
Readability	Can lead to nested code if used repeatedly, reducing readability.	Provides a clean, early exit, leading to less nesting and clearer code structure.
Placement	Typically used within the main body of the code, leading to nested conditions.	Typically placed at the beginning of a function or block for input validation or preconditions.
Error Handling	Used to conditionally handle errors or alternative paths inside the block.	the if let block. The unwrapped value is available after the guard block (in the rest of
Unwrapped Value Scope	The unwrapped value is only available inside the if let block.	The unwrapped value is available after the guard block (in the rest of the function).
Syntax	if let value = optional { ... }	guard let value = optional else { ... }
Typical Use Cases	For local unwrapping with conditions where continuing execution depends on the unwrapped value.	For validating inputs or preconditions at the start of functions/methods for clearer control flow.

Optional Chaining

```swift
// Optional chaining
struct Person {
    var address: Address?
}

struct Address {
    var street: String?
}

let person: Person? = Person(address: nil)

// Without optional chaining
if let p = person {
    if let a = p.address {
        if let s = a.street {
            print("Street: \(s)")
        }
    }
}

// With optional chaining
if let street = person?.address?.street {
    print("Street: \(street)")
}

// Multiple operations
let streetLength = person?.address?.street?.count ?? 0
```

Features:

1. Nested optionals
2. Safe access
3. Concise syntax
4. Nil coalescing

Accessing Nested Optionals

Swift Closures

Functions that can capture their Environment

Introduction

Closures are self-contained blocks of functionality that can be passed around and used in your code

Key Features:

1. First-class citizens - Functions as variables/parameters
2. Can capture values - Remember external variables/state
3. Multiple syntax styles - Flexible writing formats
4. Type inference - Automatic type detection.

Introduction

Closures are self-contained blocks of functionality that can be passed around and used in your code

Best Used For:

1. Completion handlers - Async operation results
2. Higher-order functions - Functions processing functions
3. Callbacks - Response to events
4. Custom functionality - Flexible behaviour definition.

Closure Syntax

```swift
// Regular function
func add(a: Int, b: Int) -> Int {
    return a + b
}

// Full closure syntax
let addClosure = { (a: Int, b: Int) -> Int in
    return a + b
}

// Closure with explicit type annotation (recommended)
let addShort: (Int, Int) -> Int = { $0 + $1 }

// Usage is the same for all
let result1 = add(a: 5, b: 3)      // Function call
let result2 = addClosure(5, 3)     // Closure call
let result3 = addShort(5, 3)       // Short syntax
```

Syntax Features:

1. Parameter and return type specification
2. in keyword separates definition from body
3. Shorthand argument names ($0, $1)
4. Type inference capabilities

From Functions to Closures

Higher-Order Functions

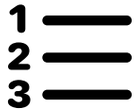
```swift
let numbers = [1, 2, 3, 4, 5]

// map: Transform each element
let doubled = numbers.map { $0 * 2 }
// [2, 4, 6, 8, 10]

// filter: Keep matching elements
let evenNumbers = numbers.filter { $0 % 2 == 0 }
// [2, 4]

// reduce: Combine elements
let sum = numbers.reduce(0) { $0 + $1 }
// 15

// Chaining higher-order functions
let sumOfDoubledEvens = numbers
    .filter { $0 % 2 == 0 }
    .map { $0 * 2 }
    .reduce(0, +)
```

Common Functions:

1. map: Transform elements
2. filter: Select elements
3. reduce: Combine elements
4. forEach: Iterate elements
5. compactMap: Transform and filter nil
6. flatMap: Transform and flatten

Functions That Work With Closures

Closure Best Practices



1. Use trailing closure syntax
2. Keep closures simple
3. Use type inference
4. Consider readability over brevity
5. Handle retain cycles
6. Document complex closures
7. Break long closures into named functions

Error Handling

Responding to and Recovering from Errors

Introduction

Swift provides first-class support for throwing, catching, and propagating errors

Key Features:

1. Error protocols - Define custom error types easily
2. Throwing functions - Functions that report errors
3. Error propagation - Pass errors up the call chain
4. Error handling - Try-catch blocks for error management

Introduction

Swift provides first-class support for throwing, catching, and propagating errors

Benefits:

1. Type-safe error handling - Catch specific error types safely
2. Clear error boundaries - Know exactly where error occur
3. Explicit error cases - Define all possible error scenarios
4. Pattern matching - Match and handle specific errors

Error Handling Keywords

```swift
// throws: Indicates function can throw errors
func riskyOperation() throws

// try: Used when calling throwing functions
try riskyOperation()

// do: Creates a new error handling scope
do {
    try riskyOperation()
}

// catch: Handles specific errors
catch {
    // Handle errors here
}
```

Keywords:

1. throws: Function declaration
2. try: Call throwing functions
3. do: Start error handling block
4. catch: Handle errors
5. throw: Raise an error

Basic Error Pattern

```
do {
    try riskyOperation()
    print("Success!")
} catch {
    print("Something went wrong: \(error)")
}
```

Putting It All Together

Creating Custom Errors

```swift
enum PaymentError: Error {
    case insufficientFunds
    case invalidCard
    case networkError(String)
}

func processPayment(amount: Int) throws {
    guard hasBalance(amount) else {
        throw PaymentError.insufficientFunds
    }

    // Process payment...
}

// Using optional try
if let result = try? processPayment(amount: 100) {
    print("Payment successful")
}

// Force try (avoid in production)
try! processPayment(amount: 50)
```

Error Types:

1. Custom enumerations
2. Associated values
3. Protocol conformance
4. Nested errors

Best Practices:

1. Descriptive error types
2. Avoid force try
3. Group related errors
4. Document error cases

Defining Your Error Types

Handling Specific Errors

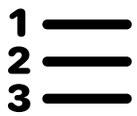
```
do {
    try processPayment(amount: 100)
} catch PaymentError.insufficientFunds {
    print("Not enough money")
} catch PaymentError.invalidCard {
    print("Card declined")
} catch {
    print("Unknown error: \(error)")
}

// Alternative pattern with switch
do {
    try processPayment(amount: 100)
} catch let error as PaymentError {
    switch error {
    case .insufficientFunds:
        print("Not enough money")
    case .invalidCard:
        print("Card declined")
    case .networkError(let message):
        print("Network error: \(message)")
    }
}
```

Pattern Matching:

1. Multiple catch blocks
2. Type casting
3. Switch statements
4. Default handling

Pattern Matching in Catch

Error Handling Best Practices

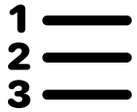


1. Use meaningful error types
2. Handle specific cases first
3. Provide error context
4. Document error conditions
5. Consider error recovery
6. Use error hierarchies
7. Avoid Overuse of try?
8. Avoid Generic error types

Object Oriented Swift

Where functions evolve into elegant objects

What We'll Cover



1. Classes vs Structures
2. Properties & Methods
3. Inheritance & Polymorphism
4. Protocols & Extensions
5. Value vs Reference Types
6. Memory Management

Classes vs Structures

```swift
// Structure
struct Point {
    var x: Int
    var y: Int
}

// Class
class Circle {
    var center: Point
    var radius: Double

    init(center: Point, radius: Double) {
        self.center = center
        self.radius = radius
    }
}

// Value vs Reference type
var point1 = Point(x: 0, y: 0)
var point2 = point1  // Creates a copy
point2.x = 5  // Only point2 changes

var circle1 = Circle(center: point1, radius: 10)
var circle2 = circle1  // Both reference same instance
circle2.radius = 20  // Both circles affected
```

Structures:

1. Value semantics
2. Automatic memberwise init
3. No inheritance
4. Stack allocation

Classes:

1. Reference semantics
2. Custom initializers
3. Inheritance
4. Heap allocation

Understanding the Differences

Properties & Methods

```swift
class BankAccount {
    // Stored properties
    var balance: Double
    let accountNumber: String


    init(accountNumber: String) {
        self.accountNumber = accountNumber
        self.balance = 0
    }

    // Computed property
    var formattedBalance: String {
        return "$\\(balance)"
    }

    // Property observer
    var transactions: Int = 0 {
        willSet {
            print("About to perform transaction #\\(newValue)")
        }
        didSet {
            print("Transaction complete")
        }
    }

    // Methods
    func deposit(_ amount: Double) {
        balance += amount
        transactions += 1
    }

    func withdraw(_ amount: Double) -> Bool {
        if amount <= balance {
            balance -= amount
            transactions += 1
            return true
        }
        return false
    }
}
```

Property Types::

1. Stored properties
2. Computed properties
3. Property observers
4. Type properties (static)

Adding Behaviour to Types

Inheritance & Polymorphism

```
class Vehicle {
    var speed: Double

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

    func description() -> String {
        return "Traveling at \\(speed) mph"
    }
}

class Car: Vehicle {
    var gear: Int

    init(speed: Double, gear: Int) {
        self.gear = gear
        super.init(speed: speed)
    }

    override func description() -> String {
        return "Car in gear \\(gear), " + super.description()
    }
}

// Polymorphism
let vehicles: [Vehicle] = [
    Vehicle(speed: 20),
    Car(speed: 50, gear: 4)
]

for vehicle in vehicles {
    print(vehicle.description())
}
```

Building Type Hierarchies

Access Control & Encapsulation

```swift
public class BankAccount {
    // Private properties
    private var balance: Double
    private let accountNumber: String

    // Public interface
    public init(accountNumber: String) {
        self.accountNumber = accountNumber
        self.balance = 0
    }

    // Internal method
    internal func validate() -> Bool {
        // Implementation
        return true
    }

    // Public methods
    public func getBalance() -> Double {
        return balance
    }

    public func deposit(_ amount: Double) {
        guard amount > 0 else { return }
        balance += amount
    }

    // File-private helper
    fileprivate func logTransaction(_ type: String) {
        print("\(type) transaction for \(accountNumber)")
    }
}
```

Access Levels:

1. private: Only within declaration
2. fileprivate: Only within file
3. internal: Within module (default)
4. public: Anyone can access
5. open: Anyone can access and subclass

Memory Management

```swift
class Parent {
    var name: String
    weak var child: Child?    // Weak reference

    init(name: String) {
        self.name = name
    }

    deinit {
        print("Parent \(name) deinitialized")
    }
}

class Child {
    var name: String
    weak var parent: Parent?  // Weak reference

    init(name: String) {
        self.name = name
    }

    deinit {
        print("Child \(name) deinitialized")
    }
}

// Usage
var parent: Parent? = Parent(name: "John")
var child: Child? = Child(name: "Jane")

// Set up relationships
parent?.child = child
child?.parent = parent

// Break strong reference cycle
parent = nil
child = nil
```

Reference Types:

1. strong (default)
2. weak
3. unowned

Understanding ARC

Protocols

```swift
// Protocol definition
protocol Payable {
    var salary: Double { get }
    func calculatePay() -> Double
}

protocol Identifiable {
    var id: String { get }
}

// Implementing protocols
struct Employee: Payable, Identifiable {
    let id: String
    let salary: Double
    let hourlyRate: Double
    var hoursWorked: Double

    func calculatePay() -> Double {
        return hourlyRate * hoursWorked
    }
}

// Protocol as type
func processPay(worker: Payable) {
    let pay = worker.calculatePay()
    print("Processing payment: \\(pay)")
}
```

Protocol Features:

1. Property requirements
2. Method requirements
3. Multiple conformance
4. Type constraints

Use Cases:

1. Dependency injection
2. Abstraction
3. Composition
4. API contracts

Defining Contracts

Extensions

```swift
// Extending built-in types
extension String {
    var isPalindrome: Bool {
        let chars = Array(self.lowercased())
        return chars == chars.reversed()
    }
}

// Protocol extensions
extension Collection where Element: Numeric {
    func sum() -> Element {
        return reduce(0, +)
    }
}

// Usage
let text = "radar"
print(text.isPalindrome)  // true

let numbers = [1, 2, 3, 4]
print(numbers.sum())  // 10
```

Extension Types:

1. Add computed properties
2. Add methods
3. Add protocol conformance
4. Add initializers
5. Constrained extensions
6. Protocol extensions

Adding Functionality

Protocol-Oriented Programming

```swift
protocol Vehicle {
    var speed: Double { get }
    func describe() -> String
}

extension Vehicle {
    // Default implementation
    func describe() -> String {
        return "Moving at \(speed) mph"
    }
}

// Protocol composition
protocol Named {
    var name: String { get }
}

protocol Aged {
    var age: Int { get }
}

// Using protocol composition
func greet(person: Named & Aged) {
    print("Hello \(person.name), you are \(person.age)")
}
```

Benefits:

1. Code reuse
2. Default implementations
3. Value semantics
4. Composition over inheritance

Swift Generics

Writing Flexible & Reusable Code

Generic Functions

```swift
// Without generics - duplicate code
func swapInts(_ a: inout Int, _ b: inout Int) {
    let temp = a
    a = b
    b = temp
}

func swapStrings(_ a: inout String, _ b: inout String) {
    let temp = a
    a = b
    b = temp
}

// With generics - one function for all types
func swap<T>(_ a: inout T, _ b: inout T) {
    let temp = a
    a = b
    b = temp
}

// Usage
var x = 5
var y = 10
swap(&x, &y)  // Works with integers

var hello = "Hello"
var world = "World"
swap(&hello, &world)  // Works with strings
```

Benefits:

1. Code reusability
2. Type safety
3. Reduced duplication
4. Compile-time checks

Common Uses:

1. Collections
2. Algorithms
3. Data structures
4. API design

Your First Step into Generics

Generic Types

```swift
// Generic Stack implementation
struct Stack<Element> {
    private var items: [Element] = []

    mutating func push(_ item: Element) {
        items.append(item)
    }

    mutating func pop() -> Element? {
        return items.popLast()
    }

    func peek() -> Element? {
        return items.last
    }

    var isEmpty: Bool {
        return items.isEmpty
    }
}

// Usage with different types
var numberStack = Stack<Int>()
numberStack.push(42)
numberStack.push(73)

var stringStack = Stack<String>()
stringStack.push("Hello")
stringStack.push("World")
```

Key Points:

1. Type parameter in angle brackets
2. Use placeholder throughout type
3. Type inference available
4. Multiple type parameters possible

Creating Flexible Data Structures

Any Questions?