

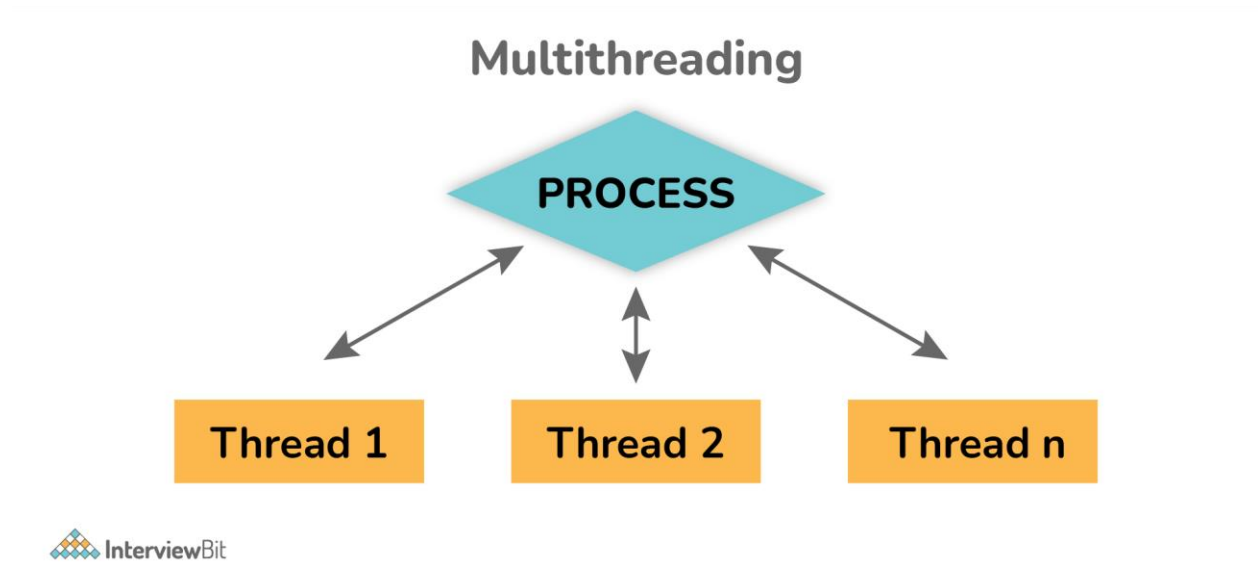
TCP-Multithreading

Contents

Thread Basics	2
Multithreaded server	3
Client	7
Thread Pool	10
Multithreaded Server	11
Client	12

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.

Multithreaded server

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

    public static void main(String[] args) throws IOException{
        try {
            serverSocket = new ServerSocket(PORT);
        } catch (IOException ioEx) {
            System.out.println("\nUnable to set up port!");
            System.exit(1);
        }
        do {
            Socket client = serverSocket.accept();
            System.out.println("\nNew client accepted.\n");
            ClientHandler handler =
                new ClientHandler(client);
            handler.start();
        } while (true);
    }

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

        public ClientHandler(Socket socket) {
            client = socket;
            try {
                input = new Scanner(client.getInputStream());
                output = new PrintWriter(
                    client.getOutputStream(), true);
            } catch (IOException ioEx) {
                ioEx.printStackTrace();
            }
        }

        public void run() {
            String received;
            do {
                received = input.nextLine();
                output.println("ECHO: " + received);
            } while (!received.equals("QUIT"));
            try {
                if (client != null) {
                    System.out.println(
                        "Closing down connection...");
                    client.close();
                }
            } catch (IOException ioEx) {
                System.out.println("Unable to disconnect!");
            }
        }
    }
}
```

- Create a class named *MultiEchoServer*.
- Create a *ServerSocket* object and define port number.
- Create *main* method and initialize *serverSocket* object inside *try* – *catch* block.

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

    public static void main(String[] args) throws IOException {
        try {
            serverSocket = new ServerSocket(PORT);
        } catch (IOException ioEx) {
            System.out.println("\nUnable to set up port!");
            System.exit(1);
        }
    }
}
```

- 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("\nNew client accepted.\n");
    ClientHandler handler =
        new ClientHandler(client);
    handler.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;

    public ClientHandler(Socket socket) {

    }

    public void run() {
    }
}
```

- Inside class, override *run()* method.
- Inside the constructor, setup the *socket* variable to the *socket* object passed as a parameter.
- Setup *try – catch* block.
- Inside *try*, initialize the *input* object and *output* object to get the input/output streams from socket.
- Catch the exception.

```
public ClientHandler(Socket socket) {
    client = socket;
    try {
        input = new Scanner(client.getInputStream());
        output = new PrintWriter(
            client.getOutputStream(), true);
    } catch (IOException ioEx) {
        ioEx.printStackTrace();
    }
}
```

- 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 received;

    do {
        received = input.nextLine();
        output.println("ECHO: " + received);
    } while (!received.equals("QUIT"));
}
```

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

```
try {
    if (client != null) {
        System.out.println(
            "Closing down connection...");
        client.close();
    }
} catch (IOException ioEx) {
    System.out.println("Unable to disconnect!");
}
```

- ✓ 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 MultiEchoClient {
    private static InetAddress host;
    private static final int PORT = 1234;

    public static void main(String[] args) {
        try {
            host = InetAddress.getLocalHost();
        } catch (UnknownHostException uhEx) {
            System.out.println("\nHost ID not found!\n");
            System.exit(1);
        }
        sendMessages();
    }

    private static void sendMessages() {
        Socket socket = null;
        try {
            socket = new Socket(host, PORT);
            Scanner networkInput =
                new Scanner(socket.getInputStream());
            PrintWriter networkOutput =
                new PrintWriter(
                    socket.getOutputStream(), true);
            Scanner userEntry = new Scanner(System.in);
            String message, response;
            do {
                System.out.print(
                    "Enter message ('QUIT' to exit): ");
                message = userEntry.nextLine();

                networkOutput.println(message);
                response = networkInput.nextLine();
                System.out.println(
                    "\nSERVER> " + response);
            } while (!message.equals("QUIT"));
        } catch (IOException ioEx) {
            ioEx.printStackTrace();
        } finally {
            try {
                System.out.println( "\nClosing connection...");
                socket.close();
            } catch (IOException ioEx) {
                System.out.println("Unable to disconnect!");
                System.exit(1);
            }
        }
    }
}
```

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

```
public class MultiEchoClient {
    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) {
    try {
        host = InetAddress.getLocalHost();
    } catch (UnknownHostException uhEx) {
        System.out.println("\nHost ID not found!\n");
        System.exit(1);
    }
    sendMessages();
}
```

- Create a static method *sendMessages*, setup *try – catch – finally* block
- 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() {
    Socket socket = null;
    try {
        socket = new Socket(host, PORT);
        Scanner networkInput =
            new Scanner(socket.getInputStream());
        PrintWriter networkOutput = new PrintWriter(
            socket.getOutputStream(), true);

    } catch (IOException ioEx) {
        ioEx.printStackTrace();
    } finally {
    }
}
```

- Inside *try*, 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 userEntry = new Scanner(System.in);
String message, response;
do {

} while (!message.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.print("Enter message ('QUIT' to exit): ");
    message = userEntry.nextLine();

    networkOutput.println(message);
    response = networkInput.nextLine();

    System.out.println("\nSERVER> " + response);
} while (!message.equals("QUIT"));
```

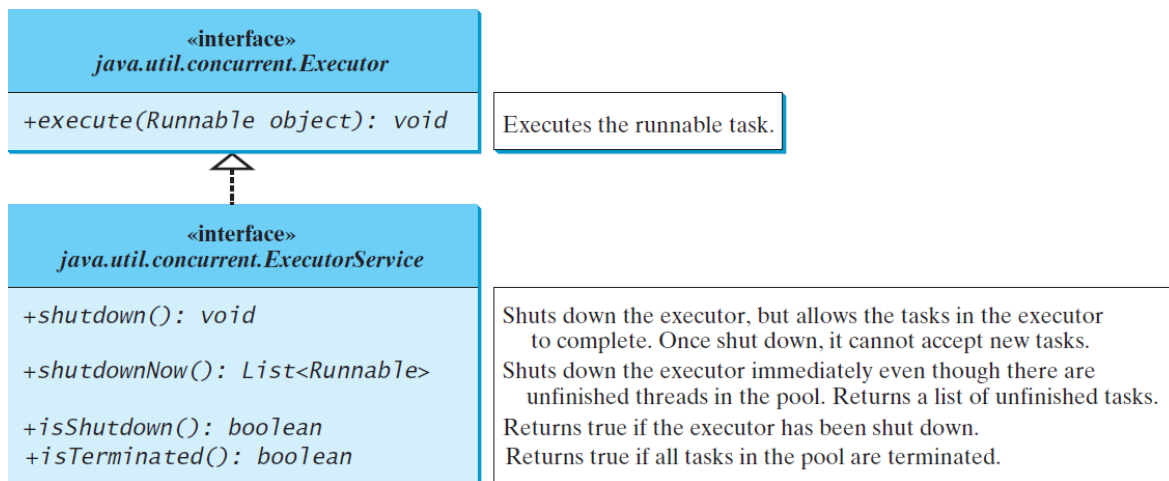
- Catch the exception.
- In the *finally* block, close the connection.

```
catch (IOException ioEx) {
    ioEx.printStackTrace();
} finally {
    try {
        System.out.println("\nClosing connection...");
        socket.close();
    } catch (IOException ioEx) {
        System.out.println("Unable to disconnect!");
        System.exit(1);
    }
}
```

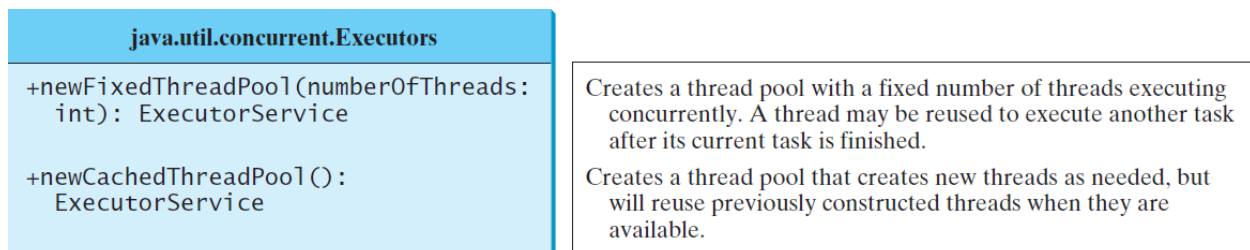
- Now, run the server and then run multiple instances of the client.

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*.



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



Multithreaded Server

```
public class MultiThreadExecutorServer {
    public static void main(String[] args) throws Exception {
        ServerSocket server = new ServerSocket(1234);
        ExecutorService pool = Executors.newFixedThreadPool(3);
        System.out.println("waiting for clients.....");
        while (true) {
            Socket link = server.accept();
            Thread t1 = new Thread(new Client_Handler(link));
            pool.execute(t1);
        }
    }

    class Client_Handler implements Runnable {
        Socket myClient = null;

        Client_Handler(Socket link) {
            this.myClient = link;
        }

        public void run() {
            try {
                System.out.println("Client " +
myClient.getRemoteSocketAddress().toString() + " has been connected");
                DataInputStream input = new
DataInputStream(myClient.getInputStream());
                DataOutputStream output = new
DataOutputStream(myClient.getOutputStream());

                String NString = input.readUTF();
                int N = Integer.parseInt(NString);
                int sum = 0;
                for (int i = 1; i <= N; i++) {
                    sum += i;
                }
                TimeUnit.SECONDS.sleep(10);
                output.writeUTF(String.valueOf(sum));
                System.out.println("The client " +
myClient.getRemoteSocketAddress().toString() + " has finished");
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
```

Client

```
public class MultiThreadExecutorClient {
    public static void main(String[] args) throws Exception {
        Socket c = new Socket("localhost", 1234);
        DataInputStream input = new DataInputStream(c.getInputStream());
        DataOutputStream output = new DataOutputStream(c.getOutputStream());
        BufferedReader userInput = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("Enter your Range: ");
        String number = userInput.readLine();

        output.writeUTF(number);

        String result = input.readUTF();
        System.out.println("The sum of 1 to " + number + " = " + result);
    }
}
```