day-025-multithreading

1. Multithreading refers to the concurrent execution of multiple threads (smaller units of a process) within a single process. It is important because it enables the efficient utilization of resources and improved performance of applications by allowing multiple tasks to be executed simultaneously.

2. The benefits of using multithreading include increased performance, reduced response time, improved resource utilization, and enhanced concurrency and responsiveness of applications.

3. In Java, a thread is a lightweight unit of execution that can run concurrently with other threads within a single process. Threads can be used to perform multiple tasks simultaneously, resulting in improved performance and resource utilization.

4. Here are the two ways of implementing threads in Java, along with code examples:

   1. Extending the Thread class:
      - In this approach, you create a new class that extends the Thread class and override the run() method with the code that you want to execute in the new thread. Then, you create an instance of this class and call the start() method to start the new thread.

      Example code:

```
public class MyThread extends Thread {
    public void run() {
        // Code to be executed in the new thread
    }
}

// Create a new instance of MyThread and start the new thread
MyThread myThread = new MyThread();
myThread.start();
```

   2. Implementing the Runnable interface:
      - In this approach, you create a new class that implements the Runnable interface and override the run() method with the code that you want to execute in the new thread. Then, you create an instance of this class and pass it to a new Thread

object. Finally, you call the start() method on the new Thread object to start the new thread.

Example code:

```
public class MyRunnable implements Runnable {
    public void run() {
        // Code to be executed in the new thread
    }
}

// Create a new instance of MyRunnable and pass it to a new
Thread object
MyRunnable myRunnable = new MyRunnable();
Thread myThread = new Thread(myRunnable);
myThread.start();
```

5. In computing, both threads and processes are used to achieve concurrency, which is the ability of a program to perform multiple tasks simultaneously. However, there are some key differences between threads and processes:

A process is an instance of a program that is running on a computer. It has its own memory space, system resources, and a unique process ID (PID) that identifies it. Each process runs independently of other processes, and any communication between processes must be done through inter-process communication (IPC) mechanisms such as pipes, sockets, or shared memory.

A thread, on the other hand, is a lightweight unit of execution within a process. A process can have multiple threads, and each thread shares the same memory space and system resources as the other threads in the same process. Threads are scheduled by the operating system's scheduler, which allocates CPU time to each thread based on its priority and other factors.

So, in summary, the main difference between a thread and a process is that a process is a separate instance of a program that runs independently of other processes, while a thread is a unit of execution within a process that shares the same memory space and system resources as other threads in the same process.

It's worth noting that threads are generally more lightweight and efficient than processes since they share resources and can be scheduled more quickly by the operating system. However, processes offer better isolation and security, since

each process has its own memory space and can't interfere with other processes unless explicit communication is established through IPC mechanisms.

6. In Java, we can create daemon threads by calling the `setDaemon()` method on a `Thread` object before starting it. A daemon thread is a thread that runs in the background and does not prevent the program from exiting. When all non-daemon threads have finished executing, the JVM will exit, even if there are still daemon threads running.

   Here's an example of how to create a daemon thread in Java:

```
public class MyThread extends Thread {
    public void run() {
        // Code to be executed by the daemon thread
    }
}

// Create a new instance of MyThread and set it as a daemon
thread
MyThread myThread = new MyThread();
myThread.setDaemon(true);

// Start the daemon thread
myThread.start();
```

   In this example, we create a new class `MyThread` that extends the `Thread` class and override the `run()` method with the code to be executed by the daemon thread. Then, we create a new instance of `MyThread` and call the `setDaemon(true)` method to set it as a daemon thread. Finally, we start the daemon thread by calling the `start()` method on the `myThread` object.

   It's important to note that once a thread is set as a daemon thread, it cannot be changed back to a non-daemon thread. Additionally, any threads created by a daemon thread will also be daemon threads by default, unless explicitly set otherwise.

7.

The wait() and sleep() methods are methods provided by Java for thread synchronization and time delay, respectively.

The wait() method is used to pause a thread's execution until a certain condition is met. It is typically used in multi-threaded programs to ensure that a thread does not continue until it has access to a shared resource. When a thread calls the wait() method on an object, it releases the lock on that object and waits until another thread calls the notify() or notifyAll() method on that same object.

Here is an example of how to use the wait() method:

```
synchronized (obj) {
    while (condition) {
        obj.wait();
    }
    // other code to be executed once the condition is met
}
```

The sleep() method is used to pause a thread's execution for a specified amount of time. It can be used for various purposes such as delay execution, simulate real-time behavior, and so on. When a thread calls the sleep() method, it suspends its execution for the specified amount of time, after which it resumes execution.

Here is an example of how to use the sleep() method:

```
try {
    Thread.sleep(1000); // sleep for 1 second
} catch (InterruptedException e) {
    // handle the exception
}
```

Note that the sleep() method can throw an InterruptedException if the thread is interrupted while it is sleeping.