## 1. 实现Runnable线程案例

使用() -> {} 替代匿名类：

//Before Java 8:

**new** **Thread**(**new** Runnable() {

**@Override**

**public** **void** **run**() {

System.out.println("Before Java8 ");

}

}).start();

//Java 8 way:

**new** **Thread**( () -> System.out.println("In Java8!") ).start();

**Output:**

too much code, **for** too little to **do**

Lambda expression rocks !!

你可以使用 下面语法实现Lambda:

(params) -> expression  
(params) -> statement  
(params) -> { statements }

如果你的方法并不改变任何方法参数，比如只是输出，那么可以简写如下：

() -> System.out.println("Hello Lambda Expressions");

如果你的方法接受两个方法参数，如下：

(int even, int odd) -> even + odd

## 2.实现事件处理

如果你曾经做过Swing 编程，你将永远不会忘记编写事件侦听器代码。使用lambda表达式如下所示写出更好的事件侦听器的代码。

// Before Java 8:

JButton show = **new** JButton("Show");

show.addActionListener(**new** ActionListener() {

**@Override**

**public** **void** **actionPerformed**(ActionEvent e) {

System.out.println("without lambda expression is boring");

}

});

// Java 8 way:

show.addActionListener((e) -> {

System.out.println("Action !! Lambda expressions Rocks");

});

在java 8中你可以使用Lambda表达式替代丑陋的匿名类。

## 3.使用Lambda表达式遍历List集合

//Prior Java 8 :

**List** features **=** **Arrays**.asList("Lambdas", "Default Method",   
"Stream API", "Date and Time API");

**for** (**String** feature **:** **features**) {

**System**.out.println(feature);

}

//In Java 8:

**List** features **=** **Arrays**.asList("Lambdas", "Default Method", "Stream API",  
 "Date and Time API");

features.forEach(n -> **System**.out.println(n));

// Even better use Method reference feature of Java 8

// method reference is denoted by :: (double colon) operator

// looks similar to score resolution operator of C++

features.forEach(**System**.out::println);

**Output:**

**Lambdas**

**Default** **Method**

**Stream** **API**

**Date** and **Time** **API**

方法引用是使用两个冒号::这个操作符号。

## 4.使用Lambda表达式和函数接口

为了支持函数编程，Java 8加入了一个新的包java.util.function，其中有一个接口java.util.function.Predicate是支持Lambda函数编程：

**public** **static** **void** **main**(args[]){

List languages = Arrays.asList("Java", "Scala", "C++", "Haskell", "Lisp");

System.out.println("Languages which starts with J :");

filter(languages, (str)->str.startsWith("J"));

System.out.println("Languages which ends with a ");

filter(languages, (str)->str.endsWith("a"));

System.out.println("Print all languages :");

filter(languages, (str)->**true**);

System.out.println("Print no language : ");

filter(languages, (str)->**false**);

System.out.println("Print language whose length greater than 4:");

filter(languages, (str)->str.length() > **4**);

}

**public** **static** **void** **filter**(List names, Predicate condition) {

**for**(String **name:** names) {

**if**(condition.test(name)) {

System.out.println(name + " ");

}

}

}

}

**Output:**

Languages which starts with J :

Java

Languages which ends with a

Java

Scala

Print all languages :

Java

Scala

C++

Haskell

Lisp

Print no language :

Print language whose length greater than **4**:

Scala

Haskell

//Even better

**public** **static** **void** **filter**(List names, Predicate condition) {

names.stream().filter((name) -> (condition.test(name)))  
 .forEach((name) -> {System.out.println(name + " ");

});

}

你能看到来自Stream API 的filter方法能够接受 Predicate参数, 能够允许测试多个条件。

## 5.复杂的结合Predicate 使用

java.util.function.Predicate提供and(), or() 和 xor()可以进行逻辑操作，比如为了得到一串字符串中以"J"开头的4个长度：

// We can even combine Predicate using and(), or() And xor() logical functions

// for example to find names, which starts with J and four letters long, you

// can pass combination of two Predicate

**Predicate**<**String**> startsWithJ **=** (n) -> n.startsWith("J");

**Predicate**<**String**> fourLetterLong **=** (n) -> n.length() == **4**;

names.stream()

.filter(startsWithJ.***and***(fourLetterLong))

.forEach((n) -> **System**.out.print("\nName, which starts with  
 'J' and four letter long is : " + n));

其中startsWithJ.***and***(fourLetterLong)是使用了AND逻辑操作。

## 6.使用Lambda实现Map 和 Reduce

最流行的函数编程概念是map，它允许你改变你的对象，在这个案例中，我们将costBeforeTeax集合中每个元素改变了增加一定的数值，我们将Lambda表达式 x -> x\*x传送map()方法，这将应用到stream中所有元素。然后我们使用 forEach() 打印出这个集合的元素.

// applying 12% VAT on each purchase

// Without lambda expressions:

**List** costBeforeTax **=** **Arrays**.asList(**100**, **200**, **300**, **400**, **500**);

**for** (**Integer** cost **:** **costBeforeTax**) {

double price **=** cost + .**12**\*cost;

**System**.out.println(price);

}

// With Lambda expression:

**List** costBeforeTax **=** **Arrays**.asList(**100**, **200**, **300**, **400**, **500**);

costBeforeTax.stream().map((cost) -> cost + .**12**\*cost)  
 .forEach(**System**.out::println);

**Output**

**112.0**

**224.0**

**336.0**

**448.0**

**560.0**

**112.0**

**224.0**

**336.0**

**448.0**

**560.0**

reduce() 是将集合中所有值结合进一个，Reduce类似SQL语句中的sum(), avg() 或count(),

// Applying 12% VAT on each purchase

// Old way:

**List** costBeforeTax **=** **Arrays**.asList(**100**, **200**, **300**, **400**, **500**);

double total **=** **0**;

**for** (**Integer** cost **:** **costBeforeTax**) {

double price **=** cost + .**12**\*cost;

total **=** total + price;

}

**System**.out.println("Total : " + total);

// New way:

**List** costBeforeTax **=** **Arrays**.asList(**100**, **200**, **300**, **400**, **500**);

double bill **=** costBeforeTax.stream().map((cost) -> cost + .**12**\*cost)  
 .reduce((sum, cost) -> sum + cost)  
 .get();

**System**.out.println("Total : " + bill);

**Output**

**Total** **:** 1680**.**0

**Total** **:** 1680**.**0

## 7.通过filtering 创建一个字符串String的集合

Filtering是对大型Collection操作的一个通用操作，Stream提供filter()方法，接受一个Predicate对象，意味着你能传送lambda表达式作为一个过滤逻辑进入这个方法：

// Create a List with String more than 2 characters

**List**<**String**> filtered **=** strList.stream().filter(x -> x.length()> **2**)  
 .collect(**Collectors**.toList());

**System**.out.printf("Original List : %s, filtered list : %s %n",   
 strList, filtered);

**Output** **:**

**Original** **List** **:** [**abc**, , **bcd**, , **defg**, **jk**], filtered list **:** [**abc**, bcd, defg]

## 8.对集合中每个元素应用函数

我们经常需要对集合中元素运用一定的功能，如表中的每个元素乘以或除以一个值等等.

// Convert String to Uppercase and join them using coma

**List**<**String**> G7 **=** **Arrays**.asList("USA", "Japan", "France", "Germany",

"Italy", "U.K.","Canada");

**String** G7Countries **=** G7.stream().map(x -> x.toUpperCase())  
 .collect(**Collectors**.joining(", "));

**System**.out.println(G7Countries);

**Output** **:**

**USA**, **JAPAN**, **FRANCE**, **GERMANY**, **ITALY**, U.K., **CANADA**

上面是将字符串转换为大写，然后使用逗号串起来。

## 9.通过复制不同的值创建一个子列表

使用Stream的distinct()方法过滤集合中重复元素。

// Create List of square of all distinct numbers

**List**<**Integer**> numbers **=** **Arrays**.asList(**9**, **10**, **3**, **4**, **7**, **3**, **4**);

**List**<**Integer**> distinct **=** numbers.stream().map( i -> i\*i).distinct()  
 .collect(**Collectors**.toList());

**System**.out.printf("Original List : %s, Square Without duplicates :  
 %s %n", numbers, distinct);

**Output** **:**

**Original** **List** **:** [9, 10, 3, 4, 7, 3, 4], **Square** **Without**   
 duplicates **:** [81, **100**, **9**, **16**, **49**]

## 10.计算List中的元素的最大值，最小值，总和及平均值

//Get count, min, max, sum, and average for numbers

**List**<**Integer**> primes **=** **Arrays**.asList(**2**, **3**, **5**, **7**, **11**, **13**, **17**, **19**, **23**, **29**);

**IntSummaryStatistics** stats **=** primes.stream().mapToInt((x) -> x)  
 .summaryStatistics();

**System**.out.println("Highest prime number in List : " + stats.getMax());

**System**.out.println("Lowest prime number in List : " + stats.getMin());

**System**.out.println("Sum of all prime numbers : " + stats.getSum());

**System**.out.println("Average of all prime numbers : " + stats.getAverage());

**Output** **:**

**Highest** **prime** **number** **in** **List** **:** 29

**Lowest** **prime** **number** **in** **List** **:** 2

**Sum** **of** **all** **prime** **numbers** **:** 129

**Average** **of** **all** **prime** **numbers** **:** 12**.**9

Lambda语法篇：http://www.cnblogs.com/figure9/archive/2014/10/24/4048421.html