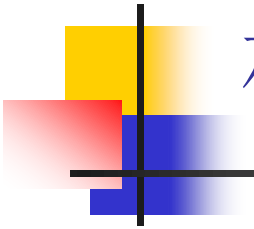
第三部分　面向对象基础

授课教师：杨文川

面向对象基础（6学时）

本章目的在于掌握面向对象的基础知识，从而为面向对象的程序设计打下基础。内容包括：

- 枚举类型（定义、关联值及原始值的定义和用法）
- 结构体和类（共同点和不同点，以及各自的应用场景）
- 属性（存储属性、计算属性、属性观察器以及类型属性）
- 方法（实例方法、类型方法、方法下标）
- 继承性、基类、子类、重载。
- 构造器和析构器
- 错误处理



枚举类型

- 枚举类型定义一种普通类型的一组相关的值，从而使我们能够在类型安全的使用这些值。
- Swift中的枚举类型非常灵活，不需要给每一个枚举成员一个对应的值。如果要提供给枚举成员一个对应的关联值，那么值的类型可以是字符串、字符、整型或者浮点型。
- Swift中的枚举型具有一些类的特征，如：计算属性、实例方法、构造器以及协议等。

枚举型的定义

- 定义一个枚举类型需要使用关键字enum
- 具体格式如下：

enum Name {

case ValueName

}

- 枚举类型的名字必须以大写字母开头。
- 每一个值前面都必须用关键字case标示。

```
enum Month {
    case January
    case February
    case March
    case April
    case May
    case June
    case July
    case August
    case September
    case October
    case November
    case December
}
```

紧凑写法

```
enum Month {
case January,February,March,April,May,June,
    July,August,September,October,November,
    December
}
```

代码参见iosSwift的Chapter08-1

枚举型的使用

- 枚举类型变量的赋值

```
var thisMonth = Month.August
var nextMonth : Month
nextMonth = .September
```

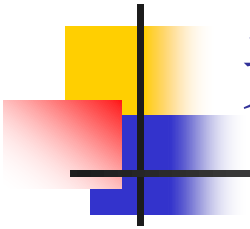
```
August

September
```

- 枚举值具有很好的可读性，常常应用于switch语句中，用来进行条件匹配。

```
switch nextMonth {
case .January, .February, .March, .April, .May
    ,| .June : print("It belongs first half
    year")
default : print("It belongs second half year")
}
```

```
"It belongs second half year\n"
```

代码参见iosSwift的Chapter08-1



关联值

- 枚举类型支持为每一个枚举成员设置一个常量或变量，我们称为关联值。
- 每一个枚举成员都可以有一个关联值。
- 关联值和成员值可以配对存储起来。
- 每个枚举成员的关联值都可以是不同的数据类型。

关联值

- 关联值定义

```
enum transportFee {
    case byAir(Int,Int,Int)
    case byCar(Int,Int)
    case byTrain(Int)
}
```

- 关联值的赋值

```
var fromShanghaiToBeijing = transportFee.
    byTrain(299)

fromShanghaiToBeijing = .byAir(800, 230, 50)
```

```
byTrain(299)

byAir(800, 230, 50)
```

代码参见iosSwift的Chapter08-2

- 关联值的获取

```
switch fromShanghaiToBeijing {
case .byAir(let ticketFee, let tax, let
    insurance) : print("The sum fee is \
    (ticketFee+tax+insurance) by air")
case .byCar(let fuelFee, let highwayFee) :
    print("The sum fee is \(fuelFee
    +highwayFee) by car")
case .byTrain(let ticketFee) : print("The sum
    fee is \(ticketFee) by Train")
}
```

```
"The sum fee is 1080 by air\n"
```

- 关联值的提取

```
switch fromShanghaiToBeijing {
case let .byAir(ticketFee, tax, insurance) :
    print("The sum fee is \(ticketFee+tax+
    insurance) by air")
case let .byCar(fuelFee, highwayFee) :
    print("The sum fee is \(fuelFee
    +highwayFee) by car")
case let .byTrain(ticketFee) : print("The sum
    fee is \(ticketFee) by Train")
}
```

代码参见iosSwift的Chapter08-3

原始值

- 枚举成员的原始值必须为相同的数据类型。
- 枚举成员的原始值可以是字符串、字符、整型及浮点型。
- 原始值的类型必须在定义枚举类型的时候显式的定义，具体的格式为：

  enum Name : DataTypeOfRawValue { case valueName = value }
- 每个原始值在枚举声明中必须是唯一的，不存在两个枚举成员具有相同的原始值。
- 原始值与关联值的区别
  - 原始值是在定义枚举类型的时候预先被设置的值，可以理解为缺省值。对于某一个枚举成员，它的原始值是不会改变的。
  - 关联值则是在创建枚举成员的关联的新常量或变量时设置的，它的值是可以改变的。

原始值实例

- 原始值的定义

```
enum car : Int {
    case truck = 0
    case sportsCar
    case SUV
    case MPV
    case limo
}
```

- 原始值的获取

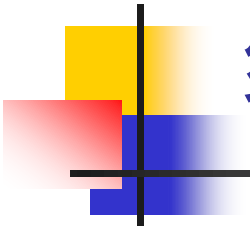
```
var myCar = car.SUV                 SUV
var rawValue = myCar.rawValue       2
myCar = car.limo                    limo
rawValue = myCar.rawValue           4

let yourCar = car(rawValue: 1)      sportsCar
```

代码参见iosSwift的Chapter08-4

第三部分　面向对象基础

本章目的在于掌握面向对象的基础知识，从而为面向对象的程序设计打下基础。内容包括：

- 枚举类型（定义、关联值及原始值的定义和用法）
- 结构体和类（共同点和不同点，以及各自的应用场景）
- 属性（存储属性、计算属性、属性观察器以及类型属性）
- 方法（实例方法、类型方法、方法下标）
- 继承性、基类、子类、重载。
- 构造器和析构器
- 错误处理



结构体和类

- 结构体和类都是实现面向对象的重要手段，两者非常类似，又有一定的差异。
- 可以将结构体理解为一种轻量级的类。
- 在Swift中，通常通过一个单独的文件来定义一个类或者一个结构体，然后由系统自动生成外部接口，而不像其它的语言那样，要手动去创建接口或者实现文件。

共同点

- 结构体拥有很多类的特点：属性、方法，下标语法、构造器等
- 结构体不支持类的继承、析构、引用计数及类型转换等特点。
- 结构体和类的定义方式非常类似，都是关键字（结构体为struct，类为class）加上名字，然后花括号定义具体内容
- 结构体定义的格式：

```
struct NameOfStruct {
        struct definition
}
```

- 类定义的格式：

```
class NameOfClass {
        class definition
}
```

```
struct Book {
    var name = ""
    var price = 0
    var category :String
}

class Reader {
    var name = ""
    var age = 16
    var favorite :String?
}
```

代码参见iosSwift的Chapter09-1

- 官方命名方式
  - 定义结构体或类的名称时，需要大写首字母。
  - 定义结构体或类中的属性和方法时，要小写首字母。

不同点——结构体

- 结构体是值类型，也就是说当一个结构体变量的值赋给另一个变量时，是通过复制结构体变量值来实现的。
- 所有的基本类型，包括：整型、浮点型、布尔型、字符串、字符型、数组以及字典，都属于值类型

```
var anotherBook = theBook
print(theBook)
print(anotherBook)
anotherBook.category = "history"
anotherBook.price = 136
anotherBook.name = "Empiror Kangxi"
print(theBook)
print(anotherBook)
```

```
Book
"Book(name: "Life of Pi", price: 62, category: "adventure")\n"
"Book(name: "Life of Pi", price: 62, category: "adventure")\n"
Book
Book
Book
"Book(name: "Life of Pi", price: 62, category: "adventure")\n"
"Book(name: "Empiror Kangxi", price: 136, category: "history")\n"
```

代码参见iosSwift的Chapter09-1

不同点——类

- 类是引用类型的。所谓引用类型，就是引用类型变量在被赋值给另一个变量时，是将该引用类型变量的物理存储地址赋值给另一个变量。赋值后，该变量和被赋值的变量所指向的物理地址相同。
- 类似于C语言的指针

```
theReader.name = "Tommy"
var anotherReader = theReader
print("\(theReader.name) is \
    (theReader.age)")
print("\(anotherReader.name) is \
    (anotherReader.age)")

anotherReader.name = "Jerry"
anotherReader.age = 39
print("\(theReader.name) is \
    (theReader.age)")
print("\(anotherReader.name) is \
    (anotherReader.age)")
```

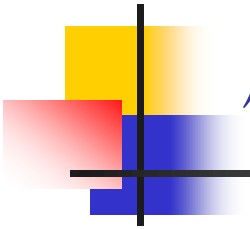
```
Reader
Reader
"Tommy is 16\n"

"Tommy is 16\n"


Reader
Reader
"Jerry is 39\n"

"Jerry is 39\n"
```

代码参见iosSwift的Chapter09-1

第三部分　面向对象基础

本章目的在于掌握面向对象的基础知识，从而为面向对象的程序设计打下基础。内容包括：

- 枚举类型（定义、关联值及原始值的定义和用法）
- 结构体和类（共同点和不同点，以及各自的应用场景）
- 属性（存储属性、计算属性、属性观察器以及类型属性）
- 方法（实例方法、类型方法、方法下标）
- 继承性、基类、子类、重载。
- 构造器和析构器
- 错误处理



属性

- 类、结构体和枚举类型都有属性。
- 属性将值和这些类型进行关联。
- 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。存储属性和计算属性与具体的实例相关联。
  - 类型属性是属于类型本身的，而不属于特定的实例。
- 属性观察者，它用来监视属性值的变化，并据此来触发特定的操作。

存储属性

- 存储属性就是存储在一个类或者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指定一个默认值，也可以在构造器中设置或修改存储属性的值。
- 常量存储属性被初始化后，值是不能改变的。
- 变量存储属性是可以修改的

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

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

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

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

代码参见iosSwift的Chapter10-1

延迟存储属性

- 在存储属性声明前加上一个关键字lazy，表示该属性为延迟存储属性。
- 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。
- 使用场景
  - 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，常常声明该计算属性为延迟属性。
  - 一个属性的值依赖于实例初始化完以后的外部因素是，我们也将其声明为延迟属性。

```
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"
//Till now, instance of image hasn't been
    created.In the following, let's create it.
print("the name of image is \(theUser.image.
    name)")
```

```
user
user
user


"the name of image is default name\n"
```

代码参见iosSwift的Chapter10-2

计算属性

- 计算属性不直接存储值，它通过getter和setter方法来获取和设置值。
- 类、结构体和枚举类型都可以定义计算属性。

```
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")
```

```
"Tony has worked for 3 years and he has holiday: 8 days\n"
```

代码参见iosSwift的Chapter10-3

属性观察器

- 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，都可以触发属性观察器。
- 除了延迟存储属性，所有其它属性都可以增加一个属性观察器，对其值的变化进行监控。
- 属性观察器有两个方法：willSet和didSet。
  - willSet在属性的值被改变前调用。willSet会将新值作为常量参数传入，缺省名称为newValue，也可以为该参数自定义一个名称。
  - didSet在属性的值被改变后调用。didSet则会将旧值作为参数传入oldValue，同样也可以为该参数自定义一个名称。

```
class Website {
    var domain : String = ""
    var clicks : Int = 0 {
        willSet(newClicks){
            print("clicks will be set to \
                (newClicks)")
        }
        didSet {
            print("did set clicks from \
                (oldValue) to \(clicks)")
        }
    }
}

let theWebsite = Website()
theWebsite.domain = "www.buaa.edu.cn"
theWebsite.clicks = 100
theWebsite.clicks = 200
```

(2 times)

(2 times)

Website
Website
Website
Website

```
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
```

代码参见iosSwift的Chapter10-4

类型属性

- 每次类型实例化后，每个实例都拥有一套自己的属性值，各实例的属性值互相独立。
- 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。
- 类型属性是用来定义某个类型的所有实例都共享的数据的
- 类型属性使用关键字static来标识。
- 类型属性的作用域为类型的内部。
- 类型属性的访问也是通过点号运算符来进行的。

```
class Visitor {
    var name : String = ""
    var stayTime : Int = 0
    static var permission : String = "visitor"
}

let theVisitor = Visitor()
theVisitor.name = "Tom"
theVisitor.stayTime = 5
print("Current permission is \(Visitor.
    permission)")
let anotherVisitor = Visitor()
anotherVisitor.name = "Sam"
anotherVisitor.stayTime = 9
Visitor.permission = "administrator"
print("Now permission is \
    (Visitor.permission)")
```

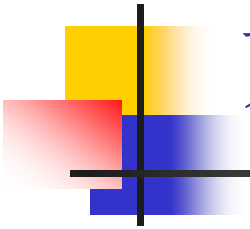
```
Visitor
Visitor
Visitor
"Current permission is visitor\n"

Visitor
Visitor
Visitor

"Now permission is administrator\n"
```

代码参见iosSwift的Chapter10-5

第三部分　面向对象基础

本章目的在于掌握面向对象的基础知识，从而为面向对象的程序设计打下基础。内容包括：

- 枚举类型（定义、关联值及原始值的定义和用法）
- 结构体和类（共同点和不同点，以及各自的应用场景）
- 属性（存储属性、计算属性、属性观察器以及类型属性）
- 方法（实例方法、类型方法、方法下标）
- 继承性、基类、子类、重载。
- 构造器和析构器
- 错误处理



方法

- 方法是类、结构体或枚举中定义实现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，和该类型的实例无关。

实例方法

- 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。实例方法的定义方法和函数完全一致。
- 实例方法的定义要写在类型定义的花括号内。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。

```
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
```

代码参见iosSwift的Chapter11-1

函数参数的外部名称

- 函数参数除了有局部名称外，还可以有一个外部名称。同样，方法的参数除了有局部名称外，也可以有一个外部名称。在Swift中，默认情况下，方法的第一个参数只有局部名称，第二个及后续的参数同时有局部名称和外部名称。
- 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。

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

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

```
1
["Tommy"]
Website


Website
Website

1
["Tommy"]
"2016-6-1"
```

代码参见iosSwift的Chapter11-2

类型方法

- 类型方法是只能由类型本身调用的方法。
- 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。

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

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

```
1
["Tommy"]
Website.Type

1
["Tommy"]
"2016-6-1"
```

代码参见iosSwift的Chapter11-3

方法下标

- 下标是一种通过下标的索引来获取值的快捷方法。
- 下标最典型的例子就是在数组中，使用下标来进行数组元素的读写，例如：Array[Index]。
- 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。
- 下标通过实例后面的括号中传入一个或者多个的索引值来对实例进行访问和赋值的。
- 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setter和getter方法的定义，可以实现读写或者只读。
- 具体格式为：

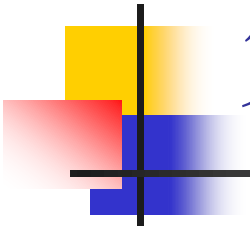
```
subscript ( index : Int ) -> Int {
                get {
                        // return index
                }
                set ( newValue ) {
                        // set new value
                }
}
```

方法下标——实例

- 在下标中定义了get方法和set方法。get方法要根据索引值index来返回visitor数组中的某一个访客的名字，set方法实现向索引值对应的变量进行赋值。
- 创建了website的实例sina，并通过方法visiting向数组visitor中写入了两个访客的名字。
- 通过下标语法来快速方便的对特定元素进行访问和赋值了。这里通过sina[0]将visitor数组中的第一个值读出。然后通过sina[2]="Pennie"，实现了对visitor数组中写入值。

```
class Website {
     var visitCount = 0
     var visitor = [String]()
     var visitDate = ""
     func visiting(visitor visitor:String,
         visitDate : String){
        ++visitCount                          (2 times)
        self.visitor.append(visitor)          (2 times)
        self.visitDate = visitDate            (2 times)
    }
    subscript(index : Int) -> String {
        get {
            return visitor[index]             "Tom"
        }
        set {
            visitor[index] = newValue         "Pennie"
        }
    }
}

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

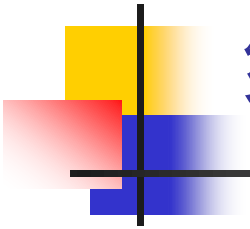
代码参见iosSwift的Chapter11-5



第三部分　面向对象基础

本章目的在于掌握面向对象的基础知识，从而为面向对象的程序设计打下基础。内容包括：

- 枚举类型（定义、关联值及原始值的定义和用法）
- 结构体和类（共同点和不同点，以及各自的应用场景）
- 属性（存储属性、计算属性、属性观察器以及类型属性）
- 方法（实例方法、类型方法、方法下标）
- 继承性、基类、子类、重载
- 构造器和析构器
- 错误处理



继承性

- 继承性是指一个类可以继承别的类的方法和属性，这个继承的类称为子类，被继承的类称为父类。
- 继承性是类的一个基本特征。
- 在Swift中，一个类可以调用父类的方法，访问父类的属性和下标，还可以重载父类的方法、属性和下标。

基类

- 没有父类的类，称为基类。
- 在Swift中，所有的类并不是从一个通用的基类继承而来的。如果不为一个类指定一个父类的话，那么这个类就是基类。

```
class Student {
    var name = ""
    var age = 0
    var id = ""
    var basicInfo : String {
        return "\(name) is \(age)
            years old, the id is \
            (id)"
    }
    func chooseClass(){
        print("\(name) choose a
            class.")
    }
    func haveClass(){
        print("\(name) have a
            class.")
    }
}

let theStudent = Student()
theStudent.name = "Tommy"
theStudent.age = 19
theStudent.id = "37060115"
print(theStudent.basicInfo)
```

```
"Tommy is 19 years old, the id is 37060115"

Student
Student
Student
Student
"Tommy is 19 years old, the id is 37060115\n"
```

代码参见iosSwift的Chapter12-1

子类

- 子类就是在一个已有类的基础上创建的一个新类，它继承了父类的全部特性，并且还有自己的特性。
- 定义一个子类的格式为：
- class SonClass : FatherClass {
- class definition
- }
- 子类的定义和一般类的定义几乎一样，除了需要在子类的类名后标注其父类的类名，并用冒号分隔。

```
class Graduate : Student {
    var supervisor = ""
    var researchTopic = ""
    func chooseSuperVisor(superVisor:String){
        self.supervisor = superVisor
    }
}

let theGraduate = Graduate()
theGraduate.name = "Sam"
theGraduate.age = 23
theGraduate.id = "SY0602115"
theGraduate.haveClass()
theGraduate.researchTopic = "Graphics"
theGraduate.chooseSuperVisor("Ian")
print("Graduate \(theGraduate.name) is \
    (theGraduate.age) and the id is \
    (theGraduate.id), The research topic is \
    (theGraduate.researchTopic) and
    supervisor is \(theGraduate.supervisor)")
```

```
Graduate Sam is 23 and the id is SY0602115, The
research topic is Graphics and supervisor is Ian
```

代码参见iosSwift的Chapter12-1

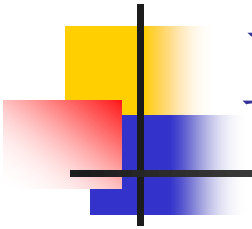
子类的子类

- 子类也可以有子类。
- 子类的子类也继承父类及父类的父类的全部属性和方法。

```
class Doctor: Graduate {
    var articles = [String]()
    func publishArticle(article : String){
        articles.append(article)
    }
}

let theDoctor = Doctor()
theDoctor.name = "Pennie"
theDoctor.age = 26
theDoctor.id = "BY0607120"
theDoctor.basicInfo
theDoctor.chooseSuperVisor("Ellis")
theDoctor.supervisor
theDoctor.publishArticle("Petri nets theory")
theDoctor.publishArticle("Process
    management")
theDoctor.articles
```

```
(2 times)


Doctor
Doctor
Doctor
Doctor
"Pennie is 26 years old, the id is BY0607120"
Doctor
"Ellis"
Doctor
Doctor

["Petri nets theory", "Process management"]
```

代码参见iosSwift的Chapter12-1



重载

- 重载是面向对象中的一个非常重要的概念。重载指的是子类将从父类继承来的实例方法、类方法、实例属性及下标等提供自己的实现。
- 在重载一个父类的特性时，需要在该特性前面加上关键字override。
  - 对于系统来说，关键字override意味着要重新定义一个特性，该特性是继承自父类的。如果不加关键字override，子类中同名的特性会导致编译阶段报错。
- 在子类中重载父类的方法、属性或下标时，可以通过super前缀语法来引入父类的实现，从而使子类的代码专注实现新增的功能。

方法重载

- 在重载父类的方法method时，先通过super.method来访问父类的method方法，然后再定义子类中新增的实现。
- 在重载父类中属性property的getter和setter方法时，通过super.property来访问父类的property属性。
- 在重载父类中的下标时，通过super[index]来访问父类中的相同下标。
- 在子类中，可以提供一个新的实现来重载父类的实例方法或类方法。重载后，父类中的方法将会被完全覆盖。

```
class Graduate : Student {
    var supervisor = ""
    var researchTopic = ""
    func chooseSuperVisor(superVisor:
        String){
        self.supervisor = superVisor
    }
    override func chooseClass() {
        print("graduate \(name) choose a
            class")
    }
}
let theGraduate = Graduate()
theGraduate.name = "Sam"
theGraduate.chooseClass()
```

```
"graduate Sam choose a class\n"
Graduate
Graduate
Graduate
```

代码参见iosSwift的Chapter12-2

属性重载

- 在子类中可以重载父类的实例属性和类属性，为其提供新的getter和setter方法，并添加属性观察器。
- 在重载父类的属性时，子类需要把它的名字和类型都写出来，这样编译器就会去匹配重载的属性和父类中同名同类的属性。
- 如果要将父类中的只读属性重载为一个读写属性，只需要在子类中为该属性提供getter和setter方法。
- 不能将一个父类的读写属性重载为一个只读属性。
  - 如果在重载父类属性时提供了setter，就一定要提供getter。
  - 如果不想在子类重载属性property的getter中修改属性值，可以直接通过super.property来返回属性值。

```
class Graduate : Student {
    var supervisor = ""
    var researchTopic = ""
    override var basicInfo : String {
        return super.basicInfo + ",
            supervisor is \(supervisor),
            research topic is \
            (researchTopic)"
    }
    func chooseSuperVisor(superVisor:String){
        self.supervisor = superVisor
    }
    override func chooseClass() {
        print("graduate \(name) choose a
            class")
    }
}
let theGraduate = Graduate()
theGraduate.name = "Sam"
theGraduate.age = 25
theGraduate.id = "SY06011125"
theGraduate.supervisor = "Ian"
theGraduate.researchTopic = "PetriNet"
theGraduate.basicInfo
```

```
Sam is 25 years old, the id is SY06011125, supervisor
is Ian, research topic is PetriNet
```

代码参见iosSwift的Chapter12-3

重载属性的观察器

- 在子类中重载一个属性的时候，还可以为该属性增加属性观察器，从而监控该属性值的变化了。
- 注意：
  - 不能为继承来的常量存储属性或只读计算型属性增加属性观察器，因为这些属性的值是不能被修改的，所以也不会发生改变。

```
class Graduate : Student {
    var supervisor = ""
    var researchTopic = ""
    override var age : Int {
        didSet {
            print("age is set from \(oldValue)
                to \(age)")
        }
        willSet {
            print("original age will be set to
                \(newValue)")
        }
    }
    override var basicInfo : String {
        return super.basicInfo + ", supervisor
            is \(supervisor), research topic is
            \(researchTopic)"
    }
    func chooseSuperVisor(superVisor:String){
        self.supervisor = superVisor
    }
    override func chooseClass() {
        print("graduate \(name) choose a
            class")
    }
}
let theGraduate = Graduate()
theGraduate.name = "Sam"
theGraduate.age = 25
```

```
"age is set from 0 to 25\n"
"original age will be set to 25\n"
Graduate
Graduate
Graduate
```

代码参见iosSwift的Chapter12-3

第三部分　面向对象基础

本章目的在于掌握面向对象的基础知识，从而为面向对象的程序设计打下基础。内容包括：

- 枚举类型（定义、关联值及原始值的定义和用法）
- 结构体和类（共同点和不同点，以及各自的应用场景）
- 属性（存储属性、计算属性、属性观察器以及类型属性）
- 方法（实例方法、类型方法、方法下标）
- 继承性、基类、子类、重载。
- 构造器和析构器
- 错误处理

构造器和析构器

- 构造就是在使用类、结构体或枚举类型的实例前，要先通过构造器来设置每个存储型属性的初始值，同时执行其它必要的设置初始化工作。这种初始化工作可以通过构造器来实现。
- 构造器不需要返回值，它的主要任务就是为新实例的第一次使用做好准备工作。
- 只有引用类型——类，才有析构器。
- 当一个类的实例使用结束，要被释放之前，析构器就会被调用。

构造器定义

- 在创建一个类型的实例时，构造器会被调用。最简单的构造器格式为：

  init () {

  //initialization process

  }

- 构造器要以关键字init来命名，可以不带任何参数。

```
class Student {
    var name : String
    var age : Int
    var id : String
    var basicInfo : String {
        return "\(name) is \(age) years old,
            the id is \(id)"
    }
    init(){
        name = "no name"
        age = 16
        id = "no id"
    }
}

var theStudent = Student()
```

```
name "no
age 16
id "no id"
```

代码参见iosSwift的Chapter13-1

定制构造器（一）

- 除了前面介绍的不带参数的缺省构造器以外，也可以自定义构造器，并定义构造器参数，包括：参数的类型和名字。

```
class Student {
    var name : String
    var age : Int
    var id : String
    var basicInfo : String {
        return "\(name) is \(age) years old,
            the id is \(id)"
    }
    init(){
        name = "no name"
        age = 16
        id = "no id"
    }
    init(name : String, age : Int){
        self.name = name
        self.age = age
        self.id = "no id"
    }
    init(name : String, age : Int, id :
        String){
        self.name = name
        self.age = age
        self.id = id
    }
}
```

```
var theStudent = Student(name: "Tom", age:
    25)
```

```
name "Tom"
age 25
id "no id"
```

```
var anotherStudent = Student(name: "Sam",
    age: 29, id: "BY0602115")
```

```
name "Sam"
age 29
id "BY0602115"
```

代码参见iosSwift的Chapter13-2

定制构造器（二）

- 在构造器中，参数也存在一个构造器内部使用的参数名和在调用构造器时使用的外部参数名。
- 如果在定义构造器的时候没有提供参数的外部名称，那么系统会自动为每个参数自动生成一个跟内部名称同名的外部名称。

```
class Student {
    var name : String
    var age : Int
    var id : String
    var basicInfo : String {
        return "\(name) is \(age) years old,
            the id is \(id)"
    }
    init(_ name : String, _ age : Int, _ id :
        String){
        self.name = name
        self.age = age
        self.id = id
    }
}

var anotherStudent = Student("Sam", 29,
    "BY0602115")
```

```
name
age 29
id
```

代码参见iosSwift的Chapter13-3

定制构造器（三）

- 如果类型定义中包含一个允许值为空的属性，那么必须要定义为可选类型。
- 系统会自动为可选类型的属性初始化，初始化后值为nil。对于可选类型的属性，不需要在构造器里进行初始化。

```
class Student {
    var name : String
    var age : Int
    var id : String?
    var basicInfo : String {
        return "\(name) is \(age) years old,
            the id is \(id)"
    }
    init(){
        name = "no name"
        age = 16
    }
    init(name : String, age : Int){
        self.name = name
        self.age = age
    }
}

var theStudent = Student(name: "Tom", age:
    25)
```

```
name "Tom"
age 25
nil
```

代码参见iosSwift的Chapter13-4

构造器代理（一）

- 在定义复杂构造器的时候，我们可以通过调用已经定义的构造器来完成部分构造工作，从而简化重复代码，这个过程称之为构造器代理。
- 值类型和引用类型的构造器代理的实现方式有所不同
  - 值类型的构造器代理只能提供给类型本身的构造器使用。我们通过self.init在自定义的构造器中调用类型中已经定义的其它构造器。

```
struct Student {
    var name : String
    var age : Int
    var id : String
    var basicInfo : String {
        return "\(name) is \(age) years old,
            the id is \(id)"
    }
    init(){
        name = "no name"
        age = 16
        id = ""
    }
    init(N name : String, A age : Int){
        self.name = name
        self.age = age
        id = ""
    }
    init(name : String, age : Int, id :
        String){
        self.init(N : name, A : age)
        self.id = id
    }
}

let theStudent = Student(name: "Sam", age:
    28, id: "BY0601115")
```

```
name
age 28
id
```

代码参见iosSwift的Chapter13-5

构造器代理（二）

- 引用类型（主要指类）里的所有存储属性，包括从父类继承的属性，都必须在构造器中进行初始化。
- 有两种类型的引用类型构造器：指定构造器和便利构造器。
- 指定构造器是类中最主要的构造器，它负责初始化类中定义的所有属性，并调用父类的构造器来实现父类中属性的初始化。每个类都必须有一个指定构造器。
- 便利构造器是类中一个重要的辅助构造器。它可以调用同一个类中的指定构造器。

- 指定构造器的定义格式为：

```
init(parameters list) {
                    some statements
}
```

- 便利构造器格式和指定构造器基本相同，只是在init前面多了一个关键字convenience，格式为：

```
convenience  init(parameters list) {
                    some statements
}
```

指定构造器和便利构造器

- 对于指定构造器和便利构造器的使用，有三个规则：

  1. 指定构造器必须调用其直接父类的指定构造器。

  2. 便利构造器必须调用同一个类中的其它构造器。

  3. 便利构造器必须最终以调用一个指定构造器结束。

- 指定构造器是纵向代理的的，即向父类代理。便利构造器则是横向代理，即：向同一个类中的构造器代理。
- 子类是不会自动继承父类的构造器的，需要使用关键字override来重载父类的构造器。

```
class Student {
    var name : String
    var age : Int
    var id : String
    var basicInfo : String {
        return "\(name) is \(age) years old,
            the id is \(id)"
    }
    init(){
        name = "no name"
        age = 16
        id = ""
    }
    convenience init(name : String, age :
        Int, id : String){
        self.init()
        self.name = name
        self.age = age
        self.id = id
    }
}
```

```
class Gradute : Student {
    var supervisor : String
    var researchTopic : String
    override init() {
        supervisor = ""
        researchTopic = ""
        super.init()
    }
    convenience init(name : String, age :
        Int, id : String,supervisor : String,
        researchTopic : String) {
        self.init(name : name, age : age,
            id : id)
        self.supervisor = supervisor
        self.researchTopic = researchTopic

    }
}
```

```
var theGraduate = Gradute(name : "Tom", age :
    29, id : "BY0602115",supervisor: "Ian",
    researchTopic: "Petri net")
```

```
▼{name "Tom", age 29, id "BY0602115"}
    name "Tom"
    age 29
    id "BY0602115"
    supervisor "Ian"
    researchTopic "Petri net"
```

代码参见iosSwift的Chapter13-7

析构器

- 只有引用类型——类，才有析构器。
- 当一个类的实例使用结束，要被释放之前，析构器就会被调用。
- 定义析构器要使用关键字deinit，定义方式和构造器类似。每个类最多只能有一个析构器，析构器不带任何参数。
- 格式如下：

```
deinit {
                    //release some
resource
}
```

- Swift会自动释放不再需要的实例资源，而不需要手动地去清理。但是，当使用自己的资源时，就需要在析构器里进行一些额外的清理工作了。
  - 例如，创建了一个自定义的类来打开一个文件，并写入了一些数据，此时就需要在通过析构器在类实例被释放之前去手动关闭该文件。
- 析构器不允许被主动调用，它是在系统释放实例资源前被自动调用的。
- 子类会继承父类的析构器，在子类析构器实现的最后部分，父类的析构器会被自动调用。
- 如果子类没有定义自己的析构器，父类的析构器也会被自动调用。
- 由于析构器被调用时，实例资源还没有被释放，所以析构器可以访问所有请求实例的属性。

析构器实例

```
class Student {
    var name : String
    var age : Int
    var id : String
    var basicInfo : String {
        return "\(name) is \(age) years old,
            the id is \(id)"
    }
    init(){
        name = "no name"
        age = 16
        id = ""
    }
    convenience init(name : String, age :
        Int, id : String){
        self.init()
        self.name = name
        self.age = age
        self.id = id
    }
    deinit {
        print("call deinit of Class Student")
    }
}
```

```
class Gradute : Student {
    var supervisor : String
    var researchTopic : String
    override init() {
        supervisor = ""
        researchTopic = ""
        super.init()
    }
    convenience init(name : String, age :
        Int, id : String,supervisor : String,
        researchTopic : String) {
        self.init(name : name, age : age,
            id : id)
        self.supervisor = supervisor
        self.researchTopic = researchTopic

    }
    deinit {
        print("call deinit of Class
            Graduate")
    }
}
```

```
var theGraduate : Gradute? = Gradute(name :
    "Tom", age : 29, id : "BY0602115",
    supervisor: "Ian", researchTopic: "Petri
    net")
theGraduate = nil
```

```
call deinit of Class Graduate
call deinit of Class Student
```

代码参见iosSwift的Chapter13-8

第三部分　面向对象基础

本章目的在于掌握面向对象的基础知识，从而为面向对象的程序设计打下基础。内容包括：

- 枚举类型（定义、关联值及原始值的定义和用法）
- 结构体和类（共同点和不同点，以及各自的应用场景）
- 属性（存储属性、计算属性、属性观察器以及类型属性）
- 方法（实例方法、类型方法、方法下标）
- 继承性、基类、子类、重载。
- 构造器和析构器
- 错误处理

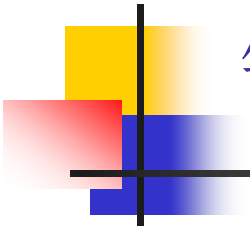
错误定义

- 程序执行过程中，总会出现出错的情况。错误处理就是当程序过程中发生了错误时，捕获错误，并进行适当的错误处理。
- 要处理错误先要定义错误，在Swift中，用遵循协议ErrorType类型的值来表示。
  - ErrorType是一个空协议，专门用作错误处理。一般用枚举类型来定义一组错误情形。
- 在程序中，当发生错误时，通过抛出一个错误来将错误报告给错误处理程序。

```
enum CreatingStudentError : ErrorType {
    case NoName
    case InvalidAge
    case InvalidId
}
```

```
throw CreatingStudentError.InvalidAge
```

代码参见iosSwift的Chapter14-1



错误处理

- 当一个错误被抛出后，将由特定的错误处理程序来对其进行适当的善后，比如向用户提示错误。在Swift中有四种错误处理方式：
  1. 将函数抛出的错误传递给调用该函数的代码来处理。
  2. 通过do-catch结构语句来处理错误。
  3. 将错误定义为可选类型来处理。
  4. 断言错误根本就不会发生。

错误处理（一）

- 第一种情况，要实现将函数中的错误传递给调用该函数的代码，就要用关键字throws来标识这个可能抛出错误的函数。
- 在函数声明的参数列表后面加上throws关键字。我们称一个标识了throws关键字的函数为throwing函数。
- 如果这个函数有返回值，那么throws关键字应该写在符号“->”之前。
- throwing函数从函数内部抛出错误，并将其传递到调用它的地方。

```
func createSomeStudents(var counts : Int) throws -> [Student]{
    var studentsArray = [Student]()
    while counts > 0 {
        let theStudent = Student()
        guard theStudent.age > 15 else {
            throw CreatingStudentError.InvalidAge
        }
        guard theStudent.id.characters.count == 8 else {
            throw CreatingStudentError.InvalidId
        }
        guard !theStudent.name.isEmpty else {
            throw CreatingStudentError.NoName
        }
        studentsArray.append(theStudent)
        counts = counts - 1
    }
    return studentsArray
}

let studentsArray = try createSomeStudents(5)
```

代码参见iosSwift的Chapter14-1

错误处理（二）

- 第二种处理错误的方式就是通过do-catch结构的语句来对各种错误情况逐一进行匹配和处理。将可能抛出错误的语句放到do语句中，然后通过catch语句来匹配各种错误的类型，并进行相应的处理。
- 如果catch语句中没有errorType，那么表示匹配所有错误类型。catch语句不一定要处理do语句中抛出的所有可能的错误。
- 如果没有一个catch语句与错误类型匹配的话，错误会传播到周围的作用域。

具体的格式为：

```
do {
                    try  statement
} catch errorType1 {
                    statements
} catch errorType2 {
                    statements
}
```

```
do {
    let studentsArray = try createSomeStudents(1)
}catch CreatingStudentError.InvalidAge {
    print("Invalid age error!")
}catch CreatingStudentError.InvalidId {
    print("Invalid id error!")
}catch CreatingStudentError.NoName {
    print("No name error!")
}
```

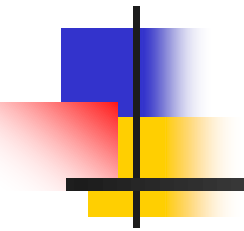
代码参见iosSwift的Chapter14-2

错误处理（三、四）

- 第三种错误处理的情形就是使用try?通过将错误转换成一个可选值来处理。如果在try?语句中抛出了一个错误，那么这个表达式的值为nil。
- 第四种情况就是已知某个throwing函数在运行时不会抛出错误，可以在表达式前用try!使错误传递失效。如果实际中抛出了错误，那么就会得到一个运行时错误。

```
func theThrowingFunction() throws -> Int {
    //some statements
    return 0
}

let x = try? theThrowingFunction()
```

代码参见iosSwift的Chapter14-3



谢　　谢