Lecture 5

COMP 3717- Mobile Dev with Android Tech

第5讲

COMP 3717 - 使用Android技术进行移动开发

Association

- Association is a *has-a* relationship between classes
- There are two ways to achieve association
  - Composition
  - Aggregation

A关联

- 关联是类之间的一种拥有关系
- 有两种方式可以实现关联
  - 组合
  - 聚合

Association (cont.)

- Composition is when an objects lifecycle is determined by its parent

```
class ClassA{
    val b = ClassB()
}

class ClassB
```

- When a *ClassA* object is created/destroyed, so is *b*

A关联（续）

- 组合是指 对象的生命周期由其父类决定

```
class ClassA{
    val b = ClassB()
}

class ClassB
```

- 当创建/销毁一个*ClassA*对象时，*b*也会随之创建/销毁

Association (cont.)

- Aggregation is when the objects lifecycle is not determined by its parent

```
class ClassA(val b:ClassB)

class ClassB
```

- *b* is created/destroyed outside of *ClassA*

A关联（续）

- 聚合是指 对象 的生命周期不由其父类决定

```
class ClassA(val b:ClassB)

class ClassB
```

- *b*在 *ClassA* 之外被创建/销毁

Association vs Inheritance

- Inheritance has it’s use cases, but often a *has-a* relationship is better
- Consider *POSSystem* and *Restaurant* below

```
interface POSSystem{
    fun processOrder()
    fun calculateFoodInventory()
}
```

```
open class Restaurant : POSSystem{
    override fun processOrder() {
        println("Processing Order...")
    }
    override fun calculateFoodInventory() {
        println("Calculating food inventory...")
    }
}
```

关联与继承

- 继承有其适用场景，但通常 *has-a* 关系更优
- 考虑 POSSystem 和 Restaurant如下

```
interface POSSystem{
    fun processOrder()
    fun calculateFoodInventory()
}
```

```
open class Restaurant : POSSystem{
    override fun processOrder() {
        println("Processing Order...")
    }
    override fun calculateFoodInventory() {
        println("Calculating food inventory...")
    }
}
```

Association vs Inheritance (cont.)

- First off, we can achieve the same functionality using either relationship
- is-a

```
class KrustyKrab : Restaurant()
```

- has-a with delegation (*r* is the delegate)

```
class KrustyKrab{
    private val r = Restaurant()
    fun processOrder() = r.processOrder()
    fun calculateFoodInventory() = r.calculateFoodInventory()
}
```

关联与继承（续）

- 首先，我们可以使用任一种关系实现相同的功能
- 是一种

```
class KrustyKrab : Restaurant()
```

- 具有-并使用 委托 （*r*是被委托者）

```
class KrustyKrab{
    private val r = Restaurant()
    fun processOrder() = r.processOrder()
    fun calculateFoodInventory() = r.calculateFoodInventory()
}
```

Delegation

- Delegation is a way of achieving *association* (has-a) between two objects rather than *inheriting* (is-a)
- Delegation is a design pattern that Kotlin supports natively with interfaces
  - Uses the **by** keyword
- Delegation means delegating responsibilities to another object

委托

- 委托是一种在两个对象之间实现关联（拥有）关系的方式，而不是通过继承（是）关系
- 委托是一种 Kotlin 通过接口原生支持的设计模式
  - 使用**by**关键字
- 委托意味着将责任委托给另一个对象

Delegation (cont.)

- If we use *POSSystem* as the delegate instead of *Restaurant*

```
class KrustyKrab(private val posSystem: POSSystem){
    fun processOrder() = posSystem.processOrder()
    fun calculateFoodInventory() = posSystem.calculateFoodInventory()
}
```

- The delegate becomes more flexible
  - This is harder to achieve with inheritance

委托 (续)

- 如果我们使用 *POSSystem* 作为委托者，而不是 *Restaurant*

```
class KrustyKrab(private val posSystem: POSSystem){
    fun processOrder() = posSystem.processOrder()
    fun calculateFoodInventory() = posSystem.calculateFoodInventory()
}
```

- 委托变得更加灵活
  - 这在继承中较难实现

Delegation (cont.)

- We can achieve the same logic by implementing *POSSystem* and overriding its members

```
class KrustyKrab(private val posSystem: POSSystem) : POSSystem{
    override fun processOrder() {
        posSystem.processOrder()
    }

    override fun calculateFoodInventory() {
        posSystem.calculateFoodInventory()
    }
}
```

委托 （续）

- 我们可以通过 实现*POSSystem* 并重写其成员来实现相同的逻辑


```
class KrustyKrab(private val posSystem: POSSystem) : POSSystem{
    override fun processOrder() {
        posSystem.processOrder()
    }

    override fun calculateFoodInventory() {
        posSystem.calculateFoodInventory()
    }
}
```

Delegation (cont.)

- Furthermore, Kotlin supports interface delegation natively by reducing a lot of the boilerplate code

```
class KrustyKrab(posSystem: POSSystem) : POSSystem by posSystem
```

- The *by* keyword means *provided by* the delegate

委托 (续)

- 此外，Kotlin 通过减少大量样板代码


```
class KrustyKrab(posSystem: POSSystem) : POSSystem by posSystem
```


- 这个 *by*关键字意味着 由 委托方提供

Delegation (cont.)

- Summary
  - With inheritance, we are limited to the parent class type
  - With delegation, the delegate can be swapped easily making our class more flexible and reusable
  - The *by* keyword in Kotlin is used to delegate responsibility to something else

委托（续）

- 摘要
  - 使用继承时，我们受限于父类的类型
  - 使用委托时，可以轻松更换委托对象，使我们的类更加灵活且可重用
  - Kotlin 中的 *by* 关键字用于将责任委托给其他对象

Delegation (cont.)

- Kotlin also provides some standard delegates
  - Lazy
  - Observable
  - Vetoable

委托（续）

- Kotlin 还提供了一些标准委托
  - 延迟（Lazy）
  - 可观察（Observable）
  - 可否决（Vetoable）

Lazy delegate

- All properties can use the lazy delegate
- Initializing a property as lazy means it will only be initialized when it is first used
- Since it is only initialized the first time is it used, by nature it can only be declared as *val*

惰性委托

- 所有属性都可以使用延迟委托
- 将属性初始化为惰性意味着它仅在首次使用时才会被初始化，首次使用
- 由于它仅在第一次使用时才被初始化，因此本质上只能声明为 *val*

Lazy delegate(cont.)

- When we make a property lazy, we delegate its getter to the *lazy* delegate

```
fun main() {

    val lazyVal: String by lazy{
        "Hello World"
    }

    println(lazyVal)
    println(lazyVal)
```

- The first call to get will execute the lambda and remember it's result
- Any call to get afterwards will return the remembered result

惰性委托（续）

- 当我们使一个属性变为延迟时，我们将其 getter 委托给 *lazy* 委托器

```
fun main() {

    val lazyVal: String by lazy{
        "Hello World"
    }

    println(lazyVal)
    println(lazyVal)
```

- 第一次调用 get 将执行 lambda 并记住其结果
- 之后对 get 的任何调用都将返回已记住的结果

Lazy delegate(cont.)

- Use the lazy delegate if you need
  - a read-only property, where you want to delay or avoid its initialization

- Let's say we have a regular class that has-a property that performs a heavy operation

```
fun main() {
    RegularClass()
}

class HeavyClass{
    init{
        println("Some heavy processing...")
    }
}

class RegularClass{
    val heavy = HeavyClass()
}
```

```
"C:\Program Files\Android\Android Stud
Some heavy processing...

Process finished with exit code 0
```

惰性委托（续）

- 如果你需要
  - 一个只读属性，并希望延迟或避免其初始化

- 假设我们有一个普通类，它拥有一个 执行繁重操作的属性

```
fun main() {
    RegularClass()
}

class HeavyClass{
    init{
        println("Some heavy processing...")
    }
}

class RegularClass{
    val heavy = HeavyClass()
}
```

```
"C:\Program Files\Android\Android Stud
Some heavy processing...

Process finished with exit code 0
```

Lazy delegate(cont.)

- By making the heavy property lazy, we avoid its initialization

```
fun main() {
    RegularClass()
}

class HeavyClass{
    init{
        println("Some heavy processing...")
    }
}

class RegularClass{
    //val heavy = HeavyClass()
    val heavy by lazy {
        HeavyClass()
    }
}
```

```
"C:\Program Files\Android\Android Stu

Process finished with exit code 0
```

惰性委托（续）

- 通过将重量级属性 延迟化，我们可以避免其初始化

```
fun main() {
    RegularClass()
}

class HeavyClass{
    init{
        println("Some heavy processing...")
    }
}

class RegularClass{
    //val heavy = HeavyClass()
    val heavy by lazy {
        HeavyClass()
    }
}
```

```
"C:\Program Files\Android\Android Stu

Process finished with exit code 0
```

Observable delegate

- To observe the changes to a property you can use the observable delegate

```
class Sponge{
    var name: String by Delegates.observable( initialValue: "Bob"){ _, oldValue, newValue ->
        println("old value: $oldValue")
        println("new value: $newValue")
    }
}
```

- We delegate the property's setter to the *observable* delegate
  - Provides the old and new value whenever we set our property

可观察委托

- 要观察属性的变化，可以使用 observable 委托

```
class Sponge{
    var name: String by Delegates.observable( initialValue: "Bob"){ _, oldValue, newValue ->
        println("old value: $oldValue")
        println("new value: $newValue")
    }
}
```

- 我们将属性的 setter 委托给 *observable* 委托
  - 在设置属性时提供旧值和新值

Observable delegate (cont.)

- Each time we set *name*, the observable delegate is invoked

```
fun main() {

    val sponge = Sponge()
    sponge.name = "Spongebob"
    sponge.name = "Mr. SquarePants"
}
```

```
"C:\Program Files\Android\Android Stu
old value: Bob
new value: Spongebob
old value: Spongebob
new value: Mr. SquarePants

Process finished with exit code 0
```

可观察委托（续）

- 每次我们设置 *name* 时，可观察委托都会被调用

```
fun main() {

    val sponge = Sponge()
    sponge.name = "Spongebob"
    sponge.name = "Mr. SquarePants"
}
```

```
"C:\Program Files\Android\Android Stu
old value: Bob
new value: Spongebob
old value: Spongebob
new value: Mr. SquarePants

Process finished with exit code 0
```

Vetoable delegate

- To veto the changes to a property, we use the vetoable delegate

```
class Sponge{
    var friends:Int by Delegates.vetoable( initialValue: 3){_, oldValue, newValue ->
        println("old value: $oldValue")
        println("new value: $newValue")
        newValue >= 0 ^vetoable
    }
}
```

- We delegate the property's setter to a *vetoable* delegate
  - Provides the old and new value (like observable)
  - If the lambda returns false, then the changes will be vetoed

可否决的委托

- 为了否决对属性的更改，我们使用可否决的委托

```
class Sponge{
    var friends:Int by Delegates.vetoable( initialValue: 3){_, oldValue, newValue ->
        println("old value: $oldValue")
        println("new value: $newValue")
        newValue >= 0 ^vetoable
    }
}
```

- 我们将属性的 setter 委托给一个 *vetoable* delegate
  - 提供旧值和新值（类似于 observable）
  - 如果 lambda 返回 false，则更改将被否决

Vetoable delegate (cont.)

- When we try to set *friends* to a vetoable condition, the value doesn't change

```
fun main() {

    val sponge = Sponge()
    sponge.friends = -4
    println(sponge.friends)
    sponge.friends = 4
    println(sponge.friends)
}
```

```
"C:\Program Files\Android\Android St
old value: 3
new value: -4
3
old value: 3
new value: 4
4

Process finished with exit code 0
```

可否决的委托（续）

- 当我们尝试将 *friends* 设置为可否决状态时，值不会改变

```
fun main() {

    val sponge = Sponge()
    sponge.friends = -4
    println(sponge.friends)
    sponge.friends = 4
    println(sponge.friends)
}
```

```
"C:\Program Files\Android\Android St
old value: 3
new value: -4
3
old value: 3
new value: 4
4

Process finished with exit code 0
```

Extension Functions

- Often, we are working with classes that we don't own

```
fun main() {

    val str: String = "free krabby patties"
}
```

- E.g., we didn't create the *String* class, it comes from the Kotlin standard library

扩展函数

- 通常，我们使用的类并非由我们自己拥有

```
fun main() {

    val str: String = "free krabby patties"
}
```

- 例如，我们并没有创建 *String* 类，它来自 Kotlin 标准库

Extension Functions (cont.)

- We might need some custom functionality from that class
- Usually in this case we would create our own function

```
fun myGetAllWords(str:String): List<String>{
    return str.split( ...delimiters: " ")
}
```

- It would be better though if we could call this function as part of the original class

扩展函数（续）

- 我们可能需要该类的一些自定义功能
- 通常在这种情况下，我们会创建自己的函数

```
fun myGetAllWords(str:String): List<String>{
    return str.split( ...delimiters: " ")
}
```

- 但如果我们能将此函数作为原始类的一部分来调用，会更好一些

Extension Functions (cont.)

- Below is how we can transform that logic into an extension function

```
fun String.getAllWords() : List<String>{
    return this.split( ...delimiters: " ")
}
```

- The class you want to extend
  - aka. the receiver
- The instance of the class you are extending

扩展函数（续）

- 以下是我们如何将该逻辑转换为扩展函数

```
fun String.getAllWords() : List<String>{
    return this.split( ...delimiters: " ")
}
```

- 你想要扩展的类
  - 又称接收者
- 被扩展类的实例对象

Extension Functions (cont.)

- You can also drop *this*

```
fun main() {

    val str = "free krabby patties"
    println(str.getAllWords())
}

fun String.getAllWords() : List<String>{
    return split( ...delimiters: " ")
}
```

- In summary, extension functions
  - allow us to add functionality to classes we don't own; and
  - call these new functions as if they were part of the original class

扩展函数（续）

- 你也可以丢弃 这个

```
fun main() {

    val str = "free krabby patties"
    println(str.getAllWords())
}

fun String.getAllWords() : List<String>{
    return split( ...delimiters: " ")
}
```

- 总之，扩展函数
  - 允许我们向不属于我们的类添加功能；并且
  - 可以像调用原始类的一部分那样调用这些新函数

Extension Properties

- We can create extension properties as well
  - The *receiver is a String*
  - The instance

```
fun main() {

    val str = "free krabby patties"
    println(str.capitalizeFirstLetter)
}

val String.capitalizeFirstLetter : String
    get() = this.substring(0, 1).uppercase() + substring( startIndex: 1)
```

扩展属性

- 我们也可以创建扩展属性
  - 接收者是字符串类型
  - 实例对象

```
fun main() {

    val str = "free krabby patties"
    println(str.capitalizeFirstLetter)
}

val String.capitalizeFirstLetter : String
    get() = this.substring(0, 1).uppercase() + substring( startIndex: 1)
```

Lambda with Receiver

- You can also create *extension function literals*
  - aka. Function literal with a receiver; or
  - *lambda with a receiver*

```
val reverseDigits: Int.() -> Int = { this: Int
    toString().reversed().toInt()
}
```

带接收者的 Lambda

- 你还可以创建 扩展函数字面量
  - 也称为 函数字面量 带有 接收者；或者
  - 带接收者的 lambda

```
val reverseDigits: Int.() -> Int = { this: Int
    toString().reversed().toInt()
}
```

Lambda with Receiver

- Providing the reciever can be done in two ways
  - Standard

```
println(145.reverseDigits())
```

  - More explicit

```
println(reverseDigits(145))
```

Provides the reciever as p1 (parameter 1) which is what is happening behind the scenes when working with a *lambda with reciever*

带接收者的 Lambda

- 提供接收者有两种方式
  - 标准方式

```
println(145.reverseDigits())
```

  - 更明确的方式

```
println(reverseDigits(145))
```

将接收者作为 p1（第一个参数）提供，这正是在使用 带接收者的 lambda 时幕后发生的过程

Lambda with Receiver (cont.)

- Furthermore, they can help give us scope to object members

```
//class we don't have access too
class Sponge{
    val name = "Spongebob"
}

fun Sponge.fact(action: Sponge.()->Unit){
    this.action()
}
```

```
fun main() {
    val sponge = Sponge()
    sponge.fact {
        println("$name lives in a pineapple")
    }
}
```

带接收者的 Lambda（续）

- 此外，它们可以为我们提供对 对象成员 的作用域

```
//class we don't have access too
class Sponge{
    val name = "Spongebob"
}

fun Sponge.fact(action: Sponge.()->Unit){
    this.action()
}
```

```
fun main() {
    val sponge = Sponge()
    sponge.fact {
        println("$name lives in a pineapple")
    }
}
```

Generics

- Generics allows us to define functions or classes that can work with different data types (Int, String, Double, etc)

- This is useful because we only have to write one piece of code for multiple types

泛型

- 泛型允许我们定义能够处理不同数据类型（如整数、字符串、双精度浮点数等）的函数或类

- 这很有用，因为我们只需编写一次代码即可适用于多种类型

Generics

- Let's say we want to create some custom sorting behaviour for a list

- We want to be able to find an element in our list and sort it to the first index
  1.) First, we find the index of the element we want to sort
  2.) Remove it from the list
  3.) Add it to the first index

```
val indexOfElement = list.indexOf(element)
list.removeAt(indexOfElement)
list.add( index: 0, element)
```

泛型

- 假设我们想要为一个列表创建一些自定义的排序行为

- 我们希望能够找到列表中的某个元素，并将其排序到第一个索引位置
  1.) 首先，我们找到要排序的元素的索引
  2.) 将其从列表中移除
  3.) 将其添加到第一个索引位置

```
val indexOfElement = list.indexOf(element)
list.removeAt(indexOfElement)
list.add( index: 0, element)
```

Generics (cont.)

- Let's add this code to a class so we can easily reuse it
- We will pass the list into the constructor and the element into our sort function
- *indexOf* returns -1 if the element doesn't exist
  - We will just return the original list in this case

```
class IntListUtils(private val data: List<Int>) {

    fun sortElementFirst(element: Int): List<Int> {
        val list = data.toMutableList()
        val indexOfElement = list.indexOf(element)

        return if (indexOfElement != -1) {
            list.removeAt(indexOfElement)
            list.add( index: 0, element)
            return list
        } else {
            println("element doesn't exist")
            data
        }
    }
}
```

泛型（续）

- 让我们将这段代码添加到一个类中，以便可以轻松地重复使用
- 我们将把 列表传入 构造函数，并将 元素传入我们的排序函数
- *indexOf* 如果元素不存在则返回 -1 不存在
  - 在这种情况下，我们直接返回 原始列表 即可


```
class IntListUtils(private val data: List<Int>) {

    fun sortElementFirst(element: Int): List<Int> {
        val list = data.toMutableList()
        val indexOfElement = list.indexOf(element)

        return if (indexOfElement != -1) {
            list.removeAt(indexOfElement)
            list.add( index: 0, element)
            return list
        } else {
            println("element doesn't exist")
            data
        }
    }
}
```

Generics (cont.)

- This code works well, it finds the first given element and sorts it to the front of the list

```
fun main() {

    val list = listOf(1, 2, 3, 4, 5, 6)
    val customSortedList = IntListUtils(list).sortElementFirst( element: 3)
    println(customSortedList)
```

```
"C:\Program Files\Android\Android St
[3, 1, 2, 4, 5, 6]

Process finished with exit code 0
```

泛型（续）

- 这段代码运行良好，它会找到第一个给定的元素并将其排序到列表的开头

```
fun main() {

    val list = listOf(1, 2, 3, 4, 5, 6)
    val customSortedList = IntListUtils(list).sortElementFirst( element: 3)
    println(customSortedList)
```

```
"C:\Program Files\Android\Android St
[3, 1, 2, 4, 5, 6]

Process finished with exit code 0
```

Generics (cont.)

- The only problem is that our code only works for integers
  - What if we want to use this code for other types?
- When we make a class use generics, we need to add <T> to the end of the name

```
class ListUtils<T>(private val data: List<Int>) {
```

泛型（续）

- 唯一的问题是我们的代码仅适用于整数
  - 如果我们想将此代码用于其他类型该怎么办?
- 当我们让一个类使用泛型时，需要在名称末尾添加 <T>

```
class ListUtils<T>(private val data: List<Int>) {
```

Generics (cont.)

- You can use any letter for the generic type
- Some commonly used type parameter names are
  - T - Type
  - K - Key
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types

泛型（续）

- 你可以使用任意字母作为泛型类型
- 一些常用的类型参数名称包括
  - T - 类型
  - K - 键
  - V - 值
  - S、U、V等——第二、第三、第四种类型

Generics (cont.)

- Once you have created a generic type for your class
- All other classes, objects and functions in that class can use that generic type as parameters

```
class ListUtils<T>(private val data: List<T>) {

    fun sortElementFirst(element: T): List<T> {
        val list = data.toMutableList()
        val indexOfElement = list.indexOf(element)

        return if (indexOfElement != -1) {
            list.removeAt(indexOfElement)
            list.add( index: 0, element)
            return list
        } else {
            println("element doesn't exist")
            data
        }
    }
}
```

泛型（续）

- 一旦你为类创建了泛型类型
- 该类中的所有其他类、对象以及函数都可以使用该泛型类型作为参数

```
class ListUtils<T>(private val data: List<T>) {

    fun sortElementFirst(element: T): List<T> {
        val list = data.toMutableList()
        val indexOfElement = list.indexOf(element)

        return if (indexOfElement != -1) {
            list.removeAt(indexOfElement)
            list.add( index: 0, element)
            return list
        } else {
            println("element doesn't exist")
            data
        }
    }
}
```

Generics (cont.)

- Our class is using generics now and can handle any type of list

```
fun main() {

    val list = listOf("1", "2", "3", "4", "5", "6")
    val customSortedList = ListUtils(list).sortElementFirst( element: "3")
    println(customSortedList)
```

```
"C:\Program Files\Android\Android Stu
[3, 1, 2, 4, 5, 6]

Process finished with exit code 0
```

泛型（续）

- 我们的类现在使用了泛型，可以处理任何类型的列表

```
fun main() {

    val list = listOf("1", "2", "3", "4", "5", "6")
    val customSortedList = ListUtils(list).sortElementFirst( element: "3")
    println(customSortedList)
```

```
"C:\Program Files\Android\Android Stu
[3, 1, 2, 4, 5, 6]

Process finished with exit code 0
```

Generics (cont.)

- We can also just make a specific function generic, rather than a whole class
- The *<T>* is added to the front of the function name

```
class ListUtils {

    fun <T>sortElementFirst(data: List<T>, element: T): List<T> {
        val list = data.toMutableList()
        val indexOfElement = list.indexOf(element)

        return if (indexOfElement != -1) {
            list.removeAt(indexOfElement)
            list.add( index: 0, element)
            return list
        } else {
            println("element doesn't exist")
            data
        }
    }
}
```

```
fun main() {

    val list = listOf("1", "2", "3", "4", "5", "6")
    val customSortedList = ListUtils().sortElementFirst(list,  element: "3")
    println(customSortedList)
```

泛型（续）

- 我们也可以仅使特定函数具有泛型，而不是整个类
- 将 *<T>* 添加到函数名称的前面

```
class ListUtils {

    fun <T>sortElementFirst(data: List<T>, element: T): List<T> {
        val list = data.toMutableList()
        val indexOfElement = list.indexOf(element)

        return if (indexOfElement != -1) {
            list.removeAt(indexOfElement)
            list.add( index: 0, element)
            return list
        } else {
            println("element doesn't exist")
            data
        }
    }
}
```

```
fun main() {

    val list = listOf("1", "2", "3", "4", "5", "6")
    val customSortedList = ListUtils().sortElementFirst(list,  element: "3")
    println(customSortedList)
```

Generics (cont.)

- Why can't we just use *Any*?

```
fun sortElementFirst(data: List<Any>, element: Any): List<Any> {
    val list = data.toMutableList()
```

- Generics provides type inference, so it will detect the specific type
- *Any* does not have any type inference making it harder to work with

泛型（续）

- 为什么我们不能直接使用*Any*?

```
fun sortElementFirst(data: List<Any>, element: Any): List<Any> {
    val list = data.toMutableList()
```

- 泛型提供类型推断，因此它会检测到具体类型
- *Any*没有任何类型推断，使其更难使用

Comparing objects

- Sometimes you might want to compare objects rather than strings or primitive types

```
class Species(private val name:String)

fun main() {

    val sponge1 = Species( name: "Bob")
    val sponge2 = Species( name: "Bob")

    println(sponge1 == sponge2)
```

```
"C:\Program Files\Android\Android St
false

Process finished with exit code 0
```

比较对象

- 有时你可能希望比较对象而不是字符串或基本类型进行比较

```
class Species(private val name:String)

fun main() {

    val sponge1 = Species( name: "Bob")
    val sponge2 = Species( name: "Bob")

    println(sponge1 == sponge2)
```

```
"C:\Program Files\Android\Android St
false

Process finished with exit code 0
```

Comparing objects (cont.)

- In the previous example you might wonder why the result is false
- Object variables, or any object that is an instance of a class, *store their values as references* (aka. a location in memory)
- Primitive types *store their values with the data directly*
- When you compare two variables using the equality operator (==), we compare their values

比较对象（续）

- 在前面的例子中，你可能会疑惑为什么结果是 false
- 对象变量或任何类的实例 存储其值为引用（即内存中的位置）
- 基本类型直接以其数据存储值
- 当你使用相等运算符（==）比较两个变量时，我们比较的是它们的值

Pass by value

- Like Java, Kotlin passes its variables by value to functions (aka. *Pass-by-value*)

```
fun doSomething(s:Species){
    s.name = "Mr. SquarePants"
}

fun main() {

    val sponge = Species( name: "Bob")
    doSomething(sponge)
    println(sponge.name)
```

```
"C:\Program Files\Android\Android Studio\j
Mr. SquarePants

Process finished with exit code 0
```

- We pass in the value (*a copy* of the memory location) of *sponge* then alter the sponge object's data

按值传递

- 与 Java 类似，Kotlin 将变量按值传递给函数（即 按值传递）

```
fun doSomething(s:Species){
    s.name = "Mr. SquarePants"
}

fun main() {

    val sponge = Species( name: "Bob")
    doSomething(sponge)
    println(sponge.name)
```

```
"C:\Program Files\Android\Android Studio\j
Mr. SquarePants

Process finished with exit code 0
```

- 我们传入的是 值（内存位置的副本）*sponge*，然后修改 sponge 对象的数据


SEE THE TIME ON THE CLOCK?
IT'S BREAK TIME!


SEE THE TIME ON THE CLOCK?
IT'S BREAK TIME!

Data Class

- In Java, it is common to create a data class (POJO)
  - A simple class mainly used to hold data
- It is also common in Java for the POJO to
  - Implement getters/setters for each data member
  - Override the superclass's (*Object*) *toString*, *equals*, and *hashcode* functions
    - Kotlin uses Any not Object
- Doing all this helps us manage and transfer our data

数据类

- 在 Java 中，创建一个数据类（POJO）是很常见的
  - 一种主要用于保存数据的简单类
- 在 Java 中，POJO 通常还会
  - 为每个数据成员实现 getter/setter 方法
  - 重写父类（*Object*）的 *toString*、*equals* 和 *hashcode* 函数
    - Kotlin 使用 Any 而不是 Object
- 完成所有这些操作有助于我们管理和传输数据

Data Class (cont.)

- For example, we might want all our data to be represented in a String

```
class Species(
    val name:String,
    val height:Int,
    val occupation:String
)

fun main() {

    val sponge = Species( name: "Bob",  height: 4,  occupation: "Cook")
    println(sponge)
```

```
"C:\Program Files\Android\Android Studio
com.bcit.lecture5.Species@6cd8737

Process finished with exit code 0
```

- But you can see the output isn't very helpful

数据类（续）

- 例如，我们可能希望所有数据都以字符串形式表示

```
class Species(
    val name:String,
    val height:Int,
    val occupation:String
)

fun main() {

    val sponge = Species( name: "Bob",  height: 4,  occupation: "Cook")
    println(sponge)
```

```
"C:\Program Files\Android\Android Studio
com.bcit.lecture5.Species@6cd8737

Process finished with exit code 0
```

- 但可以看出输出并没有太大帮助

Data Class (cont.)

- We also saw earlier that it is common to compare two objects

```
class Species(private val name:String)

fun main() {

    val sponge1 = Species( name: "Bob")
    val sponge2 = Species( name: "Bob")

    println(sponge1 == sponge2)
```

```
"C:\Program Files\Android\Android St
false

Process finished with exit code 0
```

- If we only care about comparing the data in the two objects, then we would want this to print true

数据类（续）

- 我们之前还看到，比较两个对象是很常见的

```
class Species(private val name:String)

fun main() {

    val sponge1 = Species( name: "Bob")
    val sponge2 = Species( name: "Bob")

    println(sponge1 == sponge2)
```

```
"C:\Program Files\Android\Android St
false

Process finished with exit code 0
```

如果我们只关心比较两个对象中的数据，那么我们希望这会打印 true

Data Class (cont.)

- One way to get around this is to override those superclass functions
  - toString, equals, hashCode
- We would need to write similar boilerplate code for all our POJO's in Java

```
fun main() {

    val sponge1 = Species( name: "Bob")
    val sponge2 = Species( name: "Bob")
    println(sponge1)
    println(sponge1 == sponge2)
```

```
class Species(val name:String){

    override fun toString(): String {
        return "Species(name=$name)"
    }

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other == null) return false
        if (other !is Species) return false
        return name == other.name
    }

    override fun hashCode(): Int {
        return name.hashCode()
    }
}
```

```
"C:\Program Files\Android\Android S
Species(name=Bob)
true

Process finished with exit code 0
```

数据类（续）

- 解决此问题的一种方法是重写那些超类函数
  - toString, equals, hashCode
- 我们需要为 Java 中所有的 POJO 编写类似的 Java 中所有 POJO 的样板代码

```
fun main() {

    val sponge1 = Species( name: "Bob")
    val sponge2 = Species( name: "Bob")
    println(sponge1)
    println(sponge1 == sponge2)
```

```
class Species(val name:String){

    override fun toString(): String {
        return "Species(name=$name)"
    }

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other == null) return false
        if (other !is Species) return false
        return name == other.name
    }

    override fun hashCode(): Int {
        return name.hashCode()
    }
}
```

```
"C:\Program Files\Android\Android S
Species(name=Bob)
true

Process finished with exit code 0
```

Data Class (cont.)

- In Kotlin, all we must do is make the class a *data class* and we can achieve the same thing

```
data class Species(val name:String, val height:Int, val occupation:String)

fun main() {

    val sponge1 = Species( name: "Bob",  height: 4,  occupation: "Cook")
    val sponge2 = Species( name: "Bob",  height: 4,  occupation: "Cook")
    println(sponge1)
    println(sponge1 == sponge2)
```

```
"C:\Program Files\Android\Android Studio\jbr\b
Species(name=Bob, height=4, occupation=Cook)
true

Process finished with exit code 0
```

数据类（续）

- 在 Kotlin 中，我们只需将类定义为 数据类，即可实现相同的效果

```
data class Species(val name:String, val height:Int, val occupation:String)

fun main() {

    val sponge1 = Species( name: "Bob",  height: 4,  occupation: "Cook")
    val sponge2 = Species( name: "Bob",  height: 4,  occupation: "Cook")
    println(sponge1)
    println(sponge1 == sponge2)
```

```
"C:\Program Files\Android\Android Studio\jbr\b
Species(name=Bob, height=4, occupation=Cook)
true

Process finished with exit code 0
```

Data Class (cont.)

- Can't be *open* or *abstract*
- Can't have a blank constructor (needs at least one parameter)
- Can be deconstructed

```
data class Species(val name:String, val height:Int, val occupation:String)

fun main() {

    val sponge = Species( name: "Bob",  height: 4,  occupation: "cook")
    val (name, height, occupation) = sponge
    println("$name is a $occupation who is $height feet tall")
}
```

```
"C:\Program Files\Android\Android Stud
Bob is a cook who is 4 feet tall

Process finished with exit code 0
```

数据类（续）

- 不能是 开放的 或 抽象的
- 不能有空白构造函数（至少需要一个参数）
- 可以被解构

```
data class Species(val name:String, val height:Int, val occupation:String)

fun main() {

    val sponge = Species( name: "Bob",  height: 4,  occupation: "cook")
    val (name, height, occupation) = sponge
    println("$name is a $occupation who is $height feet tall")
}
```

```
"C:\Program Files\Android\Android Stud
Bob is a cook who is 4 feet tall

Process finished with exit code 0
```

Scope functions

- Scope functions are functions that execute a block of code within the context of an object
- The give *temporary scope* to an object where *specific operations* can be applied
- Scope functions make our code more concise and readable

作用域函数

- 作用域函数是指在对象的上下文
- 它们为对象提供临时作用域，以便在其中执行特定操作可以应用
- 作用域函数使我们的代码更加简洁和易读

Scope functions (cont.)

- There are five scope functions provided in the Kotlin standard library
  - *let*, *run*, *with*, *apply*, *also*

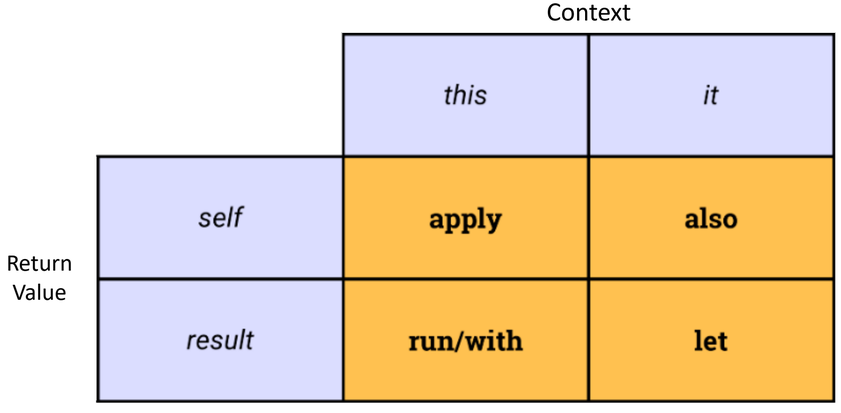
- We can differentiate these scope functions in two ways
  - Context: *it* or *this*
  - Return Value: *lambda result* or *itself (context object)*

作用域函数（续）

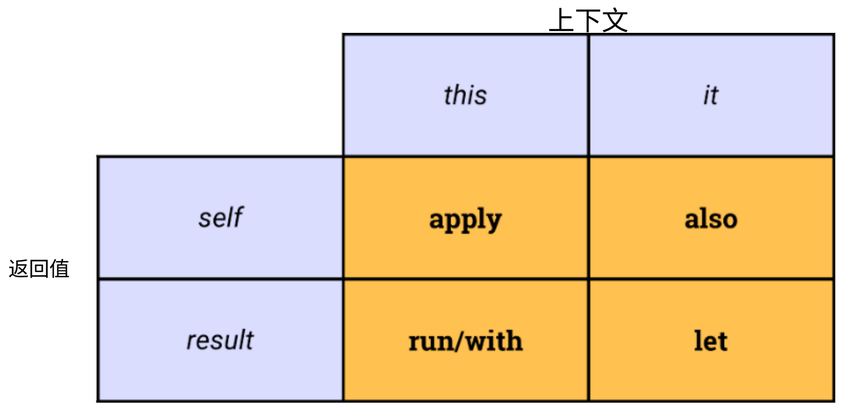
- Kotlin 标准库提供了五个作用域函数
  - 让，运行，使用，应用，还

- 我们可以通过两种方式来区分这些作用域函数
  - 上下文：*it* 或 *this*
  - 返回值： lambda 结果 或 自身（上下文对象）

Scope functions (cont.)



作用域函数（续）



Scope functions (cont.)

- Consider this class below

```
class Car(
    private var make:String? = null,
    private var color:String? = null,
    private var kilometers:Int
){
    fun drive(k:Int){
        kilometers += k
        println("The car drove $k kilometers")
    }

    fun paint(c:String){
        color = c
        println("Car has been panted $c")
    }

    override fun toString(): String {
        return "Car(make=$make, color=$color, kilometers=$kilometers)"
    }
}
```

- We will use this to explain the scope functions

作用域函数（续）

- 请考虑下面这个类

```
class Car(
    private var make:String? = null,
    private var color:String? = null,
    private var kilometers:Int
){
    fun drive(k:Int){
        kilometers += k
        println("The car drove $k kilometers")
    }

    fun paint(c:String){
        color = c
        println("Car has been panted $c")
    }

    override fun toString(): String {
        return "Car(make=$make, color=$color, kilometers=$kilometers)"
    }
}
```

- 我们将使用它来解释作用域函数

Scope function let

- *let*
  - Context: *it*
  - Return Value: *lambda result*

- Here we create a new Car object with *.let{}* at the end
  - The *scope* is defined in the curly braces

```
val car = Car( make: "ford",  color: "red",  kilometers: 50).let{ it: Car

}
```

作用域函数 let

- let
  - 上下文：*it*
  - 返回值：*lambda result*

- 在这里，我们使用 *.let{}* 结尾创建一个新的 Car 对象
  - 作用域 在花括号中定义

```
val car = Car( make: "ford",  color: "red",  kilometers: 50).let{ it: Car

}
```

Scope function let (cont.)

- Inside the scope block, *it* refers to the Car object itself
- Since let *returns the lambda result*, the last line in the block is what it returned

```
val car = Car( make: "ford", color: "red", kilometers: 50).let{ it: Car
    it.drive( k: 50)
    it.paint( c: "Blue")
    it ^let
}
println(car)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\
The car drove 50 kilometers
Car has been panted Blue
Car(make=ford, color=Blue, kilometers=100)

Process finished with exit code 0
```

作用域函数 let（续）

- 在作用域代码块内部，*it* 指向 Car 对象本身
- 由于 let 返回 lambda 的结果，因此 代码块中的最后一行 就是其返回值

```
val car = Car( make: "ford", color: "red", kilometers: 50).let{ it: Car
    it.drive( k: 50)
    it.paint( c: "Blue")
    it ^let
}
println(car)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\
The car drove 50 kilometers
Car has been panted Blue
Car(make=ford, color=Blue, kilometers=100)

Process finished with exit code 0
```

Scope function let (cont.)

- The *let* function is often used with null safety

```
var car1 = Car( make: "ford", color: "red", kilometers: 50)
val car2: Car? = null

car2?.let { car1 = it }

println(car1)
```

- Since *car2* is null, the let function won't be applied

作用域函数 let（续）

- *let* 函数通常与空安全一起使用

```
var car1 = Car( make: "ford", color: "red", kilometers: 50)
val car2: Car? = null

car2?.let { car1 = it }

println(car1)
```

- 由于 *car2* 为 null，因此不会应用 let 函数

Scope function run

- *run*
  - Context: *this*
  - Return Value: *lambda result*

- Very similar to *let* but we don't use *it*, we use *this*

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).run { this: Car
    this.drive( k: 60)
    this.paint( c: "Grey")
    this ^run
}
println(car1)
```

作用域函数执行

- 执行
  - 上下文：*this*
  - 返回值：lambda 结果

- 与*let*非常相似，但我们不使用*it*，而是使用*this*

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).run { this: Car
    this.drive( k: 60)
    this.paint( c: "Grey")
    this ^run
}
println(car1)
```

Scope function run (cont.)

- When the context is using *this*, we can drop *this* completely

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).run { this: Car
    drive( k: 60)
    paint( c: "Grey")
    this ^run
}
println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr
The car drove 60 kilometers
Car has been panted Grey
Car(make=ford, color=Grey, kilometers=110)

Process finished with exit code 0
```

作用域函数运行（续）

- 当上下文使用 *this* 时，我们可以 完全省略 *this*

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).run { this: Car
    drive( k: 60)
    paint( c: "Grey")
    this ^run
}
println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr
The car drove 60 kilometers
Car has been panted Grey
Car(make=ford, color=Grey, kilometers=110)

Process finished with exit code 0
```

Scope function with

- *with*
  - Context: *this*
  - Return Value: *lambda result*
- *with* is similar to *run* but the syntax is slightly different
- We can also omit returning *this* if we want

```
fun main() {

    val car1 = Car( make: "ford",  color: "red",  kilometers: 50)

    with(car1){ this: Car
        drive( k: 80)
        paint( c: "Blue")
        drive( k: 40)
    }

    println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr\
The car drove 80 kilometers
Car has been panted Blue
The car drove 40 kilometers
Car(make=ford, color=Blue, kilometers=170)

Process finished with exit code 0
```

带有作用域函数的

- with
  - 上下文：*this*
  - 返回值：*lambda result*
- *with* 与 *run* 类似，但语法上略有不同
- 我们也可以省略返回 *this* 如果我们 want

```
fun main() {

    val car1 = Car( make: "ford",  color: "red",  kilometers: 50)

    with(car1){ this: Car
        drive( k: 80)
        paint( c: "Blue")
        drive( k: 40)
    }

    println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr\
The car drove 80 kilometers
Car has been panted Blue
The car drove 40 kilometers
Car(make=ford, color=Blue, kilometers=170)

Process finished with exit code 0
```

Scope function apply

- *apply*
  - Context: *this*
  - Return Value: *itself (context object)*

- Since we return the context object, we don't need to return anything in the block

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).apply { this: Car
    drive( k: 20)
    paint( c: "Black")
}

println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin
The car drove 20 kilometers
Car has been panted Black
Car(make=ford, color=Black, kilometers=70)

Process finished with exit code 0
```

作用域函数 apply

- apply
  - 上下文：*this*
  - 返回值：自身（上下文对象）

- 由于我们返回的是上下文对象，因此在代码块中无需显式返回任何内容


```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).apply { this: Car
    drive( k: 20)
    paint( c: "Black")
}

println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin
The car drove 20 kilometers
Car has been panted Black
Car(make=ford, color=Black, kilometers=70)

Process finished with exit code 0
```

Scope function also

- *also*
  - Context: *it*
  - Return Value: *itself (context object)*

- Similar to *apply* but we use *it* instead of *this*

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).also { it: Car
    it.drive( k: 30)
    it.paint( c: "White")
    it.drive( k: 60)
}

println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\j
The car drove 30 kilometers
Car has been panted White
The car drove 60 kilometers
Car(make=ford, color=White, kilometers=140)

Process finished with exit code 0
```

作用域 函数也

- 也
  - 上下文：它
  - 返回值：自身（上下文对象）

- 类似于 *apply*，但我们使用 它 而不是 *this*

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).also { it: Car
    it.drive( k: 30)
    it.paint( c: "White")
    it.drive( k: 60)
}

println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\j
The car drove 30 kilometers
Car has been panted White
The car drove 60 kilometers
Car(make=ford, color=White, kilometers=140)

Process finished with exit code 0
```

Class Activity 1

- Given the Dog class, refactor the code in the main

```
class Dog{
    var name:String = "fluffy"
    var toys:MutableList<String> = mutableListOf()
    override fun toString(): String {
        return "Dog(name=$name, toys=$toys)"
    }
}
```

- Your code must use two different scope functions

```
fun main() {

    val dog1 = Dog()
    val dog2 = Dog()

    dog1.name = "sparky"
    dog2.toys.add("ball")
    dog2.toys.add("stick")
    dog1.toys = dog2.toys

    println(dog1)
```

课堂活动 1

- 根据 Dog 类，重构 main 中的代码


```
class Dog{
    var name:String = "fluffy"
    var toys:MutableList<String> = mutableListOf()
    override fun toString(): String {
        return "Dog(name=$name, toys=$toys)"
    }
}
```


- 你的代码必须使用两种不同的作用域函数


```
fun main() {

    val dog1 = Dog()
    val dog2 = Dog()

    dog1.name = "sparky"
    dog2.toys.add("ball")
    dog2.toys.add("stick")
    dog1.toys = dog2.toys

    println(dog1)
```


Class Activity 1 Answer

```
fun main() {

    val dog1 = Dog()
    val dog2 = Dog()

    with(dog1){ this: Dog
        name = "sparky"
        toys = dog2.toys.also { it: MutableList<String>
            it.add("ball")
            it.add("stick")
        }
    }

    println(dog1)
```

课堂活动1答案