Java 8 New Features

Reference

- Java 8：（先阅读课件，然后再阅读这三个链接，只看跟课件提到的topics。其余topics一律不看）
    - JDK8 新特性 (yuque.com)
    - https://juejin.cn/post/6962035387787116551
    - https://www.cnblogs.com/wmyskxz/p/13527583.html
- Optional
    - https://medium.com/swlh/how-to-write-better-code-with-java-8s-optional-b6d862f28862

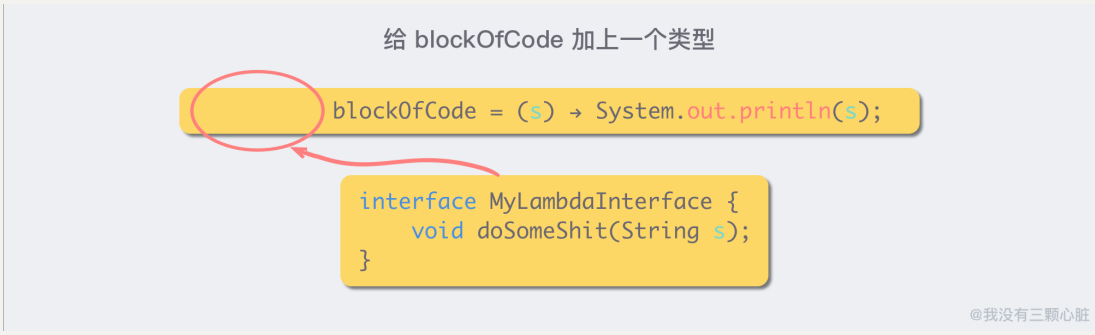
Default method and static method in interface

```java
public interface DIMLearn {
    static final String BLOG = "is Chuwa a";

    int add(int a, int b);

    default int substract(int a, int b) {
        return a - b;
    }

    static String blogName() {
        return BLOG;
    }
}
```

Lambda & Functional Interface（重点，easy）

Functional Interface

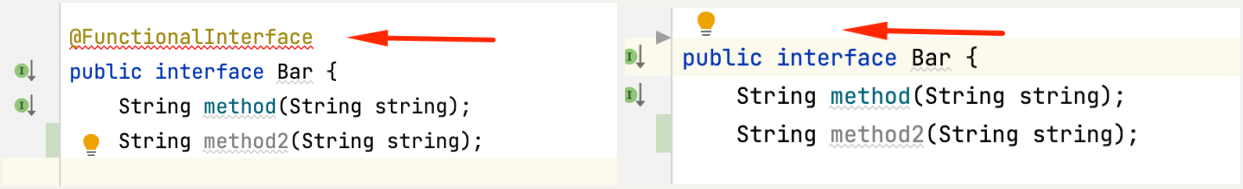
- Has one single Abstract method
- @FunctionalInterface - for sanity check
- Can have a lot of default methods
- Lambda is the implementation of the abstract method
    -




```java
@FunctionalInterface
public interface Bar {
    // 唯一一个abstract method
    String method(String string);


    default String defaultBar() {
        String s = "default Bar method";
        System.out.println(s);
        return s;
    }

    default String defaultCommon() {
        String s = "defaultCommon from Bar";
        System.out.println(s);
        return s;
    }
}
```

If I write two abstract methods with @FunctionalInterface, it will throw errors

If no @FunctionalInterface, it will not do sanity check, and we also can not use lambda

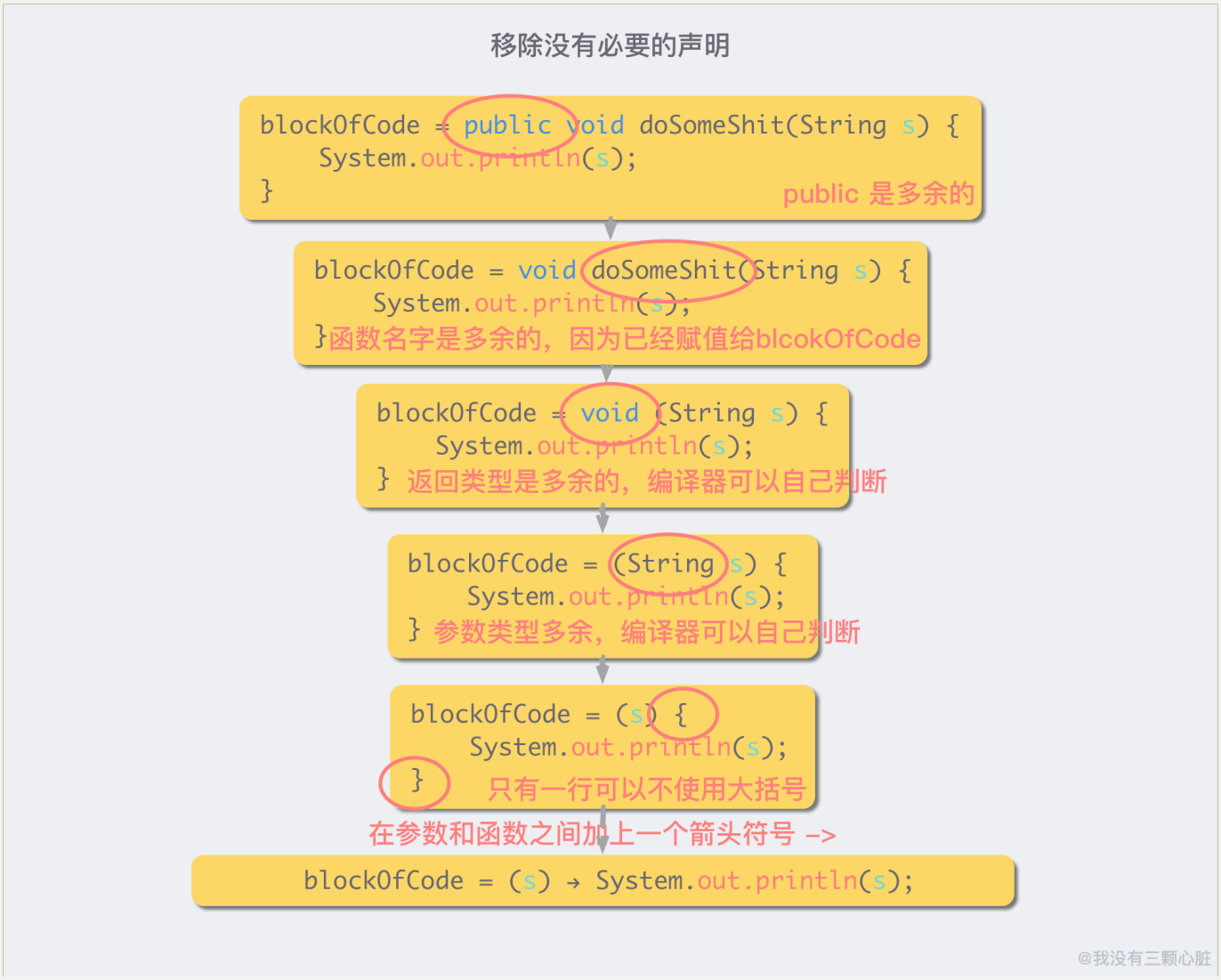


Lambda

- To Replace anonymous inner class
- Work with functional interface

- Lambda 表达式其实就是实现 SAM（**Single Abstract Method**，即该接口中有且仅有一个抽象方法，当然该接口中还可以包含非抽象方法，在 JDK 8 中增加了静态方法和默认方法）接口的语法糖。
- 其实，只要满足 SAM 特征的接口就是函数式接口，就可以使用 Lambda 表达式了，但是如果在声明函数式接口的时候，使用 `@FunctionalInterface` 注解来声明，那么编译器就会强制检查该接口是否确实有且仅有一个抽象方法，如果不是，将报错。
- 之前学过的SAM接口中，标记了 `@FunctionalInterface` 注解的接口有：Runnable、Comparator、FileFilter。
- JDK 8 在 `java.util.function` 包中新增了很多函数式接口，主要分为消费型、供给型、断言型（判断型）、函数型（功能型）。基本上可以满足我们实际的开发，当然，我们也可以自定义函数式接口。

To Replace anonymous inner class



Before Java 8, we can use Anonymous Class to override this method.

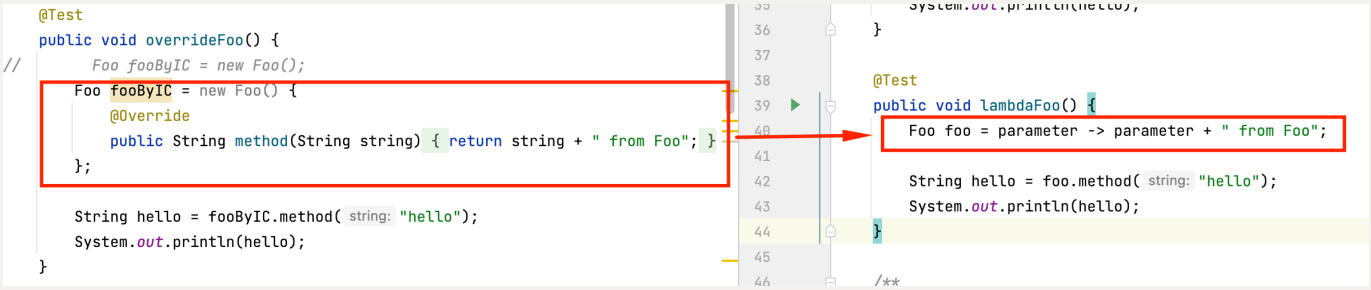
Notice that in Foo(), we have abstract method: `String method(String string);`

```
1
2       @Test
3       public void overrideFoo() {
4           // Before Java 8, we can use Anonymous Class to override
    this method
5           // Interface var = new Class
6           // List<Integer> var = new ArrayList<>()
7           // Foo 这个interface有一个abstract method, 所以在Anonymous
    Class里需要override来提供method body
8           Foo fooByIC = new Foo() {
```

```
            @Override
            public String aMethod(String string) {
                return string + " from Foo";
            }
        };

        String hello = fooByIC.aMethod("hello");
        System.out.println(hello);
    }

```

In Java 8, we can use lambda to instead of override.

```
    @Test
    public void lambdaFoo() {
        // Foo.aMethod() 是abstract method,缺少method body. lambda提供method body.
        // 比Anonymous class 简洁很多。
        Foo foo = parameter -> parameter + " from Foo";

        String hello = foo.aMethod("hello");
        System.out.println(hello);
    }

    @Test
    public void lambdaFoo2() {
        // 可以提供任何method body
        Foo foo = parameter -> parameter.toUpperCase() + " from Foo";

        String hello = foo.aMethod("hello");
        System.out.println(hello);
    }
```



再看一个大家熟悉的例子Comparator。

```java
@Test
public void test2(){
    //未使用Lambda表达式的写法
    Comparator<Integer> com1 = new Comparator<Integer>() {
        @Override
        public int compare(Integer o1, Integer o2) {
            return Integer.compare(o1,o2);
        }
    };

    int compare1 = com1.compare(12, 32);
    System.out.println(compare1);//-1
    System.out.println("====================");

    //Lambda表达式的写法
    Comparator<Integer> com2 = (o1,o2) -> Integer.compare(o1,o2);

    int compare2 = com2.compare(54, 21);
    System.out.println(compare2);//1
    System.out.println("====================");

    //方法引用
    Comparator<Integer> cpm3 = Integer::compareTo;
    int compare3 = cpm3.compare(12, 12);
    System.out.println(compare3);//0
}
```

Lambda 表达式的语法

- 说明：
    - `(形参列表)`：就是要赋值的函数式接口的抽象方法的`(形参列表)`。
    - `{Lambda体}`：就是实现这个抽象方法的方法体。
    - `->`：Lambda操作符。
- 优化：
    - 当`{Lambda体}`只有一条语句的时候，可以省略`{}`和`{;}`。
    - 当`{Lambda体}`只有一条语句的时候，并且这个语句还是 return 语句的时候，return 也可以省略，但是如果`{;}`没有省略，那么 return 是不可以省略的。
    - `(形参列表)`的类型可以省略。
    - 当`(形参列表)`的形参个数只有一个，那么可以将数据类型和`()`一起省略，但是形参名不能省略。
    - 当`(形参列表)`是空参的时候，`()`不能省略。

Lambda can use unchanged variable outside of lambda

只要该变量的内存地址不变，则该变量可被lambda用。

- Final variable
- Non-final variable however never changed
- Object variable (if we use new to reassign a new memory, then this variable is changed, can not be used in lambda)

```
    @Test
    public void testFinal() {
        final String localVariable = "Local";
        Foo foo = parameter -> {
            return parameter + " " + localVariable;
        };
```

```
7
8           System.out.println(foo.aMethod("hello"));
9       }
10
11      /**
12       * Use “Effectively Final” Variables
13       * 当variable只赋值一次，没有任何变动的时候，Java默认是final。
14       * 注意，在lambda expression的前后都不能被改变
15       */
16      @Test
17      public void testEffectivelyFinal() {
18          String localVariable = "Local";
19          Foo foo = parameter -> {
20              return parameter + " " + localVariable;
21          };
22
23          System.out.println(foo.aMethod("hello"));
24      }
25
26      /**
27       * 换object会报错，因为换了内存地址
28       */
29      @Test
30      public void testFinal21() {
31          String localVariable = "Local";
32          localVariable = "LOCAL"; // 新的内存地址
33
34          Foo foo = parameter -> {
35              return parameter + " " + localVariable; // 会报错
36          };
37
38          System.out.println(foo.aMethod("hello"));
39      }
40
41      @Test
42      public void testFinal22() {
```

```
43          String localVariable = "Local";
44
45          Foo foo = parameter -> {
46              return parameter + " " + localVariable; // 会报错
47          };
48
49          localVariable = "LOCAL"; // 新的内存地址
50
51          System.out.println(foo.aMethod("hello"));
52      }
53
54      /**
55       * Object 的set方法不会报错
56       */
57      @Test
58      public void testFinal3() {
59          List<Employee> employees = EmployeeData.getEmployees();
60
61          Employee employee = employees.get(0);
62          employee.setAge(55);
63          Foo foo = parameter -> {
64              return parameter + " " + employee;
65          };
66
67          System.out.println(foo.aMethod("hello"));
68      }
69
```

Method Reference

> *类名::方法名*

```
1  Arrays.sort(stringsArray,(s1,s2) -> s1.compareToIgnoreCase(s2));
2  Arrays.sort(stringsArray, String::compareToIgnoreCase);
```

方法引用是用来直接访问类或者实例的已经存在的方法或者构造方法。方法引用提供了一种引用而不执行方法的方式，它需要由兼容的函数式接口构成的目标类型上下文。计算时，方法引用会创建函数式接口的一个实例。**我们需要把握的重点是：函数引用只是简化Lambda表达式的一种手段而已。**

类型	示例
引用静态方法	ContainingClass::staticMethodName
引用某个对象的实例方法	containingObject::instanceMethodName
引用某个类型的任意对象的实例方法	ContainingType::methodName
引用构造方法	ClassName::new
数组类型::new	数组类型::new

Code Example

```java
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.function.BiFunction;
import java.util.function.Function;
import java.util.function.Supplier;

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public static int compareByName(Person p1, Person p2) {
        return p1.name.compareTo(p2.name);
    }
```

```
20
21      public int getAge() {
22          return age;
23      }
24
25      public String getName() {
26          return name;
27      }
28
29      @Override
30      public String toString() {
31          return name + " (" + age + ")";
32      }
33  }
34
35  public class MethodReferenceExample {
36      public static void main(String[] args) {
37          // 1. 静态方法引用
38          List<Person> people = Arrays.asList(
39                  new Person("Alice", 30),
40                  new Person("Bob", 25),
41                  new Person("Charlie", 35));
42
43          // 使用Lambda表达式
44          people.sort((p1, p2) -> Person.compareByName(p1, p2));
45
46          // 使用静态方法引用
47          people.sort(Person::compareByName);
48
49          // 2. 实例方法引用（特定对象的实例方法）
50          Comparator<Person> byAgeComparator =
    Comparator.comparingInt(Person::getAge);
51          people.sort(byAgeComparator);
52
53          // 3. 类的实例方法引用
54          // 不要尝试理解Function,将会被stream使用。
```

```
55          Function<Person, String> getNameFunction =
    Person::getName;
56          List<String> names = Arrays.asList("Alice", "Bob",
    "Charlie");
57          names.sort(String::compareToIgnoreCase);
58
59          // 4. 构造方法引用
60          // 不要尝试理解BiFunction,将会被stream使用。
61          BiFunction<String, Integer, Person> personCreator =
    Person::new;
62          Person newPerson = personCreator.apply("David", 40);
63
64          System.out.println(people);
65      }
66  }
67
```

Optional（重点，easy）

> *To avoid Null checks and run time NullPointerExceptions*
>
> - *到目前为止，臭名昭著的空指针异常是导致 Java 应用程序失败的最常见的原因。以前，为了解决空指针异常，Google 公司著名的 Guava 项目引入了 Optional 类，Guava 通过使用检查 null 值的方式来防止代码污染，它鼓励程序员写干净的代码。受到 Google 的 Guava 的启发，Optional 类已经成为了 JDK 8 类库的一部分。*
> - *Optional 实际上是个容器：它可以保存类型 T 的值，或者仅仅保存 null 。Optional 提供了很多有用的方法，这样我们就不用显示进行 null 值检测。*

Why Optional Better?

> *更优雅、更安全的方式处理可能为null的值。使用Optional对象，我们可以避免NullPointerException，并使代码更简洁、易读。*

```java
public class OptionalDemo {
    public static void main(String[] args) {
        Map<Integer, User> userMap = new HashMap<>();
        userMap.put(1, new User("Alice", new Address("Main Street")));
        userMap.put(2, new User("Bob", null));

        // Without Optional
        String streetName = "Unknown";
        User user = userMap.get(2);
        if (user != null) {
            Address address = user.getAddress();
            if (address != null) {
                streetName = address.getStreet();
            }
        }
        System.out.println("Street name without Optional: " + streetName);

        // With Optional
        streetName = Optional.ofNullable(userMap.get(2))
                .map(User::getAddress)
                .map(Address::getStreet)
                .orElse("Unknown");
        System.out.println("Street name with Optional: " + streetName);
    }
}
```

Optional Methods

Create Optional Object

1. `Optional.empty()`
    a. 创建一个空的Optional对象，表示没有值。这种方法用于需要表示缺少值的场景。

```java
Optional<String> emptyOptional = Optional.empty();
// 非Optional对象创建缺少值的场景
List<String> lst = new ArrayList<>();
TreeNode node = null;
```

2. `Optional.of(T value)`
    a. 根据一个非空值创建一个Optional对象。如果传入的值为null，这个方法会抛出NullPointerException。

```java
String name = "Alice";
Optional<String> nameOptional = Optional.of(name);
String name2 = null;
Optional<String> nameOptional = Optional.of(name2);
// 会抛出NullPointerException
```

3. `Optional.ofNullable(T value)`
    a. 根据一个值（可以为null）创建一个Optional对象。如果传入的值为null，它会创建一个空的Optional对象。这种方法适用于值可能为null的场景。

```java
String name = "Alice";
Optional<String> nameOptional =
Optional.ofNullable(name);
String name2 = null;
Optional<String> nameOptional =
Optional.ofNullable(name2); // 完全不会抛出
NullPointerException
```

Get Element From Optional

1. `get()`
    a. 如果Optional对象包含一个值，该方法会返回这个值；如果Optional对象是空的（不包含值），则抛出一个`NoSuchElementException`异常。

```
Optional<String> optionalName =
Optional.of("Alice");
String name = optionalName.get(); // 返回 "Alice"
```

2. `orElse(T other)`
    a. 如果Optional对象包含一个值，该方法返回这个值；如果Optional对象是空的，返回指定的默认值。

```
Optional<String> optionalName = Optional.empty();
String name = optionalName.orElse("Unknown"); // 返回
"Unknown"

// question
String name = optionalName.get(); // 返回什么?
```

3. `orElseGet(Supplier<? extends T> other)`
    a. 如果Optional对象包含一个值，该方法返回这个值；如果Optional对象是空的，使用指定的Supplier函数生成并返回一个默认值。

```
Optional<String> optionalName = Optional.empty();
String name = optionalName.orElseGet(() ->
"Unknown"); // 返回 "Unknown"
```

4. `orElseThrow()`
    a. 如果Optional对象包含一个值，该方法返回这个值；如果Optional对象是空的，抛出一个`NoSuchElementException`异常。这个方法在Java 10中引入，与`get()`方法的功能相同，但命名更符合实际行为。

```java
Optional<String> optionalName =
Optional.of("Alice");
String name = optionalName.orElseThrow(); // 返回
"Alice"

String name2 = null;
Optional<String> optionalName2 =
Optional.ofNullable(name);
String name = optionalName.orElseThrow(() ->
"UNKNOWN"); // 返回
```

Optional Other methods

1. `isPresent()`
    a. 该方法用于检查Optional对象是否包含值。如果包含值，返回`true`；如果Optional对象为空（不包含值），返回`false`。
2. `ifPresent(Consumer<? super T> action)`
    a. 该方法用于在Optional对象包含值时执行指定的操作。如果Optional对象包含值，将该值传递给作为参数的Consumer函数；如果Optional对象为空（不包含值），不执行任何操作。
3. Code example
    a. 在这个示例中，我们使用`isPresent()`方法检查Optional对象是否包含User实例。如果包含值，我们从Optional对象中获取值并执行与用户相关的业务逻辑。接下来，我们使用`ifPresent()`方法在Optional对象包含值时执行一个操作，例如向用户发送邮件。我们还在这个操作中使用了嵌套的Optional对象，以确保在执行操作时用户的电子邮件地址不为null。这些方法使我们能够更安全地处理可能为null的值，避免NullPointerException。

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

public class OptionalExample {
```

```
6       public static void main(String[] args) {
7           Map<Integer, User> userMap = new HashMap<>
    ();
8           userMap.put(1, new User("Alice",
    "alice@example.com"));
9           userMap.put(2, new User("Bob", null));
10
11          Optional<User> optionalUser =
    Optional.ofNullable(userMap.get(2));
12
13          // Using isPresent() to check if the
    Optional contains a value
14          if (optionalUser.isPresent()) {
15              User user = optionalUser.get();
16              // Perform some business logic with the
    user object
17              System.out.println("Performing business
    logic with user: " + user.getName());
18          } else {
19              System.out.println("User not found.");
20          }
21
22          // Using ifPresent() to perform an action
    when the Optional contains a value
23          optionalUser.ifPresent(user -> {
24              // Only perform the action if the email
    is not null
25
     Optional.ofNullable(user.getEmail()).ifPresent(ema
    il -> {
26                  System.out.println("Sending email
    to: " + email);
27                  // Call a method to send email to
    the user
28                  // sendEmail(email);
29              });
```

```
        });
    }
}

class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }
}

```

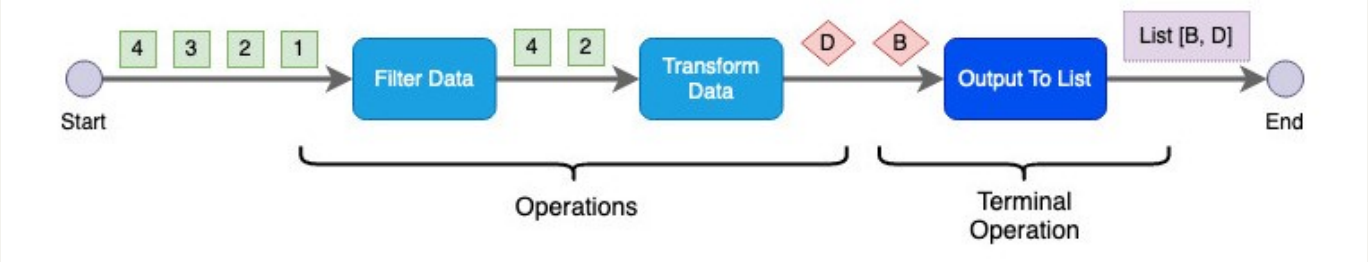
Stream API (重点，hard)

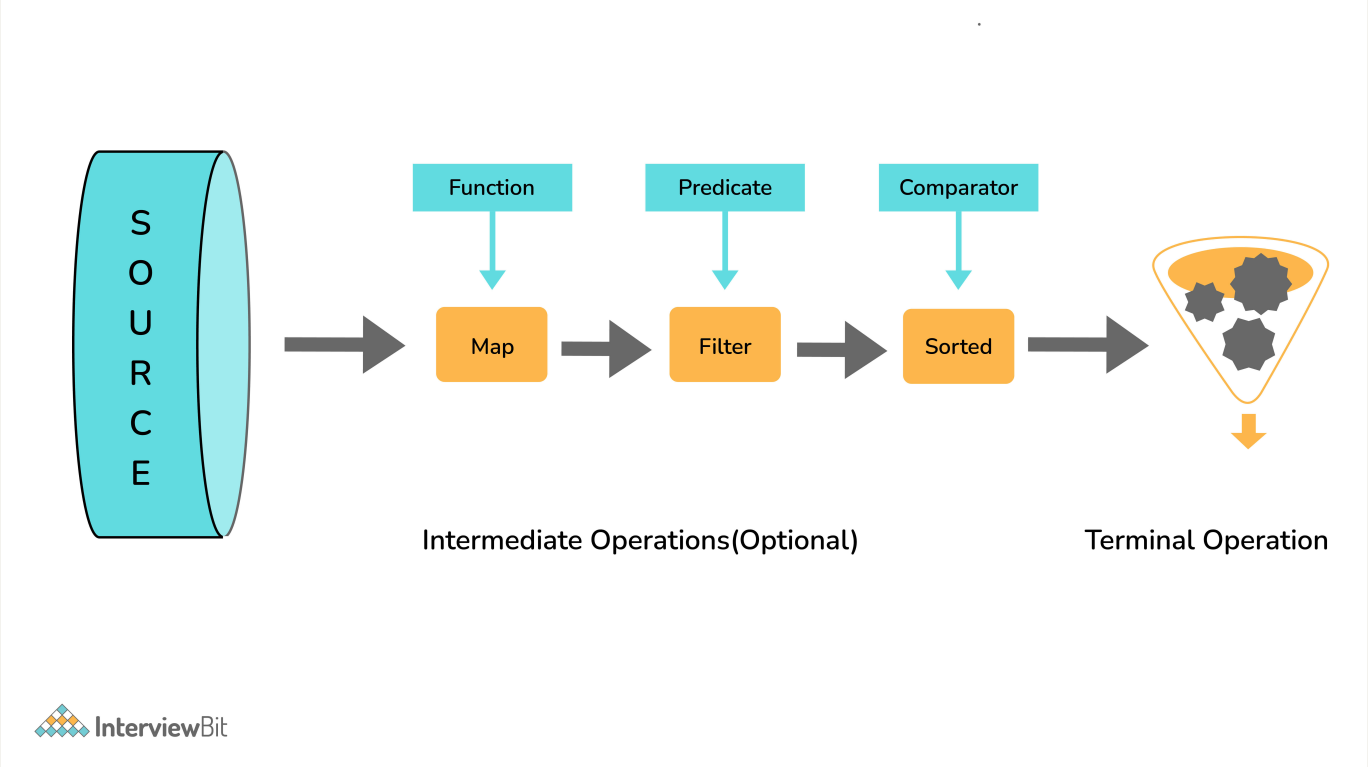
> *Feature:*
>
> - *There are many ways to create a stream instance of different sources. Once created, the instance **will not modify its source**, therefore allowing the creation of multiple instances from a single source.*
> - *Java 8 streams **can't be reused**. That is, One stream only can be used once. Details is at section*

Code: https://github.com/TAIsRich/chuwa-eij-tutorial/tree/main/02-java-core/src/main/java/com/chuwa/tutorial/t06_java8

Stream的使用主要由三部分构成。

1. Start: 构建stream 对象
2. Intermediate operation: process数据，比如filter筛选，map转换数据
3. Terminate operation：对数据搜集。 一旦有终止操作，该stream pipeline会终止。





Create stream object

1. 从集合（Collection）创建Stream
    a. 几乎所有的集合类（如List和Set）都可以通过调用`stream()`方法创建一个Stream对象。这是创建Stream的最常见方式。

```java
List<String> list = Arrays.asList("apple", "banana", "cherry");
Stream<String> stream = list.stream();
```

2. 使用`Stream.of()`方法
    a. 使用`Stream.of()`方法可以直接从多个元素创建一个Stream对象。

```java
Stream<String> stream = Stream.of("apple", "banana", "cherry");
```

3. 从数组创建Stream
    a. 通过使用`Arrays.stream()`方法，可以从数组创建一个Stream对象。

```java
String[] array = {"apple", "banana", "cherry"};
Stream<String> stream = Arrays.stream(array);
```

4. 使用`Stream.iterate()`方法
    a. `Stream.iterate()`方法允许创建一个无限的、有规律的序列。通常与`limit()`方法一起使用，以限制生成的元素数量。

```java
Stream<Integer> stream = Stream.iterate(0, n -> n + 2).limit(5); // Generates 0, 2, 4, 6, 8
```

5. 使用`Stream.generate()`方法
    a. `Stream.generate()`方法允许创建一个无限的、无规律的序列。通常与`limit()`方法一起使用，以限制生成的元素数量。

```java
Stream<Double> stream = Stream.generate(Math::random).limit(5);
```

Intermediate operation

```
* 一，筛选
* 1. filter(Predicate p) - 接受lambda，从流中排出某些元素
* 2. limit(n) - 截断流，使其元素不超过给定的数量
* 3. skip(n) - 跳过前n个元素
* 4. distinct() - 筛选，通过元素的hashcode(), equals()去除重复元素
* <p>
* 二，映射
* 1, map(function f) element -> black box(f) -> return new element
* 2, flatMap(function f)
* 三，  排序
* 1, sort
```

Terminate operation

predictable -> true/false: filter, anyMatch, allMatch, noneMatch

Map -> fucntion -> return an element

```
* 一，  匹配与查找
* 1,  allMatch(Predicate p) - 检查是否匹配所有的元素
* 2,  anyMatch(Predicate p) - 检查是否至少匹配一个元素
* 3,  noneMatch(Predicate p) - 检查是否没有匹配的元素
* 4, findFirst - 返回第一个元素
* 5,  findAny - 返回当前流中的任意元素
* 6, count - 返回流中元素的个数
* 7, max(Comparator c) - 返回流中的最大值
* 8, min(Comparator c) - 返回流中的最小值
* 9, forEach(Consumer c) - 内部迭代

*
* 二，归约
* 1, reduce(T identity, BinaryOperator) - 可以将流中的元素反复结合起来，得到一个值
* 2, reduce(BinaryOperator) - 可以将流中的元素反复结合起来，得到一个值
```

```
*
* 三， 收集
* 1, collect(Collectors c)
```

Stream API Chain

> *Stream API允许我们通过链式操作对集合进行复杂的操作*

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamChainExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eva", "Frank");

        List<String> filteredNames = names.stream()          // Convert the list to a stream
            .filter(name -> name.length() >= 4)              // Keep only the names with at least 4 characters
            .map(String::toUpperCase)                        // Convert all names to uppercase
            .sorted((name1, name2) -> name2.length() - name1.length()) // Sort names by decreasing length
            .collect(Collectors.toList());                   // Collect the result back to a list

        System.out.println("Filtered names: " + filteredNames);
    }
}

```

Question：终止操作之后，还能继续么？如果我想继续，该怎么做？

Conclusion

1. Stream 自己不会存储元素。
2. Stream 不会改变源对象，每次处理读会返回一个持有结果的新的 Stream 。
3. Stream 操作是延迟执行的，意味着会等到需要结果的时候才执行。

Collection vs. Stream API

> *集合讲的是数据，负责存储数据，Stream 流讲的是计算，负责处理数据！*

COLLECTIONS	STREAMS
Data structure holds all the data elements	No data is stored. Have the capacity to process an infinite number of elements on demand
External Iteration	Internal Iteration
Can be processed any number of times	Traversed only once
Elements are easy to access	No direct way of accessing specific elements
Is a data store	Is an API to process the data

Intermediate Operation

方 法	描 述
filter(Predicate p)	接收 Lambda ， 从流中排除某些元素。
distinct()	筛选，通过流所生成元素的equals() 去除重复元素。
limit(long maxSize)	截断流，使其元素不超过给定数量。
skip(long n)	跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。与 limit(n) 互补。
peek(Consumeraction)**	接收Lambda，对流中的每个数据执行Lambda体操作。
sorted()	产生一个新流，其中按自然顺序排序。
sorted(Comparator com)	产生一个新流，其中按比较器顺序排序。
map(Function f)	接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
mapToDouble(ToDoubleFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 DoubleStream。
mapToInt(ToIntFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 IntStream。
mapToLong(ToLongFunction f)	接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的LongStream。
flatMap(Function f)	接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流。

Terminate operation

方法	描述
boolean allMatch(Predicate p)	检查是否匹配所有元素
boolean anyMatch(Predicate p)	检查是否至少匹配一个元素
boolean noneMatch(Predicate p)	检查是否没有匹配所有元素
`Optional<T>` **findFirst()**	返回第一个元素
`Optional<T>` **findAny()**	返回当前流中的任意元素
long count()	返回流中元素总数
`Optional<T>` **max(Comparator c)**	返回流中最大值
`Optional<T>` **min(Comparator c)**	返回流中最小值
void forEach(Consumer c)	迭代
T reduce(T iden, BinaryOperator b)	可以将流中元素反复结合起来，得到一个值。返回 T
U reduce(BinaryOperator b)	可以将流中元素反复结合起来，得到一个值。返回 Optional
R collect(Collector c)	将流转换为其他形式。接收一个 Collector接口的实现，用于给Stream中元素做汇总的方法

Collectors

方法	说明	重要性	示例
toList()	收集到 List	5	`stream.collect(Collectors.toList())`
groupingBy(Function)	分组	5	`stream.collect(Collectors.groupingBy(String::length))`
groupingBy(Function, Collector)	分组（指定下游收集器）	4	`stream.collect(Collectors.groupingBy(String::length, Collectors.toList()))`
toSet()	收集到 Set	4	`stream.collect(Collectors.toSet())`
joining()	字符串拼接	4	`stream.collect(Collectors.joining())`
joining(CharSequence)	字符串拼接（指定分隔符）	4	`stream.collect(Collectors.joining(", "))`
counting()	计数	4	`stream.collect(Collectors.counting())`
summingInt(ToIntFunction)	计算 int 总和	4	`stream.collect(Collectors.summingInt(Integer::valueOf))`
toCollection(Supplier)	收集到自定义集合	3	`stream.collect(Collectors.toCollection(HashSet::new))`
minBy(Comparator)	查找最小值	3	`stream.collect(Collectors.minBy(Comparator.naturalOrder()))`
maxBy(Comparator)	查找最大值	3	`stream.collect(Collectors.maxBy(Comparator.naturalOrder()))`
summingLong(ToLongFunction)	计算 long 总和	3	`stream.collect(Collectors.summingLong(Long::valueOf))`
summingDouble(ToDoubleFunction)	计算 double 总和	3	`stream.collect(Collectors.summingDouble(Double::valueOf))`
averagingInt(ToIntFunction)	计算 int 平均值	3	`stream.collect(Collectors.averagingInt(Integer::valueOf))`
averagingDouble(ToDoubleFunction)	计算 double 平均值	3	`stream.collect(Collectors.averagingDouble(Double::valueOf))`
partitioningBy(Predicate)	分区	3	`stream.collect(Collectors.partitioningBy(s -> s.length() > 5))`
partitioningBy(Predicate, Collector)	分区（指定下游收集器）	3	`stream.collect(Collectors.partitioningBy(s -> s.length() > 5, Collectors.toList()))`
averagingLong(ToLongFunction)	计算 long 平均值	2	`stream.collect(Collectors.averagingLong(Long::valueOf))`

Complicated Code Example

```java
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class StreamExample {
```

```
7
8       public static void main(String[] args) {
9           List<String> words = Arrays.asList("hello", "world",
    "java", "stream", "api", "example");
10
11          // 使用Stream API按字符串长度分组，并将字符串转换为大写
12          Map<Integer, List<String>> groupedWords = words.stream()
13                  .collect(Collectors.groupingBy(String::length));
14
15          // 在collect()操作后，为每个分组的单词列表创建一个新的Stream
16          groupedWords.forEach((length, wordList) -> {
17              List<String> upperCaseWords = wordList.stream()
18                      .map(String::toUpperCase)
19                      .collect(Collectors.toList());
20              System.out.println("Words with length " + length + ":
    " + upperCaseWords);
21          });
22
23          // 在collect()操作后，计算每个分组单词的平均长度
24          Map<Integer, Double> averageLengthPerGroup =
    groupedWords.entrySet().stream()
25                  .collect(Collectors.toMap(
26                          Map.Entry::getKey,
27                          e ->
    e.getValue().stream().mapToInt(String::length).average().orElse(0.
    0)
28                  ));
29          System.out.println("Average length per group: " +
    averageLengthPerGroup);
30      }
31  }
32
```

CompletableFuture

Would be explained in multiple threading class.