# 第3章

# 条件判断

本章介绍如何定义代码应该在何时执行的规则。这种语言特性被称为**控制流程**(**control flow**),它描述了程序的特定部分在何时运行的条件,重点介绍 if/else 语句及其表达式、when 表达式等,以及如何使用比较运算符和逻辑运算符编写 true/false 测试。

为了理解这些功能的实际操作,可以继续 bounty-board 项目的开发,创建呈现给玩家的任务。当玩家变得更强时,任务的难度会增加,这也是条件判断可以发挥作用的地方。

## 3.1 if/else 语句

就 bounty-board 项目而言,玩家的实力由第 2 章中创建的 playerLevel 变量的值来决定。该值越低表示英雄人物离其史诗之旅的出发点越近,该值越大表示英雄人物获得了更多的经验并变得更加强大了。

当前的目标是根据玩家的当前等级为其提供任务。例如,如果玩家处于第 1 级(新角色的起点),则需要给他们分配简单一点的任务,就像入门的教程。

在 **main**()函数中,编写第一个 if/else 语句,如程序清单 3.1 所示。输入新代码后再对其进行分解。

**程序清单 3.1　输出玩家的任务(Main.kt)**

```
const val HERO_NAME = "Madrigal"
fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 4
    println(playerLevel)

    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else {
        println("Locate the enchanted sword.")
    }

    println("The hero embarks on her journey to locate the enchanted sword.")
    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

首先,添加了一个 if/else 语句。在 if 关键字之后的括号中指定了一个条件。这里的条件提出了以

下 true/false 问题："玩家的 playerLevel 是否为 1?"可以使用**结构相等运算符**(**structural equality operator**)==来表示,读作"等于",所以,该语句可以读作"如果 playerLevel 等于 1"。

if 语句后面是大括号{}中的语句。如果 if 条件的计算结果为布尔值 true(本例中,如果 playerLevel 的值正好为 1),那么大括号内的代码就是希望程序执行的操作。

```
if (playerLevel == 1) {
    println("Meet Mr. Bubbles in the land of soft things.")
}
```

该语句中包含了熟悉的 **println**()函数,用于将特定的内容输出至控制台。简而言之,到目前为止,if/else 语句表示,如果 Madrigal 处于第 1 级,程序应该输出一个初学者级别的任务。

虽然本例的 if 语句的大括号中仅包含了一条语句,但如果希望在 if 取值为 true 时执行多个操作,则可以在大括号中包含更多的语句。

如果 playerLevel 的值不是 1,怎么办呢? 在此情形下,if 语句的计算结果为 false,程序将跳过 if 后面的大括号中的表达式,并转至 else 语句。可以将 else 看作"否则"的意思:如果某个条件成立,就执行某个操作;否则就执行其他的操作。else 语句和 if 语句一样,后面是大括号中的一组表达式,告诉编译器应该做什么。但与 if 语句不同,它不需要定义条件。只要 if 条件不满足,就会执行,所以大括号紧跟在关键字 else 之后,如下所示。

```
else {
    println("Locate the enchanted sword.")
}
```

从 Kotlin 语言的角度来看,else 程序块是可选的。可以只有 if 语句而没有 else 分支,后面的章节中会看到类似的情形。在该情形下,当 if 语句的计算结果为 false 时,程序将执行 if 语句后面的任何语句。也可以声明一个空的 else 程序块,同样有效。

程序代码中分支之间的唯一区别是调用 **println**()函数时所包含的请求不同。对游戏进度来说,else 分支中分配给玩家的不是一个微不足道的任务,而是要求玩家"找到魔法剑"(locate the enchanted sword)。到目前为止,所看到的大多数函数调用都只用于将字符串输出至控制台。在第 4 章中将了解更多的函数,包括如何自定义函数。

通俗地来说,以上代码其实就是告诉编译器:"如果英雄当前的等级处于第 1 级,输出 Meet Mr. Bubbles in the land of soft things. 至控制台。否则,输出 Locate the enchanted sword. 至控制台。"

结构相等运算符==是一个**比较运算符**(**comparison operator**)。表 3.1 列出了 Kotlin 语言中的比较运算符。现在还不需要了解所有列出的这些运算符,稍后将学习关于它们的更多的知识。当需要使用运算符表示一定的条件时,可以参阅表 3.1。

**表 3.1 比较运算符**

| 运 算 符 | 描 述 |
|---|---|
| < | 计算左侧的值是否小于右侧的值 |
| <= | 计算左侧的值是否小于或等于右侧的值 |
| > | 计算左侧的值是否大于右侧的值 |
| >= | 计算左侧的值是否大于或等于右侧的值 |
| == | 计算左侧的值是否等于右侧的值 |
| != | 计算左侧的值是否不等于右侧的值 |
| === | 计算两个实例是否指向同一引用 |
| !== | 计算两个实例是否没有指向同一引用 |

言归正传，单击 **main**()函数左侧的“运行”按钮，运行 Main.kt。应该可以看到以下的输出：

```
The hero announces her presence to the world.
Madrigal
4
Locate the enchanted sword.
Time passes...
The hero returns from her quest.
5
```

由于定义的条件 playerLevel==1 为 false，因此，跳过了 if/else 语句中的 if 分支，执行的是 else 分支(此处，使用了术语**分支**(**branch**)，根据是否满足指定的条件，程序执行的流程进行了分支)。现在，尝试将变量 playerLevel 的值更改为 1，如程序清单 3.2 所示。

**程序清单 3.2　修改变量 playerLevel（Main.kt）**

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    ~~var playerLevel = 4~~
    var playerLevel = 1
    println(playerLevel)

    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else {
        println("Locate the enchanted sword.")
    }

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

再次运行该程序，将可以看到以下输出：

```
The hero announces her presence to the world.
Madrigal
1
Meet Mr. Bubbles in the land of soft things.
Time passes...
The hero returns from her quest.
2
```

现在，定义的条件取值为 true(变量 playerLevel 等于 1)，因此将触发 if 分支。

**1. 添加更多的条件判断**

确定任务的代码给出了英雄人物应该采取什么行动的粗略概念。入门级是一个很好的起点，但对于任何级别大于 1 的玩家来说，只有一个任务——找到魔法剑。一旦拥有了一把魔法剑，当然就不需要再找一把了。

为了使 if/else 语句更加精细，可以添加更多的条件判断来进行检查，并包含尽可能多的分支。可以使用 else if 分支来执行此类操作，其语法与 if 分支类似，但位于 if 语句和 else 语句之间。更新 if/else 语句以包括 4 个 else if 分支，检查变量 playerLevel 的中间值。此时，将赋给变量 playerLevel 的值改回

4，如程序清单 3.3 所示。

**程序清单 3.3 检查更多的玩家条件（Main.kt）**

```
const val HERO_NAME = "Madrigal"
fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 1
    var playerLevel = 4
    println(playerLevel)

    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        println("Save the town from the barbarian invasions.")
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("Locate the enchanted sword.")
        println("There are no quests right now.")
    }

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

更新后的逻辑见表 3.2。

**表 3.2 更新后的逻辑**

| 假设 Madrigal 的级别 | 输出如下消息 |
|---|---|
| 1 | Meet Mr. Bubbles in the land of soft things.<br>（在松软的土地上与 Bubbles 先生会面。） |
| 2～5 | Save the town from the barbarian invasions.<br>（从蛮族的入侵中拯救该城市。） |
| 6 | Locate the enchanted sword.<br>（找到魔法剑。） |
| 7 | Recover the long-lost artifact of creation.<br>（找回丢失已久的手工艺品。） |
| 8 | Defeat Nogartse，bringer of death and eater of worlds.<br>（击败死亡的使者、世界的食者——Nogartse。） |
| 9＋ | There are no quests right now.<br>（暂时没有任务。） |

再次运行该程序。由于 Madrigal 的 playerLevel 的值为 4，第一个 if 条件的值为 false，所以对应的

分支不会被执行。但是，else if (playerLevel <= 5)的值为 true，所以将会看到 Save the town from the barbarian invasions. 输出至控制台。

编译器会自上而下地计算 if/else 的条件，并在计算结果为 true 时立即停止检查。如果提供的条件均不是 true，那么将执行 else 分支。这就意味着条件判断的顺序是非常重要的：如果在检查 playerLevel==1 之前先检查了 playerLevel<=5，那么第 1 级的任务将永远不会被执行(请勿对代码进行此类更改。这里仅作为示例)。

```
if (playerLevel <= 5) {                    // Triggered for any value 5 or less
    println("Save the town from the barbarian invasions.")
} else if (playerLevel == 1) {             // Only triggered for a value of 1
    println("Meet Mr. Bubbles in the land of soft things.")
} else if (playerLevel == 6) {
    println("Locate the enchanted sword.")
} else if (playerLevel == 7) {
    println("Recover the long-lost artifact of creation.")
} else if (playerLevel == 8) {
    println("Defeat Nogartse, bringer of death and eater of worlds.")
} else {
    println("There are no quests right now.")
}
```

本例中，任何小于或等于 5 的变量 playerLevel 都将触发第一个条件，但只有值 1 才会触发第二个分支。因为第一个 if 条件会满足，所以 else if(playerLevel==1)分支永远不会被计算。

通过在初始的 if 条件取值为 false 时，编写更多带有条件判断的 else if 语句，可以在如何报告玩家等级方面增加更多精确的描述。试着改变变量 playerLevel 的值，以触发所定义的每个分支。完成后，将变量 playerLevel 的值改回 4。

**2. 嵌套的 if/else 语句**

bounty-board 中有一个任务：Save the town from the barbarian invasions.，这有点太抽象了。可以实现预期结果的方法有好几种，例如外交手段就是其中之一。假设玩家与蛮族部落的首领关系良好，他们可能就会通过与部落进行友好协商以消除误解。

为了让玩家更清楚地了解这些可能性，任务的名称将根据玩家是否与蛮族是朋友而改变。在更新确定任务的逻辑之前，还需要添加一个变量来跟踪玩家与蛮族之间是否存在友谊。

在定义了跟踪玩家与蛮族之间友谊的变量之后，需要再次调整 if/else 语句。当玩家的等级为 2～5 时，可以使用额外的嵌套 if/else 来输出正确的任务标题，当在程序清单 3.4 的代码中进行调整时，不要忘记在 else if(playerLevel==6)之前添加一个大括号 }。

**程序清单 3.4　检查玩家与蛮族之间的友谊(Main.kt)**

```
const val HERO_NAME = "Madrigal"
fun main() {
    println("The hero announces her presence to the world.")
    println(HERO_NAME)
    var playerLevel = 4
    println(playerLevel)

    val hasBefriendedBarbarians = true
```

```
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5){
        if (hasBefriendedBarbarians) {
          println("Convince the barbarians to call off their invasion.")
        } else {
          println("Save the town from the barbarian invasions.")
        }
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("There are no quests right now.")
    }

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

此处添加了一个 val 变量，其值为布尔型，表示玩家是否与蛮族交过朋友。还添加了一条 if/else 语句，当与蛮族是好友且玩家的等级介于 2～5 时，创建一个新的输出。记住变量 playerLevel 的值为 4，因此在运行程序时应该会看到一条新的消息。运行程序并查看，输出应为：

```
The hero announces her presence to the world.
Madrigal
4
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

如果看到的是任何其他的输出，检查代码是否与程序清单 3.4 中的完全一致——特别是变量 playerLevel 是否被赋值为 4。

嵌套的条件判断允许在分支中创建逻辑分支，这样检查的条件判断就可以更加精确。

**3. 更优雅的条件判断**

如果不对条件判断进行精确的控制，条件判断会泛滥成灾的。Kotlin 允许在保持条件判断简洁易读的同时，充分利用其有用性(usefulness)。

**1）逻辑运算符**

在 bounty-board 项目中，可能会出现更复杂的需要进行判断的情形。例如，假设玩家与蛮族是朋友，或者玩家自己就是蛮族，那么使用外交手段就是可能的；否则，如果激怒了蛮族部落，那么外交手段就会受阻。

可以使用一系列的 if/else 语句确定到底需要显示哪个任务，但最终会出现大量重复的代码，并且掩盖了条件判断的逻辑。有一种更优雅、对读者更友好的方法：在条件判断中使用逻辑运算符。

添加两个新的变量，并更新嵌套 if 语句的条件判断，以增强任务的逻辑，如程序清单 3.5 所示。

**程序清单 3.5　在条件判断中使用逻辑运算符(Main.kt)**

```
const val HERO_NAME = "Madrigal"

fun main() {
    println("The hero announces her presence to the world.")

    println(HERO_NAME)
    var playerLevel = 4
    println(playerLevel)

    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5){
        if (hasBefriendedBarbarians) {
        // Check whether diplomacy is an option
        if (!hasAngeredBarbarians &&
                (hasBefriendedBarbarians ||playerClass == "barbarian")) {
            println("Convince the barbarians to call off their invasion.")
        } else {
            println("Save the town from the barbarian invasions.")
        }
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("There are no quests right now.")
    }

    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

此处添加了两个名为 hasAngeredBarbarians 和 playerClass 的 val 变量，以跟踪该条件(这两个变量是只读的，因为不会在 bounty-board 项目运行时更改其值)。

首先，//之后的代码称为**代码注释(code comment)**。//右边的所有内容都是注释，编译器会将其忽略，因此，注释中不用讲究语法。注释有助于组织和添加有关代码的相关信息，使其他人(或将来的自己，到时候自己也未必记得所有的细节)更容易阅读。

接下来，在 if 语句中使用了若干**逻辑运算符(logical operator)**。逻辑运算符可以将比较运算符组合成更复杂的语句。

!称为**逻辑“非”运算符(logical ‘not’ operator)**，返回与布尔值相反的值：如果相应表达式的值为 true，则逻辑“非”运算的结果为 false，反之亦然。&&称为**逻辑“与”运算符(logical ‘and’ operator)**，只有当表达式左侧的条件与右侧的条件均为 true 时，逻辑“与”运算的结果才为 true。||称为**逻辑“或”运算符(logical ‘or’ operator)**，如果表达式左边的条件或右边的条件有一个(或两者均)为 true，则逻辑“或”运算的结果为 true。

表 3.3 给出了 Kotlin 的逻辑运算符。

**表 3.3 逻辑运算符**

| 运 算 符 | 描 述 |
|---|---|
| && | 逻辑“与”：当且仅当两者均为 true 时为 true(否则为 false) |
| \|\| | 逻辑“或”：如果其中一个为 true，则为 true(仅当两者均为 false 时才为 false) |
| ! | 逻辑“非”：返回布尔值的相反值 |

**注意**：当逻辑运算符进行组合时，其优先级别决定了计算的顺序。相同优先级的逻辑运算符从左到右运算。也可以通过将作为一个组进行计算的逻辑运算符放在一个圆括号中，来实现对操作的分组。

以下是逻辑运算符优先级的顺序，从最高到最低依次为：

!(逻辑“非”)

<(小于)，<=(小于或等于)，>(大于)，>=(大于或等于)

==(结构相等)，!=(不等于)

&&(逻辑“与”)

||(逻辑“或”)

回到 bounty-board 项目，查看新的条件判断：

```
if (!hasAngeredBarbarians &&
        (hasBefriendedBarbarians || playerClass == "barbarian")) {
    println("Convince the barbarians to call off their invasion.")
}
```

换言之，如果玩家没有激怒蛮族，且他们或者与蛮族是好朋友，或者他们自己就是蛮族，那么 bounty-board 项目将采取外交手段来阻止入侵。

Madrigal 没有激怒蛮族，她自己不是蛮族，但她和蛮族是好朋友。因此，满足判断条件，bounty-board 应当告知 Madrigal 与蛮族进行对话。运行程序并进行检查，应该可以看到以下输出：

```
The hero announces her presence to the world.
Madrigal
4
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

考虑一下在不使用逻辑运算符的情形下表达该逻辑所需的嵌套条件语句。这些运算符提供了一个清晰表达复杂逻辑的工具。

逻辑运算符不仅可用于条件判断，也可以用在许多表达式中，包括变量的声明中。添加一个新的布尔型变量，该变量封装了与蛮族进行对话所需的条件，并**重构**(**refactor**)(即在不改变行为的情形下重写)条件判断以使用新的变量，如程序清单 3.6 所示。

**程序清单 3.6 在变量声明中使用逻辑运算符(Main.kt)**

```
...
fun main() {
    ...
    if (playerLevel == 1) {
        println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        // Check whether diplomacy is an option
        val canTalkToBarbarians = !hasAngeredBarbarians &&
```

```
                (hasBefriendedBarbarians ||playerClass == "barbarian")

        if (!hasAngeredBarbarians &&
                (hasBefriendedBarbarians ||playerClass == "barbarian")) {
        if (canTalkToBarbarians) {
            println("Convince the barbarians to call off their invasion.")
        } else {
            println("Save the town from the barbarian invasions.")
        }
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long - lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("There are no quests right now.")
    }
    ...
}
```

此处，已将条件检查移至一个名为 canTalkToBarbarians 的新的 val 变量中，并更改了 if/else 语句以检查其值。这在功能上等同于之前编写的代码，但此处已将规则用赋值语句进行了替代。该值的名称清楚地表明了所定义的规则用“可读的”术语表达了什么：玩家是否与蛮族保持着对话。当程序的规则变得复杂时，这是一种特别有用的技术，有助于给将来的读者传达规则的明确含义。

再次运行程序，确保其功能与之前相同，输出也应是相同的。

### 2）条件表达式

现在，if/else 语句可以正确地给出适当的任务，并且有一些微妙之处。

另外，对其进行更改可能会显得有些笨拙，因为每个分支中都重复了一个类似的 **println** 语句。如果想要更改给出任务的整体格式怎么办？当前的程序状态需要在 if/else 语句中查找每个分支，并将每个 **println**()函数更改为新的格式。

可以通过将编写的 if/else 语句更改为条件表达式来解决该问题。**条件表达式**(**conditional expression**)类似于条件语句，只是将 if/else 赋值给一个稍后可以使用的值。在蛮族任务的分支中使用一个条件表达式，看看会是什么样子，如程序清单 3.7 所示。

**程序清单 3.7　使用条件表达式(Main.kt)**

```
...
fun main() {
    ...
    if (playerLevel == 1) {
      println("Meet Mr. Bubbles in the land of soft things.")
    } else if (playerLevel <= 5) {
        // Check whether diplomacy is an option
        val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")

        val barbarianQuest: String = if (canTalkToBarbarians) {
            println("Convince the barbarians to call off their invasion.")
            "Convince the barbarians to call off their invasion."
        } else {
            println("Save the town from the barbarian invasions.")
            "Save the town from the barbarian invasions."
```

```
        }
        println(barbarianQuest)
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
    } else {
        println("There are no quests right now.")
    }
    ...
}
```

通过 if/else 表达式，根据 canTalkToBarbarians 的值，为新变量 barbarianQuest 分配一个来自 if 语句中 case 的字符串值，这就是条件表达式的美妙之处。因为现在可以使用 barbarianQuest 变量输出蛮族的任务，所以可以使用单个 println 调用来处理这两种情形。

通过对复杂的 if/else 语句进行相同的更改，可以进一步理清任务逻辑。重构确定任务的逻辑，就可以让 6 个几乎相同的输出语句消失，如程序清单 3.8 所示。

**程序清单 3.8　使用条件表达式来确定任务（Main.kt）**

```
...
fun main() {
    ...
    val hasBefriendedBarbarians = true
    val hasAngeredBarbarians = false
    val playerClass = "paladin"
    val quest: String = if (playerLevel == 1) {
      println("Meet Mr. Bubbles in the land of soft things.")
      "Meet Mr. Bubbles in the land of soft things."
    } else if (playerLevel <= 5) {
        // Check whether diplomacy is an option
        val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")

        val barbarianQuest: String = if (canTalkToBarbarians) {
            "Convince the barbarians to call off their invasion."
        } else {
            "Save the town from the barbarian invasions."
        }
        println(barbarianQuest)
    } else if (playerLevel == 6) {
        println("Locate the enchanted sword.")
        "Locate the enchanted sword."
    } else if (playerLevel == 7) {
        println("Recover the long-lost artifact of creation.")
        "Recover the long-lost artifact of creation."
    } else if (playerLevel == 8) {
        println("Defeat Nogartse, bringer of death and eater of worlds.")
        "Defeat Nogartse, bringer of death and eater of worlds."
    } else {
        println("There are no quests right now.")
        "There are no quests right now."
    }

    println("The hero approaches the bounty board. It reads:")
    println(quest)
```

```
    println("Time passes...")
    println("The hero returns from her quest.")

    playerLevel += 1
    println(playerLevel)
}
```

如果厌倦了在更改程序时保持代码缩进，IntelliJ 可以提供帮助。选择 Code→Auto-Indent Lines 命令，就可以享受清晰缩进带来的简单乐趣了。

当需要根据条件分配变量时，就可以使用条件表达式。然而，请记住，当从每个分支分配的值是相同类型（如 quest 字符串）时，条件表达式通常是最直观的。

再次运行代码，以确保一切按预期运行。应该可以看到一些熟悉的输出（增加了一个输出），但现在的代码更加优雅且易于阅读了。

```
The hero announces her presence to the world.
Madrigal
4
The hero approaches the bounty board. It reads:
Convince the barbarians to call off their invasion.
Time passes...
The hero returns from her quest.
5
```

### 3）从 if/else 表达式中删除大括号

当匹配的条件只有一个符合的情形下，可以省略表达式的大括号（至少在语法上是这样的，稍后会详细介绍）。当一个分支仅包含一个表达式时，只能省略{}。在具有多个表达式的分支中省略它们将影响代码的计算方式，Kotlin 不允许使用没有语句或一对大括号的 if 语句。

先看一下没有括号的 quest 版本：

```
val quest: String = if (playerLevel == 1)
    "Meet Mr. Bubbles in the land of soft things."
else if (playerLevel <= 5) {
    // Check whether diplomacy is an option
    val canTalkToBarbarians = !hasAngeredBarbarians &&
            (hasBefriendedBarbarians || playerClass == "barbarian")
    if (canTalkToBarbarians) "Convince the barbarians to call off their invasion."
    else "Save the town from the barbarian invasions."
} else if (playerLevel == 6) "Locate the enchanted sword."
else if (playerLevel == 7) "Recover the long-lost artifact of creation."
else if (playerLevel == 8)
    "Defeat Nogartse, bringer of death and eater of worlds."
else "There are no quests right now."
```

该版本的 quest 条件表达式与原版本中的代码执行了相同的操作。它甚至用更少的代码表达了相同的逻辑。但觉得哪个版本更容易阅读和理解呢？如果选择的是带大括号的版本，那么选择的就是 Kotlin 社区更推荐的样式。

建议不要省略跨多行的条件语句或表达式的大括号。首先，如果没有大括号，会越来越难以理解分支的起始位置和结束位置以及添加的每个条件。其次，省略大括号会增加将来程序维护者更新分支时误读程序的风险。为此节省几行代码是不值得的。

此外，尽管上面的代码中使用或不使用大括号都可以表示相同的内容，但并非每个示例都是如此。如果在一个分支中有多个表达式，并且删除了该分支的大括号，则在该分支中仅执行第一个表达式。举

例如下：

```
var arrowsInQuiver = 2
if (arrowsInQuiver >= 5) {
    println("Plenty of arrows")
    println("Cannot hold any more arrows")
}
```

如果英雄已经有了5支或以上的箭，那么箭就足够多了，再也装不下了。若英雄只有两支箭，那么控制台不会有任何输出。但是，如果没有大括号，逻辑会发生变化，如下所示：

```
var arrowsInQuiver = 2
if (arrowsInQuiver >= 5)
    println("Plenty of arrows")
    println("Cannot hold any more arrows")
```

如果没有大括号，第二个 println 语句不再是 if 分支的一部分。虽然 Plenty of arrows 只在 arrowsInQuiver 至少为5时才会输出，但 Cannot hold any more arrows 总是会输出出来，无论英雄携带了多少支箭。

对于一个单行表达式，总体原则应该是："哪种表达方式对新手来说最清楚？"通常，对于单行表达式，删除大括号更易于阅读。例如，删除大括号后有助于澄清如下所示的简单的单行条件表达式：

```
val healthSummary = if (healthPoints != 100) "Need healing!" else "Looking good."
```

如果正在想："好吧，但我仍然不喜欢 if/else 语法，即使有大括号。它很**丑陋**(**ugly**)。"其实还可以用一种不那么冗长但更清晰的语法再次重写任务表达式。

## 3.2 区间

所有在 if/else 表达式中为 quest 编写的条件都是基于整数变量 playerLevel 的值来进行分支的。大多数分支都使用了结构相等运算符来检查变量 playerLevel 是否等于某个值，分支中使用多个比较运算符来检查变量 playerLevel 是否位于两个数值的区间范围内。对于后者，有一个更好的替代方案：Kotlin 使用**区间**(**range**)来表示一个线性值序列。

**区间运算符**(**range to operator**)(..)可用于创建一个闭区间。一个闭区间包括从..运算符左边值到右边值之间的所有值，因此 1..5 表示 1、2、3、4、5。同时，闭区间也可以表示字符序列。

可以使用 in 运算符检查某个值是否位于某个区间内。使用区间而不是<=来重构 quest 条件表达式，如程序清单 3.9 所示。

**程序清单 3.9　使用区间来重构 quest(Main.kt)**

```
...
fun main() {
    ...
    val quest: String = if (playerLevel == 1) {
        "Meet Mr. Bubbles in the land of soft things."
    } else if (playerLevel <= 5) {
    } else if (playerLevel in 2..5) {
        // Check whether diplomacy is an option
        val canTalkToBarbarians = !hasAngeredBarbarians &&
                (hasBefriendedBarbarians || playerClass == "barbarian")

        if (canTalkToBarbarians) {
            "Convince the barbarians to call off their invasion."
```

```
        } else {
            "Save the town from the barbarian invasions."
        }
    } else if (playerLevel == 6) {
        "Locate the enchanted sword."
    } else if (playerLevel == 7) {
        "Recover the long-lost artifact of creation."
    } else if (playerLevel == 8) {
        "Defeat Nogartse, bringer of death and eater of worlds."
    } else {
        "There are no quests right now."
    }
    ...
}
```

**好处**是在以上的条件判断中使用区间运算符可以很好地解决本章前面遇到的多个 else if 的问题。使用区间运算符，分支可以按任意顺序排列，代码的计算结果都是一样的。

除区间运算符外，还有一些用于创建闭区间的函数。例如，**downTo**()函数就可以创建一个降序而不是升序的闭区间。**until**()函数可以创建一个开区间，该区间不包括指定区间的上限。本章末尾的挑战之处可以看到更多类似的函数，在第 9 章中可以了解更多有关区间的内容。

## 3.3　when 表达式

when 表达式是 Kotlin 中另一种控制流程的机制。与 if/else 类似，when 表达式也可以用来编写需进行检查的条件判断，如果条件判断的计算结果为 true，则执行相应的代码。when 表达式提供了更简洁的语法，特别适合具有 3 个或更多分支的条件判断。

假设玩家是某个奇幻种族(fantasy race)的成员，如兽人(orc)或侏儒(gnome)等，这些奇幻种族在派系斗争中相互结盟。when 表达式用来判断所属的奇幻种族，并返回其所属派系的名称，如下所示：

```
val race = "gnome"
val faction: String = when (race) {
    "dwarf" -> "Keepers of the Mines"
    "gnome" -> "Tinkerers of the Underground"
    "orc", "human" -> "Free People of the Rolling Hills"
    else -> "Shadow Cabal of the Unseen Realm" // Unknown race
}
```

首先声明一个 val 变量 race，接下来声明第二个 val 变量 faction，其值由 when 表达式确定。when 表达式根据**箭头运算符**(**arrow operator**)->左侧的值检查 race 的值，当找到匹配项时，会将右侧的值分配给 faction。具有相同输出的多个 case(如 orc 和 human)可以放在一起，在->运算符之前用逗号进行分隔。

**注意**：箭头运算符->的用法在其他语言中不尽相同。实际上，正如将在本书后续章节中看到的，它在 Kotlin 语言中也有其他的用法。

默认情形下，when 表达式的作用类似于圆括号中提供的参数与大括号中指定的条件之间有一个结构相等运算符==。**参数**(**argument**)是作为输入提供给代码的数据。第 4 章中将了解更多关于参数的内容。

此处的 when 表达式示例中，race 就是作为参数出现的。因此，when 表达式将 race 的值(gnome)与第一个条件(dwarf)进行比较，以检查它们是否相等。它们并不相等，因此比较的结果为 false，when 表达式将转移至下一个条件判断。

下一个比较的结果是 true，因此，相应的分支 Tinkerers of the Underground 被赋给变量 faction。

**注意**：这里使用的是 when 表达式来为变量 faction 赋值。因为赋值发生在 when 表达式之外，所以一定会对变量 faction 进行赋值，这也就意味着 when 表达式一定会有返回值。

当将 When 语句用作表达式时（例如，对其执行赋值时），编译器将要求 When 语句**穷举**（**exhaustive**）所有可能的输入。在此情形下，如果没有 else 分支，when 语句就不可能穷尽所有可能的输入。因为 race 有太多未知的字符串可以取值。但是，如果代码中存在异常值，else 分支会添加一个回退选项（fallback option），这样，编译器就不会有问题了。

有时，when 表达式也可以在没有 else 分支的情形下穷举所有可能的输入。第 16 章中将可以看到相关的示例。

已经学习了如何使用 when 表达式，现在就可以细化 quest 逻辑的实现方式了。以前使用的是 if/else 表达式，但在本例中，when 表达式可使代码更简洁、可读性更强。一个实用的经验法则是，只要代码中包含 else if 分支，就可以用 when 表达式进行替换。

使用 when 表达式更新 quest 逻辑，如程序清单 3.10 所示。

**程序清单 3.10　使用 when 表达式重构 quest（Main.kt）**

```
...
fun main() {
    ...
    val quest: String = if (playerLevel == 1) {
        "Meet Mr. Bubbles in the land of soft things."
    val quest: String = when (playerLevel) {
        1 -> "Meet Mr. Bubbles in the land of soft things."
    } else if (playerLevel in 2..5) {
        in 2..5 -> {
            // Check whether diplomacy is an option
            val canTalkToBarbarians = !hasAngeredBarbarians &&
                    (hasBefriendedBarbarians || playerClass == "barbarian")

            if (canTalkToBarbarians) {
                "Convince the barbarians to call off their invasion."
            } else {
                "Save the town from the barbarian invasions."
            }
    } else if (playerLevel == 6) {
        "Locate the enchanted sword."
        6 -> "Locate the enchanted sword."
    } else if (playerLevel == 7) {
        "Recover the long-lost artifact of creation."
        7 -> "Recover the long-lost artifact of creation."
    } else if (playerLevel == 8) {
        "Defeat Nogartse, bringer of death and eater of worlds."
    }
        8 -> "Defeat Nogartse, bringer of death and eater of
worlds."
    else {
        "There are no quests right now."
        else -> "There are no quests right now."
    }
    ...
}
```

when 表达式和 if/else 表达式都定义了基于条件为 true 时需执行的条件判断和分支，从这一点来说二者的工作方式很类似。when 表达式的不同之处在于，不管 when 的参数是什么，都会自动**检查**

(**scope**)并与条件判断的左侧进行匹配。第 4 章和第 13 章中将更深入地讨论该检查机制(scoping)。想要快速了解,请参阅程序清单 3.10 中的 in 2..5 分支条件。

前面已经介绍了如何使用 in 关键字检查某个值是否位于某个区间内,类似于这里检查变量 playerLevel 的值。因为->运算符左侧的区间就是变量 playerLevel 的作用域,所以编译器在计算 when 表达式时,就好像变量 playerLevel 包含在每个分支条件中一样。

通常来说,when 表达式可以更好地表达代码背后的逻辑。在本例中,若使用 if/else 表达式实现相同的逻辑则需要 4 个 else if 分支,而 when 表达式要简洁得多。在 when 表达式的分支中嵌套 if/else 的模式并不常见,但 Kotlin 语言中的 when 表达式确实可以提供所需的所有灵活性。

运行 bounty-board 项目,以确认使用 when 表达式重构 quest 后,程序逻辑并没有任何改变。

**1. 带有变量声明的 when 表达式**

有时,会使用一个带有参数的 when 表达式,而这个参数只是为了计算 when 表达式而存在的。在 when 表达式的条件中使用变量的值通常非常方便。

例如,假设想给玩家分配一个可以反映其等级的头衔,但只有一个类型为 Int 的 totalExperience 变量。为了简单起见,要求必须累计 100 个经验点才能晋级(因此,等级 1 表示玩家的经验点范围为 0～99,等级 2 表示玩家的经验点范围为 100～199,以此类推)。用来解决此头衔生成的 when 表达式参考如下:

```
val playerLevel: Int = totalExperience / 100 + 1
val playerTitle: String = when (playerLevel) {
  1 -> "Apprentice"
  in 2..8 -> "Level " + playerLevel + " Warrior"
  9 -> "Vanquisher of Nogartse"
  else -> "Distinguished Knight"
}
```

但是,通过将变量声明移至 when 表达式的参数中,这样可以进一步简化代码,如下所示:

```
val playerTitle = when ( val playerLevel = totalExperience / 100 + 1 ) {
    1 -> "Apprentice"
    in 2..8 -> "Level " + playerLevel + " Warrior"
    9 -> "Vanquisher of Nogartse"
    else -> "Distinguished Knight"
}
```

在以上带有变量声明的 when 表达式中,变量 playerLevel 的值仅在 when 表达式内部有效,当表达式执行完后就会被清除。这样在每次需要使用该变量值时无须重新进行计算,还可以避免其他同名的变量将代码的其余逻辑搞乱。

**2. 无参数的 when 表达式**

截至目前,所有用到的 when 表达式都有一个参数。这在基于单个变量来决定应用程序行为的情形下很有效。但是,带参数的 when 表达式存在一些局限。

(1) when 表达式不能接收多个参数。

(2) 带参数的 when 表达式中只允许使用==、in 或 is 运算符。

如果条件判断中涉及多个参数或需要使用不同的比较运算符时,就不能使用带参数的 when 表达式了。在这些情形下,可以有两个选择:使用 if/else 语句,就像在本章前面看到的那样;或者使用不带参数的 when 表达式。

假设想要告知玩家需要多少经验点才能进入游戏的下一个级别。有两个名为 experiencePoints 和 requiredExperiencePoint 的 Int 变量，升级所需经验量的 when 表达式参考如下：

```
val levelUpStatus: String = when {
    experiencePoints > requiredExperiencePoints -> {
        "You already leveled up!"
    }
    experiencePoints == requiredExperiencePoints -> {
        "You have enough experience to level up!"
    }
    requiredExperiencePoints - experiencePoints < 20 -> {
        // The player needs less than 20 experience points to level up
        "You are very close to leveling up!"
    }
    else -> "You need more experience to level up!"
}
```

这种灵活性意味着 if/else 语句和 when 表达式可以互换。任何用 if 语句进行检查的条件判断，也可以表示为不带参数的 when 表达式中的条件，甚至也可以使用表 3.3 所示的逻辑运算符。

## 3.4 挑战之处：灵活使用区间

区间是 Kotlin 中一个强大的工具，经过一定的练习，可以发现其语法非常直观。对于此处的简单挑战，请打开 Kotlin REPL（Tools→Kotlin→REPL），探索更多关于区间的语法，包括 toList()、downTo 和 until 函数。逐个输入以下区间，再按下组合键 Ctrl＋Enter 执行程序清单 3.11 中的程序并查看结果。

**程序清单 3.11 探索区间（REPL）**

```
1 in 1..3
(1..3).toList()
1 in 3 downTo 1
1 in 1 until 3
3 in 1 until 3
2 in 1..3
2 !in 1..3
'x' in 'a'..'z'
```