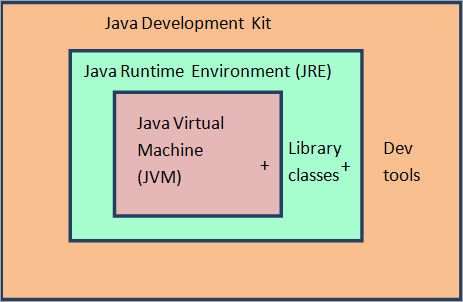
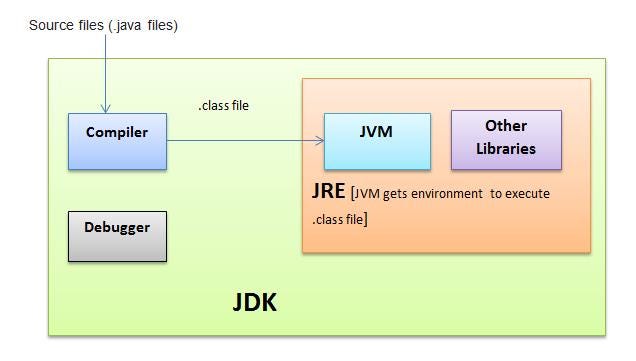
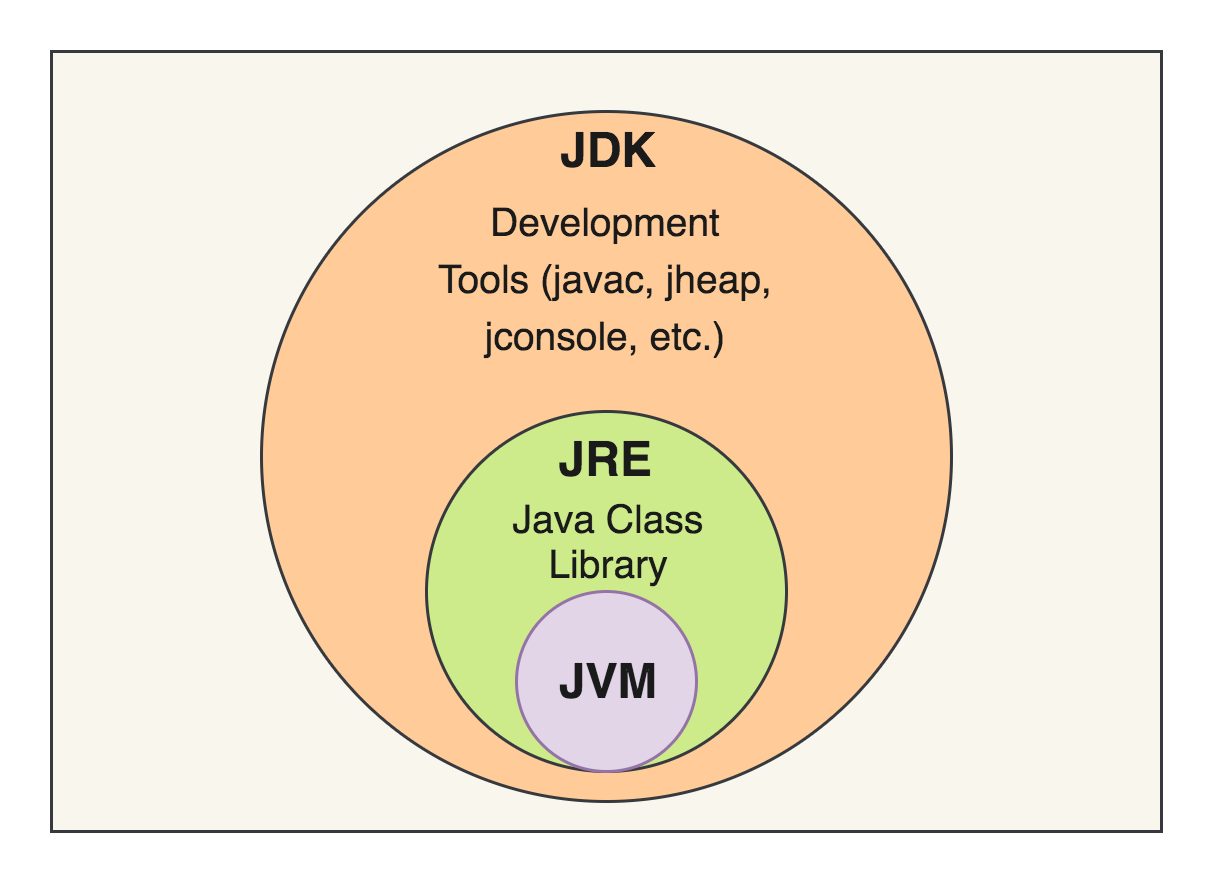
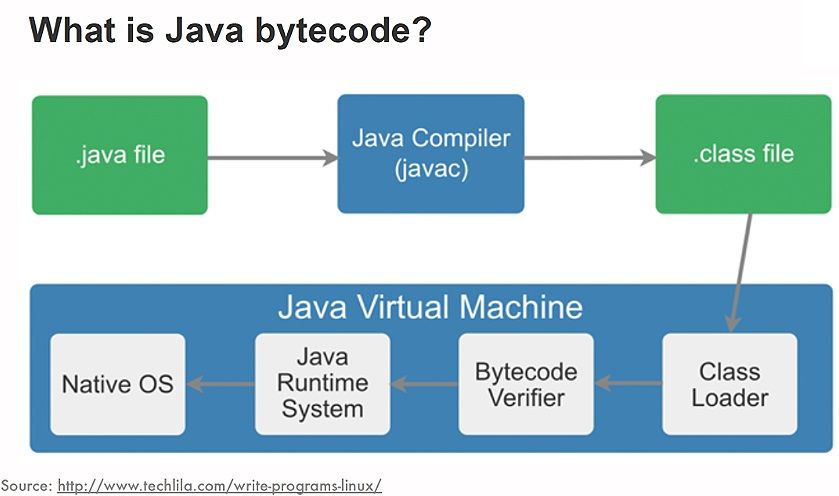
**Java**









**Java Memory Management**

This article will focus on Java memory management, how the heap works, reference types, garbage collection, and also related concepts.

**Why Learn Java Memory Management?**  
We all know that Java itself manages the memory and needs no explicit intervention of the programmer. Garbage collector itself ensures that the unused space gets cleaned and memory can be freed when not needed. So what’s the role of programmer and why a programmer needs to learn about the Java Memory Management ? Being a programmer, you don’t need to bother with problems like destroying objects, all credits to the garbage collector. However the automatic garbage collection doesn’t guarantee everything. If we don’t know how the memory management works, often we will end up amidst things that are not managed by JVM (Java Virtual Machine). There are some objects that aren’t eligible for the automatic garbage collection.

Hence knowing the memory management is essential as it will benefit the programmer to write high performance based programs that will not crash, or if does so, the programmer will know how to debug or overcome the crashes.

**Introduction:**

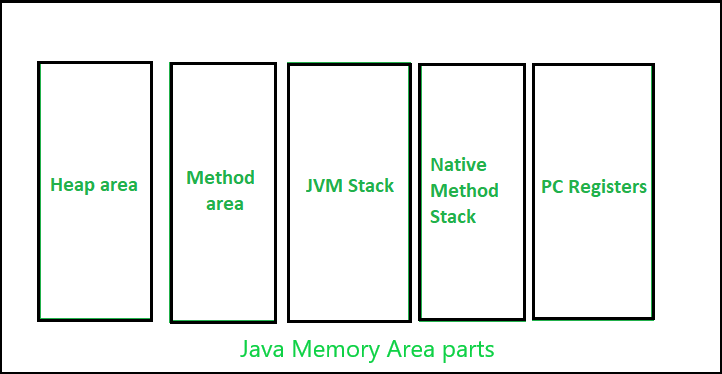
In every programming language, the memory is a vital resource and is also scarce in nature. Hence it’s essential that the memory is managed thoroughly without any leaks. Allocation and deallocation of memory is a critical task and requires a lot of care and consideration. However in Java, unlike other programming language, the JVM and to be specific Garbage Collector has the role of managing memory allocation so that the programmer needs not to. Whereas in other programming languages such as C the programmer has direct access to the memory who allocates memory in his code, thereby creating a lot of scope for leaks.

The major concepts in Java Memory Management :

* JVM Memory Structure
* Working of Garbage Collector

**Java Memory Structure:**

JVM defines various run time data area which are used during execution of a program. Some of the areas are created by the JVM whereas some are created by the threads that are used in a program. However, the memory area created by JVM is destroyed only when the JVM exits. The data areas of thread are created during instantiation and destroyed when the thread exits.



JVM Memory area parts

Let’s study these parts of memory area in detail:

**Heap :**

* It is a shared runtime data area and stores the actual object in a memory. It is instantiated during the virtual machine startup.
* This memory is allocated for all class instances and array. Heap can be of fixed or dynamic size depending upon the system’s configuration.
* JVM provides the user control to initialize or vary the size of heap as per the requirement. When a new keyword is used, object is assigned a space in heap, but the reference of the same exists onto the stack.
* There exists one and only one heap for a running JVM process.

Scanner sc = new Scanner(System.in);

The above statement creates the object of Scanner class which gets allocated to heap whereas the reference ‘sc’ gets pushed to the stack.

**Note:** Garbage collection in heap area is mandatory.

**Method Area:**

* It is a logical part of the heap area and is created on virtual machine startup.
* This memory is allocated for class structures, method data and constructor field data, and also for interfaces or special method used in class. Heap can be of fixed or dynamic size depending upon the system’s configuration.
* Can be of a fixed size or expanded as required by the computation. Needs not to be contiguous.

**Note:** Though method area is logically a part of heap, it may or may not be garbage collected even if garbage collection is compulsory in heap area.

**JVM Stacks:**

* A stack is created at the same time when a thread is created and is used to store data and partial results which will be needed while returning value for method and performing dynamic linking.
* Stacks can either be of fixed or dynamic size. The size of a stack can be chosen independently when it is created.
* The memory for stack needs not to be contiguous.

**Native method Stacks:**

Also called as C stacks, native method stacks are not written in Java language. This memory is allocated for each thread when its created. And it can be of fixed or dynamic nature.

**Program counter (PC) registers:**

Each JVM thread which carries out the task of a specific method has a program counter register associated with it. The non native method has a PC which stores the address of the available JVM instruction whereas in a native method, the value of program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.

**Working of a Garbage Collector:**

* JVM triggers this process and as per the JVM garbage collection process is done or else withheld. It reduces the burden of programmer by automatically performing the allocation or deallocation of memory.
* Garbage collection process causes the rest of the processes or threads to be paused and thus is costly in nature. This problem is unacceptable for the client but can be eliminated by applying several garbage collector based algorithms. This process of applying algorithm is often termed as **Garbage Collector tuning** and is important for improving the performance of a program.
* Another solution is the generational garbage collectors that adds an age field to the objects that are assigned a memory. As more and more objects are created, the list of garbage grows thereby increasing the garbage collection time. On the basis of how many clock cycles the objects have survived, objects are grouped and are allocated an ‘age’ accordingly. This way the garbage collection work gets distributed.
* In the current scenario, all garbage collectors are generational, and hence, optimal.

**Note:** **System.gc()** and **Runtime.gc()** are the methods which requests for Garbage collection to JVM explicitly but it doesn’t ensures garbage collection as the final decision of garbage collection is of JVM only.

Knowing how the program and it’s data is stored or organized is essential as it helps when the programmer intends to write an optimized code in terms of resources and it’s consumption. Also it helps in finding the memory leaks or inconsistency, and helps in debugging memory related errors. However, the memory management concept is extremely vast and therefore one must put his best to study it as much as possible to improve the knowledge of the same.

**2. Questions**

**Q1. What Does the Statement “Memory Is Managed in Java” Mean?**

Memory is the key resource an application requires to run effectively and like any resource, it is scarce. As such, its allocation and deallocation to and from applications or different parts of an application require a lot of care and consideration.

However, in Java, a developer does not need to explicitly allocate and deallocate memory – the JVM and more specifically the Garbage Collector – has the duty of handling memory allocation so that the developer doesn't have to.

This is contrary to what happens in languages like C where a programmer has direct access to memory and literally references memory cells in his code, creating a lot of room for memory leaks.

**Q2. What Is Garbage Collection and What Are Its Advantages?**

Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

The biggest advantage of garbage collection is that it removes the burden of manual memory allocation/deallocation from us so that we can focus on solving the problem at hand.

**Q3. Are There Any Disadvantages of Garbage Collection?**

Yes. Whenever the garbage collector runs, it has an effect on the application's performance. This is because all other threads in the application have to be stopped to allow the garbage collector thread to effectively do its work.

**AD**

Depending on the requirements of the application, this can be a real problem that is unacceptable by the client. However, this problem can be greatly reduced or even eliminated through skillful optimization and garbage collector tuning and using different GC algorithms.

**Q4. What Is the Meaning of the Term “Stop-The-World”?**

When the garbage collector thread is running, other threads are stopped, meaning the application is stopped momentarily. This is analogous to house cleaning or fumigation where occupants are denied access until the process is complete.

Depending on the needs of an application, “stop the world” garbage collection can cause an unacceptable freeze. This is why it is important to do garbage collector tuning and JVM optimization so that the freeze encountered is at least acceptable.

**Q5. What Are Stack and Heap? What Is Stored in Each of These Memory Structures, and How Are They Interrelated?**

The stack is a part of memory that contains information about nested method calls down to the current position in the program. It also contains all local variables and references to objects on the heap defined in currently executing methods.

This structure allows the runtime to return from the method knowing the address whence it was called, and also clear all local variables after exiting the method. Every thread has its own stack.

**AD**

The heap is a large bulk of memory intended for allocation of objects. When you create an object with the *new* keyword, it gets allocated on the heap. However, the reference to this object lives on the stack.

**Q6. What Is Generational Garbage Collection and What Makes It a Popular Garbage Collection Approach?**

Generational garbage collection can be loosely defined as the strategy used by the garbage collector where the heap is divided into a number of sections called generations, each of which will hold objects according to their “age” on the heap.

Whenever the garbage collector is running, the first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not. This can be a very time-consuming process if all objects in a system must be scanned.

As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short-lived.

With generational garbage collection, objects are grouped according to their “age” in terms of how many garbage collection cycles they have survived. This way, the bulk of the work spread across various minor and major collection cycles.

Today, almost all garbage collectors are generational. This strategy is so popular because, over time, it has proven to be the optimal solution.

**Q7. Describe in Detail How Generational Garbage Collection Works**

To properly understand how generational garbage collection works, it is important to first **remember how Java heap is structured** to facilitate generational garbage collection.

The heap is divided up into smaller spaces or generations. These spaces are Young Generation, Old or Tenured Generation, and Permanent Generation.

The **young generation hosts most of the newly created objects**. An empirical study of most applications shows that majority of objects are quickly short lived and therefore, soon become eligible for collection. Therefore, new objects start their journey here and are only “promoted” to the old generation space after they have attained a certain “age”.

The term **“age”** in generational garbage collection **refers to the number of collection cycles the object has survived**.

The young generation space is further divided into three spaces: an Eden space and two survivor spaces such as Survivor 1 (s1) and Survivor 2 (s2).

The **old generation hosts objects that** **have lived in memory longer than a certain “age”**. The objects that survived garbage collection from the young generation are promoted to this space. It is generally larger than the young generation. As it is bigger in size, the garbage collection is more expensive and occurs less frequently than in the young generation.

The **permanent generation** **or more commonly called, *PermGen,* contains metadata required by the JVM** to describe the classes and methods used in the application. It also contains the string pool for storing interned strings. It is populated by the JVM at runtime based on classes in use by the application. In addition, platform library classes and methods may be stored here.

First, **any new objects are allocated to the Eden space**. Both survivor spaces start out empty. When the Eden space fills up, a minor garbage collection is triggered. Referenced objects are moved to the first survivor space. Unreferenced objects are deleted.

**A Definition of Java Garbage Collection**

Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

**How Java Garbage Collection Works**

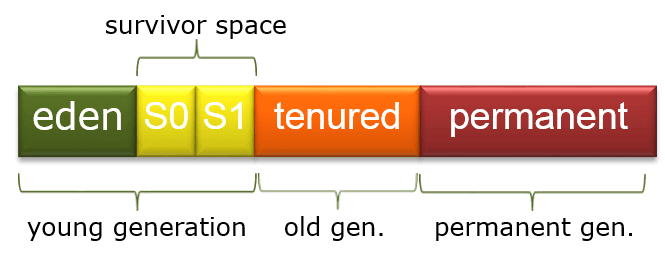
Java garbage collection is an automatic process. The programmer does not need to explicitly mark objects to be deleted. The garbage collection implementation lives in the JVM. Every JVM can implement garbage collection however it pleases. The only requirement is that it should meet the JVM specification. Although there are many JVMs, Oracle’s HotSpot is by far the most common. It offers a robust and mature set of garbage collection options.

**What are the Various Steps During the Garbage Collection?**

While HotSpot has multiple garbage collectors that are optimized for various use cases, all its garbage collectors follow the same basic process. In the first step, [unreferenced objects](https://www.javatpoint.com/Garbage-Collection) are identified and marked as ready for garbage collection. In the second step, marked objects are deleted. Optionally, memory can be compacted after the garbage collector deletes objects, so remaining objects are in a contiguous block at the start of the heap. The compaction process makes it easier to allocate memory to new objects sequentially after the JVM allocates the memory blocks to existing objects.

**How Generational Garbage Collection Strategy Works**

All of HotSpot’s garbage collectors implement a generational garbage collection strategy that categorizes objects by age. The rationale behind generational garbage collection is that most objects are short-lived and will be ready for garbage collection soon after creation.



*Image via* [*Wikipedia*](https://de.wikipedia.org/wiki/Datei:JavaGCgenerations.png)

**What are Different Classification of Objects by Garbage Collector?**

We can divide the heap into [three sections](https://plumbr.eu/handbook/garbage-collection-in-java):

* **Young Generation**: Newly created objects start in the Young Generation. The garbage collector further subdivides Young Generation into an Eden space, where all new objects start, and two Survivor spaces, where it moves objects from Eden after surviving one garbage collection cycle. When objects are garbage collected from the Young Generation, it is a minor garbage collection event.
* **Old Generation:** Eventually, the garbage collector moves the long-lived objects from the Young Generation to the Old Generation. When objects are garbage collected from the Old Generation, it is a major garbage collection event.
* **Permanent Generation:** The JVM stores the metadata, such as classes and methods, in the Permanent Generation. JVM garbage collects the classes from the Permanent Generation that are no longer in use.

During a full garbage collection event, unused objects from all generations are garbage collected.

**What are Different Types of Garbage Collector?**

HotSpot has four garbage collectors:

* **Serial:** All garbage collection events are conducted serially in one thread. JVM executes the compaction after each garbage collection.
* **Parallel:** JVM uses multiple threads for minor garbage collection. It uses a single thread for major garbage collection and Old Generation compaction. Alternatively, the Parallel Old variant uses multiple threads for major garbage collection and Old Generation compaction.
* **CMS (Concurrent Mark Sweep):** Multiple threads are used for minor garbage collection using the same algorithm as Parallel. Major garbage collection is multi-threaded, like Parallel Old. Still CMS runs concurrently alongside application processes to minimize “stop the world” events (i.e., when the garbage collector running stops the application). Here, the JVM does not perform compaction of memory.
* **G1 (Garbage First):** The newest garbage collector is intended as a replacement for CMS. It is parallel and concurrent, like CMS. However, it works quite differently under the hood than older garbage collectors.

**Benefits of Java Garbage Collection**

The biggest benefit of Java garbage collection is that it automatically handles the deletion of unused objects or objects that are [out of reach](http://beginnersbook.com/2013/04/java-garbage-collection/) to free up vital memory resources. Programmers working in languages without garbage collection (like C and C++) must implement manual memory management in their code.

Despite the extra work required, some programmers argue in favor of manual memory management over garbage collection, primarily for reasons of [control and performance](http://www.javaworld.com/article/2078645/java-se/jvm-performance-optimization-part-3-garbage-collection.html). While the debate over memory management approaches continues to rage on, garbage collection is now a standard component of many popular programming languages. For scenarios in which the garbage collector is negatively impacting performance, Java offers many options for tuning the garbage collector to improve its efficiency.

**What Triggers Garbage Collection?**

The Garbage Collection process is triggered by a variety of events that signal to the Garbage Collector that memory needs to be reclaimed.

Here are some common events that trigger Java Garbage Collection:

1. **Allocation Failure:** When an object cannot be allocated in the heap because there is not enough contiguous free space available, the JVM triggers the Garbage Collection to free up memory.
2. **Heap Size:** When the heap reaches a certain capacity threshold, the JVM triggers Garbage Collection to reclaim memory and prevent an OutOfMemoryError.
3. **System.gc():** Calling the System.gc()  method can trigger Garbage Collection, although it does not guarantee that Garbage Collection will occur.
4. **Time-Based:** Some Garbage Collection algorithms, such as G1 Garbage Collection, use time-based triggers to initiate Garbage Collection.

**Ways for requesting JVM to run Garbage Collector**

There are several ways to request the JVM to run Garbage Collector in a Java application:

**System.gc() method:**

Calling this method is the most common way to request Garbage Collection in a Java application. However, it does not guarantee that Garbage Collection will occur as it is only a suggestion to the JVM.

**Runtime.getRuntime().gc() method:**

This method provides another way to request Garbage Collection in a Java application. This method is similar to the System.gc() method, and it also suggests that the JVM should run Garbage Collector, but again it does not guarantee that Garbage Collection will occur.

**JConsole or VisualVM:**

JConsole or VisualVM is a profiling tool that is included with the Java Development Kit. These tools provide a graphical user interface that allows developers to monitor the memory usage of their Java application in real-time. They also provide a way to request Garbage Collection on-demand by clicking a button.

**Command-Line Options:**

The JVM can be configured with various command-line options to control Garbage Collection. For example, the -Xmx option can be used to specify the maximum heap size, which can affect the frequency and duration of Garbage Collection events. The -XX:+DisableExplicitGC option can be used to disable explicit calls to System.gc() or Runtime.getRuntime().gc().

**Heap Dumps:**

Heap dumps are snapshots of the Java heap that can be taken at any time during the application’s execution. They can be analyzed to identify memory leaks or other memory-related issues. Heap dumps can be requested using command-line options or profiling tools.

It is worth noting that requesting Garbage Collection too frequently can negatively impact the performance of the application. It is important to monitor the memory usage of the application and only request Garbage Collection when it is necessary. By using profiling tools and selecting appropriate Garbage Collection algorithms, developers can ensure that Garbage Collection is triggered in a way that minimizes the impact on the application’s performance.

**Why Does a Programmer need to Understand Garbage Collection?**

For many simple applications, Java garbage collection is not something that a programmer needs to consciously consider. However, for programmers who want to advance their Java skills, it is important to understand how Java garbage collection works and the ways in which it can be tuned.

Besides the basic mechanisms of garbage collection, one of the most important points to understand about garbage collection in Java is that it is non-deterministic, and there is no way to predict when garbage collection will occur at run time. It is possible to include a hint in the code to run the garbage collector with the System.gc()  or Runtime.getRuntime().gc()  methods, but they provide no guarantee that the garbage collector will actually run.



**Java Garbage Collection Best Practices**

The best approach to tuning Java garbage collection is setting flags on the JVM. Various flags such as the initial and maximum size of the heap, the size of the heap sections (e.g. Young Generation, Old Generation), can adjust the garbage collector to be used (e.g. Serial, G1, etc.). The nature of the application being tuned is a good initial guide to settings. For example, the Parallel garbage collector is efficient but will frequently cause “stop the world” events, making it better suited for backend processing where long pauses for garbage collection are acceptable.

On the other hand, the CMS garbage collector is designed to minimize pauses, making it ideal for GUI applications where responsiveness is important. Additional fine-tuning can be accomplished by changing the size of the heap or its sections and measuring garbage collection efficiency using a tool like jstat.

Try Stackify’s free code profiler, [Prefix](https://stackify.com/prefix/), to write better code on your workstation. Prefix works with .NET, Java, PHP, Node.js, Ruby, and Python.

During the next minor GC, the same thing happens to the Eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S2).

In addition, objects from the last minor GC in the first survivor space (S1) have their age incremented and are moved to S2. Once all surviving objects have been moved to S2, both S1 and Eden space are cleared. At this point, S2 contains objects with different ages.

**AD**

At the next minor GC, the same process is repeated. However this time the survivor spaces switch. Referenced objects are moved to S1 from both Eden and S2. Surviving objects are aged. Eden and S2 are cleared.

After every minor garbage collection cycle, the age of each object is checked. Those that have reached a certain arbitrary age, for example, 8, are promoted from the young generation to the old or tenured generation. For all subsequent minor GC cycles, objects will continue to be promoted to the old generation space.

This pretty much exhausts the process of garbage collection in the young generation. Eventually, a major garbage collection will be performed on the old generation which cleans up and compacts that space. For each major GC, there are several minor GCs.

**Q8. When Does an Object Become Eligible for Garbage Collection? Describe How the Gc Collects an Eligible Object?**

An object becomes eligible for Garbage collection or GC if it is not reachable from any live threads or by any static references.

The most straightforward case of an object becoming eligible for garbage collection is if all its references are null. Cyclic dependencies without any live external reference are also eligible for GC. So if object A references object B and object B references Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.

Another obvious case is when a parent object is set to null. When a kitchen object internally references a fridge object and a sink object, and the kitchen object is set to null, both fridge and sink will become eligible for garbage collection alongside their parent, kitchen.

**Q9. How Do You Trigger Garbage Collection from Java Code?**

**You, as Java programmer, can not force garbage collection in Java**; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size.

Before removing an object from memory garbage collection thread invokes finalize()method of that object and gives an opportunity to perform any sort of cleanup required. You can also invoke this method of an object code, however, there is no guarantee that garbage collection will occur when you call this method.

**AD**

Additionally, there are methods like System.gc() and Runtime.gc() which is used to send request of Garbage collection to JVM but it’s not guaranteed that garbage collection will happen.

**Q10. What Happens When There Is Not Enough Heap Space to Accommodate Storage of New Objects?**

If there is no memory space for creating a new object in Heap, Java Virtual Machine throws *OutOfMemoryError* or more specifically ***java.lang.OutOfMemoryError* heap space.**

**Q11. Is It Possible to «Resurrect» an Object That Became Eligible for Garbage Collection?**

When an object becomes eligible for garbage collection, the GC has to run the *finalize* method on it. The *finalize* method is guaranteed to run only once, thus the GC flags the object as finalized and gives it a rest until the next cycle.

In the *finalize* method you can technically “resurrect” an object, for example, by assigning it to a *static* field. The object would become alive again and non-eligible for garbage collection, so the GC would not collect it during the next cycle.

The object, however, would be marked as finalized, so when it would become eligible again, the finalize method would not be called. In essence, you can turn this “resurrection” trick only once for the lifetime of the object. Beware that this ugly hack should be used only if you really know what you're doing — however, understanding this trick gives some insight into how the GC works.

**Q12. Describe Strong, Weak, Soft and Phantom References and Their Role in Garbage Collection.**

Much as memory is managed in Java, an engineer may need to perform as much optimization as possible to minimize latency and maximize throughput, in critical applications. Much as **it is impossible to explicitly control when garbage collection is triggered** in the JVM, **it is possible to influence how it occurs as regards the objects we have created.**

Java provides us with reference objects to control the relationship between the objects we create and the garbage collector.

By default, every object we create in a Java program is strongly referenced by a variable:

**AD**

StringBuilder sb = new StringBuilder();

In the above snippet, the *new* keyword creates a new *StringBuilder* object and stores it on the heap. The variable *sb* then stores a **strong reference** to this object. What this means for the garbage collector is that the particular *StringBuilder* object is not eligible for collection at all due to a strong reference held to it by *sb*. The story only changes when we nullify *sb* like this:

sb = null;

After calling the above line, the object will then be eligible for collection.

We can change this relationship between the object and the garbage collector by explicitly wrapping it inside another reference object which is located inside *java.lang.ref* package.

A **soft reference** can be created to the above object like this:

StringBuilder sb = new StringBuilder();

SoftReference<StringBuilder> sbRef = new SoftReference<>(sb);

sb = null;

In the above snippet, we have created two references to the *StringBuilder* object. The first line creates a **strong reference** *sb* and the second line creates a **soft reference** *sbRef*. The third line should make the object eligible for collection but the garbage collector will postpone collecting it because of *sbRef*.

The story will only change when memory becomes tight and the JVM is on the brink of throwing an *OutOfMemory* error. In other words, objects with only soft references are collected as a last resort to recover memory.

A **weak reference** can be created in a similar manner using *WeakReference* class. When *sb* is set to null and the *StringBuilder* object only has a weak reference, the JVM's garbage collector will have absolutely no compromise and immediately collect the object at the very next cycle.

A **phantom reference** is similar to a weak reference and an object with only phantom references will be collected without waiting. However, phantom references are enqueued as soon as their objects are collected. We can poll the reference queue to know exactly when the object was collected.

**AD**

**Q13. Suppose We Have a Circular Reference (Two Objects That Reference Each Other). Could Such Pair of Objects Become Eligible for Garbage Collection and Why?**

Yes, a pair of objects with a circular reference can become eligible for garbage collection. This is because of how Java's garbage collector handles circular references. It considers objects live not when they have any reference to them, but when they are reachable by navigating the object graph starting from some garbage collection root (a local variable of a live thread or a static field). If a pair of objects with a circular reference is not reachable from any root, it is considered eligible for garbage collection.

**Q14. How Are Strings Represented in Memory?**

A *String* instance in Java is an object with two fields: a *char[] value* field and an *int hash* field. The *value* field is an array of chars representing the string itself, and the *hash* field contains the *hashCode* of a string which is initialized with zero, calculated during the first *hashCode()* call and cached ever since. As a curious edge case, if a *hashCode* of a string has a zero value, it has to be recalculated each time the *hashCode()* is called.

Important thing is that a *String* instance is immutable: you can't get or modify the underlying *char[]* array. Another feature of strings is that the static constant strings are loaded and cached in a string pool. If you have multiple identical *String* objects in your source code, they are all represented by a single instance at runtime.

**Q15. What Is a Stringbuilder and What Are Its Use Cases? What Is the Difference Between Appending a String to a Stringbuilder and Concatenating Two Strings with a + Operator? How Does Stringbuilder Differ from Stringbuffer?**

*StringBuilder* allows manipulating character sequences by appending, deleting and inserting characters and strings. This is a mutable data structure, as opposed to the *String* class which is immutable.

When concatenating two *String* instances, a new object is created, and strings are copied. This could bring a huge garbage collector overhead if we need to create or modify a string in a loop. *StringBuilder* allows handling string manipulations much more efficiently.

*StringBuffer* is different from *StringBuilder* in that it is thread-safe. If you need to manipulate a string in a single thread, use *StringBuilder* instead.

**System Design**

**A screenshot of a computer

Description automatically generated with medium confidenceA picture containing text, diagram, plan, line

Description automatically generatedA diagram of a load balancer

Description automatically generated with medium confidenceA diagram of a database

Description automatically generated with low confidence**