Java Threads

Threads are a fundamental concept used for concurrent programming. They allow multiple tasks to be executed concurrently within a single program. Threads are lightweight units of execution that share the same memory space and resources of a process.

Here's an in-depth overview of Java threads:

1. **Thread Creation**: Threads in Java can be created in two main ways:
    - Extending the Thread class: You can create a new class that extends the Thread class and override the run() method to define the task the thread should perform.
    - Implementing the Runnable interface: You can create a class that implements the Runnable interface and implement the run() method. Then, you can pass an instance of this class to the Thread constructor and create a new thread.
2. **Thread States:** Threads in Java have several states:
    - New: The thread has been created but not started yet.
    - Runnable: The thread is eligible to run, and it's waiting for the CPU to execute its task.
    - Running: The thread is currently executing its task.
    - Blocked: The thread is waiting for a monitor lock to be released.
    - Waiting: The thread is waiting indefinitely for another thread to perform a specific action.
    - Timed Waiting: The thread is waiting for a specific period of time.
    - Terminated: The thread has completed its execution and is terminated.
3. **Thread Synchronization:** When multiple threads share the same resources, synchronization is necessary to ensure thread safety and prevent data corruption. Java provides several mechanisms for thread synchronization:
    - synchronized keyword: By using the synchronized keyword, you can synchronize access to methods or blocks of code, allowing only one thread to execute them at a time.
    - wait() and notify() methods: These methods are used in combination with the synchronized keyword to implement inter-thread communication and coordination. The wait() method pauses the current thread until another thread calls notify() or notifyAll() to wake it up.
    - Lock interface: The Lock interface and its implementations, such as ReentrantLock, provide more advanced locking mechanisms compared to synchronized. They offer

additional features like fairness, multiple condition variables, and non-blocking attempts to acquire a lock.

4. **Thread Interference and Race Conditions:** When multiple threads access shared data concurrently, it can lead to thread interference and race conditions. Thread interference occurs when two or more threads access shared data simultaneously, resulting in unexpected behavior. Race conditions occur when the final outcome depends on the relative execution order of threads. Proper synchronization using techniques like locks or synchronization blocks is essential to avoid these issues.
5. **Thread Communication:** Java provides various mechanisms for thread communication:
    - wait() and notify() methods: As mentioned earlier, these methods allow threads to communicate and coordinate their actions.
    - join() method: The join() method allows one thread to wait for the completion of another thread before proceeding.
    - Thread class methods: The Thread class provides methods like sleep(), yield(), and interrupt(), which can be used to control and communicate with threads.
6. **Thread Pools**: Creating and managing a large number of threads can be inefficient and resource-consuming. Java's Executor framework provides thread pools, which are a pool of pre-initialized threads that can be reused. By utilizing thread pools, you can avoid the overhead of creating new threads for every task and manage the execution of tasks efficiently.
7. **Thread Safety and Volatile:** Thread safety refers to ensuring that data access and manipulation by multiple threads are properly synchronized to avoid inconsistencies. In Java, the volatile keyword is used to mark a variable as volatile, which guarantees that changes to the variable are immediately visible to other threads. It provides a weaker form of synchronization than locks but is useful in certain scenarios.

These are some of the key aspects of Java threads. Thoroughly understanding thread creation, synchronization, communication, and thread safety is crucial for effective concurrent programming in Java.

Programs in session:

```
package threads;

public class MyThread extends Thread{

   public MyThread(String name) {
      setName(name);
   }

   @Override
```

```
    public void run() {
        //This method contains logic to execute
        // in thread
        //print numbers from 1 to 10
        for(int i=0;i<10;i++){
            System.out.println("Printing number "
                    +i+" from thread "+getName());
        }

    }
}

package threads;
public class ThreadRunner {
    public static void main(String[] args)
            throws InterruptedException {
        MyThread t1 = new MyThread("t1");
        MyThread t2 = new MyThread("t2");
        MyThread t3 = new MyThread("t3");

        System.out.println("starting thread t1 and " +
                "t2 and t3");
        //Here sequence of thread execution is random
        //This depends on OS and processor
        t1.start();
        t2.start();
        t3.start();

        //This will pause main thread
        Thread.sleep(9);

        System.out.println("t1 and t2 and " +
                "t3 started");
    }
}
```

```
package runnable;

public class MyRunnable implements Runnable{

    private String name;

    public MyRunnable(String name) {
        this.name = name;
    }
```

```
    @Override
    public void run() {
        //This method contains logic to execute
        // in thread
        //print numbers from 1 to 10
        for(int i=0;i<10;i++){
            System.out.println("Printing number "
                    +i+" from thread "+name);
        }
    }
}

package runnable;
public class RunnableDemo {
    public static void main(String[] args) {
        MyRunnable r1 = new MyRunnable("R1");
        MyRunnable r2 = new MyRunnable("R2");
        MyRunnable r3 = new MyRunnable("R3");
        Thread tr1 = new Thread(r1);
        Thread tr2 = new Thread(r2);
        Thread tr3 = new Thread(r3);
        System.out.println("starting thread t1 and " +
                "t2 and t3");
        //Here sequence of thread execution is random
        //This depends on OS and processor
        tr1.start();
        tr2.start();
        tr3.start();
        //This will pause main thread
        try {
            Thread.sleep(9);
        } catch (InterruptedException e) {
            System.out.println("Thread interrupted !!");
        }
        System.out.println("t1 and t2 and " +
                "t3 started");

    }


}
```

Assignment 1

Create class Car which extends Thread

Override run method and print “Car <thread name > reached destination>”
Create 4 threads with name C1 to C4
Start 4 threads and look at the output which car reaches destination first

Assignment 2
Create class CarRunnable and do same process above, with implementing Runnable interface.