Swift程序设计实践

Swift Programming Experiments

张 亮 | 计算机学院 | 北京航空航天大学

属性

存储属性

计算属性

属性观察器

类型属性

属性

属性的定义

类、结构体和枚举型都可以有属性。

属性将值和类型进行了关联。

属性种类

属性分为存储属性、计算属性和类型属性。

存储属性通过常量或者变量来存储实例的值。

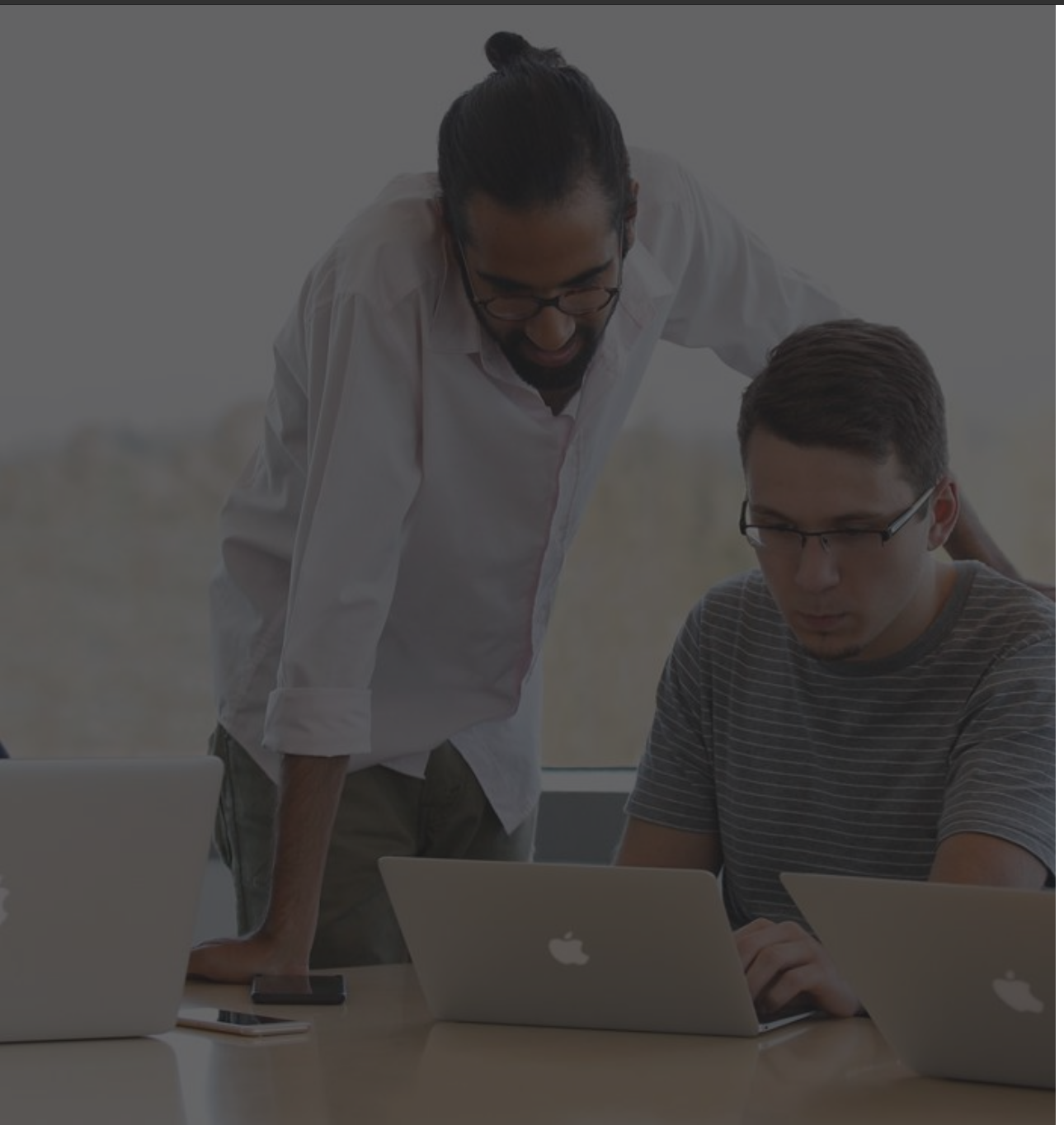
计算属性是用来计算值的。

类型属性是属于类型本身的，而不属于特定的实例。

而存储属性和计算属性与具体的实例相关联。

属性观察器

用来监听属性值的变化，并由此来触发特定的操作。



存储属性

常量存储属性

变量存储属性

常量实例的存储属性

延迟存储属性

存储属性

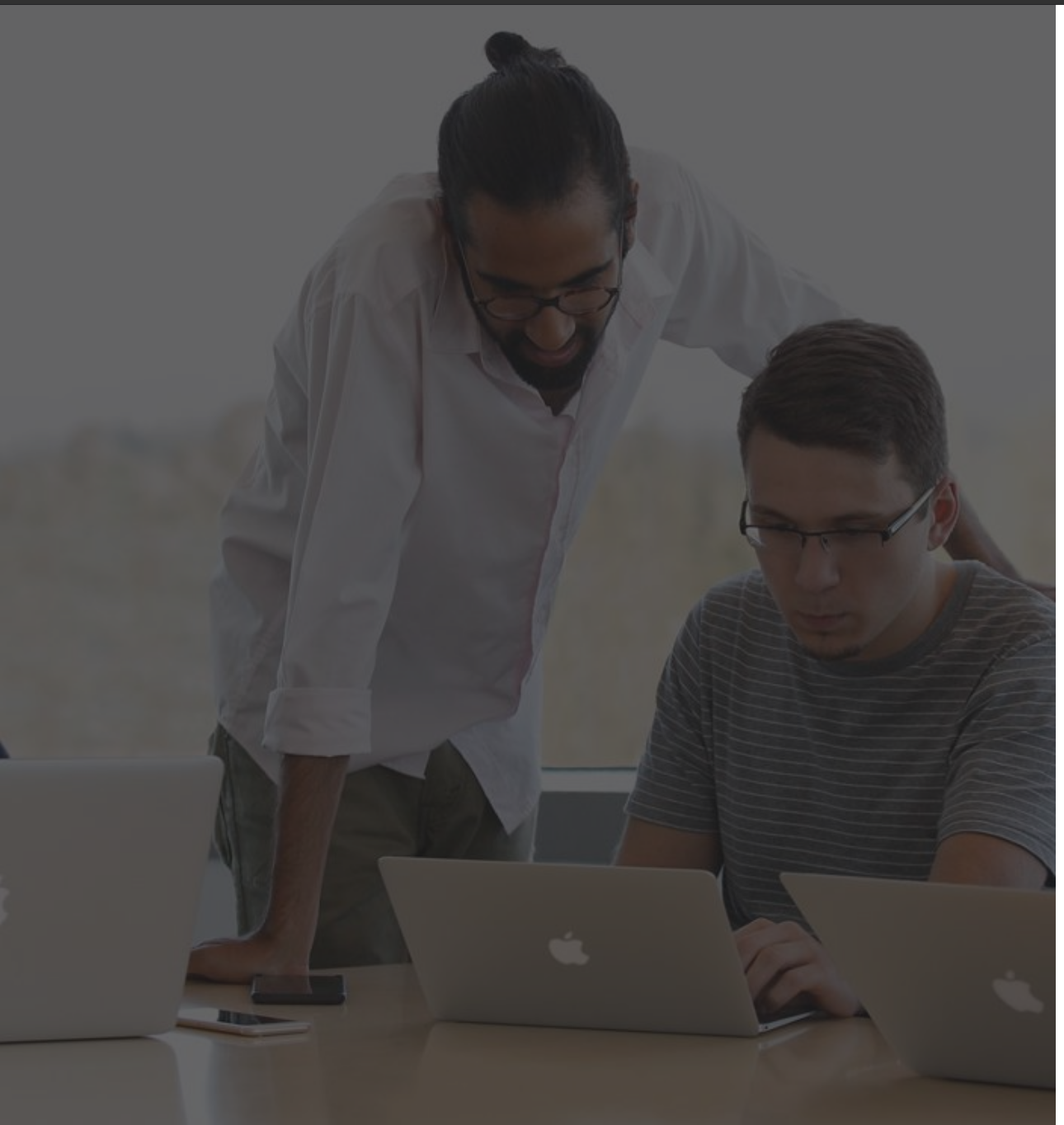
存储属性就是存储在一个类或一个结构体中的变量或常量，既可以在定义存储属性的时候赋一个默认值，也可以在构造器中设置或修改存储属性的值。

例子

```
struct User {
    let id : Int
    var name : String
    var password : String
    var email : String
}

var theUser = User(id: 16, name: "Tommy", password:
    "bhq963", email: "tommy@gmail.com")

theUser.id = 19    Cannot assign to property: 'id' is a 'let' constant
```



变量存储属性

修改实例中的变量存储属性是允许的

例子

```
theUser.name = "pennie"
theUser.password = "5263tt"
theUser.email = "pennie@gmail.com"
```

结构体：常量实例的存储属性

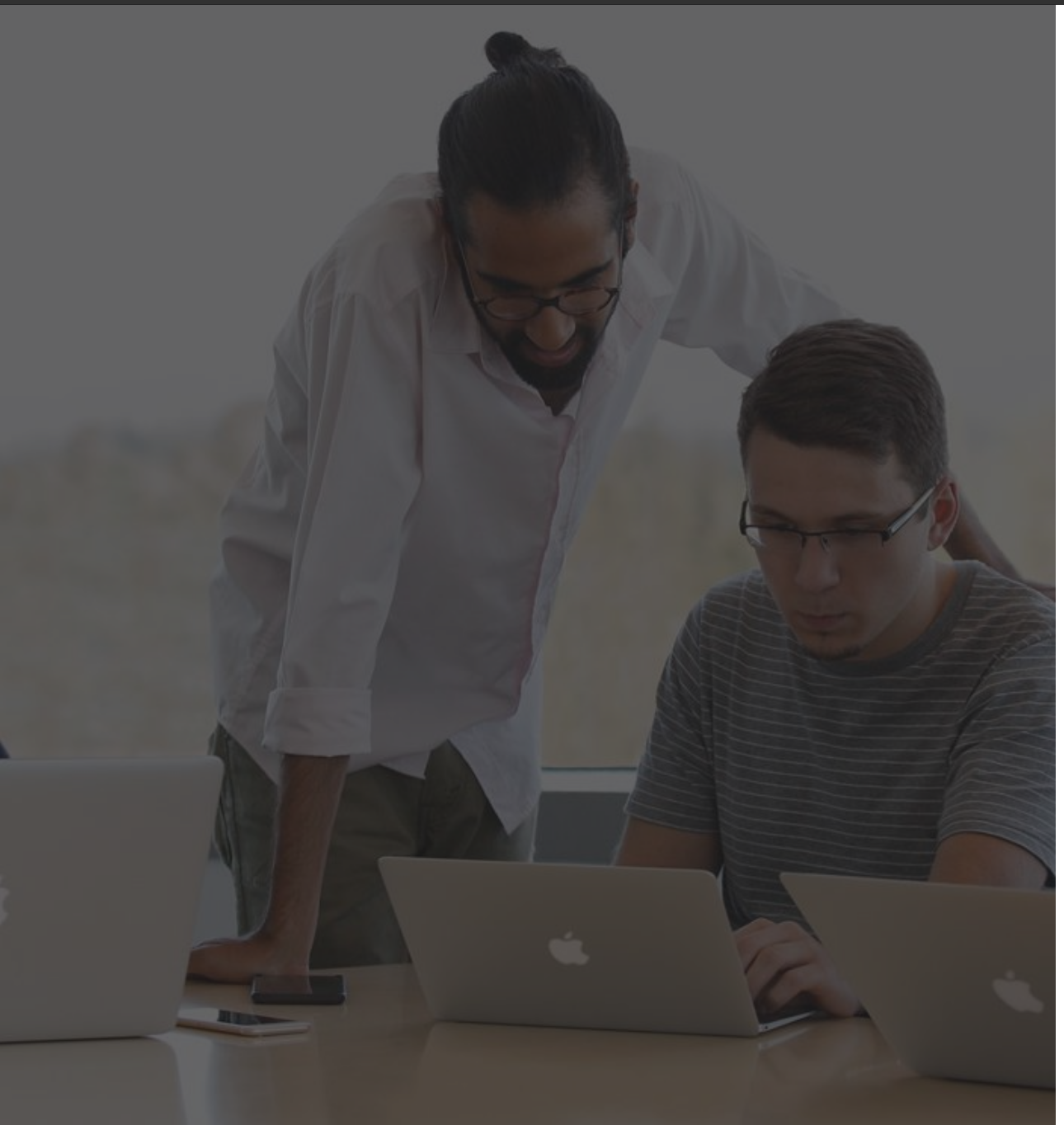
如果将结构体实例赋值给一个常量，那么该实例中的存储属性（包括常量和变量存储属性）的值都不能被改变。

这一条不适用于类（为什么？）

例子

```
let anotherUser = User(id: 17, name: "Sammy", password:
    "urs33", email: "sammy@gmail.com")
```

```
anotherUser.id = 18 ⓪ Cannot assign to property: 'id' is a 'let' constant
anotherUser.name = "sam" ⓪ Cannot assign to property: 'anotherUse...
anotherUser.password = "newp" ⓪ Cannot assign to property: 'anot...
anotherUser.email = "new@gmail.com" ⓪ Cannot assign to propert...
```



延迟存储属性

定义延迟存储属性

在存储属性声明前加上一个关键字lazy，则表示该属性为延迟存储属性。

必须声明为变量

延迟存储属性在实例第一次被调用的时候才会计算初始值。因此，延迟存储属性必须声明为变量。而常量属性在实例初始化完成后必须有值。

应用场景

当一个属性在初始化时需要占用大量系统资源（时间或空间）时，就可以声明为延迟属性。

当一个属性的值依赖于实例初始化完成后的外部数据时，也将其声明为延迟属性。

例子

```
class User {
    var id = 0
    var name = ""
    var password = ""
    var email = ""
    lazy var image = ImageInfo()
}

struct ImageInfo {
    var name = "default name"
    var path = "default path"
}

var theUser = User()
theUser.id = 18
theUser.name = "sammy"
```

```
id 18
name "sammy"
password ""
email ""
nil
```

```
print("the name of image is \(theUser.image.name)")
```

```
the name of image is default
name...
```

延迟存储属性

定义延迟存储属性

在存储属性声明前加上一个关键字lazy，则表示该属性为延迟存储属性。

必须声明为变量

延迟存储属性在实例第一次被调用的时候才会计算初始值。因此，延迟存储属性必须声明为变量。而常量属性在实例初始化完成后必须有值。

应用场景

当一个属性在初始化时需要占用大量系统资源（时间或空间）时，就可以声明为延迟属性。

当一个属性的值依赖于实例初始化完成后的外部数据时，也将其声明为延迟属性。

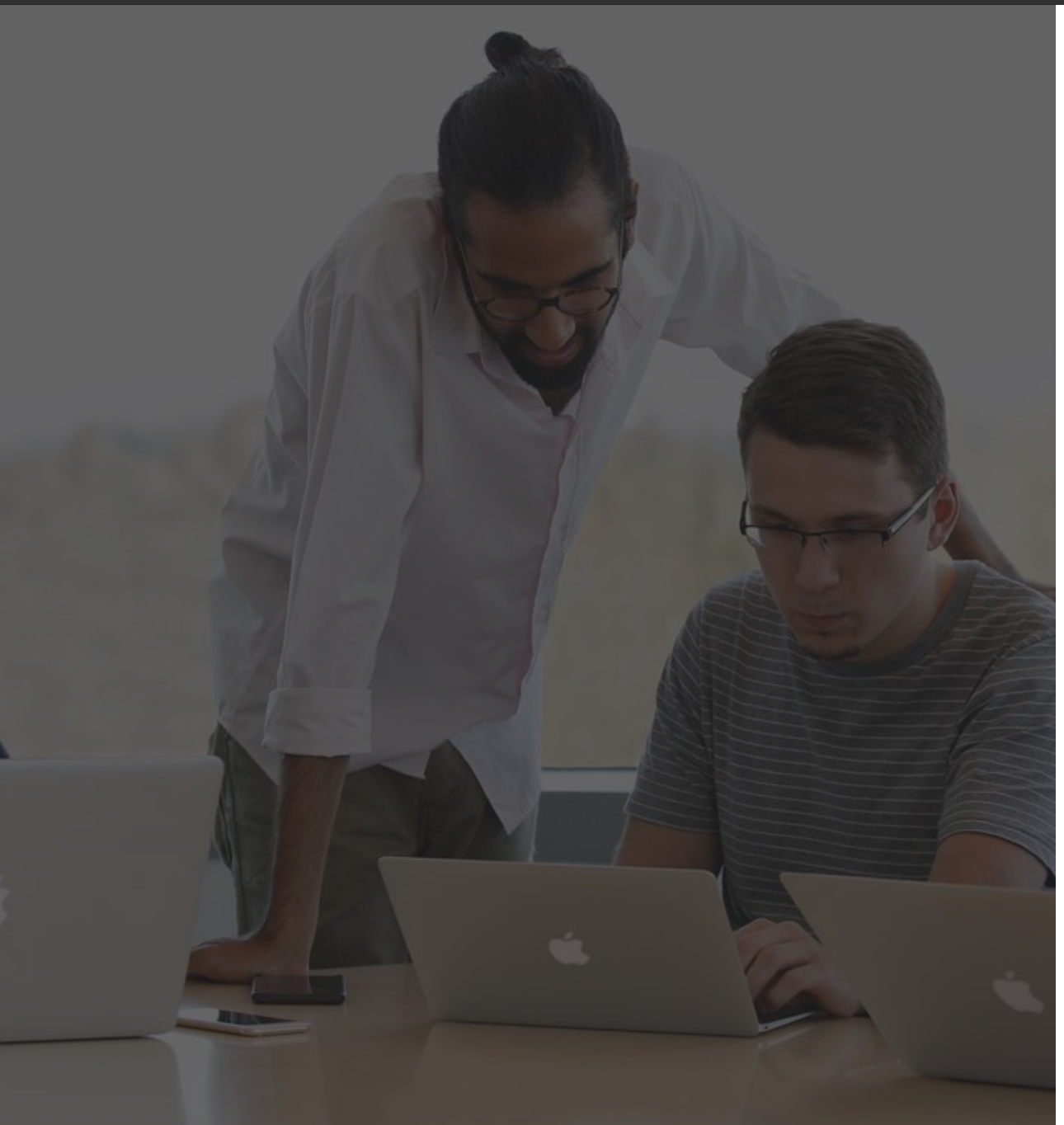
计算属性

计算属性需要经过计算后，才能返回属性值。

存储属性可以为常量或变量，而计算属性只能为变量。

计算属性是Swift中特有的一种属性，它不直接存储值，而是通过get和set方法来获取和设置值。

类、结构体和枚举型都可以定义计算属性。



计算属性：get方法

通过计算属性的get方法来读取属性的值。

例子

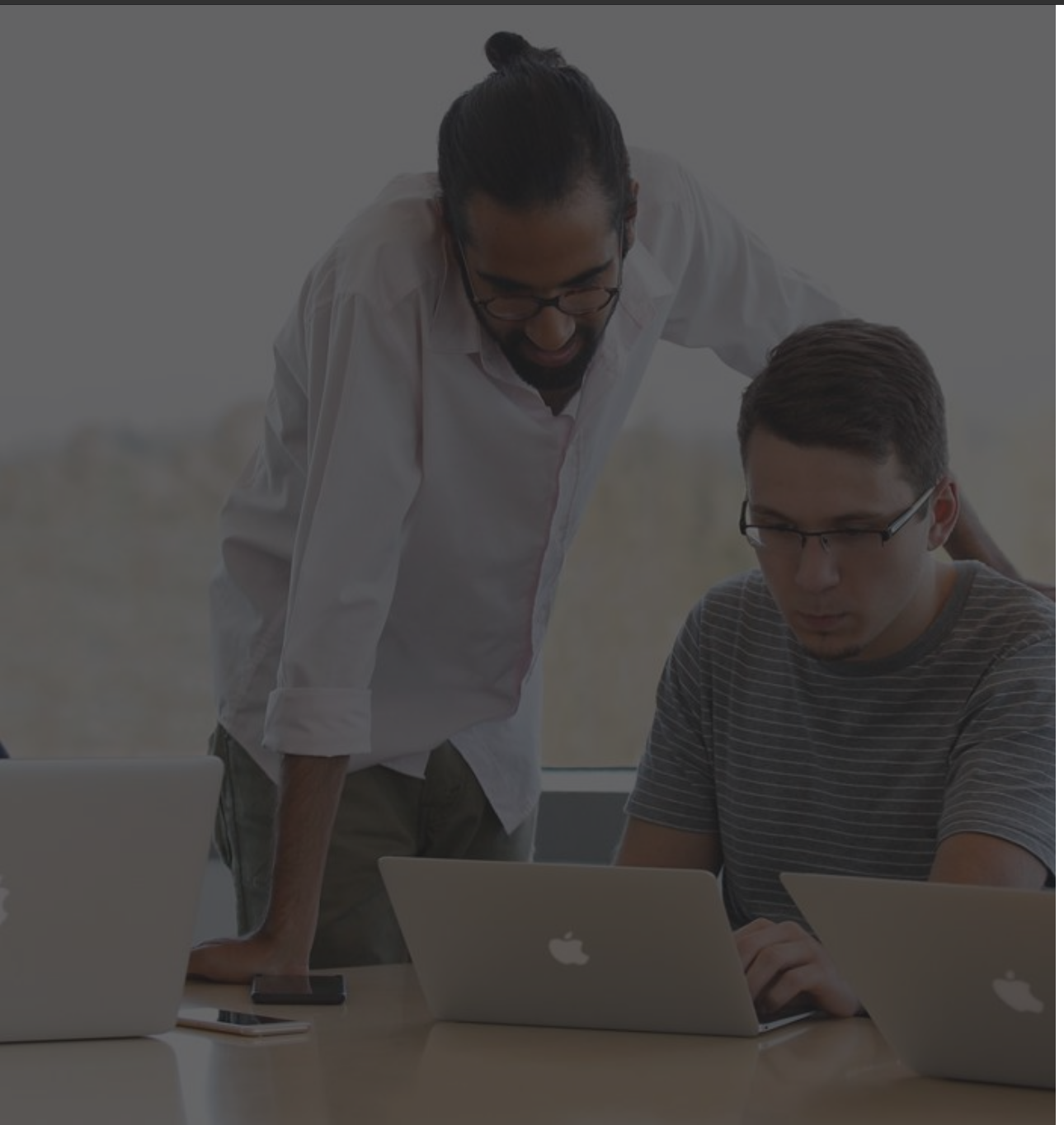
```
struct ImageInfo {
    var name = "default name"
    var path = "default path"
}
class User {
    var id = 0
    var name = ""
    var password = ""
    var email = ""
    lazy var image = ImageInfo()
    var workingYear = 0
    var holiday : Int {
        get {
            var days : Int
            switch workingYear {
                case 0: days = 5
                case 1...5: days = 5 + workingYear
                default: days = 12
            }
            return days
        }
    }
}
var theUser = User()
theUser.name = "Tony"
theUser.workingYear = 3
print("Tony has worked for \(theUser.workingYear) years and he has holiday:
    \(theUser.holiday) days")
```

计算属性：set方法

计算属性的set方法可以根据计算属性的值来设置其他相关存储属性的值。

注意：计算属性的set方法是没有返回值的。

```
var holiday : Int {
    get {
        var days : Int
        switch workingYear {
            case 0: days = 5
            case 1...5: days = 5 + workingYear
            default: days = 12
        }
        return days
    }
    set {
        switch newValue {
            case 5: workingYear = 0
            case 6...11: workingYear = newValue -
                5
            default: workingYear = 6
        }
    }
}
```



属性观察器

监听属性值的变化

Swift提供了一种叫做属性观察器的机制来监控属性值的变化。

每当要改变一个属性值的之前和之后，都会触发属性观察器。

使用范围

除了延迟存储属性，所有其他属性都可以增加一个属性观察器，对其值的变化进行监控。

属性观察器的方法

属性观察器有两个方法：willSet和didSet。

方法willSet在属性的值被改变前触发或被调用。

方法willSet会将新值作为常量参数传入，缺省名称为newValue，也可以自定义参数名称。

方法didSet在属性的值被改变后触发或被调用。

方法didSet则会将旧值作为参数传入oldValue，同样也可以自定义参数名称。

例子

```
3  class Website {
4      var domain : String = ""
5      var maxClicks = 10000
6      var clicks : Int = 0 {
7          willSet(newClicks){
8              print("clicks will be set to \(newClicks)")
9          }
10         didSet {
11             if clicks > maxClicks {
12                 print("\(clicks) is too high. Fall Back to
                       \(oldValue)")
13                 clicks = oldValue
14             } else {
15                 print("did set clicks from \(oldValue) to
                       \(clicks)")
16             }
17         }
18     }
19 }
20
21 let theWebsite = Website()
22 theWebsite.domain = "www.buaa.edu.cn"
23 theWebsite.clicks = 100
24 theWebsite.clicks = 200
25 theWebsite.clicks = 20000
```

```
clicks will be set to 100
did set clicks from 0 to 100
clicks will be set to 200
did set clicks from 100 to 200
clicks will be set to 20000
20000 is too high. Fall Back to 200
```

属性观察器

监控属性

Swift提供了一种叫做属性观察器的机制来监控属性值的变化。

每当要改变一个属性值的之前和之后，都会触发属性观察器。

使用范围

除了延迟存储属性，所有其他属性都可以增加一个属性观察器，对其值的变化进行监控。

属性观察器的方法

属性观察器有两个方法：willSet和didSet。

方法willSet在属性的值被改变前触发或被调用。

方法willSet会将新值作为常量参数传入，缺省名称为newValue，也可以自定义参数名称。

方法didSet在属性的值被改变后触发或被调用。

方法didSet则会将旧值作为参数传入oldValue，同样也可以自定义参数名称。

类型属性

应用场景

每次类型实例化后，每个实例都拥有自己独立的属性值。如果要让某个类型的所有实例都共享同一个属性的话，就需要引入类型属性的概念。

定义

类型属性是用来表示一个类型的所有实例都共享的属性。

类型属性使用关键字static来标识。

作用域

类型属性作为类型定义的一部分，它的作用域为类型的内部。

用法

跟实例属性一样，类型属性的访问也是通过点号运算符来进行的。但是，类型属性只能通过类型本身来获取和修改，不能通过实例来访问。

类型属性

用途

每次类型实例化后，每个实例都拥有自己独立的属性值。如果要让某个类型的所有实例都共享同一个属性的话，就需要引入类型属性的概念。

定义

类型属性是用来表示一个类型的所有实例都共享的属性。

类型属性使用关键字static来标识。

作用域

类型属性作为类型定义的一部分，它的作用域为类型的内部。

用法

跟实例属性一样，类型属性的访问也是通过点号运算符来进行的。但是，类型属性只能通过类型本身来获取和修改，不能通过实例来访问。

例子

```
3  class Visitor {
4      var name : String = ""
5      var stayTime : Int = 0
6      static var permission : String = "visitor"
7  }
8
9  let theVisitor = Visitor()
10 theVisitor.name = "Tom"
11 theVisitor.stayTime = 5
12 print("Current permission is \(Visitor.permission)")
13 let anotherVisitor = Visitor()
14 anotherVisitor.name = "Sam"
15 anotherVisitor.stayTime = 9
16 Visitor.permission = "administrator"
17 print("Now permission is \(Visitor.permission)")
```

```
Current permission is visitor
Now permission is administrator
```

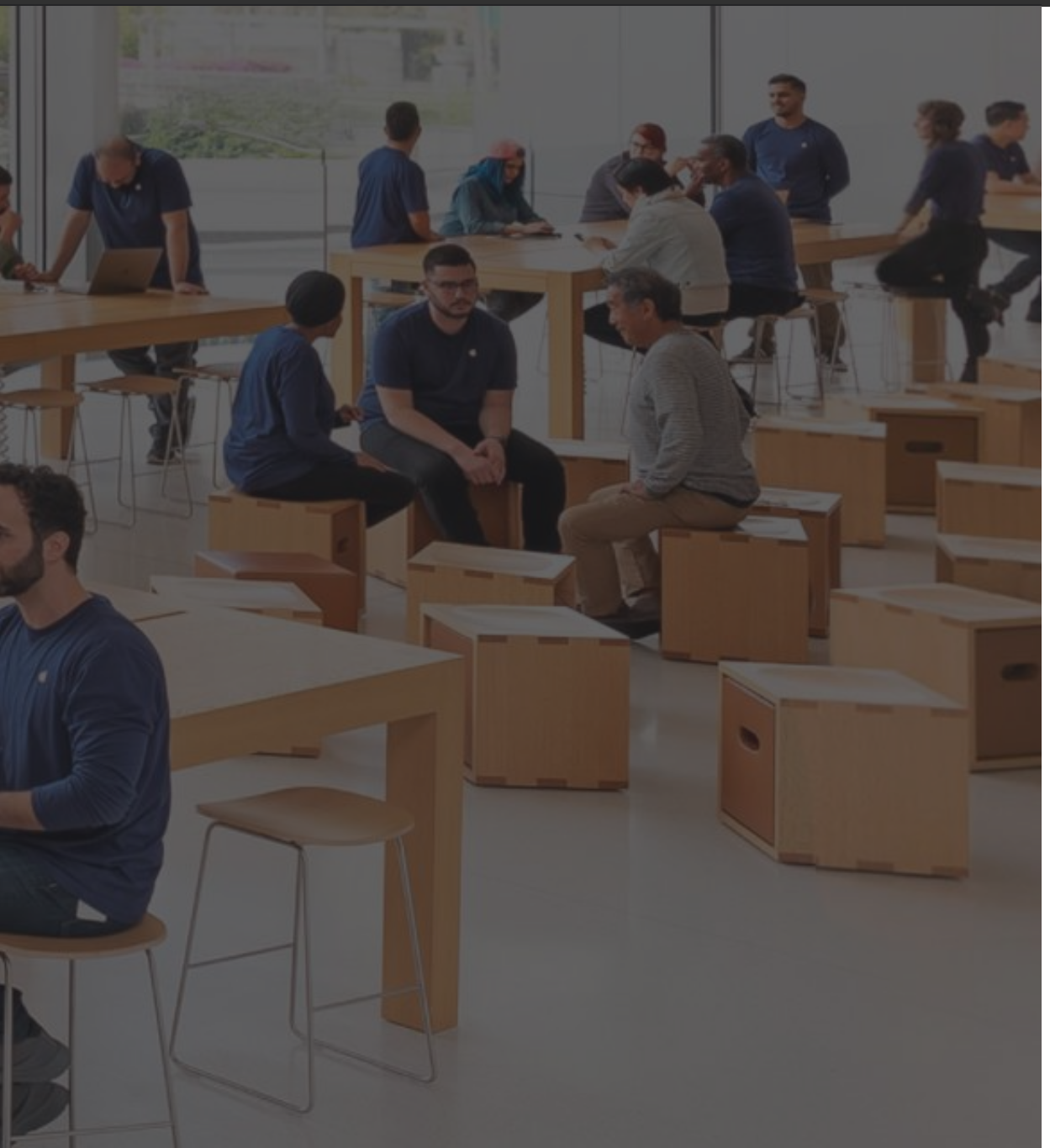
方法

实例方法

类型方法

可变方法

下标方法



方法

方法是类、结构体或枚举中定义具体任务或功能的函数。

方法分为实例方法和类型方法。

实例方法与实例相关联，而类型方法与类型本身相关联，和具体实例无关。

实例方法的定义和调用

定义

实例方法指类、结构体或枚举类型的实例的方法。实例方法可以访问和修改实例的属性，实现特定的功能。

声明方式

实例方法的声明方式和函数完全一致。实例方法的定义要写在类型定义的花括号内。

用法

实例方法可以隐式的访问属于同一个类型的其它实例方法和属性。实例方法只能被实例来调用。

实例方法的定义和调用

定义

实例方法指类、结构体或枚举类型的实例的方法。实例方法可以访问和修改实例的属性，实现特定的功能。

声明方式

实例方法的声明方式和函数完全一致。实例方法的定义要写在类型定义的花括号内。

用法

实例方法可以隐式的访问属于同一个类型的其它实例方法和属性。实例方法只能被一个实例来调用。

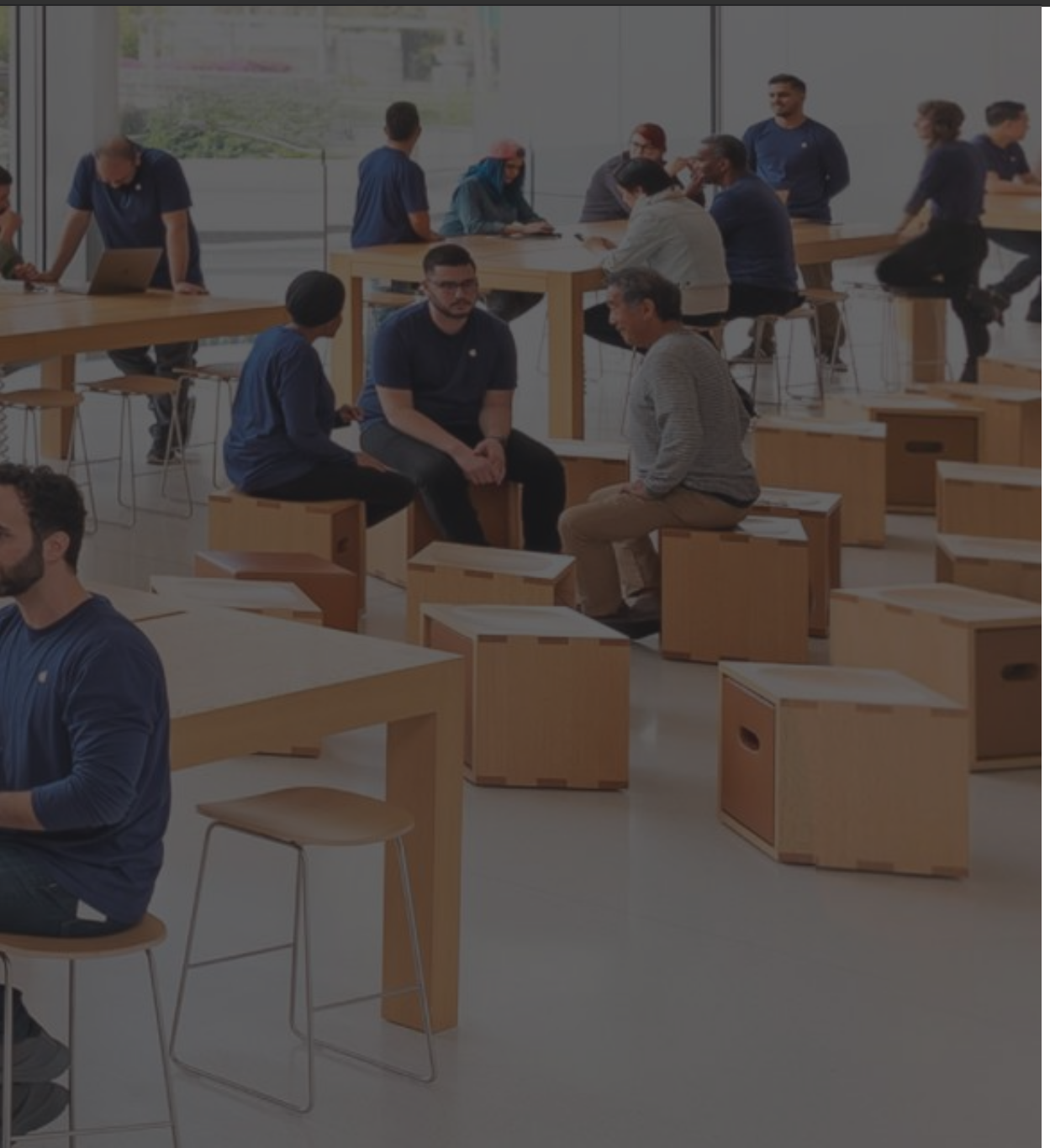
例子

```
class Website {
    var visitCount = 0
    func visiting(){
        ++visitCount
    }
}

let sina = Website()
sina.visitCount
sina.visiting()
sina.visitCount
sina.visiting()
sina.visitCount
```

```
(2 times)

Website
0
Website
1
Website
2
```



带参数的方法

方法既可以没有参数，也可以有一个或多个参数。

注意

从Swift 4开始，去掉了局部参数名和外部参数名的概念，将两者统一为方法的参数名。

参数名不仅可以在方法内访问，也可以在外部调用方法时作为提示参数输入的信息出现。

隐式属性self

每个实例都有一个隐式的属性self，表示这个实例本身。在实例的方法中可以通过self来引用实例自己。

例子

```
class Website {
    var visitCount = 0
    var visitor = [String]()
    var visitDate = ""
    func visiting(visitor: String, visitDate : String){
        visitCount += 1
        self.visitor.append(visitor)
        self.visitDate = visitDate
    }
}

let sina = Website()
sina.visiting(visitor: "Tommy", visitDate: "2016-6-1")
sina.visitCount
sina.visitor
sina.visitDate
```

带参数的方法

方法既可以没有参数，也可以有一个或多个参数。

注意

从Swift 4开始，去掉了局部参数名和外部参数名的概念，将两者统一为方法的参数名。

参数名不仅可以在方法内访问，也可以在外部调用方法时作为提示参数输入的信息出现。

隐式属性self

每个实例都有一个隐式的属性self，表示这个实例本身。在实例的方法中可以通过self来引用实例自己。

类型方法

定义

类型方法是只能由类型本身调用的方法。

声明方式

在类、结构体和枚举中声明类型方法是通过在方法的前面加上关键字static。

类型方法 VS 实例方法

相同之处：

类型方法和实例方法一样都是采用点号语法来进行调用。

不同之处：

类型方法只能由类型本身对其调用。

实例方法只能由实例对其调用。

在类型方法中，self指向类型本身。

在实例方法中，self指向实例本身。

类型方法

定义

类型方法是只能由类型本身调用的方法。

声明方式

在类、结构体和枚举中声明类型方法是通过在方法的前面加上关键字static。

在类中可以使用关键字class来实现子类重写父类的方法。

类型方法与实例方法的差别

类型方法和实例方法一样都是采用点语法来进行调用，不同之处：

类型方法是类型本身对该方法的调用。

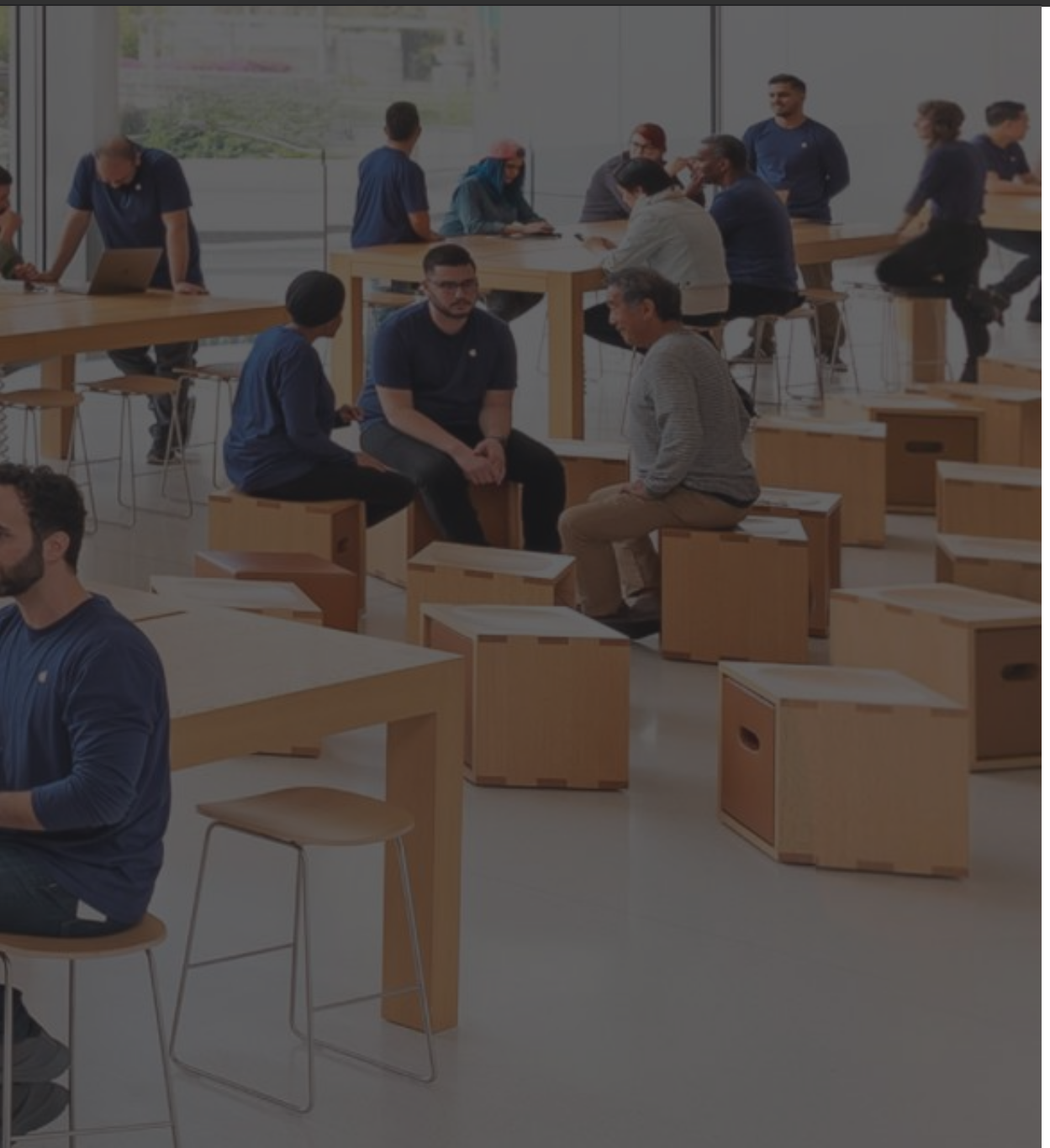
实例方法只能是实例对该方法的调用。

在类型方法中，self指向类型本身，而不是实例。

例子

```
class Website {
    static var visitCount = 0
    static var visitor = [String]()
    static var visitDate = ""
    static func visiting(visitor:String, visitDate : String){
        visitCount += 1
        self.visitor.append(visitor)
        self.visitDate = visitDate
    }
}

Website.visiting(visitor:"Tommy", visitDate: "2016-6-1")
Website.visitCount
Website.visitor
Website.visitDate
```



可变方法

在结构体中修改属性

在类Website中，可以通过实例方法或者类型方法来修改属性的值。如果将Website改写为结构体，编译器则会报错，提示结构体的方法不能修改属性。

可变方法mutating

如果要使结构体中的方法能够修改属性值，就必须在方法名前面加上关键字“mutating”。

可变方法

示例1

```
struct Website {
    var visitCount = 0
    func visiting() {
        visitCount += 1
    }
}
```

Left side of mutating operator isn't mutable: 'self' is immutable

Mark method 'mutating' to make 'self' mutable Fix

示例2

```
struct Website {
    var visitCount = 0
    mutating func visiting() {
        visitCount += 1
    }
}
var sohu = Website()
sohu.visiting()
print("\(sohu.visitCount)")
```

Website
Website
"1\n"

结构体中方法修改属性

在类Website中，可以通过实例方法或者类型方法来修改属性的值。如果将Website改写为结构体，编译器则会报错，提示结构体的方法不能修改属性。

可变方法mutating

如果要使结构体中的方法能够修改属性值，就必须在方法名前面加上关键字“mutating”。

下标方法

下标的定义

下标是一种通过下标的索引来获取值的快捷方法。

典型例子：数组中，使用下标来进行数组元素的读写，例如：Array[Index]。

自定义下标

在类、结构体和枚举类型中，可以自定义下标，从而实现对实例属性的赋值和访问。

自定义的下标可以有多种索引值类型，来实现按照不同索引进行实例属性的赋值和访问。

自定义下标的方式

自定义一个下标要使用关键字subscript，显式的声明一个或多个传入参数和返回类型。

自定义下标通过set和get方法的定义，来实现读写或者只读。

下标方法

下标的定义

下标是一种通过下标的索引来获取值的快捷方法。

典型例子：数组中，使用下标来进行数组元素的读写，例如：Array[Index]。

自定义下标

在类、结构体和枚举类型中，可以自定义下标，从而实现对实例属性的赋值和访问。

自定义的下标可以有多种索引值类型，来实现按照不同索引进行实例属性的赋值和访问。

自定义下标的方式

自定义一个下标要使用关键字subscript，显式的声明一个或多个传入参数和返回类型。

自定义下标通过set和get方法的定义，来实现读写或者只读。

格式

```
subscript ( index : DataType) -> DataType{
        get {
            // return index
        }
        set ( newValue ) {
            // set new value
        }
}
```

下标方法

下标的定义

下标是一种通过下标的索引来获取值的快捷方法。

典型例子：数组中，使用下标来进行数组元素的读写，例如：Array[Index]。

自定义下标

在类、结构体和枚举类型中，可以自定义下标，从而实现对实例属性的赋值和访问。

自定义的下标可以有多种索引值类型，来实现按照不同索引进行实例属性的赋值和访问。

自定义下标的方式

自定义一个下标要使用关键字subscript，显式的声明一个或多个传入参数和返回类型。

自定义下标通过set和get方法的定义，来实现读写或者只读。

例子

```
class Website {
     var visitCount = 0
     var visitor = [String]()
     var visitDate = ""
     func visiting(visitor visitor:String,
         visitDate : String){
        ++visitCount
        self.visitor.append(visitor)
        self.visitDate = visitDate
    }
    subscript(index : Int) -> String {
        get {
            return visitor[index]
        }
        set {
            visitor[index] = newValue
        }
    }
}

var sina = Website()
sina.visiting(visitor: "Tom", visitDate:
    "2016-6-3")
sina.visiting(visitor: "Sam", visitDate:
    "2016-6-9")
print("\(sina[0])")
sina[2] = "Pennie"
print("\(sina[2])")
```

```
(2 times)
(2 times)
(2 times)

"Tom"

"Pennie"

Website
Website

Website

"Tom\n"
"Pennie"
```

高级运算符

位运算符

溢出运算符

运算符函数

位运算符

位运算符指对一个数据中的每个位进行操作：按位取反、按位与、按位或、按位异或、按位左移和右移

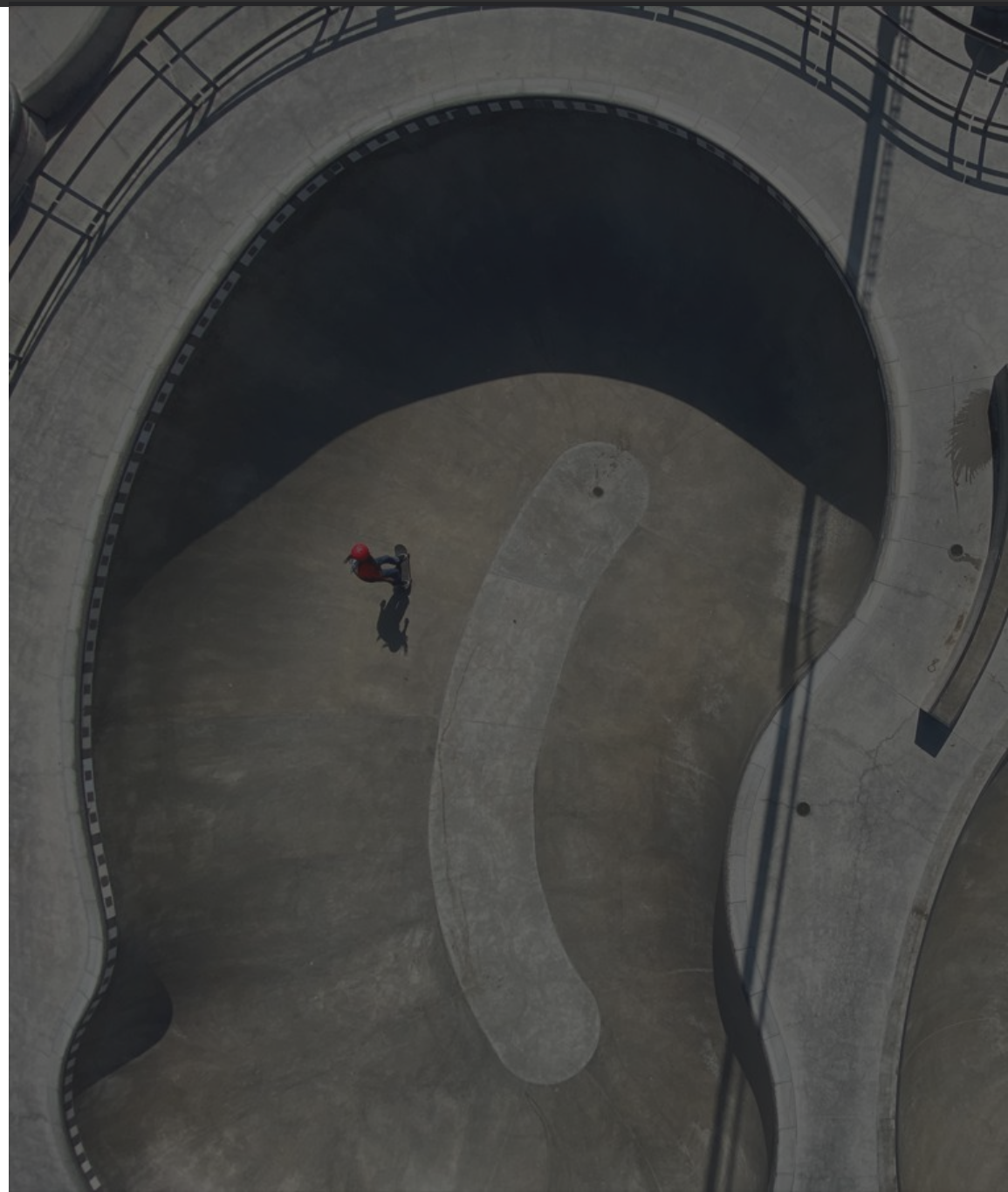
```
var origin : UInt8 = 0b00001111          15
var result = ~origin // = 0b11110000     240

origin = 240                             240
result = ~origin                         15
```

```
let shift : UInt8 = 0b10101010           170
result = shift << 1                      84
result = shift << 2                      168
result = shift >> 1                      85
result = shift >> 2                      42
```

```
var operatorA : UInt8 = 0b11001100       204
var operatorB : UInt8 = 0b10101010       170

result = operatorA & operatorB           136
result = operatorA | operatorB           238
result = operatorA ^ operatorB           102
```



溢出运算符

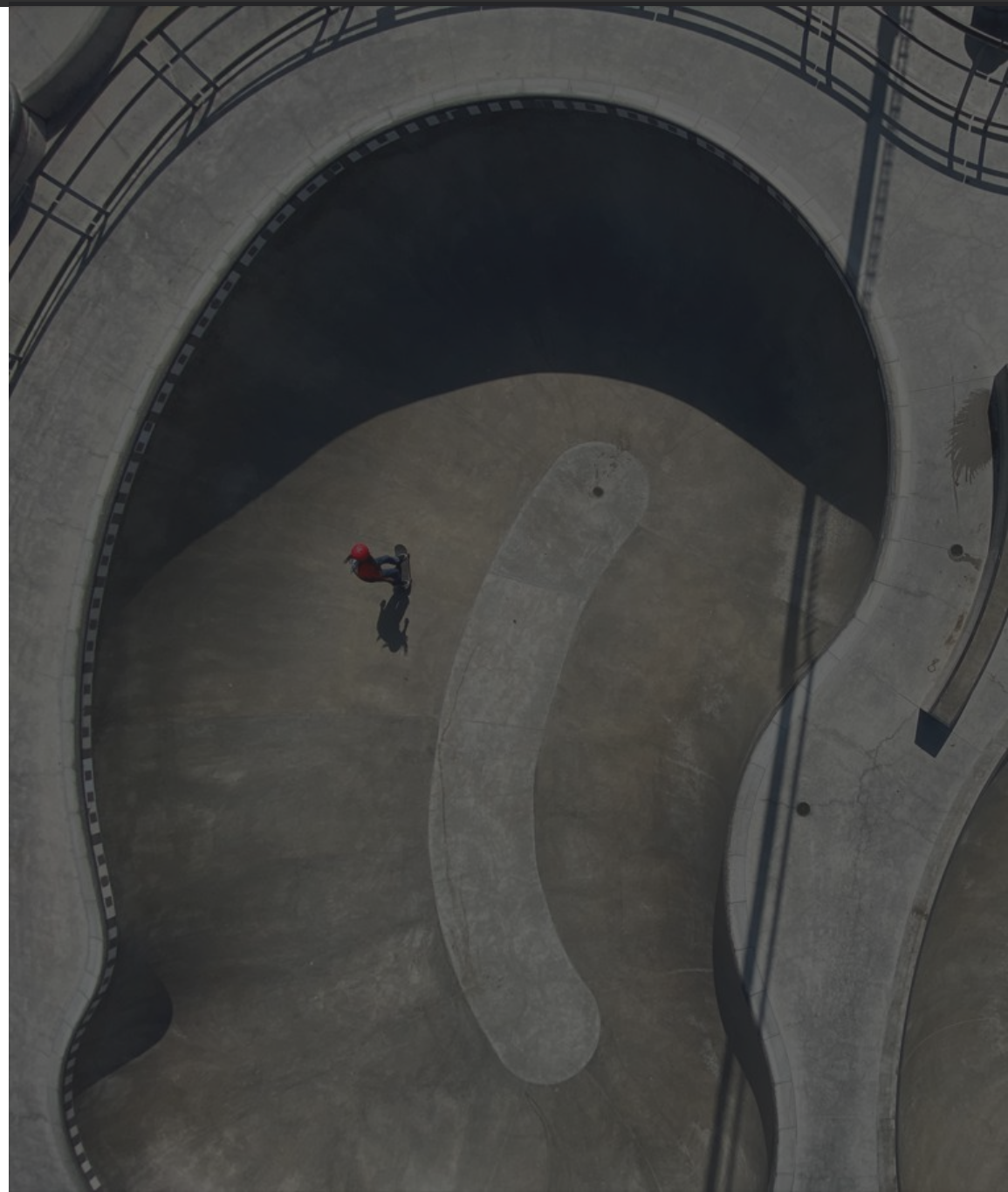
当一个整数的赋值超过了它的最大限度时，系统会报错。

溢出操作符支持整数的溢出运算，即：如果发生了上例中出现的溢出运算，系统不会报错，而会将溢出部分舍弃。

溢出操作符包括：溢出加法“&+”，溢出减法“&-”，溢出乘法“&*”。

```
origin = UInt8.max                    255
origin = UInt8.min                    0
origin = 266
  Integer literal '266' overflows when stored into 'UInt8'
```

```
result = UInt8.max &+ 1               0
result = UInt8.min &- 1               255
result = UInt8.max &* 2               254
```



运算符重载

Swift支持在类和结构体中对现有的操作符提供自定义的实现，这个过程称为运算符重载。

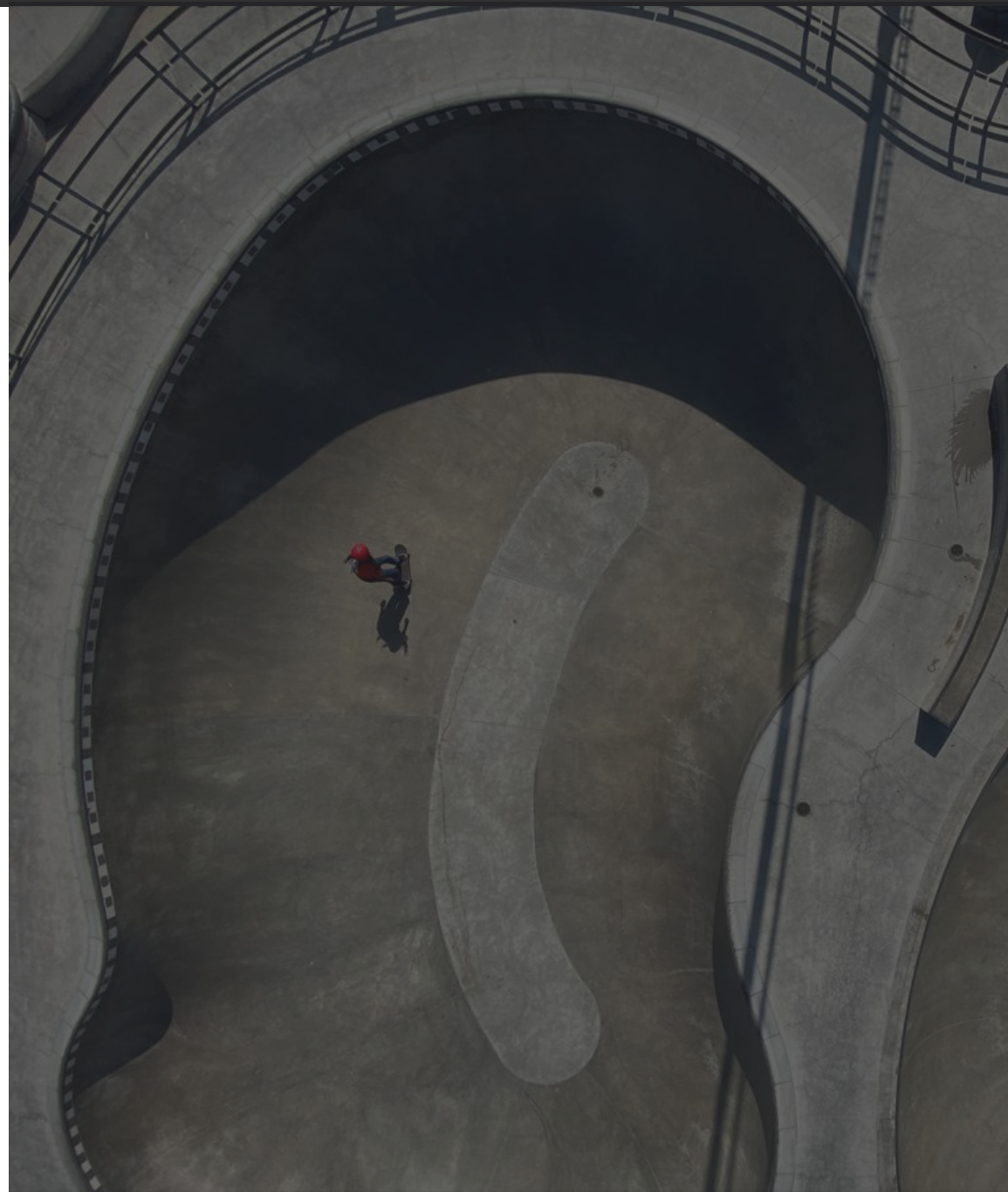
例子（双目运算符重载）

```
struct Point {
    var x = 0
    var y = 0
}

func -(a: Point,b: Point)-> Point {
    return Point(x: a.x-b.x, y:
        a.y-b.y)
}

let a = Point(x: 3, y: 5)
let b = Point(x: 1, y: 6)
let c = a - b
```

```
x 2
y -1
```



重载单目运算符

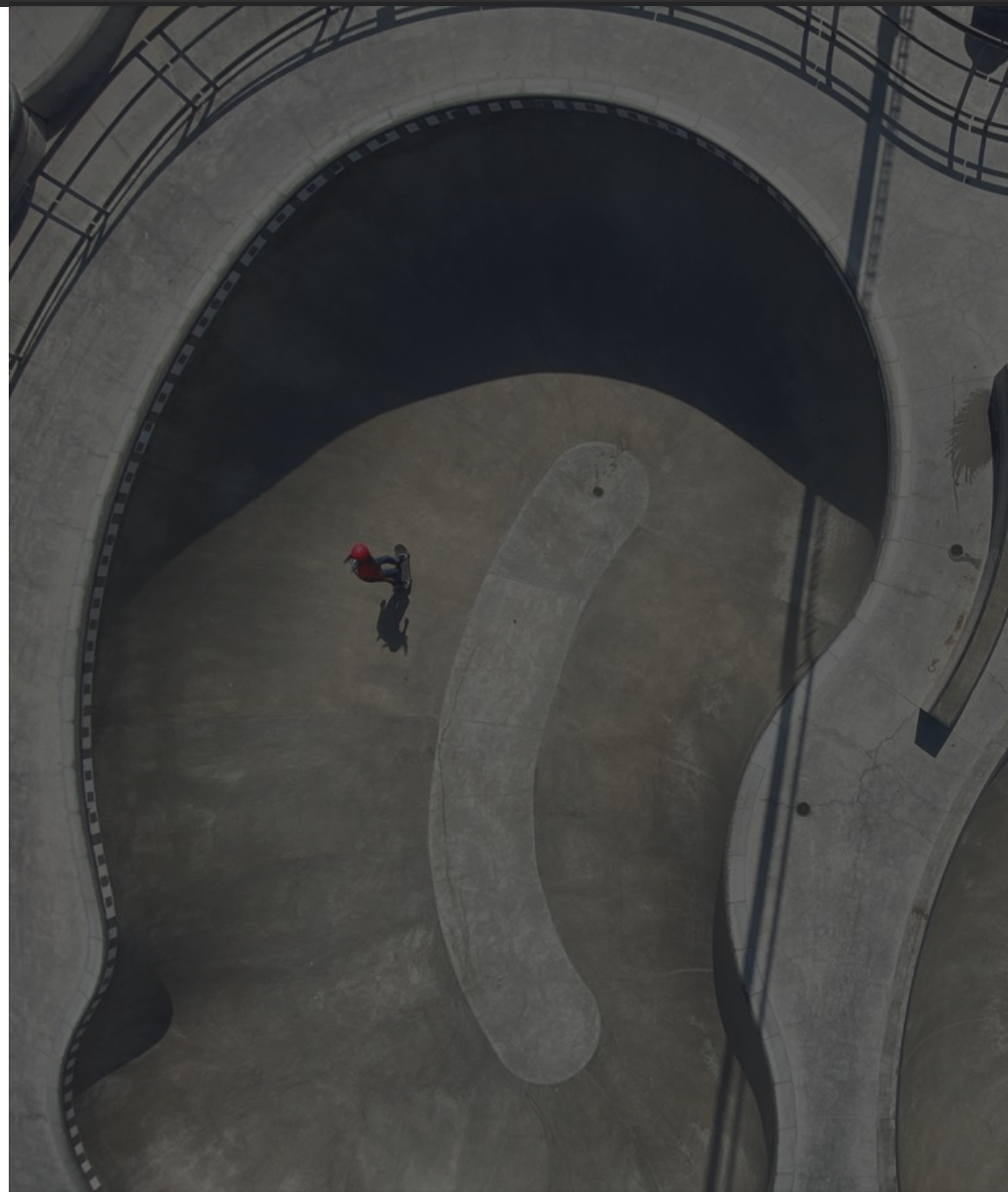
Swift除了支持双目中缀运算符的重载，也支持单目运算符的重载，即：只有一个操作数。

前缀/后缀运算符

根据运算符在操作数的前面和后面，又分为前缀运算符和后缀运算符。

标识单目运算符的重载

要重载一个前缀或后缀运算符时，须在声明运算符函数的func前加上关键字prefix或postfix。



重载单目运算符

Swift除了支持双目中缀运算符的重载，也支持单目运算符的重载，即：只有一个操作数。

前缀/后缀运算符

根据运算符在操作数的前面和后面，又分为前缀运算符和后缀运算符。

标识单目运算符的重载

要重载一个前缀或后缀运算符时，须在声明运算符函数的func前加上关键字prefix或postfix。

例子

```
struct Point {
    var x = 0
    var y = 0
}

prefix func -(point: Point)-> Point{
    return Point(x: -point.x, y: -
        point.y)
}

let point = Point(x: 3, y: 4)
let pointN = -point
```

```
x -3
y -4
```

重载复合赋值运算符

将赋值运算符“=”与其他运算符结合起来形成的运算符，称为复合赋值运算符。

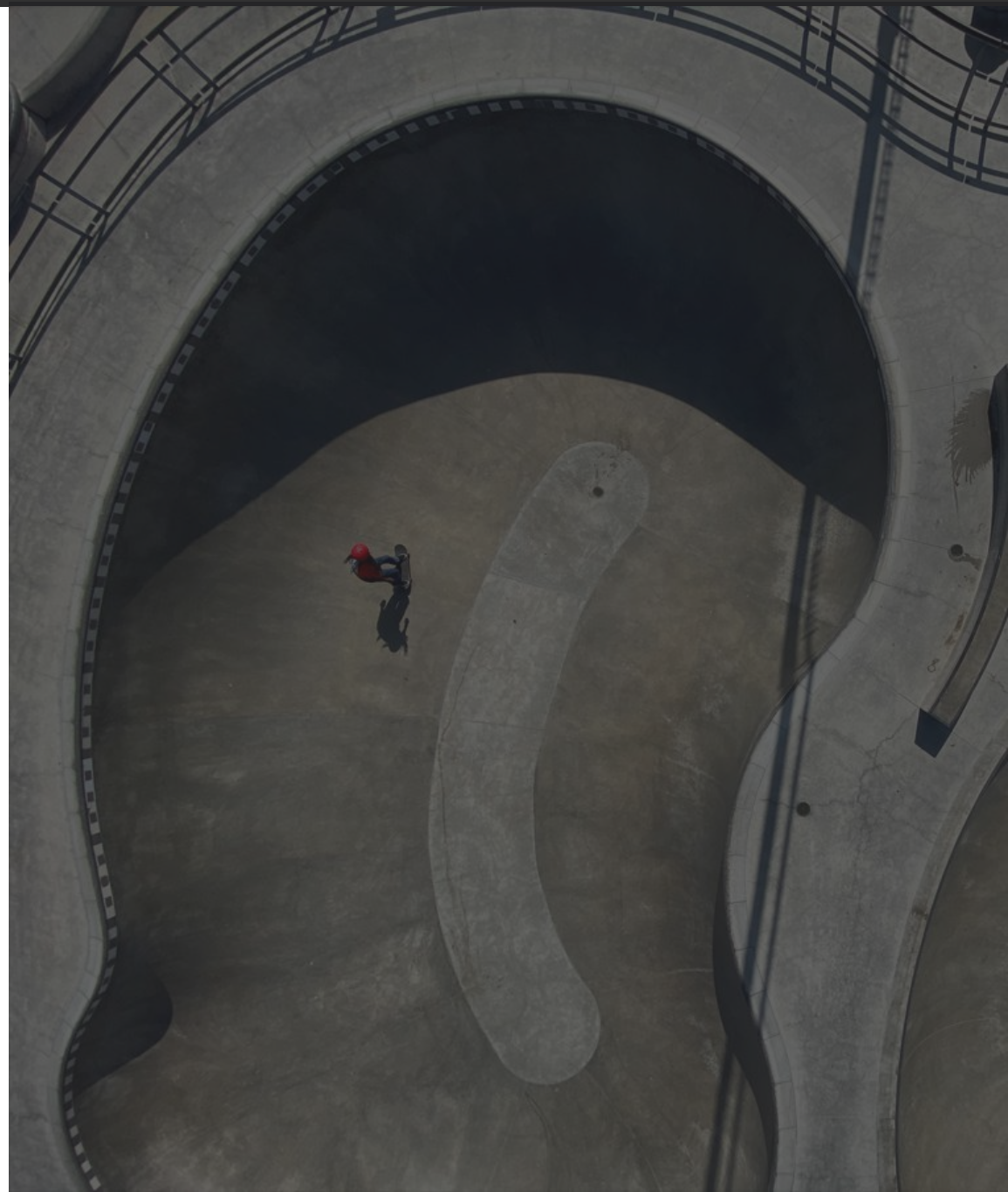
例子

```
struct Point {
    var x = 0
    var y = 0
}

func -=(inout origin:Point, decrement:
    Point){
    origin.x = origin.x - decrement.x
    origin.y = origin.y - decrement.y
}

var point = Point(x: 3, y: 6)
let decrement = Point(x: 2, y: 8)
point-=decrement
```

```
x 1
y -2
```

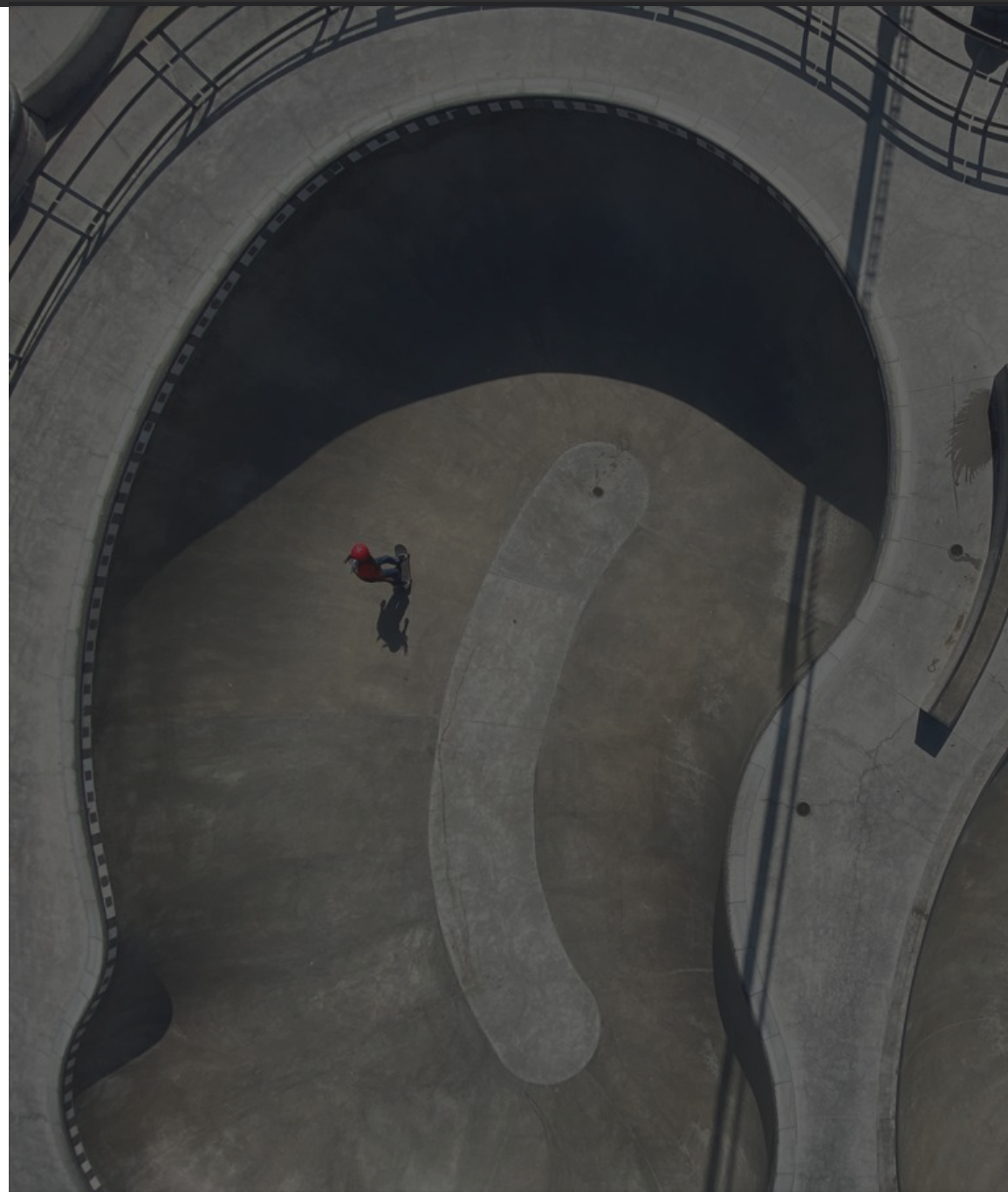


等价运算符

在Swift中没有提供用来判断类或结构体等价的操作符。

重载等价运算符

可以通过重载等价运算符“==”或者不等价操作符“！=”的方式来实现对两个类或者结构体是否等价的运算。



等价运算符

在Swift中没有提供用来判断类或结构体等价的操作符。

重载等价运算符

可以通过重载等价运算符“==”或者不等价操作符“！=”的方式来实现对两个类或者结构体是否等价的运算。

```
struct Point {
    var x = 0
    var y = 0
}

func == (point1: Point, point2: Point)-
    >Bool{
    return (point1.x == point2.x) &&
        (point1.y==point2.y)
}

let point1 = Point(x: 1, y: 2)
let point2 = Point(x: 2, y: 3)
let point3 = Point(x: 1, y: 2)

if point1 == point2 {
    print("point1 is equal to point2")
}
if point1 == point3 {
    print("point1 is equal to point3")
```

```
point1 is equal to point3
```

```
}
```

大作业的评价维度

评分点	权重
文档质量	0.25
用户友好性	0.15
新颖性	0.25
功能性	0.25
代码风格	0.1