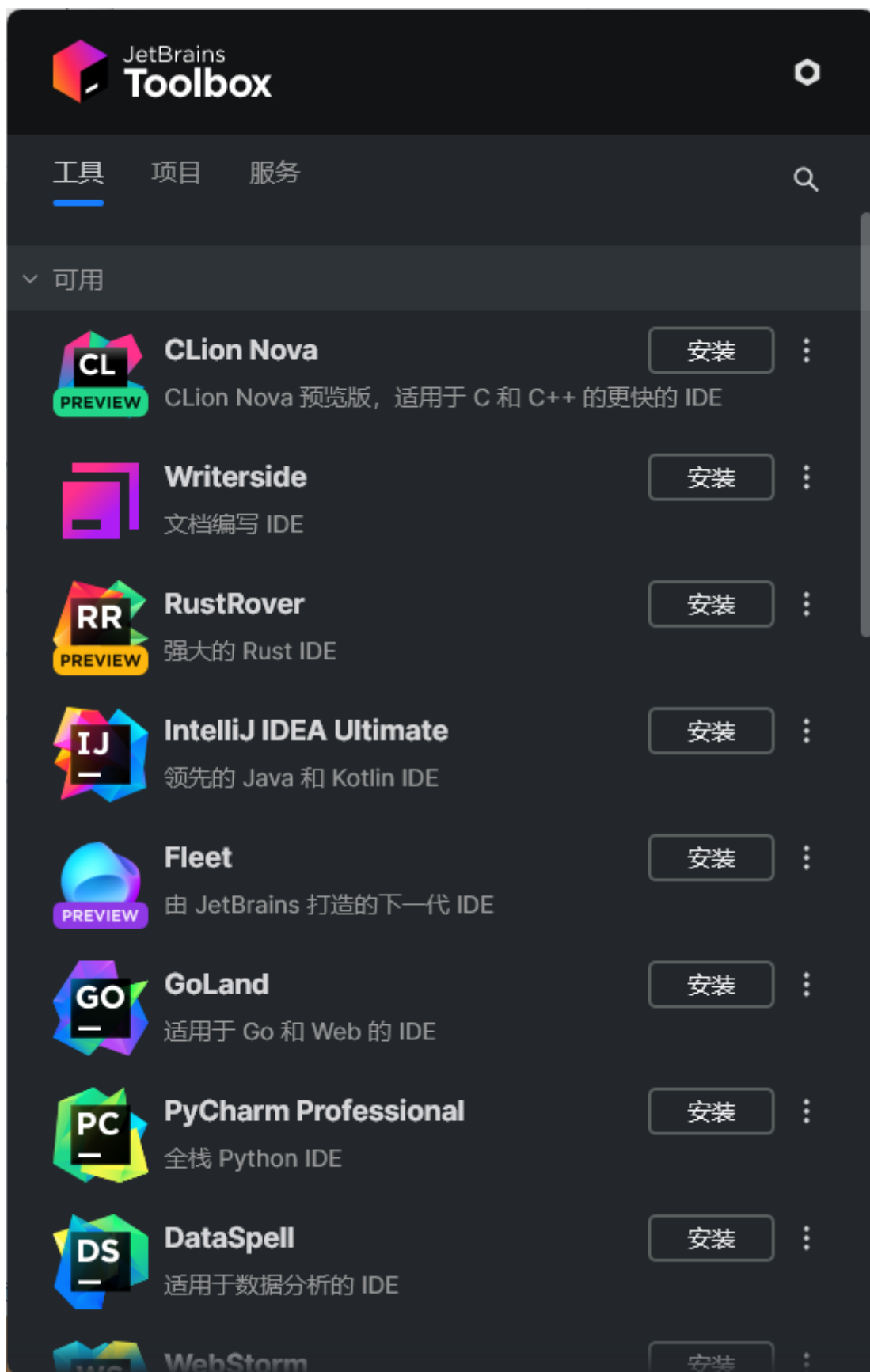


Kotlin入门

相信大家经历了学习的学习，应该对我们下学期要学习的Kotlin语言有过一定的听闻。大家应该或多或少对Kotlin这么语言有些问题，那么，接下来我们将真正踏入Kotlin安卓开发学习的过程。

Kotlin的一些历史

Kotlin是一种静态类型的编程语言，由JetBrains团队于2011年开发。JetBrains公司相信大家应该并不陌生了，我们编程过程总会或多或少用到JetBrains公司的产品。



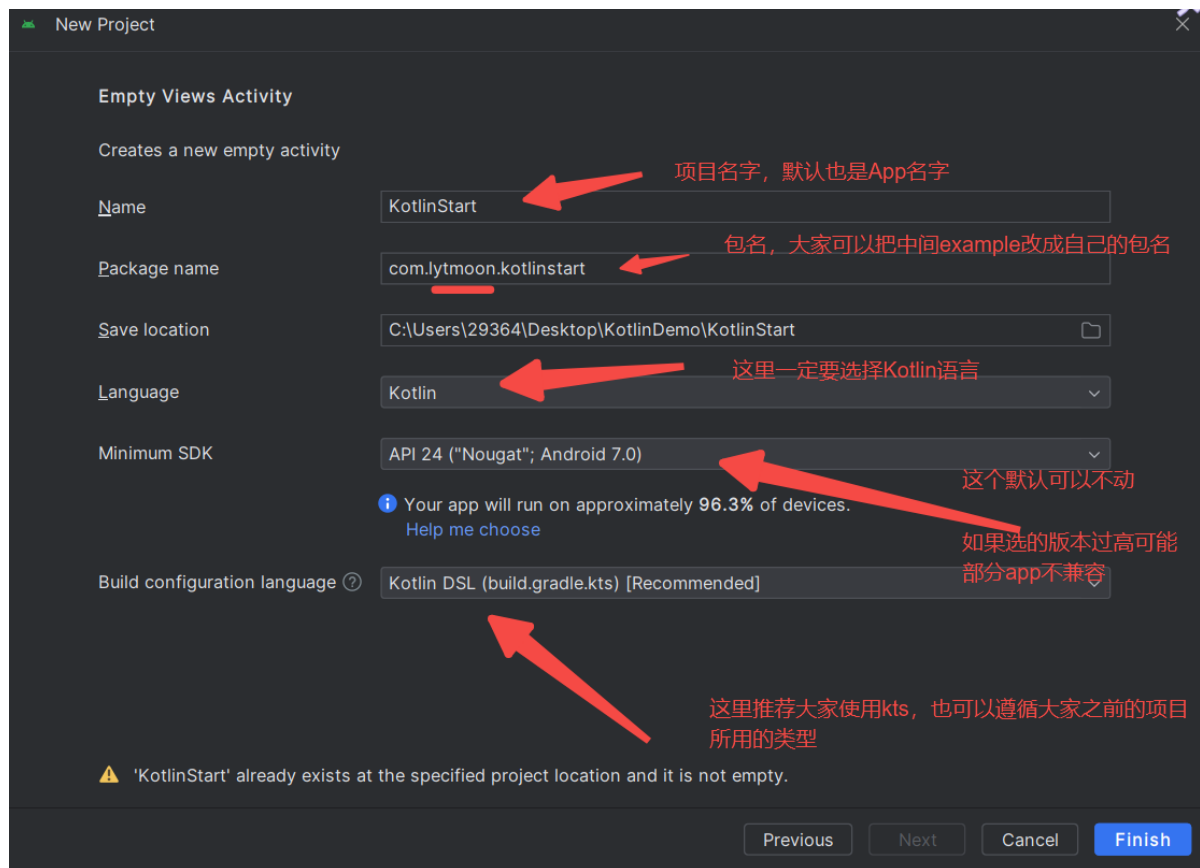
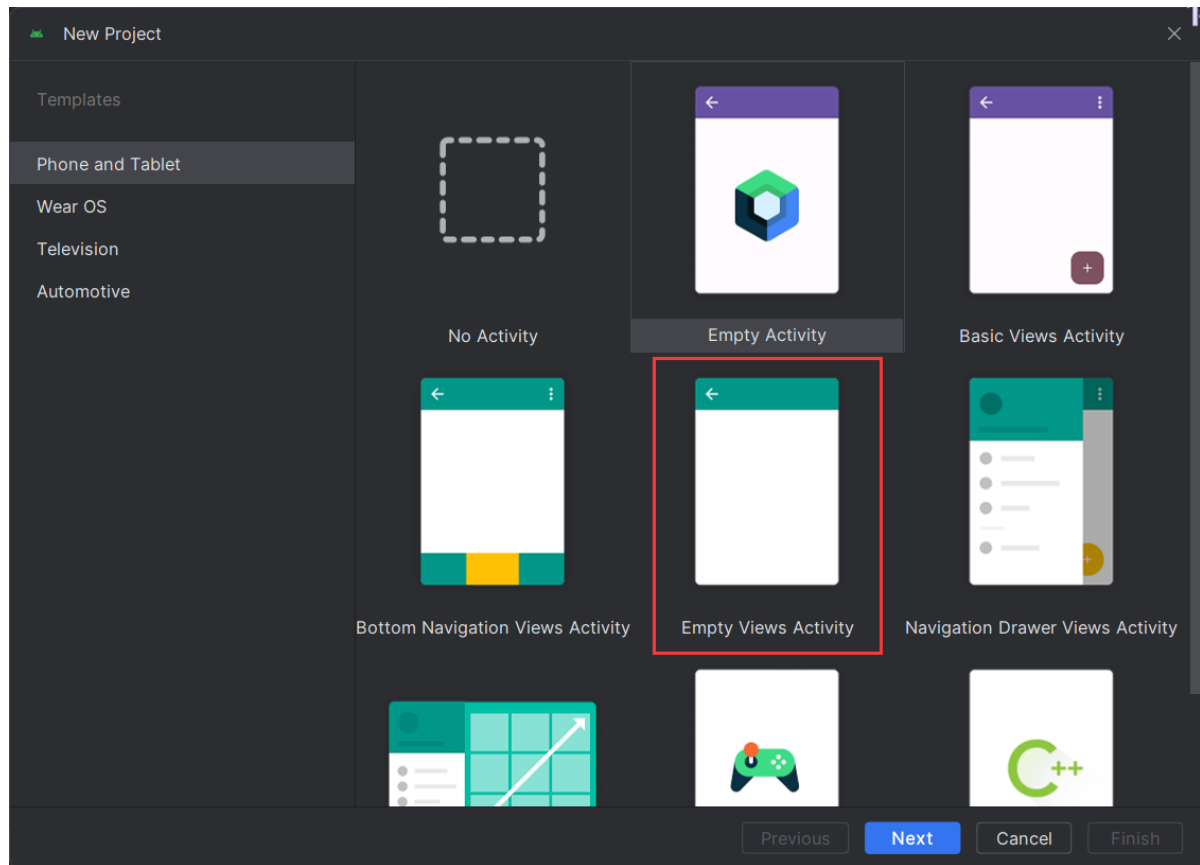
Kotlin诞生之后，在2017年的Google I/O大会上，Google宣布在Android上为Kotlin提供最佳支持，这标志着Kotlin成为Android开发的官方语言，我们Google的英文官网上阅读相关文档的时候也都能发现示例代码都是由Kotlin实现。

Kotlin旨在提供比Java更简洁更安全的编程体验，它对Java提供了百分百的支持兼容。

了解过Kotlin的一些发展历史后，接下来我们来正式学习这门语言。

使用Kotlin写安卓教程

我们现在AS的 new project界面选择 empty views activity (不要选择empty activity,现在AS更新, 部分老教程可能并不太适用)。



这里大家如果想要导入其他依赖的话依照这种写法就行了，与之前的不同，这里需要加入大括号、单引号变成双引号，写法不同。

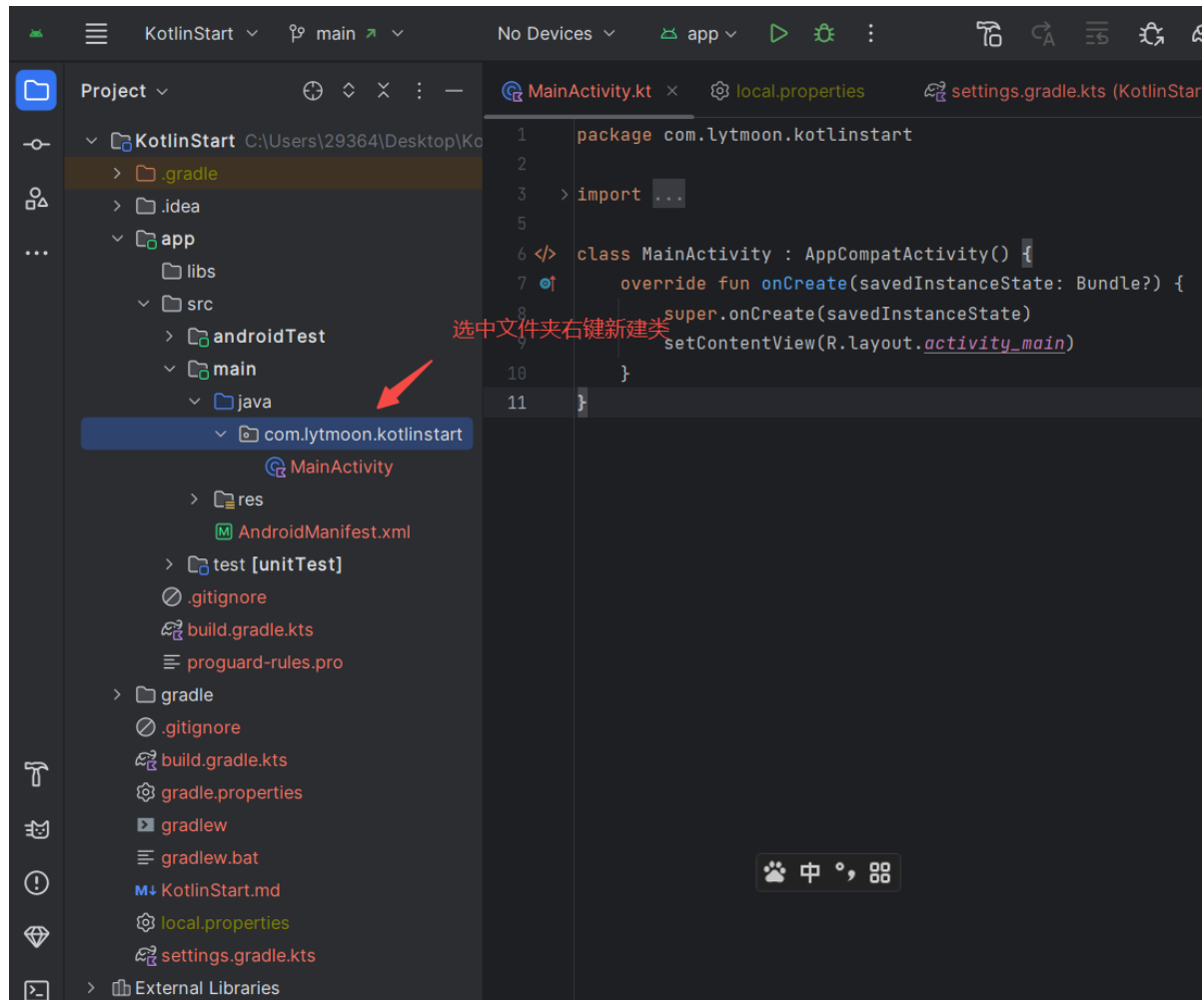
下面我们来学习一下Kotlin这门语言的一些语法。)

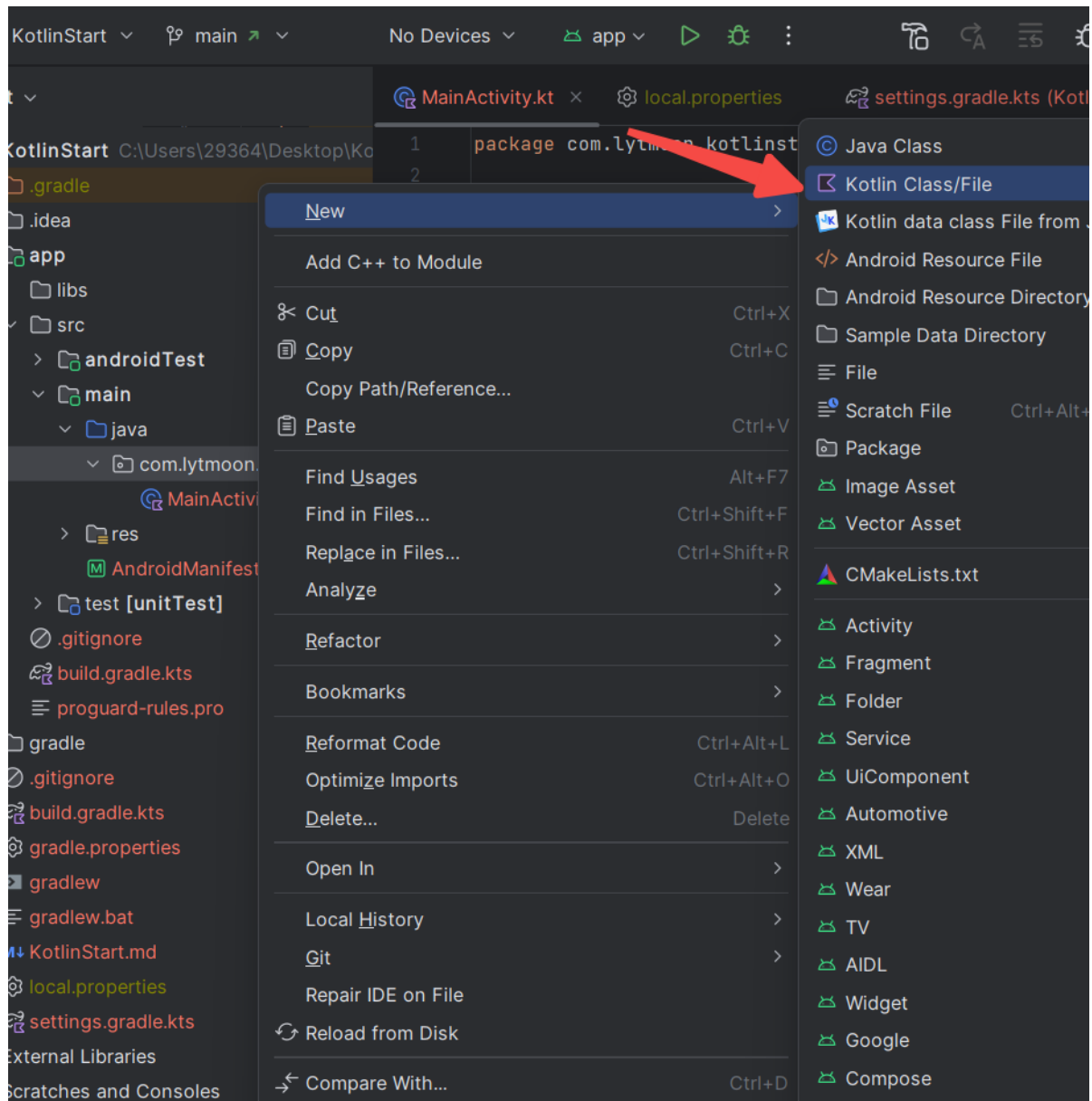
Kotlin语法

变量和函数

变量

为了运行Kotlin代码，我们可以先新建一个Kotlin类，方便我们写Kotlin代码





```
1 package com.lytmoon.kotlinstart
2
3 new *
4 class Test {
5
6
7 new *
8 fun main() {
9
}
```

与Java不同，Kotlin的程序运行入口是main方法，这个main方法是可以不在类里面的，而是在这个Kotlin文件中（Kotlin中的方法可以不依附类存在）。关于main方法大家了解就行，在安卓开发中不会用到。

与Java不同，在Kotlin中，我们使用 `var` 和 `val` 关键字来声明变量，这其实得益于Kotlin独具一格的类型推导机制。

例如在Java中我们可能这样声明变量。

```
int age = 18;
String name = "周博";
```

但是在Kotlin中，我们可以这么写

```
val age : Int = 18
val name : String = "周博"
```

相信大家已经发现了，在Kotlin中，在每行代码结束是没有分号的，这点大家需要注意一下(加了也没事，智能的编译器会提示我们删除。)

上面我们说过，Kotlin具有独具特色的类型推导机制，那么我们：后面的Int和String是不是也可以省略的（作用可以理解为表示数据的类型），当然可以，我们可以尝试一下

```
val age = 18
val name = "周博"
```

虽然我们没有显式声明变量 `age` 的类型，Kotlin的类型推断机制会根据赋值 `30` 自动推断出 `age` 是 `Int` 类型的。这是因为 `30` 是一个整数字面量，而Kotlin会将整数字面量推断为 `Int` 类型。

那么上面我们提到都可以声明变量的 `val` 和 `var` 有什么区别呢？

- `var`：声明一个可变变量，其值可以被改变。
- `val`：声明一个只读变量，其值一旦初始化后不能被改变。

例如

```
val age = 18
val name = "周博"

age=11
```

因为我们使用的 `val` 来声明变量，后面变不能修改了，编译器这时候提醒我们换成 `var`。或许你会觉得 `var` 有点万能，那么我以后的开发过程中是不是就可以全部使用`var`来声明变量，其实不是如此，在安卓开发中我们在声明对象的时候常常使用的 `val`。如果不是一些特殊情况，大家在以后开发中尽量默认使用 `val` 来声明引用型变量。（直接用 `val`，反正不通过后面编译器也会提示我们，怕啥 😊）

相信大家上面还发现了一些不一样的，比如Java声明使用int而Kotlin使用的是Int。这是因为Kotlin完全摒弃了Java的基本数据类型，大家可以看一下下面的表

Java类型	Kotlin类型
byte	Byte
short	Short
int	Int
long	Long
float	Float
double	Double
char	Char
boolean	Boolean

函数

函数是执行特定任务的代码块。在Kotlin中，函数使用 `fun` 关键字声明。函数可以有参数和返回值。

下面我们来看一下Java和Kotlin对函数写法的不同

```
fun greet(name: String, greeting: String = "Hello"): String {  
    return "$greeting, $name!"  
}
```

```
public static String greet(String name, String greeting) {  
    return greeting + ", " + name + "!";  
}
```

与Java不用，Kotlin使用`fun` 来声明一个函数，语法大致如下：

`fun + 方法名(小驼峰命名) + (arg1:String...) :String (返回值)`

接受的参数类型我们使用 `:参数类型` 表示

当然我们也可以没有参数或者没有返回值。

```
fun greet() {  
  
}
```

下面我们在main函数里面调用一下

```
7 fun main() {  
8       
9       
10      
11      
12    println(greet(name: "Android"))  
13      
14      
15      
16      
17    }
```

```
"E:\JetBrains\Android Studio\jbr\bin\java.exe" ...  
Hello, Android!
```

这里补充一下Java中打印我们使用 `System.out.println()`；对应Kotlin中的 `println`

下面来说一个语法糖，

当一个函数中只有一行代码的时候Kotlin允许我们不必编写函数体，可以直接将唯一的一行代码写在函数定义的尾部，中间用等号连接

```
23
24 fun greet(name: String, greeting: String = "Hello"): String = "$greeting, $name!"
25
26
```

也就是这个函数的返回值为等号后面的内容，这个了解就行，一般可用可不用。

逻辑控制语句

条件控制语句

if 语句

if 是最基本的条件控制语句，它可以单独使用，也可以与 else 配合使用来执行条件判断。

举个栗子，我们还是来写个看谁小的方法

```
fun minChoose(a: Int, b: Int): Int {
    if (a < b) {
        return a
    } else {
        return b
    }
}
```

逻辑比较清楚，这是编译器提示我们有一种更为简便的写法

```
new *
fun minChoose(a: Int, b: Int): Int {
    if (a < b) {
        return a
    } else {
        return b
    }
}
```

我们光标移动到黄色的位置，`alt+enter` 看一下编译器给我们改成了什么代码

```
fun minChoose(a: Int, b: Int): Int {
    return if (a < b) {
        a
    } else {
        b
    }
}
```

显然这里Kotlin允许我们return与if的配合使用，大家目前了解一下就好。

其实结合我们之前提到的语法糖我们也可以不写 `return` 语句直接用等号和if配合返回结果，这里不再赘述

```
fun minChoose(a: Int, b: Int): Int = if (a < b) a else b
```

when 语句

`when` 类似于 Java 中的 `switch` 语句，但更加强大。它可以用作表达式或语句，并支持多种类型的条件匹配。

```
val x = 1
when (x) {
    1 -> println("x 等于 1")
    2 -> println("x 等于 2")
    else -> println("x 不是 1 也不是 2")
}
```

大家需要掌握这种用法。

如果我们使用when语句后的每种情况需要执行的逻辑不止一行代码，我们也可以加{}

```
val x = 1
when (x) {
    1 -> {
        fun1()
    }
    2 -> {} println("x 等于 2")
    else -> println("x 不是 1 也不是 2")
}
```

相信大家也发现了，与switch不同when是没有break的,而且，switch的default 在这里是else

都说when很强大，哪里强大呢？

when()括号中可以加任何类型的参数，并且when还支持类型匹配，这在我们的后面开发中会使用到，举个简单的例子。

when is 用法

```
fun printTypeInfo(obj: Any) {
    when (obj) {
        is String -> println("这是一个字符串")
        is Int -> println("这是一个整数")
        is Boolean -> println("这是一个布尔值")
        else -> println("未知类型")
    }
}

// 使用函数
printTypeInfo("Hello") // 输出：这是一个字符串
```

```
printTypeInfo(123) // 输出: 这是一个整数
printTypeInfo(true) // 输出: 这是一个布尔值
```

还有一个例子，这个例子大家能理解就行，理解不了就忽视，在后面的开发中可能会用到。

加入我有两个这时我有个需求我需要根据不同的类调用不同的方法，进行不同逻辑判断，这个怎么实现呢？

这里的holder是 `ViewHolder` 类型的但是 `ViewHolder` 有很多子类，而且这个 `onBindViewHolder()` 疯狂回调往里面传递各种类型的 `ViewHolder`，这里就要用到 `when is` 语句，根据不同的对象类型执行不同的逻辑，怎么样，是不是很酷。

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    when (holder) {
        is RvNewsTopViewHolder -> {}

        is RvNewsViewHolder -> {
            val itemData = getItem(position)
            holder.bind(itemData)
        }
    }
}
```

循环控制语句

for 循环

`for` 循环可以遍历任何提供迭代器（iterator）的对象，如集合或区间。

```
for (i in 1..5) {
    println(i)
}
```

while 和 do-while 循环

`while` 循环在给定条件为真时重复执行代码块。`do-while` 循环至少执行一次代码块，然后再检查条件。

```
var i = 1
while (i <= 5) {
    println(i)
    i++
}

do {
    println(i)
    i++
} while (i <= 5)
```

这些跟Java基本相同。

需要注意的是在for循环的遍历方式。大家只需要记住Kotlin中遍历用的是 `for in` 语句

```
for (i in 1..5) {
    println(i)
}
```

这里就是在【1-5】之间，下面列举一些遍历会用到的数据遍历区间范围

表达方式	描述	示例
<code>..</code>	闭区间运算符，包含起始值和结束值	<code>for (i in 1..5)</code>
<code>until</code>	半开区间运算符，包含起始值但不包含结束值	<code>for (i in 1 until 5)</code>
<code>downTo</code>	降序区间运算符，包含起始值和结束值	<code>for (i in 5 downTo 1)</code>
<code>step</code>	步进值运算符，定义区间内的步进值	<code>for (i in 1..5 step 2)</code>

了解一下就行。

面向对象

类与对象

这么快就到了面向对象这里，不知道大家学习到了现在，大家对面向对象的理解如何了？应该不会像之前那么抽象了吧，虽然面向对象思想一直是个抽象的概念，这里我们再来看一下Kotlin的面向对象。

首先我们先来回顾一下什么是面向对象编程，举个简单的例子，我们学校的每个教室其实就可以理解为面向对象编程中的类，不同的教室有不同的编号，那么每个具体的教室实例（比如“教室2101”、“教室2102”等）就是对象。我们在生成了类的对象后就可以去访问自己类的属性和方法。

```
// 定义一个 Classroom 类
class Classroom(val roomNumber: String, val capacity: Int) {
    fun printDetails() {
        println("教室门牌号: $roomNumber, 容纳人数: $capacity")
    }
}

// 创建 Classroom 对象
val classroom2101 = Classroom("2101", 100)
val classroom2102 = Classroom("2102", 100)
val classroom3102 = Classroom("3102", 120)

// 打印教室详情
classroom2101.printDetails() // 输出: 教室门牌号: 2101, 容纳人数: 100
classroom2102.printDetails() // 输出: 教室门牌号: 2102, 容纳人数: 100
classroom3102.printDetails() // 输出: 教室门牌号: 3102, 容纳人数: 120
```

前面我们提到了，我们一般使用 `val` 来声明变量，并且依赖于Kotlin出色的类型推导机制，我们不需要写 `val classroom2101: Classroom = Classroom("2101", 100)` 显示声明。相信大家也已经发现了，在Kotlin里面创建对象的时候是没有 `new` 关键字的，这一点大家需要注意。

继承和构造函数

面向对象思想中另外一个比较重要的特征便是**继承和构造函数**

在Kotlin中，继承是面向对象编程的一个核心概念，它允许一个类（称为子类）继承另一个类（称为父类）的属性和方法。构造函数是类的特殊函数，用于初始化新创建的对象。

在Kotlin中，所有类默认都是 `final` 的(抽象类除外，毕竟使用 `abstract` 关键字就是为了让其他类继承的)，这意味着它们不能被继承。要允许一个类被继承，你需要使用 `open` 关键字来标记它，这一点是与Java不同的，在Java中一个类本身不做声名便是可以被继承的。

在Kotlin中使用了 `open` 字后表示该类可以被继承，那么我们如何用另外一个类继承它呢？与使用 `extends` 关键字不同在Java中我们不论是继承还是实现接口统一用冒号 `:`，也就是说Kotlin中是没有 `extends` 关键字的

```
open class Animal {
    fun eat() {
    }
}

class Dog : Animal() {
    fun bark() {
    }
}
```

相信大家对这种写法有点疑惑，这里为什么要使用 `()`，使用 `()` 与类的构造函数有关下面我们重点讲解一下。(关于类的继承的覆写我们后面再讲)

相信大家对构造函数的概念(可以在类初始化的时候对一些变量进行赋值)都已经了解了。在Kotlin中，构造函数分为**主构造函数**和**次构造函数**。下面要讲的概念可以有点抽象，大家可以尝试着去理解一下，对安卓开发理解别人代码这一方面还是有一定的帮助的。

Kotlin中的类可以有一个主构造函数和一个或多个**次构造函数**。主构造函数是类头的一部分，它跟在类名后面。

主构造函数在类后面小括号里面进行变量的声明

```
class Person(val name: String, var age: Int) {
    // 主构造函数
}
```

上面的写法等效于在Java中

```
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
    }
}
```

```

        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

```

可以看到我们并没有在类里面去声明这些变量我们只是通过 `(val name: String, var age: Int)` 代替了Java的那种写法，并且也是可以不用写set和get方法的。(这里的age使用var声明是让大家更好理解Kotlin可以不用像Java那样显示去写get和set方法，应为val对应Java里面的final，是没有set方法的。var声明的变量可以有set和get方法，这些默认Kotlin在反编译成Java代码之后之后会帮我写好，我们不用去显示的去添加，当然这不并不是说Kotlin没有get和set方法。)

我们来在main方法里面调用一下

```

val person= Person("卷娘",18)
println(person.name)
println(person.age)

```

卷娘

18

这便是Kotlin的主构造函数。

下面我们来看一下Kotlin的次构造函数。

次构造函数使用 `constructor` 关键字定义，并且必须直接或间接地调用主构造函数。

```

class Person(val name: String, var age: Int) {
    constructor(name: String) : this(name, 0) {
        // 次构造函数
    }
}

```

也就是说，Kotlin的次构造函数一定会调用主构造函数

上面的代码边说明了这一点

加入我在初始化Person实例的时候不想给age赋值，只想给name:String赋值的话，我们便可以使用constructor关键字。让后通过: `this(name, 0)` 调用主构造函数。这个时候我们使用次构造函数初始化的对象，name便是我们传入的值，而age由于次构造函数一定会调用主构造函数，根据代码，这里我们的age默认设置为了0，，在实际的开发中，假如我们不想给它赋值，我们可以在主构造函数中设置一个

变量，次构造函数设置两个变量，这里只是举个例子让大家更好的理解Kotlin的主构造函数和次构造函数。

当然了，次构造函数可以有多个：

```
//次构造函数
constructor(name: String) : this(name, 0)

//次构造函数
constructor() : this("无", 0)

//次构造函数
constructor(name: String, age: Int, weight: Int) : this(name, age)
```

分别对应

```
val person = Person( name: "卷娘", age: 18)
val person2 = Person()
val person3 = Person( name: "卷娘", age: 18, weight: 60)
```

上面我们提到次构造函数一定会调用主构造函数，并且使用**this**关键词如果没有一个类没有主构造函数怎么办呢？

在Kotlin中，如果一个类没有显式定义主构造函数，那么它可以有一个或多个次构造函数。每个次构造函数都需要通过 **super** 关键字来调用父类的构造函数（如果有的话），因为Kotlin中所有类都有一个超类 **Any**，它是所有类的默认父类。

如果类没有父类（除了 **Any**），次构造函数就不需要使用 **this** 或 **super** 来调用另一个构造函数。以下是一个没有主构造函数的类，只有次构造函数的例子：

```
open class Person(val name: String, val age: Int) {
    // 主构造函数

    //次构造函数
    constructor(name: String) : this(name, 0)

    //次构造函数
    constructor() : this("无", 0)

    //次构造函数
    constructor(name: String, age: Int, weight: Int) : this(name, age)
}

class Person1 : Person {

    //次构造函数
    constructor(name: String, age: Int) : super(name, age)
```



```
}
```

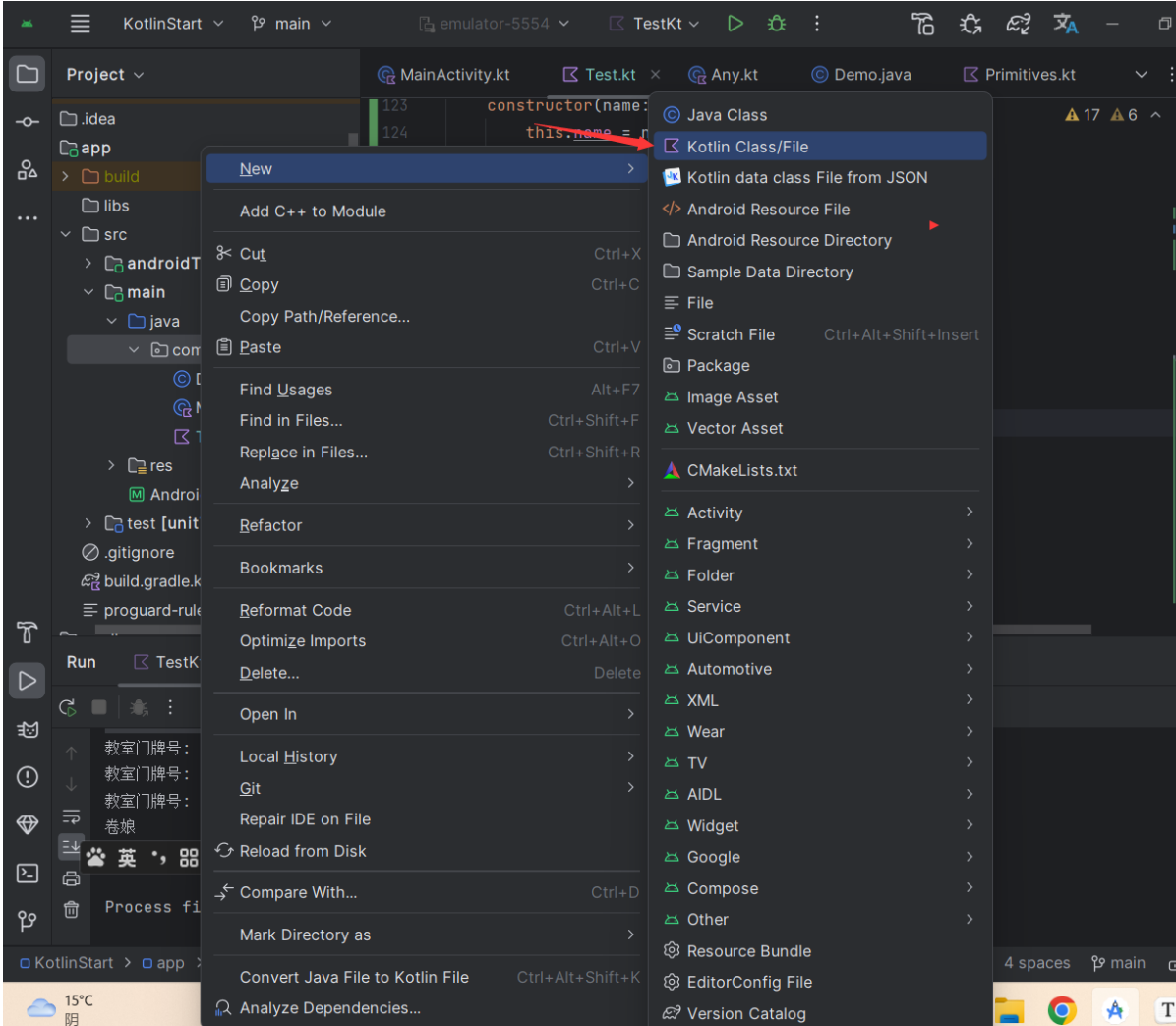
如果类没有父类（除了 Any），次构造函数就不需要使用 this 或 super 来调用另一个构造函数。以下是一个没有主构造函数的类，只有次构造函数的例子：（实际情况一般不会发生，我们不做讨论，下面的例子看不懂就不看，实际开发中可一般会遇到上面的正常情况，下面的目前只需要知道/当一个类中没有主构造函数的时候，()是可以省略的）

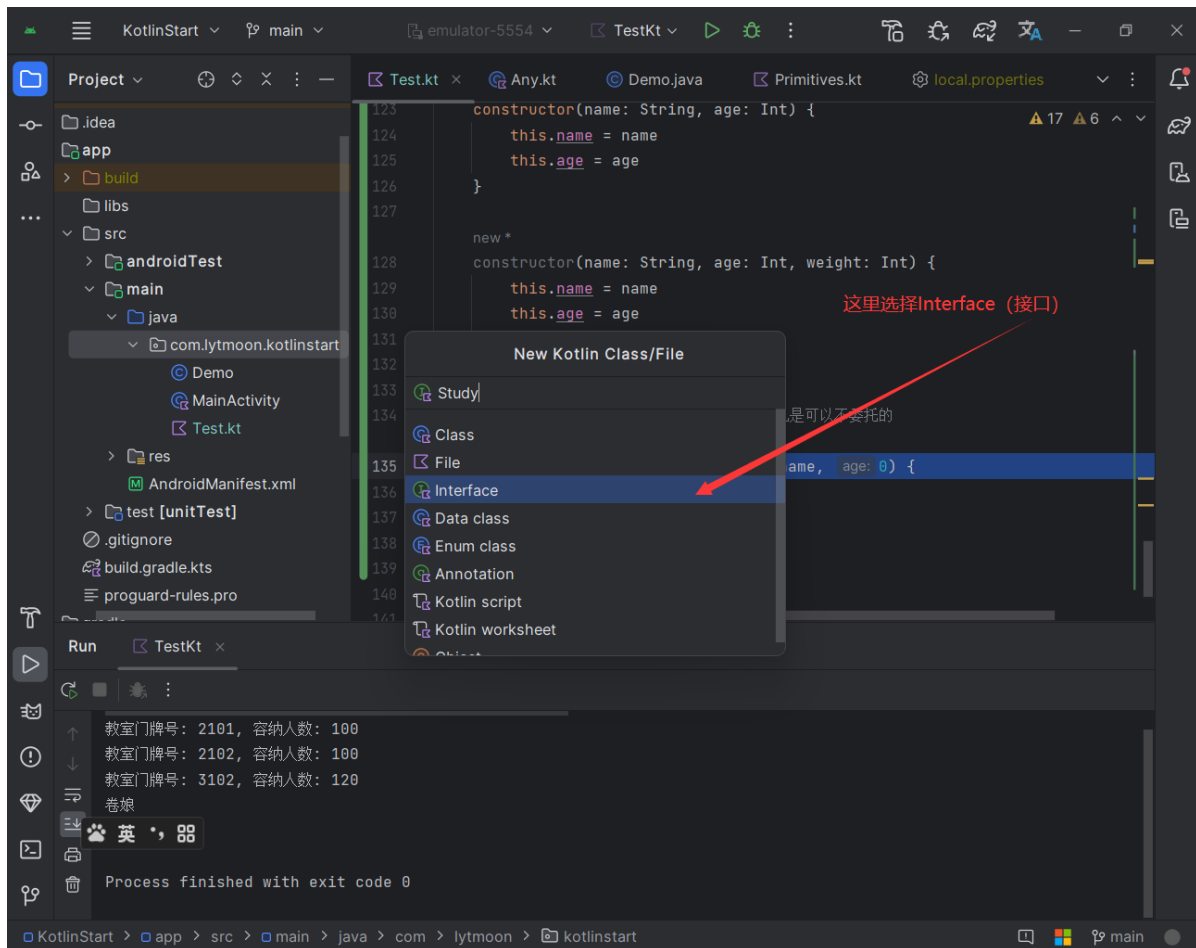
```
class Person2 {  
    var name: String  
    var age: Int  
    var weight: Int = 0  
  
    constructor(name: String, age: Int) {  
        this.name = name  
        this.age = age  
    }  
  
    constructor(name: String, age: Int, weight: Int) {  
        this.name = name  
        this.age = age  
        this.weight=weight  
    }  
  
    //这里的this会调用适合的次构造函数,当然也是可以不委托的  
    constructor(name: String) : this(name, 0) {  
        // 可以在这里添加额外的初始化代码  
    }  
}
```

接口

接口的学习比较简单但是很重要，并且Kotlin中的接口是和Java中几乎完全一致的。

接口的存在价值之一便是服务于多态编程，我们可以在接口中定义一些的抽象行为(具体表现为只有方法名，没有具体实现逻辑。)，而具体的实现逻辑可以交给具体的实现接口的类去完成。下面我们来学习一下如何使用接口。

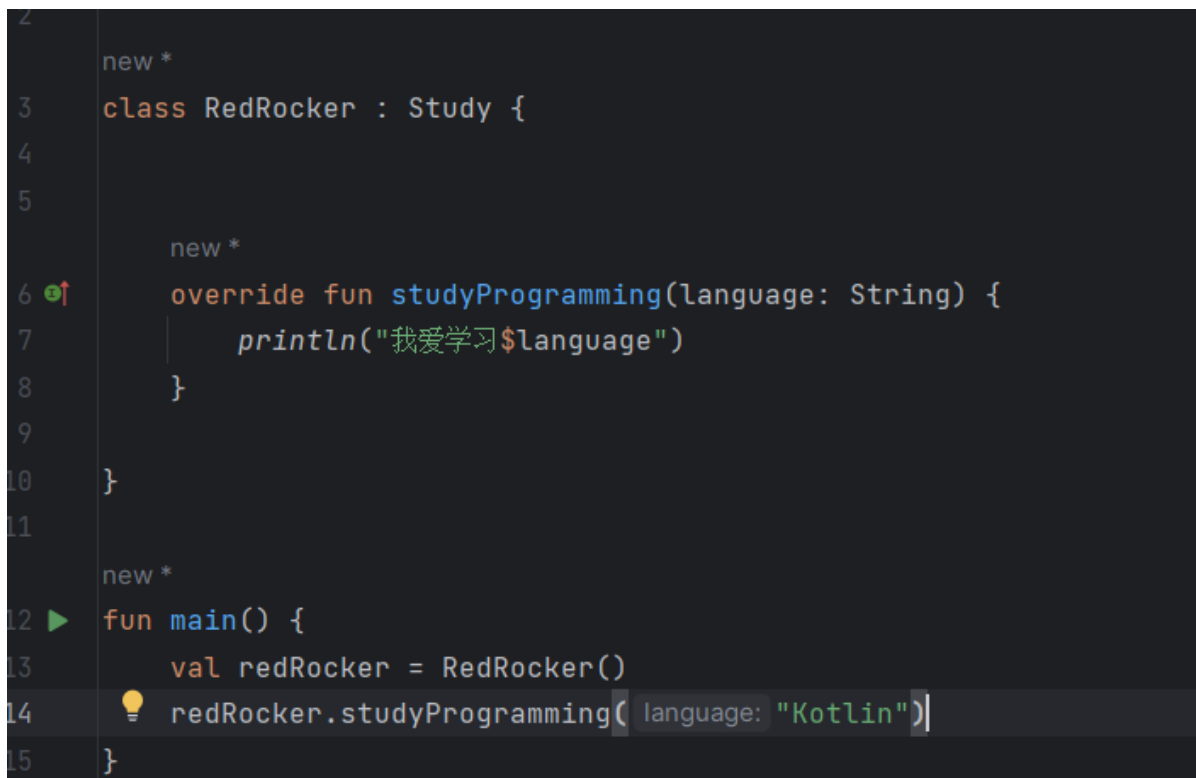
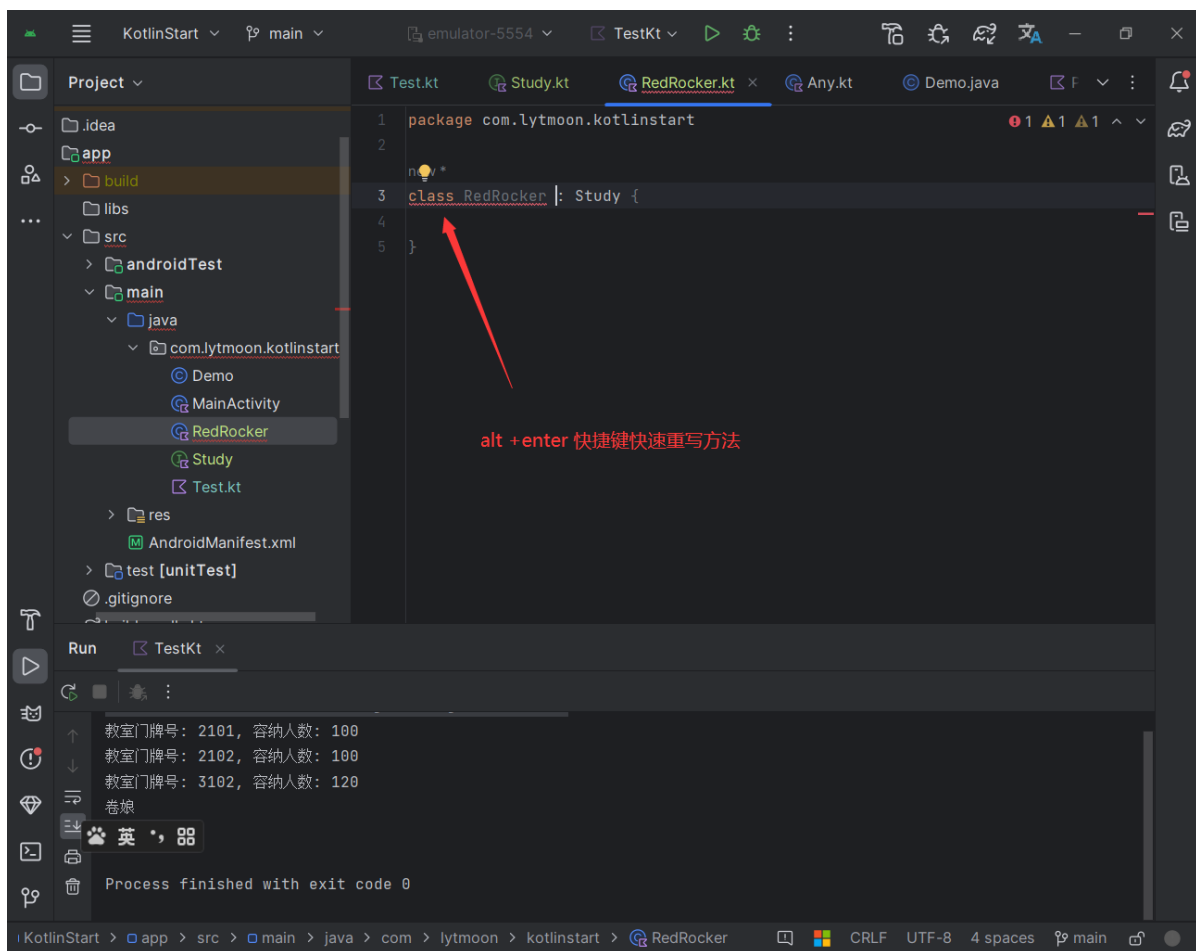




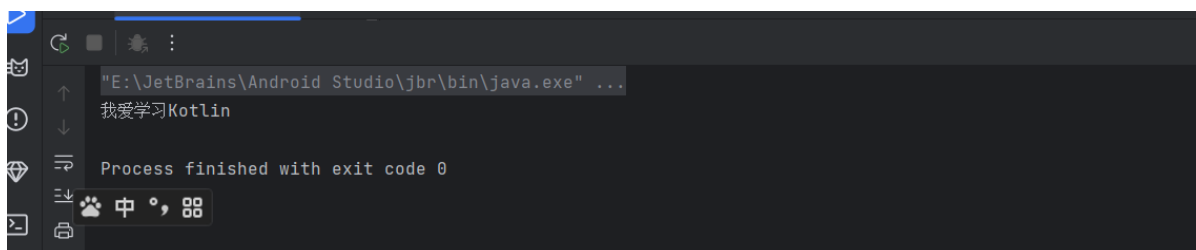
定义一组方法

```
3  
4 new *  
5 interface Study {  
6     new *  
7     fun studyProgramming(language: String)  
8 }
```

让后我们可以新建一个类来实现这个接口，上面我们提到过，继承或者实现都使用的是：



最后打印出来的便是



上面提到接口可以服务于多态编程，那么什么是多态编程呢？（下面内容可以试着理解一下，与Java其实一样。）

多态是面向对象编程中的一个核心概念，它允许不同的对象对同一消息做出响应。具体来说，多态性意味着同一个接口可以被不同的对象以不同的方式实现。

我们对上面的代码简单加工一下

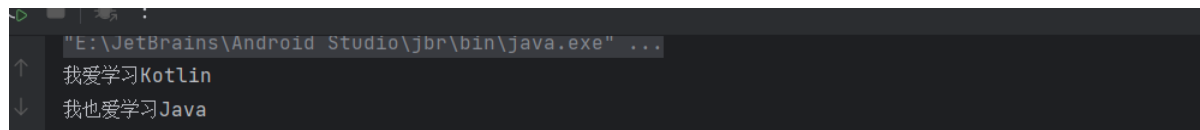
```
interface Study {
    fun studyProgramming()
}

class RedRocker1(val language: String) : Study {
    override fun studyProgramming() {
        println("我爱学习$language")
    }
}

class RedRocker2(val language: String) : Study {
    override fun studyProgramming() {
        println("我也爱学习$language")
    }
}

fun main() {
    val study1: Study = RedRocker1("Kotlin")
    val study2: Study = RedRocker2("Java")
    study1.studyProgramming() // 输出：我爱学习Kotlin
    study2.studyProgramming() // 输出：我也爱学习Java
}
```

最后结果



```
"E:\JetBrains\Android Studio\jbr\bin\java.exe" ...
↑ 我爱学习Kotlin
↓ 我也爱学习Java
```

这里应该不难理解，其实多态编程写法有很多，我们只需要理解这种思想就可以了。

相信大家学过Java应该都知道我们在定义变量或者方法的时候前面会使用一些修饰符来表示该方法或者变量的可见性。当然在Kotlin中也有这些东西，不过有些是与Java有所区别

Java修饰符	Kotlin修饰符	描述
public	public	没有限制，任何地方都可以访问
protected	protected	当前类及其子类可以访问
default	(无直接对应)	同一包内可以访问
private	private	当前类内可以访问

另外Kotlin还多了一个

Kotlin修饰符	描述
internal	同一模块内可以访问

Kotlin中无default关键字，这点大家需要注意。

一般情况下，如果我们在一个类里面编写的方法或者变量我们不希望暴露在外或者我们目前不清楚是否需要被其他类调用或者改变，都要加上private关键字，希望大家养成良好的编程习惯。

数据类与单例类

数据类

数据类是我们在后面开发安卓的时候经常会用的一种类，常见的使用场景是作为网络请求的数据接收类。

Kotlin的数据类是一种简化版的类，专门用来存储数据。当你在类前面加上 data 关键字时，Kotlin会自动为你生成 equals()、hashCode() 和 toString() 等方法。这些方法对于比较对象、在集合中使用对象以及打印对象信息非常有用。例如在Java中我们会这样定义一个数据类：

```
public class User {
    private final String name;
    private final int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        User user = (User) o;

        if (age != user.age) return false;
        return name != null ? name.equals(user.name) : user.name == null;
    }

    @Override
    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;
        result = 31 * result + age;
        return result;
    }
}
```

```

    }

    @Override
    public String toString() {
        return "User{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
    }
}

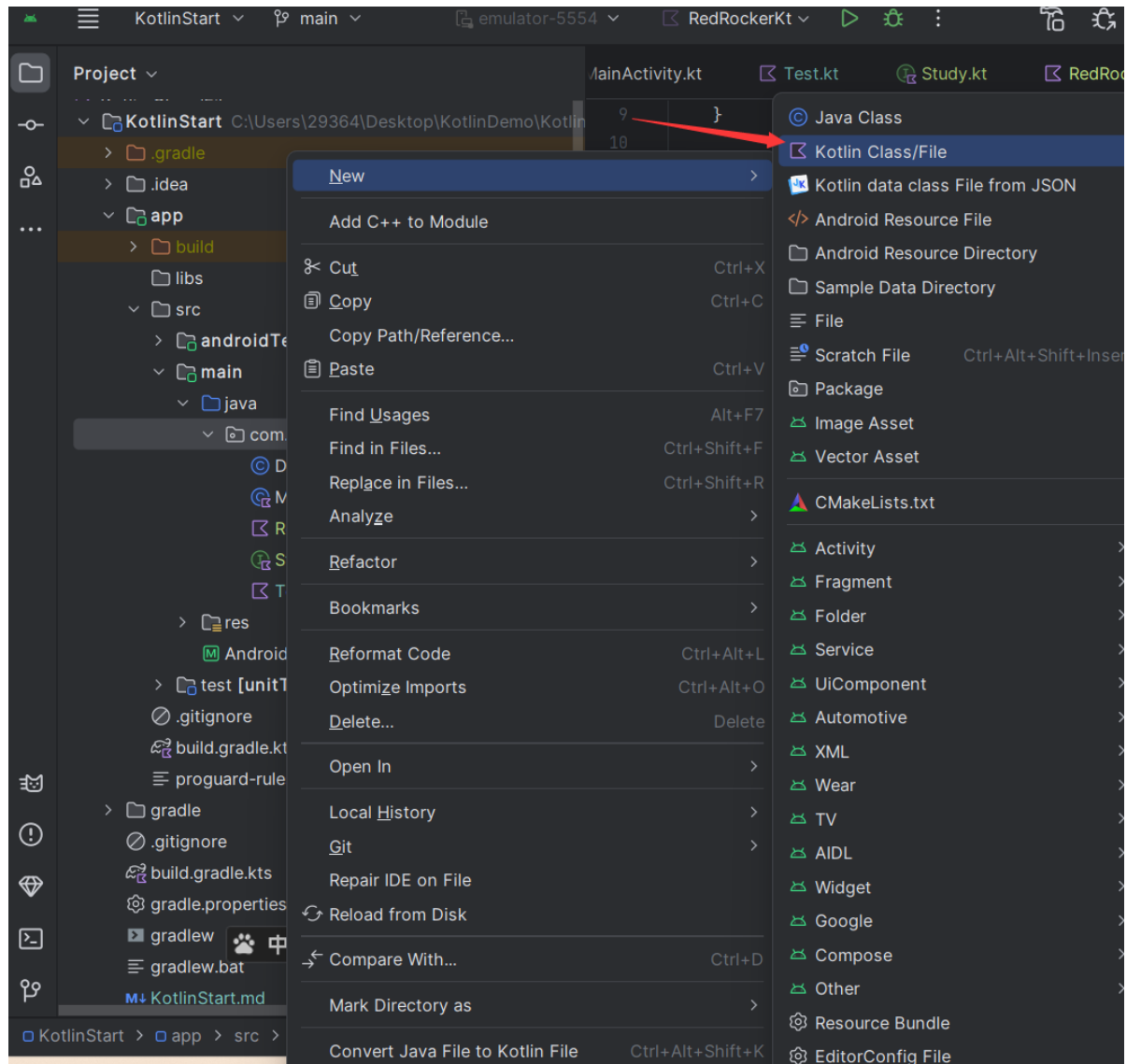
```

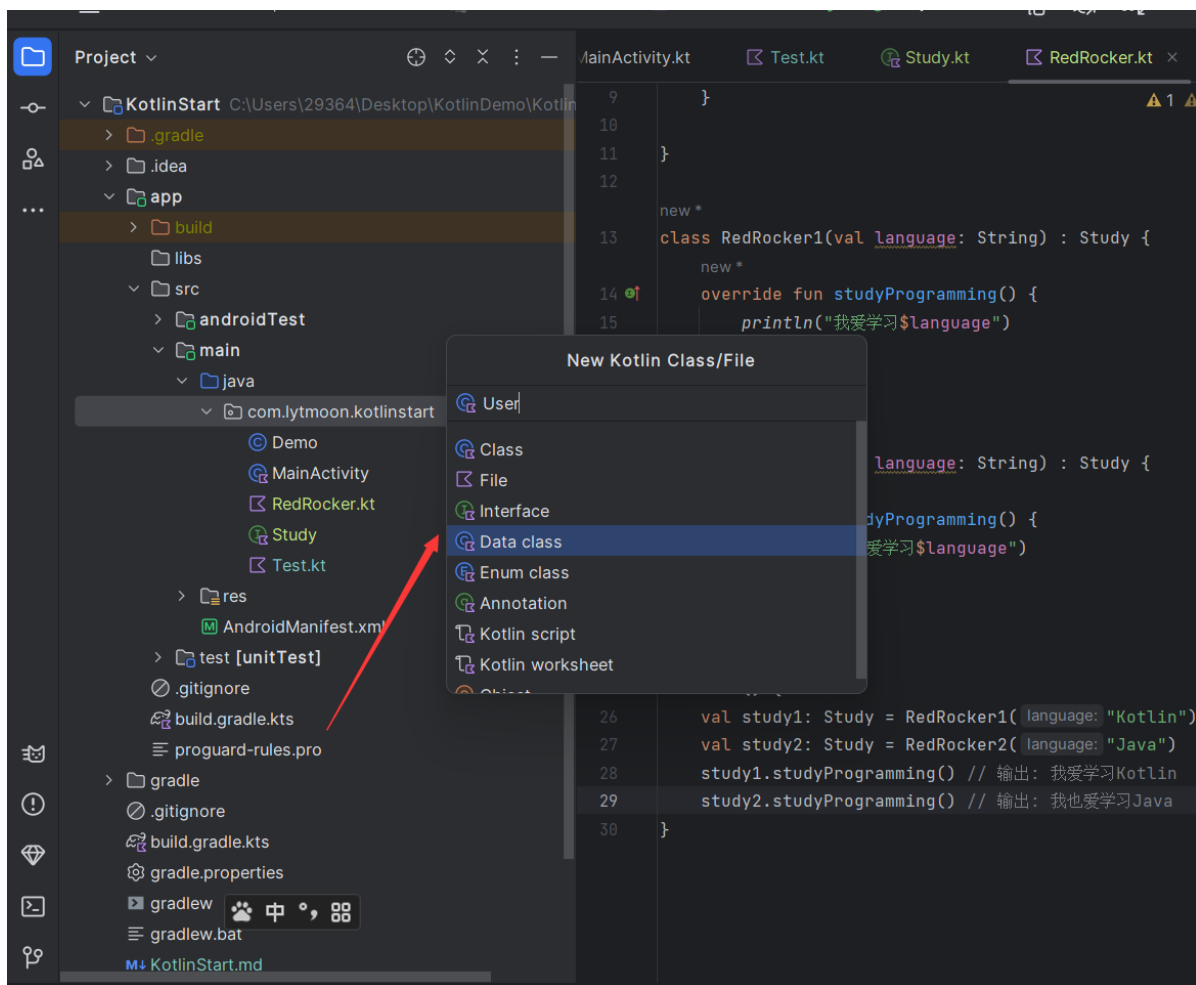
这里面我们有很多方法都需要自己去重写

但是在Kotlin里面，我们只用到了一个关键字**data**。

```
data class User(val name: String, val age: Int)
```

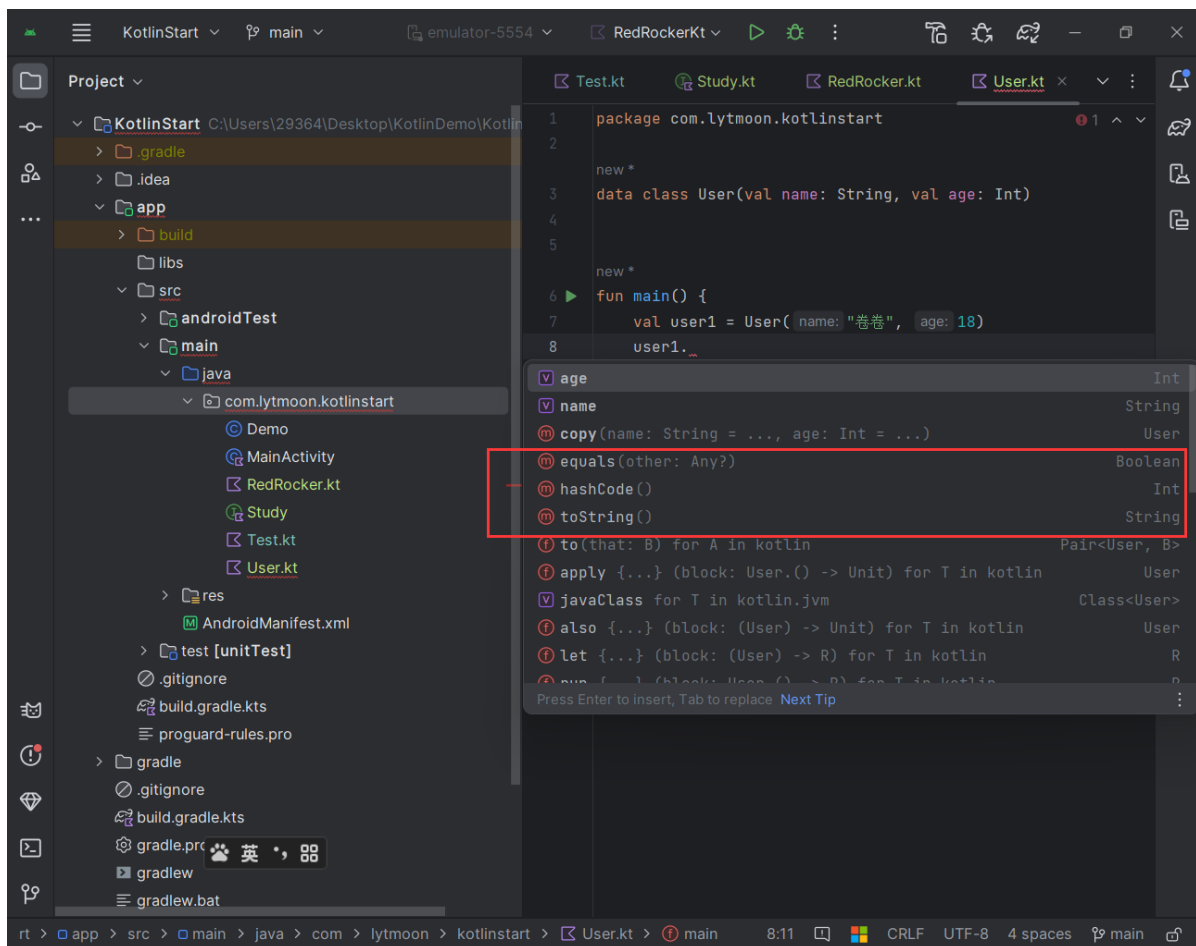
在反编译的时候会自动变成上面的Java代码（帮我们为主构造函数里面的参数生成一些必要的方法）。新建数据类的过程如下。



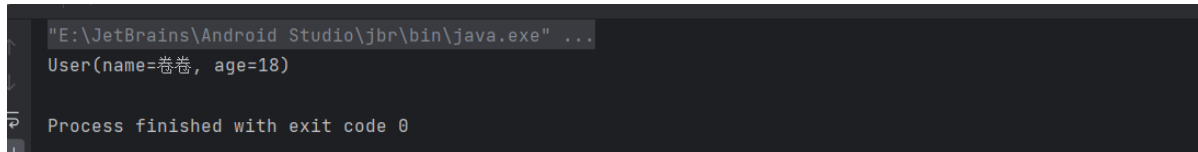


我们在main方法里面测试一下。

这里我们可以看到Kotlin已经给我们自动生成了这些方法。




```
fun main() {  
    val user1 = User("卷卷", 18)  
    println(user1.toString())  
}
```



```
"E:\JetBrains\Android Studio\jbr\bin\java.exe" ...  
User(name=卷卷, age=18)  
  
Process finished with exit code 0
```

了解完数据类之后我们再来了解另外一个Kotlin中特有的功能--单例类

单例类

相比大家都或多或少听说过单例模式，单例模式确保了全局中只有一个类实例存在。我们经常把一些工具类的方法放在单例类里面(也就是把工具类变成单例类)

我们先来看一种Java实现单例类方法

```
public class Singleton {  
    // 私有静态变量，保存类的唯一实例  
    private static Singleton instance;  
  
    // 私有构造函数，防止外部直接创建对象  
    private Singleton() {}  
  
    // 公共静态方法，返回类的唯一实例  
    public static Singleton getInstance() {  
        if (instance == null) {  
            // 如果实例为空，表示还没创建过实例  
            instance = new Singleton();  
        }  
        // 返回已创建的实例  
        return instance;  
    }  
}
```

相信大家应该都能看懂其中的逻辑:在这个例子中，`Singleton` 类有一个私有静态变量 `instance`，它用来保存类的实例。`Singleton` 的构造函数是私有的，所以不能从类外部创建 `Singleton` 对象。`getInstance()` 方法确保只创建一个 `Singleton` 实例。如果 `instance` 为空，它会创建一个新的 `Singleton` 实例，否则它会返回已经创建的实例。

这种方式被称为懒汉式单例，因为它在第一次需要实例时才创建实例。这种方法在单线程环境下工作得很好，但在多线程环境下可能会出现一些问题。如果多个线程同时访问 `getInstance()` 方法，可能会创建多个实例。为了在多线程环境中安全使用，需要添加同步机制，例如使用 `synchronized` 关键字。

这里我们不对这些写法做过多了解，有兴趣的可以看一下[这里](#)

[Java单例模式](#)

好，那么回归Kotlin，在Kotlin中我们实现单例模式只需要用使用**Object**关键字

例如

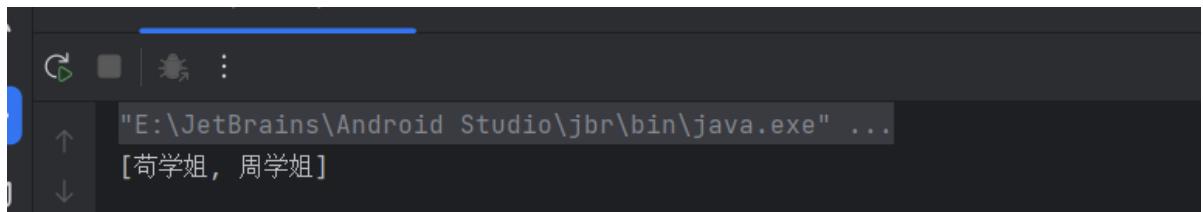
```
object Repository {
    private val data = mutableListOf<String>()

    fun addData(item: String) {
        data.add(item)
    }
    fun getData():String{
        return data.toString()
    }
}
```

在这个例子中，`Repository` 是一个单例类，它有一个可变列表 `data` 和两个方法 `addData`、`getData()`。由于 `Repository` 是一个单例，所以无论你在代码中何处访问它，它都会引用同一个实例。

关于访问这个方法的话，我们可以直接通过大写的类名来访问。

```
fun main() {
    Repository.addData("苟学姐")
    Repository.addData("周学姐")
    println(Repository.getData())
}
```



关于面向对象目前大家可以了解到这里。

Lambda编程（不仅仅是一种语法糖！）

Kotlin 的设计初衷之一就是要解决 Java 语言的一些痛点。例如，Kotlin 提供了更简洁的语法来处理 null 安全、集合操作和函数式编程等。Lambda 表达式在 Java 中也存在，但是直到 Java 8 才被引入，而 Kotlin 从一开始就内置了 Lambda 表达式的支持。

我们今天只学习一些 Lambda 编程的基础知识，稍微了解一些常用的高阶函数（只讨论其用法）。

Kotlin Lambda 表达式是 Kotlin 语言中的一个强大特性，它允许你以匿名函数的形式编写代码块，并将其作为参数传递给函数或作为结果返回。Lambda 表达式使得代码更加简洁和灵活，特别是在处理集合、线程或任何需要自定义行为的场景中。

在了解 Lambda 表达式之前，我们先来看一下这种情况：

假如现在有一个需求，我需要创建一个包含 2022 级所有学长的集合，我们会怎么去创建呢？

你可能会这么写

```
val list= ArrayList<String>()
list.add("苟云东学姐")
list.add("苟云东学姐")
.....
```

但是在Kotlin中，有一个特殊提醒的内置的 `listOf()` 函数来帮助我们快速初始化创建集合对象，这个集合 api 在我们以后的开发中也会经常用到。使用这个函数我们可以改成下面的写法

```
val list = listOf("苟云东学姐", "苟云东学姐"...)
```

怎么样，是不是很方便。不过使用 `listOf()` 函数创建的是一个不可变的集合，也就是不能对数据及逆行增删改操作，只能读取。

那么有没有创建一个可变集合的 api 呢？当然是有的

```
val list = mutableListOf("苟云东学姐", "苟云东学姐"...)
```

使用 `mutableListOf()` 创建的便是一个可变的集合

```
val list = mutableListOf("苟学姐", "周学姐", ".....")
list.add("胡学姐")
println(list)
```



这是List集合的一些用法，当然还有Set集合的一些用法，这些用法之间几乎一模一样，只是函数的名字不同罢了。

```
val set = setOf<String>("苟学姐", "周学姐", "胡学姐", ".....")
for (it in set){
    println(it)
}
```



当然我们也可以使用map

```
// 创建一个Map，其中包含三个键值对
val capitals = mapOf("France" to "Paris", "Japan" to "Tokyo", "Brazil" to "Brasilia")

// 打印整个Map
println(capitals)

// 访问特定的键
println("The capital of Japan is: ${capitals["Japan"]}")
```

了解玩这些后，我们来正式看一下Kotlin Lambda

Kotlin Lambda 表达式的语法

{参数1: 参数类型, 参数2:参数类型-> 函数体}

如果Lambda表达式有参数，参数类型可以省略，因为Kotlin有很强的类型推断能力。如果函数体只有一行，那么它的结果就是这行代码的结果。如果函数体有多行，那么最后一行的结果就是Lambda表达式的结果。

如果没有参数的话那么直接使用{}编写函数体

也就是说在->之前是参数列表，

lambda表达式经常跟函数式编程联系起来，什么是函数式变成呢?简单一点来理解就是把一个函数当成一个参数来用，而Lambda表达式就可以直接作为函数的参数传递。这个概念可能有点抽象，我们先来看一下lambda的常见用法。

用法之一：函数式编程

比方说我们来看一个集合的api

```
listOf(1, 2, 3).forEach { item -> println(item) }
```

首先我们来看一下 `listOf(1, 2, 3)`，这里返回了一个不可变的集合对象。让后调用这个集合的 `forEach` 方法来进行遍历每一个元素，我们先来看一下这个函数是怎么实现遍历的。点击这个 `forEach` 我们看到了下面的方法。

```
/**
 * Performs the given [action] on each element.
 */
@kotlin.internal.HidesMembers
public inline fun <T> Iterable<T>.forEach(action: (T) -> Unit): Unit {
    for (element in this) action(element)
}
```

这里大家可能理解的有点吃力，这里使用了内联函数关键词 `inline`，我们先不管这个我们看到这个方法接受了一个叫做action的参数，但是这个名字叫做action的参数的函数类型不知道大家有没有感到疑惑。

action: (T) -> Unit

这里其实便是函数式编程的体现，这里的action其实就是一个函数类型的参数，说的直白点，就是我们可以把这个函数当成一个参数来用，我们知道一个函数的最终目的也是执行这个函数里面的逻辑，只要这个参数能执行函数的方法就行呗。

`action: (T) -> Unit` 是一个函数类型的声明，它表示一个接受单个参数 `T` 并且不返回任何有意义值（即返回 `Unit`）的函数。在Kotlin中，所有函数都会有返回值。如果你没有显式指定返回类型，那么函数会默认返回 `Unit` 类型，这表示该函数不返回任何有意义的值。`Unit` 类型在Kotlin中类似于Java中的 `void` 类型，但它实际上是一个对象，代表了“无返回值”的概念。

那么这个action既然是一个函数类型的变量，那么就可以把它当成一个函数来用，也就是 `action()`

可以看到在for each里面 使用了for in来遍历这个集合，最后执行 `action()`，并把每个元素传递其中。而这样的一个函数类型的变量的具体逻辑是什么呢？你可以这么理解 Kotlin中的函数类型的变量默认支持Lambda表达式也就是在上面的代码中

```
`aciton参数`={ item -> println(item) }
```

```
forEach { item -> println(item) }
```

我们知道这个action接受了一个参数，在lambda中便是 item 由于不返回有意义的值，我们的函数体是自己写的 `println(item)`

这样我们回过头来看，我们定义的Lambda表达式里面的内容就可以被当成一个函数类型的变量，最后调用了action(element)执行了函数中的具体的逻辑（也就是参数也能当成一个函数来用）

我们也可以举个简单的例子：

声明函数式类型的变量的格式 **（接受参数类型） -> 返回值类型**

```
val sum: (Int, Int) -> Int = { x, y -> x + y }  
println(sum(1, 2)) // 输出结果为3
```

这里的知识有点抽象，可能大家在现在可能不能很好的理解，不过在后面的安卓开发中，随着我们使用Lambda次数增多，我们也会慢慢理解这种写法和思想。

用法之二：匿名内部类

下面我们来看一下Lambda与安卓开发的具体应用。

回想我们以往的开发经历，如果我们要在安卓主线程中新开一个线程的话，我们会怎么写

简单一点，我们直接new 一个Thread对象，

```
new Thread(new Runnable(){  
    @override  
    public void run(){  
  
    }  
}).start()
```

这里使用匿名类的写法，我们创建了一个Runnable接口的匿名类实例，并将它传给了Thread类的构造方法。最后调用Thread类的**start()**方法来开启这个线程，并执行我们重写在run方法中的逻辑。

Kotlin中的匿名类的写法与Java不用，Kotlin完全舍弃了new关键字，因此创建匿名类实例的时候就不能再使用new关键字了，而是改成了object关键字。

我们来看一下Kotlin中的写法

```
Thread(object :Runnable{  
    override fun run() {  
        //执行自己的逻辑  
    }  
}).start()
```

Thread要求接受一个Runnable的对象，我们使用object关键字通过匿名内部类的方式传入，顺便重写里面的run方法。

这么一对比的话可能大家看不到什么差别。

但是你要知道强大的lambda表达式！

如果我们在Kotlin代码中调用了一个Java方法，并且该方法接收一个Java单抽象方法接口参数。就可以使用函数式API，即Lambda表达式，来简化代码。

单抽象方法（Single Abstract Method，简称SAM）是指在Java和类似语言中，只有一个抽象方法的接口。这种接口通常用于表示一个特定的操作或行为，并且可以通过匿名内部类或Lambda表达式来实现。

在Java中，SAM接口是只有一个抽象方法的接口，常用于创建匿名内部类的实例。而在Kotlin中，你可以直接传递一个Lambda表达式来代替整个匿名内部类，这使得代码更加简洁和易读。

把上面的简单总结一下就是需要有 **Java方法 一个接口 一个方法**

我们来看一个这个Runnable

```
public interface Runnable {  
    void run();  
}
```

显然这个接口自定义了一组方法。符合我们使用使用函数式API的条件。

也就是我们可以改成

```
Thread({  
    println("")  
}).start()
```

显然这里我们直接用lambda表达式代替了整个匿名内部类。这个Lambda表达式你可以理解为一个变成了一个实现了Runnable接口的对象

难道就知道这么简洁了吗？当然不是，我们需要知道 当Lambda表达式是方法的最后一个参数的时候，可以把Lambda表达式移动到()的外面。如果Lambda表达式还是唯一的一个参数，那么这个()也可以不要。那么上面的代码可以修改成下面的代码。

```
Thread(){  
    println("")  
}.start()  
//最终简化结果  
Thread{  
    println("")  
}.start()
```

这是一种常见的应用场景，当然还有一种我们后面会经常使用到的场景--点击事件。

还记得我们Java中点击事件吗？

```
button.setOnClickListener(new View.OnClickListener() {  
    @Override  
    public void onClick(View v) {  
        // 处理点击事件  
    }  
});
```

我们来看一下这个点击方法 `setOnClickListener`

```
public void setOnClickListener(@Nullable OnClickListener l) {
    if (!isClickable()) {
        setClickable(true);
    }
    getListenerInfo().mOnClickListener = l;
}
```

显然这里是接受一个实现了OnClickListener接口的对象。

```
public interface OnClickListener {
    /**
     * Called when a view has been clicked.
     *
     * @param v The view that was clicked.
     */
    void onClick(View v);
}
```

显然这也是一个SAM方法。按照上面操作我们可以进行简化。

```
button.setOnClickListener { v ->
    // 处理点击事件
}
```

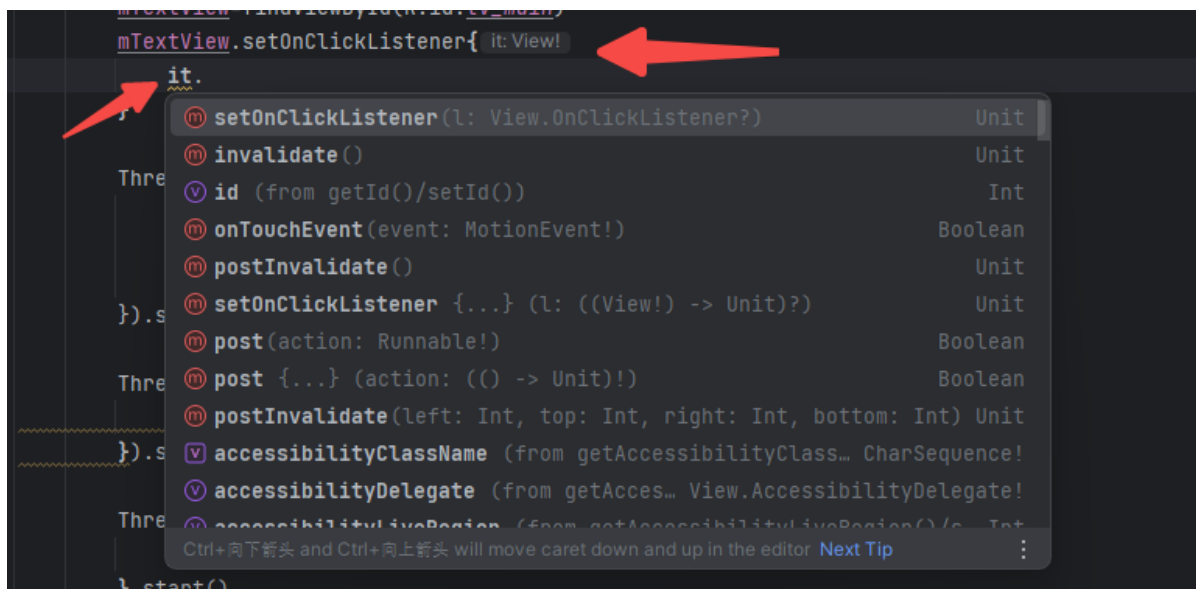
关于如何简化到这一步的大家可以按照上面的一种示例操作来

这里的 `{ v -> ... }` 就是一个lambda表达式。它接收一个参数 `v`，这是被点击的视图对象。在箭头（`->`）后面，你可以添加执行的代码。如果lambda表达式的参数未使用，甚至可以省略参数名和箭头，如下所示：

```
button.setOnClickListener {
    // 处理点击事件
}
```

那么如果这个时候我又想使用这个参数该怎么办呢？

在lambda表达式中如果参数列表只有一个参数，我们可以把它省略掉。在后面需要使用的时候直接用 `it`（不需要声明）



关于Kotlin Lambda的学习远不止如此，作为初学者的我们，虽然可以把它当成一种语法糖，但是Kotlin对Lambda中还有专门的高级函数来实现更加强大的自定义函数式API功能。而不是仅仅借助Java的SAM规范。

空安全

Kotlin十分重视对空指针的检查，甚至把空指针异的检查提前到了编译时期。而空安全特性也是Kotlin这么语言的一大特性。

下面我们来了解一些Kotlin空安全中的几个关键概念：

- **可空类型和非空类型:**

- 在Kotlin中，所有类型默认都是非空的。如果你想允许一个变量为null，你需要在类型后面加上？

来表示它是可空的。例如：

```
var nonNullable: String = "Hello" // 非空类型
var nullable: String? = null      // 可空类型
```

- **安全调用操作符 (?.) :**

- 使用安全调用操作符可以避免在变量为null时执行操作，从而防止抛出异常。如果变量不是null，就执行后面的操作；如果是null，则整个表达式返回null

```
val length = nullable?.length
```

- **Elvis操作符 (?:) :**

- Elvis操作符允许你在表达式为null时提供一个默认值。如果左侧的表达式不是null，就返回它；否则返回右侧的值。

例如：

```
val length = nullable?.length ?: 0
```


- **非空断言操作符 (!!) :**

- 当你确定一个变量绝对不会为null时, 可以使用 `!!` 操作符。如果变量为null, 会抛出 `NullPointerException`。这个操作符应谨慎使用, 因为它会去除空安全检查。例如:

```
val length = nullable!!.length
```

- **安全类型转换 (as?) :**

- 安全类型转换操作符尝试将一个实例转换为指定的类型, 如果实例不是目标类型, 则返回null。这比传统的类型转换更安全!(强制类型转化常常伴随着风险,在Java中我们时常是用try catch代码块进行捕获和处理, 现在只需要在as后面加个?)因为它不会抛出 `ClassCastException`。例如:

```
val value: Int? = nullable as? Int
```

- **let函数:**

- let函数与安全调用操作符结合使用时, 可以在变量不为 `null` 的情况下执行代码块。例如:

```
nullable?.let {  
    // 如果nullable不为null, 则执行这里的代码  
}
```

通过这些机制, Kotlin帮助开发者在编译时期就捕获可能的空引用错误, 从而减少运行时异常的发生。这些特性使得Kotlin代码更加安全和清晰。

Kotlin中的小魔术

在《第一行代码》中还有一些"Kotlin中的小魔术"--字符串内嵌表达式、函数的参数默认值。下面简单介绍一下

字符串内嵌表达式是Kotlin语言中的一个非常有用的特性, 它允许你在字符串中直接插入变量或表达式。这种方式不仅使代码更简洁, 而且提高了可读性。在Kotlin中, 你可以使用 `$` 符号来引用变量, 或者使用 `${}` 来引用更复杂的表达式。以下是一些例子:

```
val name = "Alice"  
// 使用变量  
println("Hello, $name!") // 输出: Hello, Alice!  
  
// 使用表达式  
val count = 5  
println("I have ${count + 1} apples.") // 输出: I have 6 apples.  
  
// 使用对象的属性  
val person = Person("Bob", 30)  
println("Name: ${person.name}, Age: ${person.age}") // 输出: Name: Bob, Age: 30
```

你可以理解为在java中我们可以通过`$`引用单个变量, 如果变量比较长或者需要进行一些操作, 使用`${}`, 这在我们以后开发中会经常用到, 如进行一些日志的输出。

函数的参数默认值也就是在传参的时候如果我们不指定的话可以使用原来定义好的默认值

在Kotlin中，函数的参数可以有默认值，这是一种非常方便的特性，它可以减少函数重载的需要。当调用一个函数时，如果某个参数被省略，那么将使用该参数的默认值。这样，你就不必为每个可能的参数组合编写一个单独的函数。

例如，假设我们有一个打印消息的函数，我们可以为其参数提供默认值：

```
fun printMessage(message: String, prefix: String = "Info", suffix: String = "End") {  
    println("[$prefix] $message [$suffix]")  
}
```

在这个例子中，`prefix` 和 `suffix` 参数有默认值 "Info" 和 "End"。当调用 `printMessage` 时，你可以只传递 `message` 参数：

```
printMessage("Hello, Kotlin!") // 输出: [Info] Hello, Kotlin! [End]
```

如果你想覆盖默认值，可以像这样传递所有参数：

```
printMessage("Hello, Kotlin!", "Start", "Finish") // 输出: [Start] Hello, Kotlin!  
[Finish]
```

或者，如果你只想覆盖其中一个默认值，可以使用命名参数：

```
printMessage("Hello, Kotlin!", suffix = "Done") // 输出: [Info] Hello, Kotlin!  
[Done]
```