Memory Management & Optimization

Garbage Collection

Garbage collection is an **automatic memory management** technique where the system reclaims memory occupied by objects that are no longer in use, preventing memory leaks. It relieves the programmer from doing **manual memory management**, where the programming specifies what objects to do de-allocate and return to the memory system and when to do so. Similar **automatic memory management** techniques include: - Stack allocation - Region inference - Memory ownership

Advantages: GC frees programmers from manually deallocating the memory, to prevent some errors: - **Dangling pointers**, which occur when a piece of memory is freed but there are still pointers to it, and one of those pointers is dereferenced. By then the memory may have been reassigned to another use, with unpredictable results. - **Double free bugs**, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again. - Certain kind of **memory leaks**, in which a program fails to free memory occupied by objects that have become unreachable, causing memory exhaustion

Disadvantages: Garbage collection may take a significant proportion of a program's total processing time, and affect performance as a result. The reason is because it consumes the **CPU cycles** to execute the garbage collector's code, that could otherwise be used to run the application's code. The work includes: - **Identifying live objects:** Determining which objects in memory are still being referenced by the program. - **Identifying dead objects:** Identifying objects that are no longer referenced and are candidates for collection. - **Reclaiming memory:** Freeing up the memory occupied by dead objects. - **Potentially compacting memory:** Moving live objects around to reduce fragmentation and improve allocation speed (in some GC algorithms).

As GC uses computer resources to decide which memory to free, the penalty for the convenience of not annotating object lifetime manually in the source code is **overhead**, which impairs system performance. A peer-reviewed paper from 2005 concluded that GC needs five times the memory to compensate for this overhead and to perform as fast as the same program using idealized explicit memory management.

The comparison however is made to a program generated by deallocation calls using a **oracle**, implementing by collecting traces from programs run under a profiler, and the program is only correct for one particular execution of the program. Interaction with memory hierarchy effects can make this overhead intolerable in circumstances that are hard to predict or to detect in routine testing.

Resources other than memory, such as network sockets, database handles, windows, file descriptors, and device descriptors, are not typically handled by garbage collection, but rather by other methods (e.g. destructors). Some such methods de-allocate memory also.

Common Algorithms / Strategies :

Tracing: Most commonly seen GC method, overall strategy consists of determining which objects should be garbage collected by tracing which objects are *reachable* by a chain of references from certain root objects, and considering the rest as garbage and collecting them.

Reachability of an object: Informally, an object is reachable if it is referenced by at least one variable in the program, either directly or through references from other reachable objects: 1. **A distinguished set of roots:** objects that are assumed to be reachable. Typically these include all the objects referenced anywhere from the call stack (local variables, parameters in the functions being invoked), and any global variables. 2. Anything referenced from a reachable object itself is reachable, more formally, reachability is a **transitive closure**.

The reachability definition of "garbage" is not optimal, insofar as the last time a program uses an object could be long before that object falls out of the environment scope. A distinction is sometimes drawn between **syntactic garbage**, those objects the program cannot possibly reach, and **semantic garbage** , those objects the program will in fact never again use. For example:

```
Object x = new Foo();
Object y = new Bar();
x = new Quux();

/* At this point, we know that the Foo object
 * originally assigned to x will never be
 * accessed: it is syntactic garbage.
 */

/* In the following block, y *could* be semantic garbage;
 * but we won't know until x.check_something() returns
 * some value -- if it returns at all.
 */
```

```
if (x.check_something()) {
    x.do_something(y);
}
System.exit(0);
```

The problem of precisely identifying semantic garbage can easily be shown to be partially decidable: a program that allocates an object X , runs an arbitrary input program P, and uses X if and only if P finishes would require a semantic garbage collector to solve the halting problem. Although conservative heuristic methods for semantic garbage detection remain an active research area, essentially all practical garbage collectors focus on syntactic garbage.

Strong and weak references: The garbage collector can reclaim only objects that have no references pointing to them either directly or indirectly from the root set. However, some programs require **weak references**, which should be usable for as long as the object exists but should not prolong its lifetime. In discussions about weak references, ordinary references are sometimes called **strong references**. An object is eligible for garbage collection if there are no strong (i.e ordinary) references to it, even though there might be some weak references to it.

A weak reference is not merely just any pointer to the object that a garbage collector does not care about. The term is usually reserved for properly managed category of special reference objects which are safe to use even after the object disappears because they *lapse* to a safe value (usually `null`). An unsafe reference that is not known to the garbage collector will simply remain dangling by continuing to refer to the address where the object previously resided.

Tracing Algorithms: Tracing collectors are so called because they trace through the working set of memory, and performs collection in cycles. It is common for cycles to be triggered when there is not enough free memory for the memory manager to satisfy an allocation request, but cycles can often be requested by the mutator directly or run on a time schedule. - **Mark and Sweep**: Traverse object graph → Mark reachable → Sweep unmarked. In the naive mark-and-sweep method, each object in memory has a flag (typically a single bit) reserved for garbage collection use only. This flag is always *cleared*, except during the collection cycle.

1. **Mark Phase:** When an object is created, its mark bit is set to 0 (false bit). In the Mark phase, we set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true). We would do a graph traversal for this operation, like DFS would work for us. Here we can consider every object as a node and then all the nodes (objects) that are reachable from this node (object) are visited and it goes on till we have visited all the reachable nodes.
    - The root is a variable that refers to an object and is directly accessible by a local variable.

- We can access the mark bit for an object by `markedBit(obj)`. **Algorithm**: Mark phase

```
Mark(root)
If markedBit(root) = false then
               markedBit(root) = true
                                    For each v
referenced by root
                                     Mark(v)
```

2. **Sweep Phase**: After mark phase, we would sweep all the unreachable objects (i.e it clears the heap memory for all the unreachable objects.) All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to true. Now the mark value for all the reachable objects is set to false since we will run the algorithm (if required) and again we will go through the mark phase to mark all the reachable objects. **Algorithm**: Sweep phase

```
Sweep()
For each object p in heap
If markedBit(p) = true then
               markedBit(p) = false
                             else
                                 heap.release(p)
```

Advantages: - It handles the case with cyclic references, even in the case of a cycle, this algorithm never ends in an infinite loop - No additional overheads incurred during execution

Disadvantages: - Normal program execution is suspended while the garbage collection runs (consumed by CPU cycles to run GC - After the algorithm is being run several times, reachable objects end up being separated by many, small unused memory regions. `Heap memory []. [ ].. [ ]. [ ]... []` Here the dots represents the free memory, while the bracket blocks represent the memory taken by all the reachable objects.

Now the free segments (dots) are of varying sizes let's say the 5 free segments are of size 1, 1, 2, 3, 5(size in units). Now we need to create an object which takes 10 units of memory, now assuming that memory can be allocated only in the **contiguous** form on blocks, the creation of an object is not possible although we have an available memory space of 12 units, and it will cause **OutOfMemory** error.

This problem is termed **Fragmentation**. We have memory available in "fragments" but we are unable to utilize that memory space. We can reduce the fragmentation by compaction; we shuffle the memory content to place all the memory blocks together to form one large block. Now consider the above example, after compaction we have a contiguous block of free memory size 12 units, so we can now allocate memory to an object size of 10 units.

Deterministic: Java's GC implementation, even when using a single-threaded GC does not make it perfectly deterministic in all aspects: 1. **Trigging Heuristics**: The fundamental decision of when to trigger a garbage collection cycle still relies on the JVM's internal heuristics, which is influenced by the application's behavior and memory usage patterns. Therefore an exact STW pause is not entirely predictable. 2. **Heap State**: The duration of a GC cycle will depend on the amount of live and dead objects in the heap at the time of collection. This heap state is influenced by the application's allocation and object lifecycle, which can vary.

Variants of mark-and-sweep:

Serial GC: A simple, single-threaded mark-and-sweep GC that operates in a stop-the-world manner, which pauses the application's execution during garbage collection, and also does compaction to reduce fragmentation after. It uses the same logic with mark-and sweep, with a single thread approach: - **Mark Phase**: Single thread traverses roots and marks all objects directly referenced by the application's active threads to identify all reachable objects. - **Sweep Phase**: Once the marking is complete, the algorithm single threadly sweeps the memory regions, deallocating memory occupied by unreferenced objects. - **Compaction**: Compacts the memory by moving live objects together after sweep, creating a contiguous free memory space. - **Free Memory**: After compaction, the algorithm updates the free memory space pointer to indicate the new location for object allocation.

Pros: 1. **Simplicity**: The Serial GC is straightforward to implement and understand, making it an ideal choice for beginners or smaller applications. 2. **Low Overhead**: The algorithm has lower CPU and memory overhead compared to more complex GC algorithms, making it suitable for resource-constrained environments. 3. **Predictable Pauses**: As a stop-the-world collector, the Serial GC provides predictable pauses during garbage collection, allowing for better control over the application behavior. 4. **Single-threaded Execution:** The Serial GC performs garbage collection using a single thread, ensuring more **determinism** and avoiding potential multi-threading issues. (Note: Java's implementation of GC is not deterministic)

Cons: 1. **Longer Pause Times:** The stop-the-world nature of the Serial GC can result in longer pause times, causing interruptions in application responsiveness for larger heaps or memory-intensive applications. 2. **Limited Scalability:** The single-threaded nature of the Serial GC limits its scalability on modern multi-core processors, where parallelism can significantly improve GC performance. 3. **Not Suitable for Large Applications:** The Serial GC may not be the optimal choice for large-scale applications or systems with high memory requirements, as its performance may degrade due to increased garbage collection times.

Parallel GC: Also uses mark-and-sweep, but taking advantage of multi-core processors and improving GC performance by parallelizing certain tasks, such that the parallel GC threads make collection faster, and wait lesser time during Stop-the-World period. - **Mark phase**: Similar to Serial GC, the algorithm identifies and marks all objects directly referenced by the application's active threads. This marking process occurs while the application is paused

momentarily. - **Concurrent Marking**: While the application is running, the algo concurrently marks for reachable objects in memory by utilizing multiple threads. his concurrency marking phase reduces pause time for garbage collection. - **Remark**: After the concurrent marking phase, the algorithm allows the application to continue running while it performs a final marking process to account for objects that may have become reachable during the concurrent phase. - **Concurrent Sweep and Compaction**: The Parallel GC concurrently sweeps and deallocates memory regions occupied by unreferenced objects. Simultaneously, it compacts the memory, moving live objects together to create contiguous free memory space. This concurrent sweep and compaction phase helps reduce the pause time further. **Pros:** 1. **Improved Throughput**: Parallel GC leverages multiple threads to execute garbage collection tasks concurrently, leading to improved throughput by utilizing the processing power of multi-core processors. 2. **Reduced Pause Times**: By parallelizing the marking, sweeping, and compaction phases, the Parallel GC minimizes the pause times required for garbage collection, resulting in improved application responsiveness. 3. **Scalability**: The algorithm's ability to utilize multiple threads makes it highly scalable, allowing it to handle larger heaps and memory-intensive applications more efficiently. 4. **Enhanced Performance**: Parallel GC performs well in scenarios where the application generates a significant amount of garbage, as it can leverage parallelism to expedite garbage collection and keep up with memory demands.

Cons: 1. **Increased CPU Utilization**: The Parallel GC algorithm utilizes multiple threads, leading to higher CPU utilization compared to single-threaded garbage collection algorithms. This increased utilization may affect the overall system performance for applications with limited CPU resources. 2. **Longer Individual Pause Times**: While the overall pause time may be reduced, the individual pause times for each garbage collection cycle might be longer compared to other algorithms. This aspect may impact the responsiveness of time-sensitive applications. 3. **Not Suitable for Small Systems**: The Parallel GC algorithm's multi-threaded nature and increased resource utilization make it less suitable for small-scale systems with limited resources or single-core processors. 4. **Non-deterministic**: The exact scheduling timing of parallel threads by the OS are not deterministic, as they are determined by the OS scheduler to decide how CPU cores are allocated to these threads, and this can be influenced by other processes running on the system. Hence slight variations in the duration and different timing of the GC phases.

CMS (Concurrent Mark-and-Sweep): A popular GC algorithm used in Java, aims to minimize application pauses by running certain phases concurrently with the application threads. - **Initial Mark**: Pauses the application briefly to identify objects directly reachable from the root set. - **Concurrent mark**: Concurrently traverses object graphs, marking objects that are still in use. - **Concurrent Preclean**: Continues marking objects concurrently while accounting for changes made during concurrent marking. - **Final Remark**: Pauses the application again to identify objects modified during concurrent marking and completes marking. - **Concurrent Sweep**: Concurrently reclaims memory by sweeping through and freeing unused objects. - **Concurrent Reset**: Concurrently resets internal data structures and prepares for the next garbage collection cycle.

The initial and final mark causes short pause to analyze root objects and marking reachable objects, while the concurrent marking occurs alongside application threads, identifying additional reachable objects. The concurrent preclean phase handles any object modifications during concurrent marking, then concurrent sweeping frees up memory by reclaim unused object while the application continues running. Lastly, concurrent reset prepares the garbage collector for the next cycle, ensuring it is ready for future garbage collection.

Pros: 1. **Reduced Pause Times:** CMS aims to minimize pauses by performing garbage collection concurrently with the application, resulting in better application responsiveness. 2. **Improved Scalability:** CMS is well-suited for large applications with high thread concurrency, as it strives to run concurrently with application threads. 3. **Effective for Mixed Workloads**: CMS performs well in scenarios where the application generates a significant amount of short-lived objects.

Cons: 1. **Increased CPU Utilization**: Concurrent execution can lead to higher CPU usage due to the additional threads involved in garbage collection. 2. **Fragmentation Concerns**: CMS may suffer from memory fragmentation issues, leading to less efficient memory utilization. 3. **Limited Pause Reduction**: While CMS reduces pauses compared to other algorithms, it may not eliminate pauses entirely, and long-running concurrent phases can still impact application performance. 4. **Deprecated in Java 9+**: CMS is deprecated in Java 9 and later versions, with the intention to be removed in future releases. The introduction of the Garbage-First (G1) GC aims to provide a more efficient and scalable alternative.

Problem with concurrency: In a traditional stop-the-world mark and sweep, the garbage collector has exclusive access to the object graph, but however in a concurrent GC, the application (i.e "mutator") continues to run while the GC is trying to mark live objects. This introduces a challenge: the mutator can change object references while the collector is in the middle of its work, potentially leading to incorrect identification of live vs. dead objects.

Tri-color marking: Tri-color marking is an abstraction that helps manage this concurrency by categorizing objects into three distinct states (colors) during the marking phase: 1. **White:** These objects have not yet been visited by the garbage collector in the current marking cycle. They are considered potentially garbage. 2. **Gray:** These objects have been reached by the garbage collector and need to have their children (the objects they point to) visited and processed. They are on the "to-do" list of the collector. 3. **Black:** These objects have been visited by the garbage collector, and all of their reachable children have also been processed (marked gray and then black). These objects are definitely live and will not be collected in the current cycle.

How it works: 1. **Initialization:** At the start of the marking phase, all objects in the heap are initially colored **white**. 2. **Root Set Coloring:** The garbage collector starts by identifying the root objects (e.g., objects on the stack, global variables). These roots are immediately colored **gray**. 3. **Traversal and Coloring:** The collector then picks a **gray** object, colors it **black**, and examines all the objects it points to. - If a pointed-to object is **white**, it is colored **gray** and added to the set of gray objects to be processed later. - If a

pointed-to object is already **gray** or **black**, it is left as is (no need to revisit). 4. **Iteration:** Step 3 is repeated until there are no more **gray** objects. At this point, all reachable objects from the roots have been colored **black**, and all unreachable objects remain **white**. 5. **Sweeping:** After the marking phase is complete, all **white** objects are considered garbage and can be swept (their memory reclaimed).

The Tri-Color Invariant and Write Barriers: To ensure correctness in a concurrent environment, tri-color marking algorithms need to maintain a crucial invariant: **no black object should point to a white object.** If this invariant is violated, a live white object might become unreachable by the collector and incorrectly swept.

To maintain this invariant while the mutator is running, concurrent tri-color marking GCs often employ **write barriers**. A write barrier is a small piece of code that is executed whenever the mutator writes a pointer (i.e., changes an object's reference to another object). The write barrier's job is to ensure the tri-color invariant is preserved. Common strategies for write barriers include:

1. **Dijkstra's Write Barrier:** If a black object is about to point to a white object, the white object is immediately colored gray. This ensures the collector will eventually visit it.
2. **Yuasa's Write Barrier:** When a pointer from a black object to any other object is overwritten, the target object of the overwritten pointer is colored gray. This ensures that objects potentially disconnected from the live set are revisited.

Link for detail read: http://www.yuasa.kuis.kyoto-u.ac.jp/~yuasa/language-system/RTGC.pdf

Implementation in Golang:

```
package main

import (
"fmt"
"runtime"
"sync"
"time"
)

// Object represents an object that needs to be garbage collected.
type Object struct {
mu sync.Mutex // Mutex for synchronization
Marked int // Marked indicates the color of the object.
Next *Object // Next represents a reference to another object.
}
```

```
// Constants for colors used in tricolor marking.
const (
white = iota // White objects are unmarked and unreachable.
gray
black // Black objects are marked and reachable.
)

// Global variables for managing the heap and objects.
var (
heap []*Object // Simulated heap.
rootSet []*Object // Root set of objects (global variables).
mu sync.Mutex // Mutex to protect global data structures.
)

func main() {
// Initialize the heap and objects.
initHeap()

// Construct an object graph.
obj1 := &Object{}
obj2 := &Object{}
obj3 := &Object{}

obj1.Next = obj2
obj2.Next = obj3
rootSet = []*Object{obj1} // Set the root objects.

// Simulate the tricolor mark-and-sweep garbage collection cycle.
for i := 0; i < 3; i++ {
gcCycle()
fmt.Printf("After GC cycle %d:\n", i+1)
printObjectStatus()
time.Sleep(1 * time.Second)
}
}

// initHeap initializes the heap with objects.
func initHeap() {
    // Initialize the heap with objects.
    heap = make([]*Object, 100)

    // Populate the heap with objects.
```

```
    for i := 0; i < 100; i++ {
    heap[i] = &Object{}
    }
}

// gcCycle simulates a complete tricolor mark-and-sweep garbage collection cycle.
func gcCycle() {
    // Step 1: Initialization - Reset the color of all objects to white.
    resetColors()

    // Step 2: Initial Marking - Mark objects directly reachable from the root set as gray.
    initialMark()

    // Step 3: Concurrent Marking - Mark other objects as gray concurrently.
    concurrentMark()

    // Step 4: Termination of Concurrent Marking - No additional work needed.
    // In our simplified example, the termination of concurrent marking is implicitly
    // handled by the fact that goroutines for marking have finished their work
    // when they mark all reachable objects as black.

    // Step 5: Sweeping - Deallocate memory for unreachable (white) objects.
    sweep()

    // Step 6: Reclamation - No additional work needed.
    // In our simplified example, we don't explicitly implement a reclamation phase
    // because it's typically handled by the Go runtime and not something that
    // application-level code would manage directly. The Go runtime is responsible
    // for efficiently managing memory and returning it to the operating system
    // when appropriate.
}

// resetColors resets the color of all objects to white.
func resetColors() {
    mu.Lock()
    defer mu.Unlock()

    for i := range heap {
    heap[i].Marked = white
    }
}
```

```
// initialMark marks objects directly reachable from the root set as gray.
func initialMark() {
    mu.Lock()
    defer mu.Unlock()

    for _, obj := range rootSet {
    obj.Marked = gray
    }
}

// concurrentMark marks objects as gray concurrently.
func concurrentMark() {
    var wg sync.WaitGroup

    mu.Lock()
    for _, obj := range rootSet {
        if obj.Marked == gray {
        // Start a goroutine to mark objects from the gray root set.
        wg.Add(1)
        go func(o *Object) {
        defer wg.Done()
        mark(o)
        }(obj)
    }
}
mu.Unlock()

// Wait for all goroutines to finish.
wg.Wait()
}

// mark recursively marks reachable objects using depth-first search.
func mark(obj *Object) {
    if obj == nil || obj.Marked == black {
    return
    }

    // Mark the current object as reachable.
    obj.Marked = gray

    // Recursively mark its references.
    mark(obj.Next)
```

```
    // After marking references, mark the object as black.
    obj.Marked = black
}

// sweep deallocates memory for unreachable (white) objects.
func sweep() {
    mu.Lock()
    defer mu.Unlock()

    for i := range heap {
        obj := heap[i]
        if obj != nil && obj.Marked == white {
        // Free the unmarked (white) object.
        heap[i] = nil
        }
    }
}

// printObjectStatus prints the status of each object (black, gray, or white).
func printObjectStatus() {
    mu.Lock()
    defer mu.Unlock()

    for i, obj := range heap {
        if obj == nil {
        fmt.Printf("Object %d: Freed\n", i)
        } else if obj.Marked == black {
        fmt.Printf("Object %d: Marked (Black)\n", i)
        } else if obj.Marked == gray {
        fmt.Printf("Object %d: Marked (Gray)\n", i)
        } else {
        fmt.Printf("Object %d: Unmarked (White)\n", i)
        }
    }
}
```

How root objects are decided ?
In Go's garbage collector, root objects are objects that are directly reachable from outside the garbage collection process. Identifying root objects is essential because they serve as the starting point for the garbage collection process. Root objects typically include:

1. **Global Variables**: Global variables in your Go program can reference objects. These

global variables are considered root objects because they can be directly accessed by the garbage collector.

2. **Goroutine Stacks**: Active goroutine stacks can also contain references to objects. Since goroutines are concurrently executing, their stacks are considered root sets. The garbage collector scans goroutine stacks to identify additional root objects.

To identify root objects, We don't need to do anything special in our Go code. The Go runtime and garbage collector handle this automatically. However, if we want to analyze or understand which objects are root objects, we can use tools like Go's built-in pprof package for memory profiling.

Here's how you can use pprof to identify root objects:

1. Import the `net/http/pprof` package:

```
import _ "net/http/pprof"
```

2. Add an HTTP server to your program to expose profiling endpoints:

```
go func() {
log.Println(http.ListenAndServe("localhost:6060", nil))
}()

//This starts an HTTP server on port 6060.
```

3. Start your program and access the profiling endpoint in a web browser or use a tool like `go tool pprof` from the command line:

```
go tool pprof http://localhost:6060/debug/pprof/heap

//This will provide a heap profile that includes information about root objects and their re
```

4. Analyze the heap profile to understand which objects are root objects and what references they hold. This can help you identify potential memory leaks or objects that are being held longer than expected.

Generational GC: Divide heap into young (new objects) and old generations (long-lived objects). New objects are collected more often (cheaper). It has been empirically observed that in many programs, the most recently created objects are also those most likely to become unreachable quickly (**weak generational hypothesis**). Only a small fraction of objects tend to live longer.

To exploit this behavior, generational GC divides the heap (the area of memory where objects are allocated) into **multiple generations**, typically at least two: - **Young Generation (Nursery):** This is where all new objects are initially allocated. Because most objects die young, this generation is designed to be collected frequently and efficiently. It's often further divided into: - **Eden Space:** Where most new objects are created. - **Survivor Spaces (S0 and S1):** Used to hold objects that survive a young generation collection. Objects move between survivor spaces and eventually to the old generation if they survive enough collections. - **Old Generation (Tenured Generation):** Objects that have survived multiple young generation collections are promoted (moved) to the old generation. Objects in this generation are assumed to have longer lifespans and are therefore collected less frequently.

Non-Heap Memory: - **Code Cache:** This area stores compiled native code. When the JVM interprets bytecode, frequently executed parts are compiled into optimized machine code for faster execution. This compiled code is stored in the Code Cache. - **Thread Stacks:** Each thread in a Java application has its own private stack. These stacks store information about the thread's execution, including local variables, method call history, and parameters. While technically part of the JVM's memory, the allocation and management of thread stacks are often handled differently from the heap. - **Native Memory:** This encompasses memory allocated directly by native code (e.g., through JNI calls) or by the JVM for its internal operations that aren't categorized above. This can include memory used for I/O buffers, direct byte buffers (off-heap), and other internal structures.

Metaspace: A storage area for class metadata in Java 8 and later. It's a significant change from the older Permanent Generation (PermGen) in several ways: - **Out of the Heap:** Unlike PermGen, which was a fixed-size part of the heap, Metaspace is allocated **outside of the main Java heap**, in native memory. This means it's not directly affected by the `-Xmx` and `-Xms` heap size settings. - **Dynamically Resized:** Metaspace can grow dynamically as the application loads more classes. The JVM will allocate more native memory for Metaspace as needed, up to a limit you can configure using the `-XX:MaxMetaspaceSize` JVM argument. If this limit is not set, Metaspace can theoretically grow until all available system memory is exhausted (though the OS would likely intervene before that). - **Garbage Collection:** While Metaspace is outside the main heap, the metadata stored in it can still become garbage (e.g., when a classloader is no longer active and its loaded classes are no longer referenced). The JVM performs garbage collection on Metaspace to reclaim this unused metadata. This Metaspace GC can be triggered independently or as part of a full GC. - **No More `OutOfMemoryError: PermGen space`:** One of the most common `OutOfMemoryError` exceptions in older Java versions was due to the fixed size of PermGen being exhausted. Metaspace's dynamic resizing and allocation in native memory have largely eliminated this specific error. Now, if you run out of Metaspace, you'll typically see `OutOfMemoryError: Metaspace`.

How Generational GC works: 1. **New Object Allocation:** All new objects are allocated in the young generation (typically in the Eden space). Allocation in the young generation is usually very fast. 2. **Young Generation Collection (Minor GC):** When the young generation fills up, a minor garbage collection is triggered. This collection is usually fast

because it only needs to examine a relatively small portion of the heap, and most objects in the young generation are likely to be garbage. - Live objects in Eden and one of the Survivor spaces are copied to the other Survivor space. - Objects that have survived a certain number of minor GCs are promoted to the old generation. - The Eden and the "from" Survivor space are then cleared. 3. **Promotion:** Objects that survive multiple minor GC cycles in the survivor spaces are eventually moved (promoted) to the old generation. The number of survival cycles before promotion is typically configurable. 4. **Old Generation Collection (Major GC or Full GC):** When the old generation fills up, a major garbage collection (or sometimes a full GC, which might involve collecting the entire heap) is performed. This collection is typically much slower and can result in longer pause times because it involves examining a larger portion of the heap with potentially many long-lived objects.

Why is Generational GC Effective? - Focus on the Common Case: By focusing frequent and efficient collection on the young generation where most objects die quickly, the overall cost of garbage collection is significantly reduced. - R**educed Scoping:** Minor GCs only need to examine a small part of the heap, making them faster than collections that scan the entire heap. - **Less Frequent Full GCs:** By promoting long-lived objects to the old generation, the frequency of expensive major GCs is reduced.

Reference Counting: Objects are deallocated when their reference count drops to zero (issue: circular references). It is a a memory management technique where each object in memory keeps track of how many references (pointers or variables) are currently pointing to it. This count is called the **reference count**.

How it works: 1. **Object Creation:** When an object is created, its reference count is initialized to 1 (because there's at least one reference pointing to it, the one that was just created). 2. **Reference Assignment:** - When a new reference is made to an existing object (e.g., assigning a variable to point to the same object), the object's reference count is incremented. - When a reference to an object is removed (e.g., a variable goes out of scope, is assigned a new value, or is explicitly set to `null`), the object's reference count is decremented. 3. **Garbage Collection (Reclamation):** When an object's reference count drops to zero, it means that no part of the program holds a reference to it anymore. At this point, the object is considered garbage and can be safely deallocated (its memory can be freed). The deallocation often happens immediately when the reference count reaches zero.

Pros: 1. **Immediate Reclamation:** Memory for an object is reclaimed as soon as its reference count becomes zero. This can lead to more predictable memory usage and potentially reduce the need for long garbage collection pauses. 2. **Simplicity:** The concept is relatively straightforward to understand and implement. 3. **Locality of Reclamation:** The work of garbage collection is often distributed throughout the program's execution as references are added and removed, potentially improving cache locality compared to stop-the-world GC. 4.

Predictable Timing of Count Changes: The incrementing and decrementing of reference counts are directly linked to specific program actions: creating a new reference or an existing reference going out of scope or being overwritten. These actions are deterministic points in the program's execution.

Cons: 1. **Cycle Detection Problem:** This is the most significant drawback. If a group of objects forms a cycle of references where no external reference points into the cycle, their reference counts will never drop to zero, even though they are no longer reachable by the program. This leads to memory leaks. For example: - Object A has a reference to Object B. - Object B has a reference to Object A. - If the only references to A and B are within this cycle, their reference counts will be at least 1, and they will never be deallocated by pure reference counting. 2. **Overhead of Maintaining Counts:** Incrementing and decrementing reference counts every time a reference is created or destroyed can introduce performance overhead. This overhead can be significant in programs with many reference manipulations. 3. **Complexity in Concurrent Environments:** Managing reference counts correctly in a multi-threaded environment requires careful synchronization (e.g., using **atomic operations**) to avoid race conditions, which can add complexity and potential performance bottlenecks. 4. **Space Overhead:** Each object needs to store its reference count, which adds a small amount of memory overhead to every object.

How the Cycle Problem is Addressed (in systems that use reference counting): Languages that primarily use reference counting often employ additional mechanisms to detect and break reference cycles. For example, Python uses a separate garbage collector that periodically scans for and breaks cycles of unreachable objects.