翻译：numbbbbb　校对：shinyzhu

Swift 初见

本页内容包括：

通常来说，编程语言教程中的第一个程序应该在屏幕上打印“Hello, world”。在 Swift 中，可以用一行代码实现：

```
println("Hello, world")
```

如果你写过 C 或者 Objective-C 代码，那你应该很熟悉这种形式——在 Swift 中，这行代码就是一个完整的程序。你不需要为了输入输出或者字符串处理导入一个单独的库。全局作用域中的代码会被自动当做程序的入口点，所以你也不需要`main`函数。你同样不需要在每个语句结尾写上分号。

这个教程会通过一系列编程例子来让你对 Swift 有初步了解，如果你有什么不理解的地方也不用担心——任何本章介绍的内容都会在后面的章节中详细讲解。

注意：

为了获得最好的体验，在 Xcode 当中使用代码预览功能。代码预览功能可以让你编辑代码并实时看到运行结果。

简单值

使用`let`来声明常量，使用`var`来声明变量。一个常量的值在编译时并不需要获取，但是你只能为它赋值一次。也就是说你可以用常量来表示这样一个值：你只需要决定一次，但是需要使用很多次。

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

常量或者变量的类型必须和你赋给它们的值一样。然而，声明时类型是可选的，声明的同时赋值的话，编译器会自动推断类型。在上面的例子中，编译器推断出`myVariable`是一个整数（integer）因为它的初始值是整数。

如果初始值没有提供足够的信息（或者没有初始值），那你需要在变量后面声明类型，用冒号分割。

```
let implicitInteger = 70
let implicitDouble = 70.0
let explicitDouble: Double = 70
```

练习：

创建一个常量，显式指定类型为`Float`并指定初始值为4。

值永远不会被隐式转换为其他类型。如果你需要把一个值转换成其他类型，请显式转换。

```
let label = "The width is"
let width = 94
let widthLabel = label + String(width)
```

练习：

删除最后一行中的`String`，错误提示是什么？

有一种更简单的把值转换成字符串的方法：把值写到括号中，并且在括号之前写一个反斜杠。例如：

```
let apples = 3
let oranges = 5
```

```
let appleSummary = "I have \(apples) apples."
let fruitSummary = "I have \(apples + oranges) pieces
of fruit."
```

练习：

使用`\()`来把一个浮点计算转换成字符串，并加上某人的名字，和他打个招呼。

使用方括号`[]`来创建数组和字典，并使用下标或者键（key）来访问元素。

```
var shoppingList = ["catfish", "water", "tulips",
"blue paint"]
shoppingList[1] = "bottle of water"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

要创建一个空数组或者字典，使用初始化语法。

```
let emptyArray = String[]()
let emptyDictionary = Dictionary<String, Float>()
```

如果类型信息可以被推断出来，你可以用`[]`和`[:]`来创建空数组和空字典——就像你声明变量或者给函数传参数的时候一样。

```
shoppingList = []   // 去逛街并买点东西
```

控制流

使用`if`和`switch`来进行条件操作，使用`for-in`、`for`、`while`和`do-while`来进行循环。包裹条件和循环变量括号可以省略，但是语句体的大括号是必须的。

```
let individualScores = [75, 43, 103, 87, 12]
var teamScore = 0
```

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

在`if`语句中，条件必须是一个布尔表达式——这意味着像`if score { ... }`这样的代码将报错，而不会隐形地与 0 做对比。

你可以一起使用`if`和`let`来处理值缺失的情况。有些变量的值是可选的。一个可选的值可能是一个具体的值或者是`nil`，表示值缺失。在类型后面加一个问号来标记这个变量的值是可选的。

```
var optionalString: String? = "Hello"
optionalString == nil

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

练习：

把`optionalName`改成`nil`，greeting会是什么？添加一个`else`语句，当`optionalName`是`nil`时给greeting赋一个不同的值。

如果变量的可选值是`nil`，条件会判断为`false`，大括号中的代码会被跳过。如果不是`nil`，会将值赋给`let`后面的常量，这样代码块中就可以使用这个值了。

`switch`支持任意类型的数据以及各种比较操作——不仅仅是整数以及测试相等。

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
```

```
    let vegetableComment = "Add some raisins and make
ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good
tea sandwich."
case let x where x.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(x)?"
default:
    let vegetableComment = "Everything tastes good in
soup."
}
```

练习：

删除`default`语句，看看会有什么错误？

运行`switch`中匹配到的子句之后，程序会退出`switch`语句，并不会继续向下运行，所以不需要在每个子句结尾写`break`。

你可以使用`for-in`来遍历字典，需要两个变量来表示每个键值对。

```
let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]
var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}
largest
```

练习：

添加另一个变量来记录哪种类型的数字是最大的。

使用`while`来重复运行一段代码直到不满足条件。循环条件可以在开头也

可以在结尾。

```
var n = 2
while n < 100 {
    n = n * 2
}
n

var m = 2
do {
    m = m * 2
} while m < 100
m
```

你可以在循环中使用`..`来表示范围，也可以使用传统的写法，两者是等价的：

```
var firstForLoop = 0
for i in 0..3 {
    firstForLoop += i
}
firstForLoop

var secondForLoop = 0
for var i = 0; i < 3; ++i {
    secondForLoop += 1
}
secondForLoop
```

使用`..`创建的范围不包含上界，如果想包含的话需要使用`...`。

函数和闭包

使用`func`来声明一个函数，使用名字和参数来调用函数。使用`->`来指定函数返回值。

```
func greet(name: String, day: String) -> String {
```

```
    return "Hello \(name), today is \(day)."
}
greet("Bob", "Tuesday")
```

练习：

删除day参数，添加一个参数来表示今天吃了什么午饭。

使用一个元组来返回多个值。

```
func getGasPrices() -> (Double, Double, Double) {
    return (3.59, 3.69, 3.79)
}
getGasPrices()
```

函数的参数数量是可变的，用一个数组来获取它们：

```
func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
        sum += number
    }
    return sum
}
sumOf()
sumOf(42, 597, 12)
```

练习：

写一个计算参数平均值的函数。

函数可以嵌套。被嵌套的函数可以访问外侧函数的变量，你可以使用嵌套函数来重构一个太长或者太复杂的函数。

```
func returnFifteen() -> Int {
    var y = 10
    func add() {
        y += 5
    }
    add()
    return y
}
returnFifteen()
```

函数是一等公民，这意味着函数可以作为另一个函数的返回值。

```
func makeIncrementer() -> (Int -> Int) {
    func addOne(number: Int) -> Int {
        return 1 + number
    }
    return addOne
}
var increment = makeIncrementer()
increment(7)
```

函数也可以当做参数传入另一个函数。

```
func hasAnyMatches(list: Int[], condition: Int ->
Bool) -> Bool {
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, lessThanTen)
```

函数实际上是一种特殊的闭包，你可以使用`{}`来创建一个匿名闭包。使用`in`来分割参数并返回类型。

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

练习：

重写闭包，对所有奇数返回 0.

有很多种创建闭包的方法。如果一个闭包的类型已知，比如作为一个回调

函数，你可以忽略参数的类型和返回值。单个语句闭包会把它语句的值当做结果返回。

你可以通过参数位置而不是参数名字来引用参数——这个方法在非常短的闭包中非常有用。当一个闭包作为最后一个参数传给一个函数的时候，它可以直接跟在括号后面。

```
sort([1, 5, 3, 12, 2]) { $0 > $1 }
```

对象和类

使用`class`和类名来创建一个类。类中属性的声明和常量、变量声明一样，唯一的区别就是它们的上下文是类。同样，方法和函数声明也一样。

```
class Shape {
    var numberOfSides = 0
    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

练习：

使用`let`添加一个常量属性，再添加一个接收一个参数的方法。

要创建一个类的实例，在类名后面加上括号。使用点语法来访问实例的属性和方法。

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

这个版本的`Shape`类缺少了一些重要的东西：一个构造函数来初始化类实例。使用`init`来创建一个构造器。

```
class NamedShape {
    var numberOfSides: Int = 0
    var name: String
```

```
    init(name: String) {
        self.name = name
    }

    func simpleDescription() -> String {
        return "A shape with \(numberOfSides) sides."
    }
}
```

注意`self`被用来区别实例变量。当你创建实例的时候，像传入函数参数一样给类传入构造器的参数。每个属性都需要赋值——无论是通过声明（就像`numberOfSides`）还是通过构造器（就像`name`）。

如果你需要在删除对象之前进行一些清理工作，使用`deinit`创建一个析构函数。

子类的定义方法是在它们的类名后面加上父类的名字，用冒号分割。创建类的时候并不需要一个标准的根类，所以你可以忽略父类。

子类如果要重写父类的方法的话，需要用`override`标记——如果没有添加`override`就重写父类方法的话编译器会报错。编译器同样会检测`override`标记的方法是否确实在父类中。

```
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() ->  Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \
```

```
(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test
square")
test.area()
test.simpleDescription()
```

练习：

创建NamedShape的另一个子类Circle，构造器接收两个参数，一个是半径一个是名称，实现area和describe方法。

属性可以有 getter 和 setter 。

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }

    var perimeter: Double {
    get {
        return 3.0 * sideLength
    }
    set {
                sideLength = newValue / 3.0
    }
    }

    override func simpleDescription() -> String {
        return "An equilateral triagle with sides of
length \(sideLength)."
    }
}
var triangle = EquilateralTriangle(sideLength: 3.1,
```

```
name: "a triangle")
triangle.perimeter
triangle.perimeter = 9.9
triangle.sideLength
```

在`perimeter`的 setter 中，新值的名字是`newValue`。你可以在`set`之后显式的设置一个名字。

注意`EquilateralTriangle`类的构造器执行了三步：

1. 设置子类声明的属性值
2. 调用父类的构造器
3. 改变父类定义的属性值。其他的工作比如调用方法、getters和setters也可以在这个阶段完成。

如果你不需要计算属性但是需要在设置一个新值之前运行一些代码，使用`willSet`和`didSet`。

比如，下面的类确保三角形的边长总是和正方形的边长相同。

```
class TriangleAndSquare {
    var triangle: EquilateralTriangle {
    willSet {
        square.sideLength = newValue.sideLength
    }
    }
    var square: Square {
    willSet {
        triangle.sideLength = newValue.sideLength
    }
    }
    init(size: Double, name: String) {
        square = Square(sideLength: size, name: name)
        triangle = EquilateralTriangle(sideLength:
size, name: name)
    }
}
var triangleAndSquare = TriangleAndSquare(size: 10,
name: "another test shape")
```

```
triangleAndSquare.square.sideLength
triangleAndSquare.triangle.sideLength
triangleAndSquare.square = Square(sideLength: 50,
name: "larger square")
triangleAndSquare.triangle.sideLength
```

类中的方法和一般的函数有一个重要的区别，函数的参数名只在函数内部使用，但是方法的参数名需要在调用的时候显式说明（除了第一个参数）。默认情况下，方法的参数名和它在方法内部的名字一样，不过你也可以定义第二个名字，这个名字被用在方法内部。

```
class Counter {
    var count: Int = 0
    func incrementBy(amount: Int, numberOfTimes
times: Int) {
        count += amount * times
    }
}
var counter = Counter()
counter.incrementBy(2, numberOfTimes: 7)
```

处理变量的可选值时，你可以在操作（比如方法、属性和子脚本）之前加`?`。如果`?`之前的值是`nil`，`?`后面的东西都会被忽略，并且整个表达式返回`nil`。否则，`?`之后的东西都会被运行。在这两种情况下，整个表达式的值也是一个可选值。

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

枚举和结构体

使用`enum`来创建一个枚举。就像类和其他所有命名类型一样，枚举可以包含方法。

```
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.toRaw())
        }
    }
}
let ace = Rank.Ace
let aceRawValue = ace.toRaw()
```

练习：

写一个函数，通过比较它们的原始值来比较两个`Rank`值。

在上面的例子中，枚举原始值的类型是`Int`，所以你只需要设置第一个原始值。剩下的原始值会按照顺序赋值。你也可以使用字符串或者浮点数作为枚举的原始值。

使用`toRaw`和`fromRaw`函数来在原始值和枚举值之间进行转换。

```
if let convertedRank = Rank.fromRaw(3) {
    let threeDescription =
convertedRank.simpleDescription()
}
```

枚举的成员值是实际值，并不是原始值的另一种表达方法。实际上，如果原始值没有意义，你不需要设置。

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

```
    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 heartsDescription = hearts.simpleDescription()
```

练习：

给Suit添加一个color方法，对spades和clubs返回“black”，对hearts和diamonds返回“red”。
注意，有两种方式可以引用Hearts成员：给hearts常量赋值时，枚举成员Suit.Hearts需要用全名来引用，因为常量没有显式指定类型。在switch里，枚举成员使用缩写.Hearts来引用，因为self的值已经知道是一个suit。已知变量类型的情况下你可以使用缩写。

使用struct来创建一个结构体。结构体和类有很多相同的地方，比如方法和构造器。它们之间最大的一个区别就是 结构体是传值，类是传引用。

```
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)
let threeOfSpadesDescription =
threeOfSpades.simpleDescription()
```

练习：

给`Card`添加一个方法，创建一副完整的扑克牌并把每张牌的 rank 和 suit 对应起来。

一个枚举成员的实例可以有实例值。相同枚举成员的实例可以有不同的值。创建实例的时候传入值即可。实例值和原始值是不同的：枚举成员的原始值对于所有实例都是相同的，而且你是在定义枚举的时候设置原始值。

例如，考虑从服务器获取日出和日落的时间。服务器会返回正常结果或者错误信息。

```
enum ServerResponse {
    case Result(String, String)
    case Error(String)
}

let success = ServerResponse.Result("6:00 am", "8:09
pm")
let failure = ServerResponse.Error("Out of cheese.")

switch success {
case let .Result(sunrise, sunset):
    let serverResponse = "Sunrise is at \(sunrise)
and sunset is at \(sunset)."
case let .Error(error):
    let serverResponse = "Failure...  \(error)"
}
```

练习：

给`ServerResponse`和`switch`添加第三种情况。

注意如何从`ServerResponse`中提取日升和日落时间。

接口和扩展

使用`protocol`来声明一个接口。

```
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()
}
```

类、枚举和结构体都可以实现接口。

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple
class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {
    var simpleDescription: String = "A simple
structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

练习:

写一个实现这个接口的枚举。

注意声明`SimpleStructure`时候`mutating`关键字用来标记一个会修改结构体的方法。`SimpleClass`的声明不需要标记任何方法因为类中的方法经

常会修改类。

使用`extension`来为现有的类型添加功能，比如添加一个计算属性的方法。你可以使用扩展来给任意类型添加协议，甚至是你从外部库或者框架中导入的类型。

```
extension Int: ExampleProtocol {
    var simpleDescription: String {
    return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
7.simpleDescription
```

练习：

给`Double`类型写一个扩展，添加`absoluteValue`功能。

你可以像使用其他命名类型一样使用接口名——例如，创建一个有不同类型但是都实现一个接口的对象集合。当你处理类型是接口的值时，接口外定义的方法不可用。

```
let protocolValue: ExampleProtocol = a
protocolValue.simpleDescription
// protocolValue.anotherProperty  // Uncomment to see the error
```

即使`protocolValue`变量运行时的类型是`simpleClass`，编译器会把它的类型当做`ExampleProtocol`。这表示你不能调用类在它实现的接口之外实现的方法或者属性。

泛型

在尖括号里写一个名字来创建一个泛型函数或者类型。

```
func repeat<ItemType>(item: ItemType, times: Int) ->
```

```
ItemType[] {
    var result = ItemType[]()
    for i in 0..times {
        result += item
    }
    return result
}
repeat("knock", 4)
```

你也可以创建泛型类、枚举和结构体。

```
// Reimplement the Swift standard library's optional
type
enum OptionalValue<T> {
    case None
    case Some(T)
}
var possibleInteger: OptionalValue<Int> = .None
possibleInteger = .Some(100)
```

在类型名后面使用where来指定一个需求列表——例如，要限定实现一个协议的类型，需要限定两个类型要相同，或者限定一个类必须有一个特定的父类。

```
func anyCommonElements <T, U where T: Sequence, U:
Sequence, T.GeneratorType.Element: Equatable,
T.GeneratorType.Element == U.GeneratorType.Element>
(lhs: T, rhs: U) -> Bool {
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true
            }
        }
    }
    return false
}
anyCommonElements([1, 2, 3], [3])
```

练习：

修改`anyCommonElements`函数来创建一个函数，返回一个数组，内容是两个序列的共有元素。
简单起见，你可以忽略`where`，只在冒号后面写接口或者类名。`<T: Equatable>`和`<T where T: Equatable>`是等价的。