**1. What do you mean by Multithreading? Why is it important?**

**Multithreading** in Java refers to the ability of a CPU to provide multiple threads of execution concurrently. A thread is a lightweight process, and multithreading allows a program to run multiple tasks in parallel. It is an essential feature for efficient CPU usage, especially in modern systems where multiple processors or cores are present.

**Importance of Multithreading**:

* **Improved performance**: Multithreading allows tasks to run concurrently, leading to better performance, especially on multi-core processors.
* **Responsiveness**: It helps maintain the responsiveness of applications, especially in GUI applications, where the main thread is not blocked by lengthy tasks.
* **Resource Sharing**: Threads share the same memory space and resources, which leads to efficient resource management and reduces the overhead compared to processes.

**2. What are the benefits of using Multithreading?**

The benefits of multithreading in Java include:

* **Better CPU Utilization**: Multithreading ensures that the CPU remains busy by running multiple tasks simultaneously, maximizing resource usage.
* **Enhanced Performance**: In applications that require heavy computation, multithreading allows operations to be divided into smaller chunks that can be processed concurrently, speeding up execution.
* **Improved Responsiveness**: For GUI applications, multithreading ensures that the UI remains responsive while background tasks are running (e.g., downloading files or processing data).
* **Simplified Program Structure**: Multithreading can simplify the design of complex programs, as it allows you to handle different tasks independently.

**3. What is a Thread in Java?**

A **Thread** in Java is a small unit of a process that can run independently. Each thread in a Java program represents an individual task, and multiple threads can run concurrently within the same program. A thread shares the same memory space and resources but executes a specific part of the code in parallel with other threads.

Threads are instances of the Thread class or implementations of the Runnable interface.

**4. What are the Two Ways of Implementing a Thread in Java?**

In Java, there are two primary ways to implement a thread:

1. **Extending the Thread class**:
   * Create a new class that extends the Thread class and override its run() method, which contains the code to be executed by the thread.

Example:

java

CopyEdit

class MyThread extends Thread {

public void run() {

System.out.println("Thread is running");

}

}

1. **Implementing the Runnable interface**:
   * Create a new class that implements the Runnable interface and override its run() method. The Runnable object is then passed to a Thread object to create a new thread.

Example:

java

CopyEdit

class MyRunnable implements Runnable {

public void run() {

System.out.println("Runnable thread is running");

}

}

**5. What's the Difference Between a Thread and a Process?**

* **Thread**:  
  A thread is the smallest unit of execution within a process. Multiple threads can exist within a single process and share the same memory space. Threads are lightweight and are used to perform tasks concurrently.
* **Process**:  
  A process is an independent program that runs in its own memory space. Each process has its own resources and memory, and they do not share memory with other processes. Processes are more resource-intensive than threads and are used to run separate applications.

**Key Differences**:

* Threads share memory; processes do not.
* Threads are lighter and faster to create compared to processes.
* Threads are typically used for concurrent execution within a single application, whereas processes are used for executing independent applications.

**6. How Can We Create Daemon Threads?**

A **Daemon Thread** in Java is a thread that runs in the background and is used for tasks like garbage collection or background monitoring. Daemon threads are terminated when the program ends, even if they are still running.

To create a daemon thread:

1. Create a thread.
2. Set the thread as a daemon using the setDaemon(true) method before starting it.

Example:

java

CopyEdit

class MyDaemonThread extends Thread {

public void run() {

while (true) {

System.out.println("Daemon thread is running");

}

}

}

public class DaemonThreadExample {

public static void main(String[] args) {

MyDaemonThread daemonThread = new MyDaemonThread();

daemonThread.setDaemon(true); // Mark the thread as a daemon

daemonThread.start();

}

}

In this example, the MyDaemonThread is set as a daemon thread.

**7. What are the wait() and sleep() Methods?**

* **wait()**:
  + The wait() method is used in synchronization contexts to make the current thread wait until another thread sends a notification using notify() or notifyAll(). The thread that calls wait() will release the lock it holds and enter the waiting state until it is awakened.

Example:

java

CopyEdit

synchronized (obj) {

obj.wait();

}

* **sleep()**:
  + The sleep() method is used to pause the execution of the current thread for a specified time (in milliseconds). It does not release any locks or monitor resources. It simply pauses the thread's execution for the given duration.

Example:

java

CopyEdit

try {

Thread.sleep(1000); // Sleep for 1 second

} catch (InterruptedException e) {

e.printStackTrace();

}

**Key Differences**:

* wait() is used for inter-thread communication and synchronization, whereas sleep() is used to pause the thread temporarily.
* wait() releases the lock on the object, while sleep() does not release the lock.