Partially Constructed Object

A common misconception is that instantiating an object is one clean atomic step, but it is three major steps:

instance = new SingleObject();

1. Allocate memory
2. Execute the constructor
3. Assign the reference

The JVM can reorder steps 2 and 3 for performance because, once the memory is allocated, it can be assigned before or after the constructor executes. In that case, another thread could examine the instance variable between steps 2 and 3 and see that it is null, yet not fully initialized. This is known as being partially constructed at that moment.

The JVM is allowed to do this for optimization because the Java Memory Model supports instruction reordering, CPU optimization, and register-based cache writes. As long as single-threaded behavior stays correct, it can reorder the operations. It does not assume another thread is watching the fields unless your code tells it. Keyword volatile does that.

Example (I had generative AI create this example):

```java
public class ConfigurationManager {
    private Map<String, String> config;

    // notice it is not volatile
    private static ConfigurationManager instance;

    private ConfigurationManager() {
        config = new HashMap<>();
        config.put("mode", "PROD");
    }

    public static ConfigurationManager getInstance() {
        if (instance == null) {
            synchronized (ConfigurationManager.class) {
                if (instance == null) {
                    instance = new ConfigurationManager();
                }
            }
        }
        return instance;
    }

    public String get(String key) {
        return config.get(key);
    }
}
```

What could happen:

Thread A	Thread B
1. Enters synchronized block	waits
2. Allocates memory for ConfigurationManager	waits
3. Assigns reference to instance	waits
4. (Context switch happens here)	calls getInstance()
5. waits	Sees that instance is not null
6. waits	Calls get("mode")
7. waits	Uses config, thus NullPointerException
8. Instantiates config	

Adding volatile fixes this.

```
private static volatile ConfigurationManager instance;
```

Volatile guarantees visibility (all writes are visible to other threads) and ordering (it forces the JVM not to reorder the assignment and constructor steps).

Today, the Holder pattern (see the lecture) prevents the issue and is preferred. It is synchronized by the JVM, thus you do not need to worry about synchronization or volatile issues.

```
private static class Holder {
    private static final ConfigurationManager INSTANCE =
        new ConfigurationManager();
}
```