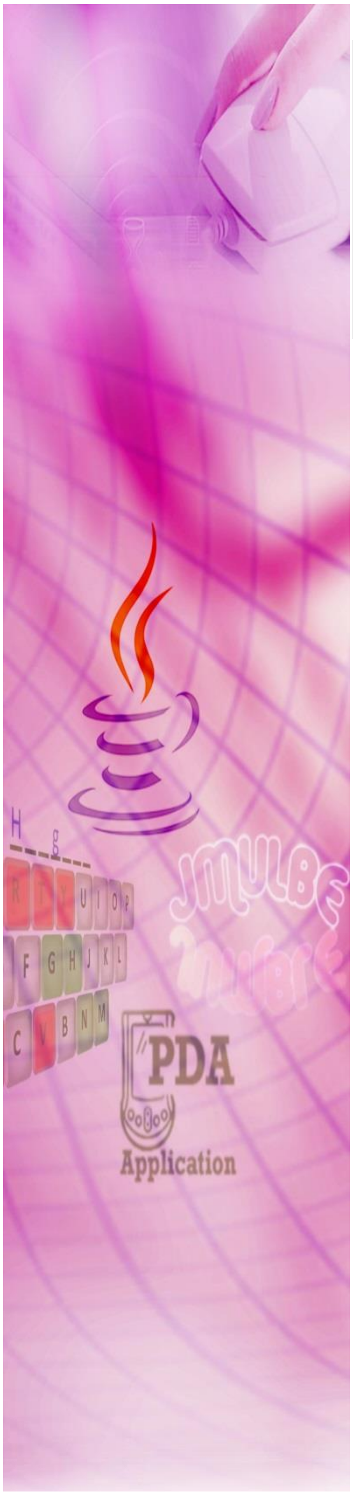

Session 8

Instructor Inputs

Session Overview

This session covers the Identifying Concurrency Synchronizers, Identifying Concurrency Collections, Implementing ExecutorService, and Implementing Fork/Join Framework topics of the Implementing Concurrency section of Chapter 6. In addition, it covers the Working with Input Stream section of Chapter 7 of the book – Programming in Java – SG.

In this session, the students will explore the concept of concurrency in Java. They will learn implementing atomic variables and locks in a multithreaded environment. Furthermore, they will learn reading data from various input streams.

Implementing Concurrency

Handling Tips

Start the session by telling the students that to achieve coordination and concurrency among the threads of an application, they can use the synchronizer classes provided in the `java.util.concurrent` package. Tell them that the use of synchronizer classes also minimizes data inconsistency and errors in an application. Discuss each of the classes and ensure that the students have gained a clear understanding. Focus on the situations where each of the preceding classes can be implemented. While explaining this topic, you can discuss the code given under the Identifying Concurrency Synchronizers subtopic in the Additional Examples topic.

Next, discuss the concurrency classes briefly by referring to the scenario given in the SG. Emphasize on the point that the use of the collection classes improves the performance of a concurrent application and also improves its scalability by allowing multiple threads to share a data collection. Next, tell them that in a multithreaded application, where there is a requirement of a large number of threads to perform operations, it is a cumbersome task to efficiently manage the resources of a computer and the performance of the application. In such a case, an executor can be used to improve the efficiency. Then, discuss the runnable and callable tasks and explain the difference between the preceding types of tasks. Further, explain all the executor interfaces along with their purpose, as given in the SG. Focus on how the executor service is used to overcome the limitations of an executor. While explaining this topic, you can discuss the code given under the Implementing ExecutorService subtopic in the Additional Examples topic.

Finally, discuss with the students about the concept of the fork/join framework. Tell the students that the preceding framework can be utilized to achieve the maximum CPU utilization in an environment where there are multiple processors available for a concurrent application. Focus on the concept of work stealing and how it can prove to be beneficial for improving the performance of an application. Briefly discuss the classes that can be used for implementing the fork/join framework. Then, explain the code to implement the fork/join framework, as given in the SG. While discussing the fork/join framework, you can discuss the information given under the Additional Inputs topic.

Additional Inputs

Implementing Fork/Join Framework

The pseudocode for decomposing a computationally intensive problem by using the fork/join framework is:

```
// pseudocode
Result solve(Problem problem) {
  if (problem.size < SEQUENTIAL_THRESHOLD)
    return solveSequentially(problem);
  else {
    Result left, right;
    INVOKE-IN-PARALLEL {
      left = solve(extractLeftHalf(problem));
      right = solve(extractRightHalf(problem));
    }
    return combine(left, right);
  }
}
```

In the preceding pseudocode, a problem to be solved is identified. Then, it is identified whether the problem needs to be solved sequentially or by applying the fork/join technique. If the fork/join technique is required, the problem is decomposed into smaller tasks, namely `left` and `right`. Further, the results of the preceding tasks are then combined to produce the final result.

Additional Examples

Identifying Concurrency Synchronizers

The following code demonstrates how to implement `CountDownLatch` to simulate the race of three players in a concurrent manner:

```
import java.util.concurrent.*;

public class Countdown {

    public static void main(String a[]) throws InterruptedException {
        CountDownLatch counter = new CountDownLatch(3);
        new Runner(counter, "Carl");
        new Runner(counter, "Joe");
        new Runner(counter, "Smith");
        System.out.println("Starting the race...");
        long countval = counter.getCount();
        while (countval > 0) {
            Thread.sleep(1000);
            System.out.println(countval);
            if (countval == 1) {
                System.out.println("Go...");
            }
            counter.countDown();
            countval = counter.getCount();
        }
    }
}
```

```
class Runner extends Thread {

    CountDownLatch timer;

    public Runner(CountDownLatch a, String b) {
        timer = a;
        this.setName(b);
        System.out.println(this.getName() + " is ready to run ");
        start();
    }

    public void run() {
        try {
            timer.await();
        } catch (InterruptedException e) {
            System.out.println("Death before start");
        }
        System.out.print("\n"+ this.getName() + " started running");
    }
}
```

In the preceding code, a new countdown latch is created. The number of operations after which the waiting threads will start executing is specified as `3`. Further, three threads are created where each thread represents a runner. All the runners wait until the countdown reaches zero, and then the race starts. Finally, the runners start to run.

Once the preceding code is executed, the following output is displayed:

```
Carl is ready to run
Joe is ready to run
Smith is ready to run
Starting the race...
3
2
1
Go...

Carl started running
Smith started running
Joe started running
```

Implementing ExecutorService

The following code demonstrates how to use `ExecutorService` with a fixed thread pool:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class Task implements Runnable {

    String taskname;

    public Task(String name) {
        taskname = name;
    }

    public void run() {

        System.out.println("The task name is - " + taskname);
        if (taskname.equals("Display stars")) {
            for (int i = 1; i <= 5; i++) {
                for (int j = 0; j < (5 - i); j++) {
                    System.out.print(" ");
                }
                for (int j = 1; j <= i; j++) {
                    System.out.print("*");
                }
                for (int k = 1; k < i; k++) {
                    System.out.print("*");
                }
                System.out.println();

            }
            for (int i = 5 - 1; i >= 1; i--) {
                for (int j = 0; j < (5 - i); j++) {
                    System.out.print(" ");
                }
                for (int j = 1; j <= i; j++) {
                    System.out.print("*");
                }
                for (int k = 1; k < i; k++) {
                    System.out.print("*");
                }
                System.out.println();
            }

        }
    }
}

class TaskExecutor {

    public static void main(String a[]) {
        Task task1 = new Task("Display stars");
        ExecutorService threadexecutor = Executors.newFixedThreadPool(1);
        System.out.println("Executor started");
        try {
            threadexecutor.execute(task1);
        } finally {
            threadexecutor.shutdown();
            try {
               threadexecutor.awaitTermination(Long.MAX_VALUE,
TimeUnit.NANOSECONDS);
                System.out.println("Executor shutdown");
            } catch (InterruptedException e) {
            }
        }
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Executor started
The task name is - Display stars
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
Executor shutdown
```

Working with Input Stream

Handling Tips

Start the discussion by referring to the scenario of the Java application, which converts a document from one format to another, such as .txt to .pdf, with the students. Then, start explaining the importance of the input stream in a programming language. While discussing about the input stream, you can explain the information given in the Additional Inputs topic. While discussing the `FileInputStream`, `BufferedInputStream`, and `BufferedReader` classes, explain the code given under the subtopics, Using the FileInputStream Class, Using the BufferedInputStream Class, Using the BufferedReader Class and Using the FileReader Class, respectively, as given in the Additional Examples topic.

Additional Inputs

Some of the additional input stream classes inside `java.io` package are:

- **The ByteArrayInputStream Class**: Contains an internal buffer that contains the bytes that may be read from the stream. An internal counter keeps a track of the next byte to be supplied by the `read()` method.
- **The DataInputStream Class**: Lets an application read the primitive Java data types from an input stream in a machine-independent way. The `DataInputStream` class is not necessarily safe for multithreaded access.
- **The FilterInputStream Class**: The `FilterInputStream` class contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality. The class overrides all methods of `InputStream` with the versions that pass all the requests to the contained input stream.

Additional Examples

Using the FileInputStream Class

Consider the following code that demonstrates whether a file is a valid file or not:

```
import java.io.FileDescriptor;
import java.io.FileInputStream;
import java.io.IOException;
```

```
public class FileInputStreamDemo
{
    public static void main(String[] args)
    {
        FileDescriptor fd = null;
        boolean bool = false;
        try (FileInputStream fis = new FileInputStream("file.txt"))
        {
            fd = fis.getFD();
            bool = fd.valid();
            System.out.println("Valid file: " + bool);
        }
        catch (IOException ex)
        {
            System.out.println(ex);
        }
    }
}
```

Once the preceding code is executed, the following output is displayed:

```
Valid file: true
```

In the preceding code, the `getFD()` method is used to return a `FileDescriptor` object that represents the connection to the actual file in the file system being used by this `FileInputStream`. The `valid()` method tests whether the file descriptor object is valid or not. The `System.out.println("Valid file: " + bool);` statement prints the output.

Using the BufferedInputStream Class

Consider the following code that demonstrates the `mark()`, `reset()`, and `markSupported()` methods of the `BufferedInputStream` class:

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class BufferedInputStreamDemo
{
    public static void main(String[] args)
    {
        boolean bool;
        try (BufferedInputStream bis = new BufferedInputStream(new
FileInputStream("file.txt")))
        {
            bool = bis.markSupported();
            System.out.println("Support for mark() and reset() : " + bool);
            System.out.println("Char : " + (char) bis.read());
            System.out.println("Char : " + (char) bis.read());
            System.out.println("Char : " + (char) bis.read());
            bis.mark(0);
            System.out.println("Char : " + (char) bis.read());
            System.out.println("reset() invoked");
            bis.reset();
```

```
                System.out.println("Char : " + (char) bis.read());

            }
            catch (IOException ex)
            {
                System.out.println(ex);
            }
        }
    }
```

In the preceding code, the `markSupported()` method is used to test whether the input stream supports the `mark()` and `reset()` methods or not. The `mark()` method is used to mark the current position in the input stream. A subsequent call to the `reset()` method repositions this stream at the last marked position. Further, the `System.out.println("Char : " + (char) bis.read());` statement prints the output.

Using the BufferedReader Class

Consider the following code that demonstrates whether the stream is ready for reading or not:

```
import java.io.*;

public class BufferedReaderDemo
{
    public static void main(String args[]) throws IOException
    {
        InputStream is;
        InputStreamReader isr;
        is = new FileInputStream("file.txt");
        isr = new InputStreamReader(is);
        try (BufferedReader br = new BufferedReader(isr))
        {
            int value;
            if (br.ready())
            {
                System.out.println("Stream is ready");
                while ((value = br.read()) != -1)
                {
                    char c = (char) value;
                    System.out.println(c);
                }
            }

        }
    }
}
```

In the preceding code, the `BufferedReaderDemo` class is created. The `ready()` method is used to test whether the stream is ready for the purpose of reading or not. The `System.out.println(c);` statement prints the output.

Using the FileReader Class

Consider the following code that reads the data from a file:

```
import java.io.*;
public class FileReaderDemo
{
    public static void main(String[] args)
    {
        try{
        File f=new File("test.txt");
        FileReader fr1=new FileReader(f);
        char[] c1=new char[(int)f.length()];
        fr1.read(c1,0,c1.length);
        System.out.println("\n\nSecond way\n");
        for(int i=0;i<c1.length;i++)
        {
            System.out.print(c1[i]);
        }
        fr1.close();
        }
        catch(Exception e){
            System.out.println(e);
        }

    }
}
```

In the preceding code, an object of `File` class is created to represent a file stream. This object is passed to the `FileReader` class constructor. The `read()` method is used to read the file.

FAQs

- *Is it possible to open the same file for reading as well as writing?*
  Ans: Yes, it is possible.

- *Which exception will be thrown if we attempt to read the data from a closed stream?*
  Ans: The `java.io.IOException` exception will be thrown.

- *Is it possible to schedule tasks to run once or periodically?*
  Ans: Yes, it is possible by using the `ScheduledExecutorService` interface. This interface extends `ExecutorService`.

- *What will happen if the number of threads in a fixed thread pool is specified as zero?*
  Ans: In this case, the `IllegalArgumentException` exception will be thrown.

- *What will happen if an executor service is shut down before executing a task?*

  Ans: In this case, the `java.util.concurrent.RejectedExecutionException` exception will be thrown.

- *How can we estimate the number of threads that are executing or stealing tasks in an application that implements the fork/join framework?*

  Ans: The `getActiveThreadCount()` method can be used to estimate the number of threads that are executing or stealing tasks in an application that implements the fork/join framework.