翻译：vclwei, @coverxit, @NicePiao　校对：coverxit

控制流

本页包含内容：

Swift提供了类似 C 语言的流程控制结构，包括可以多次执行任务的`for`和`while`循环，基于特定条件选择执行不同代码分支的`if`和`switch`语句，还有控制流程跳转到其他代码的`break`和`continue`语句。

除了 C 语言里面传统的for条件递增（`for-condition-increment`）循环，Swift 还增加了`for-in`循环，用来更简单地遍历数组（array），字典（dictionary），区间（range），字符串（string）和其他序列类型。

Swift 的`switch`语句比 C 语言中更加强大。在 C 语言中，如果某个 case 不小心漏写了`break`，这个 case 就会贯穿（fallthrough）至下一个 case，Swift 无需写`break`，所以不会发生这种贯穿（fallthrough）的情况。case 还可以匹配更多的类型模式，包括区间匹配（range matching），元组（tuple）和特定类型的描述。`switch`的 case 语句中匹配的值可以是由 case 体内部临时的常量或者变量决定，也可以由`where`分句描述更复杂的匹配条件。

For 循环

`for`循环用来按照指定的次数多次执行一系列语句。Swift 提供两种`for`循环形式：

- `for-in`用来遍历一个区间（range），序列（sequence），集合

（collection），系列（progression）里面所有的元素执行一系列语句。

- for条件递增（`for-condition-increment`）语句，用来重复执行一系列语句直到达成特定条件达成，一般通过在每次循环完成后增加计数器的值来实现。

For-In

你可以使用`for-in`循环来遍历一个集合里面的所有元素，例如由数字表示的区间、数组中的元素、字符串中的字符。

下面的例子用来输出乘 5 乘法表前面一部分内容：

```
for index in 1...5 {
    println("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

例子中用来进行遍历的元素是一组使用闭区间操作符（`...`）表示的从`1`到`5`的数字。`index`被赋值为闭区间中的第一个数字（`1`），然后循环中的语句被执行一次。在本例中，这个循环只包含一个语句，用来输出当前`index`值所对应的乘 5 乘法表结果。该语句执行后，`index`的值被更新为闭区间中的第二个数字（`2`），之后`println`方法会再执行一次。整个过程会进行到闭区间结尾为止。

上面的例子中，`index`是一个每次循环遍历开始时被自动赋值的常量。这种情况下，`index`在使用前不需要声明，只需要将它包含在循环的声明中，就可以对其进行隐式声明，而无需使用`let`关键字声明。

注意：

`index`常量只存在于循环的生命周期里。如果你想在循环完成后访问`index`的值，又或者想让`index`成为一个变量而不是常量，你必须在循环

之前自己进行声明。
如果你不需要知道区间内每一项的值，你可以使用下划线（_）替代变量名来忽略对值的访问：

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
println("\(base) to the power of \(power) is \
(answer)")
// 输出 "3 to the power of 10 is 59049"
```

这个例子计算 base 这个数的 power 次幂（本例中，是3的10次幂），从1（3的0次幂）开始做3的乘法， 进行10次，使用0到9的半闭区间循环。这个计算并不需要知道每一次循环中计数器具体的值，只需要执行了正确的循环次数即可。下划线符号_（替代循环中的变量）能够忽略具体的值，并且不提供循环遍历时对值的访问。

使用for-in遍历一个数组所有元素：

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    println("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

你也可以通过遍历一个字典来访问它的键值对（key-value pairs）。遍历字典时，字典的每项元素会以(key, value)元组的形式返回，你可以在for-in循环中使用显式的常量名称来解读(key, value)元组。下面的例子中，字典的键（key）解读为常量animalName，字典的值会被解读为常量legCount：

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
```

```
    println("\(animalName)s have \(legCount) legs")
}
// spiders have 8 legs
// ants have 6 legs
// cats have 4 legs
```

字典元素的遍历顺序和插入顺序可能不同，字典的内容在内部是无序的，所以遍历元素时不能保证顺序。关于数组和字典，详情参见集合类型。

除了数组和字典，你也可以使用`for-in`循环来遍历字符串中的字符（`Character`）：

```
for character in "Hello" {
    println(character)
}
// H
// e
// l
// l
// o
```

For条件递增（for-condition-increment）

除了`for-in`循环，Swift 提供使用条件判断和递增方法的标准 C 样式`for`循环:

```
for var index = 0; index < 3; ++index {
    println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
```

下面是一般情况下这种循环方式的格式：

```
for `initialization`; `condition`; `increment` {
    `statements`
}
```

和 C 语言中一样，分号将循环的定义分为 3 个部分，不同的是，Swift 不

需要使用圆括号将“initialization; condition; increment”包括起来。

这个循环执行流程如下：

1. 循环首次启动时，初始化表达式（*initialization expression*）被调用一次，用来初始化循环所需的所有常量和变量。
2. 条件表达式（*condition expression*）被调用，如果表达式调用结果为`false`，循环结束，继续执行`for`循环关闭大括号（`}`）之后的代码。如果表达式调用结果为`true`，则会执行大括号内部的代码（*statements*）。
3. 执行所有语句（*statements*）之后，执行递增表达式（*increment expression*）。通常会增加或减少计数器的值，或者根据语句（*statements*）输出来修改某一个初始化的变量。当递增表达式运行完成后，重复执行第 2 步，条件表达式会再次执行。

上述描述和循环格式等同于：

```
`initialization`
while `condition` {
    `statements`
    `increment`
}
```

在初始化表达式中声明的常量和变量（比如`var index = 0`）只在`for`循环的生命周期里有效。如果想在循环结束后访问`index`的值，你必须要在循环生命周期开始前声明`index`。

```
var index: Int
for index = 0; index < 3; ++index {
    println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
println("The loop statements were executed \(index)
times")
// 输出 "The loop statements were executed 3 times
```

注意`index`在循环结束后最终的值是`3`而不是`2`。最后一次调用递增表达式`++index`会将`index`设置为`3`，从而导致`index < 3`条件为`false`，并终止

循环。

While 循环

`while`循环运行一系列语句直到条件变成`false`。这类循环适合使用在第一次迭代前迭代次数未知的情况下。Swift 提供两种`while`循环形式：

- `while`循环，每次在循环开始时计算条件是否符合；
- `do-while`循环，每次在循环结束时计算条件是否符合。

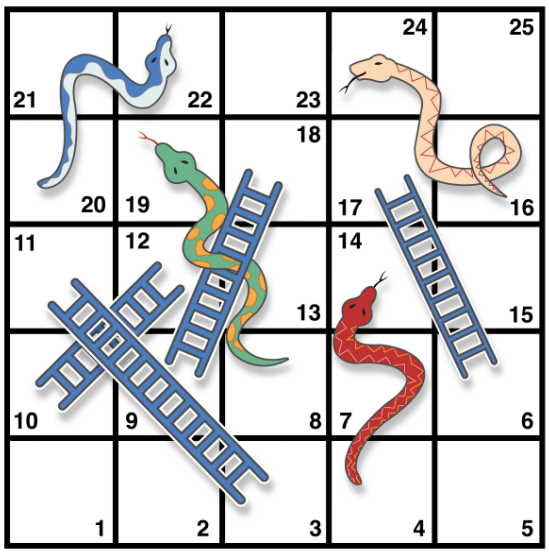
While

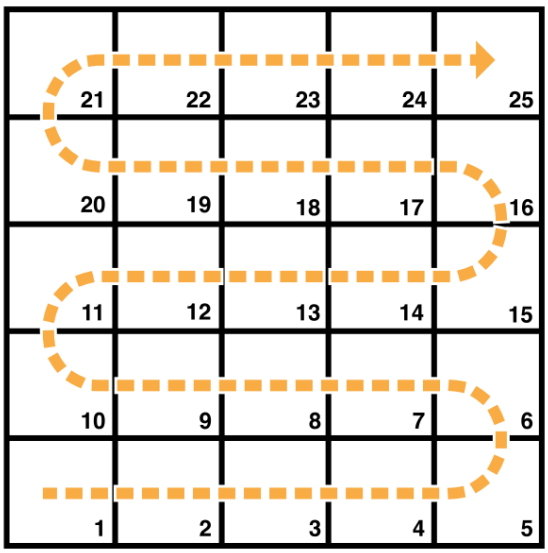
`while`循环从计算单一条件开始。如果条件为`true`，会重复运行一系列语句，直到条件变为`false`。

下面是一般情况下 `while` 循环格式：

```
while `condition` {
    `statements`
}
```

下面的例子来玩一个叫做蛇和梯子（*Snakes and Ladders*）的小游戏，也叫做滑道和梯子（*Chutes and Ladders*）：





游戏的规则如下：

- 游戏盘面包括 25 个方格，游戏目标是达到或者超过第 25 个方格；
- 每一轮，你通过掷一个 6 边的骰子来确定你移动方块的步数，移动的路线由上图中横向的虚线所示；
- 如果在某轮结束，你移动到了梯子的底部，可以顺着梯子爬上去；
- 如果在某轮结束，你移动到了蛇的头部，你会顺着蛇的身体滑下去。

游戏盘面可以使用一个`Int`数组来表达。数组的长度由一个`finalSquare`常量储存，用来初始化数组和检测最终胜利条件。游戏盘面由 26 个 `Int` 0 值初始化，而不是 25 个（由`0`到`25`，一共 26 个）：

```
let finalSquare = 25
var board = Int[](count: finalSquare + 1,
repeatedValue: 0)
```

一些方块被设置成有蛇或者梯子的指定值。梯子底部的方块是一个正值，使你可以向上移动，蛇头处的方块是一个负值，会让你向下移动：

```
board[03] = +08; board[06] = +11; board[09] = +09;
board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02;
board[24] = -08
```

3 号方块是梯子的底部，会让你向上移动到 11 号方格，我们使用`board[03]`等于`+08`（来表示`11`和`3`之间的差值）。使用一元加运算符（`+i`）是为了和一元减运算符（`-i`）对称，为了让盘面代码整齐，小于 10 的数字都使用 0 补齐（这些风格上的调整都不是必须的，只是为了让代码看起来更加整洁）。

玩家由左下角编号为 0 的方格开始游戏。一般来说玩家第一次掷骰子后才会进入游戏盘面：

```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // 掷骰子
    if ++diceRoll == 7 { diceRoll = 1 }
    // 根据点数移动
```

```
    square += diceRoll
    if square < board.count {
        // 如果玩家还在棋盘上，顺着梯子爬上去或者顺着蛇滑下去
        square += board[square]
    }
}
println("Game over!")
```

本例中使用了最简单的方法来模拟掷骰子。`diceRoll`的值并不是一个随机数，而是以`0`为初始值，之后每一次`while`循环，`diceRoll`的值使用前置自增操作符(`++i`)来自增 1 ，然后检测是否超出了最大值。`++diceRoll`调用完成后，返回值等于`diceRoll`自增后的值。任何时候如果`diceRoll`的值等于7时，就超过了骰子的最大值，会被重置为`1`。所以`diceRoll`的取值顺序会一直是`1`，`2`，`3`，`4`，`5`，`6`，`1`，`2`。

掷完骰子后，玩家向前移动`diceRoll`个方格，如果玩家移动超过了第 25 个方格，这个时候游戏结束，相应地，代码会在`square`增加`board[square]`的值向前或向后移动（遇到了梯子或者蛇）之前，检测`square`的值是否小于`board`的`count`属性。

如果没有这个检测（`square < board.count`），`board[square]`可能会越界访问`board`数组，导致错误。例如如果`square`等于`26`， 代码会去尝试访问`board[26]`，超过数组的长度。

当本轮`while`循环运行完毕，会再检测循环条件是否需要再运行一次循环。如果玩家移动到或者超过第 25 个方格，循环条件结果为`false`，此时游戏结束。

`while` 循环比较适合本例中的这种情况，因为在 `while` 循环开始时，我们并不知道游戏的长度或者循环的次数，只有在达成指定条件时循环才会结束。

Do-While

`while`循环的另外一种形式是`do-while`，它和`while`的区别是在判断循环条件之前，先执行一次循环的代码块，然后重复循环直到条件为`false`。

下面是一般情况下 do-while循环的格式：

```
do {
    `statements`
} while `condition`
```

还是蛇和梯子的游戏，使用do-while循环来替代while循环。finalSquare、board、square和diceRoll的值初始化同while循环一样：

```
let finalSquare = 25
var board = Int[](count: finalSquare + 1,
repeatedValue: 0)
board[03] = +08; board[06] = +11; board[09] = +09;
board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02;
board[24] = -08
var square = 0
var diceRoll = 0
```

do-while的循环版本，循环中第一步就需要去检测是否在梯子或者蛇的方块上。没有梯子会让玩家直接上到第 25 个方格，所以玩家不会通过梯子直接赢得游戏。这样在循环开始时先检测是否踩在梯子或者蛇上是安全的。

游戏开始时，玩家在第 0 个方格上，board[0]一直等于 0， 不会有什么影响：

```
do {
    // 顺着梯子爬上去或者顺着蛇滑下去
    square += board[square]
    // 掷骰子
    if ++diceRoll == 7 { diceRoll = 1 }
    // 根据点数移动
    square += diceRoll
} while square < finalSquare
println("Game over!")
```

检测完玩家是否踩在梯子或者蛇上之后，开始掷骰子，然后玩家向前移动

`diceRoll`个方格，本轮循环结束。

循环条件（`while square < finalSquare`）和`while`方式相同，但是只会在循环结束后进行计算。在这个游戏中，`do-while`表现得比`while`循环更好。`do-while`方式会在条件判断`square`没有超出后直接运行`square += board[square]`，这种方式可以去掉`while`版本中的数组越界判断。

条件语句

根据特定的条件执行特定的代码通常是十分有用的，例如：当错误发生时，你可能想运行额外的代码；或者，当输入的值太大或太小时，向用户显示一条消息等。要实现这些功能，你就需要使用条件语句。

Swift 提供两种类型的条件语句：`if`语句和`switch`语句。通常，当条件较为简单且可能的情况很少时，使用`if`语句。而`switch`语句更适用于条件较复杂、可能情况较多且需要用到模式匹配（pattern-matching）的情境。

If

`if`语句最简单的形式就是只包含一个条件，当且仅当该条件为`true`时，才执行相关代码：

```
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing a
scarf.")
}
// 输出 "It's very cold. Consider wearing a scarf."
```

上面的例子会判断温度是否小于等于 32 华氏度（水的冰点）。如果是，则打印一条消息；否则，不打印任何消息，继续执行`if`块后面的代码。

当然，`if`语句允许二选一，也就是当条件为`false`时，执行 *else* 语句：

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing a
scarf.")
} else {
    println("It's not that cold. Wear a t-shirt.")
}
// 输出 "It's not that cold. Wear a t-shirt."
```

显然，这两条分支中总有一条会被执行。由于温度已升至 40 华氏度，不算太冷，没必要再围围巾——因此，`else`分支就被触发了。

你可以把多个`if`语句链接在一起，像下面这样：

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing a
scarf.")
} else if temperatureInFahrenheit >= 86 {
    println("It's really warm. Don't forget to wear
sunscreen.")
} else {
    println("It's not that cold. Wear a t-shirt.")
}
// 输出 "It's really warm. Don't forget to wear
sunscreen."
```

在上面的例子中，额外的`if`语句用于判断是不是特别热。而最后的`else`语句被保留了下来，用于打印既不冷也不热时的消息。

实际上，最后的`else`语句是可选的：

```
temperatureInFahrenheit = 72
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing a
scarf.")
} else if temperatureInFahrenheit >= 86 {
    println("It's really warm. Don't forget to wear
sunscreen.")
}
```

在这个例子中，由于既不冷也不热，所以不会触发`if`或`else if`分支，也就不会打印任何消息。

Switch

`switch`语句会尝试把某个值与若干个模式（pattern）进行匹配。根据第一个匹配成功的模式，`switch`语句会执行对应的代码。当有可能的情况较多时，通常用`switch`语句替换`if`语句。

`switch`语句最简单的形式就是把某个值与一个或若干个相同类型的值作比较：

```
switch `some value to consider` {
case `value 1`:
    `respond to value 1`
case `value 2`,
`value 3`:
    `respond to value 2 or 3`
default:
    `otherwise, do something else`
}
```

`switch`语句都由多个 *case* 构成。为了匹配某些更特定的值，Swift 提供了几种更复杂的匹配模式，这些模式将在本节的稍后部分提到。

每一个 case 都是代码执行的一条分支，这与`if`语句类似。与之不同的是，`switch`语句会决定哪一条分支应该被执行。

`switch`语句必须是完备的。这就是说，每一个可能的值都必须至少有一个 case 分支与之对应。在某些不可能涵盖所有值的情况下，你可以使用默认（`default`）分支满足该要求，这个默认分支必须在`switch`语句的最后面。

下面的例子使用`switch`语句来匹配一个名为`someCharacter`的小写字符：

```
let someCharacter: Character = "e"
```

```
switch someCharacter {
case "a", "e", "i", "o", "u":
    println("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l",
"m",
"n", "p", "q", "r", "s", "t", "v", "w", "x", "y",
"z":
    println("\(someCharacter) is a consonant")
default:
    println("\(someCharacter) is not a vowel or a
consonant")
}
// 输出 "e is a vowel"
```

在这个例子中，第一个 case 分支用于匹配五个元音，第二个 case 分支用于匹配所有的辅音。

由于为其它可能的字符写 case 分支没有实际的意义，因此在这个例子中使用了默认分支来处理剩下的既不是元音也不是辅音的字符——这就保证了`switch`语句的完备性。

不存在隐式的贯穿（No Implicit Fallthrough）

与 C 语言和 Objective-C 中的`switch`语句不同，在 Swift 中，当匹配的 case 分支中的代码执行完毕后，程序会终止`switch`语句，而不会继续执行下一个 case 分支。这也就是说，不需要在 case 分支中显式地使用`break`语句。这使得`switch`语句更安全、更易用，也避免了因忘记写`break`语句而产生的错误。

注意：

你依然可以在 case 分支中的代码执行完毕前跳出，详情请参考Switch 语句中的 break。

每一个 case 分支都必须包含至少一条语句。像下面这样书写代码是无效的，因为第一个 case 分支是空的：

```
let anotherCharacter: Character = "a"
```

```
switch anotherCharacter {
case "a":
case "A":
    println("The letter A")
default:
    println("Not the letter A")
}
// this will report a compile-time error
```

不像 C 语言里的`switch`语句，在 Swift 中，`switch`语句不会同时匹配`"a"`和`"A"`。相反的，上面的代码会引起编译期错误：`case "a": does not contain any executable statements`——这就避免了意外地从一个 case 分支贯穿到另外一个，使得代码更安全、也更直观。

一个 case 也可以包含多个模式，用逗号把它们分开（如果太长了也可以分行写）：

```
switch `some value to consider` {
case `value 1`,
`value 2`:
    `statements`
}
```

注意：如果想要贯穿至特定的 case 分支中，请使用`fallthrough`语句，详情请参考贯穿（Fallthrough）。

区间匹配（Range Matching）

case 分支的模式也可以是一个值的区间。下面的例子展示了如何使用区间匹配来输出任意数字对应的自然语言格式：

```
let count = 3_000_000_000_000
let countedThings = "stars in the Milky Way"
var naturalCount: String
switch count {
case 0:
    naturalCount = "no"
case 1...3:
    naturalCount = "a few"
```

```
case 4...9:
    naturalCount = "several"
case 10...99:
    naturalCount = "tens of"
case 100...999:
    naturalCount = "hundreds of"
case 1000...999_999:
    naturalCount = "thousands of"
default:
    naturalCount = "millions and millions of"
}
println("There are \(naturalCount) \
(countedThings).")
// 输出 "There are millions and millions of stars in
the Milky Way."
```

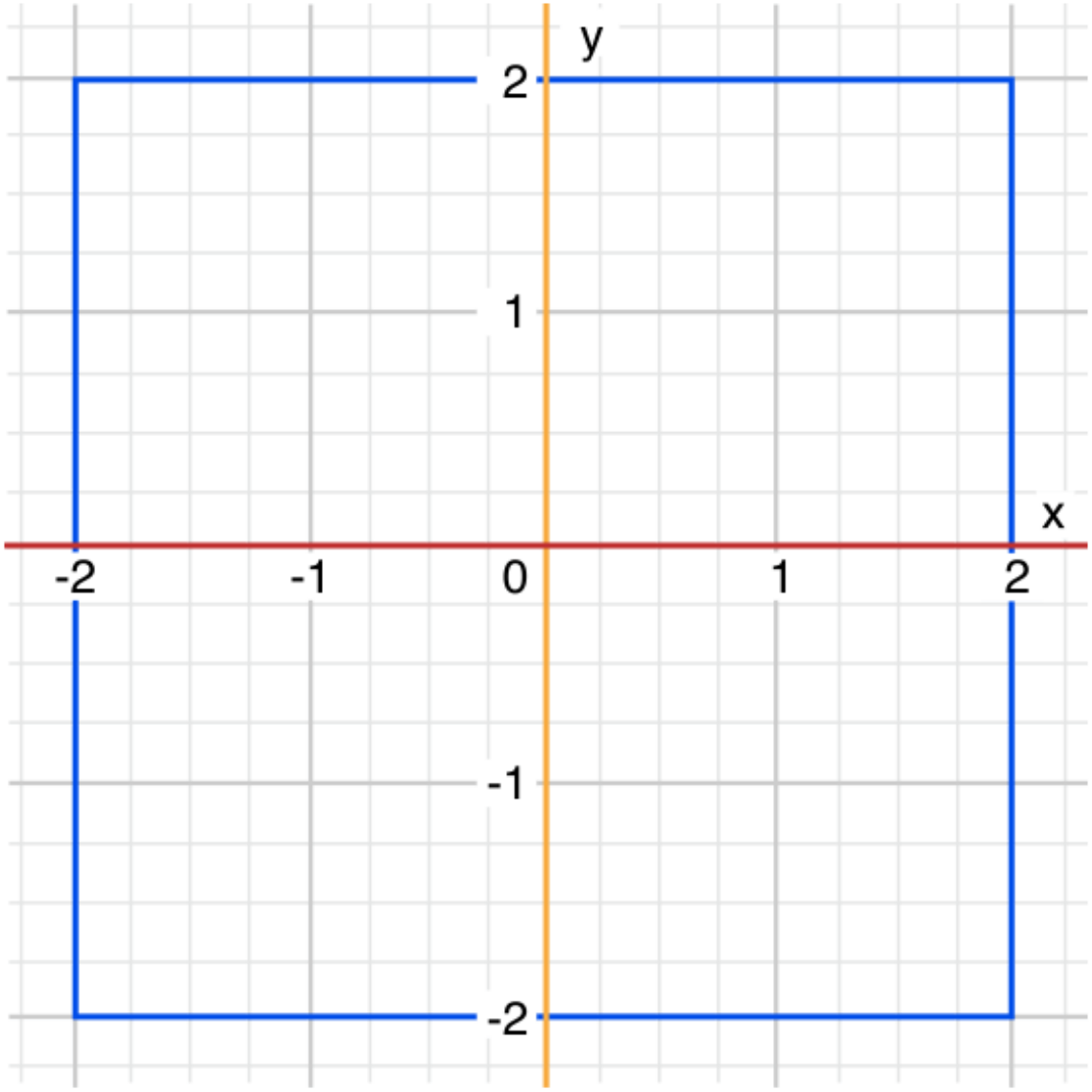
元组（Tuple）

你可以使用元组在同一个`switch`语句中测试多个值。元组中的元素可以是值，也可以是区间。另外，使用下划线（`_`）来匹配所有可能的值。

下面的例子展示了如何使用一个`(Int, Int)`类型的元组来分类下图中的点(x, y)：

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    println("(0, 0) is at the origin")
case (_, 0):
    println("(\(somePoint.0), 0) is on the x-axis")
case (0, _):
    println("(0, \(somePoint.1)) is on the y-axis")
case (-2...2, -2...2):
    println("(\(somePoint.0), \(somePoint.1)) is
inside the box")
default:
    println("(\(somePoint.0), \(somePoint.1)) is
```

```
outside of the box")
}
// 输出 "(1, 1) is inside the box"
```



在上面的例子中，`switch`语句会判断某个点是否是原点(0, 0)，是否在红色的x轴上，是否在黄色y轴上，是否在一个以原点为中心的4x4的矩形里，或者在这个矩形外面。

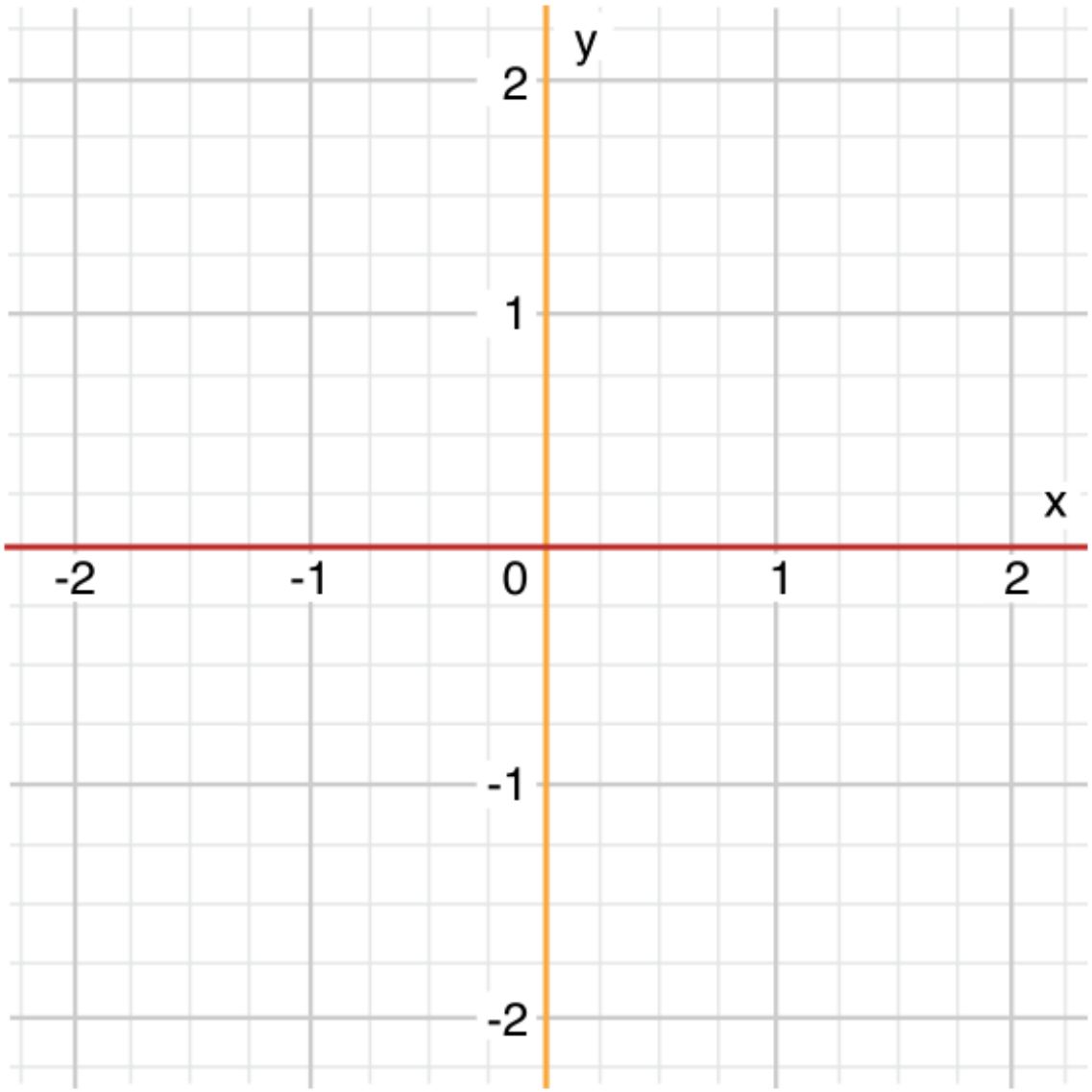
不像 C 语言，Swift 允许多个 case 匹配同一个值。实际上，在这个例子中，点(0, 0)可以匹配所有四个 *case*。但是，如果存在多个匹配，那么只会执行第一个被匹配到的 case 分支。考虑点(0, 0)会首先匹配`case (0, 0)`，因此剩下的能够匹配(0, 0)的 case 分支都会被忽视掉。

值绑定（Value Bindings）

case 分支的模式允许将匹配的值绑定到一个临时的常量或变量，这些常量或变量在该 case 分支里就可以被引用了——这种行为被称为值绑定（value binding）。

下面的例子展示了如何在一个`(Int, Int)`类型的元组中使用值绑定来分类下图中的点(x, y)：

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    println("on the x-axis with an x value of \(x)")
case (0, let y):
    println("on the y-axis with a y value of \(y)")
case let (x, y):
    println("somewhere else at (\(x), \(y))")
}
// 输出 "on the x-axis with an x value of 2"
```



在上面的例子中，`switch`语句会判断某个点是否在红色的x轴上，是否在黄色y轴上，或者不在坐标轴上。

这三个 case 都声明了常量`x`和`y`的占位符，用于临时获取元组`anotherPoint`的一个或两个值。第一个 case ——`case (let x, 0)`将匹配一个纵坐标为`0`的点，并把这个点的横坐标赋给临时的常量`x`。类似的，第二个 case ——`case (0, let y)`将匹配一个横坐标为`0`的点，并把这个点的纵坐标赋给临时的常量`y`。

一旦声明了这些临时的常量，它们就可以在其对应的 case 分支里引用。在这个例子中，它们用于简化`println`的书写。

请注意，这个`switch`语句不包含默认分支。这是因为最后一个 case ——

`case let(x, y)`声明了一个可以匹配余下所有值的元组。这使得`switch`语句已经完备了，因此不需要再书写默认分支。

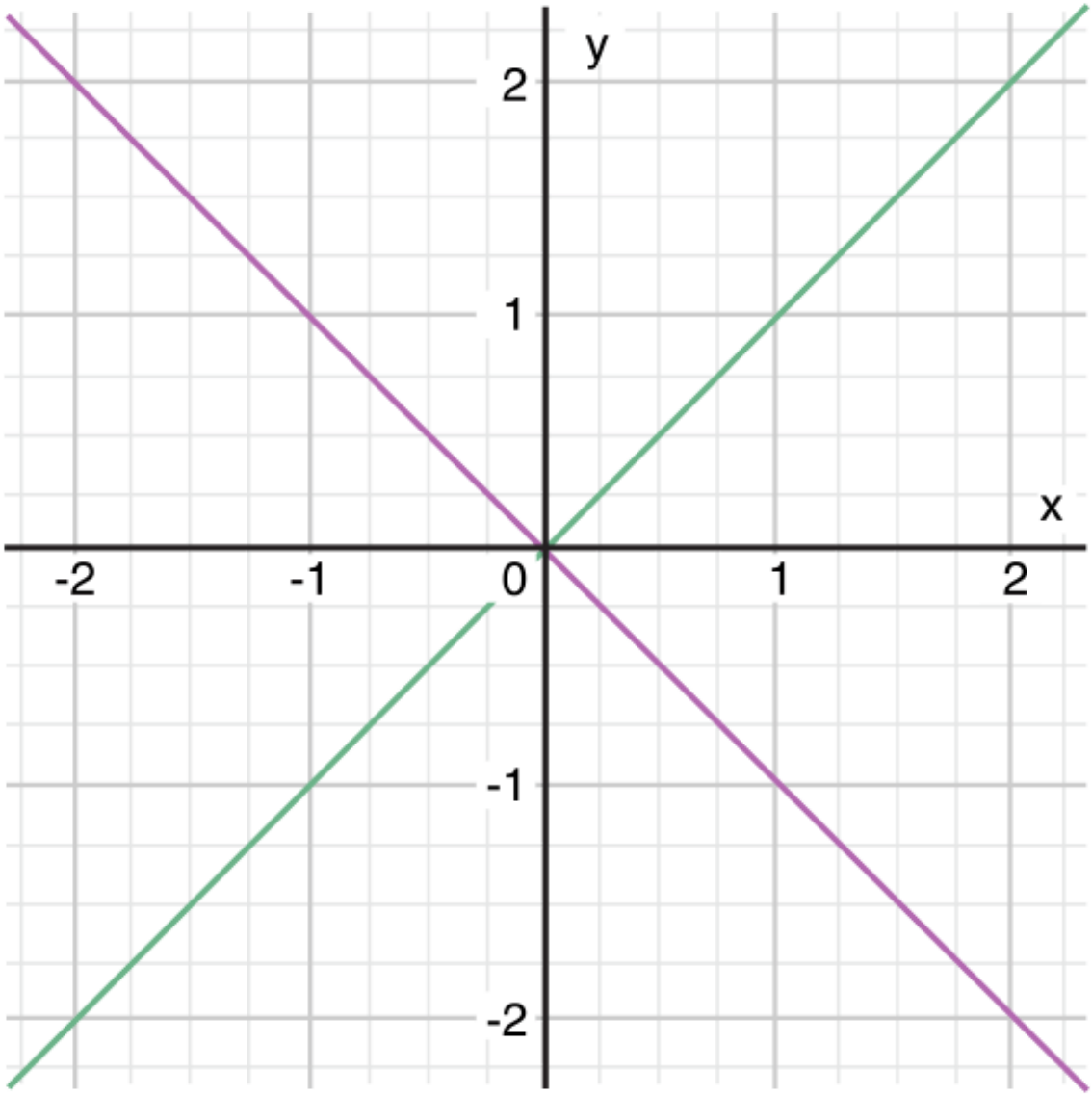
在上面的例子中，`x`和`y`是常量，这是因为没有必要在其对应的 case 分支中修改它们的值。然而，它们也可以是变量——程序将会创建临时变量，并用相应的值初始化它。修改这些变量只会影响其对应的 case 分支。

Where

case 分支的模式可以使用`where`语句来判断额外的条件。

下面的例子把下图中的点(x, y)进行了分类：

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    println("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    println("(\(x), \(y)) is on the line x == -y")
case let (x, y):
    println("(\(x), \(y)) is just some arbitrary
point")
}
// 输出 "(1, -1) is on the line x == -y"
```



在上面的例子中，`switch`语句会判断某个点是否在绿色的对角线`x == y`上，是否在紫色的对角线`x == -y`上，或者不在对角线上。

这三个 case 都声明了常量`x`和`y`的占位符，用于临时获取元组`yetAnotherPoint`的两个值。这些常量被用作`where`语句的一部分，从而创建一个动态的过滤器(filter)。当且仅当`where`语句的条件为`true`时，匹配到的 case 分支才会被执行。

就像是值绑定中的例子，由于最后一个 case 分支匹配了余下所有可能的值，`switch`语句就已经完备了，因此不需要再书写默认分支。

控制传递语句（Control Transfer Statements）

控制转移语句改变你代码的执行顺序，通过它你可以实现代码的跳转。Swift有四种控制转移语句。

- continue
- break
- fallthrough
- return

我们将会在下面讨论`continue`、`break`和`fallthrough`语句。`return`语句将会在函数章节讨论。

Continue

`continue`语句告诉一个循环体立刻停止本次循环迭代，重新开始下次循环迭代。就好像在说“本次循环迭代我已经执行完了”，但是并不会离开整个循环体。

注意：

在一个for条件递增（`for-condition-increment`）循环体中，在调用`continue`语句后，迭代增量仍然会被计算求值。循环体继续像往常一样工作，仅仅只是循环体中的执行代码会被跳过。

下面的例子把一个小写字符串中的元音字母和空格字符移除，生成了一个含义模糊的短句：

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput {
    switch character {
    case "a", "e", "i", "o", "u", " ":
        continue
    default:
```

```
        puzzleOutput += character
    }
}
println(puzzleOutput)
    // 输出 "grtmndsthnklk"
```

在上面的代码中，只要匹配到元音字母或者空格字符，就调用`continue`语句，使本次循环迭代结束，从新开始下次循环迭代。这种行为使`switch`匹配到元音字母和空格字符时不做处理，而不是让每一个匹配到的字符都被打印。

Break

`break`语句会立刻结束整个控制流的执行。当你想要更早的结束一个`switch`代码块或者一个循环体时，你都可以使用`break`语句。

循环语句中的 break

当在一个循环体中使用`break`时，会立刻中断该循环体的执行，然后跳转到表示循环体结束的大括号(`}`)后的第一行代码。不会再有本次循环迭代的代码被执行，也不会再有下次的循环迭代产生。

Switch 语句中的 break

当在一个`switch`代码块中使用`break`时，会立即中断该`switch`代码块的执行，并且跳转到表示`switch`代码块结束的大括号(`}`)后的第一行代码。

这种特性可以被用来匹配或者忽略一个或多个分支。因为 Swift 的`switch`需要包含所有的分支而且不允许有为空的分支，有时为了使你的意图更明显，需要特意匹配或者忽略某个分支。那么当你想忽略某个分支时，可以在该分支内写上`break`语句。当那个分支被匹配到时，分支内的`break`语句立即结束`switch`代码块。

注意：

当一个switch分支仅仅包含注释时，会被报编译时错误。注释不是代码语句而且也不能让switch分支达到被忽略的效果。你总是可以使用break来忽略某个分支。

下面的例子通过switch来判断一个Character值是否代表下面四种语言之一。为了简洁，多个值被包含在了同一个分支情况中。

```
let numberSymbol: Character = "三"  // 简体中文里的数字
3
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "١", "一", "๑":
    possibleIntegerValue = 1
case "2", "٢", "二", "๒":
    possibleIntegerValue = 2
case "3", "٣", "三", "๓":
    possibleIntegerValue = 3
case "4", "٤", "四", "๔":
    possibleIntegerValue = 4
default:
    break
}
if let integerValue = possibleIntegerValue {
    println("The integer value of \(numberSymbol) is
\(integerValue).")
} else {
    println("An integer value could not be found for
\(numberSymbol).")
}
// 输出 "The integer value of 三 is 3."
```

这个例子检查numberSymbol是否是拉丁，阿拉伯，中文或者泰语中的1到4之一。如果被匹配到，该switch分支语句给Int?类型变量possibleIntegerValue设置一个整数值。

当switch代码块执行完后，接下来的代码通过使用可选绑定来判断possibleIntegerValue是否曾经被设置过值。因为是可选类型的缘故，

possibleIntegerValue有一个隐式的初始值nil，所以仅仅当possibleIntegerValue曾被switch代码块的前四个分支中的某个设置过一个值时，可选的绑定将会被判定为成功。

在上面的例子中，想要把Character所有的的可能性都枚举出来是不现实的，所以使用default分支来包含所有上面没有匹配到字符的情况。由于这个default分支不需要执行任何动作，所以它只写了一条break语句。一旦落入到default分支中后，break语句就完成了该分支的所有代码操作，代码继续向下，开始执行if let语句。

贯穿（Fallthrough）

Swift 中的switch不会从上一个 case 分支落入到下一个 case 分支中。相反，只要第一个匹配到的 case 分支完成了它需要执行的语句，整个switch代码块完成了它的执行。相比之下，C 语言要求你显示的插入break语句到每个switch分支的末尾来阻止自动落入到下一个 case 分支中。Swift 的这种避免默认落入到下一个分支中的特性意味着它的switch功能要比 C 语言的更加清晰和可预测，可以避免无意识地执行多个 case 分支从而引发的错误。

如果你确实需要 C 风格的贯穿（fallthrough）的特性，你可以在每个需要该特性的 case 分支中使用fallthrough关键字。下面的例子使用fallthrough来创建一个数字的描述语句。

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe)
is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
println(description)
```

```
// 输出 "The number 5 is a prime number, and also an integer."
```
这个例子定义了一个String类型的变量description并且给它设置了一个初始值。函数使用switch逻辑来判断integerToDescribe变量的值。当integerToDescribe的值属于列表中的质数之一时，该函数添加一段文字在description后，来表明这个是数字是一个质数。然后它使用fallthrough关键字来“贯穿”到default分支中。default分支添加一段额外的文字在description的最后，至此switch代码块执行完了。

如果integerToDescribe的值不属于列表中的任何质数，那么它不会匹配到第一个switch分支。而这里没有其他特别的分支情况，所以integerToDescribe匹配到包含所有的default分支中。

当switch代码块执行完后，使用println函数打印该数字的描述。在这个例子中，数字5被准确的识别为了一个质数。

注意：

fallthrough关键字不会检查它下一个将会落入执行的 case 中的匹配条件。fallthrough简单地使代码执行继续连接到下一个 case 中的执行代码，这和 C 语言标准中的switch语句特性是一样的。

带标签的语句（Labeled Statements）

在 Swift 中，你可以在循环体和switch代码块中嵌套循环体和switch代码块来创造复杂的控制流结构。然而，循环体和switch代码块两者都可以使用break语句来提前结束整个方法体。因此，显示地指明break语句想要终止的是哪个循环体或者switch代码块，会很有用。类似地，如果你有许多嵌套的循环体，显示指明continue语句想要影响哪一个循环体也会非常有用。

为了实现这个目的，你可以使用标签来标记一个循环体或者switch代码块，当使用break或者continue时，带上这个标签，可以控制该标签代表对象的中断或者执行。

产生一个带标签的语句是通过在该语句的关键词的同一行前面放置一个标

签，并且该标签后面还需带着一个冒号。下面是一个`while`循环体的语法，同样的规则适用于所有的循环体和`switch`代码块。

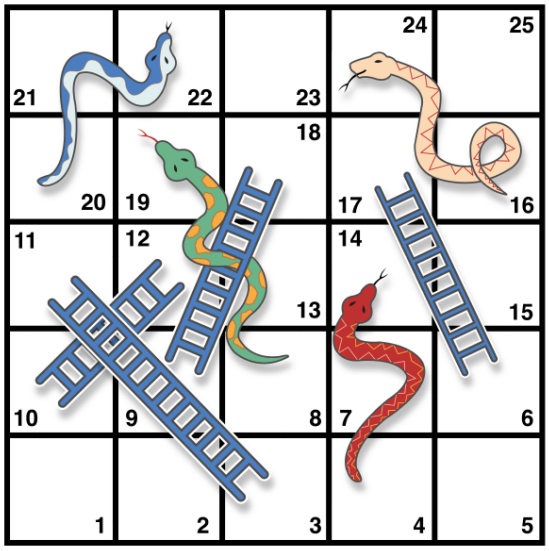
```
`label name`: while `condition` {
    `statements`
}
```

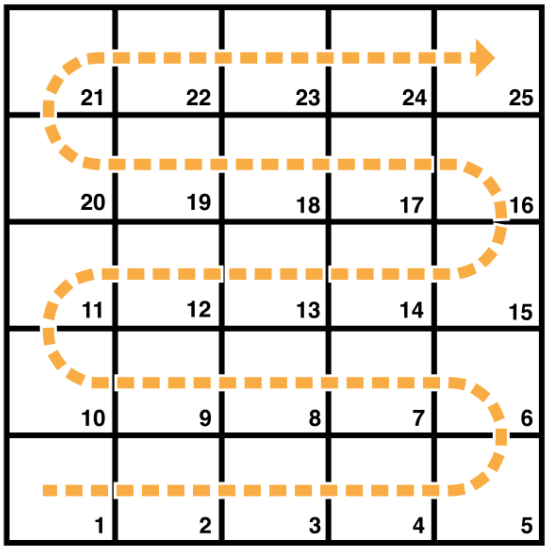
下面的例子是在一个带有标签的`while`循环体中调用`break`和`continue`语句，该循环体是前面章节中蛇和梯子的改编版本。这次，游戏增加了一条额外的规则：

- 为了获胜，你必须刚好落在第 25 个方块中。

如果某次掷骰子使你的移动超出第 25 个方块，你必须重新掷骰子，直到你掷出的骰子数刚好使你能落在第 25 个方块中。

游戏的棋盘和之前一样：





值`finalSquare`、`board`、`square`和`diceRoll`的初始化也和之前一样：

```
let finalSquare = 25
var board = Int[](count: finalSquare + 1,
repeatedValue: 0)
board[03] = +08; board[06] = +11; board[09] = +09;
board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02;
board[24] = -08
var square = 0
```

```
var diceRoll = 0
```
这个版本的游戏使用`while`循环体和`switch`方法块来实现游戏的逻辑。`while`循环体有一个标签名`gameLoop`，来表明它是蛇与梯子的主循环。

该`while`循环体的条件判断语句是`while square !=finalSquare`，这表明你必须刚好落在方格25中。

```
gameLoop: while square != finalSquare {
    if ++diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:
        // 到达最后一个方块，游戏结束
        break gameLoop
    case let newSquare where newSquare > finalSquare:
        // 超出最后一个方块，再掷一次骰子
        continue gameLoop
    default:
        // 本次移动有效
        square += diceRoll
        square += board[square]
    }
}
println("Game over!")
```
每次循环迭代开始时掷骰子。与之前玩家掷完骰子就立即移动不同，这里使用了`switch`来考虑每次移动可能产生的结果，从而决定玩家本次是否能够移动。

- 如果骰子数刚好使玩家移动到最终的方格里，游戏结束。`break gameLoop`语句跳转控制去执行`while`循环体后的第一行代码，游戏结束。
- 如果骰子数将会使玩家的移动超出最后的方格，那么这种移动是不合法的，玩家需要重新掷骰子。`continue gameLoop`语句结束本次`while`循环的迭代，开始下一次循环迭代。
- 在剩余的所有情况中，骰子数产生的都是合法的移动。玩家向前移动骰子数个方格，然后游戏逻辑再处理玩家当前是否处于蛇头或者梯子的底部。本次循环迭代结束，控制跳转到`while`循环体的条件判断语句处，再决定是否能够继续执行下次循环迭代。

注意：

如果上述的`break`语句没有使用`gameLoop`标签，那么它将会中断`switch`代码块而不是`while`循环体。使用`gameLoop`标签清晰的表明了`break`想要中断的是哪个代码块。 同时请注意，当调用`continue gameLoop`去跳转到下一次循环迭代时，这里使用`gameLoop`标签并不是严格必须的。因为在这个游戏中，只有一个循环体，所以`continue`语句会影响到哪个循环体是没有歧义的。然而，`continue`语句使用`gameLoop`标签也是没有危害的。这样做符合标签的使用规则，同时参照旁边的`break gameLoop`，能够使游戏的逻辑更加清晰和易于理解。