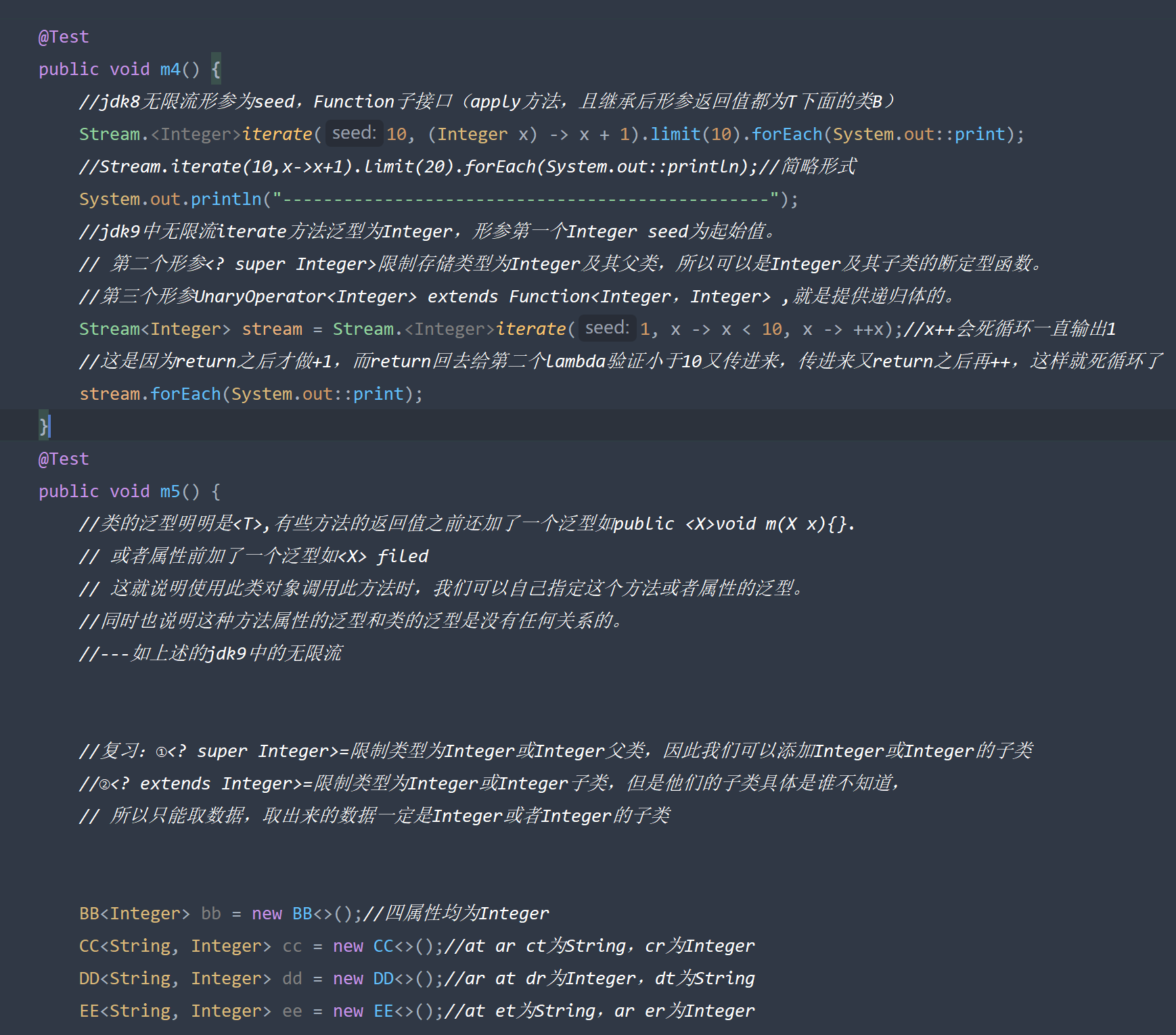
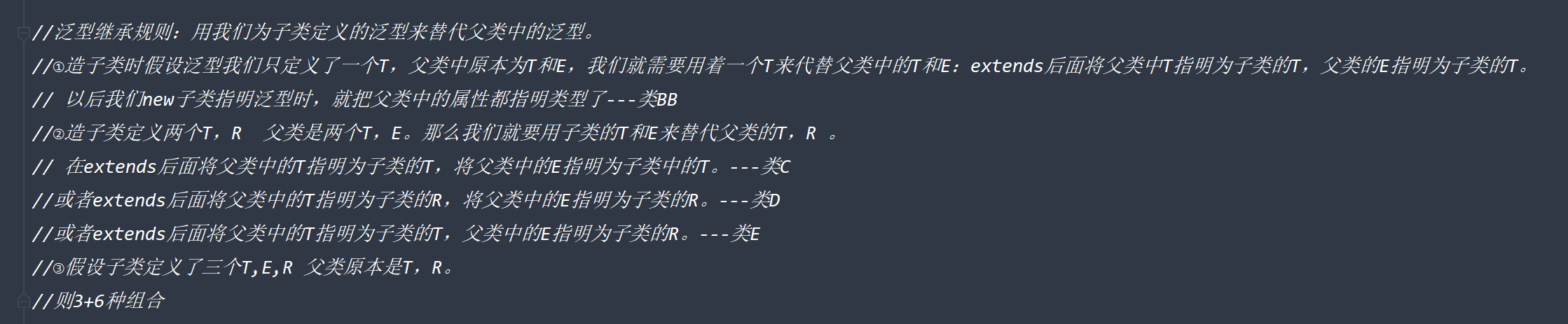
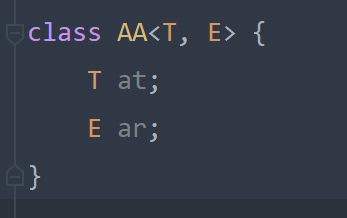
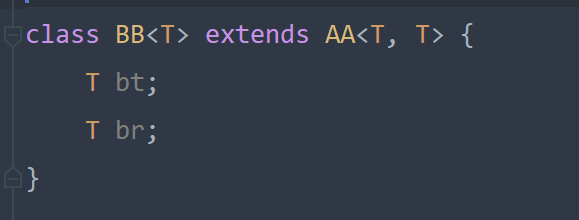
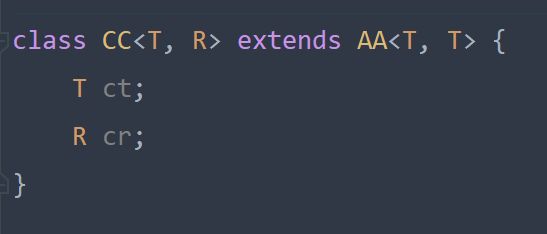
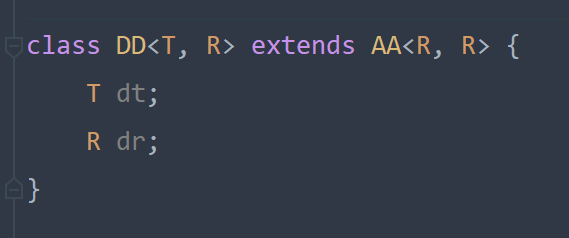
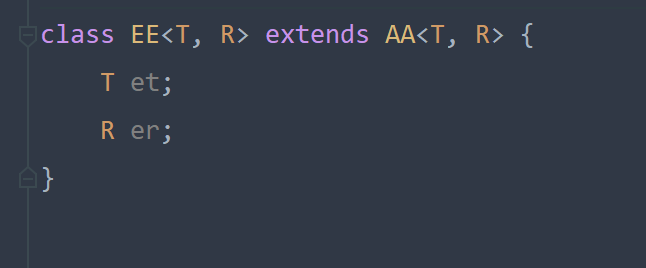
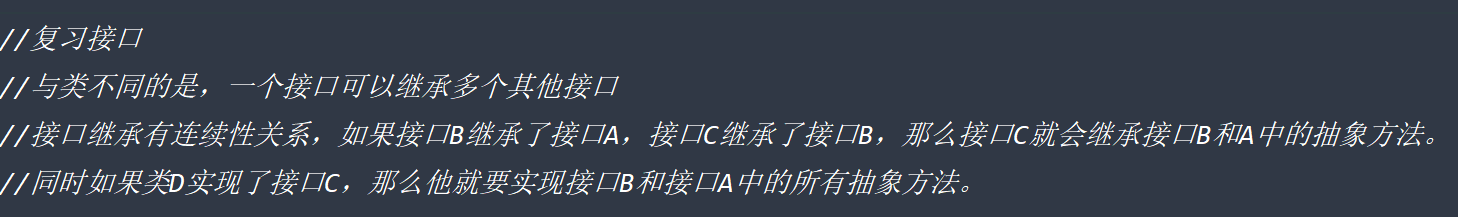
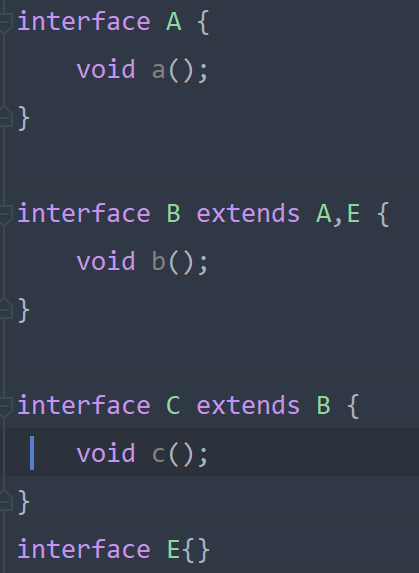
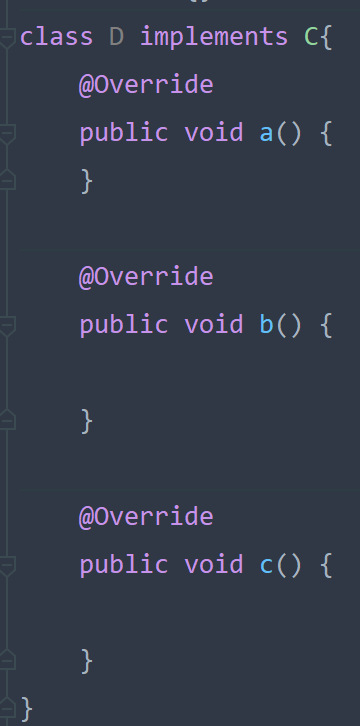
# 泛型复习

# 接口复习



# Java9新特性

相较于8目录结构改变.

## 模块化系统

模块a下的b包的c类想要使用模块a1下的b1包下的c1类：

首先两个模块均使用Jdk9以上的源和依赖。

在模块的src下新建module-info文件



在模块a1的src下的info文件中写入export b1.c1或者export b1表明对别的模块暴露b1包下的c1类或者暴露b1包

在模块a的info文件中写入requires a1; 表明需要a1模块对外暴露的包或者类。

这样a模块就能使用a1模块暴露出来的各种类了。在c类中import导入c1类即可。

因为模块化系统使得各个模块解除耦合性，所以使用Junit时需要requires Junit，并且需要暴露本类供Junit使用，export xxx.xxx。不requires Junit就不能import Junit。

如果测试完毕不再使用Junit可以删除暴露来取消暴露。并且此时Junit测试方法必须声明为public的供Junit模块调用。

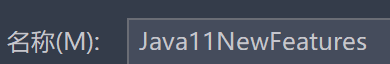
好处：使得一个项目的多个模块之间可以暴露想要暴露的部分实现对接，而不必加载别的整个模块来使得别的模块被完全暴露。

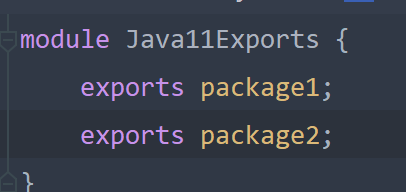
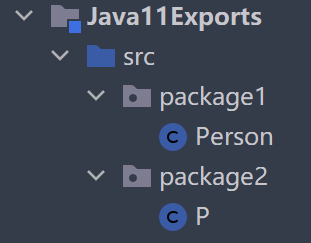
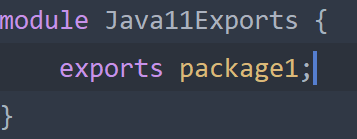
总结：

原本模块a使用模块b中类时，需对模块a添加对模块b的依赖，然后直接import模块b的任意内容。

a模块只想使用b模块的某个类而已，但是b模块所有内容对a模块暴露了。显然会造成b模块的安全性问题和加载对b模块的依赖的运行效率问题。

模块化系统添加了b的依赖后，在b的src下的module\_info中export 什么，就对外暴露什么，a模块就只能requires到b模块暴露出来的那部分，这就解决的ab模块的强耦合性，b模块的安全性，且提高运行的效率。

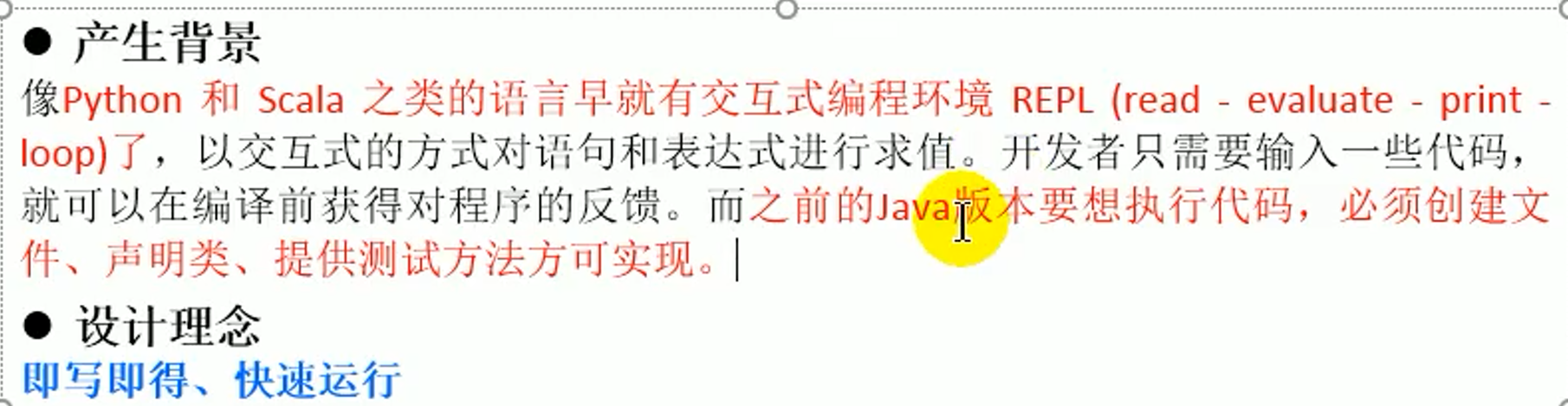




## jShell

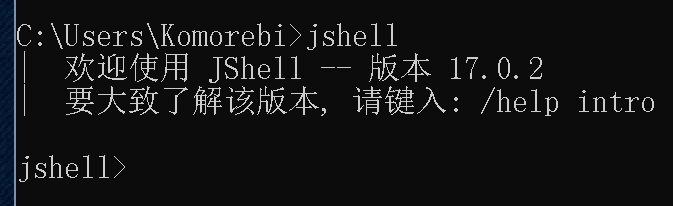
引入jshell使得java不需要在编译环境下可以直接写代码类似于命令行直接运行。

我们可以在jshell中定义简单的类，方法等并执行。一般测试方法数据使用。



在jdk的bin下类似java.exe一样有一个jshell.exe





**建议每条命令完成后加一个分号**

**使用/help调出菜单**

**使用/edit 可以查看已定义的方法，类，变量等，已经执行的所有命令。并且可以在此书写命令。**

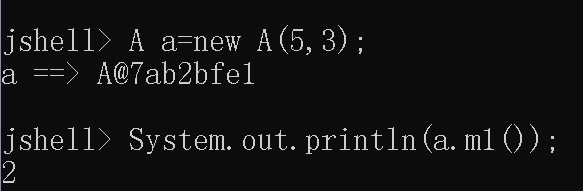
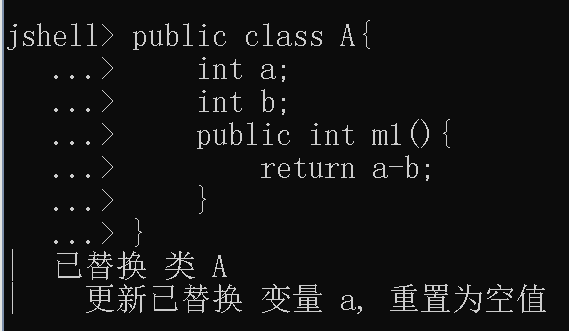
**使用Tab键补全代码（类似idea中alt+enter）**

**使用/imports 查看已导入的包**

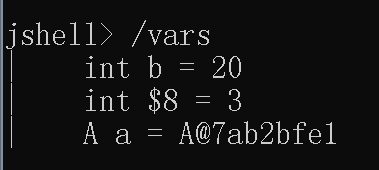
**使用/methods**

jshell定义方法后是可以覆盖此方法的，只需要写一个同名同参同返回值的方法即可。

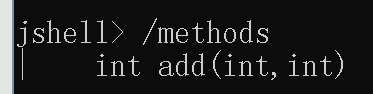
类写一个同名类即可。



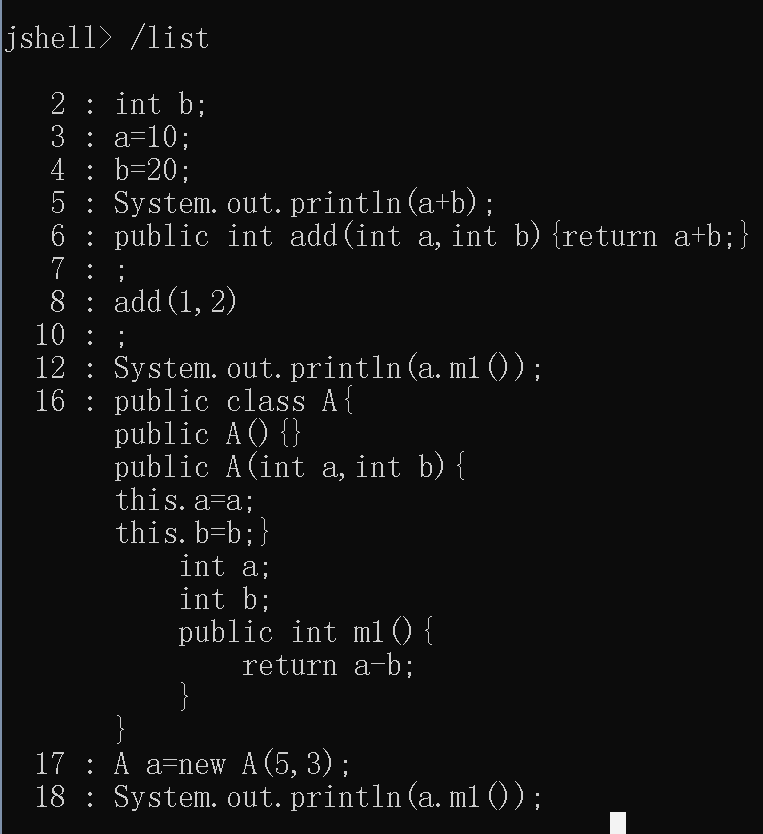
**使用/vars 查看定义过的所有变量（除类中定义的）**



**使用/methods查看定义过的所有方法（除类中的）**



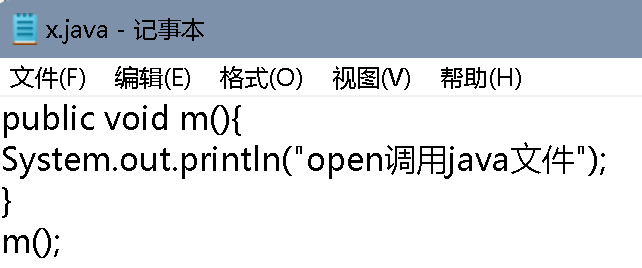
**使用/list查看开始到现在声明过的所有操作的列表**



**使用/open +空格+路径\\xxx.java 直接执行某个路径下的.java文件。**

文件中只需要方法结构，后写调用即可。

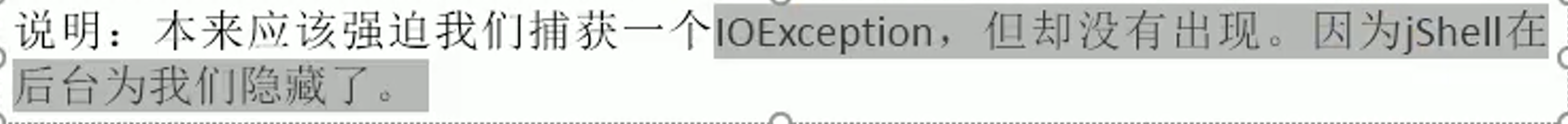






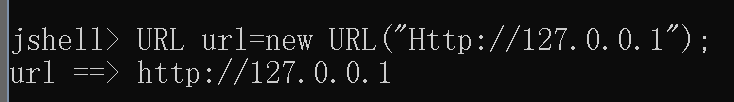
需要注意，java文件必须为GBK格式，否则其中的汉字显示乱码在jShell中

**jshell中不会有编译时强迫的异常处理**

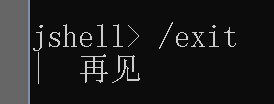


但是如果后续找不到此地址还是会抛出异常。





**使用/exit 退出jshell**

****

## 接口中定义私有方法

java8引默认方法以后，如果某个方法比较冷门不是很需要实现类去实现，则直接在接口中定义默认方法实现方法体即可。可以使用任意实现类对象调用此默认方法。

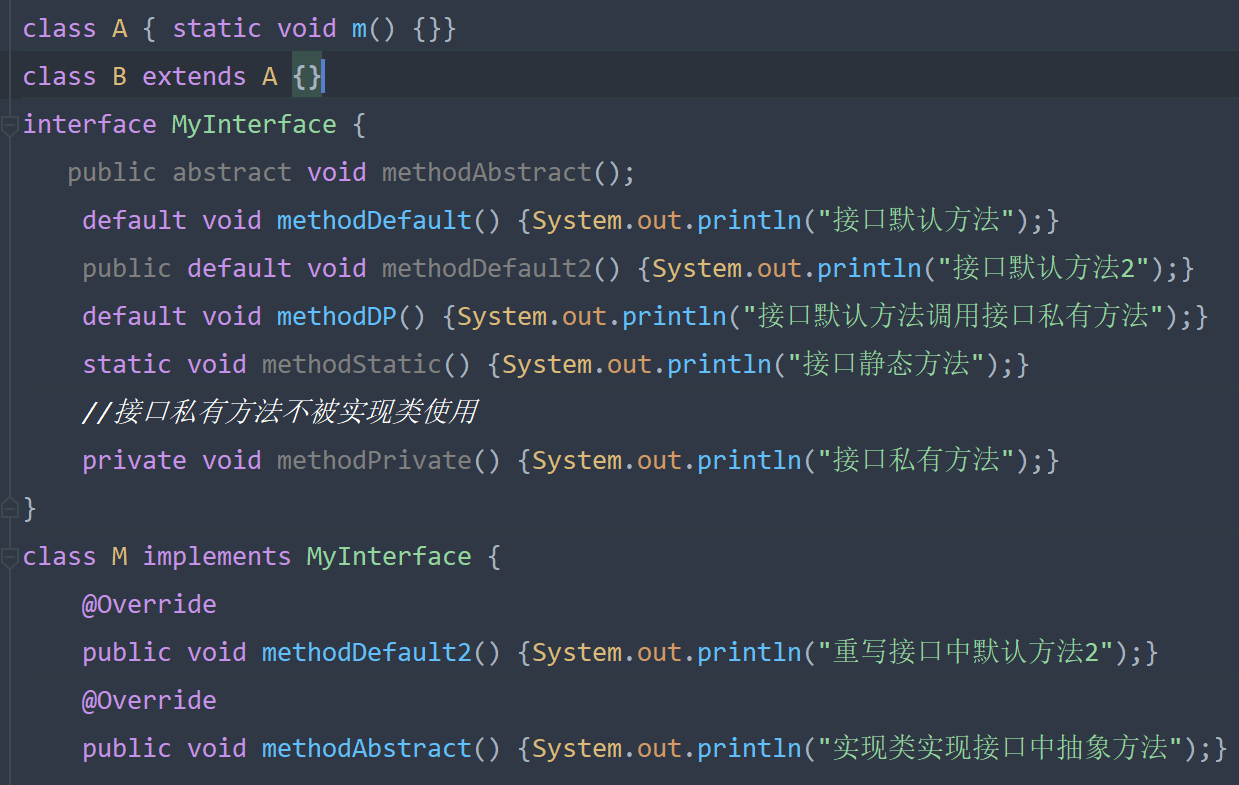
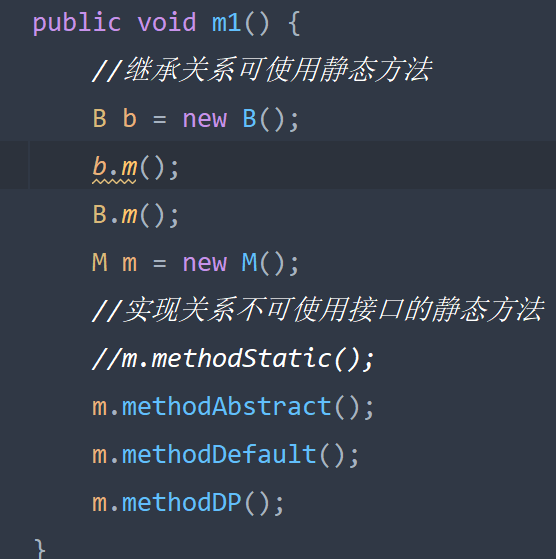
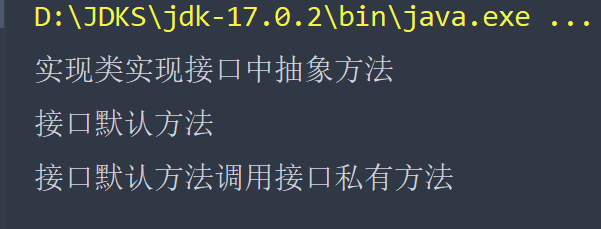
在8版本中还可以在接口中定义静态方法，由于接口与类是并列结构，可以直接使用接口调用静态方法。

且接口中静态方法不被继承，即实现类不能调用接口的默认方法，因为接口是多继承性质，如果多个接口中定义相同的静态方法，实现类实现多个接口时就不知道该使用哪个静态方法。

jdk9版本中新增私有方法，仅被接口内的默认方法使用。实现类可以通过调用接口的默认方法间接调用私有方法。

实现类实现接口时可选择重写接口中的默认方法。必须重写接口中抽象方法。

注意重写接口中默认方法时，会去掉default关键字，因为该关键字只在接口中起一个声明作用。就像abstract一样，实现以后就没这个关键字了。

## 钻石操作符使用升级

接口中泛型符号<>因为像钻石所以命名为钻石操作符。

在jdk8中如果使用匿名实现类的方式实现接口，则构造器部分的钻石操作符的泛型必须指定为声明的变量的泛型，即前后泛型一致都要写上。此时类型推断还没优化。只实现了单纯new类对象的类型推断，没有实现new接口匿名实现类的类型推断。

jdk9之后类型推断优化构造器部分指定的泛型则可以省略不写，钻石操作符还是要写。

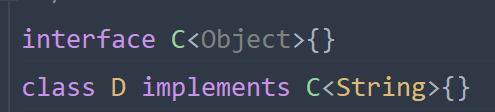
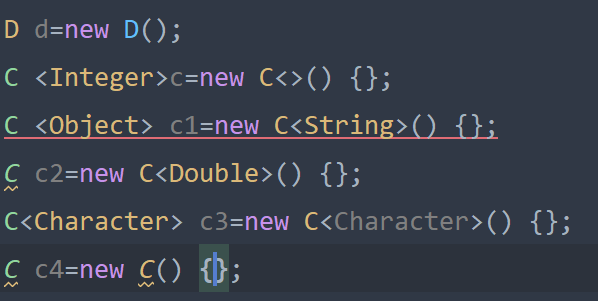
**规则：**

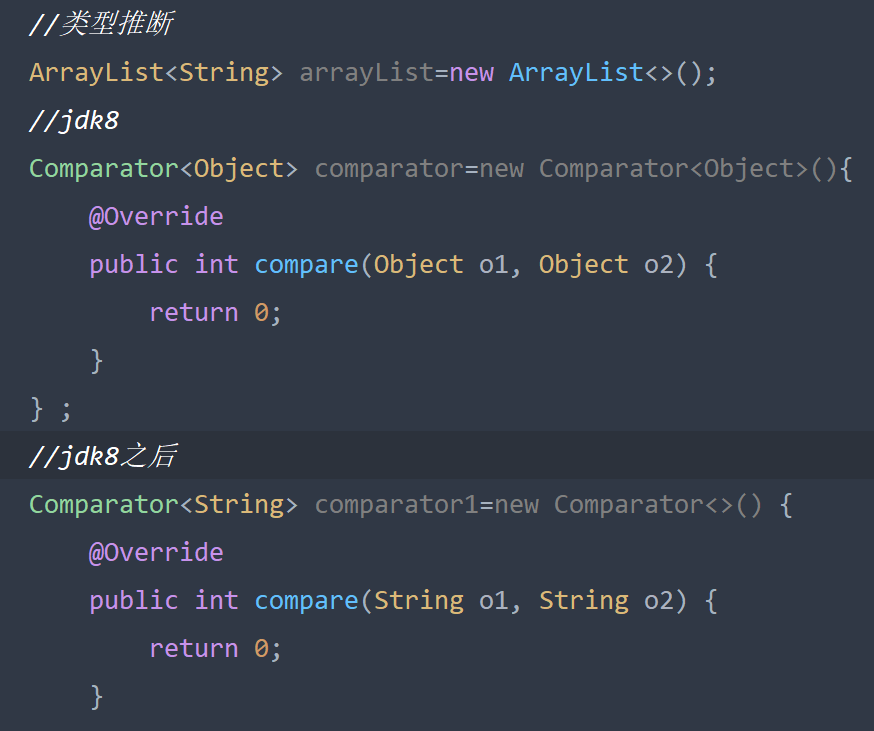
如果匿名实现类与变量都指定泛型，前后泛型类型不一致，不符合语法规则。c1。因为按照类型推断的话实例出来的应为Object类型，但是构造器实际实例出来的是String类型。前后矛盾。如果想让实现类的泛型是接口泛型的子类，就不要写前面的Object，直接构造器泛型写Sting即可。

即本质上匿名实现类的泛型是由后面的构造器的泛型决定的。

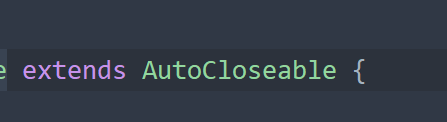
要么使用类型推断前面声明使用泛型，构造器泛型部分空着。然后由编译器推断出后面的构造器泛型为同泛型。

要么不使用类型推断，构造器部分指明匿名实现类的泛型，此时可以为接口泛型的子类。此时如果前面也加泛型的话必须与后面的构造器的泛型类型一致。

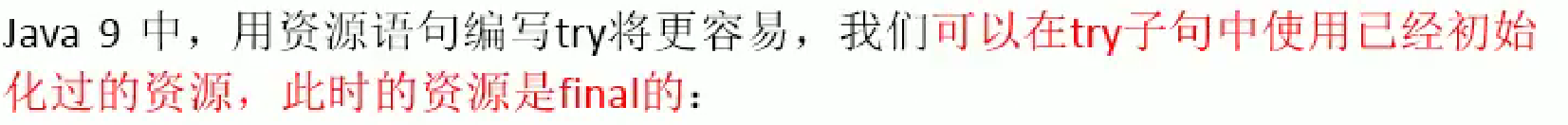


## try结构语法升级

如果一个类或接口直接或间接实现或继承了此接口

那么就可以使用jdk8和9为我们提供的更便捷的try结构。





8和9中try地形参列表中将形参隐式地声明为final。放入资源对象后，都不能在try{}块中再修改资源对象的值，比如将资源对象置为null是不可以的。



## String底层存储结构变更

由于使用String时大多数情况下都是存储abcd这种拉丁字符，他只占用一个字节。

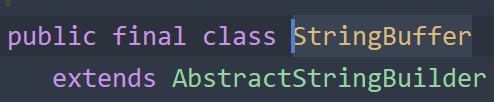
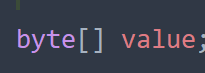
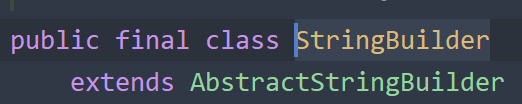
原本String底层是char[] 一个char两个字节，存储拉丁字符浪费一个字节的空间。

所以jdk9取消了底层使用char[] 而是直接使用最轻量的byte[]。

如果检测到要存储汉字，那就在第一个byte做标记，连续使用下一个byte。这样就相当于使用两个byte—一个char存储一个汉字。

同时StringBuffer和StringBuilder底层也使用byte[]

HotSport虚拟机 string相关也是byte[]



## 创建只读集合

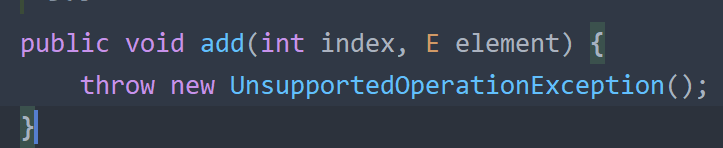
在Arrays工具类中有方法asList，将任意类型对象转为List集合，这个集合是后续不可添加任意数据的。调用add虽然不报错，是因为add方法仍然存在，但是方法内直接回抛出一个异常。

**asList方法内部**



返回的ArrayList不是Until下的，而是Arrays中的一个内部类。其与Until下的ArrayList都继承了父类

这个类下的add方法实际上是抛出异常的



Until下的ArrayList继承后重写了才没有报异常。

**jdk8中创建只读集合：**

用Collections工具类的unmodifiablelist方法放入形参为任意list对象。

unmodifiableset方法放入形参为任意set对象

unmodifiable方法放入形参为任意map对象

然后返回一个新的对象对应的内部类集合对象。数据都拷贝一遍。



## InputStream加强

用类的加载器生成一个InputStream流用来读取文件。

这个InputStream对象在jdk9中可以直接调用transfer（转移的意思）方法，形参传入一个OutputStream类型及其子类的对象就可以自动将数据写出。

因为transfer方法内部实际上还是造byte数组，i，while循环，判断i=当前对象的read（byte）是否不为-1，不是-1就调用形参OutputStream类型及其子类对象的write方法。

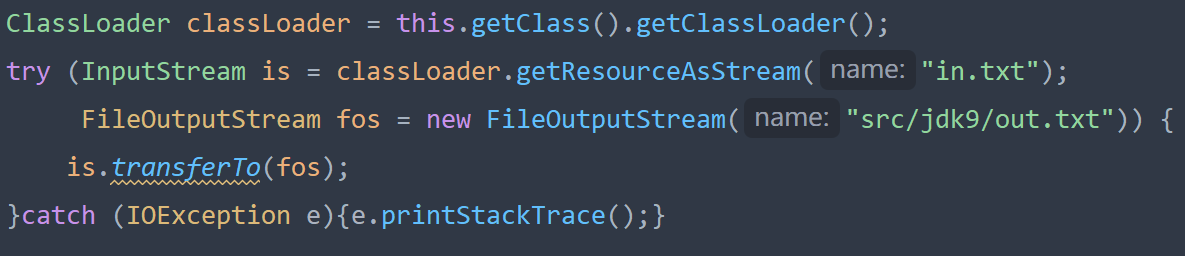


所以我们只需要用类的加载器调用getResourceAsStream(“读取文件路径”)

方法拿到InputStream is ，再造一个OutputStream os构造器指定文件路径。

再调用is的transfer方法将os传入即可完成数据自动写出。

注意加载器生成的InputStream路径为模块src下的文件，OutputStream路径相对于src任意



## StreamAPI新增4个方法

使用sql类数据库可以直接在java层面使用sql语句，然后去sql数据库中查找数据时进行中间操作。

但是nosql数据库不支持在数据库进行中间操作，这就使得必须在java层面进行中间操作。jdk9对StreamAPI进行了4个方法的新增，使得java层面的中间操作更加便捷。

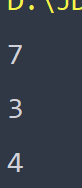
**takewhile（）中间操作**

该方法为list.stream流的方法，方法形参为断定型接口对象，使用lambda实现接口，若lambda为list对象满足某一条件，则takewhile会从list从前到后寻找，如果对象满足条件，则保留该对象返回到新stream流中，如果对象不满足条件，则终止takewhile方法。

**dropwhile中间操作**

形参为断定型接口实例，使用lambda写条件方法体。该方法会在list中从第一个对象开始，按照lambda条件检测对象，如果对象满足条件，去除该对象，直至某一个对象满足条件，则该对象后面的对象不论满不满足条件都会保留。

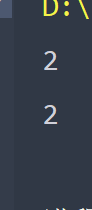
 

**ofNullable实例化stream**

由于在jdk8中Stream.of()方法如果形参列表只放了一个对象数据，那么这个对象不能为null。

其余情况任意。

那么在jdk9中就出现了另一个方法ofNullable（），该方法补充了jdk8中of的缺点，且该方法只允许放一个对象数据，这个对象是不是null都可以。（如果允许放多个还要of方法干啥）

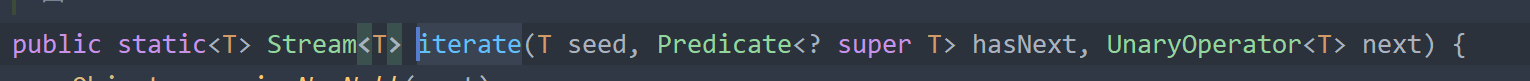
****

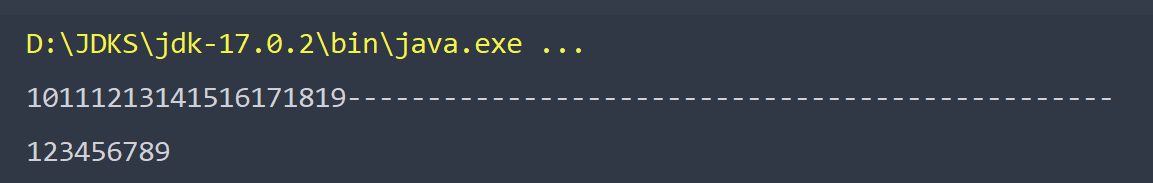
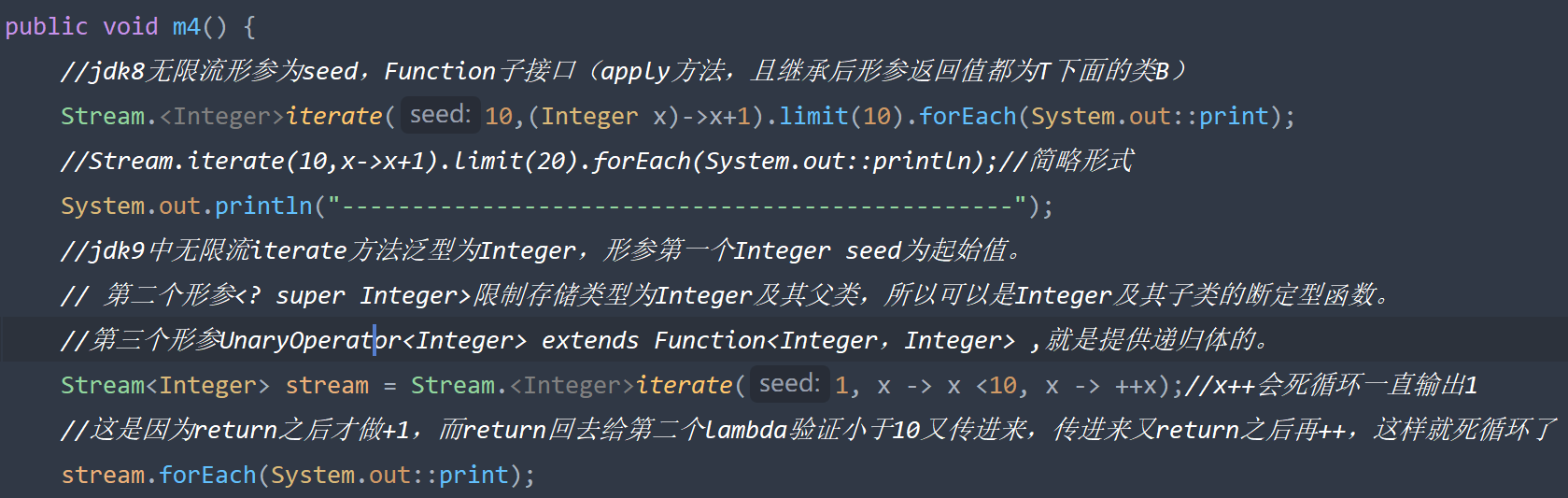
**重写的iterate无限流方法**

jdk8中的无限流递归只能通过加上中间操作limit(次数)来限制循环次数。

jdk9中，重载了iterate方法，将形参变更为三个，类型为方法泛型的seed开始数一个，类型为方法泛型子类的断定型函数实例一个，然后就是类型为方法泛型的Function子接口一个

中间新增的这个断定型就是用来作为循环的限制条件的，这个限制条件是将Function子函数返回值作为形参传入判定的。





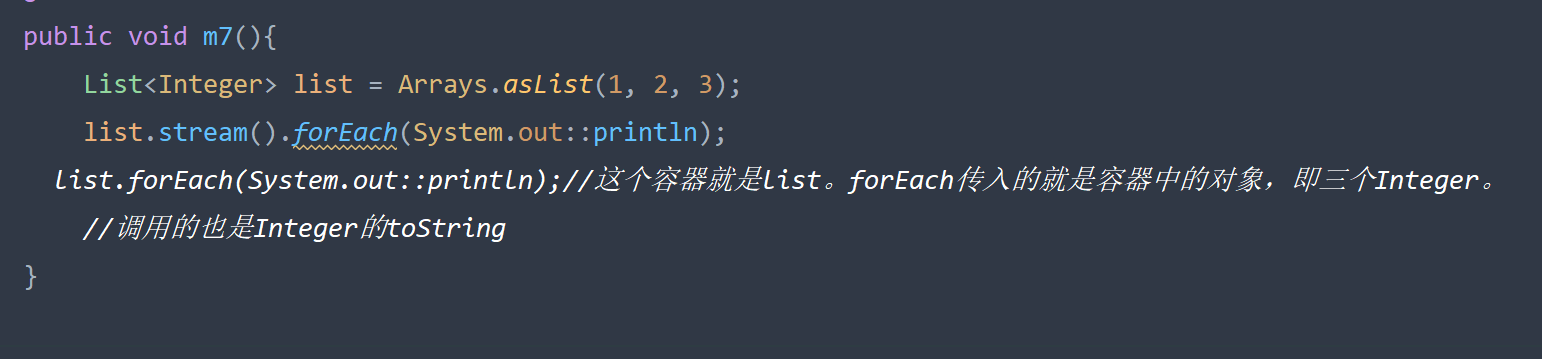
## Optional获取stream

Optional虽然只能存储一个对象数据，但是也是个容器，可以将一个list对象添加进去啊，这样本质上就有很多数据了。

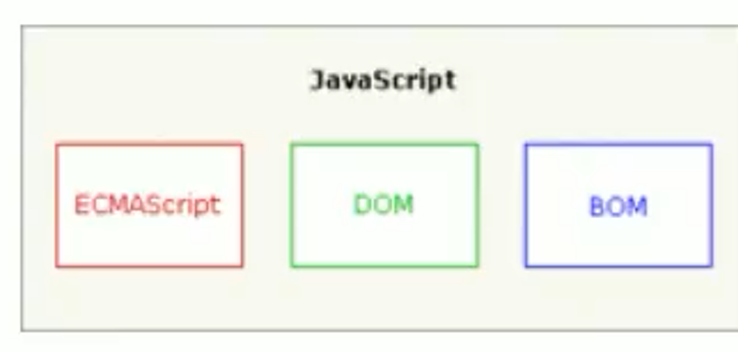
所以jdk9中为Optional添加了获取stream流的方法 stream（）

需要使用Optional对象点stream（），从而对其存储的对象数据包含的第二层数据进行中间终止操作。





## JS引擎升级



第一部分是js语法树，引擎升级就是为了不依赖框架直接使用引擎API解析js语法树。

第二部分DOM是js与后台交互----后端web学习

第三部分BOM是js与浏览器内核交互

# Java10新特性

## 局部变量的类型推断

### 概述

现在我们要在方法中造一个集合对象，通常是先new ArrayList<String> 然后让编译器帮我们补全前面的类型。

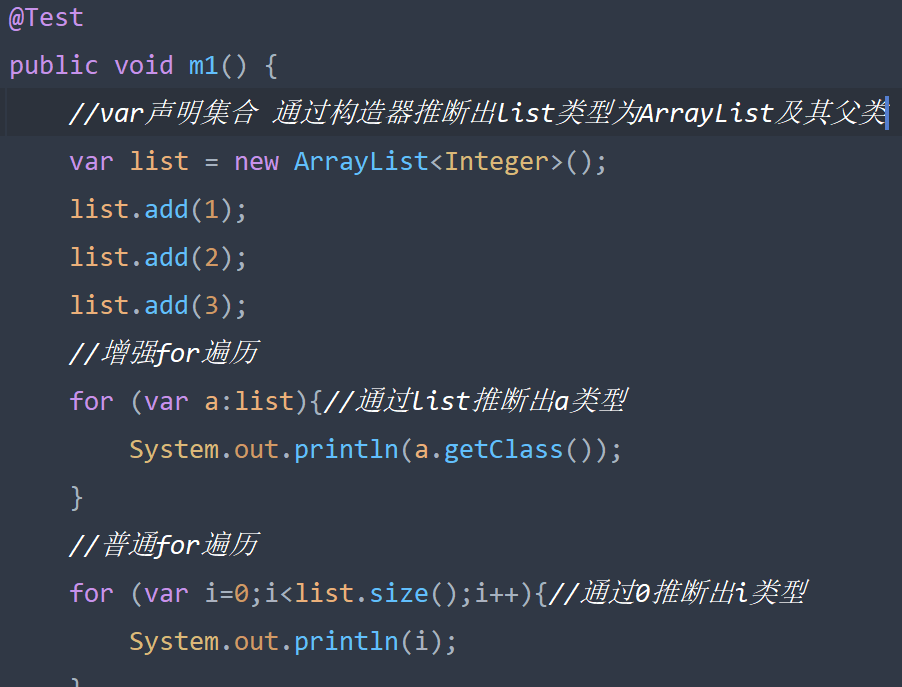
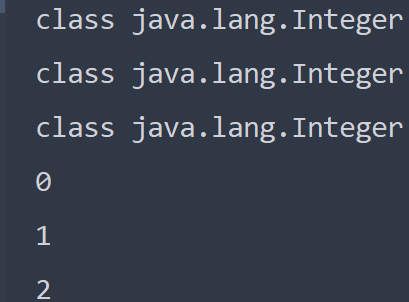
这就说明，前面的类型是可以通过后面的构造器推断得出的，要么是同级，要么是他的父类。

日常中为了简化声明，并将注意力更多放在变量=右边的方法构造器等操作而不是被变量类型困惑，于是推出了一种新类型—var

var list=new ArrayList<String>()；

众多类型都可以用var表示，包括基本数据类型。

var还是不能省去的在声明时，这代表我们是第一次声明。去掉就是已声明过的变量了。

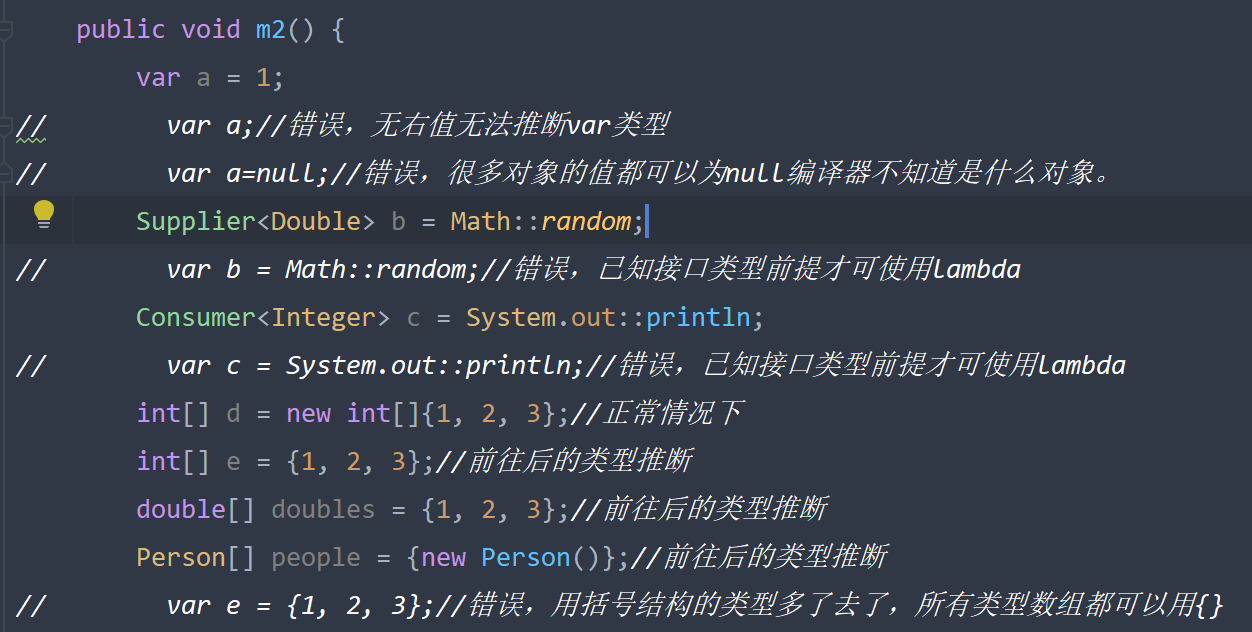
 

### 不适用情况：

1. 由于var类型推断是给出右值往左推类型，所以推断变量类型时必须给变量赋值。这也就说明了不能直接var a就完了，这就没法推断var实际代表的类型了，所以当一个变量是局部变量时，必须得有值。
2. 对于lambda表达式和方法引用，他是根据已经声明的接口类型，进而知道其抽象方法参数，返回值等信息，然后来重写抽象方法的，也就是说得先知道声明的是什么接口，才能使用lambda。如果只是写一个lambda。然后左边用var，鬼知道这是哪个接口的lambda。此时的lambda也没有任何意义。
3. 数组可以声明类型=右边一对{ }在里面直接放对象。

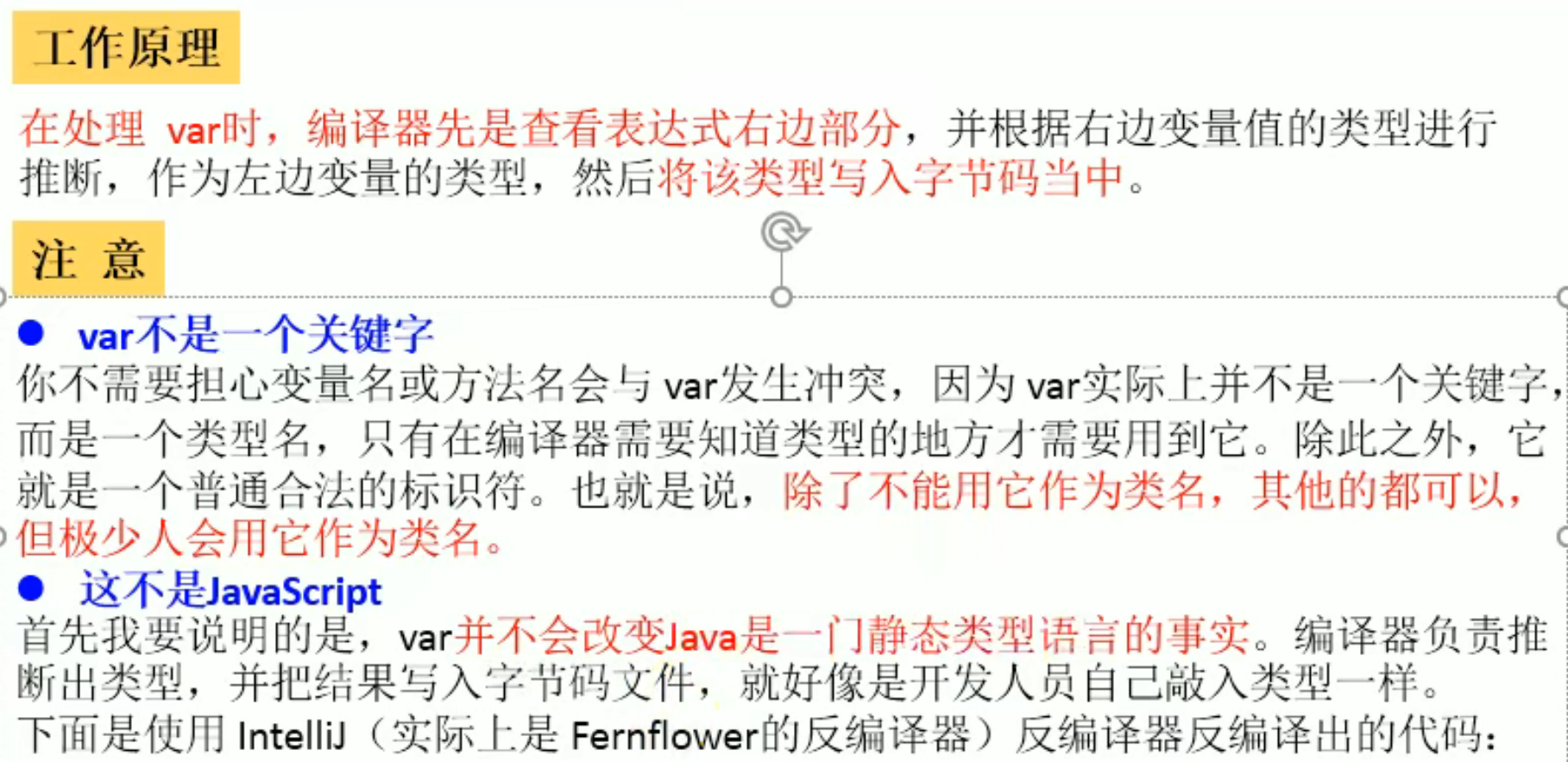
但是不给类型给个var，用{ }的数组多了去了编译器不知道是哪个类型的数组。

而变量是必须要给一个确定类型的。



1. 不能用作属性的声明，即便属性赋值了。因为属性总有不赋值有默认值0或者null的时候，0还好说，null就不行了，是个引用类型的值都可以是null。属性是必须要确定类型的。
2. 不能用作方法形参，方法形参的类型是用来限制传入方法的实参类型的，如果使用了var，相当于是根据实参来确定形参类型了，那么形参列表就起不到限制形参类型的作用，乱传对象到方法体中肯定出错。
3. 不能用作方法返回值，这个跟5一样的道理，返回值类型是用来限制返回值的。要是成了var，岂不是返回什么值都可以，必然会导致使用此方法后一系列的报错。
4. 不能用作构造器的形参 跟5一样的道理
5. 不能在catch块的形参列表中使用，如果使用了var，鬼知道你try块是包括的什么异常，如果我认为var是个小IO异常，结果try包括的实际是个大的RuntimeErxception，那出了异常就不能处理后继续执行了，会直接断掉程序，try还有什么意义。

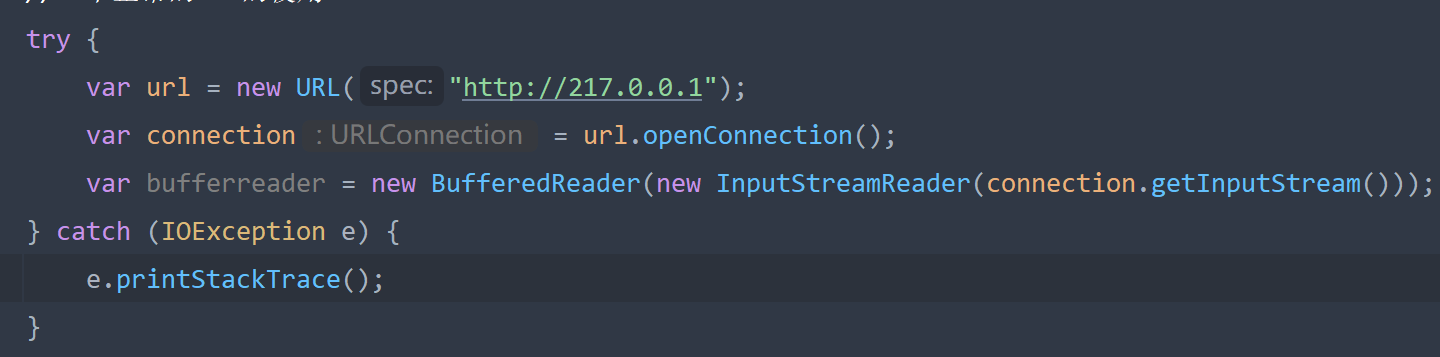
### 原理



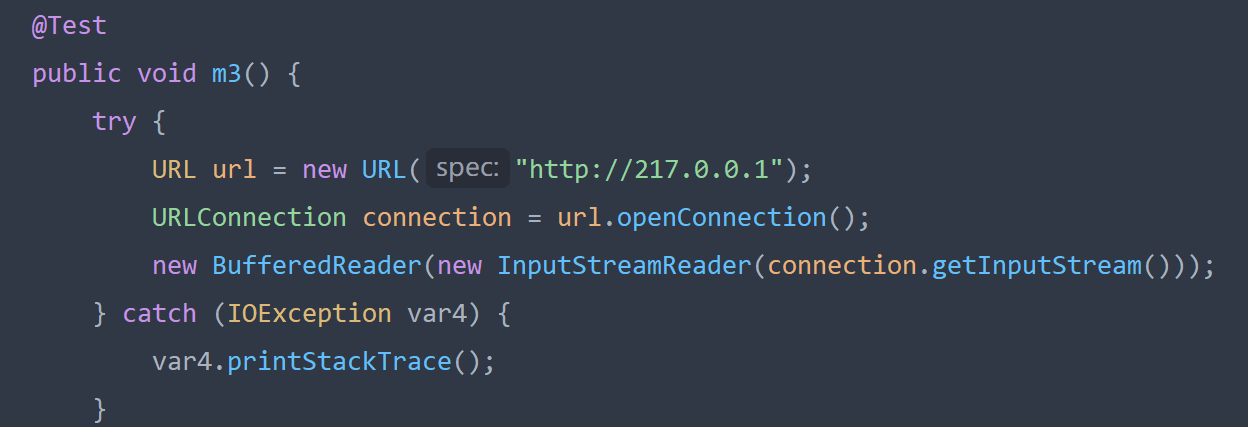
var其实就是3个很普通的字母，不是关键字int double这种，所以var是可以当作方法名，属性名，变量名等名称的。但是var不能当类名（这样使用var时会声明为自定义的var类）。

编译器工作时如果碰到var类型，马上从右往左推，进而推出var实际上是什么类型。进而把这个类型写道字节码文件中，所以尽管代码中是var，字节码文件中是具体的类型。通过idea的反编译class文件也可以看到具体的类型。

**代码中**

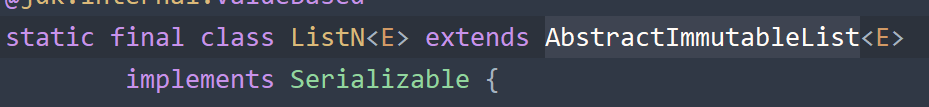


**反编译的字节码中**



## 新增创建不可变集合

jdk9中List，Set，Map三大接口的of方法返回的只读集合都是这个。该方法是往里面放对象数据生成一个只读集合。



jdk10中新增一个copyof方法，将任意集合对象放入形参列表，返回的就是一个上面的只读集合对象。

但是如果放入的集合对象本身就是只读的，copyof方法返回的还是放入的集合本身地址。

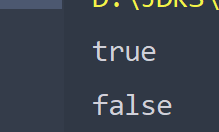
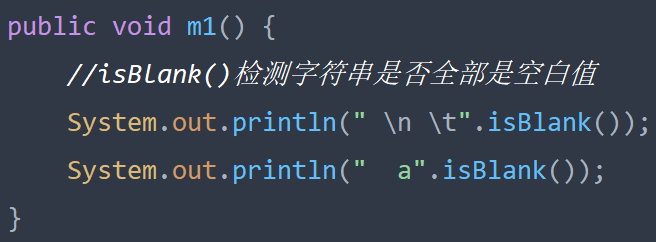
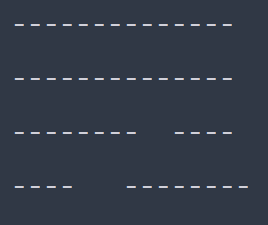
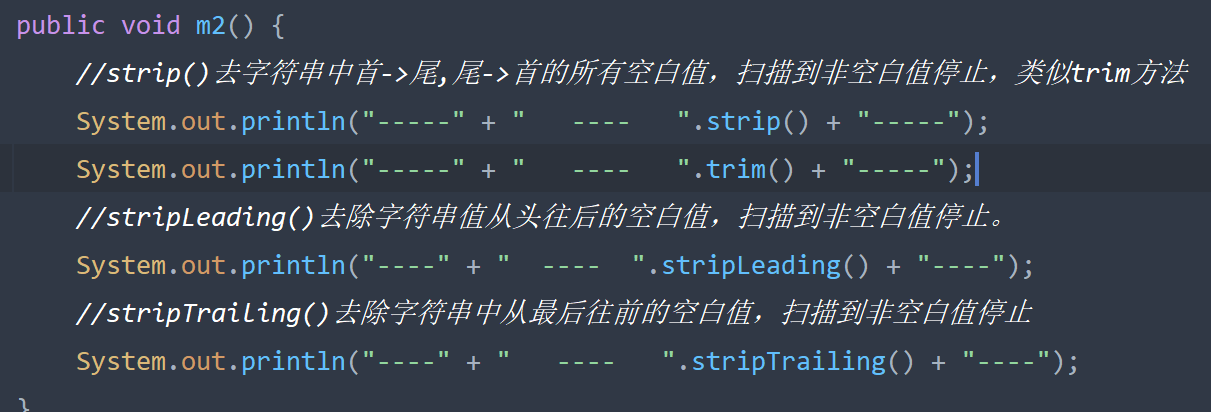
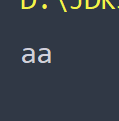
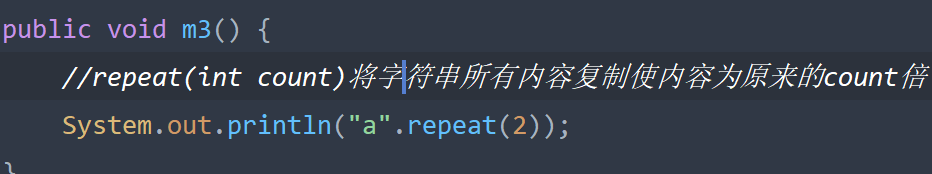
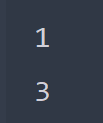
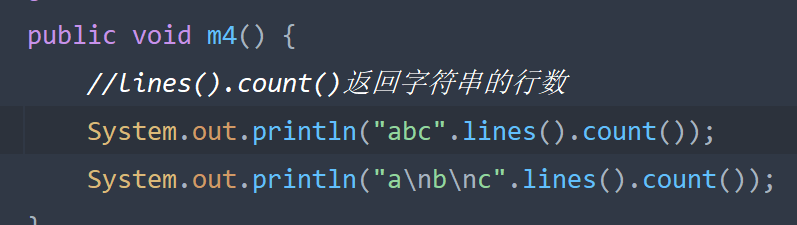
放入的集合对象是可写的，返回的就是一个新的只读集合对象，将原集合数据复制了一份。

注意Arrays中的asList方法虽然返回的也是只读集合，但是跟三大接口的of方法返回的不是一种只读集合结构asList返回的是内部类

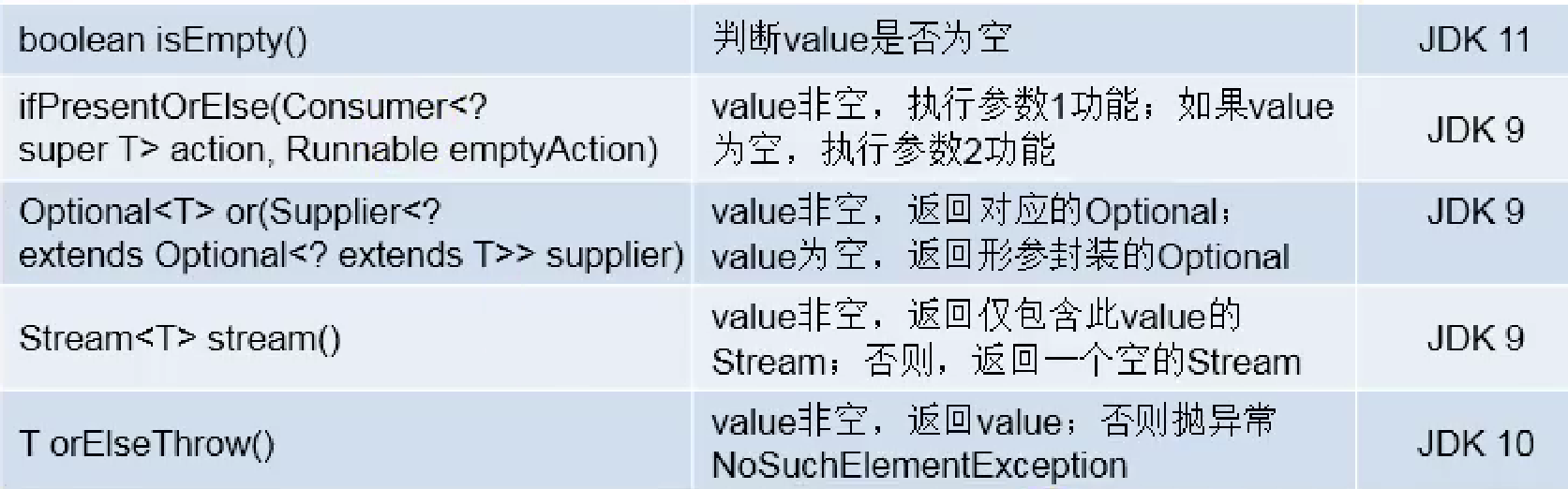


# Java11新特性

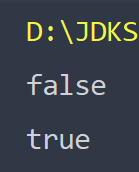
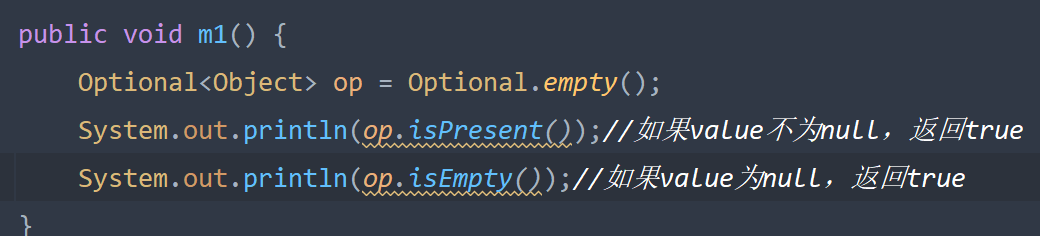
## String新增的方法

1. 
2. 
3. 
4. 

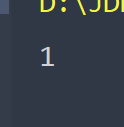
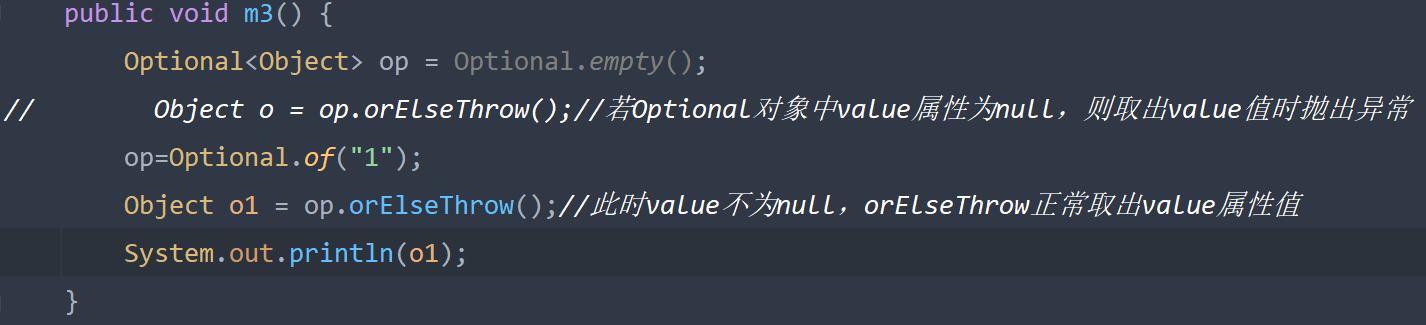
## Optional的再次加强



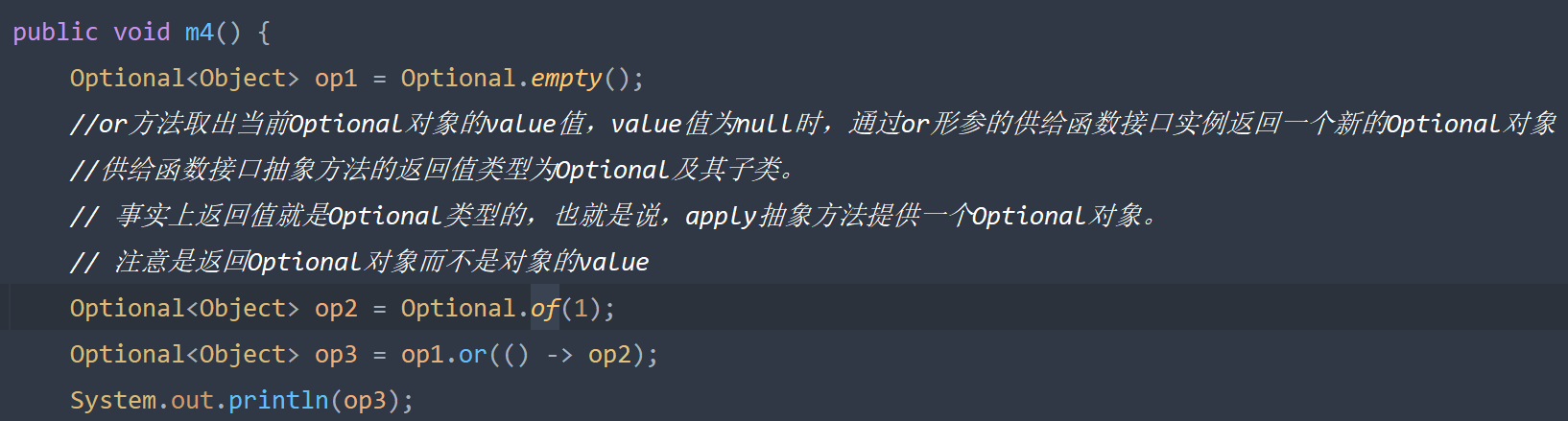
1. **isempty（）**

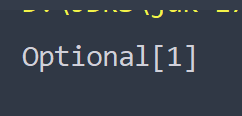


1. **orElseThrow()**



1. **or()**

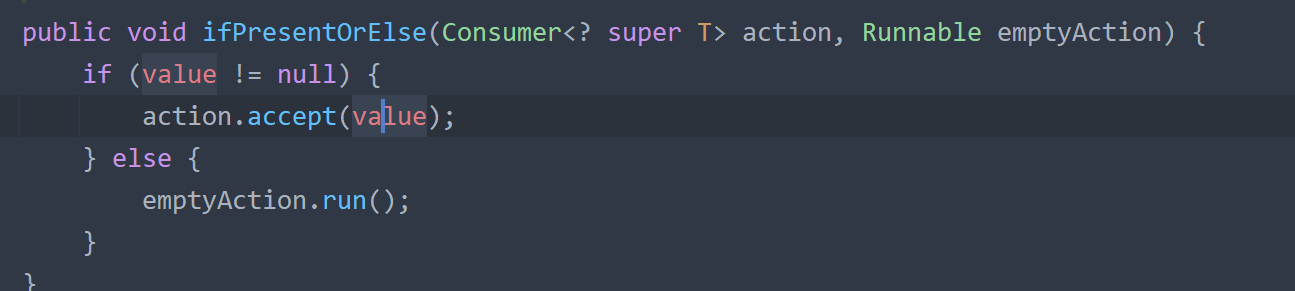


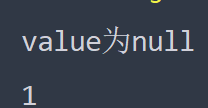
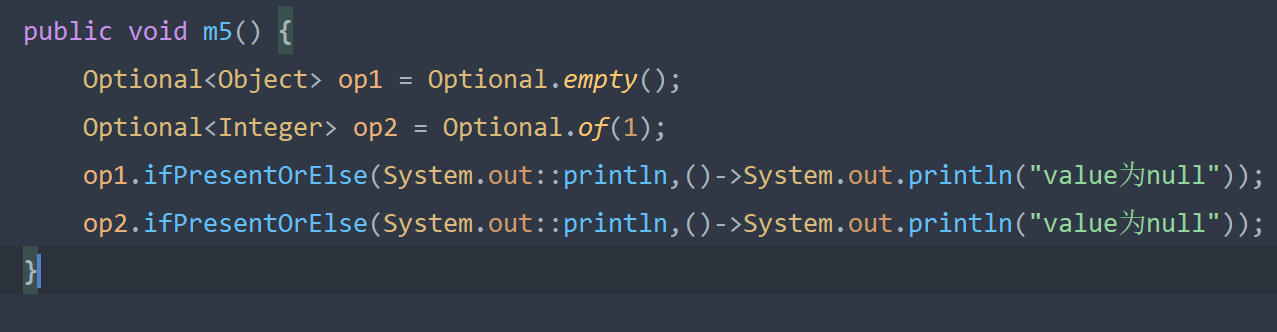


1. **ifPresentOrElse（）**

形参为两个函数式接口，第一个是消费者类型的，第一个接口抽象方法为accept（），形参类型是value值的类型，第二个是Runnbale接口，方法是run（），无参无返回值。

如果取当前Optional对象的value值时不为null，就调用第一个接口实例的accept，并将value值当作参数传入。 为null就调用第二个接口实例的run。



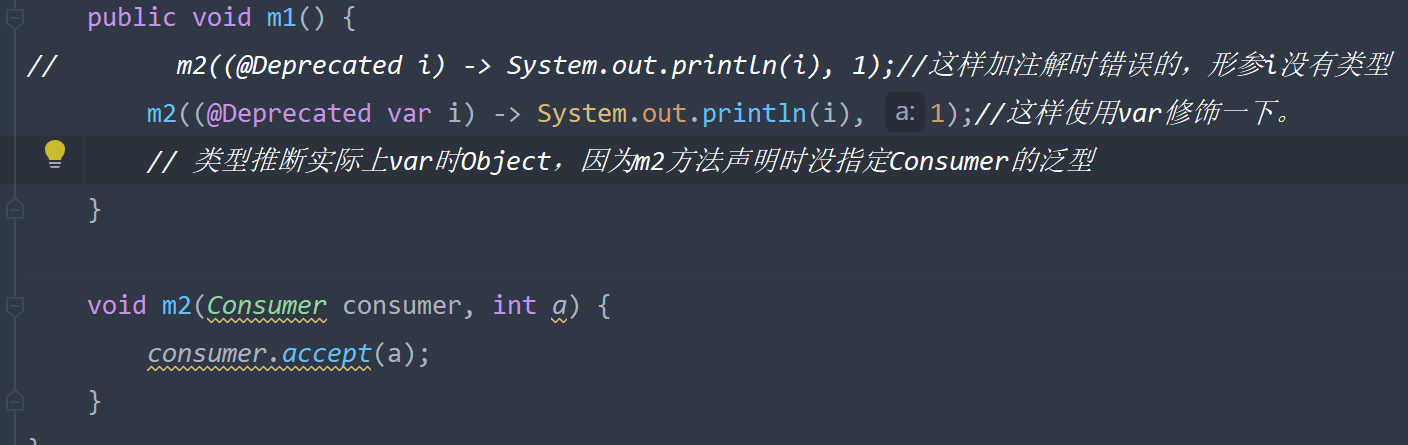
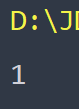


## 局部变量升级

如果我们想使用注解修饰形参，那么形参必须得有类型比如void m（@Deprecated int a）{}

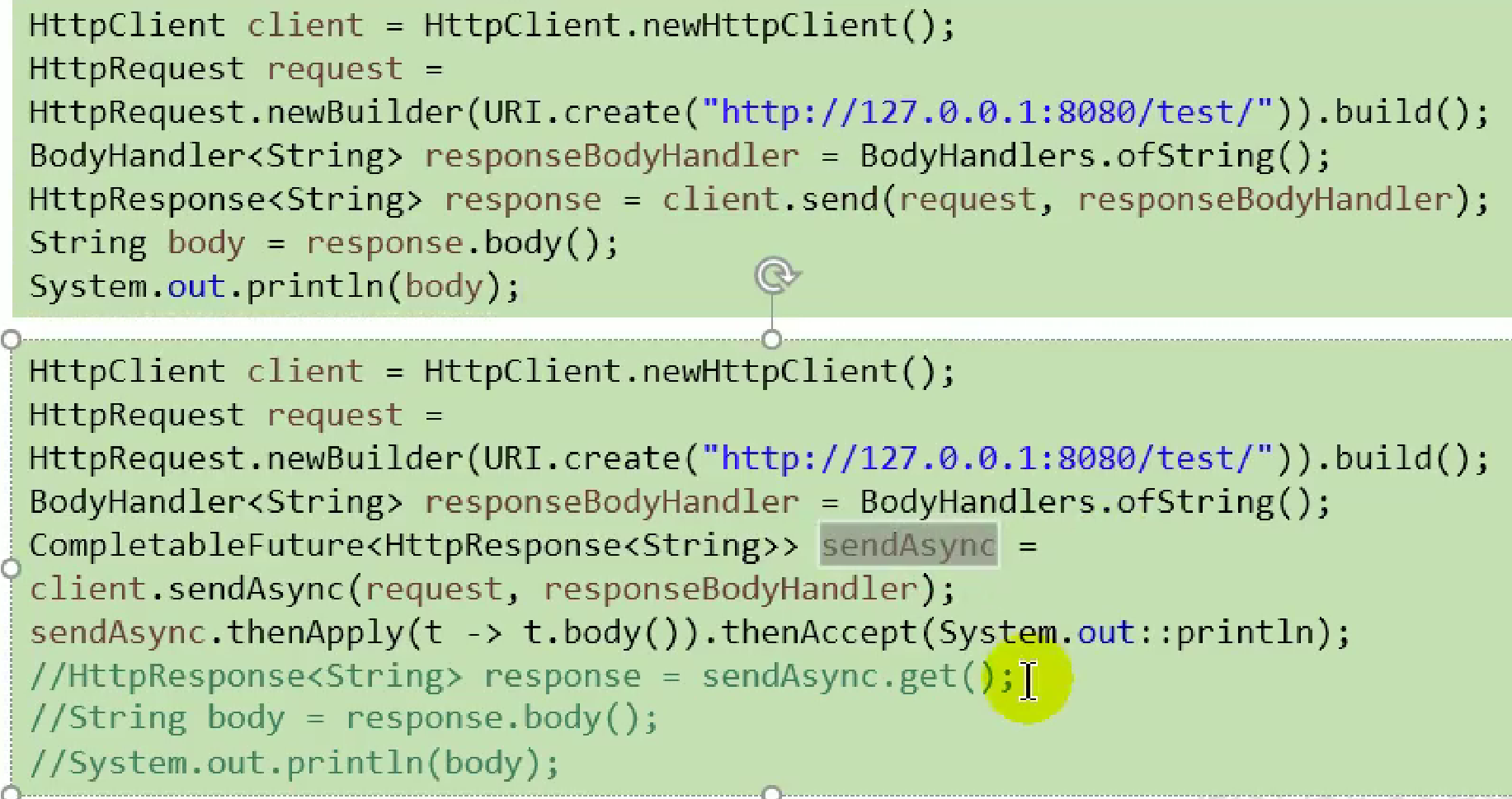
在lambda中由于绝大部分接口带有泛型，所以一般在形参列表中都省略形参类型，由接口泛型进行类型推断出形参类型，但是如果用注解修饰形参列表时，就报错因为没有显示声明类型。

这时如果lambda是作为一个方法的形参出现的，我们是匿名实例化，那么在不确定函数接口的泛型到底是什么不方便给lambda中形参显示加上类型时可以使用var 修饰一下形参列表。而且在类型推断中var会自动被推断成接口的泛型具体的类型。

## HttpClient

第一个同步方式第二个异步方式



使用HttpClient替换原来的URL对象调用的connection方法返回的链接对象。



需要在module的info中requires一下http包

## 更简化的编译运行程序

写好hello.java文件后，在cmd中，我们可以使用javac hello.java命令生成hello.class字节码文件。

然后在cmd中使用java hello命令运行字节码文件从而运行java程序。

但是jdk11允许我们直接使用java hello.java 允许java文件。看似省去了生成字节码文件的步骤，实际上只是隐藏了。

要求：

1. hello.java中必须有main方法，并且第一个类一定要有main方法。因为这种方式运行只会找第一个类中的main方法进入，而不会去找下面的类。

即使第一个类不是主类，也是找第一个类中的main方法，找不到就报错。

1. java文件中不能去实例其他java文件的类，或者使用其他java文件中的结构。这个java文件只能使用自己文件中的类。

所以这种方式只适合一个java文件中有一个类，一个main方法，然后main方法测试自己这个类中的结构。

## 垃圾回收

java两大利器：jvm，可以运行任何语言 和强大的垃圾回收器

垃圾回收器就像空气，我们感知不到却无处不在

最开始的G1是重量级使用时间最久的垃圾回收器

后来诞生了GC，有着处理下至几百兆上至几个T的堆中的垃圾的能力，并且延迟超级低<10ms 。因为垃圾回收器工作时整个java程序都会暂停，给回收器让路。

