

var、val

var 可读、可写变量

val 只读变量；不是常量！

java 中使用 final 修饰的静态变量的字符串，是个编译器常量（编译器编译的时候，已经确定并且不可改变的）

```
1 private static final String LALALA = "lalala";
```

kotlin 中的表示方式

```
1 companion object {
2     const val LALALA = "lalala"
3 }
```

定义变量

变量名后 “：“ 跟着他的类型，类型可以省略掉

定义方法

```
1 // “：“ 后面跟返回值
2 // 参数里面，参数名右面跟着参数类型
3 fun doSomething(x: Int): String{
4     return x + 4
5 }
```

定义 for 循环

可以利用 IDEA 提示，集合打点 for 就可以看到

```
1 for (lesson :Lesson in lessonList){
2     ...
3 }
```

创建对象（直接调用构造器）

```
1 java: Java = Java();
```

返回值

java void

kotlin Unit

kotlin 的类型推断

```
1 // 数据类型 首字母大写
2 var age: Int = 18
3 // 因为类型推断，基本数据类型可以省略
4 var age = 18
```

修饰符

internal 类似 public、private

internal 新增的可见性修饰符，表示当前模块可见。防止跨模块访问

声明类

也是用 class

静态内部类

java 中用 static 修饰；静态内部类可以写静态函数

kotlin 静态内部类是没有关键字修饰；

嵌套内部类

不用 static 修饰的话默认的内部类就是嵌套内部类，嵌套内部类可以获取外部类的引用，不可以写静态函数；

kotlin 嵌套内部类是有关键字修饰，用 inner 修饰

实现接口

实现接口和继承都是没有关键字，实现接口不需要 ()；接口也可以写在继承类的前面；

继承类

```
1 ClassName : 要继承的 ClassName () {  
2  
3 }
```

枚举类型的声明

java

```
1 public enum State {  
2 }
```

kotlin

```
1 enum class State{  
2 }
```

kotlin 不可空类型 和 可空类型？

kotlin 不可空类型；没有问号，不接受 null 值；

var user: User

对应 java 里的 @noneNull

kotlin 可空类型 需要在后面加个 “?”

var user: User? = null

对应 java @nullable

调用符

!! 强行调用符；是和 java 一样的，会报空指针；

? 安全调用符，会判断对象会不会为 null；变量不为 null 的时候才会执行代码；

lateinit

晚些初始化的对象；

```
1 private lateinit var mEtName: EditText
```

java instanceof 对应 kotlin

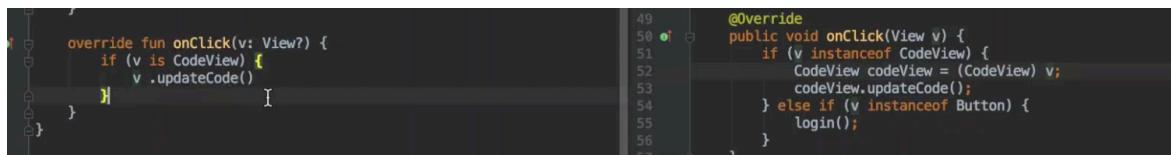
```
1 if(View instanceof TextView){}
2 if(View is TextView){}
```

类型强转

java (TextView) v
kotlin v as TextView 一般都省略，比如判断条件里面



省略后



页面跳转

kotlin 中获取外部类对象使用 [this@ClassName](#)

```
1 // kotlin class
2 startActivityForResult(Intent(this, OtherClassActivity::class))
3 // java class
4 startActivityForResult(Intent(this, OtherClassActivity.class.java))
```

kotlin 无法识别 Context/ this

应该为

```
1 override fun onClick(widget: View) {
2     //正确写法
3     val intent = Intent(this@ActivityName, BActivity::class.java)
4     startActivity(intent)
5 }
```

实体类的构造器声明

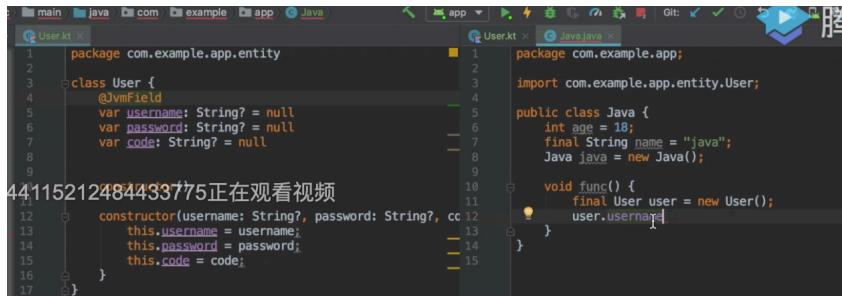
```
1 // 孔构造
2 constructor(){
3 }
4 // 简化为
5 constructor()
6
```

```
1 // 有参构造
2 constructor(userName: String?, passWord: String?){}  
3
```

```
3 ...
4 }
```

kotlin 每个成员属性，都生成了私有的成员变量，和两个公开的 set get 函数；

如果在代码中调用实体的属性，只能 set() get()，想直接调用具体属性，需要在属性上加 `@JvmField` 注解；



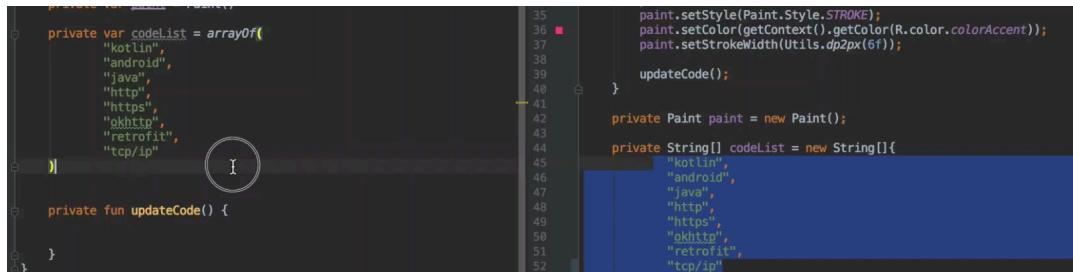
```
1 package com.example.app.entity
2
3 class User {
4     @JvmField
5     var username: String? = null
6     var password: String? = null
7     var code: String? = null
8
9     constructor(username: String?, password: String?, code: String?) {
10        this.username = username
11        this.password = password
12        this.code = code
13    }
14
15 }
16
17 
```

```
1 package com.example.app;
2
3 import com.example.app.entity.User;
4
5 public class Java {
6     int age = 18;
7     final String name = "java";
8     Java java = new Java();
9
10     void func() {
11         final User user = new User();
12         user.username
13     }
14
15 }
```

后来kotlin的新版本中已经没有 get set，直接就是属性名的形式，可以调用了

java 中的 Object 在 kotlin 中为 Any

定义数组



```
private var codeList = arrayOf(
    "kotlin",
    "android",
    "java",
    "http",
    "https",
    "okhttp",
    "retrofit",
    "tcp/ip"
)

private fun updateCode() {
```

```
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52 
```

```
paint.setStyle(Paint.Style.STROKE);
paint.setColor(getContext().getColor(R.color.colorAccent));
paint.setStrokeWidth(Utils.dp2px(6f));
updateCode();

private Paint paint = new Paint();
private String[] codeList = new String[]{
    "kotlin",
    "android",
    "java",
    "http",
    "https",
    "okhttp",
    "retrofit",
    "tcp/ip"
}
```

基本数据类型的 `ArrayOf()`,在前面添加如 `IntArrayOf()`、`FloatArrayOf()`...

java 里的基本数据类型，都是直接的值，而 kotlin 中的 Int, String... 都是一个对象，都是可以调用方法的。

kotlin 里的基本数据类型

Int 是不可空类型 对应 java 里的基本数据类型 但是可以调用包装类的方法

Int? 可空类型 对应 java 里的包装类型 Integer...

kotlin 定义静态函数的三种方式

创建 kotlin 文件

创建 kotlin 文件，在文件里创建函数，以及调用方式。

kotlin 中不需要类名点调用

这种方式叫做 顶层函数；

```

1 package com.example.core.utils
2
3 import android.content.res.Resources
4 import android.util.TypedValue
5
6 private val displayMetrics = Resources.getSystem().getDisplayMetrics()
7
8 fun dp2px(dp: Float): Float {
9     return TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP, dp,
10        null)
11 }

```

```

1 package com.example.app.widget
2
3 import android.content.Context
4 import android.graphics.Canvas
5 import android.graphics.Color
6 import android.graphics.Paint
7 import android.util.AttributeSet
8 import android.util.TypedValue
9 import android.view.Gravity
10 import androidx.appcompat.widget.AppCompatTextView
11 import com.example.app.R
12 import com.example.core.utils.dp2px
13 import java.util.*
14
15 class CodeView : AppCompatTextView {
16     constructor(context: Context) : super(context, null)
17
18     constructor(context: Context, attrs: AttributeSet?) : super(context, attrs)
19     setTextSize(TypedValue.COMPLEX_UNIT_SP, size: 18f)
20     gravity = Gravity.CENTER
21     setBackgroundColor(getContext().getColor(R.color.colorPrimary))
22     setTextColor(Color.WHITE)
23
24     paint.isAntialias = true
25     paint.style = Paint.Style.STROKE
26     paint.color = getContext().getColor(R.color.colorAccent)
27     paint.strokeWidth = dp2px(dp: 6f)
28
29     updateCode()
30 }

```

在 java 里调用 kotlin 中的这种顶层函数方式

```

1 package com.example.core.utils
2
3 import android.content.res.Resources
4 import android.util.TypedValue
5
6 private val displayMetrics = Resources.getSystem().getDisplayMetrics()
7
8 fun dp2px(dp: Float): Float {
9     return TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP, dp,
10        null)
11 }

```

```

1 package com.example.app;
2
3 import com.example.core.utils.UtilsKt;
4
5 public class Java {
6
7     void func() {
8         UtilsKt.dp2px(10f);
9     }
10 }

```

使用 object 声明静态函数

用 **object** 修饰一个类，可以让一个类生成一个单例对象；也就是这个类里所有的属性和方法就是静态的了；如：

```

1 package com.example.core.utils
2
3 import android.annotation.SuppressLint
4 import android.content.Context
5 import com.example.core.BaseApplication
6 import com.example.core.R
7
8 @SuppressLint("StaticFieldLeak")
9 object Cachetools {
10     val context = BaseApplication.currentApplication()
11     val SP = context.getSharedPreferences(context.getString(R.string.app_name), 0)
12
13     fun save(key: String?, value: String?) {
14         SP.edit().putString(key, value).apply()
15     }
16
17     fun get(key: String?): String? {
18         return SP.getString(key, null)
19     }
20 }

```

```

1 package com.example.core.utils;
2
3 import ...
4
5 public class CacheUtils {
6     @SuppressLint("StaticFieldLeak")
7     private static Context context = BaseApplication.currentApplication();
8
9     private static SharedPreferences SP = context.getSharedPreferences("Cache", 0);
10
11     public static void save(String key, String value) {
12         SP.edit().putString(key, value).apply();
13     }
14
15     public static String get(String key) {
16         return SP.getString(key, null);
17     }
18 }

```

kotlin 调用

```
1 str: String = CacheUtils.get(...)
```

java 中调用 kotlin 静态函数

```
1 // 假设 object 创建了 Utils 类, Utils 类中有 toast 公共方法
2 // object 首先是一个类, 然后自动创建了这个类的单利对象
3 Utils.INSTANCE.toast(...);
```

使用 companion object 创建一个类的伴生对象

内部维护单例对象

```
1 class BaseApplication: Application{
2
3     override fun attachBaseContext(base: Context?){
4         super.attachBaseContext(base)
5         currentApplication = this
6     }
7 }
```

```
6    }
7
8    // 创建 伴生对象实现单利
9    companion object {
10        private lateinit var currentApplication: Context
11
12        fun currentApplication(): Context {
13            return currentApplication
14        }
15    }
16 }
```

kotlin 中调用

```
1 val context = BaseApplication.currentApplication()
```

java 中调用

```
1 BaseApplication.Companion.currentApplication()
```

拓展：

为了在 java 中不使用类似下面这种操作

```
1 BaseApplication.Companion.currentApplication()
2 Utils.INSTANCE.toast("lalala");
```

也想直接使用类名点的形式调用，可以通过使用注解来解决

@JvmStatic 只有在 object 声明的才有用，kotlin 编译器会把这个注解对应的函数编译成 java 中真正的静态函数；

@file:JvmName 在 kotlin 创建的文件中声明的方法，想在 java 中类名打点调用，可以把这个注解写在类中 package 包名上面；kotlin 调名的形式调用，没有变化。**@file:JvmName** 注解的目标是文件；

kotlin 声明匿名内部类

使用 **object**:

```
1 // Callback 是接口，后面不需要 ()
2 call.enqueue(object: Callback {
3     override fun onFailure(call: Call, e: Exception){
4
5     }
6
7     override fun onResponse(call: Call, response: Response){
8
9     }
10 })
```

对比修改的一个示例

```

1 import java.lang.reflect.Type
2
3 object HttpClient : OkHttpClient() {
4     private val gson = Gson()
5
6     private fun <T> convert(json: String?, type: Type): T {
7         return gson.fromJson(json, type)
8     }
9
10    fun <T> get(path: String?, type: Type, entityCallback: EntityCallback<T>): T {
11        val request = Request.Builder()
12            .url("https://api.hencoder.com/" + path)
13            .build()
14        val call: Call = this.newCall(request)
15
16        call.enqueue(object : Callback {
17            override fun onFailure(call: Call, e: IOException) {
18                entityCallback.onFailure("网络异常")
19            }
20
21            override fun onResponse(call: Call, response: Response) {
22                val code: Int = response.code()
23                if (code == 200 && code < 300) {
24                    val body:ResponseBody? = response.body()
25                    var json: String? = null
26                    try {
27                        json = body.string()
28                    } catch (e: IOException) {
29                        e.printStackTrace()
30                    }
31                    entityCallback.onSuccess(convert(json, type))
32                } else if (code == 400 && code < 500) {
33                    entityCallback.onFailure("客户端错误")
34                } else if (code > 500 && code < 600) {
35                    entityCallback.onFailure("服务器错误")
36                } else {
37                    entityCallback.onFailure("未知错误")
38                }
39            }
40        });
41    }
42
43    @Override
44    public void onResponse(Call call, Response response) {
45        final int code = response.code();
46        if (code >= 200 && code < 300) {
47            final ResponseBody body = response.body();
48            String json = null;
49            try {
50                json = body.string();
51            } catch (IOException e) {
52                e.printStackTrace();
53            }
54            entityCallback.onSuccess(convert(json, type));
55        } else if (code >= 400 && code < 500) {
56            entityCallback.onFailure("客户端错误");
57        } else if (code > 500 && code < 600) {
58            entityCallback.onFailure("服务器错误");
59        } else {
60            entityCallback.onFailure("未知错误");
61        }
62    }
63}
64

```

简化后代码

```

1 HttpClient : OkHttpClient() {
2     private val gson = Gson()
3
4     private fun <T> convert(json: String?, type: Type): T {
5         return gson.fromJson(json, type)
6     }
7
8     fun <T> get(path: String?, type: Type, entityCallback: EntityCallback<T>): T {
9         val request = Request.Builder()
10            .url("https://api.hencoder.com/$path")
11            .build()
12        val call: Call = this.newCall(request)
13
14        call.enqueue(object : Callback {
15            override fun onFailure(call: Call, e: IOException) {
16                entityCallback.onFailure("网络异常")
17            }
18
19            override fun onResponse(call: Call, response: Response) {
20                when (response.code()) {
21                    in 200..299 -> entityCallback.onSuccess(convert(json, type) as T)
22                    in 400..499 -> entityCallback.onFailure("客户端错误")
23                    in 500..599 -> entityCallback.onFailure("服务器错误")
24                    else -> entityCallback.onFailure("未知错误")
25                }
26            }
27        });
28    }
29
30    @Override
31    public void onResponse(Call call, Response response) {
32        final int code = response.code();
33        if (code >= 200 && code < 300) {
34            final ResponseBody body = response.body();
35            String json = null;
36            try {
37                json = body.string();
38            } catch (IOException e) {
39                e.printStackTrace();
40            }
41            entityCallback.onSuccess(convert(json, type));
42        } else if (code >= 400 && code < 500) {
43            entityCallback.onFailure("客户端错误");
44        } else if (code > 500 && code < 600) {
45            entityCallback.onFailure("服务器错误");
46        } else {
47            entityCallback.onFailure("未知错误");
48        }
49    }
50}
51

```

左侧代码甚至可以简化的更短

```

1 HttpClient.kt
2 import okhttp3.OkHttpClient
3 import java.io.IOException
4 import java.lang.reflect.Type
5
6 object HttpClient : OkHttpClient() {
7     private val gson = Gson()
8
9     private fun <T> convert(json: String?, type: Type): T {
10         return gson.fromJson(json, type)
11     }
12
13     fun <T> get(path: String?, type: Type, entityCallback: EntityCallback<T>): T {
14         val request = Request.Builder()
15             .url("https://api.hencoder.com/$path")
16             .build()
17         val call: Call = this.newCall(request)
18
19         call.enqueue(object : Callback {
20             override fun onFailure(call: Call, e: IOException) {
21                 entityCallback.onFailure("网络异常")
22             }
23
24             override fun onResponse(call: Call, response: Response) {
25                 when (response.code()) {
26                     in 200..299 -> entityCallback.onSuccess(convert(response.body().string(), type))
27                     in 400..499 -> entityCallback.onFailure("客户端错误")
28                     in 500..599 -> entityCallback.onFailure("服务器错误")
29                     else -> entityCallback.onFailure("未知错误")
30                 }
31             }
32         });
33     }
34
35     @Override
36     public void onResponse(Call call, Response response) {
37
38 }
39

```

围绕上面代码的补充：

字符串拼接使用 \$ 符号

表示范围使用

in num...num

如 code in 10...200 意思是 code >= 10 && code < 200

when 条件语句替换了 switch