Heap:

- Stores actual object in memory
- There exists only one heap memory for each running jvm process. Therefore this is shared part of memory regardless of how many threads are running.

Stack:

- Holds references to heap objects
- Store value of primitive types
- Maintain scope
- Only objects from active scope are used.
- Once method completes and returns, the top of the stack pops out, and active scope changes.
- Stack memory in java is allocated per thread.

Reference Types:
The differences between type of references is that the objects on the heap they refer to, are eligible for garbage collection.

- Strong reference:
  - The object on the heap is not garbage collected while there is a strong reference pointing to it, or if it is strongly reachable through a chain of strong references.
- Weak reference:
  - A weak reference to an object from the heap is most likely to not survive after the next garbage collection: new WeakReference<>(), WeakHashMap<K,V>()
  - Use Case: caching scenarios
- Soft reference:
  - Used for more memory sensitive scenarios
  - Those are going to be garbage collected only when your application is running low on memory.
  - As long as there is no critical need to free up some space, the garbage collector will not touch softly reachable objects.
  - Java guarantees that all soft references objects are cleaned up before it throws an OutOfMemoryError.
  - E.g.: new SoftReference<>()
- Phantom reference:
  - Used to schedule post-mortem cleanup actions.
  - Preferable to finalizers.

How Strings are referenced:

- Strings created using string literals are stored in string pool.
- String created using new keyword are stored in heap.

Garbage Collector types:

- Serial GC: A single thread collector. -XX:+UseSerialGC
- Parallel GC: multiple threads to perform garbage collecting process. -XX:+UseParallelGC
- Mostly Concurrent GC: works concurrent to application. It does not work 100% concurrent to the application, hence the name mostly concurrent.
- Garbage First: High throughput with a reasonable application pause time. -XX:+UseG1GC
- Concurrent Mark Sweep: The application pause time is

Garbage collection:

- If an objects reference from stack is lost, it cannot be accessed anymore, so it is considered garbage and eligible to be collected by garbage collector.
- Garbage collector is triggered by Java, it is upto java when and whether or not to start this process.
- It is actually an expensive process. When the garbage collector runs, all threads in the application are paused.
- You may ask java explicitly to run garbage collector by calling System.gc() and it upto java whether or not to run.

Mark and Sweep:

- Java analyzes the variables from the stack and marks all the objects that need to be kept alive. Then all the unused objects are cleaned up.
- To make this more optimized, heap memory actually consists of multiple parts.
- We can visualize the memory usage and other useful things with JVisualVM, a tool that comes with Java JDK.
- The only thing to do is to install a plugin named Visual GC, which allows you to see how the memory is actually structured.
- **Eden(1)**: when an object is created, it is allocated on the Eden(1) space. Because the Eden space is not that big, it gets full quite fast. The garbage collector runs on the Eden space and marks objects as alive.
- **S0(2):** once an object survives a garbage collecting process, it gets moved into so-called survivor space S0(2).
- **S1(3)**: The second time the garbage collector runs on Eden space, it moves all surviving objects to S1(3) space. Also everything that is currently on S0(2) is moved into S1(3)
- **Old(4)**: If an object survives for X rounds of garbage collection. It is most likely that it will survive forever, and it gets moved into Old(4) space.
- **Metaspace(5)** is used to store the metadata about your loaded classes in the JVM. Prior to Java 8, the metaspace is actually called PermGen. In Java 6, this space is used to store the memory for string pool. If you have too many strings in Java 6 application, it might crash.

kept to minimum. -XX:+UserConcMarkSweepGC. ● As of JDK 9, this GC type is deprecated. Default is G1GC.	
Tips & Tricks: ● To minimize the memory footprint, limit the scope of variables as much as possible. ● Explicitly refer to num obsolete references. ● Avoid finalizers. Prefer phantom references for cleanup work. ● Do not use strong references where weak or soft references apply. The most common memory pitfalls are caching scenarios, where data is held in memory even if it might not be needed. ● JVisualVM also has the functionality to make a heap dump at certain point, so you can analyze per class, how much memory it occupies. ● Configure JVM based on your application requirements. ● Explicitly specify the heap size for the JVM when running the application. ○ Initial heap size -Xms512m ○ Maximus heap size -Xmx1024m ○ Thread stack size -Xss128m ○ Young generation size -Xmn256m ● If application crashed with an OutOfMemoryError and you need some extra info to detect the leak, run the process with the -XX:HeapDumpOnOutOfMemory parameter. ● Use the -verbose:gc option to get the garbage collection output.	Java Performance tuning tips: ● Don't optimize before you know it is necessary. ○ In most cases, premature optimization takes up a lot of time and makes code hard to read and maintain. ○ First, define how fast your application code has to be. For e.g. specify the maximum response time for all the API calls. Or the number of records you want to import in the specified time frame. ● Use a profiler to find the real bottleneck. Profiler e.g. ○ Standard JVM profilers: Visual Vm and JProfiler. ○ LightWeight Java Transaction profilers: XRebel and Stackify ○ Low overhead, Java JVM profiling in production: AppDynamics and Dynatrace. ● Create a performance test suite for the whole application. ● Work on the biggest bottleneck first ● Use StringBuilder to concatenate Strings programmatically. ● Use + to concatenate Strings in one statement. ● Use primitives where possible. To store value on the stack instead of heap. ● Try to avoid BigInteger and BigDecimal. ● Check the current log level first. ● Use standard utility methods from apache commons. ● Cache expensive resource like database connections.

Note: Information gathered in this document has been collected from various sources on Internet.