Module 16 of 20

Memory and the JVM runtime

Stack frames and the heap, aliasing, object layout, reachability and reference types, generational GC and the collectors, the string pool and Integer cache, class loading and the JIT, and diagnosing leaks.

Java hides memory management so well that most programmers meet it for the first time in an interview — "where do objects live?", "how does the garbage collector decide?", "why is 127 == 127 but 128 != 128?" — or at two in the morning when a service dies with OutOfMemoryError. This module makes the runtime visible.

It starts with the two places every value lives, the stack frame and the heap, and what a reference really is, because aliasing and pass-by-value follow from that picture. It then follows an object from allocation to reclamation: reachability from GC roots, the four reference strengths, and what a leak actually is in a language without delete. Garbage collection gets a full lesson — mark, sweep, copy, compact; generations; Serial, Parallel, G1 and ZGC; reading a GC log. The string pool, interning and the Integer cache explain the == puzzles once and for all, with StringBuilder's growth rule alongside. Class loading covers the load–link–initialise sequence, the exact initialisation order, parent-first delegation and the JIT's tiered compilation. The last lesson is the practical toolkit: OOM messages, heap dumps and histograms, the six leak shapes, and the LRU cache that keeps memory bounded by construction.

The exercises simulate the machinery — a call stack, a mark phase, generations, a class-loader chain — so that the model becomes something you have built rather than read about.

CONTENTS

1	Stack and heap — where every value lives	14 min
2	Object lifecycle, reachability and reference types	14 min
3	Garbage collection — generations, pauses and the collectors	15 min
4	The string pool, interning and the Integer cache	13 min
5	Class loading, initialisation order and the JIT	15 min
6	Diagnosing memory problems — leaks, dumps and bounded caches	14 min
✓	Checkpoint — Memory and the JVM runtime	30 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Stack and heap — where every value lives

Every Java value lives in one of two places. Local variables, parameters and the bookkeeping of a method call live on a **thread's stack**, in a frame that is created on call and discarded on return. Objects — every `new`, every array, every string — live on the **heap**, shared by all threads and reclaimed by the garbage collector. A reference is a stack (or field) value that *points at* a heap object. Almost every "why did this happen" question about Java memory — aliasing, `StackOverflowError`, `OutOfMemoryError`, why passing an array to a method lets the method change it — is answered by drawing this picture correctly. This lesson draws it.

The call stack

Each thread has its own stack. A method call pushes a **frame** holding the method's local variables (including parameters and `this`), an operand stack for intermediate results, and a return address. The frame is fixed-size — the compiler computed how many slots the method needs — and it is popped in one step on return, which is why locals are "free": no allocation, no collection, no fragmentation.

```
static int area(int w, int h) {   // frame: w, h, and a slot for the product
    int a = w * h;
    return a;
}                                 // frame popped; a, w, h cease to exist
```

The stack is small — 512 KB to 1 MB per thread by default (`-Xss`), a few thousand to a few tens of thousands of frames. Recursion without a base case, or a legitimate recursion that is simply too deep (a linked list of a million nodes walked recursively), exhausts it:

```
static int depth(int n) { return depth(n + 1); }   // StackOverflowError afte
```

`StackOverflowError` is an `Error`, not an `Exception`: it means the program's structure is wrong, and you fix it by making the recursion iterative or bounded, not by catching it. Each thread's stack is its own, so a deep recursion in one thread does not affect another.

The heap

`new` allocates on the heap and returns a reference. The heap is shared by every thread, sized by `-Xms` (initial) and `-Xmx` (maximum), and managed by the collector, which finds objects nothing refers to and reclaims them (next two lessons). Objects never live on the stack in the language model — even a `Point` created and dropped inside one method is a heap object, although the JIT's *escape analysis* may quietly avoid the allocation when it can prove the object never leaves the method.

An object's memory is its **header** (12 bytes on a 64-bit JVM with compressed class pointers: a *mark word* for the identity hash, lock state and GC age, plus a pointer to its class) followed by its fields, padded to a multiple of 8. So `new Object()` is 16 bytes; an `Integer` is 16 bytes for 4 bytes of payload; an `int[10]` is 16 bytes of header-plus-length and 40 bytes of data, padded to 56. A `String` is two objects — the `String` and its `byte[]` — around 40 bytes before a single character. These numbers are why `int[]` beats `List<Integer>` by a factor of four or five in memory, and why "a million small objects" is where Java programs discover they have a memory budget.

References: the arrow, not the box

```
int[] a = {1, 2, 3};
int[] b = a;          // b is a second arrow to the SAME array
b[0] = 99;
System.out.println(a[0]);   // 99
```

`a` and `b` are two stack slots holding the same reference. There is one array. This is **aliasing**, and it is the single most common source of "my data changed and I did not change it" bugs: a getter that returns the internal array, a constructor that stores the caller's list, a `static` collection handed to two callers. `int[] c = a.clone()` or `new ArrayList<>(list)` makes a second object — a shallow copy, so the *elements* are still shared if they are themselves references.

Method calls copy the reference, not the object:

```
static void fill(int[] arr) { arr[0] = 7; }      // changes the caller's array
static void swap(int[] arr) { arr = new int[]{0}; }   // rebinds the local copy
```

Java is strictly pass-by-value; the value happens to be an arrow. The method can follow the arrow and modify the object; it cannot make the caller's variable point somewhere else.

Primitives versus references

A primitive local (`int` , `double` , `boolean` , …) sits in the frame itself — the number is the slot. A reference local is a pointer to the heap. Fields follow the same rule inside the object: an `int` field is stored in the object; a `String` field is a pointer to another object. `null` is a reference that points nowhere; following it is a `NullPointerException` , whose helpful message since Java 14 names the variable that was null.

Boxing crosses the line: `Integer x = 5` allocates an object (or fetches a cached one — lesson 4). An `Integer[]` is an array of pointers to sixteen-byte objects scattered across the heap; an `int[]` is a contiguous block of numbers. The cache behaviour of the CPU, not just the byte count, is why the primitive array is faster.

Two errors, two causes

Error	Where	Why	Fix
`StackOverflowError`	one thread's stack	recursion too deep	make it iterative, bound the depth, or raise `-Xss` (rarely right)
`OutOfMemoryError: Java heap space`	the heap	live objects exceed `-Xmx`	find the leak or the oversized structure; raising `-Xmx` only delays a leak

Neither should be caught in ordinary code. Both are `Error` s precisely because the program cannot sensibly continue.

Threads and the heap

Every thread sees the same heap, which is what makes shared data possible — and what makes the concurrency module necessary. The stack is private: a local variable can never be seen by another thread, so anything that lives only in locals is automatically thread-safe. "Keep it in a local" is the cheapest concurrency strategy there is.

Interview angle

"Where are objects stored? Where are local variables?" Objects on the shared heap; locals and parameters in the thread's stack frame; a reference local is a pointer into the heap.

"Is Java pass-by-reference?" No — pass-by-value; for objects the value copied is the reference, so the callee can mutate the object but cannot rebind the caller's variable.

"What causes `StackOverflowError`?" Frames exhausting the thread stack — usually unbounded recursion.

"How big is `new Object()`?" About 16 bytes: a 12-byte header padded to 8-byte alignment.

"Why is `int[]` so much smaller than `List<Integer>`?" One contiguous block of 4-byte values versus an array of pointers to 16-byte objects.

Key takeaways

Stack: per thread, frames of locals pushed on call and popped on return; small and fast.

Heap: shared, holds every object and array, sized by `-Xmx`, cleaned by the GC.

A reference is an arrow; copying it makes an alias, not a copy. Methods receive the arrow.

Object header ≈ 12 bytes plus 8-byte alignment; boxing multiplies memory and scatters it.

`StackOverflowError` = recursion; `OutOfMemoryError` = too many live objects. Fix the cause, never catch.

EXERCISE 1

Frames by hand

Simulate a thread's call stack with an `ArrayDeque<String>`. Read an integer `n` and `n` commands: `call <name>` pushes a frame and prints `> <name> (depth <d>)` where `d` is the stack size after the push; `return` pops the top frame and prints `< <name>`, or prints `error: empty stack` if there is nothing to pop. After all commands print `maxDepth=<the deepest the stack got>`.

Example: `5` then `call main`, `call parse`, `return`, `call run`, `return` →

```
> main (depth 1)
> parse (depth 2)
< parse
> run (depth 2)
< run
maxDepth=2
```

INPUT 1

```
5
call main
call parse
return
call run
return
```

OUTPUT 1

```
> main (depth 1)
> parse (depth 2)
< parse
> run (depth 2)
< run
maxDepth=2
```

INPUT 2

```
3
return
call a
return
```

OUTPUT 2

```
error: empty stack
> a (depth 1)
< a
maxDepth=1
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = Integer.parseInt(in.nextLine().trim());
        Deque<String> stack = new ArrayDeque<>();
        int maxDepth = 0;
        for (int i = 0; i < n; i++) {
            String[] cmd = in.nextLine().trim().split("\\s+");
            if (cmd[0].equals("call")) {
                stack.push(cmd[1]);
                maxDepth = Math.max(maxDepth, stack.size());
                System.out.println("> " + cmd[1] + " (depth " + stack.size() + ")");
            } else if (stack.isEmpty()) {
                System.out.println("error: empty stack");
            } else {
                System.out.println("< " + stack.pop());
            }
        }
        System.out.println("maxDepth=" + maxDepth);
    }
}
```

EXERCISE 2

Alias or copy?

Read an integer `n` and `n` integers into an array `a`; let `b` start as a **copy** of `a` (`a.clone()`). Then read an integer `m` and `m` commands: `alias` makes `b` refer to the **same** array as `a`; `copy` makes `b` a fresh clone of `a`; `set <i> <v>` assigns `b[i] = v`; `show` prints `a=<Arrays.toString(a)> b=<Arrays.toString(b)>`. Watch which assignments show through `a`.

Example: `3` then `1 2 3`, then `6` then `alias`, `set 0 9`, `show`, `copy`, `set 1 7`, `show` →

```
a=[9, 2, 3] b=[9, 2, 3]
a=[9, 2, 3] b=[9, 7, 3]
```

INPUT 1

```
3
1 2 3
6
alias
set 0 9
show
copy
set 1 7
show
```

OUTPUT 1

```
a=[9, 2, 3] b=[9, 2, 3]
a=[9, 2, 3] b=[9, 7, 3]
```

INPUT 2

```
2
5 5
3
set 0 1
show
show
```

OUTPUT 2

```
a=[5, 5] b=[1, 5]
a=[5, 5] b=[1, 5]
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int[] a = new int[n];
        for (int i = 0; i < n; i++) a[i] = in.nextInt();
        int[] b = a.clone();
        int m = in.nextInt();
        for (int i = 0; i < m; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "alias" -> b = a;
                case "copy" -> b = a.clone();
                case "set" -> b[in.nextInt()] = in.nextInt();
                default -> System.out.println("a=" + Arrays.toString(a) + " b=" + Ar
            }
        }
    }
}
```

QUIZ

1. In `int[] a = new int[3];` inside a method, where do `a` and the array live?

A. Both on the stack

B. `a` in the stack frame, the array on the heap

C. Both on the heap

D. `a` on the heap, the array on the stack

2. Predict the output:

```
static void f(int[] x) { x[0] = 1; x = new int[]{5}; }
int[] a = {0}; f(a); System.out.println(a[0]);
```

A. `0`

B. `1`

C. `5`

D. Compile error

3. `StackOverflowError` is caused by…

A. Too many objects on the heap

B. Too many frames on one thread's stack — usually unbounded recursion

C. A full string pool

D. A slow garbage collector

4. Roughly how much heap does `new Object()` take on a 64-bit HotSpot JVM with compressed class pointers?

A. 0 bytes

B. 16 bytes (12-byte header padded to 8-byte alignment)

C. 64 bytes

D. 1 byte

5. Why is an `int[1_000_000]` much smaller than an `ArrayList<Integer>` of a million values?

A. Lists store strings

B. The array is 4 bytes per value in one block; the list holds pointers to 16-byte `Integer` objects

C. Arrays are compressed

D. They are the same size

ANSWER KEY · STACK AND HEAP — WHERE EVERY VALUE LIVES

1. B Locals live in the frame; every object and array lives on the heap. `a` is an arrow into it.

2. B The method receives a copy of the reference: it can change the shared array (`x[0] = 1`) but rebinding `x` does not affect `a`.

3. B Each call pushes a frame into a stack of about 0.5–1 MB; fix the recursion rather than catching the error.

4. B The header holds the mark word and class pointer; objects are aligned to 8 bytes.

5. B Boxing multiplies memory by four or five and scatters values across the heap, hurting cache behaviour too.

LESSON 2 · 14 MIN

Object lifecycle, reachability and reference types

An object is born at `new`, lives while something can reach it, and is reclaimed some time after nothing can. Java has no `delete`; you cannot free an object, you can only stop referring to it. That single rule — **reachability, not reference counting** — decides what a "memory leak" means in Java, why cycles are harmless, and why `WeakHashMap` and `SoftReference` exist. This lesson follows an object from allocation to collection and introduces the four strengths of reference.

Reachability and GC roots

The collector starts from the **roots**: every local variable and operand on every thread's stack, every `static` field of every loaded class, JNI handles, and a few runtime internals. Anything a root points to is live; anything a live object points to is live; and so on transitively. Everything else is **garbage**, no matter how many other garbage objects point to it.

```
Node a = new Node(); Node b = new Node();
a.next = b; b.next = a;      // a cycle
a = null; b = null;          // nothing on the stack reaches the pair: both ar
```

Reference counting would keep that cycle alive forever; tracing from roots does not — the count of incoming references is irrelevant, only the path from a root matters. This is the first thing to say when asked "does Java leak with circular references?": no.

What a leak is, then

A Java leak is an object that is **reachable but never used again**. The collector cannot tell the difference between "still needed" and "forgotten in a collection", so the classic leaks are all forms of forgetting:

- A `static List` or `Map` that only ever grows (a cache with no eviction, a registry nothing unregisters from).

Listeners and callbacks registered with a long-lived object and never removed — the subject keeps the observer alive, and the observer keeps everything it references alive.

`ThreadLocal` values in a thread pool: the thread outlives the request, so does the value, unless `remove()` is called.

Inner-class instances holding an implicit `this` to a large outer object (a lambda or anonymous class stored somewhere long-lived).

Substrings before Java 7u6 shared the parent's `char[]`; a habit of keeping a two-character substring of a 10 MB string kept the 10 MB. Fixed now — but the shape of the bug, *a small handle retaining a large structure*, recurs everywhere (an `Iterator` kept alive, a `Map.Entry`, a view from `subList`).

Unclosed resources: a `FileInputStream` that is not closed holds a native file descriptor the collector may take minutes to release.

The fix is always the same: remove the reference when you are done, or use a reference type the collector may clear.

The four reference strengths

Reference	Collected when…	Typical use
Strong (`Object o = …`)	never, while reachable	everything by default
`SoftReference<T>`	memory is short — cleared before an `OutOfMemoryError`	memory-sensitive caches
`WeakReference<T>`	the next GC finds no strong refs	canonicalising maps, listener lists, `WeakHashMap` keys
`PhantomReference<T>`	after finalization, before memory reuse	resource cleanup scheduling (with a `ReferenceQueue`)

```
WeakReference<byte[]> ref = new WeakReference<>(new byte[1_000_000]);
byte[] data = ref.get();        // the array, or null if it has been collected
```

`get()` on a weak or soft reference returns `null` once the collector has cleared it; code using them must always handle that. A `ReferenceQueue` passed to the constructor is where the

reference object itself is enqueued after clearing, so a cache can learn that an entry died without polling.

`WeakHashMap` holds its *keys* weakly: when a key is no longer strongly referenced elsewhere, the entry disappears at the next collection. It is right for metadata attached to objects you do not own (a map from `Thread` to some per-thread info) and wrong for anything whose keys are `String` literals or `Integer`s — those are interned or cached and never become unreachable, so the map never shrinks.

Soft references are a tempting cache but a poor one in practice: the JVM clears them all at once under pressure, giving a cache that is either full or empty. A bounded LRU (`LinkedHashMap` with `removeEldestEntry`, lesson 6) or a real cache library is nearly always better.

Finalization is gone; use `Cleaner` or `try-with-resources`

`Object.finalize()` was deprecated in Java 9 and marked for removal in 18: it ran at an unpredictable time on an unspecified thread, could resurrect objects, slowed collection, and an exception in it was silently swallowed. Resource cleanup is the job of `AutoCloseable` + **try-with-resources**, deterministic and immediate. For the rare case where a class must clean up native memory even if a careless caller forgets `close()`, `java.lang.ref.Cleaner` registers a cleanup action that runs after the object becomes phantom-reachable:

```
class Handle implements AutoCloseable {
    private static final Cleaner CLEANER = Cleaner.create();
    private final Cleaner.Cleanable cleanable;
    Handle(long nativePtr) { this.cleanable = CLEANER.register(this, () → fre
    @Override public void close() { cleanable.clean(); }
}
```

The safety net runs *eventually*; `close()` runs *now*. Always provide and prefer `close()`.

The lifecycle, end to end

Allocation — `new` bumps a pointer in the thread's local allocation buffer (TLAB): a few instructions, no lock. This is why "allocation is expensive" is folklore from another language; in Java it is *collection* that costs, in proportion to how many objects survive.

Use — the object is strongly reachable from a root.

Unreachable — the last strong reference is overwritten or its frame is popped. Nothing happens yet.

Discovered — a collection runs, traces from roots, does not find the object. Weak references to it are cleared and enqueued.

Reclaimed — its memory is reused. If it had a `Cleaner`, the action runs on the cleaner thread.

Between 3 and 5 the memory is still occupied; `System.gc()` *suggests* a collection and is ignored by many collectors (`-XX:+DisableExplicitGC`). Code that depends on an object being collected at a particular moment is wrong by design.

Interview angle

"How does Java decide what to collect?" Reachability from GC roots — stack locals, statics, JNI handles — traced transitively; not reference counts.

"Can circular references leak?" No; an unreachable cycle is garbage.

"What is a memory leak in Java?" A reachable object no one will use again — static collections, unremoved listeners, `ThreadLocal`s in pools.

"Weak versus soft reference?" Weak: cleared at the next GC once unreferenced. Soft: kept until memory is short.

"Why not `finalize()`?" Unpredictable, slow, unsafe, deprecated; use try-with-resources, and `Cleaner` as a last-resort safety net.

Key takeaways

Live = reachable from a root; everything else is garbage, cycles included.

Leaks are forgotten references: static collections, listeners, `ThreadLocal`s, small handles to big structures.

Strong / soft / weak / phantom — decreasing strength, increasing willingness to be cleared; `get()` may return `null`.

`WeakHashMap` weak-keys; useless with interned or cached keys.

No `finalize`; `AutoCloseable` now, `Cleaner` as the safety net; never rely on *when* a collection happens.

EXERCISE 1

Mark from the roots

Implement the mark phase. Read an integer `n` and `n` object names; an integer `r` and `r` root names; an integer `e` and `e` edges `from to` (object `from` holds a reference to `to`). Print `live=<reachable objects in natural order, space-separated>` and `garbage=<the rest, natural order>` (print `-` for an empty list). Cycles among unreachable objects are still garbage.

Example: `5` then `a b c d e`, `1` then `a`, `4` then `a b`, `b c`, `d e`, `e d` →

```
live=a b c
garbage=d e
```

INPUT 1

```
5
a b c d e
1
a
4
a b
b c
d e
e d
```

OUTPUT 1

```
live=a b c
garbage=d e
```

INPUT 2

```
3
x y z
0
2
x y
y z
```

OUTPUT 2

```
live=-
garbage=x y z
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> objects = new ArrayList<>();
        for (int i = 0; i < n; i++) objects.add(in.next());
        int r = in.nextInt();
        List<String> roots = new ArrayList<>();
        for (int i = 0; i < r; i++) roots.add(in.next());
        int e = in.nextInt();
        Map<String, List<String>> refs = new HashMap<>();
        for (int i = 0; i < e; i++) refs.computeIfAbsent(in.next(), k -> new ArrayLi

        Set<String> marked = new HashSet<>();
        Deque<String> work = new ArrayDeque<>(roots);
        while (!work.isEmpty()) {
            String o = work.pop();
            if (!marked.add(o)) continue;
            for (String child : refs.getOrDefault(o, List.of())) work.push(child);
        }
        TreeSet<String> live = new TreeSet<>(marked);
        TreeSet<String> garbage = new TreeSet<>(objects);
        garbage.removeAll(marked);
        System.out.println("live=" + (live.isEmpty() ? "-" : String.join(" ", live))
        System.out.println("garbage=" + (garbage.isEmpty() ? "-" : String.join(" ",
    }
}
```

EXERCISE 2

A cache with weak entries

Simulate a map whose entries can be held strongly or weakly. Read an integer `n` and `n` commands: `put <k> <v> strong|weak` stores the entry with that strength (replacing any previous); `gc` removes every **weak** entry and prints `gc cleared <count>`; `get <k>` prints `<k>=<v>` or `<k>=null`. After all commands print `size=<entries>`.

Example: `6` then `put a 1 strong`, `put b 2 weak`, `get b`, `gc`, `get b`, `get a` →

```
b=2
gc cleared 1
b=null
a=1
size=1
```

INPUT 1

```
6
put a 1 strong
put b 2 weak
get b
gc
get b
get a
```

OUTPUT 1

```
b=2
gc cleared 1
b=null
a=1
size=1
```

INPUT 2

```
4
put k v weak
put k v strong
gc
get k
```

OUTPUT 2

```
gc cleared 0
k=v
size=1
```

REFERENCE SOLUTION

```
import java.util.*;

record Entry(String value, boolean weak) { }

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Map<String, Entry> cache = new LinkedHashMap<>();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "put" -> {
                    String k = in.next(), v = in.next();
                    cache.put(k, new Entry(v, in.next().equals("weak")));
                }
                case "gc" -> {
                    int before = cache.size();
                    cache.values().removeIf(Entry::weak);
                    System.out.println("gc cleared " + (before - cache.size()));
                }
                default -> {
                    String k = in.next();
                    Entry e = cache.get(k);
                    System.out.println(k + "=" + (e == null ? "null" : e.value()));
                }
            }
        }
        System.out.println("size=" + cache.size());
    }
}
```

QUIZ

1. Two objects that reference each other and nothing else references them are…

A. Leaked forever

B. Garbage — reachability from roots, not reference counts, decides

C. Collected only by a full GC

D. Kept until `System.gc()`

2. Which is a genuine memory leak in Java?

A. A local variable in a long method

B. A `static List` that only ever grows

C. A string literal

D. A closed `FileInputStream`

3. `WeakReference<T>.get()` may return…

A. Only the object

B. The object, or `null` once the collector has cleared it

C. A copy

D. An `Optional`

4. A `WeakHashMap<String, X>` whose keys are string literals…

A. Shrinks at every GC

B. Never shrinks — literals are interned and always strongly reachable

C. Throws

D. Becomes a `HashMap`

5. Why was `finalize()` deprecated?

A. It was too fast

B. It ran at an unpredictable time on an unspecified thread, slowed collection and could resurrect objects

C. It leaked memory

D. It was never called

ANSWER KEY · OBJECT LIFECYCLE, REACHABILITY AND REFERENCE TYPES

1. B Tracing collectors start from the roots; an unreachable cycle is never marked.

2. B Reachable-but-forgotten objects are the Java leak: static collections, unremoved listeners, `ThreadLocal`s in pools.

3. B Code holding weak or soft references must always handle the cleared case.

4. B Weak keys help only when the key objects can actually become unreachable.

5. B Resource cleanup belongs in `AutoCloseable`/try-with-resources, with `Cleaner` as a safety net.

LESSON 3 · 15 MIN

Garbage collection — generations, pauses and the collectors

The garbage collector is the part of the JVM people are most often asked about and least often understand beyond "it frees memory". The good news: the ideas are few. Every collector must *find* the live objects (mark), *reclaim* the rest (sweep or copy), and usually *defragment* (compact). The generational hypothesis says most objects die young, so the heap is split by age and the young part is collected often and cheaply. The differences between Serial, Parallel, G1 and ZGC are about **how many threads** do that work and **how long the application is paused** while they do. This lesson gives you the model and the vocabulary to read a GC log.

Three primitive algorithms

Mark–sweep. Trace from the roots and mark every live object; then walk the heap and free everything unmarked. Simple; leaves the free space fragmented, so a later large allocation may fail although the total free memory is enough.

Mark–compact. Mark, then slide the live objects together to one end and update every reference to them. No fragmentation, allocation becomes a pointer bump, but every survivor is moved.

Copying. Divide space in two; copy live objects from the active half to the empty half, leaving garbage behind; swap. Cost is proportional to *live* data, not heap size — which is superb when most objects are dead, and terrible when most are alive.

Every real collector is a combination: copying for the young generation (few survivors), mark–sweep or mark–compact for the old (many survivors).

The generational hypothesis

Measured across decades of programs: **most objects die almost immediately** (an iterator, a temporary string, a boxed integer), and objects that survive a while tend to survive a long time (the cache, the session, the connection pool). So the heap is divided:

Young generation: *Eden*, where everything is allocated, plus two *survivor* spaces. A **minor GC** copies Eden's survivors to a survivor space, incrementing each object's *age* in its header. Objects that reach the tenuring threshold (default 15, adjusted dynamically) are **promoted** to the old generation.

Old generation (tenured): long-lived objects. Collected by a **major** (or *full*) GC — rarer, larger, slower.

Minor collections are frequent and fast precisely because Eden is mostly dead by the time it fills. A **premature promotion** problem — survivor spaces too small, medium-lived objects promoted and then dying in the old generation — makes full GCs frequent; that is the commonest tuning issue in practice.

Stop-the-world, and why pauses matter

To trace the object graph safely the collector must stop application threads at a **safepoint** — every thread reaches a point where its stack is well described and pauses. The classic collectors stop the world for the whole collection; a full GC of a 30 GB heap can pause for seconds. Modern collectors do most of the marking **concurrently** with the application and stop the world only briefly, trading throughput (CPU spent on GC) for latency (pause length). Which trade is right depends on the program: a batch job wants throughput; a trading system wants a bounded pause.

The collectors you will be asked about

Collector	Flag	Threads	Pauses	Use when
Serial	`-XX:+UseSerialGC`	one	stop-the-world	small heaps, single-core containers, client tools
Parallel (throughput)	`-XX:+UseParallelGC`	many, for young and old	stop-the-world but shorter	batch jobs where total throughput matters, pauses do not
G1 (Garbage-First)	`-XX:+UseG1GC` (default since 9)	many; concurrent marking	short, targeted by `-XX:MaxGCPauseMillis` (200 ms default)	the general-purpose default: multi-GB heaps, servers
ZGC	`-XX:+UseZGC` (production since 15)	concurrent almost everything	sub-millisecond, independent of heap size	very large heaps, latency-sensitive services
Shenandoah	`-XX:+UseShenandoahGC`	concurrent compaction	low, like ZGC	same niche; Red Hat's collector

G1 divides the heap into equal regions (1–32 MB) rather than fixed contiguous generations; regions are young or old by role, collection picks the regions with the most garbage first ("garbage first"), and marking of the old generation runs concurrently. It is the right default answer to "which collector would you use?" — with "ZGC if the heap is huge and pauses must stay tiny".

Reading a GC log

Turn logging on with `-Xlog:gc` (Java 9+; `-verbose:gc` still works):

```
[0.512s][info][gc] GC(3) Pause Young (Normal) (G1 Evacuation Pause) 124M→31M(
[2.130s][info][gc] GC(7) Pause Full (System.gc()) 240M→88M(512M) 141.207ms
```

Read: at 0.512 s, the fourth collection, a young pause; heap went from 124 MB used to 31 MB used, of a 512 MB heap, in 8.4 ms. A log full of short `Pause Young` lines is healthy. `Pause Full` lines mean the old generation filled (or someone called `System.gc()`); frequent ones mean a leak, an undersized heap, or premature promotion. Use `-Xlog:gc*` for details and `jstat -gcutil <pid> 1s` for a live view.

Sizing and the flags that matter

`-Xms` / `-Xmx` — initial and maximum heap. Setting them equal avoids resize pauses on servers. In a container, `-XX:MaxRAMPercentage=75` is usually better than a fixed number.

`-Xmn` or `-XX:NewRatio` — young generation size. Larger young = fewer minor GCs and less premature promotion.

`-XX:MaxGCPauseMillis` — G1's target; tighter targets cost throughput.

`-XX:+HeapDumpOnOutOfMemoryError` — always on in production; a dump is the only way to diagnose the OOM after the fact.

`-XX:+DisableExplicitGC` — turn `System.gc()` into a no-op; libraries that call it are a known source of pauses.

The most effective tuning is not a flag: allocate less. Fewer temporaries, primitive arrays instead of boxed lists, reused buffers, `StringBuilder` instead of concatenation in a loop.

What the collector cannot fix

A **leak** — reachable objects no one uses. The GC will faithfully keep them and eventually throw `OutOfMemoryError`.

Off-heap memory: direct `ByteBuffer`s, native libraries, thread stacks, Metaspace (class metadata). `OutOfMemoryError: Metaspace` means classes are being loaded and never unloaded — usually a class-loader leak in a redeploying container.

`GC overhead limit exceeded`: the JVM spent more than 98% of recent time collecting and recovered under 2% — the heap is effectively full of live data.

Interview angle

"Explain generational GC." Most objects die young; Eden + survivors collected often by copying, promotion to old after surviving several minors; old collected rarely.

"Minor versus major GC?" Minor: young generation only, fast, frequent. Major/full: whole heap, slow, should be rare.

"What is stop-the-world?" All application threads paused at a safepoint while the collector works; modern collectors make most of it concurrent.

"Which collector is the default?" G1 since Java 9; ZGC for huge heaps with tight latency.

"Does `System.gc()` collect?" It suggests; it may be ignored, and it is a full pause when honoured — do not call it.

Key takeaways

Mark, then sweep / compact / copy; real collectors copy the young and mark–compact the old.

Generational: Eden → survivors (age) → old; minor GCs cheap, full GCs expensive; premature promotion is the common problem.

Serial (one thread), Parallel (throughput), G1 (default, region-based, pause target), ZGC/Shenandoah (concurrent, sub-ms).

`-Xlog:gc` and `jstat` show the story; `Pause Full` lines are the alarm.

Tune by allocating less; the GC cannot fix a leak or off-heap growth.

EXERCISE 1

Survivors and promotion

Simulate a generational heap. Read the tenuring threshold `T`, then an integer `n` and `n` events: `alloc <id>` puts an object of age 0 into the young generation; `free <id>` marks it garbage (wherever it is); `gc` is a **minor** collection: remove garbage from the young generation only, increment every young survivor's age, and **promote** (move to the old generation, keeping order) those whose age reaches `T`; `full` removes garbage from both generations without ageing anything. After each `gc` or `full` print `young=[id:age, …] old=[id, …]` in insertion order.

Example: `T = 2`, then `7` events: `alloc a`, `alloc b`, `gc`, `free a`, `gc`, `free b`, `full` →

```
young=[a:1, b:1] old=[]
young=[] old=[b]
young=[] old=[]
```

INPUT 1

```
2
7
alloc a
alloc b
gc
free a
gc
free b
full
```

OUTPUT 1

```
young=[a:1, b:1] old=[]
young=[] old=[b]
young=[] old=[]
```

INPUT 2

```
1
3
alloc x
gc
gc
```

OUTPUT 2

```
young=[] old=[x]
young=[] old=[x]
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static void show(Map<String, Integer> young, List<String> old) {
        StringJoiner y = new StringJoiner(", ", "[", "]");
        young.forEach((id, age) -> y.add(id + ":" + age));
        System.out.println("young=" + y + " old=" + old);
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int threshold = in.nextInt();
        int n = in.nextInt();
        Map<String, Integer> young = new LinkedHashMap<>();   // id -> age
        List<String> old = new ArrayList<>();
        Set<String> garbage = new HashSet<>();
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            switch (cmd) {
                case "alloc" -> young.put(in.next(), 0);
                case "free" -> garbage.add(in.next());
                case "gc" -> {
                    young.keySet().removeIf(garbage::contains);
                    Iterator<Map.Entry<String, Integer>> it = young.entrySet().itera
                    while (it.hasNext()) {
                        Map.Entry<String, Integer> e = it.next();
                        e.setValue(e.getValue() + 1);
                        if (e.getValue() ⩾ threshold) {
                            old.add(e.getKey());
                            it.remove();
                        }
                    }
                    show(young, old);
                }
                default -> {
                    young.keySet().removeIf(garbage::contains);
                    old.removeIf(garbage::contains);
                    show(young, old);
                }
            }
        }
    }
}
```

EXERCISE 2

GC log summary

Read lines of a simplified GC log until end of input. Each line has the form `Pause <Young|Full> <before>M→<after>M(<total>M) <millis>ms`, for example `Pause Young 124M→31M(512M) 8.4ms`. Print:

```
pauses=<count> young=<count> full=<count>
totalMs=<sum, one decimal> maxMs=<largest, one decimal>
reclaimedMB=<sum of before - after>
```

Use a regular expression to pull the fields out. Assume at least one line.

INPUT 1

```
Pause Young 124M→31M(512M) 8.4ms
Pause Young 130M→28M(512M) 6.1ms
Pause Full 240M→88M(512M) 141.2ms
```

OUTPUT 1

```
pauses=3 young=2 full=1
totalMs=155.7 maxMs=141.2
reclaimedMB=347
```

INPUT 2

```
Pause Young 10M→2M(64M) 1ms
```

OUTPUT 2

```
pauses=1 young=1 full=0
totalMs=1.0 maxMs=1.0
reclaimedMB=8
```

REFERENCE SOLUTION

```
import java.util.*;
import java.util.regex.*;

public class Main {
    public static void main(String[] args) throws java.io.IOException {
        Pattern p = Pattern.compile("Pause (Young|Full) (\\d+)M→(\\d+)M\\((\\d+)M\\
        String text = new String(System.in.readAllBytes());
        int pauses = 0, young = 0, full = 0;
        double total = 0, max = 0;
        long reclaimed = 0;
        for (String line : text.split("\\R")) {
            Matcher m = p.matcher(line.trim());
            if (!m.find()) continue;
            pauses++;
            if (m.group(1).equals("Young")) young++; else full++;
            reclaimed += Long.parseLong(m.group(2)) - Long.parseLong(m.group(3));
            double ms = Double.parseDouble(m.group(5));
            total += ms;
            max = Math.max(max, ms);
        }
        System.out.println("pauses=" + pauses + " young=" + young + " full=" + full)
        System.out.println(String.format(Locale.ROOT, "totalMs=%.1f maxMs=%.1f", tot
        System.out.println("reclaimedMB=" + reclaimed);
    }
}
```

QUIZ

1. The generational hypothesis says…

A. All objects live equally long

B. Most objects die young; those that survive tend to live long

C. Old objects die first

D. Objects never die

2. A minor GC collects…

A. The whole heap

B. The young generation only

C. The old generation only

D. Metaspace

3. Which collector is the default since Java 9 and targets a pause goal with region-based collection?

A. Serial

B. Parallel

C. G1

D. CMS

4. A GC log full of `Pause Full` lines suggests…

A. A healthy application

B. A leak, an undersized heap or premature promotion

C. The JIT is off

D. Too many threads

5. `System.gc()` …

A. Frees memory immediately

B. Only suggests a collection — often a full stop-the-world pause when honoured, and ignorable with `-XX:+DisableExplicitGC`

C. Is required before `exit`

D. Runs a minor GC

6. Predict the log line meaning: `Pause Young 124M→31M(512M) 8.4ms`

A. Heap grew from 124 to 512 MB

B. A young collection took the heap from 124 MB used to 31 MB used, of a 512 MB heap, in 8.4 ms

C. 31 objects survived

D. 8.4 MB were freed

ANSWER KEY · GARBAGE COLLECTION — GENERATIONS, PAUSES AND THE COLLECTORS

1. B Hence Eden + survivor spaces collected often and cheaply, and an old generation collected rarely.

2. B Survivors are copied to a survivor space and aged; objects past the tenuring threshold are promoted to the old generation.

3. C G1 divides the heap into regions, marks concurrently and collects the garbage-richest regions first; ZGC is the choice for huge heaps with sub-millisecond pauses.

4. B Full collections should be rare; frequent ones mean the old generation keeps filling with live data.

5. B Code must never depend on when a collection happens.

6. B before->after(total) time is the format shared by every modern JVM GC log.

LESSON 4 · 13 MIN

The string pool, interning and the Integer cache

Two interview questions live here, and both are really questions about the heap. *"Why is* `"java" == "java"` *true but* `new String("java") == "java"` *false?"* — the string pool. *"Why is* `Integer.valueOf(127) == Integer.valueOf(127)` *true but* `Integer.valueOf(128) == Integer.valueOf(128)` *false?"* — the Integer cache. Both are the JVM sharing immutable objects behind your back, both are invisible until you compare with `==`, and both reward knowing exactly where the line is drawn. This lesson also covers the growth rule of `StringBuilder`, because it is the same idea — memory the runtime manages for you, with behaviour you can predict.

The string pool

Every string **literal** in a class file is interned: when the class is loaded, `"java"` becomes one `String` object in the JVM's string table, and every literal with the same characters — in this class or any other — refers to that one object.

```
String a = "java";
String b = "java";
System.out.println(a == b);              // true — same pooled object
String c = new String("java");
System.out.println(a == c);              // false — `new` always makes a fresh
System.out.println(a.equals(c));         // true — same characters
System.out.println(a == c.intern());     // true — intern() returns the pooled
```

`intern()` looks the string up in the pool, adds it if absent, and returns the pooled reference. It is how you get identity equality for strings that arrive at runtime — from a file, a socket, a `Scanner` — and it is mostly used to save memory when millions of strings have few distinct values (city names in a dataset). It is *not* a way to make `==` correct; `equals` is.

Compile-time constant folding extends the pool to expressions: `"ja" + "va"` is a constant expression, folded to `"java"` by `javac`, so `("ja" + "va") == "java"` is `true`. But add a variable and the concatenation happens at run time, producing a new object:

```
String ja = "ja";
System.out.println((ja + "va") == "java");          // false — runtime concater
final String JA = "ja";
System.out.println((JA + "va") == "java");          // true — a final String wi
```

The pool lives in the heap (since Java 7; it was in PermGen before), so pooled strings are collectable when nothing refers to them — though literals stay reachable from their classes. Its size is a hash table tunable with `-XX:StringTableSize`.

Why the rule matters: `==` on strings is a bug

Because pooling makes `==` *sometimes* true, code that compares strings with `==` passes its tests with literals and fails in production with input from anywhere else. The rule has no exceptions: **compare strings with `equals`** (or `equalsIgnoreCase`, or `compareTo`). Use `==` only to test for `null`, or when you have deliberately interned both sides and can prove it.

The Integer cache

`Integer.valueOf(int)` — which autoboxing calls — returns a **cached** instance for values in `-128..127` and a new object outside that range:

```
Integer p = 127, q = 127;   System.out.println(p == q);   // true — same cache
Integer r = 128, s = 128;   System.out.println(r == s);   // false — two objec
System.out.println(r.equals(s));                           // true
```

The lower bound is fixed; the upper bound is guaranteed to be *at least* 127 and can be raised with `-XX:AutoBoxCacheMax=…`, so `128` giving `false` is the default HotSpot behaviour, not a language guarantee. `Long`, `Short` and `Byte` cache the same range; `Character` caches `0..127`; `Boolean` has exactly two instances; `Double` and `Float` cache nothing. `new Integer(127)` (deprecated) bypasses the cache entirely.

The lesson is identical to the string one: **compare boxed numbers with `equals` or unbox first.** `Integer` in a `List<Integer>` or a `Map<String, Integer>` is exactly where `==` sneaks in — `if (map.get(a) == map.get(b))` works for small counts and silently breaks past 127.

Autoboxing allocates

Every boxing outside the cache is an allocation. `Integer sum = 0; for (…) sum += x;` allocates a new `Integer` per iteration (`sum + x` is an `int`, boxed back into a new object). A `Map<Integer, Integer>` counting frequencies boxes on every `merge`. It is rarely the bottleneck at interview scale and always the first thing to fix in a real hot loop: `int` / `long` locals, `int[]` arrays, `IntStream` instead of `Stream<Integer>`.

`StringBuilder` growth

A `StringBuilder` is a resizable `byte[]` plus a count. Its default capacity is 16 (or `16 + s.length()` when built from a string). When an append needs more room than the current capacity, the array is replaced by one of capacity `max(needed, 2 × old + 2)` — the `+2` is a historical quirk from the days when the minimum growth had to accommodate a two-character append. So 16 → 34 → 70 → 142 → 286 → …: a doubling that makes repeated appends O(1) amortised, the same argument as `ArrayList`.

```
StringBuilder sb = new StringBuilder();      // capacity 16
sb.append("0123456789ABCDEFG");              // 17 chars: needs 17, 2×16+2 = 3
sb.append("x".repeat(40));                   // 57 chars: needs 57, 2×34+2 = 7
```

`sb.capacity()` reports it; `new StringBuilder(n)` pre-sizes when you know the final length, avoiding every copy. `s += x` in a loop, by contrast, allocates a new `String` of the full length each time — O(n²) characters copied for n appends — which is why "use `StringBuilder` in loops" is on every Java checklist. The compiler does turn a *single* expression `a + b + c` into one efficient concatenation; it is the loop it cannot fix.

Hash codes are cached too

`String.hashCode()` is computed once and stored in the object (strings are immutable, so it never changes) — which is part of why strings make excellent map keys, and why a mutable object as a key is a disaster: its hash code is cached in the *map*, in the bucket index, and a mutation leaves it in the wrong bucket forever.

Interview angle

"`"a" == "a"` ?" True: literals are interned into the pool. `new String("a") == "a"` is false; `.intern()` restores identity.

"What does `intern()` do?" Returns the pooled instance for that content, adding it if absent.

"`Integer a = 127, b = 127; a == b` ? And 128?" True for 127 (cached −128..127); false for 128 by default — always `equals`.

"Why `StringBuilder` in a loop?" Concatenation copies the whole string each time (O(n²)); the builder grows geometrically and copies once per doubling.

"Is `"ja" + "va" == "java"` ?" True — constant folding at compile time. With a non-final variable it is false.

Key takeaways

Literals and compile-time constants are pooled; `new String` and runtime concatenation are not; `intern()` fetches the pooled one.

`==` on strings is a bug by construction; `equals` always.

`Integer.valueOf` caches −128..127 (upper bound tunable); `Boolean` two instances; `Double` none. Compare boxed values with `equals`.

Boxing allocates; go primitive in hot code.

`StringBuilder` grows to `max(needed, 2·old + 2)`; pre-size it when you know the length; never `+=` a string in a loop.

EXERCISE 1

Same or equal?

Read an integer `n` and `n` words. For each word `w` print `<w>: equals=<w.equals("java")> same=<w == "java"> interned=<w.intern() == "java">`. Then read an integer `m` and `m` integers; for each `x` print `<x>: boxedSame=<Integer.valueOf(x) == Integer.valueOf(x)>`. Predict every line before you run it: a token read from input is never the pooled literal, and the Integer cache ends at 127.

Example: `2` then `java Java`, `2` then `127 128` →

```
java: equals=true same=false interned=true
Java: equals=false same=false interned=false
127: boxedSame=true
128: boxedSame=false
```

INPUT 1

```
2
java Java
2
127 128
```

OUTPUT 1

```
java: equals=true same=false
interned=true
Java: equals=false same=false
interned=false
127: boxedSame=true
128: boxedSame=false
```

INPUT 2

```
1
java
3
-128 -129 0
```

OUTPUT 2

```
java: equals=true same=false
interned=true
-128: boxedSame=true
-129: boxedSame=false
0: boxedSame=true
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        for (int i = 0; i < n; i++) {
            String w = in.next();
            System.out.println(w + ": equals=" + w.equals("java") + " same=" + (w ==
        }
        int m = in.nextInt();
        for (int i = 0; i < m; i++) {
            int x = in.nextInt();
            Integer a = Integer.valueOf(x), b = Integer.valueOf(x);
            System.out.println(x + ": boxedSame=" + (a == b));
        }
    }
}
```

EXERCISE 2

Watch the builder grow

Read an integer `n` and `n` words. Append each to one `StringBuilder` created with the default constructor, and after every append print `len=<length()> cap=<capacity()>`. Finally print `reallocations=<how many times the capacity changed>`. Predict the capacities first with the rule `max(needed, 2 × old + 2)` starting from 16, then let the JDK confirm it.

Example: `3` then `abcdefghij klmnopqrst u` →

```
len=10 cap=16
len=20 cap=34
len=21 cap=34
reallocations=1
```

INPUT 1

```
3
abcdefghij klmnopqrst u
```

OUTPUT 1

```
len=10 cap=16
len=20 cap=34
len=21 cap=34
reallocations=1
```

INPUT 2

```
1
hi
```

OUTPUT 2

```
len=2 cap=16
reallocations=0
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        StringBuilder sb = new StringBuilder();
        int reallocations = 0;
        for (int i = 0; i < n; i++) {
            String w = in.next();
            int before = sb.capacity();
            sb.append(w);
            if (sb.capacity() != before) reallocations++;
            System.out.println("len=" + sb.length() + " cap=" + sb.capacity());
        }
        System.out.println("reallocations=" + reallocations);
    }
}
```

QUIZ

1. Predict the output:

```
String a = "ja" + "va";
String b = "java";
System.out.println(a == b);
```

A. `false`

B. `true` — the concatenation of two literals is folded at compile time and pooled

C. Compile error

D. Depends on the JVM

2. `new String("x") == "x"` is…

A. `true`

B. `false` — `new` always creates a fresh object

C. A compile error

D. `true` after a GC

3. `Integer a = 200, b = 200; a == b` is…

A. `true`

B. `false` by default — 200 is outside the −128..127 cache

C. A compile error

D. `true` only on 64-bit JVMs

4. A `StringBuilder` of capacity 34 holding 34 characters receives `append("x")`. Its new capacity is…

A. 35

B. 68

C. 70

D. 64

5. Why is `s += x` in a loop O(n²)?

A. Strings are linked lists

B. Each concatenation copies the whole current string into a new one

C. The pool fills up

D. It is not — the compiler fixes it

ANSWER KEY · THE STRING POOL, INTERNING AND THE INTEGER CACHE

1. B Constant expressions are folded by `javac`; with a non-final variable operand the concatenation happens at runtime and produces a new object.

2. B `.intern()` on the new string would return the pooled instance and make the comparison true.

3. B Compare boxed values with `equals`, or unbox. The upper bound can be raised with `-XX:AutoBoxCacheMax`, which is why `==` is never safe.

4. C `max(needed, 2 × 34 + 2) = max(35, 70) = 70`.

5. B The compiler optimises a single expression, not a loop. `StringBuilder` copies only on geometric growth.

LESSON 5 · 15 MIN

Class loading, initialisation order and the JIT

Two more things the runtime does behind your back. **Class loading** brings a `.class` file into the JVM the first time it is needed, through a chain of loaders that decides which version of a class you get and when its `static` initialisers run — the mechanics behind `ClassNotFoundException`, `NoClassDefFoundError`, the lazy singleton idiom, and every "why did my static block run *there*?" surprise. **Just-in-time compilation** turns hot bytecode into machine code while the program runs, which is why Java is fast after warming up, why microbenchmarks lie, and why a method that "does nothing" can take zero time. Both are asked about; both are simpler than their reputations.

Loading, linking, initialising

A class goes through three phases, lazily, the first time the program touches it:

Loading — a class loader finds the bytes (from a jar, a directory, the network, or generated in memory) and creates the `Class` object. Nothing runs yet.

Linking — *verification* (the bytecode is well-formed and type-safe: this is the sandbox), *preparation* (static fields get their default zero values), and lazy *resolution* of symbolic references to other classes.

Initialisation — the static initialisers run: `static` field initialisers and `static { }` blocks, in textual order, exactly once, under a lock so that two threads racing to use the class both see a fully initialised one.

Initialisation is triggered by the *first active use*: `new`, a static method call, a non-constant static field access, `Class.forName`, or initialising a subclass (the superclass initialises first). It is **not** triggered by referring to a compile-time constant (`Foo.MAX` where `MAX` is `static final int MAX = 10` is inlined by `javac`) nor by `Foo.class` or declaring a variable of the type.

```
class Config {
    static final int PORT = 8080;                 // a constant: using it does
    static { System.out.println("Config loaded"); }
}
int p = Config.PORT;          // prints nothing
Config c = null;              // prints nothing
new Config();                 // prints "Config loaded"
```

The order, spelled out

For `new Child()` where `Child extends Parent`, the first time:

`Parent` static field initialisers and static blocks, textual order.

`Child` static field initialisers and static blocks.

Then, for every `new`: `Parent` instance field initialisers and instance `{ }` blocks, then the `Parent` constructor body; then `Child` instance initialisers, then the `Child` constructor body.

Static parts once per class, top of the hierarchy first; instance parts per object, parent first. A constructor's implicit `super()` is what makes step 3 happen in that order, and calling an overridable method from a parent constructor sees the child's fields *still at their defaults* — the classic trap from the inheritance module, now explained by the order above.

The lazy-holder idiom

Because initialisation is lazy, once-only and thread-safe by specification, it is the cleanest way to build a lazily created singleton with no locks in your own code:

```
class Registry {
    private Registry() {}
    private static class Holder { static final Registry INSTANCE = new Registr
    static Registry get() { return Holder.INSTANCE; }   // Holder initialises
}
```

No `synchronized`, no `volatile`, no double-checked locking; the JVM's class-initialisation lock does the work. Enums get the same guarantee for free, which is why "an enum with one constant" is the other standard singleton.

The class-loader hierarchy

Loaders form a parent chain and use **parent-first delegation**: asked for a class, a loader first asks its parent; only if the whole chain above fails does it look itself.

Bootstrap loader (native, shown as `null`) — the core modules: `java.base` (`java.lang`, `java.util`, …).

Platform loader — the rest of the JDK modules (`java.sql`, `java.xml`, …).

Application (system) loader — your classpath / module path.

Custom loaders — application servers, plugin systems, IDEs; each web app in Tomcat has its own, so two apps can ship different versions of the same library.

Delegation is why you cannot replace `java.lang.String` by putting your own on the classpath — the bootstrap loader answers first — and why a class is identified by *loader plus name*: the same `.class` loaded by two loaders is two different classes, and casting between them fails with a bewildering `ClassCastException: com.x.Foo cannot be cast to com.x.Foo`.

`ClassNotFoundException` versus `NoClassDefFoundError`

`ClassNotFoundException` — a checked exception from an explicit lookup by name (`Class.forName("com.mysql.Driver")`, `loader.loadClass`) that found nothing. Usually a missing jar or a typo in a string.

`NoClassDefFoundError` — an `Error` when the JVM itself needs a class that was present at compile time and is missing at run time — or, the sneaky case, was found but its **static initialiser threw**: the first attempt raises `ExceptionInInitializerError`, and every later use of the class fails with `NoClassDefFoundError: Could not initialize class X`. When you see the second message, look for the first.

The JIT: interpreted, then compiled

`javac` produces bytecode; the JVM starts by **interpreting** it. HotSpot counts invocations and loop iterations, and when a method gets hot it is compiled to native code — first by **C1** (fast compile, modest optimisation), then, if it stays hot, by **C2** (slow compile, aggressive optimisation). This **tiered compilation** gives quick start-up and peak speed. Compiled code is kept in the *code cache*; `-XX:+PrintCompilation` shows it happening.

What C2 does is why "Java is slow" stopped being true around 2005:

Inlining — small methods (getters, lambdas, `Integer.intValue`) are copied into the caller; the call vanishes. This is what makes streams as fast as loops when warm.

Escape analysis — an object that provably never leaves the method may be allocated on the stack or dissolved into locals (*scalar replacement*); short-lived `Point`s and iterators often cost nothing.

Speculative optimisation — if a call site has only ever seen one class, compile a direct call with a *guard*; if a new class shows up, **deoptimise** back to the interpreter and recompile. Virtual calls in Java are usually as cheap as static ones for this reason.

Loop unrolling, bounds-check elimination, dead-code elimination — a benchmark whose result is never used may be optimised to nothing, taking 0 ns and proving nothing.

Why microbenchmarks lie, and what to do

Time a method once with `System.nanoTime()` and you measure the interpreter, class loading and a cold cache. Time it in a loop and the JIT compiles it half-way through, or removes it because the result is unused. Use **JMH** (the OpenJDK harness): it warms up, runs many forks, prevents dead-code elimination with `Blackhole`, and reports a distribution, not a number. In an interview, the sentence "I would measure that with JMH after warm-up" earns more than any guess about which of two snippets is faster.

Interview angle

"When does a static block run?" At class initialisation — first active use (`new`, static method, non-constant static field), once, thread-safely; not for constants or `Foo.class`.

"Order for `new Child()`?" Parent statics, Child statics (once); then Parent instance init + constructor, Child instance init + constructor.

"How does class loading work?" Bootstrap → platform → application loaders, parent-first delegation; identity is loader + name.

"`ClassNotFoundException` versus `NoClassDefFoundError`?" Explicit lookup failed versus the JVM needing a class that is missing or failed to initialise.

"What does the JIT do?" Interprets first, compiles hot code with C1 then C2; inlining, escape analysis, speculative optimisation with deoptimisation.

Key takeaways

Load → link (verify, prepare, resolve) → initialise; initialisation is lazy, once, thread-safe, triggered by first active use.

Statics top-down once; per object, parent init + constructor before child. The holder idiom rides on this.

Parent-first delegation; a class is (loader, name); core classes cannot be shadowed.

`NoClassDefFoundError: Could not initialize class` means a static initialiser threw earlier.

Tiered JIT: interpreter → C1 → C2; inlining and escape analysis make abstractions free once warm; benchmark with JMH.

EXERCISE 1

Parent-first delegation

Simulate a class-loader chain. Read an integer `k` and `k` lines `<loader> <parent> <classes>`, where `parent` is `-` for the bootstrap loader and `classes` is a comma-separated list of the class names that loader can find itself (or `-`). Then read an integer `q` and `q` requests `<loader> <class>`. Resolve each with **parent-first delegation**: the loader asks its parent first, all the way up; only if no ancestor finds the class does it look itself. Print `<class> loaded by <loader>` or `<class>: ClassNotFoundException`.

Example:

```
3
boot - String,Object
app boot Main,Util
plugin app Plugin,Util
3
plugin String
plugin Util
plugin Foo
```

→

```
String loaded by boot
Util loaded by app
Foo: ClassNotFoundException
```

INPUT 1

```
3
boot - String,Object
app boot Main,Util
plugin app Plugin,Util
3
plugin String
plugin Util
plugin Foo
```

OUTPUT 1

```
String loaded by boot
Util loaded by app
Foo: ClassNotFoundException
```

INPUT 2

```
2
boot - String
app boot String
2
app String
boot Main
```

OUTPUT 2

```
String loaded by boot
Main: ClassNotFoundException
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    static Map<String, String> parent = new HashMap<>();
    static Map<String, Set<String>> owns = new HashMap<>();

    static String load(String loader, String cls) {
        if (loader == null) return null;
        String fromParent = load(parent.get(loader), cls);
        if (fromParent != null) return fromParent;
        return owns.get(loader).contains(cls) ? loader : null;
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int k = in.nextInt();
        for (int i = 0; i < k; i++) {
            String name = in.next(), p = in.next(), classes = in.next();
            parent.put(name, p.equals("-") ? null : p);
            owns.put(name, classes.equals("-") ? Set.of() : new HashSet<>(Arrays.asL
        }
        int q = in.nextInt();
        for (int i = 0; i < q; i++) {
            String loader = in.next(), cls = in.next();
            String by = load(loader, cls);
            System.out.println(by == null ? cls + ": ClassNotFoundException" : cls +
        }
    }
}
```

EXERCISE 2

Initialisation order, proven

Write two classes, `Parent` and `Child extends Parent`, such that:

`Parent` has a static initialiser that prints `static Parent`, an instance initialiser block that prints `init Parent`, and a constructor that prints `ctor Parent`;

`Child` has the same three, printing `static Child`, `init Child` and `ctor Child`, plus a compile-time constant `static final int CONST = 7`.

The given `main` reads an integer `n`, prints `const=<Child.CONST>` (which must **not** trigger initialisation), then creates `n` `Child` objects. Make the output match the JVM's real order: statics once, parent first; per object, parent init and constructor before child.

Example: `1` →

```
const=7
static Parent
static Child
init Parent
ctor Parent
init Child
ctor Child
```

INPUT 1

```
1
```

OUTPUT 1

```
const=7
static Parent
static Child
init Parent
ctor Parent
init Child
ctor Child
```

INPUT 2

```
0
```

OUTPUT 2

```
const=7
```

REFERENCE SOLUTION

```java
import java.util.*;

class Parent {
    static { System.out.println("static Parent"); }
    { System.out.println("init Parent"); }
    Parent() { System.out.println("ctor Parent"); }
}

class Child extends Parent {
    static final int CONST = 7;
    static { System.out.println("static Child"); }
    { System.out.println("init Child"); }
    Child() { System.out.println("ctor Child"); }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        System.out.println("const=" + Child.CONST);
        for (int i = 0; i < n; i++) new Child();
    }
}
```

QUIZ

1. Which of these triggers class initialisation of `Foo`?

A. `Foo.class`

B. `int x = Foo.MAX;` where `MAX` is `static final int MAX = 3`

C. `Foo f = null;`

D. `Foo.helper();` — a static method call

2. For `new Child()` (first ever), the order is…

A. Child statics, Parent statics, Child ctor, Parent ctor

B. Parent statics, Child statics, Parent init + ctor, Child init + ctor

C. Parent ctor, Child ctor, then statics

D. Undefined

3. Parent-first delegation means…

A. The child loads first

B. A loader asks its parent before looking itself, so core classes cannot be shadowed

C. Classes load in alphabetical order

D. Only the bootstrap loader loads

4. `NoClassDefFoundError: Could not initialize class X` usually means…

A. The jar is missing

B. X's static initialiser threw earlier (`ExceptionInInitializerError`) and X is now unusable

C. X is abstract

D. X is in the wrong package

5. Tiered compilation in HotSpot means…

A. Bytecode is compiled ahead of time

B. Code is interpreted first, then compiled by C1 and, if still hot, by C2

C. Only loops are compiled

D. The JIT runs once at start-up

6. A benchmark timing a method once with `System.nanoTime()` is unreliable because…

A. `nanoTime` is inaccurate

B. It measures the interpreter and class loading, and the JIT may later compile or eliminate the code — use JMH

C. Java has no timers

D. Methods cannot be timed

ANSWER KEY · CLASS LOADING, INITIALISATION ORDER AND THE JIT

1. D First active use: `new`, a static method, a non-constant static field. Constants are inlined; `Foo.class` and declarations do nothing.

2. B Statics top-down once; then per object the parent's initialisers and constructor complete before the child's.

3. B Which is why your own `java.lang.String` on the classpath is never used.

4. B Find the first failure in the log; every later use of the class fails with this error.

5. B Quick start with the interpreter, peak speed from C2's inlining, escape analysis and speculative optimisation.

6. B Warm-up, forks, dead-code prevention and statistical reporting are what JMH provides.

LESSON 6 · 14 MIN

Diagnosing memory problems — leaks, dumps and bounded caches

Everything so far has been the model. This lesson is what you do at two in the morning when the service falls over with `OutOfMemoryError`, and, more usefully, what you build in advance so it does not. The skills: reading the different OOM messages, taking and reading a heap dump, spotting the six leak shapes in code review, and writing the bounded structures — an LRU cache above all — that keep memory finite by construction.

The messages, and what each means

Message	Meaning	First move
`Java heap space`	live objects exceed `-Xmx`	heap dump; find the biggest retained set
`GC overhead limit exceeded`	>98% of time in GC, <2% recovered — effectively the same as above	same
`Metaspace`	class metadata exhausted — classes loaded and never unloaded	class-loader leak: redeploys, dynamic proxies, generated classes
`unable to create native thread`	OS thread limit or per-thread stack memory exhausted	thread leak: unbounded `new Thread`, executors never shut down
`Direct buffer memory`	`ByteBuffer.allocateDirect` beyond `-XX:MaxDirectMemorySize`	buffers not released; NIO frameworks
`Requested array size exceeds VM limit`	asked for an array near `Integer.MAX_VALUE`	a size computation bug, or an unbounded read into one array

Only the first two are about your objects; the others are about resources the heap size does not govern. Raising `-Xmx` fixes none of them and merely delays a leak.

Taking the evidence

`-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/dumps` — the flags to have on *before* the incident. A dump is a snapshot of every object and reference; without it you are guessing.

`jcmd <pid> GC.heap_dump /tmp/heap.hprof` — on demand, from a live process. `jmap -dump:live,format=b,file=…` is the older spelling; `live` forces a GC first so you see only reachable objects.

`jcmd <pid> GC.class_histogram` (or `jmap -histo:live <pid>`) — the quick look: instance counts and bytes per class. If the top line is `[B` (byte arrays) or `String`, look at who *holds* them, not at the strings.

`jstat -gcutil <pid> 1s` — live occupancy of each generation and GC counts per second. Old-generation occupancy that climbs after every full GC and never drops is the signature of a leak.

JFR (`-XX:StartFlightRecording`) and **JDK Mission Control** — continuous low-overhead recording of allocations, GC, threads; the production-safe profiler.

Eclipse MAT or **VisualVM** to open the dump: the *dominator tree* answers "which object keeps the most memory alive", and *path to GC roots* explains why it is still reachable.

Reading a histogram

```
num     #instances         #bytes  class name
  1:       1204411       57811728  [B
  2:       1204013       28896312  java.lang.String
  3:         12034       19311776  [Ljava.lang.Object;
  4:          9021        1010352  java.util.HashMap$Node
```

A million strings is not a diagnosis; who holds a million strings is. The `Object[]` at line 3 — 12 000 arrays averaging 1.6 KB — is likelier to be the interesting party: the backing arrays of `ArrayList`s. In the dominator tree you would find, say, one `ConcurrentHashMap` in a `static` field retaining 80 MB. That field is the leak.

The six leak shapes to catch in review

The growing static collection — a cache, registry or "seen" set with `put` and no `remove`, no bound, no expiry.

The unremoved listener — `subject.addListener(this)` in a constructor with no matching removal; the long-lived subject keeps every short-lived listener.

`ThreadLocal` **in a pool** — set per request, never `remove()`d; the pooled thread lives for days.

The unclosed resource — streams, connections, `Scanner`s, `ExecutorService`s without `shutdown()`. Try-with-resources for the first three; an explicit lifecycle for the last.

The captured outer — a lambda or inner class stored in a static or a long-lived collection, dragging its enclosing object along.

The mutable key — a `HashMap` key whose `hashCode` changed after insertion: the entry is now unreachable through `get` and unremovable through `remove`, so it is a leak *and* a bug.

Each has a two-line fix. Finding them is a matter of asking, for every collection that lives longer than a request, "what removes things from this?"

Bounded by construction: the LRU cache

The standard answer to "cache without leaking" is a bounded, least-recently-used map, and Java's `LinkedHashMap` was designed to be one:

```
class Lru<K, V> extends LinkedHashMap<K, V> {
    private final int capacity;
    Lru(int capacity) {
        super(16, 0.75f, true);          // accessOrder = true: get() moves th
        this.capacity = capacity;
    }
    @Override protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capacity;        // called after every put; true evict
    }
}
```

`accessOrder = true` makes both `get` and `put` move an entry to the most-recent end; `removeEldestEntry` is consulted after each insertion. Twenty lines, O(1) per operation, and the interview's "design an LRU cache" solved with the standard library — then be ready to describe the hand-built version (a `HashMap` plus a doubly linked list) that it wraps. For a concurrent or expiry-based cache, use Caffeine; do not write one.

Other bounded shapes: a `Deque` trimmed to the last *n* events; a `PriorityQueue` kept at size *k* for top-k; `WeakHashMap` for metadata about objects you do not own; a `SoftReference` only when you have measured that it helps.

Allocation hygiene in hot paths

Prefer primitives and primitive arrays; `IntStream`, not `Stream<Integer>`; `int` counters, not `Integer`.

Reuse buffers (`byte[]`, `StringBuilder.setLength(0)`) across iterations of a hot loop when the profiler says allocation matters — and only then; premature reuse is how shared mutable state is born.

Pre-size collections when you know the count: `new ArrayList<>(n)`, `new HashMap<>(n * 4 / 3 + 1)`.

Avoid `String.format` and regex in tight loops; both allocate freely.

Stream `Files.lines` instead of `readAllLines` for large files; read a big input once with `readAllBytes` rather than line by line into a growing list you then join.

Interview angle

"How would you find a memory leak?" Watch old-gen occupancy with `jstat`; take a heap dump (`-XX:+HeapDumpOnOutOfMemoryError` or `jcmd`); open it in MAT; the dominator tree names the retaining collection; trace the path to GC roots.

"Name common leak causes." Growing static collections, unremoved listeners, `ThreadLocal`s in pools, unclosed resources, captured outer instances, mutable keys.

"Design an LRU cache." `LinkedHashMap` in access order with `removeEldestEntry`; or a `HashMap` + doubly linked list by hand, both O(1).

"What is `GC overhead limit exceeded`?" The JVM spending almost all its time collecting and recovering almost nothing — the heap is full of live data.

"Does raising `-Xmx` fix an OOM?" Only when the working set genuinely needs it; for a leak it postpones the crash.

Key takeaways

Read the OOM message: heap, Metaspace, threads and direct memory are different problems.

Have dump-on-OOM on; `jcmd`, `jstat`, `jmap -histo`, JFR and MAT are the kit; the dominator tree names the culprit.

Six leak shapes; the review question is "what removes entries from this?"

Bound everything long-lived: LRU via `LinkedHashMap(accessOrder = true)` + `removeEldestEntry`.

Allocate less in hot paths — primitives, pre-sizing, streaming input — after measuring.

EXERCISE 1

LRU cache

Build a least-recently-used cache on `LinkedHashMap` with `accessOrder = true` and `removeEldestEntry`. Read the capacity `c`, then an integer `n` and `n` commands: `put <k> <v>` (when it evicts, print `evict <key>`); `get <k>` prints `get <k> → <v>` or `get <k> → miss`. Finally print `keys=<the keys from least to most recently used, as a list>` and `hits=<n> misses=<n>`.

Example: `c = 2`, then `6` commands: `put a 1`, `put b 2`, `get a`, `put c 3`, `get b`, `get c` →

```
get a → 1
evict b
get b → miss
get c → 3
keys=[a, c]
hits=2 misses=1
```

INPUT 1

```
2
6
put a 1
put b 2
get a
put c 3
get b
get c
```

OUTPUT 1

```
get a → 1
evict b
get b → miss
get c → 3
keys=[a, c]
hits=2 misses=1
```

INPUT 2

```
1
4
put x 1
put y 2
get x
get y
```

OUTPUT 2

```
evict x
get x → miss
get y → 2
keys=[y]
hits=1 misses=1
```

REFERENCE SOLUTION

```
import java.util.*;

class Lru<K, V> extends LinkedHashMap<K, V> {
    private final int capacity;
    Lru(int capacity) {
        super(16, 0.75f, true);
        this.capacity = capacity;
    }
    @Override protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        boolean evict = size() > capacity;
        if (evict) System.out.println("evict " + eldest.getKey());
        return evict;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int c = in.nextInt();
        int n = in.nextInt();
        Lru<String, String> cache = new Lru<>(c);
        int hits = 0, misses = 0;
        for (int i = 0; i < n; i++) {
            String cmd = in.next();
            if (cmd.equals("put")) {
                cache.put(in.next(), in.next());
            } else {
                String k = in.next();
                String v = cache.get(k);
                if (v == null) { misses++; System.out.println("get " + k + " → miss
                else { hits++; System.out.println("get " + k + " → " + v); }
            }
        }
        System.out.println("keys=" + cache.keySet());
        System.out.println("hits=" + hits + " misses=" + misses);
    }
}
```

EXERCISE 2

Heap histogram

Read allocation records `<class> <bytes>` — one object each — until end of input, and print a class histogram the way `jmap -histo` does: one line per class, `<rank>: <class> instances=<count> bytes=<total>`, ordered by total bytes descending then class name, then `total=<all bytes>` and `top=<share of the top class, whole percent>%`.

Example input

```
String 40
byte[] 1000
String 40
Object[] 500
byte[] 200
```

→

```
1: byte[] instances=2 bytes=1200
2: Object[] instances=1 bytes=500
3: String instances=2 bytes=80
total=1780
top=67%
```

INPUT 1

```
String 40
byte[] 1000
String 40
Object[] 500
byte[] 200
```

OUTPUT 1

```
1: byte[] instances=2 bytes=1200
2: Object[] instances=1 bytes=500
3: String instances=2 bytes=80
total=1780
top=67%
```

INPUT 2

```
A 10
B 10
```

OUTPUT 2

```
1: A instances=1 bytes=10
2: B instances=1 bytes=10
total=20
top=50%
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        Map<String, long[]> stats = new HashMap<>();   // class -> {count, bytes}
        while (in.hasNext()) {
            String cls = in.next();
            long bytes = in.nextLong();
            long[] s = stats.computeIfAbsent(cls, k -> new long[2]);
            s[0]++;
            s[1] += bytes;
        }
        List<Map.Entry<String, long[]>> rows = new ArrayList<>(stats.entrySet());
        rows.sort((a, b) -> {
            int c = Long.compare(b.getValue()[1], a.getValue()[1]);
            return c != 0 ? c : a.getKey().compareTo(b.getKey());
        });
        long total = 0;
        for (Map.Entry<String, long[]> e : rows) total += e.getValue()[1];
        int rank = 1;
        for (Map.Entry<String, long[]> e : rows) {
            System.out.println(rank++ + ": " + e.getKey() + " instances=" + e.getVal
        }
        System.out.println("total=" + total);
        long top = rows.isEmpty() ? 0 : rows.get(0).getValue()[1];
        System.out.println("top=" + (total == 0 ? 0 : top * 100 / total) + "%");
    }
}
```

QUIZ

1. `OutOfMemoryError: unable to create native thread` is fixed by…

A. Raising `-Xmx`

B. Finding the thread leak — unbounded `new Thread` or executors never shut down

C. Calling `System.gc()`

D. Reducing string literals

2. The single most useful flag to have on in production for memory incidents is…

A. `-Xss`

B. `-XX:+HeapDumpOnOutOfMemoryError`

C. `-verbose:class`

D. `-XX:+UseSerialGC`

3. In a heap-dump analyser, the *dominator tree* tells you…

A. Which classes have the most methods

B. Which objects keep the most memory alive (retained size)

C. The order of allocation

D. GC pause times

4. `new LinkedHashMap<>(16, 0.75f, true)` — the `true` means…

A. Thread-safe

B. Access order: `get` and `put` move an entry to the most-recent end, enabling LRU eviction via `removeEldestEntry`

C. Sorted keys

D. Weak keys

5. A `HashMap` key mutated after insertion so that its `hashCode` changes…

A. Is rehashed automatically

B. Sits in the wrong bucket: unreachable via `get`, unremovable via `remove` — a leak and a bug

C. Throws `ConcurrentModificationException`

D. Is moved to the end

ANSWER KEY · DIAGNOSING MEMORY PROBLEMS — LEAKS, DUMPS AND BOUNDED CACHES

1. B Thread stacks are off-heap; the heap size has nothing to do with it.

2. B Without a dump taken at the moment of failure you are guessing at the retaining structure.

3. B Follow the largest retained set to the static field or collection at its root — that is the leak.

4. B With `false` (the default) the map keeps insertion order and eviction would be FIFO.

5. B Keys must be immutable in the fields their `hashCode` uses.

CHECKPOINT · 30 MIN

Checkpoint — Memory and the JVM runtime

This checkpoint covers the stack and the heap, aliasing and pass-by-value, object layout, reachability and the four reference strengths, the leak shapes, generational garbage collection and the collectors, the string pool and the Integer cache, `StringBuilder` growth, class loading and initialisation order, the JIT, and the diagnosis toolkit with the LRU cache.

How it works. Fifteen questions and three programs; 70% on the questions and every program accepted clears the module.

Before you start, make sure you can answer:

- Where a local `int`, a local reference and the object it points to each live.
- Why two references to one array are an alias, and what a method can and cannot do to a caller's variable.
- What makes an object garbage, and why a cycle does not keep it alive.
- Weak versus soft references, and why `WeakHashMap` with `String` literal keys never shrinks.
- Minor versus full GC, and what a log full of `Pause Full` lines means.
- Why `"a" == "a"` but `new String("a") != "a"`, and where the Integer cache ends.
- The exact order of static and instance initialisers for `new Child()`.
- The two-line `LinkedHashMap` recipe for an LRU cache.

The programs are a tracing collector that decides what reference counting would have leaked, a bounded LRU cache with hit statistics, and an intern table that reports every hit and miss.

EXERCISE 1

Tracing versus counting

Compare the two collection strategies on one object graph. Read an integer `n` and `n` object names; `r` and `r` roots; `e` and `e` edges `from to`. Compute:

- **Tracing** frees every object not reachable from a root.
- **Reference counting** gives each object a count = number of incoming edges + 1 if it is a root; frees every non-root object whose count is 0, and when an object is freed decrements the count of each object it points to, freeing those that reach 0 — repeat until stable.

Print `tracing=<freed, natural order>`, `counting=<freed, natural order>` and `leaked=<objects tracing frees but counting does not>` (`-` for an empty list). The leaked ones are exactly the garbage cycles.

Example: `5` then `a b c d e`, `1` then `a`, `4` then `a b`, `c d`, `d c`, `d e` →

```
tracing=c d e
counting=-
leaked=c d e
```

INPUT 1

```
5
a b c d e
1
a
4
a b
c d
d c
d e
```

OUTPUT 1

```
tracing=c d e
counting=-
leaked=c d e
```

INPUT 2

```
3
x y z
1
x
1
y z
```

OUTPUT 2

```
tracing=y z
counting=y z
leaked=-
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    static String show(Collection<String> c) {
        return c.isEmpty() ? "-" : String.join(" ", new TreeSet<>(c));
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        List<String> objects = new ArrayList<>();
        for (int i = 0; i < n; i++) objects.add(in.next());
        int r = in.nextInt();
        Set<String> roots = new HashSet<>();
        for (int i = 0; i < r; i++) roots.add(in.next());
        int e = in.nextInt();
        Map<String, List<String>> out = new HashMap<>();
        Map<String, Integer> count = new HashMap<>();
        for (String o : objects) count.put(o, roots.contains(o) ? 1 : 0);
        for (int i = 0; i < e; i++) {
            String from = in.next(), to = in.next();
            out.computeIfAbsent(from, k -> new ArrayList<>()).add(to);
            count.merge(to, 1, Integer::sum);
        }

        // Tracing: mark from the roots.
        Set<String> marked = new HashSet<>();
        Deque<String> work = new ArrayDeque<>(roots);
        while (!work.isEmpty()) {
            String o = work.pop();
            if (marked.add(o)) work.addAll(out.getOrDefault(o, List.of()));
        }
        Set<String> tracing = new HashSet<>(objects);
        tracing.removeAll(marked);

        // Counting: free zero-count objects and cascade.
        Set<String> counting = new HashSet<>();
        Deque<String> zero = new ArrayDeque<>();
        for (String o : objects) if (count.get(o) == 0) zero.add(o);
        while (!zero.isEmpty()) {
            String o = zero.pop();
            if (!counting.add(o)) continue;
            for (String child : out.getOrDefault(o, List.of())) {
                if (count.merge(child, -1, Integer::sum) == 0) zero.add(child);
            }
        }

        Set<String> leaked = new HashSet<>(tracing);
        leaked.removeAll(counting);
```

```
        System.out.println("tracing=" + show(tracing));
        System.out.println("counting=" + show(counting));
        System.out.println("leaked=" + show(leaked));
    }
}
```

EXERCISE 2

Compact the heap

Simulate a mark–compact collector's sliding phase. Read an integer `n` and `n` blocks `<id> <size> <live>` in address order, starting at address 0 (`live` is 1 or 0). Live blocks slide down to the lowest free address, keeping their order; dead blocks are dropped. Print one line per live block `<id>: <old address> → <new address>`, then `freed=<total size of dead blocks>` and `top=<the first free address after compaction>`.

Example: `4` then `a 16 1`, `b 32 0`, `c 8 1`, `d 24 1` →

```
a: 0 → 0
c: 48 → 16
d: 56 → 24
freed=32
top=48
```

INPUT 1

```
4
a 16 1
b 32 0
c 8 1
d 24 1
```

OUTPUT 1

```
a: 0 → 0
c: 48 → 16
d: 56 → 24
freed=32
top=48
```

INPUT 2

```
2
x 8 0
y 8 0
```

OUTPUT 2

```
freed=16
top=0
```

REFERENCE SOLUTION

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        int address = 0, top = 0, freed = 0;
        for (int i = 0; i < n; i++) {
            String id = in.next();
            int size = in.nextInt();
            boolean live = in.nextInt() == 1;
            if (live) {
                System.out.println(id + ": " + address + " -> " + top);
                top += size;
            } else {
                freed += size;
            }
            address += size;
        }
        System.out.println("freed=" + freed);
        System.out.println("top=" + top);
    }
}
```

EXERCISE 3

An intern table

Implement a string pool. Read an integer `n` and `n` words. Keep a `HashMap<String, String>` pool; for each word, if an equal string is already pooled print `hit <word>` and reuse it, otherwise print `miss <word>` and add it. After all words print `distinct=<pool size>` and `saved=<total characters of the words that were hits — the memory interning avoided>`. Then, using the pooled references, print `identical=<true if the pooled reference for the first word == the pooled reference for the last word>`.

Example: `4` then `java go java rust` →

```
miss java
miss go
hit java
miss rust
distinct=3
saved=4
identical=false
```

INPUT 1

```
4
java go java rust
```

OUTPUT 1

```
miss java
miss go
hit java
miss rust
distinct=3
saved=4
identical=false
```

INPUT 2

```
3
ab cd ab
```

OUTPUT 2

```
miss ab
miss cd
hit ab
distinct=2
saved=2
identical=true
```

REFERENCE SOLUTION

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        Map<String, String> pool = new HashMap<>();
        List<String> pooled = new ArrayList<>();
        int saved = 0;
        for (int i = 0; i < n; i++) {
            String w = in.next();
            String existing = pool.get(w);
            if (existing != null) {
                System.out.println("hit " + w);
                saved += w.length();
                pooled.add(existing);
            } else {
                System.out.println("miss " + w);
                pool.put(w, w);
                pooled.add(w);
            }
        }
        System.out.println("distinct=" + pool.size());
        System.out.println("saved=" + saved);
        System.out.println("identical=" + (pooled.get(0) == pooled.get(pooled.size()
    }
}
```

CHECKPOINT QUESTIONS

1. Where does a local `double d = 1.5;` live?

A. On the heap

B. In the current stack frame

C. In the string pool

D. In Metaspace

2. Predict the output:

```
int[] a = {1}; int[] b = a; b[0] = 2; b = new int[]{3};
System.out.println(a[0] + " " + b[0]);
```

A. `1 3`

B. `2 3`

C. `3 3`

D. `2 2`

3. Java decides an object is garbage when…

A. Its reference count is zero

B. No path of references leads to it from a GC root

C. It is older than the tenuring threshold

D. `finalize` returns

4. A `SoftReference` differs from a `WeakReference` in that it…

A. Is never cleared

B. Is cleared only when memory is short, not at the next GC

C. Is cleared sooner

D. Holds primitives

5. Objects are promoted to the old generation when…

A. They exceed 1 KB

B. They survive enough minor collections to reach the tenuring threshold

C. They are strings

D. The program calls `System.gc()`

6. Which collector aims at sub-millisecond pauses regardless of heap size?

A. Serial

B. Parallel

C. ZGC

D. The interpreter

7. `String s = new Scanner(System.in).next(); s == "java"` when the input is `java` is…

A. `true`

B. `false` — the token is a new object, not the pooled literal

C. A compile error

D. `true` after `trim()`

8. `Integer.valueOf(127) == Integer.valueOf(127)` and `Integer.valueOf(128) == Integer.valueOf(128)` are…

A. `true`, `true`

B. `true`, `false` by default

C. `false`, `false`

D. `false`, `true`

9. A `StringBuilder` at capacity 16 receives 17 characters. Its capacity becomes…

A. 17

B. 32

C. 34

D. 64

10. Reading `Foo.LIMIT` where `static final int LIMIT = 100` …

A. Initialises `Foo`

B. Does not initialise `Foo` — the constant is inlined at compile time

C. Loads `Foo` twice

D. Throws

11. Which line is printed **first** on the first `new Child()` ?

A. `init Child`

B. `static Parent`

C. `ctor Parent`

D. `static Child`

12. Two classes with the same fully qualified name loaded by different loaders are…

A. The same class

B. Different classes — a class is identified by loader plus name

C. Merged

D. A compile error

13. Escape analysis lets the JIT…

A. Detect infinite loops

B. Avoid heap allocation for objects that never leave a method

C. Skip verification

D. Compile faster

14. `OutOfMemoryError: Metaspace` points to…

A. Too many strings

B. Classes loaded and never unloaded — a class-loader leak

C. A deep recursion

D. A large array

15. The `LinkedHashMap` LRU recipe needs…

A. `accessOrder = true` and an overridden `removeEldestEntry`

B. A `TreeMap`

C. `Collections.synchronizedMap`

D. A `PriorityQueue`

ANSWER KEY · CHECKPOINT — MEMORY AND THE JVM RUNTIME

1. B Primitive locals are slots in the frame; only objects live on the heap.

2. B `b[0] = 2` writes through the alias; `b = new int[]{3}` rebinds `b` only.

3. B Tracing from roots; cycles are irrelevant.

4. B Soft references are a (poor) cache; weak references are for canonicalising maps and listener lists.

5. B Age is kept in the object header and incremented on each survival.

6. C ZGC (and Shenandoah) do almost all work concurrently; G1 targets a configurable pause of ~200 ms.

7. B Only `s.intern() == "java"` would be true. Compare with `equals`.

8. B The cache covers −128..127; larger values are freshly allocated unless `-XX:AutoBoxCacheMax` raises the bound.

9. C `max(17, 2 × 16 + 2) = 34`.

10. B Only active uses initialise: `new`, static methods, non-constant static fields.

11. B Superclass static initialisation precedes everything else.

12. B Hence the puzzling `ClassCastException: X cannot be cast to X` in plugin systems.

13. B Scalar replacement dissolves such objects into locals — one reason short-lived temporaries are cheap.

14. B Typical in containers that redeploy applications without releasing the old loader.

15. A Access order moves used entries to the tail; `removeEldestEntry` returning `size() > capacity` evicts the head.

CodeKairo · Java study plan · Module 16: Memory and the JVM runtime. Solve the exercises online to track your progress and earn XP.