**Concurrency:**

In java, java.util.concurrent package helps us to deal with concurrency.

There are two packages under it. Named java.util.concurrent.atomic and java.util.concurrent.locks.

**Atomic Variables:**

public class Counter

{

private int count;

public void increment()

{

count++;

//just because, it is a single line does not mean it is thread safe

//just because, it is a single line, it does not mean it is atomic

}

public int getValue()

{

return count;

}

}

**Full Program:**

class Counter

{

private int count;

//now, we don’t mention any constructor. So, default constructor is there. And, since, we did not explicitly override it

//it can be invoked

public void increment()

{

count++;

//just because, it is a single line does not mean it is thread safe

//just because, it is a single line, it does not mean it is atomic

}

public int getValue()

{

return count;

}

}

class IncrementerThread extends Thread

{

private Counter counter;

**//Now, if all instances passed the same counter reference by value?**

public IncrementerThread(Counter counter)

{

this.counter=counter;

}

//any class which inherits thread must override

public void run()

{

for(int i=0;i<10000;i++)

{

//but counter's private value count is not initialized to any value

counter.increment();

}

//so, we can understand that run function is not atomic

//but, if it was atomic, what would happen?

//it will increment the counter by 10000

}

}

public class ThreadTestingClass

{

public static void main(String args[]) throws InterruptedException

{

Counter counter=new Counter();

System.out.println("Getting the initial value of the counter: "+counter.getValue());

IncrementerThread it1=new IncrementerThread(counter);

IncrementerThread it2=new IncrementerThread(counter);

it1.start();

**//the thread (it1) will start it's execution. So, it's run function will be executed**

System.out.println("The current value of the counter is: "+counter.getValue());

**//Now, though thread 1 starts incrementing here, counter's variable remain 0**

it2.start();

System.out.println("The current value of the counter is: "+counter.getValue());

**//the thread (it2) will start it's execution. So, it's run function will be executed**

it1.join();

System.out.println("The current value of the counter is: "+counter.getValue());

**//the parent thread (which is main thread) here will wait for thread 1 to finish**

it2.join();

**//the parent thread (which is main thread) here will wait for thread 2 to finish**

System.out.println("the current value of the counter is: "+counter.getValue());

**//at this point it will be 20000**

**//though there could be unexpected results, too**

}

}

Now, this can generate some pretty unexpected result.

Why, because, if you think count++ is an atomic operation, it is a trap.

Because, a set of thing will happen.

Suppose, at processor level it is the INC count instruction.

Now, First, it (will it be checked by operating system. Because, operating system itself is a program) will be checked whether count is present in the page table or not. (page table is necessarily an index mechanism for all memory units present in secondary disk storage and Cache and some part of RAM. If from page table it is known that the count variable is not present in the main memory or RAM, it is needed to be loaded from secondary disk storage to main memory or RAM. Now, it is to be checked that if any page is free. Otherwise, we need some page scheduling algorithm to free some page slot. After that, it is loaded into page table. Now, if page fault happens (that count is not present in main memory) the instruction (INC) is to be fetched and decoded again. Now, finally count stored in main memory is incremented. Now, after incrementing, it needs to be reflected back in the secondary disc storage. Now, that is depended on something else. Now, the steps I describes here, might be wrong. However, that much task is required for incrementing a simple variable.

So, the simple statement count++ will be translated by java compiler into multiple java bytecode instruction. Now, JIT (just in time Compiler) based nature of most java runtime environments means you don’t know when or if the count++ statement will be translated into native CPU instructions and whether it ends up as a single instruction or several. However, we should always act as if a single line of java code takes multiple steps to complete. (Because, whether an erroneous result will be obtained is depended on many factors: The JIT (just In Time Compiler) nature of the compiler, the type of CPU(Do both threads in the example run concurrently or in sequence? Here, a large loop count was used in order to make the threads run longer and be more likely to execute concurrently instead of executing in sequence)

Now, java.util.concurrent.atomic gives you a mechanism.

The Same Example With Atomic Variable: AtomicInteger

**class Counter**

**{**

**private AtomicInteger count;**

//now, we dont mention any constructor. So, default constructor is there. And, since, we did not explicitly override it

//it can be invoked

**public void increment()**

**{**

**count.getAndIncrement();**

//just because, it is a single line does not mean it is thread safe

//just because, it is a single line, it does not mean it is atomic

**}**

**public int getValue()**

**{**

**return count.intValue;**

**}**

**}**

Here, is an implementation of counter which is thread safe and lock free.

That is why atomic variables comes into scenario.

**java.util.concurrent.atomic** defines many atomic Classes.

**AtomicBoolean**

A boolean value that may be updated atomically.

**AtomicInteger**

An int value that may be updated atomically.

**AtomicIntegerArray**

An int array in which elements may be updated atomically.

**AtomicIntegerFieldUpdater<T>**

A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.

**AtomicLong**

A long value that may be updated atomically.

**AtomicLongArray**

A long array in which elements may be updated atomically.

**AtomicLongFieldUpdater<T>**

A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.

**AtomicMarkableReference<V>**

An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.

**AtomicReference<V>**

An object reference that may be updated atomically.

**AtomicReferenceArray<E>**

An array of object references in which elements may be updated atomically.

**AtomicReferenceFieldUpdater<T,V>**

A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.

**AtomicStampedReference<V>**

An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.

**DoubleAccumulator**

One or more variables that together maintain a running double value updated using a supplied function.

**DoubleAdder**

One or more variables that together maintain an initially zero double sum.

**LongAccumulator**

One or more variables that together maintain a running long value updated using a supplied function.

**LongAdder**

One or more variables that together maintain an initially zero long sum.

Java.util.concurrent.Atomic is a small toolkit of classes that support lock-free thread-safe programming on single variables.

Now, go back to the previous example. After that, we will discuss the other functionality this **java.util.concurrent.Atomic** provides.

class Counter

{

private AtomicInteger count;

**//now, we dont mention any constructor. So, default constructor is there. And, since, we did not explicitly override it**

**//it can be invoked**

public void increment()

{

count.getAndIncrement();

**//just because, it is a single line does not mean it is thread safe**

**//just because, it is a single line, it does not mean it is atomic**

}

public int getValue()

{

return count.intValue;

}

}

Now, in reality, even a method such as getAndIncrement() still takes several steps to execute. The reason this implementation is thread safe is something called CAS (Compare and Swap)

Some common functions this atomic package offers:

**addAndGet(int)**

Atomically adds the given value to the current value.

**getAndAdd(int)**

Now, in both case, I guess the order is different.

**compareAndSet(int expect, int update)**

Atomically sets the value to the given updated value if the current value == the expected value.

**getAndDecrement()**

**getAndIncrement()**

**getAndSet(int newValue)**

Atomically sets to the given value and returns the old value.

**Locks:**

Atomic can provide thread safety (without lock) for a single variable. For two or more than two variables, we need lock.

The things java.util.concurrent.locks provide:

* The ability to duplicate traditional synchronized blocks.
* Nonblock scoped blocking:-obtain a lock in one method and release it in another .
* Multiple wait/notify/notifyAll pools per lock: thread can select which pool (condition) they wait on.
* The ability to attempt to acquire a lock and take an alternative action if locking fails
* An implementation of multiple -reader, single writer block. (Now, this is ReentrantReadWriteLock)

Now, before we proceed into details of java.util.concurrent.locks we will first revisit few concepts:

**Synchronized block in java:**

Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

(So, different synchronized block on the same object produces a version of locking. (the concept of critical section and remainder section) . The synchronized block on an object creates a critical section for that object. So, it can only have one thread executing inside them at the same time.

All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

The synchronized keyword can be used to mark four different types of blocks:

Instance methods

Static methods

Code blocks inside instance methods

Code blocks inside static methods

These blocks are synchronized on different objects. Which type of synchronized block you need depends on the concrete situation.

**Synchronized Instance Methods**

Here is a synchronized instance method:

public synchronized void add(int value)

{

this.count += value;

}

Notice the use of the synchronized keyword in the method declaration. This tells Java that the method is synchronized.

A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance. Only one thread can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

**Synchronized Static Methods**

Static methods are marked as synchronized just like instance methods using the synchronized keyword. Here is a Java synchronized static method example:

public static synchronized void add(int value)

{

count += value;

}

Also here the synchronized keyword tells Java that the method is synchronized.

Synchronized static methods are synchronized on the class object of the class the synchronized static method belongs to. Since only one class object exists in the Java VM per class, only one thread can execute inside a static synchronized method in the same class.

If the static synchronized methods are located in different classes, then one thread can execute inside the static synchronized methods of each class. One thread per class regardless of which static synchronized method it calls.

**Synchronized Blocks in Instance Methods**

You do not have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

Here is a synchronized block of Java code inside an unsynchronized Java method:

public void add(int value){

synchronized(this){

this.count += value;

}

}

This example uses the Java synchronized block construct to mark a block of code as synchronized. This code will now execute as if it was a synchronized method.

Notice how the Java synchronized block construct takes an object in parentheses. In the example "this" is used, which is the instance the add method is called on. The object taken in the parentheses by the synchronized construct is called a monitor object. The code is said to be synchronized on the monitor object. A synchronized instance method uses the object it belongs to as monitor object.

Only one thread can execute inside a Java code block synchronized on the same monitor object.

The following two examples are both synchronized on the instance they are called on. They are therefore equivalent with respect to synchronization:

public class MyClass {

public synchronized void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

public void log2(String msg1, String msg2){

synchronized(this){

log.writeln(msg1);

log.writeln(msg2);

}

}

}

Thus only a single thread can execute inside either of the two synchronized blocks in this example.

Had the second synchronized block been synchronized on a different object than this, then one thread at a time had been able to execute inside each method.

**Synchronized Blocks in Static Methods**

Here are the same two examples as static methods. These methods are synchronized on the class object of the class the methods belong to:

public class MyClass {

public static synchronized void log1(String msg1, String msg2){

log.writeln(msg1);

log.writeln(msg2);

}

**Or,**

public static void log2(String msg1, String msg2){

synchronized(MyClass.class){

log.writeln(msg1);

log.writeln(msg2);

}

}

}

Note: the difference between this and MyClass.class (The Class name is MyClass)

Only one thread can execute inside any of these two methods at the same time.

Had the second synchronized block been synchronized on a different object than MyClass.class, then one thread could execute inside each method at the same time.

Note className.class is required here to synchronize a block (independent on some variables on the class)

Now, in case of synchronized block, all synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

(Now, how this blocking achieved? Is that a spin lock mechanism? Or the thread sleeps for some time and is notified when current thread finishes it’s task. If synchronized block does not contain wait() and in some other place it does not contain notify, will it implement spin lock?

(

here a single CPU is shared among many processes. Busy waiting wastes

CPU cycles that some other process might be able to use productively. This

type of semaphore is also called a because the process "spins" while

waiting for the lock. (Spinlocks do have an advantage in that no context switch

is required when a process must wait on a lock, and a context switch may

take considerable time. Thus, when locks are expected to be held for short

times, spinlocks are useful; they are often employed on multiprocessor systems

where one thread can "spin" on one processor while another thread performs

its critical section on another processor.)

Solution Of Busy Waiting:

void wait(semaphore \*S)

{

S->value--;

If(value<0)

{

Add the current process to the S->list

Block();

}

}

signal(semaphore \*S)

{

S->value++;

If (s->value <= 0)

{

remove a process P from S->list;

wakeup(P);

}

}

)

**Wait, notify, notifyAll**

**wait**

Object wait methods has three variance, one which waits indefinitely for any other thread to call notify or notifyAll method on the object to wake up the current thread. Other two variances puts the current thread in wait for specific amount of time before they wake up.

**notify**

notify method wakes up only one thread waiting on the object and that thread starts execution. So if there are multiple threads waiting for an object, this method will wake up only one of them. The choice of the thread to wake depends on the OS implementation of thread management.

**notifyAll**

notifyAll method wakes up all the threads waiting on the object, although which one will process first depends on the OS implementation.

**General Syntax For wait mechanism:**

**wait() :** It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify(). The wait() method releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method. The wait() method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism. In other words, it is not possible for us to implement the wait() method purely in Java: it is a native method.

General syntax for calling wait() method is like this:

**synchronized( lockObject )**

**{**

**while( ! condition )**

**{**

**lockObject.wait();**

**}**

**//take the action here;**

**}**

**notify() :** It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource. It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier’s synchronized block has completed. So, if a notifier calls notify() on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though notify() had been called.

General syntax for calling notify() method is like this:

**synchronized(lockObject)**

**{**

**//establish\_the\_condition;**

**lockObject.notify();**

**//any additional code if needed**

**}**

**notifyAll() :** It wakes up all the threads that called wait() on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as notify() method above.

General syntax for calling notify() method is like this:

**synchronized(lockObject)**

**{**

**establish\_the\_condition;**

**lockObject.notifyAll();**

**}**

**Wait And Notify First Example:**

public class ThreadA

{

public static void main(String[] args)

{

ThreadB b = new ThreadB();

b.start();

//How, can this segment of code is accessed by multiple thread?

synchronized(b)

//what’s the point of synchronizing b here?

{

try

{

System.out.println("Waiting for b to complete...");

b.wait();

//Now, main thread calls b.wait() instead of b.join()

//b.join() makes the current thread(which is main thread) to wait for the thead b

//now, here, we achieve the same thing by b.wait() However, b must notify after it finishes it’s execution

}

catch(InterruptedException e)

{

e.printStackTrace();

}

System.out.println("Total is: " + b.total);

}

}

}

class ThreadB extends Thread

{

int total;

@Override

public void run()

{

synchronized(this)

{

for(int i=0; i<100 ; i++)

{

total += i;

}

notify();

}

}

}

Now, a general producer consumer solution using lock is the following:

**Producer Module:**

do

{

produce an item in nextp

wait(empty);

**//Think about it. Empty initially needed to be initialized to n (if there are n slots in the buffer)**

wait(mutex);

**//this mutex is relatively simple. This will be either acquired hen an item form the buffer is being consumed or an item is produced**

add nextp to buffer

signal(mutex);

signal(full);

} while (TRUE);

**Consumer Module:**

do

{

wait (full);

**//again full mutex will be a counting mutex and must be initialized to 0**

**//now, if 0, that means no item is produced. Hence, it will wait until some element in the buffer is produced and it can consume**

wait (mutex) ;

remove an item from buffer to nextc

signal(mutex);

signal(empty);

consume the item in nextc

} while (TRUE);

The same logic is implemented using wait, notify and synchronized block

import java.util.Vector;

class Producer extends Thread

{

static final int MAXQUEUE = 5;

**//Queue size**

private Vector messages = new Vector();

**@Override**

public void run()

{

//so, producer

try

{

while (true)

{

putMessage();

**//Now, consumer’s primary job is to put a message in a buffer**

**//sleep(5000);**

**//We will later see the implementation of it**

}

}

catch (InterruptedException e)

{

}

}

private synchronized void putMessage() throws InterruptedException

{

/**/Now, it is synchronized function. What does it mean?**

**//Means, the variables in which the function operates, those variables will be accessed in a synchronized manner**

**//Now, this variable will operate on messages vector. Hence, messages //vector will be accessed in a synchronized manner. So, when one thread is //doing some operation on it, another thread cannot**

while (messages.size() == MAXQUEUE)

{

wait();

}

messages.addElement(new java.util.Date().toString());

System.out.println("put message");

notify();

**//Later, when the necessary event happens, the thread that is running it //calls notify() from a block synchronized on the same object.**

}

**// Called by Consumer**

public synchronized String getMessage() throws InterruptedException

{

notify();

while (messages.size() == 0)

{

**wait();//By executing wait() from a synchronized block, a thread** gives up its hold on the lock and goes to sleep.

}

String message = (String) messages.firstElement();

messages.removeElement(message);

return message;

}

}

class Consumer extends Thread

{

Producer producer;

Consumer(Producer p)

{

producer = p;

}

@Override

public void run()

{

try

{

while (true)

{

String message = producer.getMessage();

System.out.println("Got message: " + message);

//sleep(200);

}

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

public static void main(String args[])

{

Producer producer = new Producer();

producer.start();

new Consumer(producer).start();

}

}

However, this design is slightly complex . though, I perfectly understand it.

**Implement Runnable vs Extend Thread in Java:**

We can define a thread in the following two ways:

* By extending Thread class
* By implementing Runnable interface

In the first approach, Our class always extends Thread class. There is no chance of extending any other class. Hence we are missing Inheritance benefits. In the second approach, while implementing Runnable interface we can extends any other class. Hence we are able to use the benefits of Inheritance.

Because of the above reasons, implementing Runnable interface approach is recommended than extending Thread class.

**The significant differences between extending Thread class and implementing Runnable interface:**

* When we extend Thread class, we can’t extend any other class even we require and When we implement Runnable, we can save a space for our class to extend any other class in future or now.
* When we extend Thread class, each of our thread creates unique object and associate with it. When we implements Runnable, it shares the same object to multiple threads.
* Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread implementing Runnable interface introduces loose coupling as the code of Thread is separate form the job of Threads.

This question already has an answer here:

“implements Runnable” vs. “extends Thread”

While reading through the significant difference between Thread and Runnable from here, I encountered a difference that is:

When you extends Thread class, each of your thread creates unique object and associate with it. where as

When you implement Runnable, it shares the same object to multiple threads..

**There is code given :**

class ImplementsRunnable implements Runnable {

private int counter = 0;

public void run() {

counter++;

System.out.println("ImplementsRunnable : Counter : " + counter);

}

}

class ExtendsThread extends Thread {

private int counter = 0;

public void run() {

counter++;

System.out.println("ExtendsThread : Counter : " + counter);

}

}

public class ThreadVsRunnable {

public static void main(String args[]) throws Exception {

//Multiple threads share the same object.

ImplementsRunnable rc = new ImplementsRunnable();

Thread t1 = new Thread(rc);

t1.start();

Thread.sleep(1000); // Waiting for 1 second before starting next thread

Thread t2 = new Thread(rc);

t2.start();

Thread.sleep(1000); // Waiting for 1 second before starting next thread

Thread t3 = new Thread(rc);

t3.start();

//Creating new instance for every thread access.

ExtendsThread tc1 = new ExtendsThread();

tc1.start();

Thread.sleep(1000); // Waiting for 1 second before starting next thread

ExtendsThread tc2 = new ExtendsThread();

tc2.start();

Thread.sleep(1000); // Waiting for 1 second before starting next thread

ExtendsThread tc3 = new ExtendsThread();

tc3.start();

}

}

The output is something like this:

ImplementsRunnable : Counter : 1

ImplementsRunnable : Counter : 2

ImplementsRunnable : Counter : 3

ExtendsThread : Counter : 1

ExtendsThread : Counter : 1

ExtendsThread : Counter : 1

It proves differences given above.

**I make a slight modification in code given below:**

**class ImplementsRunnable implements Runnable**

**{**

**private int counter = 0;**

**public void run()**

**{**

**counter++;**

**System.out.println("ImplementsRunnable : Counter : " + counter);**

**}**

**}**

**public class Runnable**

**{**

**public static void main(String args[]) throws Exception**

**{**

**//Multiple threads share the same object.**

**ImplementsRunnable rc = new ImplementsRunnable();**

**Thread t1 = new Thread(rc);**

**t1.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t2 = new Thread(rc);**

**t2.start();**

**Thread.sleep(1000); // Waiting for 1 second before starting next thread**

**Thread t3 = new Thread(rc);**

**t3.start();**

**}**

**}**

Now, this is a sample of Runnable interface using. Now, ImplementsRunnable class which implements Runnable, can be shared among various threads.

Now, how to pass that Runnable object in thread? Pass it in constructor during thread creation.

Now, notice that, none of thread in joined here with main thread

(Now, join does the following: join() is a instance method of java.lang.Thread class which we can use join() method to ensure all threads that started from main must end in order in which they started and also main should end in last. )

**Important Note about multi-threading:**

**NOTE :** In the case of multithreading we cant predict the exact order of output because it will vary from system to system or JVM to JVM.