Functional Paradigm in Java

COMP2511, CSE, UNSW

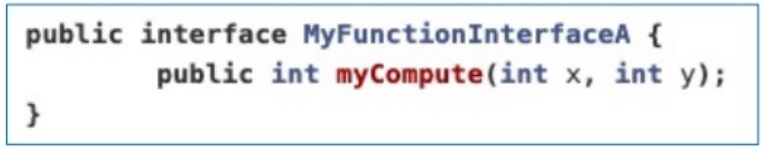
Java Lambda Expressions

- Lambda expressions allow us to
  - easily define anonymous methods,
  - treat code as data and
  - pass functionality as method argument.
- An anonymous inner class with only one method can be replaced by a lambda expression.
- Lambda expressions can be used to implement an interface with only one abstract method. Such interfaces are called *Functional Interface*s.
- Lambda expressions offer *functions as objects* - a feature from functional programming.
- Lambda expressions are less verbose and offers more flexibility.

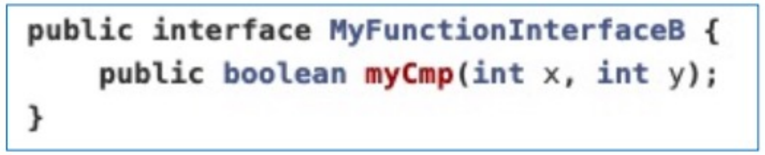
Java Lambda Expressions - Syntax

A lambda expression consists of the following:

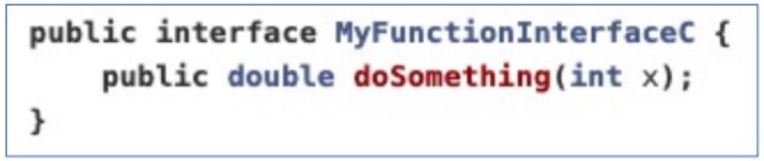
- ❖ A comma-separated list of formal parameters enclosed in parentheses. No need to provide data types, they will be inferred. For only one parameter, we can omit the parentheses.
- ❖ The arrow token, **->**
- ❖ A body, which consists of a single expression or a statement block.



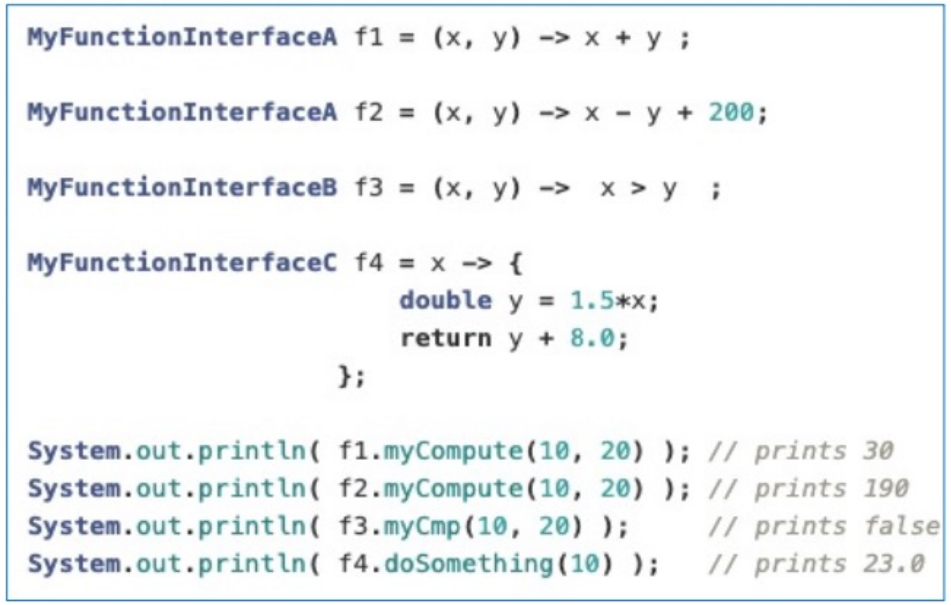
```java
public interface MyFunctionInterfaceA {
        public int myCompute(int x, int y);
}
```



```java
public interface MyFunctionInterfaceB {
    public boolean myCmp(int x, int y);
}
```



```java
public interface MyFunctionInterfaceC {
    public double doSomething(int x);
}
```



```java
MyFunctionInterfaceA f1 = (x, y) -> x + y ;

MyFunctionInterfaceA f2 = (x, y) -> x - y + 200;

MyFunctionInterfaceB f3 = (x, y) ->  x > y  ;

MyFunctionInterfaceC f4 = x -> {
                        double y = 1.5*x;
                        return y + 8.0;
                    };

System.out.println( f1.myCompute(10, 20) ); // prints 30
System.out.println( f2.myCompute(10, 20) ); // prints 190
System.out.println( f3.myCmp(10, 20) );     // prints false
System.out.println( f4.doSomething(10) );   // prints 23.0
```

Method References

We can treat an existing method as an instance of a Functional Interface.

There are multiple ways to refer to a method, using `::` operator.

❖ A **static** method (`ClassName::methName`)

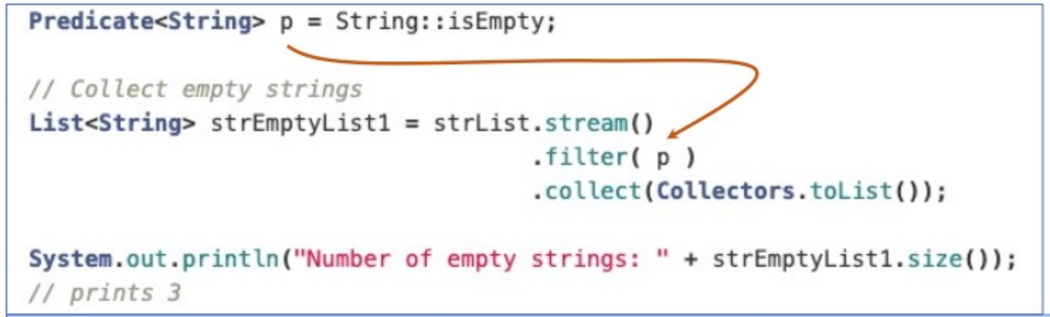
❖ An **instance** method of a particular object (`instanceRef::methName`) or (`ClassName::methName`)

❖ A class **constructor** reference (`ClassName::new`)

❖ Etc.

Function Interfaces in Java

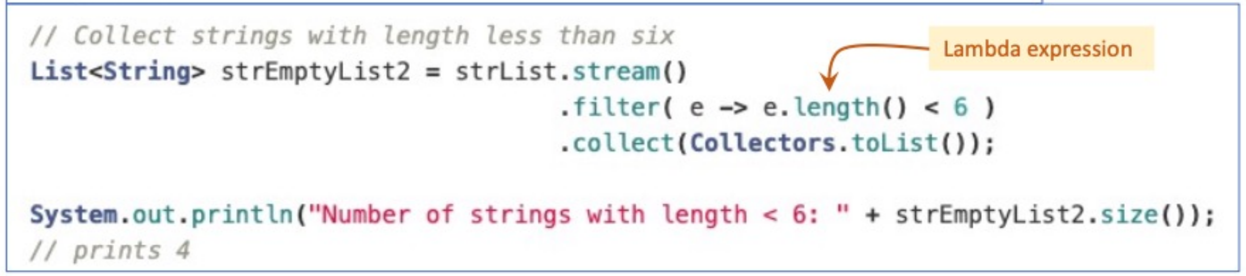
- Functional interfaces, in the package `java.util.function,` provide predefined target types for lambda expressions and method references.
- Each functional interface has a single abstract method, called the functional method for that functional interface, to which the lambda expression's parameter and return types are matched or adapted.
- Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, etc. For example,



```java
Predicate<String> p = String::isEmpty;

// Collect empty strings
List<String> strEmptyList1 = strList.stream()
                                    .filter( p )
                                    .collect(Collectors.toList());

System.out.println("Number of empty strings: " + strEmptyList1.size());
// prints 3
```



```java
// Collect strings with length less than six
List<String> strEmptyList2 = strList.stream()
                                    .filter( e -> e.length() < 6 )
                                    .collect(Collectors.toList());

System.out.println("Number of strings with length < 6: " + strEmptyList2.size());
// prints 4
```

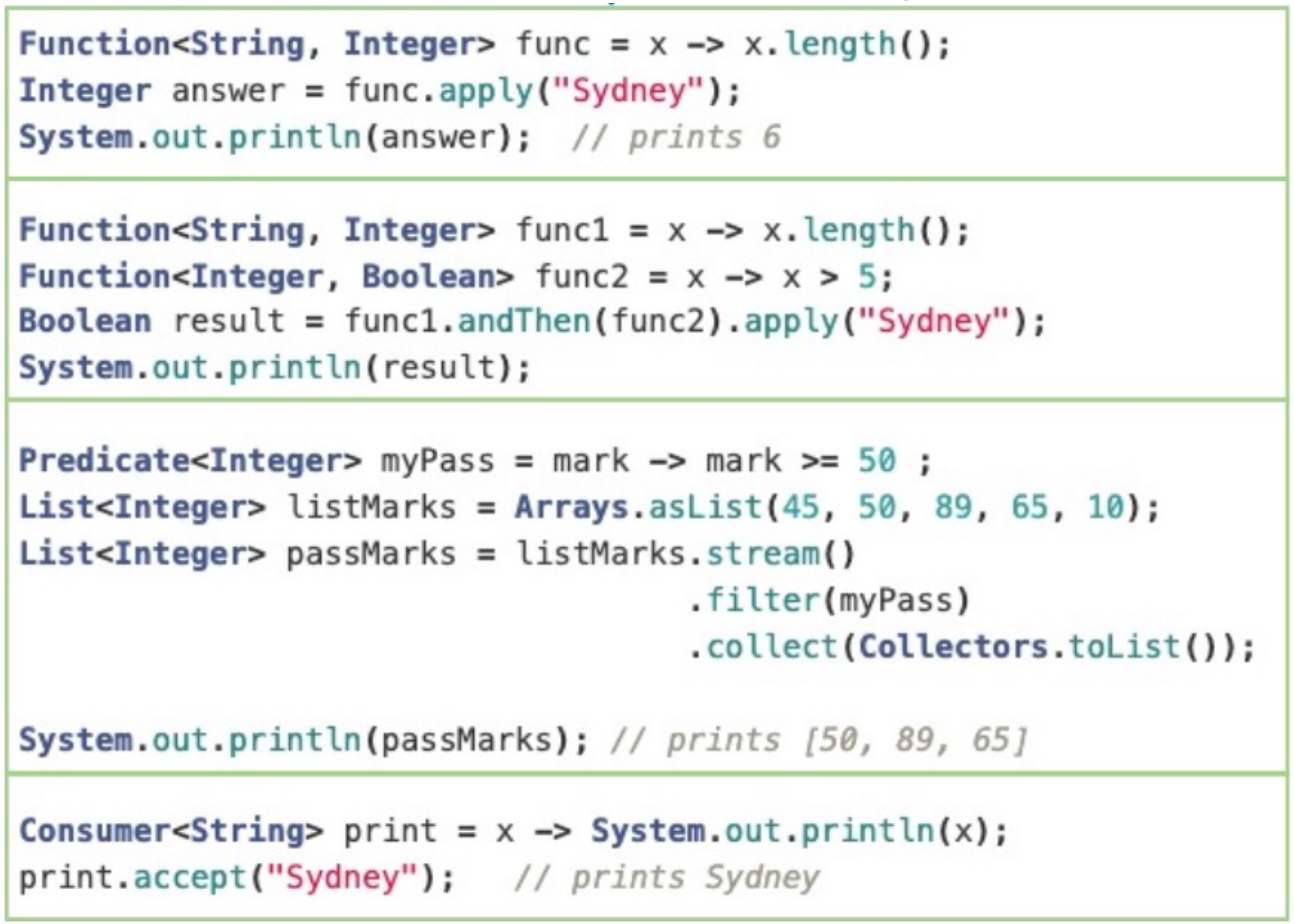
Function Interfaces in Java

- There are several basic *function shapes*, including
  - **Function** (unary function from T to R),
  - **Consumer** (unary function from T to void),
  - **Predicate** (unary function from T to boolean), and
  - **Supplier** (nilary function to R).

- More information at the package summary page

  https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

Function Interfaces in Java: Examples



```java
Function<String, Integer> func = x -> x.length();
Integer answer = func.apply("Sydney");
System.out.println(answer);  // prints 6
```

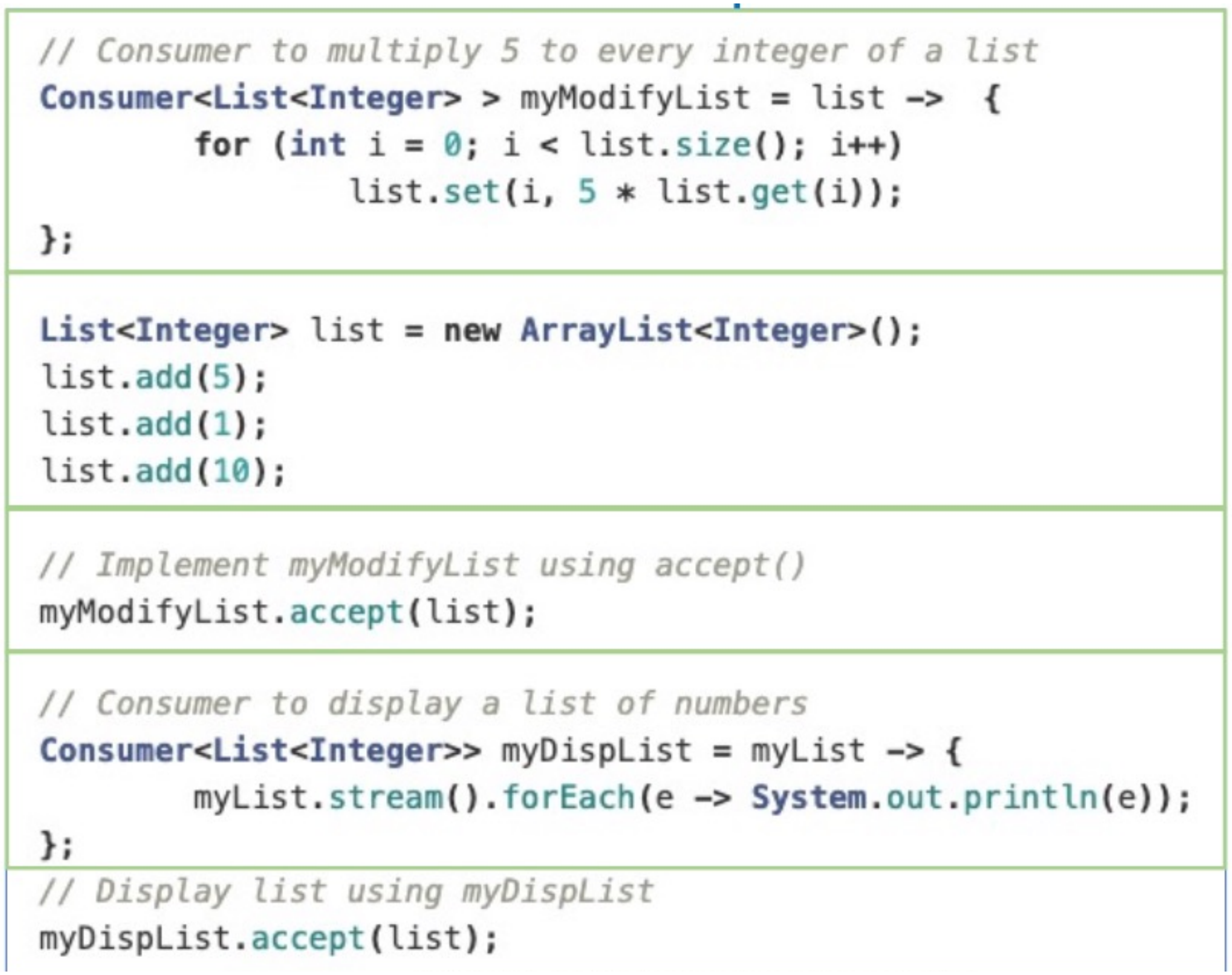
```java
Function<String, Integer> func1 = x -> x.length();
Function<Integer, Boolean> func2 = x -> x > 5;
Boolean result = func1.andThen(func2).apply("Sydney");
System.out.println(result);
```

```java
Predicate<Integer> myPass = mark -> mark >= 50 ;
List<Integer> listMarks = Arrays.asList(45, 50, 89, 65, 10);
List<Integer> passMarks = listMarks.stream()
                                   .filter(myPass)
                                   .collect(Collectors.toList());

System.out.println(passMarks); // prints [50, 89, 65]
```

```java
Consumer<String> print = x -> System.out.println(x);
print.accept("Sydney");   // prints Sydney
```

Function Interfaces in Java: Examples



```
// Consumer to multiply 5 to every integer of a list
Consumer<List<Integer> > myModifyList = list ->  {
        for (int i = 0; i < list.size(); i++)
                list.set(i, 5 * list.get(i));
};
```

```
List<Integer> list = new ArrayList<Integer>();
list.add(5);
list.add(1);
list.add(10);
```

```
// Implement myModifyList using accept()
myModifyList.accept(list);
```

```
// Consumer to display a list of numbers
Consumer<List<Integer>> myDispList = myList -> {
        myList.stream().forEach(e -> System.out.println(e));
};
```

```
// Display list using myDispList
myDispList.accept(list);
```

Comparator using Lambda Expression: Example

```
//Using an anonymous inner class
Comparator<Customer> myCmpAnonymous  = new Comparator<Customer>() {
        @Override
        public int compare(Customer o1, Customer o2) {
                return o1.getRewardsPoints() -  o2.getRewardsPoints() ;
        }
} ;
custA.sort( myCmpAnonymous );
```

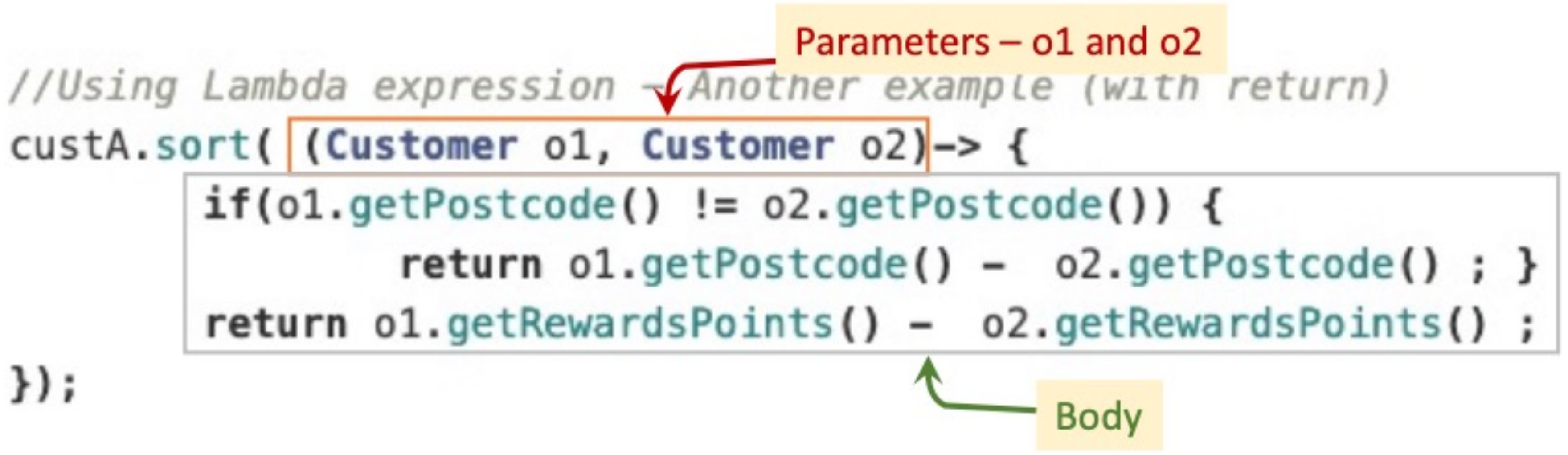
Only one line!

```
//Using Lambda expression - simple example (only one line)
custA.sort((Customer o1, Customer o2)->o1.getRewardsPoints() - o2.getRewardsPoints());
```

```
custA.forEach( (cust) -> System.out.println(cust) );
```

Print using Lambda expression

Comparator using Lambda Expression: Another Example



```
//Using Lambda expression - Another example (with return)
custA.sort( (Customer o1, Customer o2)-> {
        if(o1.getPostcode() != o2.getPostcode()) {
                return o1.getPostcode() -  o2.getPostcode() ; }
        return o1.getRewardsPoints() -  o2.getRewardsPoints() ;
});
```

Pipelines and Streams

- A **pipeline** is a sequence of aggregate operations.
- The following example prints the male members contained in the collection `roster` with a pipeline that consists of the aggregate operations `filter` and `forEach`:

Using pipeline and aggregate ops:

```
roster
     .stream()
     .filter( e -> e.getGender() == Person.Sex.MALE )
     .forEach( e -> System.out.println(e.getName()) );
```

Traditional approach, using a `for-each` loop:

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

- Please note that, in a pipeline, operations are loosely coupled, they only rely on their incoming streams and can be easily rearranged/replaced by other suitable operations.
- Just to clarify, the "." (dot) operator in the above syntax has a very different meaning to the "." (dot) operator used with an instance or a class.

Pipelines and Streams

- A **pipeline** contains the following components:
  - A **source**: This could be a collection, an array, a generator function, or an I/O channel. Such as *roster* in the example.
  - Zero or **more** **intermediate operations**. An intermediate operation, such as **filter**, produces a new stream.
- A **stream** is a sequence of elements. The method **stream** creates a stream from a collection (*roster*).
- The **filter** operation returns a new stream that contains elements that match its predicate. The filter operation in the example returns a stream that contains all male members in the collection roster.
- A **terminal** operation. A terminal operation, such as forEach, produces a non-stream result, such as a primitive value (like a double value), a collection, or in the case of forEach, no value at all.

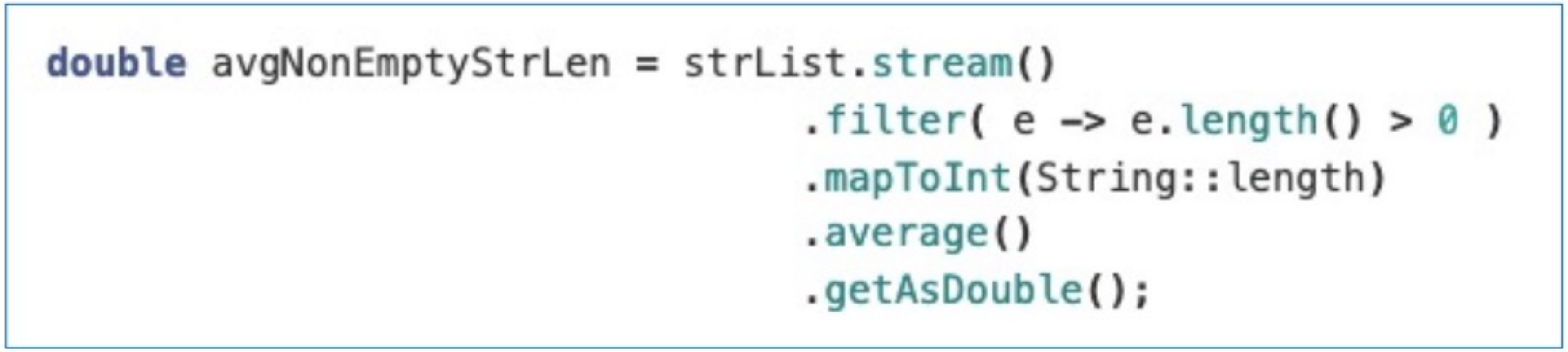
```
roster
    .stream()
    .filter( e -> e.getGender() == Person.Sex.MALE )
    .forEach( e -> System.out.println(e.getName()) );
```

Pipelines and Streams: Example

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

- The above example calculates the average age of all male members contained in the collection roster with a pipeline that consists of the aggregate operations filter, mapToInt, and average.
- The mapToInt operation returns a new stream of type IntStream (which is a stream that contains only integer values). The operation applies the function specified in its parameter to each element in a particular stream.
- As expected, the average operation calculates the average value of the elements contained in a stream of type IntStream.
- There are many terminal operations such as average that return one value by combining the contents of a stream. These operations are called reduction operations; see the section Reduction for more information at https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html

Pipelines and Streams: Another Example



```java
double avgNonEmptyStrLen = strList.stream()
                                  .filter( e -> e.length() > 0 )
                                  .mapToInt(String::length)
                                  .average()
                                  .getAsDouble();
```

End