var coins: Int = 10

coins = 20

let pi: Double = 3.14159

var coins = 10 // Swift infers that the coins variable is of type Int

// An array of 4 integers  
var numbers: [Int] = [1, 2, 3, 4]  
  
// An empty array with type inference  
var emptyArray = [Double()]  
  
// A dictionary of type [Int: String]  
var codes: [Int: String] = [200: "OK", 404: "Not Found"]

// A set of strings  
var fruits: Set<String> = ["Kiwi", "Plum", "Fig"]

let hello = "Hello, "  
let name = "Engineer"  
let salutation = hello + name // Output: "Hello, Engineer"

let multilineString = """

This is a multi-line string literal.

It can span multiple lines without needing escape characters.

You can include special characters like \n, \r, etc. directly.

"""

let myIQ = 180

let message = "My IQ is \(myIQ)." // Output: "My IQ is 180."

let stateTuple = ("Hawaii", 50, true)  
let stateNamedTuple = (name: "Alaska", number: 49, visited: false)

print(stateNamedTuple.name) // Alaska  
print(stateTuple.1) // 50

var height: Float? // height is an optional Float type

let optionalNumber: Int? = nil

let unwrappedNumber: Int = optionalNumber!

let optionalName: String? = "Lee"  
if let unwrappedName = optionalName {  
 print("Name: \(unwrappedName)")  
} else {  
 print("No name provided.")  
}guard let secondName = optionalName else {  
 print("Second Name is nil")  
 return  
}  
print("Second Name: \(secondName)")

if let carCount = jim.house?.numberOfCars {  
 print("Jim's house has \(carCount) car(s).")  
} else {  
 print("Not able to obtain the number of Jim’s cars.")

}

let optionalNumber: Int? = nil  
let result = optionalNumber ?? -1  
print(result) // Output: -1

extension String {

func invert() -> String {

return String(self.reversed())

}

}

let ourString = "ABYZ"

print(ourString.invert()) // Output: "ZYBA"

let age = 25  
if age >= 18 {

print("You are defined as an adult!")  
}

let score = 85

if score >= 90 {

print("Superb!")

} else if score >= 80 {  
 print("Good job!")

} else if score >= 70 {  
 print("You can do better!")

}

switch value {  
case pattern1:  
 // Code to execute when value matches pattern1

case pattern2:  
 // Code to execute when value matches pattern2

default:  
 // Code to execute when none of the cases match  
}

switch dayOfTheWeek {  
case "Monday", "Tuesday", "Wednesday", "Thursday", "Friday":

print("It is a Weekday.")

case "Saturday", "Sunday":  
 print("It is a Weekend.")

default:  
 print("Day is invalid.")

}

switch number {

case let num where num % 2 == 0:

print("\(number) is even.")

case let num where num % 2 != 0:

print("\(number) is odd.")

default:

print("\(number) is neither odd nor even.")

}

switch selectedFruit {  
case .apple, .orange, .banana:  
 print("Edible Fruit.")

@unknown default:  
 print("Selected an unknown fruit.")

}

let isLoggedIn = true  
let message = isLoggedIn ? "Welcome back!" : "Please log in."  
print(message) // Output: "Welcome back!"

for item in [1, 2, 3, 4, 5, 6, 7, 8] {

print(item)

}

for index in 0..<8 {  
 print(index)

}

let numbers = [1, 2, 3, 4, 5, 6, 7, 8]

// traverse the array in reverse order:

for number in numbers.reversed() {  
 print("Number is \(number)")

}

// the same, but using the forEach() function:

numbers.reversed().forEach { number in

print("Number is \(number)")

}

// count backwards by 2’s using the stride() and from, to and by:

for number in stride(from: 8, to: 0, by: -2) {

print("Number is \(number)")

}

for char in "Hello, World!" {

print(char)

}

let scores = ["Tom": 210, "Robert": 280, "Harry": 116]

for (name, score) in scores {  
 print("\(name): \(score)")

}

while condition {

// Code to be executed for as long as the condition evaluates to true

}

var userAge: Int?

while userAge == nil || userAge! <= 1 {  
 print("Enter your age:")  
 userAge = Int(readLine() ?? "")

}

print("You have entered your age as: \(userAge!)")

repeat {

// Code...

} while condition

let maxRolls = 10  
var roll: Int  
var attempts = 0

repeat {

roll = Int.random(in: 1...6)

attempts += 1

print("Attempt Number: \(attempts): Rolled: \(roll)")

} while attempts < maxRolls && roll != 1

outerLoop: for x in 1...7 {

for y in 1...7 {

if y == x {  
 continue outerLoop // Note that y is always < x  
 }

print("x = \(x), y = \(y)")

}

}

let simplestClosure = {  
 print("This is a simple closure.")  
}

simplestClosure() // Output: "This is a simple closure."

**let** greetingClosure = { (name: String) **in**  
 print("Hello, \(name)!")  
}  
greetingClosure ("Donald") // Output: "Hello, Donald!"

let addClosure = { (x: Int, y: Int) -> Int in  
 return x + y  
}

let result = addClosure(2, 7)  
print(result) // Output: 9

func makeCounter() -> () -> Int {

var myCounter = 0  
 let count: () -> Int = {

myCounter += 1  
 return myCounter

}

return count  
}

let counter1 = makeCounter()  
print(counter1 ()) // Output: 1  
print(counter1 ()) // Output: 2  
print(counter1 ()) // Output: 3

let counter2 = makeCounter()  
print(counter2 ()) // Output: 1

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

let result1 = performOp(a: 5, b: 3) { (a, b) in  
 return a \* b  
}  
print(result1) // Output: 15

let result2 = performOp(a: 8, b: 4) { $0 + $1 } // add 1st and 2nd argument  
print(result2) // Output: 12

func performLater(completion: @escaping () -> Void) {

// Perform operation after 3 seconds

DispatchQueue.main.asyncAfter(deadline: .now() + 3) {

completion()

}

}

performLater {

print("Delayed execution!")

}

func printResult(\_ resultClosure: @autoclosure () -> Int) {  
 print("Result: \(resultClosure())")  
}  
printResult(7 + 2) // Result: 9

func executeLater(\_ closure: @escaping @autoclosure () -> Void) {

DispatchQueue.main.asyncAfter(deadline: .now() + 2) {

closure()

}

}

executeLater(print("Delayed execution closure done!"))

let fiveNumbers = [1, 2, 3, 4, 5]

let squaredNumbers = fiveNumbers.map { $0 \* $0 }

// squaredNumbers is now [1, 4, 9, 16, 25]

let fiveNumbers = [1, 2, 3, 4, 5]

let oddNumbers = fiveNumbers.filter { $0 % 2 == 1 }

// oddNumbers is now [1, 3, 5]

let fiveNumbers = [1, 2, 3, 4, 5]

let sum = fiveNumbers.reduce(0) { $0 + $1 }

// sum is now 15

let fiveNumbers = ["1", "2", "3", "5", "six"]

let integerNumbers = fiveNumbers.compactMap { Int($0) }

// integerNumbers is now [1, 2, 3, 5]

let names = ["Eli", "André", "Jane", "Bobby"]

let sortedNames = names.sorted { $0 < $1 }

// sortedNames is now ["André", "Bobby", "Eli", "Jane"]

let fiveNumbers = [1, 2, 3, 4, 5]

fiveNumbers.forEach { print($0) }

// Output: 1

// 2

// 3

// 4

// 5

let fiveNumbers = [1, 2, 3, 4, 5]

let result = fiveNumbers.filter { $0 % 2 == 0 }.map { $0 \* $0 }

// result is now [4, 16]

class Auto {

var make: String

var model: String

var year: UInt32

var manufacturer: AutoManufacturer?

init(make: String, model: String, year: UInt32) {

self.make = make

self.model = model

self.year = year

}

}

let myCar = Auto(make: "Ferrari", model: "Berlinetta", year: 2025)

print("\(myCar.make) \(myCar.model) (\(myCar.year))")

// Output: Ferrari Berlinetta (2025)

protocol FullyElectric {  
 var batteryCapacity: Double { get }  
 func charge()  
}

class ElectricAuto: Auto, FullyElectric {  
 var batteryCapacity: Double

init(make: String, model: String, year: UInt32, capacity: Double) {  
 self.batteryCapacity = capacity  
 super.init(make: make, model: model, year: year)  
 }

func charge() {  
 print("Charging the electric car.")  
 }  
}

class Auto {

// Properties

// make, model, year, manufacturer ...

// and initializer(s) ...

func startEngine() {

print("\(make) \(model) engine started.")

}

}

enum AutoManufacturer {  
 case porsche, jaguar, corvette, bmw, mercedes, none  
}

let myCar = Auto(make: "BMW", model: "530i", year: 2024)  
myCar.manufacturer = .bmw

print("My car's manufacturer: \(myCar.manufacturer)")  
// Output: My car's manufacturer: bmw

func multiply(\_ x: Int, \_ y: Int) -> Int { return x \* y }

class Tractor {  
 var model: String  
 var horsepower: Int

init() { // the default initializer

model = "John Deere"  
 horsepower = 400  
 }

}

let newTractor = Tractor()  
print(newTractor.model) // John Deere  
print(newTractor.horsepower) // 400

class Person {  
 var firstName: String  
 var weight: UInt

// Default initializer  
 init() {

self.firstName = "Jimmy Durante"  
 self.weight = 140

}

// Custom initializer  
 init(name: String, weight: Int) {

self.firstName = name  
 self.weight = weight

}

}  
  
let defaultPerson = Person()  
let customPerson = Person(name: "Jill", weight: 110)

class Cat {

var name: String

// ...

func makeNoise() {

// default implementation for all Cats

print("meow")

}

}

class Lion: Cat {

override func makeNoise () {

print("Roar!")

}

}

**func** swapValues<T>(\_ x: **inout** T, \_ y: **inout** T) {

**let** temp = x

x = y

y = temp

}

struct MyStack<StackItem> {

private var items: [StackItem] = []

mutating func push(\_ item: StackItem) {  
 items.append(item)

}

  mutating func pop() -> StackItem? {

return items.popLast()

}

}

enum Result<Success, Failure: Error> {  
 case success(Success)  
 case failure(Failure)  
}

var names: Array<String> = ["Alice", "Bob", "Charlie"] // same as names: [String]

enum Optional<Wrapped> {

case none

case some(Wrapped)

}

func encodeToJSONString<T: Codable>(\_ value: T) -> String? {

let encoder = JSONEncoder()  
 // Encoding functionality ...  
}

func findIndex<T: Equatable>(of value: T, in array: [T]) -> Int? {

for (index, element) in array.enumerated() {

if element == value {

return index

}

}

return nil

}

// Good:

let maximumNumberOfRetries = 5  
func fetchServerData(from: ...) { ... }  
class CustomerProfileViewController { ... }

// Not So Good:

let maxR = 5  
func srvrData() { ... }  
class CPVC { ... }

// Good:

class DataModel { ... }

var userName: String  
 func calculateTotal() { ... }  
}

// Not So Good:  
class datamodel { ... }

var Username: String  
 func calculate\_total\_amount() { ... }  
}// Good:

if condition {

print("True condition")

} else {

print("False condition")

}

// Not So Good:

if condition {

print("True condition")

} else {

print("False condition")

}

let result = functionWithLotsOfParameters(

firstParameter: "First",

secondParameter: "Second",

thirdParameter: "Third"

)

// Good:  
func calculateArea(of item: Shape) -> Double { ... } 

// Not So Good:

// Function where we calculate the area of shapes  
func area(\_ s: Shape) -> Double { ... }