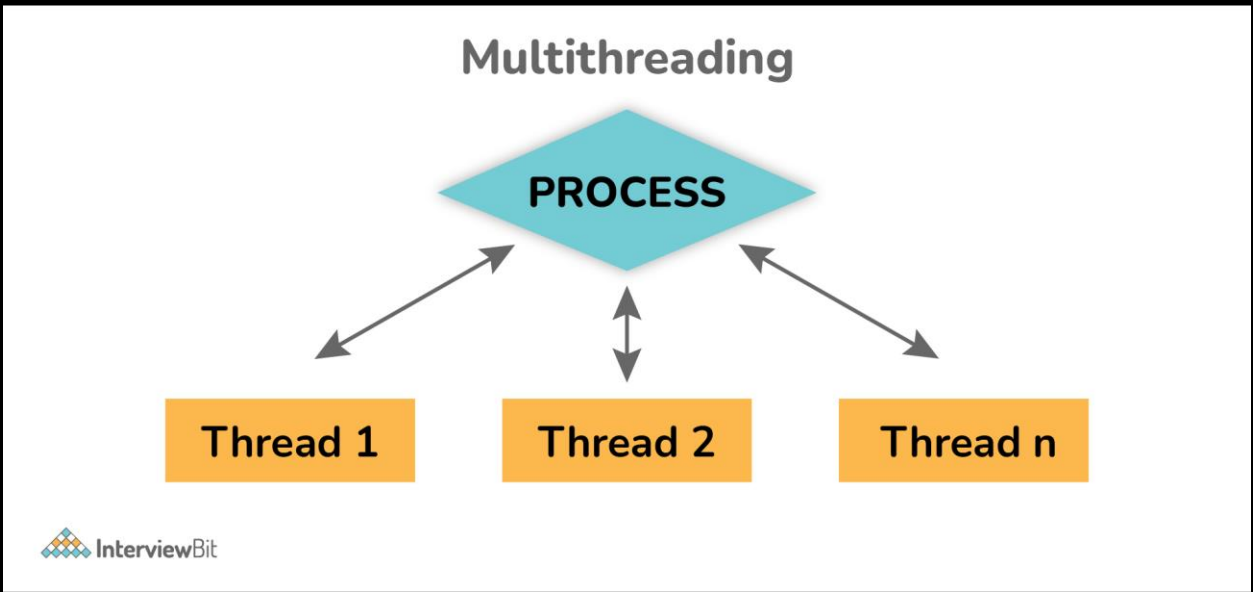
Multithreading and Multiplexing

Contents

Thread Basics

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

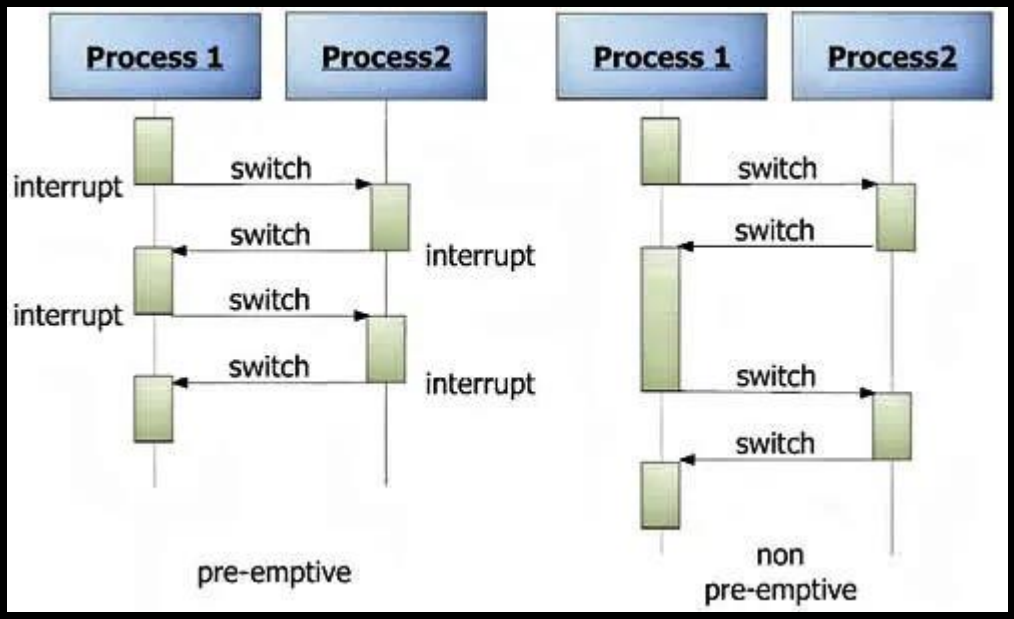


- Unlike a process, a thread does not have a separate allocation of memory but shares memory with other threads created by the same application.
    - This means that servers using threads do not exhaust their supply of available memory, as they were prone to do when creating many separate processes.
    - In addition, the threads created by an application can share global variables
- The operating system has the role do 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 scheduling is **pre-emptive** or **cooperative**.

Interviews Q: Why use multithreaded servers

✓ What is the difference between parallelization, multitasking and multithreading?

- ✓ A pre-emptive scheduler will determine when a thread has had its fair share of CPU time (possibly via simple time allocation) and will then pause it (temporarily).
- ✓ A cooperative scheduler (non-preemptive) will wait for the running thread to pause itself before giving control of the CPU to another thread.
    - A JVM using cooperative scheduling is thus much more susceptible to thread **starvation**.



Extending the Thread Class

- We can apply multithreading whether by creating thread objects and pass a class implements the $Runnable$ interface or creating a class that extends $Thread$ class.
- The $Thread$ class has seven constructors, the four most common are:
    - $Thread()$: Allocates a new $Thread$ object.
    - $Thread(Runnable\ target)$: Allocates a new $Thread$ object given an instance of $Runnable$ object.
    - $Thread(String\ name)$: Allocates a new $Thread$ object, given its name as a $String$.
    - $Thread(Runnable\ target, String\ name)$: Allocates a new Thread object, given a $Runnable$ object and a name.

- Example 1: write a multithreaded application to get the name of the task being executed.

```
public class ThreadShowName extends Thread {
    public static void main(String[] args) {
        ThreadShowName th1 = new ThreadShowName();
        ThreadShowName th2 = new ThreadShowName();
        th1.start();
        th2.start();
    }
    @Override
    public void run() {
        int pause;
        for (int i = 0; i < 10 ; i++) {
            try{
                System.out.println(getName()+
                                   " being executed");
                pause = (int) (Math.random()*3000);
                sleep(pause);
            }catch (InterruptedException ex){
                ex.printStackTrace();
            }
        }
    }
}
```

- Example 2: create two threads, but we have one thread display the message 'Hello' five times and the other thread output integers 0-4.

```
public class ThreadHelloCount {
    public static void main(String[] args) {
        HelloThread hello = new HelloThread();
        CountThread count = new CountThread();
        hello.start();
        count.start();
    }
}

class HelloThread extends Thread{
    @Override
    public void run() {
        int pause;
        for (int i = 0; i < 5; i++) {
            try{
                System.out.println("Hello!");
                pause = (int)(Math.random()*3000);
                sleep(pause);
            }catch (InterruptedException ex){
                ex.getMessage();
            }
        }
    }
}
class CountThread extends Thread{
    @Override
    public void run() {
        int pause;
        for (int i = 0; i < 5; i++) {
            try{
                System.out.println(i);
                pause = (int)(Math.random()*3000);
                sleep(pause);
            }catch (InterruptedException ex){
                ex.getMessage();
            }}}}
```

✓ The previous two programs have something incorrect? Can you guess what is it? How to fix it? Why it is incorrect?

Task

- Create the same application above but by extending the *Runnable* interface and print the output to a text area (GUI).

Multithreaded Servers

- In this section, we will create a multithreaded server that can handle multiple clients simultaneously.

Running multiple clients for a serial (one-threaded) server

- Before creating the multithreaded server, let's experiment the normal $TCPEchoServer$ and $TCPEchoClient$ by running multiple clients.
- Steps:
    1. Configure the IDE to run multiple instances of the same class (if supported).
        - If it is not possible, create multiple classes with different name but with the same code.
    2. Run the server.
    3. Run an instance of the client.
    4. Run another instance of the client.
    5. From the first client, send messages to the server.
    6. From the second client, send messages to the server.
    7. What happen? Can you explain such behavior?
- Now, let's create the multithreaded server

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.
- Inside class, override $run()$ method.

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

    public ClientHandler(Socket socket) {

    }

    public void run() {
    }
}
```

- 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), it 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"));
```

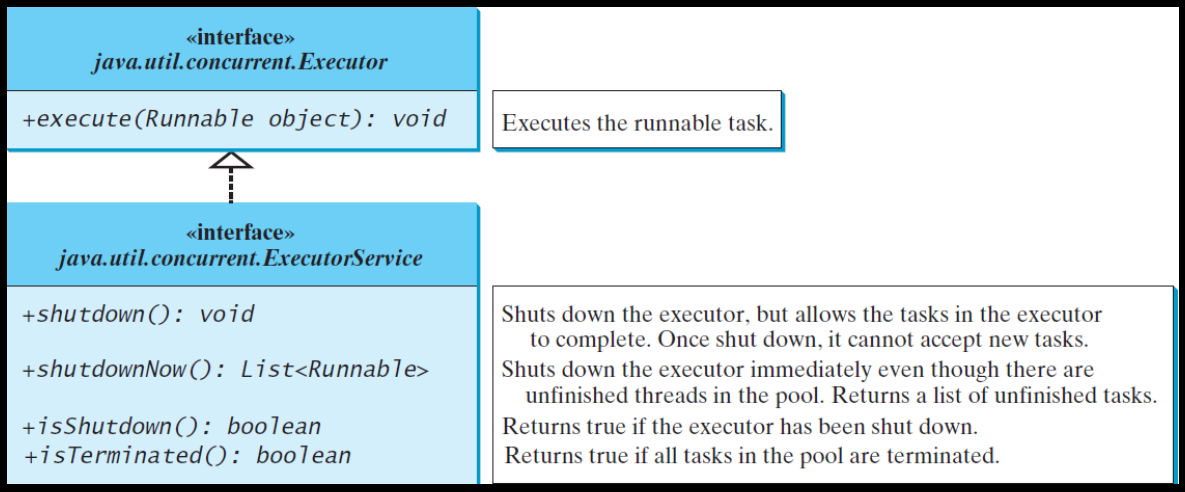
- Catch the exception.
- In the $finally$ block, close the connection.

```
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);
    }
}
```

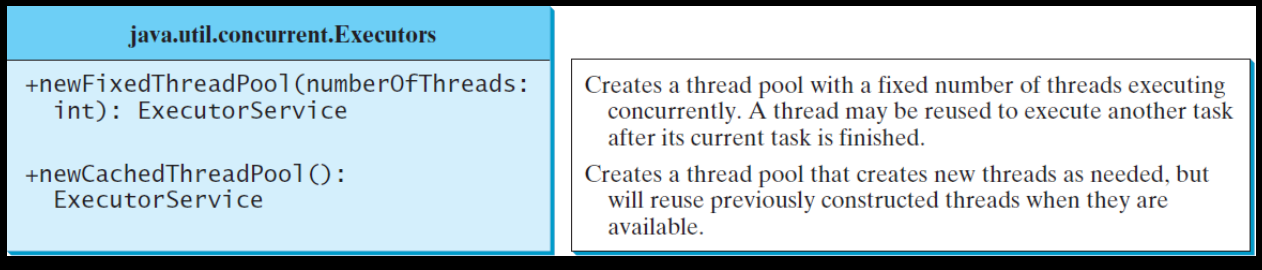
- Now, run the server and then run multiple instances of the client.

Thread Pool

- In the previous discussions, 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.



- Example 5: use *ExecutorService*.

```
public class ExecutorDemo {
    public static void main(String[] args) {
        ExecutorService executorService =
Executors.newFixedThreadPool(3);
        executorService.execute(new PrintNum(100));
        executorService.execute(new PrintChar('a', 100));
        executorService.execute(new PrintChar('b', 100));

        executorService.shutdown();
    }
}

class PrintChar implements Runnable {
    private char ch;
    private int times;

    public PrintChar(char ch, int times) {
        this.ch = ch;
        this.times = times;
    }

    @Override
    public void run() {
        for (int i = 0; i < this.times; i++) {
            System.out.println(ch);
        }
    }
}

class PrintNum implements Runnable {
    private int lastNum;

    public PrintNum(int lastNum) {
        this.lastNum = lastNum;
    }

    @Override
    public void run() {
        for (int i = 0; i < lastNum; i++) {
            System.out.println(" " + i);
        }
    }
}
```

- Example 6: a server application that computes the sum of integers from 1 to N, and sends the result to the client.

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);
    }
}
```

Task

- Create a flow chart (sequence diagram) for the server and the client to show the execution and communication sequence.