Java Multithreading

MODULE NAME: OBJECT ORIENTED PROGRAMMING WITH JAVA

MODULE CODE: CST 05206

INSTRUCTORS: JAMILA L.

Multithreading definition

Multithreading is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer.

- Multithreading is a programming concept in which the application can create a small unit of tasks to execute in parallel.
- Multithreading in Java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

Multithreading Vs. Multiprocessing

- Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation, etc.

Why is Multithreading needed?

Multithreading enables us to improve the responsiveness of a system.

- For example in a web browser, if everything ran in a single thread, then system would be completely unresponsive whenever data was being fetched to display.
- For example, if it takes 10 seconds to fetch the data, then in that 10 seconds we wont be able to do anything else in the web browser like opening new tabs, or even closing the web browser.
- So running different parts of a program in different threads concurrently helps improve the responsiveness of a system.

What is Thread Priority?

When we create a thread, we can assign its priority.

We can set different priorities to different Threads but it doesn't guarantee that a higher priority thread will execute first than a lower priority thread.

The thread scheduler is the part of Operating System implementation and when a Thread is started, its execution is controlled by Thread Scheduler and JVM doesn't have any control over its execution.

How does Java Support Multithreading?

- Java has great support for multithreaded applications.
- Java supports multithreading through Thread class.
- Java Thread allows us to create a lightweight process that executes some tasks.
- We can create multiple threads in our program and start them.
- Java runtime will take care of creating machine-level instructions and work with OS to execute them in parallel.

How to write Multithreaded Programs in Java?

We can create threads in Java using the following

- Extending the thread class
- Implementing the runnable interface
- Implementing the callable interface
- By using the executor framework along with runnable and callable tasks

How to write Multithreaded Programs in Java?

```
Thread t = new Thread(new Runnable(){

    @Override

    public void run() {

    }

});
```

How to write Multithreaded Programs in Java?

Here we are creating a Runnable as an anonymous class. If you are familiar with lambda expressions.

```
Runnable runnable = () -> System.out.println("Hello");

runnable.start();
```

Java Thread class

- Java provides Thread class to achieve thread programming.
- Thread class provides constructors and methods to create and perform operations on a thread.
- Thread class extends Object class and implements Runnable interface.

Extending the Thread Class

- In order to create a piece of code which can be run in a thread, we create a class and then extend the **thread** class.
- The task being done by this piece of code needs to be put in the **run()** function.
- In the below code you can see that **worker** is a class which extends the **thread** class, and the task of printing numbers 0 to 5 is being done inside the **run()** function.

Extending the Thread Class

```
class Worker extends Thread {
	@Override
	public void run() {
		for (int i = 0; i <= 5; i++) {

	System.out.println(Thread.currentThread().getName() + ": " +
i);
		}
	}
}
```

Thread.currentThread().getName() is used to get the name of the current thread which is running the code.

Extending the Thread Class

In order to create a thread, we just need to create an instance of the worker class. And then we can start the thread using the start() function.

```
public class ThreadClassDemo {
      public static void main(String[] args) {
            Thread t1 = new Worker();
            Thread t2 = new Worker();
            Thread t3 = new Worker();
            t1.start();
            t2.start();
            t3.start();
      }
} //In the above code, we are creating 3 threads (t1,t2 and t3) from the worker class.
Then we are starting the threads using the start() function.
```

Extending the Thread Class

```
// Here is the final code for creating a thread by extending a thread class:
class Worker extends Thread {
        @Override
        public void run() {
                for (int i = 0; i <= 5; i++) {
                        System.out.println(Thread.currentThread().getName() + ": " + i);
                }
        }
}
public class ThreadClassDemo {
        public static void main(String[] args) {
                Thread t1 = new Worker();
                Thread t2 = new Worker();
                Thread t3 = new Worker();
                t1.start();
                t2.start();
                t3.start();
        }
}
```

Advantages of Java Multithreading

- It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
- You can perform many operations together, so it saves time.
- Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Concepts of Streams

Introduction to Stream in Java

- Java provides I/O Streams to read and write data where, a Stream represents an input source or an output destination which could be a file, i/o devise, other program etc.
- In general, a Stream will be an input stream or, an output stream.

1. InputStream − This is used to read data from a source.
2. OutputStream − This is used to write data to a destination.

Types of Stream in Java

Based on the data they handle there are two types of streams −

1. Byte Streams − These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.
2. Character Streams − These handle data in 16 bit Unicode. Using these you can read and write text data only.

Standard Streams

Java provides 3 standard streams representing the input and, output devices.

Standard Input − This is used to read data from user through input devices. keyboard is used as standard input stream and represented as System.in.

Standard Output − This is used to project data (results) to the user through output devices. A computer screen is used for standard output stream and represented as System.out.

Standard Error − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

Example

Following Java program reads the data from user using BufferedInputStream and writes it into a file using BufferedOutputStream.

```
import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.FileOutputStream;

import java.io.IOException;

public class BufferedInputStreamExample {

   public static void main(String args[]) throws IOException {

      //Creating an BufferedInputStream object

      BufferedInputStream inputStream = new BufferedInputStream(System.in);

      byte bytes[] = new byte[1024];
```

Example cont…

```
System.out.println("Enter your data ");
        //Reading data from key-board
        inputStream.read(bytes);
        //Creating BufferedOutputStream object
        FileOutputStream out= new FileOutputStream("D:/myFile.txt");
        BufferedOutputStream outputStream = new BufferedOutputStream(out);
        //Writing data to the file
        outputStream.write(bytes);
        outputStream.flush();
        System.out.println("Data successfully written in the specified file");
    }
}
```

Java Networking

What Is Java Networking?

Java networking is the concept of connecting two or more computing devices to share resources.

The application layer of a Java program communicates with the network. The java.net package contains all of the Java networking classes and interfaces.

Java Networking Terminologies

The widely used Java networking terminologies are given below:

- IP Address
- Protocol
- Port Number
- MAC Address
- Connection-oriented and connection-less protocol
- Socket

Network Protocols

The java.net package supports 2 protocols. These are their names:

1. **The Transmission Control Protocol (TCP):**

TCP enables secure communication between two applications. It is commonly applied together with the Internet Protocol, which is referred to as TCP/IP.

2. **User Datagram Protocol (UDP):**

UDP is a connectionless protocol allowing data packets to be sent between applications.

Java Networking Classes

The Java programming language's java.net package contains a number of classes that make it simple to access network resources. The following classes are included in the java.net package:

- CacheRequest
- CookieHandler
- CookieManager
- DatagramPacket
- InetAddress
- ServerSocket
- Socket
- Proxy
- URL & URLConnection etc…

Socket

In Java, a socket is one end of a two-way communication link between two programs running on a network.

- A socket is always associated with a port number to allow the TCP layer to identify the application to which data is being sent.
- The Socket class is used to create socket objects that aid users in performing all basic socket operations. Users can perform various networking tasks such as sending data, reading data, and closing connections.

Socket Programming

Socket programming is a method of communicating between two nodes on a network.

One socket (node) listens on a specific port at an IP address, while the other socket reaches out to form a connection. While the client attempts to contact the server, the server creates the listener socket.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Socket Programming cont…

The steps involved in establishing a TCP connection between two computers using socket programming are as follows:

Step 1 - The server creates a ServerSocket object and specifies which port number communication will take place.

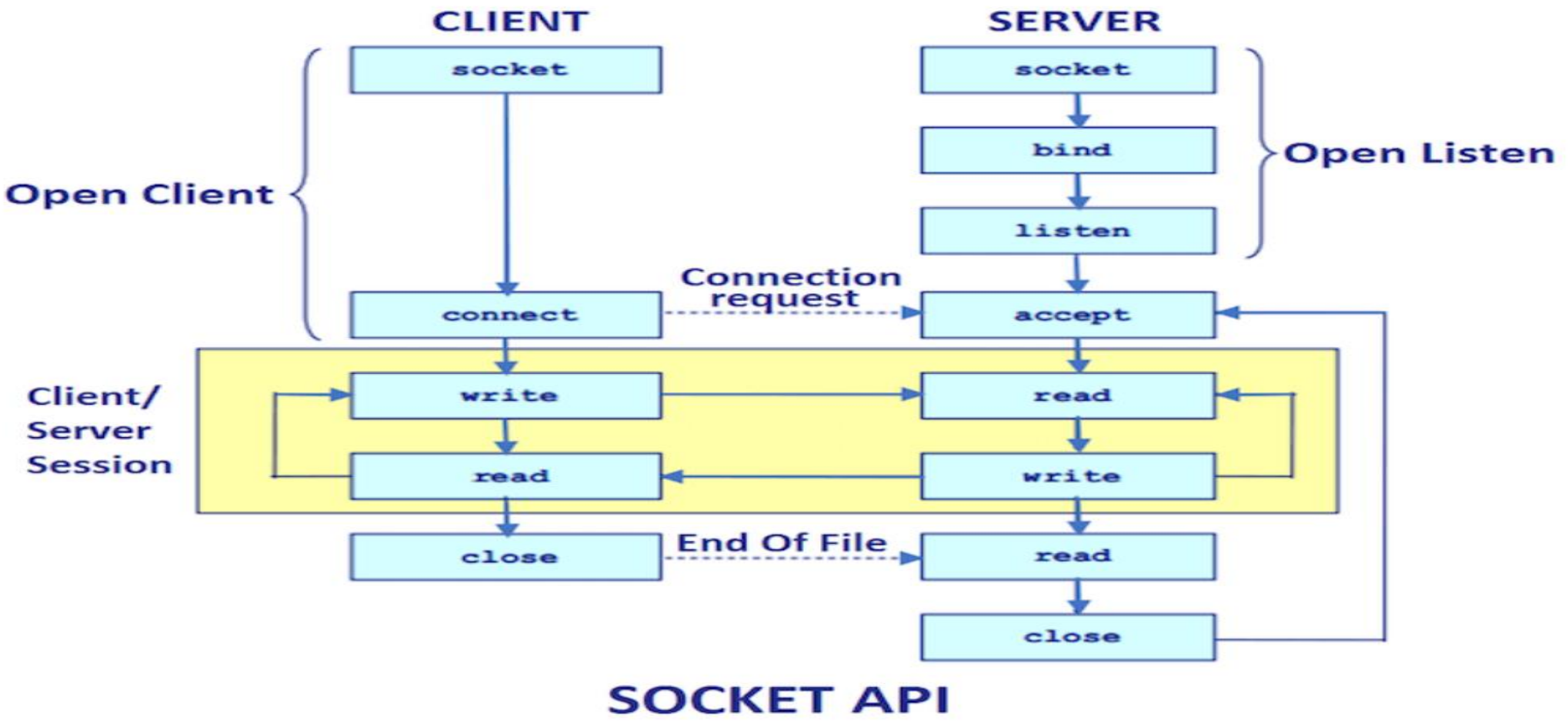
Step 2 - After the ServerSocket object is instantiated, the server invokes the ServerSocket class's accept() method. This program waits for a client to connect to the server on the specified port.

Step 3 - While the server is idle, a client creates an object of the Socket class and specifies the server name and port number to connect to.

Step 4 - Following the preceding step, the function Object() { [native code] } of the Socket class attempts to connect the client to the specified server and port number. If communication is authenticated, the client receives a Socket object capable of interacting with the server.

Step 5 - The accept() method on the server returns a reference to a new socket on the server that is connected to the client's socket.

Socket API


CLIENT
SERVER
socket
socket
bind
listen
Open Client
Open Listen
Connection request
connect
accept
Client/ Server Session
write
read
read
write
End Of File
close
read
close
SOCKET API

Socket Programming cont…

After the connections have been stabilized, communication can take place via I/O streams.

- A socket class object has an OutputStream as well as an InputStream.
- The OutputStream of the client is connected to the server's InputStream, which is then combined with the server's OutputStream.
- TCP or Transmission Control Protocol is a two-way communication protocol. As a result, information can be transmitted over both streams simultaneously.

Example of Java Socket Programming

Creating Server:

To create the server application, we need to create the instance of ServerSocket class. Here, we are using 6666 port number for the communication between the client and server. You may also choose any other port number. The accept() method waits for the client. If clients connects with the given port number, it returns an instance of Socket.

```
ServerSocket ss=new ServerSocket(6666);

Socket s=ss.accept();//establishes connection and waits for the client
```

Example of Java Socket Programming

Creating Client:

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

```
Socket s=new Socket("localhost",6666);
```

Example: File: MyServer.java

```
import java.io.*;
import java.net.*;
public class MyServer {
  public static void main(String[] args){
     try{
     ServerSocket ss=new ServerSocket(6666);
     Socket s=ss.accept();//establishes connection
     DataInputStream dis=new DataInputStream(s.getInputStream());
     String  str=(String)dis.readUTF();
     System.out.println("message= "+str);
     ss.close();
     }catch(Exception e){System.out.println(e);}
  }
}
```

Example: File: MyClient.java

```
import java.io.*;
import java.net.*;
public class MyClient {
 public static void main(String[] args) {
    try{
    Socket s=new Socket("localhost",6666);
    DataOutputStream dout=new DataOutputStream(s.getOutputStream());
    dout.writeUTF("Hello Server");
    dout.flush();
    dout.close();
    s.close();
    }catch(Exception e){System.out.println(e);}
 }
}
```

URL Processing

The Java URL class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web.

The java.net.URL class represents a URL and has a complete set of methods to manipulate URL in Java.

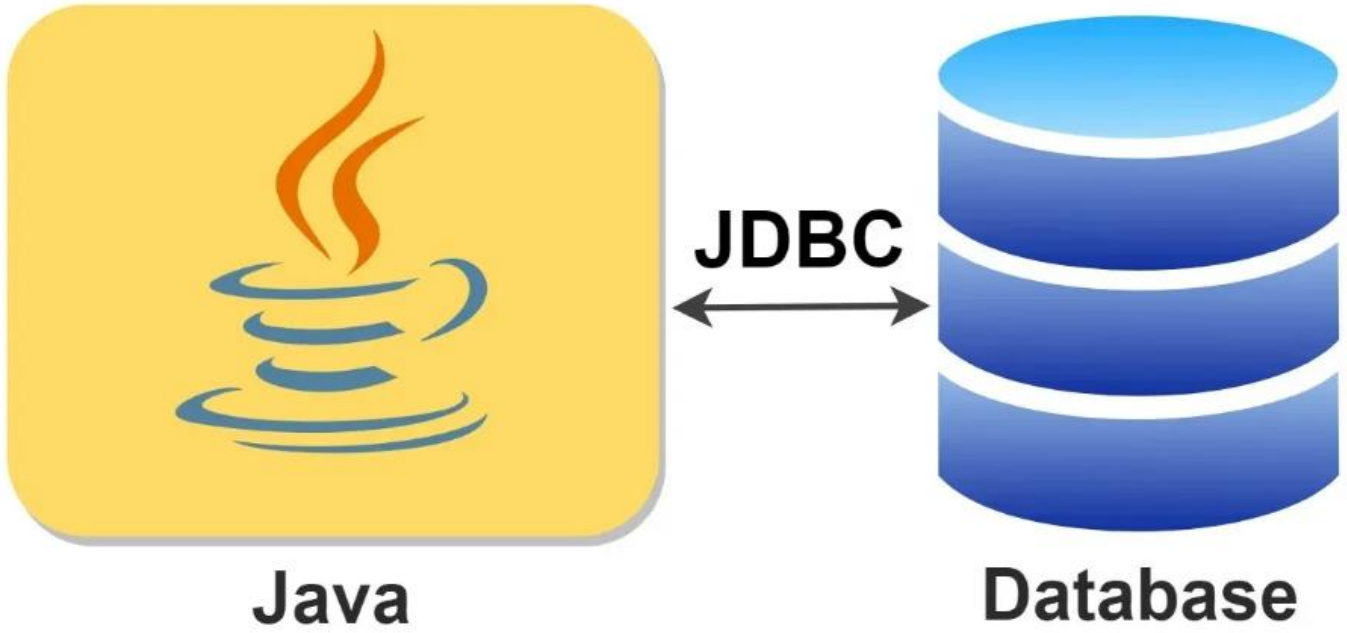
URL Constructors

Sr.No.	Constructors & Description
1	**public URL(String protocol, String host, int port, String file) throws MalformedURLException** Creates a URL by putting together the given parts.
2	**public URL(String protocol, String host, String file) throws MalformedURLException** Identical to the previous constructor, except that the default port for the given protocol is used.
3	**public URL(String url) throws MalformedURLException** Creates a URL from the given String.
4	**public URL(URL context, String url) throws MalformedURLException** Creates a URL by parsing together the URL and String arguments.

Java Database connection

JDBC

JDBC is the Java API that manages connecting to a database, issuing queries and commands, and handling result sets obtained from the database.



JDBC

The steps for connecting to a database with JDBC are as follows:

i. Install or locate the database you want to access.

ii. Include the JDBC library.

iii. Ensure the JDBC driver you need is on your classpath.

iv. Use the JDBC library to obtain a connection to the database.

v. Use the connection to issue SQL commands.

vi. Close the connection when you are finished.

*** **For more explanation See the JDBC steps.pdf**

Thank you!

for listen