

Swift Whirlwind Trip

Yanqiao ZHU

<https://sxkdz.org>

Apple Workshop 2017

School of Software Engineering, Tongji University

Based upon *The Swift Programming Language*.

Outline

- Values
 - Variables and Constants
 - Compound Data Types
 - Partial Primitive Data Types
- Control Flow
 - Conditionals and Optionals
 - Switches
 - Loops
 - Ranges
- Functions and Closures
- Higher Order Functions
 - Map
 - Filter
 - Reduce

Outline (cont.)

- Objects and Classes
- Enumerations and Structures
- Protocols and Extensions
- Error Handling
- Generics

Introduction to Swift

- Swift is a safe, fast, and interactive programming language that combines the best in modern language thinking with wisdom from the wider Apple engineering culture and the diverse contributions from its open-source community.
- The compiler is optimized for performance and the language is optimized for development, without compromising on either.
- Swift is friendly to new programmers. It's an industrial-quality programming language that's as expressive and enjoyable as a scripting language. Writing Swift code in a playground lets you experiment with code and see the results immediately, without the overhead of building and running an app.

Programming-Language Level Features

- Variables are always initialized before use.
- Array indices are checked for out-of-bounds errors.
- Integers are checked for overflow.
- Optionals ensure that nil values are handled explicitly.
- Memory is managed automatically.
- Error handling allows controlled recovery from unexpected failures.

Version Compatibility of Swift 4

- The following features are available only to Swift 4 code:
 - Substring operations return an instance of the `Substring` type, instead of `String`.
 - The `@objc` attribute is implicitly added in fewer places.
 - Extensions to a type in the same file can access that type's private members.

Hello World!

- Don't need to import a separate library for functionality like input/output or string handling.
- Code written at global scope is used as the entry point for the program, so you don't need a `main()` function.
- You also don't need to write semicolons at the end of every statement.

```
1.print "Hello, world!"
```

Values

Swift Whirlwind Trip

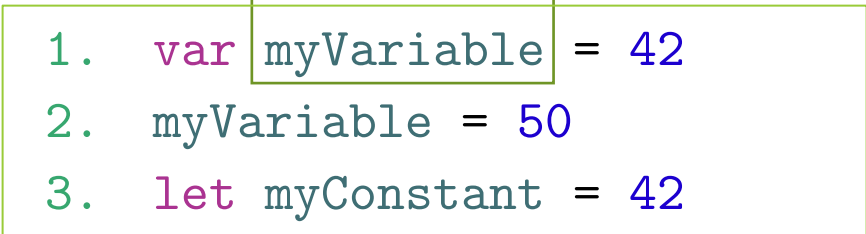
Variables and Constants

- Use `let` to make a constant and `var` to make a variable.
- The value of a constant **doesn't need to be known at compile time**.
 - But must assign it a value exactly once.
 - Can use constants to name a value that you determine once but use in many places.

Variables and Constants (cont.)

- A constant or variable must have the same type as the value you want to assign to it.
 - Don't always have to write the type explicitly.
 - Providing a value when you create a constant or variable lets the compiler infer its type.

integer deduced by the compiler



```
1. var myVariable = 42
2. myVariable = 50
3. let myConstant = 42
```

Variables and Constants (cont.)

- If the initial value doesn't provide enough information (or if there is no initial value), specify the type by writing it after the variable, separated by a colon.

```
1.let implicitInteger = 70  
2.let implicitDouble = 70.0  
3.let explicitDouble: Double = 70
```

Variables and Constants (cont.)

- Values are never implicitly converted to another type.
- If you need to convert a value to a different type, explicitly make an instance of the desired type.

```
1.let label = "The width is "  
2.let width = 94  
3.let widthLabel = label + String(width)
```

Variables and Constants (cont.)

- There's an even simpler way to include values in strings: Write the value in parentheses, and write a backslash (\) before the parentheses.

```
1.let apples = 3
2.let oranges = 5
3.let appleSummary = "I have \ (apples) apples."
4.let fruitSummary = "I have \ (apples + oranges) pieces of fruit."
```

Long Strings

- Use three double quotes (""") for strings that take up multiple lines. Indentation at the start of each quoted line is removed, as long as it matches the indentation of the closing quote.

```
1.let quotation = """
2.Even though there's whitespace to the left,
3.the actual lines aren't indented.
4.Except for this line.
5.Double quotes (") can appear without being escaped.
6.I still have \ (apples + oranges) pieces of fruit.
7. """
```

Compound Data Types

- Swift provides three primary *compound data types* (aka., *collection types*), known as arrays, sets, and dictionaries, for storing collections of values.
 - Arrays are ordered collections of values.
 - Sets are unordered collections of unique values.
 - Dictionaries are unordered collections of key-value associations.

Compound Data Types (cont.)

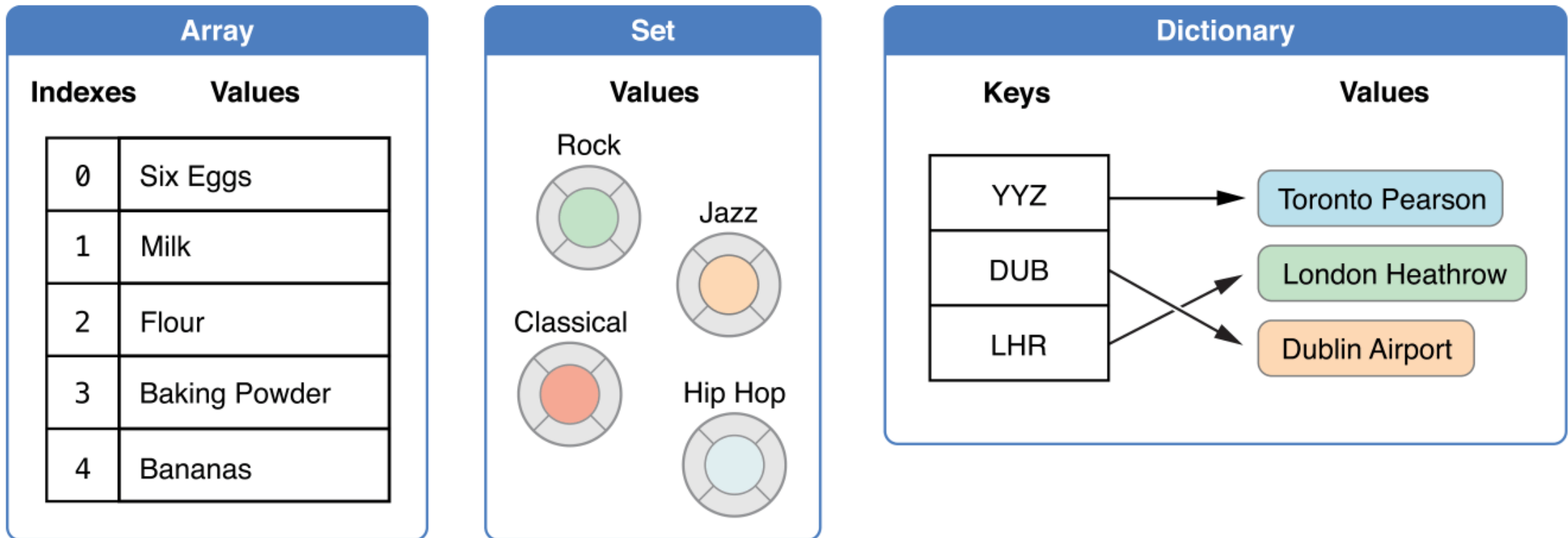


Figure 1: Compound Data Types in Swift.

Compound Data Types (cont.)

- Create arrays and dictionaries using brackets ([]).
- Access elements by writing the index or key in brackets.
 - A comma is allowed after the last element.

```
1.var shoppingList = ["catfish", "water", "tulips", "blue paint"]
2.shoppingList[1] = "bottle of water"
3.var occupations = [
4.    "Malcolm": "Captain",
5.    "Kaylee": "Mechanic",
6.]
7.occupations["Jayne"] = "Public Relations"
```

Compound Data Types (cont.)

- To create an empty array or dictionary, use the *initializer* syntax.

```
1.let emptyArray = [String]()  
2.let emptyDictionary = [String: Float]()
```

Compound Data Types (cont.)

- If type information can be inferred, you can write an empty array as `[]` and an empty dictionary as `[:]`.
 - For example: when you set a new value for a variable or pass an argument to a function.

```
1.shoppingList = []  
2.occupations = [:]
```

Partial Primitive Data Types

- Certain standard library types like `Int` and `Float` are special in that they map to basic CPU operations. (And in Swift, the compiler doesn't offer any other means to directly access those operations.)

Partial Primitive Data Types (cont.)

- Data types are implemented internally as struct and conforms with the protocols (check [the source code](#) where Bool is implemented).
 - BitwiseOperationsType
 - CVarArgType
 - Comparable
 - CustomStringConvertible
 - Equatable
 - Hashable
 - MirrorPathType
 - RandomAccessIndexType
 - SignedIntegerType
 - SignedNumberType

Partial Primitive Data Types (cont.)

- No difference **at the language level** between the things one thinks of as “primitives” in other languages.
 - Behave just like primitives.
 - Passed by value.

Control Flow

Swift Whirlwind Trip

Control Flow Statements

- Use `if` and `switch` to make conditionals.
- Use `for-in`, `for`, `while`, and `repeat-while` to make loops.
 - Parentheses around the condition or loop variable are **optional**.
 - Braces around the body are **required**.

Control Flow Statements (cont.)

```
1.let individualScores = [75, 43, 103, 87, 12]
2.var teamScore = 0
3.for score in individualScores {
4.    if score > 50 {
5.        teamScore += 3
6.    } else {
7.        teamScore += 1
8.    }
9.}
10.print(teamScore)
```

Conditionals

- The conditional must be a *Boolean expression*.
 - Means that code such as `if score { ... }` is an error.
 - The conditional is NOT an **implicit** comparison to zero.

Optionals

- If and let can be used together to work with values that might be missing.
 - These values are represented as *optionals*.
- An optional value either contains a value or contains `nil` to indicate that a value is missing.
- Write a question mark (?) after the type of a value to mark the value as optional.

Optionals (cont.)

```
1.var optionalString: String? = "Hello"
2.print(optionalString == nil)
3.var optionalName: String? = "John Appleseed"
4.var greeting = "Hello!"
5.if let name = optionalName {
6.    greeting = "Hello, \(name)"
7.}
```

Optionals (cont.)

- If the optional value is `nil`, the conditional is `false` and the code in braces is skipped.
- Otherwise, the optional value is unwrapped and assigned to the constant after `let`, which makes the unwrapped value available inside the block of code.

Optionals (cont.)

- Another way to handle optional values is to provide a default value using the ?? operator.
- If the optional value is missing, the default value is used instead.

```
1.let nickName: String? = nil
2.let fullName: String = "John Appleseed"
3.let informalGreeting = "Hi \ \(nickName ?? fullName)"
```

Switches

- Switches support **any kind** of data and a wide variety of comparison operations—they aren't limited to integers and tests for equality.

```
1. let vegetable = "red pepper"
2. switch vegetable {
3. case "celery":
4.     print("Add some raisins and make ants on a log.")
5. case "cucumber", "watercress":
6.     print("That would make a good tea sandwich.")
7. case let x where x.hasSuffix("pepper"):
8.     print("Is it a spicy \(x)?")
9. default:
10.    print("Everything tastes good in soup.")
11. }
```

Notice how `let` can be used in a *pattern* to assign the value that matched the pattern to a constant.

Switches (cont.)

- After executing the code inside the switch case that matched, the program **exits** from the switch statement.
- Execution doesn't continue to the next case.
 - There is no need to **explicitly** break out of the switch at the end of each case's code.

Loops

- Use `for-in` to iterate over items in a dictionary by providing a pair of names to use for each *key-value* pair.
- Dictionaries are an **unordered** collection, so their keys and values are iterated over in **an arbitrary order**.

Loops (cont.)

- Use `while` to repeat a block of code until a condition changes.
- The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

```
1. var m = 2
2. repeat {
3.     m *= 2
4. } while m < 100
5. print(m)
```

Ranges

- Keep an index in a loop by using range operators to make a range of indexes.
 - *Closed range operator*: $(a \dots b)$ defines a range that runs from a to b , and includes the values a and b .
 - *Half-open range operator*: $(a \dots < b)$ defines a range that runs from a to b , but does not include b .

Ranges (cont.)

```
1.let names = ["Anna", "Alex", "Brian", "Jack"]
2.let count = names.count
3.for i in 0..
```

Functions and Closures

Swift Whirlwind Trip

Functions Declaration

- Use `func` to declare a function.
- Call a function by following its name with a list of arguments in parentheses.
- Use `->` to separate the parameter names and types from the function's return type.

```
1.func greet(person: String, day: String) -> String {  
2.    return "Hello \$(person), today is \$(day)."  
3.}  
4.greet(person: "Bob", day: "Tuesday")
```

Functions Declaration (cont.)

- By default, functions use their parameter names as *labels* for their arguments. Write a custom argument label before the parameter name, or write `_` to use no argument label.

```
1.func greet(_ person: String, on day: String) -> String {  
2.    return "Hello \$(person), today is \$(day)."  
3.}  
4.greet("John", on: "Wednesday")
```

Functions Declaration (cont.)

- Use a tuple to make a compound value.
 - For example, to return multiple values from a function.
 - The elements of a tuple can be referred to either by name or by number.
- Functions can also take a variable number of arguments, collecting them into an array.

Functions Declaration (cont.)

```
1.func sumOf(numbers: Int...) -> Int {  
2.    var sum = 0  
3.    for number in numbers {  
4.        sum += number  
5.    }  
6.    return sum  
7.}  
8.sumOf()  
9.sumOf(numbers: 42, 597, 12)
```

Nested Functions

- Nested functions have access to variables that were declared in the outer function.
- Organize the complex or long code by using nested functions.

```
1.func returnFifteen() -> Int {  
2.    var y = 10  
3.    func add() {  
4.        y += 5  
5.    }  
6.    add()  
7.    return y  
8.}  
9.returnFifteen()
```

Nested Functions (cont.)

- Functions are a *first-class type*.
 - A function can return another function as its value.

```
1. func makeIncrementer() -> ((Int) -> Int) {  
2.     func addOne(number: Int) -> Int {  
3.         return 1 + number  
4.     }  
5.     return addOne  
6. }  
7. var increment = makeIncrementer()  
8. increment(7)
```

Nested Functions (cont.)

- A function can take another function as one of its arguments.

```
1. func hasAnyMatches(list: [Int], condition: (Int) -> Bool) -> Bool {  
2.     for item in list {  
3.         if condition(item) {  
4.             return true  
5.         }  
6.     }  
7.     return false  
8. }  
9. func lessThanTen(number: Int) -> Bool {  
10.     return number < 10  
11. }  
12. var numbers = [20, 19, 7, 12]  
13. hasAnyMatches(list: numbers, condition: lessThanTen)
```

In-Out Parameters

- Function parameters are constants by default. Trying to change the value of a function parameter from within the body of that function results in a compile-time error. This means that you can't change the value of a parameter by mistake. If you want a function to modify a parameter's value, and you want those changes to persist after the function call has ended, define that parameter as an *in-out parameter* instead.

In-Out Parameters (cont.)

- You write an in-out parameter by placing the `inout` keyword right before a parameter's type. An in-out parameter has a value that is passed *in* to the function, is modified by the function, and is passed back *out* of the function to replace the original value.

```
1.func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
2.    let temporaryA = a  
3.    a = b  
4.    b = temporaryA  
5.}
```

In-Out Parameters (cont.)

- You can call the `swapTwoInts(_:_:)` function with two variables of type `Int` to swap their values. Note that the names of `someInt` and `anotherInt` are prefixed with an ampersand when they are passed to the `swapTwoInts(_:_:)` function.

```
1.var someInt = 3
2.var anotherInt = 107
3.swapTwoInts(&someInt, &anotherInt)
4.print("someInt is now \someInt, and anotherInt is now \anotherInt")
5.// Prints "someInt is now 107, and anotherInt is now 3"
```

In-Out Parameters (cont.)

- In-out parameters are passed as follows:
 - When the function is called, the value of the argument is copied.
 - In the body of the function, the copy is modified.
 - When the function returns, the copy's value is assigned to the original argument.
- This behavior is known as *copy-in copy-out* or *call by value result*. For example, when a computed property or a property with observers is passed as an in-out parameter, its getter is called as part of the function call and its setter is called as part of the function return.

In-Out Parameters (cont.)

- As an optimization, when the argument is a value stored at a physical address in memory, the same memory location is used both inside and outside the function body.
- The optimized behavior is known as *call by reference*; it satisfies all of the requirements of the copy-in copy-out model while removing the overhead of copying. Write your code using the model given by copy-in copy-out, without depending on the call-by-reference optimization, so that it behaves correctly with or without the optimization.

In-Out Parameters (cont.)

- Do not access the value that was passed as an in-out argument, even if the original argument is available in the current scope. When the function returns, your changes to the original are overwritten with the value of the copy.
- Do not depend on the implementation of the call-by-reference optimization to try to keep the changes from being overwritten.

In-Out Parameters (cont.)

- A closure or nested function that captures an in-out parameter must be **nonescaping**. If you need to capture an in-out parameter without mutating it or to observe changes made by other code, use a capture list to explicitly capture the parameter immutably.

```
1.func someFunction(a: inout Int) -> () -> Int {  
2.    return { [a] in return a + 1 }  
3.}
```

Functions: A Special Case of Closures

- The blocks of a function's code that can be called later.
- The code in a *closure* has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it is executed.
- Write a closure without a name by surrounding code with braces (`{}`).
- Use `in` to separate the arguments and return type from the body.

```
1. numbers.map({ (number: Int) -> Int in  
2.     let result = 3 * number  
3.     return result  
4. })
```

Concise Closures

- When a closure's type is already known, such as the callback for a delegate, you can omit the type of its parameters, its return type, or both.
- Single statement closures implicitly return the value of their only statement.

```
1.let mappedNumbers = numbers.map({ number in 3 * number })  
2.print(mappedNumbers)
```

Concise Closures (cont.)

- Refer to parameters by number instead of by name.
 - Especially useful in very short closures.
- A closure passed as the last argument to a function can appear immediately after the parentheses.
- When a closure is the **only** argument to a function, can omit the parentheses entirely.

```
1.let sortedNumbers = numbers.sorted { $0 > $1 }  
2.print(sortedNumbers)
```

Higher Order Functions

Swift Whirlwind Trip

Map

- Loops over a collection and applies the same operation to each element in the collection.
- Case study: multiplying each item by 2 in an array `numberArray`.

```
1.let numberArray = [1, 2, 3, 4, 5]
2.var emptyArray:[Int] = []
3.for number in numberArray {
4.    emptyArray.append(number * 2)
5.}
6.// emptyArray = [2, 4, 6, 8, 10]
```


Map (cont.)

```
1.var numberArray = [1, 2, 3, 4, 5]
2.// Map 1
3.var newArray1 = numbers.map({ (value: Int) -> Int in
4.    return value * 2
5.})
6.// Map 2
7.var newArray2 = numbers.map({ (value: Int) in
8.    return value * 2
9.})
10.// Map 3
11.var newArray3 = numbers.map{value in value * 2}
12.// Map 4
13.var newArray4 = numbers.map{ $0 * 2 }
```

Filter

- Loops over a collection and returns an array that contains elements that meet a condition.
- Case study: filter even numbers only in an array.

```
1.let numberArray = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2.var emptyArray:[Int] = []
3.for number in numberArray {
4.    if number % 2 == 0 {
5.        emptyArray.append(number * 2)
6.    }
7.}
8.// emptyArray = [2, 4, 6, 8]
```

Filter (cont.)

- Instead, use `filter` for the same result.

```
1.let numberArray = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2.let emptyArray = numberArray.filter{ $0 % 2 == 0 }
```

Reduce

- Combines all items in a collection to create a single value.
- Case study: get the sum of all the numbers in an array of integers.

```
1.let numberArray = [1, 2, 3, 4, 5]
2.var sumIs:Int = 0
3.for number in numberArray {
4.    sumIs += number
5.}
6.// sumIs = 15
```

Reduce (cont.)

- Use reduce for the same result.
- The two parameters in reduce function are a **starting value** and a **function** respectively.
 - The function takes a running total and an element of the array as parameters, and returns a new running total.

```
1.let numberArray = [1, 2, 3, 4, 5]
2.var sumIs = numberArray.reduce(0, { $0 + $1 })
3.// sumIs = 15
```

Reduce (cont.)

- A more concise way is omitting the argument.
 - The + operator is substituted for the $\{ \$0 + \$1 \}$ closure, creating an even more simplified and accessible syntax.

```
1.let numberArray = [1, 2, 3, 4, 5]
2.var sumIs = numberArray.reduce(0, +)
3.// sumIs = 15
```

FlatMap

- When implemented on sequences: Flattens a collection of collections.
- Assume that two arrays are within an array, how to combine into a single array?

```
1.let numberArray = [[1, 2, 3], [4, 5, 6]]
2.var emptyArray:[Int] = []
3.for number in numberArray {
4.    emptyArray += numberArray
5.}
6.// emptyArray = [1, 2, 3, 4, 5, 6]
```

```
1.let numberArray = [[1, 2, 3], [4, 5, 6]]
2.let flattenedArray = numberArray.flatMap{$0}
3.// flattenedArray = [1, 2, 3, 4, 5, 6]
```

FlatMap (cont.)

- When implemented on optionals: Removes `nil` from the collection.

```
1.let people: [String?] = ["Bob", nil, "Jack", nil, "Alice"]  
2.let validPeople = people.flatMap{$0}  
3.// validPeople = ["Bob", "Jack", "Alice"]
```


Chaining

- Add the squares of all the even numbers from an array of arrays.
- Combine all those HOFs in one line of code.

```
1.// Chaining - flatMap + filter + map + reduce
2.let arrayInArray = [[11, 12, 13], [14, 15, 16]]
3.var newValue = arrayInArray.flatMap{$0}.filter{ $0 % 2 == 0 }
    .map{$0 * $0}.reduce(0, +)
4.// newValue = 596
```

Why Use Higher Order Functions?

- Read and understand complex functional programming.
- Write more **elegantly** and **maintainable** code that has better **readability**.

Objects and Classes

Swift Whirlwind Trip

Classes Declaration

- Use `class` followed by the class's name to create a class.
- A property declaration in a class is written the same way as a constant or variable declaration, except that it is in the context of a class. Likewise, method and function declarations are written the same way.

```
1.class Shape {  
2.    var numberOfSides = 0  
3.    func simpleDescription() -> String {  
4.        return "A shape with \$(numberOfSides) sides."  
5.    }  
6.}
```

Instances

- Create an instance of a class by putting parentheses after the class name. Use dot syntax to access the properties and methods of the instance.
- This version of the Shape class is missing something important: an initializer to set up the class when an instance is created. Use `init` to create one.

```
1. var shape = Shape()  
2. shape.numberOfSides = 7  
3. var shapeDescription = shape.simpleDescription()
```

Class_INITIALIZER

```
1.class NamedShape {  
2.    var numberOfSides: Int = 0  
3.    var name: String  
4.    init(name: String) {  
5.        self.name = name  
6.    }  
7.    func simpleDescription() -> String {  
8.        return "A shape with \$(numberOfSides) sides."  
9.    }  
10.}
```

Class_INITIALIZER and Deinitializer

- Notice how `self` is used to distinguish the `name` property from the `name` argument to the initializer.
- The arguments to the initializer are passed like a function call when you create an instance of the class.
 - Every property needs a value assigned—either in its declaration (as with `numberOfSides`) or in the initializer (as with `name`).
- Use `deinit` to create a deinitializer if you need to perform some cleanup before the object is deallocated.

Class Inheritance

- *Subclasses* include their *superclass* name after their class name, separated by a colon. There is no requirement for classes to subclass any standard root class, so you can include or omit a superclass as needed.
- Methods on a subclass that override the superclass's implementation are marked with `override`—overriding a method by accident, without `override`, is detected by the compiler as an error. The compiler also detects methods with `override` that don't actually override any method in the superclass.

Class Inheritance (cont.)

```
1. class Square: NamedShape {
2.     var sideLength: Double
3.     init(sideLength: Double, name: String) {
4.         self.sideLength = sideLength
5.         super.init(name: name)
6.         numberOfSides = 4
7.     }
8.     func area() -> Double {
9.         return sideLength * sideLength
10.    }
11.    override func simpleDescription() -> String {
12.        return "A square with sides of length \(sideLength)."
13.    }
14.}
15.let test = Square(sideLength: 5.2, name: "my test square")
16.test.area()
17.test.simpleDescription()
```

Getter and Setter in Classes

- In addition to simple properties that are stored, properties can have a getter and a setter.
- In the setter for `perimeter`, the new value has the implicit name `newValue`. You can provide an explicit name in parentheses after `set`.
- Notice that the initializer for the `EquilateralTriangle` class has three different steps:
 1. Setting the value of properties that the subclass declares.
 2. Calling the superclass's initializer.
 3. Changing the value of properties defined by the superclass. Any additional setup work that uses methods, getters, or setters can also be done at this point.

Getter and Setter in Classes (cont.)

```
1. class EquilateralTriangle: NamedShape {
2.     var sideLength: Double = 0.0
3.     init(sideLength: Double, name: String) {
4.         self.sideLength = sideLength
5.         super.init(name: name)
6.         numberOfSides = 3
7.     }
8.     var perimeter: Double {
9.         get { return 3.0 * sideLength }
10.        set { sideLength = newValue / 3.0 }
11.    }
12.    override func simpleDescription() -> String {
13.        return "An equilateral triangle with sides of length \$(sideLength)."
14.    }
15.}
16.var triangle = EquilateralTriangle(sideLength: 3.1, name: "a triangle")
17.print(triangle.perimeter)
18.triangle.perimeter = 9.9
19.print(triangle.sideLength)
```

Getter and Setter in Classes (cont.)

- If you don't need to compute the property but still need to provide code that is run before and after setting a new value, use `willSet` and `didSet`.
- The code you provide is run any time the value changes outside of an initializer. For example, the class below ensures that the side length of its triangle is **always** the same as the side length of its square.

Getter and Setter in Classes (cont.)

```
1.class TriangleAndSquare {
2.    var triangle: EquilateralTriangle {
3.        willSet { square.sideLength = newValue.sideLength }
4.    }
5.    var square: Square {
6.        willSet { triangle.sideLength = newValue.sideLength }
7.    }
8.    init(size: Double, name: String) {
9.        square = Square(sideLength: size, name: name)
10.        triangle = EquilateralTriangle(sideLength: size, name: name)
11.    }
12.}
13.var triangleAndSquare = TriangleAndSquare(size: 10, name: "another test shape")
14.print(triangleAndSquare.square.sideLength)
15.print(triangleAndSquare.triangle.sideLength)
16.triangleAndSquare.square = Square(sideLength: 50, name: "larger square")
17.print(triangleAndSquare.triangle.sideLength)
```

More about Optionals

- When working with optional values, write `?` before operations like methods, properties, and subscripting.
- If the value before the `?` is `nil`, everything after the `?` is ignored and the value of the whole expression is `nil`. Otherwise, the optional value is unwrapped, and everything after the `?` acts on the unwrapped value. In both cases, the value of the whole expression is an optional value.

```
1.let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional square")
2.let sideLength = optionalSquare?.sideLength
```

Enumerations and Structures

Swift Whirlwind Trip

Enumerations

- Use `enum` to create an enumeration. Like classes and all other named types, enumerations can have methods associated with them.

```
1. enum Rank: Int {  
2.     case ace = 1  
3.     case two, three, four, five, six, seven, eight, nine, ten  
4.     case jack, queen, king  
5.     func simpleDescription() -> String {  
6.         switch self {  
7.             case .ace: return "ace"  
8.             case .jack: return "jack"  
9.             case .queen: return "queen"  
10.            case .king: return "king"  
11.            default: return String(self.rawValue)  
12.        }  
13.    }  
14.}  
15.let ace = Rank.ace  
16.let aceRawValue = ace.rawValue
```


Enumerations (cont.)

- By default, Swift assigns the raw values starting at zero and incrementing by one each time, but you can change this behavior by explicitly specifying values.
 - In the example above, Ace is explicitly given a raw value of 1, and the rest of the raw values are assigned in order. You can also use strings or floating-point numbers as the raw type of an enumeration. Use the `rawValue` property to access the raw value of an enumeration case.

Enumerations (cont.)

- Use the `init?(rawValue:)` initializer to make an instance of an enumeration from a raw value. It returns either the enumeration case matching the raw value or `nil` if there is no matching Rank.

```
1. if let convertedRank = Rank(rawValue: 3) {  
2.     let threeDescription = convertedRank.simpleDescription()  
3. }
```

Enumerations (cont.)

- The case values of an enumeration are actual values, not just another way of writing their raw values. In fact, in cases where there isn't a meaningful raw value, you don't have to provide one.

```
1. enum Suit {  
2.     case spades, hearts, diamonds, clubs  
3.     func simpleDescription() -> String {  
4.         switch self {  
5.             case .spades: return "spades"  
6.             case .hearts: return "hearts"  
7.             case .diamonds: return "diamonds"  
8.             case .clubs: return "clubs"  
9.         }  
10.    }  
11.}  
12.let hearts = Suit.hearts  
13.let heartsDescription = hearts.simpleDescription()
```

Enumerations (cont.)

- Notice the two ways that the `hearts` case of the enumeration is referred to above: When assigning a value to the `hearts` constant, the enumeration case `Suit.hearts` is referred to by its full name because the constant doesn't have an explicit type specified.
- Inside the switch, the enumeration case is referred to by the abbreviated form `.hearts` because the value of `self` is already known to be a suit. You can use the abbreviated form anytime the value's type is already known.

Enumerations (cont.)

- If an enumeration has raw values, those values are determined as part of the declaration, which means every instance of a particular enumeration case always has the same raw value.
- Another choice for enumeration cases is to have values associated with the case—these values are determined when you make the instance, and they can be different for each instance of an enumeration case. You can think of the associated values as behaving like stored properties of the enumeration case instance. For example, consider the case of requesting the sunrise and sunset times from a server. The server either responds with the requested information, or it responds with a description of what went wrong.

Enumerations (cont.)

- Notice how the sunrise and sunset times are extracted from the `ServerResponse` value as part of matching the value against the switch cases.

Enumerations (cont.)

```
1.enum ServerResponse {  
2.    case result(String, String)  
3.    case failure(String)  
4.}  
5.let success = ServerResponse.result("6:00 am", "8:09 pm")  
6.let failure = ServerResponse.failure("Out of cheese.")  
7.switch success {  
8.case let .result(sunrise, sunset):  
9.    print("Sunrise is at \(sunrise) and sunset is at \(sunset).")  
10.case let .failure(message):  
11.    print("Failure... \(message)")  
12.}
```

Structures

- Use `struct` to create a structure.
- Structures support many of the same behaviors as classes, including methods and initializers.
- One of the most important differences between structures and classes is that structures are always **copied** when they are passed around in your code, but classes are **passed by reference**.

Structures (cont.)

```
1. struct Card {  
2.     var rank: Rank  
3.     var suit: Suit  
4.     func simpleDescription() -> String {  
5.         return "The \(rank.simpleDescription()) of  
        \ (suit.simpleDescription())"  
6.     }  
7. }  
8. let threeOfSpades = Card(rank: .three, suit: .spades)  
9. let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

Protocols and Extensions

Swift Whirlwind Trip

Protocols

- Use `protocol` to declare a protocol.
- Classes, enumerations, and structs can all adopt protocols.

```
1.protocol ExampleProtocol {  
2.    var simpleDescription: String { get }  
3.    mutating func adjust()  
4.}
```

Protocols (cont.)

```
1.class SimpleClass: ExampleProtocol {  
2.    var simpleDescription: String = "A very simple class."  
3.    var anotherProperty: Int = 69105  
4.    func adjust() { simpleDescription += " Now 100% adjusted." }  
5.}  
6.var a = SimpleClass()  
7.a.adjust()  
8.let aDescription = a.simpleDescription  
9.struct SimpleStructure: ExampleProtocol {  
10.    var simpleDescription: String = "A simple structure"  
11.    mutating func adjust() { simpleDescription += " (adjusted)" }  
12.}  
13.var b = SimpleStructure()  
14.b.adjust()  
15.let bDescription = b.simpleDescription
```

Protocols (cont.)

- Notice the use of the `mutating` keyword in the declaration of `SimpleStructure` to mark a method that modifies the structure.
 - The declaration of `SimpleClass` doesn't need any of its methods marked as mutating because methods on a class can always modify the class.

Extensions

- Use `extension` to add functionality to an existing type, such as new methods and computed properties.
- You can use an extension to add protocol conformance to a type that is declared elsewhere, or even to a type that you imported from a library or framework.

Extensions (cont.)

```
1.extension Int: ExampleProtocol {  
2.    var simpleDescription: String {  
3.        return "The number \$(self)"  
4.    }  
5.    mutating func adjust() {  
6.        self += 42  
7.    }  
8.}  
9.print(7.simpleDescription)
```

Scope of Protocols

- You can use a protocol name just like any other named type—for example, to create a collection of objects that have different types but that all conform to a single protocol. When you work with values whose type is a protocol type, methods outside the protocol definition are not available.

```
1.let protocolValue: ExampleProtocol = a
2.print(protocolValue.simpleDescription)
3.// print(protocolValue.anotherProperty) // Uncomment to see the error
```


Scope of Protocols (cont.)

- Even though the variable `protocolValue` has a runtime type of `SimpleClass`, the compiler treats it as the given type of `ExampleProtocol`. This means that you can't accidentally access methods or properties that the class implements in addition to its protocol conformance.

Error Handling

Swift Whirlwind Trip

Error Handling

- You represent errors using any type that adopts the Error protocol.

```
1. enum PrinterError: Error {  
2.     case outOfPaper  
3.     case noToner  
4.     case onFire  
5. }
```

Error Handling (cont.)

- Use `throw` to throw an error and `throws` to mark a function that can throw an error. If you throw an error in a function, the function returns immediately and the code that called the function handles the error.

```
1.func send(job: Int, toPrinter printerName: String) throws -> String {  
2.    if printerName == "Never Has Toner" {  
3.        throw PrinterError.noToner  
4.    }  
5.    return "Job sent"  
6.}
```

Error Handling (cont.)

- There are several ways to handle errors. One way is to use `do-catch`. Inside the `do` block, you mark code that can throw an error by writing `try` in front of it. Inside the `catch` block, the error is automatically given the name `error` unless you give it a different name.

```
1.do {  
2.    let printerResponse = try send(job: 1040, toPrinter: "Bi Sheng")  
3.    print(printerResponse)  
4.} catch {  
5.    print(error)  
6.}
```

Error Handling (cont.)

- You can provide multiple catch blocks that handle specific errors. You write a pattern after catch just as you do after case in a switch.

```
1.do {  
2.    let printerResponse = try send(job: 1440, toPrinter: "Gutenberg")  
3.    print(printerResponse)  
4.} catch PrinterError.onFire {  
5.    print("I'll just put this over here, with the rest of the fire.")  
6.} catch let printerError as PrinterError {  
7.    print("Printer error: \(printerError).")  
8.} catch {  
9.    print(error)  
10.}
```

Error Handling (cont.)

- Another way to handle errors is to use `try?` to convert the result to an optional. If the function throws an error, the specific error is discarded and the result is `nil`. Otherwise, the result is an optional containing the value that the function returned.

```
1.let printerSuccess = try? send(job: 1884, toPrinter: "Mergenthaler")  
2.let printerFailure = try? send(job: 1885, toPrinter: "Never Has Toner")
```

Error Handling (cont.)

- Use `defer` to write a block of code that is executed after all other code in the function, just before the function returns. The code is executed regardless of whether the function throws an error. You can use `defer` to write setup and cleanup code next to each other, even though they need to be executed at different times.

Error Handling (cont.)

```
1.var fridgeIsOpen = false
2.let fridgeContent = ["milk", "eggs", "leftovers"]
3.func fridgeContains(_ food: String) -> Bool {
4.    fridgeIsOpen = true
5.    defer {
6.        fridgeIsOpen = false
7.    }
8.    let result = fridgeContent.contains(food)
9.    return result
10.}
11.fridgeContains("banana")
12.print(fridgeIsOpen)
```

Generics

Swift Whirlwind Trip

Generics

- Write a name inside angle brackets to make a generic function or type.

```
1. func makeArray<Item>(repeating item: Item, numberOfTimes: Int) -> [Item] {  
2.     var result = [Item]()  
3.     for _ in 0..  
4.         result.append(item)  
5. }  
6.     return result  
7. }  
8. makeArray(repeating: "knock", numberOfTimes: 4)
```

Generics (cont.)

- You can make generic forms of functions and methods, as well as classes, enumerations, and structures.

```
1.// Reimplement the Swift standard library's optional type
2.enum OptionalValue<Wrapped> {
3.    case none
4.    case some(Wrapped)
5.}
6.var possibleInteger: OptionalValue<Int> = .none
7.possibleInteger = .some(100)
```

Generics (cont.)

- Use `where` right before the body to specify a list of requirements—for example, to require the type to implement a protocol, to require two types to be the same, or to require a class to have a particular superclass.
- Writing `<T: Equatable>` is the same as writing `<T> ... where T: Equatable`.

Generics (cont.)

```
1. func anyCommonElements<T: Sequence, U: Sequence>(_ lhs: T, _ rhs: U) -> Bool
2.     where T.Iterator.Element: Equatable, T.Iterator.Element ==
   U.Iterator.Element {
3.         for lhsItem in lhs {
4.             for rhsItem in rhs {
5.                 if lhsItem == rhsItem {
6.                     return true
7.                 }
8.             }
9.         }
10.    return false
11.}
12.anyCommonElements([1, 2, 3], [3])
```