ENTERPRISE JAVA PROGRAMMING / PROGRAMMING II

MODULES CT545 / CT875, SEMESTER 2, ACADEMIC YEAR 2020-2021
NATIONAL UNIVERSITY OF IRELAND, GALWAY

Lecture 3b

Lecturer: Dr Patrick Mannion

TOPICS

- More about multithreading in Java

MULTITHREADING
wait() and notify()

- Sometimes, although a thread has acquired a lock it cannot continue because some condition is not yet true.
  E.g., a thread might require some data produced by another thread which is not available yet because the other thread hasn't finished yet.
- "*Condition variables*" allow a thread to block (suspend execution) until *notified* by another thread that the necessary condition is fulfilled (e.g., that "data is available now").
  This way multiple threads "communicate" with each other.

- Remark: "Condition variables" is misleading - it's an official term, but also a misnomer, as they have nothing to do with Java variables.

MULTITHREADING
wait() and notify()

- We look at the classic *Consumer-Producer* scenario:
  - A thread called the *producer* provides data in a queue (called "buffer").
  - Another thread (the *consumer)* removes this data from the queue.
  - First, we look only at the Consumer code and just assume that the Producer "fills" the queue with new content whenever it is empty.
  - In method `getValue()` (next slide), the consumer waits until the buffer is full and then "consumes" and returns the buffer contents.
  - Note that on both sides, the queue (`buffer`) is a shared resource!

MULTITHREADING
wait() and notify()

First we *try* to create the "consume" method of the consumer task (erroneous code!):

```
public Object getValue() { // part of the consumer class
        Object val;
        boolean consumed = false;
        synchronized(buffer) { // <- because buffer is shared with the producer
                while (!consumed) { // loop until something has been consumed…
                        if(!buffer.isEmpty()) { // New data? Consume it!
                                consumed = true;
                                val = buffer.getBufferContents();
                        }
                }
                return val; // return the consumed data
        }
}
```

In the code above, `buffer` could be any object accessed by the producer and consumer (e.g. a data structure such as an instance of the `Vector` class)

What doesn't work here, and why doesn't it work…?
(You can answer this question even without knowing the rest of the program!)

MULTITHREADING
wait() and notify()

- Access to shared resource `buffer` is (correctly!) guarded using `synchronized`.
  However, the problem is that in the producer code, any access of `buffer` (including any code which fills the buffer) must *also* be guarded using `synchronized` (remember that any access to a shared resource should be synchronized, everywhere in the program).
- Therefore, if the `synchronized(buffer)` –block in `getValue` is entered by the consumer thread, this blocks off the producer from adding any data to buffer. So, the buffer never gets filled and the consumer (in method `getValue`) loops forever…

- Further (secondary) problem: the waiting thread consumes unnecessary processing time (*busy waiting*)

MULTITHREADING
wait() and notify()

- Solution:
  Allow the consumer-thread to wait for the condition to become true (here: producer has produced data) without blocking the buffer (and without "busy waiting").
- Achieved by build-in method `wait()`, called on the shared resource (i.e., the object which is used to lock out other threads in the `synchronized(...)` block).

(Remark: if an entire method is synchronized, remember that the lock-object is `this`, i.e., the object on which the method was called)

MULTITHREADING

wait() and notify()

- `wait` suspends execution of the thread until it receives a notification from some other thread
- After a while, the other thread signals that the condition became true (here: the producers signals that data is ready for consuming). This ends the waiting. Signaling is done using `notify()` or `notifyAll()` in the other thread (the producer), called on the lock-object
- Both `wait` and `notify/notifyAll` must be called within a `synchronized`-block (!!). This works because `wait` temporarily gives up control of the lock, so that other threads can access any block of code which is inside `synchronized(lock){...}`

MULTITHREADING
wait() and notify()

- The same needs to be done the other way round too:

The producer needs to wait until the consumer is ready to consume more data (that is, until the client has consumed=emptied the buffer), and the consumer needs to signal (notify) the producer that it has consumed the data and emptied the buffer, so that the producer can write new data into the buffer, and so on...

So we the consumer waits for the producer and the producer waits for the consumer. Each notifies the respective other when it can stop waiting.

MULTITHREADING

wait() and notify()

- Correct consumer code:

```
public Object getValue() {
    synchronized(buffer) { // <- we still need this too!
            while(buffer.isEmpty())
                    buffer.wait();  // wait until producer
                                    // signals that data is
                                    // available

            Object val = buffer.getBufferContents();
            buffer.clear();  // ^ we "consume" the data
            buffer.notifyAll();  // tell producer that buffer
                                 // contents has been consumed
            return val;
    }
}
```

MULTITHREADING

wait() and notify()

- Producer code:

```
public void putValue(Object x) {


    synchronized(buffer) {
          while(buffer.isFull())
                buffer.wait();
        // ("buffer.isFull" means that the consumer hasn't consumed
        // the buffer content yet)


          buffer.putBuffer(x);  // put new data x into buffer


          buffer.notify (); // tell the consumer that
                            // new data is ready to be
                              // consumed, so the consumer
                              // can stop waiting.
     } }
```

MULTITHREADING
wait() and notify()

- wait/notify in more detail:
  - `o.wait()` temporary releases the lock of object `o` which it had acquired before and blocks the thread until *notified*. Afterwards, the lock is re-acquired by the thread.
  - `o.notify()` wakes up one a randomly selected thread that is currently blocked via `o.wait()` (or the only thread if there is just one thread waiting)
  - `o.notifyAll()` wakes up all threads that are currently blocked via `o.wait()`
  - Note that, like synchronized, wait/notify are specific to a given object (the `o` above)

MULTITHREADING

notify() vs. notifyAll()

- notify() wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the JVM implementation. A thread waits on an object's monitor by calling one of the wait methods.
- https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notify--
- notifyAll() wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods. The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.
- https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#notifyAll--

MULTITHREADING

notify() vs. notifyAll()

- Use notifyAll() for situations where more than one of the waiting threads will be able to do something useful when the lock has been released on the object (e.g. a number of threads waiting for a task to finish)
- Use notify() for situations where only one thread will be able to do something useful when the lock has been released on the object (e.g. when the released object needs to be locked again immediately by the next thread)

MULTITHREADING

wait() and notify()

- `o.wait()`, `o.notify()` and `o.notifyAll()` can only be invoked within a synchronized method or block!
- The invoking thread must have acquired the lock of object `o` (otherwise, an exception is thrown).

- Question: Why do we still use while-loops here, although `wait()` does all the waiting for us?

```
while(buffer.isEmpty())
        buffer.wait();
```

???

MULTITHREADING
wait() and notify()

- There are two reasons why `wait()` should always be put inside a while-loop:
  - The *event* (i.e., the condition) which "wakes up" `wait()` is unspecific. In some situations (large program with many threads) you might not know why exactly the other thread called `notify()/notifyAll()`. It could signal the fulfillment of some other condition instead of the condition your method is waiting for. The end-condition of the while-loop catches these signals.
  - In rare cases, it might be possible that the condition was true when sending the notification with `notify()/notifyAll()`, but changed again a moment later when `wait()` returns (that is, there was a "false alarm" which "woke up" `wait`).

MULTITHREADING

Synchronization wrappers

- Older Java collections such as `Vector` are thread-safe.
- Newer collection types such as `HashMap` or `ArrayList` are not(!) thread-safe (i.e., parallel access by multiple threads can trigger a race condition), so you need to protect their access yourself using synchronized if they are a shared resource
- Reason: synchronization slows down things. Collections should work as fast as possible

MULTITHREADING

Synchronization wrappers

- In order to make access to such data structures thread-safe, you could create `synchronized` access methods for collections yourself.
- As an alternative, the Collections library provides *synchronization wrappers*. https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html
- E.g.,
  - `List arrayList = Collections.synchronizedList(new ArrayList());`
- This causes that (only!) the methods within these collections are protected by a lock on the respective collection object.
- But observe the next slide…

MULTITHREADING

- Problem with concurrency if you have something like this (or in principle any code which uses a shared resource):

```
if(list.size() < 10)
    list.add(newItem);
```

- … but no synchronization of the surrounding block or method.
- Obtaining `list` using `Collection.synchronizedList(…)` (or using `Vector`) is **not** sufficient here, because it only synchronizes the code within individual collection-methods (like add() or remove() of ArrayList), but not your own code which calls them
- Fixed code:

```
synchronized(list){
  if(list.size() < 10)
      list.add(newItem);
}
```