# 彻底掌握Kotlin

互操作性与可空性

属性、异常互操作

函数类型

A

B

C

# 讲师简介

## 宁传奇

**Jason**

曾任华为高级工程师、架构师，全栈开发爱好者。

**技术这条路，不和你一比高下，只伴你一同成长。**

# 互操作性与可空性

# 互操作性与可空性

- Java世界里所有对象都可能是null，当一个Kotlin函数返回String类型值，你不能想当然地认为它的返回值就能符合Kotlin关于空值的规定。

```
8       val adversary = Jhava()
9       println(adversary.utterGreeting())
10      //返回String!类型的值，这里的感叹号表示返回值是String或者String?，
11      //至于Java方法返回的是String类型值是null还是其他什么，Kotlin编译器并不知道。
12      //这种模棱两可的返回值类型，我们称之为平台类型。
13      val level : String! = adversary.determineFriendshipLevel()
14      //null值问题最有可能来自互操作，所以从Kotlin里调用Java代码时，一定要小心谨慎。
15      level?.toLowerCase()
```

# 类型映射

- 代码运行时，所有的映射类型都会重新映射回对应的Java类型。

```
17      val hitPoints: Int = adversary.hitPoints
18      println(hitPoints.dec())
19      println(hitPoints.javaClass)
```

# 属性、异常互操作

# 属性访问

- 不需要调用相关setter方法，你可以使用赋值语法来设置一个Java字段值了。

```
21        adversary.greeting = "Hello"
```

# @JvmName

- 可以使用@JvmName注解指定编译类的名字。

```
//Kotlin顶层函数在Java里都被当作静态方法看待和调用。
System.out.println(Hero.makeProclamation());
```

# @JvmField

- 在Java里，不能直接访问spells字段，所以必须调用getSpells，然而，你可以给Kotlin属性添加@JvmField注解，暴露它的支持字段给Java调用者，从而避免使用getter方法。

```
class Spellbook {
    @JvmField
    val spells = listOf("Magic Ms. L","Lay on Hans")
}
```

# @JvmOverloads

- @JvmOverloads注解协助产生Kotlin函数的重载版本。设计一个可能会暴露给Java用户使用的API时，记得使用@JvmOverloads注解，这样，无论你是Kotlin开发者还是Java开发者，都会对这个API的可靠性感到满意。

```
37  //带默认参数的函数
38  //英雄能在handOverFood这个函数里送出食物，因为带默认参数，所以函数调用者可自由选择怎么调用这个函数
39  //例如，调用者可以指定英雄左手或右手拿什么食物，或者使用默认的配置——左手拿浆果，右手拿牛肉。
40  //只用简单两行代码，Kotlin就给了函数调用者选择的自由。
41  @JvmOverloads
42  fun handOverFood(leftHand: String = "berries", rightHand: String = "beef") {
43      println("Mmmm... you hand over some delicious $leftHand and $rightHand")
44  }
```

# @JvmStatic

- @JvmField注解还能用来以静态方式提供伴生对象里定义的值。
- @JvmStatic注解的作用类似于@JvmField，允许你直接调用伴生对象里的函数。

```
class Spellbook {
    companion object{
        @JvmField
        val MAX_SPELL_COUNT = 10
        @JvmStatic
        fun getSpellbookGreeting() = println("I am the Great Grimoire!")
    }
}
```

# @Throws

- 抛出一个需要检查的指定异常，Java和Kotlin有关异常检查的差异让@Throws注解给解决掉了，在编写供Java开发者调用的Kotlin API时，要考虑使用@Throws注解，这样，用户就知道怎么正确处理任何异常了。

```
49  @Throws(IOException::class)
50  fun acceptApology() {
51      throw IOException()
52  }
```

# 函数类型

# 函数类型操作

- 函数类型和匿名函数能提供高效的语法用于组件间的交互，是Kotlin编程语言里比较新颖的特性。他们简洁的语法因->操作符而实现，但Java8之前的JDK版本并并不支持lambda表达式。在Java里，Kotlin函数类型使用FunctionN这样的名字的接口来表示的，FunctionN中的N代表值参数目。这样的Function接口由23个，从Function0到Function22，每一个FunctionN都包含一个invoke函数，专用于调用函数类型函数，所以，任何时候需要调一个函数类型，都用它调用invoke。