MultiThreading in JAVA

Threading in Java is a way to have multiple tasks running concurrently within a single program. It allows you to execute different parts of your code simultaneously, making your program more efficient and responsive.

In Java, threading is accomplished using the Thread class and the Runnable interface. The Thread class represents a separate thread of execution, while the Runnable interface defines the task that the thread will perform. You can create threads by either extending the Thread class or implementing the Runnable interface.

By using threads, you can make your program take advantage of the available system resources, such as multiple processor cores or processors. This means that tasks can be executed in parallel, leading to improved performance.

Some important concepts related to threading in Java include:

- Thread Synchronization: When multiple threads access shared resources, you need to ensure that they don't interfere with each other. Synchronization mechanisms, like locks or synchronized blocks, help you control the access to shared data and prevent conflicts.

- Thread States: Threads in Java can be in different states, such as "new," "runnable," "blocked," "waiting," or "terminated." These states indicate what the thread is currently doing or waiting for.

- Thread Interactions: Threads can communicate and coordinate with each other through methods like wait(), notify(), and notifyAll(). These methods allow threads to pause, resume, or signal each other.

- Java provides additional features and utilities to manage threads effectively, like the java.util.concurrent package, which offers higher-level concurrency utilities and thread pools.

Threading in Java is an essential concept for developing concurrent and multi-threaded applications. It allows you to execute tasks concurrently, making your program faster and more efficient by leveraging the available system resources.

```
class Thread1 extends Thread
{
	@Override
	public void run()
	{
		while(true)
		{
			System.out.println("This is Thread 1");
			System.out.println("Executing thread 1");
		}
	}
}


class Thread2 extends Thread
{
	@Override
```

```
        public void run()
        {
                while(true)
                {
                        System.out.println("This is Thread 2");
                        System.out.println("Executing thread 2");
                }
        }
}


class MultiExam
{
        public static void main(String args[])
        {
                Thread1 t1=new Thread1();
                Thread2 t2=new Thread2();

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

Real Life-based scenario of Multi-Threading

```
class CustomerThread extends Thread {
    private final BankAccount account;

    public CustomerThread(BankAccount account) {
        this.account = account;
    }

    @Override
    public void run() {
        // Simulate a scenario where the customer deposits and withdraws money
        for (int i = 0; i < 5; i++) {
            int amountToDeposit = (int) (Math.random() * 1000);
            account.deposit(amountToDeposit);

            int amountToWithdraw = (int) (Math.random() * 1000);
            account.withdraw(amountToWithdraw);
        }
    }
}

class BankAccount {
    private int balance = 0;

    public synchronized void deposit(int amount) {
        balance += amount;
        System.out.println("Deposited: " + amount);
        System.out.println("Current balance: " + balance);
    }

    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
            System.out.println("Current balance: " + balance);
        } else {
            System.out.println("Insufficient balance for withdrawal: " + amount);
        }
    }
}

public class MultithreadingExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        // Create two customer threads
        CustomerThread customer1 = new CustomerThread(account);
        CustomerThread customer2 = new CustomerThread(account);

        // Start the customer threads
        customer1.start();
        customer2.start();
    }
}
```

To know more about Multi threading functions
Click here : List of functions in JAVA Multi-Threading

```
//Demonstrating Multithreading using Runnable
public class MultithreadingExample {

    public static void main(String[] args) {
        // Create two threads
        Thread thread1 = new Thread(new Task("Thread 1"));
        Thread thread2 = new Thread(new Task("Thread 2"));

        // Start the threads
        thread1.start();
        thread2.start();
    }

    // A simple task to be executed by the threads
    static class Task implements Runnable {
        private final String name;

        public Task(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread: " + name + ", Count: " + i);

                try {
                    Thread.sleep(1000); // Pause for 1 second
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```