JAVA THREADS

Definition

A Java thread is the smallest unit of execution within a program. It is a lightweight subprocess that runs independently but shares the same memory space of the process, allowing multiple tasks to execute concurrently.

For example: In MS Word, one thread formats the document while another takes user input. Multithreading also keeps applications responsive, since other threads can continue running even if one gets stuck.

Create Threads in Java

We can create threads in java using two ways

1. Extending Thread Class
2. Implementing a Runnable interface

1. By Extending Thread Class

Create a class that extends Thread. Override the `run()` method, this is where you put the code that the thread should execute. Then create an object of your class and call the `start()` method. This will internally call `run()` in a new thread.

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

        ThreadExample t1 = new ThreadExample("one");
        ThreadExample t2 = new ThreadExample("two");
        t1.start();
        t2.start();
    }
}

class ThreadExample extends Thread {
    String name;
    ThreadExample(String name) {
        this.name = name;
    }
    public void run() {
        System.out.println("Thread " + name + " is running");
    }
}
```

2. Using Runnable Interface

Create a class that implements Runnable. Override the `run()` method, this contains the code for the thread. Then create a Thread object, pass your Runnable object to it and call `start()`.

```
public class Main {
    public static void main(String[] args) {
        ThreadExample te1 = new ThreadExample("one");
        ThreadExample te2 = new ThreadExample("two");
        Thread t1 = new Thread(te1);
        Thread t2 = new Thread(te2);

        t1.start();
        t2.start();
    }
}

class ThreadExample implements Runnable {
    String name;
    ThreadExample(String name) {
        this.name = name;
    }
    public void run() {
        System.out.println("Thread " + name + " is running");
    }
}
```

Note: We use `start()` to launch a new thread, which then calls the `run()` method in parallel. If we call `run()` directly, it works like a normal method call and no new thread is created.

Java Thread Class

The Thread class (in `java.lang package`) is used to create and control threads in Java. Each object of this class represents a single thread of execution.

Constructors of Thread Class

Constructor	Action Performed
`Thread()`	Allocates a new Thread object.
`Thread(Runnable target)`	Allocates a new Thread object.
`Thread(Runnable target, String name)`	Allocates a new Thread object.
`Thread(String name)`	Allocates a new Thread object.

Methods of Thread Class

Method	Action Performed
activeCount()	Returns estimate of active threads in current thread group & subgroups.
currentThread()	Returns reference of currently executing thread.
getId()	Returns unique identifier of thread.
getName()	Returns thread's name.
getPriority()	Returns thread's priority.
getState()	Returns state of this thread (NEW, RUNNABLE, etc.).
interrupt()	Interrupts this thread.
interrupted()	Tests if current thread is interrupted (clears status).
isAlive()	Tests if thread is alive (started & not dead).
isInterrupted()	Tests if thread is interrupted (without clearing status).
join()	Waits for this thread to die.
join(long millis)	Waits max millis for this thread to die.
run()	Executes run() method (if Runnable given).
setName(String name)	Changes thread's name.
setPriority(int newPriority)	Changes thread's priority.
sleep(long millis)	Pauses current thread for given milliseconds.
start()	Starts thread (JVM calls run()).
toString()	Returns string with thread name, priority & group.
yield()	Hints scheduler to pause current thread & give chance to others.
wait()	Release the lock and go to the waiting queue until notified
notify()	Wake one waiting thread
notifyAll()	Wake all waiting threads

Example

```
import static java.lang.Thread.*;

public class Main {
    public static void main(String[] args) throws InterruptedException {
        ThreadExample te1 = new ThreadExample("One");
        ThreadExample te2 = new ThreadExample("Two");
        Thread t1 = new Thread(te1, "One");
        Thread t2 = new Thread(te2, "Two");

        t1.start();
        t2.start();
```

```
        // Check if the threads are alive
        System.out.println(t1.getName() + " alive? " + t1.isAlive());
        System.out.println(t2.getName() + " alive? " + t2.isAlive());

        // Make thread One to yeild
        t1.yield();

        // Make thread One to sleep for 2 seconds
        t1.sleep(2000);


        t1.join();
        t2.join();

        // Print the number of active threads
        System.out.println("Current active count " + Thread.activeCount());
        // Print the name of the current thread
        System.out.println("Current thread running  " +
Thread.currentThread());
        // Print the ID and name of the current thread
        System.out.println("ID of the current thread - " +
Thread.currentThread().getId());
        // Print the name of the current thread
        System.out.println("Name of the current thread - " +
Thread.currentThread().getName());

    }
}

class ThreadExample implements Runnable {
    String name;
    ThreadExample(String name) {
        this.name = name;
    }
    public void run() {
        System.out.println("Thread " + name + " is running");
    }
}
```

Concurrency Problems

- **Race Condition**: Occurs when multiple threads access shared data simultaneously, leading to inconsistent results. It happens when the code is not thread-safe.
- **Deadlock**: Happens when two or more threads are blocked forever, each waiting for the other to release a lock.
- **Livelock**: Threads are active but unable to make progress because they keep responding to each other in an endless loop.
- **Thread Starvation**: A thread is perpetually denied access to resources because other threads are given priority.

- **Priority Inversion**: Occurs when a low-priority thread holds a lock needed by a high-priority thread, blocking its progress.

Race Condition

```
class UnsafeCounter {
    private int value = 0;

    // Not synchronized. Read-modify-write is racy.
    void increment() {
        int current = value;    // read
        Thread.yield();         // make the race more likely
        value = current + 1;
        System.out.println("Value: " + value);// write
    }

    int get() {
        return value; }
}

// Simple thread class that bumps the counter many times.
class MyThread extends Thread {
    private final UnsafeCounter counter;
    private final int loops;

    MyThread(String name, UnsafeCounter counter, int loops) {
        super(name);
        this.counter = counter;
        this.loops = loops;
    }

    @Override
    public void run() {
        for (int i = 0; i < loops; i++) {
            counter.increment(); // NOT thread-safe
        }
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        UnsafeCounter counter = new UnsafeCounter();
        int loops = 100; // try raising this if the race doesn't show

        // Three threads explicitly:
        Thread t1 = new MyThread("t1", counter, loops);
        Thread t2 = new MyThread("t2", counter, loops);
        Thread t3 = new MyThread("t3", counter, loops);

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();
```

```
        int expected = 3 * loops;
        int actual = counter.get();
        System.out.println("expected=" + expected +
                ", actual=" + actual +
                ", lost=" + (expected - actual));
    }
}
```

Race Condition Safe Code

```
class UnsafeCounter {
    private int value = 0;

    // Not synchronized. Read-modify-write is racy.
    synchronized void increment() {
        int current = value;    // read
        Thread.yield();         // make the race more likely
        value = current + 1;
        System.out.println("Value: " + value);// write
    }

    int get() {
        return value; }
}

// Simple thread class that bumps the counter many times.
class MyThread extends Thread {
    private final UnsafeCounter counter;
    private final int loops;

    MyThread(String name, UnsafeCounter counter, int loops) {
        super(name);
        this.counter = counter;
        this.loops = loops;
    }

    @Override
    public void run() {
        for (int i = 0; i < loops; i++) {
            counter.increment(); // NOT thread-safe
        }
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        UnsafeCounter counter = new UnsafeCounter();
        int loops = 100; // try raising this if the race doesn't show

        // Three threads explicitly:
        Thread t1 = new MyThread("t1", counter, loops);
        Thread t2 = new MyThread("t2", counter, loops);
        Thread t3 = new MyThread("t3", counter, loops);
```

```
        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();

        int expected = 3 * loops;
        int actual = counter.get();
        System.out.println("expected=" + expected +
                ", actual=" + actual +
                ", lost=" + (expected - actual));
    }
}
```