Java 8 new features

1. Lambda Expressions
2. Functional Interfaces
3. Method Reference
4. Streams
5. Comparable and Comparator
6. Optional Class
7. Date/Time API

Java Lambda Expressions

Lambda expressions in Java, introduced in Java SE 8. It represents the instances of functional interfaces (interfaces with a single abstract method). They provide a concise way to express instances of single-method interfaces using a block of code.

Key Functionalities of Lambda Expression

Lambda Expressions implement the only abstract function and therefore implement functional interfaces. Lambda expressions are added in Java 8 and provide the following functionalities.

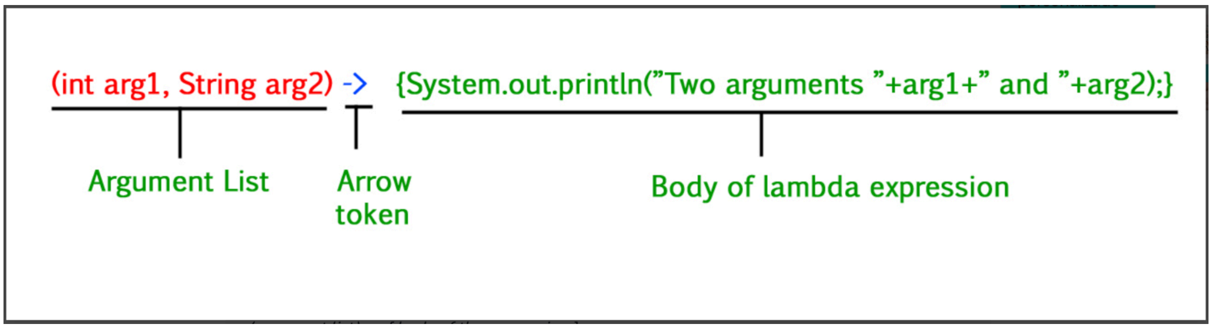
- **Functional Interfaces:** A functional interface is an interface that contains only one abstract method.
- **Code as Data:** Treat functionality as a method argument.
- **Class Independence:** Create functions without defining a class.
- **Pass and Execute:** Pass lambda expressions as objects and execute on demand.

Example: Implementing a Functional Interface with Lambda

The below Java program demonstrates how a lambda expression can be used to implement a user-defined functional interface.

```
interface FuncInterface {
    // An abstract function
    void abstractFun(int x);

    // A non-abstract (or default) function
    default void normalFun() {
       System.out.println("Hello");
    }
}

class Test {
    public static void main(String args[]) {
        // lambda expression to implement the above functional
interface.
        FuncInterface fobj = (int x)->System.out.println(2*x);

        // This calls the above lambda expression and prints 10.
        fobj.abstractFun(5);
    }
}

Output
10
```

Structure of Lambda Expression
Below diagram demonstrates the structure of Lambda Expression:


(int arg1, String arg2) -> {System.out.println("Two arguments "+arg1+" and "+arg2);}
Argument List
Arrow token
Body of lambda expression

Syntax of Lambda Expressions
Java Lambda Expression has the following syntax:

```
(argument list) -> { body of the expression }
```

Components:

- Argument List: Parameters for the lambda expression
- Arrow Token (->): Separates the parameter list and the body
- Body: Logic to be executed.

Types of Lambda Parameters
There are three Lambda Expression Parameters are mentioned below:

- Lambda with Zero Parameter
- Lambda with Single Parameter
- Lambda with Multiple Parameters

1. Lambda with Zero Parameter

Syntax:

```
() -> System.out.println("Zero parameter lambda");
```

Example: The below Java program demonstrates a Lambda expression with zero parameter.

```
@FunctionalInterface
interface ZeroParameter {
    void display();
}

public class Main {
    public static void main(String[] args){
        // Lambda expression with zero parameters
```

```
            ZeroParameter zeroParamLambda = () -> System.out.println(
                "This is a zero-parameter lambda expression!");

        zeroParamLambda.display();
    }
}

Output
This is a zero-parameter lambda expression!
```

2. Lambda with a Single Parameter

Syntax:

```
(p) -> System.out.println("One parameter: " + p);
```

It is not mandatory to use parentheses if the type of that variable can be inferred from the context.

Example: The below Java program demonstrates the use of lambda expression in two different scenarios with an ArrayList. We are using lambda expressions to iterate through and print all elements of an ArrayList. We are using lambda expression with a condition to selectively print an even number of elements from an ArrayList.

```
import java.util.ArrayList;

class Test {
    public static void main(String args[]) {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(1);
        al.add(2);
        al.add(3);
        al.add(4);

        System.out.println("Elements of the ArrayList: ");
        al.forEach(n -> System.out.println(n));

        System.out.println("Even elements of the ArrayList: ");
        al.forEach(n -> {
            if (n % 2 == 0)
                System.out.println(n);
        });
    }
}
```

```
Output
Elements of the ArrayList:
1
2
3
4
Even elements of the ArrayList:
2
4
```

Note: In the above example, we are using lambda expression with the foreach() method and it internally works with the consumer functional interface. The Consumer interface takes a single parameter and performs an action on it.

3. Lambda Expression with Multiple Parameters

Syntax:

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

Example: The below Java program demonstrates the use of lambda expression to implement functional interface to perform basic arithmetic operations.

```
@FunctionalInterface
interface Functional {
    int operation(int a, int b);
}

public class Test {

    public static void main(String[] args) {

        // Using lambda expressions to define the operations
        Functional add = (a, b) -> a + b;
        Functional multiply = (a, b) -> a * b;

        // Using the operations
        System.out.println(add.operation(6, 3));
        System.out.println(multiply.operation(4, 5));
    }
}

Output
9
20
```

Note: Lambda expressions are just like functions and they accept parameters just like functions.

Common Built-in Functional Interfaces

- Comparable<T>: int compareTo(T o);
- Comparator<T>: int compare(T o1, T o2);

These are commonly used in sorting and comparisons.

Note: Other commonly used interfaces include Predicate<T>, it is used to test conditions, Function<T, R>, it represents a function that takes the argument of type T and returns a result of type R and Supplier<T>, it represents a function that supplies results.

Based on the syntax rules just shown, which of the following is/are NOT valid lambda expressions?

1. () -> {};
2. () -> "Test";
3. () -> { return "Test";};
4. (Integer i) -> {return "Test" + i;}
5. (String s) -> {return "Test";};
6. () -> { return "Hello" }
7. x -> { return x + 1; }
8. (int x, y) -> x + y

Explanation of above lambda expressions:

1. **() -> {}:** (valid). No parameters, no return value (empty body). It is valid
2. **() -> "Test":** (valid). This lambda takes no parameters and returns a String without using a return keyword or braces. This is allowed for single-expression lambdas.
3. **() -> { return "Test"; }:** (valid). This lambda takes no parameters and returns a value using a code block. Since it uses braces {}, the return keyword is required. It is valid.
4. **(Integer i) -> {return "Test" + i;}:** (valid). This lambda takes one parameter i of type Integer and returns a concatenated string. It uses a code block with return, which is correct. It is valid.
5. **(String s) -> {return "Test";}:** (valid). This lambda takes one parameter s of type String, but does not use it. That's still perfectly valid.
6. **() -> { return "Hello" }:** (invalid). Missing semicolon after the return statement inside the block.
7. **x -> { return x + 1; }:** (invalid). This is missing the type of parameter if used in a context where type inference is not possible
8. **(int x, y) -> x + y:** (invalid). If you specify the type of one parameter, you must specify the type of all parameters.

Conclusions:

1. A lambda expression can have zero or more number of parameters(arguments).

   Ex:
   ```
   ()  sop(“hello”);
   (int a )  sop(a);
   (inta, int b)  return a+b;
   ```

2. Usually we can specify the type of parameter.If the compiler expects the type based on the context then we can remove type. i.e., a programme is not required.

   Ex:
   ```
   (inta, int b)  sop(a+b);
   (a,b)  sop(a+b);
   ```
3. If multiple parameters are present then these parameters should be separated with comma(,).
4. If there are zero parameters available then we have to use empty parameters [like ()].

   Ex:
   ```
   ()  sop(“hello”);
   ```
5. If only one parameter is available and if the compiler can expect the type then we can remove the type and parenthesis also.

   Ex:
   ```
   (int a)  sop(a);
   (a) sop(a);
   A  sop(a);
   ```
6. Similar to method body lambda expression body also can contain multiple statements.if more than one statement present then we have to enclose inside within curly braces.if one statement present then curly braces are optional.

7. Once we write a lambda expression we can call that expression just like a method, for this functional interfaces are required.

Functional Interfaces

A functional interface in Java is an interface that contains only one abstract method. Functional interfaces can have multiple default or static methods, but only one abstract method. From Java 8 onwards, lambda expressions and method references can be used to represent the instance of a functional interface.

Example: Using a Functional Interface with Lambda Expression

```
public class Main {
    public static void main(String[] args) {
        // Using lambda expression to implement Runnable
        new Thread(() ->
                   System.out.println("New thread created")).start();
    }
}
Output
New thread created
```

Explanation:
The above program demonstrates use of lambda expression with the Runnable functional interface.

- Runnable has one abstract method run(), so it qualifies as a functional interface.
- Lambda ()-> System.out.println("New thread created") defines the run() method.
- new Thread().start() starts a new thread that executes the lambda body

Note: A functional interface can also extend another functional interface.

@FunctionalInterface Annotation
@FunctionalInterface annotation is used to ensure that the functional interface cannot have more than one abstract method. In case more than one abstract method is present, the compiler flags an "Unexpected @FunctionalInterface annotation" message. However, it is not mandatory to use this annotation.

Note: @FunctionalInterface annotation is optional but it is a good practice to use. It helps catching the error in the early stage by making sure that the interface has only one abstract method.

Example: Defining a Functional Interface with @FunctionalInterface Annotation

```
@FunctionalInterface
interface Square {
    int calculate(int x);
}

class Main {
    public static void main(String args[]) {
        int a = 5;

        // lambda expression to define the calculate method
        Square s = (int x) -> x * x;

        int ans = s.calculate(a);
        System.out.println(ans);
    }
}

Output
25
```

Explanation:

- Square is a functional interface with a single method calculate(int x).
- A lambda expression (int x) -> x * x is used to implement the calculate method.
- Lambda takes x as input and returns x * x.

Types of Functional Interfaces in Java

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations as mentioned below:

1. Consumer
2. Predicate
3. Function
4. Supplier

1. Consumer:

The Consumer Interface is a part of the java.util.function package which has been introduced since Java 8, to implement functional programming in Java. It represents a function which takes in one argument and produces a result. However these kinds of functions don't return any value.
Hence this functional interface which takes in one generic namely:-

- T: denotes the type of the input argument to the operation

The lambda expression assigned to an object of Consumer type is used to define its accept() which eventually applies the given operation on its argument.

Consumers are useful when it is not needed to return any value as they are expected to operate via side-effects.

Functions in Consumer Interface
The Consumer interface consists of the following two functions:

1. accept()
This method accepts one value and performs the operation on the given argument

Syntax:
```
void accept(T t)
```

Parameters: This method takes in one parameter:
t- the input argument

Returns: This method does not return any value.

Below is the code to illustrate accept() method:
Program 1:
```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Consumer;

public class Main {
    public static void main(String args[]) {
        // Consumer to display a number
        Consumer<Integer> display = a -> System.out.println(a);

        // Implement display using accept()
        display.accept(10);

        // Consumer to multiply 2 to every integer of a list
        Consumer<List<Integer> > modify = list -> {
            for (int i = 0; i < list.size(); i++)
                list.set(i, 2 * list.get(i));
        };

        // Consumer to display a list of numbers
        Consumer<List<Integer> >
            dispList = list -> list.stream().forEach(a ->
System.out.print(a + " "));
```

```
        List<Integer> list = new ArrayList<Integer>();
        list.add(2);
        list.add(1);
        list.add(3);

        // Implement modify using accept()
        modify.accept(list);

        // Implement dispList using accept()
        dispList.accept(list);
    }
}

Output:
10
4 2 6
```

2. andThen()
It returns a composed Consumer wherein the parameterized Consumer will be executed after the first one. If evaluation of either function throws an error, it is relayed to the caller of the composed operation.

Note: The function being passed as the argument should be of type Consumer.

Syntax:

```
default Consumer <T> andThen(Consumer<? super T> after)
```

Parameters: This method accepts a parameter after which is the Consumer to be applied after the current one.

Return Value: This method returns a composed Consumer that first applies the current Consumer first and then the after operation.

Exception: This method throws NullPointerException if the after operation is null.

Program 1:

```java
// Java Program to demonstrate
// Consumer's andThen() method

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Consumer;

public class Main {
    public static void main(String args[]) {
        // Consumer to multiply 2 to every integer of a list
        Consumer<List<Integer> > modify = list -> {
            for (int i = 0; i < list.size(); i++)
                list.set(i, 2 * list.get(i));
        };

        // Consumer to display a list of integers
        Consumer<List<Integer> >
            dispList = list -> list.stream().forEach(a ->
System.out.print(a + " "));

        List<Integer> list = new ArrayList<Integer>();
        list.add(2);
        list.add(1);
        list.add(3);

        // using addThen()
        modify.andThen(dispList).accept(list);
        ;
    }
}

Output:
4 2 6
```

Program 2: To demonstrate when NullPointerException is returned.

```
// Java Program to demonstrate Consumer's andThen() method
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Consumer;

public class Main {
    public static void main(String args[]) {
        // Consumer to multiply 2 to every integer of a list
        Consumer<List<Integer> > modify = list -> {
            for (int i = 0; i < list.size(); i++)
                list.set(i, 2 * list.get(i));
        };

        // Consumer to display a list of integers
        Consumer<List<Integer>> dispList = list ->
list.stream().forEach(a -> System.out.print(a + " "));

        List<Integer> list = new ArrayList<Integer>();
        list.add(2);
        list.add(1);
        list.add(3);

        try {
            // using addThen()
            modify.andThen(null).accept(list);
        } catch (Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}

Output:
Exception: java.lang.NullPointerException
```

Program 3: To demonstrate how an Exception in the after function is returned and handled.

```
// Java Program to demonstrate Consumer's andThen() method
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.function.Consumer;

public class Main {
    public static void main(String args[]) {
        // Consumer to multiply 2 to every integer of a list
        Consumer<List<Integer> > modify = list -> {
            for (int i = 0; i <= list.size(); i++)
                list.set(i, 2 * list.get(i));
        };

        // Consumer to display a list of integers
        Consumer<List<Integer> >
            dispList = list -> list.stream().forEach(a ->
System.out.print(a + " "));
        System.out.println();

        List<Integer> list = new ArrayList<Integer>();
        list.add(2);
        list.add(1);
        list.add(3);

        // using addThen()
        try {
            dispList.andThen(modify).accept(list);
        } catch (Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}

Output:
2 1 3 Exception: java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
```

2. Predicate:
The Functional Interface PREDICATE is defined in the java.util.function package. It improves manageability of code, helps in unit-testing them separately, and contain some methods like:

```
isEqual(Object targetRef) : Returns a predicate that tests if two
arguments are equal according to Objects.equals(Object, Object).
static  Predicate isEqual(Object targetRef)
Returns a predicate that tests if two arguments are
equal according to Objects.equals(Object, Object).
T : the type of arguments to the predicate
Parameters:
targetRef : the object reference with which to
compare for equality, which may be null
Returns: a predicate that tests if two arguments
are equal according to Objects.equals(Object, Object)
```

```
and(Predicate other) : Returns a composed predicate that represents a
short-circuiting logical AND of this predicate and another.
default Predicate and(Predicate other)
Returns a composed predicate that represents a
short-circuiting logical AND of this predicate and another.
Parameters:
other: a predicate that will be logically-ANDed with this predicate
Returns : a composed predicate that represents the short-circuiting
logical AND of this predicate and the other predicate
Throws: NullPointerException - if other is null
negate() : Returns a predicate that represents the logical negation of
this predicate.
```

```
default Predicate negate()
Returns:a predicate that represents the logical
negation of this predicate
```

```
or(Predicate other) : Returns a composed predicate that represents a
short-circuiting logical OR of this predicate and another.
default Predicate or(Predicate other)
Parameters:
other : a predicate that will be logically-ORed with this predicate
Returns: a composed predicate that represents the short-circuiting
logical OR of this predicate and the other predicate
Throws : NullPointerException - if other is null
```

```
test(T t) : Evaluates this predicate on the given argument.boolean
test(T t)
test(T t)
Parameters:
t - the input argument
Returns:
true if the input argument matches the predicate, otherwise false
```

Example 1: Simple Predicate

```
// Java program to illustrate Simple Predicate
import java.util.function.Predicate;
public class PredicateInterfaceExample1 {
    public static void main(String[] args) {
        // Creating predicate
        Predicate<Integer> lesserthan = i -> (i < 18);

        // Calling Predicate method
        System.out.println(lesserthan.test(10));
    }
}
Output:
True
```

Example 2: Predicate Chaining

```
// Java program to illustrate Predicate Chaining
import java.util.function.Predicate;

public class PredicateInterfaceExample2 {
    public static void main(String[] args) {
        Predicate<Integer> greaterThanTen = (i) -> i > 10;

        // Creating predicate
        Predicate<Integer> lowerThanTwenty = (i) -> i < 20;
        boolean result = greaterThanTen.and(lowerThanTwenty).test(15);
        System.out.println(result);
        // Calling Predicate method
        boolean result2 =
greaterThanTen.and(lowerThanTwenty).negate().test(15);
        System.out.println(result2);
    }
}
Output:
True
False
```

Example 3: Predicate in to Function

```
// Java program to illustrate passing Predicate into function
import java.util.function.Predicate;

class PredicateInterfaceExample3 {
    static void pred(int number, Predicate<Integer> predicate) {
        if (predicate.test(number)) {
            System.out.println("Number " + number);
        }
    }
    public static void main(String[] args) {
        pred(10, (i) -> i > 7);
    }
}
Output:
Number 10
```

Example 4: Predicate OR

```
// Java program to illustrate OR Predicate
import java.util.function.Predicate;

class PredicateInterfaceExample4 {
    public static Predicate<String> hasLengthOf10 = new
Predicate<String>() {
        @Override
        public boolean test(String t) {
            return t.length() > 10;
        }
    };

    public static void predicate_or() {
        Predicate<String> containsLetterA = p -> p.contains("A");
        String containsA = "And";
        boolean outcome =
hasLengthOf10.or(containsLetterA).test(containsA);
        System.out.println(outcome);
    }
    public static void main(String[] args) {
        predicate_or();
    }
}
Output:
True
```

Example 5: Predicate AND

```
// Java program to illustrate AND Predicate
import java.util.function.Predicate;
import java.util.Objects;

class PredicateInterfaceExample5 {
    public static Predicate<String> hasLengthOf10 = new
Predicate<String>() {
        @Override
        public boolean test(String t) {
            return t.length() > 10;
        }
    };

    public static void predicate_and() {
        Predicate<String> nonNullPredicate = Objects::nonNull;

        String nullString = null;

        boolean outcome =
nonNullPredicate.and(hasLengthOf10).test(nullString);
        System.out.println(outcome);

        String lengthGTThan10 = "Welcome to the machine";
        boolean outcome2 = nonNullPredicate.and(hasLengthOf10).
        test(lengthGTThan10);
        System.out.println(outcome2);
    }
    public static void main(String[] args) {
        predicate_and();
    }
}

Output:
False
True
```

Example 6: Predicate negate()

```
// Java program to illustrate negate Predicate
import java.util.function.Predicate;

class PredicateInterfaceExample6 {
    public static Predicate<String> hasLengthOf10 = new
Predicate<String>() {
        @Override
        public boolean test(String t) {
            return t.length() > 10;
        }
    };

    public static void predicate_negate() {

        String lengthGTThan10 = "Thunderstruck is a 2012 children's "
                                + "film starring Kevin Durant";

        boolean outcome = hasLengthOf10.negate().test(lengthGTThan10);
        System.out.println(outcome);
    }
    public static void main(String[] args) {
        predicate_negate();
    }
}

Output:
False
```

Example 7: Predicate in Collection

```
import java.util.function.Predicate;
import java.util.*;

class User {
    String name, role;
    User(String a, String b) {
        name = a;
        role = b;
    }
    String getRole() { return role; }
    String getName() { return name; }
    public String toString() {
       return "User Name : " + name + ", Role :" + role;
    }

    public static void main(String args[]) {
        List<User> users = new ArrayList<User>();
        users.add(new User("John", "admin"));
        users.add(new User("Peter", "member"));
        List admins = process(users, (User u) ->
u.getRole().equals("admin"));
        System.out.println(admins);
    }

    public static List<User> process(List<User> users,
                            Predicate<User> predicate) {
        List<User> result = new ArrayList<User>();
        for (User user: users)
            if (predicate.test(user))
                result.add(user);
        return result;
    }
}

Output:
[User Name : John, Role :admin]
```

3. Function:
The Function Interface is a part of the java.util.function package that has been introduced since Java 8, to implement functional programming in Java. It represents a function that takes in one argument and produces a result. Hence, this functional interface takes in 2 generics, namely as follows:

- T: denotes the type of the input argument
- R: denotes the return type of the function

Note: The lambda expression assigned to an object of Function type is used to define its apply() which eventually applies the given function on the argument.

Methods in Function Interface
The Function interface consists of the following 4 methods, as listed, which are later discussed as follows:

1. apply()
2. andThen()
3. compose()
4. identity()

Method 1: apply()

Syntax:

```
R apply(T t)
```

Parameters: This method takes in only one parameter t, which is the function argument

Return Type: This method returns the function result, which is of type R.

Example:

```
import java.util.function.Function;

public class Main {
    public static void main(String args[]) {
        // Function which takes in a number and returns half of it
        Function<Integer, Double> half = a -> a / 2.0;

        // Applying the function to get the result
        System.out.println(half.apply(10));
    }
}

Output
5.0
```

Method 2: andThen()
It returns a composed function wherein the parameterized function will be executed after the first one. If evaluation of either function throws an error, it is relayed to the caller of the composed function.

Syntax:

```
default <V> Function<T, V> andThen(Function<? super R, ? extends V>
after)
```

where V is the type of output of the after function, and of the composed function

Parameters: This method accepts a parameter after which is the function to be applied after the current one.

Return Value: This method returns a composed function that applies the current function first and then the after function
Exception: This method throws NullPointerException if the after function is null.

Example 1:

```
import java.util.function.Function;

public class Main {
    public static void main(String args[]) {
        // Function which takes in a number and returns half of it
        Function<Integer, Double> half = a -> a / 2.0;

        // Now triple the output of half function
        half = half.andThen(a -> 3 * a);

        // Applying the function to get the result and printing on
console
        System.out.println(half.apply(10));
    }
}

Output
15.0
```

Example 2: To demonstrate when a NullPointerException is returned.

```
import java.util.function.Function;

public class Main {
    public static void main(String args[]) {
        // Function which takes in a number and returns half of it
        Function<Integer, Double> half = a -> a / 2.0;

        // Try block to check for exceptions
        try {
            // Trying to pass null as parameter
            half = half.andThen(null);
        } catch (Exception e) {
            // Print statement
            System.out.println("Exception thrown " + "while passing
null: " + e);
        }
    }
}

Output
Exception thrown while passing null: java.lang.NullPointerException
```

Method 3: compose()
It returns a composed function wherein the parameterized function will be executed first and then the first one. If evaluation of either function throws an error, it is relayed to the caller of the composed function.

Syntax:

```
default <V> Function<V, R> compose(Function<? super V, ? extends T>
before)
```

Where V is the type of input of the before function, and of the composed function

Parameters: This method accepts a parameter before which is the function to be applied first and then the current one

Return Value: This method returns a composed function that applies the current function after the parameterized function

Exception: This method throws NullPointerException if the before function is null.

Example 1:

```
import java.util.function.Function;

public class Main {
    public static void main(String args[]) {
        // Function which takes in a number and returns half of it
        Function<Integer, Double> half = a -> a / 2.0;

        // Triple the value given to half function
        half = half.compose(a -> 3 * a);

        // Applying the function to get the result
        System.out.println(half.apply(5));
    }
}

Output
7.5
```

Example 2: When NullPointerException is returned.

```
import java.util.function.Function;

public class Main {
    public static void main(String args[]) {
        // Function which takes in a number and returns half of it
        Function<Integer, Double> half = a -> a / 2.0;

        // Try block to check for exceptions
        try {
            // Trying to pass null as parameter
            half = half.compose(null);
        } catch (Exception e) {
            // Print statement
            System.out.println("Exception thrown "
                               + "while passing null: "
                               + e);
        }
    }
}

Output
Exception thrown while passing null: java.lang.NullPointerException
```

Method 4: identity()
This method returns a function that returns its only argument.

Syntax:

```
static <T> Function<T, T> identity()
```

where T denotes the type of the argument and the value to be returned.

Returns: This method returns a function that returns its own argument.

Example:

```
import java.util.function.Function;
public class Main {
    public static void main(String args[]) {
        Function<Integer, Integer> i = Function.identity();
        System.out.println(i.apply(10));
    }
}
Output
10
```

Applying Function on Objects and Handling Multiple Conditions

1. Using Function on Objects

Example:

```
// Demonstrates the working of functional interface
import java.util.function.Function;

// create a Person class with properties
// name and age
class Person {
    String name;
    int age;

    // creates a constructor
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    // override the toString() method to
    // display the person details
    @Override
    public String toString() {
        return "Name: " + name + ", Age: " + age;
    }
}

public class Main {
    public static void main(String args[]) {

        // create a Function that takes object and
        // returns a greeting with the person's name
        Function<Person, String> greet = person -> "Hello, " +
person.name;

        Person p = new Person("Test", 25);
        System.out.println(greet.apply(p));
    }
}

Output
Hello, Test
```

Explanation: In the above example we are using a Function interface to transform the Person object into a greeting string which contains the person's name.

2. Handling Multiple Conditions Using Function

Example:

```
// Handling Multiple Conditions
// with the help of Function
import java.util.function.Function;

public class Test {
    public static void main(String args[]) {
        Function<Integer, Integer> addFive = a -> a + 5;
        Function<Integer, Integer> multiplyByTwo = a -> a * 2;

        // Applying functions sequentially: Add five, then multiply by
two
        Function<Integer, Integer> result =
addFive.andThen(multiplyByTwo);

        System.out.println(result.apply(3));
    }
}

Output
16
```

Explanation: In the above example, two functions are declared with the help of Function interface. The first function takes an integer and adds 5 to it and the second function takes an integer and then multiplies it with 2. These two functions are combined with the andThen() method. This method makes sure that the addFive() function is applied first and then multipleByTwo() function is applied.

4. Supplier:
The Supplier Interface is a part of the java.util.function package which has been introduced since Java 8, to implement functional programming in Java. It represents a function which does not take in any argument but produces a value of type T. Hence this functional interface takes in only one generic namely:-

- T: denotes the type of the result

The lambda expression assigned to an object of Supplier type is used to define its get() which eventually produces a value. Suppliers are useful when we don't need to supply any value and obtain a result at the same time. The Supplier interface consists of only one function:
1. get()
This method does not take in any argument but produces a value of type T.
Syntax:

```
T get()
```

Returns: This method returns a value of type T. Below is the code to illustrate get() method:

Program:

```
import java.util.function.Supplier;

public class Main {
    public static void main(String args[]) {
        // This function returns a random value.
        Supplier<Double> randomValue = () -> Math.random();

        // Print the random value using get()
        System.out.println(randomValue.get());
    }
}
Output:
0.5685808855697841
```

Method Reference

In Java, a method is a collection of statements that perform some specific task and return the result to the caller. A method reference is the shorthand syntax for a lambda expression that contains just one method call. In general, one does not have to pass arguments to method references.

Why Use Method References?
Method references are used for the following reasons, which are listed below:

- Method references enhance readability, which makes the code easier to understand.
- It supports a functional programming style that works well with streams and collections.
- Reusability increases because we can directly use the existing methods.

Note: Functional Interfaces in Java and Lambda Function are prerequisites required in order to grasp a grip over Method References in Java.

Example:

```
import java.util.Arrays;

public class Main {
    // Method
    public static void print(String s) {
        System.out.println(s);
    }

    public static void main(String[] args) {
        String[] names = {"Test1", "Test2", "Test3"};

        // Using method reference to print each name
        Arrays.stream(names).forEach(Main::print);
    }
}

Output
Test1
Test2
Test3
```

Explanation: In the above example, we are using method reference to print items. The print method is a static method which is used to print the names. In the main method we created an array of names and printed each one by calling the print method directly.

Key Benefits of Method References
The key benefits of method references are listed below:

- Improved Readability: Method references simplify the code by removing boilerplate syntax.
- Reusability: Existing methods can be directly reused, enhancing modularity.
- Functional Programming Support: They work seamlessly with functional interfaces and lambdas.

Function as a Variable
In Java 8 we can use the method as if they were objects or primitive values, and we can treat them as a variable.

```
// This square function is a variable getSquare.
Function<Integer, Integer> getSquare = i -> i * i ;

// Pass function as an argument to another function easily
SomeFunction(a, b, getSquare) ;
```

Sometimes, a lambda expression only calls an existing method. In those cases, it looks clear to refer to the existing method by name. The method references can do this, they are compact, easy-to-read as compared to lambda expressions.

Generic Syntax for Method References

Aspect	Syntax
Refer to a method in an object	Object :: methodName
Print all elements in a list	list.forEach(s -> System.out.println(s));
Shorthand to print all elements in a list	list.forEach(System.out::println);

Types of Method References
There are four type method references that are as follows:

- Static Method Reference
- Instance Method Reference of a particular object
- Instance Method Reference of an arbitrary object of a particular type
- Constructor Reference

To look into all these types we will consider a common example of sorting with a comparator which is as follows:

1. Reference to a Static Method

A static method lets us use a method from a class without writing extra code. It is a shorter way to write a lambda that just calls that static method.

Syntax:

```java
// Lambda expression
(args) -> Class.staticMethod(args);
// Method reference
Class::staticMethod;
```

Example:

```java
// Reference to a static method
import java.io.*;
import java.util.*;

class Person {
    private String name;
    private Integer age;

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

    // Getter-setters
    public Integer getAge() { return age; }
    public String getName() { return name; }
}


public class Main {
    // Static method to compare with name
    public static int compareByName(Person a, Person b) {
        return a.getName().compareTo(b.getName());
    }

    // Static method to compare with age
    public static int compareByAge(Person a, Person b) {
        return a.getAge().compareTo(b.getAge());
    }
    public static void main(String[] args) {
        List<Person> personList = new ArrayList<>();

        personList.add(new Person("Vicky", 24));
        personList.add(new Person("Poonam", 25));
        personList.add(new Person("Sachin", 19));

        // Using static method reference to sort array by name
        Collections.sort(personList, Main::compareByName);

        // Display message only
        System.out.println("Sort by Name :");

        // Using streams over above object of Person type
        personList.stream()
          .map(x -> x.getName()).forEach(System.out::println);
        System.out.println();

        // Now using static method reference to sort array by age
        Collections.sort(personList, Main::compareByAge);

        // Display message only
        System.out.println("Sort by Age :");

        // Using streams over above object of Person type
        personList.stream()
          .map(x -> x.getName()).forEach(System.out::println);
    }
}

Output
Sort by Name :
Poonam
Sachin
Vicky

Sort by Age :
Sachin
Vicky
Poonam
```

Explanation: This example shows how to use static method references to sort items. We have a person class with attributes like name and age and there are two methods to compare people by name and by age. In the main method we created a list of people and sorted them by name and then sorted them by age and then printing the name again.

2. Reference to an Instance Method of a Particular Object
This type of method means using a method from a certain object which we already have. We do not need to write another function to call that particular method, we can just simply refer to it directly.

Syntax:

```
// Lambda expression
(args) -> obj.instanceMethod(args);

// Method reference
obj::instanceMethod;
```

Example:

```
// Reference to an Instance Method of a Particular Object
import java.io.*;
import java.util.*;

class Person {
    private String name;
    private Integer age;

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

    public Integer getAge() { return age; }
    public String getName() { return name; }
}

// Helper class Comparator class
class ComparisonProvider {
    // To compare with name
    public int compareByName(Person a, Person b) {
        return a.getName().compareTo(b.getName());
    }

    // To compare with age
    public int compareByAge(Person a, Person b) {
        return a.getAge().compareTo(b.getAge());
    }
}

// Main class
```

```java
public class Main {
    public static void main(String[] args) {
        List<Person> personList = new ArrayList<>();

        personList.add(new Person("Vicky", 24));
        personList.add(new Person("Poonam", 25));
        personList.add(new Person("Sachin", 19));

        // A comparator class with multiple comparator methods
        ComparisonProvider comparator = new ComparisonProvider();

        // Now using instance method reference to sort array by name
        Collections.sort(personList, comparator::compareByName);

        System.out.println("Sort by Name :");

        // Using streams
        personList.stream()
          .map(x -> x.getName()).forEach(System.out::println);
        System.out.println();

        // Using instance method reference to sort array by age
        Collections.sort(personList, comparator::compareByAge);
        System.out.println("Sort by Age :");

        personList.stream()
          .map(x -> x.getName()).forEach(System.out::println);
    }
}
```

```
Output
Sort by Name :
Poonam
Sachin
Vicky

Sort by Age :
Sachin
Vicky
Poonam
```

Explanation: This example shows how to use an instance method reference to sort a list of people. We have created a Person class with name and age and we also created a ComparisonProvider class, it has methods to compare people by name or age. In the main method we created a list of people and we are using the ComparisonProvider instance to sort and print the names first by name, then by age.

3. Reference to an Instance Method of an Arbitrary Object of a Particular Type
It means calling a method on any object that belongs to a certain group or class, not just one specific object. It helps us write less code when we want to do the same thing for many objects.

Syntax:

```
// Lambda expression
(obj, args) -> obj.instanceMethod(args);

// Method reference
ObjectType::instanceMethod;
```

Example:

```
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> personList = new ArrayList<>();

        personList.add("Vicky");
        personList.add("Poonam");
        personList.add("Sachin");

        // Method reference to String type
        Collections.sort(personList, String::compareToIgnoreCase);

        // Printing the elements(names) on console
        personList.forEach(System.out::println);
    }
}

Output
Poonam
Sachin
Vicky
```

Explanation: This example shows how to use a method reference to sort a list of names. We created a list of names and then sorting them ignoring uppercase or lowercase with the help of compareToIgnoreCase method of the String class and then we are printing the sorted names

4. Constructor Method Reference
It lets us quickly create a new object without writing extra code. It is a shortcut to call the class a new method.

Syntax:

```
// Lambda expression
(args) -> new ClassName(args);

// Method reference
ClassName::new;
```

Example:

```
import java.io.*;
import java.nio.charset.Charset;
import java.util.*;
import java.util.function.*;

class Person {
    private String name;
    private Integer age;

    // Constructor
    public Person() {
        Random ran = new Random();

        this.name = ran
                    .ints(97, 122 + 1)
                    .limit(7)
                    .collect(StringBuilder::new,
                             StringBuilder::appendCodePoint,
                             StringBuilder::append)
                    .toString();
    }

    public Integer getAge(){
        return age;
    }
    public String getName() {
        return name;
    }
}
```

```
public class Main {
    public static <T> List<T> getObjectList(int length, Supplier<T>
objectSupply) {
        List<T> list = new ArrayList<>();

        for (int i = 0; i < length; i++)
            list.add(objectSupply.get());
        return list;
    }

    public static void main(String[] args) {
        List<Person> personList = getObjectList(5, Person::new);

        // Print names of personList
        personList.stream()
            .map(x -> x.getName())
            .forEach(System.out::println);
    }
}

Output
ilvxzcv
vdixqbs
lmcfzpj
dxnyqej
zeqejcn
```
Explanation: This example shows how to use a constructor method reference to create objects. We have created a Person class and it gives each person a random name. The getObjectList method creates a list of objects by using a supplier, which means it uses the Person constructor to make new Person objects. In the main method we created a list of people and then we are printing their random names.

Common Use Cases

There are some common cases where we use Method References in Java as mentioned below:

- Iterating over collections: Simplifying operations like printing or processing elements.
- Stream API operations: Enhancing readability in filtering, mapping, and reducing operations.
- Custom utilities: Using predefined methods for frequently used tasks like sorting and comparisons.

Streams

Stream was introduced in Java 8, the Stream API is used to process collections of objects. A stream in Java is a sequence of objects that supports various methods that can be pipelined to produce the desired result.

Use of Stream in Java
The uses of Stream in Java are mentioned below:

- Stream API is a way to express and process collections of objects.
- Enable us to perform operations like filtering, mapping, reducing, and sorting.

How to Create a Java Stream?
Java Stream Creation is one of the most basic steps before considering the functionalities of the Java Stream. Below is the syntax given for declaring a Java Stream.

Syntax

```
Stream<T> stream;
```

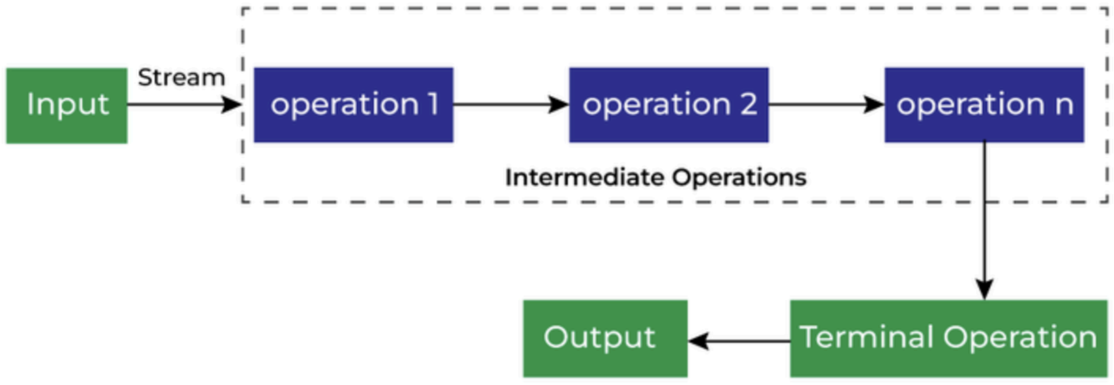
Here, T is either a class, object, or data type depending upon the declaration.

Java Stream Features
The features of Java streams are mentioned below:

- A stream is not a data structure; instead, it takes input from the Collections, Arrays, or I/O channels.
- Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence, various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams
There are two types of Operations in Streams:

1. Intermediate Operations
2. Terminal Operations



Intermediate Operations are the types of operations in which multiple methods are chained in a row.

Characteristics of Intermediate Operations

- Methods are chained together.
- Intermediate operations transform a stream into another stream.
- It enables the concept of filtering where one method filters data and passes it to another method after processing.

Benefit of Java Stream

There are some benefits because of which we use Stream in Java as mentioned below:

- No Storage
- Pipeline of Functions
- Laziness
- Can be infinite
- Can be parallelized
- Can be created from collections, arrays, Files Lines, Methods in Stream, IntStream etc.

Important Intermediate Operations

There are a few Intermediate Operations mentioned below:

1. map(): The map method is used to return a stream consisting of the results of applying the given function to the elements of this stream.

Syntax:

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

2. filter(): The filter method is used to select elements as per the Predicate passed as an argument.

Syntax:

```
Stream<T> filter(Predicate<? super T> predicate)
```

3. sorted(): The sorted method is used to sort the stream.

Syntax:

```
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
```

4. flatMap(): The flatMap operation in Java Streams is used to flatten a stream of collections into a single stream of elements.

Syntax:

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

5. distinct(): Removes duplicate elements. It returns a stream consisting of the distinct elements (according to Object.equals(Object)).

Syntax:

```
Stream<T> distinct()
```

6. peek(): Performs an action on each element without modifying the stream. It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

Syntax:

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

Java program that demonstrates the use of all the intermediate operations:

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

public class StreamIntermediateOperationsExample {
    public static void main(String[] args) {
        List<List<String>> listOfLists = Arrays.asList(
            Arrays.asList("Reflection", "Collection", "Stream"),
            Arrays.asList("Structure", "State", "Flow"),
            Arrays.asList("Sorting", "Mapping", "Reduction", "Stream")
        );
        // Create a set to hold intermediate results
        Set<String> intermediateResults = new HashSet<>();

        // Stream pipeline demonstrating various intermediate operations
        List<String> result = listOfLists.stream().flatMap(List::stream)
            .filter(s -> s.startsWith("S"))
            .map(String::toUpperCase)
            .distinct()
            .sorted()
            .peek(s -> intermediateResults.add(s))
            .collect(Collectors.toList());

        System.out.println("Intermediate Results:");
        intermediateResults.forEach(System.out::println);

        System.out.println("Final Result:");
        result.forEach(System.out::println);
    }
}

Output
Intermediate Results:
STRUCTURE
STREAM
STATE
SORTING
Final Result:
SORTING
STATE
STREAM
STRUCTURE
```

Explanation of the above Program:
List of Lists Creation:

- The listOfLists is created as a list containing other lists of strings.

Stream Operations:

- **flatMap(List::stream):** Flattens the nested lists into a single stream of strings.
- **filter(s -> s.startsWith("S")):** Filters the strings to only include those that start with "S".
- **map(String::toUpperCase):** Converts each string in the stream to uppercase.
- **distinct():** Removes any duplicate strings.
- **sorted():** Sorts the resulting strings alphabetically.
- **peek(...):** Adds each processed element to the intermediateResults set for intermediate inspection.
- **collect(Collectors.toList()):** Collects the final processed strings into a list called result.

The program prints the intermediate results stored in the intermediateResults set. Finally, it prints the result list, which contains the fully processed strings after all stream operations. This example showcases how Java Streams can be used to process and manipulate collections of data in a functional and declarative manner, applying transformations and filters in a sequence of operations.

Terminal Operations
Terminal Operations are the type of Operations that return the result. These Operations are not processed further just return a final result value.

Important Terminal Operations
There are a few Terminal Operations mentioned below:

1. collect(): The collect method is used to return the result of the intermediate operations performed on the stream.

Syntax:

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

2. forEach(): The forEach method is used to iterate through every element of the stream.

Syntax:

```
void forEach(Consumer<? super T> action)
```

3. reduce(): The reduce method is used to reduce the elements of a stream to a single value. The reduce method takes a BinaryOperator as a parameter.

Syntax:

```
T reduce(T identity, BinaryOperator<T> accumulator)
Optional<T> reduce(BinaryOperator<T> accumulator)
```

4. count(): Returns the count of elements in the stream.

Syntax:

```
long count()
```

5. findFirst(): Returns the first element of the stream, if present.

Syntax:

```
Optional<T> findFirst()
```

6. allMatch(): Checks if all elements of the stream match a given predicate.

Syntax:

```
boolean allMatch(Predicate<? super T> predicate)
```

7. anyMatch(): Checks if any element of the stream matches a given predicate.

Syntax:

```
boolean anyMatch(Predicate<? super T> predicate)
```

Note: Intermediate Operations are running based on the concept of Lazy Evaluation, which ensures that every method returns a fixed value(Terminal operation) before moving to the next method.

Java Program Using all Terminal Operations:

```java
import java.util.*;
import java.util.stream.Collectors;

public class StreamTerminalOperationsExample {
    public static void main(String[] args) {
        // Sample data
        List<String> names = Arrays.asList(
            "Reflection", "Collection", "Stream",
            "Structure", "Sorting", "State"
        );

        // forEach: Print each name
        System.out.println("forEach:");
        names.stream().forEach(System.out::println);

        // collect: Collect names starting with 'S' into a list
        List<String> sNames = names.stream()
                                   .filter(name -> name.startsWith("S"))
                                   .collect(Collectors.toList());
        System.out.println("\ncollect (names starting with 'S'):");
        sNames.forEach(System.out::println);

        // reduce: Concatenate all names into a single string
        String concatenatedNames = names.stream().reduce(
            "",
            (partialString, element) -> partialString + " " + element
        );
        System.out.println("\nreduce (concatenated names):");
        System.out.println(concatenatedNames.trim());

        // count: Count the number of names
        long count = names.stream().count();
        System.out.println("\ncount:");
        System.out.println(count);

        // findFirst: Find the first name
        Optional<String> firstName = names.stream().findFirst();
        System.out.println("\nfindFirst:");
        firstName.ifPresent(System.out::println);

        // allMatch: Check if all names start with 'S'
        boolean allStartWithS = names.stream().allMatch(
            name -> name.startsWith("S")
        );
        System.out.println("\nallMatch (all start with 'S'):");
        System.out.println(allStartWithS);

        // anyMatch: Check if any name starts with 'S'
        boolean anyStartWithS = names.stream().anyMatch(
            name -> name.startsWith("S")
        );
        System.out.println("\nanyMatch (any start with 'S'):");
        System.out.println(anyStartWithS);
    }
}

Output:
forEach:
Reflection
Collection
Stream
Structure
Sorting
State
collect (names starting with 'S'):
Stream
Structure
Sorting
State
reduce (concatenated names):
Reflection Collection Stream Structure Sorting State
count:
6
findFirst:
Reflection
allMatch (all start with 'S'):
false
anyMatch (any start with 'S'):
true
```

Explanation of the above Program:

List Creation:

- The names list is created with sample strings.

Stream Operations:

- **forEach:** Prints each name in the list.
- **collect:** Filters names starting with 'S' and collects them into a new list.
- **reduce:** Concatenates all names into a single string.
- **count:** Counts the total number of names.
- **findFirst:** Finds and prints the first name in the list.

- **allMatch:** Checks if all names start with 'S'.
- **anyMatch:** Checks if any name starts with 'S'.

The program prints each name, names starting with 'S', concatenated names, the count of names, the first name, whether all names start with 'S', and whether any name starts with 'S'.

Real-World Use Cases of Java Streams

Streams are widely used in modern Java applications for:

- Data Processing
- For processing JSON/XML responses
- For database Operations
- Concurrent Processing

Important Points:

- A stream consists of a source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
- Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

Comparable and Comparator

In Java, both Comparable and Comparator interfaces are used for sorting objects. The main difference between Comparable and Comparator is:

- Comparable: It is used to define the natural ordering of the objects within the class.
- Comparator: It is used to define custom sorting logic externally.

Difference Between Comparable and Comparator

The table below demonstrates the difference between comparable and comparator in Java.

Features	Comparable	Comparator
Definition	It defines natural ordering within the class.	It defines external or custom sorting logic.
Method	compareTo()	compare()
Implementation	It is implemented in the class.	It is implemented in a separate class.
Sorting Criteria	Natural order sorting	Custom order sorting
Usage	It is used for a single sorting order.	It is used for multiple sorting orders.

Example of Comparable

In this example, we use the Comparable interface to sort Movies by their release year using compareTo() method.

```java
import java.util.ArrayList;
import java.util.Collections;

// Movie class implements Comparable
// interface to define default sorting
class Movie implements Comparable<Movie> {
    private String name;
    private double rating;
    private int year;

    // Constructor
    public Movie(String name, double rating, int year) {
        this.name = name;
        this.rating = rating;
        this.year = year;
    }

    // Implementation of the compareTo method
    // for default sorting by year
    public int compareTo(Movie m) {
        // Sort movies in ascending order of year
        return this.year - m.year;
    }

    // Getter methods
    public String getName() {
        return name;
    }

    public double getRating() {
        return rating;
    }

    public int getYear() {
        return year;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Create a list of movies
        ArrayList<Movie> l = new ArrayList<>();
        l.add(new Movie("Star Wars", 8.7, 1977));
        l.add(new Movie("Empire Strikes Back", 8.8, 1980));
        l.add(new Movie("Return of the Jedi", 8.4, 1983));

        // Sort movies using Comparable's compareTo method by year
        Collections.sort(l);

        // Display the sorted list of movies
        System.out.println("Movies after sorting by year:");
        for (Movie m : l) {
            System.out.println(m.getName() + " " + m.getRating() + " " +
m.getYear());
        }
    }
}

Output
Movies after sorting by year:
Star Wars 8.7 1977
Empire Strikes Back 8.8 1980
Return of the Jedi 8.4 1983
```

Explanation: In the above example, the compareTo() method sorts the Movie objects by their release year, where they return negative if the current movie year is earlier, return positive if the current movie year is later and return zero if the year is the same. The Collections.sort() method uses the compareTo() method to compare and sort the movies in ascending order.

Now, suppose we want to sort movies by their rating and names as well. When we make a collection element comparable (by having it implement Comparable), we get only one chance to implement the compareTo() method. The solution is using Comparator.

Example of Comparator

In this example, we use Comparator interface to define custom sorting logic to sort movies first by rating and then by name.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

class Movie {
    private String name;
    private double rating;
    private int year;

    // Constructor to initialize movie details
    public Movie(String name, double rating, int year) {
        this.name = name;
        this.rating = rating;
        this.year = year;
    }

    public String getN() {
      return name;
    }
    public double getR() {
      return rating;
    }
    public int getY() {
      return year;
    }
}

// Comparator to sort movies by rating
class Rating implements Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {
        // Sort by rating in descending order
        return Double.compare(m2.getR(), m1.getR());
    }
}

// Comparator to sort movies by name
class NameCompare implements Comparator<Movie> {
    public int compare(Movie m1, Movie m2) {

        // Sort by name in alphabetical order
        return m1.getN().compareTo(m2.getN());
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        ArrayList<Movie> m = new ArrayList<>();
        m.add(new Movie("Force Awakens", 8.3, 2015));
        m.add(new Movie("Star Wars", 8.7, 1977));
        m.add(new Movie("Empire Strikes Back", 8.8, 1980));
        Collections.sort(m, new Rating());
        System.out.println("Movies sorted by rating:");
        for (Movie m1 : m) {
            System.out.println(m1.getR() + " " + m1.getN() + " " +
m1.getY());
        }
        Collections.sort(m, new NameCompare());
        System.out.println("\nMovies sorted by name:");
        for (Movie m1 : m) {
            System.out.println(m1.getN() + " " + m1.getR() + " " +
m1.getY());
        }
    }
}

Output
Movies sorted by rating:
8.8 Empire Strikes Back 1980
8.7 Star Wars 1977
8.3 Force Awakens 2015

Movies sorted by name:
Empire Strikes Back 8.8 1980
Force Awakens 8.3 2015
Star Wars 8.7 1977
```

Explanation: In the above example, the Comparator interface is used to sort the movies first by rating and then by name. The Rating and NameCompare classes implement custom sorting logic. The Collections.sort() method uses these comparators to sort the list by multiple criteria.

Important Point:
Comparator is a functional interface. So, now the question is why comparator is a functional interface?

Answer: Java 8 introduced a functional interface. These interfaces have only one abstract method. But the comparator has two methods that are compare(T o1, T o2) and equals(Object obj). Here, only compare() is an abstract method. The equals() method is inherited from the Object class and is not considered abstract in the interface.

Optional Class

Optional is a container class introduced in Java 8 to represent optional (nullable) values. It is used to avoid NullPointerException by providing methods to check whether a value is present or not and handle it safely. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts that were hidden are now visible to the developer.

Example: Java program without Optional Class

```
public class OptionalDemo {
    public static void main(String[] args) {
        String[] words = new String[10];
        String word = words[5].toLowerCase();
        System.out.print(word);
    }
}
Output:
Exception in thread "main" java.lang.NullPointerException
```

To avoid abnormal termination, we use the Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

Example: Program using Optional Class

```
import java.util.Optional;

public class OptionalDemo {
    public static void main(String[] args) {
        String[] words = new String[10];

          Optional<String> checkNull = Optional.ofNullable(words[5]);

          if (checkNull.isPresent()) {
            String word = words[5].toLowerCase();
            System.out.print(word);
          }
        else
            System.out.println("word is null");
    }
}

Output
word is null
```

Common Methods in Optional

The Optional class offers methods such as isPresent(), orElse(), and ifPresent() to handle potentially null values more safely and functionally. The table below lists commonly used Optional methods and their descriptions.

Method	Description
empty()	Returns an empty Optional instance
of(T value)	Returns an Optional with the specified present non-null value.
ofNullable(T value)	Returns an Optional describing the specified value if non-null, else empty
equals(Object obj)	Checks if another object is "equal to" this Optional.
filter(Predicate<? super T> predicate)	If a value is present and matches the predicate, returns an Optional with the value; else returns empty.
flatMap(Function<? super T, Optional<U>> mapper)	If a value is present, apply a mapping function that returns an Optional; otherwise returns empty.
get()	Returns the value if present, else throws NoSuchElementException.
hashCode()	Returns the hash code of the value if present, otherwise returns 0.
ifPresent(Consumer<? super T> consumer)	Returns true if a value is present, else false.
map(Function<? super T, ? extends U> mapper)	If a value is present, apply the mapping function and return an Optional describing the result (if non-null).
orElse(T other)	Returns the value if present, otherwise returns the provided default value.
orElseGet(Supplier<? extends T> other)	Returns the value if present, otherwise invokes the supplier and returns its result.
orElseThrow(Supplier<? extends X> exceptionSupplier)	Returns the value if present, otherwise throws an exception provided by the supplier

toString()	Returns a non-empty string representation of this Optional for debugging.

Example 1: Java program to illustrate some optional class methods.

```
import java.util.Optional;

class Main {
    public static void main(String[] args){
        // creating a string array
        String[] str = new String[5];

        // Setting value for 2nd index
        str[2] = "Test.";

        // It returns an empty instance of Optional class
        Optional<String> empty = Optional.empty();
        System.out.println(empty);

        // It returns a non-empty Optional
        Optional<String> value = Optional.of(str[2]);
        System.out.println(value);
    }
}

Output
Optional.empty
Optional[Test.]
```

Example 2: Java program to illustrate some optional class methods

```
import java.util.Optional;

class Main {
    public static void main(String[] args) {
        // creating a string array
        String[] str = new String[5];

        // Setting value for 2nd index
        str[2] = "Test.";

        // It returns a non-empty Optional
        Optional<String> value = Optional.of(str[2]);



         // It returns value of an Optional.If value is not present, it
throws an NoSuchElementException
        System.out.println(value.get());

        // It returns hashCode of the value
        System.out.println(value.hashCode());

        // It returns true if value is present, otherwise false
        System.out.println(value.isPresent());
    }
}

Output
Test.
1967487235
true
```

Date/Time API

New date-time API is introduced in Java 8 to overcome the following drawbacks of old date-time API :

1. **Not thread safe :** Unlike old java.util.Date which is not thread safe the new date-time API is immutable and doesn't have setter methods.
2. **Less operations :** In the old API there are only few date operations but the new API provides us with many date operations.

Java 8 under the package java.time introduced a new date-time API, most important classes among them are :

1. **Local :** Simplified date-time API with no complexity of timezone handling.
2. **Zoned :** Specialized date-time API to deal with various timezones.

LocalDate/LocalTime and LocalDateTime API : Use it when time zones are NOT required.

```java
// Java code for LocalDate LocalTime Function
import java.time.*;
import java.time.format.DateTimeFormatter;

public class Date {

public static void LocalDateTimeApi() {
    // the current date
    LocalDate date = LocalDate.now();
    System.out.println("the current date is "+ date);


    // the current time
    LocalTime time = LocalTime.now();
    System.out.println("the current time is "+ time);


    // will give us the current time and date
    LocalDateTime current = LocalDateTime.now();
    System.out.println("current date and time : "+ current);


    // to print in a particular format
    DateTimeFormatter format =
      DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    String formatedDateTime = current.format(format);

    System.out.println("in formatted manner "+ formatedDateTime);
    // printing months days and seconds
    Month month = current.getMonth();
    int day = current.getDayOfMonth();
    int seconds = current.getSecond();
    System.out.println("Month : "+month+" day : " +
                       day +" seconds : "+ seconds);

    // printing some specified date
    LocalDate date2 = LocalDate.of(1950,1,26);
    System.out.println("the republic day :"+date2);

    // printing date with current time.
    LocalDateTime specificDate =
        current.withDayOfMonth(24).withYear(2016);

    System.out.println("specific date with "+
                       "current time : "+specificDate);
}

    public static void main(String[] args) {
        LocalDateTimeApi();
    }
}

Output
the current date is 2021-09-23
the current time is 20:52:39.954238
current date and time : 2021-09-23T20:52:39.956909
in formatted manner 23-09-2021 20:52:39
Month : SEPTEMBER day : 23 seconds : 39
the republic day :1950-01-26
specific date with current time : 2016-09-24T20:52:39.956909
```

Zoned date-time API : Use it when time zones are to be considered

```java
// Java code for Zoned date-time API
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;

public class Zone {
public static void ZonedTimeAndDate() {
    LocalDateTime date = LocalDateTime.now();
    DateTimeFormatter format1 =
      DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    String formattedCurrentDate = date.format(format1);

    System.out.println("formatted current Date and"+
                      " Time : "+formattedCurrentDate);

    // to get the current zone
    ZonedDateTime currentZone = ZonedDateTime.now();
    System.out.println("the current zone is "+
                        currentZone.getZone());

    // getting time zone of specific place
    // we use withZoneSameInstant(): it is
    // used to return a copy of this date-time
    // with a different time-zone,
    // retaining the instant.
    ZoneId tokyo = ZoneId.of("Asia/Tokyo");

    ZonedDateTime tokyoZone =
            currentZone.withZoneSameInstant(tokyo);

    System.out.println("tokyo time zone is " +
                        tokyoZone);

    DateTimeFormatter format =
        DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    String formatedDateTime = tokyoZone.format(format);

    System.out.println("formatted tokyo time zone "+
                        formatedDateTime);

}
```

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

Output:
formatted current Date and Time : 09-04-2018 06:21:13
the current zone is Etc/UTC
tokyo time zone is 2018-04-09T15:21:13.220+09:00[Asia/Tokyo]
formatted tokyo time zone 09-04-2018 15:21:13
```

Period and Duration classes :

- Period : It deals with date based amount of time.
- Duration : It deals with a time based amount of time.

```
// Java code for period and duration
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.Month;
import java.time.Duration;
import java.time.Period;

public class Main {
    public static void checkingPeriod() {
        LocalDate date1 = LocalDate.now();

        LocalDate date2 =
            LocalDate.of(2014, Month.DECEMBER, 12);

        Period gap = Period.between(date2, date1);
        System.out.println("gap between dates "+
                           "is a period of "+gap);
    }

    public static void checkingDuration() {

        LocalTime time1 = LocalTime.now();
        System.out.println("the current time is " +
                            time1);

        Duration fiveHours = Duration.ofHours(5);

        // adding five hours to the current
        // time and storing it in time2

        LocalTime time2 = time1.plus(fiveHours);

        System.out.println("after adding five hours " +
                           "of duration " + time2);

        Duration gap = Duration.between(time2, time1);
        System.out.println("duration gap between time1" +
                           " & time2 is " + gap);
}

    public static void main(String[] args) {
        checkingPeriod();
        checkingDuration();
    }
}

Output
gap between dates is a period of P6Y6M25D
the current time is 18:34:24.813548
after adding five hours of duration 23:34:24.813548
duration gap between time1 & time2 is PT-5H
```

ChronoUnits Enum : java.time.temporal.ChronoUnit enum is added in Java 8 to replace integer values used in old API to represent day, month etc.

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;

public class Main{
    public static void checkingChronoEnum() {
        LocalDate date = LocalDate.now();
        System.out.println("current date is :" +
                            date);

        // adding 2 years to the current date
        LocalDate year =
             date.plus(2, ChronoUnit.YEARS);

        System.out.println("next to next year is " +
                            year);

        // adding 1 month to the current date
        LocalDate nextMonth =
                date.plus(1, ChronoUnit.MONTHS);

        System.out.println("the next month is " + nextMonth);
        // adding 1 week to the current date
        LocalDate nextWeek =
                  date.plus(1, ChronoUnit.WEEKS);

        System.out.println("next week is " + nextWeek);

        // adding 2 decades to the current date
        LocalDate Decade =
                  date.plus(2, ChronoUnit.DECADES);

        System.out.println("20 years after today " +
                            Decade);
    }

    public static void main(String[] args) {
        checkingChronoEnum();
    }
}

Output:
current date is :2018-04-09
next to next year is 2020-04-09
the next month is 2018-05-09
next week is 2018-04-16
20 years after today 2038-04-09
```

TemporalAdjuster: It is used to perform various date related operations.

```
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
import java.time.DayOfWeek;

public class main {
    // Function to check date and time
    // according to our requirement
    public static void checkingAdjusters()
    {

        LocalDate date = LocalDate.now();
        System.out.println("the current date is "+
                            date);

        // to get the first day of next month
        LocalDate dayOfNextMonth =
              date.with(TemporalAdjusters.
                        firstDayOfNextMonth());
        System.out.println("firstDayOfNextMonth : " +
                           dayOfNextMonth );

        // get the next saturday
        LocalDate nextSaturday =
                date.with(TemporalAdjusters.
                          next(DayOfWeek.SATURDAY));

        System.out.println("next saturday from now is "+
                           nextSaturday);

        // first day of current month
        LocalDate firstDay =
                  date.with(TemporalAdjusters.
                  firstDayOfMonth());

        System.out.println("firstDayOfMonth : " +
                           firstDay);

        // last day of current month
        LocalDate lastDay =
                  date.with(TemporalAdjusters.
                            lastDayOfMonth());

        System.out.println("lastDayOfMonth : " +
                           lastDay);
}

    public static void main(String[] args) {
        checkingAdjusters();
    }
}

Output
the current date is 2021-07-09
firstDayOfNextMonth : 2021-08-01
next saturday from now is 2021-07-10
firstDayOfMonth : 2021-07-01
lastDayOfMonth : 2021-07-31
```