Edwin Maraví
emaravi@cjavaperu.com

CJAVA

Misión

Nuestro equipo trabaja para integrar la tecnología Java en la sociedad como solución a todas sus necesidades.

CJAVA
siempre para apoyarte

Visión

Poder aportar al desarrollo del País usando tecnología Java.

Servicios Académicos

Programer (80 horas - Certificación Java 11)

[Certificado: Java Programer]

Developer (80 horas - Spring FrameWork y Angular)

[Certificado: Java Developer]

Expert (80 horas – Microservicios y DevOps)

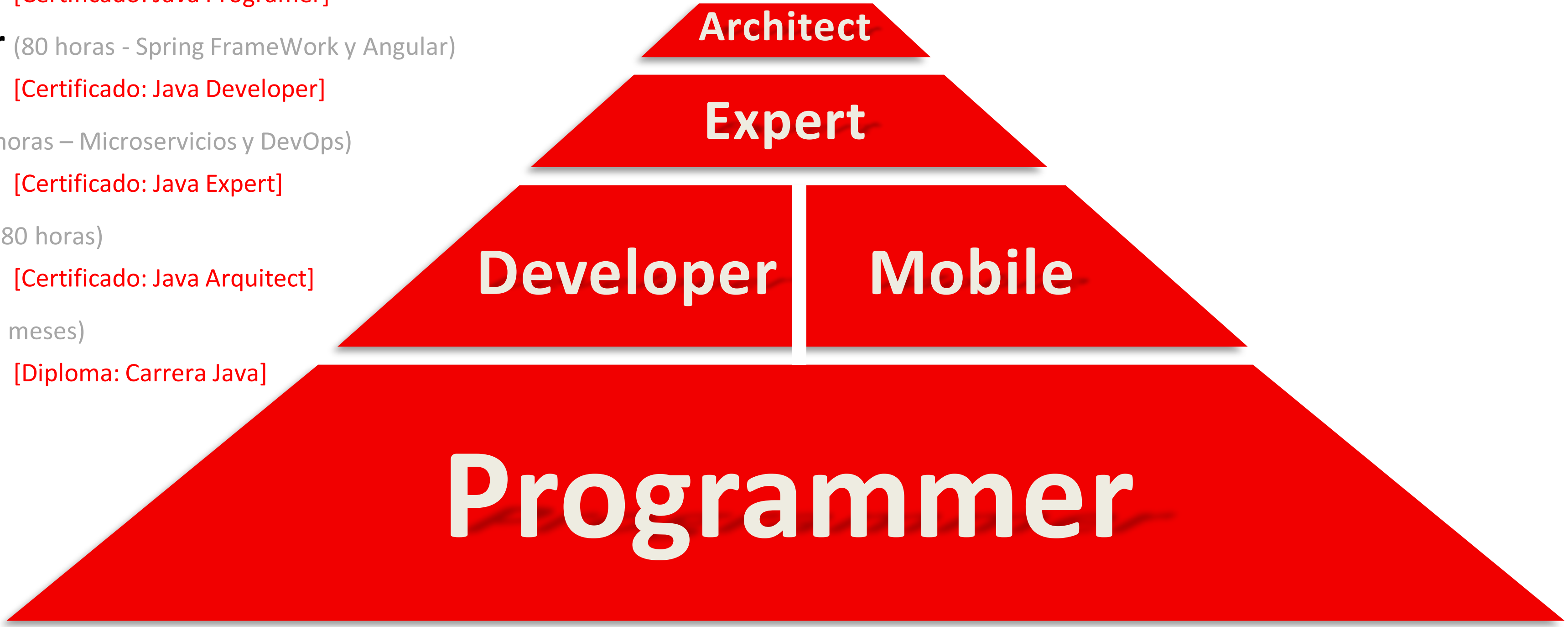
[Certificado: Java Expert]

Architect (80 horas)

[Certificado: Java Arquitect]

Carrera (12 meses)

[Diploma: Carrera Java]



Architect

Expert

Developer

Mobile

Programmer

CJAVA
siempre para apoyarte

Quienes Somos

Somos una organización orientada a desarrollar, capacitar e investigar tecnología JAVA a través de un prestigioso staff de profesionales a nivel nacional.

CJAVA

Contáctenos

Av. Arenales 395 oficina 405
Santa Beatriz - Lima 01 - Perú

Teléfono: 433-6948

RPC / WhatsApp: 932 656 459

Email: info@cjavaperu.com

Síguenos

/cjava.peru.1

/cjava_peru

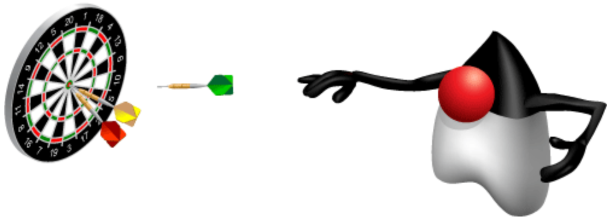
/cjavaperu

Collections, Streams y Filters

Objectives

After completing this lesson, you should be able to:

- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods
- Define pipelines in terms of lambdas and collections



Collections, Streams, and Filters

- Iterate through collections using forEach
- Streams and Filters

The Person Class

- Person class
  - Attributes like name, age, address, etc.
- Class created by using the Builder pattern
  - Generates a collection persons for examples
- RoboCall Example
  - An app for contacting people via mail, phone, email
  - Given a list of people query for certain groups
  - Used for test and demo
- Groups queried for
  - Drivers: Persons over the age of 16
  - Draftees: Male persons between 18 and 25 years old
  - Pilots: Persons between 23 and 65 years old

Person Properties

A Person has the following properties:

```
public class Person {
    private String givenName;
    private String surName;
    private int age;
    private Gender gender;
    private String eMail;
    private String phone;
    private String address;
    private String city;
    private String state;
    private String code;
```

Builder Pattern

Allows object creation by using method chaining

- Easier-to-read code
- More flexible object creation
- Object returns itself
- A fluent approach

Example

```
people.add(
 new Person.Builder()
    .givenName("Betty")
    .surName("Jones")
    .age(85)
    .gender(Gender.FEMALE)
    .email("betty.jones@example.com")
    .phoneNumber("211-33-1234")
    .build()
);
```

Collection Iteration and Lambdas

RoboCall06 Iterating with forEach

```
public class RoboCallTest06 {

 public static void main(String[] args){

  List<Person> pl = Person.createShortList();

  System.out.println("\n=== Print List ===");
  pl.forEach(p -> System.out.println(p));

 }
}
```

RoboCallTest07: Stream and Filter

```
public class RoboCallTest07 {

  public static void main(String[] args){

    List<Person> pl = Person.createShortList();
    RoboCall05 robo = new RoboCall05();

    System.out.println("\n=== Calling all Drivers Lambda ===");
    pl.stream()
        .filter(p -> p.getAge() >= 23 && p.getAge() <= 65)
        .forEach(p -> robo.roboCall(p));

  }
}
```

RobocallTest08: Stream and Filter Again

```
public class RoboCallTest08 {

  public static void main(String[] args){

    List<Person> pl = Person.createShortList();
    RoboCall05 robo = new RoboCall05();

    // Predicates
    Predicate<Person> allPilots =
        p -> p.getAge() >= 23 && p.getAge() <= 65;

    System.out.println("\n=== Calling all Drivers Variable ===");
    pl.stream().filter(allPilots)
        .forEach(p -> robo.roboCall(p));
  }
```

SalesTxn Class

- Class used in examples and practices to follow
- Stores information about sales transactions
  - Seller and buyer
  - Product quantity and price
- Implemented with a Builder class
- Buyer class
  - Simple class to represent buyers and their volume discount level
- Helper enums
  - BuyerClass: Defines volume discount levels
  - State: Lists the states where transactions take place
  - TaxRate: Lists the sales tax rates for different states

Java Streams

- Streams
  - `java.util.stream`
  - A sequence of elements on which various methods can be chained
- Method chaining
  - Multiple methods can be called in one statement
- Stream characteristics
  - They are immutable.
  - After the elements are consumed, they are no longer available from the stream.
  - A chain of operations can occur only once on a particular stream (a pipeline).
  - They can be serial (default) or parallel.

The Filter Method

- The Stream class converts collection to a pipeline
  - Immutable data
  - Can only be used once and then tossed
- Filter method uses Predicate lambdas to select items.
- Syntax:

```
System.out.println("\n== CA Transations Lambda ==");
tList.stream()
    .filter(t -> t.getState().equals("CA"))
    .forEach(SalesTxn::printSummary);
```

Method References

In some cases, the lambda expression merely calls a class method.

```
.forEach(t -> t.printSummary())
```

- Alternatively, you can use a method reference

```
.forEach(SalesTxn::printSummary));
```

- You can use a method reference in the following situations:
  - Reference to a static method
    - `ContainingClass::staticMethodName`
  - Reference to an instance method
  - Reference to an instance method of an arbitrary object of a particular type (for example, `String::compareToIgnoreCase`)
  - Reference to a constructor
    - `ClassName::new`

Method Chaining

- Pipelines allow method chaining (like a builder).
- Methods include filter and many others.
- For example:

```
tList.stream()
    .filter(t -> t.getState().equals("CA"))
    .filter(t -> t.getBuyer().getName()
        .equals("Acme Electronics"))
    .forEach(SalesTxn::printSummary);
```

Method Chaining

- You can use compound logical statements.
- You select what is best for the situation.

```
        System.out.println("\n== CA Transations for ACME ==");
        tList.stream()
            .filter(t -> t.getState().equals("CA")  &&
                t.getBuyer().getName().equals("Acme Electronics"))
            .forEach(SalesTxn::printSummary);

        tList.stream()
            .filter(t -> t.getState().equals("CA"))
            .filter(t -> t.getBuyer().getName()
                .equals("Acme Electronics"))
            .forEach(SalesTxn::printSummary);
```

Pipeline Defined

- A stream pipeline consists of:
  - A source
  - Zero or more intermediate operations
  - One terminal operation
- Examples
  - Source: A Collection (could be a file, a stream, and so on)
  - Intermediate: Filter, Map
  - Terminal: `forEach`

Summary

After completing this lesson, you should be able to:

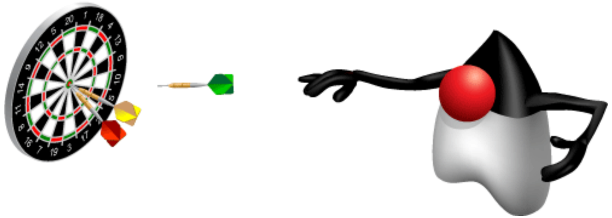
- Describe the Builder pattern
- Iterate through a collection by using lambda syntax
- Describe the Stream interface
- Filter a collection by using lambda expressions
- Call an existing method by using a method reference
- Chain multiple methods together
- Define pipelines in terms of lambdas and collections

Lambda Built-in Functional Interfaces

Objectives

After completing this lesson, you should be able to:

- List the built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces



Built-in Functional Interfaces

- Lambda expressions rely on functional interfaces
  - Important to understand what an interface does
  - Concepts make using lambdas easier
- Focus on the purpose of main functional interfaces
- Become aware of many primitive variations
- Lambda expressions have properties like those of a variable
  - Use when needed
  - Can be stored and reused

The `java.util.function` Package

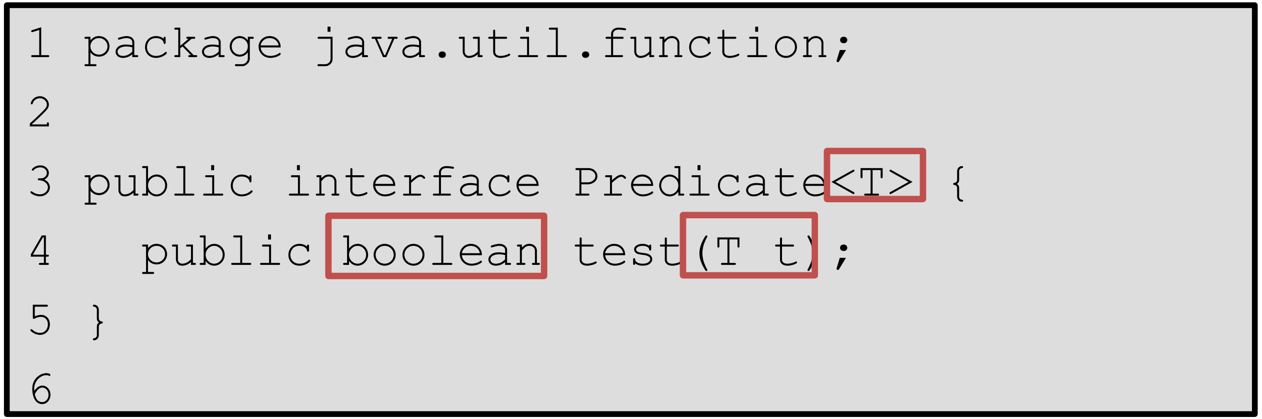
- `Predicate`: An expression that returns a `boolean`
- `Consumer`: An expression that performs operations on an object passed as argument and has a void return type
- `Function`: Transforms a T to a U
- `Supplier`: Provides an instance of a T (such as a factory)
- Primitive variations
- Binary variations

Example Assumptions

The following two declarations are assumed for the examples that follow:

```
List<SalesTxn> tList = SalesTxn.createTxnList();
SalesTxn first = tList.get(0);
```

Predicate



```
package java.util.function;

public interface Predicate<T> {
   public boolean test(T t);
}

```

Predicate: Example

```java
Predicate<SalesTxn> massSales =
    t -> t.getState().equals(State.MA);

System.out.println("\n== Sales - Stream");
tList.stream()
    .filter(massSales)
    .forEach(t -> t.printSummary());

System.out.println("\n== Sales - Method Call");
for(SalesTxn t:tList){
    if (massSales.test(t)){
        t.printSummary();
    }
}
```

Consumer

```java
package java.util.function;

public interface Consumer<T> {

    public void accept(T t);

}
```

Consumer: Example

```java
    Consumer<SalesTxn> buyerConsumer = t ->
        System.out.println("Id: " + t.getTxnId()
            + " Buyer: " + t.getBuyer().getName());

    System.out.println("== Buyers - Lambda");
    tList.stream().forEach(buyerConsumer);

    System.out.println("== First Buyer - Method");
    buyerConsumer.accept(first);
```

Function

```
package java.util.function;

public interface Function<T,R> {

      public R apply(T t);
}

```

Function: Example

```
 Function<SalesTxn, String> buyerFunction =
   t -> t.getBuyer().getName();

 System.out.println("\n== First Buyer");
 System.out.println(buyerFunction.apply(first));
}
```

Supplier

```java
package java.util.function;

public interface Supplier<T> {

    public T get();
}

```

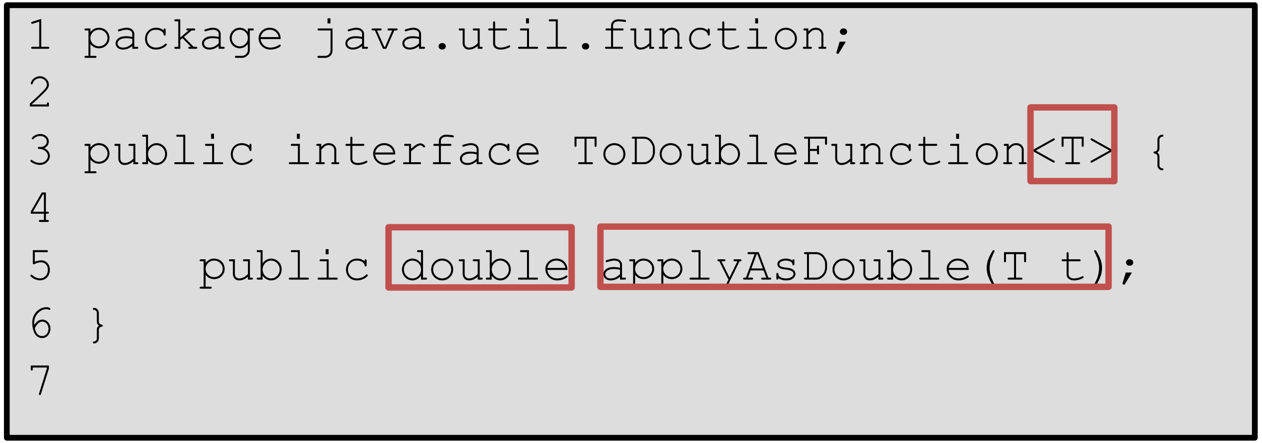
Supplier: Example

```
        List<SalesTxn> tList = SalesTxn.createTxnList();
        Supplier<SalesTxn> txnSupplier =
            () -> new SalesTxn.Builder()
                .txnId(101)
                .salesPerson("John Adams")
.buyer(Buyer.getBuyerMap().get("PriceCo"))
                .product("Widget")
                .paymentType("Cash")
                .unitPrice(20)
//... Lines ommited
                .build();

        tList.add(txnSupplier.get());
        System.out.println("\n== TList");
        tList.stream().forEach(SalesTxn::printSummary);
```

Primitive Interface

- Primitive versions of all main interfaces
  - Will see these a lot in method calls
- Return a primitive
  - Example: `ToDoubleFunction`
- Consume a primitive
  - Example: `DoubleFunction`
- Why have these?
  - Avoids auto-boxing and unboxing

Return a Primitive Type

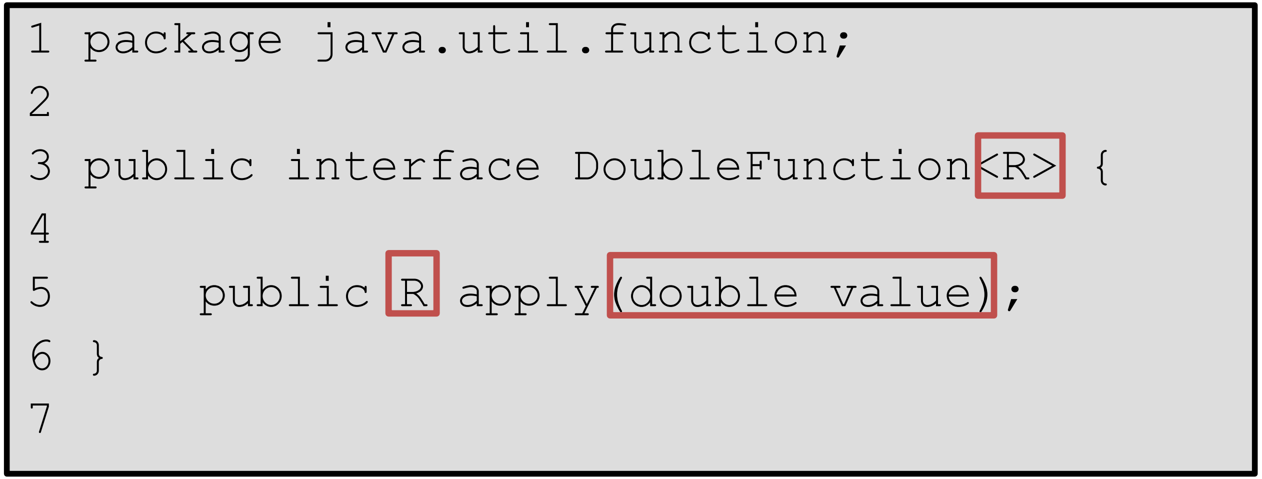


```
package java.util.function;

public interface ToDoubleFunction<T> {

    public double applyAsDouble(T t);
}

```

Return a Primitive Type: Example

```
    ToDoubleFunction<SalesTxn> discountFunction =
        t -> t.getTransactionTotal()
            * t.getDiscountRate();

    System.out.println("\n== Discount");
    System.out.println(
        discountFunction.applyAsDouble(first));
```

Process a Primitive Type

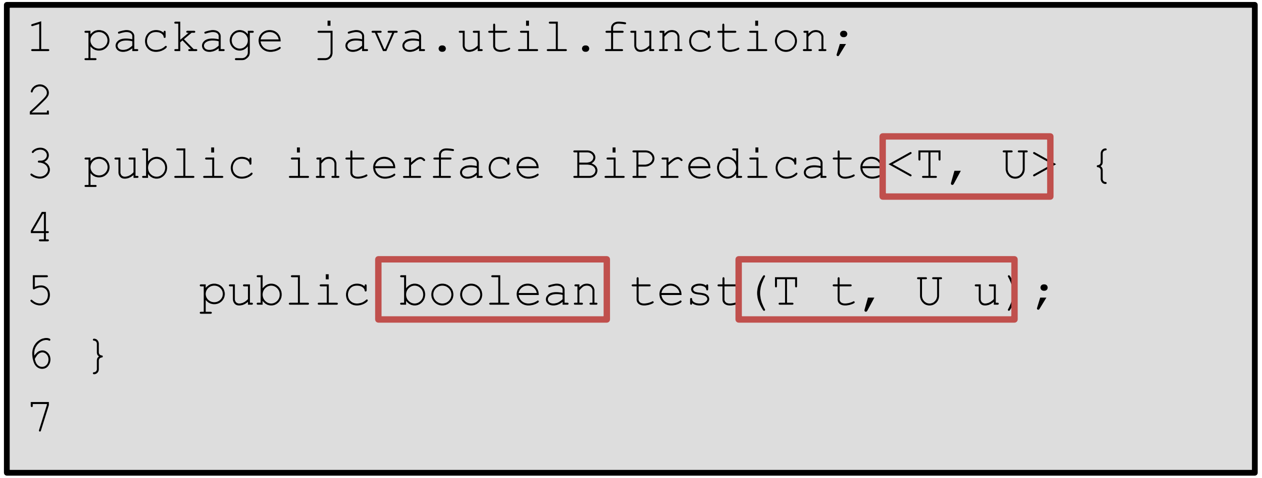


```java
package java.util.function;

public interface DoubleFunction<R> {

    public R apply(double value);
}

```

Process Primitive Type: Example

```
A06DoubleFunction test = new A06DoubleFunction();

DoubleFunction<String> calc =
      t -> String.valueOf(t * 3);

String result = calc.apply(20);
System.out.println("New value is: " + result);
```

Binary Types



```
package java.util.function;

public interface BiPredicate<T, U> {

     public boolean test(T t, U u);
}

```

Binary Type: Example

```
List<SalesTxn> tList = SalesTxn.createTxnList();
SalesTxn first = tList.get(0);
String testState = "CA";

BiPredicate<SalesTxn,String> stateBiPred =
  (t, s) -> t.getState().getStr().equals(s);

System.out.println("\n== First is CA?");
System.out.println(
  stateBiPred.test(first, testState));
```

Unary Operator

```
package java.util.function;

public interface UnaryOperator<T>
extends Function<T,T> {
    @Override
    public T apply(T t);
}
```

UnaryOperator: Example

If you need to pass in something and return the same type, use the UnaryOperator interface.

```
      UnaryOperator<String> unaryStr =
          s -> s.toUpperCase();

        System.out.println("== Upper Buyer");
        System.out.println(
          unaryStr.apply(first.getBuyer().getName()));
```

Wildcard Generics Review

- Wildcards for generics are used extensively.
- `? super T`
  - This class and any of its super types
- `? extends T`
  - This class and any of its subtypes

Summary

After completing this lesson, you should be able to:

- List the built-in interfaces included in `java.util.function`
- Use primitive versions of base interfaces
- Use binary versions of base interfaces

Lambda Operations

Objectives

After completing this lesson, you should be able to:

- Extract data from an object by using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class

Streams API

- Streams
  - `java.util.stream`
  - A sequence of elements on which various methods can be chained
- The Stream class converts collection to a pipeline.
  - Immutable data
  - Can only be used once
  - Method chaining
- Java API doc *is your friend*
- Classes
  - `DoubleStream`, `IntStream`, `LongStream`

Types of Operations

- Intermediate
  - `filter() map() peek()`
- Terminal
  - `forEach() count() sum() average() min() max() collect()`
- Terminal short-circuit
  - `findFirst() findAny() anyMatch() allMatch() noneMatch()`

Extracting Data with Map

`map(Function<? super T,? extends R> mapper)`

- A map takes one `Function` as an argument.
  - A `Function` takes one generic and returns something else.
- Primitive versions of `map`
  - `mapToInt()` `mapToLong()` `mapToDouble()`

Taking a Peek

`peek(Consumer<? super T> action)`

- The peek method performs the operation specified by the lambda expression and returns the elements to the stream.
- Great for printing intermediate results

Search Methods: Overview

- `findFirst()`
  - Returns the first element that meets the specified criteria
- `allMatch()`
  - Returns `true` if all the elements meet the criteria
- `noneMatch()`
  - Returns `true` if none of the elements meet the criteria
- All of the above are short-circuit terminal operations.

Search Methods

- Nondeterministic search methods
  - Used for nondeterministic cases. In effect, situations where parallel is more effective.
  - Results may vary between invocations.
- `findAny()`
  - Returns the first element found that meets the specified criteria
  - Results may vary when performed in parallel.
- `anyMatch()`
  - Returns true if any elements meet the criteria
  - Results may vary when performed in parallel.

Optional Class

- `Optional<T>`
  - A container object that may or may not contain a non-null value
  - If a value is present, `isPresent()` returns true.
  - `get()` returns the value.
  - Found in `java.util`.
- Optional primitives
  - `OptionalDouble OptionalInt OptionalLong`

Lazy Operations

- Lazy operations:
  - Can be optimized
  - Perform only required operations

```
== First CO Bonus ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
```

```
== CO Bonuses ==
Stream start
Stream start
Stream start
Stream start
Stream start
Stream start
Executives
CO Executives
  Bonus paid: $7,200.00
Stream start
Executives
CO Executives
  Bonus paid: $6,600.00
Stream start
Executives
CO Executives
  Bonus paid: $8,400.00
```

Stream Data Methods

- **`count()`**
  - Returns the count of elements in this stream
- **`max(Comparator<? super T> comparator)`**
  - Returns the maximum element of this stream according to the provided `Comparator`
- **`min(Comparator<? super T> comparator)`**
  - Returns the minimum element of this stream according to the provided `Comparator`

Performing Calculations

- `average()`
  - Returns an optional describing the arithmetic mean of elements of this stream
  - Returns an empty optional if this stream is empty
  - Type returned depends on primitive class.
- `sum()`
  - Returns the sum of elements in this stream
  - Methods are found in primitive streams:
    - `DoubleStream`, `IntStream`, `LongStream`

Sorting

- `sorted()`
  - Returns a stream consisting of the elements sorted according to natural order
- `sorted(Comparator<? super T> comparator)`
  - Returns a stream consisting of the elements sorted according to the `Comparator`

Comparator Updates

`comparing(Function<? super T,? extends U> keyExtractor)`

Allows you to specify any field to sort on based on a method reference or lambda

Primitive versions of the Function also supported

`thenComparing(Comparator<? super T> other)`

Specify additional fields for sorting.

`reversed()`

Reverse the sort order by appending to the method chain.

Saving Data from a Stream

`collect(Collector<? super T,A,R> collector)`

- Allows you to save the result of a stream to a new data structure
- Relies on the `Collectors` class
- Examples
  - `stream().collect(Collectors.toList());`
  - `stream().collect(Collectors.toMap());`

CJAVA
siempre para apoyarte

Collectors Class

averagingDouble(ToDoubleFunction<? super T> mapper)

Produces the arithmetic mean of a double-valued function applied to the input elements

groupingBy(Function<? super T,? extends K> classifier)

A "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a map

joining()

Concatenates the input elements into a String, in encounter order

partitioningBy(Predicate<? super T> predicate)

Partitions the input elements according to a Predicate

Quick Streams with `Stream.of`

The `Stream.of` method allows you to easily create a stream.

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

  Stream.of("Monday", "Tuesday","Wedensday", "Thursday")
    .filter(s -> s.startsWith("T"))
    .forEach(s -> System.out.println("Matching Days: " + s));
}
```

Flatten Data with `flatMap`

Use the `flatMap` method to flatten data in a stream.

```
Path file = new File("tempest.txt").toPath();

try{

  long matches = Files.lines(file)
    .flatMap(line -> Stream.of(line.split(" ")))
    .filter(word -> word.contains("my"))
    .peek(s -> System.out.println("Match: " + s))
    .count();

  System.out.println("# of Matches: " + matches);
```

Summary

After completing this lesson, you should be able to:

- Extract data from an object using map
- Describe the types of stream operations
- Describe the Optional class
- Describe lazy processing
- Sort a stream
- Save results to a collection by using the `collect` method
- Group and partition data by using the `Collectors` class

CJAVA

siempre para apoyarte

#CJavaNoPara

Gracias

CJAVA

Contáctenos

📍 Av. Arenales 395 oficina 405
Santa Beatriz - Lima 01 - Perú

☎ Teléfono: 433-6948

📱 RPC / WhatsApp: 932 656 459

✉ Email: info@cjavaperu.com

Síguenos

/cjava.peru.1

/cjava_peru

/cjavaperu