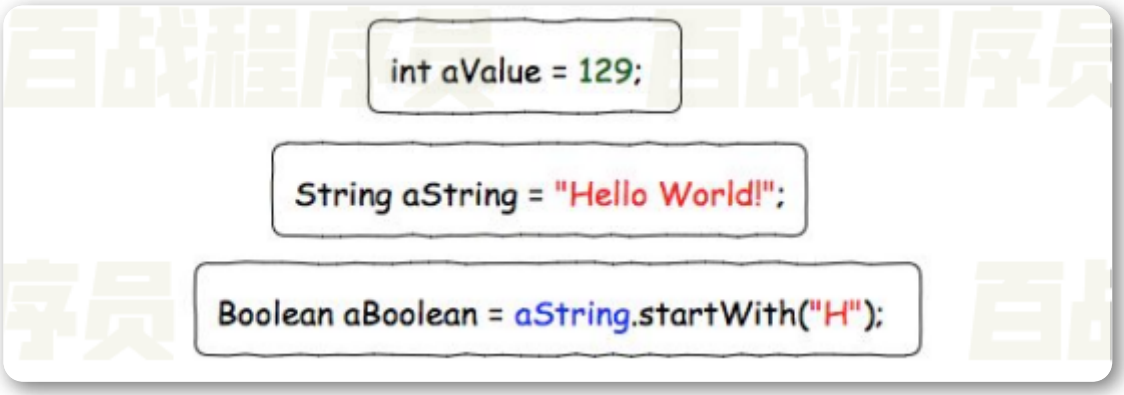
Lambda表达式介绍

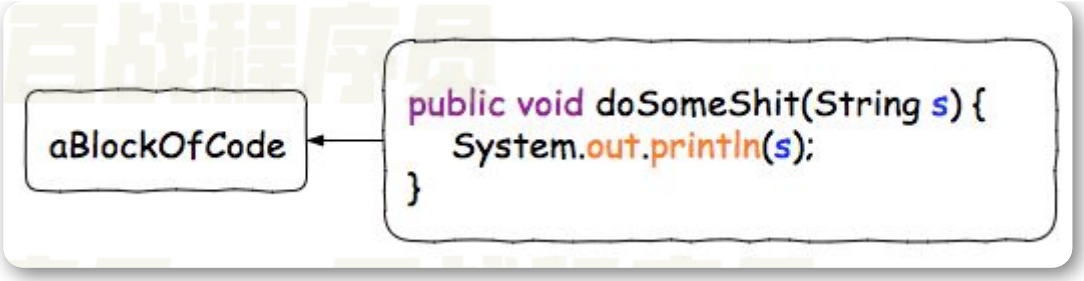
Lambda简介

Lambda 表达式是 JDK8 的一个新特性，可以取代大部分的匿名内部类，写出更优雅的 Java 代码，尤其在集合的遍历和其他集合操作中，可以极大地优化代码结构。

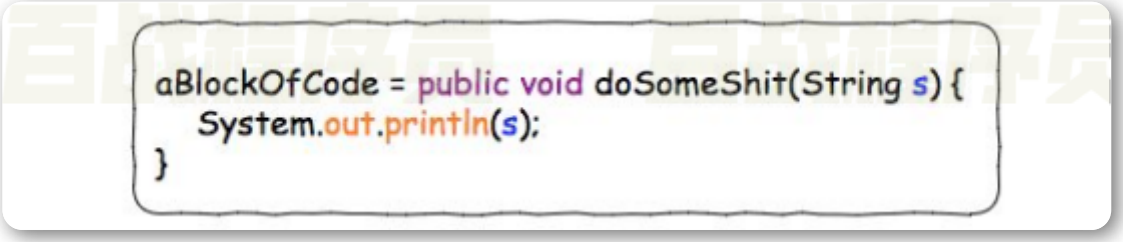
在Java语言中，可以为变量赋予一个值:



能否把一个代码块赋给一变量吗?

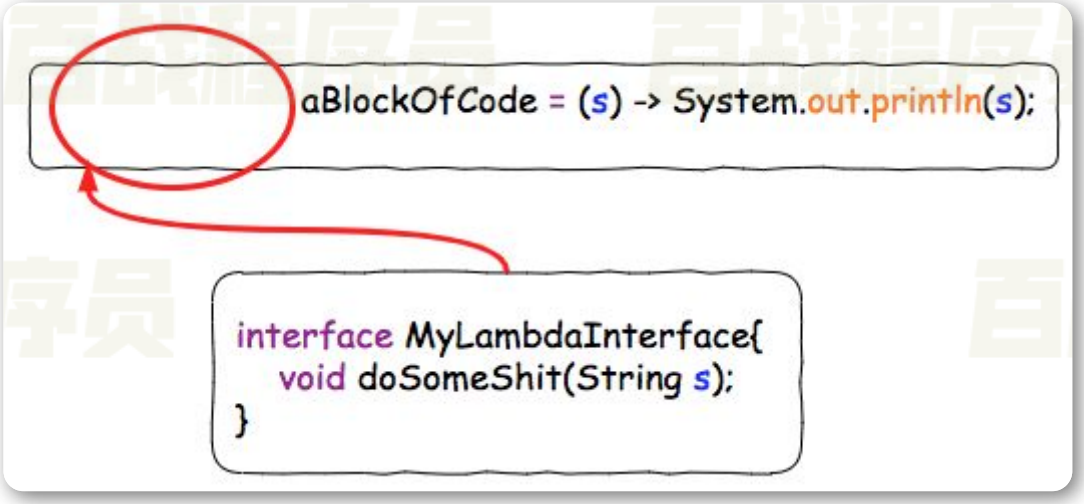


在Java 8之前，这个是做不到的。但是Java 8问世之后，利用Lambda特性，就可以做到了。

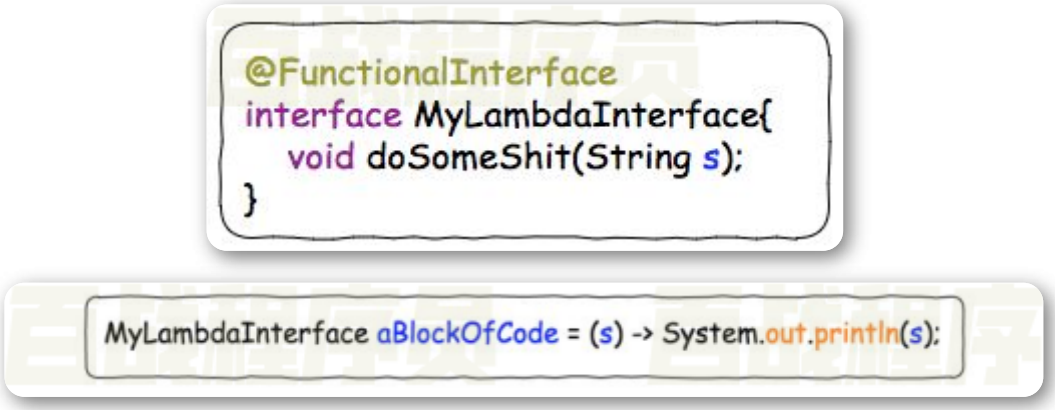


甚至我们可以让语法变得更简洁。

在Java 8里面，所有的Lambda的类型都是一个接口，而Lambda表达式本身，也就是”那段代码“，需要是这个接口的实现。这是我认为理解Lambda的一个关键所在，简而言之就是，Lambda表达式本身就是一个接口的实现。直接这样说可能还是有点让人困扰，我们继续看看例子。我们给上面的aBlockOfCode加上一个类型：

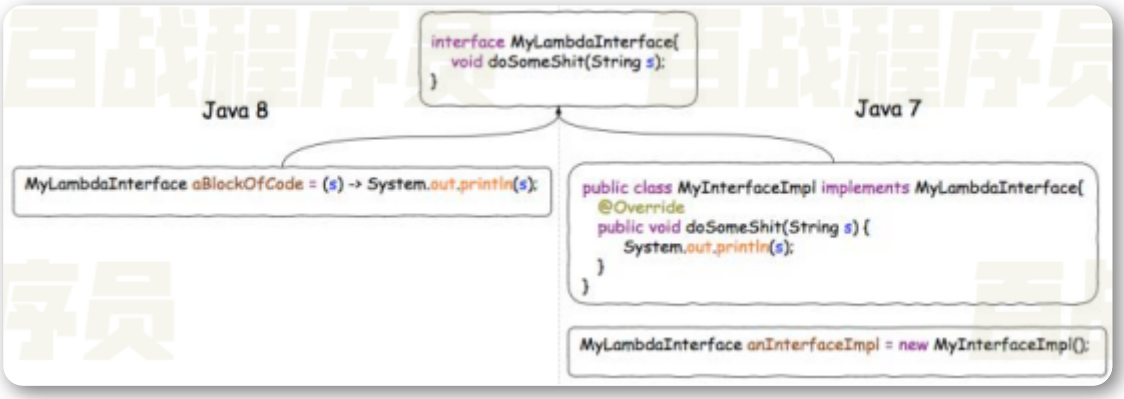


这种只有一个接口函数需要被实现的接口类型，我们叫它”函数式接口“。为了避免后来的人在这个接口中增加接口函数导致其有多个接口函数需要被实现，变成"非函数接口”，我们可以在这个上面加上一个声明@FunctionalInterface, 这样别人就无法在里面添加新的接口函数了。



Lambda作用

最直观的作用就是使得代码变得异常简洁。



接口要求

虽然使用 Lambda 表达式可以对某些接口进行简单的实现，但并不是所有的接口都可以使用 Lambda 表达式来实现。Lambda 规定接口中只能有一个需要被实现的方法，不是规定接口中只能有一个方法。

jdk 8 中有另一个新特性：default， 被 default 修饰的方法会有默认实现，不是必须被实现的方法，所以不影响 Lambda 表达式的使用。

@FunctionalInterface注解作用

@FunctionalInterface标记在接口上，“函数式接口”是指仅仅只包含一个抽象方法的接口。

实时效果反馈

1.Lambda 表达式是哪个版本JDK的新特性?

A JDK5.0;

B JDK6.0;

C JDK7.0;

D JDK8.0;

2.Lambda 表达式可以对哪些接口进行简单的实现?

A 接口中可以包含多个抽象方法;

B 接口中只能含有一个方法;

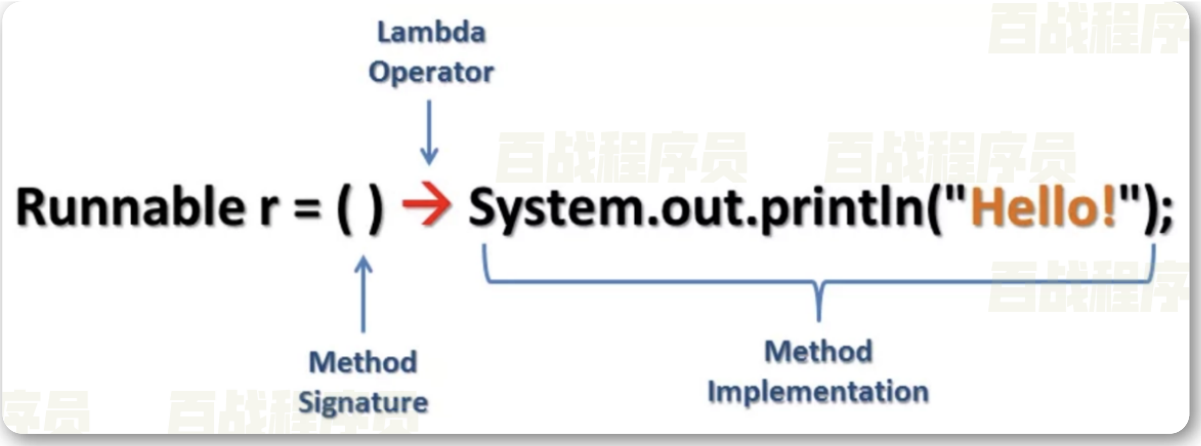
C 接口中只能含有一个抽象方法;

D 以上都可以;

答案

1=>D 2=>C

Lambda表达式语法



语法结构

```
(parameters) -> expression
或
(parameters) ->{ statements;}
```

语法形式为 () -> {}：

() 用来描述参数列表，如果有多个参数，参数之间用逗号隔开，如果没有参数，留空即可；

-> 读作(goes to)，为 lambda运算符 ，固定写法，代表指向动作；

{} 代码块，具体要做的事情，也就是方法体内容；

Lambda表达式的重要特征

可选类型声明：不需要声明参数类型，编译器可以统一识别参数值。

可选的参数圆括号：一个参数无需定义圆括号，但多个参数需要定义圆括号。

可选的大括号：如果主体包含了一个语句，就不需要使用大括号。

可选的返回关键字：如果主体只有一个表达式返回值则编译器会自动返回值，大括号需要指定明表达式返回了一个数值。

Lambda案例

```
// 1. 不需要参数,返回值为 5
() -> 5

// 2. 接收一个参数(数字类型),返回其2倍的值
x -> 2 * x

// 3. 接受2个参数(数字),并返回他们的差值
(x, y) -> x - y

// 4. 接收2个int型整数,返回他们的和
(int x, int y) -> x + y

// 5. 接受一个 string 对象,并在控制台打印,不返回任何值(看起来像是返回void)
(String s) -> System.out.print(s)
```

实时效果反馈

1.如下哪个符号不是Lambda 表达式中的符号?

A {};

B ();

C [];

D ->;

答案

1=>C

Lambda表达式入门案例

定义函数接口

```
/**
 * 无返回值，无参数
 */
@FunctionalInterface
interface NoReturnNoParam{
    void method();
}

/**
 * 无返回值，有一个参数
 */
@FunctionalInterface
interface NoReturnOneParam{
    void method(int a);
}

/**
 * 无返回值，有多个参数
 */
@FunctionalInterface
interface NoReturnMultiParam{
    void method(int a,int b);
}

/**
 * 有返回值，无参数
```

```
 */
@FunctionalInterface
interface ReturnNoParam{
    int method();
}

/**
 * 有返回值，有一个参数
 */
@FunctionalInterface
interface ReturnOneParam{
    int method(int a);
}

/**
 * 有返回值，有多个参数
 */
@FunctionalInterface
interface ReturnMultiParam{
    int method(int a,int b);
}
```

实现函数接口

```
public static void main(String[] args) {
    /**
     * 无返回值，无参数
     */
    NoReturnNoParam noReturnNoParam = ()->{
        System.out.println("NoReturnNoParam");
```

```
    };
    noReturnNoParam.method();

    /**
     * 无返回值，有一个参数
     */
    NoReturnOneParam noReturnOneParam = (int
a)->{
        System.out.println("NoReturnOneParam
"+a);
    };
     noReturnOneParam.method(10);

    /**
     * 无返回值，有多个参数
     */
    NoReturnMultiParam noReturnMultiParam =
(int a, int b)->{
        System.out.println("NoReturnMultiParam
"+a+"\t"+b);
    };
    noReturnMultiParam.method(10,20);

    /**
     * 有返回值，无参数
     */
    ReturnNoParam returnNoParam = ()->{
        System.out.print("ReturnNoParam ");
        return 10;
    };
    System.out.println(returnNoParam.method());
```

```
    /**
     * 有返回值，有一个参数
     */
    ReturnOneParam returnOneParam = (int a)->{
        System.out.print("ReturnOneParam ");
        return a;
    };

 System.out.println(returnOneParam.method(10));

    /**
     * 有返回值，有多个参数
     */
    ReturnMultiParam returnMultiParam = (int a
,int b)->{
        System.out.print("ReturnMultiParam ");
        return a+b;
    };

 System.out.println(returnMultiParam.method(10,
20));

}
```

Lambda语法简化

```
/**
     * 无返回值，无参数
     */
   /* NoReturnNoParam noReturnNoParam = ()->{
```

```
        System.out.println("NoReturnNoParam");
    };*/
    /**
     * 简化版
     */
    NoReturnNoParam noReturnNoParam = ()->
System.out.println("NoReturnNoParam");
    noReturnNoParam.method();

    /**
     * 无返回值，有一个参数
     */
   /* NoReturnOneParam noReturnOneParam = (int
a)->{
        System.out.println("NoReturnOneParam
"+a);
    };*/
    /**
     * 简化版
     */
    NoReturnOneParam noReturnOneParam = a ->
System.out.println("NoReturnOneParam "+a);
     noReturnOneParam.method(10);

    /**
     * 无返回值，有多个参数
     */
   /* NoReturnMultiParam noReturnMultiParam =
(int a, int b)->{
        System.out.println("NoReturnMultiParam
"+a+"\t"+b);
    };*/
```

```
    NoReturnMultiParam noReturnMultiParam =
(a,b)-> System.out.println("NoReturnMultiParam
"+a+"\t"+b);
    noReturnMultiParam.method(10,20);

    /**
     * 有返回值，无参数
     */
   /* ReturnNoParam returnNoParam = ()->{
        System.out.print("ReturnNoParam ");
        return 10;
    };*/
    /**
     * 简化版
     */
    ReturnNoParam returnNoParam = ()->10+20;
    System.out.println(returnNoParam.method());

    /**
     * 有返回值，有一个参数
     */
  /*  ReturnOneParam returnOneParam = (int a)->
{
        System.out.print("ReturnOneParam ");
        return a;
    };*/
    /***
     * 简化版
     */
    ReturnOneParam returnOneParam = a->a;

 System.out.println(returnOneParam.method(10));
```

```
    /**
     * 有返回值，有多个参数
     */
    /*ReturnMultiParam returnMultiParam = (int
a ,int b)->{
        System.out.print("ReturnMultiParam ");
        return a+b;
    };*/
    /**
     * 简化版
     */
    ReturnMultiParam returnMultiParam = (a ,b)-
>a+b;

 System.out.println(returnMultiParam.method(10,
20));

}
```

Lambda表达式的使用

Lambda表达式引用方法

有时候我们不是必须使用Lambda的函数体定义实现，我们可以利用 lambda表达式指向一个已经存在的方法作为抽象方法的实现。

要求

- 参数的个数以及类型需要与函数接口中的抽象方法一致。

- 返回值类型要与函数接口中的抽象方法的返回值类型一致。

语法

方法归属者::方法名 静态方法的归属者为类名，非静态方法归属者为该对象的引用。

案例

```
/**
 * 无返回值，无参数
 */
@FunctionalInterface
interface NoReturnNoParam{
    void method();
}

/**
 * 无返回值，有一个参数
 */
@FunctionalInterface
interface NoReturnOneParam{
    void method(int a);
}

/**
 * 无返回值，有多个参数
 */
@FunctionalInterface
interface NoReturnMultiParam{
    void method(int a,int b);
}
```

```
/**
 * 有返回值，无参数
 */
@FunctionalInterface
interface ReturnNoParam{
    int method();
}

/**
 * 有返回值，有一个参数
 */
@FunctionalInterface
interface ReturnOneParam{
    int method(int a);
}

/**
 * 有返回值，有多个参数
 */
@FunctionalInterface
interface ReturnMultiParam{
    int method(int a,int b);
}
public class Test {
    public static void main(String[] args) {
        /**
         * 无返回值，无参数
         */
       /* NoReturnNoParam noReturnNoParam = ()-
>{

 System.out.println("NoReturnNoParam");
```

```
        };*/
        /**
         * 简化版
         */
        NoReturnNoParam noReturnNoParam = ()->
System.out.println("NoReturnNoParam");
        noReturnNoParam.method();

        /**
         * 无返回值，有一个参数
         */
       /* NoReturnOneParam noReturnOneParam =
(int a)->{

 System.out.println("NoReturnOneParam "+a);
        };*/
        /**
         * 简化版
         */
        NoReturnOneParam noReturnOneParam = a -
> System.out.println("NoReturnOneParam "+a);
         noReturnOneParam.method(10);

        /**
         * 无返回值，有多个参数
         */
       /* NoReturnMultiParam noReturnMultiParam
= (int a, int b)->{

 System.out.println("NoReturnMultiParam
"+a+"\t"+b);
        };*/
```

```
        NoReturnMultiParam noReturnMultiParam =
(a,b)-> System.out.println("NoReturnMultiParam
"+a+"\t"+b);
        noReturnMultiParam.method(10,20);

        /**
         * 有返回值，无参数
         */
       /* ReturnNoParam returnNoParam = ()->{
            System.out.print("ReturnNoParam ");
            return 10;
        };*/
        /**
         * 简化版
         */
        ReturnNoParam returnNoParam = ()-
>10+20;

 System.out.println(returnNoParam.method());

        /**
         * 有返回值，有一个参数
         */
      /*  ReturnOneParam returnOneParam = (int
a)->{
            System.out.print("ReturnOneParam
");
            return a;
        };*/
        /***
         * 简化版
         */
```

```
        ReturnOneParam returnOneParam = a->a;

 System.out.println(returnOneParam.method(10));

        /**
         * 有返回值，有多个参数
         */
        /*ReturnMultiParam returnMultiParam =
(int a ,int b)->{
            System.out.print("ReturnMultiParam
");
            return a+b;
        };*/
        /**
         * 简化版
         */
        ReturnMultiParam returnMultiParam = (a
,b)->a+b;

 System.out.println(returnMultiParam.method(10,
20));

    }
    /**
     * 要求:
     * 1,参数的个数以及类型需要与函数接口中的抽象方法一
致。
     * 2,返回值类型要与函数接口中的抽象方法的返回值类型
一致。
     * @param a
     * @return
     */
```

```
    public static int doubleNum(int a){
        return 2*a;
    }
    public int addTwo(int a){
        return a+2;
    }
}
```

```
public class Test2 {
    public static void main(String[] args) {
        ReturnOneParam returnOneParam =
Test::doubleNum;
        int value = returnOneParam.method(10);
        System.out.println(value);

        Test test = new Test();
        ReturnOneParam returnOneParam1 =
test::addTwo;
        int value2 =
returnOneParam1.method(10);
        System.out.println(value2);
    }
}
```

Lambda表达式创建线程

```
public class Test3 {
    public static void main(String[] args) {
```

```
 System.out.println(Thread.currentThread().getN
ame()+" 开始");
        new Thread(()->{
            for(int i=0;i<20;i++){
                try {
                    Thread.sleep(500);
                } catch (InterruptedException
e) {
                    e.printStackTrace();
                }

 System.out.println(Thread.currentThread().getN
ame()+" "+i);
            }
        },"Lambda Thread ").start();

 System.out.println(Thread.currentThread().getN
ame()+" 结束");
    }
}
```

Lambda 表达式中的闭包问题

什么是闭包

闭包的本质就是代码片断。所以闭包可以理解成一个代码片断的引用。在Java中匿名内部类也是闭包的一种实现方式。

在闭包中访问外部的变量时，外部变量必须是final类型，虚拟机会帮我们加上 final 修饰关键字。

```
public class Test4 {
    public static void main(String[] args) {
        final int num  =10;
        NoReturnNoParam noReturnNoParam = ()-
>System.out.println(num);
        noReturnNoParam.method();
    }
}
```

常用的函数接口

Consumer接口的使用

Consumer 接口是JDK为我们提供的一个函数式接口，该接口也被称为消费型接口。

遍历集合

我们可以调用集合的 public void forEach(Consumer<? super E> action) 方法，通过 lambda 表达式的方式遍历集合中的元素。以下是 Consumer 接口的方法以及遍历集合的操作。

```
public class Test4 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
        list.add("d");
        list.forEach(System.out::println);
    }
}
```

Predicate接口的使用

Predicate 是 JDK 为我们提供的一个函数式接口，可以简化程序的编写。

删除集合中的元素

我们通过public boolean removeIf(Predicate<? super E> filter)方法来删除集合中的某个元素，

```
public class Test5 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("a");
        list.add("b");
        list.add("c");
        list.add("d");
        list.removeIf(ele->ele.equals("b"));
        list.forEach(System.out::println);
    }
}
```

Comparator接口的使用

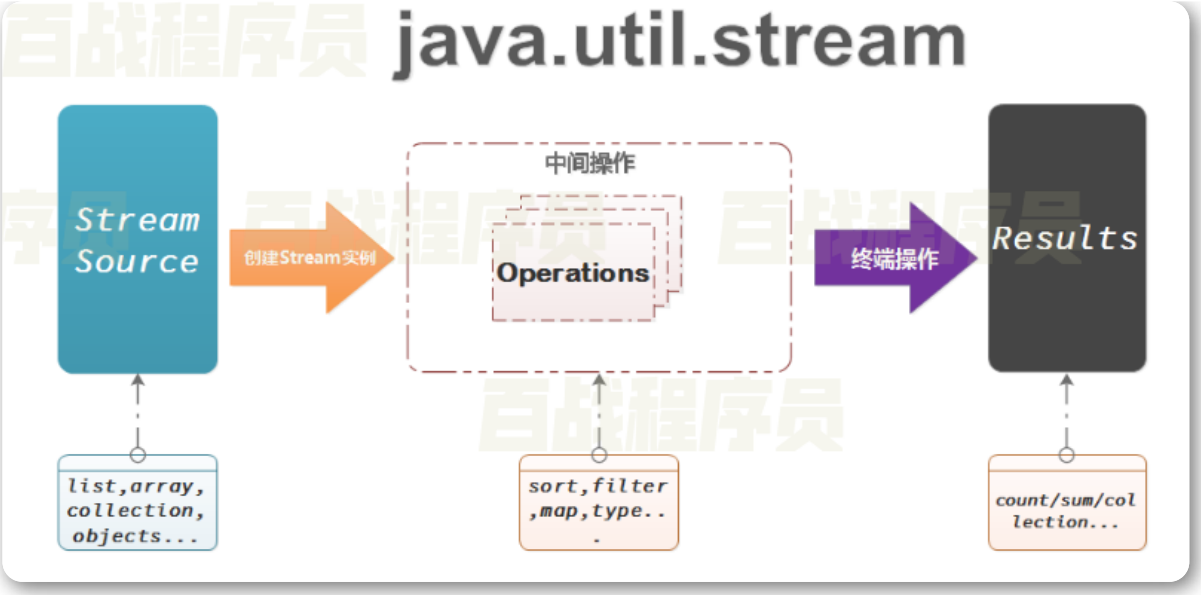
Comparator是 JDK 为我们提供的一个函数式接口，该接口为比较器接口。

元素排序

之前我们若要为集合内的元素排序，就必须调用 sort 方法，传入比较器重写 compare 方法的比较器对象，现在我们还可以使用 lambda 表达式来简化代码。

```
public class Test7 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("a");
        list.add("d");
        list.add("b");
        list.add("c");
        list.sort((o1,o2)->o1.compareTo(o2));
        list.forEach(System.out::println);
    }
}
```

Stream流介绍



Stream流简介

Stream是数据渠道，用于操作数据源所生成的元素序列，它可以实现对集合的复杂操作，例如过滤、排序和映射等。Stream不会改变源对象，而是返回一个新的结果集。

Stream流的生成方式

- 生成流：通过数据源（集合、数组等）创建一个流。
- 中间操作：一个流后面可以跟随零个或者多个中间操作，其目的主要是打开流，做出某种程度的数据过滤/映射，然后返回一个新的流，交给下一个操作使用。
- 终结操作：一旦执行终止操作，就执行中间的链式操作，并产生结果。

实时效果反馈

1.Stream流对象的作用是?

A 操作线程;

B 操作集合;

C 操作网络;

D 操作文件;

答案

1=>B

Stream流的常见方法

数据过滤

```
public class Test8 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("oldlu");
        list.add("oldlin");
        list.add("kevin");
        list.add("peter");
        //多条件and关系
        list.stream().filter(ele ->
ele.startsWith("o")).filter(ele-
>ele.endsWith("n")).collect(Collectors.toList()
).forEach(System.out::println);
        System.out.println("------------------
-----");

        //多条件or关系
        Predicate<String> predicate1 = ele ->
ele.startsWith("o");
        Predicate<String> predicate2 = ele-
>ele.endsWith("n");
```

```
 list.stream().filter(predicate1.or(predicate2)
).collect(Collectors.toList()).forEach(System.o
ut::println);
    }
}
```

数量限制

```
public class Test9 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("oldlu");
        list.add("oldlin");
        list.add("kevin");
        list.add("peter");
        //limit

 list.stream().limit(2).forEach(System.out::pri
ntln);
    }
}
```

元素排序

```
public class Test10 {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("b");
        list.add("c");
        list.add("a");
        list.add("d");
        //升序排序

 list.stream().sorted(Comparator.naturalOrder()
).forEach(System.out::println);
        System.out.println("---------------");
        //降序排序

 list.stream().sorted(Comparator.reverseOrder()
).forEach(System.out::println);
    }
}
```