# 一、学习目标

1. 能够使用@FunctionalInterface注解

2. 能够自定义无参无返回函数式接口

3. 能够自定义有参有返回函数式接口

4. 能够理解**Lambda延迟执行**的特点

5. 能够使用Lambda作为方法的参数

6. 能够使用Lambda作为方法的返回值

7. 能够使用**Supplier**函数式接口

8. 能够使用**Consumer**函数式接口

9. 能够使用**Function**函数式接口

10. 能够使用**Predicate**函数式接口

# 第一章 函数式接口

## 1.1 Lambda

一般都叫Lambda表达式（λ expression）是一个[匿名函数](https://baike.baidu.com/item/%E5%8C%BF%E5%90%8D%E5%87%BD%E6%95%B0/4337265)，Lambda表达式基于数学中的[λ演算](https://baike.baidu.com/item/%CE%BB%E6%BC%94%E7%AE%97)得名，直接对应于其中的lambda抽象（lambda abstraction），是一个匿名函数，即**没有函数名的函数**。Lambda表达式可以表示[闭包](https://baike.baidu.com/item/%E9%97%AD%E5%8C%85/10908873)（注意和数学传统意义上的不同）。

## 1.2 编程范式

常见的编程范式有**命令式编程（Imperative programming）**,**函数式编程**，**逻辑式编程**。常见的面向对象是一种命令式编程。

**命令式编程：**是面向计算机的抽象，有**变量**（对应着存储单元），**赋值语句**（获取，存储），**表达式**（内存引用和算术运算），**控制语句**（跳转、循环等），一句话：命令式编程是一个**冯诺依曼机**的**指令序列**。

**函数式编程：**是面向数学的抽象，将计算描述为一种**表达式求值**，一句话：函数式程序就是一个**表达式**。

## 1.3 函数式接口

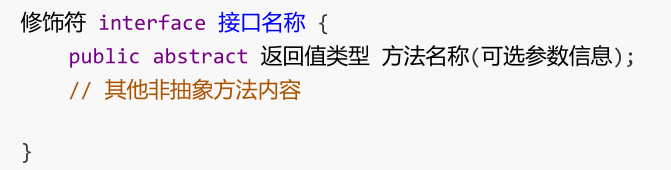
有且仅有一个抽象方法的接口。这个接口就是函数，就是一个表达式，所以可以使用Lambda表达式。

即适用于函数式编程场景的接口。而Java中的函数式编程体现就是Lambda，所以函数式接口就是可以适用于Lambda使用的接口。只有确保接口中有且仅有一个抽象方法，Java中的Lambda才能顺利地进行推导。

**“语法糖”**是指使用更加方便，但是原理不变的代码语法。例如在遍历集合时使用的for-each语法，其实底层的实现原理仍然是迭代器，这便是“语法糖”。从应用层面来讲，Java中的Lambda可以被当做是匿名内部类的“语法糖”，但是二者在原理上是不同的。

## 1.4 格式

只要确保接口中有且仅有一个**抽象**方法即可：可以包含其他方法（默认，静态，私有）



由于接口当中抽象方法的 public abstract 是可以省略的，所以定义一个函数式接口很简单：

public interface MyfactionaInterface {  
 void myMethod();  
}

## 1.5 @FunctionalInterface注解

与 @Override 注解的作用类似，Java 8中专门为函数式接口引入了一个新的注解： @FunctionalInterface 。该注解可用于一个接口的定义上：

@FunctionalInterface  
public interface MyfactionaInterface {  
 void myMethod();  
}

@FunctionalInterface作用：

是函数式接口：编译成功；

非函数式接口：编译失败；

## 1.6 自定义函数式接口

*/\*\*  
 \* 函数是接口的使用：  
 \* 1.作为参数  
 \* 2.作为返回值  
 \*  
 \** ***@author*** *: lipu  
 \** ***@since*** *: 2020-08-04 20:20  
 \*/*public class Demo {  
  
 *//定义一个方法，参数使用函数是接口MyfunctionaInterface* public static void show(MyfunctionaInterface myInter){  
 myInter.myMethod();  
 }  
  
 public static void main(String[] args) {  
 *//传递其实现类  
 show*(new MyfunctionaInterfaceImpl());  
  
 *//直接传递匿名内部类  
 show*(new MyfunctionaInterface() {  
 @Override  
 public void myMethod() {  
 System.*out*.println("我是MyfunctionaInterface的匿名内部实现类");  
 }  
 });  
  
 *//使用Lambda表达式  
 show*(()->{  
 System.*out*.println("使用完整的Lambda表达式重写接口的方法");  
 });  
 *show*(()-> System.*out*.println("使用简化的Lambda表达式重写接口的方法"));  
 }  
}

# 第二章 函数式编程

## 2.1 Lambda的延迟特性

有些场景的代码执行后，结果不一定会被使用，从而造成性能浪费。而Lambda表达式是延迟执行的，这正好可以作为解决方案，提升性能。

### 性能浪费的日志案例

注:日志可以帮助我们快速的定位问题，记录程序运行过程中的情况，以便项目的监控和优化。

一种典型的场景就是对参数进行有条件使用，例如对日志消息进行拼接后，在满足条件的情况下进行打印输出：

public class LoggerDemo01 {  
  
 public static void showLog(int level, String message){  
 *//对日志级别进行判断，如果是1级别，name输出日志信息* if (level == 1) {  
 System.*out*.println(message);  
 }  
 }  
  
 public static void main(String[] args) {  
 *//定义三个日志信息* String msg1 = "Hello";  
 String msg2 = "World";  
 String msg3 = "Java";  
  
 *showLog*(1,msg1+msg2+msg3);  
 }  
}

这段代码存在问题：无论级别是否满足要求，作为 log 方法的第二个参数，三个字符串一定会首先被拼接并传入方法内，然后才会进行级别判断。如果级别不符合要求，那么字符串的拼接操作就白做了，存在性能浪费。

**备注：**SLF4J是应用非常广泛的日志框架，它在记录日志时为了解决这种性能浪费的问题，并不推荐首先进行字符串的拼接，而是将字符串的若干部分作为可变参数传入方法中，仅在日志级别满足要求的情况下才会进行字符串拼接。例如： LOGGER.debug("变量{}的取值为{}。", "os", "macOS") ，其中的大括号 {} 为占位符。如果满足日志级别要求，则会将“os”和“macOS”两个字符串依次拼接到大括号的位置；否则不会进行字符串拼接。这也是一种可行解决方案，但Lambda可以做到更好。

### 体验Lambda的更优写法

使用Lambda必然需要一个函数式接口：

@FunctionalInterface  
public interface MessageBuilder {  
  
 String builderMessage();  
}

然后对 log 方法进行改造：

public class LoggerLambdaDemo02 {  
  
 public static void showLog(int level, MessageBuilder message){  
 *//对日志级别进行判断，如果是1级别，name输出日志信息* if (level == 1) {  
 System.*out*.println(message.builderMessage());  
 }  
 }  
  
 public static void main(String[] args) {  
 *//定义三个日志信息* String msg1 = "Hello ";  
 String msg2 = "World ";  
 String msg3 = "Java";  
  
 *showLog*(1,()->msg1+msg2+msg3);

这样一来，只有当级别满足要求的时候，才会进行三个字符串的拼接；否则三个字符串将不会进行拼接。

### 证明Lambda的延迟

下面的代码可以通过结果进行验证：

public class LoggerDelayDemo03 {  
  
 public static void showLog(int level, MessageBuilder message){  
 *//对日志级别进行判断，如果是1级别，name输出日志信息* if (level == 1) {  
 System.*out*.println(message.builderMessage());  
 }  
 }  
  
 public static void main(String[] args) {  
 *//定义三个日志信息* String msg1 = "Hello ";  
 String msg2 = "World ";  
 String msg3 = "Java";  
  
 *showLog*(2,()->{  
 System.*out*.println("Lambda执行");return msg1+msg2+msg3;});  
 }  
}

从结果中可以看出，在不符合级别要求的情况下，Lambda将不会执行。从而达到节省性能的效果。

扩展：实际上使用内部类也可以达到同样的效果，只是将代码操作延迟到了另外一个对象当中通过调用方法来完成。而是否调用其所在方法是在条件判断之后才执行的。

## 2.2使用Lambda作为参数和返回值

参数

public class Demo01Runnable {  
  
 private static void startThread(Runnable task){  
 new Thread(task).start();  
 }  
  
 public static void main(String[] args) {  
 for (int i = 0; i < 10; i++) {  
 *startThread*(() -> System.*out*.println(Thread.*currentThread*().getName() + "线程执行"));  
 }  
 }  
}

返回值

public class Demo02Comparator {  
  
 private static Comparator<String> newComparator(){  
 return (a,b)->b.length() - a.length();*//谁在前，为true则在前面* }  
  
  
 public static void main(String[] args) {  
 String[] arr = {"abc","ab","abcd","a"};  
 System.*out*.println(Arrays.*toString*(arr));  
 Arrays.*sort*(arr,*newComparator*());  
 System.*out*.println(Arrays.*toString*(arr));  
 }  
}

# 第三章 常用的函数式接口

JDK提供了大量常用的函数式接口以丰富Lambda的典型使用场景，它们主要在 java.util.function 包中被提供。

## 3.1 Supplier接口

java.util.function.Supplier<T> 接口（生产型接口）仅包含一个无参的方法： T get() 。用来获取一个泛型参数指定类型的对象数据。由于这是一个函数式接口，这也就意味着对应的Lambda表达式需要“对外提供”一个符合泛型类型的对象数据。

public class Demo01Supplier {  
  
 private static String getString(Supplier<String> supplier){  
 return supplier.get();  
 }  
  
 public static void main(String[] args) {  
 String msg1 = "Hello ";  
 String msg2 = "World";  
  
 System.*out*.println(*getString*(()->msg1+msg2));  
 }  
}

求数组元素最大值

public class Demo02Test {  
  
 private static int getMax(Supplier<Integer> supplier){  
 return supplier.get();  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {2,234,45,6,7,88,954,34,5,20};  
  
 int maxNum = *getMax*(() -> {  
 int max = arr[0];  
 for (int i : arr) {  
 if (i > max) {  
 max = i;  
 }  
 }  
 return max;  
 });  
 System.*out*.println("最大值为："+maxNum);  
 }  
}

## 3.2 Consumer接口

java.util.function.Consumer<T> 接口（消费型接口）则正好与Supplier接口相反，它不是生产一个数据，而是消费一个数据，其数据类型由泛型决定。

### 抽象方法：accept

Consumer 接口中包含抽象方法 void accept(T t) ，意为消费一个指定泛型的数据。基本使用如：

当然，更好的写法是使用方法引用。

### 默认方法：andThen

如果一个方法的参数和返回值全都是 Consumer 类型，那么就可以实现效果：消费数据的时候，首先做一个操作，然后再做一个操作，实现组合。而这个方法就是 Consumer 接口中的default方法 andThen 。下面是JDK的源代码：

备注： java.util.Objects 的 requireNonNull 静态方法将会在参数为null时主动抛出

NullPointerException 异常。这省去了重复编写if语句和抛出空指针异常的麻烦。

要想实现组合，需要两个或多个Lambda表达式即可，而 andThen 的语义正是“一步接一步”操作。例如两个步骤组

合的情况：

## 3.1 Lambda

### 第二堂课

本节知识点：

05\_使用Lambda优化日志案例

06\_函数式接口作为方法的参数案例

07\_函数式接口作为方法的返回值类型案例

视频时间：

24分55秒

本节目标 ：

1. 理解Lambda的延迟执行
2. 独立编写函数式接口作为方法的参数案例
3. 独立编写函数式接口作为方法的返回值类型案例

#### 使用Lambda优化日志案例

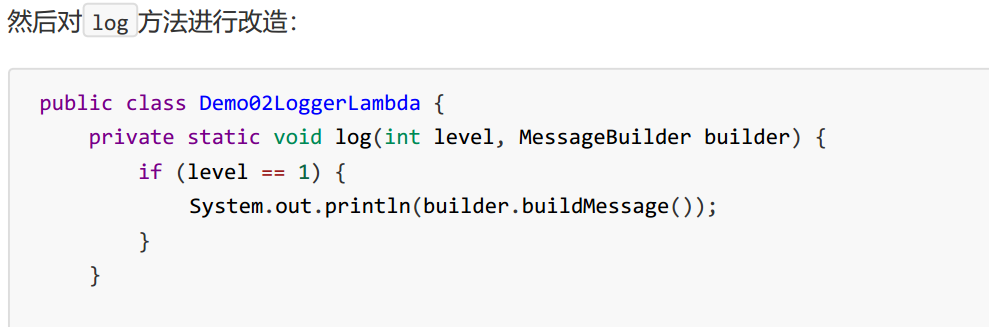
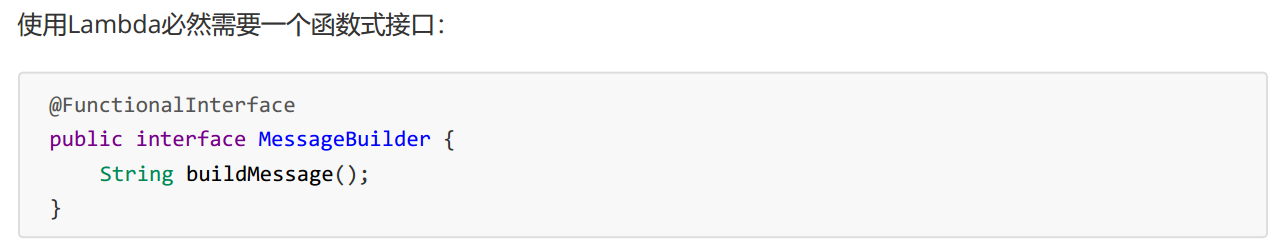
##### 2.1.1知识概述

* 使用Lambda优化日志案例

##### 2.1.2视频详情



##### 2.1.3总结与补充



##### 2.1.5习题答案

无

#### 函数式接口作为方法的参数案例

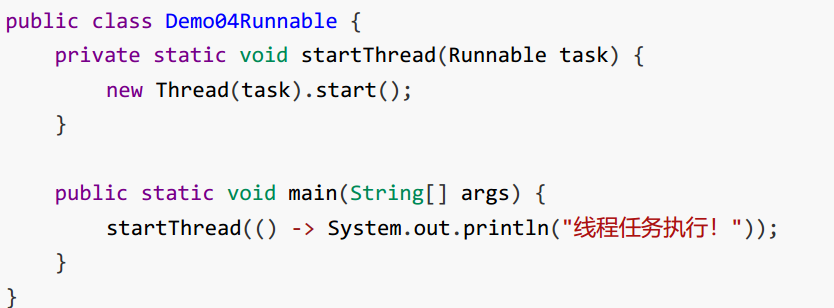
##### 2.2.1知识概述

* 使用Lambda开启线程-> Runnable

##### 2.2.2视频详情



##### 2.2.3总结与补充



##### 2.2.4课堂提问与练习

无

##### 2.2.5习题答案

无

#### 函数式接口作为方法的返回值类型案例

##### 2.3.1知识概述

* 函数式接口Comparator的使用

##### 2.3.2视频详情

##### 2.3.3总结与补充



##### 2.3.4课堂提问与练习

无

##### 2.3.5习题答案

无

### 第三堂课

本节知识点：

08\_常用的函数式接口\_Supplier接口

09\_常用的函数式接口\_Supplier接口练习\_求数组元素最大值

10\_常用的函数式接口\_Consumer接口

11\_常用的函数式接口\_Consumer接口的默认方法andThen

视频时间:

31分52秒

本节目标 ：

1. 理解生产型Supplier接口的作用
2. 独立编写使用Supplier求数组最大值案例
3. 理解消费型接口Consumer的作用
4. 掌握Consumer接口中andThen方法的使用

#### 常用的函数式接口\_Supplier接口

##### 3.1.1知识概述

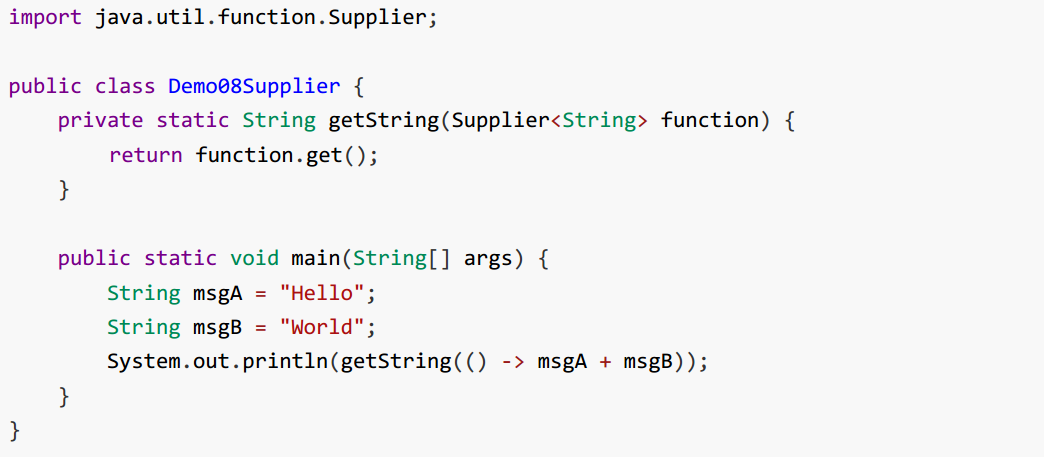
* 生产型接口Supplier

##### 3.1.2视频详情



##### 3.1.3总结与补充

java.util.function.Supplier<T> 接口仅包含一个无参的方法： T get() 。用来获取一个泛型参数指定类型的对象数据。由于这是一个函数式接口，这也就意味着对应的Lambda表达式需要“对外提供”一个符合泛型类型的对象数据。



##### 3.1.4课堂提问与练习

无

##### 3.1.5习题答案

无

#### 常用的函数式接口\_Supplier接口练习\_求数组元素最大值

##### 3.2.1知识概述

* 使用 Supplier 接口作为方法参数类型，通过Lambda表达式求出int数组中的最大值

##### 3.2.2视频详情



##### 3.2.3总结与补充



##### 3.2.4课堂提问与练习

无

##### 3.2.5习题答案

无

#### 常用的函数式接口\_Consumer接口

##### 3.3.1知识概述

* 消费型接口Consumer

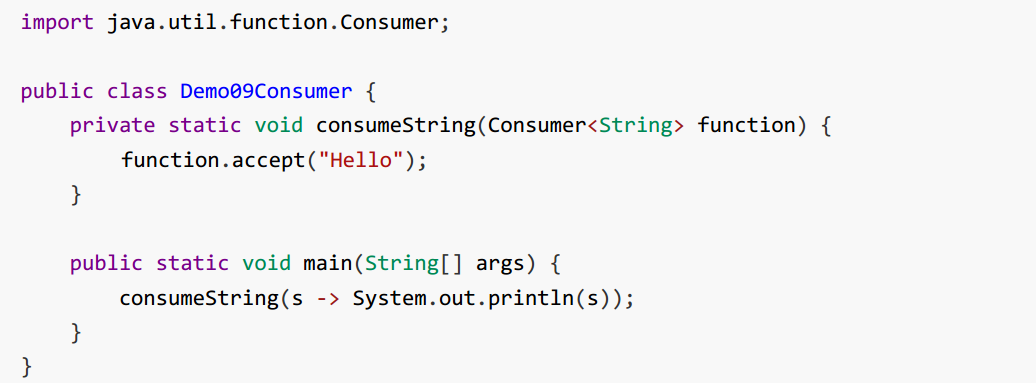
##### 3.3.2视频详情



##### 3.3.3总结与补充

java.util.function.Consumer<T> 接口则正好与Supplier接口相反，它不是生产一个数据，而是消费一个数据，其数据类型由泛型决定

Consumer 接口中包含抽象方法 void accept(T t) ，意为消费一个指定泛型的数据



##### 3.3.4课堂提问与练习

无

##### 3.3.5习题答案

无

#### 常用的函数式接口\_Consumer接口的默认方法andThen

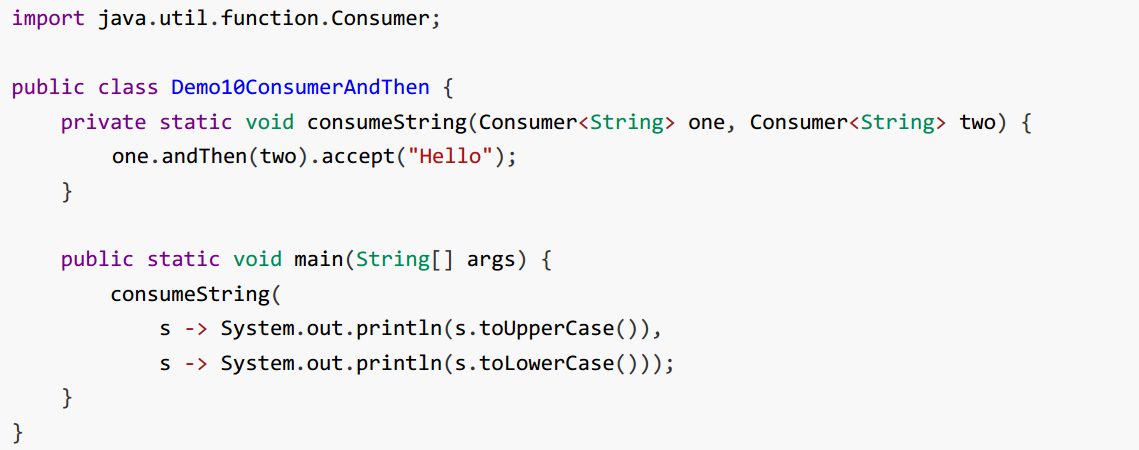
##### 3.3.1知识概述

* 如果一个方法的参数和返回值全都是 Consumer 类型，那么就可以实现效果：消费数据的时候，首先做一个操作，然后再做一个操作，实现组合

##### 3.3.2视频详情



##### 3.3.3总结与补充



##### 3.3.4课堂提问与练习

无

##### 3.3.5习题答案

无

### 第四堂课

本节知识点：

12\_常用的函数式接口\_Consumer接口练习\_字符串拼接输出

13\_常用的函数式接口\_Predicate接口

14\_常用的函数式接口\_Predicate接口\_默认方法and

15\_常用的函数式接口\_Predicate接口\_默认方法or&negate

视频时间：

36分40秒

本节目标 ：

独立编写Consumer接口完成字符串拼接案例

理解条件Predicate接口的作用

掌握Predicate中and方法的使用

掌握Predicate中or、negate方法的使用

#### 4.1 常用的函数式接口\_Consumer接口练习\_字符串拼接输出

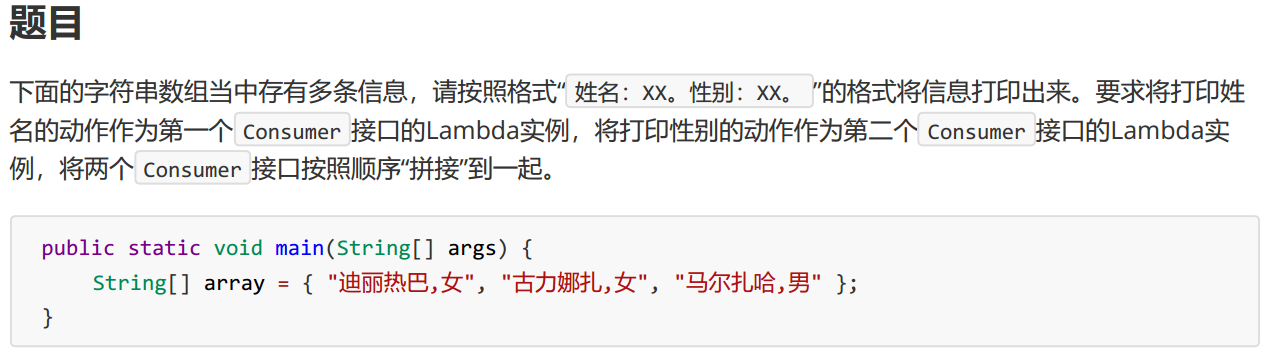
##### 4.1.1知识概述

* 常用的函数式接口\_Consumer接口练习\_字符串拼接输出

##### 4.1.2视频详情



##### 4.1.3总结与补充



##### 4.1.4课堂提问与练习

无

##### 4.1.5习题答案

无

#### 4.2 常用的函数式接口\_Predicate接口

##### 4.2.1知识概述

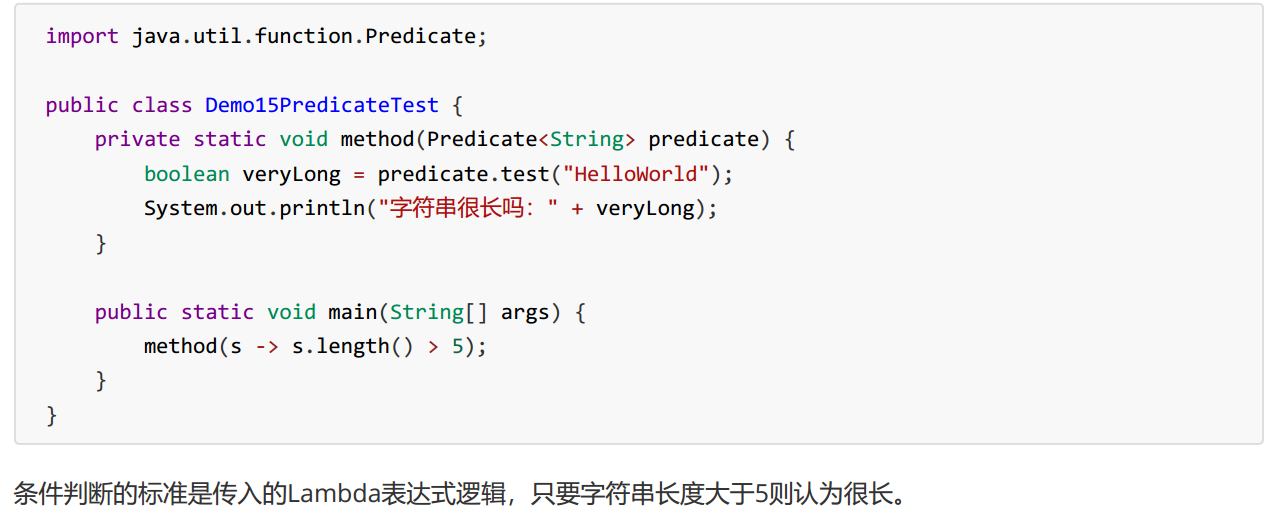
* 常用的函数式接口\_Predicate接口

##### 4.2.2视频详情



##### 4.2.3总结与补充

有时候我们需要对某种类型的数据进行判断，从而得到一个boolean值结果。这时可以使用java.util.function.Predicate<T> 接口



##### 4.2.4课堂提问与练习

无

##### 4.2.5习题答案

无

#### 4.3 常用的函数式接口\_Predicate接口\_默认方法and

##### 4.3.1知识概述

* 常用的函数式接口\_Predicate接口\_默认方法and

##### 4.3.2视频详情



##### 4.3.3总结与补充



##### 4.3.4课堂提问与练习

无

##### 4.3.5习题答案

无

#### 4.4 常用的函数式接口\_Predicate接口\_默认方法or&negate

##### 4.4.1知识概述

* Or : 或者
* negate : 非

##### 4.4.2视频详情



##### 4.4.3总结与补充

与 and 的“与”类似，默认方法 or 实现逻辑关系中的“或”

negate - > 非

##### 4.4.4课堂提问与练习

无

##### 4.4.5习题答案

无

### 第五堂课

本节知识点：

16\_常用的函数式接口\_Predicate接口练习\_集合信息筛选

17\_常用的函数式接口\_Function接口

18\_常用的函数式接口\_Function接口\_默认方法andThen

19\_常用的函数式接口\_Function接口练习\_自定义函数模型拼接

视频时间：

38分19秒

本节目标 ：

掌握Predicate集合信息筛选案例

理解转换型接口Function接口的作用

掌握Function接口andThen方法的使用

独立完成案例编写

#### 5.1 常用的函数式接口\_Predicate接口练习\_集合信息筛选

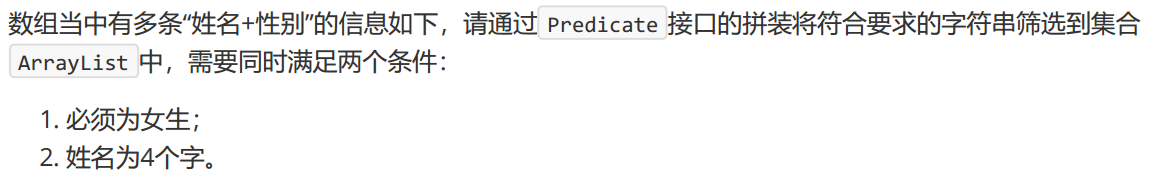
##### 5.1.1知识概述

* Predicate案例

##### 5.1.2视频详情



##### 5.1.3总结与补充



##### 5.1.4课堂提问与练习

无

##### 5.1.5习题答案

无

#### 5.2 常用的函数式接口\_Function接口

##### 5.2.1知识概述

* 转换型接口Function

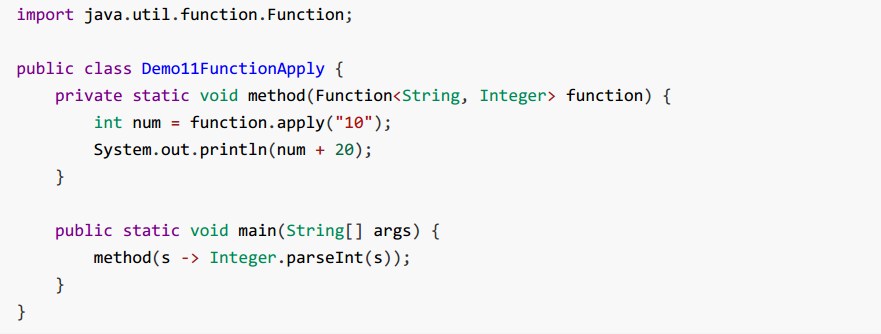
##### 5.2.2视频详情



##### 5.2.3总结与补充

java.util.function.Function<T,R> 接口用来根据一个类型的数据得到另一个类型的数据，前者称为前置条件，后者称为后置条件

Function 接口中最主要的抽象方法为： R apply(T t) ，根据类型T的参数获取类型R的结果。使用的场景例如：将 String 类型转换为 Integer 类型。



##### 5.2.4课堂提问与练习

无

##### 5.2.5习题答案

无

#### 5.3 常用的函数式接口\_Function接口\_默认方法andThen

##### 5.3.1知识概述

* Function 接口中有一个默认的 andThen 方法，用来进行组合操作。

##### 5.3.2视频详情



##### 5.3.3总结与补充



##### 5.3.4课堂提问与练习

无

##### 5.3.5习题答案

无

#### 5.4 常用的函数式接口\_Function接口练习\_自定义函数模型拼接

##### 5.3.1知识概述

* Function接口练习

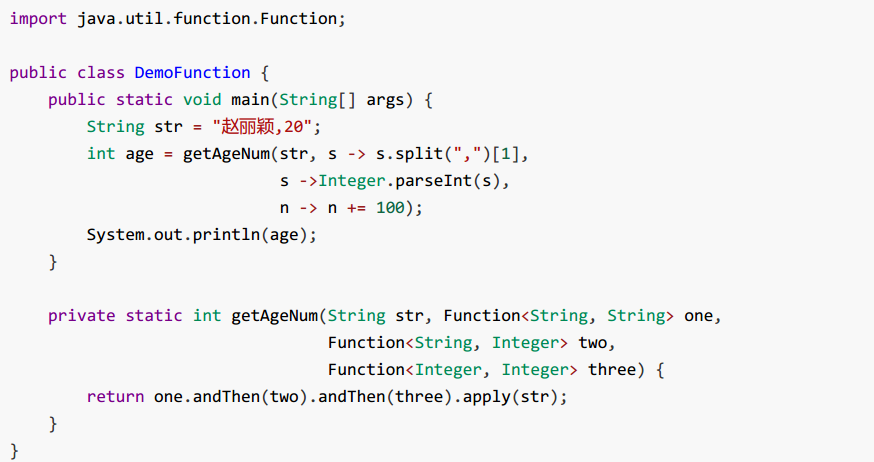
##### 5.3.2视频详情



##### 5.3.3总结与补充

请使用 Function 进行函数模型的拼接，按照顺序需要执行的多个函数操作为：  
 String str = "赵丽颖,20";

1. 将字符串截取数字年龄部分，得到字符串；  
 2. 将上一步的字符串转换成为int类型的数字；  
 3. 将上一步的int数字累加100，得到结果int数字



##### 5.3.4课堂提问与练习

无

##### 5.3.5习题答案

无