学习 **Kotlin** 编程语言

Kotlin 是一种编程语言被世界各地的 Android 开发者广泛使用。本主题可用作 Kotlin 速成课程，帮助您快速上手。

变量声明

Kotlin 使用两个不同的关键字（即 val 和 var）来声明变量。

- val 用于值从不更改的变量。使用 val 声明的变量无法重新赋值。
- var 用于值可以更改的变量。

在以下示例中，count 是一个 Int 类型的变量，初始赋值为 10：

```
var count: Int = 10
```

Int 是一种表示整数的类型，是可以用 Kotlin 表示的众多数值类型之一。与其他语言类似，您还可以使用 Byte、Short、Long、Float 和 Double，具体取决于您的数值数据。

var 关键字表示可以根据需要为 count 重新赋值。例如，可以将 count 的值从 10 更改为 15：

```
var count: Int = 10
count = 15
```

不过，有些值不应更改。假设有一个名为 languageName 的 String。如果希望确保 languageName 的值始终为“Kotlin”，则可以使用 val 关键字声明 languageName：

```
val languageName: String = "Kotlin"
```

通过这些关键字，您可以明确指出哪些变量的值可以更改。请根据需要加以利用。如果引用的变量必须可重新赋值，则将其声明为 var。否则，请使用 val。

类型推断

接着前面的示例来讲，为 languageName 赋予初始值后，Kotlin 编译器可根据

所赋值的类型来推断其类型。

由于 "Kotlin" 的值为 String 类型，因此编译器推断 languageName 也为 String。请注意，Kotlin 是一种静态类型的语言。这意味着，类型将在编译时解析且从不改变。

在以下示例中，languageName 推断为 String，因此无法对其调用任何不属于 String 类的函数：

```
val languageName = "Kotlin"
val upperCaseName = languageName.toUpperCase()

// Fails to compile
languageName.inc()
```

toUpperCase() 是一个只能对 String 类型的变量调用的函数。由于 Kotlin 编译器已将 languageName 推断为 String，因此可以安全地调用 toUpperCase()。不过，inc() 是一个 Int 运算符函数，因此无法对 String 调用它。利用 Kotlin 的类型推断，既能确保代码简洁，又能确保类型安全。

Null 安全

在某些语言中，可以声明引用类型变量而不明确提供初始值。在这类情况下，变量通常包含 null 值。默认情况下，Kotlin 变量不能持有 null 值。这意味着以下代码段无效：

```
// Fails to compile
val languageName: String = null
```

要使变量持有 null 值，它必须是可为 null 类型。可以在变量类型后面加上 ? 后缀，将变量指定为可为 null，如以下示例所示：

```
val languageName: String? = null
```

指定 String? 类型后，可以为 languageName 赋予 String 值或 null。

必须小心处理可为 null 的变量，否则可能会出现可怕的 NullPointerException。例如，在 Java 中，如果尝试对 null 值调用方法，程序会发生崩溃。

Kotlin 提供了多种机制来安全地处理可为 null 的变量。如需了解详情，请参阅 Android 平台中常见的 Kotlin 模式：可为 null 性。

条件语句

Kotlin 提供了几种实现条件逻辑的机制，其中最常见的是 if-else 语句。如果 if 关键字后面括在圆括号内的表达式求值为 true，则会执行该分支中的代码（即，紧跟在后面的括在大括号内的代码）。否则，会执行 else 分支中的代码。

```
if (count == 42) {
    println("I have the answer.")
} else {
    println("The answer eludes me.")
}
```

您可以使用 else if 表示多个条件。这样，您就可以在单个条件语句中表示更精细、更复杂的逻辑，如以下示例所示：

```
if (count == 42) {
    println("I have the answer.")
} else if (count > 35) {
    println("The answer is close.")
} else {
    println("The answer eludes me.")
}
```

条件语句对于表示有状态的逻辑很有用，但您可能会发现，编写这些语句时会出现重复。在上面的示例中，每个分支都是输出一个 String。为了避免这种重复，Kotlin 提供了条件表达式。最后一个示例可以重新编写如下：

```
val answerString: String = if (count == 42) {
    "I have the answer."
} else if (count > 35) {
    "The answer is close."
} else {
    "The answer eludes me."
}
```

```
println(answerString)
```

每个条件分支都隐式地返回其最后一行的表达式的结果，因此无需使用 return 关键字。由于全部三个分支的结果都是 String 类型，因此 if-else 表达式的结果也是 String 类型。在本例中，根据 if-else 表达式的结果为 answerString 赋予了一个初始值。利用类型推断可以省略 answerString 的显式类型声明，但为了清楚起见，通常最好添加该声明。

注意：Kotlin 不包含传统的三元运算符，而是倾向于使用条件表达式。

随着条件语句的复杂性不断增加，您可以考虑将 if-else 表达式替换为 when 表达式，如以下示例所示：

```
val answerString = when {
    count == 42 -> "I have the answer."
    count > 35 -> "The answer is close."
    else -> "The answer eludes me."
}

println(answerString)
```

when 表达式中每个分支都由一个条件、一个箭头 (->) 和一个结果来表示。如果箭头左侧的条件求值为 true，则会返回右侧的表达式结果。请注意，执行并不是从一个分支跳转到下一个分支。when 表达式示例中的代码在功能上与上一个示例中的代码等效，但可能更易懂。

Kotlin 的条件语句彰显了它的一项更强大的功能，即智能类型转换。您不必使用安全调用运算符或非 null 断言运算符来处理可为 null 的值，而可以使用条件语句来检查变量是否包含对 null 值的引用，如以下示例所示：

```
val languageName: String? = null
if (languageName != null) {
    // No need to write languageName?.toUpperCase()
    println(languageName.toUpperCase())
}
```

在条件分支中，languageName 可能会被视为不可为 null。Kotlin 非常智能，能够识别执行分支的条件是 languageName 不持有 null 值，因此您不必在该分

支中将 languageName 视为可为 null。这种智能类型转换适用于 null 检查、类型检查，或符合合约的任何条件。

函数

您可以将一个或多个表达式归入一个函数。您可以将相应的表达式封装在一个函数中并调用该函数，而不必在每次需要某个结果时都重复同一系列的表达式。

要声明函数，请使用 fun 关键字，后跟函数名称。接下来，定义函数接受的输入类型（如果有），并声明它返回的输出类型。函数的主体用于定义在调用函数时调用的表达式。

以前面的示例为基础，下面给出了一个完整的 Kotlin 函数：

```
fun generateAnswerString(): String {
    val answerString = if (count == 42) {
        "I have the answer."
    } else {
        "The answer eludes me"
    }

    return answerString
}
```

上面示例中的函数名为 generateAnswerString。它不接受任何输入。它会输出 String 类型的结果。要调用函数，请使用函数名称，后跟调用运算符 (())。在下面的示例中，使用 generateAnswerString() 的结果对 answerString 变量进行了初始化。

```
val answerString = generateAnswerString()
```

函数可以接受参数输入，如以下示例所示：

```
fun generateAnswerString(countThreshold: Int): String {
    val answerString = if (count > countThreshold) {
        "I have the answer."
    } else {
        "The answer eludes me."
```

```
    }

    return answerString
}
```

在声明函数时，可以指定任意数量的参数及其类型。在上面的示例中，generateAnswerString() 接受一个名为 countThreshold 的 Int 类型的参数。在函数中，可以使用参数的名称来引用参数。

调用此函数时，必须在函数调用的圆括号内添加一个参数：

```
val answerString = generateAnswerString(42)
```

简化函数声明

generateAnswerString() 是一个相当简单的函数。该函数声明一个变量，然后立即返回结果。函数返回单个表达式的结果时，可以通过直接返回函数中包含的 if-else 表达式的结果来跳过声明局部变量，如以下示例所示：

```
fun generateAnswerString(countThreshold: Int): String {
    return if (count > countThreshold) {
        "I have the answer."
    } else {
        "The answer eludes me."
    }
}
```

您还可以将 return 关键字替换为赋值运算符：

```
fun generateAnswerString(countThreshold: Int): String = if (count > c
        "I have the answer"
    } else {
        "The answer eludes me"
    }
```

匿名函数

并非每个函数都需要一个名称。某些函数通过输入和输出更直接地进行标识。这些函数称为“匿名函数”。您可以保留对某个匿名函数的引用，以便日后

使用此引用来调用该匿名函数。与其他引用类型一样，您也可以在应用中传递引用。

```
val stringLengthFunc: (String) -> Int = { input ->
    input.length
}
```

与命名函数一样，匿名函数也可以包含任意数量的表达式。函数的返回值是最终表达式的结果。

在上面的示例中，stringLengthFunc 包含对一个匿名函数的引用，该函数将 String 当作输入，并将输入 String 的长度作为 Int 类型的输出返回。因此，该函数的类型表示为 (String) -> Int。不过，此代码不会调用该函数。要检索该函数的结果，您必须像调用命名函数一样调用该函数。调用 stringLengthFunc 时，必须提供 String，如以下示例所示：

```
val stringLengthFunc: (String) -> Int = { input ->
    input.length
}

val stringLength: Int = stringLengthFunc("Android")
```

高阶函数

一个函数可以将另一个函数当作参数。将其他函数用作参数的函数称为“高阶函数”。此模式对组件之间的通信（其方式与在 Java 中使用回调接口相同）很有用。

下面是一个高阶函数的示例：

```
fun stringMapper(str: String, mapper: (String) -> Int): Int {
    // Invoke function
    return mapper(str)
}
```

stringMapper() 函数接受一个 String 以及一个函数，该函数根据您传递给它的 String 来推导 Int 值。

要调用 stringMapper()，可以传递一个 String 和一个满足其他输入参数的函

数（即，一个将 String 当作输入并输出 Int 的函数），如以下示例所示：

```
stringMapper("Android", { input ->
    input.length
})
```

如果匿名函数是在某个函数上定义的最后一个参数，则您可以在用于调用该函数的圆括号之外传递它，如以下示例所示：

```
stringMapper("Android") { input ->
    input.length
}
```

在整个 Kotlin 标准库中可以找到很多匿名函数。如需了解详情，请参阅高阶函数和 Lambda。

类

到目前为止提到的所有类型都已内置在 Kotlin 编程语言中。如果您希望添加自己的自定义类型，可以使用 class 关键字来定义类，如以下示例所示：

```
class Car
```

属性

类使用属性来表示状态。属性是类级变量，可以包含 getter、setter 和后备字段。由于汽车需要轮子来驱动，因此您可以添加 Wheel 对象的列表作为 Car 的属性，如以下示例所示：

```
class Car {
    val wheels = listOf<Wheel>()
}
```

请注意，wheels 是一个 public val，这意味着，可以从 Car 类外部访问 wheels，并且不能为其重新赋值。如果要获取 Car 的实例，必须先调用其构造函数。这样，您便可以访问它的任何可访问属性。

```
val car = Car() // construct a Car
val wheels = car.wheels // retrieve the wheels value from the Car
```

如果希望自定义轮子，可以定义一个自定义构造函数，用来指定如何初始化类属性：

```
class Car(val wheels: List<Wheel>)
```

在以上示例中，类构造函数将 List<Wheel> 当作构造函数参数，并使用该参数来初始化其 wheels 属性。

类函数和封装

类使用函数对行为建模。函数可以修改状态，从而帮助您只公开希望公开的数据。这种访问控制机制属于一个面向对象的更大概念，称为“封装”。

在以下示例中，doorLock 属性对 Car 类外部的一切都不公开。要解锁汽车，必须调用 unlockDoor() 函数并传入有效的“钥匙”，如以下示例所示：

```
class Car(val wheels: List<Wheel>) {

    private val doorLock: DoorLock = ...

    fun unlockDoor(key: Key): Boolean {
        // Return true if key is valid for door lock, false otherwise
    }
}
```

如果希望自定义属性的引用方式，则可以提供自定义的 getter 和 setter。例如，如果希望公开属性的 getter 而限制访问其 setter，则可以将该 setter 指定为 private：

```
class Car(val wheels: List<Wheel>) {

    private val doorLock: DoorLock = ...

    var gallonsOfFuelInTank: Int = 15
        private set

    fun unlockDoor(key: Key): Boolean {
        // Return true if key is valid for door lock, false otherwise
    }
```

```
}
```

通过结合使用属性和函数，可以创建能够对所有类型的对象建模的类。

互操作性

Kotlin 最重要的功能之一就是它与 Java 之间流畅的互操作性。由于 Kotlin 代码可编译为 JVM 字节码，因此 Kotlin 代码可直接调用 Java 代码，反之亦然。这意味着，您可以直接从 Kotlin 利用现有的 Java 库。此外，绝大多数 Android API 都是用 Java 编写的，因此可以直接从 Kotlin 调用它们。

后续措施

Kotlin 是一种灵活而实用的语言，它的支持力量和发展势头日益强劲。如果您还没有尝试过，我们建议您试一下。如需了解后续步骤，请参阅 官方 Kotlin 文档 以及如何申请 常见的 Kotlin 模式。

最后更新时间 (UTC)：2025-03-06。

- 微信
  在微信中关注 Android 开发者