JVM Command line switches

HotSpot, the primary reference Java VM implementation. (Used by both Oracle Java and OpenJDK)

Maxine Virtual Machine

Jikes Research Virtual Machine

CACAO is a research Java Virtual Machine

JAmiga is an open-source Java virtual machine for the Amiga platform

The java command supports a wide range of options that can be divided into the following categories:

- ✓ Standard Options
- ✓ Non-Standard Options
- ✓ Advanced Runtime Options
- ✓ Advanced JIT Compiler Options
- ✓ Advanced Serviceability Options
- ✓ Advanced Garbage Collection Options

Standard options are guaranteed to be supported by all implementations of the Java Virtual Machine (JVM).

They are used for common actions, such as checking the version of the JRE, setting the class path, enabling verbose output, and so on.

-agentlib:libname[=options]
Loads the specified native agent library

-client
Selects the Java HotSpot Client VM.

-server
Selects the Java HotSpot Server VM

-Dproperty=value
Sets a system property value.

-jar filename
Executes a program encapsulated in a JAR file.

-version
Displays version information and then exits.

-verbose:gc
Displays information about each garbage collection (GC) event.

Non-standard options are general purpose options that are *specific to the Java HotSpot Virtual* Machine, so they are not guaranteed to be supported by all JVM implementations, and are subject to change.

These options start with -X.

-X
Displays help for all available -X options.

-Xloggc:filename
Sets the file to which verbose GC events information should be redirected for logging.

-Xmssize
Sets the initial size of the heap

-Xmxsize
Specifies the maximum size of the memory

-Xsssize
Sets the thread stack size

Advanced options are not recommended for casual use. These are developer options used for tuning specific areas of the Java HotSpot Virtual Machine operation that often have specific system requirements and may require privileged access to system configuration parameters.

They are also not guaranteed to be supported by all JVM implementations, and are subject to change.

Advanced options start with -XX.

-XX:ErrorFile=filename
Specifies the path and file name to which error data is written when an irrecoverable error occurs.

-XX:NativeMemoryTracking=mode
Specifies the mode for tracking JVM native memory usage

-XX:+PrintCommandLineFlags
Enables printing of ergonomically selected JVM flags that appeared on the command line.

-XX:+UseLargePages
Enables the use of large page memory.

-XX:-UseBiasedLocking
Disables the use of biased locking

-XX:+UnlockCommercialFeatures
Enables the use of commercial features

Advanced JIT Compiler Options

These options control the dynamic just-in-time (JIT) compilation performed by the Java HotSpot VM.

-XX:CompileThreshold=invocations
Sets the number of interpreted method invocations before compilation.

-XX:InlineSmallCode=size
Sets the maximum code size (in bytes) for compiled methods that should be inlined

-XX:+LogCompilation
Enables logging of compilation activity to a file named hotspot.log in the current working directory

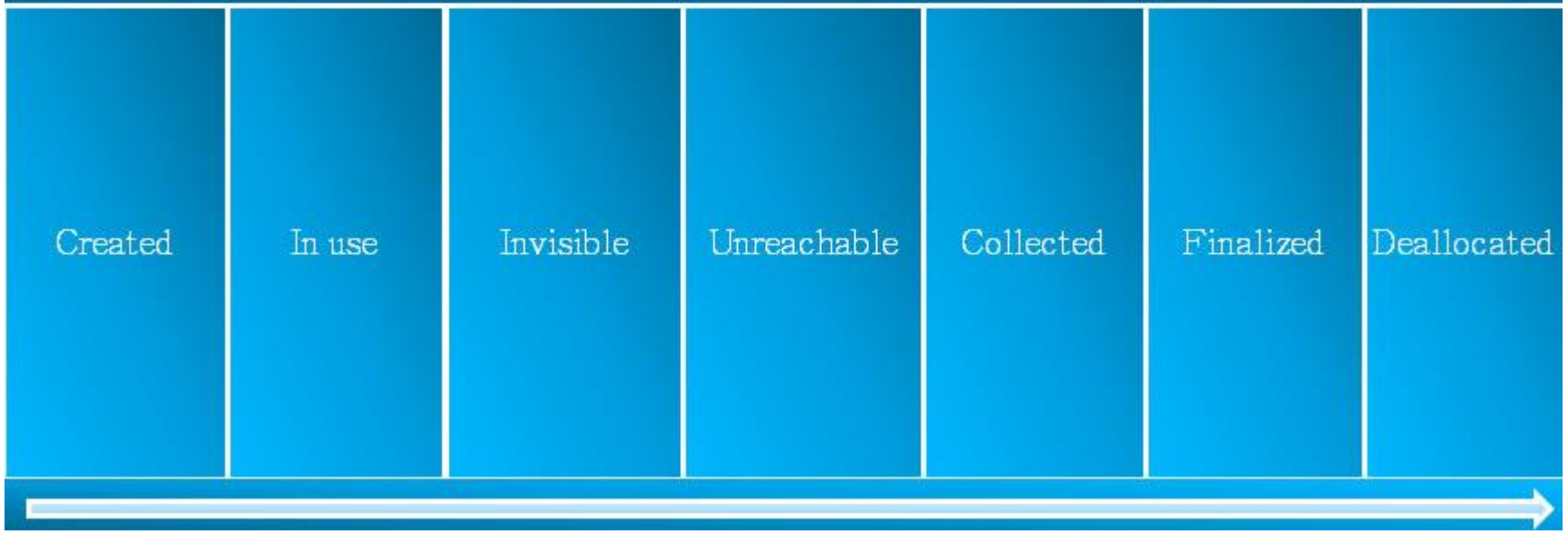
-XX:+PrintCompilation
Enables verbose diagnostic output from the JVM by printing a message to the console every time a method is compiled.

-XX:+PrintInlining
Enables printing of inlining decisions.

-XX:ReservedCodeCacheSize=size
Sets the maximum code cache size (in bytes) for JIT-compiled code.

Java Object Lifecycle

JAVA Object Life Cycle



Object Lifecycle

In Java, it has seven states in Object lifecycle.

Created
In use
Invisible
Unreachable
Collected
Finalized
De-allocated

Created:

Creation of an Object means allocation of memory, calling constructor and initializing its properties.

Account account = new Account(); // creation of Object Account

In Use:

Any Object that is held by any strong reference is said to be in use.

Account account = new Account(); // Here account is strong

Invisible:

Invisible Objects can cause unnecessary memory blockage and can impact the performance as well.

```
public void execute(){
try{
Object obj = new Object(); }


while(true){ ……}
}
```

execution of the code comes to the infinite while loop it may seem that the above Object referenced by obj is out of scope and is eligible for GC but in fact it lives in the same stack frame and occupies memory in heap area. It can cause serious memory blockage and there are chances to get OutOfMemoryException.

Fix: To fix this we have to explicitly set the references to null after using them.

Unreachable:

An Object becomes unreachable when there are no more references to it and it can’t be accessed.

```
Object obj = new Object();

obj=null;
```

Collected:

After getting recognized as unreachable Object it moves to collected state.

It is just a phase before its deallocation.

If the Object has finalize method then it is marked for finalization otherwise it directly moves to finalized state.

Finalized

After Object's finalized method is called and then also it is unreachable then it is in finalized state. The finalize() method is only called once and it's better to handle any clean up in your code elsewhere because the resources are blocked till finalizer is called by the GC.

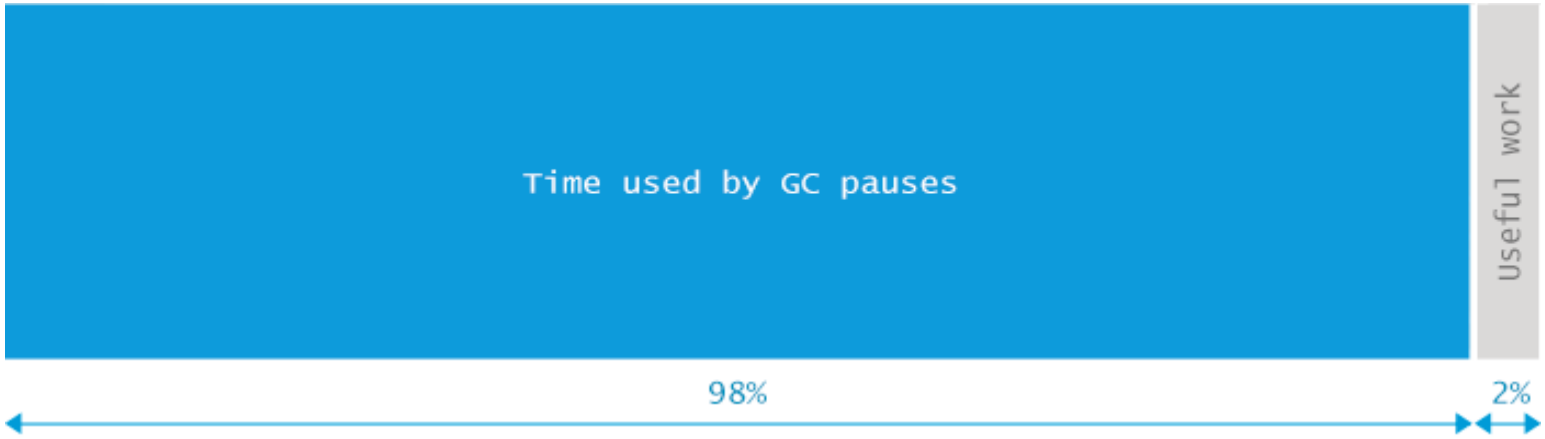
Deallocated

The deallocated state is the final step in garbage collection

java.lang.OutOfMemoryError: GC overhead limit exceeded

What is causing it?

By default the JVM is configured to throw this error if it spends more than 98% of the total time doing GC and when after the GC only less than 2% of the heap is recovered.



What is the solution?

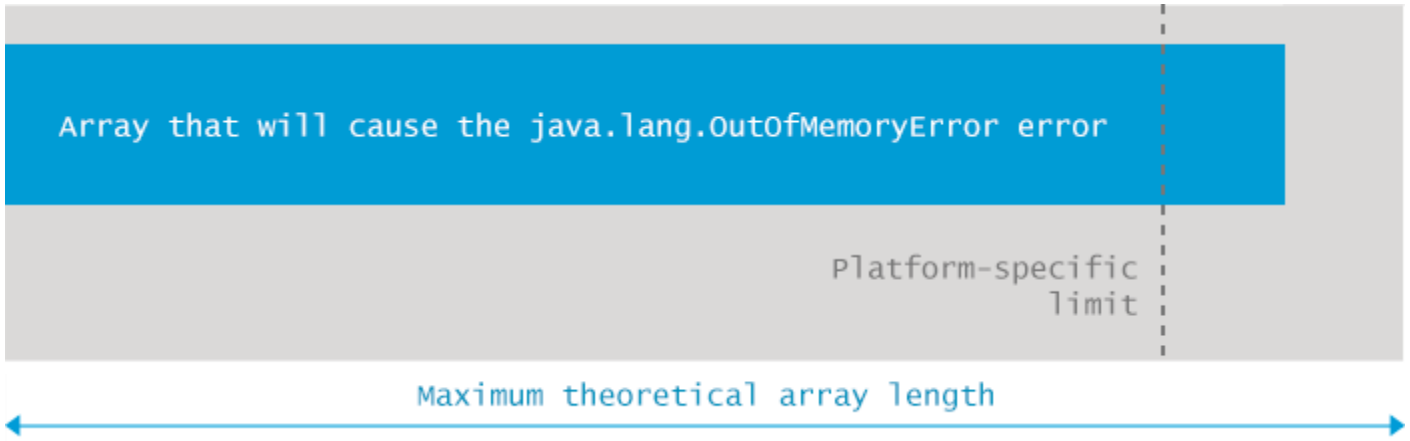
To just wished to get rid of the “java.lang.OutOfMemoryError: GC overhead limit exceeded” message, add -XX:-UseGCOverheadLimit

It is strongly suggested not to use this option though – instead of fixing the problem it just postpone.

Note : Troubleshoot the application, by identifying memory leaks

java.lang.OutOfMemoryError:
Requested array size exceeds VM limit

Java has got a limit on the maximum array size our program can allocate. The exact limit is platform-specific but is generally somewhere between 1 and 2.1 billion elements.

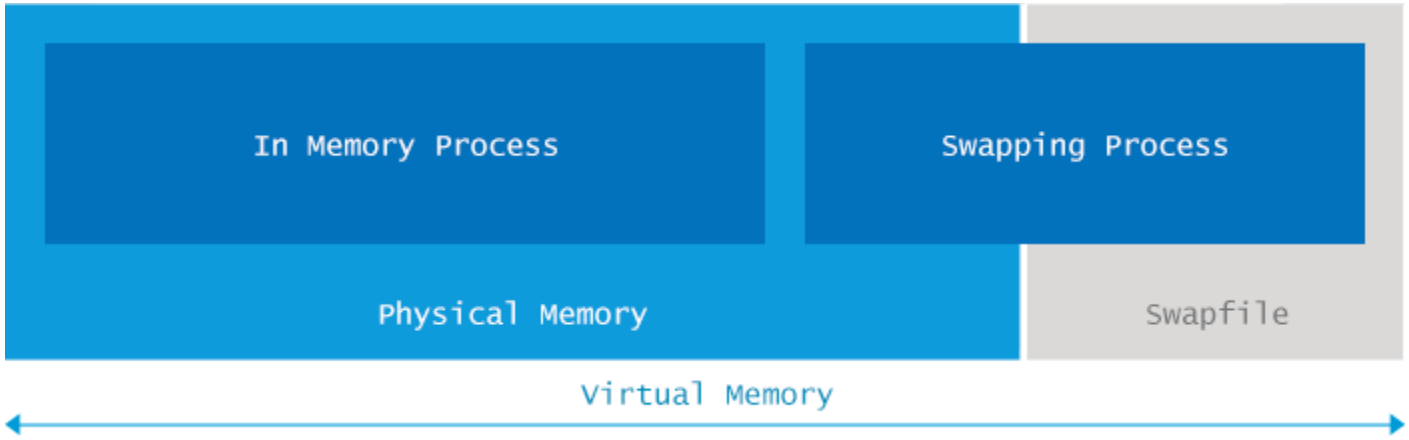


What is the solution?

Check the code base to see whether application really need arrays that large. Maybe we could reduce the size of the arrays and be done with it. Or divide the array into smaller bulks and load the data you need to work with in batches fitting into our platform limit.

java.lang.OutOfMemoryError:
Out of swap space?

In situations where the total memory requested by the JVM is larger than the available physical memory, operating system starts swapping out the content from memory to hard drive.



The *java.lang.OutOfMemoryError: Out of swap space?* error indicates that the swap space is also exhausted and the new attempted allocation fails due to the lack of both physical memory and swap space.

What is the solution?

To overcome this issue, you have several possibilities. First and often the easiest workaround is to increase swap space.

The means for this are platform specific, for example in Linux you can achieve with the following example sequence of commands, which create and attach a new swapfile sized at 640MB:

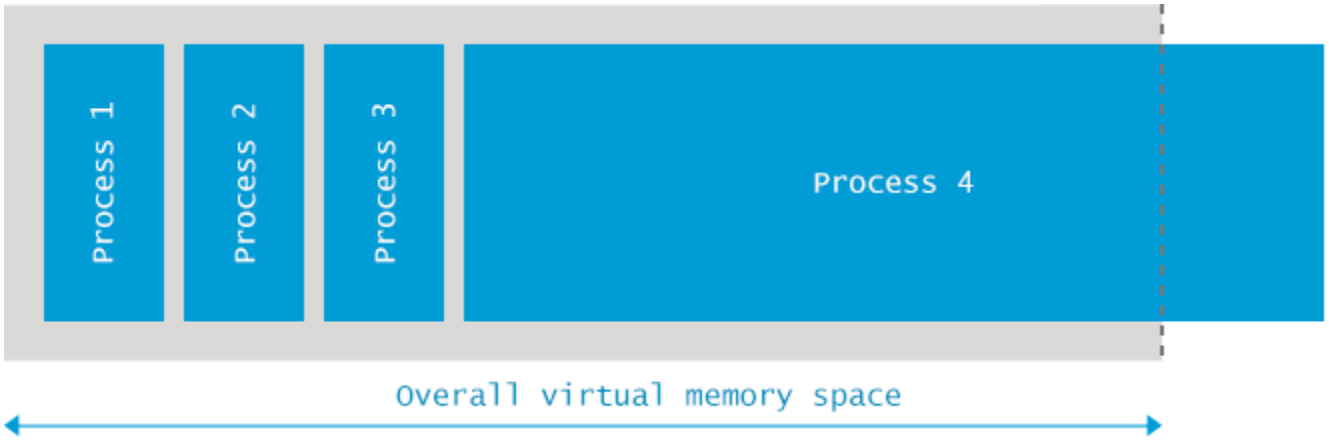
```
swapoff -a
dd if=/dev/zero of=swapfile bs=1024 count=655360
mkswap swapfile
swapon swapfile
```

Note: Running garbage collection algorithms on swapped allocations can increase the length of GC pauses by several orders of magnitude

Out of memory:
Kill process or sacrifice child

Operating systems are built on the concept of processes.

Those processes are shepherded by several kernel jobs, one of which, named “Out of memory killer”



This kernel job can destroy the processes under extremely low memory conditions.

When such a condition is detected, the Out of memory killer is activated and picks a process to kill.

The target is picked using a set of heuristics scoring all processes and selecting the one with the worst score to kill.

What is the solution?

The first and most straightforward way to overcome the issue is to migrate the system to an instance with more memory.

Other possibilities would involve fine-tuning the OOM killer, scaling the load horizontally across several small instances or reducing the memory requirements of the application.

One more solution is to increasing swap space (not recommended, GC pause takes more time)

Finding Out Why a Process Was Killed

grep -i kill /var/log/messages*
host kernel: Out of Memory: Killed process 2592 (oracle).

Configuring the OOM Killer

if we want to make our oracle process less likely to be killed by the OOM killer, we can do the following.

echo -15 > /proc/2592/oom_adj

We can make the OOM killer more likely to kill our oracle process by doing the following.

echo 10 > /proc/2592/oom_adj

If we want to exclude our oracle process from the OOM killer
echo -17 > /proc/2592/oom_adj

We can set valid ranges for oom_adj from -16 to +15, and a setting of -17 exempts a process entirely from the OOM killer. The higher the number, the more likely our process will be selected for termination if the system encounters an OOM condition.

java.lang.OutOfMemoryError: Metaspace

Metaspace usage is strongly correlated with the number of classes loaded into the JVM. Also, the metadata loaded.

What is the solution?

increase the maximum Metaspace size:
-XX:MaxMetaspaceSize=512m

OutOfMemoryError: Compressed class space

If we are working on 64-bit platforms a pointer to class metadata can be represented by a 32-bit offset (by using vm option UseCompressedClassPointers - This vm option is enabled by default).

If vm option is kept enabled then amount of space available for class metadata is fixed (i.e. specified by vm option)

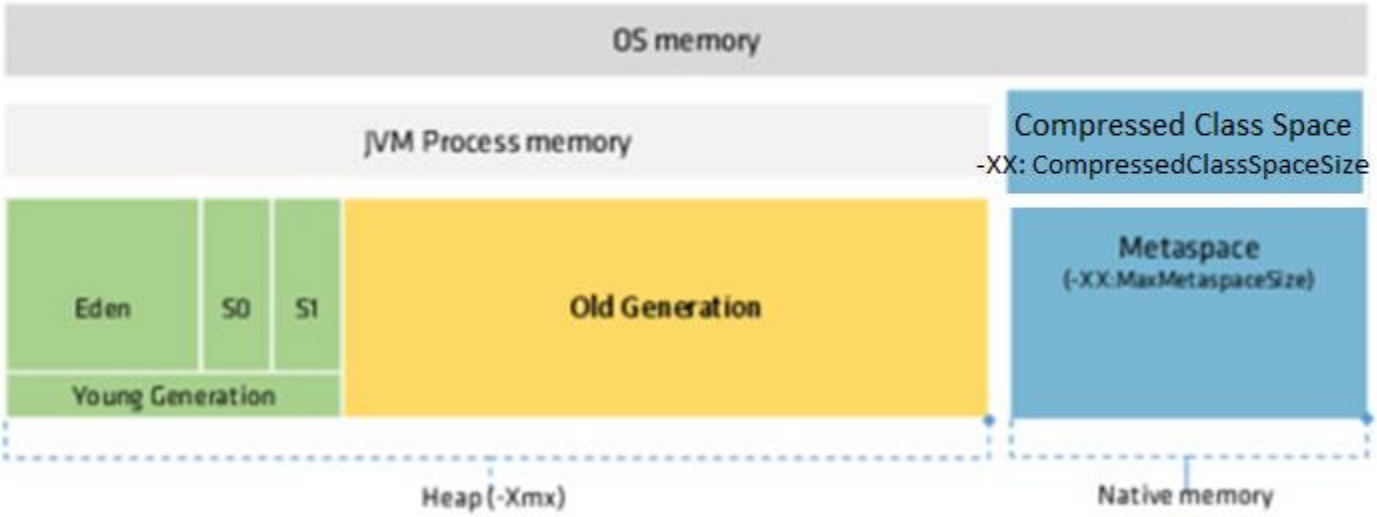
If amount of space available for class metadata is exceeds CompressedClassSpaceSize, then java.lang.OutOfMemoryError Compressed class space is thrown.

UseCompressedClassPointers vm option

-XX: CompressedClassSpaceSize=2g
It will set size of 2 gigabyte.

Now, if space available for class metadata exceeds 2 gigabyte, then java.lang.OutOfMemoryError Compressed class space is thrown.

JVM memory
JDK8



There are different type of class metadata :

- klass metadata (only it is stored in CompressedClassSpaceSize)
- other metadata (it is not stored in CompressedClassSpaceSize, it is stored in Metaspace).

Every object (except array) in memory has 2 machine word header. The first one is called **mark** word and the second one is **klass** word. Btw arrays have extra 32 bit word filled with array's length.

Mark word stores identity hashcode, bits used for garbage collection, bits used for locking.

Klass word stores ordinary object pointer (oop) to class metadata, which describes the object's layout, methods, and other type information

On 64 bit jvm every object contains 16 byte header, 2 words each 8 byte. Array has an extra 32 bit word to store array size.

```
|--------------------------------------------------------------------------------------------------------------|
|                                            Object Header (128 bits)                                          |
|--------------------------------------------------------------------------------|-----------------------------|
|                                  Mark Word (64 bits)                           |    Klass Word (64 bits)     |
|--------------------------------------------------------------------------------|-----------------------------|
| unused:25 | identity_hashcode:31 | unused:1 | age:4 | biased_lock:1 | lock:2 |    OOP to metadata object   |
|--------------------------------------------------------------------------------|-----------------------------|
| thread:54 |       epoch:2        | unused:1 | age:4 | biased_lock:1 | lock:2 |    OOP to metadata object   |
|--------------------------------------------------------------------------------|-----------------------------|
|                       ptr_to_lock_record:62                          | lock:2 |    OOP to metadata object   |
|--------------------------------------------------------------------------------|-----------------------------|
|                     ptr_to_heavyweight_monitor:62                    | lock:2 |    OOP to metadata object   |
|--------------------------------------------------------------------------------|-----------------------------|
|                                                                      | lock:2 |    OOP to metadata object   |
|--------------------------------------------------------------------------------|-----------------------------|
```

Stuck Thread

What is Stuck Thread?

A Stuck Thread is a thread which is processing a request for more than maximum time that is configured in a server.

hogging thread

A hogging thread is a thread which is taking more than usual time to complete the request and can be declared as Stuck .

In Weblogic:

A thread declared as Stuck if it runs over 600 secs

WebLogic has polar which runs every 2 secs,It checks for the number of requests completed in last two minutes

check how much times each took to complete

Then, it takes the average time of all completed request

Then multiply average time with a the value of the request taken more time.

For example –

At a particular moment, total number of completed requests in last two seconds – 4
Total time took by all 4 requests – 16 secs
Req1 took – 5 secs, Req2 took – 3 secs, Req3 took – 7 secs, Req4 took – 1 sec
Average time = 16/4 = 4 secs
7*4 = 28 secs
Now weblogic check all executed threads to see which taking more than 28 secs, if any then that thread(s) declared as Hogged Thread.

Race Condition

A race condition occurs in programming when two or more execution threads modify a shared, or critical, resource.

Race conditions can result in run time errors that are difficult to isolate and to repair.

The term "race" is used because the threads can be regarded as racing each other to complete operations on a variable or other shared resource.

Preventing Race Conditions

To prevent race conditions from occurring we must make sure that the critical section is executed as an atomic instruction. That means that once a single thread is executing it, no other threads can execute it until the first thread has left the critical section.

Race conditions can be avoided by proper thread synchronization in critical sections.

Thread synchronization can be achieved using a synchronized block of Java code.

Thread synchronization can also be achieved using other synchronization constructs like locks or atomic variables like java.util.concurrent.atomic.AtomicInteger.