Why wait() and notify() are always called from Synchronized context?

JAVINPAUL

MAR 28, 2024 • PAID

Hello guys, multithreading and concurrency are some of the most important topics for Java interviews and a good knowledge of them goes a long way in receiving a good offer.

In the past few posts, I have shared popular Java interview questions like How ConcurrentHashMap work in Java? Why String is Immutable and What is the difference between List, List<Object> and today, I am going to share another popular Java multithreading interview question, **why wait() and notify() methods are called in synchronized context?**

From synchronized context, I mean from a synchrnonized method or a synchronized block.

But before that let's find out what is wait() and notify() method in Java and why are they improtant?

What is wait(), notify(), and notifyAll() in Java?

`wait()` and `notify()` are methods in Java that are used for inter-thread communication, particularly in scenarios where one thread needs to wait for a certain condition to be met before continuing execution.

These methods are associated with the concept of a monitor, which in Java is implemented using intrinsic locks (synchronization).

They are used when one thread needs to wait for a certain condition to be met before proceeding, and another thread needs to notify the waiting thread when that condition has been fulfilled.

1. **wait() Method**:
    - The `wait()` method is defined in the `Object` class in Java and is used to make a thread wait until another thread invokes the `notify()` method or the `notifyAll()` method for the same object.
    - When a thread calls `wait()`, it releases the lock it holds on the object and enters the waiting state until it receives a notification from another thread.

- It's important to call `wait()` inside a synchronized block or method to ensure that the thread owns the intrinsic lock associated with the object. Otherwise, it will throw an `IllegalMonitorStateException`.
- `wait()` has several overloaded versions, allowing you to specify a timeout or a condition for waiting.

2. **notify() Method**:
   - The `notify()` method is also defined in the `Object` class and is used to wake up a single thread that is waiting on the object.
   - When `notify()` is called, it notifies one of the threads that are waiting on the object, allowing it to proceed. The choice of which thread to notify is not specified and is left to the implementation.
   - Like `wait()`, `notify()` should be called within a synchronized block or method to ensure proper synchronization.
3. **notifyAll() Method**:
   - The `notifyAll()` method is also defined in the `Object` class and is used to wake up all the threads that are waiting on the object.
   - When `notifyAll()` is called, all threads that are waiting on the object are notified and moved to the runnable state. This ensures that all waiting threads have an opportunity to proceed.
   - Like `wait()` and `notify()`, `notifyAll()` should be called within a synchronized block or method.

Here's a simple example demonstrating the usage of `wait()` and `notify()` in Java:

```
class SharedObject {

boolean isReady = false;

synchronized void waitForReady() throws InterruptedException {

while (!isReady) {

wait(); // Waits until another thread calls notify()

}

System.out.println("Work can now proceed...");

}
```

```
synchronized void notifyReady() {

isReady = true;

notify(); // Wakes up one waiting thread

}

}

class WorkerThread extends Thread {

SharedObject sharedObject;

WorkerThread(SharedObject sharedObject) {

this.sharedObject = sharedObject;

}

public void run() {

try {

sharedObject.waitForReady();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class Main {

public static void main(String[] args) {

SharedObject sharedObject = new SharedObject();

WorkerThread thread1 = new WorkerThread(sharedObject);

WorkerThread thread2 = new WorkerThread(sharedObject);
```

```
thread1.start();

thread2.start();

// Simulating some work

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

sharedObject.notifyReady(); // Notifies waiting threads

}

}
```

In this example, `WorkerThread` instances wait for `SharedObject` to be ready by calling `waitForReady()`, which internally calls `wait()`. The main thread then calls `notifyReady()`, which notifies the waiting threads to proceed.

Why wait() and notify() needs to be called from a synchronized context

Now that you know what is wait() and notify() in Java and what does they do, now let's answer the question, why wait() and notify() needs to be called from a synchronized context:

Here's why `wait()` and `notify()` are typically called within a synchronized context:

1. **Thread Safety**
   Both `wait()` and `notify()` operate on a shared resource or object. To ensure thread safety, it's essential to synchronize access to this shared resource.
   By calling `wait()` and `notify()` within a synchronized block or method, you ensure that only one thread can access them at a time, preventing race conditions and maintaining the integrity of the shared resource.
2. **Lock Ownership**
   When a thread calls `wait()` within a synchronized block or method, it releases the intrinsic lock it holds on the object, allowing other threads to acquire the lock and

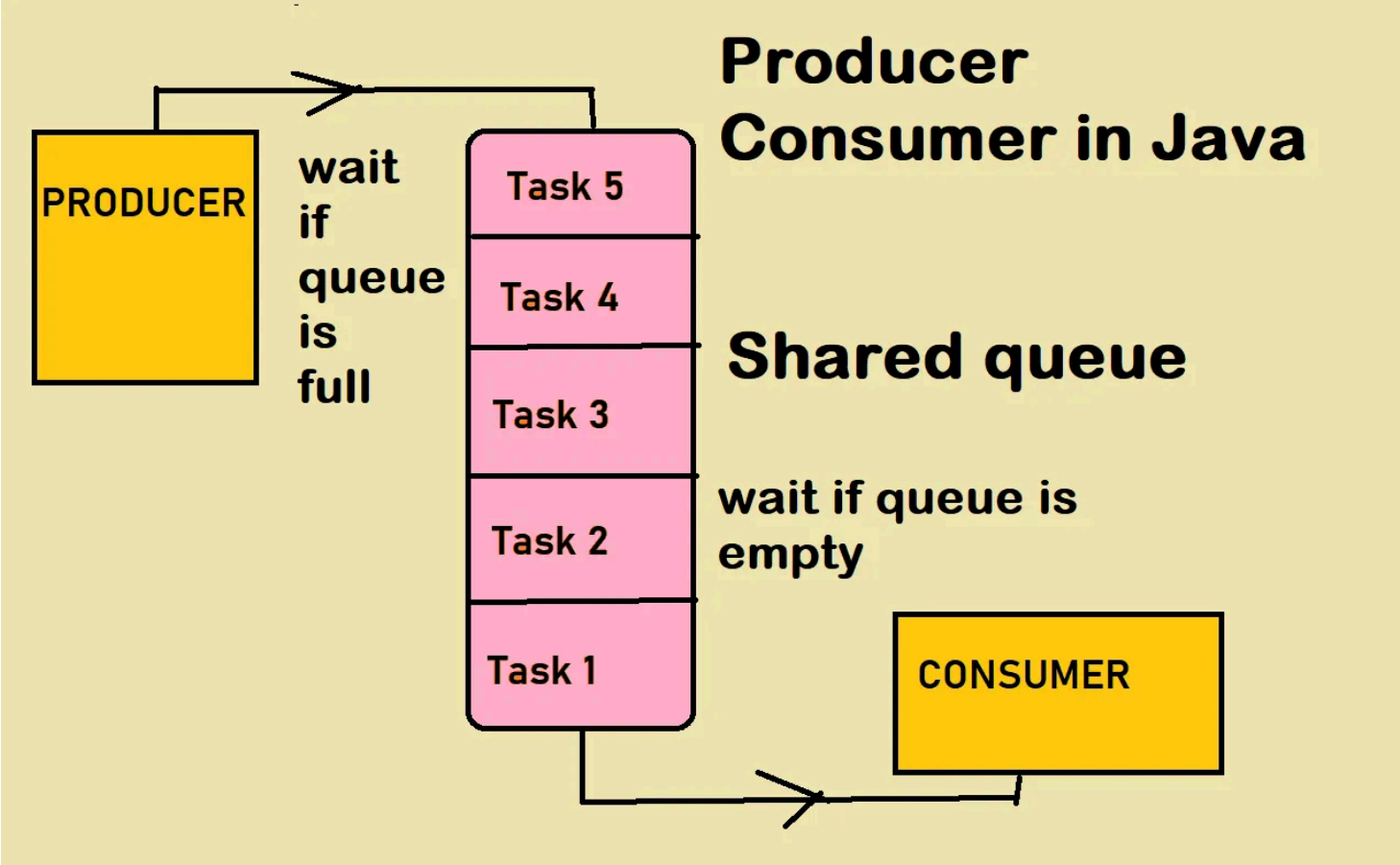
potentially modify the shared state.
Similarly, when `notify()` is called within a synchronized block or method, it notifies a waiting thread and potentially changes the shared state. Without synchronization, there's no guarantee of consistent behavior when multiple threads are accessing the shared resource.

3. **Violation of Monitor Ownership Principle**
Java's monitor model follows the principle of monitor ownership, which means that a thread must own the monitor associated with an object to invoke `wait()`, `notify()`, or `notifyAll()` on that object.
Since intrinsic locks in Java correspond to object monitors, calling `wait()` and `notify()` without owning the intrinsic lock would violate this principle and lead to runtime errors.

In summary, calling `wait()` and `notify()` from a synchronized context ensures thread safety, prevents race conditions, and adheres to the monitor ownership principle, making the code robust and reliable in multi-threaded environments.

That's all on why wait() and notify(), and notify() in Java is called from synchnrozied method or block.

Another good example of wait() and notify() is to implement Producer Consumer pattern in Java, you can actually try that to check if you can write code using wait() and notify correctly or not. This is also a followup question to see if you can actually use wait() and notify() mehods.



Let me know how did you find this article

All the best for your interviews!!

*And, if need help with your interview preparation, you can also check my Java + Spring Interview + SQL Bundle on Gumroad, use code **friends20** to get a 20% discount also*

5 Likes

← Previous

Next →

Comments

Write a comment...