# [How to write Thread-Safe Code in Java](https://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html)

**thread-safety** or **thread-safe code in Java** refers to code which can safely be used or shared in concurrent or multi-threading environment and they will behave as expected. any code, class, or object which can behave differently from its contract on the concurrent environment is not thread-safe. thread-safety is one of the risks introduced by using [threads in Java](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) and I have seen java programmers and developers struggling to *write thread-safe code*or just understanding *what is thread-safe code* and what is not?  
  
  
This will not be a very detailed article on thread-safety or low-level details of [synchronization in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) instead we will keep it simple and focus on one example of non-thread-safe code and try to understand what is thread-safety and **how to make a code thread-safe**.

Btw, if you are serious about mastering Java multi-threading and concurrency then I also suggest you take a look at the [**Java Multithreading, Concurrency, and Performance Optimization**](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fjava-multithreading-concurrency-performance-optimization%2F) course by Michael Pogrebinsy on Udemy.   
  
It's an advanced course to become an expert in Multithreading, concurrency, and Parallel programming in Java with a strong emphasis on high performance.

## How to make Thread-Safe Code in Java

Before you learn how to write a thread-safe code you need to understand what is thread-safety and there is no better way to that than looking at a non-thread-safe code. So, let's see an example of the potential, not thread-safe code, and learn how to fix that.

### Example of Non-Thread-Safe Code in Java

Here is an example of a **non-thread-safe code**, look at the code, and find out *why this code is not thread-safe*?

/\*

**\* Non Thread-Safe Class in Java**

 \*/

public class **Counter** {

    private int count;

    /\*

     \* This method is not thread-safe because ++ is not an atomic operation

     \*/

    public int getCount(){

        return count++;

    }

}

**The above example is not thread-safe** because ++ (the increment operator) is not an **atomic operation** and can be broken down into reading, update, and write operations.  
  
If multiple thread call getCount() approximately same time each of these three operations may coincide or overlap with each other for example while thread 1 is updating value, thread 2 reads and still gets old value, which eventually let thread 2 override thread 1 increment and **one count is lost** because multiple threads called it concurrently.

You can further check [**Multithreading and Parallel Computing in Java**](https://click.linksynergy.com/fs-bin/click?id=JVFxdTr9V80&subid=0&offerid=323058.1&type=10&tmpid=14538&RD_PARM1=https%3A%2F%2Fwww.udemy.com%2Fmultithreading-and-parallel-computing-in-java%2F) to learn more about essential multithreading concepts like atomicity and compound operation. It's a good course to learn multithreading in JAva.

## How to make code Thread-Safe in Java

There are multiple ways to make this code thread-safe in Java:

1) Use the [synchronized keyword in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) and lock the getCount() method so that only one thread can execute it at a time which removes the possibility of coinciding or interleaving.

2) use **Atomic Integer**, which makes this ++ operation [atomic](https://javarevisited.blogspot.com/2020/04/difference-between-atomic-volatile-and-synchronized-in-java-multi-threading.html) and since **atomic operations are thread-safe** and saves the cost of external synchronization.

Here is a thread-safe version of Counter class in Java:

/\*

 \* **Thread-Safe Example in Java**

 \*/

public class Counter {

    private int count;

    AtomicInteger atomicCount = new AtomicInteger( 0 );

    /\*

     \* **This method thread-safe now because of locking and synchornization**

     \*/

    public synchronized int getCount(){

        return count++;

    }

    /\*

     \* **This method is thread-safe because count is incremented atomically**

     \*/

    public int getCountAtomically(){

        return atomicCount.incrementAndGet();

    }

}

And, if you are serious about mastering Java multi-threading and concurrency then I also suggest you take a look at the [**Java Multithreading, Concurrency, and Performance Optimization**](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fjava-multithreading-concurrency-performance-optimization%2F) course by Michael Pogrebinsy on Udemy.   
  
This is an advanced course to become an expert in Multithreading, concurrency, and Parallel programming in Java with a strong emphasis on high performance.

[](https://click.linksynergy.com/deeplink?id=JVFxdTr9V80&mid=39197&murl=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fjava-multithreading-concurrency-performance-optimization%2F)

## Important points about Thread-Safety in Java

Here are some points worth remembering to **write thread-safe code in Java**, this knowledge also helps you to avoid some serious concurrency issues in Java-like race condition or [deadlock in Java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html):

1) Immutable objects are by default thread-safe because their state can not be modified once created. Since String is immutable in Java, it's inherently thread-safe.

2) Read-only or [final variables in Java](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html) are also thread-safe in Java.

3) Locking is one way of achieving thread-safety in Java.

4) [Static variables](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) if not synchronized properly become a major cause of thread-safety issues.

5) Example of thread-safe class in Java: Vector, Hashtable, [ConcurrentHashMap](https://www.java67.com/2020/02/25-examples-of-concurrenthashmap-in-java.html), [String](https://www.java67.com/2018/06/top-35-java-string-interview-questions.html), etc.

6) Atomic operations in Java are thread-safe like reading a 32-bit int from memory because it's an atomic operation it can't interleave with other threads.

7) local variables are also thread-safe because each thread has there own copy and using local variables is a good way to write thread-safe code in Java.

8) In order to avoid thread-safety issues minimize the sharing of objects between multiple threads.

9) [Volatile keyword in Java](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html) can also be used to instruct thread not to cache variables and read from main memory and can also instruct JVM not to reorder or optimize code from threading perspective.

[how to write thread-safe code in java example](http://javarevisited.blogspot.com/2011/11/run-java-program-from-command-prompt.html)That’s all on **how to write thread-safe class or code in Java** and avoid serious concurrency issues in Java. To be frank, thread-safety is a little tricky concept to grasp, you need to think concurrently in order to catch whether a code is thread-safe or not.   
  
Also, [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) plays a spoiler since it can **reorder code** for optimization, so the code which looks sequential and runs fine in the development environment not guaranteed to run similarly in the production environment because JVM may ergonomically adjust itself as server JVM and perform more optimization and reorder which cause **thread-safety issues**.

**Q How to create Immutable Class and Object in Java - Tutorial Example**

Writing or creating immutable classes in Java is becoming popular day by day, because of concurrency and multithreading advantage provided by immutable objects. Immutable objects offer several benefits over a conventional mutable object, especially while creating concurrent Java applications. **Immutable object not only guarantees safe publication of object’s state but also can be shared among other threads without any external synchronization**. In fact, JDK itself contains several immutable classes like String, Integer, and other wrapper classes. EX. java.lang.String, once created can not be modified like trim, uppercase, lowercase.

**immutable class in Java –**  Immutable classes are those classes, whose object can not be modified once created, it means any modification on an immutable object will result in another immutable object. the best example to understand immutable and mutable objects are, String and StringBuffer.   
  
Since A string is an immutable class, any change on existing string object will result in another string e.g. replacing a character into String, creating substring from String, all result in new objects. While in case of mutable object like StringBuffer, any modification is done on object itself and no new objects are created. Some times this immutability of String can also cause security hole, and that the reason why password should be stored on char array instead of String.

**How to write immutable class in Java:**

Despite a few disadvantages, an Immutable object still offers several benefits in multi-threaded programming and it’s a great choice to achieve thread safety in Java code. here are few rules, which helps to make a class immutable in Java :

1. The state of immutable object can not be modified after construction, any modification should result in a new immutable object.

2. **All fields of the Immutable class should be final.**

3. Object must be properly constructed i.e. object reference must not leak during construction process.

4. **Object should be final in order to restrict sub-class for altering immutability of parent class.**

By the way, we can still create an immutable object by violating few rules, like String has its hashcode in the non final field, but its always guaranteed to be the same. No matter how many times we calculate it because it’s calculated from final fields, which is guaranteed to be the same.   
This required a deep knowledge of Java memory model, and can create subtle race conditions if not addressed properly. In next section we will see simple example of writing immutable class in Java. By the way, if our Immutable class has lots of optional and mandatory fields, then we can also use Builder design pattern to make a class Immutable in Java.

Immutable Class Example in Java

Here is complete code example of writing immutable class in Java. We have followed simplest approach and all rules for making a class immutable, including it making class final to avoid putting immutability at risk due to Inheritance and Polymorphism.

public **final** **class** Contacts {

    private **final String** name;

    private **final** **String** mobile;

    public Contacts(String name, String mobile) {

        this.name = name;

        this.mobile = mobile;

    }

    public String getName(){

        return name;

    }

    public String getMobile(){

        return mobile;

    }

}

This Java class is immutable, because its state can not be changed once created. **We can see that all of it’s fields are final.** This is one of the most simple way of creating immutable class in Java, where all fields of class also remains immutable like String in above case. Some time we may need to write immutable class which includes mutable classes like java.util.Date, despite storing Date into final field it can be modified internally, if internal date is returned to the client. In order to preserve immutability in such cases, its advised to return copy of original object, which is also one of the Java best practice. here is another example of making a class immutable in Java, which includes mutable member variable.

**public final class ImmutableReminder**{

    private **final Date** remindingDate;

    public ImmutableReminder (Date remindingDate) {

        if(remindingDate.getTime() < System.currentTimeMillis()){

            throw new IllegalArgumentException("Can not set reminder” +

                        “ for past time: " + remindingDate);

        }

        this.remindingDate = new Date(remindingDate.getTime());

    }

    public Date getRemindingDate() {

        return (Date) remindingDate.clone();

    }

}

In above example of creating immutable class, Date is a mutable object. If **getRemindingDate()** returns actual Date object than despite remindingDate being final variable, internals of Date can be modified by client code. By returning clone() or copy of remindingDate, we avoid that danger and preserves immutability of class.

Benefits of Immutable Classes in Java

As I said earlier Immutable classes offers several benefits, here are few to mention:

1) Immutable objects are by default thread safe, can be shared without synchronization in concurrent environment.

2) An immutable object simplifies development, because its easier to share between multiple threads without external synchronization.

3) Immutable object boost performance of Java application by reducing synchronization in code.  
4) Another important benefit of Immutable objects is reusability, we can cache Immutable object and reuse them, much like String literals and Integers.  We can use static factory methods to provide methods like valueOf(), which can return an existing Immutable object from cache, instead of creating a new one.

Apart from the above advantages, immutable object has disadvantage of creating garbage as well. Since the immutable objects can not be reused and they are just a use and throw. String being a prime example, which can create lot of garbage and can potentially slow down application due to heavy garbage collection, but again that's extreme case and if used properly Immutable object adds lot of value.

That's all on how to write immutable class in Java. we have seen rules of writing immutable classes, benefits offered by immutable objects and how we can create immutable class in Java which involves mutable fields. Don’t forget to read more about concurrency benefit offered by Immutable object in one of the best Java book recommended to Java programmers, Concurrency Practice in Java.

**Q.) Can we make array volatile in Java?**

**Ans**. **Yes**, we can make an array volatile in Java but only the reference which is pointing to an array, not the whole array. What I mean, if one thread changes the reference variable to points to another array, that will provide a volatile guarantee, but if multiple threads are changing individual array elements they won't be having happens before guarantee provided by the volatile modifier.

**Q.) Can volatile make a non-atomic operation to atomic?**

This question is also not easy to answer because volatile is not about atomicity, but there are cases where we can use a volatile variable to make the operation atomic.

One example I have seen is having a long field in our class. If we know that a long field is accessed by more than one thread e.g. a counter, a price field or anything, we better make it volatile. Why? because reading to a long variable is not atomic in Java and done in two steps, If one thread is writing or updating long value, it's possible for another thread to see half value (fist 32-bit). While reading/writing a volatile long or double (64 bit) is atomic.

**Q) What are practical uses of volatile modifier?**

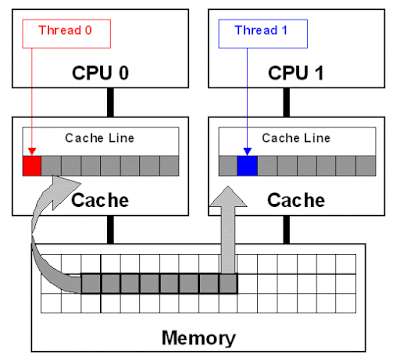
One of the practical use of the volatile variable is to make reading double and long atomic. Both double and long are 64-bit wide and they are read in two parts, first 32-bit first time and next 32-bit second time, which is non-atomic but volatile double and long read is atomic in Java. Another use of the volatile variable is to provide a memory barrier, just like it is used in Disrupter framework. Basically, Java Memory model inserts a write barrier after we write to a volatile variable and a read barrier before we read it. Which means, if we write to volatile field then it's guaranteed that any thread accessing that variable will see the value we wrote and anything we did before doing that right into the thread is guaranteed to have happened and any updated data values will also be visible to all threads, because the memory barrier flushed all other writes to the cache.

**Q. Which one would be easy to write? synchronization code for 10 threads or 2 threads?**

In terms of writing code, both will be of same complexity because synchronization code is independent of a number of threads. Choice of synchronization though depends upon a number of threads because the number of thread present more contention, so we go for advanced synchronization technique e.g. lock stripping, which requires more complex code and expertise.

**Q.) What is false sharing in the context of multi-threading?**

false sharing is one of the well-known performance issues on multi-core systems, where each process has its local cache. false sharing occurs when threads on different processor modify variables that reside on same cache line as shown in the following image:

[](https://2.bp.blogspot.com/-Tze9foqpb74/VepwCzXHGCI/AAAAAAAADtM/i4KQDaefqk4/s1600/False%2BSharing%2Bin%2BMulti-threaded%2Bapplication.gif)

False sharing is very hard to detect because the thread may be accessing completely different global variables that happen to be relatively close together in memory. Like many concurrency issues, the primary way to avoid false sharing is careful code review and aligning our data structure with the size of a cache line.  
  
**Q.) What is busy spin? Why should we use it?**Busy spin is one of the technique to wait for events without releasing CPU. It's often done to avoid losing data in CPU cached which is lost if the thread is paused and resumed in some other core. So, if we are working on low latency system where our order processing thread currently doesn't have any order, instead of sleeping or calling wait(), we can just loop and then again check the queue for new messages. It's only beneficial if we need to wait for a very small amount of time e.g. in micro seconds or nano seconds. LMAX Disrupter framework, a high-performance inter-thread messaging library has a BusySpinWaitStrategy which is based on this concept and uses a busy spin loop for EventProcessors waiting on the barrier.  
  
**Q.) How do we take thread dump in Java?**We can take a thread dump of Java application in Linux by using kill -3 PID, where PID is the process id of Java process. In Windows, we can press Ctrl + Break. This will instruct JVM to print thread dump in standard out or err and it could go to console or log file depending upon our application configuration. If we have used Tomcat then when  
  
  
**Q) Can we create an Immutable object, which contains a mutable object?**Yes, its possible to create an Immutable object which may contain a mutable object, we just need to be a little bit careful not to share the reference of the mutable component, instead, we should return a copy of it if we have to. Most common example is an Object which contain the reference of java.util.Date object.  
  
**Q.What is Volatile Variable in Java - When to use in Thread? Example**

What is Volatile variable in Java  
volatile variable in Java is a special variable which is used to signal threads, a compiler that this particular variables value are going to be updated by multiple threads inside Java application. By making a variable volatile using the volatile keyword in Java, application programmer ensures that its value should always be read from main memory and thread should not use cached value of that variable from their own stack. With the introduction of Java memory model from Java 5 onwards along with introduction of CountDownLatch, CyclicBarrier, Semaphore and ConcurrentHashMap, volatile variable also guarantees "happens-before" relationship, which means not only another thread has visibility of latest value of volatile variable but also all the variable is seen by the thread which has updated value of volatile variable before these threads sees it. What is volatile variable and when to use it is always a popular Java threading question.  
Important point related to volatile keyword in Java  
Since volatile keyword is used to make any variable volatile in Java environment, it's good to know more about What is a volatile keyword, what is its limitation and How to use the volatile keyword in Java.

The volatile keyword can only be applied to a variable, it can not be applied to class or method. using volatile keyword along with class and method is a compiler error.

A volatile is also referred as modifier in Java.

When to use Volatile variable in Java  
This is the most important thing to learn while learning about the volatile variable in Java. When to use volatile variable in Java is also a famous multi-threading interview question in Java. here are some of the scenario where we can use volatile variable in Java :  
  
1) Any variable which is shared between multiple threads should be made variable, in order to ensure that all thread must see the latest value of the volatile variable.  
2) A signal to compiler and JIT to ensure that compiler does not change ordering or volatile variable and moves them out of synchronized context.  
3) We want to save the cost of synchronization as volatile variables are less expensive than synchronization.  
That's all on What is volatile variable in Java When to use volatile variable in Java. a volatile variable is an important concept to understand and use. It's also very important in terms of  Java interview point of view.

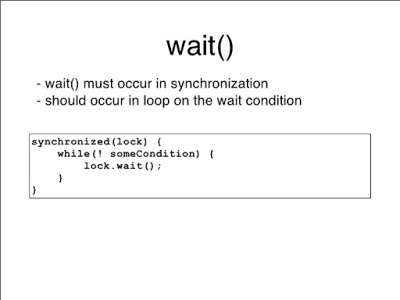
How to use wait, notify and notifyAll in Java - Producer Consumer Example

We can use wait, notify, and notifyAll methods to communicate between threads in Java. For example, if we have two threads running in our program e.g.Producer and Consumer then producer thread can communicate to the consumer that it can start consuming now because there are items to consume in the queue. Similarly, a consumer thread can tell the producer that it can also start putting items now because there is some space in the queue, which is created as a result of consumption. A thread can use the wait() method to pause and do nothing depending upon some condition. For example, in the producer-consumer problem, producer thread should wait if the queue is full and consumer thread should wait if the queue is empty.  
If some thread is waiting for some condition to become true, we can use notify and notifyAll methods to inform them that condition is now changed and they can wake up.  
Both notify() and notifyAll() method sends a notification but notifies sends the notification to only one of the waiting thread, no guarantee which thread will receive notification and notifyAll() sends the notification to all threads.  
So if only one thread is waiting for an object lock, also known as a monitor then both notify and notifyAll will send the notification to it. If multiple threads are waiting on a monitor then notify will only inform one of the lucky threads and rest will not receive any notification, but notifyAll will inform all threads.  
In this Java multi-threading tutorial we will learn how to use wait, notify and notifyAll() method in Java to implement inter-thread communication by solving the producer-consumer problem.  
And, if we are serious about mastering Java multi-threading and concurrency then I also suggest we take a look at the Java Multithreading, Concurrency, and Performance Optimization course by Michael Pogrebinsy on Udemy. It's an advanced course to become an expert in Multithreading, concurrency, and Parallel programming in Java with a strong emphasis on high performance

**Q.How to use wait and notify in code**

Even though wait and notify are quite a fundamental concept and they are defined in the object class, surprisingly, it's not easy to write code using wait and notify. We can test this during an interview by asking the candidate to write code to solve producer-consumer problem using wait and notify by hand.

I am sure many will be stuck and make mistakes e.g. synchronizing at the wrong place, not calling wait on a right object or not following standard idiom. To be honest its confusing for non-regular coders.  
  
First confusion arise from the fact, how to call wait() method? Since wait method is not defined in Thread class, we cannot simply call Thread.wait(), that won't work but since many Java developers are used to calling Thread.sleep() they try the same thing with wait() method and stuck.  
We need to call wait() method on the object which is shared between two threads, in producer-consumer problem its the queue which is shared between producer and consumer threads.  
The second confusion comes from the fact that wait needs to be a call from synchronized block or method? So if we use synchronized block, which object should be put to go inside the block? This should be the same object, whose lock we want to acquire i.e. the shared object between multiple threads. In our case it's queue.

[](https://4.bp.blogspot.com/-REiHbzOXAwM/VZPzn9IOM5I/AAAAAAAADOw/6oX4j4aBUno/s1600/wait%2Bmethod%2Bin%2BJava%2B2.jpg)

Always call wait and notify from Loop Instead of If Block

Once we know that we need to call wait from synchronized context and on the shared object, next thing is to avoid mistake made by several Java developer by calling wait() method inside if block instead of while loop.  
Since we call wait inside a conditional block e.g. producer thread should call wait() if queue is full, first instinct goes towards using if block, but calling wait() inside if block can lead to subtle bugs because it's possible for thread to wake up spuriously even when waiting condition is not changed.  
If we don't check the condition again after waking up by using a loop, we will take the wrong action which may cause problem e.g. trying to insert item on a full queue or trying to consume from an empty queue. That's why we should always call wait and notify method from a loop and not from if block.

Based upon above knowledge here is  the standard code template or idiom to call wait and notify method in Java :

// The standard idiom for calling the wait method in Java

synchronized (sharedObject) {

while (condition) {

sharedObject.wait(); // (Releases lock, and reacquires on wakeup)

}

... // do action based upon condition e.g. take or put into queue

}

As I suggested, we should always invoke wait method from a loop. The loop is used to test the condition before and after waiting. If the condition still holds and the notify (or notifyAll) method has already been invoked before a thread calls wait() method, then there is no guarantee that the thread will ever awake from the wait, potentially causing a deadlock.  
  
BTW, if we are serious about mastering concurrency and multi-threading, I strongly suggest we to read **Java Concurrency in Practice** by Brian Goetz, without reading that book our journey to Java multi-threading is not complete. It's probably one of the most recommended books to Java developers.

## Java wait(), notify() and notifyAll() Example

Here is our sample program to demonstrate how to use wait and notify method in Java. In this program, we have used the standard idiom discussed above to call wait(), notify() and notifyAll() method in Java.  
In this program, we have two threads named PRODUCER and CONSUMER and implemented using Producer and Consumer class which extends Thread class. The logic of what producer and the consumer should do is written in their respective run() method.  
Main thread starts both producer and consumer threads and also create an object of LinkedList class to share as Queue between them. If we don't know LinkedList also implements Queue interface in Java.  
Producer runs in an infinite loop and keeps inserting random integer value into Queue until the queue is full. We check this condition at while(queue.size == maxSize), remember before doing this check we synchronize on queue object so that no other thread modify the queue when we are doing this check.  
If Queue is full then our PRODUCER thread waits until CONSUMER thread consume one item and make space in our queue and call notify() method to inform PRODUCER thread. **Both wait() and notify() method are called on shared object which is queue** in our case.

import java.util.LinkedList;

import java.util.Queue;

import java.util.Random;

/\*\*

\* Simple Java program to demonstrate How to use wait, notify and notifyAll()

\* method in Java by solving producer consumer problem.

\*

\* @author Javin Paul

\*/

public class ProducerConsumerInJava {

public static void main(String args[]) {

System.out.println("How to use wait and notify method in Java");

System.out.println("Solving Producer Consumper Problem");

Queue<Integer> buffer = new LinkedList<>();

int maxSize = 10;

Thread producer = new Producer(buffer, maxSize, "PRODUCER");

Thread consumer = new Consumer(buffer, maxSize, "CONSUMER");

producer.start();

consumer.start();

}

}

/\*\*

\* Producer Thread will keep producing values for Consumer

\* to consumer. It will use wait() method when Queue is full

\* and use notify() method to send notification to Consumer

\* Thread.

\*

\* @author WINDOWS 8

\*

\*/

class Producer extends *Thread* {

private Queue<Integer> queue;

private int maxSize;

public Producer(Queue<Integer> queue, int maxSize, String name){

super(name);

this.queue = queue;

this.maxSize = maxSize;

}

@Override

public void run() {

while (true) {

synchronized (queue) {

while (queue.size() == maxSize) {

try {

System.out .println("Queue is full, "

+ "Producer thread waiting for "

+ "consumer to take something from queue");

queue.wait();

} catch (Exception ex) {

ex.printStackTrace();

}

}

Random random = new Random();

int i = random.nextInt();

System.out.println("Producing value : " + i);

queue.add(i);

queue.notifyAll();

}

}

}

}

/\*\*

\* Consumer Thread will consumer values form shared queue.

\* It will also use wait() method to wait if queue is

\* empty. It will also use notify method to send

\* notification to producer thread after consuming values

\* from queue.

\*

\* @author WINDOWS 8

\*

\*/

class Consumer extends *Thread* {

private Queue<Integer> queue;

private int maxSize;

public Consumer(Queue<Integer> queue, int maxSize, String name){

super(name);

this.queue = queue;

this.maxSize = maxSize;

}

@Override

public void run() {

while (true) {

synchronized (queue) {

while (queue.isEmpty()) {

System.out.println("Queue is empty,"

+ "Consumer thread is waiting"

+ " for producer thread to put something in queue");

try {

queue.wait();

} catch (Exception ex) {

ex.printStackTrace();

}

}

System.out.println("Consuming value : " + queue.remove());

queue.notifyAll();

}

}

}

}

Output

How to use wait and notify method in Java

Solving Producer Consumper Problem

Queue is empty,Consumer thread is waiting for producer thread to put something in queue

Producing value : -1692411980

Producing value : 285310787

Producing value : -1045894970

Producing value : 2140997307

Producing value : 1379699468

Producing value : 912077154

Producing value : -1635438928

Producing value : -500696499

Producing value : -1985700664

Producing value : 961945684

Queue is full, Producer thread waiting for consumer to take something from queue

Consuming value : -1692411980

Consuming value : 285310787

Consuming value : -1045894970

Consuming value : 2140997307

Consuming value : 1379699468

Consuming value : 912077154

Consuming value : -1635438928

Consuming value : -500696499

Consuming value : -1985700664

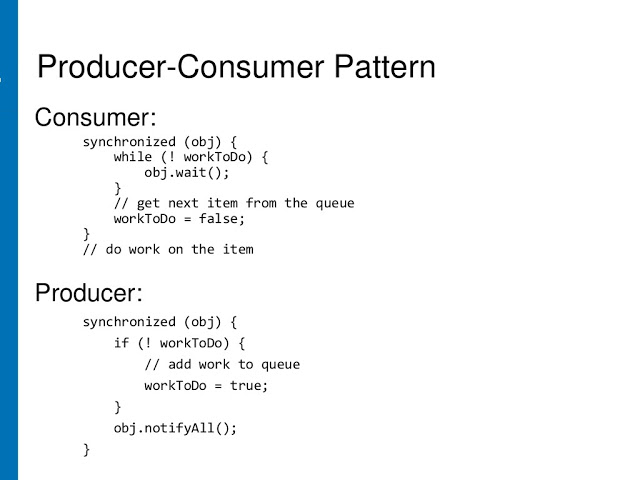
Consuming value : 961945684

Queue is empty,Consumer thread is waiting for producer thread to put something in queue

Producing value : 1182138498

In order to understand this program better, I suggest we debug it instead of running. Once we start our program in debug mode it will stop at either PRODUCER or CONSUMER thread, depending upon which one thread scheduler chose to give CPU.  
Since both threads have wait condition they will go there, now we just run it and see what it does, it will most likely print the output shown above. We can even use Step Into and Step Over buttons in Eclipse to run the program step by step to understand it better.  
Things to Remember about Using wait(), notify() and notifyAll() method

1. We can use wait() and notify() method to implement inter-thread communication in Java. Not just one or two threads but multiple threads can communicate to each other by using these methods.
2. Always call wait(), notify() and notifyAll() methods from synchronized method or synchronized block otherwise JVM will throw IllegalMonitorStateException.
3. Always call wait and notify method from a loop and never from if() block, because loop test waiting condition before and after sleeping and handles notification even if waiting for the condition is not changed.
4. Always call wait in shared object e.g. shared queue in this example.
5. Prefer notifyAll() over notify() method due to reasons given in this article.

[](https://3.bp.blogspot.com/-icSDvcAustg/VZP05Fy5RUI/AAAAAAAADO8/G4zbwhOMiy4/s1600/Producer%2BConsumer%2Bpatern%2Busing%2Bwait%2Bnotify%2Bjava.jpg)

That's all about **how to use wait, notify and notifyAll() method in Java**. We should use wait and notify for inter-thread communication in Java only if we know what we are doing otherwise there are many high-level concurrency utilities available for the different task.

Swing is not Thread-safe in Java - What Does it Mean? Event Dispatcher, SwingWorker, Multithreading and Best Practices

A couple of my readers ask questions, what does it mean by Swing is not thread-safe and How does it affect coding in Swing GUI application? This post is an effort to help those readers and several other programmers to understand Swing and thread-safety in a bit more detailed way. To keep it simple, let's revise what does it mean by being thread-safe? We say an object is thread-safe if we can call its a method, which can change its states, from multiple threads at the same time. To give we an example, java.lang.String is a thread-safe class, which means we can call any method e.g. substring(), toUpperCase() or toLowerCase() from multiple threads.  
  
By the way, String is a thread-safe because it's immutable. Let's come back to Swing now, The Core part of Swing is made up of different GUI component e.g. JLable, JPanel, JCombobox, etc. All these components are not thread-safe, which means we can not call methods of these components like the JLable.setText("new title") from any thread, other than Event Dispatcher Thread(EDT).   
  
In one word, Since Swing is not thread-safe, we can not update Swing components from any random thread, they are always updated using Event Dispatcher thread. This is, in fact, one of the most popular Java Swing Interview Question, with lot's of interesting follow-up e.g.   
  
If Swing is not thread-safe than how do we update components from other threads? and which methods of Swing API are thread-safe? etc. We will see the answer to this question in the next section.  
  
And, if we are new to Java Swing and GUI Programming world then I also recommend we go through Java Swing (GUI) Programming: From Beginner to Expert course by John Purcell on Udemy to learn Java in a better and more structured way. This is one of the best and up-to-date courses to learn Java Swing and GUI coding online.

How to update Swing Component from a different thread

Since Swing is not thread-safe by design, it's designer did provide couple of utility methods in SwingUtilities class to update any Swing component from a thread other thread Event Dispatcher Thread. We can use invokeAndWait() and invokeLater() to update a Swing component from any arbitrary thread.   
  
As the name suggests, invokeAndWait() is synchronous, blocking method and blocks until GUI is updated, while invokeLater() is an asynchronous call and doesn't wait for GUI to be updated. By the way, both of these methods make sure that Swing GUI components are updated in EDT thread only. Both of these methods takes a Runnable object, which contains code to update GUI, as shown below :

SwingUtilities.invokeLater(new Runnable() {

  public void run() {

    JLable.setText("Update Title");

  }

}

to learn more about both of these methods and How to use them, See when to use InvokeAndWait and InvokeLater in Java.

A couple of times, the Interviewer also asks about, how do we find, if a particular thread is Event Dispatcher thread or not? Well, if we are familiar with Swing API, they we may know that SwingUtilities provides isEventDispatchThread() method, which can be used to find out if the current thread is Event Dispatcher thread or not.   
  
Another follow-up question is Why Swing is not thread-safe in Java? Well, this is on similar lines like why multiple inheritance is not supported in Java or Why Java doesn’t support operator overloading.   
  
It's the decision taken by there designer, at that time. Since making an API thread-safe takes a lot of work, and it's often based upon the benefit we get. Since GUI screens are mostly updated in the response of user action e.g. when users click a button, and since events are handled in the same Event dispatcher thread, it's easy to update GUI on that thread. It's very rare when an update request for GUI comes from a different thread e.g. maybe once a network request is complete or a file is loaded.   
  
In such cases, we can use either invokeAndWait(), invokeLater(), or preferably SwingWorker class, which is designed to run these lengthy time-consuming tasks in a separate thread and same time managing inter-thread communication between a worker thread and Event dispatcher thread.   
  
Since EDT thread is most important in Swing and responsible for listening event and updating GUI, we must not do any time-consuming task on it, otherwise, we risk our application to become unresponsive or frozen. Another key thing to note, while using multi-threading in Swing development is that, not only GUI components but there model e.g. TableModel for JTable, must be updated in Event Dispatcher thread. One of the crucial difference between AWT and Swing is that AWT components are thread-safe

Summary

Here is a summary of our discussion, which is also a list of Java Swing best practices in a multithreading environment. When we use threads in Swing development, we risk of deadlock and frozen GUI. By following these simple rules, we minimize our chances of errors.

1) Since Swing components are not thread-safe, until any specifics mentioned in Javadoc, they must be created and modified from AWT Event Dispatcher thread.

2) Not only Swing components but also there model e.g. ListModel, TableModel must be modified from Event Dispatcher thread.

3) AWT Event Dispatcher thread is responsible for updating GUI and listening events, so don't block them by doing time-consuming task there.

4) We can use java.awt.EventQueue's isDispatchThread() or SwingUtilities.isEventDispatchThread() to check if current thread is EDT.

5) Use SwingWorker to perform a lengthy time-consuming tasks in a worker thread.

6) Use InvokeAndWait() and invokeLater() to update GUI components from threads other than Event Dispatcher thread.

That's all on Why Swing is not thread-safe in Java, more important what does it mean by Swing is not thread-safe. We have also touched base on Event Dispatcher thread, SwingWorker, invokeAndWait, and invokeLater. This is an extremely important topic from the Swing development and Interview point of view. I have never seen a Java Swing Interview, without any question from threading.

**ThreadLocal in Java - Example Program and Tutorial**

ThreadLocal in Java is another way to achieve thread-safety apart from writing immutable classes. If we have been writing multi-threaded or concurrent code in Java then we must be familiar with the cost of synchronization or locking which can greatly affect the Scalability of application, but there is no choice other than synchronizing if we are sharing objects between multiple threads. ThreadLocal in Java is a different way to achieve thread-safety, it doesn't address synchronization requirement, instead, it eliminates sharing by providing an explicit copy of Object to each thread. Since Object is no more shared there is no requirement of Synchronization which can improve scalability and performance of the application.  
  
In this Java ThreadLocal tutorial, we will see important points about ThreadLocal in Java, when to use ThreadLocal in Java, and a simple example of ThreadLocal in Java program.  
  
And, if we are serious about mastering Java multi-threading and concurrency then I also suggest we take a look at the Java Multithreading, Concurrency, and Performance Optimization course by Michael Pogrebinsy on Udemy. It's an advanced course to become an expert in Multithreading, concurrency, and Parallel programming in Java with a strong emphasis on high performance

**Q. When to use ThreadLocal in Java**

Many Java Programmers question where to use ThreadLocal in Java and some even argue the benefit of the ThreadLocal variable, but ThreadLocal has many genuine use cases and that's why it's added to the standard Java Platform Library. I agree though until we are not in concurrent programming, we will rarely use ThreadLocal. below are some well know usage of ThreadLocal class in Java:

1) ThreadLocal is fantastic to implement Per Thread Singleton classes or per thread context information like transaction id.

2) We can wrap any non Thread Safe object in ThreadLocal and suddenly its uses become Thread-safe, as it's only being used by Thread Safe. One of the classic examples of ThreadLocal is sharing SimpleDateForamt. Since SimpleDateFormat is not thread-safe, having a global formatter may not work but having per Thread formatter will certainly work.

3) ThreadLocal provides another way to extend Thread. If we want to preserve or carry information from one method call to another we can carry it by using ThreadLocal. This can provide immense flexibility as we don't need to modify any method.

On a basic level, ThreadLocal provides Thread Confinement which is an extension of the local variable. while the local variable is only accessible on the block they are declared, ThreadLocal is visible only in Single Thread.

No two Thread can see each other's ThreadLocal variable. A real-Life example of ThreadLocal is in J2EE application servers which uses java ThreadLocal variable to keep track of transaction and security Context. It makes a lot of sense to share heavy objects like Database Connection as ThreadLocal in order to avoid excessive creation and cost of locking in case of sharing global instance.

Java ThreadLocal Example – Code

import java.io.IOException;  
import java.text.DateFormat;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
/\*\*  
 \*  
 \* @author  
 \*/  
public class ThreadLocalTest {  
  
    public static void main(String args[]) throws IOException {  
        Thread t1 = new Thread(new Task());    
        Thread t2 = new Thread( new Task());  
       
        t1.start();  
        t2.start();        
       
    }  
     
    /\*  
     \* Thread safe format method because every thread will use its own DateFormat  
     \*/  
    public static String threadSafeFormat(Date date){  
        DateFormat formatter = PerThreadFormatter.getDateFormatter();  
        return formatter.format(date);  
    }  
     
}  
  
  
/\*  
 \* Thread Safe implementation of SimpleDateFormat  
 \* Each Thread will get its own instance of SimpleDateFormat which will not be shared between other threads. \*  
 \*/  
class PerThreadFormatter {  
  
    private static final ThreadLocal<SimpleDateFormat> dateFormatHolder = new ThreadLocal<SimpleDateFormat>() {  
  
        /\*  
         \* initialValue() is called  
         \*/  
        @Override  
        protected SimpleDateFormat initialValue() {  
            System.out.println("Creating SimpleDateFormat for Thread : " + Thread.currentThread().getName());  
            return new SimpleDateFormat("dd/MM/yyyy");  
        }  
    };  
  
    /\*  
     \* Every time there is a call for DateFormat, ThreadLocal will return calling  
     \* Thread's copy of SimpleDateFormat  
     \*/  
    public static DateFormat getDateFormatter() {  
        return dateFormatHolder.get();  
    }  
}  
  
class Task implements Runnable{  
     
    @Override  
    public void run() {  
        for(int i=0; i<2; i++){  
            System.out.println("Thread: " + Thread.currentThread().getName() + " Formatted Date: " + ThreadLocalTest.threadSafeFormat(new Date()) );  
        }        
    }  
}  
  
Output:  
Creating SimpleDateFormat for Thread: Thread-0  
Creating SimpleDateFormat for Thread: Thread-1  
Thread: Thread-1 Formatted Date: 30/05/2012  
Thread: Thread-1 Formatted Date: 30/05/2012  
Thread: Thread-0 Formatted Date: 30/05/2012  
Thread: Thread-0 Formatted Date: 30/05/2012

If we look at the output of the above program then we will find that when a different thread calls getFormatter() method of ThreadLocal class than its call its initialValue() method which creates an exclusive instance of SimpleDateFormat for that Thread.   
  
Since SimpleDateFormat is not shared between thread and essentially local to the thread which creates its own  threadSafFormat() method is completely thread-safe.

We can further join an advanced course like Concurrency, Multithreading, and Parallel Computing in Java by Holczer Balazes on Udemy to learn advanced concurrency concepts like fork-join pool, map-reduce, and parallel programming. 

Important points on Java ThreadLocal Class

1. ThreadLocal in Java is introduced on JDK 1.2 but it later generified in JDK 1.4 to introduce type safety on ThreadLocal variable.

2. ThreadLocal can be associated with Thread scope, all the code which is executed by Thread has access to ThreadLocal variables but two thread can not see each other's ThreadLocal variable.

3. Each thread holds an exclusive copy of the ThreadLocal variable which becomes eligible to Garbage collection after thread finished or died, normally or due to any Exception, Given those ThreadLocal variable doesn't have any other live references.

4. ThreadLocal variables in Java are generally private static fields in Classes and maintain its state inside Thread.

We saw how ThreadLocal in Java opens another avenue for thread-safety. Though the concept of thread-safety by confining object to Thread is there from JDK 1.0 and many programmers have their own custom ThreadLocal classes, having ThreadLocal in Java API makes it a lot more easy and standard. Think about the ThreadLocal variable while designing concurrency in our application.   
Don't misunderstand that ThreadLocal is an alternative to Synchronization, it all depends upon the design. If the design allows each thread to have their own copy of the object then ThreadLocal is there to use.  
Producer Consumer Problem with Wait and Notify - Thread Example

Producer Consumer Problem is a classical concurrency problem and in fact it is one of the concurrency design pattern. In last article we have seen solving Producer Consumer problem in Java using blocking Queue but one of my reader emailed me and requested code example and explanation of solving Producer Consumer problem in Java  with wait and notify method as well, Since its often asked as one of the top coding question in Java. In this Java tutorial, I have put the code example of wait notify version of earlier producer consumer concurrency design pattern. We can see this is much longer code with explicit handling blocking conditions like when shared queue is full and when queue is empty. Since we have replaced BlockingQueue with Vector we need to implement blocking using wait and notify and that's why we have introduced produce(int i) and consume() method.  
If we see I have kept consumer thread little slow by allowing it to sleep for 50 Milli second to give an opportunity to producer to fill the queue, which helps to understand that Producer thread is also waiting when Queue is full.

**Q.Java program to solve Producer Consumer Problem in Java**

Here is complete Java program to solve producer consumer problem in Java programming language. In this program we have used wait and notify method from java.lang.Object class instead of using BlockingQueue for flow control.

import java.util.Vector;  
import java.util.logging.Level;  
import java.util.logging.Logger;  
  
/\*\*  
 \* Java program to solve Producer Consumer problem using wait and notify  
 \* method in Java. Producer Consumer is also a popular concurrency design pattern.  
 \*  
 \* @author Javin Paul  
 \*/  
public class ProducerConsumerSolution {  
  
    public static void main(String args[]) {  
        Vector sharedQueue = new Vector();  
        int size = 4;  
        Thread prodThread = new Thread(new Producer(sharedQueue, size), "Producer");  
        Thread consThread = new Thread(new Consumer(sharedQueue, size), "Consumer");  
        prodThread.start();  
        consThread.start();  
    }  
}  
  
class Producer implements Runnable {  
  
    private final Vector sharedQueue;  
    private final int SIZE;  
  
    public Producer(Vector sharedQueue, int size) {  
        this.sharedQueue = sharedQueue;  
        this.SIZE = size;  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 7; i++) {  
            System.out.println("Produced: " + i);  
            try {  
                produce(i);  
            } catch (InterruptedException ex) {  
                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);  
            }  
  
        }  
    }  
  
    private void produce(int i) throws InterruptedException {  
  
        //wait if queue is full  
        while (sharedQueue.size() == SIZE) {  
            synchronized (sharedQueue) {  
                System.out.println("Queue is full " + Thread.currentThread().getName()

                                    + " is waiting , size: " + sharedQueue.size());  
  
                sharedQueue.wait();  
            }  
        }  
  
        //producing element and notify consumers  
        synchronized (sharedQueue) {  
            sharedQueue.add(i);  
            sharedQueue.notifyAll();  
        }  
    }  
}  
  
class Consumer implements Runnable {  
  
    private final Vector sharedQueue;  
    private final int SIZE;  
  
    public Consumer(Vector sharedQueue, int size) {  
        this.sharedQueue = sharedQueue;  
        this.SIZE = size;  
    }  
  
    @Override  
    public void run() {  
        while (true) {  
            try {  
                System.out.println("Consumed: " + consume());  
                Thread.sleep(50);  
            } catch (InterruptedException ex) {  
                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);  
            }  
  
        }  
    }  
  
    private int consume() throws InterruptedException {  
        //wait if queue is empty  
        while (sharedQueue.isEmpty()) {  
            synchronized (sharedQueue) {  
                System.out.println("Queue is empty " + Thread.currentThread().getName()

                                    + " is waiting , size: " + sharedQueue.size());  
  
                sharedQueue.wait();  
            }  
        }  
  
        //Otherwise consume element and notify waiting producer  
        synchronized (sharedQueue) {  
            sharedQueue.notifyAll();  
            return (Integer) sharedQueue.remove(0);  
        }  
    }  
}  
  
Output:  
Produced: 0  
Queue is empty Consumer is waiting , size: 0  
Produced: 1  
Consumed: 0  
Produced: 2  
Produced: 3  
Produced: 4  
Produced: 5  
Queue is full Producer is waiting , size: 4  
Consumed: 1  
Produced: 6  
Queue is full Producer is waiting , size: 4  
Consumed: 2  
Consumed: 3  
Consumed: 4  
Consumed: 5  
Consumed: 6  
Queue is empty Consumer is waiting , size: 0

That’s all on How to solve producer-consumer problem in Java using wait and notify method. I still think that using BlockingQueue to implement producer consumer design pattern is much better because of its simplicity and concise code. At the same time this problem is an excellent exercise to understand concept of wait and notify method in Java.

What is difference between wait and sleep in Java Thread?

Wait vs sleep in Java

Differences between wait and sleep method in Java multi-threading is one of the very old questions asked in Java interviews. Though both wait and sleep put thread on waiting state, they are completely different in terms of behavior and use cases. Thread.sleep(long millis) is meant for introducing pause, releasing CPU and giving another thread an opportunity to execute while wait is used for inter thread communication in Java. These methods are defined in java.lang.Object class and available to every object in Java. It is based upon object lock, if we remember every object in Java has implicit lock, also known as monitor.  
  
  
When a thread enter into a synchronized method it acquired the lock which is used to protect the critical reason e.g. it acquired lock on current object if it is going inside an instance synchronized method and lock object on class literal if its entering into a static synchronized method.  
  
By using wait() and notify() method two threads can communicate with each other which is key to solve many concurrency problems e.g. produce consumer problem, dining philosopher problem, reader and writer problem, and to implement several Concurrency designs.  
  
In this tutorial, we will learn about following this about wait() and sleep() method in Java :

What is wait() method in Java?

What is sleep() method in Java?

What is difference between wait and sleep in Java?

Where to use wait and sleep in Java?

By the way, if we are preparing for Java interview then I suggest we to take a look at Java Programming Interview Exposed, a very good book from Wrox publication, tailored for Java interviews.

**Q What is wait and sleep method in Java?**

Wait method is defined in Object class and it available to all object, wait() method is always discussed along with its counterpart notify() and notifyAll() method and used in inter thread communication in Java.  
  
The wait() method puts a thread on wait by checking some condition like in Producer Consumer problem, producer thread should wait if Queue is full or Consumer thread should wait if Queue is empty.  
  
The notify() method is used to wake up waiting thread by communicating that waiting condition is over now for example once producer thread puts an item on empty queue it can notify Consumer thread that Queue is not empty any more. On the other hand Sleep() method is used to introduce pause on Java application.  
  
We can put a Thread on sleep, where it does not do anything and relinquish the CPU for specified duration. When a Thread goes to Sleep it can be either wake up normally after sleep duration elapsed or it can be woken up abnormally by interrupting it.  
  
**Q Difference between Wait and Sleep method in Java Thread**

In last section we saw what is wait and sleep method and in this section we will see what are differences between wait and sleep method in Java. As I told before apart from waiting they are completely different to each other:

1) First and most important difference between Wait and sleep method is that wait method must be called from synchronized context i.e. from synchronized method or block in Java. If we call wait method without synchronization, it will throw IllegalMonitorStateException in Java. On the other hand there is no requirement of synchronization for calling sleep method , we can call it normally.

2) Second worth noting difference between wait and sleep method is that, wait operates on Object and defined in Object class while sleep operates on current Thread and defined in java.lang.Thread class.

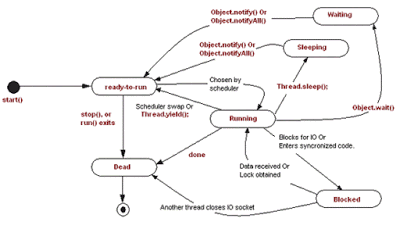
3) Third and another significant difference between wait and sleep in Java is that, wait() method releases the lock of object on which it has called, it does release other locks if it holds any while sleep method of Thread class does not release any lock at all.

4) wait method needs to be called from a loop in order to deal with false alarm i.e. waking even though waiting condition still holds true, while there is no such thing for sleep method in Java. its better not to call Sleep method from loop.

here is code snippet for calling wait and sleep method in Java

synchronized(monitor)  
while(condition == true){ monitor.wait())  //releases monitor lock  
Thread.sleep(100); //puts current thread on Sleep

5) One more difference between wait and sleep method which is not as significant as previous ones is that wait() is a non static method while sleep() is static method in Java.



Where to use wait and sleep method in Java

By reading properties and behavior of wait and sleep method it's clear that wait() method should be used in conjunction with notify() or notifyAll() method and intended for communication between two threads in Java while Thread.sleep() method is a utility method to introduce short pauses during program or thread execution. Given the requirement of synchronization for wait, it should not be used just to introduce pause or sleep in Java.

In summary wait and sleep method are completely different to each other and have different use cases. Use wait() and notify() method for inter thread communication while use sleep() method for introducing small pause during thread execution. Also remember, that wait() method will release the lock acquired when it entered into synchronized block or method, but sleep() method will keep the lock with itself. So if we design require releasing the lock during wait period then use wait() and notify method otherwise just use sleep().