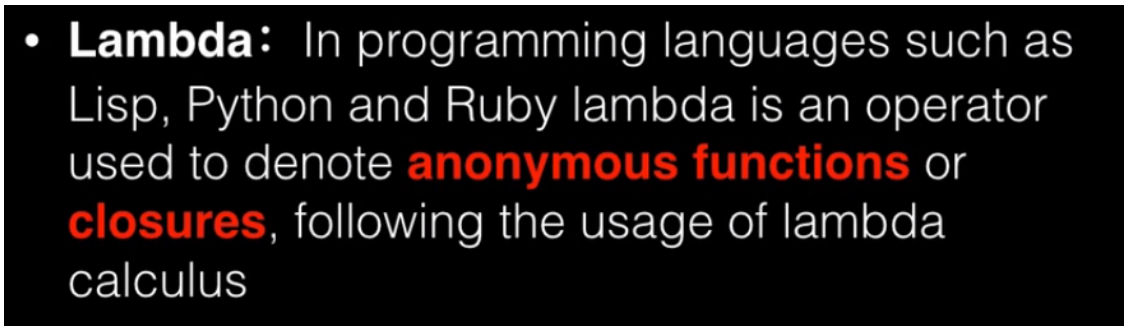
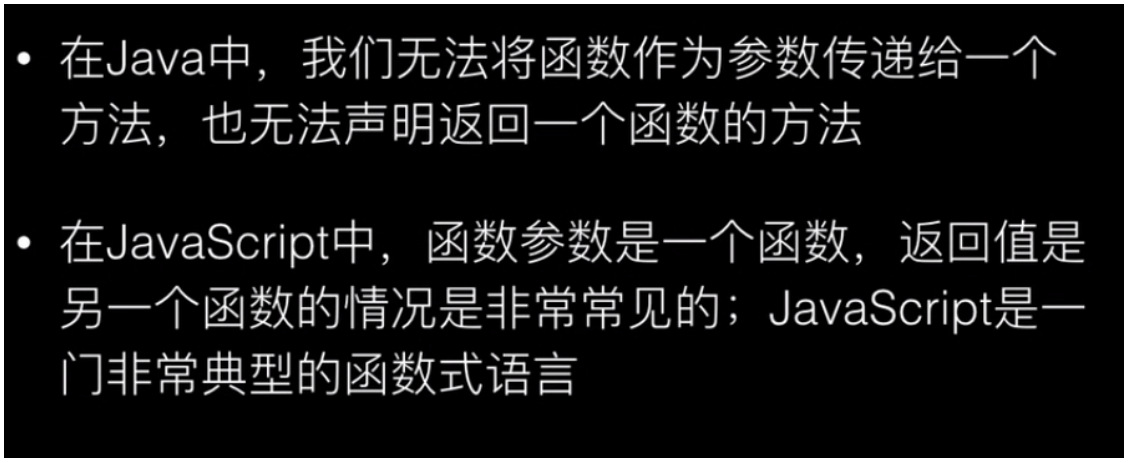
JAVA8新特性

lambda表达式与函数式接口

- **定义**



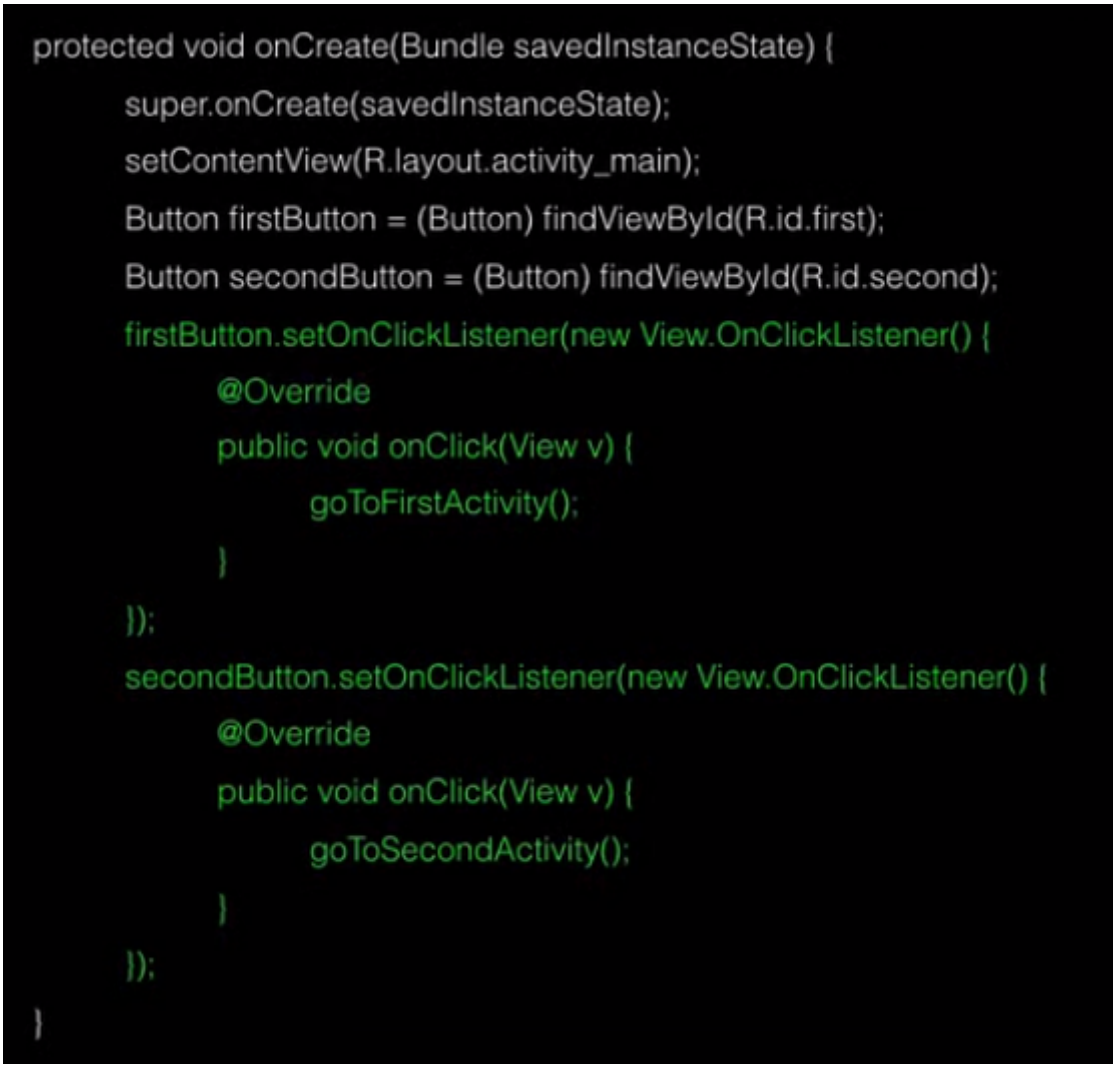
- **必要性**



- **javascript回调方法实例**

```
// javascript 中的函数作为参数
a.execute(callback(event){
          //do next
          });
```

- **Java匿名内部类实例**



- **基本表达式**

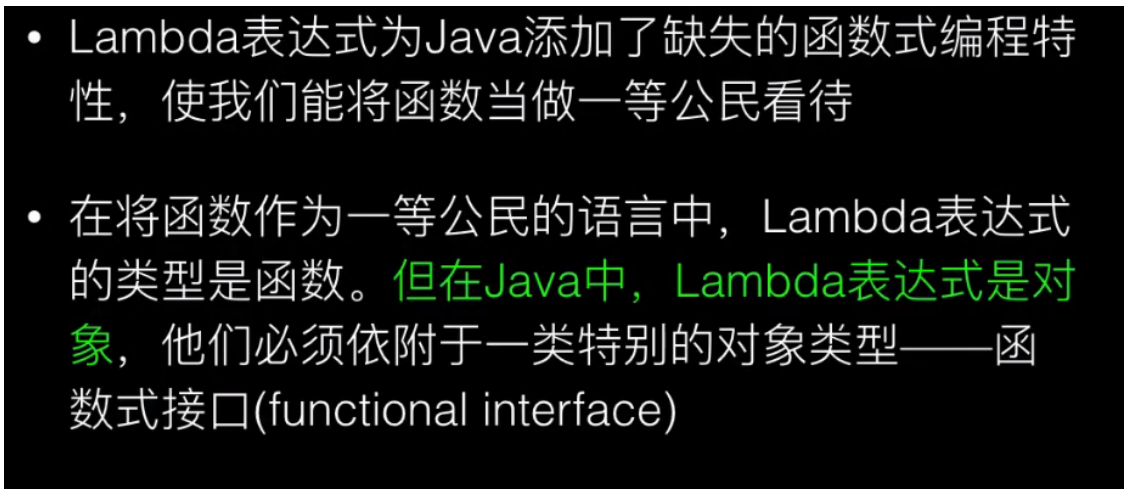
```
a.function(
(parm1,parm2,parm3)->{
    //do next
});
```

- **函数式接口**（functionInterface）

  定义：有且只有一个抽象方法的接口称为函数式接口（Conceptually, a functional interface has exactly one abstract method.）

  实例化方式：lambda表达式、方法引用、构造器引用（Note that instances of functional interfaces can be created with lambda expressions, method references, or constructor references）

- **函数式接口注解（@FunctionalInterface）：**
  - 满足函数式接口定义的接口可以显示的在该接口上加上@FunctionalInterface注解，表示这是一个函数式接口
  - 任何不满足函数式接口的接口若加上@FunctionalInterface，则编译器会报错
  - 任何满足函数式接口定义的接口但没有添加@FunctionalInterface注解，编译器同样会按照函数式接口编译该接口
- **lambda表达式的作用：**



函数式接口与方法引用

- **方法引用实例：**

```
package com.dennis.jdk8;

import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class ListTest01 {
    public static void main(String[] args) {
        List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
        for (int i = 0; i < 8; i++) {
            System.out.println(integerList.get(i));
        }
        System.out.println("=================");

        for (Integer i : integerList) {
            System.out.println(i);
        }
        System.out.println("=================");

        integerList.forEach(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) {
                System.out.println(integer);
            }
        });
        System.out.println("=================");

        integerList.forEach(integer -> {
            System.out.println(integer);
        });
        System.out.println("=================");

        // 方发引用的方式创建lambda表达式
        integerList.forEach(System.out::println);
    }
}
```

- **lambda表达式的本质：**（可以通过执行结果发现，lambda表达式本质上是一个实现了函数式接口的对象）

```
package com.dennis.jdk8;

/**
 * 一个.java文件中只能有一个public类
 */
@FunctionalInterface
interface MyFuncInterface {
    void test01();
    @Override
    public String toString();
}

public class Test02 {
    public void test(MyFuncInterface myFuncInterface) {
        System.out.println("调用函数式接口中抽象方法前");
        myFuncInterface.test01();
        System.out.println("调用函数式接口中抽象方法后");
    }

    public static void main(String[] args) {
        Test02 test02 = new Test02();

        // 传统匿名内部类的方式调用函数式接口中的方法
        test02.test(new MyFuncInterface() {
            @Override
            public void test01() {
                System.out.println("匿名内部类方式调用:实现并成功调用函数是接口中的
抽象方法");
            }
        });
        System.out.println("================================");

        // lambda方式调用
        test02.test(() -> {
            System.out.println("lambda方式调用:实现并成功调用函数是接口中的抽象方
法");
        });
        System.out.println("================================");
        // lambda表达式的本质
        MyFuncInterface myFuncInterface = () -> {
            System.out.println("实现并成功调用函数是接口中的抽象方法");
        };
        test02.test(myFuncInterface);
        System.out.println("================================");
        // myFuncInterface对象的本质
        System.out.println("myFuncInterface对象的本质");
        System.out.println(myFuncInterface.getClass());
        System.out.println(myFuncInterface.getClass().getSuperclass());

 System.out.println(myFuncInterface.getClass().getInterfaces().length);
        System.out.println(myFuncInterface.getClass().getInterfaces()[0]);
    }
}
```

输出结果:

```
调用函数式接口中抽象方法前
匿名内部类方式调用:实现并成功调用函数是接口中的抽象方法
调用函数式接口中抽象方法后
==============================
调用函数式接口中抽象方法前
lambda方式调用:实现并成功调用函数是接口中的抽象方法
调用函数式接口中抽象方法后
==============================
调用函数式接口中抽象方法前
实现并成功调用函数是接口中的抽象方法
调用函数式接口中抽象方法后
==============================
myFuncInterface对象的本质
class com.dennis.jdk8.Test02$$Lambda$2/1078694789
class java.lang.Object
1
interface com.dennis.jdk8.MyFuncInterface

Process finished with exit code 0
```

基本语法：

Java Lambda概要

- Java Lambda表达式是一种匿名函数；它是没有声明的方法，即没有访问修饰符、返回值声明和名字

Lambda表达式作用

- 传递行为，而不仅仅是值
  - 提升抽象层次
  - API重用性更好
  - 更加灵活

Java Lambda基本语法

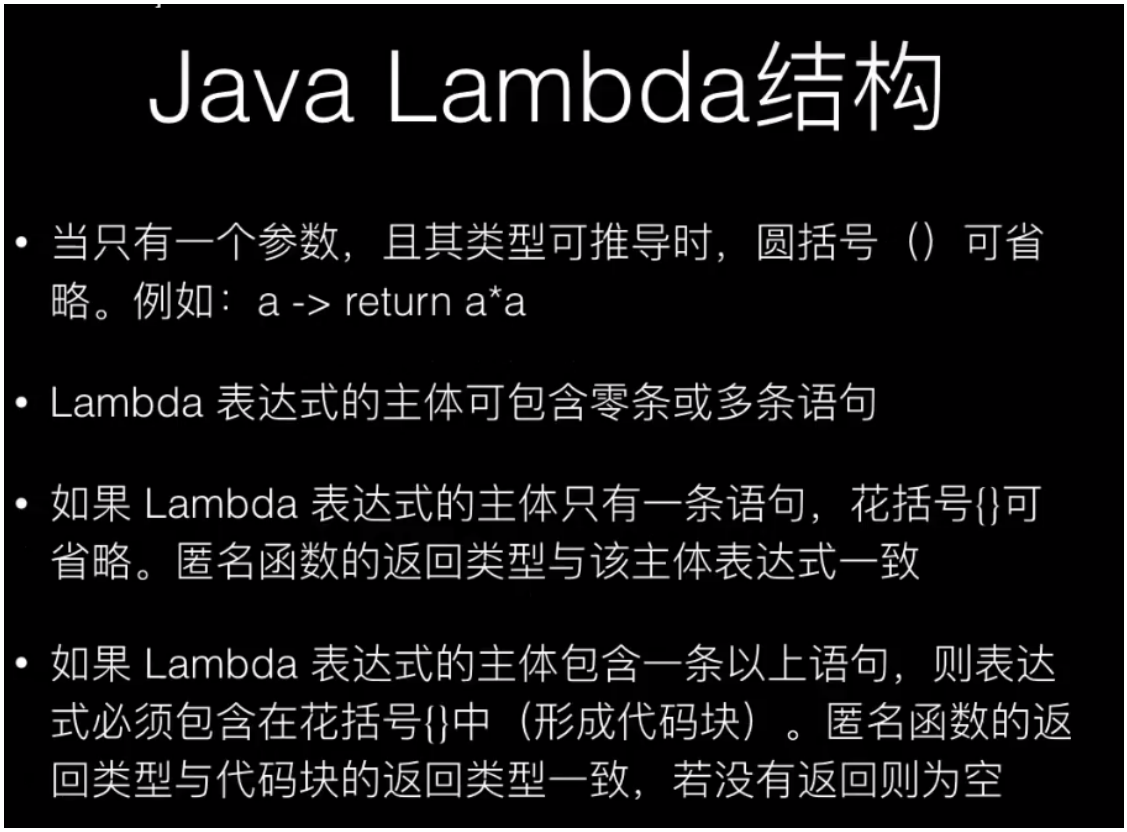
- Java中的Lambda表达式基本语法
  - (argument) -> (body)
- 比如说
  - (arg1, arg2...) -> { body }
  - (type1 arg1, type2 arg2...) -> { body }

Java Lambda示例

- Lambda示例说明
  - (int a, int b) -> { return a + b; }
  - () -> System.out.println("Hello World");
  - (String s) -> { System.out.println(s); }
  - () -> 42
  - () -> { return 3.1415 };

Java Lambda结构

- 一个 Lambda 表达式可以有零个或多个参数
- 参数的类型既可以明确声明，也可以根据上下文来推断。例如：(int a)与(a)效果相同
- 所有参数需包含在圆括号内，参数之间用逗号相隔。例如：(a, b) 或 (int a, int b) 或 (String a, int b, float c)
- 空圆括号代表参数集为空。例如：() -> 42



Function函数式接口与流初步

- **Function灵活性**：通过apply（）方法将行为作为"参数"传入

```
package com.dennis.jdk8.function;

import java.util.function.Function;

/**
 * 描述：  Function函数式接口demo(灵活性体现)
 *
 * @author Dennis
 * @version 1.0
 * @date 2020/1/12 15:12
 */
public class FunctionTest01 {
    public static void main(String[] args) {
        FunctionTest01 f1 = new FunctionTest01();
        // 加
        Integer r1 = f1.compute(2, x1 -> 8 + x1);
        System.out.println(r1);
        // 减
        Integer r2 = f1.compute(2, x1 -> 8 - x1);
        System.out.println(r2);
        // 乘
        Integer r3 = f1.compute(2, x1 -> x1 * x1);
        System.out.println(r3);
        // 除
        Integer r4 = f1.compute(2, x1 -> 8 / 2);
        System.out.println(r4);

        System.out.println(f1.add(1, 2));
        System.out.println(f1.subtract(1, 2));
```

```
        System.out.println(f1.multiply(1, 2));
        System.out.println(f1.divide(1, 2));
    }

    /**
     * 单个自变量的计算方法
     */
    public Integer compute(Integer num, Function<Integer, Integer> function)
{
        return function.apply(num);
    }

    /**
     * 传统方式实现计算方法
     */
    public Integer add(Integer a, Integer b) {
        return a + b;
    }

    public Integer subtract(Integer a, Integer b) {
        return a - b;
    }

    public Integer multiply(Integer a, Integer b) {
        return a * b;
    }

    public Double divide(Integer a, Integer b) {
        return 1D * a / b;
    }

}
```

- **andThen与compose方法的对比**
  - andThen：先执行调用者的apply()方法，再执行作为参数的Founction对象的apply方法
  - compose：与andThen执行apply的顺序相反

```
package com.dennis.jdk8.function;

import java.util.function.Function;

/**
 * 描述：  Function函数接口中andThen与compose的对比
 *
 * @author Dennis
 * @version 1.0
 * @date 2020/1/12 15:34
 */
public class FunctionTest02 {
    public static void main(String[] args) {
        FunctionTest02 test02 = new FunctionTest02();
        Integer r1 = test02.beforeCompute(2, v1 -> v1 * 3, v2 -> v2 * v2);
        System.out.println(r1); // 12
        Integer r2 = test02.afterCompute(2, v1 -> v1 * 3, v2 -> v2 * v2);
        System.out.println(r2); //36
```

```
    }

    /**
     * 先执行f2的apply(a)方法，并将其结果作为f1的apply()的参数，且最后执行f1的apply方法
     */
    public Integer beforeCompute(Integer a, Function<Integer, Integer> f1, Function<Integer, Integer> f2) {
        return f1.compose(f2).apply(a);
    }

    /**
     * 先执行f1的apply(a)方法，并将其结果作为f2的apply()的参数，且最后执行f2的apply方法
     */
    public Integer afterCompute(Integer a, Function<Integer, Integer> f1, Function<Integer, Integer> f2) {
        return f1.andThen(f2).apply(a);
    }
}
```

- **BiFunction:较之于Function只能接受一个参数， BiFunction可以接受两个参数，进一步提高了apply()方法实现时的灵活性**

  注意：lambda表达式的语法

```
package com.dennis.jdk8.function;

import lombok.AllArgsConstructor;
import lombok.Data;

import java.util.Arrays;
import java.util.List;
import java.util.function.BiFunction;
import java.util.stream.Collectors;

/**
 * 描述：    较之于Function只能接受一个参数， BiFunction可以接受两个参数，进一步提高了apply()放法实现时的灵活性
 *
 * @author Dennis
 * @version 1.0
 * @date 2020/1/12 16:28
 */
public class BiFunctionTest01 {
    public static void main(String[] args) {
        List<Person> list =
                Arrays.asList(new Person("张三", 20),
                        new Person("李四", 30),
                        new Person("王五", 50));

        BiFunctionTest01 test01 = new BiFunctionTest01();
        // 按名字条件获取
        List<Person> resultByName = test01.getPersonByName("李四", list);
        System.out.println(resultByName);
        // 按年龄条件获取
```

```
        List<Person> resultByAge = test01.getPersonByAge(40, list, (maxAge,
personList) ->
                personList.stream().filter(person -> person.getAge() <
maxAge).collect(Collectors.toList())
        );
        System.out.println(resultByAge);
    }

    public List<Person> getPersonByName(String name, List<Person>
personList) {
        BiFunction<String, List<Person>, List<Person>> biFunction =
                (n, elements) -> elements.stream().filter(person ->
person.getName().equals(n)).collect(Collectors.toList());
        return biFunction.apply(name, personList);
    }

    public List<Person> getPersonByAge(Integer maxAge, List<Person>
personList, BiFunction<Integer, List<Person>, List<Person>> biFunction) {
        return biFunction.apply(maxAge, personList);
    }

    @Data
    @AllArgsConstructor
    public static class Person {
        private String name;
        private Integer age;
    }
}
```

Predicate函数式接口

- **作用：提供判断方法test(),且提供或（or）、与(and)、非(nagete)逻辑计算**

```
package com.dennis.jdk8.predicate;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

/**
 * 描述：  predicate demo
 *
 * @author Dennis
 * @version 1.0
 * @date 2020/1/12 22:19
 */
public class PredicateTest01 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9,
10);
        PredicateTest01 test01 = new PredicateTest01();
        // 大于5
        List<Integer> r1 = test01.numFilter(numbers, num -> num > 5);
        System.out.println(r1);
        System.out.println("++++++++++++++++++++++++");
```

```
        // 小于5
        List<Integer> r2 = test01.numFilter(numbers, num -> num <= 5);
        System.out.println(r2);
        System.out.println("++++++++++++++++++++++++");

        // 偶数过滤
        List<Integer> r3 = test01.numFilter(numbers, num -> num % 2 == 0);
        System.out.println(r3);
        System.out.println("++++++++++++++++++++++++");

        // 奇数过滤
        List<Integer> r4 = test01.numFilter(numbers, num -> num % 2 != 0);
        System.out.println(r4);
        System.out.println("++++++++++++++++++++++++");

        // 大于5且为寄数
        List<Integer> r5 = test01.andFilter(numbers, num -> num % 2 != 0,
num -> num > 5);
        System.out.println(r5);
        System.out.println("++++++++++++++++++++++++");

        // 反向过滤大于5的数
        List<Integer> r6 = test01.negateFilter(numbers, num -> num > 5);
        System.out.println(r6);
        System.out.println("++++++++++++++++++++++++");

        // 全部显示
        List<Integer> r7 = test01.numFilter(numbers, num -> true);
        System.out.println(r7);
        System.out.println("++++++++++++++++++++++++");

        // 全部不显示
        List<Integer> r8 = test01.numFilter(numbers, num -> false);
        System.out.println(r8);
        System.out.println("++++++++++++++++++++++++");
    }

    // 单个判断
    public List<Integer> numFilter(List<Integer> numbers, Predicate<Integer>
predicate) {
        ArrayList<Integer> integers = new ArrayList<>();
        numbers.forEach(integer -> {
            if (predicate.test(integer)) {
                integers.add(integer);
            }
        });
        return integers;
    }

    // 与组合判断
    public List<Integer> andFilter(List<Integer> numbers, Predicate<Integer>
p1, Predicate<Integer> p2) {
        ArrayList<Integer> integers = new ArrayList<>();
        for (Integer integer : numbers) {
            if (p1.and(p2).test(integer)) {
                integers.add(integer);
            }
```

```
        }
        return integers;
    }

    // 逻辑非判断
    public List<Integer> negateFilter(List<Integer> numbers,
Predicate<Integer> p1) {
        ArrayList<Integer> integers = new ArrayList<>();
        for (Integer integer : numbers) {
            if (p1.negate().test(integer)) {
                integers.add(integer);
            }
        }
        return integers;
    }
}
```

Supplier函数式接口

- **get()方法不接受参数，返回一个泛型T类的结果对象，可用来代替工厂方法**

```
package com.dennis.jdk8.supplier;

import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.experimental.Accessors;

import javax.swing.plaf.SplitPaneUI;
import java.util.function.Supplier;

/**
 * 描述： Supplier demo：不接受参数，返回一个泛型T类的结果对象，可用来代替工厂方法
 *
 * @author Dennis
 * @version 1.0
 * @date 2020/1/12 23:10
 */
public class SupplierTest01 {
    public static void main(String[] args) {
        Supplier<Student> supplier = () -> new Student();
        Student student = supplier.get();
        System.out.println(student);

        System.out.println("------------------");
        // 方法引用的当时实现lambda表达式
        Supplier<Student> supplier1 = Student::new;
        System.out.println(supplier1.get());

    }

    @Data
    @Accessors(chain = true)
    @NoArgsConstructor
    public static class Student {
        private String name = "张三";
```

```
        private Integer age = 18;
    }
}
```

BinaryOperator函数式接口

- **BinaryOperator继承至BiFunction函数式接口，是一个参数类型和返回值类型都相同的特殊BiFunction接口**

```
package java.util.function;

import ...

/**
 * Represents an operation upon two operands of the same type, producing a result
 * of the same type as the operands.  This is a specialization of
 * {@link BiFunction} for the case where the operands and the result are all of
 * the same type.
 *
 * <p>This is a <a href="package-summary.html">functional interface</a>
 * whose functional method is {@link #apply(Object, Object)}.
 *
 * @param <T> the type of the operands and result of the operator
 *
 * @see BiFunction
 * @see UnaryOperator
 * @since 1.8
 */
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {
```

```
package com.dennis.jdk8.binaryoperator;

import java.util.Comparator;
import java.util.function.BinaryOperator;

/**
 * 描述：  BinaryOperator demo
 *
 * @author Dennis
 * @version 1.0
 * @date 2020/1/13 21:29
 */
public class BinaryOperatorTest01 {
    public static void main(String[] args) {
        BinaryOperatorTest01 test01 = new BinaryOperatorTest01();
        Integer r1 = test01.compute(2, 3, Integer::sum);
        Integer r2 = test01.compute(2, 3, (v1, v2) -> v1 * v2);
        System.out.println(r1);
        System.out.println(r2);

        String str1 = "Hello";
        String str2 = "hi";
        // 以字符串长度判断大小
        String shortOne = test01.getShortOne(str1, str2, (s1, s2) ->
s1.length() - s2.length());
        System.out.println(shortOne);

        // 以首字母的ASCII码值大小判断大小
```

```
        String shortOne1 = test01.getShortOne(str1, str2, (s1, s2) ->
s1.charAt(0) - s2.charAt(0));
        System.out.println(shortOne1);
    }

    /**
     * 计算两参数为整数的需求
     */
    public Integer compute(Integer a, Integer b, BinaryOperator<Integer>
binaryOperator) {
        return binaryOperator.apply(a, b);
    }

    /**
     * 获取较小值
     */
    public String getShortOne(String s1, String s2, Comparator<String>
comparator) {
        return BinaryOperator.minBy(comparator).apply(s1, s2);
    }
}
```