Concept behind putting wait(),notify() methods in Object class [duplicate]

Ask Question

This question already has an answer here:
How can the wait() and notify() methods be called on Objects that are not threads? *8 answers*

I am just having hard time to understand concept behind putting `wait()` in `Object` class. For this questions sake consider if `wait()` and `notifyAll()` are in `Thread` class.

```
class Reader extends Thread {
    Calculator c;
    public Reader(Calculator calc) {
        c = calc;
    }

    public void run() {
        synchronized(c) {                              //line 9
        try {
            System.out.println("Waiting for calculation...");
            c.wait();
        } catch (InterruptedException e) {}
            System.out.println("Total is: " + c.total);
        }
    }

    public static void main(String [] args) {
        Calculator calculator = new Calculator();
        new Reader(calculator).start();
        new Reader(calculator).start();
        new Reader(calculator).start();
        calculator.start();
    }
}

class Calculator extends Thread {
    int total;
    public void run() {
        synchronized(this) {                     //Line 31
            for(int i=0;i<100;i++) {
                total += i;
            }
             notifyAll();
        }
    }
}
```

My Question is that what difference it could have made? In line 9 we are acquiring lock on object c and then performing wait which satisfy the condition for wait that we need to acquire lock on the object before we use wait and so is case for notifyAll we have acquired lock on object of Calculator at line 31.

java multithreading wait notify

edited Jul 11 '16 at 14:27
Ravindra babu
25.8k 5 131 127

asked Jul 24 '13 at 16:58
Sunny
607 3 14 26

marked as duplicate by EJP java May 7 '17 at 18:23

This question has been asked before and already has an answer. If those answers do not fully address your question, please ask a new question.

4 It is hard to understand what you are asking... – assylias Jul 24 '13 at 17:01

i am asking if we have putted Wait and notify in Thread class then also i think this code might have worked. – Sunny Jul 24 '13 at 17:02

1 `.wait()` and `.notify{,All}()` are on `Object`, not `Thread`. This is what allows the implementation of many locking primitives in the JVM (`Semaphore`, `CountDownLatch`, etc) – fge Jul 24 '13 at 17:04

This code doesn't really apply to the question, since `Thread` is a subclass of `Object` just like everything else. You never attempt to call `wait()` on a non-`Thread` object, so the code is pretty irrelevant to your question. – Henry Keiter Jul 24 '13 at 17:04

11 Answers

> I am just having hard time to understand concept behind putting wait() in object class For this questions

> sake consider as if wait() and notifyAll() are in thread class

In the Java language, you `wait()` on a particular instance of an `Object` – a monitor assigned to that object to be precise. If you want to send a signal to one thread that is waiting on that specific object instance then you call `notify()` on that object. If you want to send a signal to all threads that are waiting on that object instance, you use `notifyAll()` on that object.

If `wait()` and `notify()` were on the `Thread` instead then each thread would have to know the status of every other thread. How would thread1 know that thread2 was waiting for access to a particular resource? If thread1 needed to call `thread2.notify()` it would have to somehow find out that `thread2` was waiting. There would need to be some mechanism for threads to register the resources or actions that they need so others could signal them when stuff was ready or available.

In Java, the object itself is the entity that is shared between threads which allows them to communicate with each other. The threads have no specific knowledge of each other and they can run asynchronously. They run and they lock, wait, and notify on the *object* that they want to get access to. They have no knowledge of other threads and don't need to know their status. They don't need to know that it is thread2 which is waiting for the resource – they just notify on the resource and whomever it is that is waiting (if anyone) will be notified.

In Java, we then use lock objects as synchronization, mutex, and communication points between threads. We synchronize on a lock object to get mutex access to an important code block and to synchronize memory. We wait on an object if we are waiting for some condition to change – some resource to become available. We notify on an object if we want to awaken sleeping threads.

```java
// locks should be final objects so the object instance we are synchronizing on,
// never changes
private final Object lock = new Object();
...
// ensure that the thread has a mutex lock on some key code
synchronized (lock) {
    ...
    // i need to wait for other threads to finish with some resource
    // this releases the lock and waits on the associated monitor
    lock.wait();
    ...
    // i need to signal another thread that some state has changed and they can
    // awake and continue to run
    lock.notify();
}
```

There can be any number of lock objects in your program – each locking a particular resource or code segment. You might have 100 lock objects and only 4 threads. As the threads run the various parts of the program, they get exclusive access to one of the lock objects. Again, they don't have to know the running status of the other threads.

This allows you to scale up or down the number of threads running in your software as much as you want. You find that the 4 threads is blocking too much on outside resources, then you can increase the number. Pushing your battered server too hard then reduce the number of running threads. The lock objects ensure mutex and communication between the threads independent on how many threads are running.

edited Jul 15 '17 at 23:33

answered Jul 24 '13 at 17:59
Gray
92.7k 14 216 285

3 Wow self explanatory, but as you have pointed I wish to get in detail why we call .Wait () from the synchronized block as in wait state it releases the lock for others which makes the resources accessible to other threads. – Sunny Jul 25 '13 at 5:12

That's just part of the spec @Sunny. You need to have the lock to send a `notify()` so the `wait()` has to unlock it first. – Gray Jul 25 '13 at 13:49

For better understanding why wait() and notify() method belongs to Object class, I'll give you a real life example: Suppose a gas station has a single toilet, the key for which is kept at the service desk. The toilet is a shared resource for passing motorists. To use this shared resource the prospective user must acquire a key to the lock on the toilet. The user goes to the service desk and acquires the key, opens the door, locks it from the inside and uses the facilities.

Meanwhile, if a second prospective user arrives at the gas station he finds the toilet locked and therefore unavailable to him. He goes to the service desk but the key is not there because it is in the hands of the current user. When the current user finishes, he unlocks the door and returns the key to the service desk. He does not bother about waiting customers. The service desk gives the key to the waiting customer. If more than one prospective user turns up while the toilet is locked, they must form a queue waiting for the key to the lock. Each thread has no idea who is in the toilet.

Obviously in applying this analogy to Java, a Java thread is a user and the toilet is a block of code which the thread wishes to execute. Java provides a way to lock the code for a thread which is currently executing it using the synchronized keyword, and making other threads that wish to use it wait until the first thread is finished. These other threads are placed in the waiting state. Java is NOT AS FAIR as the service station because there is no queue for waiting threads. Any one of the waiting threads may get the monitor next, regardless of the order they asked for it. The only guarantee is that all threads will get to use the monitored code sooner or later.

Finally the answer to your question: the lock could be the key object or the service desk. None of which is a Thread.

However, these are the objects that currently decide whether the toilet is locked or open. These are the objects that are in a position to notify that the bathroom is open (“notify”) or ask people to wait when it is locked wait.

answered Sep 3 '15 at 16:44
Roushan
1,837 2 9 24

1 Clear explaination..... – Manan Shah Oct 2 '16 at 3:12

People in the world are like threads on their way they use shared resources like waiting lobby chairs in railway station, petrol fuel station etc. All these people don't know who is waiting for these they acquire and release resource. It is resources announces that they are free and available, not the people , this is why object class has wait() and notify() methods. – malatesh Aug 6 '17 at 4:33

The other answers to this question all miss the key point that in Java, there is one mutex associated with **every** object. (I'm assuming you know what a mutex or "lock" is.) This is *not* the case in most programming languages which have the concept of "locks". For example, in Ruby, you have to explicitly create as many `Mutex` objects as you need.

I think I know why the creators of Java made this choice (although, in my opinion, it was a mistake). The reason has to do with the inclusion of the `synchronized` keyword. I believe that the creators of Java (naively) thought that by including `synchronized` methods in the language, it would become easy for people to write correct multithreaded code -- just encapsulate all your shared state in objects, declare the methods that access that state as `synchronized` , and you're done! But it didn't work out that way...

Anyways, since any class can have `synchronized` methods, there needs to be one mutex for each object, which the `synchronized` methods can lock and unlock.

`wait` and `notify` both rely on mutexes. Maybe you already understand why this is the case... if not I can add more explanation, but for now, let's just say that both methods need to work on a mutex. Each Java object has a mutex, so it makes sense that `wait` and `notify` can be called on any Java object. Which means that they need to be declared as methods of `Object` .

Another option would have been to put static methods on `Thread` or something, which would take any `Object` as an argument. That would have been much less confusing to new Java programmers. But they didn't do it that way. It's much too late to change any of these decisions; too bad!

answered Jul 24 '13 at 21:17
Alex D
23k 3 55 100

My answer specifically talks about one monitor per object. Also, in Java you can also use `ReentrantLock` or other locking mechanisms built into the JDK if you so wish. – Gray Jul 24 '13 at 21:20

2 OK, noted, +1 for including that point. It's true that later JRE versions include explicit lock objects, but right from day one, the implicit mutexes have been there, and that is why `wait` and `notify` were made methods on `Object` . If explicit lock objects, or better yet, condition queue objects were included in the original JRE, then `wait` and `notify` would certainly have been associated with them. – Alex D Jul 24 '13 at 21:33

Answer to your first question is As every object in java has only one `lock(monitor)` and `wait(),notify(),notifyAll()` are used for monitor sharing thats why they are part of `Object` class rather than `Thread` class.

answered Jul 24 '13 at 17:02
amicngh
6,450 2 17 39

true, but if wait was part of Thread class then also i think they could have shared the lock – Sunny Jul 24 '13 at 17:12

In simple terms, the reasons are as follows.

1. `Object` has monitors.
2. Multiple threads can access one `Object`. Only one thread can hold object monitor at a time for `synchronized` methods/blocks.
3. `wait(), notify() and notifyAll()` method being in `Object` class allows all the threads created on that `object` to communicate with other
4. Locking (using `synchronized or Lock` API) and Communication (`wait() and notify()`) are two different concepts.

If `Thread` class contains `wait(), notify() and notifyAll()` methods, then it will create below problems:

1. `Thread` communication problem
2. `Synchronization` on object won't be possible. If each thread will have monitor, we won't have any way of achieving synchronization
3. `Inconsistency` in state of object

Refer to this article for more details.

edited Dec 15 '17 at 14:44

answered Jul 11 '16 at 11:17

Ravindra babu

25.8k 5 131 127

Threads are not created "on an object". Multiple threads do not exist for one object. – Gray Dec 15 '17 at 14:16

re-phrased sentence. – Ravindra babu Dec 15 '17 at 14:45

These methods works on the locks and locks are associated with Object and not Threads. Hence, it is in Object class.

The methods wait(), notify() and notifyAll() are not only just methods, these are synchronization utility and used in communication mechanism among threads in Java.

For more detailed explanation, please visit : ***http://parameshk.blogspot.in/2013/11/why-wait-notify-and-notifyall-methods.html*** answered Nov 20 '13 at 21:14

Paramesh Korrakuti
671 1 15 17

This is just my 2 cents on this question...not sure if this holds true in its entirety.

Each Object has a monitor and waitset --> set of threads (this is probably more at OS level). This means the monitor and the waitset can be seen as private members of an object. Having wait() and notify() methods in Thread class would mean giving public access to the waitset or using get-set methods to modify the waitset. You wouldnt want to do that because thats bad designing.

Now given that the Object knows the thread/s waiting for its monitor, it should be the job of the Object to go and awaken those threads waiting for it rather than an Object of thread class going and awakening each one of them (which would be possible only if the thread class object is given access to the waitset). However, it is not the job of a particular thread to go and awaken each of the waiting threads. (This is exactly what would happened if all these methods were inside the Thread class). Its job is just to release the lock and move ahead with its own task. A thread works independently and does not need to know whihc other threads are waiting for the objects monitor (it is an unnecessary detail for the thread class object). If it started awakening each thread on its own.. it is moving from away its core functionality and that is to carry out its own task. When you think about a scene where there might 1000's of threads.. you can assume how much of a performance impact it can create. Hence, given that Object Class knows who is waiting on it, it can carry out the job awakening the waiting threads and the thread which sent notify() can carry out with its further processing.

To give an analogy (perhaps not the right one but cant think of anything else). When we have a power outage, we call a customer representative of that company because she knows the right people to contact to fix it. You as a consumer are not allowed to know who are the engineers behind it and even if you know, you cannot possibly call each one of them and tell them of your troubles (that is not ur duty. Your duty is to inform them about the outage and the CR's job is to go and notify(awaken) the right engineers for it).

Let me know if this sounds right... (I do have the ability to confuse sometimes with my words).

answered Mar 30 '14 at 17:16

BHS
9 1

wait and notify operations work on implicit lock, and implicit lock is something that make inter thread communication possible. And all objects have got their own copy of implicit object. so keeping wait and notify where implicit lock lives is a good decision.

Alternatively wait and notify could have lived in Thread class as well. than instead of wait() we may have to call Thread.getCurrentThread().wait(), same with notify. For wait and notify operations there are two required parameters, one is thread who will be waiting or notifying other is implicit lock of the object . both are these could be available in Object as well as thread class as well. wait() method in Thread class would have done the same as it is doing in Object class, transition current thread to waiting state wait on the lock it had last acquired.

So yes i think wait and notify could have been there in Thread class as well but its more like a design decision to keep it in object class.

answered Jul 4 '14 at 8:58
leoismyname
226 4 10

---

wait - wait method tells the current thread to give up monitor and go to sleep.

notify - Wakes up a single thread that is waiting on this object's monitor.

So you see wait() and notify() methods work at the monitor level, thread which is currently holding the monitor is asked to give up that monitor through wait() method and through notify method (or notifyAll) threads which are waiting on the object's monitor are notified that threads can wake up.

Important point to note here is that monitor is assigned to an object not to a particular thread. That's one reason why these methods are in Object class. To reiterate threads wait on an Object's monitor (lock) and notify() is also called on an object to wake up a thread waiting on the Object's monitor.

answered May 7 '17 at 18:01
Ankur Kunal
11 1 4

---

the `wait()` method will release the lock on the specified object and waits when it can retrieve the lock.

the `notify()` , `notifyAll()` will check if there are threads waiting to get the lock of an object and if possible will give it to them.

The reason why the locks are a part of the objects is because the resources (RAM) are defined by `Object` and not `Thread` .

The easiest method to understand this is that Threads can share Objects (in the example is calculator shared by all threads), but objects can not share Attributes (like primitives, even references itself to Objects are not shared, they just point to the same location). So in orde to make sure only one thread will modify a object, the synchronized locking system is used

answered Jul 24 '13 at 17:15
Joris W
432 3 15

Your answer is confusing locks and conditions. They are different. wait releases the lock and waits on the condition. Notify releases a thread (or threads) waiting on the condition but does not release the lock. – Gray Dec 15 '17 at 13:57

Wait and notify method always called on object so whether it may be Thread object or simple object (which does not extends Thread class) Given Example will clear your all the doubts.

I have called wait and notify on class ObjB and that is the Thread class so we can say that wait and notify are called on any object.

```
public class ThreadA {
    public static void main(String[] args){
        ObjB b = new ObjB();
        Threadc c = new Threadc(b);
        ThreadD d = new ThreadD(b);
        d.setPriority(5);
        c.setPriority(1);
        d.start();
        c.start();
    }
}

class ObjB {
    int total;
    int count(){
        for(int i=0; i<100 ; i++){
            total += i;
        }
        return total;
    }}


class Threadc extends Thread{
    ObjB b;
    Threadc(ObjB objB){
        b= objB;
    }
    int total;
    @Override
    public void run(){
        System.out.print("Thread C run method");
        synchronized(b){
            total = b.count();
            System.out.print("Thread C notified called ");
            b.notify();
        }
    }
}

class ThreadD extends Thread{
    ObjB b;
    ThreadD(ObjB objB){
        b= objB;
    }
    int total;
    @Override
    public void run(){
        System.out.print("Thread D run method");
        synchronized(b){
            System.out.println("Waiting for b to complete...");
            try {
                b.wait();
                System.out.print("Thread C B value is" + b.total);
                }
                catch (InterruptedException e) {
                    e.printStackTrace();
                }
        }
    }
}
```

edited Jan 9 '17 at 8:23
iAsghar Hussain
15 6

answered Sep 25 '16 at 18:33
Abhinav Gupta
1