//: Playground - noun: a place where people can play

//import UIKit

print("Hello World")

print("👦👧")

let π = 3.1415926

let 工资 = 5000

var yourAge: Int = 21

//类型推断

var weight = 3.5

weight = 4

print(weight)

//元组，一般值是由组合给出的，可以通过序号.0，.1，.2等给出

var 课程情况 = ("ios应用开发技术","徐慧",32)

课程情况.0

课程情况.1

课程情况.2

//也可以通过前缀给出

var 课程情况1 = (cName:"ios应用开发技术",tName:"徐慧",cTime:32)

课程情况1.cName

课程情况1.tName

课程情况1.cName

//可选类型 Optional，变量可能有值，默认为nil

var addr: String? = "江苏省徐州市"

//字符串

var a = ""

var b = " "

//判断字符串是否为空

a.isEmpty

b.isEmpty

//字符串与字符

var c = "我" //字符串

var d: Character = "我" //字符

//对一个字符串的characters属性进行循环，访问单个字符

let coureName = "ios应用开发技术"

for char in coureName {

print(char)

}

//连接字符串和字符，用+

let fName = "张"

let gName = "三丰"

let title = "一代大师"

var famous = title + fName + gName

//向字符串添加字符，用append方法

let number: Character = "1"

famous.append(number)

//字符串插值，组合常量、变量、表达式等成为一个长字符串

let name = "Nancy"

let classTime = 32

let expTime = 32

var des = "\(name)参加了ios课程的学习，一共要学习\(classTime + expTime)学时！"

//数组

//创建有重复值的数组

let array = [Int](repeatElement(3, count: 10))

//创建一个有序范围的Int数组

let array2 = Array(1...100)

//创建数组

var places = ["beijing","shanghai","nanjing","changsha"]

//元素计数

places.count

//是否为空

places.isEmpty

//添加：append

places.append("wuhan")

places.count

let otherplaces = ["Tokyo","Paris"]

places += otherplaces

//获取元素：数组名[索引]

places[6]

//插入:insert

places.insert("Newyork", at: 5)

//删除:remove

places.remove(at: 5)

places.removeSubrange(0...2)

places

//字典

//定义：Dictionary<键类型，值类型>，或[键类型：值类型]

var a: Dictionary<String,String> = ["001":"张三"]

var b: [String:String] = ["001":"张三"]

//创建字典

var airports = ["PVG":"Shanghai pudong","CHU":"Dalian","DUB":"DUBLIN Airport"]

//字典是否为空

airports.isEmpty

//字典计数

airports.count

//添加或更新：字典变量[键] = 值

airports["SHQ"] = "Hongqiao Airport"

airports["CHU"] = "大连周水子机场"

airports

//获取，可以用下标

airports["DUB"]

//移除，用下标把值设为nil

airports["DUB"] = nil

//循环一个字典for in，因为键值对有2个元素，用元组变量

for (key, value) in airports{

print(key,value)

}

//单独使用其中键或值，使用keys或values(可使用for in)

for key in airports.keys{

print(key)

}

//把键值对分类成数组，用[数组类型]（字典变量.keys），[数组类型]（字典变量.values）

let codes = [String](airports.keys)

let airportFullname = [String](airports.values )

//函数

func add(a:Int, b:Int) -> Int{

return a + b

}

var sum = add(a: 3, b: 4)

//无参数无返回值

func welcome(){

print("欢迎学习")

print("iOS应用开发技术")

}

welcome()

//多返回值（使用元组）Int默认是Int64

func maxMin() -> (Int,Int){

return(Int.min,Int.max)

}

maxMin()

//可以给某个参数以默认值

func add2(x:Int, increment:Int = 2) -> Int{

return x + increment

}

add2(x: 3)

add2(x: 3, increment: 5)

//参数类型可以是函数

func calculate(x:Int, y:Int, **method:(Int,Int) -> Int)** -> Int{

return method(x, y)

}

func add3(x:Int, y:Int) -> Int{

return x + y

}

//传递的函数类型要与参数中的函数类型一致

calculate(x: 5, y: 8, method: add3)

//参数标签

//func retangleArea(w width: Double, h height: Double) -> Double{

// let area = width \* height

// return area

//}

//retangleArea(w: 20, h: 30)

//如果不写参数标签，用\_代替

func retangleArea(\_ width: Double, h height: Double) ->Double{

let area = width \* height

return area

}

retangleArea(20, h: 30)

//闭包，是一种函数的简写形式，省去函数名，把参数和返回值放入花括号内

var city = ["xuzhou", "nanjing", "suzhou", "lianyungang"]

var cityRank1 = city.sorted()

func daoxu(a:String, b:String) -> Bool{

return a > b

}

var cityRank2 = city.sorted(by: daoxu)

//用闭包来改写，闭包里面不用写参数类型，可以自动推断，sorted方法敲两次回车

var cityRank3 = city.sorted { (a, b) -> Bool in

return a > b

}

//参数和返回类型可自动推断，单表达式可以忽略return关键词

var cityRank4 = city.sorted { (a, b) in

a > b

}

//可使用快捷参数，前缀$，从0开始递增

var cityRank5 = city.sorted { $0 > $1

}

cityRank5

//枚举

//给定一个名称，然后把每一种情况列举出来

enum week {

case Monday

case Tuesday

case Wednesday

case Thursday

case Friday

case Saturday

case Sunday

}

//使用：点语法

week.Monday

//与switch配合使用

var today = week.Friday

switch today {

case .Monday:

print("今天是周一")

case .Tuesday:

print("今天是周二")

case .Wednesday:

print("今天是周三")

case .Thursday:

print("今天是周四")

case .Friday:

print("今天是周五")

default:

print("今天是周末")

}

//类与对象

class 角色{

var id = ""

var money = 0

}

let Mike = 角色()

Mike.id = "迈克"

Mike.money = 1000

struct 坐标{

var x = 0, y = 0

}

enum 移动方式{

case 走

case 跑

case 骑

case 传送

}

//类型属性，属于类型固有的，实例不能调用，前面加static

class 生命体{

//static var 遗传方式:String{

class var 遗传方式:String{

return "RNA"

}

}

生命体.遗传方式

class Human: 生命体 {

override class var 遗传方式:String{

return "DNA"

}

}

Human.遗传方式

//计算属性：当前坐标

class 巫师:角色{

var 人物坐标 = 坐标()

var 人物移动方式 = 移动方式.走

var 当前坐标: 坐标{

get{

switch 人物移动方式 {

case .走:

人物坐标.x += 1

人物坐标.y += 1

case .跑:

人物坐标.x += 5

人物坐标.y += 5

case .骑:

人物坐标.x += 10

人物坐标.y += 10

case .传送:

人物坐标.x += 1000

人物坐标.y += 1000

}

return 人物坐标

}

set{

人物坐标 = newValue

}

}

}

var 巫师1 = 巫师()

巫师1.当前坐标

巫师1.人物移动方式 = .跑

巫师1.当前坐标

巫师1.人物移动方式 = .传送

巫师1.当前坐标

//set 设置器，给人物坐标设置新值

巫师1.当前坐标 = 坐标(x: 3000, y: 2000)

巫师1.人物坐标

//属性监视器

//willSet：事前响应，新值newValue

//didSet：事后响应，旧值oldValue

class 经验{

var 总经验 = 0{

willSet {

print("当前总经验是:\(newValue)!")

}

didSet {

print("增加了\(总经验-oldValue)点经验")

}

}

}

var 经验1 = 经验()

经验1.总经验 = 1000

经验1.总经验 = 3000

//实例方法

enum Monster : Int{

case sheep = 10,

cat = 50,

snake = 100

}

class Player{

var exp = 0

func idle(){

exp += 500

print("挂机成功1次！总经验为\(exp)")

}

func kill(monster: Monster,expTime: Int){

exp += (monster.rawValue \* expTime)

print("打了一只\(monster),现在的经验值是\(exp)") }

}

var Jack = Player()

Jack.idle()

Jack.kill(monster: .snake, expTime: 3)

//类型方法，属于类型本身，无需实例化就可以使用，方法前加关键字static

class Player1{

static var nick = "John"

//static func server(){

class func server(){

print("\(nick)，您在江苏电信1区")

}

}

Player1.server()

//对于继承，父类类型方法加关键字class，可被子类重写(override)

class WuhanPlayer : Player1{

override static func server(){

print(nick,"您在湖北移动2区")

}

}

WuhanPlayer.server()

//初始化

class RoomTemp{

var season: String

var temp: Int

init(season: String, temp: Int) {

self.season = season

self.temp = temp

}

//便利构造器，实现便利的初始化

convenience init(){

self.init(season: "Winter", temp: 4)

}

}

let temp1 = RoomTemp(season: "Spring", temp: 22)

temp1.season

temp1.temp

let temp2 = RoomTemp()

temp2.season

temp2.temp

//可失败构造器：针对有可能的初始化失败，返回nil

struct Animal{

let name: String

init?(name: String){

if name.isEmpty {

print("没有给动物命名")

return nil

}

self.name = name

}

}

let cat = Animal(name:"")

//协议：方法、属性或一段功能的预定义

protocol AProtocol{

}

protocol BProtocol{

}

//遵从协议，多个协议之间可用","分隔

struct Astruct: AProtocol, BProtocol{

}

class Name{

}

class GivenName: Name, AProtocol{

}

//结构体、枚举就可以继承了一些功能。把功能抽取，变成小的功能，灵活。

//属性协议，要求遵从者实现以指定的名称实现属性，但具体实现是实例属性或类型属性并不关心

//可以指定要求实现getter或getter+setter，属性必须定义为变量，协议中的属性必须跟{get}或者{get set}

protocol FullName{

var fName: String{ get }

var gName: String{ get }

}

//遵守协议，用:

struct Student: FullName{

var gName: String

var fName: String

}

struct Teacher: FullName{

var gName: String

var fName: String

}

var student1 = Student(gName: "三丰", fName: "张")

student1.gName

student1.fName

class SomeBody: FullName{

var title: String? //optional，可选类型

var name: String

init(title: String?, name: String) {

self.title = title

self.name = name

}

var gName: String{

return name

}

var fName: String{

//title有值，返回值，为nil时，返回默认值“普通员工”

return title ?? "普通员工"

}

}

var somebody1 = SomeBody(title: nil, name: "李四")

somebody1.gName

somebody1.fName

//方法协议，定义时没有花括号执行体，可以通过协议将方法分解为更小的组合

//类型方法协议：前缀只能是static

protocol AMethod{

static func foo()

}

class A:AMethod{

static func foo() {

print("Hello!")

}

}

//实例方法协议，协议名称一般可以用形容词，用来描述协议能做什么

protocol RandomGeneratable{

//arc4random()生成随机数，返回值类型是UInt32

func randomNumber() -> Int

}

struct RandomNumber: RandomGeneratable{

func randomNumber() -> Int {

//arc4random\_uniform(6),返回值是0...5

return Int(arc4random())%6+1

}

}

struct RandomNumberInSix: RandomGeneratable{

func randomNumber() -> Int {

return Int(arc4random\_uniform(6))

}

}

let random1 = RandomNumber()

random1.randomNumber()

let random2 = RandomNumberInSix()

random2.randomNumber()

//协议作为类型使用：可用于参数类型/返回类型，变量/常量/属性，集合类型中的元素类型

struct Dice{

var sides: Int

//协议作为数据类型使用

var randomNumber: RandomGeneratable

func play() -> Int {

return self.randomNumber.randomNumber() % sides + 1

}

}

//randomNumber传递了遵守协议的结构体RandomNumber()

let aDice = Dice(sides: 6, randomNumber: RandomNumber())

aDice.play()

mutating主要用在Swift中的：结构体（struct）、枚举（enum）、扩展（extension）、协议（protocol）中修饰方法。

Swift结构体（struct）、枚举（enum） 中均可以包含类方法和实例方法。

在使用 class 来实现带有 mutating 的方法的协议时，方法前面是不需要加 mutating 修饰的，因为 class 可以随意更改自己的成员变量。因此在协议里用 mutating 修饰方法，对于 class 的实现是完全透明，可以当作不存在的。

//枚举的协议

protocol Switchable{

mutating func onOff()

}

//注意：在修改self的值时，要在func前加mutating

enum MySwitch: Switchable{

case on, off

mutating func onOff() {

switch self{

case .off:

self = .on

default:

self = .off

}

}

}

var myswitch = MySwitch.off

myswitch.onOff()

//把遵守相同协议的实例放到一个协议类型的数组

//数组中的元素都遵守协议CustomStringConvertible

//CustomStringConvertible控制台可打印

let array:[CustomStringConvertible] = [1,2,3,"hello"]

for element in array {

print(element)

}

//协议继承：一个协议可以继承若干个协议，并可以在继承的基础上增加新需求。与class继承相似，区别是class不能多重继承，对结构体进行扩展，相当于实现了多重继承（面向协议编程）

//建议将类的继承改为协议的继承使用，可以d

//继承的多个协议间用逗号分开

//CustomPlaygroundQuickLookable是playground可预览

protocol MyPrintable: CustomStringConvertible,CustomPlaygroundDisplayConvertible{

}

struct MyContent{

var text: String

var mycumtomtext: String

}

//提供默认实现，可以给协议扩展提供一个默认的实现

extension MyPrintable{

var customPlaygroundDescription: Any{

return "Playground的默认预览文字"

}

}

extension MyContent: MyPrintable{

var description: String{

return self.text

}

//var customPlaygroundDescription:Any{

// return self.mycustomtext

//}

}

let mycontent1 = MyContent(text: "内容", mycumtomtext: "保留文字")

mycontent1.description

//类专用协议：可以把协议限制在class类型（让结构体和枚举无法使用），加关键字class到协议继承列表的第一位

protocol OnlyForClass: class, CustomStringConvertible, CustomPlaygroundQuickLookable{

}

class Mytext: OnlyForClass{

var description: String{

return "des"

}

var customPlaygroundQuickLook: PlaygroundQuickLook{

return PlaygroundQuickLook.text("playground")

}

}

protocol OnlyForClass: AnyObject, CustomStringConvertible, CustomPlaygroundDisplayConvertible{

}

class Mytext: OnlyForClass{

var playgroundDescription: Any {

return "Playground 111"

}

var description: String{

return "des"

}

var customPlaygroundDescription: Any{

return "playground"

}

}

let mytext1 = Mytext()

mytext1.customPlaygroundDescription

print(mytext1)

//结构体继承上面的协议出现错误

//struct MyStruct: OnlyForClass{

//}

//协议组合：多个协议临时组合在一起的类型

//形式：协议1 & 协议2 &...

protocol Ageable{

var age: Int {get}

}

protocol Nameable{

var name: String{get}

}

//学生

struct Student1: Ageable, Nameable{

var age: Int

var name: String

}

//教师

struct Teacher1: Ageable,Nameable{

var age: Int

var name: String

var title: String

}

//将两个协议临时组合在一起，作为参数类型

func wish(someone: Ageable & Nameable){

print("祝",someone.name,someone.age,"岁生日快乐")

}

let stu1 = Student1(age: 10, name: "Jack")

wish(someone: stu1)

//协议检查和转换：使用is和as类型转换操作符来检查协议遵从与否，或转换成特定的协议

protocol Slogan{

var desc: String{get}

}

protocol Coder: Slogan{

var name: String{get set}

}

struct JavaCoder: Coder{

var name: String

var desc: String{

return "我会java"

}

}

struct JScoder: Coder{

var name: String

var desc: String{

return "我会jS"

}

}

struct Programmer{

var name: String

}

let java1 = JavaCoder(name: "Jack")

let js1 = JScoder(name: "John")

let programmer1 = Programmer(name: "Mike")

//数组中三个元素类型不一样，可以用 as [Any]

let coders = [java1, js1, programmer1] as [Any]

for coder in coders {

//if let语法，=右边必须是个Optional类型的变量

//在条件判断语句中，把Optional值直接给一个临时常量，swift会自动检测Optional是否包含值，如果包含值，会隐式的拆包并给那个临时常量，在接下来的上下文中就能直接使用这个临时常量

if let coder = coder as? Coder{

print(coder.name, coder.desc)

}else{

print("你不是一个程序员！")

}

if let pro = coder as? Programmer {

print("你是一个新手",pro.name)

}

}

**1、HelloWord**

import UIKit

class ViewController: UIViewController {

@IBAction func touch(\_ sender: UIButton) {

print("Hello World!")

//创建基础UIAlertController

let ac = UIAlertController(title: "提示", message: "您点击了按钮", preferredStyle: .alert)

//创建动作按钮

//style的参数可选：常规(default)、取消(cancel)以及警示(destruective)

let btn1 = UIAlertAction(title: "好", style: .default, handler: nil)

let btn2 = UIAlertAction(title: "取消", style: .destructive, handler: nil)

//动作按钮添加到控制器上

ac.addAction(btn1)

ac.addAction(btn2)

//显示对话框视图控制器

self.present(ac, animated: true, completion: nil)

}

}

**1、计算器**

import UIKit

class ViewController:UIViewController{

//按住control键，将控件连线至该位置，属性选择outlet

@IBOutlet weak var num1: UITextField!

@IBOutlet weak var num2: UITextField!

@IBOutlet weak var sum: UILabel!

//方法选择action

@IBAction func calculate(\_ sender: UIButton) {

//文本框中的内容为String类型，转为Double类型

//文本框中的数据类型为String?,需要显式拆包

let i1 = Double(num1.text!)

let i2 = Double(num2.text!)

//计算结果在文本框中输出，需转为String类型

sum.text = String(i1! + i2!)

}

}

**2、图片浏览器**

import UIKit

class ViewController: UIViewController {

//定义球员姓名数组

let allDescs = ["奥尼尔","保罗","基德","吉诺比利","加内特","科比","孙悦","詹姆斯"]

//控件连线outlet，定义编号

@IBOutlet weak var imageNo: UILabel!

//控件连线outlet，定义图片

@IBOutlet weak var imageView: UIImageView!

//控件连线outlet，定义描述

@IBOutlet weak var iamgeDesc: UILabel!

//控件连线action，定义滑动条方法

@IBAction func sliderValueChange(\_ sender: UISlider) {

//取出滑动条的值，并转为整型

let no = Int(sender.value)

//设置图片编号

imageNo.text = String(no) + "/8"

//设置球员姓名

iamgeDesc.text = allDescs[no-1]

//设置图像

imageView.image = UIImage(named: String(no))

}

}