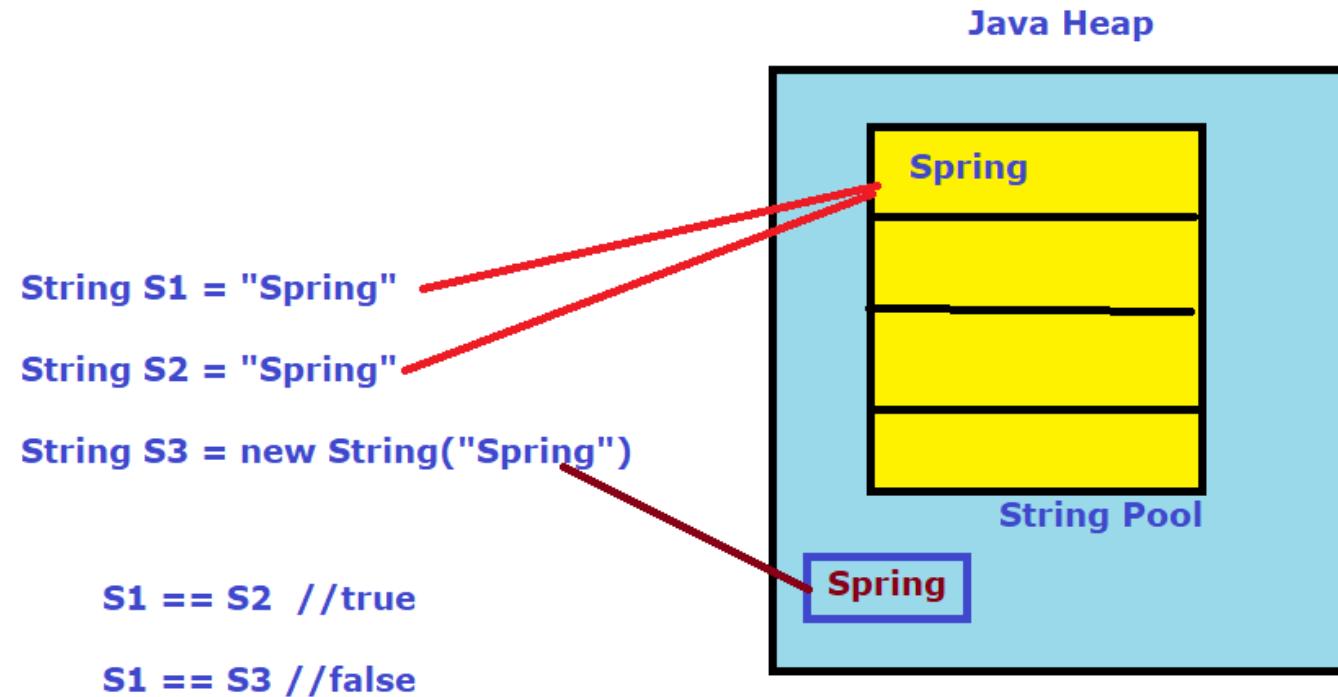


String Magic

We can create a String object using new operator :
`String s3 = new String("Spring");`

as well as providing literal value
`String s2 = "Spring";`

What is the difference ????



String Pool in java is a pool of Strings stored in Java Heap Memory

```
String str2 = new String("great");
```

Vs

```
String str3 = new String("great").intern();
```

new operator always force String class to create a new String object in heap space.

intern() method to put it into the pool or refer to other String object from string pool having same value.

Note : The intern() method helps in comparing two String objects with == operator by looking into the pre-existing pool of string literals, no doubt it is faster than equals() method.

The String pool is implemented as a fixed capacity HashMap with each bucket containing a list of strings with the same hashCode.

Default buckets are : 60013

From command-prompt:

```
java -XX:+PrintStringTableStatistics
```

Note : These Strings are loaded by the core java

Try with application:

```
java -XX:+PrintStringTableStatistics MainTest2
```

SymbolTable statistics:

Number of buckets	:	20011	=	160088 bytes, avg	8.000
Number of entries	:	12788	=	306912 bytes, avg	24.000
Number of literals	:	12788	=	494976 bytes, avg	38.706
Total footprint	:		=	961976 bytes	
Average bucket size	:	0.639			
Variance of bucket size	:	0.638			
Std. dev. of bucket size	:	0.798			
Maximum bucket size	:	6			

StringTable statistics:

Number of buckets	:	60013	=	480104 bytes, avg	8.000
Number of entries	:	1122	=	26928 bytes, avg	24.000
Number of literals	:	1122	=	91008 bytes, avg	81.112
Total footprint	:		=	598040 bytes	
Average bucket size	:	0.019			
Variance of bucket size	:	0.019			
Std. dev. of bucket size	:	0.137			
Maximum bucket size	:	2			

Lab - ex4.1-StringPool-Performance

Now, Try with:

```
java -XX:+PrintStringTableStatistics -XX:StringTableSize=120121
```

Note : compare the Elapsed Time

Tuning:

```
java -XX:+PrintFlagsFinal MainTest2
```

Check the InitialHeapSize & MaxHeapSize


```
java -XX:+PrintStringTableStatistics -  
XX:StringTableSize=120121 -XX:MaxHeapSize=600m MainTest2
```

Note : Out of memory exception comes

Now, try with

```
java -XX:+PrintStringTableStatistics -XX:StringTableSize=120121  
-XX:InitialHeapSize=1g
```

compare the Elapsed Time

Shortcut Flags

-XX:InitialHeapSize: -Xms

-XX:MaxHeapSize: -Xmx

String , StringBuffer & StringBuilder

String

Vs

StringBuffer

Vs

StringBuilder

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you concat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

No.	StringBuffer	StringBuilder
1)	StringBuffer is <i>synchronized</i> i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is <i>non-synchronized</i> i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is <i>less efficient</i> than StringBuilder.	StringBuilder is <i>more efficient</i> than StringBuffer.

String Deduplication

String Deduplication:

String class keeps string value in char[] array.

The char array is not accessible and modified from outside, this means that char array can be used safely by multiple instances of String at the same time.

Deduplication may happen during minor GC.

When GC detects another String object with the same hash code, it compares two strings char by char. When they match, one String's char array will be re-assigned to the char array of the second string and the unreferenced char array of the first string becomes available for GC.

Enable String Deduplication:

`-XX:+UseG1GC -XX:+UseStringDeduplication`

Important Points:

- ☐ This option is only available from Java 8 Update 20 JDK release.
- ☐ This feature will only work along with the G1 garbage collector.
- ☐ You need to provide both `-XX:+UseG1GC` and `-XX:+StringDeduplication` JVM options to enable this feature.
- ☐ To check if it happens in your system you can use `--XX:+PrintStringDeduplicationStatistics` parameter.
- ☐ You can control this by using `-XX:StringDeduplicationAgeThreshold=3` option to change when Strings become eligible for deduplication.

[GC concurrent-string-deduplication, 2893.3K->2672.0B(2890.7K), avg 97.3%, 0.0175148 secs]

[Last Exec: 0.0175148 secs, Idle: 3.2029081 secs, Blocked: 0/0.0000000 secs]

[Inspected: 96613]

[Skipped: 0(0.0%)]

[Hashed: 96598(100.0%)]

[Known: 2(0.0%)]

[New: 96611(100.0%) 2893.3K]

[Deduplicated: 96536(99.9%) 2890.7K(99.9%)]

[Young: 0(0.0%) 0.0B(0.0%)]

[Old: 96536(100.0%) 2890.7K(100.0%)]

[Total Exec: 452/7.6109490 secs, Idle: 452/776.3032184 secs, Blocked: 11/0.0258406 secs]

[Inspected: 27108398]

[Skipped: 0(0.0%)]

[Hashed: 26828486(99.0%)]

[Known: 19025(0.1%)]

[New: 27089373(99.9%) 823.9M]

[Deduplicated: 26853964(99.1%) 801.6M(97.3%)]

[Young: 4732(0.0%) 171.3K(0.0%)]

[Old: 26849232(100.0%) 801.4M(100.0%)]

[Table]

[Memory Usage: 2834.7K]

[Size: 65536, Min: 1024, Max: 16777216]

[Entries: 98687, Load: 150.6%, Cached: 415, Added: 252375, Removed: 153688]

[Resize Count: 6, Shrink Threshold: 43690(66.7%), Grow Threshold: 131072(200.0%)]

[Rehash Count: 0, Rehash Threshold: 120, Hash Seed: 0x0]

[Age Threshold: 3]

[Queue]

[Dropped: 0]

These are the results after running the app for 10 minutes. As we can see String Deduplication was executed 452 times and "deduplicated" 801.6 MB Strings. String Deduplication inspected 27 000 000 Strings.

When we compare memory consumption from Java 7 with the standard Parallel GC to Java 8u20 with the G1 GC and enabled String Deduplication the heap dropped approximately 50%:

The best practices for Object Allocation

These points revolve around following principles :

- ✓ Create Objects with Care
- ✓ Free up When not Required
- ✓ Help Java Garbage Collector to do Job Easily

Minimize Scope of Variables:

Minimum scope means availability of object for garbage collection quickly

- ❑ variable defined at method level, after method execution is no longer referenced
- ❑ For Class scope, garbage collector waits until all references of class are removed
- ❑ Request scope, Session scope, Application scope, Conversation scope etc.,

Note : Optimum scope of an attribute is needed.

Initialize When You Actually Need:

This is allocating memory just before you actually use the object first time.

Sometimes it is needed to declare some of the attributes at the beginning of method.

While declaring such attributes, we tend to initialize those during declaration itself.

In this scenario if anything goes wrong (e.g. exception occurs) before first use of this variable, then this is unnecessary initialization and waste of memory.

Allocate Only Required Memory:

The best example of this scenario is Vector (`java.util.Vector`).

This class has default initial size 10 on initialization.

Once we go on adding objects to this collection and there is need to add addition memory, this self expanding collection adds another 10 memory spaces to itself.

- ❑ Initialize with required size

Do not Declare in Loops:

This is another common mistake found in most of the programs running out of memory.

Declaring and initializing variables inside loop.

For each iteration of the loop, this variable instance is created in memory is allocated through initialization

Memory Leaks Possibilities Through Soft References in Collections:

In following example, can you spot the memory leak problem?

```
private Object[] elements;  
// Some code to add elements  
elements[--size];  
return elements;
```

Here the collection object at elements[size] location is not garbage collected because of the soft reference to this collection after reduction of size. Following code can fix this problem.

```
private Object[] elements;  
// Some code to add elements  
elements[--size];  
elements[size] = null;  
return elements;
```

Mutating Operations on String:

This is something special about String, all operations on string result in another string object.

Mostly we have string concatenation operation carried out, it can be to generate a long dynamic query

Use other alternatives which do not mutate like this. Here StringBuffer/ StringBuilder can be used to concatenate string.

Clean up Heavy Objects:

Make it little bit easy for garbage collector.

Set the heavy objects to null.

This will free up the references of heavy object and make that memory available immediately.

Difference between final, finally and finalize

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

finalize() Method Call Has Advantages and Disadvantages:

This method is available in object class.

Java suggests that this method should be called when you want to instruct garbage collector to clean up the object.

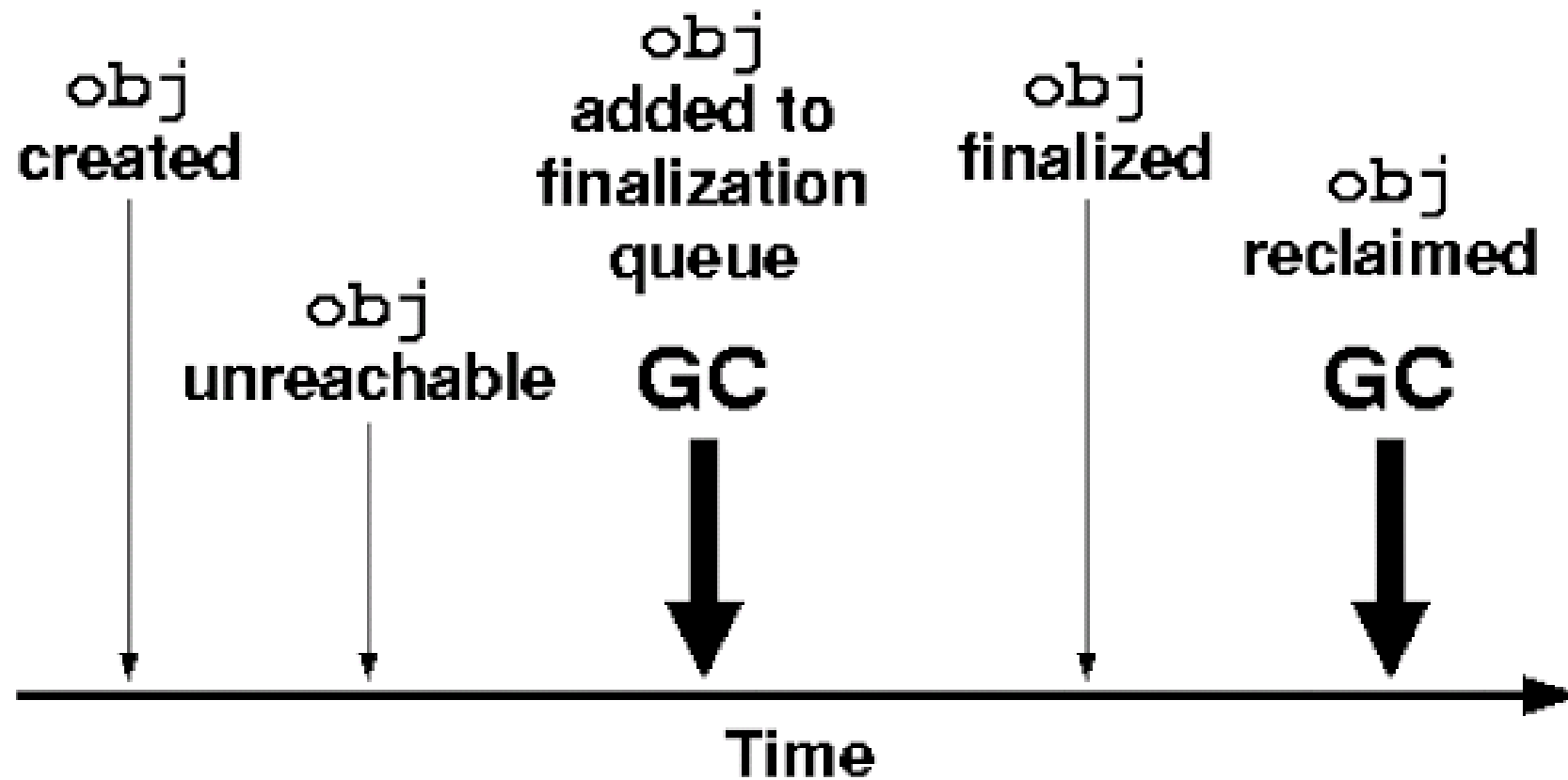
This method does not trigger the clean up immediately.
It just intimates JVM that the object is ready to free up memory.

Garbage collection happens at the JVM's/garbage collector's choice.

But too many such calls can result in triggering of garbage collection frequently and resulting in reducing performance of main program.

Hence it is better to reduce usage of memory instead of increasing load on cleanup operation.

Lifetime of finalizable object obj



Creation Of A Finalizer

The JVM will ignore a trivial `finalize()` method (e.g. one which just returns without doing anything, like the one defined in the `Object` class). Otherwise, if an instance is being created, and that instance has a non-trivial `finalize()` method defined or inherited, then the JVM will do the following:

The JVM will create the instance

The JVM will also create an instance of the `java.lang.ref.Finalizer` class, pointing to that object instance just created

The `java.lang.ref.Finalizer` class holds on to the `java.lang.ref.Finalizer` instance that was just created (so that it is kept alive, otherwise nothing would keep it alive and it would be GCed at the next GC).

The First GC

Eden gets full, and a minor GC happens.

Firstly, instead of a bunch of objects which aren't being referenced, we have lots of objects which are referenced from Finalizer objects, which in turn are referenced from the Finalizer class. So everything stays alive!

The GC will copy everything(all Test Objects) into the survivor space. And if that isn't big enough to hold all of the objects, it will have to move some to the old generation (which has a much more expensive GC cycle).

Note : Assume that there is a Test class with finalize() method

The First GC (continued)

Because the GC recognizes that nothing else points to the Test instances apart from the Finalizers

GC adds each of those Finalizer objects to the reference queue at `java.lang.ref.Finalizer.ReferenceQueue`

Now the GC is finally finished, having done quite a bit more work

Test instances are hanging around, spread all over the place in survivor space and the old generation too;

Finalizer instances are hanging around too, as they are still referenced from the Finalizer class.

The GC is finished, but nothing seems to have been cleared up!

The Finalizer Thread

Now that the minor GC is finished, the application threads start up again

In that same `java.lang.ref.Finalizer` class is an inner class called `FinalizerThread`, which starts the "Finalizer" daemon thread when the `java.lang.ref.Finalizer` is loaded in to the JVM.

"Finalizer" daemon thread sits in a loop which is blocked waiting for something to become available to be popped from the `java.lang.ref.Finalizer.ReferenceQueue` queue.

ex :

```
for(;;)
{
    Finalizer f = java.lang.ref.Finalizer.ReferenceQueue.remove();
    f.get().finalize();
}
```

The Second GC

Finalizer objects will get off the queue, and the Test instances they point to will get their finalize() methods called.

"Finalizer" daemon thread also removes the reference from the Finalizer class to that Finalizer instance it just processed - remember, that is what was keeping the Finalizer instance alive.

Now nothing points to the Finalizer instance, and it can be collected in the next GC - as can the Test instance since nothing else points to that.

So eventually, after all that, another GC will happen. And this time round, those Finalizer objects that have been processed will get cleared out by the GC. That's the end of the finalizer lifecycle

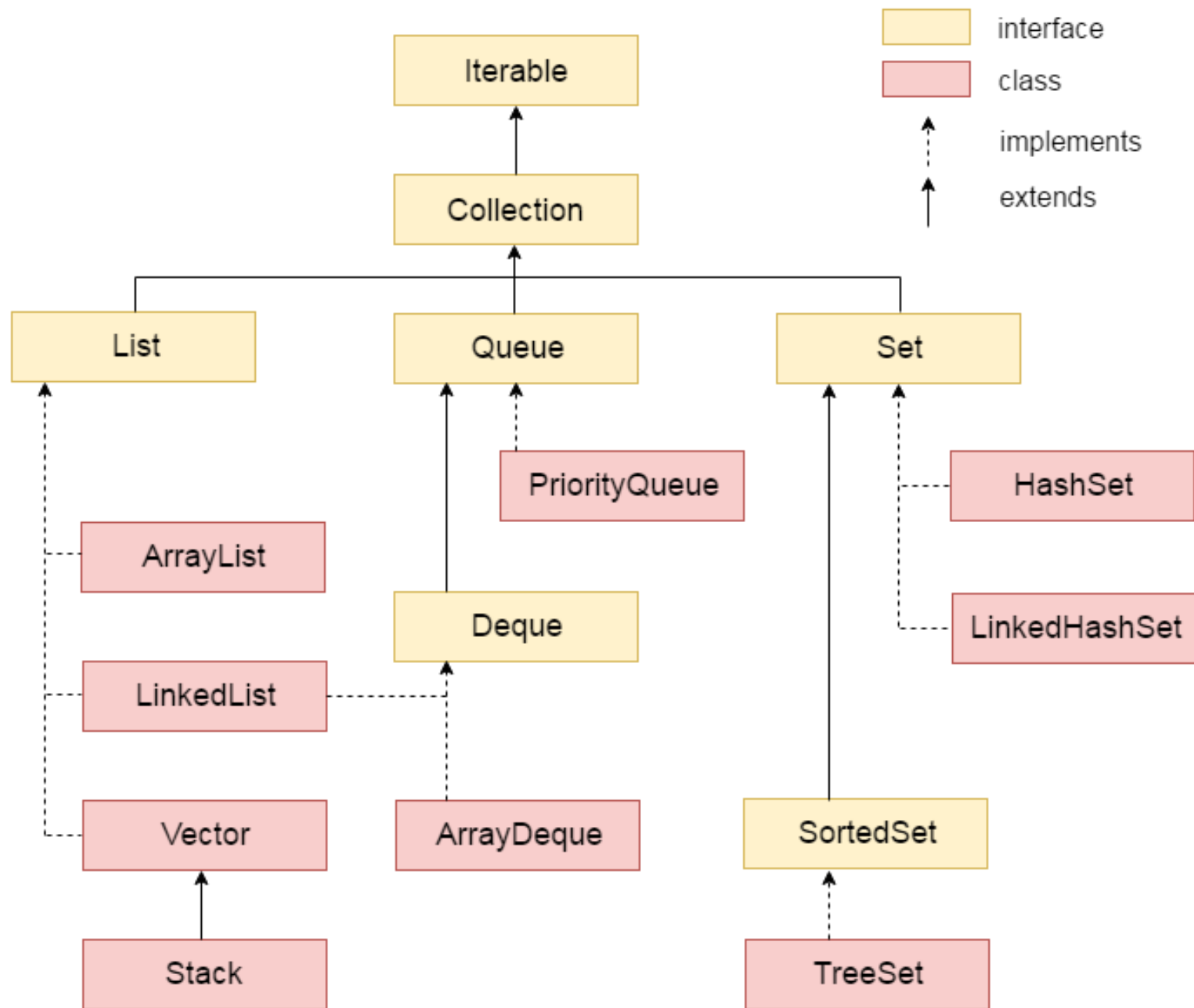
Design and Architecture requiring more Memory:

The design and architecture should be such that you have to minimum data loaded in memory.

Sometimes you have to load data in memory(cache) to avoid repeated calls to persistent store/source of data, but it is better to optimize

- > Eager vs Lazy loading
- > fetching strategies
- > Cache eviction policies

Java Collection API



ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

ArrayList	LinkedList
1) ArrayList internally uses dynamic array to store the elements.	LinkedList internally uses doubly linked list to store the elements.
2) Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3) ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4) ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

ArrayList and Vector both implements List interface and maintains insertion order.

ArrayList	Vector
1) ArrayList is not synchronized .	Vector is synchronized .
2) ArrayList increments 50% of current array size if number of element exceeds from its capacity[default size=10]	Vector increments 100% means doubles the array size if total number of element exceeds than its capacity[default size=10]. However, we can configure the capacity increment.
3) ArrayList is fast because it is non-synchronized.	Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.

The `java.util.Deque` interface is a subtype of the `java.util.Queue` interface. It represents a queue where you can insert and remove elements from both ends of the queue. Thus, "Deque" is short for "Double Ended Queue".

There are two implementations of Deque :

`java.util.ArrayDeque`
`java.util.LinkedList`

Inserting & Reading sequentially from Collection prefer
LinkedList/ArrayList

Inserting & Reading/Deleting by Search/equals from Collection prefer
HashSet

Inserting, ArrayList & LinkedList performs best while HashSet takes
double the time

Reading, HashSet performs best while ArrayList & LinkedList are
marginally less

Deleting, HashSet performs better than ArrayList & ArrayList performs
better than LinkedList. LinkedList is slow because of sequential search

Differences Between ArrayList And LinkedList In Java:

	ArrayList	LinkedList
Structure	ArrayList is an index based data structure where each element is associated with an index.	Elements in the LinkedList are called as nodes, where each node consists of three things – Reference to previous element, Actual value of the element and Reference to next element.
Insertion And Removal	Insertions and Removals in the middle of the ArrayList are very slow. Because after each insertion and removal, elements need to be shifted.	Insertions and Removals from any position in the LinkedList are faster than the ArrayList. Because there is no need to shift the elements after every insertion and removal. Only references of previous and next elements are to be changed.
	Insertion and removal operations in ArrayList are of order $O(n)$.	Insertion and removal in LinkedList are of order $O(1)$.

Retrieval(Searching or getting an element)

Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based.

Retrieval of elements in LinkedList is very slow compared to ArrayList. Because to retrieve an element, you have to traverse from beginning or end (Whichever is closer to that element) to reach that element.

Retrieval operation in ArrayList is of order of $O(1)$.

Retrieval operation in LinkedList is of order of $O(n)$.

Random Access

ArrayList is of type Random Access. i.e elements can be accessed randomly.

LinkedList is not of type Random Access. i.e elements can not be accessed randomly. you have to traverse from beginning or end to reach a particular element.

Usage	ArrayList can not be used as a Stack or Queue.	LinkedList, once defined, can be used as ArrayList, Stack, Queue, Singly Linked List and Doubly Linked List.
Memory Occupation	ArrayList requires less memory compared to LinkedList. Because ArrayList holds only actual data and it's index.	LinkedList requires more memory compared to ArrayList. Because, each node in LinkedList holds data and reference to next and previous elements.
When To Use	If your application does more retrieval than the insertions and deletions, then use ArrayList.	If your application does more insertions and deletions than the retrieval, then use LinkedList.

Difference between HashSet and ArrayList in Java

- 1. Implementation :** HashSet implements Set interface in Java while ArrayList implements List interface.
- 2. Internal implementation:** HashSet is backed by an HashMap while ArrayList is backed by an Array.
- 3. Ordering :** HashSet is an unordered collection and doesn't maintain any order while ArrayList maintains the order of the object in which they are inserted.
- 4. Duplicates :** HashSet doesn't allow duplicates though ArrayList allows duplicate values.
- 5. Index based :** The fifth difference between HashSet and ArrayList is that ArrayList is index based you can retrieve object by calling get(index) or remove objects by calling remove(index) while HashSet is completely object based. HashSet also doesn't provide get() method.

HashSet vs ArrayList contains performance

The ArrayList uses an array for storing the data. The ArrayList.contains will be of $O(n)$ complexity. So essentially searching in array again and again will have $O(n^2)$ complexity.

While HashSet uses hashing mechanism for storing the elements into their respective buckets. The operation of HashSet will be faster for long list of values. It will reach the element in $O(1)$.

We can decide when to use ArrayList or HashSet based on the performance and properties (duplicates, ordering);

Lambdas vs Anonymous Inner Classes

Performance

At runtime anonymous inner classes require class loading, memory allocation and object initialization and invocation of a non-static method while lambda expression is pure compile time activity and don't incur extra cost during runtime.

So performance of lambda expression is better as compare to anonymous inner classes.

Types Of References In Java

Depending upon how objects are garbage collected, references to those objects in java are grouped into 4 types.

They are:

- 1) Strong References
- 2) Soft References
- 3) Weak References
- 4) Phantom References

Strong References

Any object in the memory which has active **strong reference** is not eligible for garbage collection.

```
A a = new A();
```

Soft References

The objects which are softly referenced will not be garbage collected (even though they are available for garbage collection) until JVM badly needs memory.

These objects will be cleared from the memory only if JVM runs out of memory. You can create a soft reference to an existing object by using **java.lang.ref.SoftReference** class.

```
A a = new A();    //Strong Reference
```

```
//Creating Soft Reference to A-type object to which 'a' is also pointing
```

```
SoftReference<A> softA = new SoftReference<A>(a);
```

a = null; //Now, A-type object to which 'a' is pointing earlier is eligible for garbage collection. But, it will be garbage collected only when JVM needs memory.

```
a = softA.get(); //You can retrieve back the object which has been softly referenced
```

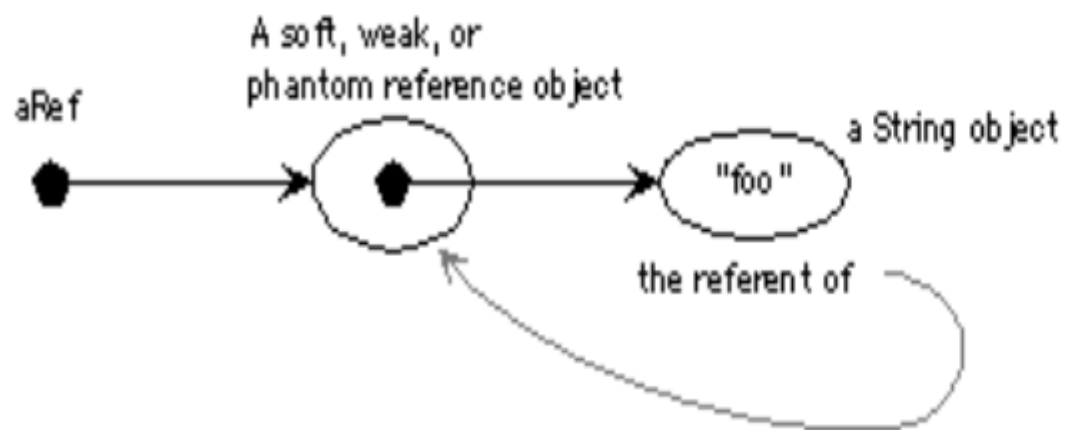
SOFT REFERENCES

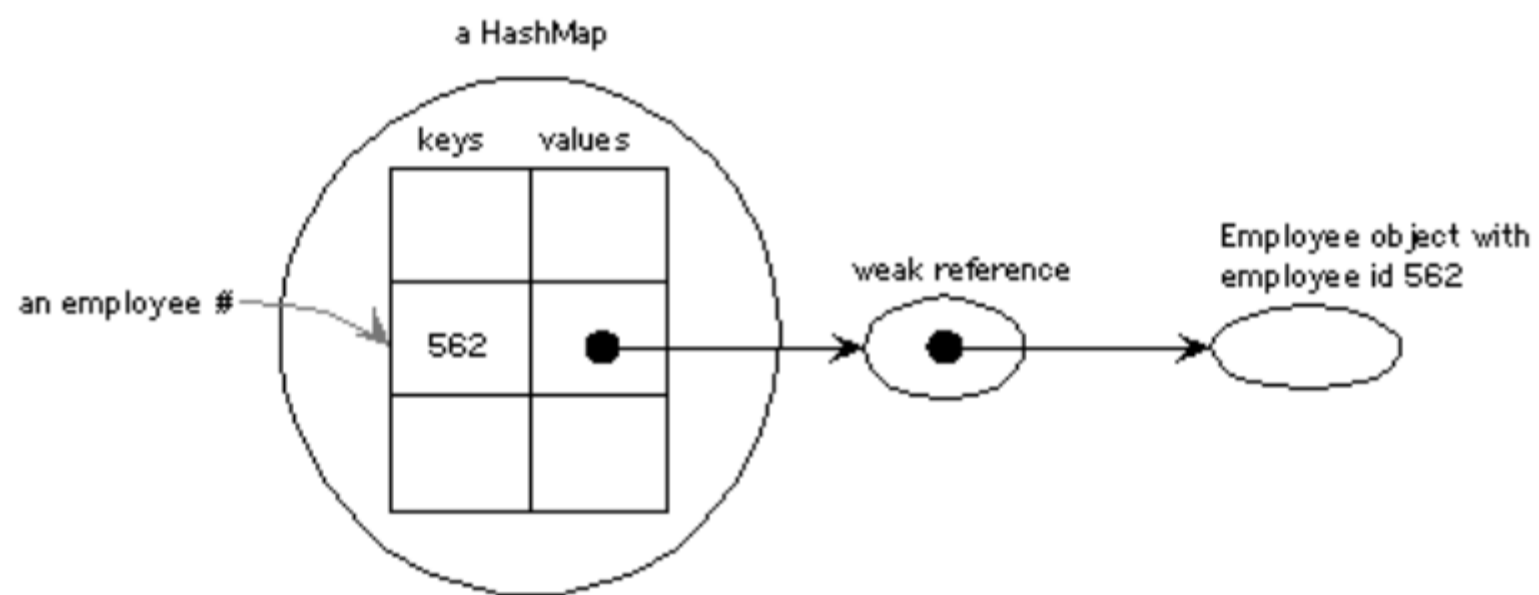
Soft references are good for providing caches of objects that can be garbage collected when the Java Virtual Machine (JVM) is running low on memory.

In particular, the JVM guarantees a couple of useful things:

- ❑ It will not garbage collect an object as long as there are normal, strong references to it.
- ❑ It will not throw an `OutOfMemoryError` until it has "cleaned up" all softly reachable objects (objects not reachable by strong references but reachable through one or more soft references).
- ❑ This allows us to use soft references to refer to objects that we could afford to have garbage collected, but that are convenient to have around until memory becomes tight.


```
Reference aRef = new SoftReference( new String( 'foo' ) );
```





org.apache.commons.pool2.impl

Class SoftReferenceObjectPool<T>

java.lang.Object

org.apache.commons.pool2.BaseObject

org.apache.commons.pool2.BaseObjectPool<T>

org.apache.commons.pool2.impl.SoftReferenceObjectPool<T>

Weak References

JVM ignores the **weak references**. That means objects which has only weak references are eligible for garbage collection. They are likely to be garbage collected when JVM runs garbage collector thread. JVM doesn't show any regard for weak references.

```
A a = new A();    //Strong Reference
```

```
    //Creating Weak Reference to A-type object to which 'a' is also pointing.
```

```
WeakReference<A> weakA = new WeakReference<A>(a);
```

```
    a = null;    //Now, A-type object to which 'a' is pointing earlier is available  
for garbage collection.
```

```
    a = weakA.get();    //You can retrieve back the object which has been  
weakly referenced.
```

Phantom References

The objects which are being referenced by **phantom references** are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called '**reference queue**'. You can't retrieve back the objects which are being phantom referenced.

```
A a = new A();    //Strong Reference
                //Creating ReferenceQueue
```

```
ReferenceQueue<A> refQueue = new ReferenceQueue<A>();
```

```
//Creating Phantom Reference to A-type object to which 'a' is also pointing
PhantomReference<A> phantomA = new PhantomReference<A>(a, refQueue);
```

a = null; //Now, A-type object to which 'a' is pointing earlier is available for garbage collection. But, this object is kept in 'refQueue' before removing it from the memory.

```
a = phantomA.get(); //it always returns null
```

```
refQueue.remove();    // This will block till it is GCd
```

Soft vs Weak vs Phantom References				
Type	Purpose	Use	When GCed	Implementing Class
Strong Reference	An ordinary reference. Keeps objects alive as long as they are referenced.	normal reference.	Any object not pointed to can be reclaimed.	default
Soft Reference	Keeps objects alive provided there's enough memory.	to keep objects alive even after clients have removed their references (memory-sensitive caches), in case clients start asking for them again by key.	After a first gc pass, the JVM decides it still needs to reclaim more space.	<code>java.lang.ref.SoftReference</code>
Weak Reference	Keeps objects alive only while they're in use (reachable) by clients.	Containers that automatically delete objects no longer in use.	After gc determines the object is only weakly reachable	<code>java.lang.ref.WeakReference</code> <code>java.util.WeakHashMap</code>
Phantom Reference	Lets you clean up after finalization but before the space is reclaimed (replaces or augments the use of <code>finalize()</code>)	Special clean up processing		

Phantom references are safe way to know an object has been removed from memory. For instance, consider an application that deals with large images. Suppose that we want to load a big image in to memory when large image is already in memory which is ready for garbage collected.

In such case, we want to wait until the old image is collected before loading a new one. Here, the phantom reference is flexible and safely option to choose.

The reference of the old image will be enqueued in the ReferenceQueue once the old image object is finalized. After receiving that reference, we can load the new image in to memory.

Phantom reference are the weakest level of reference in Java; in order from strongest to weakest, they are: strong, soft, weak, *phantom*.

Java Lock vs synchronized

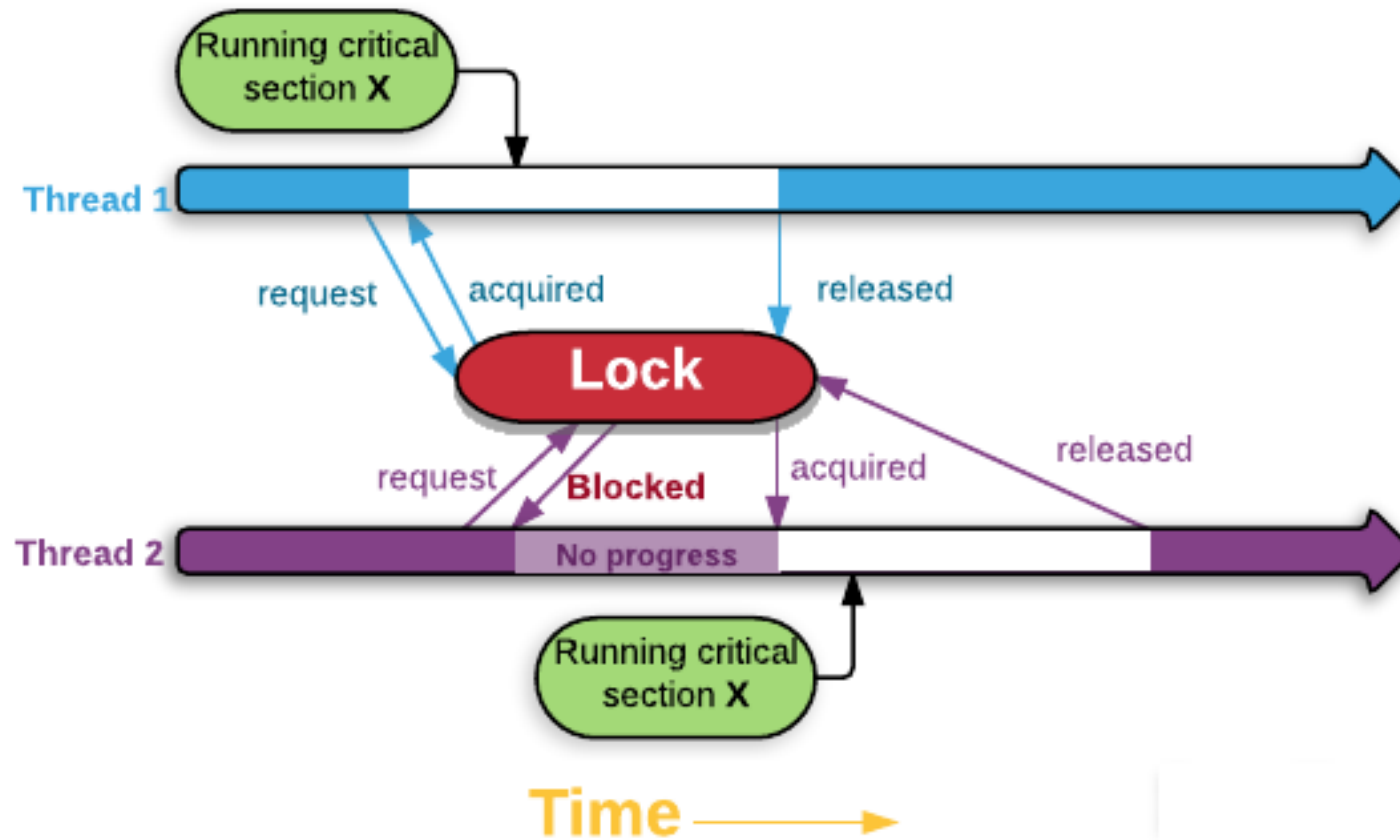
Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

In the Java virtual machine, every object and class is logically associated with a monitor. To implement the mutual exclusion capability of monitors, a lock is associated with each object and class.

Mutual Exclusion of Critical Section



Why use Synchronization?

The synchronization is mainly used to

- To prevent thread interference.

- To prevent consistency problem.

Ex : The classic example : Joint account holders are trying to withdraw the amount at the same time [money transfer Online , withdraw cash at ATM]

```
private synchronized void makeWithdrawal(int amt) {  
    if (acct.getBalance() >= amt) {  
        System.out.println(Thread.currentThread().getName() + " is going to withdraw");  
        try {  
            Thread.sleep(100);  
        } catch (InterruptedException ex) {  
        }  
        acct.withdraw(amt);  
        System.out.println(Thread.currentThread().getName() + " completes the withdrawal");  
    } else {  
        System.out.println("Not enough in account for " + Thread.currentThread().getName() + "  
    }  
}
```

There are two kinds of locks

Those with synchronized blocks, and those which use `java.util.concurrent.Lock`.

Java Lock vs synchronized

Java Lock API provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use `tryLock()` to make sure thread waits for specific time only.

Synchronization code is much cleaner and easy to maintain whereas with Lock we are forced to have try-finally block to make sure Lock is released even if some exception is thrown between `lock()` and `unlock()` method calls.

synchronization blocks or methods can cover only one method whereas we can acquire the lock in one method and release it in another method with Lock API.

synchronized keyword doesn't provide fairness whereas we can set fairness to true while creating ReentrantLock object so that longest waiting thread gets the lock first. We can create different conditions for Lock and different thread can await() for different conditions.

ReentrantLock

The class ReentrantLock is a mutual exclusion lock with the same basic behavior as the implicit monitors accessed via the synchronized keyword but with extended capabilities (like the longest waiting thread gets the lock first)

```
ReentrantLock lock = new ReentrantLock();  
int count = 0;
```

```
void increment() {  
    lock.lock();  
    try {  
        count++;  
    } finally {  
        lock.unlock();  
    }  
}
```


ReadWriteLock

The interface `ReadWriteLock` specifies another type of lock maintaining a pair of locks for read and write access.

The idea behind read-write locks is that it's usually safe to read mutable variables concurrently as long as nobody is writing to this variable. So the read-lock can be held simultaneously by multiple threads as long as no threads hold the write-lock.

This can improve performance and throughput in case that reads are more frequent than writes.

In the below code, both read tasks are executed in parallel and print the result simultaneously to the console

```
Runnable readTask = () -> {  
    lock.readLock().lock();  
    try {  
        System.out.println(map.get("foo"));  
        sleep(1);  
    } finally {  
        lock.readLock().unlock();  
    }  
};  
  
executor.submit(readTask);  
executor.submit(readTask);
```

The below example first acquires a write-lock in order to put a new value to the map after sleeping for one second.

```
ExecutorService executor = Executors.newFixedThreadPool(2);  
Map<String, String> map = new HashMap<>();  
ReadWriteLock lock = new ReentrantReadWriteLock();
```

```
executor.submit(() -> {  
    lock.writeLock().lock();  
    try {  
        sleep(1);  
        map.put("foo", "bar");  
    } finally {  
        lock.writeLock().unlock();  
    }  
});
```

Note: read tasks have to wait the whole second until the write task has finished. After the write lock has been released both read tasks are executed in parallel

ReadWriteLock Locking Rules

The rules by which a thread is allowed to lock the ReadWriteLock either for reading or writing the guarded resource, are as follows:

Read Lock

If no threads have locked the ReadWriteLock for writing, and no thread have requested a write lock (but not yet obtained it). Thus, multiple threads can the lock for reading.

Write Lock

If no threads are reading or writing.
Thus, only one thread at a time can lock the lock for writing.

StampedLock

StampedLock which also support read and write locks.

In contrast to ReadWriteLock the locking methods of a StampedLock return a stamp represented by a long value.

We can use these stamps to either release a lock or to check if the lock is still valid.

Additionally stamped locks support another lock mode called optimistic locking.

```
ExecutorService executor = Executors.newFixedThreadPool(2);  
Map<String, String> map = new HashMap<>();  
StampedLock lock = new StampedLock();
```

```
executor.submit(() -> {  
    long stamp = lock.writeLock();  
    try {  
        sleep(1);  
        map.put("foo", "bar");  
    } finally {  
        lock.unlockWrite(stamp);    } });
```

```
Runnable readTask = () -> {  
    long stamp = lock.readLock();  
    try {  
        System.out.println(map.get("foo"));  
        sleep(1);  
    } finally {  
        lock.unlockRead(stamp);    } };
```

```
executor.submit(readTask);  
executor.submit(readTask);
```

Optimistic Locking:

An optimistic read lock is acquired by calling `tryOptimisticRead()` which always returns a stamp without blocking the current thread, no matter if the lock is actually available.

If there's already a write lock active the returned stamp equals zero. You can always check if a stamp is valid by calling `lock.validate(stamp)`.

```
ExecutorService executor = Executors.newFixedThreadPool(2);  
StampedLock lock = new StampedLock();
```

```
executor.submit(() -> {  
    long stamp = lock.tryOptimisticRead();  
    try {  
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));  
        sleep(1);  
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));  
        sleep(2);  
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));  
    } finally {  
        lock.unlock(stamp);    } });
```

```
executor.submit(() -> {  
    long stamp = lock.writeLock();  
    try {  
        System.out.println("Write Lock acquired");  
        sleep(2);  
    } finally {  
        lock.unlock(stamp);  
        System.out.println("Write done");    } });
```


Semaphores

Concurrency API also supports counting semaphores. Whereas locks usually grant exclusive access to variables or resources, a semaphore is capable of maintaining whole sets of permits.

This is useful in different scenarios where you have to limit the amount concurrent access to certain parts of your application.

Semaphores – Restrict the number of threads that can access a resource. Example, limit max 10 connections to access a file simultaneously.

Mutex – Only one thread to access a resource at once. Example, when a client is accessing a file, no one else should have access the same file at the same time.

```
ExecutorService executor = Executors.newFixedThreadPool(10);
```

```
Semaphore semaphore = new Semaphore(5);
```

```
Runnable longRunningTask = () -> {  
    boolean permit = false;  
    try {  
        permit = semaphore.tryAcquire(1, TimeUnit. MILLISECONDS);  
        if (permit) {  
            System.out.println("Semaphore acquired");  
            sleep(5);  
        } else {  
            System.out.println("Could not acquire semaphore");  
        }  
    } catch (InterruptedException e) {  
        throw new IllegalStateException(e);  
    } finally {  
        if (permit) {  
            semaphore.release();  
        }  
    }  
}
```

```
IntStream.range(0, 10)  
    .forEach(i -> executor.submit(longRunningTask));
```

CPU | Threads | Deadlocks | Memory | **Monitor Usage** | Exceptions | Performance Charts | Even

Group by **C** Monitor class then group by **G** Waiting/blocked thread ☐ Show blocked threads only

⚙️ 🔍	Name
[-] Monitor of class C demo.Table	
[-] was waited by thread G Thread-2 native ID: 0x2B8 group: 'main'	
that was blocked by thread G Thread-1 native ID: 0x2BC group: 'main'	

⚙️ 🔍	Reverse Call Tree
[-] ↩ demo.Table.printTable(int) TestSynchronizedBlock1.java:7	
↩ demo.MyThread2.run() TestSynchronizedBlock1.java:33	

Reducing Locks

Try to use the `concurrent.locks` package which provide extended capabilities compared to `synchronized` regions or method calls for timed lock acquisition, fairness among threads etc.

Avoid `synchronized` methods. Go with smaller `synchronized` regions whenever possible. Try to use `synchronizing` on a specific object.

Increase the number of resources, where access to them leads to locking.

Reducing Locks (contd)

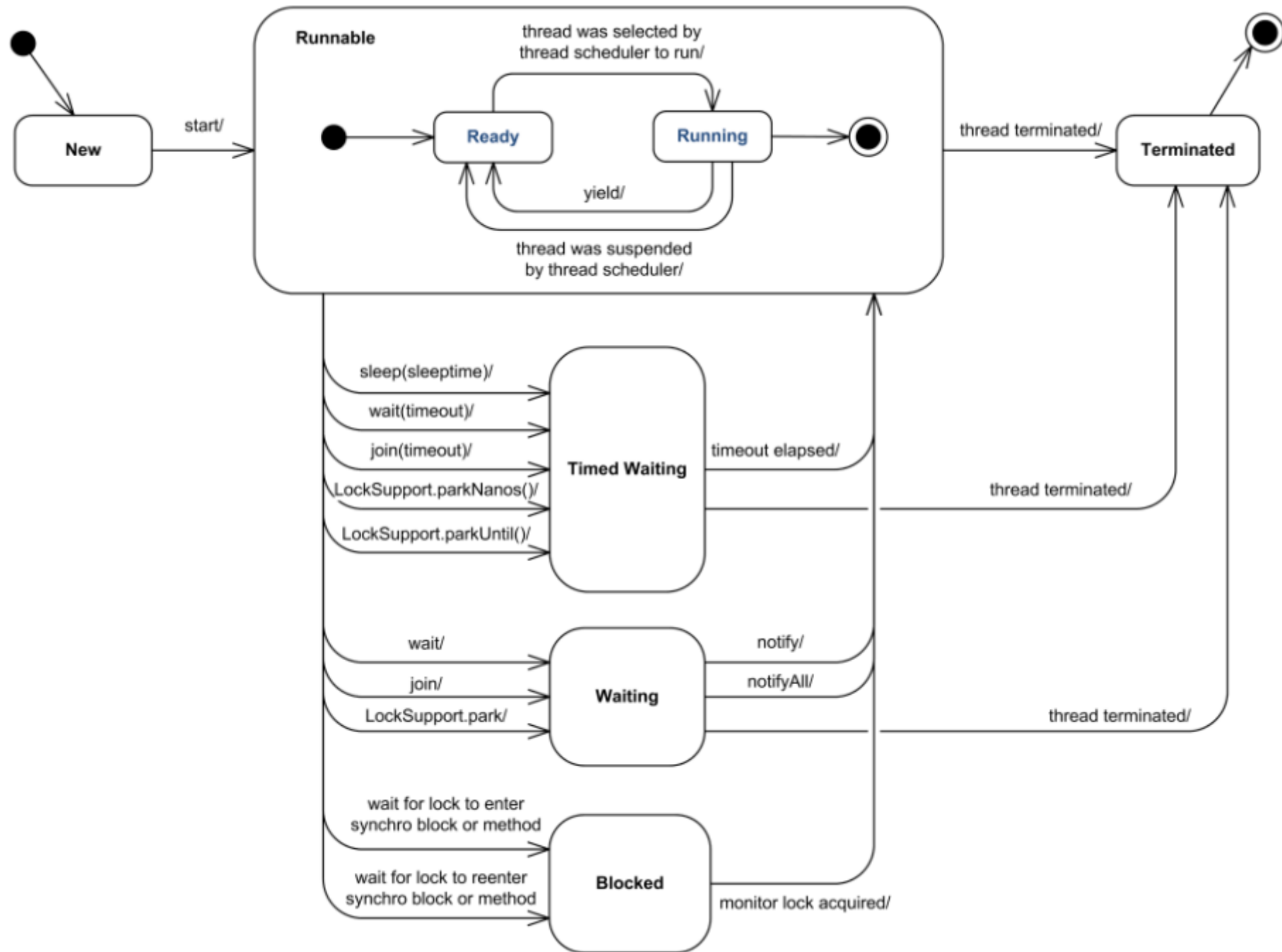
Try to cache resources if each call to create the resource requires synchronized calls.

Try to avoid the synchronized call entirely if possible by changing the logic of execution.

Try to control the order of locking in cases of deadlocks. For example, every thread has to obtain LockA before obtaining LockB and not mix up the order of obtaining locks.

If the owner of the lock has to wait for an event, then do a synchronization wait on the lock which would release the lock and put the owner itself on the blocked list for the lock automatically so other threads can obtain the lock and proceed.

Understanding Thread States



- **NEW** : A thread has not started yet.
- **RUNNABLE** : Thread is running state but it can be in state of waiting.
- **BLOCKED** : Thread is waiting to acquire monitor lock to enter into a synchronized block/method after calling `Object.wait()`
- **WAITING** : A thread is in waiting state due to calling one of the following methods
 - **Object.wait()** : It causes current thread to wait until it been notified by method *notify()* or *notifyAll()*.
 - **Object.join()** : Waits for current thread to die.
 - **LockSupport.park** : Disables the current thread for thread scheduling purposes unless the permit is available.

• **TIMED_WAITING** : Current thread is waits for another thread for specified time to perform the action.

- **Thread.sleep (long timeInMilliSecond)** : Makes current thread to cease the execution for specified time.
- **Object.wait (long timeInMilliSecond)** : Causes current thread to wait for specified time until time elapsed or get notified by notify() or notifyAll().
- **Thread.join (long millis)** : Current thread waits for specified time to die the thread.
- **LockSupport.parkNanos (long nanoSeconds)** : Disables the current thread for thread scheduling purposes, for up to the specified waiting time, unless the permit is available.
- **LockSupport.parkUntil ()**

NEW : A thread is in a new state when it has just been created and the start() method hasn't been invoked

RUNNING : A thread is in a runnable state when the start method has been invoked and the JVM actively schedules it for execution

BLOCKED / WAITING FOR MONITOR ENTRY: Thread enters this state when it isn't able to acquire the necessary monitors to enter a synchronized block. A thread waits for other threads to relinquish the monitor. A thread in this state isn't doing any work

WAITING : A thread enters this state when it relinquishes its control over the monitor by calling `Object.wait()`. It waits till the time other threads invoke `Object.notify()` or `Object.notifyAll()`. In this state too the thread isn't doing any work

TIMED WAITING : A thread enters this state when it relinquishes its control over the monitor by calling `Object.wait(long)`. It waits till the time other threads invoke `Object.notify()` / `Object.notifyAll()` or when the wait time expires. In this state too the thread isn't doing any work

SLEEPING / WAITING ON CONDITION: A thread sleeps when it calls `Thread.sleep(long)`. As the name suggests the thread isn't doing in this state either

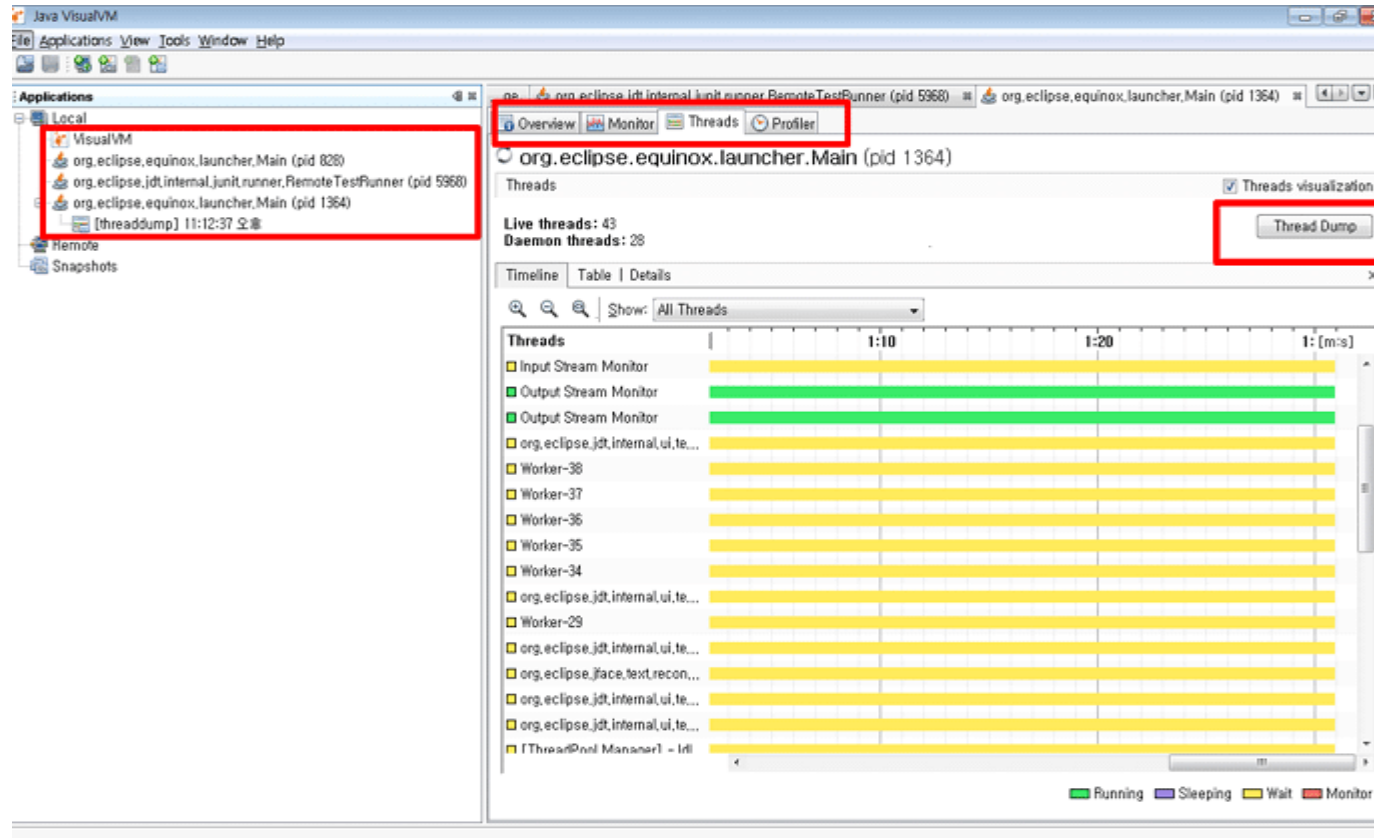
Terminated

Other

Analysing Thread Dumps

Thread Dump Using jVisualVM

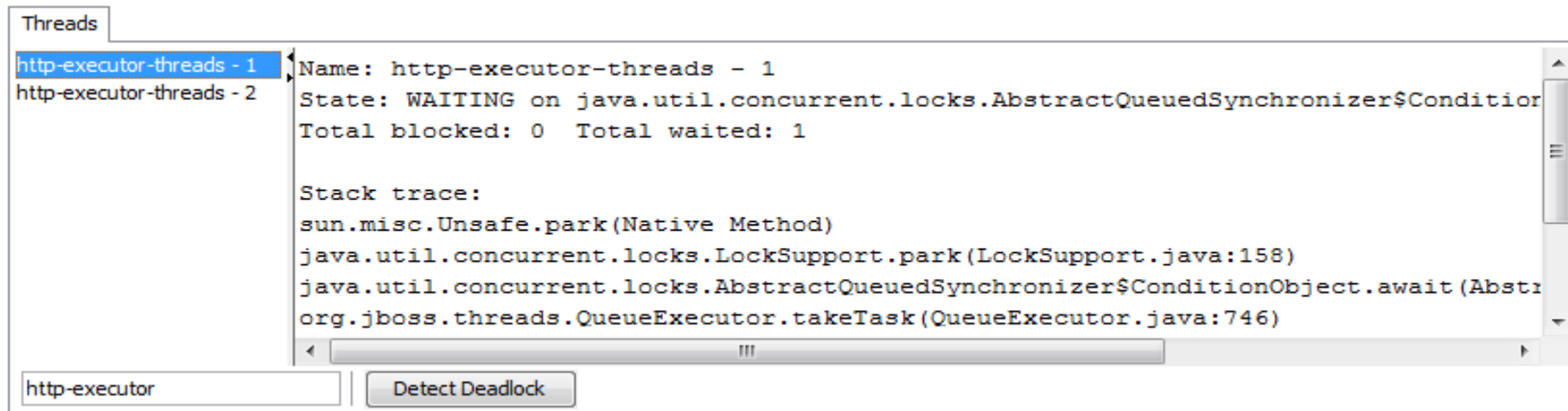
Generate a thread dump by using a program such as jVisualVM.



Ex1 : C:\Java\jdk1.8.0\bin\jvisualvm

Ex2 : C:\Java\jdk1.8.0\bin\jstat -f <process-id>

idle thread should look like, by looking at its stack trace:



The screenshot shows the 'Threads' tab in a Java IDE. A list on the left contains 'http-executor-threads - 1' (highlighted) and 'http-executor-threads - 2'. The main pane displays details for the selected thread:

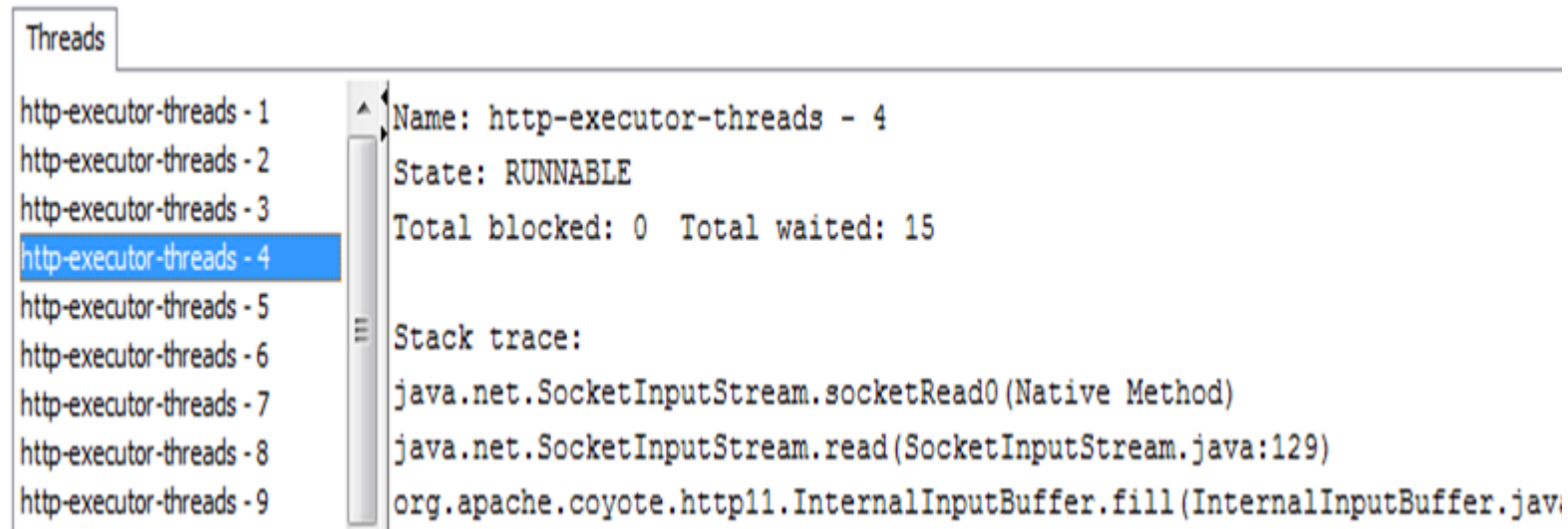
- Name: http-executor-threads - 1
- State: WAITING on java.util.concurrent.locks.AbstractQueuedSynchronizer\$Condition
- Total blocked: 0 Total waited: 1

The stack trace is as follows:

```
Stack trace:  
sun.misc.Unsafe.park(Native Method)  
java.util.concurrent.locks.LockSupport.park(LockSupport.java:158)  
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(Abst:  
org.jboss.threads.QueueExecutor.takeTask(QueueExecutor.java:746)
```

At the bottom, there is a text field containing 'http-executor' and a 'Detect Deadlock' button.

HTTP thread is **busy** at doing input/output operations which could mean, for example, the web server is acquiring data from an external resource.



The screenshot shows the 'Threads' tab in a Java IDE. A list of threads is on the left, with 'http-executor-threads - 4' selected and highlighted in blue. To the right, the details for this thread are displayed:

- Name: http-executor-threads - 4
- State: RUNNABLE
- Total blocked: 0 Total waited: 15
- Stack trace:
 - java.net.SocketInputStream.socketRead0 (Native Method)
 - java.net.SocketInputStream.read (SocketInputStream.java:129)
 - org.apache.coyote.http11.InternalInputBuffer.fill (InternalInputBuffer.java:131)

Eliminate Runnable threads: Runnable threads are already being actively scheduled and there is no need to extensively focus on them in the first pass.

Eliminate Idle threads: It is natural that some threads in the application are in the idle state. They would either be waiting for an incoming connection or sleeping for a period of time before they wake. Eliminate such threads from your analysis which would be ideally found idling cycles.

The below threads can be safely ignored :

"tcpConnection-9011-7736" daemon prio=1 tid=0x351750b8
nid=0x7eab in **Object.wait()** [0x2fdf4000..0x2fdf50b0]
at java.lang.Object.wait(Native Method)
- waiting on <0x43eef7c8> (a java.lang.Object)
at com.caucho.server.TcpServer.accept(TcpServer.java:648)

"Store org.hibernate.cache.StandardQueryCache Expiry Thread"
daemon prio=1 tid=0x3becd4e0 nid=0x1c6e **waiting on condition**
[0x35389000..0x353891b0] at java.lang.Thread.sleep(Native Method)

"Store org.hibernate.cache.UpdateTimestampsCache Expiry Thread"
daemon prio=1 tid=0x397868a8 nid=0x1c6c **waiting on condition**
[0x3548a000..0x3548b0b0] at java.lang.Thread.sleep(Native Method)

Analyse BLOCKED and WAITING threads: This is perhaps the most important part of analysis. It is important to ask the question why are these threads in either the BLOCKED or WAITING state when they should have been in the RUNNABLE state.

These threads are waiting for monitors which have been acquired by other threads.

"tcpConnection-9011-7709" in the BLOCKED state waiting to lock a monitor:

"tcpConnection-9011-7709" daemon prio=1 tid=0x35029e40
nid=0x7e61 **waiting for monitor entry** [0x2fe74000..0x2fe75e30]
at
java.beans.Introspector.getPublicDeclaredMethods(Introspector.java:1249)
- **waiting to lock <0x3d9bb6b0> (a java.lang.Class)**

"tcpConnection-9011-7640" has obtained the monitor.

"tcpConnection-9011-7640" daemon prio=1 tid=0x35177a40
nid=0x7dc7 **runnable** [0x305fa000..0x305fcf30]
at
java.beans.Introspector.findExplicitBeanInfo(Introspector.java:410)
- **locked <0x3d9bb6b0> (a java.lang.Class)**

Resource Contention

Resource contention typically happens when the threads are fighting for the same resources. If a majority of threads in the application are fighting for the same resource this could speak about poor application design.

The most common symptoms of resource contention issues are :

- > The throughput / responsiveness of the application has fallen considerably and the application is too slow
- > The CPU utilization is low
- > All the application resources (db connection pools etc) seem to free and available
- > In thread dumps most the thread would appear in the BLOCKED / WAITING FOR MONITOR ENTRY state

out-of-threads

Consider the following scenario:

- > Response time of your key pages is rapidly dropping
- > The application throughput is more or less constant
- > The application server and database servers are running with permissible limits of system parameters

It is possible that you may be seeing a sudden increase of users hitting the application. Based on the analysis the only immediate recourse would be to increase the number of threads in the thread pool to cater to this increased demand. We need to consider the impact of such changes before making the changes.

Problem 1 :

all the threads are waiting to acquire lock on the monitor
<0x44008de0>

"tcpConnection-9011-8009" daemon prio=1 tid=0x31446a80
nid=0x49f5 **waiting for monitor entry** [0x2edf5000..0x2edf6e30]
at
com.opensymphony.oscache.base.algorithm.AbstractConcurrentReadC
ache.put(AbstractConcurrentReadCache.java:1648)
waiting to lock <0x44008de0>

This lock is obtained by the below thread:

" tcpConnection-9011-7817" daemon prio=1 tid=0x24fee130
nid=0x491c **runnable** [0x1513e000..0x15140130]
at java.lang.System.identityHashCode(Native Method)
locked < 0x44008de0 >

Analysis :

operating system had run out of file descriptors and hence thread `tcpConnection-9011-7817` couldn't really persist the cached object to the disk and release the monitor.

Solutions :

Increasing the limit on the number of file descriptors (`ulimit`) and restarting the application.

BLOCKED (on object monitor) -> **(ex. JFR-Latencies)**

"Worker Thread 2" #12 prio=5 os_prio=0 tid=0x000000001b7c4000 nid=0x19b0
waiting for monitor entry [0x000000001c11f000]
java.lang.Thread.State: **BLOCKED** (on object monitor)
 at Logger.log(Logger.java:17)
 - waiting to lock <0x00000000d5de8788> (a Logger)
at WorkerThread.run(WorkerThread.java:24)

Refer to the below thread, which is holding the lock

"Worker Thread 0" #10 prio=5 os_prio=0 tid=0x000000001b7bd800 nid=0x2af8
waiting on condition [0x000000001bf1f000]
java.lang.Thread.State: **TIMED_WAITING** (sleeping)
 at java.lang.Thread.sleep(Native Method)
 at Logger.log(Logger.java:17) - locked <0x00000000d5de8788> (a Logger)
 at WorkerThread.run(WorkerThread.java:24)

Problem : (waiting on object monitor)

The below thread is unable to get database connection:

```
"Thread-1" prio=5 tid=0x00a861b8 nid=0xbdc in Object.wait()  
[0x02d0f000..0x02d0fb68]  
    at java.lang.Object.wait(Native Method)  
        - waiting on <0x22aadfc0> (a  
org.tw.testyard.thread.ConnectionPool)  
    at java.lang.Object.wait(Unknown Source)  
    at  
org.tw.testyard.thread.ConnectionPool.getConnection(ConnectionPool.  
java:39)  
        - locked <0x22aadfc0> (a org.tw.testyard.thread.ConnectionPool)
```

Analyse thread dumps over a duration

Compare the the sanapshot's :

Snapshot 1:

"tcpConnection-9011-7696" daemon prio=1 tid=0x3466c258
nid=0x7e4f in **Object.wait()** [0x2727b000..0x2727bfb0]

Snapshot 2:

"tcpConnection-9011-7696" daemon prio=1 tid=0x3466c258
nid=0x7e4f **runnable** [0x2727b000..0x2727bfb0]

Solution :

We see such things happening often in your application it is time to revisit the connection pool configuration.

In any case the connection pool size should be at least as big as the number of worker threads in your application.

We may see the similar issues when dealing with resources such as db connections, network connections or file handles.

Name: http-nio-8080-exec-32 [Tomcat idle thread]

State: WAITING on

java.util.concurrent.locks.AbstractQueuedSynchronizer\$ConditionObject@5a7b5d06

Total blocked: 1 Total waited: 21

Stack trace:

sun.misc.Unsafe.park(Native Method)

java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)

java.util.concurrent.locks.AbstractQueuedSynchronizer\$ConditionObject.await(AbstractQueuedSynchronizer.java:2039)

java.util.concurrent.LinkedBlockingQueue.take(LinkedBlockingQueue.java:442)

org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:103)

org.apache.tomcat.util.threads.TaskQueue.take(TaskQueue.java:31)

java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1074)

java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1134)

java.util.concurrent.ThreadPoolExecutor\$Worker.run(ThreadPoolExecutor.java:624)

org.apache.tomcat.util.threads.TaskThread\$WrappingRunnable.run(TaskThread.java:61)

java.lang.Thread.run(Thread.java:748)

Thread Stack Trace -> Blocked, Waited count

Blocked count is the total number of times that the thread blocked to enter or reenter a monitor. I.e. the number of times a thread has been in the `java.lang.Thread.State.BLOCKED` state.

Waited count is the total number of times that the thread waited for notification. i.e. the number of times that a thread has been in the `java.lang.Thread.State.WAITING` or `java.lang.Thread.State.TIMED_WAITING` state.

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.

JMX Architecture

What is JMX ?

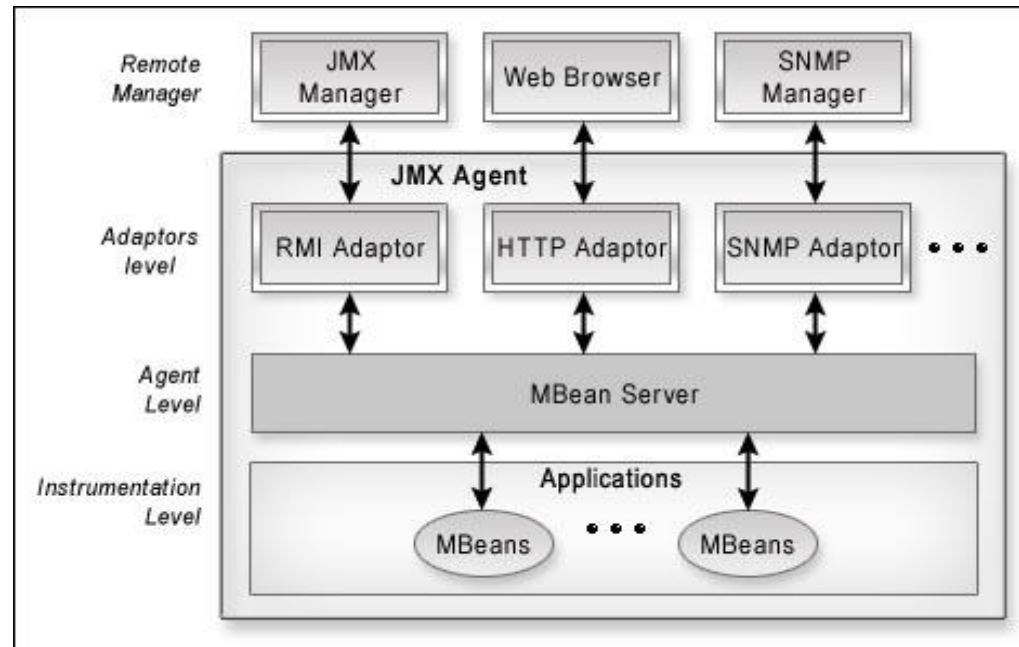
Java Management Extensions (JMX) is an upcoming open technology specification that defines the management architecture, which enables managing of applications and services.

This technology also allows Java developers to integrate their applications with existing network management solutions.

As per JMX specification, JMX architecture is divided into four levels:

1. Instrumentation level
2. Agent level
3. Adaptors level
4. Remote Manager/ distribution level

A diagram representing the JMX architecture is shown below:



JMeter

Thread Group

Thread group elements are the beginning points of any test plan. All controllers and samplers must be under a thread group.

The thread group element controls the number of threads JMeter will use to execute the test.

The controls for a thread group allow you to:

- Set the number of threads
- Set the ramp-up period
- Set the number of times to execute the test

Each thread will execute the test plan in its entirety and completely independently of other test threads.

Multiple threads are used to simulate concurrent connections to your server application.

The ramp-up period tells JMeter how long to take to "ramp-up" to the full number of threads chosen.

If 10 threads are used, and the ramp-up period is 100 seconds, then JMeter will take 100 seconds to get all 10 threads up and running.

Each thread will start 10 ($100/10$) seconds after the previous thread was begun. If there are 30 threads and a ramp-up period of 120 seconds, then each successive thread will be delayed by 4 seconds.

Ramp-up needs to be long enough to avoid too large a workload at the start of a test, and short enough that the last threads start running before the first ones finish (unless one wants that to happen).

Start with Ramp-up = number of threads and adjust up or down as needed.

By default, the thread group is configured to loop once through its elements.

Thread Pool & Types of Executors

ThreadPool Executor Types:

Single Thread Executor : A thread pool with only one thread. So all the submitted tasks will be executed sequentially. Method : `Executors.newSingleThreadExecutor()`

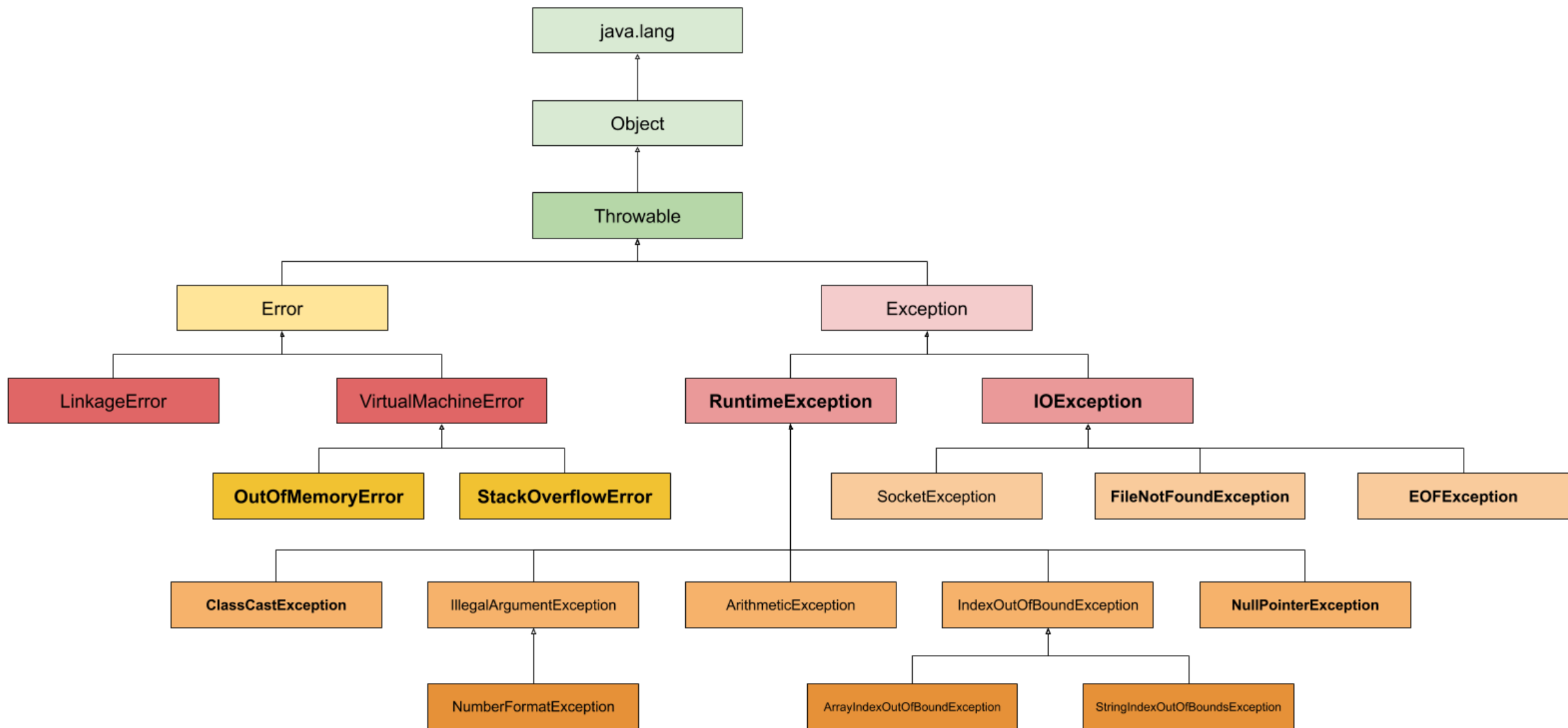
Cached Thread Pool : A thread pool that creates as many threads it needs to execute the task in parallel. The old available threads will be reused for the new tasks. If a thread is not used during 60 seconds, it will be terminated and removed from the pool. Method : `Executors.newCachedThreadPool()`

Fixed Thread Pool : A thread pool with a fixed number of threads. If a thread is not available for the task, the task is put in queue waiting for an other task to ends. Method : `Executors.newFixedThreadPool()`

Scheduled Thread Pool : A thread pool made to schedule future task. Method : `Executors.newScheduledThreadPool()`

Single Thread Scheduled Pool : A thread pool with only one thread to schedule future task. Method : `Executors.newSingleThreadScheduledExecutor()`

Exception Handling



Errors	Exceptions
<p>Errors are usually raised by the environment in which the application is running. For example, an error will occur due to a lack of system resources.</p> <p>It is not possible to recover from an error.</p> <p>Errors occur at run-time and are not known by the compiler; hence, they are classified as “unchecked.”</p> <p>“OutOfMemory” and “StackOverflow” are examples of errors.</p>	<p>Exceptions are caused by the code of the application itself.</p> <p>The use of try-catch blocks can handle exceptions and recover the application from them.</p> <p>Exceptions can be “checked” or “unchecked,” meaning they may or may not be caught by the compiler.</p> <p>“IndexOutOfBoundsException” is an example of an unchecked exception, while “ClassNotFoundException” is an example of a checked exception.</p>

Checked Exceptions

The checked exceptions are the ones that implement the `Exception` class and not the `RuntimeException`.

They are called checked exceptions because the compiler verifies them during the compile-time and refuses to compile the code if such exceptions are not handled or declared.

Such exceptions are usually used to notify the user that a method can result in a state that needs to be handled.

For example, the `FileNotFoundException` is an example of such an exception.

Unchecked Exceptions / Runtime Exceptions

The unchecked exceptions are the second type of exceptions in Java.

The unchecked exceptions in Java are the ones that implement the RuntimeException class.

Those exceptions can be thrown during the normal operation of the Java Virtual Machine.

Unchecked exceptions do not need to be declared in the method throws clause in order to be thrown.

ArithmeticException

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

[OR]

```
try{  
    //code that may raise exception  
    int data=100/0;  
}
```

```
catch(ArithmeticException e)
```

```
{System.out.println(e);}
```

Performance Side Effects of Using Exceptions

Throwing an exception in Java requires the JVM to fill up the whole call trace, list each method that comes with it, and pass it further to the code that will finally catch and handle the exception.

That's why we shouldn't use exceptions unless it is really necessary.

The code that uses an exception looks as follows:

```
public int divide(int dividend, int divisor) {  
    try {  
        return dividend / divisor;  
    } catch (Exception ex) {  
        // handle exception  
    }  
    return -1;  
}
```

The code that does a simple check using a conditional looks as follows:

```
public int divide(int dividend, int divisor)
{
    if (divisor != 0) {
        return 10 / divisor;
    }
    return -1;
}
```

Spring DataAccessException

The Data Access Object (DAO) support in Spring allows us to isolate minimal amount of code related to particular database technology easily.

The most important DAO support is `DataAccessException` hierarchy which let the client code handle the exceptions without knowing the details of the particular data access API in use (e.g. JDBC, JPA, Hibernate etc).

That means it is possible to handle an exception like `JdbcSQLException` in a generic way without knowing that JDBC is being used in the DAO layer. This allows one to switch between the persistence technologies easily without changing a lot of code.

The @Repository annotation

@Repository annotation guarantees that all Spring DAO support, including the exception translation is provided in a consistent way.

Controller Exception Handler

```
@ExceptionHandler(EmptyResultDataAccessException.class)
public ResponseEntity<String> noCityFound(DataAccessException e) {

    return ResponseEntity.status(HttpStatus.NOT_FOUND).body("No City
found");
}
```