Unit 7: Threads [3 Hrs.]

Threads in Java allow concurrent execution of tasks, enabling efficient utilization of CPU resources. This unit covers the creation, instantiation, and starting of threads, as well as advanced concepts like thread priorities, synchronization, inter-thread communication, and deadlock.

---

1. Create/Instantiate/Start New Threads

There are two ways to create and start threads in Java:

a) Extending `java.lang.Thread`

- Create a class that extends the `Thread` class.
- Override the `run()` method to define the task the thread will perform.
- Create an instance of the class and call the `start()` method to begin execution.

Lab 1: Creating, instantiating and starting new threads extending `java.lang.Thread` class

```
class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - Count: " + i);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.err.println("Thread interrupted: " + e.getMessage());
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread1 = new MyThread();
        MyThread thread2 = new MyThread();

        thread1.start(); // Starts the first thread -> calls parent(Thread) method
        thread2.start(); // Starts the second thread
        //thread1.interrupt();  -> to simulate interrupt
    }
}
```

Explanation:

- The `MyThread` class extends `Thread` and overrides the `run()` method.
- Two threads (`thread1` and `thread2`) are created and started using the `start()` method.
- The `run()` method is executed concurrently by both threads.

Sample Output:

```
Thread-0 - Count: 1
Thread-1 - Count: 1
Thread-0 - Count: 2
Thread-1 - Count: 2
Thread-0 - Count: 3
Thread-1 - Count: 3
Thread-0 - Count: 4
Thread-1 - Count: 4
Thread-0 - Count: 5
Thread-1 - Count: 5
```

---

b) Implementing `java.lang.Runnable` Interface

- Create a class that implements the `Runnable` interface.
- Override the `run()` method to define the task.
- Pass an instance of the class to a `Thread` object and call `start()`.

Lab 2: Creating, instantiating and starting new threads implementing `java.lang.Runnable` interface

```java
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - Count: " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread1 = new Thread(myRunnable);
        Thread thread2 = new Thread(myRunnable);

        thread1.start(); // Starts the first thread
        thread2.start(); // Starts the second thread
    }
}
```

Explanation:

- The `MyRunnable` class implements the `Runnable` interface and overrides the `run()` method.
- Two threads (`thread1` and `thread2`) are created using the `Thread` constructor and started using the `start()` method.
- The `run()` method is executed concurrently by both threads.

Sample Output:

```
Thread-0 - Count: 1
Thread-1 - Count: 1
Thread-0 - Count: 2
Thread-1 - Count: 2
```

```
Thread-0 - Count: 3
Thread-1 - Count: 3
Thread-0 - Count: 4
Thread-1 - Count: 4
Thread-0 - Count: 5
Thread-1 - Count: 5
```

Difference Between Extending `Thread` and Implementing `Runnable`

Feature	Extending `Thread`	Implementing `Runnable`
Inheritance	Uses up the single inheritance option.	Does not use inheritance; allows extending other classes.
Reusability	Less reusable.	More reusable; same `Runnable` instance can be passed to multiple threads.
Thread Creation	Directly creates a `Thread` object.	Requires a `Thread` object to wrap the `Runnable`.
Best Use Case	When we need to override `Thread` methods.	When we want to separate thread logic from task logic.

---

2. Thread Execution

- When a thread is started using the `start()` method, the JVM calls the `run()` method.
- Threads run concurrently, meaning multiple threads can execute simultaneously (depending on the number of CPU cores).

Lab 3: Thread Execution

```
class Task implements Runnable {
    @Override
    public void run() {
        // This code will be executed when the thread starts
        // It prints the name of the currently executing thread
        System.out.println(Thread.currentThread().getName() + " is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        Task task = new Task();

        Thread thread1 = new Thread(task, "Thread-1");// The second argument in the
Thread constructor sets the thread's name
        Thread thread2 = new Thread(task, "Thread-2");

        // Start the first thread
        thread1.start();

        // Start the second thread
```

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

Sample Output:

```
Thread-1 is running.
Thread-2 is running.
```

---

3. Thread Priorities

- Thread priorities range from `1` (lowest) to `10` (highest).
- The default priority is `5`.
- Use `setPriority(int priority)` to set a thread's priority and `getPriority()` to retrieve it.

Lab 4: Thread Priorities

```
class Task implements Runnable {
    @Override
    public void run() {
        // It prints the name of the currently executing thread and its priority
        System.out.println(Thread.currentThread().getName() + " - Priority: " +
Thread.currentThread().getPriority());
    }
}

public class Main {
    public static void main(String[] args) {
        Task task = new Task();

        Thread thread1 = new Thread(task, "Thread-1");// The second argument in the
Thread constructor sets the thread's name
        Thread thread2 = new Thread(task, "Thread-2");

        // Set the priority of thread1 to the minimum priority (1)
        // Lower priority threads are given less preference by the thread scheduler
        thread1.setPriority(Thread.MIN_PRIORITY); // Priority 1

        // Set the priority of thread2 to the maximum priority (10)
        // Higher priority threads are given more preference by the thread scheduler
        thread2.setPriority(Thread.MAX_PRIORITY); // Priority 10

        // Start the first thread
        thread1.start();

        // Start the second thread
        thread2.start();
    }
}
```

Sample Output:

```
Thread-2 - Priority: 10
Thread-1 - Priority: 1
```

---

4. Synchronization

- Synchronization ensures that only one thread can access a shared resource at a time.
- Use the `synchronized` keyword to create synchronized methods or blocks.

Lab 5: Synchronization

```
class Counter {
    private int count = 0; // Instance variable to store the count (shared between threads)

    // Synchronized method to increment the count safely->Ensures only one thread can execute this method at a time
    public synchronized void increment() {
        count++; // Increment the shared 'count' variable -> single object of Counter accessing this method using two threads(shared resources-count variable)
    }

    // Method to retrieve the current value of 'count'
    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) {
        // Create a single instance of the Counter class->This instance will be shared between multiple threads
        Counter counter = new Counter();

        // Does not guarantee which thread will execute immediately after t1 -> the order of execution between threads is non-deterministic unless explicitly controlled
        Thread t1 = new Thread(() -> {    //Thread t1: Increments the counter 1000 times->equivalent to implementing the Runnable interface and its run() method
            for (int i = 0; i < 1000; i++) {
                counter.increment(); // Safely increment the shared 'count' variable
            }
        });

        // Thread t2: Also increments the counter 1000 times
        Thread t2 = new Thread(() -> {    equivalent to implementing the Runnable interface and its run() method
            for (int i = 0; i < 1000; i++) {
                counter.increment(); // Safely increment the shared 'count' variable
            }
        });

        // Start both threads->Each thread will execute its task concurrently
```

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

        try {
            // The join() method ensures the main thread waits for t1 and t2 to
complete
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            // Handle any interruption during thread execution
            e.printStackTrace();
        }

        // Print the final value of 'count'->Since the increment() method is
synchronized, the final value will always be 2000
        System.out.println("Count: " + counter.getCount());
    }
}
```

Sample Output:

```
Count: 2000
```

Sample Output without synchronized, the final value of count could be less than 2000 due to overlapping operations. The output might look like

```
Count: 1997
```

Sample Output when the `join()` method is not used

```
Count: 1250
//The value printed will always be less than or equal to 2000.
```

Using a Lambda Expression

- The lambda expression `() -> { ... }` defines the task that the thread will execute.
- It is equivalent to implementing the `Runnable` interface and its `run()` method->The lambda expression `() -> { ... }` creates an instance of `Runnable`.
- Instead of explicitly creating a class that implements `Runnable`, the lambda offers a concise way to define the task directly.

The `join()` method ensures that the main thread waits for `t1` and `t2` to complete their execution before proceeding.

- When `t1.join()` is called, the main thread(psvm) pauses until `t1` finishes its execution.
- Similarly, when `t2.join()` is called, the main thread waits until `t2` completes its execution.
- Only after both threads have finished does the main thread proceed to print the value of `count`.
- This ensures that all increments are completed before the final value of `count` is printed.

---

5. Inter-Thread Communication

- Threads can communicate using `wait()`, `notify()`, and `notifyAll()`.
- These methods are used in synchronized contexts to coordinate thread execution.

Lab 6: Inter-Thread Communication

```
class NumberPrinter {
    private int currentNumber = 100; // Start from 100

    // Synchronized method to print even numbers
    public synchronized void printEvenNumbers() {
        while (currentNumber <= 200) {
            if (currentNumber % 2 == 0) {
                System.out.println("Even Thread: " + currentNumber);
                currentNumber++;
            }
            notify(); // Notify the other thread waiting on the same object's
monitor(lock)
            try {
                wait(); // Wait for the other thread->The thread enters a waiting
state until another thread calls notify() or notifyAll() on the same object, or until
it is interrupted.
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    // Synchronized method to print odd numbers
    public synchronized void printOddNumbers() {
        while (currentNumber <= 200) {
            if (currentNumber % 2 != 0) {
                System.out.println("Odd Thread: " + currentNumber);
                currentNumber++;
            }
            notify(); // Notify the other thread
            try {
                wait(); // Wait for the other thread
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public class EvenOddThreads {
    public static void main(String[] args) {
        NumberPrinter printer = new NumberPrinter();//Both threads share the same
printer object->shared resource

        // Thread to print even numbers
        Thread evenThread = new Thread(() -> printer.printEvenNumbers(),
```

```
"EvenThread");//Thread thread = new Thread(Runnable target, String name);

        // Thread to print odd numbers
        Thread oddThread = new Thread(() -> printer.printOddNumbers(), "OddThread");

        // Start both threads
        evenThread.start();
        oddThread.start();
    }
}
```

Sample Output:

```
Even Thread: 100
Odd Thread: 101
Even Thread: 102
Odd Thread: 103
Even Thread: 104
Odd Thread: 105
...
Even Thread: 200
```

---

6. Deadlock

Understanding deadlock in Java can be challenging because it involves multiple threads and resources, and the conditions for deadlock are subtle. Let me simplify it with a simple example and explanation.

What is Deadlock?

Deadlock occurs when two or more threads are blocked forever, waiting for each other to release resources. It happens when:

1. **Mutual Exclusion**: Resources are held by one thread at a time.
2. **Hold and Wait**: A thread holds a resource and waits for another.
3. **No Preemption**: Resources cannot be forcibly taken from a thread.
4. **Circular Wait**: Threads form a circular chain, each waiting for a resource held by the next.

Lab 7: Deadlock

```
public class DeadlockExample {
    public static void main(String[] args) {
        final String resource1 = "Resource 1";
        final String resource2 = "Resource 2";

        // Thread 1 tries to lock resource1 then resource2
        Thread thread1 = new Thread(() -> {
            synchronized (resource1) {
                System.out.println("Thread 1: Locked Resource 1");
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
```

```
                    e.printStackTrace();
                }
                synchronized (resource2) {
                    System.out.println("Thread 1: Locked Resource 2");
                }
            }
        });

        // Thread 2 tries to lock resource2 then resource1
        Thread thread2 = new Thread(() -> {
            synchronized (resource2) {
                System.out.println("Thread 2: Locked Resource 2");
                try {
                    Thread.sleep(100); // Simulate some work
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (resource1) {
                    System.out.println("Thread 2: Locked Resource 1");
                }
            }
        });

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

Sample Output

```
Thread 1: Locked Resource 1
Thread 2: Locked Resource 2
```

Explanation

1. **Thread 1** locks `resource1` and waits for `resource2`.
2. **Thread 2** locks `resource2` and waits for `resource1`.
3. Both threads are stuck waiting for each other, causing a **deadlock**.

How to Avoid Deadlock?

To avoid deadlock, we can:

1. **Lock Resources in the Same Order**: Ensure all threads request resources in the same sequence.
2. **Avoid Nested Locks**: Minimize locking multiple resources at once.

Simplified Fix for the Example

Lock resources in the same order:

```
// Thread 1 and Thread 2 both lock resource1 first, then resource2
Thread thread1 = new Thread(() -> {
    synchronized (resource1) {
        System.out.println("Thread 1: Locked Resource 1");
        try {
```

```
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (resource2) {
            System.out.println("Thread 1: Locked Resource 2");
        }
    }
});

Thread thread2 = new Thread(() -> {
    synchronized (resource1) {
        System.out.println("Thread 2: Locked Resource 1");
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (resource2) {
            System.out.println("Thread 2: Locked Resource 2");
        }
    }
});
```

Now, both threads lock `resource1` first, so no deadlock occurs.

Summary

- Threads can be created by extending `Thread` or implementing `Runnable`.
- Thread priorities determine the order of execution.
- Synchronization ensures thread-safe access to shared resources.
- Inter-thread communication allows threads to coordinate.
- Deadlock occurs when threads are stuck waiting for each other indefinitely.