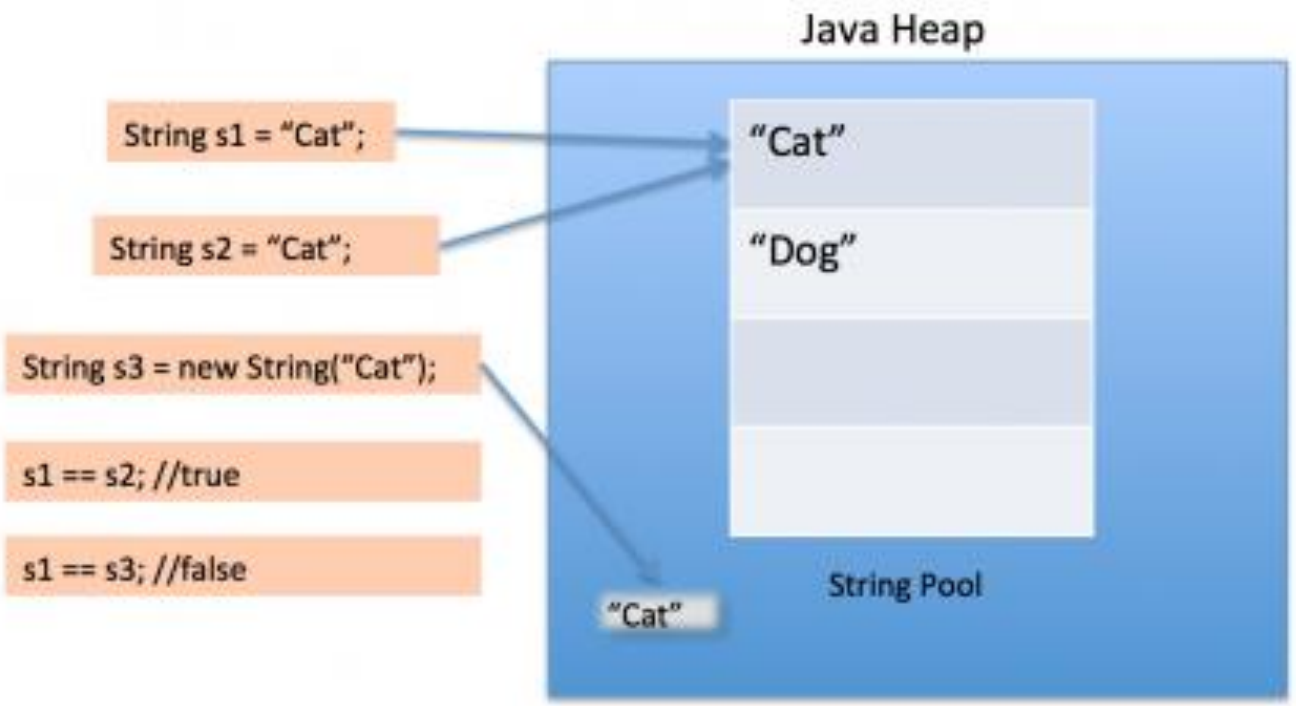
String Magic

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

as well as providing literal value
String s2 = “cat";

What is the difference ????



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

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

Vs

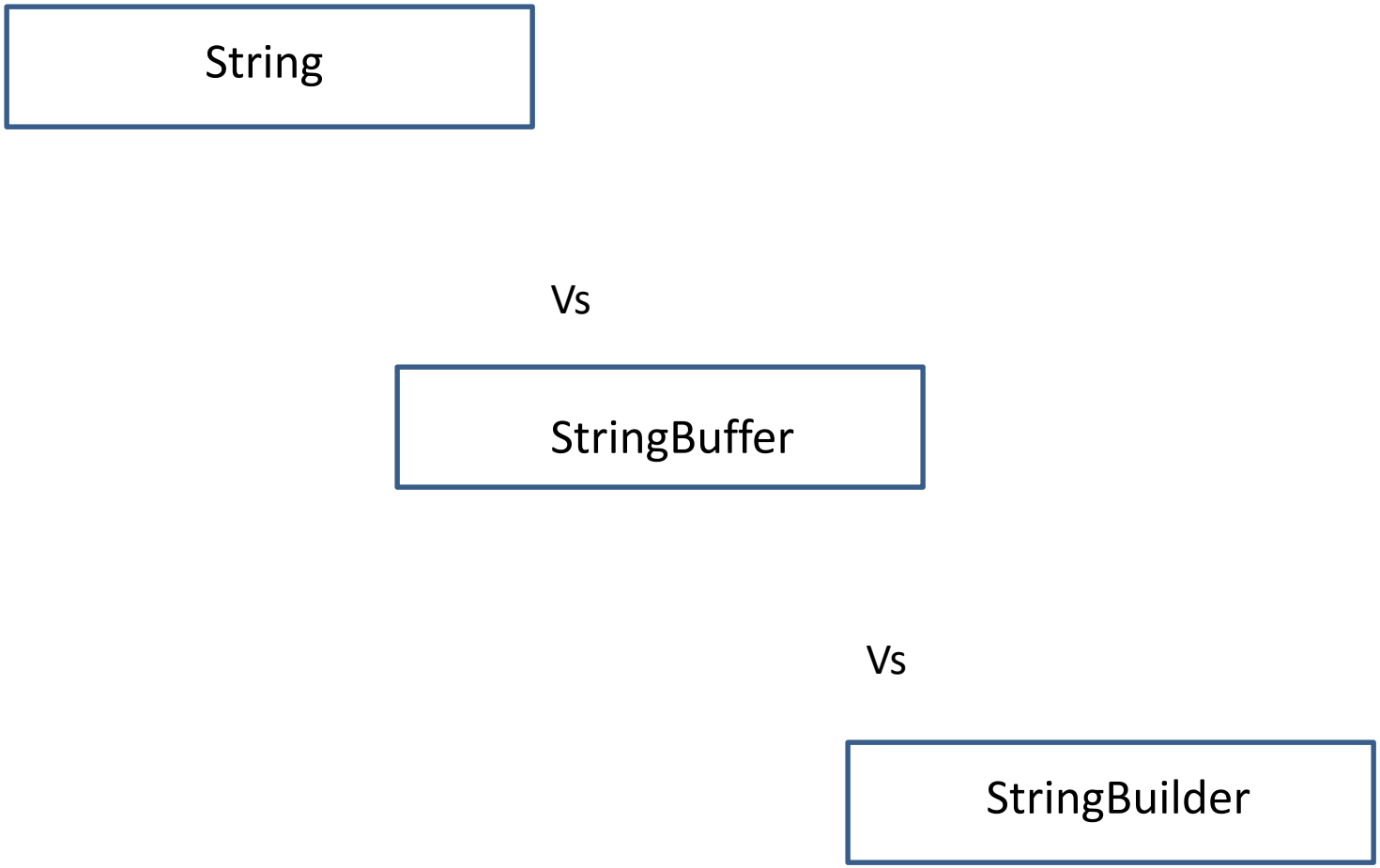
```
String str3 = new String("great").intern();
```

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.

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 cancat 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 *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

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.

finalize() Method Call Has Advantages and Disadvantages:

This method is available in object class.

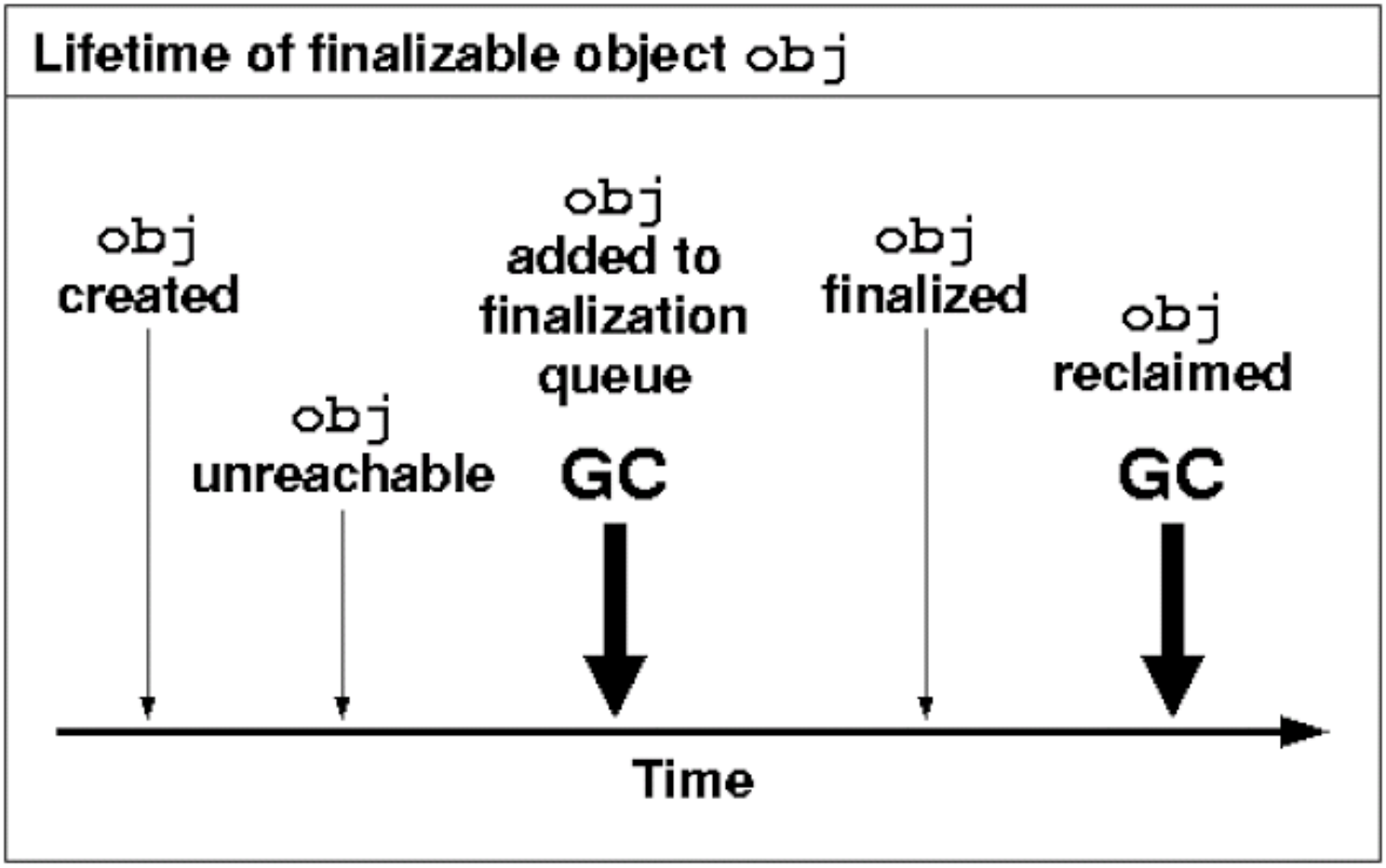
Java suggests that this method should be called when you want to instruct garbage collector to clean up the object.

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
obj created
obj unreachable
obj added to finalization queue
GC
obj finalized
obj reclaimed
GC
Time

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.

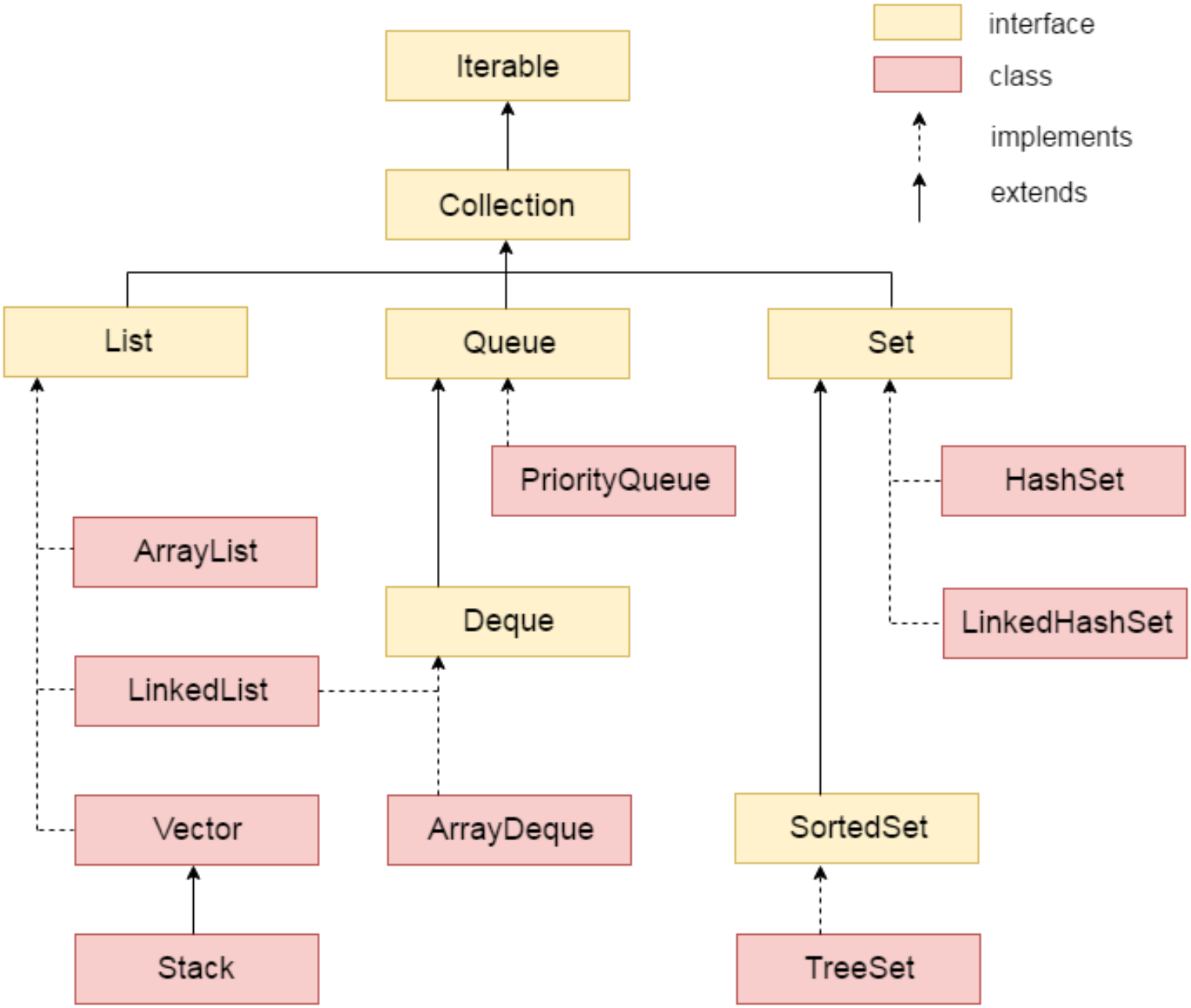
Sometimes you have to load data in memory(cache) to avoid repeated calls to persistent store/source of data, but it is better to optimize

> Eager vs Lazy loading

> fetching strategies

> Cache eviction policies

Java Collection API


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

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.	Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity.
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 implementatioons 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 10 times better than ArrayList & ArrayList performs 4 times better than LinkedList. LinkedList is slow because of sequencial search

ArrayList vs LinkedList

1) Since Array is an index based data-structure searching or getting element from Array with index is pretty fast. Array provides O(1) performance for get(index) method but remove is costly in ArrayList as you need to rearrange all elements.

On the Other hand LinkedList doesn't provide Random or index based access and you need to iterate over linked list to retrieve any element which is of order O(n).

2) Insertions are easy and fast in LinkedList as compared to ArrayList because there is no risk of resizing array and copying content to new array if array gets full which makes adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList in Java.

ArrayList also needs to update its index if you insert something anywhere except at the end of array.

ArrayList vs LinkedList (continued)

3) Removal is like insertions better in LinkedList than ArrayList.

4) LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

In the below situations, LinkedList is better choice than ArrayList :

1) Your application can live without Random access. Because if you need nth element in LinkedList you need to first traverse up to nth element O(n) and than you get data from that node.

2) Your application is more insertion and deletion driver and you insert or remove more than retrieval. Since insertion or removal doesn't involve resizing its much faster than ArrayList.

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).

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
```

Weak References

JVM ignores the **weak references**. That means objects which has only week 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.	`java.lang.ref.SoftReference`
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	`java.lang.ref.WeakReference` `java.util.WeakHashMap`
Phantom Reference	Lets you clean up after finalization but before the space is reclaimed (replaces or augments the use of finalize())	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.
Why use Synchronization

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

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

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.

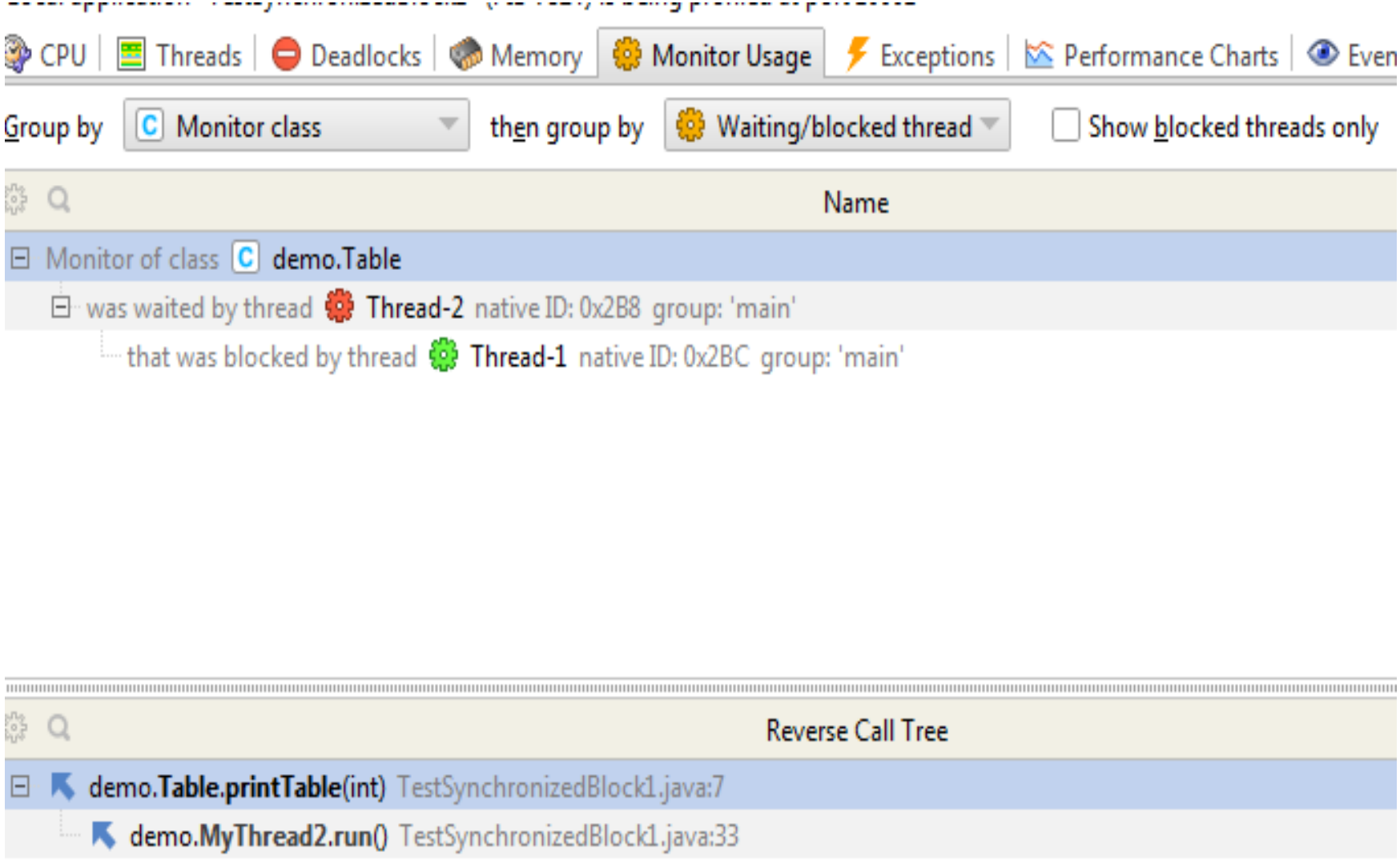
This is useful in different scenarios where you have to limit the amount concurrent access to certain parts of your application.

```
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
Monitor class
then group by
Waiting/blocked thread
Show blocked threads only
Name
Monitor of class demo.Table
was waited by thread Thread-2 native ID: 0x2B8 group: 'main'
that was blocked by thread 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.

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 work-load 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 parrallel. 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()