//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)

}

}