
Java
8

FUNCTIONAL INTERFACES

Functional interface

- An interface with only one method.
- **@FunctionalInterface**
  - Used to enforce the intent of the interface
  - Existing interfaces are annotated with @FunctionalInterface
  - Adding another method to the interface definition, will not be functional and compilation process will fail.
- ***Used with lambda expressions.***

Built-in Functional Interfaces

- Earlier Versions of Java has several functional interfaces
  - ***Prominent because of Lambda expressions***

- Existing functional interfaces
  - Comparator
  - Callable
  - Runnable
  - ActionListener

- Collections API has been rewritten and new Stream API uses a lot of functional interfaces.

- java.util.function
  - *Consumer, Supplier, Function and Predicate.*

LAMBDAS

Functional Programming

- Java always remained **Object first language.**
- Functions are not important for Java.
  - cannot live on their own in Java world.
- *JavaScript is one of the best example of an FP language.*
- There is no way of passing a method as argument or returning a method body
- invList.forEach((inv)->{System.***out.println(inv);});***
- **Lambda expression adds that missing link of functional programming** to Java.

Benefits of Lambda Expression

- Fewer Lines of Code
  - It reduced amount of code.
- Sequential and Parallel Execution
  - Support by passing behavior in methods
- Higher Efficiency
  - Utilizing Multicore CPU's using Streams
- Streams API
  - Handling Collection of data much efficiently

Where to use Lambda expressions

- Lambda expressions can be used where there is a target type.
- The following are the target type in Java
  - Variable declarations and assignments
  - Return statements
  - Method or constructor arguments

Lambda Expressions

- ***Lambda Expressions are:***
- ***Anonymous***
  - It doesn't have an explicit name like a method
- ***Function***
  - *Not associated with a particular class like a method*
  - Has a list of parameters, a body, a return type, and list of exceptions that can be thrown.
- ***Passed around***
  - *Can be passed as argument to a method or stored in a* variable.
- ***Concise***
  - No need to write a lot of boilerplate like anonymous classes.

Lambda Expressions

- A lambda expression is composed of three parts.
  - **Argument List** **(int x, int y)**
  - **Arrow Token** **->**
  - **Body** **x + y**

Lambda – Argument List

- ***Can have zero, one or more parameters.***
  - **(int x, int y)**
- **Parameters can be explicitly declared or it can be inferred**
  - (int a) is same as just (a)
- **Parameters are separated by commas.**
  - (a, b) or (int a, int b) or (String a, int b, float c)
- **Empty parentheses to represent an empty set**
  - () -> 42
- **It is not mandatory to use parentheses.**
  - a -> return a*a

Lambda Expression - Body

- The body of the lambda expressions can contain zero, one or more statements.
- Lambdas may return a value.
- The type of the return value will be inferred by compiler.
  - Body is evaluated like a method body
  - return statement returns control to the caller of the anonymous method.
- The **return** statement is not required if the lambda body is just a one-liner.
- **If the return is an expression curly braces are required**

Lambda with Built-in Functional Interface

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

 Runnable task = () -> {

for(int i=0;i<10;i++) {
       System.out.println("Hello World");
   }
};

Thread t = new Thread(task);

     t.start();
   }
```

Custom Functional Interface

```
interface Converter<T,R>{
        public R calculate(T frm);
}
public static void  doConvert(Converter<Double,Double> conv,Double frm){

 System.out.println(conv.calculate(frm));
}

public static void main(String[] args) {

Converter<Double,Double> currencyConverter = (val)->{return val * 45.0;};

doConvert(currencyConverter, 100.00);

   Converter<Double,Double> farenToCel = (faren)->{return (faren-32) * 5/9;};

      doConvert(farenToCel, 84.00);
}
```

BUILT-IN FUNCTIONAL INTERFACES

Need for Built In Functional Interfaces

- Lambda expressions must correspond to one functional interface.
- Java 8 also has new functional interfaces covering the most common scenarios usages.
  - **Each developer need not define** functional interface of types which are already in this package.
  - They are **Defined with Generic types** and are re-usable for specific use cases.

Built-in Functional Interfaces

- Located inside the **java.util.function** package.
  - **Predicate<T>**
  - **Consumer<T>**
  - **Function<T, R>**
  - **Supplier<T>**
- Where T and R represent generic types
  - **T** represents a parameter type
  - **R** the return type.

Predicate<T>

- A predicate is a function that receives a value and evaluates it.
  - It has a method test(T obj) to evaluate and return a Boolean value.
  - It also has some default Methods

```
@FunctionalInterface
    public interface Predicate<T> {
        boolean test(T t);
        // Other default and static methods
    }
```

Predicate<T>

- **Using Anonymous Class**

```
Predicate<String> startsWithA = new Predicate<String>() {

    public boolean test(String t) {
            return t.startsWith("A");
        } };

boolean result = startsWithA.test("Anand");
```

- **Using Lambda Expression**

```
Predicate<String> testWithLambda = (name)->name.startsWith("A");
        System.out.println(testWithLambda.test("Anand"));
```

Predicate<T>

```
public void usingPredicate(){

 List<String> countries =
          Arrays.asList("India","Nepal","srilanka","SouthAfrica","Indonesia");

 Predicate<String> pref = (country)->country.startsWith("I");

 countries.forEach((eachCountry)->{
          if(pref.test(eachCountry))   {
         System.out.println(eachCountry);
         }
 });
}
```

Predicate<T> Default Methods

- **negate()**
  - returns a predicate that represents the logical negation of the predicate.

- Predicate<String> predicate = (s) -> s.length() > 0;

  System.***out.println(predicate.test("Ramesh"));***

  System.***out.println(predicate.negate().test("Ramesh"));***

Predicate<T> Default Methods

- **default Predicate or(Predicate other)**
  - Returns a composed predicate that represents logical OR of two predicates
  - When evaluating composed predicate
    - If the first predicate is true, then the other predicate is not evaluated.
- **default Predicate and(Predicate other)**
  - Returns a composed predicate that represents a logical AND of two predicates.
  - When evaluating the composed predicate
    - if the first predicate is false, then the other predicate is not evaluated.
- **default Predicate isEqual(Object targetRef)**
  - Returns a predicate that tests if two arguments are equal
  - Equality is determined by Objects.equals()

Predicate<T> Methods

```
Predicate<String> maxLength = (s) -> s.length() < 8;

    System.out.println(maxLength.test("Ramesh"));

    System.out.println(maxLength.negate().test("Ramesh"));

    Predicate<String> minLength = (s) -> s.length() > 3;


    System.out.println("is name is more than 3 and less than
8 chars := "+ maxLength.and(minLength).test("Ramesh"));
```

Predicate<T> Methods

- Can combine predicates

- Predicate<String> startsWithA = t -> t.startsWith("A");

  Predicate<String> endsWithA = t -> t.endsWith("A");

  boolean result = startsWithA.and(endsWithA).test("Hi");

- static <T> Predicate<T> isEqual(Object targetRef)
  - Returns a Predicate that tests if two arguments are equal according to Objects.equals(Object, Object).

Supplier<T>

- Represents a supplier of results.
- It does not take any arguments.
- Used for lazy generation of values.

- **T get():**

  - Does not accept any argument
  - Returns newly generated values, T, in the stream.
  - No requirement that new or distinct results be returned each time the supplier is invoked.

Supplier<T>

```
public void usingSupplier(){

Supplier<Invoice> invSupplier = ()->{

   return new Invoice(200,"Rakesh",7888);

};

Invoice inv = invSupplier.get();

System.out.println(inv);

}
```

Consumer<T>

- Represents an operation that accepts a single input and returns no result
- It has one Method void accept(T t)
  - Performs operation on the given argument (T t)
- Lambda passed to the *List.forEach* method implements *Consumer*

```
public static void printNames(String name){
System.out.println(name);
}

public void usingConsumer(){

Consumer<String> consumer = Example::printNames;
      consumer.accept("Ramesh");
      consumer.accept("Suresh");
      consumer.accept("Maghesh");
}
```

Consumer<T>

- Specialized versions of the *Consumer* that receive primitive values
  - *DoubleConsumer*,
  - *IntConsumer*
  - *LongConsumer*

- ***BiConsumer* interface takes** two arguments

```
Map<String, Integer> ages = new HashMap<>();
ages.put(“Ramesh", 25);
ages.put(“Suresh", 24);

ages.forEach((name, age) -> System.out.println(name + " is " + age
    + " years old"));
```

Consumer<T>

- **default Consumer<T> andThen(Consumer<? super T> after)**
  - returns a composed Consumer that performs, in sequence, the operation of the consumer followed by the operation of the parameter.

- Used to combine Consumers and make the code more readable

- Consumer<String> first = t -> System.out.println("First:" + t);

- Consumer<String> second = t -> System.out.println("Second:" + t);

- first.andThen(second).accept("Hi");

- The output is:
- First: Hi
- Second: Hi

Function<T,R>

- Represents a function that accepts one argument and produces a result

- Object of a particular type is the input, an operation is performed on it and object of another type is returned as output,

  - Can be used without the need to define a new functional interface every time.

- R apply(T t)
  - Applies this function to the given argument (T t)
  - Returns the function result

Function<T,R>

```
Function<Integer, String> function = (t) -> {
        if (t % 2 == 0) {
            return t+ " is even number";
}
else {
 return t+ " is odd number";
}
};
System.out.println(function.apply(5));
System.out.println(function.apply(8));
```

Method References

- Used to store a reference on the function
  - But don't want to *invoke* it right away.
- Lambda expressions are for ***passing parameter to an instance method***
- Points to existing methods by their names.
  - Described using **::** **(double colon) symbol.**
- ***When Used With Lambdas expressions, creates a compact and concise code.***
  - ***Can't be used for any method.***
- **Can only be used to replace a single-method lambda expression.**

Method References

- List names = new ArrayList();

  names.add("Mahesh");

  names.add("Suresh");

- **names.forEach((e)->System.*out.println(e));***

  - *Passes parameter* as an argument to the println method

- *names.forEach(System.out::println);*

  - Can replace the pass-through lambda with a method reference.
  - ***A method reference names the method to which the parameter is passed***

referenceToInstance::methodName.

- Common Parts are removed to Change Lambda to a Method reference
  - The parameter
  - The argument
  - Dot is replaced with a colon on the method call.

```
String[] nameList =
                { "Ramesh", "anand", "Suresh", "Anand", "Magesh" };

Arrays.sort(nameList, String::compareToIgnoreCase);

for(String name:nameList) {
      System.out.println(name);
}
```

Method Reference

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


FirstExample ex = new FirstExample();


Function<Double,Double> reference = ex::invoke;

double result = reference.apply(200.00);

System.out.println(result);


}
```

DEFAULT AND STATIC METHODS IN INTERFACES

Need for default Methods

- When a interface has one or multiple implementations
- Adding one or more methods to the interface
  - All the implementations will be forced to implement them too.
  - If not the design will break down.

- Default interface methods **allows to add new methods to an interface that are automatically available in the implementations**.
  - Thus, there's no need to modify the implementing classes.

- **Backward compatibility is neatly preserved** without having to refactor the implementers.

Default Method

- D**efault methods are implicitly public**
- They are **declared with the *default* keyword**
- They **provide an implementation**.

- Default methods in interfaces helps to **incrementally provide additional functionality to a given type without breaking down the implementing classes.**

- Can be used to **provide additional functionality around an existing abstract method**:

Default Interface Methods

```
public interface Vehicle {

    String getBrand();

    String slowDown();

    default String turnAlarmOn() {
        return "Turning the vehicle alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the vehicle alarm off.";
    }
}
```

Default Interface Methods

```
public class Car implements Vehicle {

    private String brand;

    // constructors/getters

    @Override
    public String getBrand() {
        return brand;
    }


    @Override
    public String slowDown() {
        return "The car is slowing down.";
    }
}
```

Default Interface Methods

```
public static void main(String[] args) {
    Vehicle car = new Car("BMW");
    System.out.println(car.getBrand());
    System.out.println(car.slowDown());
    System.out.println(car.turnAlarmOn());
    System.out.println(car.turnAlarmOff());
}
```

Accessing Default Interface Methods

- Default method can be accessed from each instance including anonymous objects.
- Default methods **cannot** be accessed from within lambda expressions.

Multiple Interface Inheritance Rules

- Classes can implement multiple interfaces,
- **When a class implements several interfaces that define the same *default* methods**.

```
public interface Alarm {
    default String turnAlarmOn() {
        return "Turning the alarm on.";
    }

    default String turnAlarmOff() {
        return "Turning the alarm off.";
    }
}

public class Car implements Vehicle, Alarm {
    // ...
}
```

Multiple Interface Inheritance Rules

- There will be a conflict caused by multiple interface inheritance
- **Solved by explicitly providing an implementation for the methods:**

```
@Override
public String turnAlarmOn() {
    // custom implementation
}

@Override
public String turnAlarmOff() {
    // custom implementation
}
```

  - Can also **have class use the *default* methods of one of the interfaces**.

Multiple Interface Inheritance Rules

The class can use the *default* methods defined within the *Alarm* interface:

```
@Override
public String turnAlarmOn() {
    return Vehicle.super.turnAlarmOn();
}
```

```
@Override
public String turnAlarmOn() {
    return Alarm.super.turnAlarmOn();
}
```

Multiple Interface Inheritance Rules

Possible to make the *Car* class use both sets of default methods:

```
@Override
public String turnAlarmOn() {
    return Vehicle.super.turnAlarmOn() + " " + Alarm.super.turnAlarmOn();
}

@Override
public String turnAlarmOff() {
    return Vehicle.super.turnAlarmOff() + " " + Alarm.super.turnAlarmOff();

}
```

Static Interface Methods

- *static* methods don't belong to a particular object,
- They are not part of the API of the classes implementing the interface
  - **Example : utility methods for null check, collection sorting etc**

- They have to be **called by using the interface name preceding the method name**.

```
public interface Vehicle {

    // regular / default interface methods

    static int getHorsePower(int rpm, int torque) {
        return (rpm * torque) / 5252;
    }
}
```

Static Interface Methods

- Provide a mechanism that allows by putting together related methods in one single place without having to create an object.
  - **Can also be done with abstract classes.**
- Difference is that **abstract classes can have constructors, state, and behavior**.
- Can group related utility methods, without having to create artificial utility classes that are simply placeholders for static methods.

Interface -Default and Static Methods

- Default methods
  - Allows adding new methods to existing interfaces without breaking the binary compatibility with the code written for older versions of those interfaces.
- Difference between default methods and abstract methods
  - Abstract methods are required to be implemented.
  - Default methods are not.

OPTIONAL CLASS

Java.util.Optional Class

- It is a public final class used to deal with NullPointerException
- A container object used to contain not-null objects.
- Optional object is used to represent null with absent value.
- Has methods to handle values as ‘available’ or ‘not available’ instead of checking null values
- The class does not define any constructors.

Java.util.Optional Class

- **Optional.empty()**
  - Used to create an empty Optional

- Optional<String> emptyString = Optional.empty();

- **get()**
  - To obtain a value present in an Optional instance.
  - If doesn't contain a value, the method would throw NoSuchElementException.
- **Optional.ofNullable()**
  - To create an optional that can accept null.
  - Integer x = null;
  - Optional<Integer> optional = Optional.ofNullable(x);

Java.util.Optional Class

- **orElse(defaultValue)**
  - returns the value if present
  - otherwise returns the default value provided as parameter.

- **isPresent()**
  - determine if a value is present
  - returns boolean value true or false.

- **ifPresent()**
  - performs given action if the given Optional object is non-empty.
  - Otherwise it returns false.

Example

```
public static Object getStringObject(int key)
{
 switch (key) {
case 1:

return new String("Vannila String");
case 2:
return new StringBuffer("Buffer String");
case 3:
return new StringBuilder("Builder String");
default:
return null;
}

}
```

Example

```
Optional<Object> obj = Optional.ofNullable(getStringObject(2));

Object strType=   obj.orElse("Invalid Choice");

System.out.println("String :="+strType.toString());

obj.ifPresent((o)->{System.out.println( o.toString().toUpperCase() );});


if(obj.isPresent())  {
System.out.println("Value is present");

}  else  {
 System.out.println("Value is absent");

}
```

Java.util.Optional Class

- **orElseGet(getFunc)**
  - returns the value if present
  - otherwise returns the value obtained from getFunc.

- **orElseThrow(excFunc)**
  - returns the value if present,
  - otherwise throws the exception generated by excFunc.

Example

```
String[] names = new String[10];

Optional<String> checkNull = Optional.ofNullable(names[4]);

if(checkNull.isPresent())  {

System.out.println(checkNull.get());

}   else  {

      System.out.println("No Such Element");
}

checkNull.ifPresent(System.out::println);
```

Example

```
import java.util.Optional;

public class BankAccount {

private long accountNumber;
private String customerName;
private Double overDraft;

public BankAccount(long accountNumber, String customerName, Double
    overDraft) {
super();
this.accountNumber = accountNumber;
this.customerName = customerName;
this.overDraft = overDraft;

  }
}
```

Example

```
BankAccount regAccount = new BankAccount(200,"Ramesh",null);

  Optional<Double> regAcHasOD = Optional.ofNullable(regAccount.getOverDraft());

    regAccount.setOverDraft(regAcHasOD.orElse(4500.00));

System.out.println(regAccount);


  BankAccount salAccount = new BankAccount(201,"Rajesh",2000.00);

  Optional<Double> salAcHasOD = Optional.ofNullable(salAccount.getOverDraft());

  salAccount.setOverDraft(salAcHasOD.orElse(4500.00));

System.out.println(salAccount);
```