Java Multithreading Basics

Kindly explain below points with respect to multithreading in Java:Creating Threads
 - Extending the Thread Class
 - Implementing the Runnable Interface
 - Thread Lifecycle

Multithreading in Java: Detailed Explanation

Java provides multithreading capabilities to execute multiple threads simultaneously, enabling efficient use of CPU and better application performance. Below is a detailed explanation of the topics:

---

1. Creating Threads

Threads in Java can be created in two main ways:

a. Extending the `Thread` Class

- In this approach, a class extends the `Thread` class and overrides its `run()` method.
- The `run()` method contains the code that the thread will execute.
- To start the thread, the `start()` method is called, which internally invokes the `run()` method.

Example:

```java
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Thread is running...");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Starts the thread
    }
}
```

Key Points:

- Each thread gets its own call stack.
- You cannot extend any other class if you extend `` `Thread` `` (single inheritance limitation).

---

b. Implementing the `` `Runnable` `` Interface

- In this approach, a class implements the `` `Runnable` `` interface and overrides its `` `run()` `` method.
- The `` `Runnable` `` object is passed to a `` `Thread` `` object, and the `` `start()` `` method is used to execute the thread.

Example:

```java
class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread is running using Runnable...");
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable runnable = new MyRunnable();
        Thread thread = new Thread(runnable);
        thread.start(); // Starts the thread
    }
}
```

Key Points:

- This approach is preferred as it allows the class to extend another class.
- It promotes better separation of thread logic from thread management.

---

2. Thread Lifecycle

A thread in Java passes through several states during its lifecycle. The `` `Thread` `` class provides constants for these states, and they can be understood as follows:

a. New (Created):

- A thread is in the "new" state when it is created but not yet started.
- Example:

  ```java
  Thread t = new Thread(); // New state
  ```

b. Runnable:

- The thread is ready to run but waiting for CPU time.
- This happens after calling the `` `start()` `` method.
- Example:

  ```java
  t.start(); // Moves to Runnable state
  ```

c. Running:

- The thread is in execution.

- The CPU assigns time to the thread to execute its `` `run()` `` method.

d. Blocked/Waiting:

- The thread is waiting for a resource or a signal to proceed.
- This can occur due to methods like `` `sleep()` ``, `` `wait()` ``, or resource contention.

e. Terminated:

- A thread moves to the "terminated" state after completing its task or due to an unhandled exception.
- Once terminated, a thread cannot be restarted.

Thread Lifecycle Diagram:

```
rust


  New -> Runnable -> Running -> Waiting/Blocked -> Terminated
```

Example Demonstrating Lifecycle:

```
java



  class MyThread extends Thread {
      @Override
      public void run() {
          try {
              System.out.println("Thread is running...");
              Thread.sleep(1000); // Moves to Waiting/Blocked state
              System.out.println("Thread is terminated.");
          } catch (InterruptedException e) {
              e.printStackTrace();
          }
      }
  }

  public class Main {
      public static void main(String[] args) {
          MyThread thread = new MyThread();
          System.out.println("Thread created."); // New state
```

```
            thread.start(); // Moves to Runnable state
        }
    }
```

Comparison: `Thread` vs `Runnable`

Aspect	Thread	Runnable
Inheritance Limitation	Can’t extend other classes	Can extend any other class
Code Organization	Mixing thread logic and thread management	Better separation of concerns
Flexibility	Less flexible	More flexible and reusable

This explanation covers how to create threads, the lifecycle stages of threads, and their respective examples for clarity. Let me know if you'd like further details or examples!

Explain the process creation with respect to Java8

In **Java 8**, process creation is managed using the **`java.lang.Process`** and **`java.lang.ProcessBuilder`** classes. These classes allow you to execute external programs, scripts, or commands from within a Java application. Java 8 improved the process creation capabilities introduced in earlier versions by adding features like **`ProcessBuilder`**, which offers better control over process creation and management.

Steps for Process Creation

1. **Creating a Process Using `Runtime.exec()`**
   The **`Runtime`** class provides the **`exec()`** method to create and execute a new process. However, this approach is less preferred in Java 8 due to limited configuration options.

   Example:

```java
import java.io.*;

public class ProcessExample {
    public static void main(String[] args) throws IOException {
        Process process = Runtime.getRuntime().exec("notepad.exe");
        System.out.println("Process started.");
    }
}
```

Key Points:

- The `exec()` method launches an external process.
- Input/output redirection is harder to configure compared to `ProcessBuilder`.

---

2. **Creating a Process Using `ProcessBuilder`**
   Java 8's `ProcessBuilder` is the preferred way to create and manage processes because it provides more flexibility, such as redirection of input/output streams and setting environment variables.

   Example:

```java
import java.io.*;

public class ProcessBuilderExample {
    public static void main(String[] args) {
        try {
            // Creating a ProcessBuilder instance
            ProcessBuilder pb = new ProcessBuilder("ping", "google.com");

            // Redirecting the output to a file
            pb.redirectOutput(new File("output.txt"));

            // Starting the process
            Process process = pb.start();

            // Waiting for the process to complete
            int exitCode = process.waitFor();
            System.out.println("Process exited with code: " + exitCode);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Features of `ProcessBuilder`:

- Allows setting environment variables using `pb.environment()`.
- Enables redirection of input/output streams using `redirectInput()`, `redirectOutput()`, and `redirectError()`.
- Accepts a list of commands instead of a single string.

Reading Process Output

To capture the output of the process (stdout and stderr), you can use the process's input stream.

Example:

```java
import java.io.*;

public class ProcessOutputExample {
    public static void main(String[] args) {
        try {
            ProcessBuilder pb = new ProcessBuilder("ping", "-c", "4", "google.com");
            Process process = pb.start();

            // Reading the standard output
            BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }

            // Wait for the process to complete
            int exitCode = process.waitFor();
            System.out.println("Exit code: " + exitCode);
        } catch (Exception e) {
            e.printStackTrace();
        }
        }
    }
```

Redirecting Streams

Java 8 allows you to redirect streams for better control.

Redirecting Output to a File:

```java
ProcessBuilder pb = new ProcessBuilder("ping", "google.com");
pb.redirectOutput(new File("output.txt")); // Redirect stdout to a file
pb.start();
```

Redirecting Input from a File:

```java
ProcessBuilder pb = new ProcessBuilder("sort");
pb.redirectInput(new File("unsorted.txt")); // Provide input from a file
pb.redirectOutput(new File("sorted.txt"));  // Redirect output to a file
pb.start();
```

Advantages of `ProcessBuilder` Over `Runtime.exec()`

Feature	`Runtime.exec()`	`ProcessBuilder`
Input/Output Redirection	Limited	Comprehensive
Environment Variable Control	Limited	Flexible
Command Argument Handling	Single String	List of Arguments
Code Readability	Less intuitive	Clear and modular

Handling Process Termination

The `waitFor()` method can be used to block the current thread until the process finishes. You can also forcibly destroy a process using the `destroy()` method.

Example:

```java
Process process = new ProcessBuilder("notepad.exe").start();
Thread.sleep(5000); // Let the process run for 5 seconds
process.destroy(); // Terminate the process
System.out.println("Process terminated.");
```

Key Enhancements in Java 8

- **Stream Handling:** Java 8 introduced the ability to use `try-with-resources` to manage process input/output streams efficiently.
- **ProcessBuilder API Improvements:** Enhanced support for flexible process creation and management.
- **Environment Variables:** Easy modification and inspection of the process environment.

These features make Java 8's process creation robust, flexible, and more developer-friendly compared to earlier versions.

How do we create threads through Lambda expression

In Java 8, **lambda expressions** introduced a concise way to create threads without the need for writing verbose code. Instead of implementing the `Runnable` interface explicitly or extending the `Thread` class, you can use a lambda expression to define the `run()` method inline.

Steps to Create Threads Using Lambda Expressions

1. Using Lambda with `Thread` Constructor

The `Thread` class accepts a `Runnable` as an argument. Since `Runnable` is a functional interface (it has only one abstract method `run()`), we can pass a lambda expression instead of implementing the interface.

Example:

```java
public class LambdaThreadExample {
    public static void main(String[] args) {
        // Create a thread using a lambda expression
        Thread thread = new Thread(() -> {
            System.out.println("Thread is running using lambda expression.");
        });

        // Start the thread
        thread.start();
    }
}
```

---

2. Multiple Threads with Lambda

You can create multiple threads easily by using lambdas and defining the behavior for each thread.

Example:

```java
public class MultiThreadExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> System.out.println("Thread 1 is running"));
        Thread thread2 = new Thread(() -> System.out.println("Thread 2 is running"));

        thread1.start();
        thread2.start();
    }
}
```

---

3. Lambda with `ExecutorService`

In real-world applications, you might use an `ExecutorService` to manage threads. You can use lambda expressions for defining tasks (`Runnable`).

Example:

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

public class ExecutorLambdaExample {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        // Submit tasks using lambda expressions
        executorService.submit(() -> System.out.println("Task 1 executed"));
        executorService.submit(() -> System.out.println("Task 2 executed"));

        // Shut down the executor
        executorService.shutdown();
    }
}
```

Benefits of Using Lambdas for Thread Creation

1. **Conciseness:** Lambdas reduce the boilerplate code required to implement the `Runnable` interface or extend the `Thread` class.
2. **Readability:** The behavior of the thread is clearly visible inline.
3. **Flexibility:** You can easily define one-time-use thread logic without creating separate classes.

Example: Computing a Task in a Thread Using Lambda

Here’s an example where a lambda is used to perform a calculation in a separate thread:

```java
public class CalculationThread {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            int sum = 0;
            for (int i = 1; i <= 10; i++) {
                sum += i;
            }
            System.out.println("Sum of numbers from 1 to 10 is: " + sum);
        });

        thread.start();
    }
}
```

Comparison: Traditional Runnable vs Lambda

Aspect	Traditional Runnable	Lambda Expression
Boilerplate Code	Requires a separate class or an anonymous class	Defined inline with minimal syntax
Readability	Relatively verbose	Compact and concise
Functional Interface Use	Indirectly used	Explicit and natural use

---

By leveraging lambda expressions, thread creation becomes simpler and more elegant in Java 8, making it easier to focus on the task logic rather than the surrounding boilerplate code.