In-depth Exploration of Garbage Collector (GC)

Ben Khalfallah Héla

#garbage-collector

Hello !

Bio

Ben Khalfallah Héla
Senior Frontend Expert

I love:

- Software Engineering Practices
- Architecture, Clean Code & Craft
- Sharing
- Data Structures
- Graph Algorithm
- Linking Hardware and Software
- Creating tools

“There are no solutions. There are **only trade-offs** .”

—Thomas Sowell

01

Mark-Sweep

Operating principle, algorithm and real use cases

02

Copying

Operating principle, algorithm and real use cases

03

Mark-Compact

Operating principle, algorithm and real use cases

04

Generational

Operating principle, algorithm and real use cases

05

G1 & Z

Operating principle, algorithm and real use cases

06

Experimental GC

Epsilon & Shenandoah

01

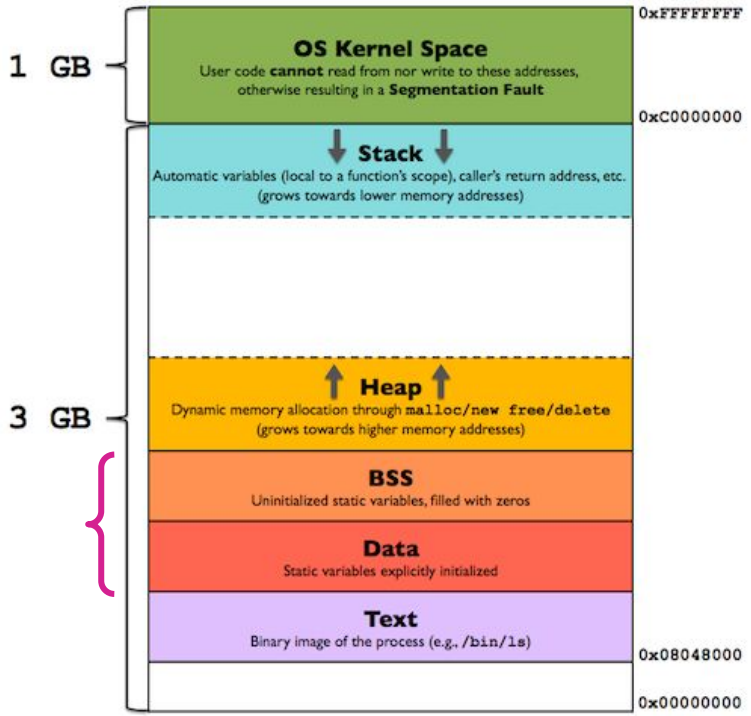
Why GC?

A tidbit of history

Running Program's Memory

Static Memory = BSS segment + Data segment

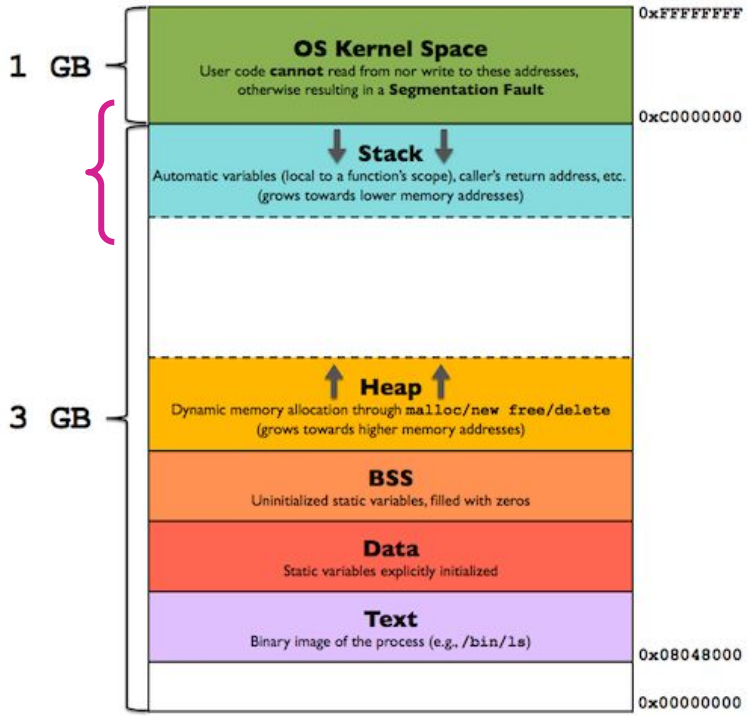
- **BSS**: Uninitialized or zero-initialized static data.
- **Data**: Explicitly initialized static data (with non-zero values).
- **Static Memory** : Exists for the entire duration of the program.



Running Program's Memory

Stack Memory:

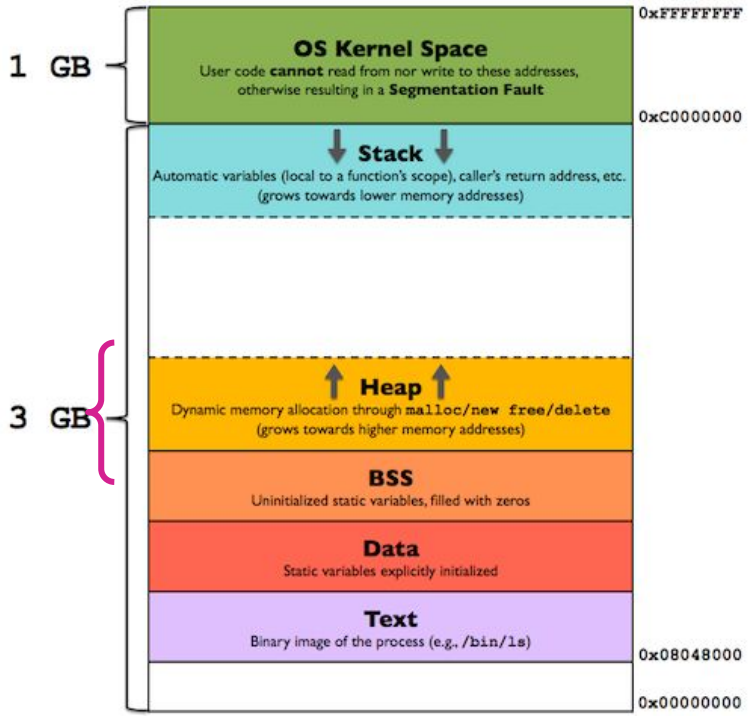
- **Automatically** managed **temporary** storage for function calls.
- Holds **local variables, parameters, and return addresses** .
- Organized in **stack frames** , one per function call.
- **LIFO** (Last-In, First-Out) data structure.
- **Limited capacity** (risk of stack overflow).
- **Non-resizable** memory allocations.

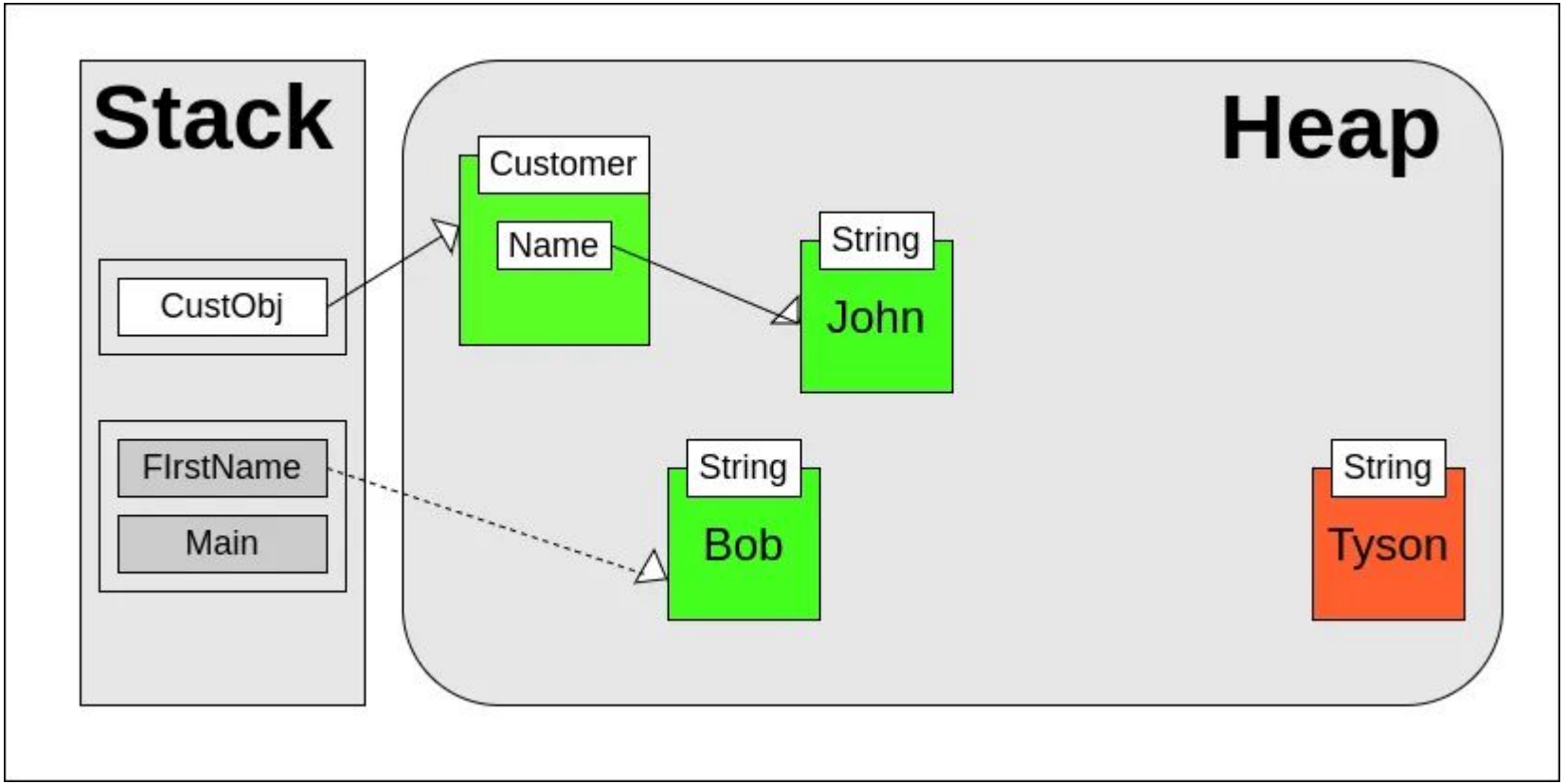


Running Program's Memory

Heap Memory:

- Dynamically allocated memory requiring **explicit manual management** (e.g., **malloc**, **free**).
- Accessed via **references** (**pointers**).
- Used for data with **runtime-determined size**, persistent beyond function scope, or with potential **resizing** needs.



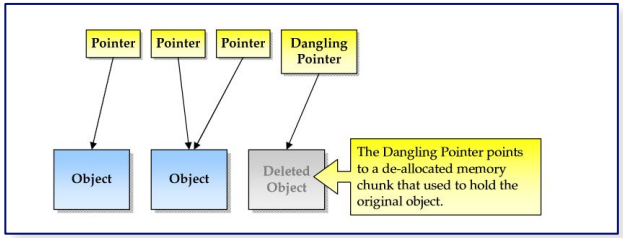


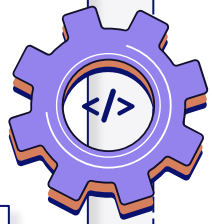
https://coderstea.in/post/java/get-ready-to-deep-dive-java-memory-management-garbage-collector/

Manual Memory Errors

Dangling Pointers

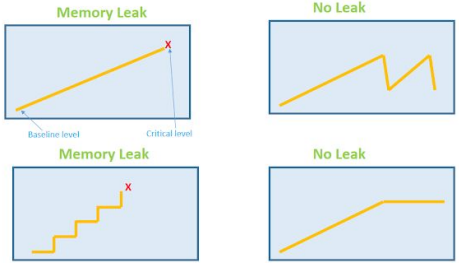
Occur when memory is deallocated (freed) while there are still active references (pointers) to it.





Memory Leaks

Happen when allocated memory is not released even when it's no longer needed.



The rise of GC

01

Centralized Management

Memory management is **centralized in a single tool** , the Garbage Collector, which is **runtime-controlled** (generally by a Virtual Machine).

Efficient Debugging and Profiling

GC provides tools and information to track memory usage and identify potential problems.

02

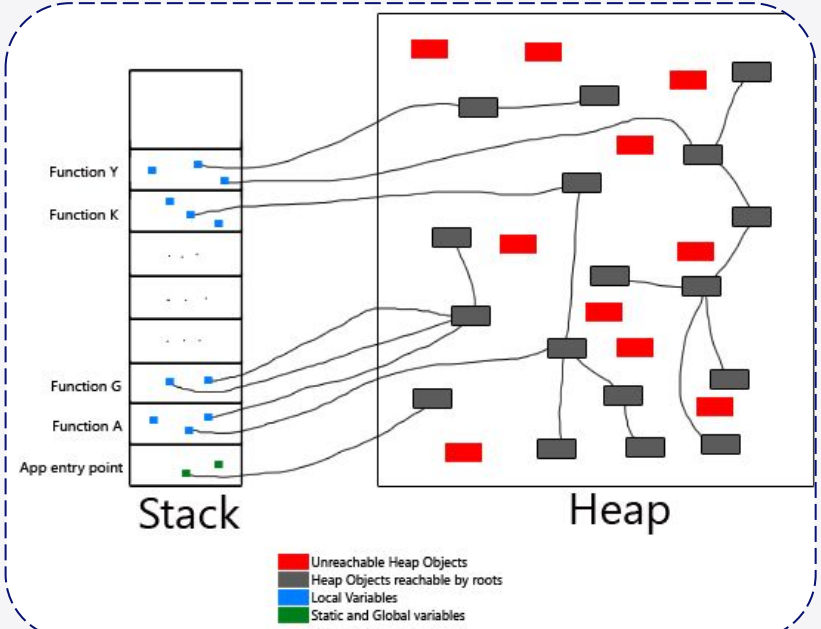
03

Scalability, Universality, and Efficiency

The **upgrade** to the Garbage Collector and the automatic memory management **algorithms** has become scalable, universal, and efficient.

GC missions

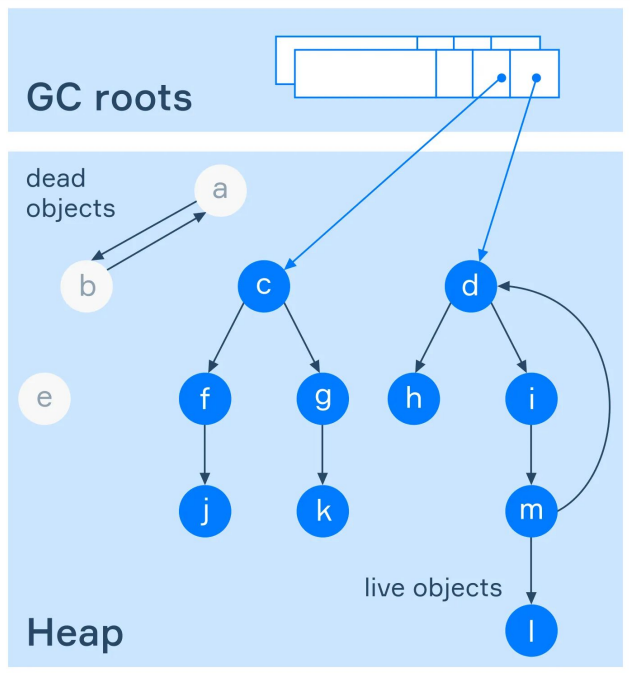
- GC **finds and frees unused** objects.
- **Unused** objects are those with **no active references** .
- No active references means **no part of the program can access the object** .
- Heap objects can be **referenced from local, global, and static variables, thread stacks, and the constant pool**.
- The garbage collector must consider all these references to accurately identify unused objects.



https://www.javacodegeeks.com/2013/06/garbage-collection-in-java-2.html

GC Roots

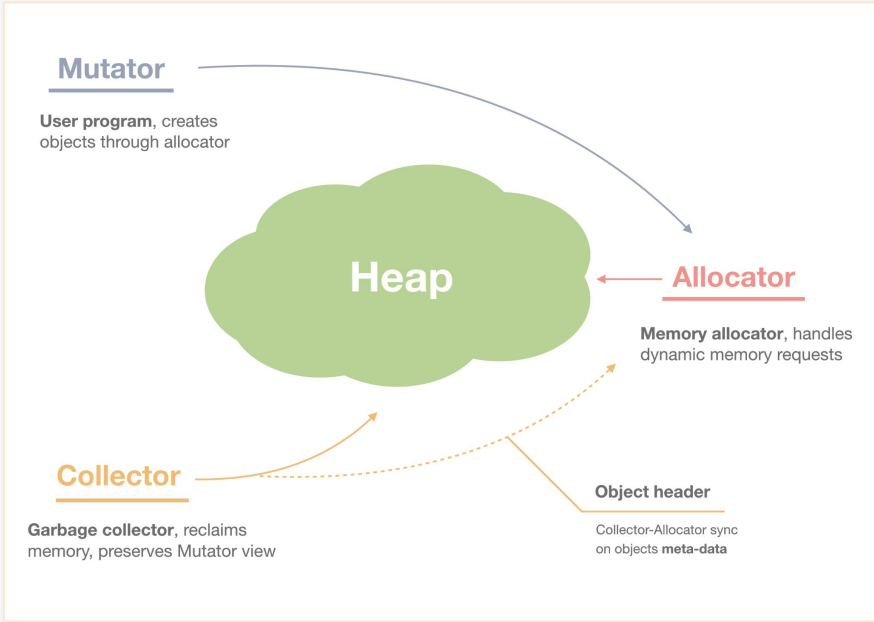
- **GC roots** are **external pointers to heap objects** .
- **Reachable from a root** = alive (not garbage).
- **Unreachable = garbage** (to be collected).



https://hyperskill.org/learn/step/17271

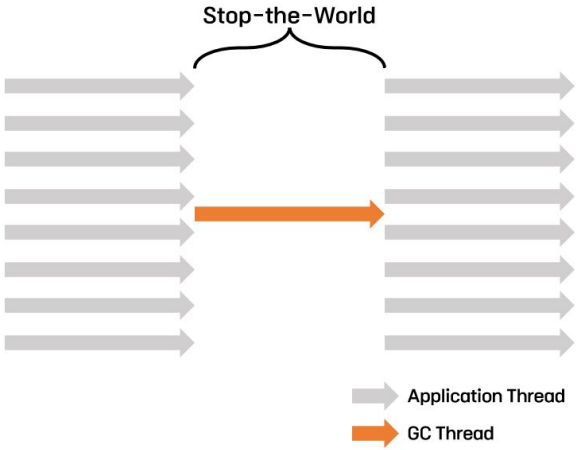
Managed memory systems

- Managed memory = **Mutator (the program)** + Garbage Collector.
- Mutator **allocates** , Garbage Collector **reclaims** .
- They work together to manage the heap.



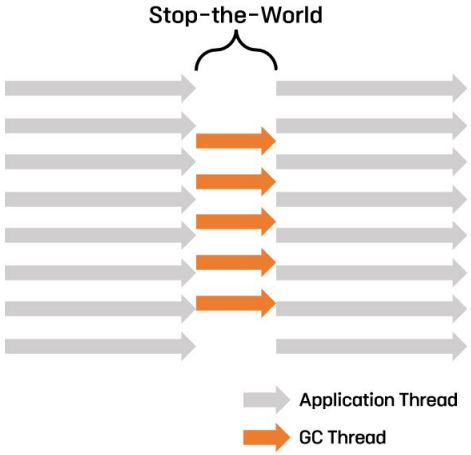
http://dmitrysoshnikov.com/compilers/writing-a-memory-allocator/

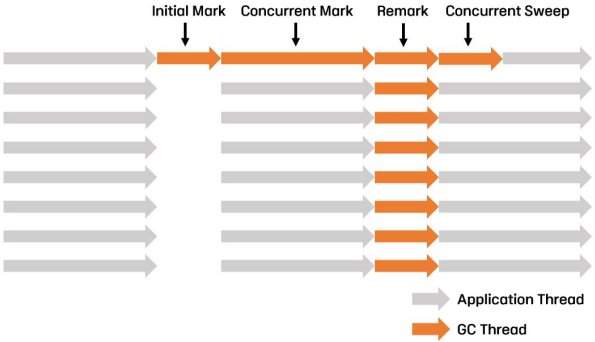
Garbage Collector Types



Serial

Parallel





Concurrent

Garbage Collector Characteristics

Fast

Minimal execution time to avoid slowing down your program.

Efficient

Optimal space usage with low memory overhead.

Responsive

Minimal pause time, crucial for real-time applications.

Locality-aware

Improves how your program accesses memory.

Scalable

Handles growing heaps and complex objects efficiently.

Algorithm

Different garbage collection algorithms have different strengths and weaknesses, impacting these traits.

02

Mark-Sweep

Operating principle, algorithm and real use cases

Operating principle and algorithm

1 — 2 — 3 — 4

stop-the-world

Pauses the program to perform garbage collection.

Prepare the root list

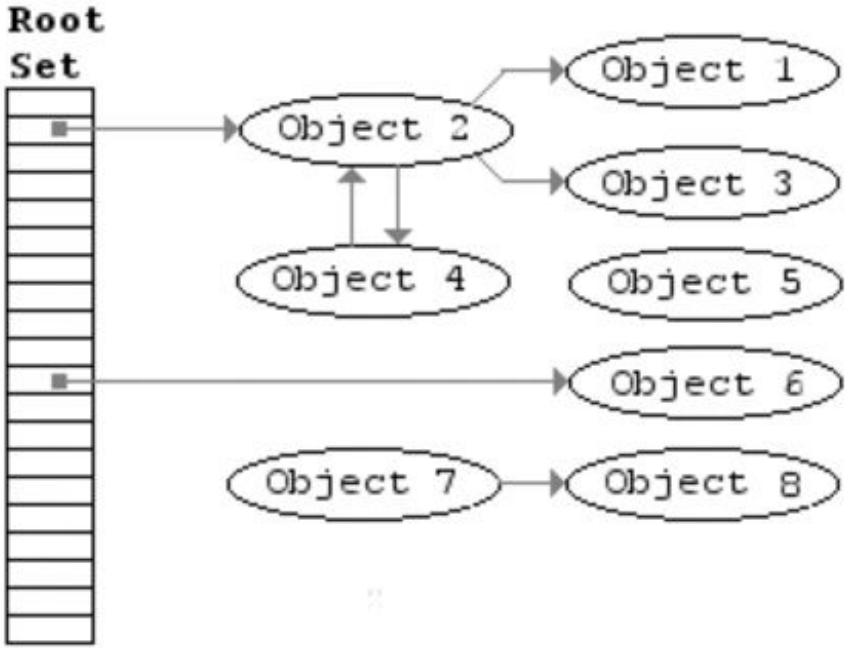
The roots could be local variables, global variables, static variables, or variables referenced in the thread stack.

Mark phase (DFS)

From those roots, it follows the connections (references) between objects, marking every object it can reach as "alive."

Sweep phase

Any object that's not marked is considered unreachable and gets swept away, freeing up that memory space.



https://en.wikipedia.org/wiki/Tracing_garbage_collection#Na%C3%AFve_mark-and-sweep

Mark-Sweep in Action

uLisp

“The type of garbage collector used in uLisp is called mark and sweep.”

Early versions of Ruby

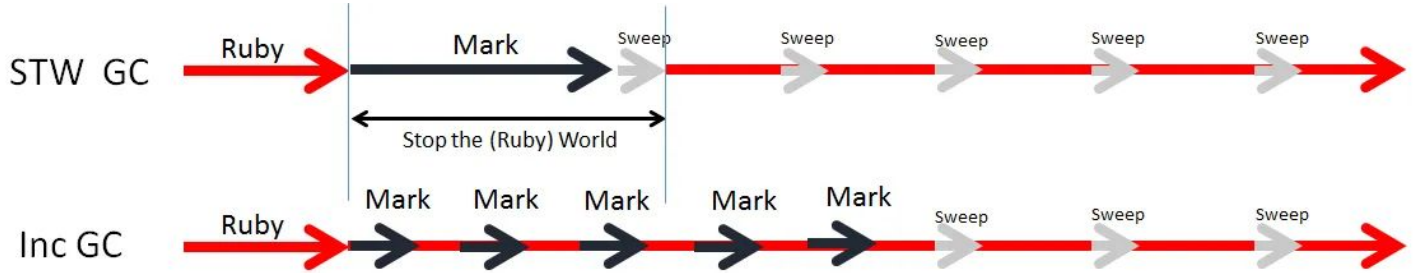
“The first version of Ruby already has a GC using Mark and Sweep (M&S) algorithm.”

Mozilla SpiderMonkey

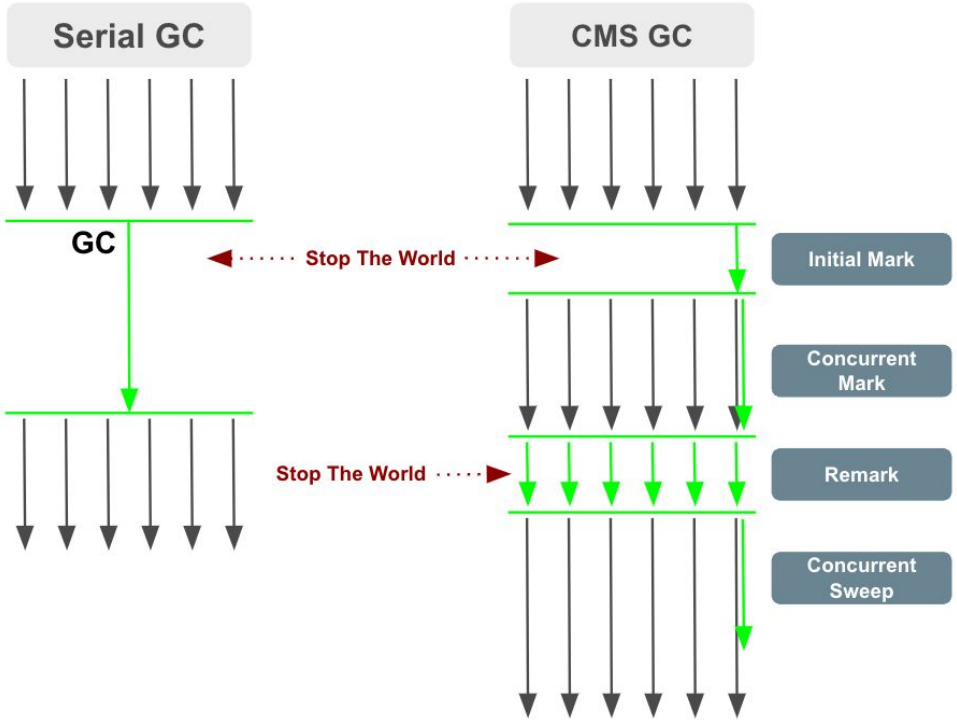
“SpiderMonkey has a mark-sweep garbage collection (GC) with incremental marking mode, generational collection, and compaction.”

Incremental Mark-Sweep

This variation, used in systems like **Ruby 2.2** and **OCaml**, breaks down garbage collection into smaller increments, interleaving them with program execution.



STW: stop the world vs. incremental marking



Concurrent Mark & Sweep vs Serial GC

Mark-Sweep Limitations

1

Costly sweep phase

Scanning the entire heap

2

"Stop-the-world" pauses

Program freezes during GC

3

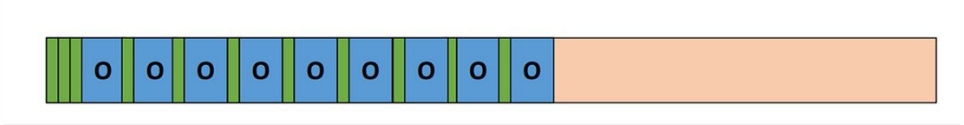
Memory fragmentation

Scattered free space

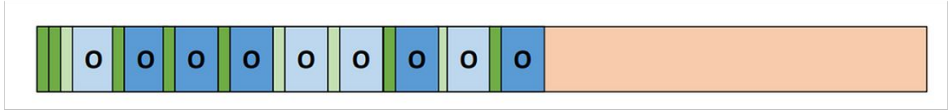
4

Not ideal for real-time systems

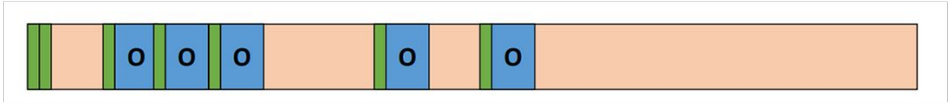
Due to pauses and unpredictable performance



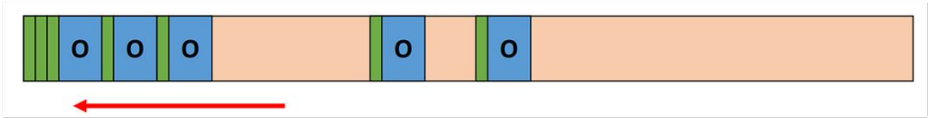
Heap before GC: Green is header, Blue is object, Orange is free heap space



Marking phase identifies Dark Blue objects as still alive and Lighter objects as dead (unmarked)



Space is free, but live objects need to be compacted



https://web.eecs.utk.edu/~mrjantz/garbage_collection.pdf

03

Copying

Operating principle, algorithm and real use cases

Operating principle and algorithm

1 — 2 — 3 — 4

Two Worlds

The heap is divided into two equal halves: "from-space" and "to-space."

Fill "From"

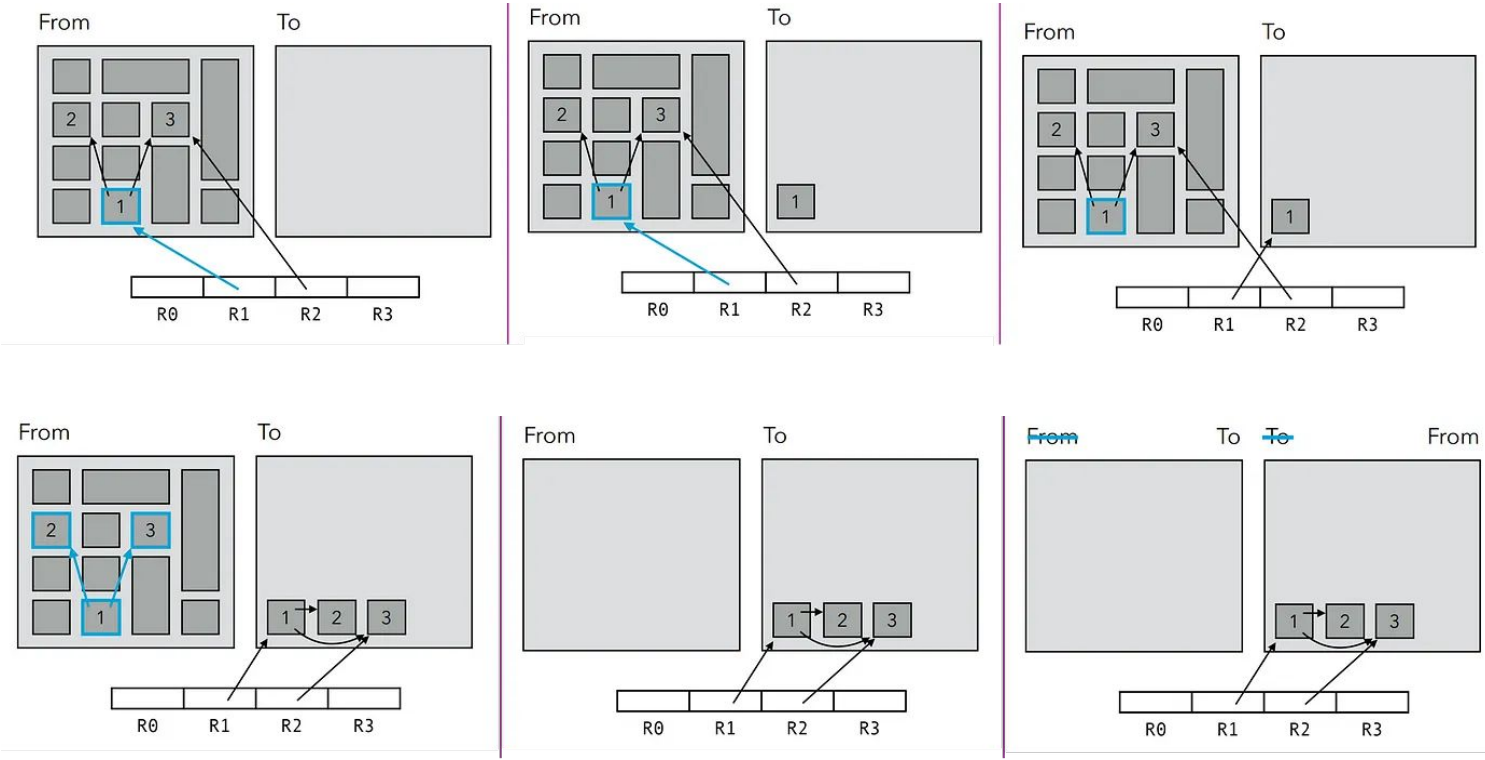
Memory is allocated in "from-space."

Copy Survivors

When "from-space" is full, reachable objects are copied to "to-space."

Space Swap

The roles of "from-space" and "to-space" are then swapped.


From
To
2
3
1
R0
R1
R2
R3
From
To
2
3
1
1
R0
R1
R2
R3
From
To
2
3
1
1
R0
R1
R2
R3
From
To
2
3
1
1
2
3
R0
R1
R2
R3
From
To
1
2
3
R0
R1
R2
R3
From
To
To
From
1
2
3
R0
R1
R2
R3

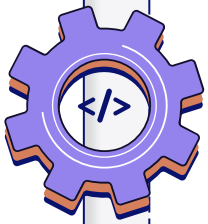
Copying algorithm variations

Feature	Naïve Copying GC	Cheney's Algorithm
Traversal Method	Depth-First Search (DFS)	Breadth-First Search (BFS)
Stack Overflow Risk	High (due to recursion)	Low (no recursion)
Extra Memory	May need a separate queue	Uses "to-space" as a queue
Efficiency	Lower (potential for stack overflow and extra memory usage)	Higher (avoids stack overflow and optimizes memory usage)
Complexity	Simpler to understand conceptually	Slightly more complex to grasp initially

Copying in Action

OCaml

- Uses a generational garbage collector with separate **minor** and **major** heaps.
- Employs copying garbage collection to move live objects from the minor heap to the major heap.



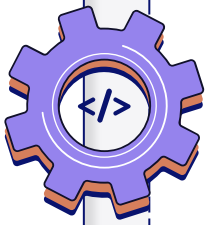
Lisp

- **Fenichel** and **Yochelson** have described how performance will degrade over time in LISP systems utilizing virtual memory.
- **Their solution , copying garbage collection** , as further modified by Cheney, was widely adopted in modern LISP systems.

Copying limitations

Pros

- **No Fragmentation:** Copying GC keeps memory organized, making it easy to find space for new objects.
- **Fast Allocation:** Allocating memory is super quick, almost as fast as stack allocation. This can significantly boost performance.
- **Focus on the Living:** The GC only spends time dealing with "live" objects, ignoring the ones that are no longer needed. This can make the process more efficient.



Cons

- **Stop-the-world:** Like traditional mark-and-sweep, copying GC also requires pausing the program's execution to perform garbage collection.
- **Double the Memory:** It needs twice the memory space because it uses two separate areas ("from-space" and "to-space").
- **Copying Costs:** Copying objects can be expensive, especially if they are large or if there are many of them.

04

Mark-Compact

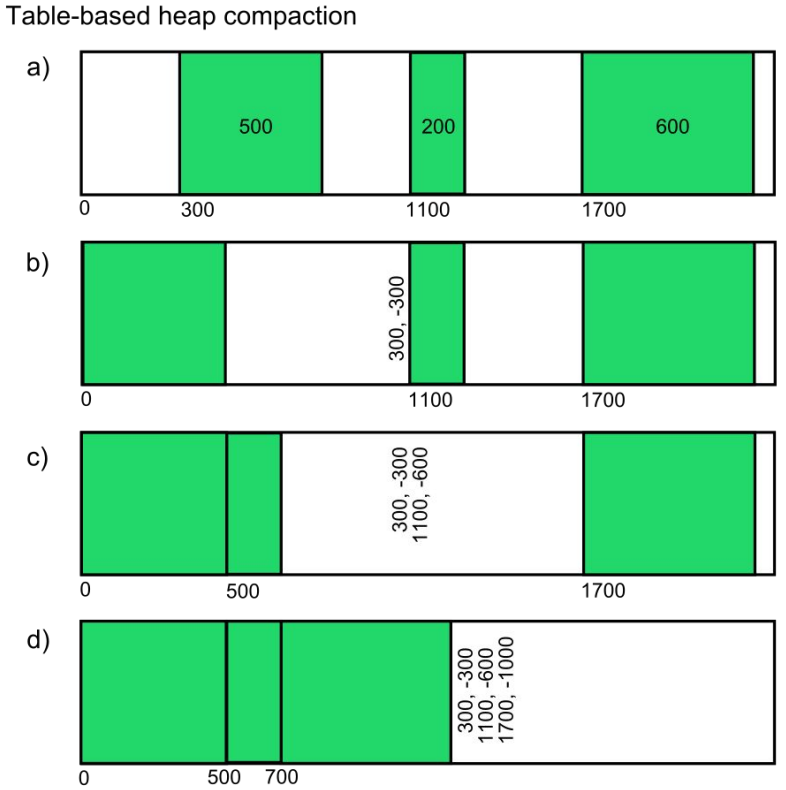
Operating principle, algorithm and real use cases

“**Mark–compact** algorithms can be regarded as a **combination of the mark–sweep algorithm and Cheney’s copying algorithm** . First, reachable objects are marked, then a compacting step relocates the reachable (marked) objects towards the beginning of the heap area.”

https://en.wikipedia.org/wiki/Mark%E2%80%93compact_algorithm

Operating principle and algorithm

- **Two Main Phases:**
  - **Marking** : The algorithm identifies and "marks" all the actively used (live) pieces of data in memory.
  - **Compacting** : The marked live data is then moved together. This frees up larger contiguous blocks of memory, reducing fragmentation.
- **Compacting Methods** :
  - Arbitrary.
  - Linearizing.
  - Sliding.

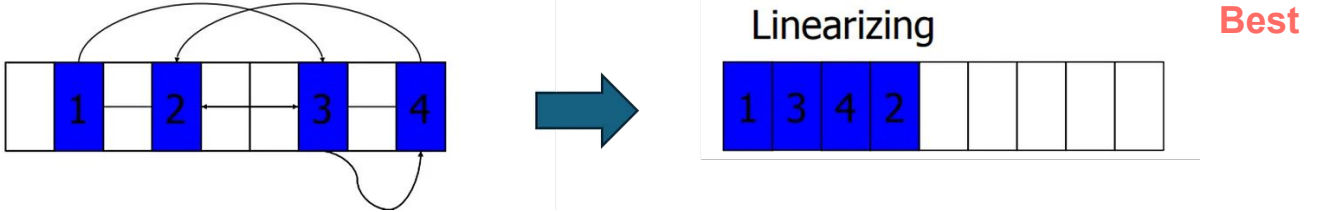


https://www.wikiwand.com/en/articles/Mark%E2%80%93compact_algorithm

Data is moved without concern for its original order or relationships.

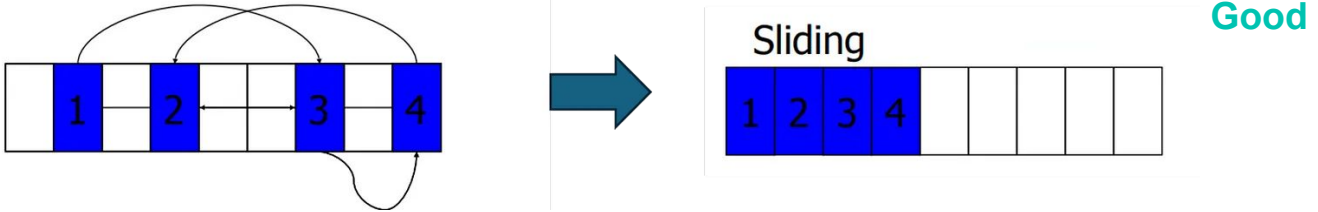
Worst

Objects that reference each other are placed closer together.



Best

The original order in which the data was allocated is preserved.

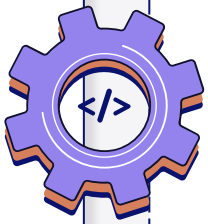


Good

Mark-Compact in Action

Uni-processor compaction

- **Two-finger algorithm** (Saunders – 1974): it's not really used in practice due to performance issues.
- **Lisp 2 algorithm** .
- **Jonkers' threaded** algorithm (1979).



Parallel compaction

- **SUN's parallel algorithm** (Flood-Detlefs-Shavit-Zhang – 2001).
- **IBM's parallel algorithm** (Abuaiadh-Ossia-Petrank-Silbershtein – 2004).
- **The Compressor** (Kermany-Petrank – 2006).

Characteristics of the Uni-processor compaction algorithms:

Algorithm	Space	Passes	Obj size	Order
Two-finger	None	2	Fixed	Arbitrary
LISP2	1 pointer-sized per object	3	Variable	Sliding
Threaded	(Pointer-sized headers)	2	Variable	Sliding

Jonkers' threaded algorithm, IBM's Parallel Compaction algorithm restricted to a single thread and IBM's Parallel Compaction algorithm fully parallel:

Compaction type	Compaction time		#Requests per second	
Triggering	≈ 90 gc default	20gc	≈ 90 gc default	20gc
Threaded	1698	1671	219.8	224.5
Parallel-restricted	1387	1251	221.7	226.1
Parallel	499	440	222.4	229.1

Mark-Compact Limitations

1

Pause Times

The compacting phase requires stopping the program's execution.

2

CPU Overhead

With complex object relationships and varying object sizes.

3

Not Ideal for All Workloads

Not suitable for applications with large heaps and high object churn.

4

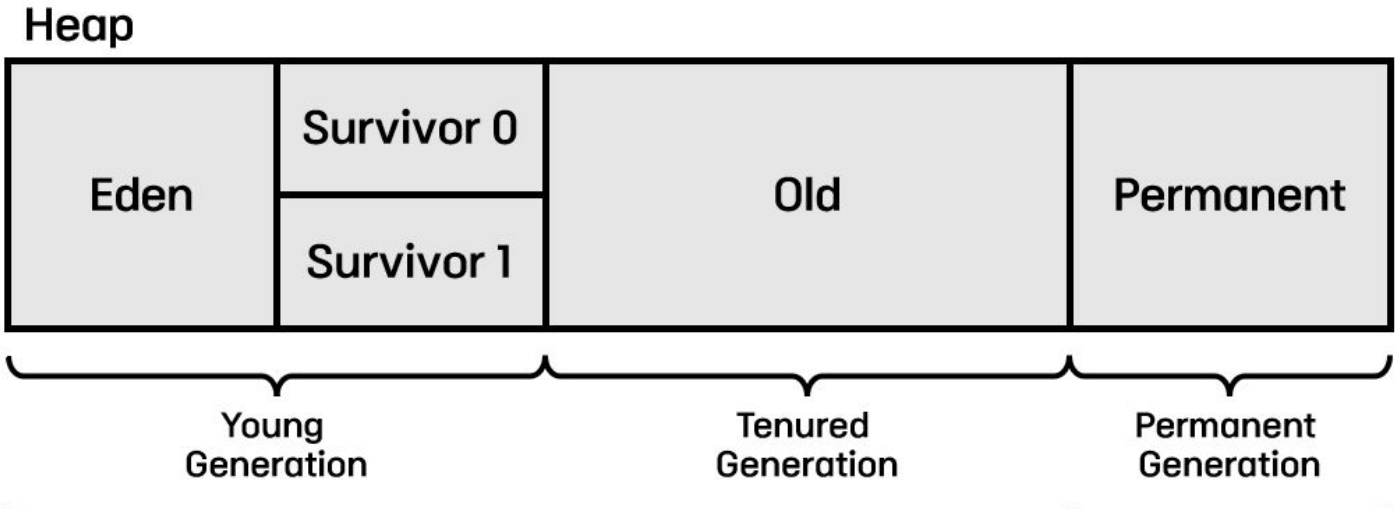
Complexity

Implementing Mark-Compact GC efficiently can be complex.

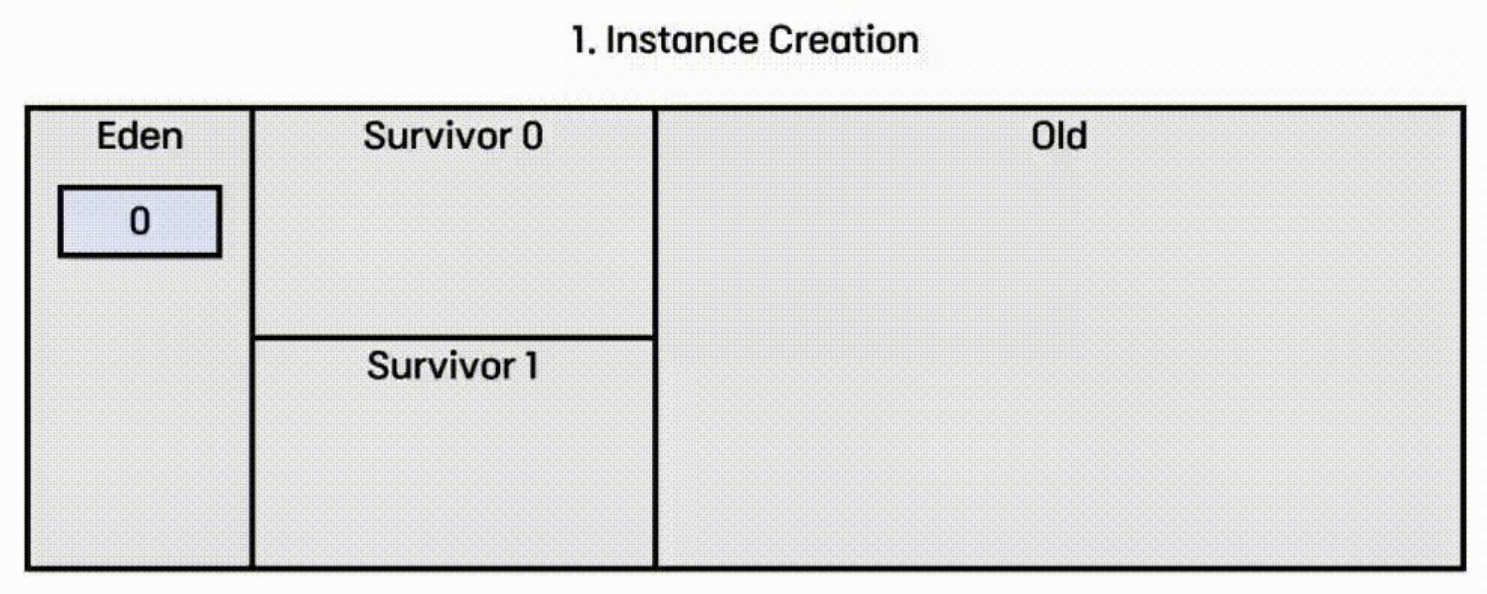
05

Generational

Operating principle, algorithm and real use cases



https://velog.io/@impala/JAVA-JVM-Garbage-Collection



https://velog.io/@impala/JAVA-JVM-Garbage-Collection

Operating principle and algorithm

1

Create an Instance

New objects are initially allocated in the young generation of the heap

2

Minor GC (Stop and Copy)

The young generation has a **limited size** . As the program keeps creating objects, the young generation eventually fills up. **This triggers a minor GC** .

3

Promotion

Objects that survive a certain number of minor GCs in the young generation are considered "long-lived."
These objects are promoted to the old generation .

4

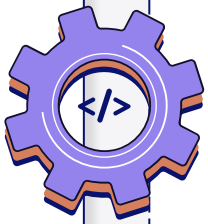
Major GC (Stop-the-World)

When the old generation also fills up this triggers a major GC, which is a more comprehensive garbage collection process.

Generational in Action

V8 (JavaScript Engine)

- **2 generations** : **young** (small, new objects) and **old** (large, long-lived objects).
- Young generation uses a **copying** strategy (semi-space) for fast collection.
- Old generation uses **mark-and-sweep** with optimizations for larger heaps.



Java Generational Collectors

- **Up to Java 6 :** Serial Collector as **default** with Parallel Collector as **optional** .
  - This was the earliest garbage collector in Java, **designed for single-threaded environments and smaller heaps** .
- **Java 7 and 8 :** Parallel GC became the **default** for server-class machines.
  - This GC uses multiple threads to perform garbage collection, improving throughput (the amount of work done in a given time).

Generational Limitations

+

Faster Minor GCs

Focus on the young generation leads to more frequent but quicker collections, minimizing pauses.

+

Less Copying

Long-lived objects are spared from repeated copying, saving processing time.

-

Inter-generational Pointer Tracking

Keeping track of references across generations adds complexity to the GC process.

4

Nepotism Risk

Since old generations are not collected as often as young ones, it is possible for dead old objects to prevent the collection of dead young objects.

06

G1(Garbage First)

Operating principle, algorithm and real use cases

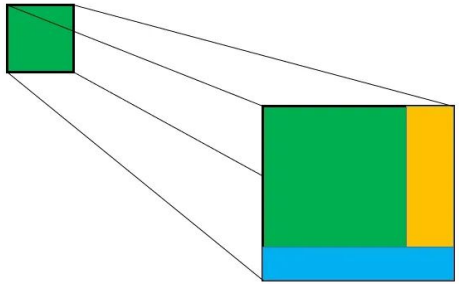
G1: Heap Regions

- G1 is a **generational** , **regionalized** , **incremental** , **parallel** , **mostly concurrent, stop-the-world** , and evacuating (**compacting**) garbage collector.
- The Heap is divided into **multiple regions** , each of **equal size** .
- The **Eden, Survivor, and Old** regions are **not required to be contiguous** like the older garbage collectors (**logical** set of regions).
- Eden regions (‘E’) and Survivor regions (‘S’) are part of the **Young Generation** .

https://velog.io/@impala/JAVA-JVM-Garbage-Collection

G1: Region

- Every region is composed of different components:
  - **Space**: the space allocated for each region can vary from 1MB to 32MB, depending on the maximum Heap size.
  - **Alive**: a portion of the objects within a region will still be in use.
  - **Garbage** : some objects in the region are no longer needed and can be classified as garbage.
  - **RSet**: the Remembered Set serves as a form of metadata that helps keep track of which objects are alive and which are no longer needed.
  - This data aids the JVM in calculating the percentage of live objects in a region at any given time (**Liveness % = Live Size / Region Size)** .

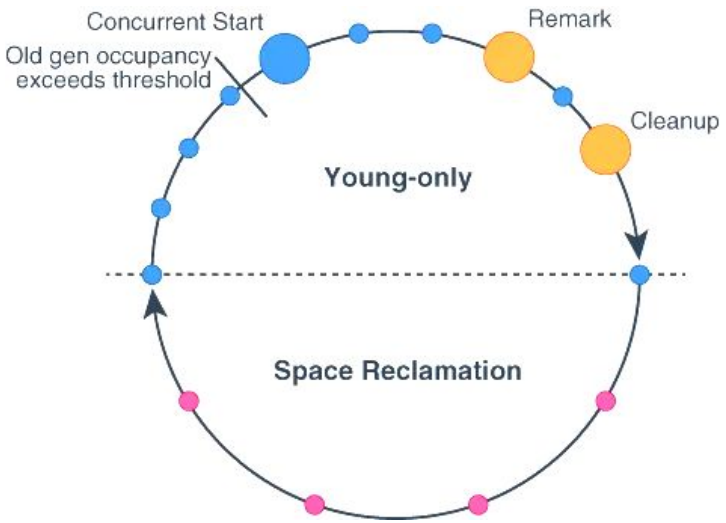


Area	1 MB to 32 MB
	Alive area
	Garbage area
	Remembered Set

https://itnext.io/in-depth-exploration-of-garbage-collector-gc-828fcef9fe5d#39ce

G1: Phases

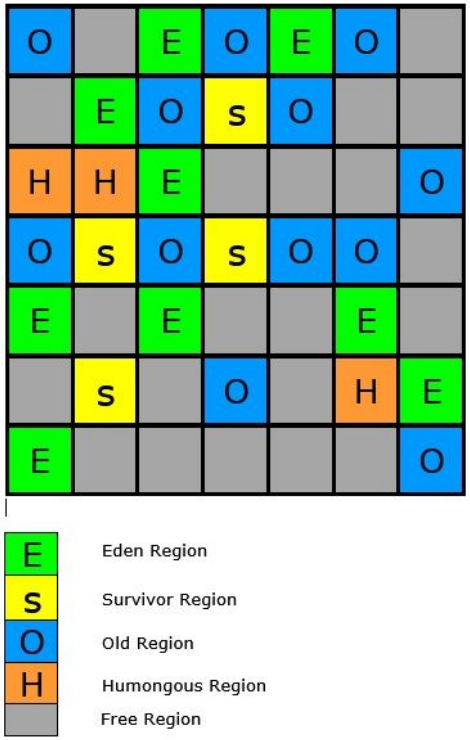
- The G1 collector **alternates between two phases** : the **young-only phase and the space-reclamation phase** .
- **Young-Only Phase:**
  - **Eden Regions Fill:** New objects are allocated in Eden regions.
  - **Young-Only Collection:** When Eden regions fill up, a young-only collection (stop-the-world) evacuates live objects to Survivor or Old regions.
- **Space-Reclamation Phase:**
  - **Concurrent Marking** : G1GC concurrently marks live objects in the old generation.
  - **Remark** : A stop-the-world pause to finalize marking.
  - **Cleanup** : A stop-the-world pause to prepare for mixed collections.
  - **Mixed Collections** : G1GC performs several mixed collections that evacuate live objects from both young and old generations.
  - **Cycle Completion** : The space-reclamation phase ends when G1GC determines that further evacuation wouldn't yield significant free space.



https://itnext.io/in-depth-exploration-of-garbage-collector-gc-828fcef9fe5d#39ce

G1 in Action

- G1GC has been the **default** garbage collector in Java **since JDK 9** (released in September 2017) and continues to be the default in the latest versions, including **JDK 21**.
- It replaced the Parallel GC, which was the default in JDK 7 and 8 for server-class machines.



https://www.dhaval-shah.com/g1-gc-primer/

G1 Pros

1

Predictable pause times

G1GC tries to meet user-defined pause-time goals, making it more predictable than earlier collectors.

2

High throughput

It achieves good throughput by performing many tasks concurrently and using parallel threads for stop-the-world phases.

3

Scalability

G1GC is designed to handle large heaps efficiently.

4

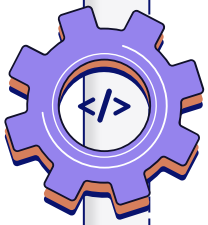
Adaptability

It can adjust its behavior based on the application's needs and the desired pause-time targets.

G1 Limitations

Memory Fragmentation

Although G1GC reduces fragmentation compared to mark-sweep collectors, it can still experience some fragmentation, **especially with objects of varying sizes and lifespans** .



Tuning Complexity

G1GC has **many tunable parameters** , making it potentially complex to configure for optimal performance. **Finding the right balance** can require careful tuning and monitoring.

07

Z

Operating principle, algorithm and real use cases

ZGC: A High-Performance Garbage Collector

- **Goal**: Handle **very large heaps** (terabytes) with minimal latency.
- **Generational ZGC** : While initially non-generational, ZGC **is moving to a generational model in Java 21** for even better performance.
- **Concurrency** : ZGC performs **most garbage collection tasks concurrently** with the application, **minimizing disruptions** .

	Serial	Parallel	G1	CMS	ZGC
Marking	-	-	✓*	✓*	✓
Compaction	-	-	-	-	✓
Reference Processing	-	-	-	-	✓
Relocation Set Selection	-	-	-	-	✓
StringTable Cleaning	-	-	-	-	✓
JNI WeakRef Cleaning	-	-	-	-	✓
JNI GlobalRefs Scanning	-	-	-	-	✓
Class Unloading	-	-	-	-	✓
Thread Stack Scanning	-	-	-	-	-**

https://cr.openjdk.org/~pliden/slides/ZGC-OracleDevLive-2020.pdf

ZGC: Main features

1

Heap Regions

ZPages.

2

Colored pointers

Extra-information about an object's life cycle within its pointer.

3

Heap Multi-Mapping

As a virtualization mechanism.

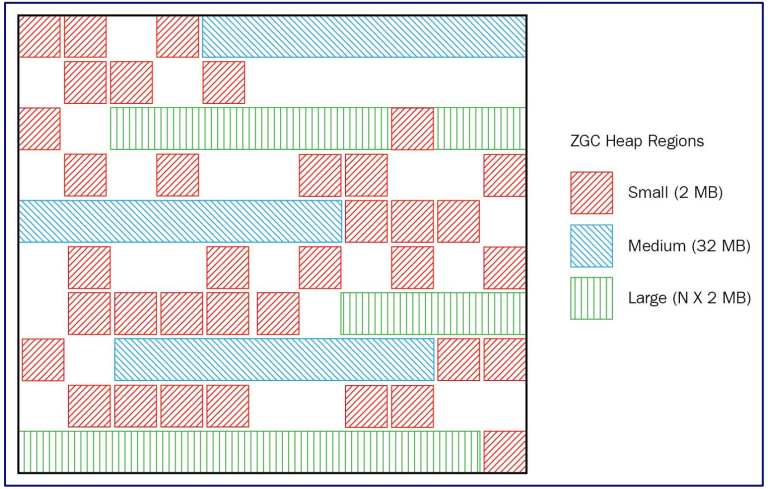
4

Load Barriers

ZGC's Traffic Controllers

ZGC: Heap Regions

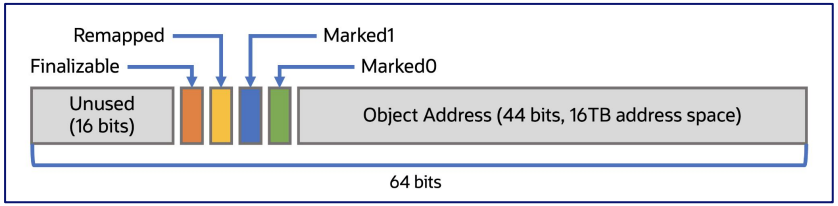
- Instead of dividing the heap into separate spaces (like **young and old generations**), ZGC **uses a single, unified space** .
- **ZGC** divides Heap into regions, also called **ZPages** .
- **ZPages** can be **dynamically created and destroyed** .
- They can be **dynamically sized** .
- ZPages come in **different sizes** to accommodate various object sizes:
  - **Small** (2 MiB – object size up to 256 KiB).
  - **Medium** (32 MiB – object size up to 4 MiB).
  - **Large** (4+ MiB – object size > 4 MiB).
- A ZGC heap can have **many ZPages of each size** .
- This organization reduces memory **fragmentation** .



https://hub.packtpub.com/getting-started-with-z-garbage-collectorzgc-in-java-11-tutorial/

ZGC: Colored Pointers

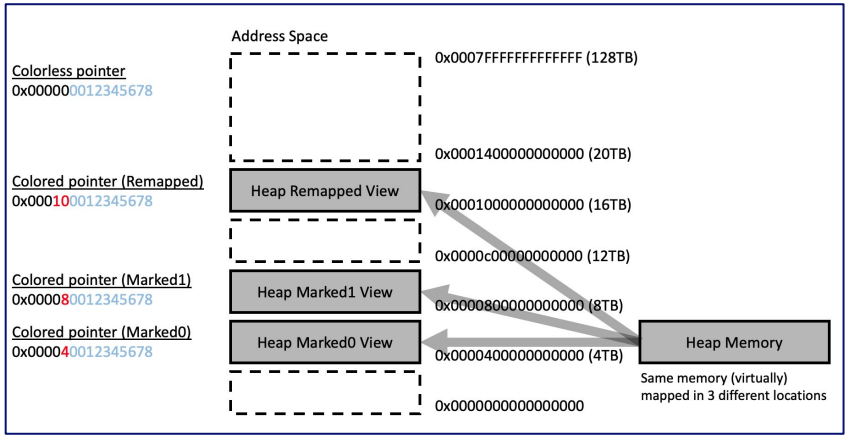
- Each object has a **colored pointer** that acts as a tag, providing information about its **state and location** .
- **The colors tell ZGC if an object is:**
  - **Marked0 & Marked1** : are used to tag **objects to collect** .
  - **Remapped** : indicates that the reference has been **relocated** .
  - **Finalizable** : The object needs **special cleanup** before being removed.
- ZGC uses these colors to perform garbage **collection tasks without stopping** the program for long periods.



https://dev.java/learn/jvm/tool/garbage-collection/zgc-deepdive/

ZGC: Heap Multi-Mapping

- ZGC uses a clever trick called "**heap multi-mapping** " to move objects around without interrupting the program.
- Multi-memory mapping allows **multiple virtual addresses to point to the same physical address** .
- The ZGC needs this technique since ZGC **can move the physical location** of an object in Heap memory while the application is running.
- **Colors** tell ZGC what to do with the object (e.g., move it, leave it alone).
- **ZGC creates multiple "virtual" copies of each object** , like reflections in a mirror. **Each copy has a different colored label** .
- When the program tries to access an object using a colored pointer, **ZGC uses the multi-mapping to find the correct "virtual" copy and redirect it to the real object** .



https://dev.java/learn/jvm/tool/garbage-collection/zgc-deepdive/

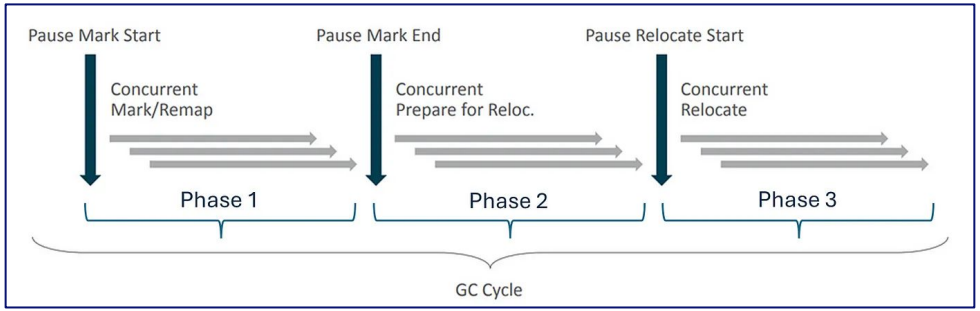
ZGC: Load Barriers

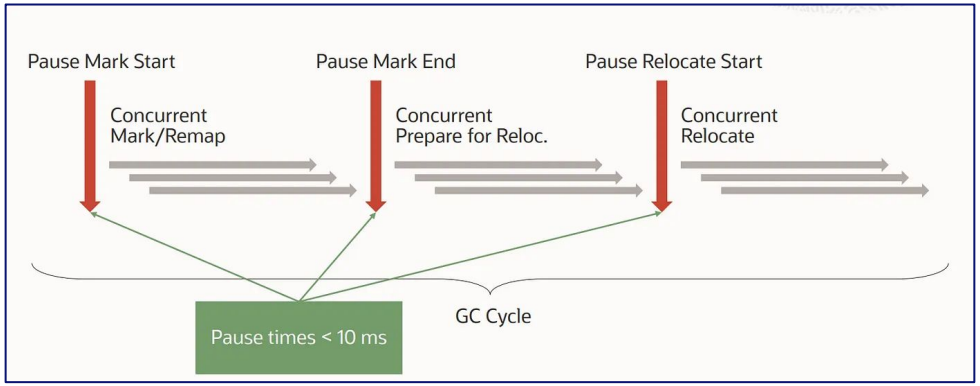
- Load barriers **are small segments** of code that the JIT compiler **injects** at strategic points, specifically when an object reference is being loaded from the Heap.
- If the object is **in the same place** , the load barrier lets the program access it directly.
- If the object has **moved** , the load barrier provides the new location.
- Load barriers **ensure** that the program always has **the correct address** for an object, even if it's been moved.

```
String n = person.name;          // Loading an object reference from heap
<load barrier needed here>
String p = n;                    // No barrier, not a load from heap
n.isEmpty();                     // No barrier, not a load from heap
int age = person.age;            // No barrier, not an object reference
```

https://cr.openjdk.org/~pliden/slides/ZGC-OracleDevLive-2020.pdf

ZGC: Operating Principle





https://cr.openjdk.org/~pliden/slides/ZGC-OracleDevLive-2020.pdf

Z Limitations

1

Platform Support

ZGC primarily targets Linux and x86-64 architectures.

2

Memory Overhead

ZGC's heap multi-mapping technique can lead to increased virtual memory usage.

3

Not Always the Best Choice

For smaller heaps or applications with different performance priorities, other GCs might be more suitable.

4

Relatively New

Since its initial release (2018), ZGC has undergone significant development and improvements, including **the addition of generational ZGC in JDK 21 (released in September 2023)** .

09

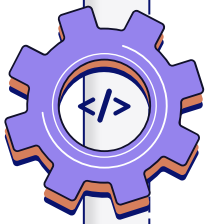
Experimental GC

Not Suitable for Production!

Experimental GC

Shenandoah

Due to concerns about its readiness for production, the generational version of Shenandoah was removed from JDK 21.



Epsilon

Epsilon GC is an experimental garbage collector introduced in Java 11. Epsilon GC **simply allocates memory and does nothing else** . It doesn't reclaim any unused objects, eventually leading to an OutOfMemoryError if the application runs out of memory. **Epsilon GC is not suitable for production environment!**

10

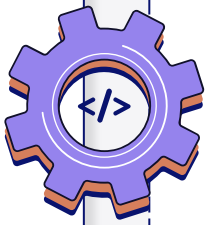
Comparing GC

Evaluating the efficiency of GC algorithms

Choosing the Right Garbage Collector

Serial GC

- **Pros:** Simple, low memory overhead.
- **Cons:** Long pauses, limited scalability.
- **Best for:** Small applications, single-threaded environments, or situations where pause times aren't critical.



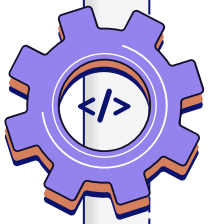
Parallel GC

- **Pros:** High throughput, efficient on multi-core systems.
- **Cons:** Longer pauses compared to concurrent collectors.
- **Best for:** Applications prioritizing throughput, batch processing, or scenarios where response time is less critical.

Choosing the Right Garbage Collector

Garbage-First (G1) GC

- **Pros:** Balanced performance (throughput and latency), predictable pauses, scalable for large heaps.
- **Cons:** Some overhead due to regions and Remembered Sets, can still have pause time fluctuations.
- **Best for:** Most server-side applications, especially those with large heaps and a need for predictable performance. A good default choice in many cases.



Z Garbage Collector (ZGC)

- **Pros:** Ultra-low latency (pauses typically under 10ms), high scalability, concurrent operation.
- **Cons:** Higher memory overhead compared to some collectors, might have platform limitations.
- **Best for:** Applications with very large heaps and strict latency requirements, such as real-time systems or high-performance transaction processing.

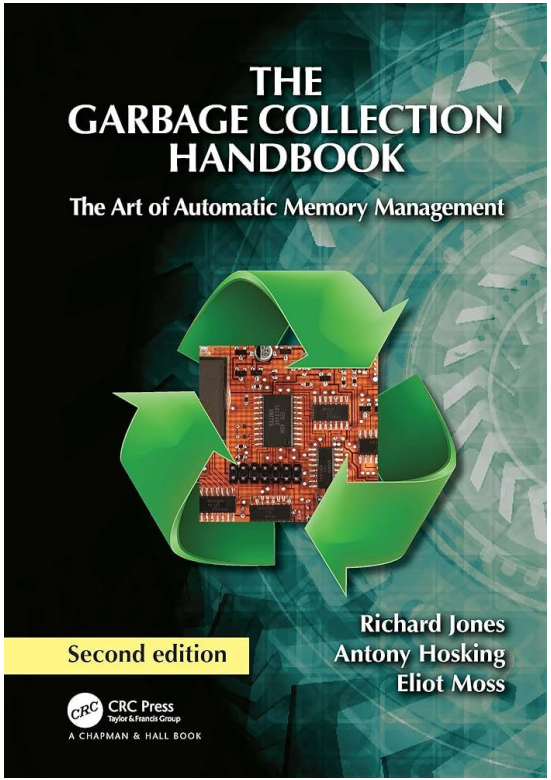
Key Considerations

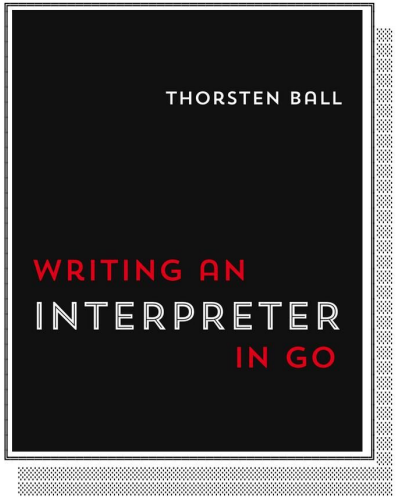
- **Application Needs:** Analyze your application's performance goals (throughput vs. latency), heap size, and sensitivity to pauses.
- **Hardware:** Consider the number of processors, memory capacity, and architecture of your system.
- **JDK Version:** Be aware of the default garbage collector and available options in your JDK version.
- **Tuning:** Most garbage collectors have tunable parameters. Experiment and monitor your application to find the optimal settings.

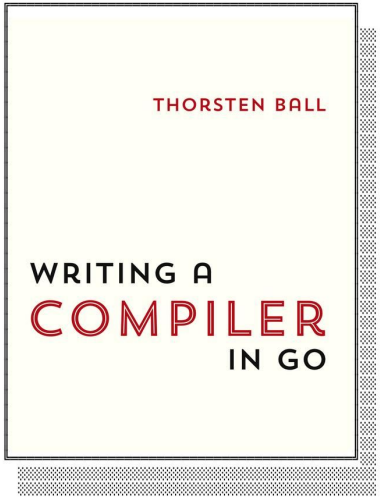
“Garbage collection isn't magic. It's a **clever** solution with **trade-offs** .”

—Ben Khalfallah Héla

Books


THE
GARBAGE COLLECTION
HANDBOOK
The Art of Automatic Memory Management
Second edition
Richard Jones
Antony Hosking
Eliot Moss
CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

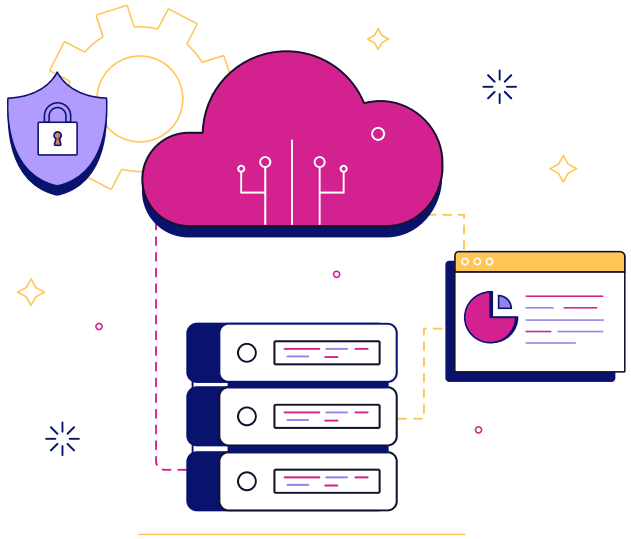

THORSTEN BALL
WRITING AN
INTERPRETER
IN GO


THORSTEN BALL
WRITING A
COMPILER
IN GO

Thanks !

Do you have any questions

https://helabenkhalfallah.com/
https://helabenkhalfallah.medium.com/



#garbage-collector