**MultiThreading:**

# Provide an example of threading and synchronization in Java

The best way to really understand threading and the need for synchronization is through a great example. Here we will present an example of an online banking system to really help see the potential problems with multi-threading, and their solutions through the use of a thread synchronization construct like a monitor.

|  |
| --- |
|  |

Let’s suppose we have an online banking system, where people can log in and access their account information. Whenever someone logs in to their account online, they receive a separate and unique thread so that different bank account holders can access the central system simultaneously.

Now let’s create a Java class to represent those individual bank accounts. Instances of this class are created when people actually log in online. Let’s name the class BankAccount. This class has a method called “deposit” that’s used to deposit funds into the bank account. This class also has another method called “transfer” to transfer funds from the bank to another account.

Here is some simple Java code that represents the BankAccount class:

public class BankAccount {

int accountNumber;

double accountBalance;

// to withdraw funds from the account

public boolean transfer (double amount)

{

double newAccountBalance;

if( amount > accountBalance)

{

//there are not enough funds in the account

return false;

}

else

{

newAccountBalance = accountBalance - amount;

accountBalance = newAccountBalance;

return true;

}

}

public boolean deposit(double amount)

{

double newAccountBalance;

if( amount < 0.0)

{

return false; // can not deposit a negative amount

}

else

{

newAccountBalance = accountBalance + amount;

accountBalance = newAccountBalance;

return true;

}

}

## Example of a race condition in Java

You've now seen the code above, but let's get into the problems that we can run into  
when we have a multi-threaded application. The problem that we will be presenting below is what's called a  
race condition. A race condition occurs when a program or application malfunctions because of an unexpected ordering of events that produces contention over the same resource. That sounds confusing, but it will make a lot more sense once you read the example below.

|  |
| --- |
|  |

So, let’s get into the actual problem. Let’s say that there’s a husband and wife - Jack and Jill - who share a joint account. They currently have $1,000 in their account. They both log in to their online bank account at the same time, but from different locations.

They both decide to deposit $200 each into their account through a wire transfer from other bank accounts that they have at the same time. So, the total account balance after these 2 deposits should be $1,000 + ($200 \* 2), which equals $1,400.

Let’s say Jill’s transaction goes through first, but Jill's thread of execution is switched out (to Jack’s transaction thread) right after executing this line of code in the deposit method:

newAccountBalance = accountBalance + amount;

Now, the processor is running the thread for Jack, who is also depositing $200 into their account. When Jack’s thread deposits $200, the account balance is still only $1,000, because the variable **accountBalance** has not yet been updated in Jill’s thread. Remember that Jill’s thread stopped execution right before the accountBalance variable was updated.

So, Jack’s thread runs until it completes the deposit function, and then updates the value of the accountBalance variable to $1200. After this, control returns to Jill’s thread, where newAccountBalance has the value of $1200. Then, it just assigns this value of $1,200 to accountBalance and returns. And that is the end of execution.

What is the result of these 2 deposits of $200? Well, the accountBalance variable ends up being set to only $1200, when it should have been $1400. This means Jack and Jill lost $200. This is good for the bank, but a huge problem for Jack and Jill, and any other of the bank's customers.

## The cause of the race condition

|  |
| --- |
|  |

So, do you see how the problem was caused here? Because Jill’s thread switched out (to Jack’s thread) right before the **accountBalance**variable was updated, Jill’s deposit was not counted.

If you remember the definition of a race condition, the example we just gave should clear it up. Here's the definition of a race condition again, in case you forgot: A race condition occurs when a program or application malfunctions because of an unexpected ordering of events that produces contention over the same resource. Hopefully, now it makes a lot more sense.

## Synchronization fixes race conditions in multi-threaded programs

But the real question is how can this problem be fixed? Well, it should be clear that the code needs to allow the deposit function to run to completion without switching to run a different thread. This is what synchronization is all about - fixing issues like this! This can be accomplished with a synchronization construct like a monitor.

## Example of using the synchronized keyword in Java

This problem is easily fixed in Java. In the code below, all we do is add the **synchronized**keyword to the transfer and deposit methods to create a monitor. If you need a refresher on monitors then you can read this article: [Monitors vs semaphores](http://www.programmerinterview.com/index.php/operating-systems/monitors-vs-semaphores/).

public class BankAccount {

int accountNumber;

double accountBalance;

// to withdraw funds from the account

public synchronized boolean transfer (double amount)

{

double newAccountBalance;

if( amount > accountBalance)

{

//there are not enough funds in the account

return false;

}

else

{

newAccountBalance = accountBalance - amount;

accountBalance = newAccountBalance;

return true;

}

}

public synchronized boolean deposit(double amount)

{

double newAccountBalance;

if( amount < 0.0)

{

return false; // can not deposit a negative amount

}

else

{

newAccountBalance = accountBalance + amount;

accountBalance = newAccountBalance;

return true;

}

}

## Synchronized keyword locks methods

What does the synchronized keyword do for us here? Well, if a thread is executing inside either the deposit or transfer blocks, then it is now impossible for any other threads to enter either of those methods. This means that only one thread can execute those functions at a time - which is exactly what we want to prevent the problem with the accountBalance variable that we described earlier.

First, it is not possible for two invocations of synchronized methods on the same object to interleave - so one thread can not interrupt another thread until it is done executing *all* of the code in a synchronized method. So, when one thread is executing a synchronized method all other threads are blocked from entering that method.

**=========================================**

**http://www.questiondiscussion.com/articles/738/threading-and-thread-synchronization-in-java-part-2**

[[QD Article] Threading and Thread Synchronization in Java - Part 2](http://www.questiondiscussion.com/articles/738/threading-and-thread-synchronization-in-java-part-2)

|  |  |
| --- | --- |
| **4**  **1** | In the QD Article - [Thread and Thread Synchronization in Java Part 1](http://www.questiondiscussion.com/articles/731/threading-and-thread-synchronization-in-java), I went through basic concepts of how Java supports multithreading, how to implement Threads using Threads class or Runnable interface, and how to achieve thread synchronization. This article is an extension to that, where I have tried to visit more involved concept, using a real life problem and by trying to implement a solution for that.  **Problem: (This problem was asked in an interview)** Implement a Queue in Java. Following conditions should be satisfied:   * Queue should have add and remove methods. * You should be able to share one Queue object across multiple threads (queue should act as a shared resource) * There should be a maximum size defined for the queue and it should print proper 'UNDERFLOW' or 'OVERFLOW' message on the console, in respective situations * In the scenario of UNDERFLOW/OVERFLOW, the current thread should be blocked and it should be unblocked (interrupted) only when another thread tries to access the queue.   **Java implementation of the above problem:**  Please go thorugh the output generated by this program (which I have pasted below), which will make things clearer.  import java.io.\*;  import java.util.\*;  import java.util.concurrent.atomic.\*;  /\*  \* This queue should have following properties  \*  \* - Fixed max size, containing Integers  \* - More than one thread can access it at a time, so the queue should be synchronized  \* - In case of overflow or underflow: block the current thread,  \* and this should be interrupted when a new thread tries to access the queue.  \*/  class MyQueue {  Vector<Integer> myqueue;  AtomicInteger counter;  Integer max\_size;  public MyQueue(int size){  myqueue = new Vector<Integer>(size);  // Also initialize the counter  counter = new AtomicInteger(0);  max\_size = new Integer(size);  }  public void printqueue(){  System.out.print("Queue size: " + counter.get() + ", ");  System.out.print("Current queue is -> ");  for(int i=0; i < counter.get(); i++){  System.out.printf("%d, ", myqueue.get(i));  }  System.out.println();  System.out.println("-------------------------");  }  public void add(int value){  synchronized (this){  System.out.println("Notifying the threads");  this.notifyAll();  }  // Check for overflow  if (counter.get() >= max\_size){  System.out.println("<OVERFLOW>. Blocking the thread");  try{  synchronized(this) {  this.wait();  }  }  catch (InterruptedException ex){  System.out.println("Failed to block thread: " + ex);  }  return;  }  // Else, add this number  myqueue.add(counter.get(), value);  counter = new AtomicInteger(counter.addAndGet(1)); // Increment counter  System.out.printf("Added the element - %d\n", value);  printqueue();  // Done  }  public Integer remove(){  synchronized (this){  System.out.println("Notifying the threads");  this.notifyAll();  }  // Check for underflow  if (counter.get() == 0){  System.out.println("<UNDERFLOW>. Blocking the thread");  try{  synchronized (this){  this.wait();  }  }  catch (InterruptedException ex){  System.out.println("Failed to block thread: " + ex);  }  return null;  }  // Else, get this number  Integer element = new Integer(myqueue.get(0));  // Reduce the counter  myqueue.remove(0);  counter = new AtomicInteger(counter.addAndGet(-1));  System.out.printf("Removed element - %d\n", element);  printqueue();  return element;  }  }  class HelloThread implements Runnable {  // Create a static queue, to be shared across threads  public static MyQueue myqueue = new MyQueue(4);  int curr\_size; // Specific to each thread  public HelloThread(int size){  curr\_size = size;  }  public void run(){  // Add data to queue  // Get data from queue  for(int i=0; i<curr\_size; i++){  System.out.printf("Thread ID - %d ==> ", Thread.currentThread().getId());  myqueue.add(i);  }  for(int i=0; i<curr\_size; i++){  System.out.printf("Thread ID - %d ==> ", Thread.currentThread().getId());  myqueue.remove();  }  }  }  public class SyncQueue {  public static void main(String args[]) throws IOException {  // Let's create two threads and run them  Thread t1 = new Thread(new HelloThread(6));  Thread t2 = new Thread(new HelloThread(4));  t1.start();  t2.start();  }  }  **Output generated by the program:**  Thread ID - 8 ==> Notifying the threads  Thread ID - 9 ==> Notifying the threads  Added the element - 0  Added the element - 0  Queue size: 2, Current queue is -> 0, 0,  -------------------------  Thread ID - 9 ==> Notifying the threads  Added the element - 1  Queue size: 3, Current queue is -> 0, 0, 1,  -------------------------  Thread ID - 9 ==> Notifying the threads  Added the element - 2  Queue size: 4, Current queue is -> 0, 0, 1, 2,  -------------------------  Thread ID - 9 ==> Notifying the threads  <OVERFLOW>. Blocking the thread  Queue size: 2, Current queue is -> 0, 0, 1, 2,  -------------------------  Thread ID - 8 ==> Notifying the threads  <OVERFLOW>. Blocking the thread  Thread ID - 9 ==> Notifying the threads  Removed element - 0  Queue size: 3, Current queue is -> 0, 1, 2,  -------------------------  Thread ID - 9 ==> Notifying the threads  Removed element - 0  Queue size: 2, Current queue is -> 1, 2,  -------------------------  Thread ID - 9 ==> Notifying the threads  Removed element - 1  Queue size: 1, Current queue is -> 2,  -------------------------  Thread ID - 9 ==> Notifying the threads  Removed element - 2  Queue size: 0, Current queue is ->  -------------------------  Thread ID - 8 ==> Notifying the threads  Added the element - 2  Queue size: 1, Current queue is -> 2,  -------------------------  Thread ID - 8 ==> Notifying the threads  Added the element - 3  Queue size: 2, Current queue is -> 2, 3,  -------------------------  Thread ID - 8 ==> Notifying the threads  Added the element - 4  Queue size: 3, Current queue is -> 2, 3, 4,  -------------------------  Thread ID - 8 ==> Notifying the threads  Added the element - 5  Queue size: 4, Current queue is -> 2, 3, 4, 5,  -------------------------  Thread ID - 8 ==> Notifying the threads  Removed element - 2  Queue size: 3, Current queue is -> 3, 4, 5,  -------------------------  Thread ID - 8 ==> Notifying the threads  Removed element - 3  Queue size: 2, Current queue is -> 4, 5,  -------------------------  Thread ID - 8 ==> Notifying the threads  Removed element - 4  Queue size: 1, Current queue is -> 5,  -------------------------  Thread ID - 8 ==> Notifying the threads  Removed element - 5  Queue size: 0, Current queue is ->  -------------------------  Thread ID - 8 ==> Notifying the threads  <UNDERFLOW>. Blocking the thread  **In this above program, you can see implementation of the following concepts:**   * **Access of Shared resource by multiple threads**: In the class *HelloThread*, I have defined myqueue as *static*, thus, it can be shared by both the threads t1 and t2 that we create in *SyncQueue*. * **Use of synchronous DS**: I have used *java.util.Vector* and *java.util.concurrent.atomic.AtomicInteger*. These are synchronous implementations and thus, two threads t1 and t2 can't modify them concurrently. * **Thread communication**: In *MyQueue.add* method, I have used a synchronized section to block current thread, in case of OVERFLOW, and similarly in *MyQueue.remove* method in case of UNDERFLOW.   When the non-blocked thread tries to access the same method, it calls *notifyAll()* method first, which is used to unblock all blocked threads.   * **Synchronous block**: For method calls *wait()* and *notifyAll()*, I have used *synchronous* block, so that the object myqueue (remember the *this* keyword?) is not attempted to be blocked or interrupted by multiple threads at a time. If not for synchronous block, I would have got following exception - *java.lang.IllegalMonitorStateException*   Please feel free to post any comments, in case of any query or suggestion. |