Java 7 Automatic Resource Management with try-with-resources

**The old way of resource cleanup (Before java 7)**

We have been doing this from long time. e.g. read a file from file system. Code may look different but flow will be like below example:

**public class** ResourceManagementBeforeJava7 {  
 **public static void** main(String[] args) {  
 BufferedReader br = **null**;  
 **try** {  
 String sCurrentLine;  
 br = **new** BufferedReader(**new** FileReader(**"C:/temp/test.txt"**));  
 **while** ((sCurrentLine = br.readLine()) != **null**) {  
 System.***out***.println(sCurrentLine);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **finally** {  
 **try** {  
 **if** (br != **null**) **br.close()**;  
 } **catch** (IOException ex) {  
 ex.printStackTrace();  
 }  
 }  
 }  
}

These types of code are very common in application code base where there is lots of IO operations.

Code inside try and catch blocks are essentially important and have some application specific logics. But what about finally block?? **Most of the time, finally blocks are just copy pasted** for sake of saving the resources from corruption, by closing them.

These finally blocks **looks more ugly, when you have 3-4 such resources to close in single finally block**. Don’t you think these finally blocks are unnecessarily there when we know, we have to close the resource anyhow without any exceptional case??

## ****The new fancy way with try-with-resources (syntax example)****

Now look at the new way of opening and closing a resource in java 7.

**public class** ResourceManagementInJava7 {  
 **public static void** main(String[] args) {  
 **try** (BufferedReader br = **new** BufferedReader(**new** FileReader(**"C:/temp/test.txt"**))) {  
 String sCurrentLine;  
 **while** ((sCurrentLine = br.readLine()) != **null**) {  
 System.***out***.println(sCurrentLine);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

There are two things to closely watch:

1. File resource (BufferedReader) is opened in try block in special manner (inside small brackets).
2. Finally block is completely gone.

And last but not the least, code looks pretty and easy to read. It’s good ,right?? But how actually it works??

**How actually it works?**

In java 7, we have a new super interface [**java.lang.AutoCloseable**](http://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html). This interface have one method:

|  |
| --- |
| void close() throws Exception; |

Java docs recommend this interface to be **implemented on any resource that must be closed when it is no longer needed**.

When we open any such AutoCloseable resource in special try-with-resource block, immediately after finishing the try block, **JVM calls this close() method on all resources initialized in “try()” block**.

For example, BufferedReader has implemented close() method file this:

|  |
| --- |
| public void close() throws IOException {      synchronized (lock) {          if (in == null)              return;          in.close();          in = null;          cb = null;      }  } |

Due to above method definition, any underlying stream or IO resource is closed when this method is invoked by JVM.

## ****Adding functionality to custom resources****

Well, this is a good resource cleanup design. But is it available to JDK native classes only?? NO. You can use it also to your custom resources.

For example, I have create a custom resource in below code:

**public class** CustomResource **implements** AutoCloseable {  
 **public void** accessResource() {  
 System.***out***.println(**"Accessing the resource"**);  
 }  
   
 @Override  
 **public void** close() **throws** Exception {  
 System.***out***.println(**"CustomResource closed automatically"**);  
 }  
}

Now I will use it in my example code:

**public class** TryWithCustomResource {  
 **public static void** main(String[] args) {  
 **try** (CustomResource cr = **new** CustomResource()) {  
 cr.accessResource();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

Output in console:

Accessing the resource

CustomResource closed automatically

Output in console clearly proves that resource was closed down automatically as soon as try block was finished.

## ****Final notes****

That’s all regarding automatic resource management with try-with-resources in java 7. Let’s note down highlights point by point:

* Before java 7, we had to use finally blocks to cleanup the resources. Finally blocks were not mandatory, but resource clean up was to prevent the system from being corrupt.
* With java 7, no need to explicit resource cleanup. Its done automatically.
* Automatic resource cleanup done when initializing resource in try-with-resources block (try(…) {…}).
* Cleanup happens because of new interface AutoCloseable. Its close method is invoked by JVM as soon as try block finishes.
* If you want to use this in custom resources, then implementing AutoCloseable interface is mandatory. otherwise program will not compile.
* You are not supposed to call close() method in your code. This should be called automatically bu JVM. Calling it manually may cause unexpected results.

[What is a race condition?](http://stackoverflow.com/questions/34510/what-is-a-race-condition)

<http://stackoverflow.com/questions/34510/what-is-a-race-condition>

**A race condition occurs when two or more threads can access shared data and they try to change it at the same time**. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g:

if (x == 5) // The "Check"

{

y = x \* 2; // The "Act"

// If another thread changed x in between "if (x == 5)" and "y = x \* 2" above,

// y will not be equal to 10.

}

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.

In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

// Obtain lock for x

if (x == 5)

{

y = x \* 2; // Now, nothing can change x until the lock is released.

// Therefore y = 10

}

// release lock for x

How AtomicInteger or AtomicLong works in Java

Before we dig into CAS (Compare And Swap) strategy and how is it used by atomic constructs like [AtomicInteger](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html), first consider this code:

public class MyApp

{

private volatile int count = 0;

public void upateVisitors()

{

++count; //increment the visitors count

}

}

This sample code is tracking the count of visitors to the application. Is there anything wrong with this code? What will happen if multiple threads try to update count? Actually the problem is simply marking count as volatile does not guarantee atomicity and ++count is not an atomic operations. To read more check [this](http://flex4java.blogspot.in/2015/03/volatile-atomicity-visibility-and.html).

Can we solve this problem if we mark the method itself synchronized as shown below:

public class MyApp

{

private int count = 0;

public synchronized void upateVisitors()

{

++count; //increment the visitors count

}

}

Will this work? If yes then what changes have we made actually?

Does this code guarantee atomicity? Yes.

Does this code guarantee visibility? Yes.

Then what is the problem?

It makes use of locking and that introduces lot of delay and overhead. Check this [article](http://flex4java.blogspot.in/2015/03/is-multi-threading-really-worth-it.html). This is very expensive way of making things work.

To overcome these problems atomic constructs were introduced. If we make use of an AtomicInteger to track the count it will work.

public class MyApp

{

private AtomicInteger count = new AtomicInteger(0);

public void upateVisitors()

{

count.incrementAndGet(); //increment the visitors count

}

}

The classes that support atomic operations e.g. AtomicInteger, AtomicLong etc. makes use of CAS. CAS does not make use of locking rather it is very optimistic in nature. It follows these steps:

* Compare the value of the primitive to the value we have got in hand.
* If the values do not match it means some thread in between has changed the value. Else it will go ahead and swap the value with new value.

Check the following code in AtomicLong class:

public final long incrementAndGet() {

for (;;) {

long current = get();

long next = current + 1;

if (compareAndSet(current, next))

return next;

}

}

In JDK 8 the above code has been changed to a single intrinsic:

public final long incrementAndGet() {

return unsafe.getAndAddLong(this, valueOffset, 1L) + 1L;

}

**public** **final** **boolean** [http://grepcode.com/static/app/images/1x1.gif](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLong.java)compareAndSet(**long** expect, **long** update) { //As for JDK 7

**return** [unsafe](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLong.java#AtomicLong.0unsafe).[compareAndSwapLong](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/sun/misc/Unsafe.java#Unsafe.compareAndSwapLong%28java.lang.Object%2Clong%2Clong%2Clong%29)(**this**, [valueOffset](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLong.java#AtomicLong.0valueOffset), expect, update);

    }

**What advantage this single intrinsic have?**

Actually this single line is JVM intrinsic which is translated by JIT into an optimized instruction sequence. In case of x86 architecture it is just a single CPU instruction LOCK XADD which [might yield better performance than classic load CAS loop](https://blogs.oracle.com/dave/entry/atomic_fetch_and_add_vs).

Now think about the possibility when we have high contention and a number of threads want to update the same atomic variable. In that case there is a possibility that locking will outperform the atomic variables but in realistic contention levels atomic variables outperform lock. There is one more construct introduced in Java 8, [LongAdder](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html). As per the documentation:

**This class is usually preferable to AtomicLong when multiple threads update a common sum that is used for purposes such as collecting statistics, not for fine-grained synchronization control. Under low update contention, the two classes have similar characteristics. But under high contention, expected throughput of this class is significantly higher, at the expense of higher space consumption.**

So LongAdder is [not always a replacement](http://blog.palominolabs.com/2014/02/10/java-8-performance-improvements-longadder-vs-atomiclong/) for AtomicLong. We need to consider the following aspects:

* When no contention is present AtomicLong performs better.
* LongAdder will allocate Cells (a final class declared in abstract class [Striped64](http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/jsr166e/Striped64.java?view=co)) to avoid contention which consumes memory. So in case we have a tight memory budget we should prefer AtomicLong.

What is Mutex in Java

<http://searchnetworking.techtarget.com/definition/mutex>

In computer programming, a mutex (mutual exclusion object) is a program object that is created so that multiple program [thread](http://searchcio-midmarket.techtarget.com/definition/thread) can take turns sharing the same resource, such as access to a file. Typically, when a program is started, it creates a mutex for a given resource at the beginning by requesting it from the system and the system returns a unique name or ID for it. After that, any thread needing the resource must use the mutex to lock the resource from other threads while it is using the resource. If the mutex is already locked, a thread needing the resource is typically queued by the system and then given control when the mutex becomes unlocked (when once more, the mutex is locked during the new thread's use of the resource).

<http://stackoverflow.com/questions/34524/what-is-a-mutex>

When I am having a big heated discussion at work, I use a rubber chicken which I keep in my desk for just such occasions. The person holding the chicken is the only person who is allowed to talk. If you don't hold the chicken you cannot speak. You can only indicate that you want the chicken and wait until you get it before you speak. Once you have finished speaking, you can hand the chicken back to the moderator who will hand it to the next person to speak. This ensures that people do not speak over each other, and also have their own space to talk.

Replace Chicken with Mutex and person with thread and you basically have the concept of a mutex.

Of course, there is no such thing as a rubber mutex. Only rubber chicken. My cats once had a rubber mouse, but they ate it.

Of course, before you use the rubber chicken, you need to ask yourself whether you actually need 5 people in one room and would it not just be easier with one person in the room on their own doing all the work. Actually, this is just extending the analogy, but you get the idea.

**In this case it is a microphone.**

**A Mutex is a mutually exclusive flag. It acts as a gate keeper to a section of code allowing one thread in and blocking access to all others.** This ensures that the code being controlled will only be hit by a single thread at a time. Just be sure to release the mutex when you are done.

**Semaphore**:- **It is a variable used for controlling access to a common resource by multiple processes/threads in a concurrent system**. In other words, semaphore is used to manage how many units of resources are available in multi-process or multi-threaded environment.

Possible value of a semaphore can vary from 0 to N, where is N is maximum units of shared resources allowed in the given system.  
There are two type of semaphore -

1. Counting semaphore (Values varies from 0 to N)

2. Binary semaphore (Values either 0  or  1)

*Note*:- Java's semaphore is a counting semaphore.

**Mutex**:- It is basically mutual exclusion. In other words, only one thread is allowed to acquire the resource at a time. When one thread acquires the resource, no other thread is allowed to acquire the resource until the thread owning the resource releases. All threads waiting for acquiring resource would be blocked.

At high level, binary semaphore is similar to Mutex. Binary semaphore can act as Mutex. However, there is fundamental difference between Mutex and Semaphore :-  the concept of "**ownership**".

Semaphores have no notion of ownership, this means that any thread can release a semaphore, Whereas a mutex does have the concept of ownership The process that locked the mutex is supposed to unlock it.

Mutex primary use is **to guard shared resources** in multi-threaded environment.

**When do we use semaphore and Mutex** :- The usefulness of Mutex and Semaphore lies in - how many threads are allowed to acquire the resource at once ?

If answer is one - Mutex is suitable candidate.   
Otherwise Semaphore as it allows multiple threads(equal to number of permitted semaphore values) to access shared resources.