Swift Overview
Introducing Swift’s features
[1] Swift tour
1. Simple Values
Implicit declaration

```swift
import Swift

1. Simple Values

1.1 Implicit Declaration

var for variables

var language = "Swift"
language = "Objective C"

let for constants

let constantLanguage = "Swift"
```

4
Explicit declaration

1.2 Explicit Declaration

```javascript
let explicitString: String = "Hello"
let explicitInteger: Int = 38
let explicitConstantString: String = "fixed"
```

"Hello"

10

"fixed"
1.3 Variable typecasting
You must explicitly typecast

\begin{verbatim}
let label = "The width is 
let width = 94
let widthLabel = label + String(width)

Use \texttt{\textbackslash\textbackslash\textbackslash} to evaluate values within strings.

let widthLabel = label + "\textbackslash\textbackslash\textbackslash\\width"
\end{verbatim}

"The width is 94"
Array - Dictionaries

1.4 Arrays
Use []

```javascript
var summerMonths = ["June", "July", "August"]
var summerMonths[2] = "July"
```

1.5 Dictionaries
Use {},

```javascript
var courseGrades = {
  "Eng101": "A",
  "Eng102": "B"
}
var courseGrades["Eng101"] = "A"
```
## 1.6 Empty collections

### Arrays

```javascript
var emptyArray = []
emptyArray = []
```

### Dictionaries

```javascript
var emptyDictionary = {};
emptyDictionary = {};
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td></td>
<td>0 elements</td>
</tr>
<tr>
<td>Dictionary</td>
<td></td>
<td>0 key/value pairs</td>
</tr>
</tbody>
</table>
2. Control Flow
Conditionals - if

2. Control Flow

2.1 Conditionals
Use if and switch for conditionals

// Generate random int in range 1...9
let random = arc4random_uniform(8)

if random < 5 {
    print("low")
} else {
    print("high")
}

var optionalString: String? = "Hello"
prompt?("Please enter your name")
var optionalName: String? = "John Appleseed"
var greeting = "Hello"

if let name = optionalName {
    greeting = "Hello \(name)"
}

println("Hello")
println(""Hello John Appleseed"")
// switches are really powerful
let veg = "red pepper"

switch veg {
  case "lemon":
    let comment = "Add some raisins and make ants on a log."
    break;
  case "cucumber", "watermelon":
    let comment = "That would make a good tea sandwich."
    break;
  case let x: where x.name == "pepper":
    let comment = "Is it the right pepper?"
    break;
  default:
    let comment = "Everything tastes good in soup."
}

"red pepper"
"Is it a spicy red pepper?"
Loops - for-in

2.2 Loops
Use for-in, for, while, do-while to make loops

for-in with arrays

```javascript
// for-in enumerates through all the values
let individualScores = [75, 43, 183, 87, 12]
var teamScore = 0

for score in individualScores {
    if score > 50 { teamScore += 3 }
    else { teamScore += 1 }
}

println(teamScore)
```

"11"
Loops - for-in

```javascript
for-in with dictionaries

let interestingNumbers = {
    "Prime": [2, 3, 5, 7, 11, 13],
    "Palindrome": [1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25]
}

var largestNumberOverall = 0
var largestNumberKind = ""

for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number in collection {
            largestNumberOverall = number
            largestNumberKind = kind
        }
    }
}

println("The largest number \n\(largestNumberOverall\) is of the \(largestNumberKind\) kind")
```

"The largest number 25 is of the Square kind"
Loops - while/do-while

### while loop

```javascript
var n = 2;
while (n < 100) {
    n += 2;
}
print(n);
```

### do-while loop

```javascript
var m = 2;
do {
    m += 2;
} while (m < 100);
print(m);
```
Loops - for

```java
for loop - normal
var firstLoop = 0
for var i = 0; i < 10; i++ { firstLoop += 1 }
print(firstLoop)

for loop - using a range
var secondLoop = 0
for i in 0...10 { secondLoop += 1 }
print(secondLoop)
```
3. Functions and Closures
Functions

3. Functions and Closures

3.1 Functions

Use `func` to declare a function

```java
func greet(name: String, day: String) -> String {
    return "Hello \(name), today is \(day)"
}

// Notice the absence of the argument labels

// Hi Bob, today is Tuesday
```
Functions can return tuples

```scala
18
#define calculateStatistics(scores: [Int]) -> (min: Int, max: Int, sum: Int, ave: Double) {
19 var min = scores[0]
20 var max = scores[0]
21 var sum = 0
22 var ave = 0.0
23 for score in scores {
24 min = score < min ? score : min
25 max = score > max ? score : max
26 sum += score
27 ave = Double(sum) / Double(scores.count)
28 }
29 return (min, max, sum, ave)
30 } let statistics = calculateStatistics([5, 3, 100, 3, 9])
31 printf(\"min = \%(statistics.min)\"")
32 printf(\"max = \%(statistics.max)\"")
33 printf(\"ave = \%(statistics.ave)\"")
34 ```
Functions

Functions can take a variable number of parameters

```haskell
> func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
        sum += number
    }
    return sum
}

func sumOf([2, 9, 1, 2])
```

Functions

Functions can be nested

```javascript
// Function returns an integer that takes an int and returns an int
func makeIncrementer() => int x => int y => int {
    return x + y
}
```

Functions can return another function (first-class types)

```javascript
// Function returns a function that takes an int and returns an int
func makeIncrementer() => int x => int y => int {
    return x + y
}
```

```javascript
// get the Incrementer function
var Incrementer = makeIncrementer()
```

```javascript
// now use it
Incrementer(1)
```
Functions can take functions (closures) as arguments

```javascript
function hasAnyMatches(list: [Int], condition: Int => Boolean): Int {
  var count = 0;
  var hasMatch = false
  for (let item of list) {
    if (condition(item)) hasMatch = true; count++
  }
  return hasMatch ? count : 0;
}

// The condition function
function lessThanTen(number: Int) => Boolean {
  return number < 10;
}

var numbers = [5, 4, 7, 12]
print(hasAnyMatches(numbers, lessThanTen)) // prints 2, as 5 and 4 match the condition
```

"There are 2 matches"
Closures

3.2 Closures

Closures - blocks of code that can be called later

May use a name-function, or not ()

```javascript
// using the array numbers from above, using a closure for the argument to map
let numbers = [1, 2, 3, 4, 5];
let map = function (number) {
  return function (times) {
    return number * times;
  }
};
let result = map(3);
console.log(result(5));
```

30, 37, 21, 36
Closures

```javascript
if parameters and return type can be inferred, even better notice no need for the return (single statement closure)

let inferredTripleNumbers = numbers.map((number in 3 * number))

You can use $0, $1,.. for the parameters instead

let numerator(parametersTripleNumbers = numbers.map(3 * $0))
let numerator(parametersTripleNumbers)
```
4. Objects and Classes
Create a class

4. Objects and Classes

4.1 Create a class

```java
class Shape {
    val numberSides = 0

    fun describe() : String {
        return "A shape with $numberSides sides."}
}
```

"A shape with 7 sides."
4.2 Instantiate a class

```java
// Non default initializer: init()

class Shape

    def __init__(self, numSides):
        self.numSides = numSides

    def simpleDescription(self):
        return f'A shape with {self.numSides} sides.'
```

```python
# Non default initializer: init()

class Shape

    def __init__(self, numSides):
        self.numSides = numSides

    def simpleDescription(self):
        return f'A shape with {self.numSides} sides.'
```
Using initializers

```java
4.3 Using initializers

```AnonymousShape`

<table>
<thead>
<tr>
<th>Initializer: String</th>
<th><code>setShape = name</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function: <code>String()</code></td>
<td><code>return &quot;A triangle with \$\{number\} sides.&quot;</code></td>
</tr>
<tr>
<td>Name: <code>nameShape</code></td>
<td><code>nameShape = &quot;triangle&quot;</code></td>
</tr>
</tbody>
</table>

A polygon with 3 sides.
Triangle with 3 sides.
A triangle with 3 sides.
4.4 Subclassing

```java
class Square: Shape {
    double sideLength; // side length of square

    Square(double sideLength) {
        // initialize side length
    }

    Square(double sideLength, String name) {
        // initialize side length and name
    }

    String getDescription() {
        return "A square with side length \"" + sideLength + \"\".
    }

    // other methods...
}
```

```java
// Example usage:
Square square = new Square(5.2, "My square");
System.out.println(square.getDescription());
```

A square with side length 5.2.
4.5 Computed properties can have getters/setters

```java
class EquilateralTriangle extends Shape {
    // Stored property
    var sideLength = 5.0;

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        self.name = name
    }

    // Computed properties
    var perimeter: Double {
        get {
            self.sideLength * 3.0
        }
        set (newValue: Double) {
            self.sideLength = newValue
        }
    }

    override func shapeDescription() -> String {
        return "An equilateral triangle with side of length \(sideLength).
    }

    init(sideLength: Double, name: String) {
        super.init(sideLength: sideLength, name: name)
    }

    var triangle = EquilateralTriangle(sideLength: 5.0, name: "a triangle")
    triangle.perimeter = 15.0
    triangle.sideLength = 5.0
    triangle.sideLength
```
4.6 Stored properties can be observed

```java
class Triangle
```
```java
4.6 Stored properties can be observed
```
Named parameters

4.7 Internal vs external parameter names

- **Functions**: use only internal names by default
- **Initializers**: use external names for all parameters
- **Methods**: use external names for all but the first parameter

```java
class Counter {
    var count = 0

    func increment(amount: Int, numberOfTimes: Int) { count += numberOfTimes; amount }
}

counter.increment(12, numberOfTimes: 7)
```

(count 8) (count 14)
5. Enumerations and Structures
enum - raw values

5. Enumerations and Structures

5.1 enums

Enumerations may include methods.

```swift
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King

    func simpleDescription() -> String {
        switch self {
            case .Ace: return "Ace"
            case .Jack: return "Jack"
            case .Queen: return "Queen"
            case .King: return "King"
            default: return String(self.rawValue)
        }
    }
}

let ace = Rank.Ace
let aceRawValue = ace.rawValue
```
enum - no raw values

```swift
enum Suit {
    case Spades, Hearts, Diamonds, Clubs

    func color() -> String {
        switch self {
            case .Spades, .Clubs: return "black"
            case .Hearts, .Diamonds: return "red"
        }
    }

    func simpleDescription() -> String {
        switch self {
            case .Spades: return "spades"
            case .Hearts: return "hearts"
            case .Diamonds: return "diamonds"
            case .Clubs: return "clubs"
        }
    }
}  

let hearts = Suit.Hearts
let hearts, simpleDescription()
let hearts, color()
enum - associated values

```swift
enum ServerResponse {
    case success(String, String)
    case error(String)
}

let success = ServerResponse.success("6:00 am", "6:00 pm")
let failure = ServerResponse.error("Out of cheese")

// change to failure to see other response
switch success {
    case .success(let sunrise, let sunset):
        let serverResponse = "Sunrise is at \(sunrise) and sunset is at \(sunset)"
    case .error(let error):
        let serverResponse = "Failure... \(error)"
    default:
        break
}
```

5.2 Structures

- structures are value types, classes reference types
- structures may have methods and properties

```swift
struct Card {
    var rank: Rank
    var suit: Suit

    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \(suit.simpleDescription())"
    }
}

let threeOfSpades = Card(rank: .Three, suit: .Spades)
threeOfSpades.simpleDescription() // "The 3 of spades"
```
6. Protocols and Extensions
Declare a protocol

6. Protocols and Extensions

6.1 Protocols

```swift
protocol ExampleProtocol {
    var simpleDescription: String { get }
    var makingTestsAdvisable: Bool { get }
}
```
Adopt a protocol

Classes, enumerations and structs can adopt protocols

```swift
class SimpleClass: ExampleProtocol {
    var simpleDescription = "A very simple class."
    var anotherProperty = 01234

    func adjust() {
        simpleDescription = "Now 100% adjusted." // Not required
    }
}

var a = SimpleClass()
a.adjust()

struct SimpleStructures: ExampleProtocol {
    var simpleDescription = "A simple structure"
    // Noting is required; structs are value types
    var adjusted: SimpleStructures {
        simpleDescription = "Adjusted" // Not required
    }
}

var b = SimpleStructures()
b.adjust()
```

```swift
"simpleDescription" = "A very simple class. anotherProperty 01234"
"simpleDescription" = "A very simple class. Now 100% adjusted. anotherProperty 012345"
"simpleDescription" = "A very simple class. Now 100% adjusted."
"simpleDescription" = "A simple structure"
"simpleDescription" = "A simple structure (adjusted)
"simpleDescription" = "A simple structure (adjusted)"
Use a protocol as a type

```swift
// assign the a struct of type SomeClass
let protocolValue: SomeProtocol = a

// another struct description
// A very simple class. Now '100% adjusted.' anotherProperty double
```
6.2 Extensions

Use extensions to add methods or computed properties to a type, or even conformance to a protocol.

```swift
extension Int16 { // extension
    var description: String {
        return "The number " + self
    }
}

let seven = 7
seven = description
seven = adjust() // The number 7
```

41
7. Generics
<T> - type parameters
Generic enum

7.2 You can make generic methods, functions, enum, classes and structs

```cpp
// Implement the built standard library’s optional type
enum Valueutoff { kNone, kSome, kOther }

// optional integer
enum possibleInt { SomeValueOne = 1, SomeValueTwo = 2, None = Valueutoff::kNone, OtherValue = Valueutoff::kOther }

switch (possibleInt) {
    case SomeValueOne:
        // do something
        break;
    case SomeValueTwo:
        // do something else
        break;
    default:
        // do nothing
        break;
}
```
Generic enum

7.3 Using constraints
Use the optional where after the type name to require
- the type to implement a protocol
- two types to be the same
- a class to have a particular superclass

```swift
    for item in T.GranuleElement {
        if matchedHere = false
            for value in T.GranuleElement {
                matchedHere = true
            }
    }
    return matchedHere, items
}
```
var vs let

```javascript
// var - declare a variable that can change
var x = 10;

// let - declare a constant; value that cannot change
let allowableNumberOfTries = 3;
```
Type Annotations

```plaintext
var welcomeMessage: String
welcomeMessage = "Hello y'all"

var red, green, blue: Double
```
Type Aliases

```typescript
// Type Aliases

type AudioSample = 255;
var noisedSample: AudioSample = AudioSample.MAX;
```
Naming and Printing

NOTE
You should really allow inference whenever possible

Naming Constants and Variables
Pretty much any Unicode code point may be used. Except for first character. Not a good idea.

Printing constants and variables
```java
printIn("Today's message is \{welcomeMessage\}")
```

"Today's message is Hello y'all"
Data Types

- Int (int of 8, 16, 32, or 64 bytes)
- Double, Float
- Bool
- String
- Array
- Dictionary
# Tuples

Groups of different or same types

```python
def process_data(data):
    # Process the data
    return processed_data
```
Tuple elements

You can ignore parts of the tuple

```plaintext
let {(justTheStatusCode, _)} = httpResponse

"The status code is \"\httpError, 404\"
```

You can name the tuple's element when defining

```plaintext
let httpResponseStatus = HttpStatusCode 200, description: "OK"

httpResponseStatus.description
```

404

"The status code is 404"

200

"OK"
<table>
<thead>
<tr>
<th>Optionals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use to indicate the possible absence of a value</td>
</tr>
</tbody>
</table>

```swift
let possibleNumber = "123"
let convertedNumber = possibleNumber.toInt11
let nilValue = "456".toInt11
```

Press `%` and click the constant convertedNumber to see its type.
nil

Only optional types can be assigned nil.

```swift
var serverResponseCode: Int? = 404
var serverResponseCode = nil
var defaultsURL: String? = nil
```
Forced unwrapping

```swift
if statements and forced unwrapping

Check for nil with an if

if convertedNumber is Some
    "converted number contains \(convertedNumber)"
else
    "forcibly unwrapping \(convertedNumber)"

converted number contains Optional<123>
forcibly unwrapping b: 123
```
### Optional Binding

*Can be used with an `if` or `while` statement*

```python
if let actualNumber = possibleNumber.toDouble() {
    actualNumber
}
```
Implicitly Unwrapped Optionals

Use when you know the optional always has a value

```swift
let possibleString: String? = "An optional string"
let forcedString: String = possibleString! // An explicit unwrapping
let assumeString: String? = "An implicitly unwrapped optional string"
let implicitString: String = assumeString ?? "An implicitly unwrapped optional string"
```
[3] Basic operators
# Tuples

## Operators

<table>
<thead>
<tr>
<th>Decomposing tuples automatically</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>let (x, y) = (1, 2)</code></td>
</tr>
<tr>
<td><code>&quot;x = \(x\) and y = \(y\)&quot;</code></td>
</tr>
</tbody>
</table>

The assignment does not return a value

// if `x = y` not allowed

Arithmetic operators do not allow overflow &coco do
The remainder operator (%) works with floating points as well.

```python
let intRemainder = 9 % 4
let doubleRemainder = 8 % 2.5

++i, -- operators work with integers and floating-point types
```
comparison

Comparison operators: == (equals), !=, <, <=, =>, <=> (identical), !==

Careful with the ternary operator: include surrounding spaces question ? answer1 : answer2

```javascript
let contentHeight = 40
let hasHeader = true
let rowHeight = contentHeight + (hasHeader ? 50 : 20)
```

<table>
<thead>
<tr>
<th>40</th>
<th>True</th>
<th>90</th>
</tr>
</thead>
</table>

nil coalescing

```swift
nil coalescing operator
a ?? b : unwrap a or return b if a is nil same as a ?? nil ? a : b

let defaultColorName = "red"
var userDefinedColorName: String?
var colorNameToUse = userDefinedColorName ?? defaultColorName

// change the userDefinedColorName to green above to see the effect on colorNameToUse
```

64
range

<table>
<thead>
<tr>
<th>Range Operators</th>
<th>a..b: half-open</th>
<th>a...b: closed-range</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i in 1..5; { print i; }</td>
<td>for i in 1...5; { print i; }</td>
<td></td>
</tr>
</tbody>
</table>
[4] Strings and characters
Strings are value types

- Value types
- Bridged with NSString
- Fast and capable

```
var someString = "A simple string"
let emptyString = ""
let anotherEmptyString = String()
if emptyString.isEmpty { println("it is empty") }

String Mutability: var - yes, let - no

someString = "", is a simple string

'A simple string, is a simple string'
```
Characters

copies are made only when needed. Swift optimization.

```
var myString = someString
myString = "myString is different than someString"
someString

Characters are single letter strings

let exclamationMark: Character = "!"
let catCharacters: [Character] = ["c","w","m"]
let catString = String(catCharacters)
```

"A simple string, is a simple string"
"myString is different than someString"
"A simple string, is a simple string"
String interpolation

Strings can be concatenated, but characters appended

```python
use message = "Hello \n to all"
mmessage.append(exclamationMark)
```

"Hello to all"

String interpolation

use \()\) within a string literal \"\" to interpolate expressions

```python
let tw = 2
print("By a \{tw\} by \{tw\} pine stud\")
```

"By a 2 by 2 pine stud"

Unicode characters may add some humor.

```python
print("\e(\u1F429)\")
```

"😊"
Counting

```plaintext
var word = "cafe"
count(word)  // 'cafe'
4

adding another character, but the count is still the same

word = "cafe345"  // 'cafe'
count(word)  // 'cafe'
4
```
Because of the Unicode representation of strings, strings cannot be indexed by integral values.

```rust
let greeting = "Guten Tag"
println("\count(greeting) characters")
let si = greeting.startIndex
let ei = greeting.endIndex
println("The first character: \(greeting[si.startIndex])")
greeting[si.successor()].unwrap().is_zero
println("The second character: \(greeting[advance(si, 1)])")

Use the global `advance(start: n)` function to access any index.

for index in indices(greeting) {
    println("\(greeting[index])")
}
```

Use `indices(_.1)` to create a `Range` of indices you can navigate through.
Inserting/Removing

```javascript
var welcome = "Hello";

// Insert a character using the insert(charAtIndex) method
welcome.insert(" ", atIndex: welcome.endIndex)

// Insert a string using the splice(charAtIndex) method
welcome.splice(" there!", atIndex: welcome.endIndex, predecessor())

// Remove a character using the removeAtIndex(\_) method
welcome.removeAtIndex(1)

// Remove a substring using the removeRange(\_\_) method
let range = advance(welcome.endIndex, -3)..<welcome.endIndex.
welcome.removeRange(range)
```

Inserting/Removing

<table>
<thead>
<tr>
<th>Comparing Strings</th>
<th>Three ways to compare:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- string and character equality</td>
</tr>
<tr>
<td></td>
<td>- prefix equality</td>
</tr>
<tr>
<td></td>
<td>- suffix equality</td>
</tr>
</tbody>
</table>

- **String and character equality**

```java
if "hi" == "hi" { "same" }

"same"
```

- **Prefix equality: use the hasPrefix(_:_:) method**

```java
if "http://www. slain.edu", hasPrefix("http://") { "HTTP protocol" }

"HTTP protocol"
```

- **Suffix equality: use the hasSuffix(_:_:) method**

```java
if "www.cs.slain.edu", hasSuffix(".edu") { "Educational TLD" }

"Educational TLD"
```
[5] Collection types
Arrays

Three types of collections

- Array - ordered values
- Set - unordered and distinct values
- Dictionary - unordered values

Arrays

```javascript
var somemts = [0n1]
somemts.push(1)
somemts = []
somemts = [0n1] (count: 3, repeatedValue: 3)
somemts = [2,2]
somemts.count
somemts.isEmpty
somemts (8,2) = (4,1)
somemts
somemts.insert(0, atIndex: 2)
somemts.removeAtIndex(0)
somemts
somemts.removeLast()
somemts
```
Iterating arrays

Iterating over an array

```python
for num in someInts { println("\(\{\text{num}\}\)\n\") }
println();
for (index, num) in enumerate(someInts) {
    println("index: \(\{\text{index}\}\) value: \(\{\text{num}\}\)\n")
}
```
Sets

Sets do not have a shorthand form

```javascript
var vowels = Set<Character>\{ "a", "e", "i", "o" \}
vowels.insert('u') // not added
vowels.insert('u') // still a set, not an array

var favoriteBands = Set\{ "Dire Straits", "Bee Gees", "Maroon Five" \}
favoriteBands.isEmpty()
favoriteBands.contains("Aerosmith")
favoriteBands.insert("Aerosmith")
```

77
Iterating sets

```python
print()
for (index, band) in enumerate(favoriteBands):
    print(f"{index}: \{band\}\n")

print()
for (index, band) in enumerate(sorted(favoriteBands)):
    print(f"{index}: \{band\}\n")

For operations on sets see help (intersections, unions, ...)

A type must be hashable in order to be stored in a set
  * It must conform to both the Equatable (==) and Hashable protocols
```
Dictionaries

```kotlin
// Dictionaries
The key must be Hashable.

var intMapping = [Int: String]()
intMapping = []
intMapping = [0: "zero", 1: "one"]
intMapping.count
intMapping.isEmpty
intMapping[3] = "three"

// Trying to update key = 2 returns nil, since it does not exist as of yet
intMapping.updateValue("two", forKey: 2)
intMapping[2] = nil
intMapping
```

79
Dictionaries

```ruby
# Iterating over a dictionary
print("!!")
for key, stringMapping in intMappings:
    print("!\"", key, ",", stringMapping,"!\"")

let keys = [Int(intMappings.keys)]
let mappings = [String(intMappings.values)]
```
[6] Control flow
For

```
for i in 1...5 { println(i) }
for i in stride(from: 5, through: 1, by: -1) { println(i) }
var p = 2
for _ in 1...3 { p += p }
for var i = 1; i <= 5; i++ { println(i) }
```
```c
while and do-while

// Nothing more to show for these guys. :-)

if

// Again, nothing to add for this one. :-)
switch - Range

```javascript
let val = 1_000
switch val {
  case 0...9: "ones"
  case 10...99: "tens"
  case 100...999: "hundreds"
  case 1000...9999: "thousands"
  default: "more than I can count"
}
```
name the tuple values for easy identification

```javascript
let pt = {x: 1, y: 1}

switch pt {
    case {0, 0}: "origin"
    case (1, 0): "x-axis"
    case (0, 1): "y-axis"
    case (-2, -2, -2): "((-x.x), (-pt.y)) inside the box"
    default: "outside the box"
}
```

"(1,1) inside the box"
switch - binding

```rust
define pt2 = {1, 0}
define pt3 = {0, 1}
define pt4 = {1, 1}
define pt5 = {0, 0}

switch pt2 {
  case (let x, 0): "on x-axis"
  case (0, let y): "on y-axis"
  case (let x, y): "nt (\(x), \(y))"}
```
switch - where

```javascript
let pt3 = (1, -1)
switch pt3 {
  case let (x, y) where x == y: "on the line x = y"
  case let (x, y) where x == -y: "on the line x = -y"
  case let (x, y): "somewhere else"
}
```
[7] Functions
# Internal Names

## Functions

Each function has a type consisting of:
- parameter types
- return type

### Function with internal name only

```swift
func sayHello(personName: String) -> String {
    return "Hello, \(personName)"
}
sayHello("Baba")
```

### Function with two parameters

```swift
func halfOpenRangeLength(start: Int, end: Int) -> Int {
    return end - start
}
halfOpenRangeLength(1, 10)
```
function with no parameters

```swift
func sayHelloWorld() -> String {
    return "Hello, world"
}
sayHelloWorld()
```

"Hello, world"
return tuples
optional return tuple

```javascript
function minMax(array: number[]): [number, number] {
    if (array.length > 0) {
        var min = array[0];
        var max = array[0];
        for (var element of array) {
            if (element < min) min = element;
            if (element > max) max = element;
        }
        return [min, max];
    }
}

if (let bounds = minMax([8, -6, 2, 100, 3, 71])
    bounds.min
    bounds.max
} 109
```
### Function parameter names

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td><code>// function with both internal and external names</code></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td><code>func add2Ints(first num1: Int, second num2: Int) -&gt; Int {</code></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td><code>    return num1 + num2</code></td>
<td>(3 times)</td>
</tr>
<tr>
<td>57</td>
<td><code>  }</code></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td><code>add2Ints(first: 2, second: 4)</code></td>
<td>6</td>
</tr>
<tr>
<td>59</td>
<td><code>// use a # to make both external and internal names the same</code></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td><code>func sub2Ints(#first: Int, #second: Int) -&gt; Int {</code></td>
<td>(3 times)</td>
</tr>
<tr>
<td>61</td>
<td><code>    return first - second</code></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td><code>  }</code></td>
<td></td>
</tr>
<tr>
<td>63</td>
<td><code>sub2Ints(first: 2, second: 4)</code></td>
<td>-2</td>
</tr>
</tbody>
</table>
parameter names

```java
// function with default parameter values

function join(separator: String?, separatorString: String?, delimiter: String?, delimiters: String?) -> String {
    return string + delimiter + toString
}

join(string: "Hello", to: ":")
join(string: "Hello", to: "world", with: ":")

// Constant and variable parameters

// parameters are constant by default - cannot be changed within the function

func append(separator: String?, separatorString: String?, delimiters: String?, number: Int) -> String {
    return toHexString
}

append("ha", to: ":", number: 3)
```

94
in-out parameters

```swift
func swapTwoInts(inout a: Int, inout b: Int) {
    let tmpA = a
    a = tmpA
    b = tmpA
}

var a = 1, b = 2
swapTwoInts(&a, &b)
println(a)  // Output: 2
println(b)  // Output: 1
```

95
function as a type

```haskell
var mathFunctions :: Int, Int -> Int = add2Ints
    mathFunctions(2, 4)  
    mathFunction = sub2Ints
    mathFunctions(2, 4)  
    func printMathResult(mathFunction (Int, Int) -> Int, m1, Int, m2: Int)  
        printm("m1 = \$,mathFunction(m2)\$")  
    printMathResult add2Ints, 2, 4) // show the assiant editor to see results  
    printMathResult sub2Ints, 2, 4)  
```
<table>
<thead>
<tr>
<th>Line Numbers</th>
<th>Code</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>107</td>
<td>func stepForward(value: Int) -&gt; Int {</td>
<td>4 times</td>
</tr>
<tr>
<td>108</td>
<td>return value + 1</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>func stepBackward(value: Int) -&gt; Int {</td>
<td>3 times</td>
</tr>
<tr>
<td>112</td>
<td>return value - 1</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>func chooseStepFunction(backwards: Bool) -&gt; (Int) -&gt; Int {</td>
<td>2 times</td>
</tr>
<tr>
<td>116</td>
<td>return backwards ? stepBackward : stepForward</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>var currentValue = 3</td>
<td>3 times</td>
</tr>
<tr>
<td>123</td>
<td>var moveMearerToZero = chooseStepFunction(currentValue &gt; 0)</td>
<td>Function</td>
</tr>
<tr>
<td>125</td>
<td>// The constant now refers to the stepBackward function</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>print(current)</td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>while (currentValue &gt; 0) {</td>
<td>3 times</td>
</tr>
<tr>
<td>131</td>
<td>print(&quot;(currentValue =&quot;)&quot;)</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>currentValue = moveMearerToZero(currentValue)</td>
<td>3 times</td>
</tr>
<tr>
<td>135</td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

function types as return types
nested return functions

```haskell
// New function to use and return a nested function

func chooseStepFunction2_backwards: Bool => Int => Int {
  func stepForward(input: Int) => Int { return input + 1 }
  func stepBackward(input: Int) => Int { return input - 1 }
  return backwards 7 stepForward stepBackward
}

4

moveWearerToZero = chooseStepFunction2(currentValue * 8)  // Function

while currentValue != 0 {
  printStr("currentValue")  // 4 times
  currentValue = moveWearerToZero(currentValue)  // 4 times
}
[8] Closures
# Closures

Closures are functions that can access and modify the variables and state from the enclosing lexical scope.

## Forms of Closures

Closures take one of three forms:

- **Global functions**: Named and do not capture any values.
- **Nested functions**: Named and can capture values from the enclosing function.
- **Closure expressions**: Unnamed and can capture values from the surrounding context.

## Optimizations

Closures offer optimizations such as:

- Inferring parameter and return value types from context.
- Implicit returns from single-expression closures.
- Shorthand argument names.
- Trailing closure syntax.
parameter function

```
First version - parameter function

func backwards(s1: String, s2: String) -> Bool {
  return s1 > s2
}

var reversed = sorted(names, backwards)

[Ewa, Daniela, Chris, Barry, Alex]
```
inline closure

Closure expression syntax has the following form:

```haskell
{{ parameters } \to \text{return type in statements}}
```

Second version - inline closure

```haskell
reversed = sortDinames, (s1: String, s2: String) \to \text{Bool in return s1 > s2})
```

102
contextual inference
implicit returns

Fourth version - implicit returns

20 reversed -> sorted(names, { s1, s2 in s1 > s2 })
21 reversed

(10 times)

'Eva', 'Daniela', 'Chris', 'Barry', 'Alex'
shorthand argument names

Fifth version - shorthand argument names

23 reversed = sorted(names, { x @> $y } )
24 reversed
25 (10 times)
26 {'Eva', 'Daniela', 'Chris', 'Banny', 'Alex'}
Strings implement the `>` operator, thus:

```python
reversed = sorted(names, >)
['Ewa', 'Daniella', 'Chris', 'Barry', 'Alex']
```
trailing closure

Sixth version – trailing closure

```plaintext
reversed = sorted(names) { $0 > $1 }
reversed[10 times]
```

| Eire | Eamonn | Chris | Barry | Alex |
capturing values

```
func interestCalculator(rate: Double) -> (Double, Int) -> Double {
  // captures the rate
  let calculator = {
    amount: Double, years: Int -> Double in
    return amount * rate * Double(years)
  }
  return calculator
}

var interest25Percent = interestCalculator(0.05)
interest25Percent(100, 10)
```
capturing values

```swift
func interestCalculator2(rate: Double) -> (Double, Int) -> Double {
    // return function directly. Use inference to simplify
    return 4
}

interestAt5Percent = interestCalculator2(1.05)
interestAt5Percent(100, 2)
```
capturing values

```swift
func interestCalculator3(rate: Double) -> (Double, Int) -> Double {
    // use shortcut parameter names
    return { 10 * rate + Double(1) } // (2 times)
}

let interestAt5Percent = interestCalculator3(0.05)
let interestAt5Percent100 = interestCalculator3(100, 2)
```

110
[9] Enumerations
enum

Enumerations
Very much like classes.

```java
enum CompassPoint {
    case North
    case South
    case East
    case West

    // alternatively
    // case North, South, East, West
}
```
associated values

```java
enum BarCode {
    case UPCA(Int, Int, Int)
    case QRCode(String)
}

var productBarCode = BarCode.UPCA(8, 85989, 51226, 3) // (Enum Value)
var productBarCode = QRCode("ABCDEFGHIJKLMNOPQRSTUVWXYZ") // (Enum Value)

switch productBarCode {
    case .UPCA(let numberSystem,
      let manufacturer,
      let product,
      let check):
        "UPC-A, manufacturer = " + manufacturer"
    case .QRCode(let numberSystem, nfr, prod, chk):
      "QR code: " + productCode"
    default:
      "QR code: ABCDEFGHJKL..." // QR code: ABCDEFGHJKL...
```
raw values

```swift
enum PlanetOrderFromSun: Int {
    case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}

let earthsOrder = PlanetOrderFromSun.Earth.rawValue

let unknownPlanet = PlanetOrderFromSun(rawValue: 9)
if let up = unknownPlanet { "ok" } else { "oops" }
```

114
[10] Classes and structures
class v struct

Classes and Structures

Both can:

- Define stored properties
- Define methods
- Define subscripts
- Define initializer
- Be extended (extensions)
- Conform to protocols
class extras

Classes in addition can:

- Be subclassed
- Be typecast
- Use deinitializers to free resources
- Have more than one reference to their instances
Structures have a default member-wise initializer

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

let origin = Point(x: 0, y: 0)
```
class identity

```java
class Size {
    var width = 0
    var height = 0
}

let s1 = Size()
let s2 = Size()

// s1 == s2 ? "Same" : "Different" // Will overload later!

s1 == s2 ? "Identical" : "Un-identical"
```
## Properties

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stored properties:</td>
</tr>
<tr>
<td>• Provided by classes and structs</td>
</tr>
<tr>
<td>• can be observed</td>
</tr>
<tr>
<td>Computed properties:</td>
</tr>
<tr>
<td>• Provided by classes, structs, enums</td>
</tr>
<tr>
<td>Properties can:</td>
</tr>
<tr>
<td>• be associated with instances</td>
</tr>
<tr>
<td>• be associated with the type (type properties)</td>
</tr>
<tr>
<td>• inherited properties (stored/computed) can be observed</td>
</tr>
</tbody>
</table>
Lazy stored properties

```swift
class DBAccess {
    // lazy var db = someLengthyTask()

    func performLengthyTask() {
        // db.someMethod()
    }
}
```
Computed properties and stored observers

```swift
class Circle {
    var radius = 1.0 {
        willSet {
            \(\text{radius} \text{ will change to } \text{newValue}\)"
        }
        didSet {
            "\text{new radius: } \text{\(\text{radius}\)}"
        }
    }

    var area: Double {
        get {
            return M_PI * radius * radius
        }
        set {
            radius = sqrt(newValue / M_PI)
        }
    }

    var circumference: Double {
        get {
            return 2 * M_PI * radius
        }
        set(newCircumference) {
            radius = newCircumference / 2 / M_PI
        }
    }
}

let circle = Circle()
circle.radius = 2
```
Type properties

- `struct` and `enum` can define both `let/var` stored/computed properties
- `class` can define only `var` computed properties

```swift
struct SomeStruct {
    static var storedTypeProperty = "some value"
    static var computedTypeProperty: Int { return 1 }
}

class SomeClass {
    static var storedTypeProperty = "some value"
    static var computedTypeProperty: Int { return 1 }
    class var overrideableComputedTypeProperty: Int { return 1 }
}

SomeClass.storedTypeProperty

"some value"
```
[12] Methods
class Counter {
    var count = 0

    func incrementBy(amount: Int, numberOfTimes: Int) {
        // amount: internal
        // numberOfTimes: internal/external
        count += amount * numberOfTimes
    }

    func decrementBy(amount: Int, _ numberOfTimes: Int) {
        // amount: now internal/external
        // numberOfTimes: internal only
        count -= amount * numberOfTimes
    }
}

let c = Counter()
c.incrementBy(3, numberOfTimes: 3)  // (count 3)
c.decrementBy(amount: 2, 1)  // (count 1)
Mutating value types

```swift
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveByX(dx: Double, dy: Double) {
        x += dx
        y += dy
        // alternatively
        // self = Point(x: x + dx, y: y + dy)
    }
}
var pt = Point()
pt.moveByX(x: 1, y: 1)
```

Mutating enum

```swift
enum TriStateSwitch {
    case Off, Low, High

    mutating func next() {
        switch self {
        case Off: self = Low
        case Low: self = High
        case High: self = Off
        }
    }
}

var ovenLight = TriStateSwitch.Off
ovenLight == .Off ? "Off" : "?
ovenLight.next()
ovenLight == .Low ? "Low" : "?"
```
Mutating enum

```
class Alien {
  static var alienCount = 0

  // subclasses cannot override
  static func destroyAnAlien() {
    // self refers to type in this context
    self.aliencount -= 1
  }

  // subclasses can override
  class func generateAnAlien() {
    // implicit access to type property
    alienCount += 1
  }

  Alien.aliencount = 0
  Alien.generateAnAlien(); Alien.aliencount = 1
  Alien.destroyAnAlien(); Alien.aliencount = 0
```
[13] Subscripts
Subscripts

Subscripts
Allow access to members of a collection, list or sequence
- can define multiple ones (overloading) for a single type
- not limited to one dimension
- can have multiple input parameters
- use [] to access elements

```swift
struct TimesTable {
    let multiplier: Int

    subscript(index: Int) -> Int {
        // used as [index]
        return multiplier * index
    }
}

let threeTimesTable = TimesTable(multiplier: 3)

"3 * 6 = \(threeTimesTable[6])" // multiplier 3
"3 * 6 = 18"
```
options

Subscript options

- can return any type
- can use var parameters
- can use variadic parameters (…)
- cannot use in-out parameters

```swift
struct Matrix {
    let rows: Int
    let columns: Int
    var grid: [Double]

    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns

        // Create the grid, initialize with 0.0
        grid = Array(count: rows * columns, repeatedValue: 0.0)
    }

    func indexIsValidForRow(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows &&
        column >= 0 && column < columns
    }
}
```
func isValidForRow(row: Int, column: Int) -> Bool {
    return row >= 0 && row < rows && column >= 0 && column < columns
}

var matrix2By2 = Matrix(rows: 2, columns: 2)
matrix2By2[0, 0] = 0
// The next line will execute the assertion.
// Reveal the AE to see results
matrix2By2[2, 2] = 1.5
matrix2By2[1, 1] = 1.5
[14] Inheritance
Inheritance

Classes do not inherit a universal class by default.

```swift
class Vehicle {
    var currentSpeed = 0.0
    var description: String {
        return "travelling at \(currentSpeed) mph."
    }
    func makeNoise() { /* do nothing */ }
}

class Bicycle: Vehicle {
    var hasBasket = false
}

class Tandem: Bicycle {
    var currentNumberOfPassengers = 0
}

let tandem = Tandem()
let tandem, currentSpeed = 20.0
let tandem, description = "travelling at 20.0 mph."
```
Overriding - method

A subclass can override a superclass's

- instance/type methods
- instance/type properties (add, not remove accessibility)

instance method

```swift
class Train: Vehicle {}
override fun makeNoise() {
    println("Choo choo")
}
```

let train = Train()
train.makeNoise()
Override - property

```java
instance property

35 class Car: Vehicle {
36     var gear = 1
37     override var description: String {
38         return super.description + " in gear \(gear\)"
39     }
40 }
41
42 let car = Car()
43 car.currentSpeed = 25.0; car.gear = 3
44 car.description
```
Override - observer

```swift
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            // changes gear based on speed
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}

let automaticCar = AutomaticCar()
automaticCar.currentSpeed = 35.0
```
<table>
<thead>
<tr>
<th>Preventing overrides</th>
</tr>
</thead>
<tbody>
<tr>
<td>mark as final before its introducer (var, func, class, subscript)</td>
</tr>
</tbody>
</table>
[15] Initialization
Initialization

struct Celcius {
    var tempInCelcius: Double

    init(fromFahrenheit: Double) {
        tempInCelcius = (fromFahrenheit - 32.0) / 1.8
    }

    init(fromKelvin: Double) {
        tempInCelcius = fromKelvin - 273.15
    }

    init(_ celcius: Double) {
        tempInCelcius = celcius
    }
}

let boilingPointOfWaterInDegreesCelcius = Celcius(fromFahrenheit: 212.0)
let freezingPointOfWaterInDegreesCelcius = Celcius(fromKelvin: 273.15)
let bodyTemperature = Celcius(37.0)
Optional properties

```
Optional property types

```class SurveyQuestion {
    var question: String
    var response: String?

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

    func ask() { println(question) }
}

let cheeseQuestion = SurveyQuestion(question: "Do you like cheese?")
cheeseQuestion.ask()  // "Do you like cheese?"
cheeseQuestion.response = "Yes I do!"
```
Default init

Default initializers

Swift provides a default initializer for a

- struct/class having default values for all properties
- and that doesn't add its own initializer

```swift
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}

var item = ShoppingListItem()

struct Size {
    var width: Double
    var height: Double
}

let size = Size(width: 3, height: 2)
```
init delegation - Value

```swift
struct Point {
    var x = 0.0
    var y = 0.0

    init(x: Double, y: Double) { self.x = x; self.y = y }
}

var point = Point()
point = Point(x: 1, y: 1)
```
init delegation - class

<table>
<thead>
<tr>
<th>_initializer delegation - class types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rule 1</strong> - A designated init must call superclass's designated init</td>
</tr>
<tr>
<td><strong>Rule 2</strong> - A convenience init must call another init</td>
</tr>
<tr>
<td><strong>Rule 3</strong> - A convenience init must ultimately call a designated init</td>
</tr>
</tbody>
</table>
class Employee {
  var eid: Int

  // convenience init()
  init() { self.init(eid: 0) }

  // designated init
  init(eid: Int) { self.eid = eid }
}

class Manager: Employee {
  var departmentHeaded: String

  // Override super's designated init
  override convenience init(eid: Int) {
    self.init(departmentHeaded: "none", eid: eid)
  }

  init(departmentHeaded: String, eid: Int) {
    self.departmentHeaded = departmentHeaded
    super.init(eid: eid)
  }
}

let defEmployee = Employee()  // [eid 0]
let emp1 = Employee(eid: 1)    // [eid 1]

// calls inherited init() which calls overridden init(eid):
let defManager = Manager()     // [eid 0] departmentHeaded "none"

let mgr1 = Manager(eid: 1)     // [eid 1] departmentHeaded "none"
let mgr2 = Manager(departmentHeaded: "IT", eid: 2)  // [eid 2] departmentHeaded "IT"
```java
init override

<table>
<thead>
<tr>
<th>Init inheritance and overriding</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Vehicle {</td>
</tr>
<tr>
<td>var numberOfWheels = 0</td>
</tr>
<tr>
<td>var description: String { return &quot;(\text{numberOfWheels}) wheels&quot; }</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>147</td>
</tr>
</tbody>
</table>
```
[16] Deinitialization
Deinitializers

- Use to clean things up. One per class
- Always called
- Superclass deinit inherited and called automatically

```
class Bank {
    var cash = 100_000

    deinit {
        println("No money left")
    }
}
```

```
class Customer {
    let name: String

    init(name: String) {
        self.name = name
        println("\(name): init")
    }

    deinit {
        println("\(name): deinit")
    }

    var ref1: Customer? = nil
    var ref2: Customer? = nil
    var ref3: Customer? = nil

    // class init
    ref1 = Customer(name: "John Appleseed")
    ref2 = ref1
    ref3 = ref1

    ref1 = nil
    ref2 = nil
    ref3 = nil

    // calls deinit
    ref3 = nil
Avoid strong refence cycles b/w classes

```java
class Person {
    let name: String
    init(name: String) { self.name = name }

    var apartment: Apartment?
    deinit { println("(name): deinit") }
}

class Apartment {
    let number: Int
    init(number: Int) { self.number = number }

    // uncomment to see the cycle broken
    // weak var tenant: Person?

    deinit { println("(number): deinit") }
}

var john: Person? = Person(name: "John Appleseed")
var number73: Apartment? = Apartment(number: 73)

john!.apartment = number73
number73!.tenant = john

// Notice how the deinit is never called (strong ref)
john = nil
number73 = nil
```

[152]
[18] Optional chaining
Optional chaining

```java
// Optional Chaining
Alternative to forced unwrapping

class Person {
    var residence: Residence?
}

class Residence {
    var numberOfRooms = 1
}

let john = Person()

let roomCount = john.residence?.numberOfRooms

if let roomCount = john.residence?.numberOfRooms {
    roomCount
} else {
    "oops"
}
```

[19] Type casting
Class type casting

Type Casting
Implemented with the `is` and `as` operators

```java
class MediaItem {
    var name: String
    init(name: String) { self.name = name }
}

class Movie: MediaItem {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```
Downcast

```javascript
let library = [
    Movie(name: "Casablanca", director: "Michael Curtiz"),
    Song(name: "Blue Suede Shoes", artist: "Elvis Presley")
];

// The array is of MediaItem, so only superclass functionality
// is available
Library[0] is Movie
// This errors out
//library[0].director

Downcast to subclass to access specific functionality

for item in library {
    if let movie = item as? Movie {
        movie
    } else if let song = item as? Song {
        song
    }
}
```
[20] Nested types
## Nested types

Can nest classes, structs, enums

```swift
struct BlackjackCard {
    // nested Suit enumeration
    enum Suit: Character {
        case Spades = "♠", Hearts = "♥", Diamonds = "♦", Clubs = "♣"
    }

    // nested Rank enumeration
    enum Rank: Int {
        case Two = 2, Three, Four, Five, Six, Seven, Eight, Nine, Ten
        case Jack, Queen, King, Ace
        struct Values {
            let first: Int, second: Int?
        }
    }

    var values: Values {
        switch self {
        case .Ace:
            return Values(first: 1, second: nil)
        case .Jack, .Queen, .King:
            return Values(first: 10, second: nil)
        default
            return Values(first: self.rawValue, second: nil)
    }
}
```
Nested types

```swift
// BlackjackCard properties and methods
let rank: Rank
let suit: Suit

var description: String {
    var output = "suit is \(suit.rawValue),"
    output += " value is \(rank.values.first)"
    if let second = rank.values.second {
        output += " or \(second)"
    }
    return output
}

let theAceOfSpades = BlackjackCard(rank: .Ace, suit: .Spades)
theAceOfSpades.description
```

"suit is ♦, value is 1"
"suit is ♦, value is 1 or 11"
"suit is ♦, value is 1 or 11"
"suit is ♦, value is 1 or 11"
[21] Extensions
[22] Protocols
[23] Generics
[24] Access control
Access control

Restrict access to your code entities from other files and modules.

- types (class, struct, enum)
- properties
- methods
- init
- subscript
- protocols (to an extend)
- global let, var and functions

- **Module** - a single unit of code distribution
- **File** - a single source code file w/in a module
Access levels

There are three access levels:

- **Public** - access from any file w/in any module
  
  - use for the public interface to a framework

- **Internal** - access from any file w/in defining module (default)
  
  - use for app’s or framework’s internal structure

- **Private** - access from w/in file only
  
  - use to hide functionality
Access control

Guiding principle

Make sure that any entity is not restricted by any of its dependencies

```
public class SomePublicClass {}
internal class SomeInternalClass { // or class SomeInternalClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0 // or let someInternalConstant = 0
private func somePrivateFunction() {}
```
Default member access

Accessibility of type becomes the default accessibility for its members
- Public and Internal type access - Internal member access
- Private type access - Private member access

```java
public class PublicClass {
    // explicitly public
    public var publicProperty = 0

    var internalProperty = 0  // implicitly public
}

class InternalClass {
    // implicitly internal
    class InternalClass {
        var internalProperty = 0  // implicitly internal
    }
}
```
# Default member access

<table>
<thead>
<tr>
<th>Tuple access:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• tuple access is determined by most restrictive member access</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function access:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• overall access is determined by most restrictive access b/w params, return</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enum access:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• cases have same access as enum</td>
<td></td>
</tr>
<tr>
<td>• raw</td>
<td>associated values can have same or higher access</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nested types:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• have Internal access for Public or Internal types, Private otherwise</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subclass types:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• access is the same or lower than superclass (can restrict, but not open up)</td>
<td></td>
</tr>
</tbody>
</table>
[25] Advanced operators
infix binary

```
Operator Functions

struct Vector2D {
  var x = 0.0
  var y = 0.0
}

an infix binary operator

func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
} (4 times)

let vector = Vector2D(x: 3.0, y: 1.0) (x 3, y 1)
let anotherVector = Vector2D(x: 2.0, y: 4.0) (x 2, y 4)
let combinedVector = vector + anotherVector (x 5, y 5)
```
prefix unary

a prefix unary operator

```python
15  prefix func (vector: Vector2D) -> Vector2D {
16      return Vector2D(x: -vector.x, y: -vector.y)  \[x=3, y=4\]
17  }
18  let positive = Vector2D(x: 3.0, y: 4.0)  \[x=3, y=4\]
19  let negative = -positive  \[x=-3, y=-4\]
```
compound assignment

```swift
func += { inout left: Vector2D, right: Vector2D } {
    left = left + right
}(3 times)

var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
```
prefix increment

```python
prefix func ++ (inout vector: Vector2D) -> Vector2D {
    vector += Vector2D(x: 1.0, y: 1.0)
    return vector
}

var toIncrement = Vector2D(x: 3.0, y: 4.0)
let afterIncrement = ++toIncrement
```

174
equivalence

equivalence operators

```swift
func ==(left: Vector2D, right: Vector2D) -> Bool {
    return (left.x == right.x) && (left.y == right.y)
}
func != (left: Vector2D, right: Vector2D) -> Bool {
    return !(left == right)
}
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree { "equal" }
```
Custom operators

// First declare the operator

prefix operator +++ {}

// Next, define it

prefix func +++ (inout vector: Vector2D) -> Vector2D {
    var vector = vector
    return vector
}

var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++ toBeDoubled

// Can specify associativity and precedence
infix operator <- { associativity left precedence 140 }

func <- (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y - right.y)
}

let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondeVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector <- secondeVector