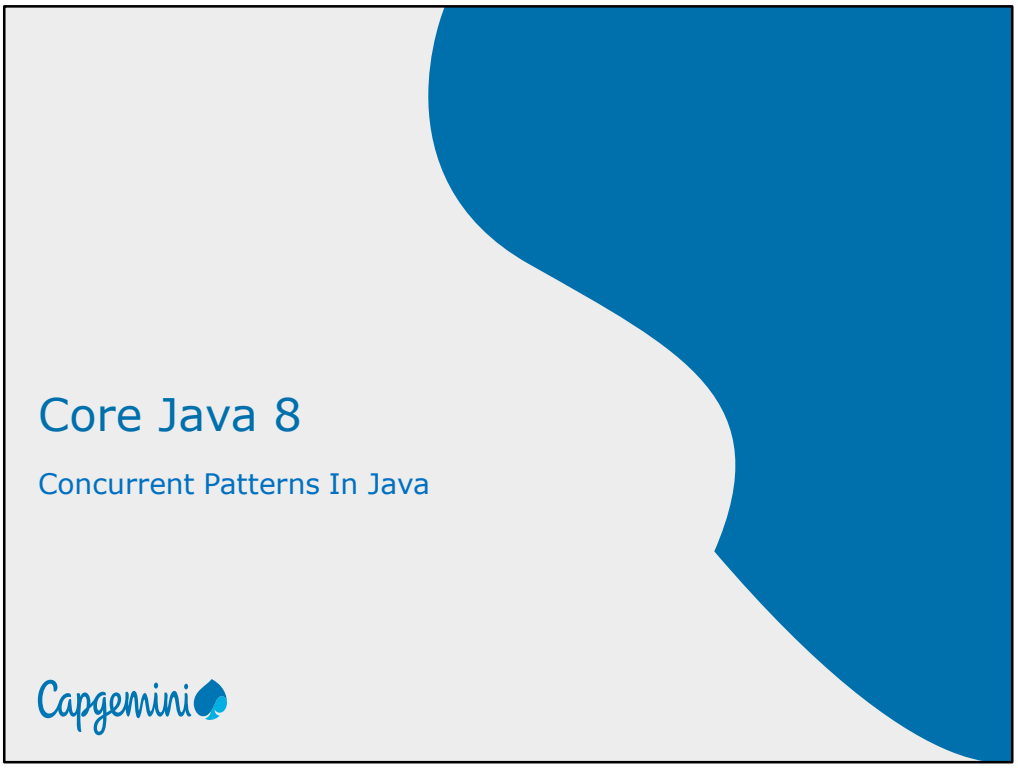
Core Java 8
Concurrent Patterns In Java
Capgemini

Lesson Objectives

After completing this lesson, participants will be able to

- Introducing Executors, What is Wrong with Runnable
- Defining Executor Pattern
- Executor Pattern Demo
- Comparing Runnable and Executor Pattern
- Understanding the Waiting Queue of the Executor Service
- Wrapping Up Executor Service Pattern
- From Runnable to Callable :What Is Wrong with Runnable
- Introducing Callable interface to Model Task
- Future Objects to Transmit Objects Between Threads

This lesson covers the concurrent Pattern in Java . It explains how to create Executor program and implement Concurrent Patterns .

Lesson outline:

Introducing Executors, What Is Wrong with the Runnable Pattern?
Defining the Executor Pattern: A New Pattern to Launch Threads
Defining the Executor Service Pattern, a First Simple Example
Comparing the Runnable and the Executor Service Patterns
Understanding the Waiting Queue of the Executor Service
Wrapping-up the Executor Service Pattern
From Runnable to Callable: What Is Wrong with Runnables?
Defining a New Model for Tasks That Return Objects
Introducing the Callable Interface to Model Tasks
Introducing the Future Object to Transmit Objects Between Threads
Wrapping-up Callables and Futures, Handling Exceptions

Introduction - Executor

Consider the below code snippet :

Runnable task = () -> System.out.println("Hello World ");

Thread thread= new Thread(task);

t

--

1
2
3
4
5
6

Executor Pattern Aims to Fix These Issues

1. A task is an instance of Runnable
2. The Thread is launched
3. The Thread is created on demand by user
4. Once the task is done ,thread dies.
5. Threads are expensive resource

Executor Pattern

How can we improve the use of Threads ,as resources ?

-By creating pools of ready to use threads ,and using them .

Instead of creating a thread with task as a parameter ,we pass a task to the pool of threads ,that will execute it .

We Need 2 patterns :

1. One to create a pool of Threads
2. Second one to pass a task to this pool

```
Executor executor =
                Executors.newSingleThreadExecutor();
Runnable task= () -> System.out.println("I Run ");
executor.execute(task);
```

Executor Service Pattern

```
public interface Executor {
        void execute(Runnable Task);
}
```

Executor interface is a simple interface to support launching new tasks.

ExecutorService is an extension of Executor

```
ExecutorService  singleThreadExecutor = Executors.newSingleExecutor();
```

-This will create a pool with only one thread in it .

The thread in SingleExecutorService pattern will be kept alive as long as pool is alive.

A pool of Thread is an instance of Executor interface

ExecutorService Pattern

ExecutorService is an extension of Executor

public interface ExecutorService extends Executor{

// 11 more methods..

}

The 2 most used executor services are :

1. Single Thread Executor

ExecutorService singleThreadExecutor = Executors.newSingleExecutor();

2. Fixed Thread pool Executor

ExecutorService fixedThreadExecutor= Executors.newFixedThreadPoolExecutor(8);

The thread in SingleExecutorService pattern will be kept alive as long as pool is alive.

Executor Service Pattern

```
Runnable task= () -> System.out.println("I run" );
```

Runnable Pattern :

```
Thread  thread= new Thread(task);
thread.start();
```

Here we have to create the thread

Thread will be created by Executors

Executor Pattern :

```
Executor executor = new Executors.singleThreadExecutor();
executor.execute(task);
```

The thread in SingleExecutorService pattern will be kept alive as long as pool is alive.

ExecutorService Pattern

```
Executor executor= Executors.newSingleThreadExecutor();
Runnable task1= ()-> someReallyLongProcess();
Runnable task2= ()-> anotherReallyLongProcess();

executor.execute(task1);
executor.execute(task2);
```

Suppose we run this code ,obviously task2 has to wait for task1 to complete.

To Handle this case ,Executor has a waiting queue :

Working of Waiting Queue :

- A task is added to waiting queue when no thread is available
- The task are executed in the order of their submission .

ExecutorService Pattern –Wrap Up

- Building an Executor is more efficient than creating threads on demand.
  - Executor will create the pool of threads and these threrads will be alive as long as the Executor is alive.
  - One given thread can execute as many task as we need.
- We can pass instances of Runnable to Executor
- The Executor has a Waiting Queue to handle multiple requests.
- A task can be removed from the waiting Queue

From Runnable to Callable

```
Runnable task= () -> someReallyLongProcess ();
    Executor executor=…;
    executor.execute (task);
```

With Runnable implementation :

- A task can not return the result
- A task can not throw the exception

There is no way to know whether task is done or not

Callable Introduction

We need a new model for our tasks with below features :

- With a method that returns a value
- And that can throw exception .
- We also need a new Object that acts as bridge between threads

```
public interface Callable<V>{
V call() throws Exception ;
}
```

Demo

Introducing the Future Object to Transmit Objects Between Threads

Executor interface does not handle Callable

ExecutorService interface has submit method to handle Callable

```
Future<T> submit (Callable<T> Task);
```

How does this Future Object Works?

- We create the Callable in main thread and the task which we want to execute .
- We pass this task to submit method of executor
- ExecutorService will execute this task and produce the result .
- This result has to be passed from ExecutorService to Main thread.
- Here comes the Future Object for your help.
- In fact the Executor will return the Future object which holds the result.

// In main thread

```
Callable<String> task= ()->buildPatientReport();

Future<String> future= executor.submit(task);
String result= future.get();
```

The future object is returned by submit() method call in main thread.

The get() method of future object can be called to return the produced string.

The get() method is blocking until the returning String is available.

Lab

Summary

In this lesson, you have learnt the following:

- Callable
- Executor

Add the notes here.

Review Question