

欢迎回来继续我们的学习。

现在我们的to-do 清单上还有不少事情要做，接下来先处理一个比较简单的：产生一个随机数，然后让它显示在屏幕上。

在游戏开发的时候，我们经常会用到随机数。当然，大家最感兴趣的随机数可能就是体彩双色球了，可惜的是彩票里面的所谓随机数未必随机啊，主任才是决定一切的力量~

自买彩票十几年以来，哥中的最大奖是210元。

当时就开始各种幻想，以为千万头奖近在咫尺，从此走上人生巅峰，可谁知道从此好运远离，连中个5元都得一等再等。

算了还是别做梦了，来说说游戏里面的随机数吧。

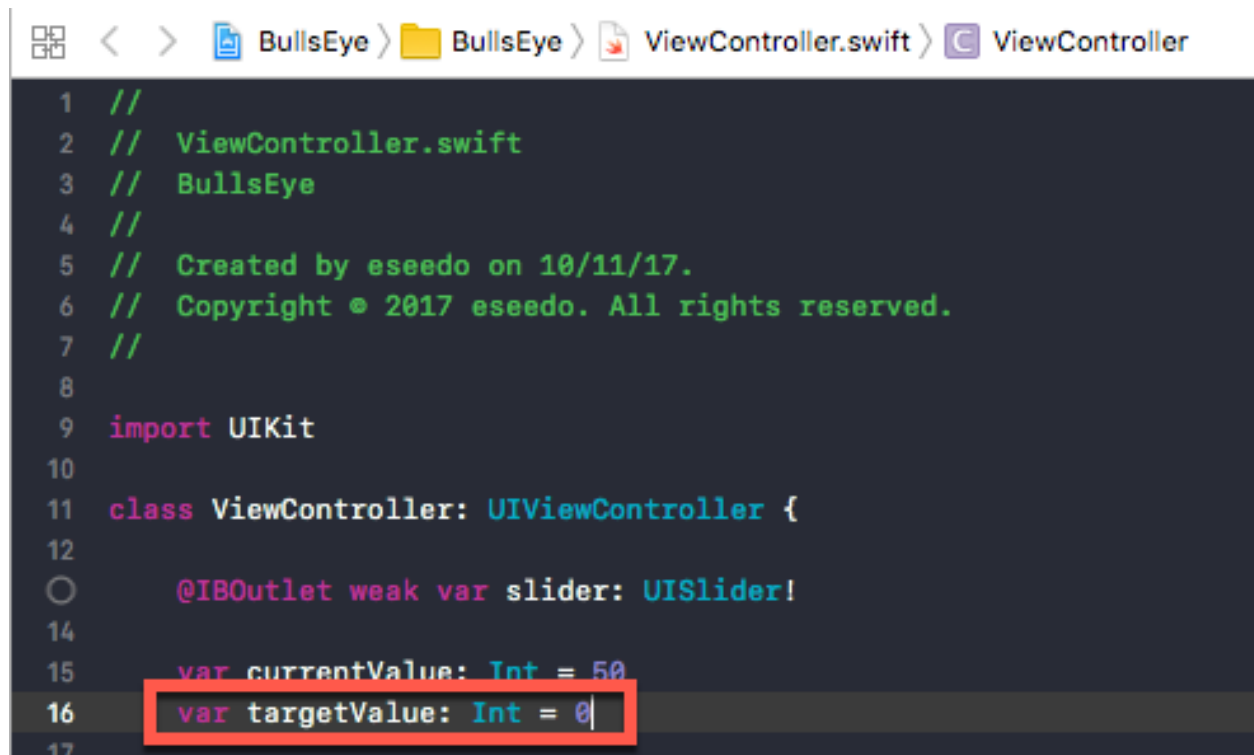
在此之前提一句，如果哪位哥中了头奖，别忘了留个联系方式，土豪我们做个朋友吧！

实际上，我们不可能让一个计算机真正产生完全随机不可预测的数字，而使用所谓的伪随机数(pseudo-random)生成器来产生貌似是随机的效果。这里我们将用到一个比较常用的，arc4random\_uniform()函数。

你应该还记得函数和方法的区别吧？不怕麻烦再重复一遍，函数不属于某个对象，而方法则必须寄生于某个对象。

在产生随机数之前，首先我们需要定义一个变量来保存该随机数。

在Xcode中打开ViewController.swift，在currentValue这个变量的定义语句之后添加一行代码：



```
1 //
2 // ViewController.swift
3 // BullsEye
4 //
5 // Created by eseedo on 10/11/17.
6 // Copyright © 2017 eseedo. All rights reserved.
7 //
8
9 import UIKit
10
11 class ViewController: UIViewController {
12
13     @IBOutlet weak var slider: UISlider!
14
15     var currentValue: Int = 50
16     var targetValue: Int = 0
17 }
```

如果我们没有告诉编译器targetValue是怎样的一种变量，那么它就不知道该为targetValue分配多少存储空间，更不会知道我们是否正确的使用了该变量。此外，Swift中的变量必须有一个数值，因此我们赋予其一个初始值0.当然，0这个数值在实际的游戏中并不会用到，我们会使用所产生的随机数来替代它。还记得之前提过的变量类型吗？这里我们所定义的目标Value就是一个实例变量。之所以将其定义为一个实例变量，是因为我们需要在多处用到它，比如在viewDidLoad()中，同时在用户触碰按钮之前记住该随机数，以及在 showAlert()方法将数值和用户所选择的数值进行比较预算。

接下来我们就需要生成这个随机数了。  
在我们这个游戏里，最适合生成随机数的地方当然就是游戏开始处。

在Xcode中打开ViewController.swift，然后在viewDidLoad()方法中添加下面的这行代码：

```
targetValue = 1 + Int(arc4random_uniform(100))
```

此时viewDidLoad()方法的完整代码如下：

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    currentValue = lroundf(slider.value)  
    targetValue = 1 + Int(arc4random_uniform(100))  
}
```

这里我们调用了arc4random\_uniform函数来产生一个随机整数，并将targetValue的最终数值范围限定在1到100之间。

实际上我们通过arc4random\_uniform函数能得到的最大数字是99，因为arc4random\_uniform()会排除超过上界的数值。这里我们设置的上界是100，因此我们得到的随机数在0-99之间（包含99但不包含100）。为了得到1-100之间的数字，我们需要在arc4random\_uniform()的结果上加上1。

希望你别被上面的数学运算吓倒。

虽然很多人说只有学好数学才能成为一个好的程序猿，但实际上绝大多数应用中所用到的数学都不会比上面的算术复杂。当然，在游戏和科学计算中用到的数学要稍微复杂一些。不过你的目的并不是成为一个算法高手或者数学家，所以也不必太担心。

接下来，让我们更改showAlert()的方法如下：

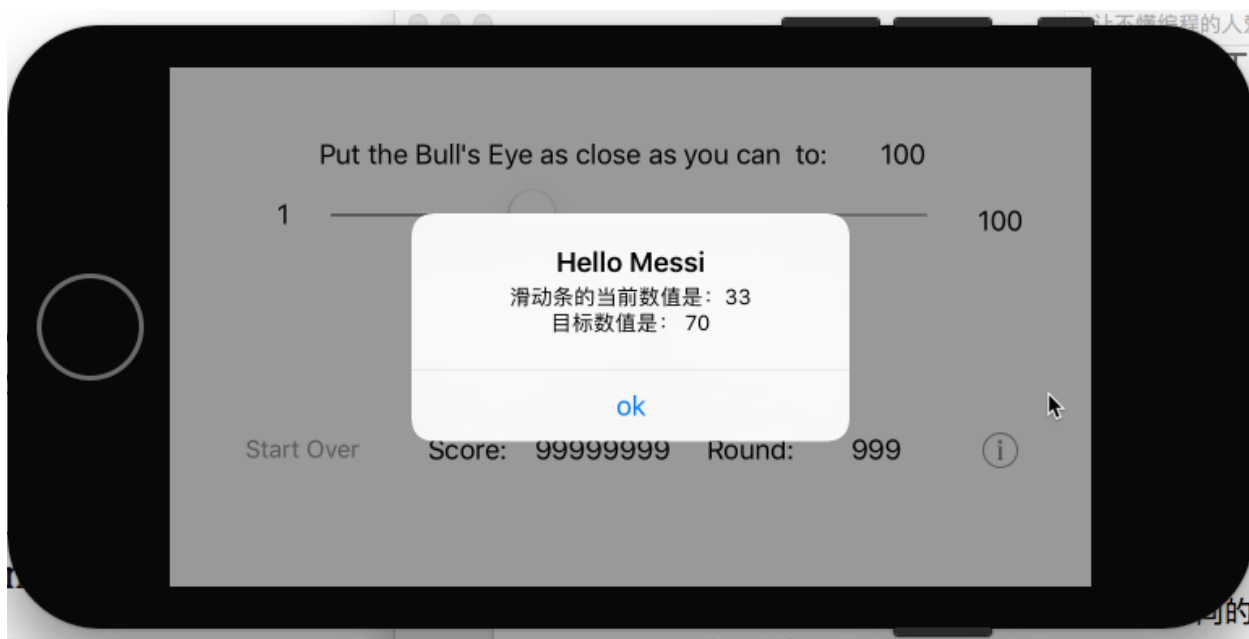
```
@IBAction func showAlert(){  
    let message = "滑动条的当前数值是： \(currentValue)" +  
        "\n目标数值是： \(targetValue)"  
  
    let alert = UIAlertController(title:"Hello Messi",  
        message:message,  
        preferredStyle: .alert)  
    let action = UIAlertAction(title:"ok",style: .default,handler: nil)  
    alert.addAction(action)  
  
    present(alert, animated: true, completion: nil)  
}
```

后面就不重复提醒大家了，黄色高亮部分代表修改或新增的代码。

在上面的代码中，我们向字符串中添加了存储在targetValue中的随机数，这个大家应该不再陌生了。\\(targetValue)将被真实的随机数替代。

\\n这个符号之前没接触过。实际上它在字符串中出现的时候，就意味着我们要换行了。

好了，点击Run运行游戏，试试看有什么变化。



注意：之前我们曾经使用+这个符号来将两个数字相加（和大家在小学数学里面学到的那样），但这里我们则使用+来将文本粘合到一起形成一个长的字符串。

在编程语言中，相同的符号有时代表着不同的作用，而具体的含义由上下文决定（跟人类

语言一样）。

## 添加新的游戏回合

如果你多试几次，会发现对话框所提示的随机数目标数值几乎就没改变过。这样的话这游戏也就没啥意思了。

还好意思说自己是随机数，一次都不变。这就好比双色球和大乐透的每期中奖号码都是重复的，还有谁愿意买？（我！！！）

这是因为我们在viewDidLoad()方法里面生成了随机数，而之后就没有再次生成。而这个方法只会在应用第一次打开创建视图控制器的时候才会调用，之后就没它的事了。我们的to-do清单上写的是，“在游戏的每个回合开始的时候生成一个随机数”。好吧，这里的回合指的是什么呢？

当游戏开始的时候，玩家的初始分数是0，游戏回合数是1。我们把滑动条设置在中间的位置（50），然后计算一个随机数。接着我们等玩家触碰按钮。当玩家触碰按钮之后，一轮游戏就结束了。

这个时候我们会计算这一轮的得分，然后把它添加到总得分里面去。接着我们让游戏回合数加1，然后开启新一轮游戏。我们再次把滑动条重置到中间的位置，然后计算一个新的随机数。就这样，只要你愿意，直到地老天荒，直到宇宙的终结（实际上直到你的手机没电为止~）。

每当你发现自己在想，“在新一轮的游戏开始后我们要做这个，做那个。。。”，这个时候就该创建一个新的方法了。这个方法会创建自己的独有功能。

记住这一点，让我们继续前进吧。

回到Xcode,切换到ViewController.swift，添加一个新的方法。

具体放的位置不是很重要，只要它是在class ViewController的花括号里面，这样编译器就知道它属于ViewController对象了。

```
func startNewRound() {  
  
    targetValue = 1 + Int(arc4random_uniform(100))  
    currentValue = 50  
    slider.value = Float(currentValue)
```

```
}
```

这里和我们之前所做的事情没有太大区别，只不过我们把开始新一轮游戏的逻辑机制放到了一个独立的方法startNewRound()里面去。这样做的好处是，我们可以在后面重复使用这部分代码的逻辑。

然后更改viewDidLoad()方法的代码如下：

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    startNewRound()  
}
```

注意我们已经把viewDidLoad()方法中原有的相关代码删除，并使用对startNewRound()方法的调用来代替。

当玩家触碰按钮显示提示对话框后，可以在 showAlert() 方法里面调用这个方法开启新一轮游戏。

让我们更改 showAlert() 方法里面的代码：

```
@IBAction func showAlert(){  
  
    let message = "滑动条的当前数值是： \(currentValue)" +  
        "\n目标数值是： \(targetValue)"  
  
    let alert = UIAlertController(title:"Hello Messi",  
                                message:message,  
                                preferredStyle: .alert)  
    let action = UIAlertAction(title:"ok",style: .default,handler: nil)  
    alert.addAction(action)  
  
    present(alert, animated: true, completion: nil)  
  
    startNewRound()  
}
```

这里添加的唯一一行新代码就是最后的startNewRound()

在此之前，我们所接触到的视图控制器里面的方法都是在某个事件发生的时候调用：比如当应用加载视图控制器的时候调用viewDidLoad()方法，当玩家触碰按钮的时候调用 showAlert() 方法，当玩家拖动滑动条的时候调用sliderMoved()方法。这就是我们之前谈

到的事件驱动模型。

以上的方法调用都是自动的，而这里开启新一轮游戏则是手动调用方法的例子。我们从对象中的某个方法给相同对象的另一个对象发送了一条消息，听起来似乎有点拗口。

秘密：在iOS开发里面，发送消息和调用方法其实是一个意思~

比如在这里，视图控制器给自己发送一条startNewRound()的消息以开启新一轮游戏。

然后iPhone就会切换到新的方法，并从头到尾执行里面的语句。当方法里面的语句执行完毕后，就会返回之前的方法，继续其中的代码。可能是第一次调用时的viewDidLoad()方法，也可能是此后每一个回合的 showAlert()方法。

有时你可能看到类似下面的方法调用方式：

```
self.startNewRound()
```

这个是之前不带self的方法调用作用相同。还记得刚才我提到视图控制器给自己发送消息吗？其实这就是self的意思。

在Swift中，当我们需要对某个对象调用方法时，通常这样来写：

```
receiver.methodName(parameters)
```

这里的receiver就是我们要发送消息的接收对象。如果我们要给自己发送消息，receiver就是self。不过因为给self发送消息是非常常见的事情，所以大部分时候这个关键词可以省略掉。

不过实际上这并非我们首次调用方法。之前的addAction()方法是UIAlertController，而present()方法则是所有视图控制器都会有的，包括你自己的视图控制器。

当我们使用Swift编程的时候，大部分时候都是让对象调用方法，因为这就是我们应用中对象的通讯方式。

这里要说明一点，把开启新一轮游戏的逻辑机制放到一个独立的方法里面是很好的编码习惯。如果我们不这样做，你就会看到下面的代码：

```
override func viewDidLoad()
{
    super.viewDidLoad()
```

```

    targetValue = 1 + Int((arc4random())_uniform(100))
    currentValue = 50
    slider.value = Float(currentValue)
}

@IBAction func showAlert()
{
    ...

    presentViewController(alert,animated:true,completion:nil)
    targetValue = 1 + Int(arc4random_uniform(100))
    currentValue = 50
    slider.value = Float(currentValue)
}

```

这样做有什么不好呢？首先，我们在两个不同的地方重新写了相同的代码。好吧，这里只是3行代码，如果是300行怎么办？如果我们打算更改这部分逻辑又该怎么办？如果我们要在10个不同的地方用这部分代码怎么办？

还有，你刚开始写这段代码的时候非常清楚它是干吗用的。但1周以后呢？1个月以后呢？半年以后呢？你还记得这样的代码究竟是干嘛的吗？假如几个月后你发现要修改其中的一两行，但又忘了其它地方的类似代码，你就完蛋了。代码重复是产品中出现bug的主要来源之一。

只要你的代码需要在两个不同的地方完成相同的事情，你就需要考虑创建一个新的方法。相信我，别偷懒。偷懒一时爽，代码死光光~

还有，方法的名称也有助于我们了解它的确切功能。如果你没看过前面的教程，我直接给你下面的三行代码：

```

    targetValue = 1 + Int((arc4random())_uniform(100))
    currentValue = 50
    slider.value = Float(currentValue)

```

你能知道这几行代码究竟是干吗用的吗？当然，你可能了解这里用随机数创建了一个数值，然后设置了一个初始值，再把初始值赋予了滑动条。

仅此而已，你能知道这其实是在开启新一轮的游戏吗？不管你能不能，我反正是不知道。有些程序猿会加上一些注释代码，但最好的方法还是学学哥：

```

startNewRound()

```

清清楚楚明明白白真真切切，比注释行还要清晰明了。虽然注释行也很有用，但让代码具有自解释性更重要。

而且在后面你可以随时查看和修改startNewRound()方法，更不用说在Xcode里面搜索它也变得简单了。

点击Run运行游戏，确保每次触碰都会产生新的1到100之间的随机数。

你或许注意到了，每个新的回合开启时，滑动条都会重置到中间的位置。这是因为在startNewRound方法中我们重置currentValue的数值到50，然后让滑动条的结点位置更改为这一点。

这个和我们最初的做法不同，之前我们是读取滑动条的位置，然后把它的数值赋予currentValue。哥这样做的目的是让每一轮新的游戏都从相同的位置开始，感觉要靠谱点。

小练习：

如果你比较无聊，可以尝试下修改代码，让滑动条不必每次开始的时候都放在中间的位置。

## 类型转换

对了，你可能对以下代码中Float(...)和Int(...)的作用有点迷糊：

```
targetValue = 1 + Int(arc4random_uniform(100))  
slider.value = Float(currentValue)
```

Swift是所谓的强类型语言，也就是说它对可以放进盒子里的形状非常挑剔（也就是说Think Different行不通？！）。比如，一个变量是Int类型的，如果你把Float类型的数值交给它保存，它会直接拒绝的，反过来也一样。

我们从UISlider所获得的数值正好是Float类型的，也就是说带小数点，之前我们曾经见识过。不过currentValue是Int类型的，因此如果你直接这样写代码是不行的：

```
slider.value = currentValue
```

编译器会直接报错。有些编程语言可能会自动帮你把Int类型转换成Float类型，但Swift是不会乐意帮你做这种事情的~

当我们使用Float(currentValue)的时候，编译器会把currentValue中保存的整数值转换成



一个新的Float浮点数，从而可以赐给UISlider。

之前在使用arc4random\_uniform()的时候我们用过类似的方法，也就是在将随机数保存到targetValue之前，首先需要把它转换成一个Int数值。

Swift虽然是一门全新的语言，但是相比大多数的流行编程语言来说，对于变量类型的使用要求非常严格。这一点对于编程入门的新手来说会造成混乱和困惑，当然更加不幸的是Swift的错误提示信息还不能准确的指出代码错在哪里。

作为入门新手只需要记住，当你看到类似这样的错误信息时，cannot assign value of type 'something' to type 'something else'，就意味着你在数据的变量类型上使用出错了。而解决方案只能是强制的把某个类型转换成另一个。

继续前进！

## 把我们的目标数值放到标签里

好吧，我们已经知道如何生成随机数，然后把它保存在一个名为targetValue的变量里。现在我们需要把这个数值放到屏幕上，不然玩家就是瞎猫去抓活老鼠，要多难有多难。

之前我们设计storyboard界面的时候，已经添加了一个用来放置目标值的标签。现在要做的事情就是读取targetValue变量的数值，然后把它放到标签里。为了做到这一点，我们需要完成两件事情：

- 1.创建一个到标签对象的outlet，这样我们才好向它发送消息
- 2.让标签显示新的文本内容

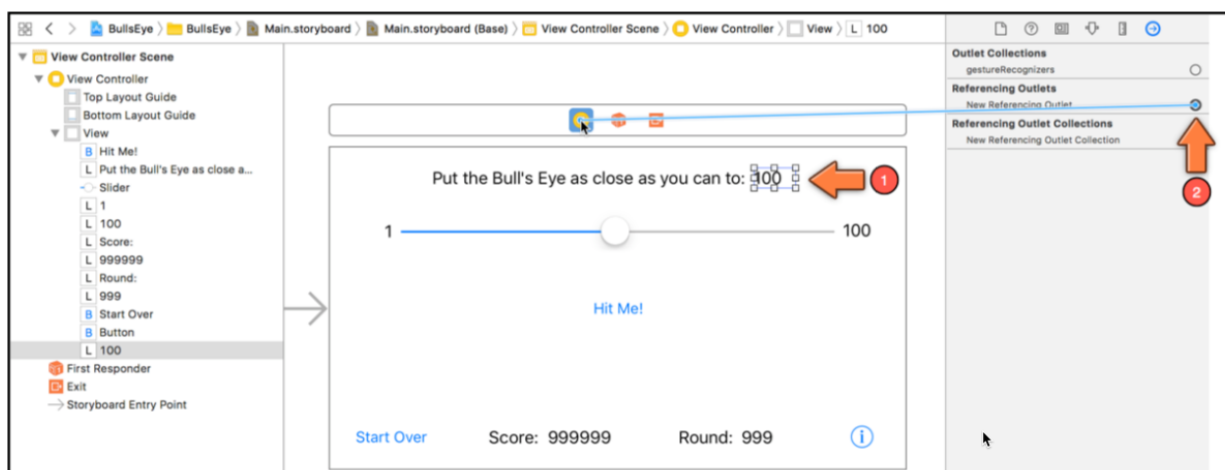
其实之前对滑动条已经做过类似的事情了。还记得我们之前曾添加过@IBOutlet这个变量吗？通过这种方式，我们可以从视图控制器的任何一个地方引用该滑动条。通过这个outlet变量，我们可以向滑动条请求获取它的数值，具体来说就是通过slider.value。对于这个标签我们将用到相同的方式来完成。

在Xcode中切换到ViewController.swift，然后在其它outlet变量定义的下面添加下面的这行代码：

```
@IBOutlet weak var targetLabel: UILabel!
```

在Main.storyboard中，点击选择代表目标值的标签

在Xcode右侧切换到Connections inspector,然后从New Referencing Outlet拖一条线到View Controller, 如下图所示:



从弹出菜单中选择targetLabel, 这样就建立了从界面视觉元素到视图控制器中相关变量的关联。

好了, 又到了代码时间。让我们在Xcode中切换回代码界面, 打开ViewController.swift

在startNewRound()方法的下面添加一个新的方法:

```
func updateLabels(){
    targetLabel.text = String(targetValue)
}
```

我们把这个逻辑放到一个单独的方法里, 这是因为在其它地方我们还会用到。

方法的名称已经很清晰的说明了它的作用: 使用它来更新标签的文本内容。

当然, 现在我们只是用它来更新单个标签的文本内容, 后面还会添加代码来更新其它标签的文本内容 (总得分, 游戏回合数)。

updateLabels这个方法里面的代码现在应该可以看懂了, 我们把一个字符串的内容赋予目标标签的文本属性。

不过你可能要问，为什么我们不能更简单一点，用下面的代码呢：

```
targetLabel.text = targetValue;
```

原因很简单，我们不能把整数类型的数值直接赋予字符串类型的变量。方井盖没法放进圆孔。

targetLabel这个outlet变量代表对UILabel对象的引用。UILabel标签对象有自己的text属性，它是一个字符串对象。我们只能把字符串类型的数据放到text里面。

但是刚才的代码试图把targetValue,一个Int类型的整数直接放到字符串类型的变量里。在Swift中这样的事情显然不可能发生。

所以我们需要先把整数类型的变量转换为一个字符串，这就是String(targetValue)的作用。。其实之前我们也做过类似的事情，现在你应该知道为什么要这么样做了吧。

但是用下面的方式是可行的：

```
targetLabel.text = "\(targetValue)"
```

需要注意的是，updateLabels()是个常规的方法，它没有作为一个动作方法关联到任何UI界面控件上，因此除非我们手动调用它，它就难有出头之日。要区分某个方法是否是动作方法很简单，只需要看前面是否有@IBAction即可。

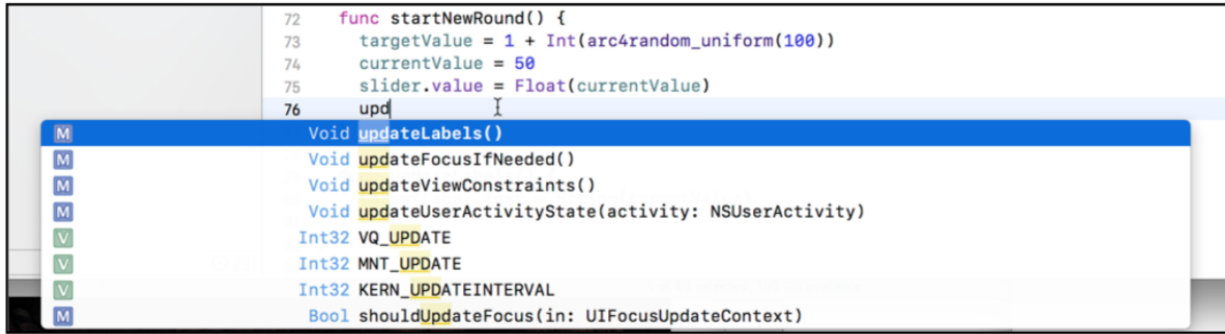
调用updateLabels()方法比较合适的地方是在每次调用startNewRound()方法后，因为在那里我们将要获取新的目标值。我们无需在两个不同的地方调用updateLabels(),只需将其包含在startNewRound()方法中即可。

因此让我们修改startNewRound()的方法如下：

```
func startNewRound() {  
    targetValue = 1 + Int(arc4random_uniform(100))  
    currentValue = 50  
    slider.value = Float(currentValue)  
    updateLabels()  
}
```

黄色高亮部分就是新增的代码。

Xcode的自动完成功能会帮你不小的帮，作为一个懒人，你只需要敲下方法名称的头几个字母，比如upd，Xcode会自动帮你完成其它的。你只需要回车确认建议即可：



点击Xcode上的Run按钮来运行游戏，现在我们可以从屏幕上看到随机生成的目标值了。

在结束之前，我们再来点理论知识恶补一下吧。

### 科普：动作方法vs一般方法

好吧，我们现在已经用到了两种方法，分别是和界面中控件关联起来的动作方法，以及有着独立意识的一般方法。那么它们之间有什么区别呢？

答案是：没什么区别。

唯一不同的就是@IBAction这个标示。当我们在方法前面加了这个东西后，Interface Builder就会知道我们可以把它和界面中的按钮，滑动条或者其它控件关联起来。

至于那些没有@IBAction标识的方法就不能和界面中的控件建立关联了。

// 基本动作方法

@IBAction func showAlert()

// 动作方法，不过添加了到触发这个动作的对象的引用

@IBAction func sliderMoved(\_ slider: UISlider)

@IBAction func buttonTapped(\_ button: UIButton)

// 不能和Interface Builder中的控件建立关联

func updateLabels()

好了，今天的学习到此结束，上福利了。



那等在季节里的容颜

吴念真

M 明星网  
mingxing.com



遥想公瑾当年。。。

