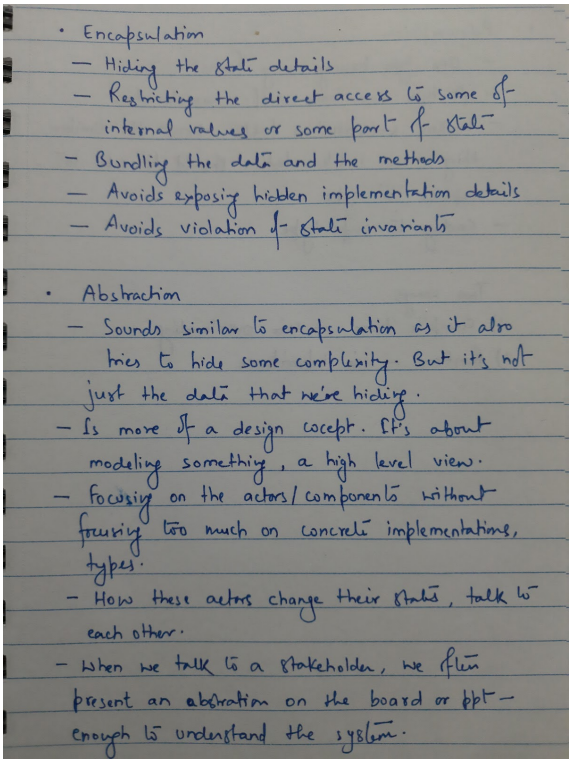
TODO

- [ ] Study weakhashmap and IdentityHashmap
- [x] ~~Run some samples with TreeMap e.g. ceiling, floor~~
- [x] ~~Try Priority Queue~~
- [x] ~~Optional's flatmap~~
- [ ] ArraysParallelSortHelpers
- [ ] CilkSort
- [ ] @HotSpotIntrinsicCandidate
- [x] ~~Arrays.copyOf vs System.arrcopy~~
- [ ] Arrays.sort variants
- [x] ~~Spliterator~~
- [ ] Parallel streams
- [ ] VarHandle
- [ ] Volatile
- [ ] Happens-before relationship
- [ ] Java Memory Model
- [ ] https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html
- [ ] @HotSpotIntrinsicCandidate
- [ ] AbstractQueuedSynchronizer
- [ ] @ReservedStackAccess
- [ ] Condition
- [ ] Two lock queue algorithm
- [ ] https://www.cs.rochester.edu/research/synchronization/pseudocode/duals.html
- [ ] https://www.cs.rochester.edu/~scott/papers/2004_DISC_dual_DS.pdf
- [ ] http://www.cs.rochester.edu/u/scott/papers/2009_Scherer_CACM_SSQ.pdf
- [ ] http://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf
- [ ] https://dl.acm.org/doi/10.1145/1073970.1074013
- [ ] http://people.csail.mit.edu/edya/publications/OptimisticFIFOQueue-journal.pdf
- [ ] https://javainterviewsquestions.wordpress.com/
- [ ] Does calling wait release all the locks help by thread or just the monitor of the object on which wait was called?
- [ ] Flow API
- [ ] CompletableFuture
- [ ] Work stealing Pool
- [ ] ForkJoin

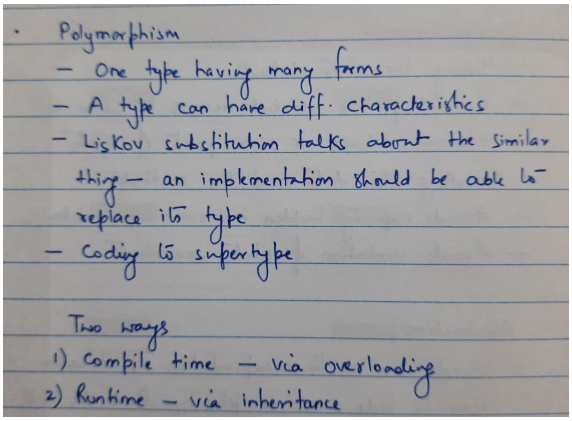
- [ ] ConcurrentHashMap - does it still use Segment?
- [ ] http://gee.cs.oswego.edu/dl/html/j9mm.html
- [ ] https://github.com/LMAX-Exchange/disruptor
- [ ] https://doc.akka.io/docs/akka/2.5/guide/actors-motivation.html?language=java
- [ ] https://docs.oracle.com/javase/specs/jls/se11/html/index.html
- [ ] http://objectlayout.org/
- [ ] https://www.dynatrace.com/resources/ebooks/javabook/performance-and-scalability/
- [ ] https://docs.oracle.com/en/java/javase/11/vm/index.html
- [ ] https://docs.oracle.com/en/java/javase/11/gctuning/index.html
- [ ] https://docs.oracle.com/javase/specs/jvms/se11/html/index.html
- [ ] http://blog.jamesdbloom.com/JVMInternals.html

Core

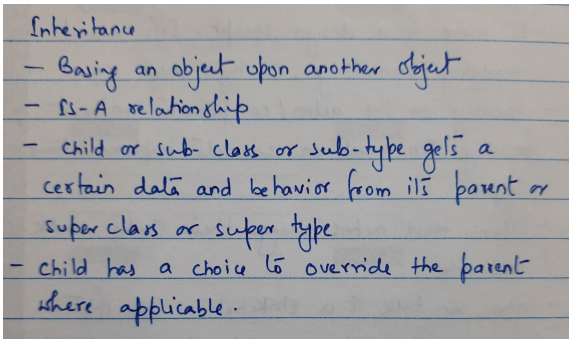
Abstraction and Encapsulation



Polymorphism



Inheritance



Overriding, Overloading

- ➔ **Initialization order -**
  - Base static blocks in order
  - Child static blocks in order
  - Base instance blocks in order
  - Base Constructor
  - Child instance blocks in order
  - Child constructor
- ➔ The child must explicitly call super arg constructor in absence of super's no-arg constructor.
- ➔ public static variables of Base can be accessed in child classes.
- ➔ Base static methods can be overloaded in Child class.
- ➔ While shadowing the base static method, either overload the method or follow all overriding rules.

Why Java doesn't allow to override static methods?

https://stackoverflow.com/questions/2223386/why-doesnt-java-allow-overriding-of-static-methods
Overriding is an object-oriented concept, when a method is overridden it behaves differently depending upon the type the object is invoking the method, for example,

the Animal class has a method called eat(), dog class overrides the eat() method because the dog eats different food, cat class also overrides the eat() method because cat eats different food.
On the other hand, let's say you want to sort cat's name and dogs name, this sort method should be static because it does not change its behavior depending on the object, it always sorts the string name, whether it is a cat's name or dog's name.

Aggregation and Composition

https://softwareengineering.stackexchange.com/questions/176049/what-is-the-use-of-association-aggregation-and-composition
Aggregation and Composition are related concepts. Both can be considered as a 'has-a' relationship.
From an implementation perspective, there's not really much difference.
Composition is just a bit more restrictive than aggregation.

- A "owns" B = Composition: B has no meaning or purpose in the system without A
- A "uses" B = Aggregation: B exists independently (conceptually) from A

A file reader - can have a buffer that can be used while reading a file.

Now, the file can exist in the directory even when the file reader is no longer in the memory but the buffer won't. Buffer object exists as long as the file reader object is active.

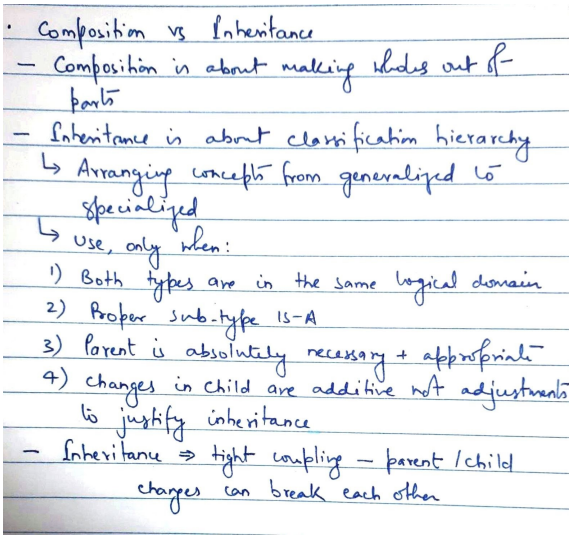
Why favor composition over inheritance

https://stackoverflow.com/questions/49002/prefer-composition-over-inheritance
https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose
http://www.javaworld.com/article/2076814/core-java/inheritance-versus-composition--which-one-should-you-choose-.html

My acid test for the above is:

- Does TypeB want to expose the complete interface (all public methods no less) of TypeA such that TypeB can be used where TypeA is expected? Indicates **Inheritance**.
  - e.g. A Cessna biplane will expose the complete interface of an airplane, if not more. So that makes it fit to derive from Airplane.
- Does TypeB want only some/part of the behavior exposed by TypeA? Indicates need for **Composition.**
  - e.g. A Bird may need only the fly behavior of an Airplane. In this case, it makes sense to extract it out as an interface / class / both and make it a member of both classes.

Update: Just came back to my answer and it seems now that it is incomplete without a specific mention of Barbara Liskov's Liskov Substitution Principle as a test for 'Should I be inheriting from this type?'



Interface vs Abstract classes

Interfaces used to be 100% abstract before Java8 but that's not the case anymore.
Starting with Java8, interfaces can have non-abstract methods in the form of default and static methods.
Default methods can be overridden by the class implementing the interface. However, static methods are the same as classes i.e. can't be overridden.
Starting with Java9, interfaces can even have a private default/static method. The idea is to DRY in case you have repetitive code in your default or static methods.
The boundary of abstract classes and interfaces is not that strict anymore. We can achieve almost everything with interfaces now.
The main reason to choose an abstract class over interface would be -

- If the state needs to be maintained because interface variables are by default constants.
- There's something that you must do in the constructor.

Exceptions

- ➔ Cannot throw any checked exception without declaring it in throws.
- ➔ Can throw any subtype of checked exception declared in throws.
- ➔ Can throw any unchecked exception without declaring it in throws.
- ➔ Either catch or duck checked exception.
- ➔ No need to catch or duck un-checked exception / Error.
- ➔ How does a compiler identify that there is a checked exception
  - ◆ It looks at the code, if it comes across an exception, looks up the inheritance hierarchy of the exception to decide if it is checked or unchecked.

Singleton vs Utility class or Enums

- ➔ **Singleton:** When you only need one live instance of an object. Having more than one instance will compromise the system. E.g. Cache implementation.
- ➔ **Utility:** When you want to provide a bunch of methods that you want to co-locate in one place. Typically such classes have static methods. As such having more than one instance of the object will not affect the integrity.
- ➔ **Enum:** They represent a fixed set of constants from which you choose a value. E.g. Days of the week.

Serialization

http://moi.vonos.net/java/serialization/
https://docs.oracle.com/en/java/javase/11/docs/specs/serialization/index.html

Strings

https://javaranch.com/journal/200409/ScjpTipLine-StringsLiterally.html
http://java-performance.info/string-intern-in-java-6-7-8/

Class loading

A typical approach for loading a class in memory relies on calling the `Class.forName(String className)` method. Notice the `String` argument representing the class name. Thanks to string immutability, the class name cannot be changed during the loading process. However, if `String` is mutable, then imagine loading `class A` (for example, `Class.forName("A")`), and, during the loading process, its name will get changed to `BadA`. Now, the `BadA` objects can do bad things!

Because Strings are immutable:

Sensitive data in strings (for example, passwords) may reside in memory (in SCP) for a long time. Being a cache, the SCP takes advantage of special treatment from the garbage collector. More precisely, the SCP is not visited by the garbage collector with the same frequency (cycles) as other memory zones. As a consequence of this special treatment, sensitive data is kept in the SCP for a long time, and can be prone to unwanted usages.

In order to avoid this potential drawback, it is advisable to store sensitive data (for example, passwords) in `char[]` instead of `String`.

Is String completely immutable?

Well, behind the scenes, `String` uses `private final char[]` to store each character of the string. By using the Java Reflection API, in JDK 8, the following code will modify this `char[]` (the same code in JDK 11 will throw `java.lang.ClassCastException`):

```
String user = "guest";
System.out.println("User is of type: " + user);

Class<String> type = String.class;
Field field = type.getDeclaredField("value");
field.setAccessible(true);

char[] chars = (char[]) field.get(user);

chars[0] = 'a';
chars[1] = 'd';
chars[2] = 'm';
chars[3] = 'i';
chars[4] = 'n';

System.out.println("User is of type: " + user);
```

So, in JDK 8, `String` is *effectively* immutable, but not *completely*.

In Java 8, all strings are encoded as arrays of 16-bit characters, regardless of the encoding of the string. This is wasteful: most Western locales can encode strings into 8-bit byte arrays, and even in a locale that requires 16 bits for all characters, strings like program constants often can be encoded as 8-bit bytes.

In Java 11, strings are encoded as arrays of 8-bit bytes unless they explicitly need 16-bit characters; these strings are known as compact strings. A similar (experimental) feature in Java 6 was known as compressed strings; compact strings are conceptually the same but differ greatly in implementation.

Hence, the size of an average Java string in Java 11 is roughly half the size of the same string in Java 8. This generally is a huge savings: on average, 50% of a typical Java heap may be consumed by string objects.
This feature is controlled by the -XX:+CompactStrings flag, which is true by default.

String deduplication

The simplest mechanism is to let the JVM find the duplicate strings and *deduplicate* them: arrange for all references to point to a single copy and then free the remaining copies. This is possible only when using G1 GC and only when specifying the `-XX:+UseStringDeduplication` flag (which by default is `false`). This feature exists in Java 8 only after version 20, and all releases of Java 11.

This feature is not enabled by default for three reasons. First, it requires extra processing during the young and mixed phases of G1 GC, making them slightly longer. Second, it requires an extra thread that runs concurrently with the application, potentially taking CPU cycles away from application threads. And third, if there are few deduplicated strings, the memory use of the application will be higher (instead of lower); this extra memory comes from the bookkeeping involved in tracking all the strings to look for duplications.

Cloning

Cloning via clone()

The `Object` class contains a method named `clone()`. This method is useful for creating shallow copies (it can be used for deep copies as well). In order to use it, a class should follow the given steps:

- Implement the `Cloneable` interface (if this interface is not implemented, then `CloneNotSupportedException` will be thrown).
- Override the `clone()` method (`Object.clone()` is `protected`).
- Call `super.clone()`.

The `Cloneable` interface doesn't contain any methods. It is just a signal for JVM that this object can be cloned. Once this interface is implemented, the code needs to override the `Object.clone()` method. This is needed because `Object.clone()` is `protected`, and, in order to call it via `super`, the code needs to override this method. This can be a serious drawback if `clone()` is added to a child class since all superclasses should define a `clone()` method in order to avoid the failure of the `super.clone()` chain invocation.

Deep Cloning

Can be done manually
Via any 3rd party library
Via Serialization
Via JSON

Cloning via JSON

Almost any JSON library in Java can serialize any **Plain Old Java Object (POJO)** without any extra configuration/mapping required. Having a JSON library in the project (and many projects have) can save us from adding an extra library to provide deep cloning. Mainly, the solution can leverage the existing JSON library to get the same effect.

The following is an example using the `Gson` library:

```
private static <T> T cloneThroughJson(T t) {

  Gson gson = new Gson();
  String json = gson.toJson(t);

  return (T) gson.fromJson(json, t.getClass());
}

Point point = new Point(...);
Point clone = cloneThroughJson(point);
```

Var(Local Variable Type Inference - LVTI)

```
// Avoid
var intNumber = 10;    // inferred as int
var longNumber = 10;   // inferred as int
var floatNumber = 10;  // inferred as int
var doubleNumber = 10; // inferred as int
```

```
// Prefer
var intNumber = 10;     // inferred as int
var longNumber = 10L;   // inferred as long
var floatNumber = 10F;  // inferred as float, 10.0
var doubleNumber = 10D; // inferred as double, 10.0
```

```
var byteNumber = 25;    // inferred as int
var shortNumber = 1463; // inferred as int
```

Unfortunately, there are no *literals* available for these two primitive types. The only approach to help the compiler to infer the correct types is to rely on an explicit downcast:

```
var byteNumber = (byte) 25;     // inferred as byte
var shortNumber = (short) 1463; // inferred as short
```

Unfortunately, LVTI cannot take advantage of the *programming to the interface* technique. In other words, when we use `var`, the inferred type is the concrete implementation, not the contract. For example, replacing `List<String>` with `var` will result in the inferred type, `ArrayList<String>`:

```
// inferred as ArrayList<String>
var playerList = new ArrayList<String>();
```

Nevertheless, there are some explanations that sustain this behavior:

- LVTI acts at the local level (local variables) where the *programming to the interface* technique is used less than method parameters/return types or field types.

```
Player p1 = new Player();
Player p2 = new Player();
var listOfPlayer = List.of(p1, p2); // inferred as List<Player>

// Don't do this!
var listOfPlayer = new ArrayList<>(); // inferred as ArrayList<Object>
listOfPlayer.add(p1);
listOfPlayer.add(p2);
```

```
var[] numberArray = new int[10];
var numberArray[] = new int[10];
```

Unfortunately, none of these two approaches will compile. The solution requires us to remove the brackets from the left-hand side:

```
// Prefer
var numberArray = new int[10]; // inferred as array of int, int[]
numberArray[0] = 3;            // works
numberArray[0] = 3.2;          // doesn't work
numbers[0] = "3";              // doesn't work
```

There is a common practice to initialize an array at declaration time, as follows:

```
// explicit type work as expected
int[] numbers = {1, 2, 3};
```

However, trying to use `var` will not work (will not compile):

```
// Does not compile
var numberArray = {1, 2, 3};
var numberArray[] = {1, 2, 3};
var[] numberArray = {1, 2, 3};
```

```
// Does not compile
public var fetchReport(Player player, Date timestamp) {

  return new Report();
}
```

- LVTI cannot be used as a method argument type—the following code doesn't compile:

```
public Report fetchReport(var player, var timestamp) {

  return new Report();
}
```

- Variables of the `var` type can be passed as method arguments or store a return method—the following code compiles successfully and it works:

```
public Report checkPlayer() {

  var player = new Player();
  var timestamp = new Date();
  var report = fetchReport(player, timestamp);

  return report;
}

public Report fetchReport(Player player, Date timestamp) {

  return new Report();
}
```

Collections

- ➔ An unmodifiable collection is not necessarily immutable. If the contained elements are mutable, the entire collection is clearly mutable, even though it might be unmodifiable.
- ➔ A collection implementation that implements the {@code Serializable} interface cannot be guaranteed to be serializable. The reason is that in general, collections contain elements of other types, and it is not possible to determine statically whether instances of some element type are actually serializable. The collection may be serializable, if it contains only elements of some serializable type, or if it is empty. Collections are thus said to be conditionally serializable, as the serializability of the collection as a whole depends on whether the collection itself is serializable and on whether all contained elements are also serializable.
- ➔ If you have two collections c1, c2, and you have to iterate over one and check contains in another(scenario like checking if these two have common elements) then:
  - ◆ Iterate over small collection min(c1.size(), c2.size())
  - ◆ If one of them is set, choose a set to check contains as that would have a better search performance, generally O(N/2).
- ➔

UnmodifiableCollection

UnmodifiableCollection is read-only only when access is through the wrapper. Changes in the backed collection will always be reflected in the unmodifiable collection.

```
13          IntStream.rangeClosed(1, 5).forEach(l::add);
14          System.out.println("l::"+l);
15
16          List<Integer> ul = Collections.unmodifiableList(l);
17          System.out.println("ul before::"+ul);
18          l.add(6);
19          System.out.println("ul after::"+ul);
```

Problems | Javadoc | Declaration | Console

```
<terminated> CollectionDemo [Java Application] C:\Users\Lenovo\.p2\pool\plugins\or
l::[1, 2, 3, 4, 5]
ul before::[1, 2, 3, 4, 5]
ul after::[1, 2, 3, 4, 5, 6]
```

Comparator consistent with Equals

The ordering imposed by a comparator on a set of elements is said to be consistent with equals if and only if c.compare(e1, e2)==0 has the same boolean value as e1.equals(e2) for every e1 and e2.

For example, suppose one adds two elements a and b such that (a.equals(b) && c.compare(a, b) != 0) to an empty TreeSet with comparator c..

The second add operation will return true (and the size of the tree set will increase) because a and b are not equivalent from the tree set's perspective, even though this is contrary to the specification of the Set.add method.

Sorted set and map internally use Tree data structure that uses comparator to compare node values rather than equals. So, in the above scenario, even though the two entries are equal as per the equals method, tree would add both of them because compareTo doesn't consider them equal which is not consistent with equals.

```
parent = t;
cmp = cpr.compare(key, t.key);
if (cmp < 0) ←
    t = t.left;
else if (cmp > 0)
    t = t.right;
else {
    V oldValue = t.value;
    if (replaceOld || oldValue == null) {
        t.value = value;
    }
    return oldValue;
```

Map

Great care must be exercised if mutable objects are used as map keys. The behaviour of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map.

https://www.nagarro.com/en/blog/post/24/performance-improvement-for-hashmap-in-java-8
https://stackoverflow.com/questions/43911369/hashmap-java-8-implementation
https://stackoverflow.com/questions/53526790/why-are-hashmaps-implemented-using-powers-of-two
https://stackoverflow.com/questions/1835976/what-is-a-sensible-prime-for-hashcode-calculation

An accurate `initialCapacity` will avoid the need to automatically rehash as the table grows. It is also possible to tweak `loadFactor`, but `0.75` (the default) provides a good balance between space and time of access. A `loadFactor` of higher than `0.75` will reduce the rehashing requirement, but access will become slower as buckets will generally become fuller. Setting the `initialCapacity` to the maximum number of elements divided by the `loadFactor` will prevent a rehash operation from occurring.

To counteract this linear effect, modern implementations of `HashMap` have a new mechanism. Once a bucket reaches a `TREEIFY_THRESHOLD`, it is converted to a bin of `TreeNode`s (and behaves similar to a `TreeMap`).

Why not do this from the beginning? `TreeNode`s are about double the size of a list node, and therefore a space cost is paid. A well-distributed hashing function will rarely cause buckets to be converted to `TreeNode`s. Should this occur in your application, it is time to consider revisiting the hash function, `initialCapacity`, and `loadFactor` settings for the `HashMap`.

Hashmap doubles the size in resizing. Since the size is power of 2, the elements from each bin either stay at the same index in the new table or move with a power of 2 offsets.

HashMap and TreeMap internally use Tree, the main difference is the ordering. TreeMap uses natural ordering or given comparator while HashMap tree uses hashes of the entries to order the nodes(and compareTo only if hashes are equal)

IdentityHashMap -
https://stackoverflow.com/questions/838528/use-cases-for-identityhashmap

Benefit of having String keys

Hash code caching

The *equals() and hashCode()* section discussed `equals()` and `hashCode()`. Hash codes should be calculated every time they are involved in hashing specific activities (for example, searching an element in a collection). Since `String` is immutable, every string has an immutable hash code that can be cached and reused as it cannot be changed after string creation. This means that hash codes of strings can be used from the cache instead of recalculating them at each usage. For example, `HashMap` hashes its keys for different operations (for example, `put()`, `get()`), and if these keys are of the `String` type, then hash codes will be reused from the cache instead of recalculating them.

Set

https://stackoverflow.com/questions/30037694/why-the-internal-implementation-of-hashset-creates-dummy-objects-to-insert-as-va

Add a custom class in Set.

- Case 1- Not implementing hashcode and equals
  - Default equality is object equality and default hashcode returns different hashcodes for diff objects. The default hashcode is simply the numerical representation of the object's memory address. So, in this case, both objects would have different hashcodes(two diff objects would have diff memory locations). However, in a rare case, there can be a hash collision if their modulo returns the same index.
  - Person p = new Person("M");
  - Person p1 = new Person("M");
  - According to default object equality, both objects are not the same. Now the set contains duplicate items. Both objects may fall into the same or diff buckets(depending on returned hashcode).
- Case 2- implementing hashcode only
  - Same as Case 1.
  - Because of default equality, both objects are different.
  - Even if we are implementing hashcode, diff objects can have the same hashcode.
  - So, if returned hashcodes are the same, duplicate objects will be added in the same bucket, otherwise, they will be in diff buckets.
- Case 3- implementing equals only
  - Depends on hashcode
  - If hashcodes are the same, duplicate objects will be ignored because of correct equals behaviour.
  - if hashcodes are diff, the set can add the same object in the diff bucket.
- Case 4- implementing equals and hashcode both
  - Correct set behaviour

PriorityQueue

Priority queue represented as a balanced binary heap: the two children of queue[n] are queue[2*n+1] and queue[2*(n+1)]

Arrays.copyOf vs System.arrayCopy

1st creates a new array as well and then internally calls System.arraycopy to copy the value. 2nd does one job - copy the elements
https://stackoverflow.com/questions/2589741/what-is-more-efficient-system-arraycopy-or-arrays-copyof
https://stackoverflow.com/questions/44487304/why-is-arrays-copyof-2-times-faster-than-system-arraycopy-for-small-arrays

Spliterator

https://www.youtube.com/watch?v=C_EuJoTWnoM

Concurrent versus synchronized collections

The obvious question is *What is the difference between a concurrent and a synchronized collection?* Well, the main difference consists of the way in which they achieve thread-safety. Concurrent collections achieve thread-safety by partitioning the data into segments. Threads can access these segments concurrently and obtain locks only on the segments that are used. On the other hand, synchronized collection locks the entire collection via *intrinsic locking* (a thread that invokes a synchronized method will automatically acquire the intrinsic lock for that method's object and release it when the method returns).

Concurrency

Misc

- Synchronized vs Locks
- Race Condition - check-then-act, read-modify-write
  - http://stackoverflow.com/questions/34510/what-is-a-race-condition
  - http://www.javacodegeeks.com/2014/08/java-concurrency-tutorial-atomicity-and-race-conditions.html
- Data Race
- A data race occurs when 2 instructions access the same memory location, at least one of these accesses is a write and there is no happens-before ordering among these accesses.
- Race condition vs Data race - https://blog.regehr.org/archives/490
- Thread-safe class loading and static initializer blocks
  http://stackoverflow.com/questions/878577/are-java-static-initializers-thread-safe
- LiveLock, starvation and deadlock
  https://richardbarabe.wordpress.com/2014/02/21/java-deadlock-livelock-and-lock-starvation-examples/
- Ways to prevent/avoid deadlock
  https://stackoverflow.com/questions/16780937/tips-to-prevent-deadlocks-in-java
- volatile and synchronized
  - http://stackoverflow.com/questions/3519664/difference-between-volatile-and-synchronized-in-java
  - http://www.infoq.com/articles/memory_barriers_jvm_concurrency

In order to write thread-safe classes, we can consider the following techniques:

- Have no *state* (classes with no instance and `static` variables)
- Have *state,* but don't share it (for example, use instance variables via `Runnable`, `ThreadLocal`, and so on)
- Have *state,* but an immutable *state*
- Use message-passing (for example, as Akka framework)
- Use `synchronized` blocks
- Use `volatile` variables
- Use data structures from the `java.util.concurrent` package
- Use synchronizers (for example, `CountDownLatch` and `Barrier`)
- Use locks from the `java.util.concurrent.locks` package

Following the Java API documentation, in cases of multithreading applications that update frequently but read less frequently, it is recommended to rely on `LongAdder`, `DoubleAdder`, `LongAccumulator`, and `DoubleAccumulator`, instead of the `AtomicFoo` classes. For such scenarios, these classes are designed to optimize the usage of threads.

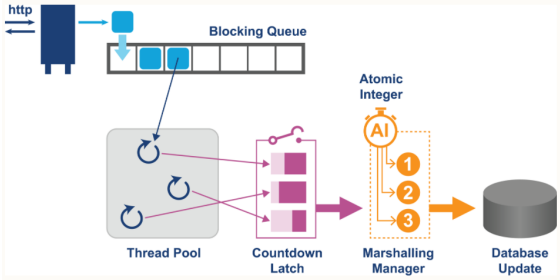


Figure 12-4. Example concurrent application

False sharing occurs because of the way CPUs handle their cache. Consider the data in this simple class:

```
public class DataHolder {
    public volatile long l1;
    public volatile long l2;
    public volatile long l3;
    public volatile long l4;
}
```

Each `long` value is stored in memory adjacent to one another; for example, `l1` could be stored at memory location 0xF20. Then `l2` would be stored in memory at 0xF28, `l3` at 0xF2C, and so on. When it comes time for the program to operate on `l2`, it will load a relatively large amount of memory—for example, 128 bytes from location 0xF00 to 0xF80—into a cache line on one of the cores of one of the CPUs. A second thread that wants to operate on `l3` will load that same chunk of memory into a cache line on a different core.

Loading nearby values like that makes sense in most cases: if the application accesses one particular instance variable in an object, it is likely to access nearby instance variables. If they are already loaded into the core's cache, that memory access is very, very fast—a big performance win.

The downside to this scheme is that whenever the program updates a value in its local cache, that core must notify all the other cores that the memory in question has been changed. Those other cores must invalidate their cache lines and reload that data from memory.

Java Memory Model

http://tutorials.jenkov.com/java-concurrency/java-memory-model.html
http://gee.cs.oswego.edu/dl/jmm/cookbook.html
https://shipilev.net/blog/2016/close-encounters-of-jmm-kind/
https://www.youtube.com/watch?v=TxqsKzxyySo&t=4s

The JMM seeks to provide answers to questions such as:

- What happens when two cores access the same data?
- When are they guaranteed to see the same value?
- How do memory caches affect these answers?

Anywhere shared state is accessed, the platform will ensure that the promises made in the JMM are honored. These promises fall into two main groups: guarantees related to ordering and those concerned with visibility of updates across threads.

At a high level, there are two possible approaches that a memory model like the JMM might take:

Strong memory model
: All cores always see the same values at all times.

Weak memory model
: Cores may see different values, and there are special cache rules that control when this may occur.

In reality, the JMM has a very weak memory model. If an application is developed on a hardware platform with a stronger memory model than the JMM, then undiscovered concurrency bugs can survive—because they do not manifest in practice due to hardware guarantees. When the same application is deployed onto weaker hardware, the concurrency bugs may become a problem as the application is no longer being protected by the hardware.

The guarantees provided by the JMM are based upon a set of basic concepts:

Happens-Before
: One event definitely happens before another.

Synchronizes-With
: The event will cause its view of an object to be synchronized with main memory.

As-If-Serial
: Instructions appear to execute in order outside of the executing thread.

Release-Before-Acquire
: Locks will be released by one thread before being acquired by another.

From this standpoint, it is immediately clear what the Java keyword `synchronized` refers to: it means that the local view of the thread holding the monitor has been Synchronized-With main memory.

Synchronized methods and blocks define touchpoints where threads must perform syncing. They also define blocks of code that must fully complete before other synchronized blocks or methods can start.

A very common mistake is to forget that operations on locked data must be treated equitably. If an application uses `synchronized` only on write operations, this can lead to lost updates.

For example, it might seem as though a read does not need to lock, but it must use `synchronized` to guarantee visibility of updates coming from other threads.

How Nested Monitor Lockout Occurs

Nested monitor lockout is a problem similar to deadlock. A nested monitor lockout occurs like this:

```
Thread 1 synchronizes on A
Thread 1 synchronizes on B (while synchronized on A)
Thread 1 decides to wait for a signal from another thread before continuing
Thread 1 calls B.wait() thereby releasing the lock on B, but not A.

Thread 2 needs to lock both A and B (in that sequence)
        to send Thread 1 the signal.
Thread 2 cannot lock A, since Thread 1 still holds the lock on A.
Thread 2 remain blocked indefinately waiting for Thread1
        to release the lock on A

Thread 1 remain blocked indefinately waiting for the signal from
        Thread 2, thereby
        never releasing the lock on A, that must be released to make
        it possible for Thread 2 to send the signal to Thread 1, etc.
```

Atomicity

Atomic support - AtomicInteger, AtomicBoolean, etc.
https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html
https://stackabuse.com/concurrency-in-java-the-volatile-keyword/

ABA Problem

The astute reader would recognize that CAS succeeds even if the value of a shared variable is changed from A to B and then back to A. Consider the following sequence:

1. A thread T1 reads the value of a shared variable as A and desires to change it to B. After reading the variable's value, thread
2. T1 undergoes a context switch. Another thread, T2 comes along, changes the value of the shared variable from A to B and then back to A from B.
3. Thread T1 is scheduled again for execution and invokes CAS with A as the expected value and B as the new value. CAS succeeds since the current value of the variable is A, even though it changed to B and then back to A in the time thread T1 was context switched.

references the same node, may not necessarily imply that the list wasn't changed. One solution to this problem is to attach a version number with the value, i.e. instead of storing the value as A, we store it as a pair (A, V1). Another thread can change the value to (B, V1) but when it changes it back to A the associated version is different i.e. (A, V2). In this way, a collision can be detected. There are two classes in Java that can be used to address the ABA problem:

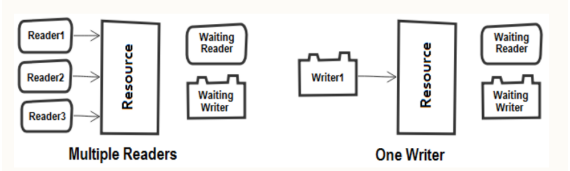
1. `AtomicStampedReference` (please see this class in our reference section for detailed explanation of ABA problem)
2. `AtomicMarkableReference`

Volatile

https://www.baeldung.com/java-volatile

ReadWriteLock

ReadLock.lock() - Acquires the read lock if the write lock is not held by another thread and returns immediately. If the write lock is held by another thread then the current thread becomes disabled for thread scheduling purposes and lies dormant until the read lock has been acquired.
WriteLock.lock() - Acquires the write lock if neither the read nor write lock are held by another thread and returns immediately, setting the write lock hold count to one. If the current thread already holds the write lock then the hold count is incremented by one and the method returns immediately. If the lock is held by another thread then the current thread becomes disabled for thread scheduling purposes and lies dormant until the write lock has been acquired, at which time the write lock hold count is set to one.



ConcurrentLinkedQueue vs LinkedBlockingQueue

https://stackoverflow.com/questions/1426754/linkedblockingqueue-vs-concurrentlinkedqueue
ConcurrentLinkedQueue is non-blocking so the poll will never be blocked if the queue is empty so not preferable in usual producer/consumer scenarios.
Should be used when you don't need to block.
Java's ConcurrentLinkedQueue is not only non-blocking, but it has the awesome property that the producer does not contend with the consumer. In a single producer / single consumer scenario (SPSC), this really means that there will be no contention to speak of. In a multiple producer / single consumer scenario, the consumer will not contend with the producers. This queue does have contention when multiple producers try to offer(), but that's concurrency by definition. It's basically a general purpose and efficient non-blocking queue.

Concurrency and Multithreading

In computer science, concurrency is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other. The computations may be executed on multiple cores in the same chip, preemptively time-shared threads on the same processor, or executed on physically separated processors.

In computer architecture, multithreading is the ability of a central processing unit or a single core in a multi-core processor to execute multiple processes or threads concurrently, appropriately supported by the operating system. Where multiprocessing systems include multiple complete processing units, multithreading aims to increase the utilization of a single core by using thread-level as well as instruction-level parallelism.
http://www.howtogeek.com/194756/cpu-basics-multiple-cpus-cores-and-hyper-threading-explained/
http://superuser.com/questions/329904/what-are-threads-and-what-do-they-do-in-the-processor

Thread

If a thread group is not passed, JVM tries to add the thread to its parent's group by default.

Interruption

Interrupt

Interrupts the thread. Interrupting a thread that is not alive need not have any effect.

```
* <p> If this thread is blocked in an invocation of the {@link
* Object#wait() wait()}, {@link Object#wait(long) wait(long)}, or {@link
* Object#wait(long, int) wait(long, int)} methods of the {@link Object}
* class, or of the {@link #join()}, {@link #join(long)}, {@link
* #join(long, int)}, {@link #sleep(long)}, or {@link #sleep(long, int)},
* methods of this class, then its interrupt status will be cleared and it
* will receive an {@link InterruptedException}.
*
* <p> If this thread is blocked in an I/O operation upon an {@link
* java.nio.channels.InterruptibleChannel InterruptibleChannel}
* then the channel will be closed, the thread's interrupt
* status will be set, and the thread will receive a {@link
* java.nio.channels.ClosedByInterruptException}.
*
* <p> If this thread is blocked in a {@link java.nio.channels.Selector}
* then the thread's interrupt status will be set and it will return
* immediately from the selection operation, possibly with a non-zero
* value, just as if the selector's {@link
* java.nio.channels.Selector#wakeup wakeup} method were invoked.
*
* <p> If none of the previous conditions hold then this thread's interrupt
* status will be set. </p>
*
```

isInterrupted - instance method

```
/**
 * Tests whether this thread has been interrupted.  The <i>interrupted
 * status</i> of the thread is unaffected by this method.
 *
 * <p>A thread interruption ignored because a thread was not alive
 * at the time of the interrupt will be reflected by this method
 * returning false.
 *
 * @return  {@code true} if this thread has been interrupted;
 *          {@code false} otherwise.
 * @see     #interrupted()
 * @revised 6.0
 */
public boolean isInterrupted() {
    return isInterrupted(false);
}
```

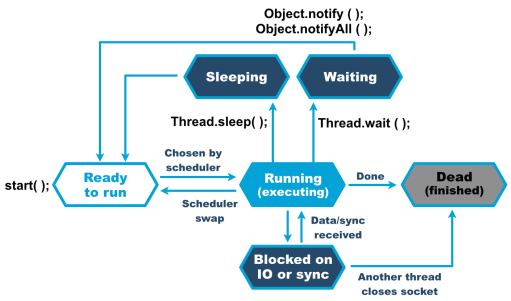
Interrupted - static method

Checks and reset the interrupt status of the thread.

```
/**
 * Tests whether the current thread has been interrupted.  The
 * <i>interrupted status</i> of the thread is cleared by this method.  In
 * other words, if this method were to be called twice in succession, the
 * second call would return false (unless the current thread were
 * interrupted again, after the first call had cleared its interrupted
 * status and before the second call had examined it).
 *
 * <p>A thread interruption ignored because a thread was not alive
 * at the time of the interrupt will be reflected by this method
 * returning false.
 *
 * @return  {@code true} if the current thread has been interrupted;
 *          {@code false} otherwise.
 * @see #isInterrupted()
 * @revised 6.0
 */
public static boolean interrupted() {
    return currentThread().isInterrupted(true);
}
```

Thread States

- New
- Runnable(executing in the Java virtual machine)
- Blocked(waiting for a monitor lock)
- Waiting(waiting for another thread to perform a particular action.)
- Timed-Waiting
- Terminated



Blocked vs Waiting

When a thread calls the Object.wait method, it releases all the acquired monitors and is put into WAITING (or TIMED_WAITING if we call the timeout versions of the wait method) state. Now when the thread is notified either by notify() or by notifyAll() call on the same object then the waiting state of the thread ends and the thread starts attempting to regain all the monitors which it had acquired at the time of the wait call. At one time there may be several threads trying to regain (or maybe gain for the first time) their monitors. If more than one threads attempt to acquire the monitor of a particular object then only one thread (selected by the JVM scheduler) is granted the monitor and all other threads are put into BLOCKED state.

Wait vs Sleep

https://stackoverflow.com/questions/1036754/difference-between-wait-and-sleep

```
synchronized(LOCK) {
    Thread.sleep(1000); // LOCK is held
}


synchronized(LOCK) {
    LOCK.wait(); // LOCK is not held
}
```

A `wait` can be "woken up" by another thread calling `notify` on the monitor which is being waited on whereas a `sleep` cannot. Also a `wait` (and `notify`) must happen in a block `synchronized` on the monitor object whereas `sleep` does not:

```
Object mon = ...;
synchronized (mon) {
    mon.wait();
}
```

At this point the currently executing thread waits *and releases the monitor*. Another thread may do

```
synchronized (mon) { mon.notify(); }
```

(on the same `mon` object) and the first thread (assuming it is the only thread waiting on the monitor) will wake up.

You can also call `notifyAll` if more than one thread is waiting on the monitor – this will wake *all of them up*. However, only one of the threads will be able to grab the monitor (remember that the `wait` is in a `synchronized` block) and carry on – the others will then be blocked until they can acquire the monitor's lock.

Another point is that you call `wait` on `Object` itself (i.e. you wait on an object's monitor) whereas you call `sleep` on `Thread`.

Yet another point is that you can get *spurious wakeups* from `wait` (i.e. the thread which is waiting resumes for no apparent reason). You should **always** `wait` **whilst spinning on some condition** as follows:

```
synchronized {
    while (!condition) { mon.wait(); }
}
```

onSpinWait()

```
/**
 * Indicates that the caller is momentarily unable to progress, until the
 * occurrence of one or more actions on the part of other activities. By
 * invoking this method within each iteration of a spin-wait loop construct,
 * the calling thread indicates to the runtime that it is busy-waiting.
 * The runtime may take action to improve the performance of invoking
 * spin-wait loop constructions.
 *
 * @apiNote
 * As an example consider a method in a class that spins in a loop until
 * some flag is set outside of that method. A call to the {@code onSpinWait}
 * method should be placed inside the spin loop.
 * <pre>{@code
 *     class EventHandler {
 *         volatile boolean eventNotificationNotReceived;
 *         void waitForEventAndHandleIt() {
 *             while ( eventNotificationNotReceived ) {
 *                 java.lang.Thread.onSpinWait();
 *             }
 *             readAndProcessEvent();
 *         }
 *
 *         void readAndProcessEvent() {
 *             // Read event from some source and process it
 *              . . .
 *         }
 *     }
 * }</pre>
 * <p>
 * The code above would remain correct even if the {@code onSpinWait}
 * method was not called at all. However on some architectures the Java
 * Virtual Machine may issue the processor instructions to address such
 * code patterns in a more beneficial way.
 *
 * @since 9
 */
@HotSpotIntrinsicCandidate
public static void onSpinWait() {}
```

ThreadLocal

ThreadLocal internally holds a ThreadLocalMap to store a private copy of an object. The key of this map is the thread calling ThreadLocal and the value is the object.

First get() call may invoke initialValue() if set() has not been called already.
If not overridden, initialValue simply returns null.

Each thread T has a reference to ThreadLocalMap that is used by all ThreadLocal objects that T will ever create.
The 1st ThreadLocal would initialize T's threadlocal map and use this(referring to ThreadLocal object) as map key.

T{
ThreadLocal1 of int;
ThreadLocal2 of string;
}
T.threadlocalmap.put(ThreadLocal1, somevalue1);
T.threadlocalmap.put(ThreadLocal2, somevalue1);

Get - get T.threadlocalmap.get(this)

ThreadLocal and Memory Leaks-
https://javapapers.com/core-java/threadlocal/
http://stackoverflow.com/questions/17968803/threadlocal-memory-leak
http://www.appneta.com/blog/introduction-to-javas-threadlocal-storage/

Future

```
/**
 * Attempts to cancel execution of this task.  This attempt will
 * fail if the task has already completed, has already been cancelled,
 * or could not be cancelled for some other reason. If successful,
 * and this task has not started when {@code cancel} is called,
 * this task should never run.  If the task has already started,
 * then the {@code mayInterruptIfRunning} parameter determines
 * whether the thread executing this task should be interrupted in
 * an attempt to stop the task.
 *
 * <p>After this method returns, subsequent calls to {@link #isDone} will
 * always return {@code true}.  Subsequent calls to {@link #isCancelled}
 * will always return {@code true} if this method returned {@code true}.
 *
 * @param mayInterruptIfRunning {@code true} if the thread executing this
 * task should be interrupted; otherwise, in-progress tasks are allowed
 * to complete
 * @return {@code false} if the task could not be cancelled,
 * typically because it has already completed normally;
 * {@code true} otherwise
 */
boolean cancel(boolean mayInterruptIfRunning);
```

Semaphore

An abstraction that allows n units to be acquired, and offers acquire and release mechanisms. Only n threads can access a resource at any given time.

> A one-permit semaphore (binary semaphore) is equivalent to a mutex, but with one distinct difference. A mutex can only be released by a thread that the mutex is locked on, whereas a semaphore can be released by a nonowning thread. A scenario where this might be necessary would be forcing the resolution of a deadlock. Semaphores also have the advantage of being able to ask for and release multiple permits. If multiple permits are being used, it is essential to use fair mode; otherwise, there is an increased chance of thread starvation.

CyclicBarrier

n threads wait for each other. When nth thread arrives, all threads resume their execution.
State can be reset, additional barrier action can be given.
Generally, the last arriving thread runs the barrier action if provided.

> This synchronizer works well for problems that rely on a task that can be divided into subtasks. Each subtask runs in a different thread and waits for the rest of the threads. When all the threads complete, they combine their results in a single result.

```
* A synchronization aid that allows a set of threads to all wait for
* each other to reach a common barrier point.  CyclicBarriers are
* useful in programs involving a fixed sized party of threads that
* must occasionally wait for each other. The barrier is called
* <em>cyclic</em> because it can be re-used after the waiting threads
* are released.
*
* <p>The {@code CyclicBarrier} uses an all-or-none breakage model
* for failed synchronization attempts: If a thread leaves a barrier
* point prematurely because of interruption, failure, or timeout, all
* other threads waiting at that barrier point will also leave
* abnormally via {@link BrokenBarrierException} (or
* {@link InterruptedException} if they too were interrupted at about
* the same time).
*
```

CountdownLatch

A single thread waits for n threads to complete some action. All other threads may be independent of each other. When all n threads complete some task, the waiting thread resumes its execution.
Different from CyclicBarrier as other threads do not wait for the waiting thread, they just call countDown() to decrement the count while in CyclicBarrier, all threads wait for each other.

```
* A synchronization aid that allows one or more threads to wait until
* a set of operations being performed in other threads completes.
```

> The idea that all threads must be fully stopped before the STW phase can commence is similar to the use of latches, such as that implemented by `CountDownLatch` in the `java.util.concurrent` library.

Executor

Interface with a single method - execute() that accepts a Runnable.

ExecutorService

A sub-interface with lifecycle methods.
shutdown()

```
/**
 * Initiates an orderly shutdown in which previously submitted
 * tasks are executed, but no new tasks will be accepted.
 * Invocation has no additional effect if already shut down.
 *
 * <p>This method does not wait for previously submitted tasks to
 * complete execution.  Use {@link #awaitTermination awaitTermination}
 * to do that.
 *
```

shutdownNow()

```
/**
 * Attempts to stop all actively executing tasks, halts the
 * processing of waiting tasks, and returns a list of the tasks
 * that were awaiting execution.
 *
 * <p>This method does not wait for actively executing tasks to
 * terminate.  Use {@link #awaitTermination awaitTermination} to
 * do that.
 *
 * <p>There are no guarantees beyond best-effort attempts to stop
 * processing actively executing tasks.  For example, typical
 * implementations will cancel via {@link Thread#interrupt}, so any
 * task that fails to respond to interrupts may never terminate.
 *
```

ThreadPoolExecutor

Pool always creates a new worker thread if workers < corepoolsize
If corepoolsize<workers<maxPoolSize - add a new worker only if the workqueue is full.
Using an unbounded queue without a predefined will cause new tasks to wait in the queue when all corePoolSize threads are busy. Thus, no more than corePoolSize threads will ever be created unless corePoolSize and maxPoolSize are the same.

prestartCoreThread/prestartAllCoreThreads - to prestart core threads before submitting a job otherwise core threads are created as tasks start coming i.e. not until the 1st task arrives.

Task Rejection
If a request cannot be queued(bounded queue or hand-off queue like SynchronousQueue), a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.
When the pool has been shut down.
Rejection Policy

- AbortPolicy
- CallerRunsPolicy
- DiscardPolicy
- DiscardOldestPolicy

How to get any exception thrown by executor service?
override afterExecute(Runnable r, Throwable t) method and use passed throwable.

States

```
* The runState provides the main lifecycle control, taking on values:
*
*   RUNNING:  Accept new tasks and process queued tasks
*   SHUTDOWN: Don't accept new tasks, but process queued tasks
*   STOP:     Don't accept new tasks, don't process queued tasks,
*             and interrupt in-progress tasks
*   TIDYING:  All tasks have terminated, workerCount is zero,
*             the thread transitioning to state TIDYING
*             will run the terminated() hook method
*   TERMINATED: terminated() has completed
*
* The numerical order among these values matters, to allow
* ordered comparisons. The runState monotonically increases over
* time, but need not hit each state. The transitions are:
*
* RUNNING -> SHUTDOWN
*    On invocation of shutdown()
* (RUNNING or SHUTDOWN) -> STOP
*    On invocation of shutdownNow()
* SHUTDOWN -> TIDYING
*    When both queue and pool are empty
* STOP -> TIDYING
*    When pool is empty
* TIDYING -> TERMINATED
*    When the terminated() hook method has completed
*
```

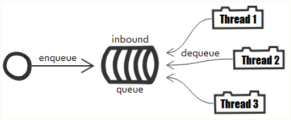
Each worker blocks for a submitted task as it calls task.run() not task.start

Worker continuously polls the workqueue:
  For each task:
    Locks its own worker object
    Runs task
    Unlocks

This lock is used to check if a worker is busy or idle. All idle workers are interrupted when shutdown is called which basically tries to lock the worker object before interrupting a worker.
On the other hand, shutdownNow doesn't care about the worker lock, it simply loops the worker set and interrupts all.

Difference b/w Old Executors and new Work Stealing Pool

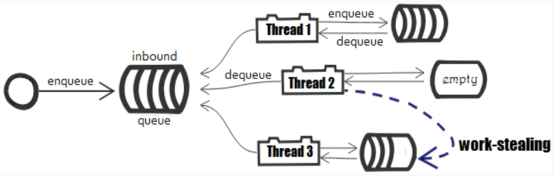
Old Executors



So, a thread pool relies on an internal inbound queue to store tasks. Each thread must dequeue a task and execute it. This is suitable for cases when the tasks are time-consuming and their number is relatively low. On the other hand, if these tasks are many and are small (they require a small amount of time to be executed), there will be a lot of contentions as well. This is not good, and even if this is a lock-free queue the problem is not entirely solved.

Work Stealing Pool/Executor

In order to reduce contentions and increase performance, a thread pool can rely on a work-stealing algorithm and a queue per thread. In this case, there is a central inbound queue for all tasks, and an extra queue (known as the local task queue) for each thread (worker thread), as in the following diagram:



So, each thread will dequeue tasks from the central queue and enqueue them in their own queue. Each thread has its own local queue of tasks. Further, when a thread wants to process a task, it simply dequeues a task from its own local queue. As long as its local queue is not empty, the thread will continue to process the tasks from it without bothering other threads (no contentions with other threads). When its local queue is empty (as in the case of **Thread 2** in the preceding diagram), it tries to steal (via a working-stealing algorithm) tasks from local queues that belong to other threads (for example, **Thread 2** steals tasks from **Thread 3**). If it doesn't find anything to steal, it accesses the shared central inbound queue.

Each local queue is actually a **deque** (short for **double-ended queue**), therefore it can be accessed efficiently from both ends. The thread sees its deque as a stack, meaning that it will enqueue (add new tasks) and dequeue (take tasks for processing) from only one end. On the other hand, when a thread tries to steal from the queue of another thread, it will access the other end (for example, **Thread 2** steals from **Thread 3** queue from the other end). So, tasks are processed from one end and stolen from the other end.

Fork/Join

Starting with JDK 8, we can do it as follows:

```
ForkJoinPool forkJoinPool = ForkJoinPool.commonPool();
```

Both approaches avoid the unpleasant situation of having too many pool threads on a single JVM, caused by the parallel operations that created their own pools.

- `RecursiveAction` for the `void` tasks
- `RecursiveTask<V>` for tasks that return a value
- `CountedCompleter<T>` for tasks that need to remember the pending task count

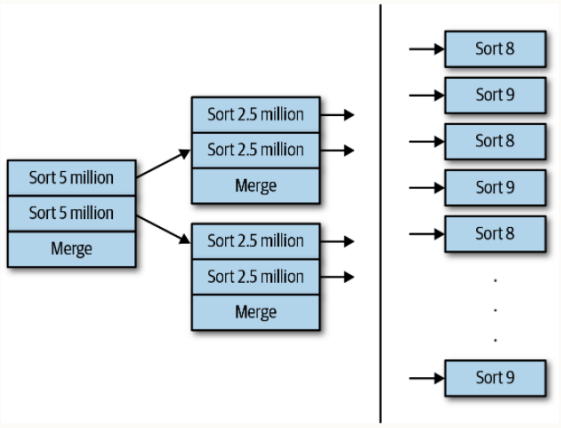


Figure 9-1. Tasks in a recursive quicksort

An additional 131,072 tasks are needed to merge those sorted arrays, 65,536 additional tasks to merge the next set of sorted arrays, and so on. In the end, there will be 524,287 tasks.

The larger point here is that none of the tasks can complete until the tasks that they have spawned have also completed. The tasks directly sorting arrays of fewer than 47 elements must be completed first, and then tasks can merge the two small arrays that they created, and so on: everything is merged up the chain until the entire array is merged into its final, sorted value.

It isn't possible to perform that algorithm efficiently using `ThreadPoolExecutor`, because a parent task must wait for its child tasks to complete. A thread inside a thread-pool executor cannot add another task to the queue and then wait for it to finish: once the thread is waiting, it cannot be used to execute one of the subtasks. `ForkJoinPool`, on the other hand, allows its threads to create new tasks and then suspend their current task. While the task is suspended, the thread can execute other pending tasks.

ConcurrentHashMap

- Retrieval operations generally do not block, so may overlap with update operations. Retrievals reflect the results of the most recently completed update operations holding upon their onset. (More formally, an update operation for a given key bears a happens-before relation with any (non-null) retrieval for that key reporting the updated value.)
- For aggregate operations such as {@code putAll} and {@code clear}, concurrent retrievals may reflect insertion or removal of only some entries.
- Similarly, Iterators, Spliterators and Enumerations return elements reflecting the state of the hash table at some point at or since the creation of the iterator/enumeration. They do not throw ConcurrentModificationException.
- However, iterators are designed to be used by only one thread at a time.
- Same as HashMap, but it locks the first node of the bin rather than the segment(not anymore).

Threaddump

Table 2-6 Thread States for a Thread Dump

Thread State	Description
NEW	The thread has not yet started.
RUNNABLE	The thread is executing in the JVM.
BLOCKED	The thread is blocked waiting for a monitor lock.
WAITING	The thread is waiting indefinitely for another thread to perform a particular action.
TIMED_WAITING	The thread is waiting for another thread to perform an action for up to a specified waiting time.
TERMINATED	The thread has exited.

Wait Set, wait/notify

Every object has a wait-set along with a monitor, add/remove from the wait set is atomic.
Wait set is manipulated by wait/notify.

When a thread t calls wait on an object m

- if t doesn't have m's lock
  - throw IllegalMonitorStateException.
- If t has m's lock
  - Is it a timed wait?
    - if yes, check time params
    - If invalid, throw IllegalArgumentException
  - If not timed wait or (timed wait and params valid)
  - check if t is interrupted
    - if interrupted throw InterruptedException
  - If no
    - Add t to m's wait set and release m's lock
- T remains in m's wait set until - notify / notifyAll / interrupt / timed wait / some spurious wake up implemented by JVM vendor

Interrupt/notify order - jvm impl dependent

Notify > interrupt - t returns from wait set with interrupt pending
Interrupt > notify - t returns from wait set with interrupt set to false and some other thread(if exists in wait set) must receive the notify call

T calls m.notify()

- t acquires m's lock
- one random thread X from wait set is selected by jvm and removed from wait set
- X tries to lock m but can't succeed until t releases m.
- t.interrupt() - sets t's interrupt state, If t is in the wait set of any object, it will be removed from the wait set and will throw an InterruptedException after relocking the monitor.

Busy Wait

Implementing guards via waiting and notification methods is nearly always superior to using an optimistic-retry-style busy-wait "spinloop" of the form:

```
protected void busyWaitUntilCond() {
  while (!cond)
    Thread.yield();
}
```

Busy-waits can waste an unbounded amount of CPU time spinning uselessly. The wait-based version rechecks conditions only when some other thread provides a notification that the object's state has changed, thus possibly affecting the guard condition.
The main exceptions are those cases in which you somehow know that the condition must become true within some very short, bounded amount of time. In such cases, the time wasted spinning might be less than the time required to suspend and resume threads.
Bounded spins, followed by suspensions, are also sometimes used inside run-time systems as an optimization for "adaptive" locks that are usually held only briefly.
Why prefer wait
For example, if the spinning task is running at a high priority but the threads that change the condition run at low priority, the spinning task may still crowd the others out. In the wait-based version, the waiting task does not run at all, and thus cannot encounter such scheduling problems. Some of the most plausible situations for using spinloops are those in which some other action can be taken within the loop if the condition is not true, rather than just yielding.
Another example,
Balking on attempts to pop an element from an empty stack is attractive since it makes the class usable in sequential settings where it would be pointless to wait for a pop: if no other threads can add an element, the program will just stall forever. On the other hand, some clients of a Stack in concurrent contexts might want to hold up and wait for an element to appear. One inefficient approach is to try to perform pop and if a StackEmptyException is caught, to try again. This is a disguised form of busy-waiting.

Spin Locks

You initialize a counter to some value and do a pthread_mutex_trylock()—that takes about 2 µs. If you don't get the lock, decrement the counter and loop. Another 2 µs. Repeat. When the counter hits zero, give up and block. If you get the mutex, you've saved a bunch of time. If you don't, you've only wasted a little time.
Spin locks can be effective in very restricted circumstances. The critical section must be short, you must have significant contention for the lock, and you must be running on more than one CPU. If you do decide you need a spin lock, test that assumption.

Locks and Spinlocks

The intrinsic locks that we have met up until now work by invoking the operating system in user code. The OS is used to put a thread into an indefinite wait until signaled. This can be a huge overhead if the contended resource is only in use for a very short period of time. In this case, it may be much more efficient to have the blocked thread stay active on a CPU, accomplish no useful work, and "burn CPU" retrying the lock until it becomes available.

This technique is known as a *spinlock* and is intended to be more lightweight than a full mutual-exclusion lock. In modern systems, spinlocks are usually implemented with CAS, assuming that the hardware supports it. Let's look at a simple

Of course, lock-free techniques also come at a cost. Occupying a CPU core is expensive in terms of utilization and power consumption: the computer is going to be busier doing nothing, but busier also implies hotter, which means more power will be required to cool down the core that's processing nothing.

Adaptive Spin Locks

A refinement of spin locks, called adaptive spin locks, is used in many kernels. If you could find out whether the thread holding the desired mutex was in fact currently running on a CPU, you could make a more reasoned judgment as to whether or not to spin. An adaptive lock can do this. If the mutex owner is running, the requestor spins. If the owner isn't, the requestor doesn't.

Spurious wakeup

In spurious wakeups, a thread might be awoken from its waiting state even though no thread signaled the condition variable. For correctness it is necessary, then, to verify that the condition is indeed true after the thread has finished waiting. Because spurious wakeup can happen repeatedly, this is achieved by waiting inside a loop that terminates when the condition is true.

Why does thread scheduler allows spurious wakeups
Just think of it... like any code, thread scheduler may experience temporary blackouts due to something abnormal happening in underlying hardware / software. Of course, care should be taken for this to happen as rare as possible, but since there's no such thing as 100% robust software it is reasonable to assume this can happen and take care on the graceful recovery in case if scheduler detects this (eg. by observing missing heartbeats).
Now, how could the scheduler recover, taking into account that during blackout it could miss some signals intended to notify the waiting threads? If the scheduler does nothing, the mentioned "unlucky" threads will just hang, waiting forever - to avoid this, the scheduler would simply send a signal to all the waiting threads.
This makes it necessary to establish a "contract" that waiting thread can be notified without a reason. To be precise, there would be a reason - scheduler blackout - but since thread is designed (for a good reason) to be oblivious to scheduler internal implementation details, this reason is likely better to present as "spurious".
Source-
https://softwareengineering.stackexchange.com/questions/186842/spurious-wakeups-explanation-sounds-like-a-bug-that-just-isnt-worth-fixing-is

wait() / notify()

Hoare vs Mesa Monitors

So far we have determined that the idiomatic usage of a monitor requires using a while loop as follows. Let's see how the design of monitors affects this recommendation.

```
while( condition == false ) {
    condVar.wait();
}
```

Once the asleep thread is signaled and wakes up, you may ask why does it need to check for the condition being false again, the signaling thread must have just set the condition to true?

In **Mesa monitors** - Mesa being a language developed by Xerox researchers in the 1970s - it is possible that the time gap between thread B calls `notify()` and releases its mutex **and** the instant at which the asleep thread A, wakes up and reacquires the mutex, ***the predicate is changed back to false by another thread different than the signaler and the awoken threads!*** The woken up thread competes with other threads to acquire the mutex once the signaling thread B **empties** the monitor. On signaling, thread B doesn't give up the monitor just yet; rather it continues to own the monitor until it exits the monitor section.

In contrast, **Hoare monitors** - Hoare being one of the original inventor of monitors - the signaling thread B ***yields*** the monitor to the woken up thread A and thread A ***enters*** the monitor, while thread B sits out. This guarantees that the predicate will not have changed and instead of checking for the predicate in a while loop an if-clause would suffice. The woken-up/released thread A immediately starts execution when the signaling thread B signals that the predicate has changed. No other thread gets a chance to change the predicate since no other thread gets to enter the monitor.

Java, in particular, subscribes to Mesa monitor semantics and the developer is always expected to check for condition/predicate in a while loop. Mesa monitors are more efficient than Hoare monitors.

The problem with calling wait() and notify() on the empty string, or any other constant string is, that the JVM/Compiler internally translates constant strings into the same object. That means, that even if you have two different MyWaitNotify instances, they both reference the same empty string instance. This also means that threads calling doWait() on the first MyWaitNotify instance risk being awakened by doNotify() calls on the second MyWaitNotify instance.

What exactly is wait and notify

Calling `start()` on a thread instance causes a new line of execution to begin, JVM creates a new thread, concurrently running with the parent thread. JVM allows an application to have multiple threads running concurrently. As a thread runs concurrently, unaware of other threads, what happens when it needs to coordinate with some other thread ? How do different threads communicate ?

State dependence

It is possible for multiple threads to share a resource. If this resource is mutable i.e. threads can update it or its state can be updated by some way, then, in particular scenarios a thread might need this shared resource to be in a desired state, in order to complete its operation / execution. It means that thread can't continue, if the resource is not in the desired state. It is also possible in a concurrent environment that another thread updates the resource meanwhile in such a way that allows the first thread to continue its execution, which was not possible earlier. This is called state-dependence.

Example, Suppose, a download server `S` (shared resource) is shared among multiple down-loader threads. A down-loader thread `T` can't download a file, if `S` is not up. `T` needs `S` to be in a particular state (`UP`, a precondition that must be true) so that it can download a file (`T`'s operation). There is another thread `W` whose job is to manage the life-cycle of `S`. When `W` starts `S` i.e. `S` is in `UP` state, now `T` can continue its execution. This is an example of **state-dependent action** in a concurrent environment, also we need some mechanism so that `W` can inform `T` that `S` is now `UP`.

Some actions may have state-based preconditions that need not always hold when a thread is trying to do something. How do we manage such state-based preconditions ?

To work this out, we can write code in such a way that actions succeed only when state-based preconditions hold, **check-then-act**. If preconditions do not hold, possible actions could be-

1. Balking : Throwing an exception if the precondition fails.
2. Guarded suspension : Suspending the current method invocation (and its associated thread) until the precondition becomes true.
3. Time-out : An upper bound is placed on how long to wait for the precondition to become true.

Guarded suspension

Guarded suspension is a software design pattern for managing operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed. The guarded suspension pattern is typically applied to method calls (guarded methods) in object-oriented programs, and involves suspending the method call, and the calling thread, until the precondition (acting as a guard) is satisfied. - Wikipedia

Guard

A guard is a boolean expression that must evaluate to true if the program execution is to continue in the branch in question.

Guarded block

Threads often have to coordinate their actions. The most common coordination idiom is the guarded block. Such a block begins by polling a condition that must be true before the block can proceed. Please refer to this link for more details on guarded blocks.

wait() / notify() mechanism in Java

`wait()/notify()` in Java can be used as a state-based concurrency control construct. It is a way to implement guarded blocks. Threads use `wait() / notify()` to communicate with each other, signal notifications and to coordinate their actions with each other.

For each condition that needs to be waited on, we write a guarded wait that causes the current thread to block if the guard condition is false.

Then, every method causing state changes that affect the precondition, notifies threads waiting for state changes, causing them to wake up and recheck their guard conditions.

How does it work?

`wait/notify` is a way to implement guarded methods in order to follow the design principle of guarded suspension. A thread checks a precondition to check if the current state allows its desired action. If the precondition doesn't hold, the thread goes to the wait state; lies dormant there and waits for the precondition to be true. As a result of some activity (or another thread) when the precondition becomes true, another thread notifies the waiting thread to resume its operation.

A thread goes to the wait state by calling `wait()` on an object, then another thread notifies the waiting thread by calling `notify()` on the same object.

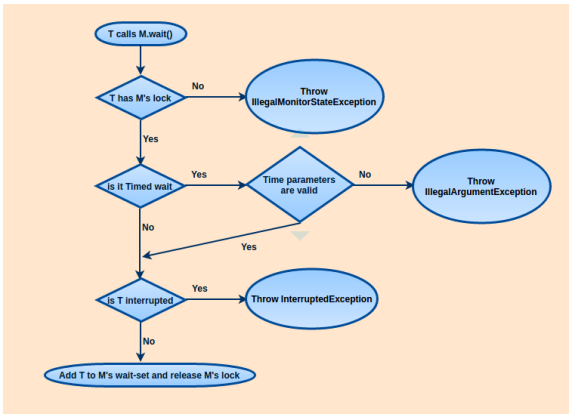
Java synchronization is implemented using monitors. Entities possessing both locks and wait-sets are generally called monitors (although almost every language defines details somewhat differently). In Java, every object is associated with a monitor, any object can serve as a monitor.

Wait-set of an object

Every object has a wait-set along with its lock. The wait set for each object is maintained internally by the JVM. Each set holds threads blocked by wait on the object until corresponding notifications are invoked or the waits are otherwise released. Add / remove from the wait set is atomic and manipulated only by methods `wait, notify, notifyAll and Thread.interrupt`.

The methods `wait, notify, and notifyAll` may be invoked only when the synchronization lock is held on their targets. Failure to comply causes these operations to throw an `IllegalMonitorStateException` at run time.

wait()



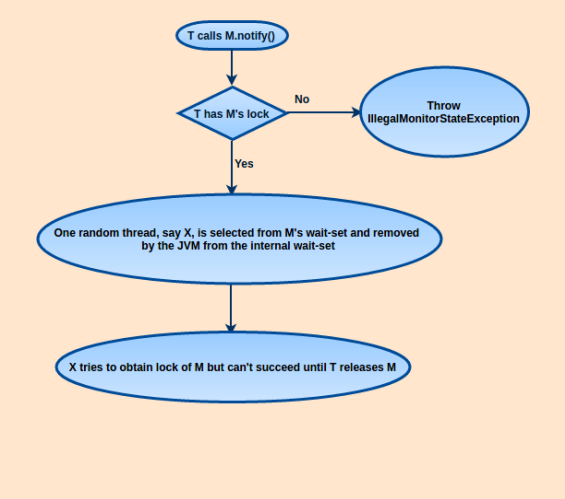
`wait()` is a final method of `Object` class that causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()`. A `wait()` invocation results in the following actions:

When a thread `T` calls `wait()` on an object `M`,

If the current thread has been interrupted, then the method exits immediately, throwing an `InterruptedException`. Otherwise, the JVM places the thread in the internal and otherwise inaccessible wait set associated with the target object.

The synchronization lock for the target object is released, but all other locks held by the threads are retained. A full release is obtained even if the lock is re-entrantly held due to nested synchronized calls on the target object. Upon later resumption, the lock status is fully restored.

notify()



`notify()` also is a final method of `Object` class same as `wait()`. It wakes up a single thread that is waiting on this object's monitor. A `notify` invocation results in the following actions:
When a Thread `T` calls `notify()` on an object `M`,
There is no guarantee about which waiting thread will be selected when the wait set contains more than one thread, choice is arbitrary and depends on the JVM implementation. `X` must re-obtain the synchronization lock for the target object, which will always cause it to block at least until the thread calling notify (`T`) releases the lock. It will continue to block if some other thread obtains the lock first.

NotifyAll
A `notifyAll` works in the same way as notify except that the steps occur (in effect,simultaneously) for all threads in the wait set for the object. However, because they must acquire the lock, threads continue one at a time.
Note that within synchronized methods, the order in which a notifyAll is placed does not matter. No awakened threads will be able to continue until the synchronization lock is released. Just as a matter of style, most people put notifications last in method bodies.

Interrupt : If `Thread.interrupt` is invoked for a thread suspended in a wait, the same `notify` mechanics apply, except that after re-acquiring the lock, the method throws an `InterruptedException` and the thread's interruption status is set to false.

Timed Waits : The timed versions of the `wait` method, `wait(long msecs) and wait(long msecs , int nanosecs)`, take arguments specifying the desired maximum time to remain in the wait set. They operate in the same way as the untimed version except that if a wait has not been notified before its time bound, it is released automatically.

Wait, notify and interrupt

We know from the previous post that the wait-set of an object holds all the blocked threads on `wait()` and wait-set is manipulated only by methods `wait, notify, notifyAll and interrupt`. There are two possible ways for a waiting thread to leave the wait-set of an object - notification and interrupt.
This post discusses the scenario when a thread is both notified and interrupted while waiting.

Interruption / Notification order
Interruption and notification order is implementation dependent, a JVM implementer is free to choose any ordering as long as it follows JLS and VM specification rules.

notify > interrupt : If `notify` is ordered before `interrupt`, then
An arbitrarily chosen thread returns normally from the wait-set with an interrupt pending.
interrupt > notify : If `interrupt` is ordered before `notify`, then
A waiting thread returns from the wait-set by throwing an InterruptedException and some other thread in the wait-set must receive the notification.

Notifications cannot be lost due to interrupts. Assume that a set `s` of threads is in the wait set of an object `m`, and another thread performs a `notify` on `m`. Then either:
at least one thread in `s` must return normally from wait , or
all of the threads in `s` must exit wait by throwing `InterruptedException`

Another distinction between `notify()` and `interrupt` could be,
`notify()` is invoked on a target object (used as a monitor whose wait-set will be used to hold waiting thread) so which thread will receive the notification in the wait-set is decided by the runtime environment; the selection process depends on the implementation.

But `interrupt` is always invoked on a known thread, `interrupt` is aware of its target thread. `interrupt` doesn't need to acquire the monitor of its target thread, and target thread is not required to be in the wait-set of any object.

Synchronized

Locking provides protection against both high-level and low-level conflicts by enforcing atomicity against methods and code-blocks synchronized on the same object.
Some default rules

1. Always lock during updates to object fields.
2. Always lock during access of possibly updated object fields.
3. Never lock when invoking methods on other objects.

Implementation of synchronization requires the existence of an atomic test and set instruction in hardware. This is true for both uniprocessor and multiprocessor machines. Because threads can be preempted at any time, between any two instructions, you must have such an instruction.
A test and set instruction tests (or just loads into a register) a word from memory and sets it to some value (typically, 1), all in one instruction, with no possibility of anything happening in between the two halves (e.g., an interrupt or a write by a different CPU). All synchronization is based upon the existence of this instruction. The one type of instruction that is substantially different is the **compare and swap** instruction, which compares one word of main memory with a register and swaps the contents of that word with a second register when equal.

POSIX defines three synchronization variables and the function pthread_join() to provide this functionality. Java provides the same functionality by encapsulating synchronization variables within every object. These synchronization variables are manipulated by means of a keyword (synchronized), thread.join(), and several methods on Object.

There are two basic things you want to do. The first is that you want to protect shared data. This is what locks do. The second is that you want to prevent threads from running when there's nothing for them to do. You don't want them spinning, wasting time. This is what semaphores, condition variables, wait sets, join(), barriers, etc., are for.

The class Object (and hence any subclass) has two private instance variables - mutex and wait-set. The class object itself is a subclass of Object; hence it too has a mutex and wait set. This mutex can be used to protect static data. It is used for static synchronized methods. The class lock may be used to protect class internals during instance creation.
One important thing,

```
public class Foo {
        static int count = 0;
        public void inc(int i) {
                synchronized (Foo.class) {
                        count = count + i;
                }
        }
        // cf: getClass()
}
```

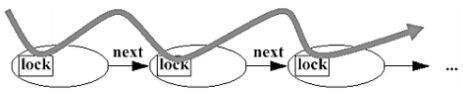
We use Foo.class to obtain the class object for Foo. Should you later define Bar, which subclasses Foo, a call to Bar.inc() will of course increment the same count variable as Foo.inc() (static variables are inherited by subclasses) and the lock from the Foo class will be locked, not the lock from Bar. This is, of course, what we want. If we had called getClass() instead of Foo.class, we would have locked the lock for Bar. That 67 would have been a mistake and we would have been using two different locks to protect the same static variable.

Synchronized vs Locks

1. There is no way to back off from an attempt to acquire a lock if it is already held, to give up after waiting for a specified time, or to cancel a lock attempt after an interrupt. This can make it difficult to recover from liveness problems.
2. There is no way to alter the semantics of a lock, for example with respect to reentrancy, read versus write protection, or fairness.
3. There is no access control for synchronization. Any method can perform synchronized(obj) for any accessible object, thus leading to potential denial-of-service problems caused by the holding of needed locks.
4. Synchronization within methods and blocks limits use to strict block-structured locking. For example, you cannot acquire a lock in one method and release it in another.

Locks

Back-offs try lock, if fails, try lock after some time. It generally relies on retries, it can livelock due to the delay between retries.
Reorderings Code requiring multiple locks can try one ordering, and if it fails, release all locks and try a different ordering.
Non-block-structured locking You can use a Mutex for hand-over-hand(lock coupling) locking across the nodes of a linked list during traversal.



General Read-Write Scenarios

- If there are already one or more active (executing) Readers, can a newly arriving Reader immediately join them even if there is also a waiting Writer? If so, a continuous stream of entering Readers will cause Writers to starve. If not, the throughput of Readers decreases.

- If both some Readers and some Writers are waiting for an active Writer to finish, should you bias the policy toward allowing Readers? A Writer? Earliest first? Random? Alternate? Similar choices are available after termination of Readers.
- Do you need a way to allow Writers to downgrade access to be Readers without having to give up locks?

Read-Write Locks

The idea behind read-write locks is that the readLock may be held simultaneously by multiple reader threads, so long as there are no writers. The writeLock is exclusive.
Read-Write locks are generally preferable to plain locks when:

- The methods in a class can be cleanly separated into those that only access (read) internally held data and those that modify (write).
- Reading is not permitted while writing methods are in progress. (If reads are permitted during writes, you may instead rely on unsynchronized read methods or copy-on-write updates.
- Target applications generally have more readers than writers.
- The methods are relatively time-consuming, so it pays to introduce a bit more overhead associated with read-write locks compared to simpler techniques in order to allow concurrency among reader threads.

The operation of RWlocks is as follows: The first reader that requests the lock will get it. Subsequent readers also get the lock, and all of them are allowed to read the data concurrently. When a writer requests the lock, it is put on a sleep queue until all the readers exit. A second writer will also be put on the writer's sleep queue. Should a new reader show up at this point, it will be put on the reader's sleep queue until all the writers have completed. Further writers will be placed on the same writer's sleep queue as the others (hence, in front of the waiting reader), meaning that writers are always favored over readers.

Performance Parameters

- Throughput - # of operations performed per unit time.
- Latency - Time elapsed between issuing a message and servicing it.
- Capacity - # of simultaneous activities that can be supported for a given target minimum throughput of maximum latency.
- Efficiency
- Scalability - Rate at which latency or throughput improves when resources are added to a system.
- Degradation - Just opposite of scalability

Thread

A thread is a call sequence that executes independently of others, while at the same time possibly sharing underlying system resources within the same program.

Just as multitasking operating systems can do more than one thing concurrently by running more than a single process, a process can do the same by running more than a single thread.
Each thread is a different stream of control that can execute its instructions independently, allowing a multithreaded process to perform numerous tasks concurrently.
A thread terminates when its run method completes by either returning normally or throwing an unchecked exception.
Threads are not restartable, even after they terminate. Invoking start more than once results in an InvalidThreadStateException.
One thread can not readily determine which other thread started it.
For communications, threads can use cheaper strategies like memory based synchronization facilities such as locks, waiting and notification mechanisms.
With multiple threads, a piece of code can be accessed by different activities in any order because all threads are independent of each other, order could be surprising - any order that is not explicitly ruled out is allowed. For instance, a field set to one value in one line may have a different value by the time the next line of code is executed because some other thread interleaved and got scheduled by the CPU.
Java threads are implemented in the JVM, which in turn is built on top of the native threads library for the specific platform. Java does not expose the native threads' APIs, only its own, very small set of functions.

The value of using Threads

- Performance gains from multiprocessing hardware (parallelism)
- Increased application throughput
- Increased application responsiveness
- Replacing process-to-process communications
- Efficient use of system resources
  Each process must maintain a complete process structure, including a full virtual memory space and kernel state. The cost of creating and maintaining this large amount of state makes each process much more expensive, in both time and space, than a thread.
- One binary that runs well on both uniprocessors and multiprocessors
- The ability to create well-structured programs.

In Java there is no thread exit function as there is in the other libraries, and the only way of exiting a thread is to return from the run() method. This seems a bit odd, but it is intentional. The basic idea is that only the run() method should make the decision to exit. Other methods lower in the call chain may decide that they are done with what they are doing, or they may encounter an error condition, but all they should do is pass that information up the call stack. Moreover, even the run() method shouldn't be exiting the thread explicitly because it doesn't "know" that it's running in a unique thread.

Why prefer Runnable interface

There are two reasons for using a runnable.
The first is that we are not changing the nature of the
thread itself, we're only changing the run() method, so subclassing Thread isn't really appropriate.

The second reason is that if we implement the Runnable interface, it's possible to subclass something else more useful.
Moreover, we can consider a Runnable to be the work to be done, while the thread is the engine
to do the work. From this point of view, it makes no sense to include the work inside the engine.

Garbage collecting Threads(need to check in new versions)

When do threads and thread objects get garbage collected? If you drop the last pointer to a thread, will it stop running and be garbage collected? No. When a thread is started, the thread object is entered into a thread group (see Thread Groups) and will remain there until it exits. The top-level thread group is one of the root GC nodes, so it never disappears.
As soon as a thread exits, its stack will be freed (this is an implementation detail), and some time after it exits and you drop the last pointer to the thread object, that thread object will be garbage collected. In other words, everything works the way you think it should and there's nothing to worry about.

Restarting Threads

Once a thread has exited, it is gone. The stack has been freed, the internal thread structures have been cleared, the underlying kernel resources have been returned. All that's left is the empty shell of the thread object. (If you extended the thread object and included your own instance variables, those will not be changed.) You cannot restart the thread; you cannot reuse the thread object.

How many threads in Java

The Java spec does not state how many threads an implementation must support. The actual number is completely implementation dependent. Presumably, the number will be the same as the limit on the underlying native library.

Join

t.join() returns when t.isAlive() is false, so it makes no sense to invoke join on a thread that has not been started.

Things to consider while designing a multithreaded solution

- Attributes
- State constraints(capacity can never be negative)
- Operations
- Connections to other objects and potential impact
- Pre/Post conditions
- Protocol - when and how operations are performed

Another set of 'how to think'

- **Allocating responsibility.** Which object has responsibility for executing the action? One of the participants? All of them? A separate coordinator?
- **Detecting conditions.** How can you tell when the participants are in the right state to perform the action? Do you ask them by invoking accessors? Do they tell you whenever they are in the right state? Do they tell you whenever they might be in the right state?
- **Programming actions.** How are actions in multiple objects arranged? Do they need to be atomic? What if one or more of them fails?
- **Linking conditions to actions.** How do you make sure that the actions occur only under the right conditions? Are false alarms acceptable? Do you need to prevent one or more participants from changing state between testing the condition and performing the action? Do the actions need to be performed when the participants enter the appropriate states, or merely whenever the conditions are noticed to hold? Do you need to prevent multiple objects from attempting to perform the action at the same time?

A system can be considered as a combination of objects and activities.
In object-centric view, a system is a collection of objects.
In an activity-centric view, a system is a collection of possible concurrent activities where each activity may involve many threads.
A given object may be involved in multiple activities and a given activity may involve multiple objects.

Cancellation

When activities in one thread fail or change course, it may be necessary or desirable to cancel activities in other threads, regardless of what they are doing. Cancellation requests introduce inherently unforeseeable failure conditions for running threads.
The asynchronous nature of cancellation leads to design tactics reminiscent of those in distributed systems where failures may occur at any time due to crashes and disconnections. Concurrent programs have the additional obligation to ensure consistent states of internal objects participating in other threads.

- **Interruption** (see interrupt section)
- **IO and resource revocation** A particular form of resource revocation. If one thread performs s.close() on an IO object (for example, an InputStream) s, then any other thread attempting to use s (for example, s.read()) will receive an IOException. Revocation affects all threads using the closed IO objects and causes the IO objects to be unusable. Most classes in java.io do not, and cannot, clean-fail upon IO exceptions. For example, if a low-level IO exception occurs in the midst of a StreamTokenizer or ObjectInputStream operation, there is no sensible recovery action that will preserve the intended guarantees. So, as a matter of policy, JVMs do not automatically interrupt IO operations.
- **Asynchronous cancellation (not in use widely)**
  The most obvious implementation for a truly asynchronous stop is to use UNIX signals or NT's equivalent. Signals in UNIX are indeed asynchronous, but not as immediate as you might imagine. The usual implementation of signals is for the caller to mark a bit in the target's process structure, and for the caller to look at that bit only when context switching on the system clock. Thus delivery of signals, hence stop notifications, occurs only at clock ticks.
- **Resource Control** You cannot control exactly what foreign code does or how long it does it. But you can and should apply standard security measures to limit undesirable effects. One approach is to create and use a SecurityManager and

related classes that deny all checked resource requests when a thread has run too long. This form of resource denial, in conjunction with resource revocation strategies can together prevent foreign code from taking any actions that might otherwise contend for resources with other threads that should continue.
- **Multiphase cancellation** Multiphase cancellation is a pattern seen at the process level in most operating systems. For example, it is used in Unix shutdowns, which first try to terminate tasks using kill -1, followed if necessary by kill -9.

Safety

Nothing bad happens to an object and all objects in a system maintain consistent states. An object is consistent if all fields obey their invariants. Each public method should lead an object from one consistent state to another. Safety failures lead to unintended behavior at run time - things just start going wrong.
Read/Write conflicts: T1 reads a field while T2 writes it. What T1 reads depends on which thread won the race.
Write/Write conflicts: Two threads try to write to the same field.

In a safe system, every object protects itself from integrity violations. We must ensure that all objects accessible from multiple threads are either immutable or employ appropriate synchronization, and that no object becomes concurrently accessible by leaking out its ownership. There are three basic strategies:

Immutability

If an object can not change state, then it can never encounter conflicts or inconsistencies. An immutable object can be shared freely without concern for synchronization or access restriction.

For added security, declare that the immutable class is final. The use of final means that when you think you have an immutable object, you really do — it's not of some subclass that supports
mutable operations as well.

Synchronization

Locking protects against low-level storage conflicts and corresponding high-level invariant failures. Every instance of class Object and its subclasses possesses a lock.
Synchronized keyword is not part of method signature, so a child class does not automatically inherit synchronized modifier.
Synchronized instance methods in subclasses employ the same lock as those in their superclasses.
Synchronization in an inner class method is independent of its outer class. However, a non-static inner class methods can lock its containing class as:
synchronized(OuterClass.this) {}

Also, interface methods and constructors can not be declared as synchronized.

Locking obeys a built-in acquire-release protocol controlled only by use of the synchronized keyword. Locks operate on a per-thread, not per-invocation basis.
A thread hitting synchronized passes if the lock is free or the thread already possesses the lock. Re-entrancy allows a synchronized method to make a self-call to another synchronized method on the same object without freezing up.
Synchronized is not equivalent to atomic, but can be used to achieve atomicity. It also has the side-effect of synchronizing the underlying memory system.
Locking an object does not automatically protect access to static fields of that object's class or any of its subclasses. Static synchronization employs the lock possessed by the Class object associated with the class the static methods are declared in. The static lock associated with each class is unrelated to that of any other class, including its subclasses. It is not effective to add a new static synchronized method in a subclass that attempts to protect static fields declared in a superclass.
The JVM internally obtains and releases the locks for Class objects during class loading and initialization, hence the class loading is thread safe.

What is a fully synchronized or atomic object?
All methods are synchronized.
There are no public fields.
All methods are finite so eventually release locks.
All fields are initialized to a consistent state in the constructor.
The state of the object is consistent after each method, even in the case of an exception.
A fully synchronized object is not sufficient in some cases:

```
for(int i=0; i<v.size(); i++) {
        SOP(v.get(i));
}
```

In this example, it may fail even if size and get are synchronized methods because the code is not atomic. It is possible for T1 to succeed in size check but T2 to change the size before T1 calls get(). There is a time gap between T1 releasing lock for size and attempting to obtain the lock again to call get.

All changes made in one synchronized method or block are atomic and visible with respect to other synchronized methods and blocks employing the same lock, and processing of synchronized methods or blocks within any given thread is in program-specific order even if the processing of statements within blocks may be out of order.

Reducing Synchronization

A set of synchronized operations might deadlock or present other liveness or lock-based performance problems if they were all waiting for the single synchronization lock associated with a single object. But they might be deadlock-free and/or run more efficiently if they are waiting on multiple distinct locks. As a general rule, the more finely you can subdivide the internal synchronization of a given class, the better will be its liveness properties across a wider range of contexts.

Two considerations enter into any decision about whether synchronization of an accessor method can be removed:
Legality. The value of the underlying field never assumes an illegal value; that is, you can ensure that the field never, even momentarily, breaks invariants.
Staleness. Clients do not necessarily require the most recently updated value of a field, but can live with possibly stale values

If a field is not always legal, then the choices are:
- Synchronize the accessor (as well as all update methods).
- Ensure somehow that clients realize when they have obtained an illegal value and take evasive action (for example via double-checks).
- Omit the accessor method. This applies surprisingly often. Ask yourself why any client would want to know the value of a field and what they could do with this value. Because object attributes can change asynchronously in concurrent programs, a value obtained by a client in one line of code may have changed before the next line of code is executed. Thus, accessor methods are not frequently useful in concurrent programs.

If a field is always legal but staleness is not acceptable, then you have the additional choice:
- Remove synchronization from the accessor, and qualify the instance variable as volatile.

If a field is always legal and staleness is acceptable, then you also have the choices:
- Remove synchronization from the accessor without any further alteration.
- If you like to live dangerously, just make the field public.

Optimistic Updates

Typically, each method takes the form:
- Get a copy of the current state representation (while holding a lock).
- Construct a new state representation (without holding any locks).
- Commit to the new state only if the old state has not changed since obtaining it.

Optimistic update techniques limit synchronization to very brief intervals — just long enough to access and later update state representations.

Confinement

Confinement ensures that the accessibility of a given object is unique to a single thread without needing to rely on dynamic locking on each access.

There are four categories to check to see if a reference r to an object x can escape from a method m executing within some activity:
- m passes r as an argument in a method invocation or object constructor.
- m passes r as the return value from a method invocation.
- m records r in some field that is accessible from another activity (in the most flagrant case, static fields that are accessible anywhere).
- m releases (in any of the above ways) another reference that can in turn be traversed to access r.

There are four sorts of confinement - **method (**If a given method invocation creates an object and does not let it escape, then it can be sure that no other threads will interfere with (or even know about) its use of that object.**)**, **thread (**threadl-ocal**)**, object confinement (using OO encapsulation techniques) and group confinement.

Exclusively held resources are analogs of physical objects in the sense that:
- If you have one, then you can do something (with it) that you couldn't do otherwise.
- If you have it, then no one else has it.
- If you give it to someone else, then you no longer have it.
- If you destroy it, then no one will ever have it.

For example, statements of the form x.r = y.s. do not cause the owner y, containing field s to lose possession after completion of the operation. The assignment instead results in both r and s still being bound.

Volatile

Declaring a field volatile affects atomicity, visibility and ordering surrounding their use.
Declaring a field as volatile differs only in that no locking is involved. In particular, composite read/write operations such as the "++" operation on volatile variables are not performed atomically.
Note - If volatile fields are accessed frequently inside methods, their use is likely to lead to slower performance than would locking the entire methods.
Declaring fields as volatile can be useful when you do not need locking for any other reason, yet values must be accurately accessible across multiple threads.

Liveness

Something actually happens in an activity. Liveness problems lead to nothing - things just stop running.
An activity may fail to make progress due to following possible reasons:
Locking, Waiting, Input(waits for input that has not yet arrived), CPU contention, Failure(error, fault)
Permanent failures could be - Deadlock, Missed signals(T1 started waiting after a notification to wake it up was produced already), Livelock(a continuously retired action continuously fails), Resource exhaustion, Starvation(JVM fails to allocate CPU time to a thread).

Deadlock

Deadlock is possible when two or more objects are mutually accessible from two or more threads, and each thread holds on lock while trying to obtain another lock already held by another thread.

Resource ordering can be used to avoid deadlock. Threads will obtain the locks in the same order. Ordering can be used whenever there is a need to arbitrarily break symmetry or force precedence in a concurrent design.

Priority

Java makes no promises about scheduling or fairness. By default, each new thread has the same priority as the thread that created it.
This is implementation dependent - vary across platforms. Some examples-

1. Some JVMs always select the highest current priority thread.
2. Some map the 10 Java thread priorities into a smaller system-supported categories, so threads with different priorities may be treated equally.

Interrupt

Each thread has an associated boolean interruption status.
t.interrupt() interrupts and sets t's status to true. If t is engaged in wait/sleep/join, interrupt causes InterruptedException and sets t's interrupt status to false.
Thread.interrupted() clears interruption status of the current thread and returns previous status.
isInterrupted() returns true if the thread has been interrupted via interrupt method but the status has not since been reset either by Thread.interrupt or wait/sleep/join throwing InterruptedException.

Interrupt is used by cancellation also. The best-supported techniques for approaching cancellation rely on per-thread interruption status that is set by method Thread.interrupt, inspected by Thread.isInterrupted, cleared (and inspected) by Thread.interrupted, and sometimes responded to by throwing InterruptedException.

Thread interrupts serve as requests that activities be canceled. Nothing stops anyone from using interrupts for other purposes, but this is the intended convention.
But there is nothing about an interrupt that forces immediate termination. This gives any interrupted thread a chance to clean up before dying, but also imposes obligations for code to check interruption status and take appropriate action on a timely basis. Lack of interruption may be used as a precondition checked at safe points before doing anything that would be difficult or impossible to undo later.

By convention, interruption status is cleared when an InterruptedException is thrown. When you need to propagate interruption status after handling an InterruptedException, you must either rethrow the exception or reset the status via Thread.currentThread().interrupt(). There are two situations in which threads remain dormant without being able to check interruption status or receive InterruptedException: blocking on synchronized locks and on IO. Threads do not respond to interrupts while waiting for a lock used in a synchronized method or block.
Basically, a call to interrupt() sets a flag and looks to see if the target thread is blocked. If it is blocked, it is forcibly awakened. When it sees the flag, it throws an exception. You may also test to see if a thread has been interrupted via Thread.interrupted() (for the current thread, this also resets the interrupted flag) or thread.isInterrupted() (for an arbitrary thread, this does not reset the flag). Once the flag is cleared, no method will throw an InterruptedException until interrupt() is called again. Catching the exception will also reset the flag.
In POSIX deferred cancellation and in Java interruption, a thread may run for an arbitrary amount of time after a cancellation (interruption) has been issued, thus allowing critical sections to execute without having to disable/enable cancellation. An interrupted thread could go off and run in a loop for hours before hitting an interruption point.

Handling Interrupts

- Disable interrupts
- Ignore interrupts
- Exit on interrupt
- Propage InterruptedException
- Re-interrupt

ThreadGroups

Each thread is a member of its parent's ThreadGroup. One purpose of groups is to enforce security policies like making it illegal to call interrupt() on threads which are not in your group.

Concurrency

Multithreading is a technique that allows one program to do multiple tasks concurrently.

A concurrent program is one that does more than one thing at a time. Different activities could be performed by different CPUs on some systems, and on others, it could be one CPU executing all activities.
Concurrency introduces following overheads:
Locks, Monitors, Threads, Context-switching, Scheduling, Locality(on multiprocessors, when multiple threads share an object, cache consistency h/w and low-level system s/w must communicate the values across all processors).

```
final class SetCheck {
        private int a = 0;
        private long b = 0;
        void set() {
                a = 1;
                b = -1;
        }
        boolean check() {
                return ((b == 0) || (b == -1 && a == 1));
        }
}
```

In a concurrent program, it's possible for check() to return false. Here are some possibilities:

- The compiler may rearrange the order of the statements, so b may be assigned before a.
- The processor may rearrange the execution order of machine instructions corresponding to the statements, or even execute them at the same time.
- The memory system (as governed by cache control units) may rearrange the order in which writes are committed to memory cells corresponding to the variables. These writes may overlap with other computations and memory actions.
- The compiler, processor, and/or memory system may interleave the machine-level effects of the two statements. For example on a 32-bit machine, the high-order word of b may be written first, followed by the write to a, followed by the write to the low-order word of b.

For example, as detailed below, check could read a value for the long b that is neither 0 nor -1, but instead a half-written in-between value. Also, out-of-order execution of the statements in set() may cause check() to read b as -1 but then read a as still 0.
Each thread has a working memory. Most rules are phrased in terms of when values must be transferred between the main memory and per-thread working memory. The rules address three
Issue:

Atomicity

Access and updates to the memory cells corresponding to fields of any type except long/double are guaranteed to be atomic. Additionally, atomicity extends to volatile long and double. Atomicity guarantees ensures that when a non long/double field is used in an expression, you will obtain either its initial value or some other value that was written by some other thread, but not some garbage value. Atomicity alone does not guarantee that you will get the most updated value due to visibility issues.

Visibility

Changes to fields made by one thread are guaranteed to be visible to other threads only under the following conditions:

1. A writing thread releases a synchronization lock and a reading thread subsequently acquires that same synchronization lock. In essence, releasing a lock forces a flush of all writes from working memory employed by the thread, and acquiring a lock forces a (re)load of the values of accessible fields.
   Note the double meaning of synchronized: it deals with locks that permit higher-level synchronization protocols, while at the same time dealing with the memory system (sometimes via low-level memory barrier machine instructions) to keep value representations in sync across threads.
2. If a field is declared as volatile, any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation (i.e., for the purposes at hand it is flushed immediately). Reader threads must reload the values of volatile fields upon each access.
3. The first time a thread accesses a field of an object, it sees either the initial value of the field or a value since written by some other thread.
   If you create and start a new thread T and then create an object X used by thread T, you cannot be sure that the fields of X will be visible to T unless you employ synchronization surrounding all references to object X. Or, when applicable, you can create X before starting T.
4. As a thread terminates, all written variables are flushed to main memory.

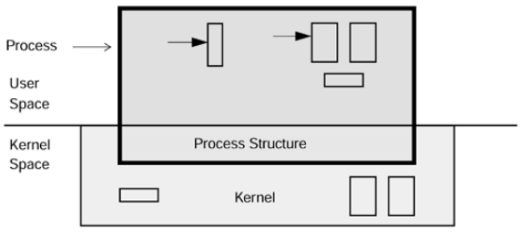
Note that visibility problems never arise when passing references to objects across methods in the same thread.
A particular update to a particular field made by one thread will eventually be visible to another. But eventually can be an arbitrarily long time. Long stretches of code in threads that use no synchronization can be hopelessly out of sync with other threads with respect to values of fields. In particular, it is always wrong to write loops waiting for values written by other threads unless the fields are volatile or accessed via synchronization
For example, it is possible to obtain a fresh value for one field of an object, but a stale value for another. Similarly, it is possible to read a fresh, updated value of a reference variable, but a stale value of one of the fields of the object now being referenced.

Critical Sections

A critical section is a section of code that must be allowed to complete atomically with no interruption that affects its completion. Other things may go on at the same time, and the thread that is executing in the critical section may even lose its processor, but no other thread may enter the critical section. Should another thread want to execute that same critical section, it will be forced to wait until the first thread finishes.

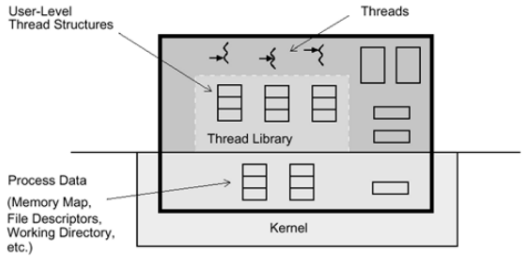
Traditional Operating System

Figure 2-2. Memory Layout for Multitasking Systems



There are actually two different modes of operation for the CPUs: user mode, which allows normal user programs to run, and kernel mode, which also allows some special instructions to run that only the kernel can execute. So a user program can execute only user-mode

instructions, and it can execute them only in user space. When it needs something from the kernel (say, it wants to read a file or find out the current time), the user program must make a system call. This is a library function that sets up some arguments, then executes a special trap instruction. This instruction causes the hardware to trap into the kernel, which then takes control of the machine.
Because the operating system has complete control over I/O, memory, processors, etc., it needs to maintain data for each process it's running. The data tells the operating system what the state of that process is. So, the concept of process in the multitasking world extends into the kernel, where this information is maintained in a process structure.
A thread is a lightweight entity, comprising the registers, stack, and some other data. The rest of the process structure is shared by all threads: the address space, file descriptors, etc. Much (and sometimes all) of the thread structure is in user space, allowing for very fast access. All threads in a process share the state of that process. They reside in exactly the same memory space, see the same functions, and see the same data.



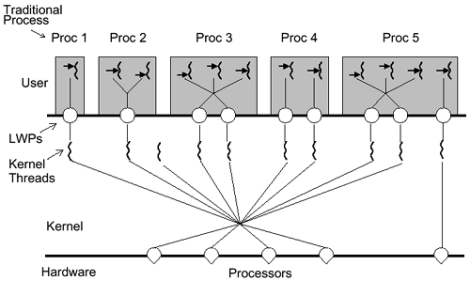
System Calls

A system call is basically a function that ends up trapping to routines in the kernel. For multithreaded programs, there is a serious issue surrounding how many threads can make system calls concurrently. For some operating systems, the answer is "one"; for others, it's "many."
When a process makes a system call, the following events occur:

1. The process traps to the kernel.
2. The trap handler runs in kernel mode and saves all the registers.
3. The handler sets the stack pointer to the process structure's kernel stack.
4. The kernel runs the system call.
5. The kernel places any requested data into the user-space structure that the programmer provided.
6. The kernel changes any process structure values affected.
7. The process returns to user mode, replacing the registers and stack pointer, and returns the appropriate value from the system call.

The POSIX Multithreaded Model

Threads are the portable application-level interface. The underlying threads library schedules the threads onto LWPs. The LWPs in turn are implemented by kernel threads in the kernel. These kernel threads are then scheduled onto the available CPUs by the standard kernel scheduling routine, completely invisible to the user. This picture is accurate for POSIX threads. It is equally applicable for Win32 and Java threads.



State Dependence

There are two general approaches to the design and implementation of any state-dependent action, that stem from liveness-first versus safety-first design perspectives:

- **Optimistic** try-and-see methods can always be tried when invoked, but do not always succeed, and thus may have to deal with failure.
  Try-and-see designs attempt actions without first ensuring that they will succeed, often because the constraints that would ensure success cannot be checked. However, optimistic methods always check postconditions (often by catching failure exceptions) and, if they fail to hold, apply a chosen failure policy. If a constraint is not under the control of the host object, then even if it is known to hold momentarily, it need not hold throughout the course of an action relying on it.

- **Conservative** check-and-act methods refuse to proceed unless preconditions hold. When preconditions do hold, the actions always succeed.

Optimistic approaches rely on the existence of exceptions and related mechanisms that indicate when postconditions do not hold. Conservative approaches rely on the availability of guard constructions that indicate when preconditions hold and guarantee that they continue to hold across the course of an action relying on them.

Thread Libraries

There are two fundamentally different ways of implementing threads -

1. The first is to write a user-level library that is substantially self-contained. It will make calls to system routines, and it may depend upon the existence of certain kernel features, but it is fundamentally a user-level library and contains no "magic" hooks into secret kernel routines.
2. The second way is to write a library that is inherently a kernel-level implementation. It may define all the same functions as in the first case, but these functions will be completely dependent upon the existence of kernel routines to support them and may well be almost entirely in kernel space.

Some implementations of the POSIX standard are of the first kind, while both OS/2 and Win32 threads are of the second type. When Java is implemented on these OSs it inherits the underlying behavior.

Process Structure

The only thing the kernel knows about is the process structure. In Solaris 2, the last couple bits have been abstracted out and placed into a new structure called a lightweight process (LWP). So a process contains all of the above, except for the runtime statistics, CPU state, and kernel stack, which are now part of the LWP structure. A process thus contains some number of LWPs (one for a "traditional" process, more for a multithreaded process). Just as the threads all share the process variables and state, the LWPs do the same.

LWPs

Each LWP is separately scheduled by the kernel and multiple LWPs in the same process can run in parallel on multiple processors. Because scheduling is done on a per-LWP basis, each LWP collects its own kernel statistics—user time, system time, page faults, etc. This also implies that a process with two LWPs will generally get twice as much CPU time as a process with only one LWP(a wild generalization).

Threads and LWPs

When it's time to context-switch two processes, the kernel saves the registers in the process structure, changes some virtual memory pointers, loads the CPU registers with data from the other process structure, and continues. When context-switching two threads, the registers are saved as before, but the memory map and the "current process" pointer remain the same.
What actually makes up a thread are its own stack and stack pointer; a program counter; some thread information, such as scheduling priority, and signal mask, stored in the thread structure; and the CPU registers (the stack pointer and program counter are actually just registers).
Everything else comes from either the process or (in a few cases) the LWP. The stack is just memory drawn from the program's heap. As threads share the same process structure, they also share most of the operating system state.

Different models of kernel scheduling

Many Threads on One LWP

The first technique is known as the many-to-one model. It is also known as co-routing. However, the thread creation, scheduling, and synchronization are all done 100% in user space, so they're done fast and cheap and use no kernel resources. This is how green threads work.

One Thread per LWP

The one-to-one model allocates one LWP for each thread. This model has the drawback that thread creation involves LWP creation; hence it requires a system call, as does scheduling and synchronization. In addition, each LWP takes up additional kernel resources, so you are limited in the total number of threads you can create.

Many Threads on many LWPs

The third model is the strict many-to-many model. Any number of threads are multiplexed onto some (smaller or equal) number of LWPs. Thread creation is done completely in user space, as are scheduling and synchronization (well, almost). Numerous threads can run in parallel on different CPUs, and a blocking system call need not block the entire process. As in the many-to-one model, the only limit on the number of threads is the size of virtual memory.

The two-level Model

The choice of the threading model is an implementation-level decision for writers of the JVM. Java itself has no concept of LWPs or threading models. This is a very reasonable choice by the Java designers; Java programs shouldn't be looking at this kind of low-level detail. Unfortunately, it brings in a very significant area of possible platform behavior difference.

Thread Scheduling

There are two basic levels to scheduling threads: process local scheduling (also known as process contention scope, or unbound threads—the many-to-many model) and system global scheduling (also known as system contention scope, or bound threads—the one-to-one model).
If an LWP is running on CPU 0 and it is context switched off for a short time, the vast majority of that cache will still be valid. So, it would be much better for that LWP to go back onto CPU 0. The normal schedulers in the various OSs endeavor to do precisely that via processor affinity.

Process contention scope scheduling means that all of the scheduling mechanism for the thread is local to the process—the threads library has full control over which thread will be scheduled on an LWP.
System contention scope scheduling means that the scheduling is done by the kernel (i.e., one-to-one binding).

Process Contention Scope

There are four ways to cause an active thread (say, T1) to context switch.

1. Synchronization
2. Preemption
3. Yielding
4. Time slicing

The LWPs that are to be used by the unbound threads are set up in a pool and are identical in all respects. Should you want a special LWP, you'd want a bound thread to run on it (not an option in Java). When an unbound thread exits or goes to sleep, and there are no more runnable threads, the LWP that was running the thread goes to sleep in the kernel. When another thread becomes runnable, the idling LWP wakes up and runs it. Should an LWP remain idle for an extended length of time (5 minutes for Solaris 2.5), the threads library may kill it.

Context Switching

It may be a bit unclear what the role of the LWP is when threads context switch. The role is invisible. The threads save and restore CPU registers with no regard to the LWP at all. The threads scheduler does not do anything to the LWP structure. Should the operating system decide to context switch the LWP, it will do so completely independently of what the LWP happens to be doing at that time. Should two threads be in the middle of context switching when the kernel decides to context switch the LWP, it still makes no difference.

Preemption

Preemption requires a system call, so the kernel has to send the signal, which takes time. Finally, the LWP, to which the signal is directed, must receive it and run the signal handler. Context switching by preemption is involuntary and is more expensive than context switching by "voluntary" means.

How to get LWPs in Java

First let's consider what we really want from our scheduler. We want all of our runnable threads to run as much as possible. We want to make as many blocking system calls as we feel like making, and we want them to execute concurrently.
One implementational technique for getting this effect is to use bound threads. Another is to ensure that the library creates a sufficient number of LWPs and guarantees that the runnable threads will be time sliced.
If you are running on a system that implements only PCS scheduling for Java threads (e.g., Solaris) there is no portable mechanism for specifying how many LWPs you'd like.
In Solaris, you are forced to make a native call to pthread_setconcurrency() to obtain the "expected" level of kernel concurrency. Obviously, this is not a good thing and makes a mess of your 100% pure Java program, but it is necessary for most high-performance MT programs

Dealing with Failure

There are six general responses to such failed actions:

1. **Abrupt termination** An extreme response to failure is to let a method die immediately, returning (usually via an exception) regardless of the state of the current object or status of the current activity. When objects are intrinsically shared across activities, and there is no way to re-establish consistent object states upon failure, and there is no possible (or practical) way to back out of a failing operation, then the only recourse is to set a broken or corrupted flag in the object encountering the failure and then abruptly terminate.
2. **Continuation** (ignoring failures)
3. **Rollback** The most desirable semantics in optimistic designs are clean-fail guarantees: Either the operation succeeds completely, or it fails in a way that leaves the object in exactly the same state as before the operation was attempted.
   There are two complementary styles for maintaining state representations that can be used in rollbacks:
   - Provisional action. Before attempting updates, construct a new representation that will, upon success, be swapped in as the current state. Methods perform updates on the tentative new version of the state representations, but do not commit to the new version until success is assured. This way, nothing needs to be undone upon failure.
   - Checkpointing. Before attempting updates, record the current state of the object in a history variable, perhaps in the form of a Memento (see the Design Patterns book). Methods directly perform updates on the current representation. But upon failure, fields can be reverted to the old values.
4. **Roll-forward**
5. **Retry** Without care, retries can consume unbounded amounts of CPU time. You can minimize the likelihood of repeated contention-based failures, as well as reduce CPU wastage, by inserting heuristic delays between attempts. One popular strategy (seen for example in Ethernet protocols) is exponential backoff, in which each delay is proportionally longer than the last one.
6. **Delegation to handlers**

Semaphores

Conceptually, a semaphore maintains a set of permits initialized in a constructor. A Semaphore initialized to 1 can be used directly as a lock, in which case "extra" releases are remembered and thus allow extra acquires. While this property is not at all desirable here, in contexts unrelated to locking it can be exploited as a cure for missed signals(how?).
A counting semaphore is a variable that can increment arbitrarily high but decrement only to zero.
A semaphore is useful for working with "trainlike" objects, that is, what you care about is whether there are either zero objects or more than zero. Buffers and lists that fill and empty are good examples. Semaphores are also useful when you want a thread to wait for something.

A semaphore is perfect for situations where you want to count things and have threads sleep when some limit is hit.
Bounded buffer using Semaphore

- Initially, for a buffer of size n, there are n put-permits and 0 take-permits.
- A take operation must acquire a take-permit and then release a put-permit.
- A put operation must acquire a put-permit and then release a take-permit.

Latch

Latches help structure solutions to initialization problems where you do not want a set of activities to proceed until all objects and threads have been completely constructed. For example, a more ambitious game-playing application might need to ensure that all players wait until the game officially begins.

Use-cases:

- **Completion indicators.** For example, to force a set of threads to wait until some other activity completes.
- **Timing thresholds.** For example, to trigger a set of threads at a certain date.
- **Event indications.** For example, to trigger processing that cannot occur until a certain packet is received or a button is clicked.
- **Error indications.** For example, to trigger a set of threads to proceed with global shut-down tasks.

Condition Variables

Monitor operations in the Java programming language maintain a single wait set for each object.
Some other languages and thread libraries (in particular POSIX pthreads) include support for multiple wait sets associated with multiple condition variables managed under a common object or lock.
A condition variable creates a safe environment for you to test your condition, sleep on it when false, and be awakened when it might have become true.
It works like this: A thread obtains a mutex (condition variables always have an associated mutex) and tests the condition under the mutex's protection. No other thread should alter any aspect of the condition without holding the mutex. If the condition is true, your thread completes its task, releasing the mutex when appropriate. If the condition isn't true, the mutex is released for you, and your thread goes to sleep on the condition variable. When some other thread changes some aspect of the condition (e.g., it reserves a plane ticket for granny), your thread then reacquires the mutex, reevaluates the condition, and either succeeds or goes back to sleep, depending upon the outcome.

Whereas in POSIX mutexes and condition variables exist as separate data types that must be associated together by the programmer, in Java they are tightly integrated with each object. As we mentioned, every Java object has associated with it a mutex, and additionally every Java object has associated with it a condition variable. The class Object defines the methods wait(), notify(), and notifyAll() to manipulate the condition variable associated with that object.
An object's condition variable is always associated with the object's mutex. Hence, before you can invoke wait() on the object, you must hold the mutex.
There are two disadvantages of wait sets vs. condition variables: With condition variables it is clear from the code what you're waking up, whereas notifyAll() will potentially wake up a lot of threads unnecessarily and waste a lot of time.

Why reevaluate the condition

First, the other thread may not have tested the complete condition before sending the wakeup. **Second**, even if the condition was true when the wakeup was sent, it could have changed before your thread got to run.
Third, condition variables allow for spurious wakeups.

Transaction Protocols

The most straightforward version of the basic protocol is:

1. For each participant p, if p cannot join, abort.
2. For each participant p, tentatively execute p's action.
3. For each participant p, if p cannot commit, abort.
4. For each participant p, commit p's effects of the transaction.

In the purest form of optimistic transactions, participants can always join, but cannot always commit. In the purest form of conservative transactions, participants cannot always join, but if they do, they can always commit. Optimistic approaches apply best when the likelihood of contention is low enough to outweigh rollback costs. Conservative approaches apply best when it is difficult or impossible to undo actions performed during transactions.

ThreadPool - Design choices

The first decision to make surrounding lightweight executable frameworks based on worker threads is whether to create or use them at all. The main question is whether there is some property of ordinary Threads that you do not need or are willing to give up. If not, it is unlikely that you will arrive at a solution that outperforms the built-in thread support on production JVM implementations.

Identity

Most worker threads must be treated "anonymously". Because the same worker thread is reused for multiple tasks, the use of ThreadLocal and other thread-specific contextual control techniques becomes more awkward.

Queuing

Runnable tasks that are sitting in queues do not run. This is one source of performance benefits in most worker-thread designs — if each action were associated with a thread, it would need to be independently scheduled by the JVM. But as a consequence, queued

execution cannot in general be used when there are any dependencies among tasks. If a currently running task blocks waiting for a condition produced by a task still waiting in the queue, the system may freeze up.

- Use as many worker threads as there are simultaneously executing tasks. In this case, the Channel need not perform any queuing, so you can use SynchronousChannels, queueless channels that require each put to wait for a take and vice versa.
- Create custom queues that understand the dependencies among the particular kinds of tasks being processed by the worker threads.

Saturation

As the request rate increases, a worker pool will eventually become saturated. All worker threads will be processing tasks and the Host object(s) using the pool will be unable to hand off work. Possible responses include:

- Increase the pool size.
- If the nature of the service allows it, use an unbounded buffered channel and let requests pile up. This risks potential system failure due to exhaustion of memory.
- Establish a back-pressure notification scheme to ask clients to stop sending so many requests.
- Drop (discard) new requests upon saturation. This can be a good option if you know that clients will retry anyway. However, unless retries are automatic, you need to add callbacks, events, or notifications back to clients to alert them of the drops so that they will know enough to retry.
- Make room for the new request by dropping old requests that have been queued but not yet run, or even canceling one or more executing tasks.
- Block until some thread is available. This can be a good option when handlers are of predictable, short-lived duration, so you can be confident that the wait will unblock without unacceptable delays.
- The Host can run the task directly itself, in its current thread. This is often the best default choice. In essence, the Host momentarily becomes single-threaded. The act of servicing the request limits the rate at which it can accept new requests, thus preventing further local breakdowns.

Thread management

Lazy construction

Activate a new thread only when a request cannot be serviced immediately by an existing idle thread. Lazy construction allows users to provide large enough pool size limits to avoid underutilization problems occurring when fewer threads are running than a given computer can handle. This comes at the minor expense of occasionally higher latencies when a new request causes a new thread to be created. The start-up effects of lazy construction can be tempered by creating a small number of "warm" threads upon construction of the pool.

Idle time-outs

Allow threads to time out waiting for work and to terminate upon time-out. This eventually causes all workers to exit if the pool is not used for prolonged periods. When coupled with lazy construction, these dead threads will be replaced with new ones if the request rate later increases.

Cancellation

You may need to distinguish cancellation of a task from cancellation of the worker thread performing that task. One approach is:

- Upon interruption, allow the current worker thread to die, but replace it if necessary with a fresh worker thread if the work queue is not empty or when a new incoming task arrives.
- Provide a shutdown method in the worker thread class that causes existing workers to die and no additional workers to be created.

Additionally, you may need to trigger some kind of error handling if a Host thread is canceled during a task hand-off.

JVM, GC, and Performance

JVM

> According to the specification that defines the Java Virtual Machine (usually called the VM Spec), the JVM is a stack-based interpreted machine. This means that rather than having registers (like a physical hardware CPU), it uses an execution stack of partial results and performs calculations by operating on the top value (or values) of that stack.
>
> The basic behavior of the JVM interpreter can be thought of as essentially "a `switch` inside a `while` loop"—processing each opcode of the program independently of the last, using the evaluation stack to hold intermediate values.

> The JVM provides three primary areas to hold data:
>
> - The *evaluation stack*, which is local to a particular method
> - *Local variables* to temporarily store results (also local to methods)
> - The *object heap*, which is shared between methods and between threads

Instance method calls are normally turned into invokevirtual instructions, except when the static type of a receiver object is only known to be an interface type. In this case, the call is instead represented by an invokeinterface opcode. Finally, in the cases (e.g., private methods or superclass calls) where the exact method for dispatch is known at compile time, an invokespecial instruction is produced.
From Java 8 onward, invokedynamic has become a crucial part of the Java language and it is used to provide support for advanced language features like lambdas.

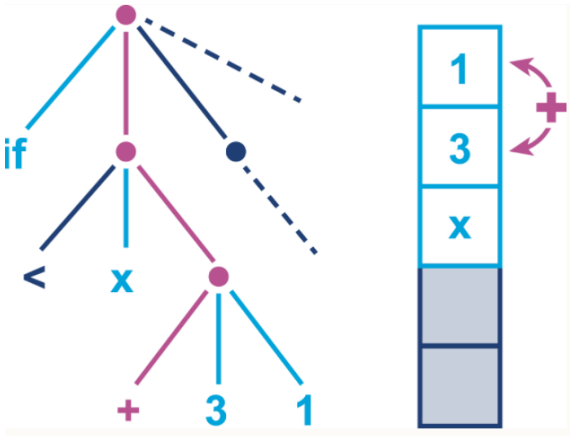


Figure 9-4. Subtree evaluation

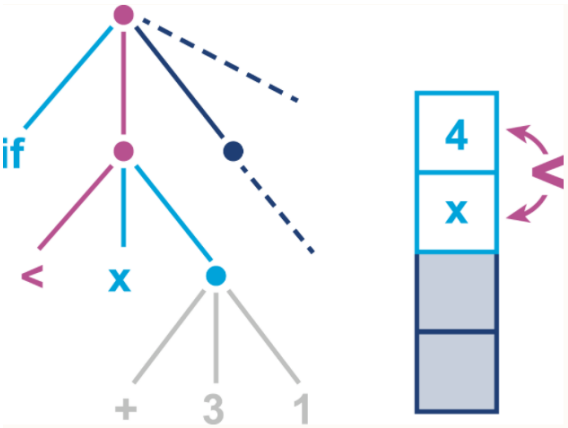


Figure 9-5. Final subtree evaluation

The entry point into the application will be the `main()` method of *HelloWorld.class*. In order to hand over control to this class, it must be loaded by the virtual machine before execution can begin.

To achieve this, the Java classloading mechanism is used. When a new Java process is initializing, a chain of classloaders is used. The initial loader is known as the Bootstrap classloader and contains classes in the core Java runtime. In versions of Java up to and including 8, these are loaded from *rt.jar*. In version 9 and later, the runtime has been modularised and the concepts of classloading are somewhat different.

The Extension classloader is created next; it defines its parent to be the Bootstrap classloader and will delegate to its parent if needed. Extensions are not widely used, but can supply overrides and native code for specific operating systems and platforms. Notably, the Nashorn JavaScript runtime introduced in Java 8 is loaded by the Extension loader.

Finally, the Application classloader is created; it is responsible for loading in user classes from the defined classpath. Some texts unfortunately refer to this as the "System" classloader. This term should be avoided, for the simple reason that it doesn't load the system classes (the Bootstrap classloader does). The Application classloader is encountered extremely frequently, and it has the Extension loader as its parent.

Java loads in dependencies on new classes when they are first encountered during the execution of the program. If a classloader fails to find a class, the behavior is usually to delegate the lookup to the parent. If the chain of lookups reaches the Bootstrap classloader and it isn't found, a `ClassNotFoundException` will be thrown. It is important that developers use a build process that effectively compiles with the exact same classpath that will be used in production, as this helps to mitigate this potential issue.

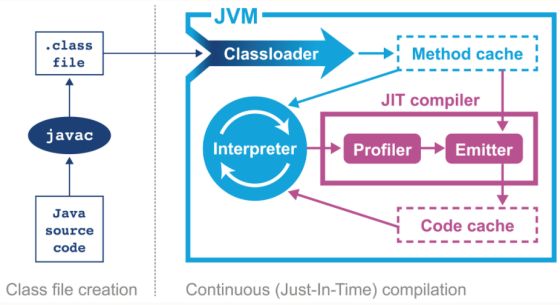


Figure 2-3. The HotSpot JVM

The JVM provides a portable execution environment that is independent of the operating system, by providing a common interface to Java code. However, for some fundamental services, such as thread scheduling (or even something as mundane as getting the time from the system clock), the underlying operating system must be accessed.

This capability is provided by native methods, which are denoted by the keyword native. They are written in C, but are accessible as ordinary Java methods. This interface is referred to as the Java Native Interface (JNI).

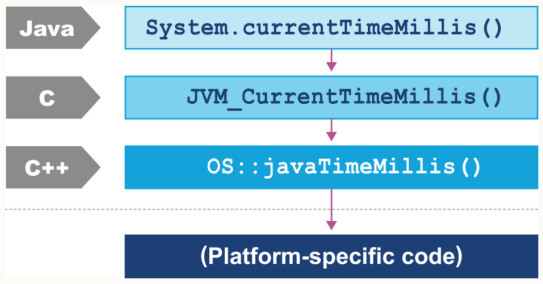


Figure 3-11. The HotSpot calling stack

JIT

Java programs begin their execution in the bytecode interpreter, where instructions are performed on a virtualized stack machine. This abstraction from the CPU gives the benefit of class file portability, but to get maximum performance your program must execute directly on the CPU, making use of its native features.

HotSpot achieves this by compiling units of your program from interpreted bytecode into native code. The units of compilation in the HotSpot VM are the method and the loop. This is known as *Just-in-Time* (JIT) compilation.

JIT compilation works by monitoring the application while it is running in interpreted mode and observing the parts of code that are most frequently executed. During this analysis process, programmatic trace information is captured that allows for more sophisticated optimization. Once execution of a particular method passes a threshold, the profiler will look to compile and optimize that particular section of code.

The basic unit of compilation in HotSpot is a whole method. HotSpot also supports the compilation of a hot loop using a technique called on-stack replacement (OSR).

OSR is used to help the case where a method is not called frequently enough to be compiled but contains a loop that would be eligible for compilation if the loop body was a method in its own right.
The overall picture is that when compilation is indicated, the method is placed on a compiler thread, which compiles in the background. When the optimized machine code is available, the entry in the vtable of the relevant klass is updated to point at the new compiled code.

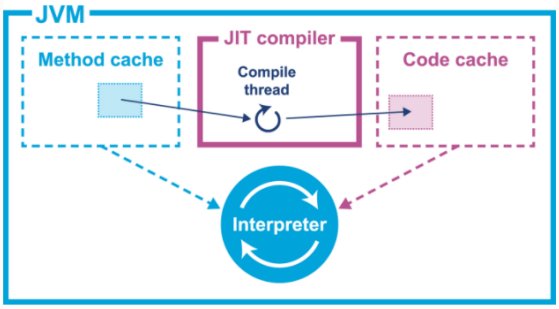


Figure 9-6. Simple compilation of a single method

Logging JIT Compilation

One important JVM switch that all performance engineers should know about is:

```
-XX:+PrintCompilation
```

This will cause a log of compilation events to be produced on `STDOUT` and will allow the engineer to get a basic understanding of what is being compiled.

Compilers Within HotSpot

The HotSpot JVM actually has not one, but two JIT compilers in it. These are properly known as C1 and C2, but are sometimes referred to as the client compiler and the server compiler, respectively. Historically, C1 was used for GUI apps and other "client" programs, whereas C2 was used for long-running "server" applications. Modern Java apps generally blur this distinction.

The general approach that both compilers take is to rely on a key measurement to trigger compilation: the number of times a method is invoked, or the invocation count. Once this counter hits a certain threshold the VM is notified and will consider queuing the method for compilation.
The compilation process proceeds by first creating an internal representation of the method. Next, optimizations are applied that take into account profiling information that has been collected during the interpreted phase. However, the internal representation of the code that C1 and C2 produce is quite different. C1 is designed to be simpler and have shorter compile times than C2. The tradeoff is that as a result C1 does not optimize as fully as C2.
One technique that is common to both is single static assignment. This essentially converts the program to a form where no reassignment of variables occurs. In Java programming terms, the program is effectively rewritten to contain only final variables.

Tiered Compilation in HotSpot

- Level 0: interpreter
- Level 1: C1 with full optimization (no profiling)
- Level 2: C1 with invocation and backedge counters
- Level 3: C1 with full profiling
- Level 4: C2

We can also see in Table 9-6 that not every level is utilized by each compilation approach.

Table 9-6. Compilation pathways

Pathway	Description
0-3-4	Interpreter, C1 with full profiling, C2
0-2-3-4	Interpreter, C2 busy so quick-compile C1, then full-compile C1, then C2
0-3-1	Trivial method
0-4	No tiered compilation (straight to C2)

In the case of the trivial method, the method starts off interpreted as usual but then C1 (with full profiling) is able to determine the method to be trivial. This means that it is clear that the C2 compiler would produce no better code than C2 and so compilation terminates.

The Code Cache

JIT-compiled code is stored in a memory region called the code cache. This area also stores other native code belonging to the VM itself, such as parts of the interpreter.

The code cache has a fixed maximum size that is set at VM startup. It cannot expand past this limit, so it is possible for it to fill up. At this point no further JIT compilations are possible and uncompiled code will execute only in the interpreter. This will have an impact on performance and may result in the application being significantly less performant than the potential maximum.

The code cache is implemented as a heap containing an unallocated region and a linked list of freed blocks. Each time native code is removed, its block is added to the free list. A process called the sweeper is responsible for recycling blocks.

When a new native method is to be stored, the free list is searched for a block large enough to store the compiled code. If none is found, then subject to the code cache having sufficient free space, a new block will be created from the unallocated space.

Native code can be removed from the code cache when:

- It is deoptimized (an assumption underpinning a speculative optimization turned out to be false).
- It is replaced with another compiled version (in the case of tiered compilation).
- The class containing the method is unloaded.

You can control the maximum size of the code cache using the VM switch:

```
-XX:ReservedCodeCacheSize=<n>
```

Note that with tiered compilation enabled, more methods will reach the lower compilation thresholds of the C1 client compiler. To account for this, the default maximum size is larger to hold these additional compiled methods.

In Java 8 on Linux x86-64 the default maximum sizes for the code cache are:

```
251658240 (240MB) when tiered compilation is enabled (-XX:+TieredComp
 50331648  (48MB) when tiered compilation is disabled (-XX:-TieredCon
```

Tuning

1. First run the app with the `PrintCompilation` switch on.
2. Collect the logs that indicate which methods are compiled.
3. Now increase the size of the code cache via `ReservedCodeCacheSize`.
4. Rerun the application.
5. Look at the set of compiled methods with the enlarged cache.

The performance engineer will need to take into account the slight nondeterminism inherent in the JIT compilation. Keeping this in mind, there are a couple of obvious tells that can easily be observed:

- Is the set of compiled methods larger in a meaningful way when the cache size is increased?
- Are all methods that are important to the primary transaction paths being compiled?

If the number of compiled methods does not increase (indicating that the code cache is not being fully utilized) as the cache size is increased, then provided the load pattern is representative, the JIT compiler is not short of resources.

At this point, it should be straightforward to confirm that all the methods that are part of the transaction hot paths appear in the compilation logs. If not, then the next step is to determine the root cause—why these methods are not compiling.

Compilation

Under the hood, HotSpot stores profile data about the running program in structures called method data objects, or MDOs for short.
MDOs are used by the bytecode interpreter and the C1 compiler to record the information the JIT compilers use when determining what optimizations to make. MDOs store information such as methods invoked, branches taken, and types observed at call sites. Counters that record the "hotness" of a profiled property are maintained, and the values in them are decayed during profiling. This ensures that methods are compiled only if they are still hot when they reach the head of the compilation queue.
Once this profiling data has been gathered and the decision has been made to compile, the specific details of the individual compiler take over. The compiler builds an internal representation of the code to be compiled; the exact nature of the representation depends on which compiler (C1 or C2) is in use.

HotSpot's JIT compilers are able to perform a wide range of modern compiler optimization techniques, including:

- **Inlining(or gateway optimization)**
  HotSpot considers several factors when determining whether to inline a method, including:
  The bytecode size of the method to be inlined
  The depth of the method to be inlined in the current call chain
  The amount of space in the code cache already taken up by compiled versions of this method.

  Table 10-1. Inlining switches

Switch	Default (JDK 8, Linux x86_64)	Explanation
`-XX:MaxInlineSize=<n>`	35 bytes of bytecode	Inline methods up to this size.
`-XX:FreqInlineSize=<n>`	325 bytes of bytecode	Inline "hot" (frequently called) methods up to this size.
`-XX:InlineSmallCode=<n>`	1,000 bytes of native code (non-tiered) 2,000 bytes of native code (tiered)	Do not inline methods where there is already a final-tier compilation that occupies more than this amount of space in the code cache.
`-XX:MaxInlineLevel=<n>`	9	Do not inline call frames deeper than this level.

- **Loop unrolling**
  Each back branch can have undesirable processor effects, as the CPU dumps its pipeline of incoming instructions due to the back branch. In general, the shorter the loop body, the higher the relative cost of the back branch.

HotSpot contains a number of specific optimizations for loop unrolling:

- It can optimize counted loops that use an `int`, `short`, or `char` loop counter.
- It can unroll loop bodies and remove safepoint polls.
- Unrolling a loop reduces the number of back branches and their associated branch prediction costs.
- Removing safepoint polls further reduces the work done each loop iteration.

- **Escape analysis**

HotSpot can perform a scope-based analysis for determining if work done within a method is visible or has side effects outside the boundary of that method.

By proving that an allocated object does not escape the method (classed as a `NoEscape`), the VM can apply an optimization called *scalar replacement*. The fields in the object become scalar values, similar to if they had all been local variables instead of object fields. They can then be arranged into CPU registers by a HotSpot component called the *register allocator*.

NOTE

If there are not enough free registers available, then the scalar values can be placed onto the current stack frame (this is known as a *stack spill*).

The aim of escape analysis is to deduce whether the heap allocation can be avoided. If it can be, the object can be automatically allocated on the stack and thus GC pressure can be reduced by a small amount.

Limitations of Escape Analysis

Escape analysis, like other optimizations, is subject to tradeoffs since every allocation not made on the heap must happen somewhere, and the CPU registers and stack space are relatively scarce resources. One limitation in HotSpot is that by default, arrays of more than 64 elements will not benefit from escape analysis. This size is controlled by the following VM switch:

```
-XX:EliminateAllocationArraySizeLimit=<n>
```

If an object is found to escape the method scope on any branch, then the optimization to avoid allocating the object on the heap will not be applied. In the following example, if both branches are seen to be taken then the object can sometimes escape the methods and must be classed as an `ArgEscape`. This will increase the object allocation rate and put additional pressure on the GC.

```
for (int i = 0; i < 100_000_000; i++) {
    Object mightEscape = new Object(i);

    if (condition) {
        result += inlineableMethod(mightEscape);
    } else {
        result += tooBigToInline(mightEscape);
    }
}
```

If it is possible in your code to localize the object allocation to within a nonescaping branch, as shown here, then you will benefit from escape analysis on that path.

```
for (int i = 0; i < 100_000_000; i++) {

    if (condition) {
        Object mightEscape = new Object(i);
        result += inlineableMethod(mightEscape);
    } else {
        Object mightEscape = new Object(i);
        result += tooBigToInline(mightEscape);
    }
}
```

- **Lock elision and lock coarsening**

Locks and Escape Analysis

HotSpot is able to use escape analysis and some related techniques to optimize the performance of locks.

NOTE

This applies only to intrinsic locks (those that use `synchronized`). The locks from `java.util.concurrent` are not eligible for these optimizations.

The key lock optimizations that are available are:

- Removing locks on nonescaping objects (*lock elision*)
- Merging consecutive locked regions that share the same lock (*lock coarsening*)
- Detecting blocks where the same lock is acquired repeatedly without an unlock (*nested locks*)

When consecutive locks on the same object are encountered, HotSpot will check if it is possible to enlarge the locked region. To achieve this, when HotSpot encounters a lock it will search backward to try to find an unlock on the same object. If a match is found, then it will consider whether the two lock regions can be joined to produce a single larger region.

- **Monomorphic dispatch**
- **Intrinsics**

An intrinsic is the name given to a highly tuned native implementation of a method that is preknown to the JVM rather than generated dynamically by the JIT subsystem.

Table 10-3. Example intrinsified methods

Method	Description
`java.lang.System.arraycopy()`	Faster copying using vector support on the CPU.
`java.lang.System.currentTimeMillis()`	Fast implementations provided by most OSs.
`java.lang.Math.min()`	Can be performed without branching on some CPUs.
Other `java.lang.Math` methods	Direct instruction support on some CPUs.
Cryptographic functions (e.g., AES)	Hardware acceleration can give significant performance improvements.

- **On-stack replacement**

Sometimes you will encounter code that contains a hot loop within a method that is not called enough times to trigger a compilation—for example, in a Java program's `main()` method.

HotSpot can still optimize this code using a technique called *on-stack replacement* (OSR). This trick counts the loop back branches in the interpreter; when these cross a threshold, the interpreted loop will be compiled and execution will switch to this compiled version.

A back branch happens when the loop reaches the end of its body, checks its exit condition, and—if the loop has not finished—branches back to the loop body start.

Safepoints Revisited

Before we leave the topic of JIT compilation, it makes sense to bring together a list of all the conditions in the JVM that require the VM to be at a safepoint. As well as GC STW events, the following activities require all threads to be at a safepoint:

- Deoptimizing a method
- Creating a heap dump
- Revoking a biased lock
- Redefining a class (e.g., for instrumentation)

In compiled code, the JIT compiler is responsible for emitting safepoint checks (as seen previously with loop unrolling) and in HotSpot it will generate them:

- At Loop back branches
- On Method return

You can see the total time a program has spent at safepoints, including waiting for all threads to reach the safepoints, by using the VM switch `-XX:+PrintGCApplicationStoppedTime`. Combine this with `-XX:+PrintSafepointStatistics` for futher information on safepoints.

Comparing AOT and JIT Compilation

AOT compilation has the advantage of being relatively simple to understand. Machine code is produced directly from the source, and the machine code that corresponds to a compilation unit is directly available as assembly.
Targeting processor-specific features during AOT compilation will produce an executable that is compatible only with that processor. This can be a useful technique for low-latency or extreme performance use cases; building on the exact same hardware as the application will run on ensures that the compiler can take advantage of all available processor optimizations. However, this technique does not scale: if you want maximum performance across a range of target architectures, you will need to produce a separate executable for each one.
By contrast, HotSpot can add optimizations for new processor features as they are released and applications will not have to recompile their classes and JARs to take advantage of them. It is not unusual to find that program performance improves measurably between new releases of the HotSpot VM as the JIT compiler is improved.

Class-File Structure

Figure 2-2. Mnemonic for class file structure

Threads

It is safe to assume that every JVM application thread is backed by a unique OS thread that is created when the `start()` method is called on the corresponding `Thread` object.

Monitoring

Different ways/tools to monitor JVM:
- ➜ JMX
- ➜ Java Agents
- ➜ The JVM Tool Interface(JVMTI)
- ➜ The Serviceability Agent(SA)

Safe Points

The runtime requires each application thread to have special execution points, called safepoints, where the thread's internal data structures are in a known-good state. At these times, the thread is able to be suspended for coordinated actions.

There are two primary rules that govern the JVM's approach to safepointing:
- ➜ The JVM cannot force a thread into the safepoint state.
- ➜ The JVM can prevent a thread from leaving the safepoint state.

This means that the implementation of the JVM interpreter must contain code to yield at a barrier if safepointing is required.

The general case for reaching safepoints then looks like this:
- ➜ The JVM sets a global "time to safepoint" flag.
- ➜ Individual application threads poll and see that the flag has been set.
- ➜ They pause and wait to be woken up again.

When this flag is set, all app threads must stop. Threads that stop quickly must wait for slower stoppers (and this time may not be fully accounted for in the pause time statistics). Normal app threads use this polling mechanism. They will always check in between executing any two bytecodes in the interpreter. In compiled code, the most common cases where the JIT compiler has inserted a poll for safepoints are exiting a compiled method and when a loop branches backward (e.g., to the top of the loop).

A thread is automatically at a safepoint if it:
- ➜ Is blocked on a monitor
- ➜ Is executing JNI code

A thread is not necessarily at a safepoint if it:
- ➜ Is partway through executing a bytecode (interpreted mode)
- ➜ Has been interrupted by the OS

Hardware in General

Modern CPUs have several layers of cache, with the most-often-accessed caches being located close to the processing core. The cache closest to the CPU is usually called L1 (for "level 1 cache"), with the next being referred to as L2, and so on. Different processor architectures have a varying number and configuration of caches, **but a common choice is for each execution core to have a dedicated, private L1 and L2 cache, and an L3 cache that is shared across some or all of the cores.**

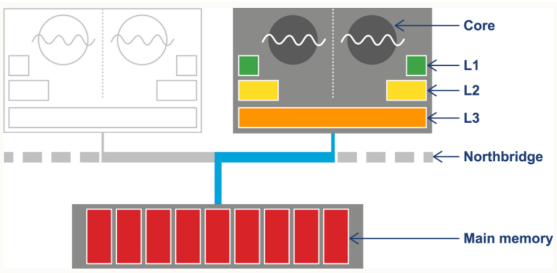


Figure 3-3. Overall CPU and memory architecture

Although the addition of a caching architecture hugely improves processor throughput, it introduces a new set of problems. These problems include determining how memory is fetched into and written back from the cache. The solutions to this problem are usually referred to as cache consistency protocols.
At the lowest level, a protocol called **MESI** (and its variants) is commonly found on a wide range of processors. It defines four states for any line in a cache. Each line (usually 64 bytes) is either:
- ➜ Modified (but not yet flushed to main memory)
- ➜ Exclusive (present only in this cache, but does match main memory)
- ➜ Shared (may also be present in other caches; matches main memory)
- ➜ Invalid (may not be used; will be dropped as soon as practical)

The idea of the protocol is that multiple processors can simultaneously be in the Shared state. However, if a processor transitions to any of the other valid states (Exclusive or Modified), then this will force all the other processors into the Invalid state.

Modern Processor Features

Translation Lookaside Buffer

One very important use is in a different sort of cache, the Translation Lookaside Buffer (TLB). This acts as a cache for the page tables that map virtual memory addresses to physical addresses, which greatly speeds up a very frequent operation—access to the physical address underlying a virtual address.

Branch Prediction and Speculative Execution

In this model a conditional branch is problematic, because until the condition is evaluated, it isn't known what the next instruction after the branch will be. This can cause the processor to stall for a number of cycles.
To avoid this, the processor can dedicate transistors to building up a heuristic to decide which branch is more likely to be taken. Using this guess, the CPU fills the pipeline based on a gamble. If it works, then the CPU carries on as though nothing had happened. If it's wrong, then the partially executed instructions are dumped, and the CPU has to pay the penalty of emptying the pipeline.

Memory Models

For example, let's suppose we have a piece of code like this:

```
myInt = otherInt;
intChanged = true;
```

There is no code between the two assignments, so the executing thread doesn't need to care about what order they happen in, and thus the environment is at liberty to change the order of instructions.

However, this could mean that in another thread that has visibility of these data items, the order could change, and the value of `myInt` read by the other thread could be the old value, despite `intChanged` being seen to be `true`.

Operating System

Scheduler

An often-overlooked feature of operating systems is that by their nature, they introduce periods of time when code is not running on the CPU. A process that has completed its time quantum will not get back on the CPU until it arrives at the front of the run queue again. Combined with the fact that CPU is a scarce resource, this means that code is waiting more often than it is running.

To put it another way, if we observe that the CPU utilization is not approaching 100% user time, then the next obvious question is, "Why not?" What is causing the program to fail to achieve that? Are involuntary context switches caused by locks the problem? Is it due to blocking caused by I/O contention?

Garbage Collection

Java classes have an internal representation within Java Hotspot VM and are referred to as class metadata.

In previous releases of Java Hotspot VM, the class metadata was allocated in the so-called permanent generation. Starting with JDK 8, the permanent generation was removed and the class metadata is allocated in native memory. The amount of native memory that can be used for class metadata is by default unlimited. Use the option `-XX:MaxMetaspaceSize` to put an upper limit on the amount of native memory used for class metadata.

Class metadata is deallocated when the corresponding Java class is unloaded. Java classes are unloaded as a result of garbage collection, and garbage collections may be induced to unload classes and deallocate class metadata. When the space committed for class metadata reaches a certain level (a high-water mark), a garbage collection is induced. After the garbage collection, the high-water mark may be raised or lowered depending on the amount of space freed from class metadata. The high-water mark would be raised so as not to induce another garbage collection too soon. The high-water mark is initially set to the value of the command-line option `-XX:MetaspaceSize`. It is raised or lowered based on the options `-XX:MaxMetaspaceFreeRatio` and `-XX:MinMetaspaceFreeRatio`. If the committed space available for class metadata as a percentage of the total committed space for class metadata is greater than `-XX:MaxMetaspaceFreeRatio`, then the high-water mark will be lowered. If it's less than `-XX:MinMetaspaceFreeRatio`, then the high-water mark will be raised.

Specify a higher value for the option `-XX:MetaspaceSize` to avoid early garbage collections induced for class metadata.

Tuning Strategy

The heap grows or shrinks to a size that supports the chosen throughput goal. Learn about heap tuning strategies such as choosing a maximum heap size, and choosing maximum pause-time goal.

Don't choose a maximum value for the heap unless you know that you need a heap greater than the default maximum heap size. Choose a throughput goal that's sufficient for your application.
A change in the application's behavior can cause the heap to grow or shrink. For example, if the application starts allocating at a higher rate, then the heap grows to maintain the same throughput.
If the heap grows to its maximum size and the throughput goal isn't being met, then the maximum heap size is too small for the throughput goal. Set the maximum heap size to a value that's close to the total physical memory on the platform, but doesn't cause swapping of the application. Execute the application again. If the throughput goal still isn't met, then the goal for the application time is too high for the available memory on the platform.
If the throughput goal can be met, but pauses are too long, then select a maximum pause-time goal. Choosing a maximum pause-time goal may mean that your throughput goal won't be met, so choose values that are an acceptable compromise for the application.
It's typical that the size of the heap oscillates as the garbage collector tries to satisfy competing goals. This is true even if the application has reached a steady state. The pressure to achieve a throughput goal (which may require a larger heap) competes with the goals for a maximum pause-time and a minimum footprint (which both may require a small heap).

The following are general guidelines regarding heap sizes for server applications:

- Unless you have problems with pauses, try granting as much memory as possible to the virtual machine. The default size is often too small.
- Setting -Xms and -Xmx to the same value increases predictability by removing the most important sizing decision from the virtual machine. However, the virtual machine is then unable to compensate if you make a poor choice.
- In general, increase the memory as you increase the number of processors, because allocation can be made parallel.

If survivor spaces are too small, then the copying collection overflows directly into the old generation. If survivor spaces are too large, then they are uselessly empty. At each garbage collection, the virtual machine chooses a threshold number, which is the number of times an object can be copied before it's old. This threshold is chosen to keep the survivors half full.

Keep the old generation large enough to hold all the live data used by the application at any given time, plus some amount of slack space (10 to 20% or more).

Selecting a Collector

Unless your application has rather strict pause-time requirements, first run your application and allow the VM to select a collector.
If necessary, adjust the heap size to improve performance. If the performance still doesn't meet your goals, then use the following guidelines as a starting point for selecting a collector:

- If the application has a small data set (up to approximately 100 MB), then select the serial collector with the option -XX:+UseSerialGC.
- If the application will be run on a single processor and there are no pause-time requirements, then select the serial collector with the option -XX:+UseSerialGC.
- If (a) peak application performance is the first priority and (b) there are no pause-time requirements or pauses of one second or longer are acceptable, then let the VM select the collector or select the parallel collector with -XX:+UseParallelGC.
- If response time is more important than overall throughput and garbage collection pauses must be kept shorter than approximately one second, then select a mostly concurrent collector with -XX:+UseG1GC or -XX:+UseConcMarkSweepGC.
- If response time is a high priority, and/or you are using a very large heap, then select a fully concurrent collector with -XX:UseZGC. ZGC is intended for applications which require low latency (less than 10 ms pauses) and/or use a very large heap (multi-terabytes). You can enable is by using the -XX:+UseZGC option.

In the HotSpot JVM (by far the most commonly used JVM) **memory is allocated at startup and managed from within user space.** That means that system calls (such as sbrk()) are not needed to allocate memory. In turn, this means that kernel-switching activity for garbage collection is quite minimal.

Thus, if a system is exhibiting high levels of **system CPU usage**, then it is definitely not spending a significant amount of its time in GC, as GC activity burns user space CPU cycles and does not impact kernel space utilization.

On the other hand, if a JVM process is using 100% (or close to that) of **CPU in user space**, then garbage collection is often the culprit. When analyzing a performance problem, if simple tools (such as vmstat) show consistent 100% CPU usage, but with almost all cycles being consumed by user space, then we should ask, "Is it the JVM or user code that is responsible for this utilization?" In almost all cases, high user space utilization by the JVM is caused by the GC subsystem, so a useful rule of thumb is to check the GC log and see how often new entries are being added to it.

Table 5-1. Support level of various GC algorithms[a]

GC algorithm	Support in JDK 8	Support in JDK 11	Support in JDK 12
Serial GC	S	S	S
Throughput (Parallel) GC	S	S	S
G1 GC	S	S	S
Concurrent Mark-Sweep (CMS)	S	D	D
ZGC	-	E	E
Shenandoah	E2	E2	E2
Epsilon GC	-	E	E

Resizing the metaspace requires a full GC, so it is an expensive operation. If there are a lot of full GCs during the startup of a program (as it is loading classes), it is often because permgen or metaspace is being resized, so increasing the initial size is a good idea to improve startup in that case. Servers, for example, typically specify an initial metaspace size of 128 MB, 192 MB, or more. This log also prints out the metaspace sizes, but those will never change during a young collection.

Out-of-Memory Errors

The JVM throws an *out-of-memory* error under these circumstances:

- No native memory is available for the JVM.
- The metaspace is out of memory.
- The Java heap itself is out of memory: the application cannot create any additional objects for the given heap size.
- The JVM is spending too much time performing GC.

The last two cases—involving the Java heap itself—are more common, but don't automatically conclude from an out-of-memory error that the heap is the problem. It is necessary to look at why the out-of-memory error occurred (that reason is part of the output of the exception).

Object representation

HotSpot represents Java objects at runtime via a structure called an oop. This is short for ordinary object pointer(one that represents instances of a Java class are called instanceOops), and is a genuine pointer in the C sense. These pointers can be placed in local variables of reference type where they point from the stack frame of the Java method into the memory area comprising the Java heap.
The memory layout of an instanceOop starts with two machine words of header present on every object. The mark word is the first of these, and is a pointer that points at instance-specific metadata. Next is the klass word, which points at class-wide metadata.

In Java 7 and before, the klass word of an instanceOop points into an area of memory called PermGen, which was a part of the Java heap.
From Java 8 onward, the klasses are held outside of the main part of the Java heap (but not outside the C heap of the JVM process).

Fundamentally the klassOop contains the virtual function table (vtable) for the class, whereas the Class object contains an array of references to Method objects for use in reflective invocation.

Compressed oops

Using simple programming, 64-bit JVMs are slower than 32-bit JVMs. This performance gap is because of the 64-bit object references: the 64-bit references take up twice the space (8 bytes) in the heap as 32-bit references (4 bytes). That leads to more GC cycles, since there is now less room in the heap for other data.

The JVM can compensate for that additional memory by using compressed oops. Oops stands for ordinary object pointers, which are the handles the JVM uses as object references. When oops are only 32 bits long, they can reference only 4 GB of memory (232), which is why a 32-bit JVM is limited to a 4 GB heap size. (The same restriction applies at the operating system level, which is why any 32-bit process is limited to 4 GB of address space.) When oops are 64 bits long, they can reference exabytes of memory, or far more than you could ever actually get into a machine.
There is a middle ground here: what if there were 35-bit oops? Then the pointer could reference 32 GB of memory (235) and still take up less space in the heap than 64-bit references. The problem is that there aren't 35-bit registers in which to store such references. Instead, though, the JVM can assume that the last 3 bits of the reference are all 0. Now every reference can be stored in 32 bits in the heap. When the reference is stored into a 64-bit register, the JVM can shift it left by 3 bits (adding three zeros at the end). When the reference is saved from a register, the JVM can right-shift it by 3 bits, discarding the zeros at the end.
This leaves the JVM with pointers that can reference 32 GB of memory while using only 32 bits in the heap. However, it also means that the JVM cannot access any object at an address that isn't divisible by 8, since any address from a compressed oop ends with three zeros. The first possible oop is 0x1, which when shifted becomes 0x8. The next oop is 0x2, which when shifted becomes 0x10 (16). Objects must therefore be located on an 8-byte boundary.
It turns out that objects are already aligned on an 8-byte boundary in the JVM; this is the optimal alignment for most processors. So nothing is lost by using compressed oops. If the first object in the JVM is stored at location 0 and occupies 57 bytes, the next object will be stored at location 64—wasting 7 bytes that cannot be allocated. That memory trade-off is worthwhile (and will occur whether compressed oops are used or not), because the object can be accessed faster given that 8-byte alignment.
But that is the reason that the JVM doesn't try to emulate a 36-bit reference that could access 64 GB of memory. In that case, objects would have to be aligned on a 16-byte boundary, and the savings from storing the compressed pointer in the heap would be outweighed by the amount of memory that would be wasted between the memory-aligned objects.
This has two implications. First, for heaps that are between 4 GB and 32 GB, use compressed oops. Compressed oops are enabled using the -XX:+UseCompressedOops flag; they are enabled by default whenever the maximum heap size is less than 32 GB. (In "Reducing Object Size", it was noted that the size of an object reference on a 64-bit JVM with a 32 GB heap is 4 bytes—which is the default case since compressed oops are enabled by default.)
Second, a program that uses a 31 GB heap and compressed oops will usually be faster than a program that uses a 33 GB heap. Although the 33 GB heap is larger, the extra space used by the pointers in that heap means that the larger heap will perform more-frequent GC cycles and have worse performance.
Hence, it is better to use heaps that are less than 32 GB, or heaps that are at least a few GB larger than 32 GB. Once extra memory is added to the heap to make up for the space used by the uncompressed references, the number of GC cycles will be reduced. No hard rule indicates the amount of memory needed before the GC impact of the uncompressed oops is ameliorated—but given that 20% of an average heap might be used for object references, planning on at least 38 GB is a good start.

Soft Reference

Soft references are used when the object in question has a good chance of being reused in the future, but you want to let the garbage collector reclaim the object if it hasn't been used recently (a calculation that also takes into consideration the amount of memory the heap has available). Soft references are essentially one large, least recently used (LRU) object pool. The key to getting good performance from them is to make sure that they are cleared on a timely basis.

```
long ms = SoftRefLRUPolicyMSPerMB * AmountOfFreeMemoryInMB;
if (now - last_access_to_reference > ms)
   free the reference
```

This code has two key values. The first value is set by the `-XX:SoftRefLRUPolicyMSPerMB= N` flag, which has a default value of 1,000.

The second value is the amount of free memory in the heap (once the GC cycle has completed). The free memory in the heap is calculated based on the maximum possible size of the heap minus whatever is in use.

So how does that all work? Take the example of a JVM using a 4 GB heap. After a full GC (or a concurrent cycle), the heap might be 50% occupied; the free heap is therefore 2 GB. The default value of `SoftRefLRUPolicyMSPerMB` (1,000) means that any soft reference that has not been used for the past 2,048 seconds (2,048,000 ms) will be cleared: the free heap is 2,048 (in megabytes), which is multiplied by 1,000:

```
long ms = 2048000; // 1000 * 2048
if (System.currentTimeMillis() - last_access_to_reference_in_ms > ms
    free the reference
```

If the 4 GB heap is 75% occupied, objects not accessed in the last 1,024 seconds are reclaimed, and so on.

Weak References

Weak references should be used when the referent in question will be used by several threads simultaneously. Otherwise, the weak reference is too likely to be reclaimed by the garbage collector: objects that are only weakly referenced are reclaimed at every GC cycle.

The interesting effect here, though, is where the weak reference ends up in the heap. Reference objects are just like other Java objects: they are created in the young generation and eventually promoted to the old generation. If the referent of the weak reference is freed while the weak reference itself is still in the young generation, the weak reference will be freed quickly (at the next minor GC). (This assumes that the reference queue is quickly processed for the object in question.) If the referent remains around long enough for the weak reference to be promoted into the old generation, the weak reference will not be freed until the next concurrent or full GC cycle.

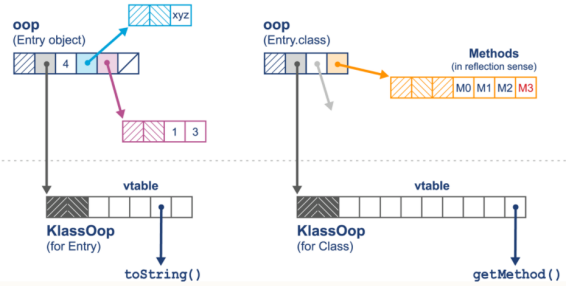


Figure 6-2. klassOops and Class objects

This means that, in general, a HotSpot object header consists of:

- ➔ Mark word at full native size
- ➔ Klass word (possibly compressed)
- ➔ Length word if the object is an array—always 32 bits
- ➔ A 32-bit gap (if required by alignment rules)

The managed environment of the JVM does not allow a Java reference to point anywhere but at an instanceOop (or null). This means that at a low level:

- ➔ A Java value is a bit pattern corresponding either to a primitive value or to the address of an instanceOop (a reference).
- ➔ Any Java reference considered as a pointer refers to an address in the main part of the Java heap.
- ➔ Addresses that are the targets of Java references contain a mark word followed by a klass word as the next machine word.
- ➔ A klassOop and an instance of Class<?> are different (as the former lives in the metadata area of the heap), and a klassOop cannot be placed into a Java variable.

Why avoid finalize()

Once the object has been registered as needing finalization, instead of being immediately reclaimed during the garbage collection cycle, the object follows this extended lifecycle:

1. Finalizable objects are moved to a queue.
2. After application threads restart, separate finalization threads drain the queue and run the `finalize()` method on each object.
3. Once the `finalize()` method terminates, the object will be ready for actual collection in the next cycle.

Overall, this means that all objects to be finalized must first be recognized as unreachable via a GC mark, then finalized, and then GC must run again in order for the data to be collected. This means that finalizable objects persist for one extra GC cycle at least. In the case of objects that have become Tenured, this can be a significant amount of time. The finalization queue processing can be seen in

GC Roots and Arenas

Some examples of the GC roots:

- ➔ Stack frames
- ➔ JNI
- ➔ Registers (for the hoisted variable case)
- ➔ Code roots (from the JVM code cache)
- ➔ Globals
- ➔ Class metadata from loaded classes

The HotSpot garbage collector works in terms of areas of memory called **arenas**.
HotSpot does not use system calls to manage the Java heap. Instead, HotSpot manages the heap size from user space code.

Weak Generational Hypothesis

One key part of the JVM's memory management relies upon an observed runtime effect of software systems, the Weak Generational Hypothesis:

> *The distribution of object lifetimes on the JVM and similar software systems is bimodal—with the vast majority of objects being very short-lived and a secondary population having a much longer life expectancy.*

This hypothesis, which is really an experimentally observed rule of thumb about the behavior of object-oriented workloads, leads to an obvious conclusion: garbage-collected heaps should be structured in such a way as to allow short-lived objects to be easily and quickly collected, and ideally for long-lived objects to be separated from short-lived objects.

HotSpot uses several mechanisms to try to take advantage of the Weak Generational Hypothesis:

- It tracks the "generational count" of each object (the number of garbage collections that the object has survived so far).
- With the exception of large objects, it creates new objects in the "Eden" space (also called the "Nursery") and expects to move surviving objects.
- It maintains a separate area of memory (the "old" or "Tenured" generation) to hold objects that are deemed to have survived long enough that they are likely to be long-lived.

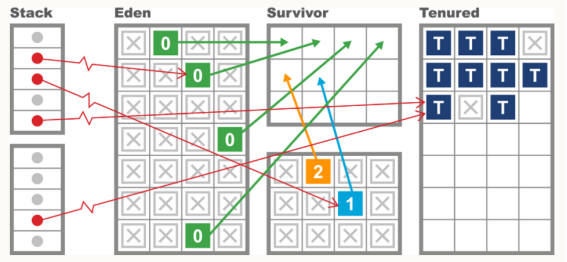


Figure 6-4. Generational collection

The main concerns that developers often need to consider when choosing a garbage collector include:

- Pause time (aka pause length or duration)
- Throughput (as a percentage of GC time to application runtime)
- Pause frequency (how often the collector needs to stop the application)
- Reclamation efficiency (how much garbage can be collected on a single GC duty cycle)
- Pause consistency (are all pauses roughly the same length?)

Thread-Local Allocation

For efficiency, the JVM partitions Eden into buffers and hands out individual regions of Eden for application threads to use as allocation regions for new objects. The advantage of this approach is that each thread knows that it does not have to consider the possibility that other threads are allocating within that buffer. These regions are called thread-local allocation buffers (TLABs).
The exclusive control that an application thread has over its TLABs means that allocation is O(1) for JVM threads. This is because when a thread creates a new object, storage is allocated for the object, and the thread-local pointer is updated to the next free memory address.
If the object will not fit in an empty TLAB, the VM will next try to allocate the object directly in Eden, in an area outside of any TLAB. If this fails, the next step is to perform a young GC (which might resize the heap). Finally, if this fails and there's still not enough space, the last resort is to allocate the object directly in Tenured.
From this we can see that the only objects that are really likely to end up being directly allocated in Tenured are large arrays (especially byte and char arrays).

Hemispheric Collection(Survivor spaces)

The central idea is to use the spaces as a temporary holding area for objects that are not actually long-lived. This prevents short-lived objects from cluttering up the Tenured generation and reduces the frequency of full GCs. The spaces have a couple of basic properties:

- ➔ When the collector is collecting the currently live hemisphere, objects are moved in a compacting fashion to the other hemisphere and the collected hemisphere is emptied for reuse.
- ➔ One half of the space is kept completely empty at all times.

The Parallel Collectors

These are fully STW for both young and full collections, and they are optimized for throughput. After stopping all application threads the parallel collectors use all available CPU cores to collect memory as quickly as possible. The available parallel collectors are:

- ➔ Parallel GC
  The simplest collector for the young generation
- ➔ ParNew
  A slight variation of Parallel GC that is used with the CMS collector
- ➔ ParallelOld
  The parallel collector for the old (aka Tenured) generation

The parallel collectors are in some ways similar to each other—they are designed to use multiple threads to identify live objects as quickly as possible and to do minimal bookkeeping.
Because multiple garbage collector threads are participating in a minor collection, some fragmentation is possible due to promotions from the young generation to the old generation during the collection. Each garbage collection thread involved in a minor collection reserves a part of the old generation for promotions and the division of the available space into these "promotion buffers" can cause a fragmentation effect. Reducing the number of garbage collector threads and increasing the size of the old generation will reduce this fragmentation effect.

Priority of Parallel Collector Goals
The maximum pause-time goal is met first. Only after it's met is the throughput goal addressed. Similarly, only after the first two goals have been met is the footprint goal considered.

Growing and shrinking the size of a generation is done by increments that are a fixed percentage of the size of the generation so that a generation steps up or down toward its desired size. Growing and shrinking are done at different rates. By default, a generation grows in increments of 20% and shrinks in increments of 5%. The percentage for growing is controlled by the command-line option -XX:YoungGenerationSizeIncrement=<Y> for the young generation and -XX:TenuredGenerationSizeIncrement=<T> for the old generation.
The percentage by which a generation shrinks is adjusted by the command-line flag -XX:AdaptiveSizeDecrementScaleFactor=<D>. If the growth increment is X%, then the decrement for shrinking is X/D%.
If the collector decides to grow a generation at startup, then there's a supplemental percentage is added to the increment. This supplement decays with the number of collections and has no long-term effect. The intent of the supplement is to increase startup performance. There isn't supplement to the percentage for shrinking.

Young Parallel Collections

This usually occurs when a thread tries to allocate an object into Eden but doesn't have enough space in its TLAB, and the JVM can't allocate a fresh TLAB for the thread. When this occurs, the JVM has no choice other than to stop all the application threads—because if one thread is unable to allocate, then very soon every thread will be unable.
Once all application (or user) threads are stopped, HotSpot looks at the young generation (which is defined as Eden and the currently nonempty survivor space) and identifies all objects that are not garbage. This will utilize the GC roots (and the card table to identify GC roots coming from the old generation) as starting points for a parallel marking scan.

The Parallel GC collector then evacuates all of the surviving objects into the currently empty survivor space (and increments their generational count as they are relocated). Finally, Eden and the just-evacuated survivor space are marked as empty, reusable space and the application threads are started so that the process of handing out TLABs to application threads can begin again.
This design has two performance advantages. First, because the young generation is only a portion of the entire heap, processing it is faster than processing the entire heap. The application threads are stopped for a much shorter period of time than if the entire heap were processed at once. You probably see a trade-off there, since it also means that the application threads are stopped more frequently than they would be if the JVM waited to perform GC until the entire heap was full; that trade-off will be explored in more detail later in this chapter. For now, though, it is almost always a big advantage to have the shorter pauses even though they will be more frequent.
The second advantage arises from the way objects are allocated in the young generation. Objects are allocated in eden (which encompasses the vast majority of the young generation). When the young generation is cleared during a collection, all objects in eden are either moved or discarded: objects that are not in use can be discarded, and objects in use are moved either to one of the survivor spaces or to the old generation. Since all surviving objects are moved, the young generation is automatically compacted when it is collected: at the end of the collection, eden and one of the survivor spaces are empty, and the objects that remain in the young generation are compacted within the other survivor space.

Old Parallel Collections

Parallel GC(young parallel collection) is a hemispheric evacuating collector, whereas ParallelOld is a compacting collector with only a single contiguous memory space. Thus, the collector can potentially be very efficient in its use of memory, and will not suffer from memory fragmentation.

The purpose of the young collections is to deal with the short-lived objects, so the occupancy of the young space is changing radically with allocation and clearance at GC events.
By contrast, the old space does not change as obviously. Occasional large objects will be created directly in Tenured directly, but apart from that, the space will only change at collections—either by objects being promoted from the young generation, or by a full rescan and rearrangement at an old or full collection.

Limitations of parallel collections

- ➔ They are fully stop-the-world.
- ➔ The design of the young parallel collectors is such that dead objects are never touched, so the length of the marking phase is proportional to the (small) number of surviving objects. This basic design, coupled with the usually small size of the young regions of the heap, means that the pause time of young collections is very short for most workloads.
- ➔ The old generation is generally by default seven times the size of the young generation. Old objects may be long-lived, so a potentially larger number of old objects may survive a full collection.
  This behavior also explains a key weakness of parallel old collection—the STW time will scale roughly linearly with the size of the heap. As heap sizes continue to increase, ParallelOld starts to scale badly in terms of pause time.

Tri-color Marking

The algorithm works like this:
- ➔ GC roots are colored gray.
- ➔ All other objects are colored white.
- ➔ A marking thread moves to a random gray node.
- ➔ If the node has any white children, the marking thread first colors them gray, then colors the node black.
- ➔ This process is repeated until there are no gray nodes left.
- ➔ All black objects have been proven to be reachable and must remain alive.
- ➔ White nodes are eligible for collection and correspond to objects that are no longer reachable.

Condition - No black object node may hold a reference to a white object node during concurrent marking.

Concurrent collection also frequently makes use of a technique called **snapshot at the beginning (SATB)**. This means that the collector regards objects as live if they were reachable at the start of the collection cycle or have been allocated since.
Consider the situation where an object has already been colored black by a marking thread, and then is updated to point at a white object by a mutator thread. It would be a problem because GC would clean that white object but it actually became reachable. It happens because application threads run concurrently with GC marking threads.
This problem can be solved in several different ways. We could, for example, change the color of the black object back to gray, adding it back to the set of nodes that need processing as the mutator thread processes the update.
An alternative approach would be to keep a queue of all changes that could potentially violate the invariant, and then have a secondary “fixup” phase that runs after the main phase has finished.

CMS

The Concurrent Mark and Sweep (CMS) collector is designed to be an extremely low-pause collector for the Tenured (aka old generation) space only. It is usually paired with a slightly modified parallel collector for collecting the young generation (called ParNew rather than Parallel GC).

The CMS collector is activated with this flag:

```
-XX:+UseConcMarkSweepGC
```

On modern versions of HotSpot, this flag will activate `ParNewGC` (a slight variation of the parallel young collector) as well.

Phases

- Initial Mark (STW)
- Concurrent Mark
- Concurrent Preclean
- Remark (STW)
- Concurrent Sweep
- Concurrent Reset

The observable effects of using CMS are as follows, for most workloads:
- Application threads don't stop for as long.
- A single full GC cycle takes longer (in wallclock time, that's why two STW events).
- Application throughput is reduced while a CMS GC cycle is running.
- GC uses more memory for keeping track of objects.
- Considerably more CPU time is needed overall to perform GC.
- CMS does not compact the heap, so Tenured can become fragmented.

By default, CMS will use half of the available threads to perform the concurrent phases of GC, and leave the other half for application threads to execute Java code.
While CMS is running, the young GC run will usually take longer than in the case of the parallel collectors, because it only has half the cores available for young generation GC (the other half of the cores are running CMS). At the end of this young collection, some objects will usually be eligible for promotion to Tenured. These promoted objects need to be moved into Tenured while CMS is still running, which requires some coordination between the two collectors. This is why CMS requires a slightly different young collector.

CMS has two separate STW phases:

Initial Mark
: Marks the immediate interior nodes—those that are directly pointed at by GC roots

* Remark
: Uses the card tables to identify objects that may require fixup work

This means that all app threads must stop, and hence safepoint, twice per CMS cycle. This effect can become important for some low-latency apps that are sensitive to safepointing behavior.

Concurrent Mode Failure(CMF)

Under normal circumstances, the young collection promotes only a small amount of objects to Tenured, and the CMS old collection completes normally, freeing up space in Tenured. However, the case of very high allocation or premature promotion can cause a situation in which the young collection has too many objects to promote for the available space in Tenured. This is known as a concurrent mode failure (CMF). It can also happen in case of heap fragmentation because CMS does not compact Tenured as it runs.
In the case of CMF, JVM has no choice at this point but to fall back to a collection using ParallelOld, which is fully STW. To avoid this, by default, a 75% heap occupancy ratio is used as a trigger to launch CMS.
To avoid CMF due to heap fragmentation, internally, CMS uses a free list of chunks of memory to manage the available free space. During the final phase, Concurrent Sweep, contiguous free blocks will be coalesced by the sweeper threads. This is to provide larger blocks of free space and try to avoid CMFs caused by fragmentation.
However, the sweeper runs concurrently with mutators. Thus, unless the sweeper and the allocator threads synchronize properly, a freshly allocated block might get incorrectly swept up. To prevent this, the sweeper locks the free lists while it is sweeping.

G1

G1 is a generational, incremental, parallel, mostly concurrent, stop-the-world, and evacuating garbage collector which monitors pause-time goals in each of the stop-the-world pauses.
G1 reclaims space mostly by using evacuation: live objects found within selected memory areas to collect are copied into new memory areas, compacting them in the process.
The Garbage-First collector is not a real-time collector. It tries to meet set pause-time targets with high probability over a longer time, but not always with absolute certainty for a given pause.

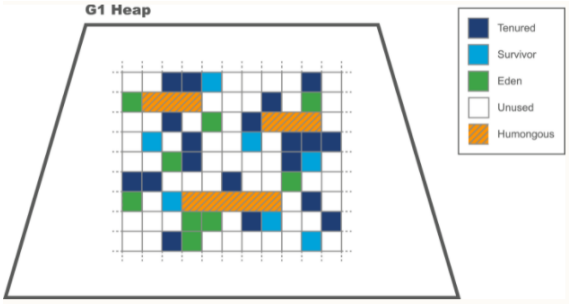


Figure 7-5. G1 regions

G1's algorithm allows regions of either 1, 2, 4, 8, 16, 32, or 64 MB. By default, it expects between 2,048 and 4,095 regions in the heap and will adjust the region size to achieve this.

To calculate the region size, we compute this value:

```
<Heap size> / 2048
```

Objects that occupy more space than half a region size are considered humongous objects. They are directly allocated in special *humongous regions*, which are free, contiguous regions that are immediately made part of the Tenured generation (rather than Eden).

G1 GC operates on discrete regions within the heap. Each region (there are by default around 2,048) can belong to either the old or new generation, and the generational regions need not be contiguous. The idea behind having regions in the old generation is that when the concurrent background threads look for unreferenced objects, some regions will contain more garbage than other regions. The actual collection of a region still requires that application threads be stopped, but G1 GC can focus on the regions that are mostly garbage and spend only a little bit of time emptying those regions. This approach—clearing out only the mostly garbage regions—is what gives G1 GC its name: garbage first.

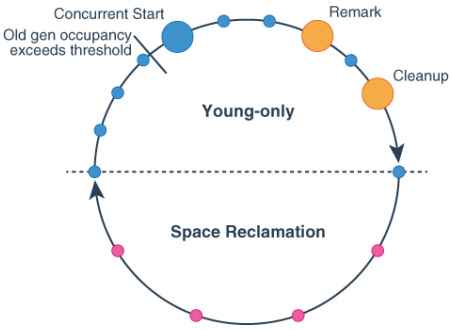
That doesn't apply to the regions in the young generation: during a young GC, the entire young generation is either freed or promoted (to a survivor space or to the old generation).

G1 garbage collection pauses can reclaim space in the young generation as a whole, and any additional set of old generation regions at any collection pause. During the pause G1 copies objects from this *collection set* to one or more different regions in the heap. The destination region for an object depends on the source region of that object: the entire young generation is copied into either survivor or old regions, and objects from old regions to other, different old regions using aging.

Regions

These are areas which are by default 1 MB in size.

Algorithm



The following list describes the phases, their pauses and the transition between the phases of the G1 garbage collection cycle in detail:

1. Young-only phase: This phase starts with a few Normal young collections that promote objects into the old generation. The transition between the young-only phase and the space-reclamation phase starts when the old generation occupancy reaches a certain threshold, the Initiating Heap Occupancy threshold. At this time, G1 schedules a Concurrent Start young collection instead of a Normal young collection.
   - Concurrent Start : This type of collection starts the marking process in addition to performing a Normal young collection. Concurrent marking determines all currently reachable (live) objects in the old generation regions to be kept for the following space-reclamation phase. While collection marking hasn't completely finished, Normal young collections may occur. Marking finishes with two special stop-the-world pauses: Remark and Cleanup.
   - Remark: This pause finalizes the marking itself, performs global reference processing and class unloading, reclaims completely empty regions and cleans up internal data structures. Between Remark and Cleanup G1 calculates information to later be able to reclaim free space in selected old generation regions concurrently, which will be finalized in the Cleanup pause.
   - Cleanup: This pause determines whether a space-reclamation phase will actually follow. If a space-reclamation phase follows, the young-only phase completes with a single Prepare Mixed young collection.
2. Space-reclamation phase: This phase consists of multiple Mixed collections that in addition to young generation regions, also evacuate live objects of sets of old generation regions. The space-reclamation phase ends when G1 determines that evacuating more old generation regions wouldn't yield enough free space worth the effort.

After space-reclamation, the collection cycle restarts with another young-only phase. As backup, if the application runs out of memory while gathering liveness information, G1 performs an in-place stop-the-world full heap compaction (Full GC) like other collectors.

The high-level picture of the collector is that G1:

- Uses a concurrent marking phase
- Is an evacuating collector
- Provides "statistical compaction"

The Remembered Sets (usually just called RSets) are per-region entries that track outside references that point into a heap region. This means that instead of tracing through the entire heap for references that point into a region, G1 just needs to examine the RSets and then scan those regions for references.

The G1 GC young collection is triggered when eden fills up (in this case, after filling four regions). After the collection, eden is empty (though regions are assigned to it, which will begin to fill up with data as the application proceeds). At least one region is assigned to the

survivor space (partially filled in this example), and some data has moved into the old generation.
In the general case, some objects from the survivor space might have been moved to the old generation, and if the survivor space were full, some objects from eden would have been promoted directly to the old generation—in those cases, the size of the old generation would increase.

G1 Phases

G1 has a collection of phases that are somewhat similar to the phases we've already met, especially in CMS:

1. Initial Mark (STW)
2. Concurrent Root Scan
3. Concurrent Mark
4. Remark (STW)
5. Cleanup (STW)

The Concurrent Root Scan is a concurrent phase that scans survivor regions of the Initial Mark for references to the old generation. This phase must complete before the next young GC can start. In the Remark phase, the marking cycle is completed. This phase also performs reference processing (including weak and soft references) and handles cleanup relating to implementing the SATB approach.

Cleanup is mostly STW, and comprises accounting and RSet "scrubbing." The accounting task identifies regions that are now completely free and ready to be reused (e.g., as Eden regions).

Basic JVM Flags for G1

The switch that you need to enable G1 (in Java 8 and before) is:

```
+XX:UseG1GC
```

```
-XX:MaxGCPauseMillis=200
```

This means that the default pause time goal is 200 ms. In practice, pause times under 100 ms are very hard to achieve reliably, and such goals may not be met by the collector. One other option that may also be of use is the option of changing the region size, overriding the default algorithm:

```
-XX:G1HeapRegionSize=<n>
```

Note that `<n>` must be a power of 2, between 1 and 64, and indicates a value in MB. We will meet other G1 flags when we discuss tuning G1 in Chapter 8.

Humongous Objects

Humongous objects are objects larger or equal the size of half a region. The current region size is determined ergonomically as described in the Ergonomic Defaults for G1 GC section, unless set using the `-XX:G1HeapRegionSize` option.

These humongous objects are sometimes treated in special ways:

- Every humongous object gets allocated as a sequence of contiguous regions in the old generation. The start of the object itself is always located at the start of the first region in that sequence. Any leftover space in the last region of the sequence will be lost for allocation until the entire object is reclaimed.
- Generally, humongous objects can be reclaimed only at the end of marking during the Cleanup pause, or during Full GC if they became unreachable. There is, however, a special provision for humongous objects for arrays of primitive types for example, `bool`, all kinds of integers, and floating point values. G1 opportunistically tries to reclaim humongous objects if they are not referenced by many objects at any kind of garbage collection pause. This behavior is enabled by default but you can disable it with the option `-XX:G1EagerReclaimHumongousObjects`.
- Allocations of humongous objects may cause garbage collection pauses to occur prematurely. G1 checks the Initiating Heap Occupancy threshold at every humongous object allocation and may force an initial mark young collection immediately, if current occupancy exceeds that threshold.
- The humongous objects never move, not even during a Full GC. This can cause premature slow Full GCs or unexpected out-of-memory conditions with lots of free space left due to fragmentation of the region space.

Comparison to Other Collectors

This is a summary of the main differences between G1 and the other collectors:

- Parallel GC can compact and reclaim space in the old generation only as a whole. G1 incrementally distributes this work across multiple much shorter collections. This substantially shortens pause time at the potential expense of throughput.
- Similar to the CMS, G1 concurrently performs part of the old generation space-reclamation concurrently. However, CMS can't defragment the old generation heap, eventually running into long Full GC's.
- G1 may exhibit higher overhead than other collectors, affecting throughput due to its concurrent nature.

Due to how it works, G1 has some unique mechanisms to improve garbage collection efficiency:

- G1 can reclaim some completely empty, large areas of the old generation during any collection. This could avoid many otherwise unnecessary garbage collections, freeing a significant amount of space without much effort.
- G1 can optionally try to deduplicate duplicate strings on the Java heap concurrently.

Reclaiming empty, large objects from the old generation is always enabled. You can disable this feature with the option `-XX:-G1EagerReclaimHumongousObjects`. String deduplication is disabled by default. You can enable it using the option `-XX:+G1EnableStringDeduplication`.

Generally, when moving to G1 from other collectors, particularly the Concurrent Mark Sweep collector, start by removing all options that affect garbage collection, and only set the pause-time goal and overall heap size by using `-Xmx` and optionally `-Xms`.

Many options that are useful for other collectors to respond in some particular way, have either no effect at all, or even decrease throughput and the likelihood to meet the pause-time target. An example could be setting young generation sizes that completely prevent G1 from adjusting the young generation size to meet pause-time goals.

GC Tuning

https://docs.oracle.com/en/java/javase/11/gctuning/garbage-first-garbage-collector-tuning.html

Switching On GC Logging

The first thing to do is to add some switches to the application startup. These are best thought of as the "mandatory GC logging flags," which should be on for any Java/JVM application (except, perhaps, desktop apps). The flags are:

```
-Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintTenuringDistribution
-XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps
```

Let's look at each of these flags in more detail. Their usage is described in Table 8-1.

Table 8-1. Mandatory GC flags

Flag	Effect
`-Xloggc:gc.log`	Controls which file to log GC events to
`-XX:+PrintGCDetails`	Logs GC event details
`-XX:+PrintTenuringDistribution`	Adds extra GC event detail that is vital for tooling
`-XX:+PrintGCTimeStamps`	Prints the time (in secs since VM start) at which GC events occurred
`-XX:+PrintGCDateStamps`	Prints the wallclock time at which GC events occurred

The engineer should also know that these four main factors should be studied and measured during a tuning exercise:

- Allocation
- Pause sensitivity
- Throughput behavior
- Object lifetime

Allocation

Experience shows that sustained allocation rates of greater than 1 GB/s are almost always indicative of performance problems that cannot be corrected by tuning the garbage collector. Allocation rates also affect the number of objects that are promoted to Tenured. If we assume that short-lived Java objects have a fixed lifetime (when expressed in wallclock time), then higher allocation rates will result in young GCs that are closer together. If the collections become too frequent, then short-lived objects may not have had time to die and will be erroneously promoted to Tenured.

```
-XX:MaxTenuringThreshold=<n>
```

This controls the number of garbage collections that an object must survive to be promoted into Tenured. It defaults to 4 but can be set anywhere from 1 to 15. Changing this value represents a tradeoff between two concerns:

- The higher the threshold, the more copying of genuinely long-lived objects will occur.
- If the threshold is too low, some short-lived objects will be promoted, increasing memory pressure on Tenured.

Pause time

1. >1 s: Can tolerate over 1 s of pause
2. 1 s–100 ms: Can tolerate more than 200 ms but less than 1 s of pause
3. < 100 ms: Cannot tolerate 100 ms of pause

If we combine the pause sensitivity with the expected heap size of the application, we can construct a simple table of best guesses at a suitable collector. The result is shown in Table 8-4.

Table 8-4. Initial collector choice

	Pause time tolerance		
>1 s	1 s–100 ms	<100 ms	< 2 GB
Parallel	Parallel	CMS	< 4 GB
Parallel	Parallel/G1	CMS	< 4 GB
Parallel	Parallel/G1	CMS	< 10 GB
Parallel/G1	Parallel/G1	CMS	< 20 GB
Parallel/G1	G1	CMS	> 20 GB

These are guidelines and rules of thumb intended as a starting point for tuning, not 100% unambiguous rules.

GC Roots

These tradeoffs include that the scanning time needed to locate GC roots is affected by factors such as:

- Number of application threads
- Amount of compiled code in the code cache
- Size of heap

High numbers of application threads will also have an impact on GC times, as they represent more stack frames to scan and more time needed to reach a safepoint. They also exert more pressure on thread schedulers in both bare metal and virtualized environments.

As well as these traditional examples of GC roots, there are also other sources of GC roots, including JNI frames and the code cache for JIT-compiled code.

Performance Tests

- ➔ Latency test
  What is the end-to-end transaction time?
- ➔ Throughput test
  How many concurrent transactions can the current system capacity deal with?
- ➔ Load test
  Can the system handle a specific load? A load test differs from a throughput test (or a stress test) in that it is usually framed as a binary test: “Can the system handle this projected load or not?”
- ➔ Stress test
  What is the breaking point of the system?
- ➔ Endurance test
  What performance anomalies are discovered when the system is run for an extended period? Some problems manifest only over much longer periods of time (often measured in days). These include slow memory leaks, cache pollution, and memory fragmentation.
- ➔ Capacity planning test
  Does the system scale as expected when additional resources are added?
- ➔ Degradation
  What happens when the system partially fails?

When deciding where to focus your effort in a performance tuning exercise, there are three golden rules that can provide useful guidance:

- ➔ Identify what you care about and figure out how to measure it.
- ➔ Optimize what matters, not what is easy to optimize.
- ➔ Play the big points first.

Some goals are obvious:

- ➔ Reduce 95% percentile transaction time by 100 ms.
- ➔ Improve the system so that 5x throughput on existing hardware is possible.
- ➔ Improve average response time by 30%.

One particularly important Java-specific insight is related to JIT compilation. Modern JVMs analyze which methods are being run to identify candidates for JIT compilation to optimized machine code. This means that if a method is not being JIT-compiled, then one of two things is true about the method:

- ➔ It is not being run frequently enough to warrant being compiled.
- ➔ The method is too large or complex to be analyzed for compilation.

Problems with a straightforward microbenchmark test

A simple test that runs and checks some start/end time difference.

The first concern with the benchmark is that it goes straight into testing the code, without any consideration for warming up the JVM. Consider the case where the sort is running in a server application in production. It is likely to have been running for hours, maybe even days. However, we know that the JVM includes a Just-in-Time compiler that will convert interpreted bytecode to highly optimized machine code. This compiler only kicks in after the method has been run a certain number of times.
The test we are therefore conducting is not representative of how it will behave in production. The JVM will spend time optimizing the call while we are trying to benchmark. A warmup period is therefore desirable—it will allow the JVM to settle down before we capture our timings. Usually, this involves running the code we are about to benchmark for a number of iterations without capturing the timing details.

Another external factor that we need to consider is garbage collection. Ideally we want GC to be prevented from running during our time capturing, and also to be normalized after setup. Due to the nondeterministic nature of garbage collection, this is incredibly difficult to control.

JMH

JMH is a Java harness for building, running, and analyzing nano/micro/milli/macro benchmarks written in Java and other languages targeting the JVM.

One issue is that if the benchmark framework were to call the user’s code for a large number of iterations, loop optimizations might be triggered. This means the actual process of running the benchmark can cause issues with reliable results.
In order to avoid hitting loop optimization constraints, JMH generates code for the benchmark, wrapping the benchmark code in a loop with the iteration count carefully set to a value that avoids optimization.

Native Memory

The heap (usually) accounts for the largest amount of memory used by the JVM, but the JVM also uses memory for its internal operations. This non heap memory is native memory. The total of native and heap memory used by the JVM yields the total footprint of an application.
Consider a heap that is specified with the parameters -Xms512m -Xmx2048m. The heap starts by using 512 MB, and it will be resized as needed to meet the GC goals of the application.

That concept is the essential difference between committed (or allocated) memory and reserved memory (sometimes called the virtual size of a process). The JVM must tell the

operating system that it might need as much as 2 GB of memory for the heap, so that memory is reserved: the operating system promises that when the JVM attempts to allocate additional memory when it increases the size of the heap, that memory will be available.

Still, only 512 MB of that memory is allocated initially, and that 512 MB is all of the memory that is being used (for the heap). That (actually allocated) memory is known as the committed memory. The amount of committed memory will fluctuate as the heap resizes; in particular, as the heap size increases, the committed memory correspondingly increases. Every time the JVM creates a thread, the OS allocates some native memory to hold that thread's stack, committing more memory to the process (until the thread exits at least).

To minimize the footprint used by the JVM, limit the amount of memory used by the following:

Heap
: The heap is the biggest chunk of memory, though surprisingly it may take up only 50% to 60% of the total footprint. Using a smaller maximum heap (or setting the GC tuning parameters such that the heap never fully expands) limits the program's footprint.

Thread stacks
: Thread stacks are quite large, particularly for a 64-bit JVM. See Chapter 9 for ways to limit the amount of memory consumed by thread stacks.

Code cache
: The code cache uses native memory to hold compiled code. As discussed in Chapter 4, this can be tuned (though performance will suffer if all the code cannot be compiled because of space limitations).

Native library allocations
: Native libraries can allocate their own memory, which can sometimes be significant.

Using the option `-XX:NativeMemoryTracking=off|summary|detail` enables this visibility. By default, Native Memory Tracking (NMT) is off. If the summary or detail mode is enabled, you can get the native memory information at any time from `jcmd`:

```
% jcmd process_id VM.native_memory summary
```

If the JVM is started with the argument `-XX:+PrintNMTStatistics` (by default, `false`), the JVM will print out information about the allocation when the program exits.

This memory usage breaks down as follows. The heap itself is (unsurprisingly) the largest part of the reserved memory at 4 GB. But the dynamic sizing of the heap meant it grew only to 268 MB (in this case, the heap sizing was `-Xms256m -Xmx4g`, so the actual heap usage has expanded only a small amount):

```
-                 Java Heap (reserved=4194304KB, committed=268288KB)
                            (mmap: reserved=4194304KB, committed=268
```

Next is the native memory used to hold class metadata. Again, note that the JVM has reserved more memory than it used to hold the 24,316 classes in the program. The committed size here will start at the value of the `MetaspaceSize` flag and grow as needed until it reaches the value of the `MaxMetaspaceSize` flag:

```
-                     Class (reserved=1182305KB, committed=150497KB)
                            (classes #24316)
                            (malloc=2657KB #35368)
                            (mmap: reserved=1179648KB, committed=147
```

Seventy-seven thread stacks were allocated at about 1 MB each:

```
-                    Thread (reserved=84455KB, committed=84455KB)
                            (thread #77)
                            (stack: reserved=79156KB, committed=7915
                            (malloc=243KB, #314)
                            (arena=5056KB, #154)
```

Then comes the JIT code cache: 24,316 classes is not very many, so just a small section of the code cache is committed:

```
-                       Code (reserved=102581KB, committed=15221KB)
                             (malloc=2741KB, #4520)
                             (mmap: reserved=99840KB, committed=12480
```

Next is the area outside the heap that GC algorithm uses for its processing. The size of this area depends on the GC algorithm in use: the (simple) serial collector will reserve far less than the more complex G1 GC algorithm (though, in general, the amount here will never be very large):

```
-                         GC (reserved=199509KB, committed=53817KB)
                             (malloc=11093KB #18170)
                             (mmap: reserved=188416KB, committed=4272
```

Similarly, this area is used by the compiler for its operations, apart from the resulting code placed in the code cache:

```
-                   Compiler (reserved=162KB, committed=162KB)
                             (malloc=63KB, #229)
                             (arena=99KB, #3)
```

Internal operations of the JVM are represented here. Most of them tend to be small, but one important exception is direct byte buffers, which are allocated here:

```
-                 Internal (reserved=10584KB, committed=10584KB)
                           (malloc=10552KB #32851)
                           (mmap: reserved=32KB, committed=32KB)
```

Symbol table references (constants from class files) are held here:

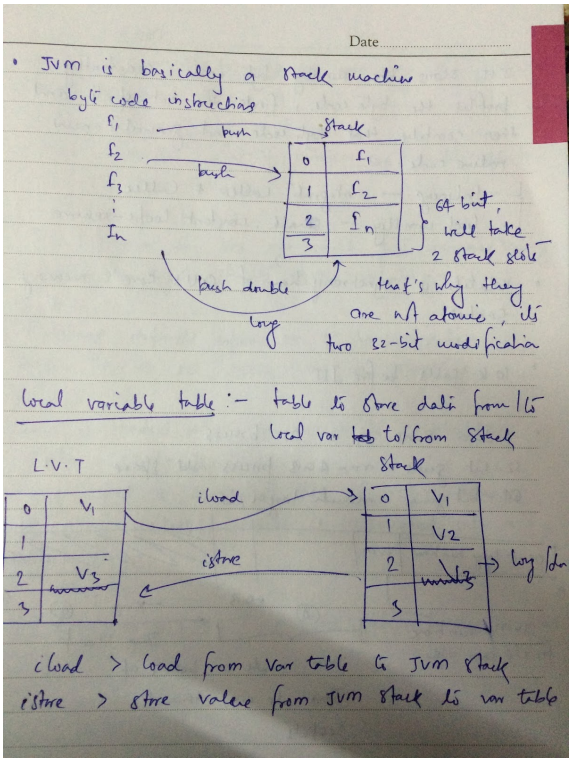
```
-                   Symbol (reserved=12093KB, committed=12093KB)
                           (malloc=10039KB, #110773)
                           (arena=2054KB, #1)
```

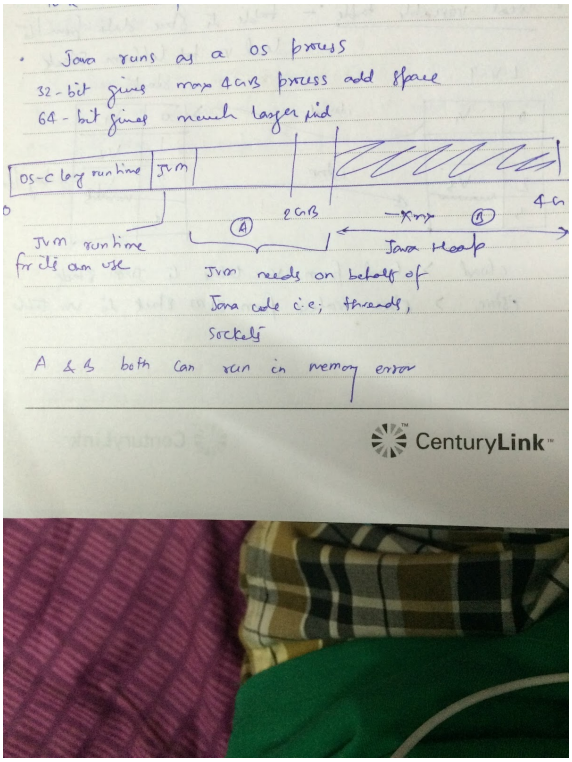
NMT itself needs some space for its operation (which is one reason it is not enabled by default):

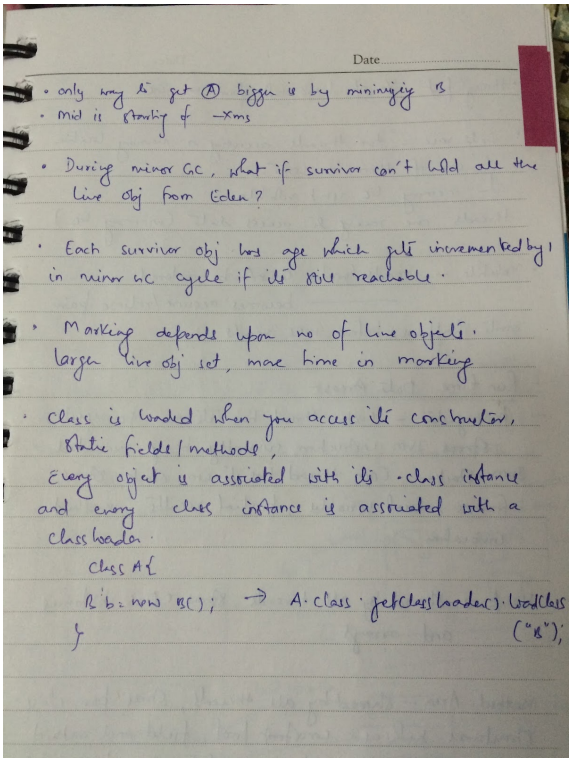
```
-    Native Memory Tracking (reserved=7195KB, committed=7195KB)
                           (malloc=16KB #199)
                           (tracking overhead=7179KB)
```

Finally, here are some minor bookkeeping sections of the JVM:

```
-               Arena Chunk (reserved=188KB, committed=188KB)
                           (malloc=188KB)
-                   Unknown (reserved=8192KB, committed=0KB)
                           (mmap: reserved=8192KB, committed=0KB)
```







Date

- throughput – % of time not spent in GC
- Data race – two threads accessing a memory location
  - one of them is write
  - memory loc isn't volatile

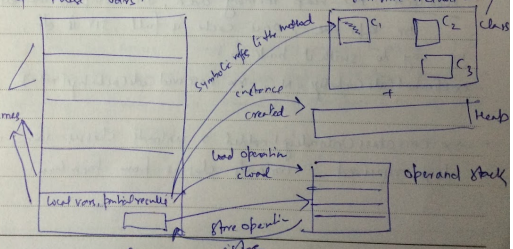
  threads are racing to access data (memory loc)
- Volatile – reads/writes cannot be ordered
  - becomes acquire/release pair

  write happens-before all reads
- Run time Data Areas –
  - pc register – Each thread has its own pc register. Stores JVM instruction currently being executed
  - JVM Stack – Each thread has its own stack, stores frames, local variables, partial results, method invocation
  - Heap – Shared by all threads, stores class instances and arrays
  - Method Area – Shared by all threads, stores per-class structures such as – constant pool, field and method data, method & constructor code. Generally part of heap. Is a part of PermGen

Date

but inclusion in garbage collection depends on the JVM implementor.

- Runtime constant pool – per-class structure, kind of Symbol-Table, stores constants, pool is created in method area when JVM creates a class.
- Native method stack – native method calls
- Frame – new frame is created on the stack by JVM when a method is invoked, stores local variables, an operand stack, a ref to the run-time constant pool. frame has symbolic refs to the method to be invoked & vars to be accessed, dynamic linking translate these refs to the concrete method refs with the run-time location of these vars.



Date

- non Array types are loaded by classloaders
  Array types – Array class is created directly by JVM
  T[]
  - created by JVM
  - created by classloader
- Get current PermGen size – using jmap -heap <PID> or KILL -3 <PID> – heap dump has memory/type info
- How to generate thread dump on out of memory error – on OOME, JVM creates heap dump automatically but not thread dump. To create thread dump – create uncaughtExceptionHandler > then check if it is OOMError, if true print thread stack
- Code cache – non-heap memory area, JIT compiled methods are kept here, once cache is full, JIT is off, no way to switch it back on.

  -XX:InitialCodeCacheSize=N, -XX:ReservedCodeCacheSize(max)
- -XX:CMSClassUnloadingEnabled > unloads classes when no longer required, unloads classes from PermGen

Date

- jmap -permstat <pid> – perm gen info
- max no. of method parameters allowed – 255
- Direct mapping b/w Java thread and native thread. When a new Java thread is created, JVM allocates a new thread stack. Once JVM has its all preparation data – synchronization/lock details, pc, allocation buffers, a new native thread on OS is created & scheduled by OS. Once Java thread completes its execution, native thread is also released.
- CMS – low-latency, short pauses, non-compacting, requires large heap size,
- We need serialization, when we need an obj to exist beyond the lifetime of JVM
- OOME can occur – heap (no more memory is allocate) / PermGen (no memory is free class data) / native thread (no memory is create native thread) / OS-space (no OS space available for JVM)