Introduction:

Swift provides a variety of control flow statements. These include `while` loops to perform a task multiple times; `if`, `guard`, and `switch` statements to execute different branches of code based on certain conditions; and statements such as `break` and `continue` to transfer the flow of execution to another point in your code

`for-in` loop

You use the `for-in` loop to iterate over a sequence, such as items in an array, ranges of numbers, or characters in a string.

```
let names = ["Anna", "Alex", "Brian", "Jack"]

for name in names {

print("Hello, \(name)!")

}

// Hello, Anna!

// Hello, Alex!

// Hello, Brian!

// Hello, Jack!
```

You can also iterate over a dictionary to access its key-value pairs

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]

for (animalName, legCount) in numberOfLegs {

print("\(animalName)s have \(legCount) legs")

}

// cats have 4 legs

// ants have 6 legs
```

```
// spiders have 8 legs
```

You can also use `for-in` loops with numeric ranges

```
for index in 1...5 {

print("\(index) times 5 is \(index * 5)")

}

// 1 times 5 is 5

// 2 times 5 is 10

// 3 times 5 is 15

// 4 times 5 is 20

// 5 times 5 is 25
```

If you don't need each value from a sequence, you can ignore the values by using an underscore in place of a variable name.

```
let base = 3

let power = 10

var answer = 1

for _ in 1...power {

answer *= base

}

print("\(base) to the power of \(power) is \(answer)")

// Prints "3 to the power of 10 is 59049"
```

In some situations, you might not want to use closed ranges, which include both endpoints. Consider drawing the tick marks for every minute on a watch face

```
let minutes = 60

for tickMark in 0..<minutes {

// render the tick mark each minute (60 times)

}
```

Some users might want fewer tick marks in their UI. They could prefer one mark every `5` minutes instead. Use the `stride(from:to:by:)` function to skip the unwanted marks.

```
let minuteInterval = 5

for tickMark in stride(from: 0, to: minutes, by: minuteInterval) {

// render the tick mark every 5 minutes (0, 5, 10, 15 ... 45, 50, 55)

}
```

Closed ranges are also available, by using `stride(from:through:by:)` instead

```
let hours = 12

let hourInterval = 3

for tickMark in stride(from: 3, through: hours, by: hourInterval) {

// render the tick mark every 3 hours (3, 6, 9, 12)

}
```

while loop

```
var index = 10

while index < 20 {

  print( "Value of index is \(index)")

  index = index + 1
```

```
}
```

repeat...while Loop

```
var i = 1, n = 5

// repeat...while loop from 1 to 5

repeat {

 print(i)

 i = i + 1

} while (i <= n)
```

Every `switch` statement consists of multiple possible *cases*, each of which begins with the `case` keyword. In addition to comparing against specific values, Swift provides several ways for each case to specify more complex matching patterns

```
let someCharacter: Character = "z"

switch someCharacter {

case "a":

print("The first letter of the alphabet")

case "z":

print("The last letter of the alphabet")

default:

print("Some other character")

}

// Prints "The last letter of the alphabet"
```

The body of each case *must* contain at least one executable statement. It isn't valid to write the following code, because the first case is empty

```
let anotherCharacter: Character = "a"

switch anotherCharacter {

case "a": // Invalid, the case has an empty body

case "A":

print("The letter A")

default:

print("Not the letter A")

}

// This will report a compile-time error.
```

To make a `switch` with a single case that matches both `"a"` and `"A"`, combine the two values into a compound case, separating the values with commas.

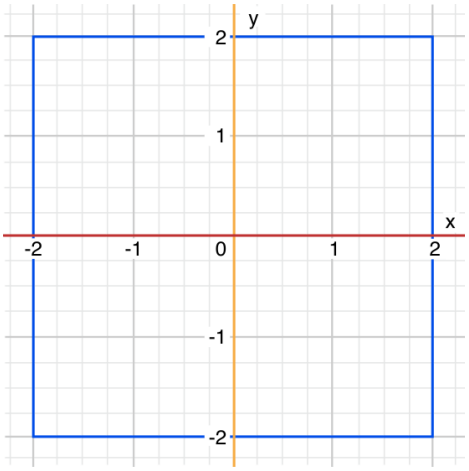
```
let anotherCharacter: Character = "a"

switch anotherCharacter {

case "a", "A":

print("The letter A")

default:

print("Not the letter A")

}

// Prints "The letter A"
```

Values in `switch` cases can be checked for their inclusion in an interval. This example uses number intervals to provide a natural-language count for numbers of any size:

```
let approximateCount = 62

let countedThings = "moons orbiting Saturn"

let naturalCount: String

switch approximateCount {

case 0:

naturalCount = "no"

case 1..<5:

naturalCount = "a few"

case 5..<12:

naturalCount = "several"

case 12..<100:

naturalCount = "dozens of"

case 100..<1000:

naturalCount = "hundreds of"

default:

naturalCount = "many"

}

print("There are \(naturalCount) \(countedThings).")

// Prints "There are dozens of moons orbiting Saturn."
```

You can use tuples to test multiple values in the same `switch` statement. Each element of the tuple can be tested against a different value or interval of values. Alternatively, use the underscore character (_), also known as the wildcard pattern, to match any possible value

```
let somePoint = (1, 1)

switch somePoint {

case (0, 0):

print("\(somePoint) is at the origin")

case (_, 0):

print("\(somePoint) is on the x-axis")

case (0, _):

print("\(somePoint) is on the y-axis")

case (-2...2, -2...2):

print("\(somePoint) is inside the box")

default:

print("\(somePoint) is outside of the box")

}

// Prints "(1, 1) is inside the box"
```



Control transfer statements change the order in which your code is executed, by transferring control from one piece of code to another. Swift has five control transfer statements:

- `continue`
- `break`
- `fallthrough`
- `return`
- `throw`

```
let puzzleInput = "great minds think alike"

var puzzleOutput = ""

let charactersToRemove: [Character] = ["a", "e", "i", "o", "u", " "]

for character in puzzleInput {

if charactersToRemove.contains(character) {

continue

}

puzzleOutput.append(character)

}

print(puzzleOutput)

// Prints "grtmndsthnklk"
```

The following example switches on a `Character` value and determines whether it represents a number symbol in one of four languages. For brevity, multiple values are covered in a single `switch` case

```
let numberSymbol: Character = "三" // Chinese symbol for the number 3

var possibleIntegerValue: Int?

switch numberSymbol {

case "1", "١", "一", "๑":

possibleIntegerValue = 1

case "2", "٢", "二", "๒":

possibleIntegerValue = 2
```

```
case "3", "٣", "三", "๓":

possibleIntegerValue = 3

case "4", "٤", "四", "๔":

possibleIntegerValue = 4

default:

break

}

if let integerValue = possibleIntegerValue {

print("The integer value of \(numberSymbol) is \(integerValue).")

} else {

print("An integer value couldn't be found for \(numberSymbol).")

}

// Prints "The integer value of 三 is 3."
```

In Swift, `switch` statements don't fall through the bottom of each case and into the next one. That is, the entire `switch` statement completes its execution as soon as the first matching case is completed. By contrast, C requires you to insert an explicit `break` statement at the end of every `switch` case to prevent fallthrough

```
let integerToDescribe = 5

var description = "The number \(integerToDescribe) is"

switch integerToDescribe {

case 2, 3, 5, 7, 11, 13, 17, 19:

description += " a prime number, and also"

fallthrough
```

```
    default:

    description += " an integer."

    }

    print(description)

    // Prints "The number 5 is a prime number, and also an integer."
```

You use an *availability condition* in an `if` or `guard` statement to conditionally execute a block of code, depending on whether the APIs you want to use are available at runtime

```
    if #available(iOS 10, macOS 10.12, *) {

    // Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS

    } else {

    // Fall back to earlier iOS and macOS APIs

    }
```