Scenario:

Imagine you have a servlet like this:

```
public class MyServlet extends HttpServlet {

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      int count = 0;
      count++;
      response.getWriter().write("Count: " + count);
   }
}
```

- The **doGet()** method (which is called by the service() method) is **NOT synchronized** by default.
- This means **multiple threads** can execute the **doGet()** method at the **same time** without waiting for each other.

✅ **Why Does Each Thread Execute Its Own service() Method?**

Key Point:

Even though the **service() method is not synchronized**, every incoming request (thread) gets its **own stack space**.

👉 This means that if **Thread A** and **Thread B** call the doGet() method at the **same time**, both will have:

- **Their own local variables**
- **Their own execution stack**
- **Their own instance of the count variable**.

💡 **What Does This Mean in Simple Terms?**

- **Thread A** will have its own local variable count.
- **Thread B** will have its own local variable count.
- Both threads can execute the method at the same time, **without blocking each other**.
- ✅ This is why **local variables are always thread-safe** in the service() method.

✅ **What Happens If the service() Method Was Synchronized?**

Now, imagine if you did this:

```
protected synchronized void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
  int count = 0;
  count++;
  response.getWriter().write("Count: " + count);
}
```

- Now, **only ONE thread at a time** can execute the doGet() method.
- If **Thread A** is inside doGet(), **Thread B** has to **wait** until Thread A is done.
- This will drastically reduce performance and slow down your application.

- If you had made the method synchronized, then **other threads** would have to **wait**.
- But since the method is **not synchronized**, all threads can execute it **at the same time** with their own local variables.

✅ **Why Is the Local Variable Thread-Safe Then?**

The key reason is:

Local variables in Java are stored in the thread's stack memory, which is not shared with any other thread.

👉 So:

- **Thread A** gets its own count variable.
- **Thread B** gets its own count variable.
- **They never share local variables.**

✅ **What About Instance Variables?**

However, if you had declared an **instance variable** like this:

```
public class MyServlet extends HttpServlet {
    private int counter = 0;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        counter++;
        response.getWriter().write("Counter: " + counter);
    }
}
```

- Now **counter** is shared across all threads (requests).
- If **Thread A** and **Thread B** both access doGet() simultaneously:
    - Thread A increases counter from **0 to 1**.
    - Thread B increases counter from **1 to 2**.
    - But both might get mixed results (Race Condition).

❌ This is **NOT thread-safe.**

✅ **What Is SingleThreadedModel Then?**

Your professor also said:

"To add synchronization, you can make the class implement SingleThreadedModel interface..."

💡 **What Is SingleThreadedModel?**

If you modify your servlet like this:

```
public class MyServlet extends HttpServlet implements SingleThreadedModel {
    private int counter = 0;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        counter++;
        response.getWriter().write("Counter: " + counter);
    }
}
```

👉 Now, the **container (Tomcat)** will ensure that:

- **Only one thread** can access the doGet() method at a time.
- Other threads have to **wait**.
- This ensures **thread safety** but kills performance.

✅ **Why Is This a Bad Idea?**

Disadvantages of SingleThreadedModel:

1. 🚫 **It blocks requests**, meaning your application will be super slow.
2. 🚫 It defeats the whole purpose of Java Servlets, which are meant for **high concurrency**.
3. 🚫 It has been deprecated in Servlet 2.4 (so we don't use it anymore).

✅ **To Summarize Everything:**

Scenario	Local Variable Thread-Safe?	Instance Variable Thread-Safe?	Explanation
✅ **No Synchronization** (default)	✅ Yes	❌ No	Local variables are stored in **thread stack**, so they're always thread-safe. But instance variables are shared, causing race conditions.
✅ **Synchronized Method**	✅ Yes	✅ Yes	Only one thread can access the method at a time, ensuring thread-safety but reducing performance.
✅ **SingleThreadedModel**	✅ Yes	✅ Yes	Container creates a new instance per request, ensuring thread-safety but reducing performance.

- **If you use synchronized,** threads will have to wait.
- But since **the method is not synchronized**, multiple threads can execute simultaneously.
- However, **instance variables** are not thread-safe unless synchronized.

✅ **Golden Rule (Easy to Remember):**

Variable Type	Thread-Safe Without Synchronization?	Why?
✅ Local Variable (inside doGet())	✅ Yes	Each thread has its own copy (stack memory).
❌ Instance Variable (outside doGet())	❌ No	Shared across all threads (heap memory).
✅ Static Variable (class-level)	❌ No	Shared across all instances.

✅ **So Final Answer: Why Is the service() Method Thread-Safe for Local Variables?**

- Because **every thread** has its **own stack** in memory.
- The local variables (int count) are stored **inside the stack**, which is **not shared** with any other thread.
- Therefore, **local variables are automatically thread-safe**.
- But **instance variables** (private int counter;) are **not thread-safe** without synchronization.

How Tomcat handles requests (using thread pools, worker threads, etc.) and why synchronization slows things down??

✅ **Step 1: What Is Tomcat Exactly Doing?**

Tomcat is a Web Container (Servlet Container)

- Tomcat is like a **restaurant manager** that handles incoming requests (customers) and assigns a **worker (thread)** to serve each request.
- Every time you type a URL like:

http://localhost:8080/MyApp/MyServlet

👉 Tomcat receives your request and **assigns a thread** to execute the service() method of your servlet.

✅ **Step 2: Tomcat's Worker Thread Model**

What happens when Tomcat receives a request?

👉 **Imagine this scenario:**

- **You** open the browser and hit the servlet URL (MyServlet).
- **Your friend** also hits the same URL from their laptop.

🚶 **What Tomcat Does Internally (Step-by-Step):**

Event	What Happens Internally in Tomcat
Request #1 from You	Tomcat creates a new **thread** (let's call it **Thread-1**). This thread will call the **service()** method of your servlet.

Event	What Happens Internally in Tomcat
Request #2 from Your Friend	Tomcat creates **another thread** (let's call it **Thread-2**) and again calls the **service()** method of the same servlet instance.
Request #3 from Someone Else	Tomcat creates **Thread-3** and does the same thing.

✅ **IMPORTANT:**

- Tomcat **does NOT create a new servlet object for each request**.
- Instead, **it uses the same servlet object** but **creates a new thread** for each request.
- This is known as the **Multithreading Model in Servlets.**

✅ **Step 3: What Is Actually Happening in Memory?**

Memory Diagram (Internally)

Request #	Thread in Tomcat	Local Variable	Instance Variable (shared)
Request #1 (You)	**Thread-1**	**count = 10 (local)**	counter = 1 (shared)
Request #2 (Your Friend)	**Thread-2**	**count = 10 (local)**	counter = 2 (shared)
Request #3 (Someone Else)	**Thread-3**	**count = 10 (local)**	counter = 3 (shared)

✅ **Explanation:**

- The **local variable (count)** is different for each thread (because it's stored in the **thread's stack memory**).
- The **instance variable (counter)** is shared across all threads (because it's stored in **heap memory**).

✅ **Step 4: Why Does Tomcat Avoid Synchronization?**

👉 Suppose Tomcat allowed only **one thread at a time** like this:

```
protected synchronized void doGet(HttpServletRequest request, HttpServletResponse response) {
    int count = 0;
    count++;
    response.getWriter().write("Count: " + count);
}
```

👉 This means:

- **Thread-1** starts executing the method.
- **Thread-2** has to WAIT until Thread-1 is done.
- **Thread-3** also has to WAIT.

🚨 **PROBLEM: This Will Kill Your Application!**

- If **100 people** hit your servlet, Tomcat will make **99 people wait**!
- Your website will hang like a dead app. 💀
- **This is why Tomcat does not synchronize the service() method.**

✅ **Step 5: Why Is Local Variable Thread-Safe Then?**

👉 Now the magic part 🔥 !

Why Is This Code Automatically Thread-Safe?

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
    int count = 0;
    count++;
    response.getWriter().write("Count: " + count);
}
```

Explanation:

- When Thread-1 executes, it creates its own local variable (count = 0) in **stack memory**.

- When Thread-2 executes, it also creates its own local variable (count = 0).
- **Since local variables belong to the thread itself**, they are 100% **thread-safe**.

✅ **Technical Reason:**

- **Local variables** are stored in **Thread Stack Memory** (which is isolated for each thread).
- **Instance variables** are stored in **Heap Memory** (which is shared across all threads).
- This is why **local variables are always thread-safe.**

✅ **Step 6: What About Instance Variables?**

Now the big question...
👉 If you did this:

```
public class MyServlet extends HttpServlet {

    private int counter = 0;


    protected void doGet(HttpServletRequest request, HttpServletResponse response) {

        counter++;

        response.getWriter().write("Counter: " + counter);

    }

}
```

👉 What happens now?

Result (BIG PROBLEM 🚨):

Thread Value of counter

Thread-1 counter = 1

Thread-2 counter = 2

Thread-3 counter = 3

Thread Value of counter

Thread-4 counter = 4

👉 Multiple threads are modifying the **same instance variable (counter)**, which leads to:

- **Race Conditions** (unexpected output).
- **Data Corruption**.
- **Incorrect Results**.

✅ **Step 7: What Is The Role Of SingleThreadedModel?**

"To make the service() method thread-safe, you can implement SingleThreadedModel."

🚀 **What Is It?**

If you do this:

```
public class MyServlet extends HttpServlet implements SingleThreadedModel {
    private int counter = 0;
}
```

👉 Tomcat will now:

- **Create a new servlet instance** for every request.
- This means: **Thread-1 → New Servlet**, **Thread-2 → New Servlet**.
- ✅ No shared instance variables = No thread-safety issues.

🚨 **Why Is This A Bad Idea?**

- Tomcat will consume **a lot of memory**.
- It will kill performance if **thousands of requests** come in.
- That's why **SingleThreadedModel is deprecated**.

✅ **Step 8: How Does Tomcat Actually Handle Threads Internally?**

👉 When Tomcat starts, it creates a **Thread Pool** with a fixed number of worker threads.

Example:

Worker Thread Name	Status
Thread-1	Handling Request 1
Thread-2	Handling Request 2
Thread-3	Handling Request 3

What Happens When All Threads Are Busy?

- Tomcat queues the incoming requests.
- When a thread is free, it picks the next request from the queue.
- **This is why Tomcat is super fast.**

✅ **Step 9: Final Answer In One Line**

Variable Type	Stored In	Thread-Safe?	Why?
✅ **Local Variable**	Thread Stack	✅ Always	Each thread has its own copy.
❌ **Instance Variable**	Heap Memory	❌ No	Shared by all threads.
✅ **Synchronized Method**	Heap Memory	✅ Yes	One thread at a time (bad performance).

✅ 💯 **Quick Fun Fact**

- **Tomcat can handle up to 10,000 requests per second** using multithreading.
- But if you synchronize the service() method, your application will be **dead slow**. 🚨
- This is why Tomcat **avoids synchronization** and uses **local variables for thread safety.**

💡 **So What Should You Do In Real Life?**

Scenario	Solution
Local Variable (inside doGet())	✅ Always thread-safe, no problem.
Instance Variable (outside doGet())	❌ Use synchronized OR avoid using instance variables.
Global Data (database access)	✅ Use synchronized or connection pooling.