访问控制

@M了个J

https://github.com/CoderMJLee

http://cnblogs.com/mjios

码拉松

访问控制（Access Control）

- 在访问权限控制这块，Swift提供了5个不同的访问级别（以下是从高到低排列， 实体指被访问级别修饰的内容）
  - open：允许在定义实体的模块、其他模块中访问，允许其他模块进行继承、重写（open只能用在类、类成员上）
  - public：允许在定义实体的模块、其他模块中访问，不允许其他模块进行继承、重写
  - internal：只允许在定义实体的模块中访问，不允许在其他模块中访问
  - fileprivate：只允许在定义实体的源文件中访问
  - private：只允许在定义实体的封闭声明中访问

- 绝大部分实体默认都是 internal 级别

访问级别的使用准则

- 一个实体不可以被更低访问级别的实体定义，比如
  - 变量\常量类型 ≥ 变量\常量
  - 参数类型、返回值类型 ≥ 函数
  - 父类 ≥ 子类
  - 父协议 ≥ 子协议
  - 原类型 ≥ `typealias`
  - 原始值类型、关联值类型 ≥ 枚举类型
  - 定义类型A时用到的其他类型 ≥ 类型A
  - ……

元组类型

- 元组类型的访问级别是所有成员类型最低的那个

```
internal struct Dog {}
fileprivate class Person {}

// (Dog, Person)的访问级别是fileprivate
fileprivate var data1: (Dog, Person)
private var data2: (Dog, Person)
```

泛型类型

- 泛型类型的访问级别是 类型的访问级别 以及 所有泛型类型参数的访问级别 中最低的那个

```
internal class Car {}
fileprivate class Dog {}
public class Person<T1, T2> {}

// Person<Car, Dog>的访问级别是fileprivate
fileprivate var p = Person<Car, Dog>()
```

成员、嵌套类型

- 类型的访问级别会影响成员（属性、方法、初始化器、下标）、嵌套类型的默认访问级别
- 一般情况下，类型为private或fileprivate，那么成员\嵌套类型默认也是private或fileprivate
- 一般情况下，类型为internal或public，那么成员\嵌套类型默认是internal

```
public class PublicClass {
    public var p1 = 0 // public
    var p2 = 0 // internal
    fileprivate func f1() {} // fileprivate
    private func f2() {} // private
}

class InternalClass { // internal
    var p = 0 // internal
    fileprivate func f1() {} // fileprivate
    private func f2() {} // private
}
```

```
fileprivate class FilePrivateClass { // fileprivate
    func f1() {} // fileprivate
    private func f2() {} // private
}

private class PrivateClass { // private
    func f() {} // private
}
```

小码哥教育 SEEMYGO

成员的重写

- 子类重写成员的访问级别必须 ≥ 子类的访问级别，或者 ≥ 父类被重写成员的访问级别

- 父类的成员不能被成员作用域外定义的子类重写

```
public class Person {
    private var age: Int = 0
}
public class Student : Person {
    override var age: Int {
        set {}
        get {10}
    }
}
```

```
public class Person {
    private var age: Int = 0

    public class Student : Person {
        override var age: Int {
            set {}
            get {10}
        }
    }
}
```

小码哥教育 SEEMYGO

下面代码能否编译通过？

```
private class Person {}
fileprivate class Student : Person {}
```

```
private struct Dog {
    var age: Int = 0
    func run() {}
}

fileprivate struct Person {
    var dog: Dog = Dog()
    mutating func walk() {
        dog.run()
        dog.age = 1
    }
}
```

```
private struct Dog {
    private var age: Int = 0
    private func run() {}
}

fileprivate struct Person {
    var dog: Dog = Dog()
    mutating func walk() {
        dog.run()
        dog.age = 1
    }
}
```

- 直接在全局作用域下定义的private等价于fileprivate

getter、setter

- getter、setter默认自动接收它们所属环境的访问级别
- 可以给setter单独设置一个比getter更低的访问级别，用以限制写的权限

```
fileprivate(set) public var num = 10
class Person {
    private(set) var age = 0
    fileprivate(set) public var weight: Int {
        set {}
        get { 10 }
    }
    internal(set) public subscript(index: Int) -> Int {
        set {}
        get { index }
    }
}
```

初始化器

- 如果一个public类想在另一个模块调用编译生成的默认无参初始化器，必须显式提供public的无参初始化器
  - 因为public类的默认初始化器是internal级别

- required初始化器 ≥ 它的默认访问级别

- 如果结构体有private\fileprivate的存储实例属性，那么它的成员初始化器也是private\fileprivate
  - 否则默认就是internal

枚举类型的case

- 不能给enum的每个case单独设置访问级别

- 每个case自动接收enum的访问级别
  - public enum定义的case也是public

协议

- 协议中定义的要求自动接收协议的访问级别，不能单独设置访问级别
  - public协议定义的要求也是public

- 协议实现的访问级别必须 ≥ 类型的访问级别，或者 ≥ 协议的访问级别

- 下面代码能编译通过么？

```
public protocol Runnable {
    func run()
}

public class Person : Runnable {
    func run() {}
}
```

小码哥教育 SEEMYGO

扩展

- 如果有显式设置扩展的访问级别，扩展添加的成员自动接收扩展的访问级别
- 如果没有显式设置扩展的访问级别，扩展添加的成员的默认访问级别，跟直接在类型中定义的成员一样
- 可以单独给扩展添加的成员设置访问级别
- 不能给用于遵守协议的扩展显式设置扩展的访问级别

小码哥教育 SEEMYGO

扩展

- 在同一文件中的扩展，可以写成类似多个部分的类型声明
- 在原本的声明中声明一个私有成员，可以在同一文件的扩展中访问它
- 在扩展中声明一个私有成员，可以在同一文件的其他扩展中、原本声明中访问它

```
public class Person {
    private func run0() {}
    private func eat0() {
        run1()
    }
}
```

```
extension Person {
    private func run1() {}
    private func eat1() {
        run0()
    }
}
```

```
extension Person {
    private func eat2() {
        run1()
    }
}
```

将方法赋值给var\let

- 方法也可以像函数那样，赋值给一个let或者var

```
struct Person {
    var age: Int
    func run(_ v: Int) { print("func run", age, v) }
    static func run(_ v: Int) { print("static func run", v) }
}
```

```
let fn1 = Person.run
fn1(10) // static func run 10

let fn2: (Int) -> () = Person.run
fn2(20) // static func run 20

let fn3: (Person) -> ((Int) -> ()) = Person.run
fn3(Person(age: 18))(30) // func run 18 30
```