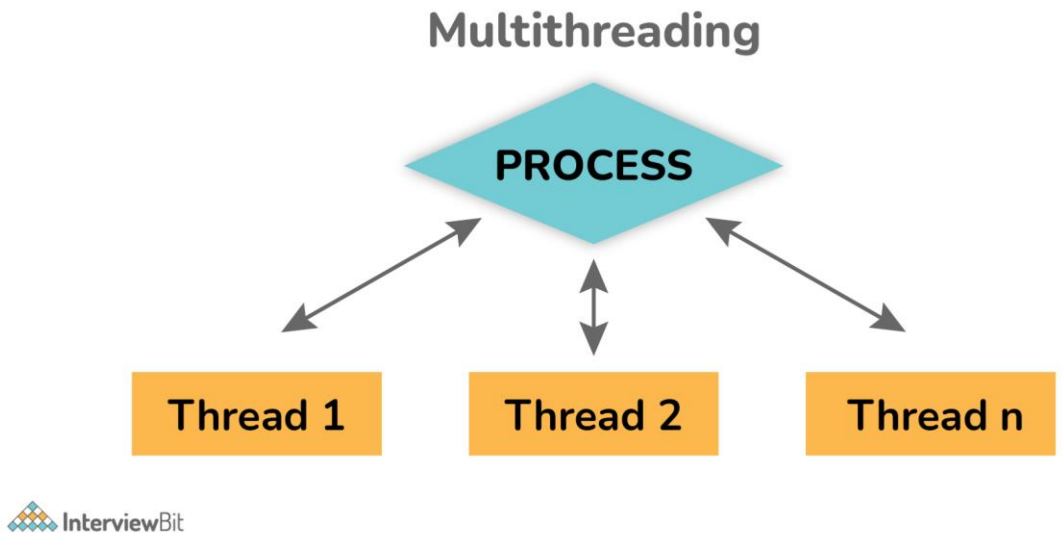
TCP-Multithreading

Contents

Thread Basics

- A **thread** is a flow of control through a program.



- Unlike a process, a thread does not have a separate allocation of memory but shares memory with other threads created by the same application.
  - This means that servers using threads do not exhaust their supply of available memory, as they were prone to do when creating many separate processes.
  - In addition, the threads created by an application can share global variables
- The operating system has the role to determine which thread to execute among the many threads running according to two factors:
  - Thread priority (1–10, in increasing order of importance) in Java.
  - Whether the CPU scheduling algorithm is preemptive or non-preemptive.

Multithreaded server

```
public class Multi_TCP_Server {
    private static ServerSocket serverSocket;
    private static final int PORT = 1234;

    public static void main(String[] args) throws IOException {
        serverSocket = new ServerSocket(PORT);
        System.out.println("The server is running...");

        do {
            Socket client = serverSocket.accept();
            System.out.println("\n*****New client connected...******");
            ClientHandler clientHandler = new ClientHandler(client);
            clientHandler.start();
        } while (true);
    }
}

class ClientHandler extends Thread {
    private Socket client;
    private Scanner input;
    private PrintWriter output;

    ClientHandler(Socket client) throws IOException {
        this.client = client;
        input = new Scanner(client.getInputStream());
        output = new PrintWriter(client.getOutputStream(), true);
    }

    @Override
    public void run() {
        String recieved;
        do {
            recieved = input.nextLine();
            output.println("ECHO: " + recieved);
        } while (!recieved.equals("QUIT"));
        if (client != null) {
            try {
                System.out.println("****Closing connection...****");
                client.close();
            } catch (IOException ex) {
                System.out.println(ex);
            }
        }
    }
}
```

- Create a class named *Multi_TCP_Server*.
- Create a *ServerSocket* object and define port number.
- Create *main* method and initialize *serverSocket*.

```
public class Multi_TCP_Server {
    private static ServerSocket serverSocket;
    private static final int PORT = 1234;

    public static void main(String[] args) throws IOException {
        serverSocket = new ServerSocket(PORT);
        System.out.println("The server is running...");
    }
}
```

- Inside *main*, create an infinite *do* − *while* loop.
- Inside *do*, create a *Socket* object to accept an incoming connection.
- Create an instance of *ClientHandler* class and pass the client’s socket to the constructor.
    - *ClientHandler* is a multi-threaded class we will write to handle the connections of the clients.
- Call *start* method of *clientHandler* object.

```
do {
    Socket client = serverSocket.accept();
    System.out.println("\n*****New client connected...******");
    ClientHandler clientHandler = new ClientHandler(client);
    clientHandler.start();
} while (true);
```

- Create a class named *ClientHandler* that extends *Thread* class.
- Inside the class, define *Socket*, *Scanner*, *PrintWriter* objects.
- Inside class, create constructor that accepts a *Socket* object.

```
class ClientHandler extends Thread {
    private Socket client;
    private Scanner input;
    private PrintWriter output;

    ClientHandler(Socket client) {

    }
```

- Inside class, override $run()$ method.
- Inside the constructor, setup the $socket$ variable to the $socket$ object passed as a parameter.
- Initialize the $input$ object and $output$ object to get the input/output streams from socket.

```
ClientHandler(Socket client) throws IOException {
    this.client = client;
    input = new Scanner(client.getInputStream());
    output = new PrintWriter(client.getOutputStream(), true);
}
```

- In the run method, define a $String$ object for receiving messages.
- Define a $do - while$ loop that will run until the client sends "QUIT" message.
- Inside do, receive the client's message. Then reply to him.

```
public void run() {
    String recieved;
    do {
        recieved = input.nextLine();
        output.println("ECHO: " + recieved);
    } while (!recieved.equals("QUIT"));
```

- After $while$, setup $try - catch$ block for closing the connection.

```
if (client != null) {
    try {
        System.out.println("****Closing connection...****");
        client.close();
    } catch (IOException ex) {
        System.out.println(ex);
    }
}
```

✓ What is the purpose of the $do - while$ loop in the $main$ and the one in the $ClientHandler$?

Client

- The same client code written before (with minor modifications), does not have to implement any multithreading mechanism.

```
public class Multi_TCP_Client {
    private static InetAddress host;
    private static final int PORT=1234;

    public static void main(String[] args) throws UnknownHostException {
        host = InetAddress.getLocalHost();
        try {
            sendMessages();
        }catch (Exception e){
            System.out.println(e);
        }
    }

    private static void sendMessages() throws IOException {
        Socket link = new Socket(host, PORT);
        Scanner input = new Scanner(link.getInputStream());
        PrintWriter output = new PrintWriter(link.getOutputStream(), true);

        Scanner userInp = new Scanner(System.in);
        String msg, rspns;
        do{
            System.out.println("Enter a message (QUIT to exit): ");
            msg = userInp.nextLine();
            output.println(msg);
            rspns = input.nextLine();
            System.out.println("SERVER> "+rspns+"\n");
        }while (!msg.equals("QUIT"));

        System.out.println("Closing connection");
        link.close();
    }
}
```

- Create a class named *MultiEchoClient*.
- Inside the class, define *InetAddress* object and port number.
- Create *main* method.

```
public class Multi_TCP_Client {
    private static InetAddress host;
    private static final int PORT = 1234;

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

- Inside *main*, initialize *InetAddress* object to return the IP address of the local machine inside a *try − catch*.
- Call the static method *sendMessages* to communicate with the server.

```
public static void main(String[] args) throws UnknownHostException {
    host = InetAddress.getLocalHost();
    try {
        sendMessages();
    } catch (Exception e) {
        System.out.println(e);
    }
}
```

- Create a static method *sendMessages*.
- Define *Socket* object to get the server's IP and service's port.
- Create *Scanner*/*PrintWriter* object for receiving/sending messages.

```
private static void sendMessages() throws IOException {
    Socket link = new Socket(host, PORT);
    Scanner input = new Scanner(link.getInputStream());
    PrintWriter output = new PrintWriter(link.getOutputStream(), true);
```

- Create *Scanner* object to read user's input from keyboard.
- Create two strings, one for sending messages and the other for receiving the messages.
- Define a *do − while* loop that runs until the user write "QUIT".

```
Scanner userInp = new Scanner(System.in);
String msg, rspns;
do {

} while (!msg.equals("QUIT"));
```

- Inside *do*, prompt the user to enter a message.
- Send the message to the server.
- Receive the reply from the server.
- Print the server's reply to the console.

```
do {
    System.out.println("Enter a message (QUIT to exit): ");
    msg = userInp.nextLine();
    output.println(msg);
    rspns = input.nextLine();
    System.out.println("SERVER> " + rspns + "\n");
} while (!msg.equals("QUIT"));
```

- After the do … $while()$ block, close the connection.

```
System.out.println("Closing connection");
link.close();
```

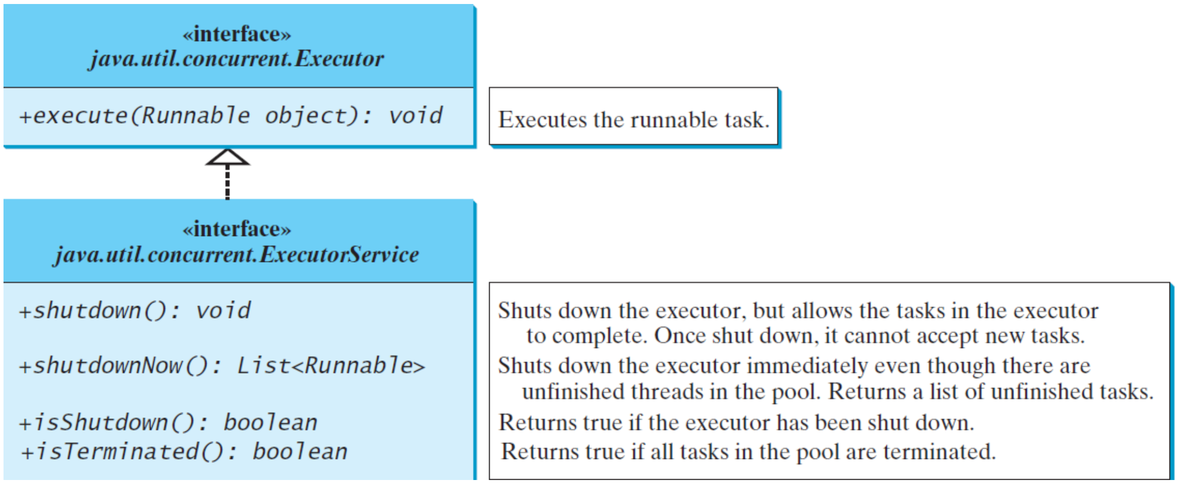
- Now, run the server and then run multiple instances of the client.

To Do

1. Replace the line: $!msg.equals("QUIT")$ by $msg\ != "QUIT"$
2. For the $PrintWriter$ object, remove the $autoFlush$ option.

Thread Pool

- In the previous discussion, you learned how to create task classes using $Runnable$ and $Thread$. This approach is convenient for a single task execution.
    - But it is not efficient for a large number of tasks because you have to create a thread for each task.
    - Starting a new thread for each task could limit throughput and cause poor performance.
- A thread pool can be used to execute tasks efficiently.
    - Using a thread pool is an ideal way to manage the number of tasks executing concurrently.
- Java provides the $Executor$ interface for executing tasks in a thread pool and the $ExecutorService$ interface for managing and controlling tasks.
    - $ExecutorService$ is a subinterface of $Executor$.


«interface»
java.util.concurrent.Executor
+execute(Runnable object): void
«interface»
java.util.concurrent.ExecutorService
+shutdown(): void
+shutdownNow(): List<Runnable>
+isShutdown(): boolean
+isTerminated(): boolean

Executes the runnable task.

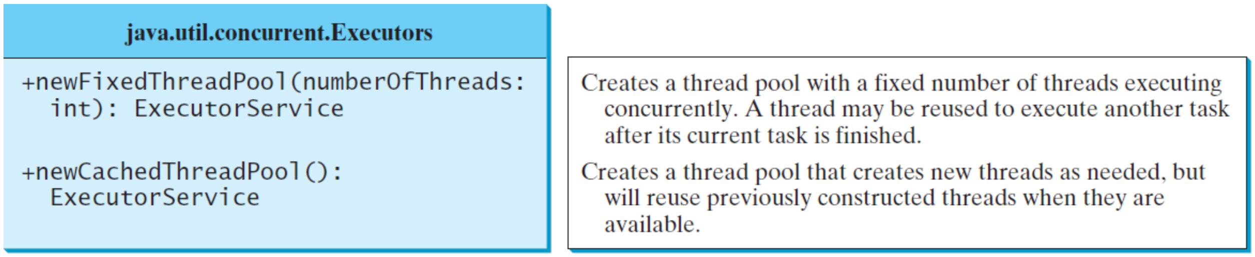
Shuts down the executor, but allows the tasks in the executor to complete. Once shut down, it cannot accept new tasks.

Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

Returns true if the executor has been shut down.

Returns true if all tasks in the pool are terminated.

- To create an $Executor$ object, use the static methods in the $Executors$ class.


java.util.concurrent.Executors
+newFixedThreadPool(numberOfThreads: int): ExecutorService
+newCachedThreadPool(): ExecutorService

Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

Multithreaded Server

```
public class TCPServerThreadPool {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(1234);
        ExecutorService pool = Executors.newFixedThreadPool(3);
        System.out.println("Waiting for clients...");
        while (true){
            Socket link = serverSocket.accept();
            Thread thread = new Thread(new ClientHandler(link));
            pool.execute(thread);
        }
    }
}

class ClientHandler implements Runnable{
    Socket link = null;
    ClientHandler(Socket link){
        this.link = link;

    }
    @Override
    public void run() {
        try{
            System.out.println("Client:
"+link.getRemoteSocketAddress()+" is connected");
            DataInputStream inputStream = new
DataInputStream(link.getInputStream());
            DataOutputStream outputStream = new
DataOutputStream(link.getOutputStream());

            int n = inputStream.readInt();
            int sum = 0;
            for (int i = 0; i < n; i++) {
                sum += i;
            }
            TimeUnit.SECONDS.sleep(10);
            outputStream.writeInt(sum);
            System.out.println("Client:
"+link.getRemoteSocketAddress()+" has finished");
            System.out.println("****************************");
        }catch (IOException ex){
            System.out.println(ex);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

- Create a class named *TCPServerThreadPool*.
- Inside the class define the *main* method.
- Inside *main*, Initialize *Serversocket* object and *ExecutorService* object.

```java
public class TCPServerThreadPool {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(1234);
        ExecutorService pool = Executors.newFixedThreadPool(3);
        System.out.println("Waiting for clients...");

    }
}
```

- Inside *main*, create a *while* loop.
- Inside *while* block, initialize *Socket* object to accept the connection from a client.
- Initialize a *Thread* object, pass the *ClientHandler* object as an argument.
- Execute the task by the *pool* object.

```java
while (true){
    Socket link = serverSocket.accept();
    Thread thread = new Thread(new ClientHandler(link));
    pool.execute(thread);
}
```

- In the same file, create *ClientHandler* class that implements *Runnable* interface.
- Initialize *Socket* object in the constructor.

```java
Socket link = null;
ClientHandler(Socket link){
    this.link = link;
}
```

- Implement the *run* method.
- Inside *run*, setup *try* … *catch* block.

```java
@Override
public void run() {
    try{

}catch (){
}
}
```

- Inside try, print a message informing the a new connection is accepted.
- Inside try, initialize $DataInputStream$ for data input.
- Inside try, initialize $DataOutputStream$ for data output.

```
try{
    System.out.println("Client: "+link.getRemoteSocketAddress()+" is connected");
    DataInputStream inputStream = new DataInputStream(link.getInputStream());
    DataOutputStream outputStream = new DataOutputStream(link.getOutputStream());
}
```

- Inside try, read the integer n from the client.
- Inside try, calculate the summation of n.

```
int n = inputStream.readInt();
int sum = 0;
for (int i = 0; i < n; i++) {
    sum += i;
}
```

- Let the server sleep for 10 seconds to simulate multi-client connection.
- Send the result to the client.

```
TimeUnit.SECONDS.sleep(10);
outputStream.writeInt(sum);
System.out.println("Client: "+link.getRemoteSocketAddress()+" has finished");
System.out.println("*****************************");
```

- Catch the exceptions.

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

Client

```
public class TCPClinetThreadPool {
    public static void main(String[] args) throws IOException {
        Socket link = new Socket("localhost", 1234);
        DataInputStream inputStream = new
DataInputStream(link.getInputStream());
        DataOutputStream outputStream = new
DataOutputStream(link.getOutputStream());
        Scanner scn = new Scanner(System.in);
        System.out.println("Enter a range: ");
        int n = scn.nextInt();
        outputStream.writeInt(n);
        int result = inputStream.readInt();
        System.out.println("The sum of 1 to "+ n + " = " + result);
    }
}
```

Task

Create a client-server GUI application that lets the client enter a username and password. If the username and password are valid, the server sends the time and date to the client. If the username and password are not valid, the server sends an image to the client.