Table of Contents

Searching within strings

Replacing substrings

Converting case

Stripping whitespace

Splitting and joining strings

Checking for substring existence

String formatting

String manipulation with regular expressions

Conditional statements & Control flow & Loops

if / else statements

switch / case statements

while loops

for loops

Loop control statements (break, continue)

Nested loops

Looping through iterable objects (lists, arrays, dictionaries, etc.)

Iterating over ranges

Infinite loops and how to handle them

Lists / Arrays

Creating lists/arrays

Accessing elements by index

Modifying elements

Slicing lists/arrays

Concatenating lists/arrays

List/array comprehension

Sorting lists/arrays

Reversing lists/arrays

Finding elements in lists/arrays

Removing elements from lists/arrays

Dictionaries / Maps

Creating dictionaries/maps

Accessing elements by key

Modifying elements

Checking for key existence

Dictionary/map comprehension

Iterating over keys

Iterating over values

Iterating over key-value pairs

Sorting dictionaries/maps

Merging dictionaries/maps

Sets

Creating sets

Adding elements to sets

Removing elements from sets

Checking for element existence

Set operations (union, intersection, difference, symmetric difference)

Set comprehension

Converting lists/arrays to sets and vice versa

Iterating over sets

Checking for subsets and supersets

Exceptions / try/catch

Handling exceptions with try/catch blocks

Catching specific exceptions

Raising exceptions

Cleaning up with finally block

Exception chaining

Functions

Defining functions

Function arguments (positional, keyword, default values)

Returning values from functions

Function overloading

Lambda functions

Recursion

Generators and iterators

Decorators

Higher-order functions

Function documentation and comments

OOP

Class creation syntax

Creating instances of a class

Class attributes vs instance attributes

Constructor method (init method)

Instance methods vs static methods vs class methods

Encapsulation (public vs private members)

Inheritance

Polymorphism (method overriding, method overloading)

Composition vs inheritance

Abstract classes/interfaces

Variable Declaration

Declaring variables

```
var myVar: Int
var myString: String
let myConst: Double
```

Assigning values to variables

```
var myVar = 10
var myString = "Hello, World!"
let myConst = 3.14
```

Initializing variables

```
var myVar: Int = 5
var myString: String = "Swift"
let myConst: Double = 2.71828
```

Declaring multiple variables in a single line

```
var x = 1, y = 2, z = 3
```

Using constants

```
let pi = 3.14159
```

Variable scope

```
var globalVar = 10

func myFunction() {
    var localVar = 5
    print(globalVar) // Accessible
    print(localVar)  // Accessible
}

print(globalVar) // Accessible
print(localVar)  // Error: localVar is not accessible outside myFunction
```

Variable naming conventions

```
var my_variable: Int
var camelCaseVariable: String
```

Implicit vs explicit declaration

```
var implicitVar = 10
var explicitVar: Int = 10
```

Type inference

```
var myVar = 10 // Inferred as Int
var myString = "Swift" // Inferred as String
```

Printing Output

Printing a single variable

```
var name = "John"
print(name)
```

Printing multiple variables

```
var age = 30
var height = 6.2
print("Age: \(age), Height: \(height)")
```

Formatting output

```
var num = 3.14159
print(String(format: "%.2f", num))
```

Printing special characters

```
print("This is a tab: \t and this is a newline: \n")
```

String methods

String concatenation

```
var str1 = "Hello"
var str2 = "World"
var combinedStr = str1 + " " + str2
```

String slicing

```
var str = "Hello, World!"
var slicedStr = str[..<5] // Output: "Hello"
```

Searching within strings

```
var str = "Hello, World!"
if str.contains("Hello") {
    print("Found")
}
```

Replacing substrings

```
var str = "Hello, World!"
var replacedStr = str.replacingOccurrences(of:
"World", with: "Swift")
```

Converting case

```
var str = "hello"
var uppercasedStr = str.uppercased()
var lowercasedStr = str.lowercased()
```

Stripping whitespace

```
var str = "  Hello, World!  "
```

```
var trimmedStr = str.trimmingCharacters(in:
.whitespacesAndNewlines)
```

Splitting and joining strings

```
var str = "Apple,Orange,Banana"
var splittedStr = str.split(separator: ",")
var joinedStr = splittedStr.joined(separator: " ")
```

Checking for substring existence

```
var str = "Hello, World!"
if let range = str.range(of: "Hello") {

   print("Substring found")
}
```

String formatting

```
var name = "John"
var age = 30
var formattedString = String(format: "Name: %@, Age:
%d", name, age)
print(formattedString)
```

String manipulation with regular expressions

```
import Foundation
var str = "Hello, World!"
```

```
var pattern = "Hello"
var regex = try! NSRegularExpression(pattern:
pattern)
if let match = regex.firstMatch(in: str, range:
NSRange(location: 0, length: str.utf16.count)) {
print("Match found")
}
```

Conditional statements & Control flow & Loops

if / else statements

```
var num = 10
if num > 0 {
    print("Positive")
} else if num < 0 {
    print("Negative")
} else {
    print("Zero")
}
```

switch / case statements

```
var grade = "A"
switch grade {
case "A":
    print("Excellent")
case "B":
    print("Good")
```

```
default:
    print("Pass")
}
```

while loops

```
var i = 0
while i < 5 {
    print(i)
    i += 1
}
```

for loops

```
for i in 0..<5 {
    print(i)
}
```

Loop control statements (break, continue)

```
for i in 0..<5 {
    if i == 3 {
        break
    }
    print(i)
}
```

Nested loops

```
for i in 0..<3 {
    for j in 0..<3 {
        print(i * j)
    }
}
```

Looping through iterable objects (lists, arrays, dictionaries, etc.)

```
var nums = [1, 2, 3, 4, 5]
for num in nums {
    print(num)
}
```

Iterating over ranges

```
for i in 1...5 {
    print(i)
}
```

Infinite loops and how to handle them

```
var i = 0
while true {
    print(i)
    if i == 10 {
        break
    }
```

```
    i += 1
}
```

Lists / Arrays

Creating lists/arrays

```
var nums = [1, 2, 3, 4, 5]
```

Accessing elements by index

```
var nums = [1, 2, 3, 4, 5]
print(nums[0]) // Output: 1
```

Modifying elements

```
var nums = [1, 2, 3, 4, 5]
nums[0] = 10
print(nums) // Output: [10, 2, 3, 4, 5]
```

Slicing lists/arrays

```
var nums = [1, 2, 3, 4, 5]
var slicedNums = nums[1..<4]
print(slicedNums) // Output: [2, 3, 4]
```

Concatenating lists/arrays

```
var nums1 = [1, 2, 3]
var nums2 = [4, 5, 6]
var combinedNums = nums1 + nums2
print(combinedNums) // Output: [1, 2, 3, 4, 5, 6]
```

List/array comprehension

```
var nums = [1, 2, 3, 4, 5]
var squaredNums = [num * num for num in nums]
print(squaredNums) // Output: [1, 4, 9, 16, 25]
```

Sorting lists/arrays

```
var nums = [5, 3, 1, 4, 2]
nums.sort()
print(nums) // Output: [1, 2, 3, 4, 5]
```

Reversing lists/arrays

```
var nums = [1, 2, 3, 4, 5]
nums.reverse()
print(nums) // Output: [5, 4, 3, 2, 1]
```

Finding elements in lists/arrays

```
var nums = [1, 2, 3, 4, 5]
if nums.contains(3) {
```

```
    print("Element found")
}
```

Removing elements from lists/arrays

```
var nums = [1, 2, 3, 4, 5]
nums.remove(at: 2)
print(nums) // Output: [1, 2, 4, 5]
```

Dictionaries / Maps

Creating dictionaries/maps

```
var ages = ["John": 30, "Jane": 25, "Doe": 40]
```

Accessing elements by key

```
var ages = ["John": 30, "Jane": 25, "Doe": 40]
print(ages["John"]!) // Output: 30
```

Modifying elements

```
var ages = ["John": 30, "Jane": 25, "Doe": 40]
ages["John"] = 35
print(ages) // Output: ["John": 35, "Jane": 25,
"Doe": 40]
```

Checking for key existence

```
var ages = ["John": 30, "Jane": 25, "Doe": 40]
if let age = ages["John"] {
    print("Age found: \(age)")
}
```

Dictionary/map comprehension

```
var nums = [1, 2, 3, 4, 5]
var squaredDict = [num: num * num for num in nums]
print(squaredDict) // Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Iterating over keys

```
var ages = ["John": 30, "Jane": 25, "Doe": 40]
for name in ages.keys {
    print(name)
}
```

Iterating over values

```
var ages = ["John": 30, "Jane": 25, "Doe": 40]
for age in ages.values {
    print(age)
}
```

Iterating over key-value pairs

```
var ages = ["John": 30, "Jane": 25, "Doe": 40]
for (name, age) in ages {
    print("\(name) is \(age) years old")
}
```

Sorting dictionaries/maps

```
var ages = ["John": 30, "Jane": 25, "Doe": 40]
var sortedAges = ages.sorted { $0.key < $1.key }
print(sortedAges) // Output: [("Doe", 40), ("Jane", 25), ("John", 30)]
```

Merging dictionaries/maps

```
var dict1: [String: Int] = ["John": 30, "Jane": 25]
var dict2: [String: Int] = ["Doe": 40, "Smith": 35]
var mergedDict = dict1.merging(dict2) { (current, _) in current }
print(mergedDict) // Output: ["John": 30, "Jane": 25, "Doe": 40, "Smith": 35]
```

Sets

Creating sets

```
var mySet: Set = [1, 2, 3, 4, 5]
```

Adding elements to sets

```
var mySet: Set = [1, 2, 3]
mySet.insert(4)
print(mySet) // Output: [1, 2, 3, 4]
```

Removing elements from sets

```
var mySet: Set = [1, 2, 3]
mySet.remove(2)
print(mySet) // Output: [1, 3]
```

Checking for element existence

```
var mySet: Set = [1, 2, 3]
if mySet.contains(2) {
    print("Element found")
}
```

Set operations (union, intersection, difference, symmetric difference)

```
var set1: Set = [1, 2, 3]
var set2: Set = [3, 4, 5]
var unionSet = set1.union(set2) // Output: [1, 2, 3, 4, 5]
var intersectionSet = set1.intersection(set2) // Output: [3]
var differenceSet = set1.subtracting(set2) // Output: [1, 2]
var symmetricDifferenceSet =
```

```
set1.symmetricDifference(set2) // Output: [1, 2, 4, 5]
```

Set comprehension

```
var nums = [1, 2, 3, 4, 5]
var mySet: Set = [num * num for num in nums]
print(mySet) // Output: [1, 4, 9, 16, 25]
```

Converting lists/arrays to sets and vice versa

```
var myList = [1, 2, 3, 4, 5]
var mySet = Set(myList)
var myArray = Array(mySet)
```

Iterating over sets

```
var mySet: Set = ["Apple", "Orange", "Banana"]
for item in mySet {
    print(item)
}
```

Checking for subsets and supersets

```
var set1: Set = [1, 2, 3, 4]
var set2: Set = [2, 3]
if set2.isSubset(of: set1) {
    print("set2 is a subset of set1")
}
```

```
if set1.isSuperset(of: set2) {
    print("set1 is a superset of set2")
}
```

Exceptions / try/catch

Handling exceptions with try/catch blocks

```
do {
    // Code that might throw an exception
    let result = try someFunction()
    print(result)
} catch {
    print("An error occurred: \(error)")
}
```

Catching specific exceptions

```
do {
    let result = try someFunction()
    print(result)
} catch SomeError.invalidInput {
    print("Invalid input error occurred")
} catch SomeError.runtimeError(let message) {
    print("Runtime error occurred: \(message)")
} catch {
    print("An error occurred: \(error)")
}
```

Raising exceptions

```
enum MyError: Error {
    case runtimeError(String)
}
func someFunction() throws -> Int {
throw MyError.runtimeError("Something went wrong")
}
```

Cleaning up with finally block

```
do {
    let result = try someFunction()
    print(result)
} catch {
    print("An error occurred: \(error)")
} finally {
    // Code to be executed regardless of whether an exception was thrown
}
```

Exception chaining

```
enum MyError: Error {
    case runtimeError(String)
}

func someFunction() throws {
    do {
        try someOtherFunction()
    } catch {
        throw MyError.runtimeError("Failed in
```

```
someOtherFunction: \(error)")
    }
}

func someOtherFunction() throws {
    // Code that might throw an exception
}
```

Functions

Defining functions

```
func greet(name: String) {
    print("Hello, \(name)!")
}
```

Function arguments (positional, keyword, default values)

```
func greet(name: String, greeting: String = "Hello")
{
    print("\(greeting), \(name)!")
}

greet(name: "John") // Output: "Hello, John!"
greet(name: "Jane", greeting: "Hi") // Output: "Hi,
Jane!"
```

Returning values from functions

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

let sum = add(a: 3, b: 5)
print(sum) // Output: 8
```

Function overloading

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

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

Lambda functions

```
let add = { (a: Int, b: Int) -> Int in
    return a + b
}

let result = add(3, 5)
print(result) // Output: 8
```

Recursion

```
func factorial(n: Int) -> Int {
```

```
    if n == 0 {
        return 1
    }
    return n * factorial(n: n - 1)
}

let fact = factorial(n: 5)
print(fact) // Output: 120
```

Generators and iterators

```

// Generator example
func fibonacci() -> AnyIterator {
    var a = 0
    var b = 1
    return AnyIterator {
        defer { (a, b) = (b, a + b) }
        return a
    }
}

for fib in fibonacci().prefix(10) {
    print(fib)
}

// Iterator example
struct Countdown: Sequence, IteratorProtocol {
    var count: Int

    mutating func next() -> Int? {
        if count == 0 {
            return nil
```

```
        } else {
            defer { count -= 1 }
            return count
        }
    }
}

var countdown = Countdown(count: 5)
for num in countdown {
    print(num)
}
```

Decorators

```
// Decorator example
func logged(originalFunction: () -> ()) {
    print("Calling function...")
    originalFunction()
}

func myFunction() {
    print("Executing myFunction...")
}

logged(originalFunction: myFunction)
```

Higher-order functions

```
// Higher-order function example
func applyOperation(a: Int, b: Int, operation: (Int,
Int) -> Int) -> Int {
```

```
    return operation(a, b)
}

let addition = applyOperation(a: 3, b: 5, operation:
{ $0 + $1 })
let multiplication = applyOperation(a: 3, b: 5,
operation: { $0 * $1 })

print(addition) // Output: 8
print(multiplication) // Output: 15
```

Function documentation and comments

```
/// This function greets a person.
/// - Parameters:
///   - name: The name of the person to greet.
///   - greeting: The greeting message (default is
"Hello").
func greet(name: String, greeting: String = "Hello")
{
    print("\(greeting), \(name)!")
}
```

OOP

Class creation syntax

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

```
    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 instances of a class

```
var person = Person(name: "John", age: 30)
person.greet() // Output: "Hello, my name is John and
I am 30 years old."
```

Class attributes vs instance attributes

```
class MyClass {
    static var classAttribute = "Class attribute"
    var instanceAttribute = "Instance attribute"
}

print(MyClass.classAttribute) // Output: "Class
attribute"

var obj = MyClass()
print(obj.instanceAttribute) // Output: "Instance
attribute"
```

Constructor method (init method)

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

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

Instance methods vs static methods vs class methods

```
class MyClass {
    var name: String

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

    func instanceMethod() {
        print("Instance method called")
    }

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

    class func classMethod() {
        print("Class method called")
```

```
    }
}

var obj = MyClass(name: "Object")
obj.instanceMethod()
MyClass.staticMethod()
MyClass.classMethod()
```

Encapsulation (public vs private members)

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

    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.")
    }
}

var person = Person(name: "John", age: 30)
person.greet() // Output: "Hello, my name is John and
I am 30 years old."
// print(person.name) // Error: 'name' is
inaccessible due to 'private' protection level
```

Inheritance

```
class Animal {
    var name: String

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

    func makeSound() {
        print("Animal makes a sound")
    }
}

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

var dog = Dog(name: "Buddy")
dog.makeSound() // Output: "Dog barks"
```

Polymorphism (method overriding, method overloading)

```
class Animal {
    func makeSound() {
        print("Animal makes a sound")
    }
}

class Dog: Animal {
```

```
    override func makeSound() {
        print("Dog barks")
    }

    func makeSound(volume: Int) {
        print("Dog barks with volume \(volume)")
    }
}

var dog = Dog()
dog.makeSound() // Output: "Dog barks"
dog.makeSound(volume: 5) // Output: "Dog barks with volume 5"
```

Composition vs inheritance

```
// Composition example
class Engine {
    func start() {
        print("Engine started")
    }
}

class Car {
    var engine = Engine()

    func start() {
        engine.start()
        print("Car started")
    }
}
```

```
var car = Car()
car.start() // Output: "Engine started" followed by "Car started"
```

Abstract classes/interfaces

```
protocol Drawable {
    func draw()
}

class Circle: Drawable {
    func draw() {
        print("Drawing circle")
    }
}

class Rectangle: Drawable {
    func draw() {
        print("Drawing rectangle")
    }
}

var circle = Circle()
var rectangle = Rectangle()
circle.draw() // Output: "Drawing circle"
rectangle.draw() // Output: "Drawing rectangle"
```