The Swift Programming Language in Chinese

https://github.com/letsswift

http://letsswift.com/

June 15, 2014

目录

1 Swift 中文教程（十四）初始化

初始化是类，结构体和枚举类型实例化的准备阶段。这个阶段设置这个实例存储的属性的初始化数值和做一些使用实例之前的准备以及必须要做的其他一些设置工作。

通过定义构造器（initializers）实现这个实例化过程，也就是创建一个新的具体实例的特殊方法。和 Objective-C 不一样的是，Swift 的构造器没有返回值。它们主要充当的角色是确保这个实例在使用之前能正确的初始化。

类实例也能实现一个析构器（deinitializer），在类实例销毁之前做一些清理工作。更多的关于析构器（deinitializer）的内容可以参考 Deinitialization。

1.1 存储属性的初始化

类和结构体必须在它们被创建时把它们所有的属性设置为合理的值。存储属性不能为不确定状态

你可以在构造方法里面给一个属性设置一个初始值，或者在定义的时候给属性设置一个默认值，这个行为将会在接下来的章节描述。

> 注意：当你对给一个属性分配一个默认值的时候，它会调用它相对应的初始化方法，这个值是对属性直接设置的，不会通知它对应的观察者

1.1.1 构造器

构造器是创建一个具体类型实例的方法。最简单的构造器就是一个没有任何参数实例方法，写作 init。

在下面的例子定义了一个叫 Fahrenheit（华氏度）的新结构体，来储存转换成华氏度的温度。Fahrenheit 结构体，有一个属性，叫 temperature（温度），它的类型为 Double（双精度浮点数）：

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
```

```
var f = Fahrenheit()
println("The default temperature is \(f.temperature)° Fahrenheit")
// prints "The default temperature is 32.0° Fahrenheit"
```

这个结构体定义了一个单一的构造方法 init，它没有任何参数，它储存的温度属性初始化为 32.0 度。(水在华氏度的温度情况下的冰点)。

1.1.2 属性的默认值

如上所述，你可以在构造器中设置它自己的储存属性的初始化值。或者在属性声明时，指定属性的默认值，你指定一个默认的属性值，会被分配到它定义的初始值。

> 注意：如果一个属性常常使用同样的初始化值，提供一个默认值会比在初始化使用一个默认值会更好。

同样的结果，但是默认值与属性的初始化在它定义地时候就紧紧地捆绑在一起。很简单地就能构造器更简洁，和可以让你从默认值中推断出这个属性的类型。默认值也能让你优化默认构造器和继承构造器变得更容易，在本章会稍候描述。

你可以在上面的 Fahrenheit（华氏度）结构体定义时，给 temperature（温度）属性提供默认值。

```
struct Fahrenheit {
    var temperature = 32.0
}
```

1.2 自定义初始化（Customizing Initialization）

你可以根据输入的参数来自定义初始化过程和可选的属性类型，或者在初始化的时候修改静态属性。在这章节将会详细叙述。

1.2.1 初始化参数

你可以在构造器定义的时候提供一部分参数，在自定义初始化过程中定义变量的类型和名称。初始化参和函数或者方法参数一样有着同样的功能。

在下面的例子中，定义了一个结构体 Celsius。储存了转换成摄氏度的温度，Celsius 结构体实现了从不同的温度初始化结构体的两个方法，init(fromFahrenheit:) 和 init(fromKelvin:)。

```
struct Celsius {
    var temperatureInCelsius: Double = 0.0
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```

第一个构造器只有一个初始化参数，形参（External Parameter Names）fromFahrenheit，和实参（Local Parameter Names）fahrenheit。第二个构造器有一个单一的初始化参数，形参（External Parameter Names）fromKenvin，和实参（Local Parameter Names）kelvin。两个构造器都把单一的参数转换为摄氏度和储存到一个 temperatureInCelsius 的属性.

1.2.2 实参名（Local Parameter Names）和形参名（External Parameter Names）

和函数参数和方法参数一样，初始化参数拥有在构造器函数体使用的实参，和在调用时使用的形参.

然而，和函数或者方法不同，构造器在圆括号前面没有一个识别函数名称。因此，构造器参数的名称和类型，在被调用的时候，很大程度上扮演一个被识别的重要角色。为此，在构造器中，当你没有提供形参名时，Swift 就会为每一个参数提供一个自动的形参名。这个形参名和实参名相同，就像和之前你写的每一个初始化参数的 hash 符号一样。

> 注意：如果你在构造器中没有定义形参，提供一个下横线（_）作为区分形参和上面说描述的重写默认行为。

在下面的例子，定义了一个结构体 Color，拥有三个静态属性 red，green 和 blue。这些属性储存了从 0.0 到 1.0 的值，这些值代表红色，绿色和蓝色的深度。

Color 提供了一个构造器，以及三个双精度（Double）类型的参数：

```
struct Color {
    let red = 0.0, green = 0.0, blue = 0.0
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
}
```

无论什么时候，你创建一个 Color 实例，你必须使用每一个颜色的形参来调用构造器：

let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)

值得注意的是，不能不通过形参名来调用构造器。在构造器定义之后，形参名必须一致使用。如果漏掉就会在编写时提示错误。

```
let veryGreen = Color(0.0, 1.0, 0.0)
// this reports a compile-time error - external names are required
```

1.2.3 可选类型

如果你储存属性使用的是自定义的类型在逻辑上允许值为空 -或者他们的值并不在构造器中初始化，或者他们被允许为空。可以定义一个可选类型的属性。可选类型属性是一个自动初始化值为 nil，表示这个属性有意在构造器中设置为“空值”（no value yet）。

在下面的例子中，定义了一个 SurveryQuestion 类，拥有一个可选的 String 属性 response。

这个回答在他们调查问题在发布之前是无法知道的，所以 response 定义为类型 String? ，或者叫可选 String（optional String）。说明它会被自动分配一个默认值 nil，意思为当 surverQuestion 初始化时还不存在。

1.2.4 在初始化时修改静态属性

当你在设置静态属性值时，只要在初始化完成之前，你都可以在初始化时随时修改静态属性。

> 注意：对于类的实例化，一个静态属性只能在初始化时被修改，这个初始化在类定义时已经确定。

你可以重写 SurveryQuestion 例子，对于问题的 text 属性，使用静态属性会比动态属性要好，因为 SurveyQuestion 实例被创建之后就无法修改。尽管 text 属性现在是静态的，但是仍然可以在构造器中被设置：

```
class SurveyQuestion {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        println(text)
    }
}
let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask()
// prints "How about beets?"
beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

1.3 默认构造器

Swift 为每一个结构或者基类提供了默认的构造器，来初始化它们所包含的所有属性。默认构造器将会创建一个新的实例然后将它们的属性设置为默认值。

下面的例子定义了一个叫 ShoppingListItem 的类，包含了名称，数量和是否已购买的属性，将会被用在购物清单中：

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
```

```
    var purchased = false
}
var item = ShoppingListItem()
```

因为 ShoppingListItem 类中所有的属性都有默认值，并且这个类是一个没有父类的基类，所以它默认拥有一个会将所有包含的属性设置为初始值的默认构造器。比如在这个例子中 name 属性是一个可选 String 属性，它会被默认设置为 nil，尽管在代码中没有指明。上面的例子使用默认构造器创建了一个 ShoppingListItem 类，记做 ShoppingListItem()，然后将它赋值给了变量 item。

1.3.1 结构类型的成员逐一构造器

除了上面提到的默认构造器之外，结构类型还有另外一种成员逐一完成初始化的构造器，可以在定义结构的时候直接指定每个属性的初始值。

成员逐一构造器是一种为结构的成员属性进行初始化的简便方法。下面的例子定义了一个叫 Size 的结构，和两个属性分别叫 width 和 height。每个属性都是 Double 类型的并且被初始化为 0.0。

因为每个存储属性都有默认值，在 Size 结构创建一个实例的时候就可以自动调用这个成员逐一构造器 init(width:height:):

```
struct Size {
    var width = 0.0, height = 0.0
}
let twoByTwo = Size(width: 2.0, height: 2.0)
```

1.4 数值类型的构造器代理

在实例的初始化过程中，构造器可以调用其他的构造器来完成初始化。这个过程叫构造器代理，可以避免多个构造器的重复代码。

对于数值类型和类来说，构造器代理的工作形式是不一样的。数值类型（结构和枚举）不支持继承，因此他们的构造器代理相对简单，因为它们只能使用自己的构造器代理。但是一个类可以继承自另外一个类，所以类需要确保在初始化的时候将它所有的存储属性都设置为正确的值。这种过程在下一节类的继承和初始化中叙述。

对于数值类型来说，可以使用 self.init 来调用其他构造器，注意只能在这个数值类型内部调用相应的构造器。

需要注意的是如果你为数值类型定义了一个构造器，你就不能再使用默认构造器了。这种特性可以避免当你提供了一个特别复杂的构造器的时候，另外一个人误使用了默认构造器而出错。

> 注意：如果你想要同时使用默认构造器和你自己设置的构造器，不要将这两种构造器写在一起，而是使用扩展形式。更多内容可以参考 Extensions 一章。

下面的示例定义了一个结构 Rect 来表示一个几何中的矩形。这个 Rect 结构需要另外两个结构来组成，包括 Size 和 Point，初始值均为 0.0：

```
struct Size {
    var width = 0.0, height = 0.0
}
struct Point {
    var x = 0.0, y = 0.0
}
```

现在你有三种初始化 Rect 结构的方式：直接使用为 origin 和 size 属性初始化的 0 值，给定一个指定的 origin 和 size，或者使用中心点和大小来初始化。下面的例子包含了这三种初始化方式：

```
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

init() 构造器和默认构造器的功能相同。这个构造器不需要任何内容，只是用来在已有其他构造器的时候表示默认构造器的依然存在。调用这个构造器创建的 Rect，根据 Point 和 Size 的结构定义，Point(x: 0.0, y: 0.0) ，Size(width: 0.0, height: 0.0) origin 和 size 都会被设置为 0。

```
let basicRect = Rect()
// basicRect's origin is (0.0, 0.0) and its size is (0.0, 0.0)
```

第二个 Rect 构造器 init(origin:size:) 和成员逐一构造器类似，它使用给定的值来初始化结构的属性：

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
// originRect's origin is (2.0, 2.0) and its size is (5.0, 5.0)
```

第三个构造器 init(center:size) 就更加复杂一些，它首先使用 center 和 size 计算出了 origin 的值，然后调用（或者是使用代理）了 init(origin:size) 构造器，设置 origin 和 size 的值：

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

init(center:size:) 构造器同样可以设置 oringin 和 size 的值，而且使用起来也非常方便，代码也比较简洁因为它使用了已有的一些构造器。

注意：可以参考 Extensions 一章，学习怎样省略 init() 和 init(origin:size:)

1.5 类的继承和初始化

译者注：本小节内容 Apple 从底层解释，十分复杂，建议有需要的读者自行阅读英文原文。

本小节主要的意思就是说：

1. 自定义初始化方法要先调用自己类默认初始化方法，自己重写默认初始化方法要先调用父类默认初始化方法
2. 应该要先调用父类的构造器或者自身的默认构造器，以防止先给属性赋值了然后才调用父类或者自身的默认构造器把以前的赋值覆盖了

一个类的所有存储属性 -包括从父类继承而来的属性 -都必须在初始化的时候设置初始值。

Swift 为 class 类型定义了两种构造器来确保它们所有的存储属性都设置了初始值。这两种方式叫做指定构造器和便捷构造器。

1.5.1 指定构造器和便捷构造器

指定构造器是一个类最主要的构造器。指定构造器通过设置所有属性的初值并且调用所有的父类构造器来根据构造链一次初始化所有的属性。

类所拥有的指定构造器很少，一般只有一个，并且是连接这父类的构造链依次完成构造的。

每个类至少有一个指定构造器，在有些情况下，需要使用继承来从父类中得到该指定构造器，更多内容可以查看后面的 Automatic Initializer Inheritance 章节。

便捷构造器是类的第二种常用构造器。你可以调用同一个类中的指定构造器来定义一个便捷构造器，使用指定构造器来设置相关的参数默认值。你还可以定义一个便捷构造器来创建这个类的实例或者是别的特殊用途。

如果你的类不需要它们，也可以不定义便捷构造器。不过对于常见初始化模型需要快捷方式的时候创建一个便捷构造器可以让你的初始化过程变成十分简单便捷。

1.5.2 构造链

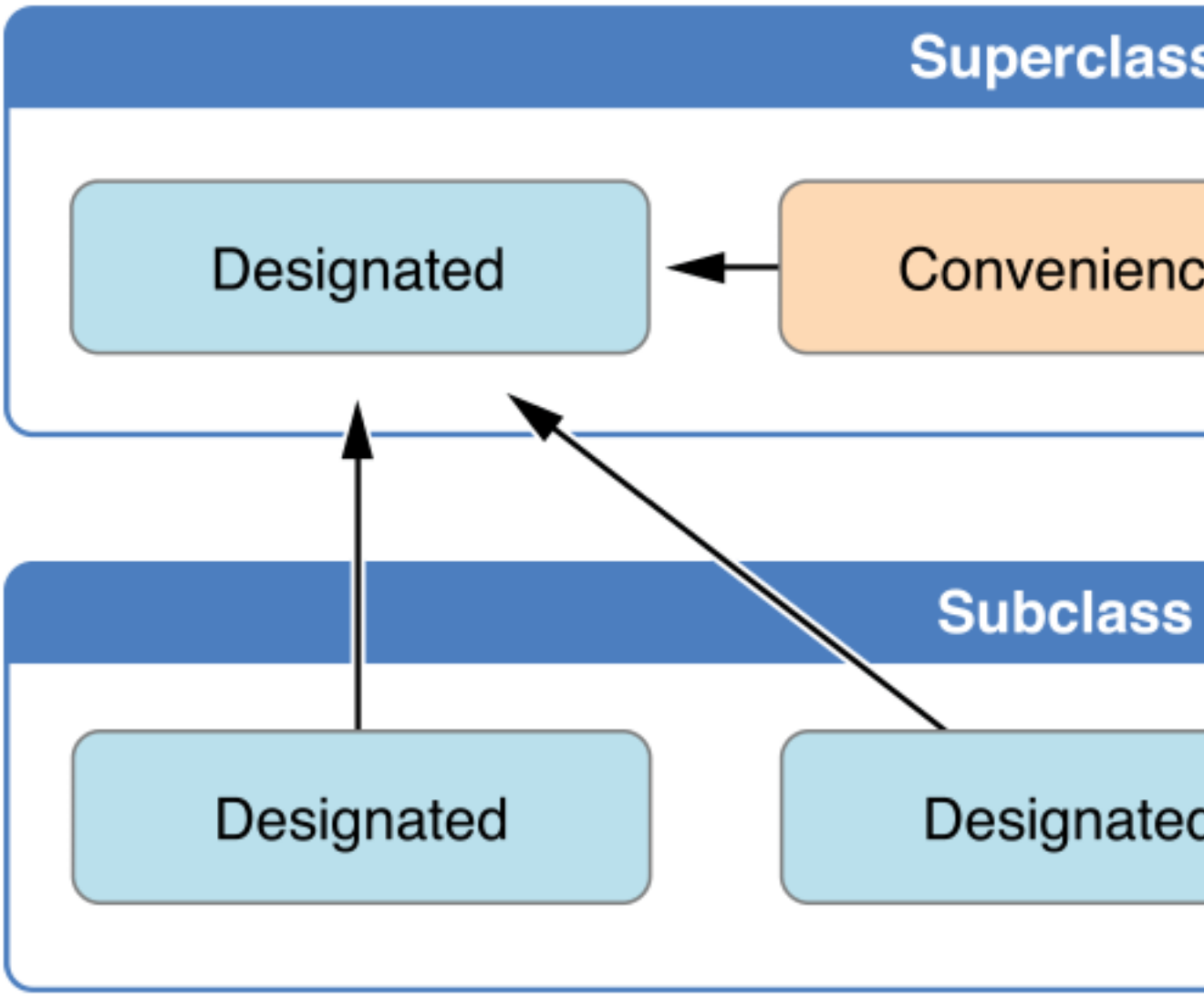
为了简化指定构造器和便捷构造器的关系，Swift 为两种构造器的代理调用设置了三个规则：

- 规则 1 指定构造器必须调用它直接父类的指定构造器
- 规则 2 便捷构造器只能调用同一个类中的其它构造器
- 规则 3 便捷构造器必须以调用一个指定构造器结束

记下这些规则的简单方法是：

- 指定构造器必须向上代理
- 便捷构造器必须横向代理
- 可以使用下面的图来表示：

父类中的两个便捷构造器依次调用直到指定构造器，子类中的指定构造器调用了父类的指定构造器。


Superclass
Designated
Convenienc
Subclass
Designated
Designated

> 注意：这些规则不会影响每个类的实例创建过程。每个构造器都可以用来创建它们各自的类的实例。这些规则只影响你如何编写类实现代码。

下图演示的是另一种更为复杂的具有四个等级的类。这个图展示了指定构造器在类的初始化过程中如何被作为 “漏斗” 节点的。这个构造链简化了类与类之间的交互关系：

1.5.3 两阶段的初始化

在 Swift 中，类的初始化要经过两个阶段。在第一个阶段，每一个存储属性都被设置了一个初始值。一旦每个存储属性的值在初始化阶段被设置了，在第二个阶段，每个类在这个实例被使用之前都会有机会来设置它们相应的存储属性。

两阶段的模式使初始化过程更加安全，还可以让每个类在类的层级关系中具有更多的可能性。两阶段初始化方法可以防止属性在被初始化之前就被使用，或者是被另一个构造器错误地赋值。

> 注意：Swift 的这种两阶段初始化方法跟 Objective-C 中的类似。主要的差别是在第一个过程中，Objective-C 为每个属性赋值 0 或者 null，而在 Swift 中，可以个性化设置这些初始值，还可以处理一些初始值不能是 0 或者 nil 的情况。

Swift 编译器通过四重检查来确保两阶段式的初始化过程是完全正确无误的：

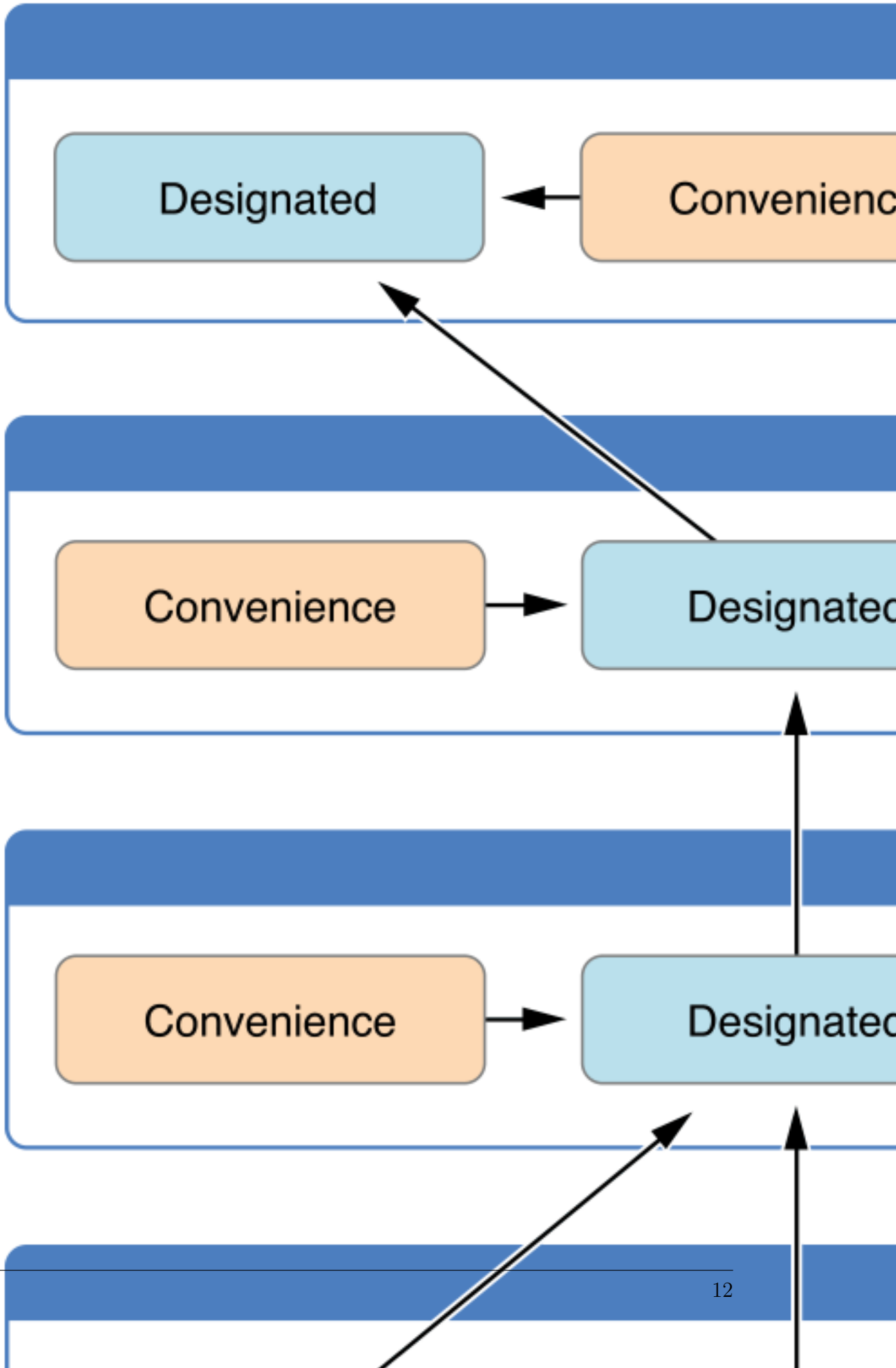
- Safety check 1

A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer. As mentioned above, the memory for an object is only considered fully initialized once the initial state of all of its stored properties is known. In order for this rule to be satisfied, a designated initializer must make sure that all its own properties are initialized before it hands off up the chain.

- Safety check 2

A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property. If it doesn't, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization.

- Safety check 3


Designated
Convenienc
Convenience
Designated
Convenience
Designated

A convenience initializer must delegate to another initializer before assigning a value to any property (including properties defined by the same class). If it doesn't, the new value the convenience initializer assigns will be overwritten by its own class's designated initializer.

- Safety check 4

An initializer cannot call any instance methods, read the values of any instance properties, or refer to selfas a value until after the first phase of initialization is complete. The class instance is not fully valid until the first phase ends. Properties can only be accessed, and methods can only be called, once the class instance is known to be valid at the end of the first phase.

Here's how two-phase initialization plays out, based on the four safety checks above:

- Phase 1

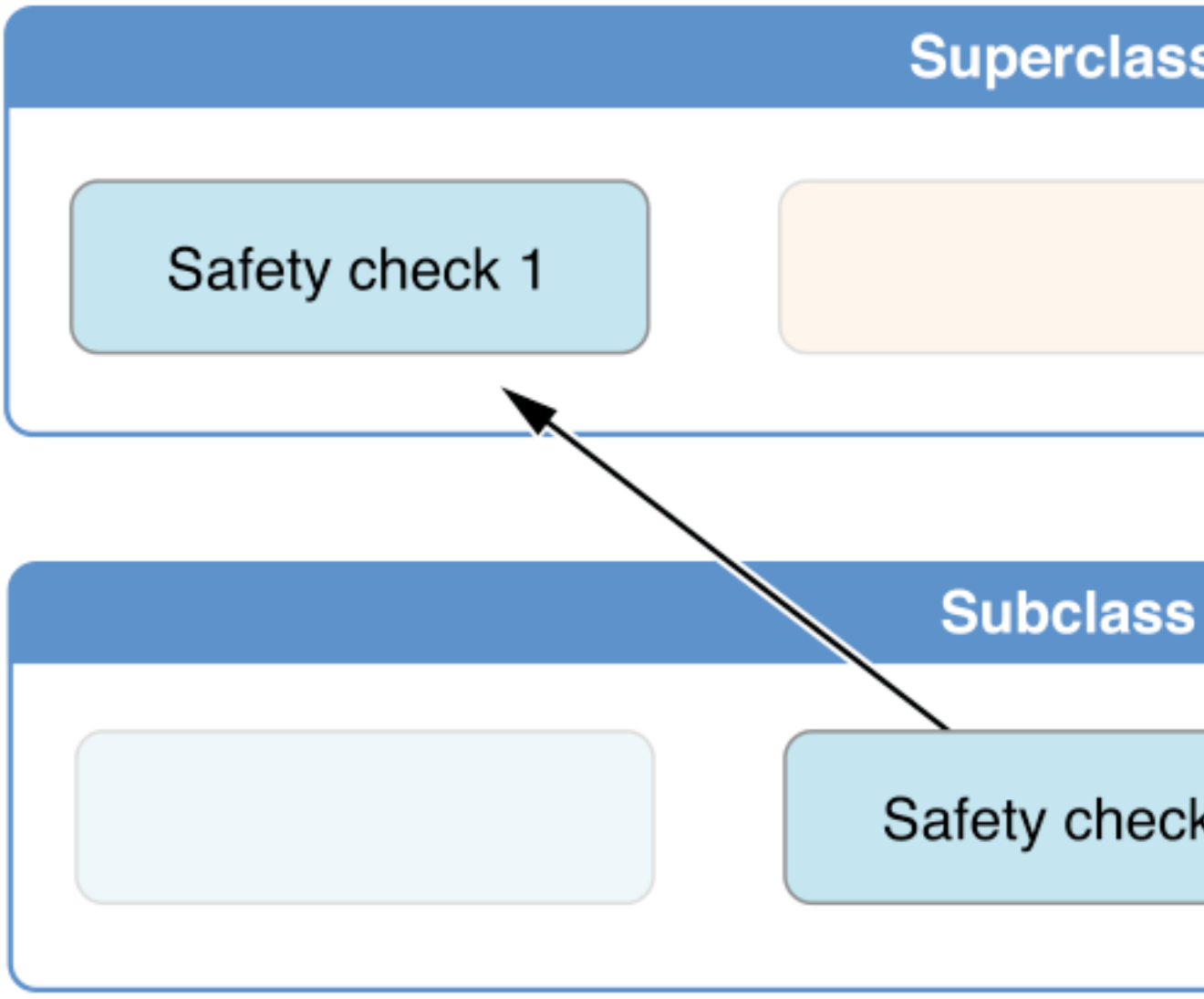
A designated or convenience initializer is called on a class. Memory for a new instance of that class is allocated. The memory is not yet initialized. A designated initializer for that class confirms that all stored properties introduced by that class have a value. The memory for these stored properties is now initialized. The designated initializer hands off to a superclass initializer to perform the same task for its own stored properties. This continues up the class inheritance chain until the top of the chain is reached. Once the top of the chain is reached, and the final class in the chain has ensured that all of its stored properties have a value, the instance's memory is considered to be fully initialized, and phase 1 is complete.

- Phase 2

Working back down from the top of the chain, each designated initializer in the chain has the option to customize the instance further. Initializers are now able to access self and can modify its properties, call its instance methods, and so on. Finally, any convenience initializers in the chain have the option to customize the instance and to work with self. Here's how phase 1 looks for an initialization call for a hypothetical subclass and superclass:

In this example, initialization begins with a call to a convenience initializer on the subclass. This convenience initializer cannot yet modify any properties. It delegates across to a designated initializer from the same class.

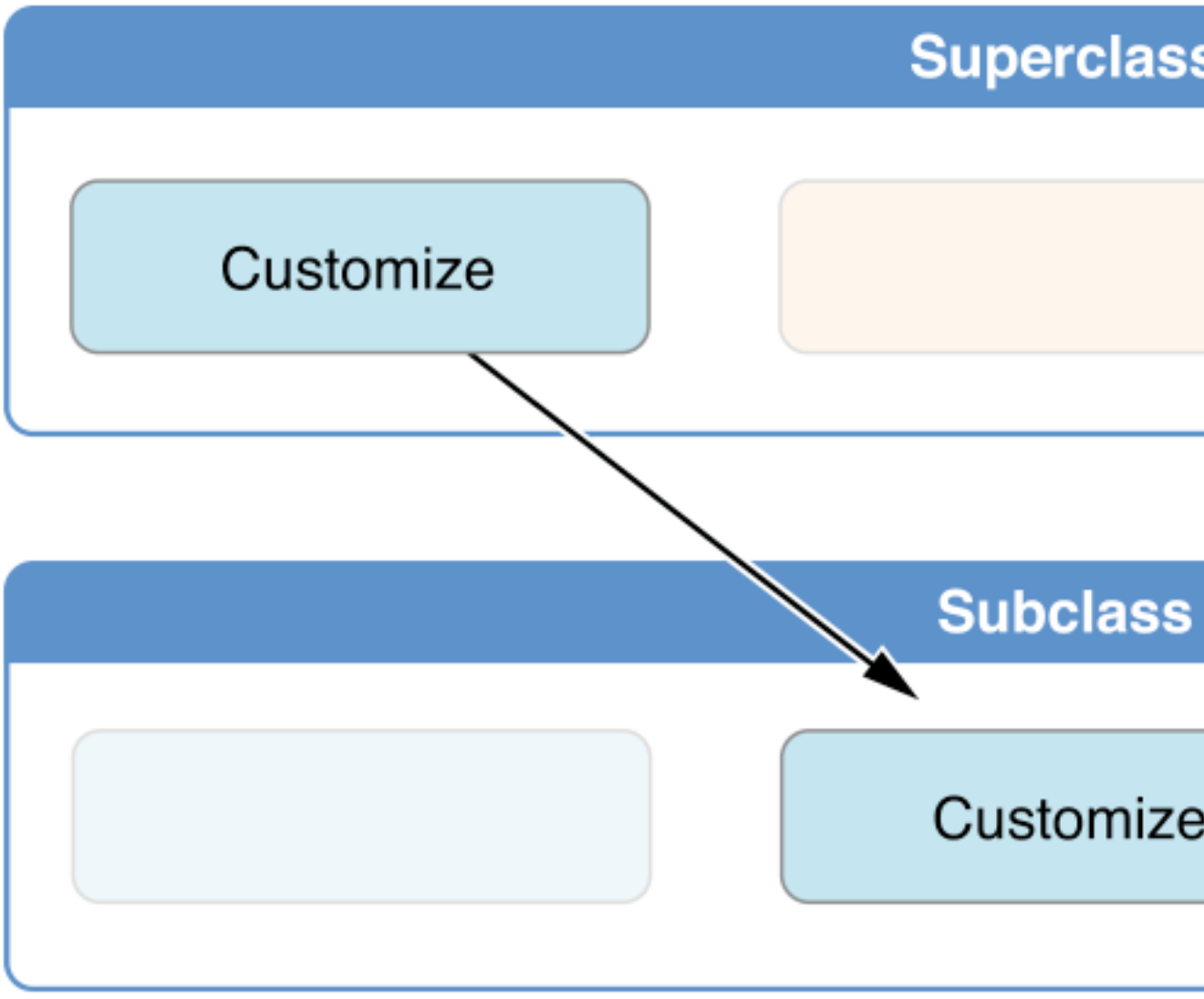
The designated initializer makes sure that all of the subclass's properties have a value, as per safety check 1. It then calls a designated initializer on its superclass to continue the initialization up the chain.


Superclass
Safety check 1
Subclass
Safety check

The superclass's designated initializer makes sure that all of the superclass properties have a value. There are no further superclasses to initialize, and so no further delegation is needed.

As soon as all properties of the superclass have an initial value, its memory is considered fully initialized, and Phase 1 is complete.

Here's how phase 2 looks for the same initialization call:



The superclass's designated initializer now has an opportunity to customize the instance further (although it does not have to).

Once the superclass's designated initializer is finished, the subclass's designated initializer can perform additional customization (although again, it does not have to).

Finally, once the subclass's designated initializer is finished, the convenience

initializer that was originally called can perform additional customization.

1.5.4 构造器的继承和重写

Unlike subclasses in Objective-C, Swift subclasses do not not inherit their superclass initializers by default. Swift's approach prevents a situation in which a simple initializer from a superclass is automatically inherited by a more specialized subclass and is used to create a new instance of the subclass that is not fully or correctly initialized.

If you want your custom subclass to present one or more of the same initializers as its superclass—perhaps to perform some customization during initialization—you can provide an overriding implementation of the same initializer within your custom subclass.

If the initializer you are overriding is a designated initializer, you can override its implementation in your subclass and call the superclass version of the initializer from within your overriding version.

If the initializer you are overriding is a convenience initializer, your override must call another designated initializer from its own subclass, as per the rules described above in Initializer Chaining.

> NOTE Unlike methods, properties, and subscripts, you do not need to write the override keyword when overriding an initializer.

1.5.5 构造器自动继承

As mentioned above, subclasses do not not inherit their superclass initializers by default. However, superclass initializers are automatically inherited if certain conditions are met. In practice, this means that you do not need to write initializer overrides in many common scenarios, and can inherit your superclass initializers with minimal effort whenever it is safe to do so.

Assuming that you provide default values for any new properties you introduce in a subclass, the following two rules apply:

- Rule 1 If your subclass doesn't define any designated initializers, it automatically inherits all of its superclass designated initializers.

- Rule 2 If your subclass provides an implementation of all of its superclass designated initializers—either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition—then it automatically inherits all of the superclass convenience initializers.

These rules apply even if your subclass adds further convenience initializers.

NOTE

A subclass can implement a superclass designated initializer as a subclass convenience initializer as part of satisfying rule 2.

1.5.6 指定初始化和便捷初始化的语法

Designated initializers for classes are written in the same way as simple initializers for value types:

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

Convenience initializers are written in the same style, but with the convenience keyword placed before the initkeyword, separated by a space:

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

1.5.7 指定初始化和便捷初始化实战

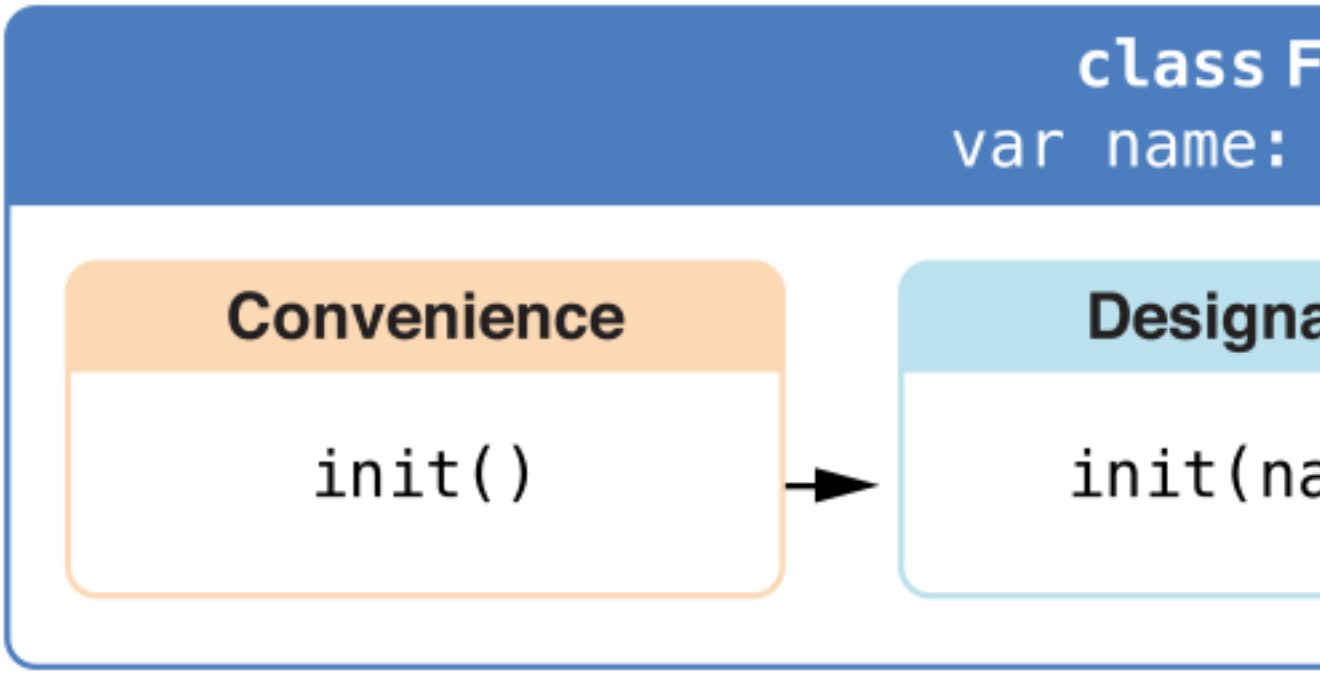
下面的例子演示的是指定构造器，便捷构造器和自动构造器继承的实战。例子中定义了三个类分别叫 Food，RecipeIngredient 和 ShoppingListItem，并给出了他们的继承关系。

基类叫做 Food，是一个简单的类只有一个 name 属性：

```
class Food {
    var name: String
    init(name: String) {
```

```
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

下图就是 Food 类的构造链：



类不存在成员逐一构造器，所以 Food 类提供了一个指定构造器，使用参数 name 来完成初始化：

```
let namedMeat = Food(name: "Bacon")
// namedMeat's name is "Bacon"
```

init(name:String) 构造器就是 Food 类中的指定构造器，因为它保证了每一个 Food 实例的属性都被初始化了。由于它没有父类，所以不需要调用 super.init() 构造器。

Food 类也提供了便捷构造器 init()，这个构造器没有参数，仅仅只是将 name 设置为了 [Unnamed]：

```
let mysteryMeat = Food()
// mysteryMeat's name is "[Unnamed]"
```

下一个类是 Food 的子类，叫做 RecipeIngredient。这个类描述的是做饭时候的配料，包括一个数量属性 Int 类型，然后定义了两个构造器：

```
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

下图表示这两个类的构造链：

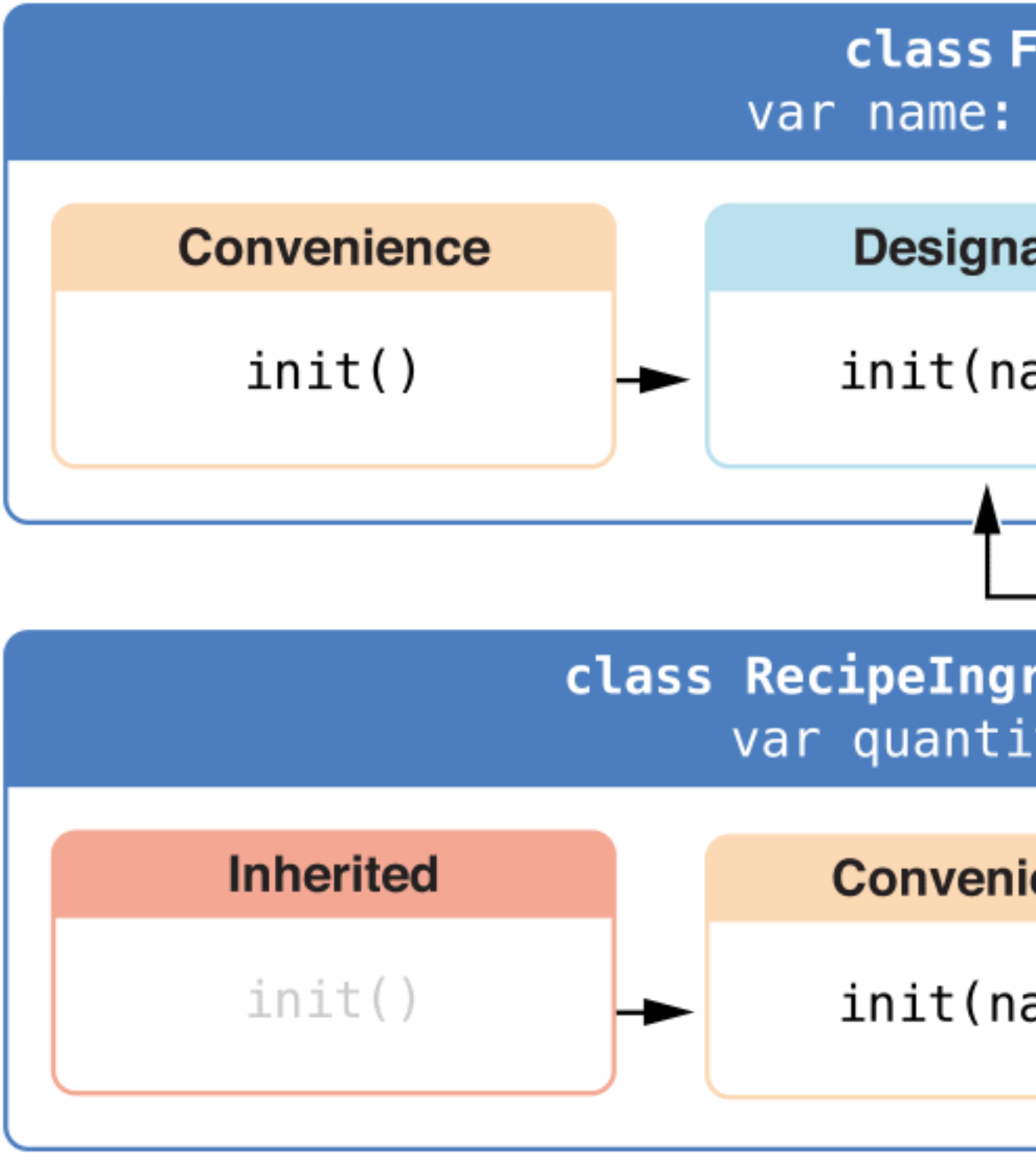
RecipeIngredient 类有它自己的指定构造器 init(name: String, quantity:Int)，用来创建一个新的 RecipeIngredient 实例。在这个指定构造器中它调用了父类的指定构造器 init(name:String)。

然后它还有一个便捷构造器，init(name)，它使用了同一个类中的指定构造器。当然它还包括一个继承来的默认构造器 init()，这个构造器将使用 RecipeIngredient 中的 init(name: String) 构造器。

RecipeIngredient also defines a convenience initializer, init(name: String), which is used to create aRecipeIngredient instance by name alone. This convenience initializer assumes a quantity of 1 for anyRecipeIngredient instance that is created without an explicit quantity. The definition of this convenience initializer makes RecipeIngredient instances quicker and more convenient to create, and avoids code duplication when creating several single-quantity RecipeIngredient instances. This convenience initializer simply delegates across to the class's designated initializer.

Note that the init(name: String) convenience initializer provided by RecipeIngredient takes the same parameters as the init(name: String) designated initializer from Food. Even though RecipeIngredient provides this initializer as a convenience initializer, RecipeIngredient has nonetheless provided an implementation of all of its superclass's designated initializers. Therefore, RecipeIngredient automatically inherits all of its superclass's convenience initializers too.

In this example, the superclass for RecipeIngredient is Food, which has a single convenience initializer calledinit(). This initializer is therefore inherited by


class F
var name:
Convenience
init()
Designa
init(na
class RecipeIngr
var quanti
Inherited
init()
Convenie
init(na

RecipeIngredient. The inherited version of init() functions in exactly the same way as the Food version, except that it delegates to the RecipeIngredient version of init(name: String) rather than the Food version.

上述三种构造器都可以用来创建 RecipeIngredient 实例：

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

最后一个类是 ShoppingListItem 继承自 RecipeIngredient，它又包括了另外两个属性，是否已购买 purchased，描述 description，描述本身还是一个计算属性：

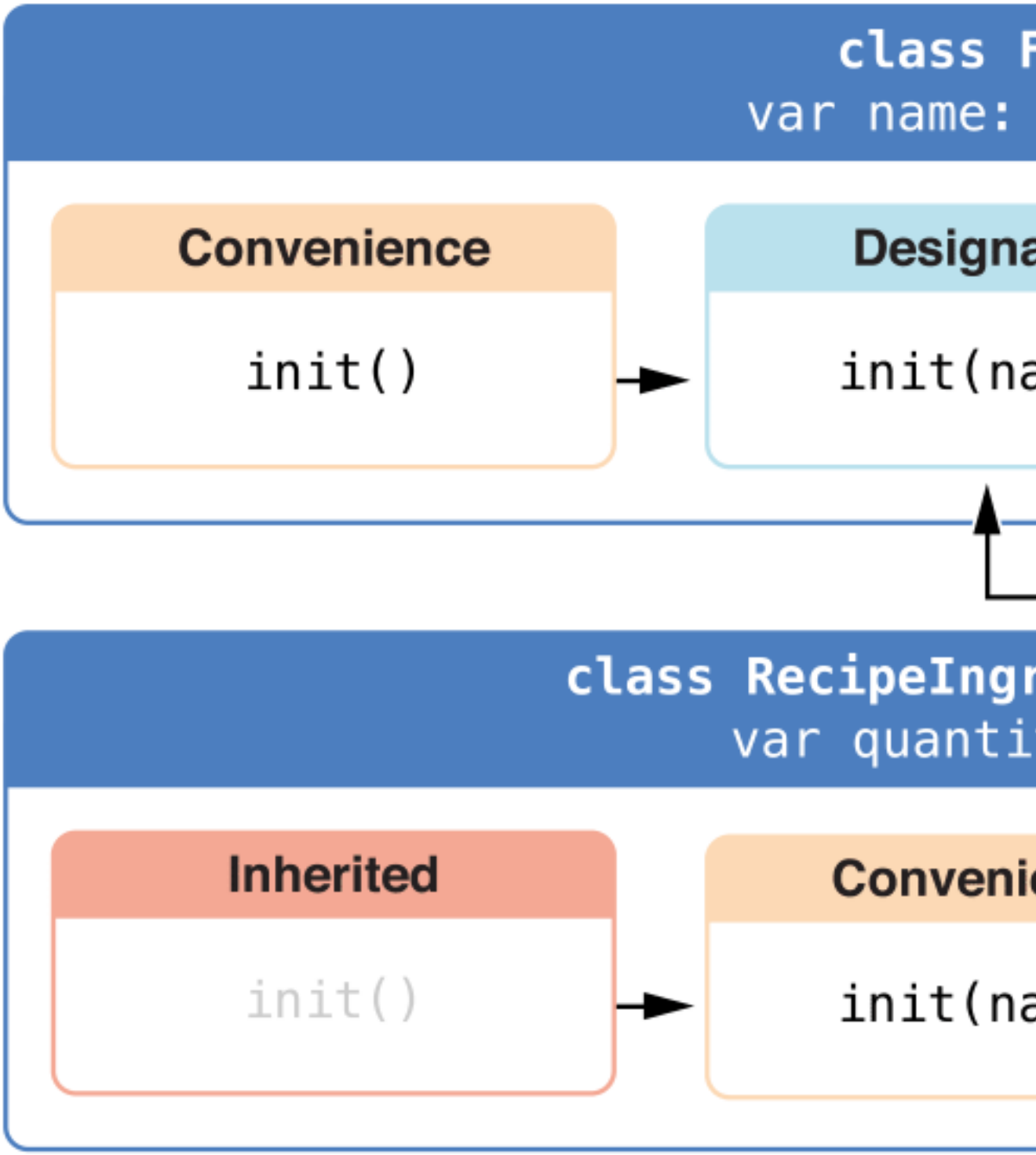
```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
    var output = "\(quantity) x \(name.lowercaseString)"
        output += purchased ? " yes" : " no"
        return output
    }
}
```

> 注意：ShoppingListItem 没有定义构造器来初始化 purchased 的值，因为每个商品在买之前 purchased 都是默认被设置为没有被购买的。

因为 ShoppingListItem 没有提供其他构造器，那么它就完全继承了父类的构造器，用下图可以说明：

你可以在创建 ShoppingListItem 实例时使用所有的继承构造器：

```
var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
```


Convenience
init()
Designa
init(na
Inherited
init()
Convenie
init(na
Inherited
Inherit

```
    println(item.description)
}
// 1 x orange juice yes
// 1 x bacon no
// 6 x eggs no
```

通过输出可以看出所有的实例在创建的时候，属性的默认值都被正确的初始化了。

1.6 通过闭包或者函数来设置一个默认属性值

如果存储属性的默认值需要额外的特殊设置，可以使用闭包或者函数来完成。

闭包或者函数会创建一个临时变量来作为返回值为这个属性赋值。下面是如果使用闭包赋值的一个示意代码：

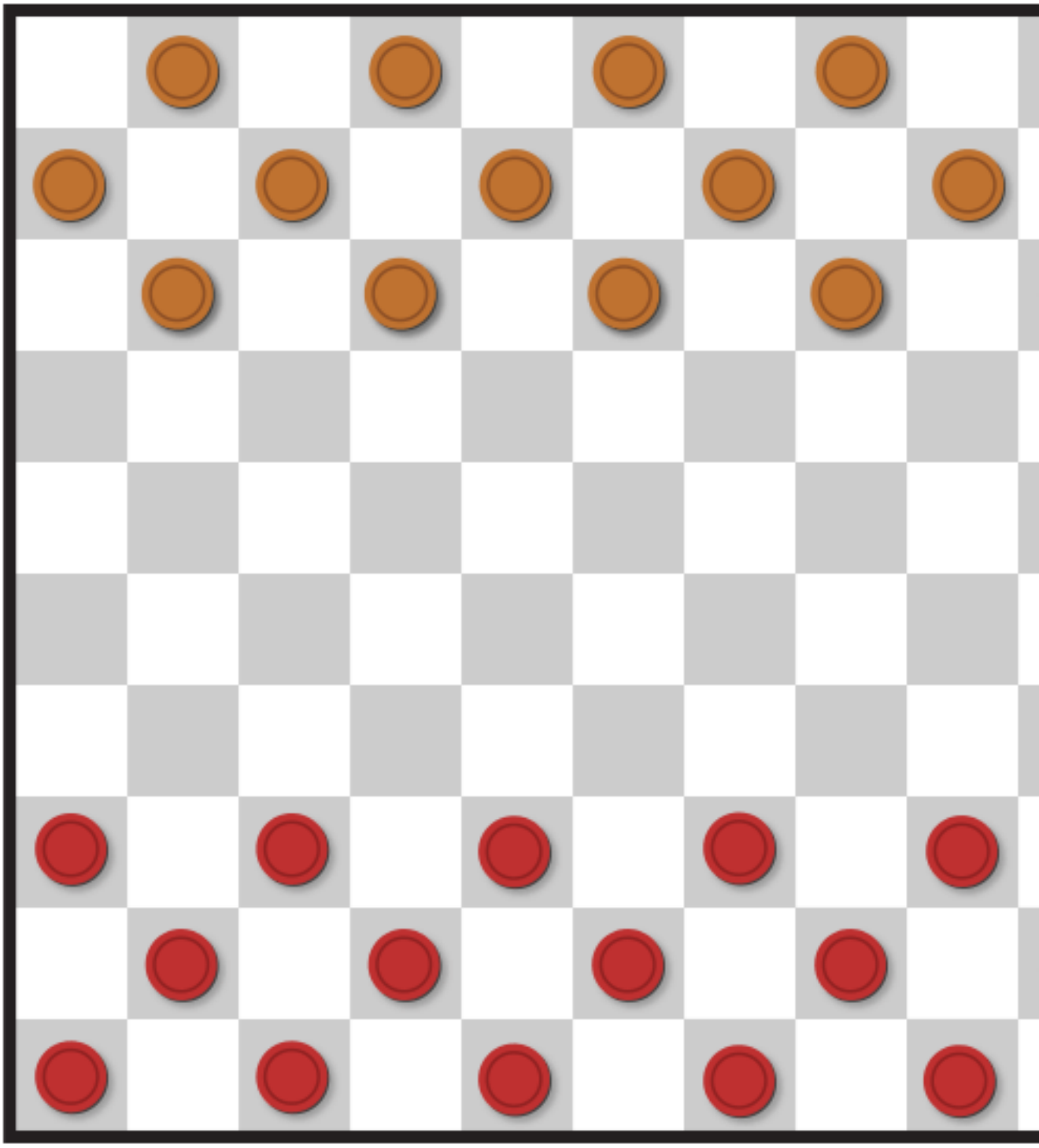
```
class SomeClass {
    let someProperty: SomeType = {
       // create a default value for someProperty inside this closure
        // someValue must be of the same type as SomeType
        return someValue
        }()
}
```

需要注意的是在闭包结尾有两个小括号，告诉 Swift 这个闭包是需要立即执行的。

> 注意：如果你时候闭包来初始化一个属性，在闭包执行的时候，后续的一些属性还没有被初始化。在闭包中不要访问任何后面的属性，一面发生错误，也不能使用 self 属性，或者其它实例方法。

下面的例子是一个叫 Checkerboard 的结构，是由游戏 Checkers 来的

这个游戏是在一个 10×10 的黑白相间的格子上进行的。来表示这个游戏盘，使用了一个叫 Checkerboard 的结构，其中一个属性叫 boardColors，是一个 100 个 Bool 类型的数组。true 表示这个格子是黑色，false 表示是白色。那么在初始化的时候可以通过下面的代码来初始化：



```
struct Checkerboard {
    let boardColors: Bool[] = {
        var temporaryBoard = Bool[]()
        var isBlack = false
        for i in 1...10 {
            for j in 1...10 {
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
            isBlack = !isBlack
        }
        return temporaryBoard
        }()
    func squareIsBlackAtRow(row: Int, column: Int) -> Bool {
        return boardColors[(row * 10) + column]
    }
}
```

当一个新的 Checkerboard 实例创建的时候，闭包会执行，然后 boardColor 的默认值将会被依次计算并且返回，然后作为结构的一个属性。通过使用 squareIsBlackAtRow 工具函数可以检测是否被正确设置：

```
let board = Checkerboard()
println(board.squareIsBlackAtRow(0, column: 1))
// prints "true"
println(board.squareIsBlackAtRow(9, column: 9))
// prints "false"
```

参考文献