FAQ

(Frequently Asked Questions)

Table of Contents

Can one define a default method for a method from `java.lang.Object` class?

`Object` class is the base class for all the Java classes. All classes inherit from the base class. The base class has higher precedence over the default methods. Default methods cannot override a method from `java.lang.Object` class. Even if you define a default method, it will not be used while execution. Hence, it is not recommended to define a default method for an `Object` class.

How do you represent time without a date by using Date-Time API?

`LocalTime` class located within the `java.time` package signifies exact time of day without a date.

Why are immutable classes declared as final?

The objects of an immutable class cannot be changed. Final classes cannot be extended. By declaring a class as final, it is ensured that the subclasses do not override methods in the class.

Why are String values used as keys in HashMap?

String objects are immutable. Hence, they are used as HashMap keys in HashMap. Immutable objects are objects that do not change their state once they are created.

What is the double colon operator used for in Java 8?

In Java 8, method references are described using double colon (::) symbol. Method references helps to refer methods by their name. It is used to replace lambda expression in functional interface implementations.

When do you use `forEach()` method in Java 8?

`forEach()` method helps in traversing the elements of a list or collection. In Java 8, a collection or list interface contains `forEach()` method by default. You can use method reference or lambda expression inside `forEach()` method. Following example shows using lambda expression in the `forEach()` method:

```
List<String>  animals  =  new  ArrayList<>(Arrays.asList("cat",  "dog",
"lion", "cheetah"));
animals.forEach (a -> System.out.println(a) );
```

The code will output all the elements in the `animals` list.

You can modify the code to use method reference as following:

```
 List<String>  animals  =  new  ArrayList<>(Arrays.asList("cat",  "dog",
"lion", "cheetah"));
animals.forEach (System.out.println);
```

How is lambda expression beneficial to developers?

Using lambda expressions, a developer can write a method exactly at its place of execution. For example, to compare the length of two strings, the lambda expression can be stated as:

```
(String one, String two)
     → Integer.compare(one.length(), two.length())
```

Thus, it saves developers from writing lengthy code.

How will one implement parallelism in non thread-safe collections?

One can implement parallelism in non thread-safe collections using aggregate operations and parallel streams.

When do you use the `Consumer` interface?

`Consumer` interface is a part of `java.util.function` package. This interface accepts single input parameter and does not return any value as output. It is used in operations when an object is taken as input and some operation is done on the input without producing any result.

The following code shows a simple usage of `Consumer` interface with lambda expression. The input string is converted into lowercase.

```
import java.util.function.Consumer;

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

    Consumer<String> c = (x) → System.out.println(x.toLowerCase());;
    c.accept("ALICE IN WONDERLAND");
    }
}
```

The output of the code will be : alice in wonderland

Can you use the same annotation name for two different methods?

Java 8 allows applying the same annotation to a declaration or a type. This feature is called repeating annotation. For example, to schedule automatic backup of your files, you can apply `@BackupSchedule` annotation to run `backupComp()` method on the first day of every month and on every Sunday as:

```
@BackupSchedule(dayOfMonth=”first”);
@BackupSchedule(dayOfWeek=”Sun”);
public void backupComp(){….};
```

What is the use of `Optional` interface in Java 8?

In Java 8, `java.util.Optional<T>` class helps to avoid null pointer exceptions errors during runtime. The compiler throws a `NullPointerException` when any variable or input parameter is not validated. `Optional` is a single-value container object that either contains a value or is empty. If a value is present, `isPresent()` method returns true and `get()` method returns the value. Following code shows the usage of `Optional` interface:

```
List<Optional<Student>>StuList=new ArrayList<Optional<Student>>();
stuList.add(Optional.empty());
stuList.add(Optional.of(new Student("Myname")));
stuList.add(Optional.of(new Student("Roll25")));
System.out.println(stuList.get(0).getName());
```

The compiler does not throw a `NullPointerException` error even if the list is empty.

How is @Functional interface useful in coding?

`@FunctionalInterface` annotation helps in detecting errors during compilation when the interface you are annotating is not a valid functional interface. An interface may have only one abstract method. Java 8 has many functional interfaces which can be used in lambda expressions. These interfaces help to refer to lambda expressions in the code. For example, in the following interface `Myprog`:

```
@FunctionalInterface
public interface Myprog{
  int addsample(int a, int b);
}
```

if you add another abstract method within the interface as follows, the compiler will show an error during compilation:

```
@FunctionalInterface
public interface Myprog{
  int addsample(int a, int b);
  int subsample(int a, int b);
}
```

The compiler shows an error as follows:

```
Unexpected @FunctionalInterface annotation
@FunctionalInterface ^ Myprog is not a functional interface
multiple non-overriding abstract methods found in interface Myprog
```

--- End of FAQ ---