JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

1. Classes and Instances
2. The Methods
3. The Constructors
4. Static Elements
5. Initialization sections
6. Package
7. Inheritance and Polymorphism
8. Abstract classes and Interfaces
9. String processing
10. Wrapper classes for primitive types
11. Exceptions and Assertions
12. Nested classes
13. Enums
14. Generics
15. Collections
16. Method overload resolution
17. Multithreads
18. Core Java classes
19. Object Oriented Design
20. **Functional Programming**

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Lists and Sets Traversing

- The traversing elements of Collection is a frequently action in programming.

See iterable\Car & ListCarIteration & SetCarIteration

- In JDK 8 the **default** Spliterator<T> spliterator() method was added to Iterable<T> interface for Spliterator object getting that allow to split collections into parts while iterating.

```
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<? super T> action);
    default void forEachRemaining(Consumer<? super T> action) {
        do { } while (tryAdvance(action));
    }
    Spliterator<T> trySplit();
…
}
```

See iterable\Car & ListCarSpliteration & ListCarSpliterationMultithreading

Map Traversing

- Java maps are not iterable.
- However in JDK 8 the **default void** forEach(BiConsumer<? **super** K, ? **super** V> action) method has been provided for traversal.
- The BiConsumer object-parameter processes the map entry's key and value.

```
default void forEach(BiConsumer<? super K, ? super V> action) {
    Objects.requireNonNull(action);
    for (Map.Entry<K, V> entry : entrySet()) {
        K k;
        V v;
        try {
            k = entry.getKey();
            v = entry.getValue();
        } catch (IllegalStateException ise) {
            throw new ConcurrentModificationException(ise);        }
        action.accept(k, v);
    }  }
```

See iterable\MapIteration

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Removing Elements from a Collection

- The Collection<E> interface **removeIf** method can be used to remove elements from a collections whose Iterator supports the remove operation.

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```

If the **filter** returns true for an element, that element is removed from the collection.

See collection\RemoveElementsExample

Replacing the Elements of List or Map

- All the elements in a list or map can be modified using the List<E> or Map<K, V> interfaces default **replaceAll** method and a UnaryOperator or BiFunction that specified how to perform the modification.

List<E> interface

```
default void replaceAll(UnaryOperator<E> operator) {
    …
}
```

Map<K, V> interface

```
default void replaceAll(BiFunction<? super K, ? super V, ? extends
V> function) {
    …
}
```

See collection\ReplaceElementsExample

Map Computations

- The Map<K, V> interface provides the following methods which perform inline computations on selected by key entry in a map.
- BiFunction or Function implementations defines these computations.

```
default V compute(K key, BiFunction<? super K, ? super V,
                                    ? extends V> remappingFunction) {…}

default V computeIfPresent(K key, BiFunction<? super K, ? super V,
                                    ? extends V> remappingFunction) {…}

default V computeIfAbsent(K key, Function<? super K,
                                    ? extends V> mappingFunction) {…}
```

computeIfAbsent method accepts a **Function** object instead of a **BiFunction**, since no existing value needs to be processed.

Map Computations

- The **compute** method calculates entry value using the specified **BiFunction** for the specified key.
- Method **compute** returns *new value* or *null* if no such key.
- If the **BiFunction** results in *null*, the entry is removed from the map.

See collection\MapComputations - 1st part

Map Computations

- The **computeIfPresent** method calculates entry value using the specified **BiFunction** for the specified key if the entry exists and its value is not null.
- If the value is null, nothing is done.
- Method **computeIfPresent** returns *new value* or *null* if no such key.
- If the **BiFunction** results in *null*, the entry is removed from the map.

See collection\MapComputations - 2nd part

Map Computations

- The **computeIfAbsent** method calculates entry value using the specified **Function** for the specified key and adds this entry to the map if the entry with such key does not exist.
- If the entry with specified key exists and value is not null, it is not rewrite.
- If the entry with specified key exists and value is null, method calculates entry new value using the specified **Function**.
- Method **computeIfAbsent** returns *the current or computed value* or *null* if computed value is *null*.
- If the **Function** results in *null*, no entry is added.

See collection\MapComputations - 3d part

Map Merging

- The Map interface's **merge(Key, Value, BiFunction)** method is used to modify an existing value for a specified key by merging a new value with it according to a mapping **BiFunction**.
- If the entry **Key** - **Value** does not exist, a new entry is created with the specified **Key** and **Value**.
- If **Key** exists with another **Value**, the method uses remapping function to update the value by function result.
- If the mapping function results in a *null* value, the entry is removed from the map.

```
default V merge(K key, V value, BiFunction<? super V, ? super V,
                                           ? extends V> remappingFunction) {...}
```

See collection\MyClass & MapMerging

Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Comparator Interface

- java.util.Comparator<T> is a functional interface that is used to compare two objects.
- Its functional method, called **compare**, takes two arguments of type T and returns an integer.

```
@FunctionalInterface
public interface Comparator<T> {
   int compare(T o1, T o2);
...
}
```

The result of the comparison method is expected to be as follows:

- for o1 > o2: a positive integer
- for o1 < o2 : a negative integer
- for o1 = o2: 0

See comparing\CompareObjects

Comparator methods

Method	Description
int **compare**(T o1, T o2)	It compares the first object with second object.
static <T,U extends Comparable<? super U>> Comparator<T> **comparing**(Function<? super T,? extends U> keyExtractor)	It accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator that compares by that sort key.
static <T,U> Comparator<T> **comparing**(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)	It accepts a function that extracts a sort key from a type T, and returns a Comparator that compares by that sort key using the specified Comparator.
static <T> Comparator<T> **comparingDouble**(ToDoubleFunction<? super T> keyExtractor)	It accepts a function that extracts a double sort key from a type T, and returns a Comparator that compares by that sort key.
static <T> Comparator<T> **comparingInt**(ToIntFunction<? super T> keyExtractor)	It accepts a function that extracts an int sort key from a type T, and returns a Comparator that compares by that sort key.
static <T> Comparator<T> **comparingLong**(ToLongFunction<? super T> keyExtractor)	It accepts a function that extracts a long sort key from a type T, and returns a Comparator that compares by that sort key.

Comparator methods

Method	Description
boolean **equals**(Object obj)	It is used to compare the current object with the specified object.
static <T extends Comparable<? super T>> Comparator<T> **naturalOrder**()	It returns a comparator that compares Comparable objects in natural order.
static <T> Comparator<T> **nullsFirst**(Comparator<? super T> comparator)	It returns a comparator that treats null to be less than non-null elements.
static <T> Comparator<T> **nullsLast**(Comparator<? super T> comparator)	It returns a comparator that treats null to be greater than non-null elements.
default Comparator<T> **reversed**()	It returns comparator that contains reverse ordering of the provided comparator.
static <T extends Comparable<? super T>> Comparator<T> **reverseOrder**()	It returns comparator that contains reverse of natural ordering.
default Comparator<T> **thenComparing**(Comparator<? super T> other)	It returns a lexicographic-order comparator with another comparator.

Comparator methods

Method	Description
default <U extends Comparable<? super U>> Comparator<T> **thenComparing**(Function<? super T,? extends U> keyExtractor)	It returns a lexicographic-order comparator with a function that extracts a Comparable sort key.
default <U> Comparator<T> **thenComparing**(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)	It returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator.
default Comparator<T> **thenComparingDouble**(ToDoubleFunction<? super T> keyExtractor)	It returns a lexicographic-order comparator with a function that extracts a double sort key.
default Comparator<T> **thenComparingInt**(ToIntFunction<? super T> keyExtractor)	It returns a lexicographic-order comparator with a function that extracts a int sort key.
default Comparator<T> **thenComparingLong**(ToLongFunction<? super T> keyExtractor)	It returns a lexicographic-order comparator with a function that extracts a long sort key.

Comparator comparing methods

```
/*Accepts a function that extracts a  Comparable sort key from a type T,
  and returns a Comparator<T> that compares by that sort key*/
public static <T, U extends Comparable<? super U>> Comparator<T>
comparing(Function<? super T, ? extends U> keyExtractor) {
    Objects.requireNonNull(keyExtractor);
        return (Comparator<T> & Serializable)
            (c1, c2) ->
keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));     }

/*Accepts a function that extracts a sort key from a type T, and
returns a Comparator<T> that compares by that sort key using the specified
Comparator - used for key not implemented Comparable interface*/
public static <T, U> Comparator<T> comparing(
            Function<? super T, ? extends U> keyExtractor,
            Comparator<? super U> keyComparator) {
        Objects.requireNonNull(keyExtractor);
        Objects.requireNonNull(keyComparator);
        return (Comparator<T> & Serializable) (c1, c2) ->
                keyComparator.compare(keyExtractor.apply(c1),
                                      keyExtractor.apply(c2));      }
```

See comparing\Student & ListWrapper & ComparingMethods

Comparator Interface

- In JDK 8 to Comparator<T> naturalOrder(), reverseOrder(), reversed() methods that return natural and reverse sorting Comparators were added. They throw NullPointerException when compare with null.
- If compared objects may be null we can use methods that return null-safe Comparators.

See comparing\Student & OtherComparatorMethods

Comparator's methods specialization

keyExtractor

- The Java API provides specializations to the Comparator's **comparing** method that extract Integer, Long or Double keys from objects before comparing them based on natural ordering.

static <T> Comparator<T>
comparingInt(ToIntFunction<? super T>**keyExtractor**);

static <T> Comparator<T>
comparingLong(ToLongFunction<? super T>**keyExtractor**);

static <T> Comparator<T>
comparingDouble(ToDoubleFunction<? super T>**keyExtractor**);

See comparing\Student & LongWrapper & ComparingMethodsSpecialization

Comparators chains

- The default **thenComparing** methods return a comparator that is used for further comparison if the calling comparator determines that the objects being compared are equal.

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

default <U extends Comparable<? super U>> Comparator<T>
    thenComparing(Function<? super T, ? extends U> keyExtractor);

default <U> Comparator<T> thenComparing(Function<? super T,
 ? extends U> keyExtractor, Comparator<? super U> keyComparator);
```

See comparing\Student & ComparatorChaining

Specializing Comparators chains

- The Java API provides specializations to the Comparator interface's **thenComparing** signature that accepts a specialized function object. These specializations extract Integer, Long or Double keys from objects before comparing them based on natural ordering.

```
default Comparator<T> thenComparingInt(ToIntFunction<? super T>
                                       keyExtractor);
default Comparator<T> thenComparingLong(ToLongFunction<?
                                        super T> keyExtractor);
default Comparator<T> thenComparingDouble(ToDoubleFunction<?
                                          super T> keyExtractor);
```

See comparing\Student & SpecializedComparatorChaining

Using Comparators to sort Lists

- Comparators will most frequently be used to sort lists and streams. The **List<E>** interface has a method named **sort** which accepts a **Comparator<E>** argument that is used for the comparisons during the sort.

```
default void sort(Comparator<? super E> c)
```

since JDK 8

See comparing\Student & ListSorting

Using Comparators to sort arrays

- Comparators can operate on Java arrays in a fashion similar to lists through the use of the **java.util.Arrays** class methods:

```
public static <T> void sort(T[] a, Comparator<? super T> c)

public static <T> void sort(T[] a, int fromIndex, int toIndex,
                            Comparator<? super T> c)

public static <T> int binarySearch(T[] a, T key,
                                   Comparator<? super T> c)
```

since JDK 1.2

See comparing\Student & ArraySorting

Using Comparators to organize Maps

- Comparators can also be used to sort ordered Map entries.
- Comparators can also be used to compare **Map.Entry** objects. There are four static methods in the **Map.Entry** interface that compare keys and values:

```
public static <K extends Comparable<? super K>, V>
                    Comparator<Map.Entry<K, V>> comparingByKey()

public static <K, V> Comparator<Map.Entry<K, V>>
                    comparingByKey(Comparator<? super K> cmp)

public static <K, V extends Comparable<? super V>>
                    Comparator<Map.Entry<K, V>> comparingByValue()

public static <K, V> Comparator<Map.Entry<K, V>>
                    comparingByValue(Comparator<? super V> cmp)
```

since JDK 8

See comparing\MapOrganizing

BinaryOperator

- **BinaryOperator** has methods to compare two objects with the same type using Comparator:

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {
   public static <T> BinaryOperator<T> minBy(Comparator<? super T>
                                                comparator) {…}
   public static <T> BinaryOperator<T> maxBy(Comparator<? super T>
                                                comparator) {… }
}
```

See comparing\CompareWithBinaryOperator

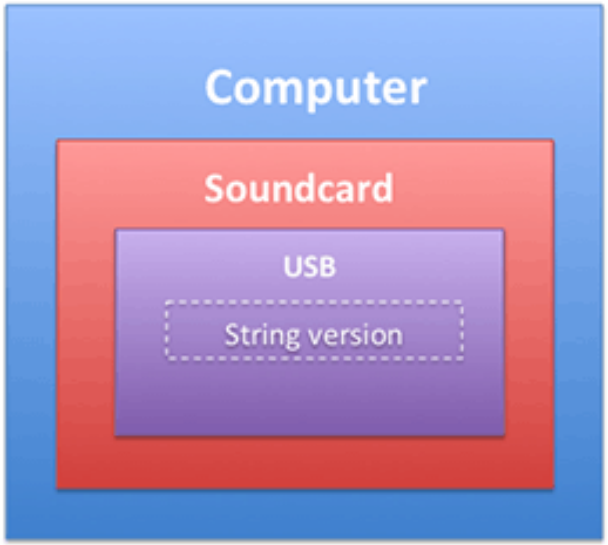
Module contents

Functional Programming

- Functional programming basics
- Functional Interfaces & Lambda Expressions
- Predicates
- Functions
- Operators
- Consumers
- Suppliers
- Method Reference
- Use in Traversing Objects
- Use in Collections
- Use in Comparing Objects
- Use in Optionals
- Use in Streams

Optional

null is often used to denote the absence of a value



In some cases, we do not know in advance whether the reference variable has a not null value

```
String version = computer.getSoundcard().getUSB().getVersion();
```

may be or may not be (returns null)

`NullPointerException - ???`

"I call it ***my billion-dollar mistake***. It was the invention of the null reference in 1965. I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement." - **Tony Hoare**

Developer of the null pointer introduced it in the ALGOL family of languages.

Optional

Checking to prevent null dereferences:

```
String version = "UNKNOWN";
if (computer != null) {
    Soundcard soundcard = computer.getSoundcard();
    if (soundcard != null) {
        Usb usb = soundcard.getUsb();
        if (usb != null) {
            version = usb.getVersion();
        }
    }
}
```

very ugly, decreasing the overall readability

"Using null to represent the absence of a value is a wrong approach"
- Raoul-Gabriel Urma (Cambridge Spark Founder, Modern Java in Action author)

Optional

Groovy:

safe navigation operator (Elvis operator)

```
String version =
computer?.getSoundcard()?.getUSB()?.getVersion()?
: "UNKNOWN";
```

if `computer` is null, or `getSoundcard()` returns null,
or `getUSB()` returns null, version will be `"UNKNOWN"`

- **Haskell** includes a **Maybe** type, which essentially encapsulates an optional value. A value of type **Maybe** can contain either a value of a given type or nothing. There is no concept of a null reference.
- **Scala** has a similar construct called **Option[T]** to encapsulate the presence or absence of a value of type **T**. You then have to explicitly check whether a value is present or not using operations available on the **Option** type, which enforces the idea of "null checking." You can no longer "forget to do it" because it is enforced by the type system.

Optional


```
Optional<Soundcard> soundcardOpt = Optional.ofNullable(soundcard);
System.out.println(soundcardOpt);
                                    //Optional[Soundcard{usb=Usb{version='3.2'}}]
System.out.println(soundcardOpt.get());
                                    //Soundcard{usb=Usb{version='3.2'}}
```

Optional


```
soundcardOpt = Optional.ofNullable(null);
System.out.println(soundcardOpt); //Optional.empty
System.out.println(soundcardOpt.get()); //NoSuchElementException
```

The purpose of the jav.util.Optional<T> class is to provide a type-level solution for representing optional values instead of *null* references.

See optionals\nullref & optionals\optional

Creating an Optional

- Optional is a container object which **may** or **may not** contain a non-null value.

```
public final class Optional<T> {
   private static final Optional<?> EMPTY = new Optional<>(null);
   private final T value;

...
}
```

- The Optional class provides the following static methods which are used **to create Optionals of type T**:

```
public static <T> Optional<T> of(T value)     //throws NPE for value=null
public static <T> Optional<T> ofNullable(T value)  //creates Optional.empty
                                                   //for value=null
public static<T> Optional<T> empty()          //creates Optional.empty
```

See optionals\OptionalCreating

Determining if an Optional is present

- The **isPresent** method returns *true* if the Optional contains a non-null object and returns *false* otherwise.

```
public boolean isPresent() {
    return value != null;
}
```

- In JDK 11, the **isEmpty** method was added. This method returns *true* if the Optional wraps a *null* object and *false* otherwise. This method gives the opposite result of isPresent.

```
public boolean isEmpty() {
    return value == null;
}
```

See optionals\OptionalIsPresent

Retrieving the contents of an Optional

- The **get** method returns the object wrapped by an Optional (NoSuchElementException for *null* value).

```
public T get() {
    if (value == null) {
        throw new NoSuchElementException("No value  present");
    }
    return value;
}
```

- If a value is present, returns the value, otherwise returns other

```
public T orElse(T other)
```

- If a value is present, returns the value, otherwise returns the result produced by the supplier

```
public T orElseGet(Supplier<? extends T> supplier)
```

See optionals\OptionalContentRetrieving

Retrieving the contents of an Optional

- If a value is present, returns the value, otherwise throws an exception produced by the exception supplying function.

```
public <X extends Throwable> T orElseThrow(Supplier<? extends X>
                                           exceptionSupplier) throws X
```

- If a value is present, returns the value, otherwise throws NoSuchElementException

```
public T orElseThrow()   [JAVA10]
```

See optionals\OptionalContentRetrieving

Creating chain of Optionals

- The Optional<T> class provides the following methods that can be used to create chains of Optionals:

```
/*If a value is present, returns an Optional wrapping the value,
  otherwise returns an Optional produced by the supplier*/
public Optional<T> or(Supplier<? extends Optional<? extends T>>
                                                    supplier)
```

See optionals\OptionalChaining

Printing the content of an Optional

- The Optional class's **ifPresent** method accepts a consumer and is frequently used to print the object that is wrapped by an Optional.

```
public void ifPresent(Consumer<? super T> action) {
    if (value != null) {
        action.accept(value);
    }
}
```

- Since ifPresent returns void, it must be the last method called if used in an Optional chain.

See optionals\OptionalIfPresent

Filtering Optionals

- The Optional class's **filter** method accepts a predicate and returns an Optional, so it can be used anywhere within an Optional chain.
- The **filter** method returns the Optional if the predicate is *true* and returns an empty Optional if the predicate is *false*.

```
public Optional<T> filter(Predicate<? super T> predicate) {
    Objects.requireNonNull(predicate);
    if (isEmpty()) {
        return this;
    } else {
        return predicate.test(value) ? this : empty();
    }
}
```

See optionals\OptionalFiltering

Optional mapping

- The Optional class's **map** method accepts a function that converts a non-null Optional<T> to an Optional<U>.

```
public <U> Optional<U> map(Function<? super T, ? extends U>
                                                    mapper) {
    Objects.requireNonNull(mapper);
    if (isEmpty()) {
        return empty();
    } else {
        return Optional.ofNullable(mapper.apply(value));
    }
}
```

- The map method can be used to modify the underlying object, since it returns a new Optional.

See optionals\Resource & OptionalMapping

Optional flatmapping

- The Optional class's **flatmap** method accepts a function that converts a non-null Optional<T> to an Optional<U>.

```
public <U> Optional<U> flatMap(Function<? super T,
                                ? extends Optional<? extends U>> mapper) {
    Objects.requireNonNull(mapper);
    if (isEmpty()) {
        return empty();
    } else {
        @SuppressWarnings("unchecked")
        Optional<U> r = (Optional<U>) mapper.apply(value);
        return Objects.requireNonNull(r);
    }
}
```

- The Optional class's flatmap method can avoid the extra level of nesting.

See optionals\OptionalFlatMapping\Student & OptionalMapVsFlatmap

Primitive Optionals

- There are **OptionalInt**, **OptionalLong** and **OptionalDouble** classes.
- The difference between OptionalDouble and Optional<Double> is that first is for a primitive and second is for the Double wrapper class.
- Working with the primitive optional class looks similar to working with the Optional class itself.

```
OptionalDouble optional = OptionalDouble.of(5.5);
optional.ifPresent(System.out::println);                    // 5.5
System.out.println(optional.getAsDouble());                 // 5.5
System.out.println(optional.orElseGet(() -> Double.NaN));   // 5.5
```

DoubleSupplier