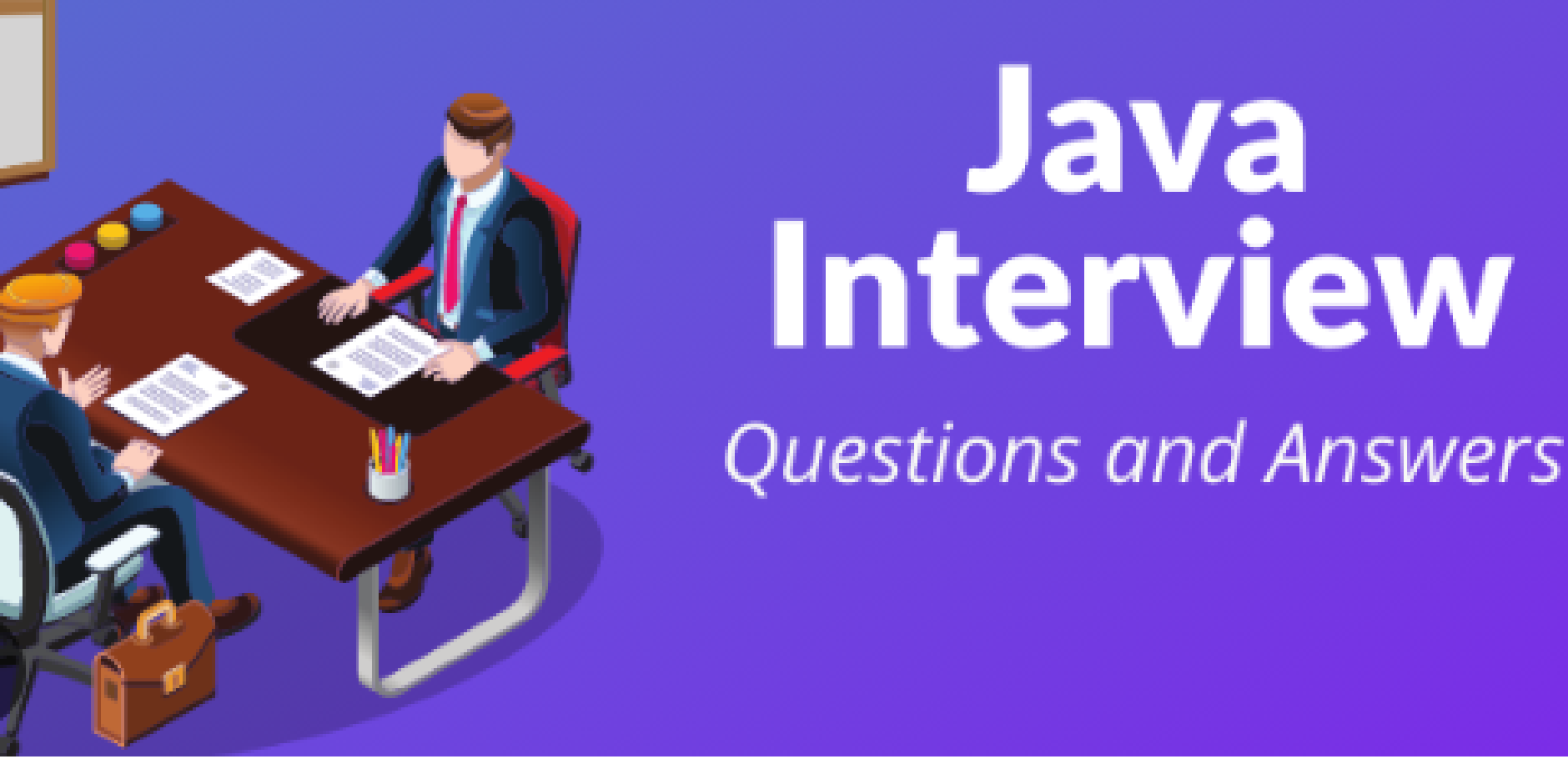
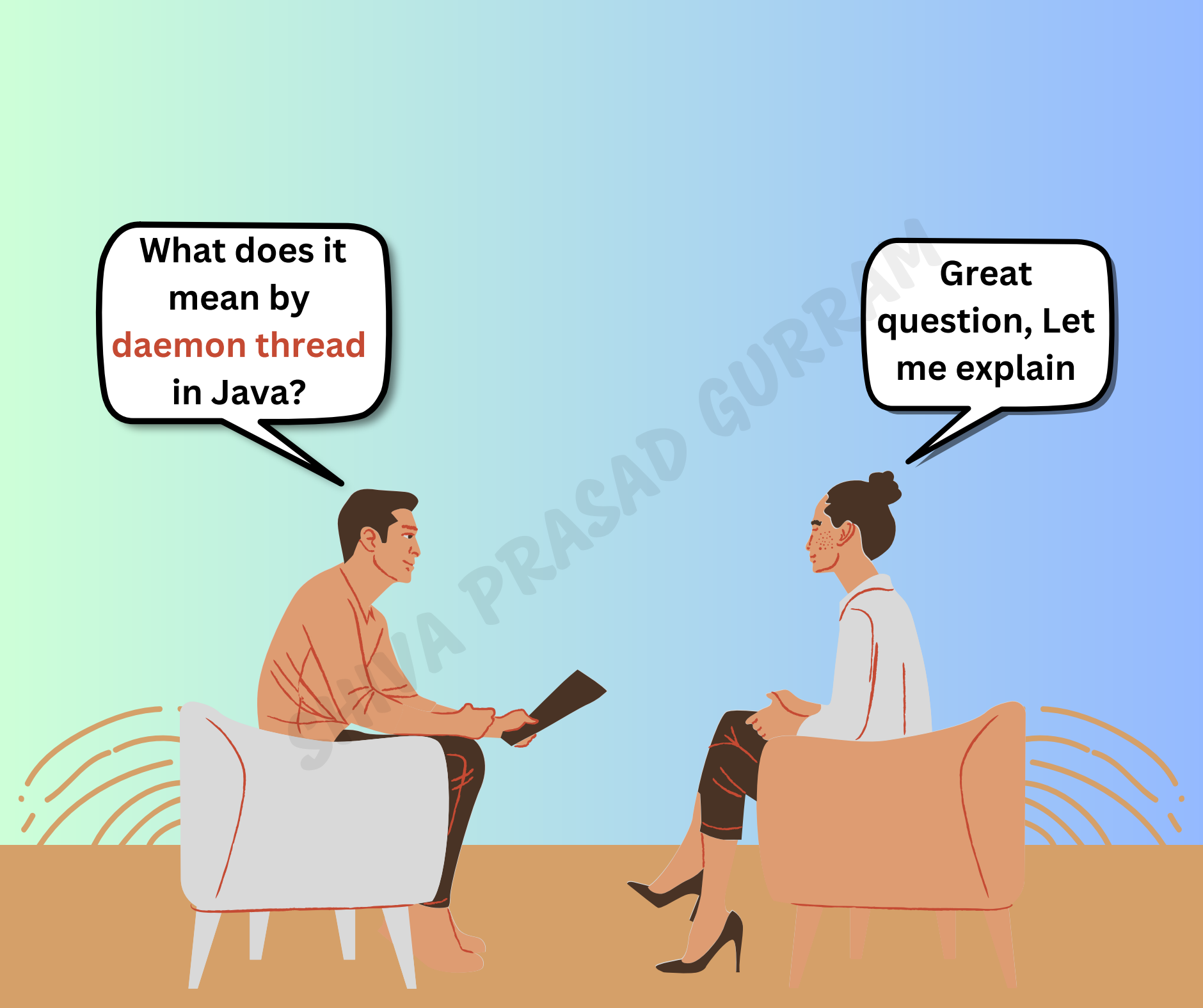


Interview Questions And Answers


What does it mean by daemon thread in Java?
Great question, Let me explain
SRINIVA PRASAD GURRAM

Understanding the Concept of Daemon Thread

Java offers two types of threads

- User thread
- Daemon thread

User Thread

User threads are high-priority threads. The JVM will wait for any user thread to complete its task before terminating it.

Daemon Thread

A Daemon thread is a type of thread that **runs in the background of another thread**. In Java, the **daemon thread has no role in life other than providing services to the user thread**. Its life depends on the mercy of user threads i.e. **when all the user threads die, JVM terminates this thread automatically**. These are **low-priority** threads.

Eg: Garbage collection in Java (gc), finalizer, etc.

Properties

- Since daemon threads are meant to serve user threads and are only needed while user threads are running, **They can not prevent the JVM from exiting** when all the user threads finish their execution.

- JVM terminates itself when all user threads finish their execution.

- If JVM finds a running daemon thread, it terminates the thread and, after that, shutdown it. **JVM does not care whether the Daemon thread is running or not**. (Until you have a **poorly designed code**).
    - For example, calling Thread.join() on a running daemon thread can block the shutdown of the application.

Nature

By default, the **main thread is always non-daemon** but for all the remaining threads, daemon nature will be inherited from parent to child. That is if the parent is Daemon, the child is also a Daemon and if the parent is a non-daemon, then the child is also a non-daemon.

Note: Whenever the last non-daemon thread terminates, all the daemon threads will be terminated automatically

Uses of Daemon Threads

Background processing

Many enterprise applications have a need for continuous background processing of tasks such as sending notifications, generating reports, or performing backups. Daemon threads can be used to perform these tasks without blocking the main thread of the application.

Resource management

In enterprise applications, resources such as database connections, network sockets, or file handles need to be managed efficiently. Daemon threads can be used to monitor and clean up unused resources, ensuring that the application runs smoothly.

Monitoring

In order to ensure high availability and reliability of enterprise applications, it's often necessary to monitor system metrics such as CPU usage, memory consumption, or network traffic. Daemon threads can be used to collect and analyze these metrics in real-time, allowing for proactive problem resolution

Messaging

In some enterprise applications, messaging systems are used for communication between different components. Daemon threads can be used to listen for incoming messages and dispatch them to the appropriate handlers.

Creating a Daemon Thread

To set a thread to be a daemon thread, all we need to do is to call **Thread.setDaemon()**. In this example, we'll use the DeamonThread class which extends the Thread class.

```
DeamonThread daemonThread = new DaemonThread();
daemonThread.setDaemon(true);
daemonThread.start();
```

Before starting the thread, we must use **setDaemon(true)**. If we do so after it has already begun, an **IllegalThreadStateException** will occur.

Can we convert the main thread to a daemon thread?

We can't because the main thread will be **created and started by JVM** and if you would like to mark it as a daemon thread you have to call **setDaemon(true) inside the main method**, that means the thread already started hence it will throw the above exception.

Does the daemon thread survive after JVM exits?

They won't survive. The JVM will exit when all the threads, except the daemon ones, have died.

When you start your application, the JVM will start a single, non-daemon thread to run your static main method.

Once the main method exits, this main thread will die, and if you spawned no other non-daemon thread, the JVM will exit.

If however, you started another thread, the JVM will not exit, it will wait for all the non-daemon threads to die before exiting.

If that thread you spawned is doing something vital, this is the right thing to do, however often you have some threads that are not that vital, maybe they are listening to some external event that may or may not happen.

So, in theory, you should place some code somewhere to stop all the threads you spawned to allow the JVM to exit.

Since this is error-prone, it's way easier to mark such non-vital threads as daemons. If they are marked as such, the JVM will not wait for them to die before exiting, the JVM will exit and kill those threads when the "main threads" (those not marked as a daemon) have died

```java
public class Spawner {
  public static void main(String[] args) {
    Thread t = new Thread(new Runnable() {
      public void run() {
        while (true) {
          System.out.println("I'm still alive");
        }
      }
    });
    // Try uncommenting/commenting this line
    // t.setDaemon(true);
    t.start();
    System.out.println("Main thread has finished");
  }
}
```

When running above code with the line commented, the thread is not a daemon, so even if your main method has finished, you'll keep on having the **console flooded until you stop it with CTRL+C**. That is, the JVM will not exit.

If you uncomment the line, **then the thread is a daemon**, and soon after the main method has finished, the thread will be killed and the JVM will exit, **without the need for CTRL+C**

Overall, daemon threads are a powerful tool for managing background tasks and resources in enterprise applications, allowing them to run more efficiently and reliably.

I tried to cover the essentials of this vast concept in this post, but because there is still much to grasp, I kindly ask that you take your time to do so.