Swift Progrmmaing Language

application

- native
  - android application using Java or Kotlin
  - platform (OS) specific
  - most of the times, developed using compile languages like C or C++ or Go or Rust
  - the code gets compiled into Assembly language
  - these apps are faster than web or hybrid applications
- web applications
  - web apps using JS or TS
  - platform or OS independent
  - most of the times, developed using interpreted languages like html, JS, TS, Perl, Python, PHP, Ruby, Java
- hybrid applications
  - application developed in Flutter or React Native

what is swift

- Chris Lattner has developed this language
- is a
  - object oriented programming language
  - functional programming language
    - functions are considered as first class citizens
      - variables of type function
    - a function can be passed as an argument to another function
    - a function can be returned as return value of a function
  - scripting language
    - interpreted language
    - simpler than real programming languages (C)
    - does not support pointers
  - compiled language
- features
  - type safe language
  - can be used for developing
    - console applications
    - server side applications
    - desktop applications
    - iOS applications
    - watchOS applications
    - tvOS applications
  - semicolon is optional when statements are written per line
  - semicolon is mandatory when multiple statements are written on same line

installation

- download the Xcode from app store
- install Xcode on your machine
- Xcode by default install swift and objective-c language compiler and interpreter

identifiers

- used to identify variables/constants/functions/classes/enums and protocols
- rules
  - can not use special characters
    - e.g. first name = "steve" <- invalid
    - e.g. firstName = "steve" <- valid
  - variable can not start with number
    - e.g. 1name = "steve" <- invalid
    - e.g. firstName = "steve" <- valid
  - can not use reserved words (keywords)
    - e.g. if = "test" <- invalid
- **variable**
  - mutable (can change its state)
  - identifier which stores value in the memory
  - can change its value
  - **declaration**
    - implicit declaration
      - variable's data type will be inferred by Swift
      - Swift infers the data type by validating value stored in it
      - syntax

        -
          ```
          // var <variable name> = <variable value>
          ```
      - e.g.
        ```
        // data type is integer
        var num = 100
        ```
      - to get data type of a variable use typeof function

```swift
var num = 100
print("num = \(num)")
print("type of num = \(type(of: num))")
```

            - explicit declaration
                - data type is explicitly give at the time of declaration
                - syntax
                    - : =
                - e.g.

```swift
let num: Int = 100
```

- **constant**
    - immutable (can not change its state)
    - variable which does not change its value
    - to declare a constant use **let** keyword
    - e.g.

```swift
// constant
let num = 100

// can not change its value
// num = 200
```

    - **NOTE**: try to use let as much as you can

data types

- system defined
    - **Int**
        - represent whole number
        - decimal is not allowed
        - types

        - signed
            - Int8 - 1 byte (short)
            - Int16 - 2 bytes
            - Int32 - 4 bytes (int)
            - Int64 - 8 bytes (long)
        - unsigned
            - UInt8 - 1 byte (short)
            - UInt16 - 2 bytes
            - UInt32 - 4 bytes (int)
            - UInt64 - 8 bytes (long)
    - e.g.

      ```
      // implicit
      let num = 100

      // explicit
      let num2: Int8 = 100
      let num3: Int16 = 10000
      ```

- **Float**
    - represents a decimal value
    - types
        - Float - 4 bytes
        - Float32 - 4 bytes
        - Float64 - 8 bytes
        - Float80 - 10 bytes
    - e.g.

      ```
      let salary: Float = 10.15
      ```

- **Double**
    - represents a large decimal value
    - takes 8 bytes to store the value
    - can store values up 2E-308
    - by default swift declares a decimal value as double

        - e.g.

          ```swift
          let value = 1231213312.13122113
          ```

    - **Character**

        - represents a single character
        - e.g.

      ```swift
      let ch2: Character = "b"
      ```

    - **String**

        - collection of characters

        - e.g.

          ```swift
          let firstName = "Steve"
          let lastName: String = "Jobs"
          ```

    - **Bool**

        - represents true or false
        - e.g.

      ```swift
      let canVote = true
      let isHidden: Bool = false
      ```

    - **Optional**

        - data type which allows a variable to store a value optionally

        - to create optional variable use ? with data type

        - e.g.

          ```swift
          // allows firstName to store nil
          var firstName: String? = nil
          ```

```
var lastName = "jobs"

// Optional("jobs")
print(type(of: lastName))
```

- **unwrap**
  - get the value stored in the optional variable
  - to unwrap a variable use ! with variable name
  - e.g.

    ```
    var num: Int? = 20

    // num = Optional(20), type = Optional<Int>
    print("num = \(num), type = \(type(of: num))")


    // unwrap the value and store inside another variable
    let num2 = num!

    // num2 = 20, type = Int
    print("num2 = \(num2), type = \(type(of: num2))")
    ```

  - **application receives a fatal error when the nil gets unwrapped**
  - e.g.

    ```
    var num: Int? = nil

    // Optional(nil)
    print(num)

    // application crashes
    print(num!)
    ```

  - while unwrapping the value from optional variable, first make sure the value is non-nil
    - e.g

    ```
    var num: Int? = nil
    ```

```
  // print num's value only when num is not nil
  if num != nil {
    // unwrap the value to get the original value stored
in the variable
    print(num!)
  }


  // print num's value only when num is not nil
  if let num2 = num {
    // num2 can not be unwrapped as it is the original
value (Int) and
    // not an optional value
    print(num2)
  }

  // print num's value only when num is not nil
  if let num = num {
    // num can not be unwrapped as it is the original
value (Int) and
    // not an optional value
    print(num)
  }
```

- user defined
  - class
  - struct
  - enum

function

- reusable code unit
- syntax

```
  func <function name> ( <external name> <internal name> : <data
type>, ..) -> <return type> {
      // function body
  }
```

- note
  - external name can not be used internally
  - internal name can not be used externally
- e.g.

```
// parameterless empty function
func emptyFunction() {}
emptyFunction()

// parameterless function
func myFunction1() {
    print("inside my function")
}

myFunction1()

// parameterized function
func myFunction2(param1: Int, param2: Int) {
    print("inside myFunction2")
    print("param1 = \(param1), param2 = \(param2)")
}

myFunction2(param1: 10, param2: 20)

func add(p1: Int, p2: Int) -> Int {
    print("inside add")
    return p1 + p2
}

let addition = add(p1: 10, p2: 20)
print("addition = \(addition)")
```

- to call functions without parameter name

```
// use external name as underscore (_)
func myFunction3(_ p1: Int, _ p2: Int) {
    print("inside myFunction3")
    print("p1 = \(p1)")
    print("p2 = \(p2)")
}

// can not pass values along with parameter name
myFunction3(10, 30)
```

collections

- array
    - collection of similar ordered values
    - respects the insertion order

- allows duplicate values
- e.g.

  ```
  // implicit
  let numbers = [10, 20, 30, 40, 50]
  print("\(numbers), type = \(type(of: numbers))")

  // explicit
  let countries: Array<String> = ["india", "usa", "uk", "japan"]
  print("\(countries), type = \(type(of: countries))")

  let salaries: [Float] = [10.15, 8.5, 6.7, 8.4]
  print("\(salaries), type = \(type(of: salaries))")
  ```

- empty can be declared with the following syntax
- e.g.

  ```
  // implicit
  // creating an empty array to store Ints
  let emptyIntArray = [Int]()
  print("\(emptyIntArray), type = \(type(of: emptyIntArray))")

  // explicit
  let emptyStringArray: [String] = []
  print("\(emptyStringArray), type = \(type(of:
  emptyStringArray))")
  ```

- mutable vs immutable array
  - mutable: dynamic modifications are allowed
    - e.g.

    ```
    // mutable
    var array1 = [10, 20, 30, 40, 50]
    array1.append(60)
    print(array1)
    ```

  - immutable: dynamic modifications are not allowed
    - e.g.

```
// immutable
let array2 = [10, 20, 30, 40, 50]
// array2.append(60)
```

- **operations**
    - **appending values**
        - used to add a new value to the end of the array

```
var array = [10, 20, 30, 40, 50]

 // appending a new value
 array.append(60)

 // appending a new value
 // usefull when adding members of one array to another
 array += [70]
```

    - **removing values**

```

var array = [10, 20, 30, 40, 50]

// remove first
array.removeFirst()

// remove last
array.removeLast()

// remove a value from array
array.remove(at: 3)

// remove all values from array
array.removeAll()
```

- **properties**
    - **count**
        - returns number of values in an array
        - e.g.

```

let array = [10, 20, 30, 40, 50]
```

```
// 5
print("count = \(array.count)")
```

- **isEmpty**
    - checks if the array is empty

```
let array = []

// true
print("isEmpty = \(array.isEmpty)")
```

- tuple
    - immutable collection of similar or dissimilar values
    - e.g.

```
  // implicit
  let mobile1 = ("iPhone XS Max", 144000)
  print("model: \(mobile1.0)")
  print("price: \(mobile1.1)")

  // explicit
  let mobile2: (String, Int) = ("iPhone XS Max", 144000)
  print("model: \(mobile2.0)")
  print("price: \(mobile2.1)")

  // implicit
  let mobile3 = (model: "iPhone XS Max", price: 144000)
  print("model: \(mobile3.model)")
  print("price: \(mobile3.price)")

  // explicit
  let mobile4: (model: String, price: Int) = (model: "iPhone XS Max", price: 144000)
  print("model: \(mobile4.model)")
  print("price: \(mobile4.price)")
```

- set
    - collection of unique values
    - may or may not keep the insertion order
    - can not be declared implicitly
    - e.g.

```swift
let s1: Set<Int> = [10, 20, 30, 40]
let s2: Set<Int> = [40, 50, 60, 70]

// [40]
print("s1 intersection s2 = \(s1.intersection(s2))")

// [40]
print("s2 intersection s1 = \(s2.intersection(s1))")

// [10, 20, 30, 40, 50, 60, 70]
print("s1 union s2 = \(s1.union(s2))")

// [10, 20, 30, 40, 50, 60, 70]
print("s2 union s1 = \(s2.union(s1))")

// [10, 20, 30]
print("s1 - s2 = \(s1.subtracting(s2))")

// [50, 60, 70]
print("s2 - s1 = \(s2.subtracting(s1))")
```

- dictionary
    - collection of key-value pairs
    - key should be always unique, value can be repeated
    - key and value can be of any data type
    - always returns optional value

built-in operators

- mathematical operators
- relational operators
- comparison operators
- logical operators
- bitwise operators
- range operator
    - operator used to create a range from lower to upper bound
- types
    - **closed range operator**
        - includes the upper bound
        - e.g.

```
// closed range operator
for index in 0...3 {
  print("index = \(index)")
}

// index = 0
// index = 1
// index = 2
// index = 3    <--- 3 (upper bound is included)
```

  - **half open range operator**
    - excludes the upper bound
    - e.g.

```
// closed range operator
for index in 0..<3 {
  print("index = \(index)")
}

// index = 0
// index = 1
// index = 2
```

built-in values

- **nil**
  - similar to null
  - value is not present

closure

object oriented programming

struct

- user defined data type
- collection of similar or dissimilar data types
- is a **value** type
- does not support inheritance
- e.g.

```
// struct declaration
struct Person {
  var name: String
  var age: Int
  var phone: String
}

// struct instantiation
let p1 = Person(name: "person1", age: 30, phone: "+91235435")
print(p1)
```

class

- user defined data type
- collection of similar or dissimilar data types
- is a **reference** type
- supports inheritance
- e.g.

```
class Mobile {
    var model: String?
    var company: String?
    var price: Float?

    // initailizer
    init() {}
    init(model: String, company: String, price: Float) {
        self.model = model
        self.company = company
        self.price = price
    }

    // deinit
    deinit {
        print("iniside deinit")
    }

    // gtter
    func setModel(model: String) { self.model = model }
    func setCompany(company: String) { self.company = company }
    func setPrice(price: Float) { self.price = price }

    // setter
    func getModel() -> String? { return self.model }
```

```
        func getCompany() -> String? { return self.company }
        func getPrice() -> Float? { return self.price }

        // facilitator
        func printInfo() {
            print("model: \(model!)")
            print("company: \(company!)")
            print("price: \(price!)")
        }
    }

    let m1 = Mobile(model: "z10", company: "BlackBerry", price: 40000)
    m1.printInfo()
```

- constructor vs initializer
  - constructor and initializer are used to initialize the data members
  - constructor has same name as that of the class (in C++, Java) while
    - initializer in Swift will always have name as init()
  -

properties

- **stored properties**
  - propertie used to store a value within a class
  - e.g.

    ```
    struct Point {
      // stored properties
      var x: Int
      var y: Int
    }
    ```

- **computed properties**
  - do not store any value
  - values will be computed instead of storing within the object
  -

protocol

- similar to interface in other languages
- blueprint of methods and properties

- e.g.

```
// protocol declaration
protocol Shape {
    func draw()
    func erase()
}

// conform to the protocol
struct Rectangle : Shape {

    func draw() {
        print("rectangle is drawing")
    }

    func erase() {
        print("rectangle is getting erased")
    }
}


// conform to the protocol
class Square: Shape {

    func draw() {
        print("Square is drawing")
    }

    func erase() {
        print("Square is getting erased")
    }
}
```