构造器

构造器是一种特殊的函数，使用保留关键字init来声明。因此构造器函数不需要使用关键字func。通常情况下，构造器也不返回任何值。构造器中需要设置实例中每个存储属性的初始值。

注意：

> 当为存储型属性分配默认值或者在构造器中设置初始值时，他们的值都是被直接设置的，不会触发任何属性观察者。

构造过程中常量属性赋值

可以在构造过程中给常量属性赋值，只要在构造过程结束时该属性设置成确定的值即可。一旦常量属性被赋值，将永远不可更改。

注意：

> 对于类的实例来说，他的常量属性只能在定义他的类的构造过程中修改，不能在子类中修改。

默认构造器

类或者结构体如果所有的属性都有默认值，且没有自定义的构造器，那么Swift会为该结构体/类提供一个默认构造器，该默认构造器会创建一个所有属性都是默认值的实例对象。举例如下：

```
class TestDefault {
    var title = "test"
    var number = 1
}

var t = TestDefault()
print(t.title,t.number)
```

输出：

```
test 1
```

这里，TestDefault()使用的就是默认构造器。

当类中某个属性没有初始化时，不能使用默认构造器，如下：

```
class TestDefault {
    var title = "test"
    var number = 1
    var x: Int
}
```

编译错误，错误信息如下：

```
Class 'TestDefault' has no initializers
```

逐一成员构造器

除默认构造器外，结构体还提供了逐一成员构造器。意思是：假如你没有提供自己的构造器，编译器会为结构体自动生成一个逐一成员构造器。

如下：

```
struct Person {
    var name: String
    var age: Int
}

var p = Person(name: "Jim", age: 18)
```

结构体Person没有实现构造器，编译期为其提供了逐一成员构造器。所谓逐一成员构造器，就是该构造器中，将每个属性都作为了参数。

指定构造器

类里面的所有存储属性，包括所有继承自父类的属性，都必须在构造过程中设置初始值。Swift为类类型提供了两种构造器来确保实例中的所有存储属性都获得初始值，分别是指定构造器和便利构造器。

一个类至少要有一个指定构造器，指定构造器是类中最主要的构造器。通常情况下，一个指定构造器就足够了，当然，声明多个指定构造器也没什么问题。

指定构造器的语法如下：

```
init {
    // statement
}
```

使用关键字init即可。

指定构造器是类的主要构造器，要在指定构造器中初始化所有的属性。

便利构造器

便利构造器的语法如下：

```
convenience init {
    // statement
}
```

在init关键字前面使用convenience。

类中可以有便利构造器，也可以没有便利构造器，对此没有限制。便利构造器，可以根据意思理解，就是为了方便构造，在便利构造器中，通常将参数赋为默认值。

指定构造器和便利构造器的使用原则

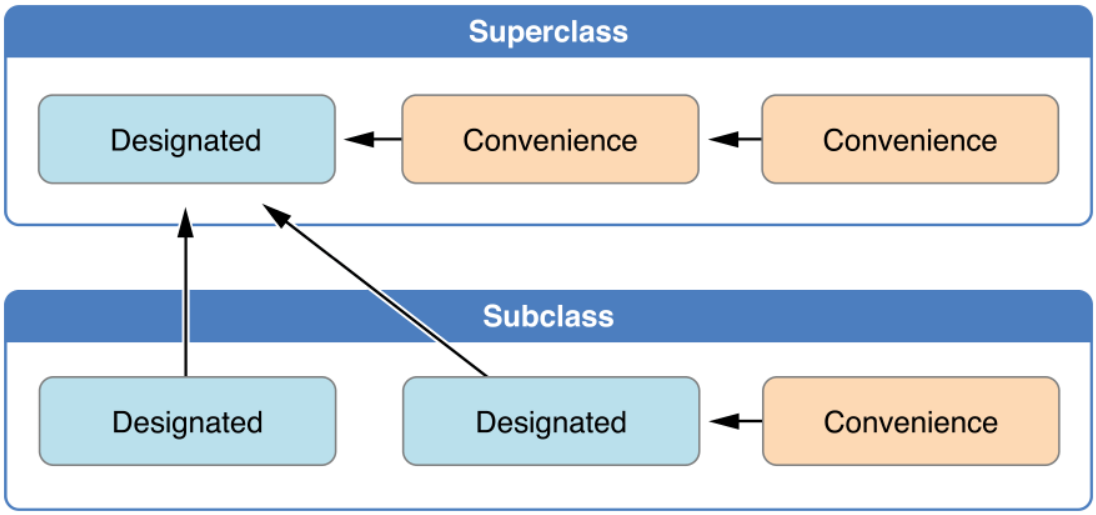
Swift中规定了指定构造器和便利构造器的使用原则，如下：

1. 一个指定构造器必须调用他父类的一个指定构造器
2. 一个便利构造器必须调用这个类自身的另一个构造器（这里既可以是指定构造器，也可以是便利构造器）
3. 一个便利构造器最终一定会调用一个指定构造器

一个简单的记忆方法：

1. 指定构造器总是向上代理
2. 便利构造器总是横向代理

上面这些规则可以用下图说明：



两段式构造过程

Swift 类使用了两段式构造过程，步骤如下：

1. 类中的每个存储属性都获得一个初始值，也就是属性被初始化
2. 给每个类一次机会，在新实例准备使用之前进一步定制存储属性的值

和Objective-C的主要区别在于，OC中所有属性不赋值会被默认初始化，为nil或者0，而Swift中所有属性都需要由开发者来指定。

Swift编译器会执行4种检查，以确保两段式构造过程正确完成：

1. 指定构造器中先初始化自己类中的所有属性，然后再调用父类的指定构造器
2. 指定构造器必须先向上代理调用父类的构造器，然后才能为继承的属性赋值
3. 便利构造器必须先调用同一个类中的其他构造器，然后才能为属性赋值
4. 构造器在第一阶段构造完成之前，不能调用任何实例方法，不能读取任何实例属性，self不能被引用

具体来说，两个阶段的过程如下。

阶段1

1. 类的某个指定构造器或便利构造器被调用
2. 完成类的新实例内存分配，但此时内存还没有被初始化
3. 指定构造器先对自身的存储属性进行赋值，存储型属性的内存完成初始化
4. 指定构造器切换到父类构造器，对其存储属性完成相同的任务
5. 这个过程沿着类的继承链一直向上执行，直到达到继承链的最顶部
6. 当到达了继承链的最顶部，而且继承链的最后一个类已确保所有存储型属性都已经赋值，这个实例的内存被认为已经完全初始化。

此时阶段1完成。

阶段2

1. 从继承链顶部向下，继承链中每个类的指定构造器都有机会进一步自定义实例。构造器此时可以访问self、修改它的属性并调用实例方法
2. 最终，继承链中任意的便利构造器有机会自定义实例和使用self

构造器的继承和重写

和Objective-C不同，Swift中子类默认不会继承父类的构造器。在子类中，如果想提供和父类相同的构造器，可以提供自定义的实现。这实际上相当于是对父类中的某个方法重写，因此需要在前面添加 override 关键字。

注意，即使重写的是父类的默认构造器，也需要添加override关键字。

构造器的自动继承

默认情况下，子类不会继承父类的构造器。但是，在一些特殊情况下，子类可以继承父类的构造器。

前提：**子类中新的属性都设置了默认值**，

以下两个规则将会适用：

1. 如果子类没有定义任何指定构造器，那么子类会自动继承父类所有的指定构造器
2. 如果子类提供了所有父类指定构造器的实现，那么子类自动继承父类所有的便利构造器

需要注意的是规则2，当子类没有定义任何指定构造器，此时子类会自动继承父类所有的指定构造器，这种情况同时也是*子类提供了所有父类指定构造器的实现*，因此：

当子类没有定义任何指定构造器时，子类会自动继承父类所有的指定构造器和便利构造器。

override关键字

子类重写父类中的方法时，必须用override关键字修饰。如果子类重写父类方法时，不使用override修饰符，那么会编译错误，比如：

```
func layoutSubviews() {
        super.layoutSubviews()
}
```

不加override修饰符，编译错误，错误信息：

```
Overriding declaration requires an 'override' keyword
```

如果使用了override修饰符，但是方法并不是父类中的方法，也会编译错误，比如：

```
override func test(){

}
```

父类中并没有test()方法，变比错误，错误信息：

```
Method does not override any method from its superclass
```

可失败构造器

构造器函数后面添加问号，表示可失败构造器。示例如下：

```
struct Person {
    var name: String
    var age: Int

    init?(name: String,age: Int){
        if(age < 0){
            return nil
        }
        self.name = name
        self.age = age
    }
}
```

注意：

> 可失败构造器的参数名和参数类型，不能与其他非可失败构造器的参数名、参数类型相同。否则会编译错误。

必须构造器

如果在类里面使用了required标记构造器，那么该构造器称为必须构造器。该类的所有子类，都需要实现该构造器。如下：

```
class TestDefault {
    var title = "test"
    var number = 1

    required init(title: String, number: Int){
        self.title = title
        self.number = number
    }
}

class CustomDefault: TestDefault {
    var x = 5

    init(x: Int){
        self.x = x
        super.init(title: "", number: 1)
    }
}
```

CustomDefault没有实现必须构造器，会提示错误，错误信息如下：

```
required' initializer 'init(title:number:)' must be provided by
subclass of 'TestDefault'
```

在CustomDefault中添加必须构造器：

```
class CustomDefault: TestDefault {
    var x = 5

    init(x: Int){
        self.x = x
        super.init(title: "", number: 1)
    }

    required init(title: String, number: Int) {
        super.init(title: title, number: number)
    }
}
```

编译没问题。需要注意的是，子类中的必须构造器，同样需要添加required关键字。这样可以在继承链上适用该特性。