Garbage **Collection**

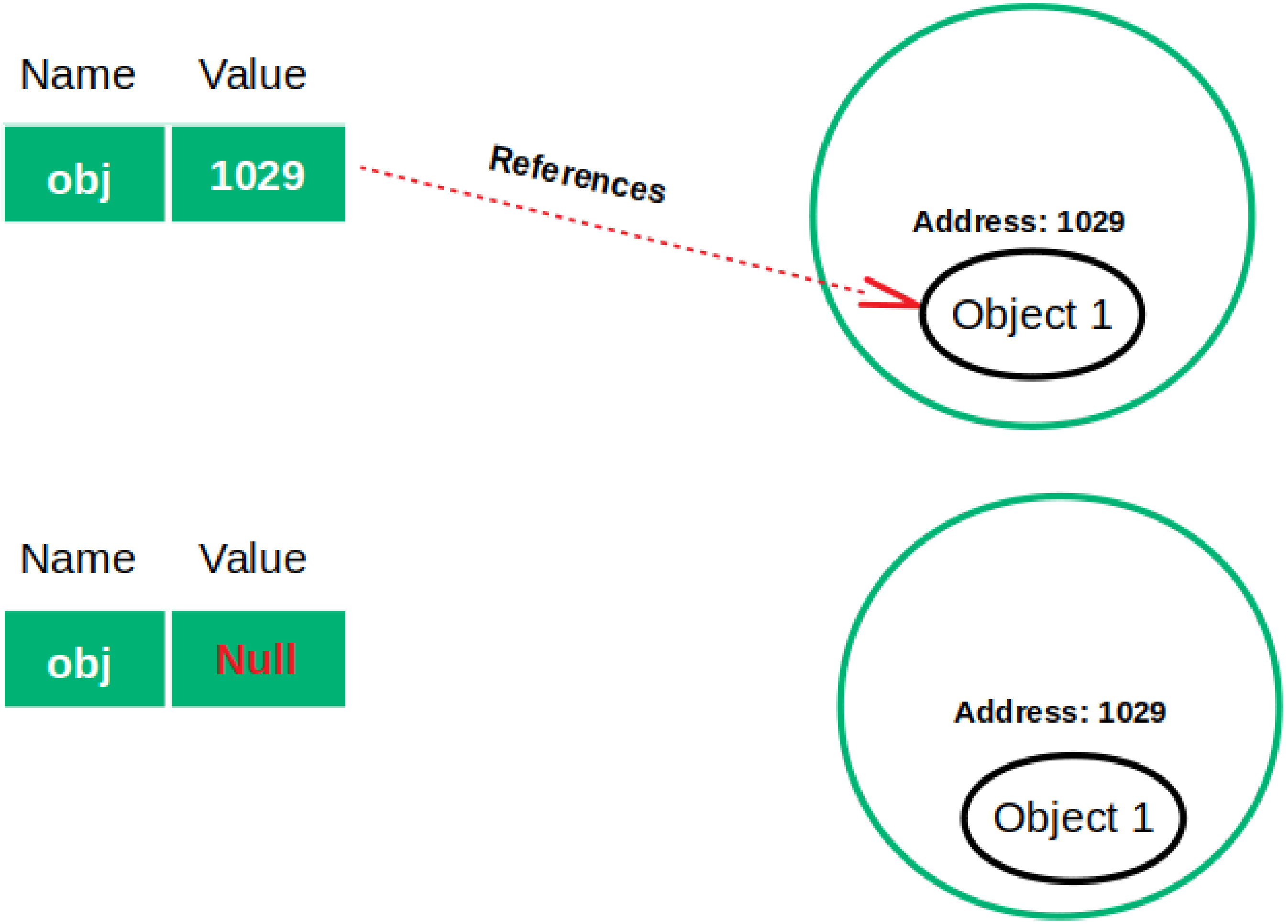
What is Garbage Collection?

- Garbage collector is used to delete unreferenced objects from heap.

- Java garbage collection is an automatic process perform by JVM.

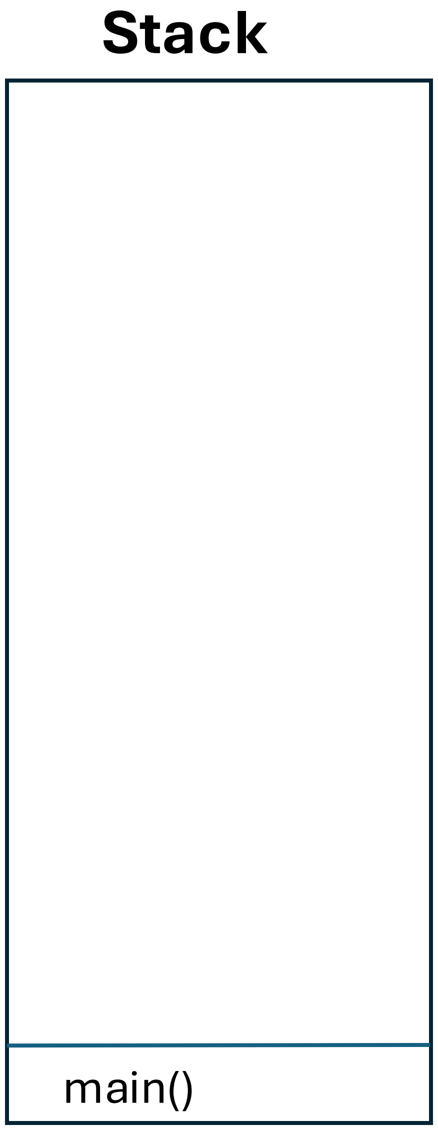
- The main objective of garbage collector is to free heap memory by destroying unreachable objects.

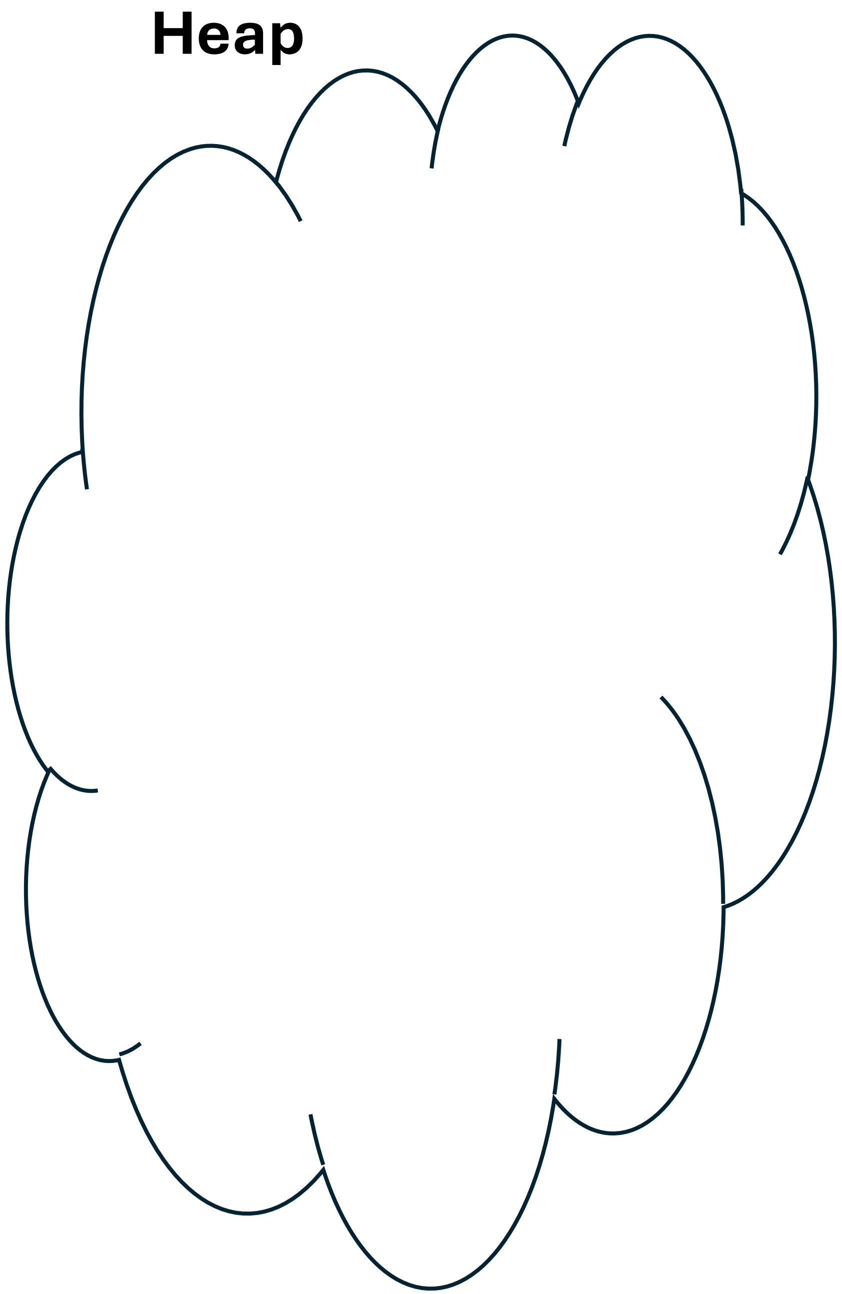
Referenced and unreferenced objects



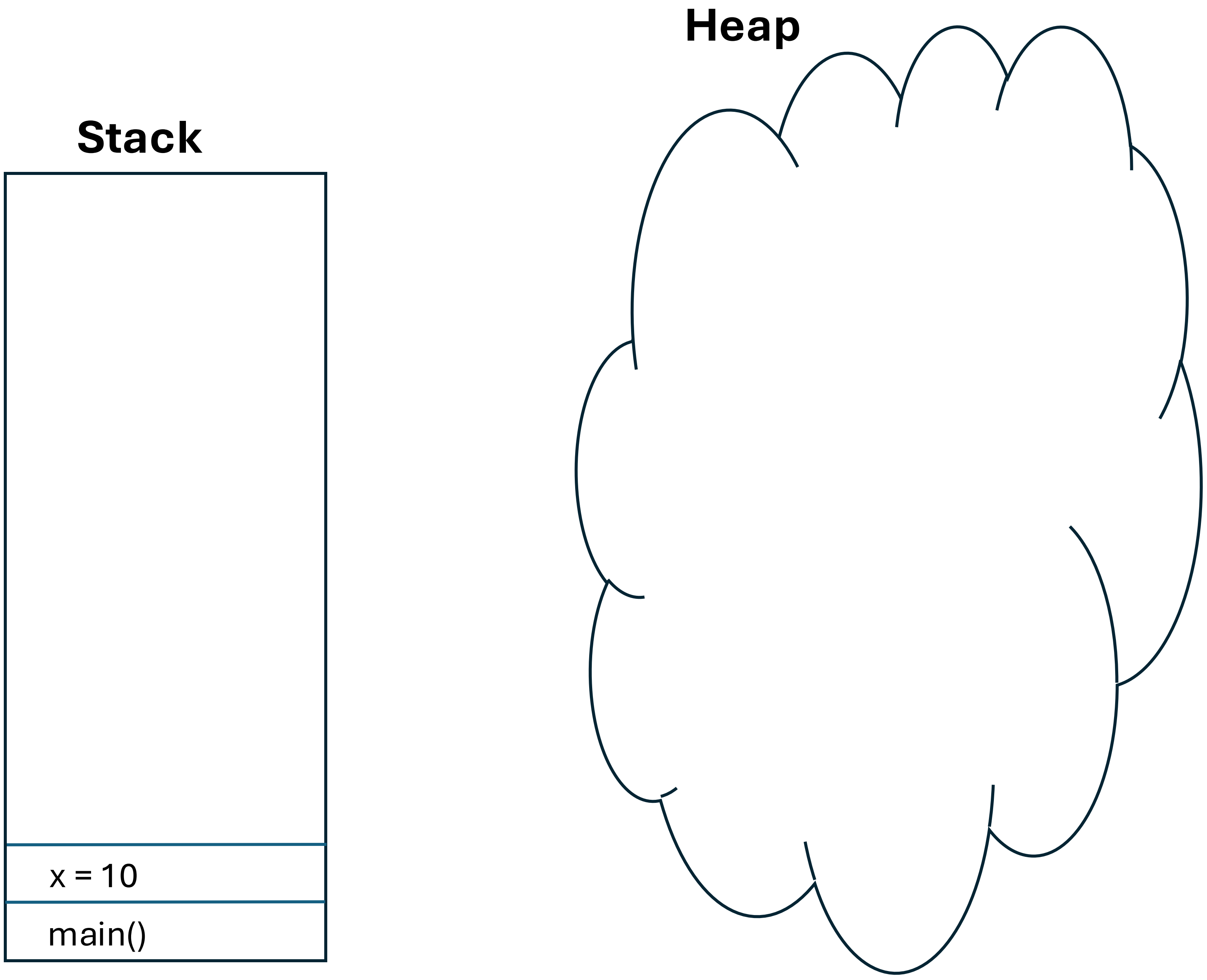

Ordered, on top of eachother!
No particular order!
Stack
Heap

```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        int x = 10;
        String s1 = "24";
        Demo demo = new Demo();
        demo.m1(demo);
    }

    private void m1(Demo demo){
        Demo demo2 = demo;
        String s2 = "24";
        String s3 = new String(original:"24");
    }
}
```

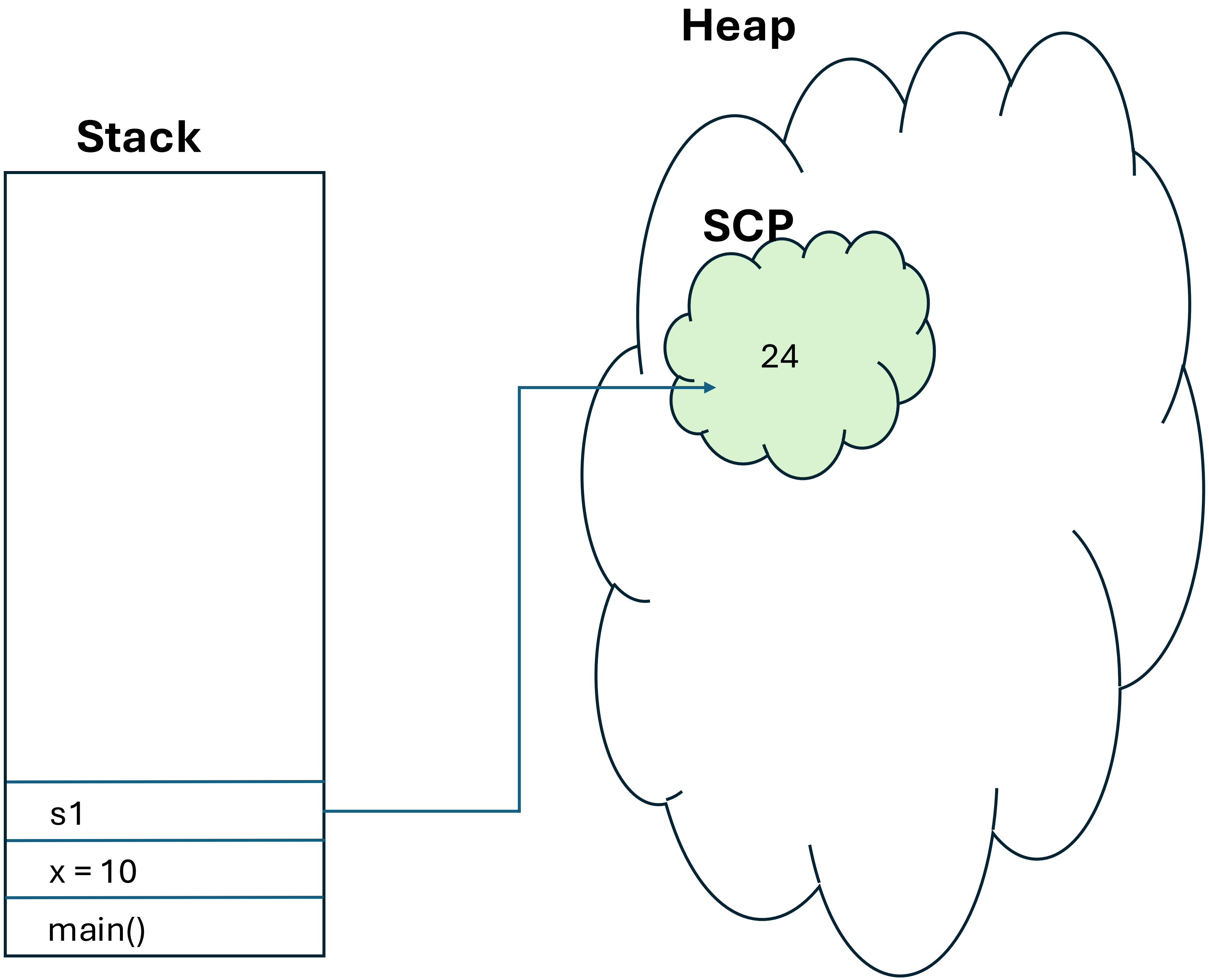




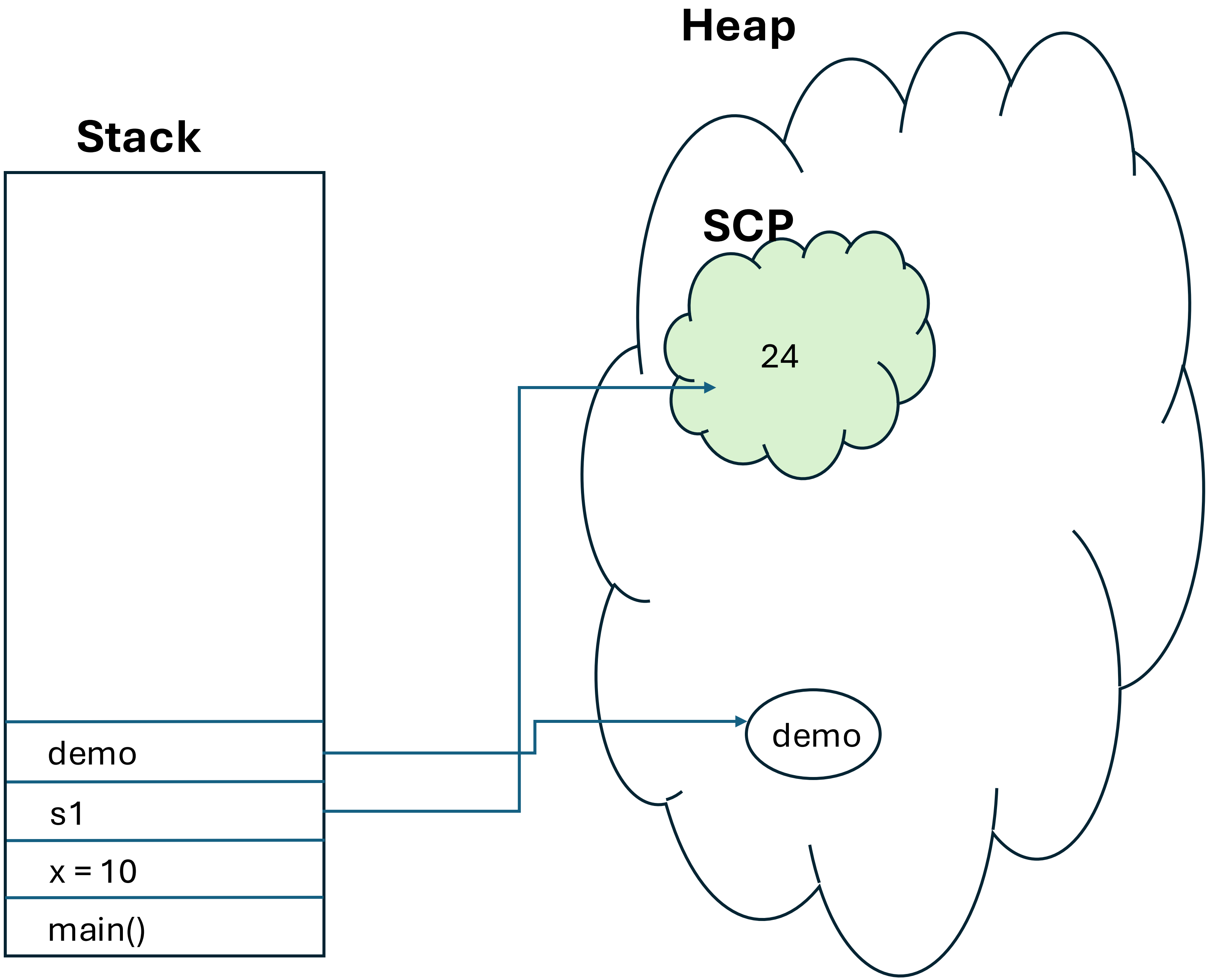
```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        int x = 10;
        String s1 = "24";
        Demo demo = new Demo();
        demo.m1(demo);
    }

    private void m1(Demo demo){
        Demo demo2 = demo;
        String s2 = "24";
        String s3 = new String(original:"24");
    }
}
```



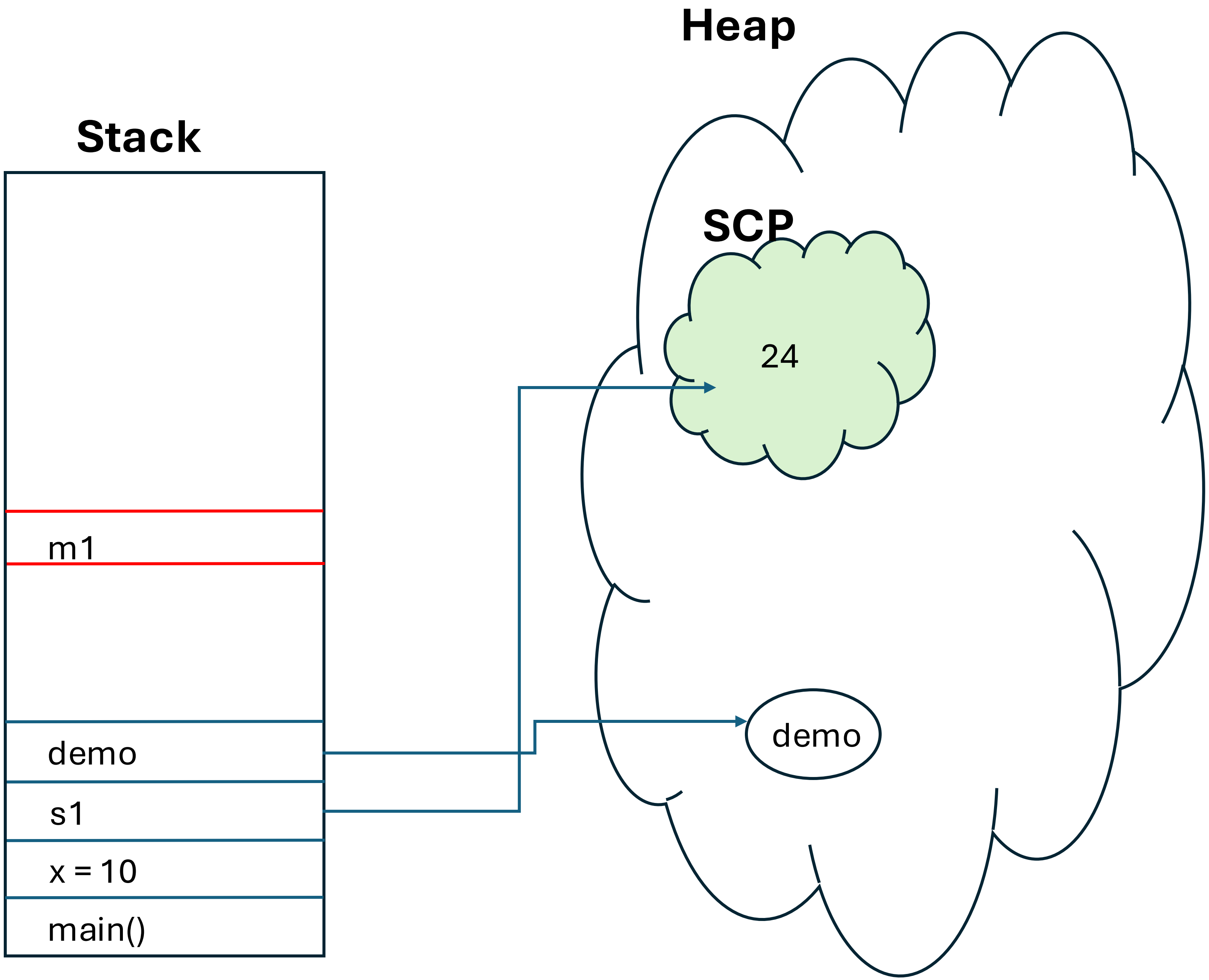
```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        int x = 10;
        String s1 = "24";
        Demo demo = new Demo();
        demo.m1(demo);
    }

    private void m1(Demo demo){
        Demo demo2 = demo;
        String s2 = "24";
        String s3 = new String(original:"24");
    }
}
```



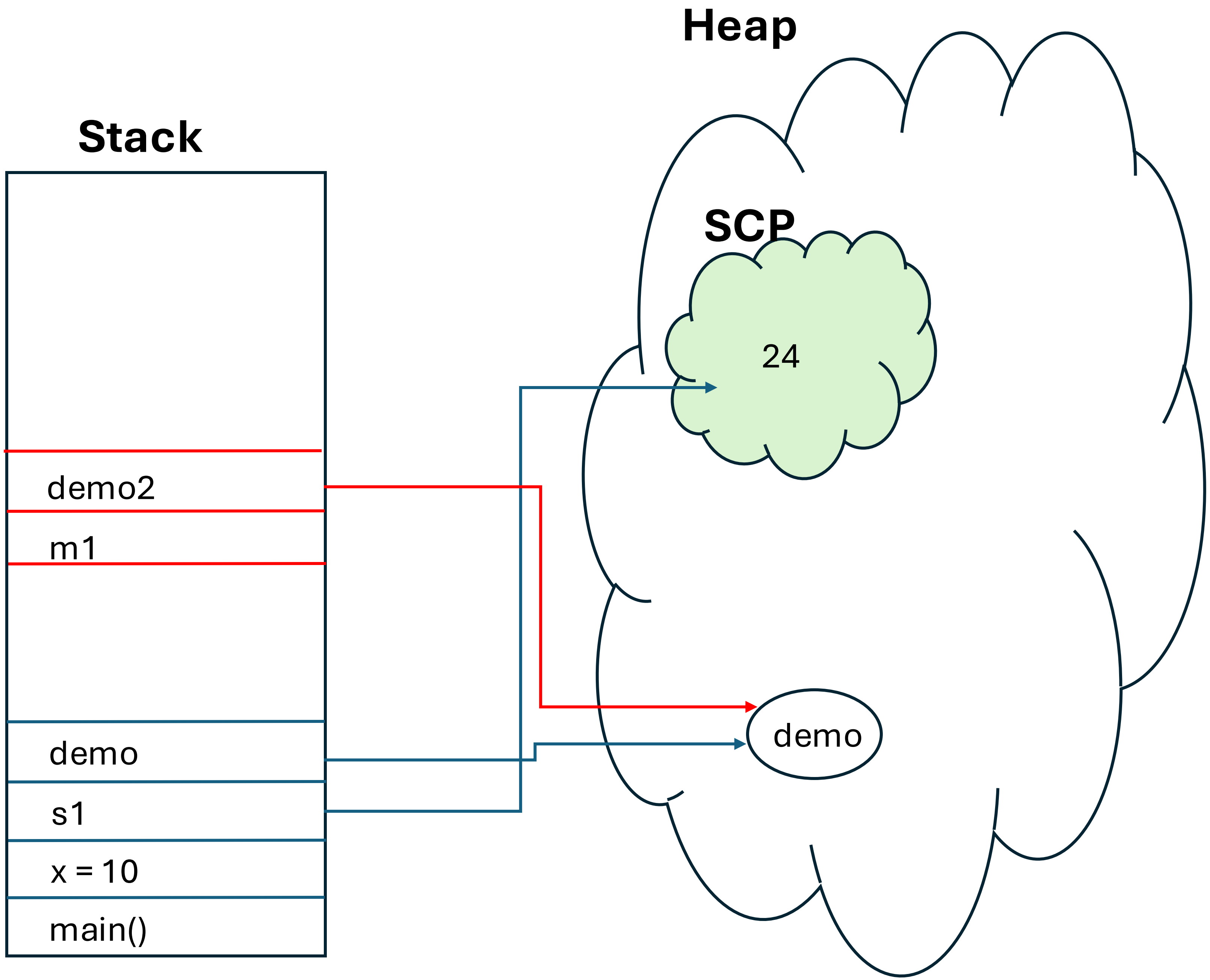
```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        int x = 10;
        String s1 = "24";
        Demo demo = new Demo();
        demo.m1(demo);
    }

    private void m1(Demo demo){
        Demo demo2 = demo;
        String s2 = "24";
        String s3 = new String(original:"24");
    }
}
```



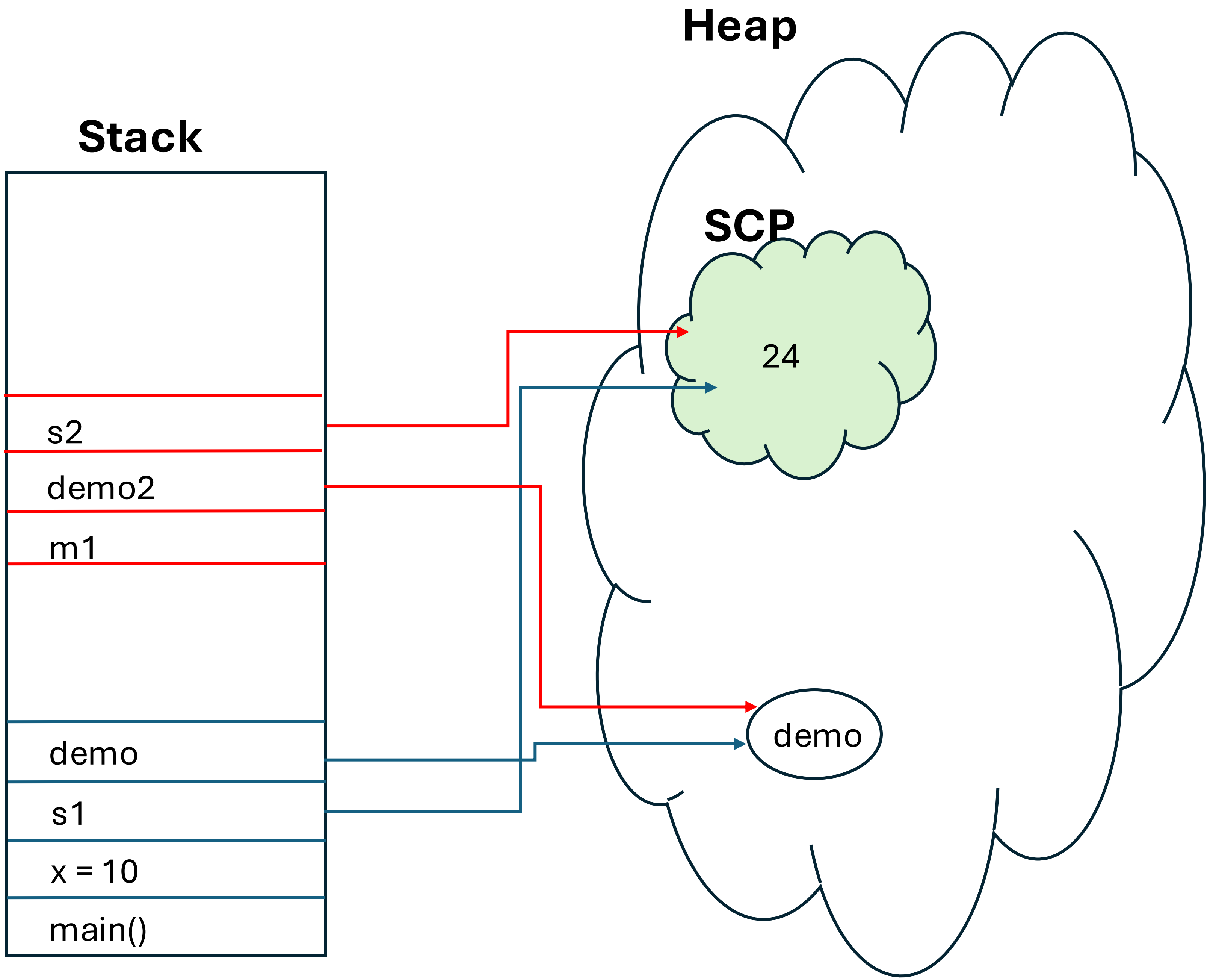
```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        int x = 10;
        String s1 = "24";
        Demo demo = new Demo();
        demo.m1(demo);
    }

    private void m1(Demo demo){
        Demo demo2 = demo;
        String s2 = "24";
        String s3 = new String(original:"24");
    }
}
```



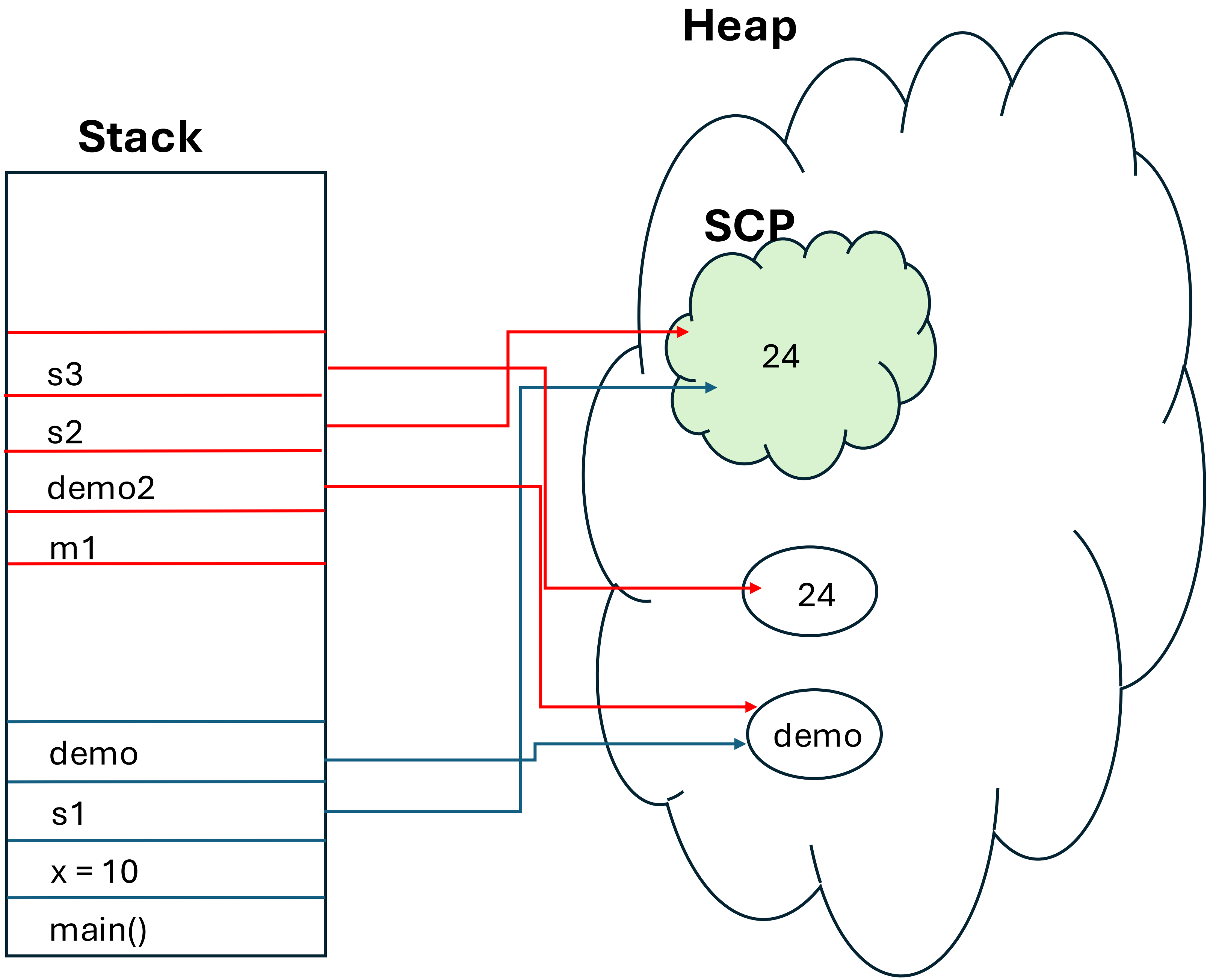
```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        int x = 10;
        String s1 = "24";
        Demo demo = new Demo();
        demo.m1(demo);
    }

    private void m1(Demo demo){
        Demo demo2 = demo;
        String s2 = "24";
        String s3 = new String(original:"24");
    }
}
```



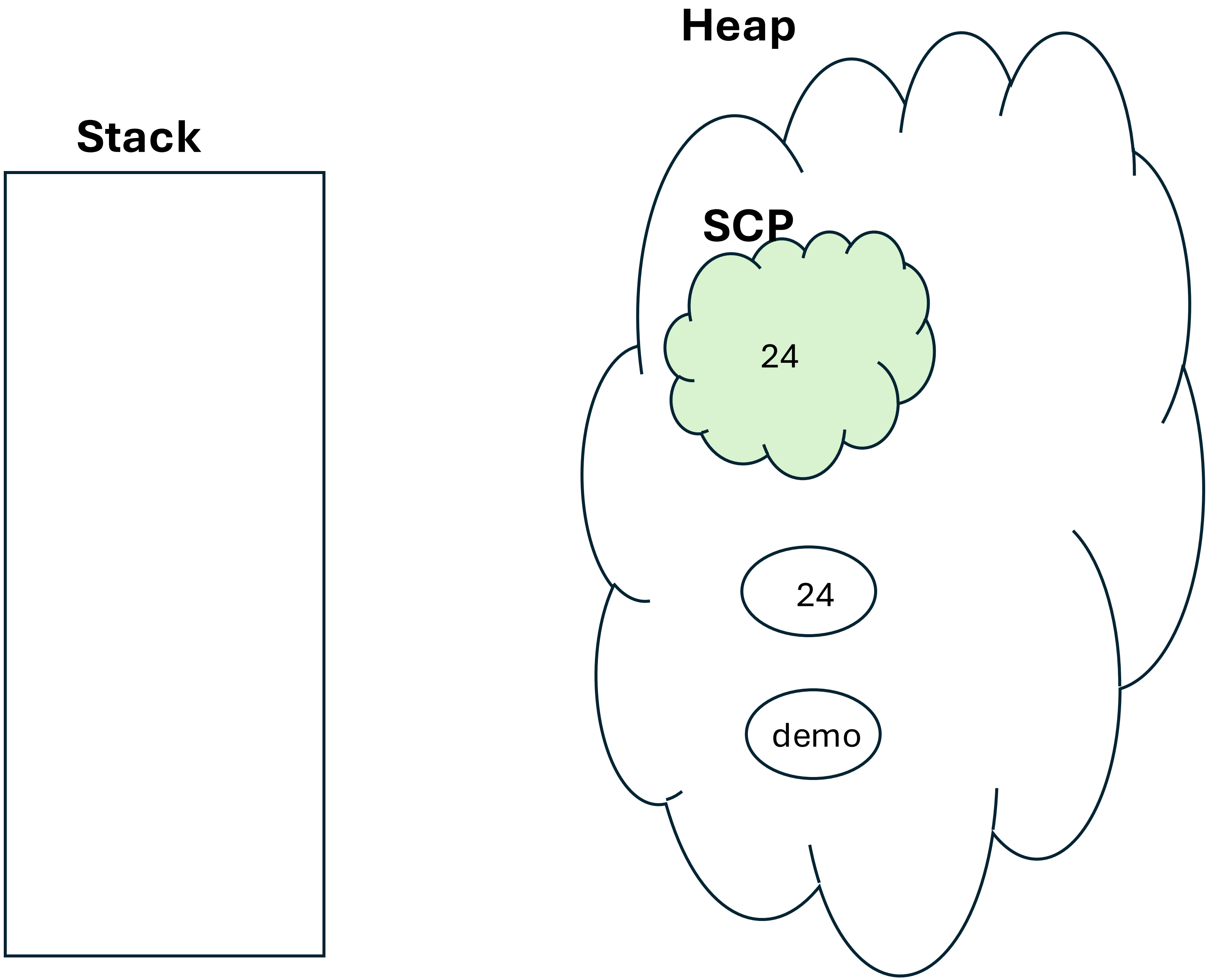
```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        int x = 10;
        String s1 = "24";
        Demo demo = new Demo();
        demo.m1(demo);
    }

    private void m1(Demo demo){
        Demo demo2 = demo;
        String s2 = "24";
        String s3 = new String(original:"24");
    }
}
```



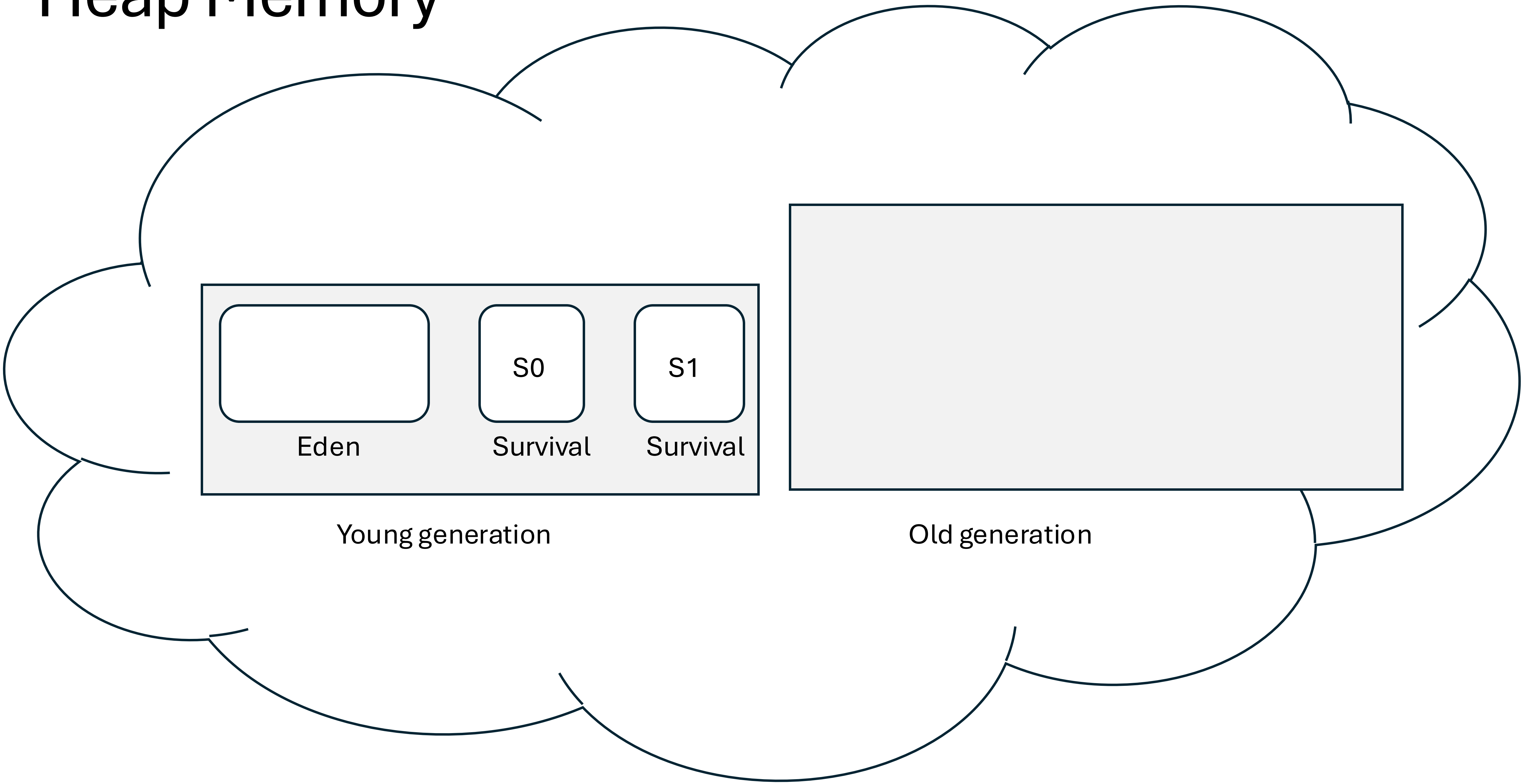
```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        int x = 10;
        String s1 = "24";
        Demo demo = new Demo();
        demo.m1(demo);
    }

    private void m1(Demo demo){
        Demo demo2 = demo;
        String s2 = "24";
        String s3 = new String(original:"24");
    }
}
```



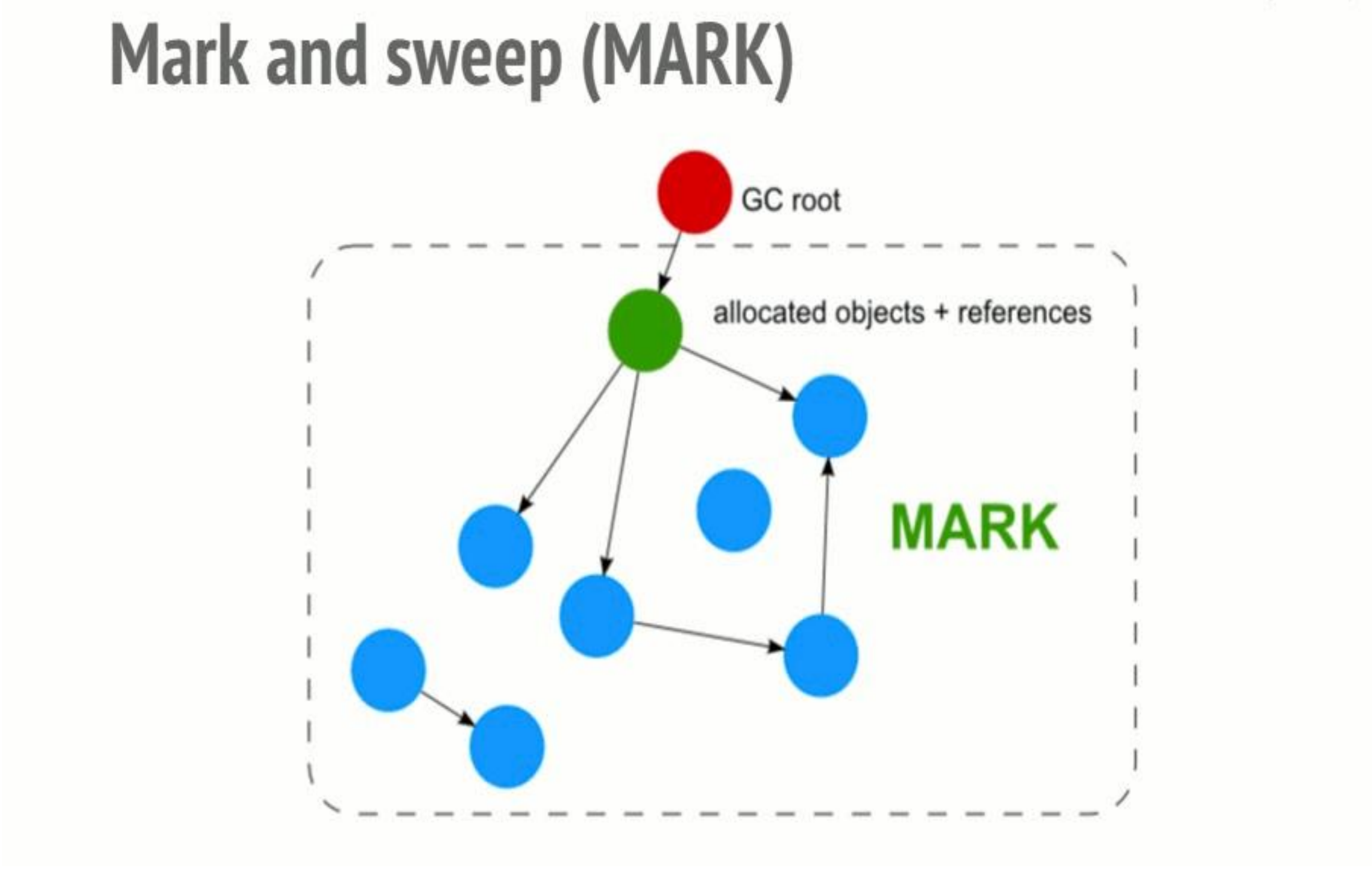
```java
public class Demo {

    Run | Debug
    public static void main(String[] args) {
        int x = 10;
        String s1 = "24";
        Demo demo = new Demo();
        demo.m1(demo);
    }

    private void m1(Demo demo){
        Demo demo2 = demo;
        String s2 = "24";
        String s3 = new String(original:"24");
    }
}
```



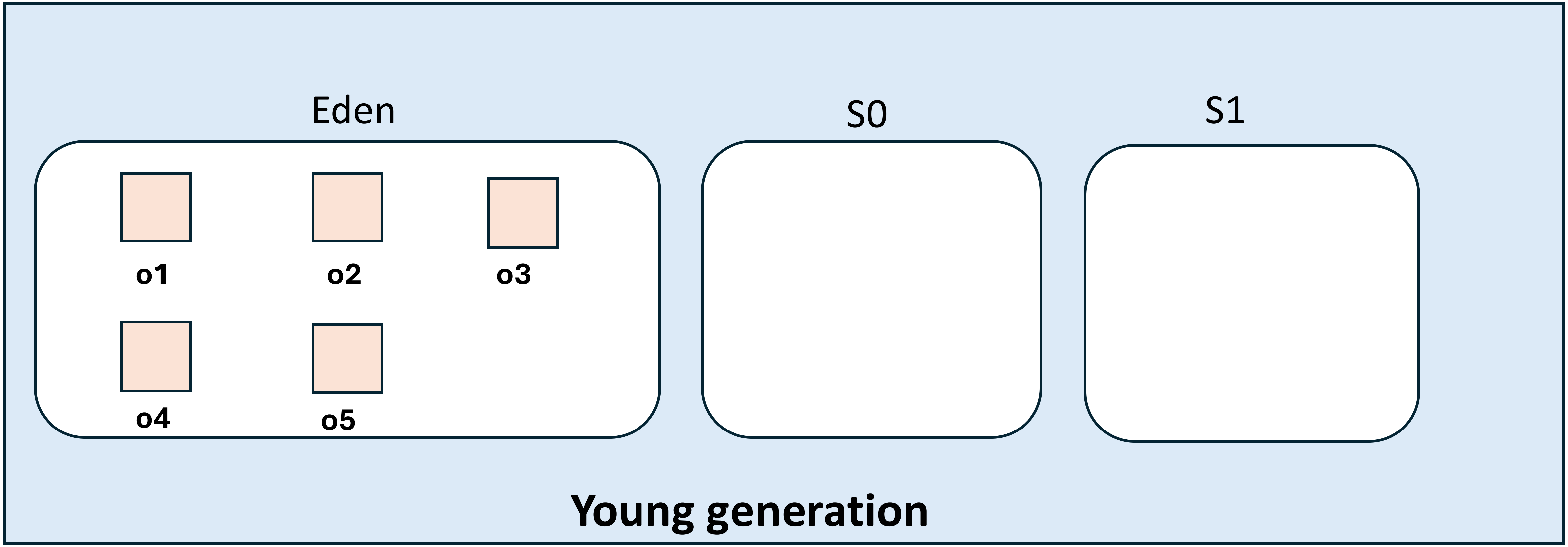
Heap Memory


S0
S1
Eden
Survival
Survival
Young generation
Old generation

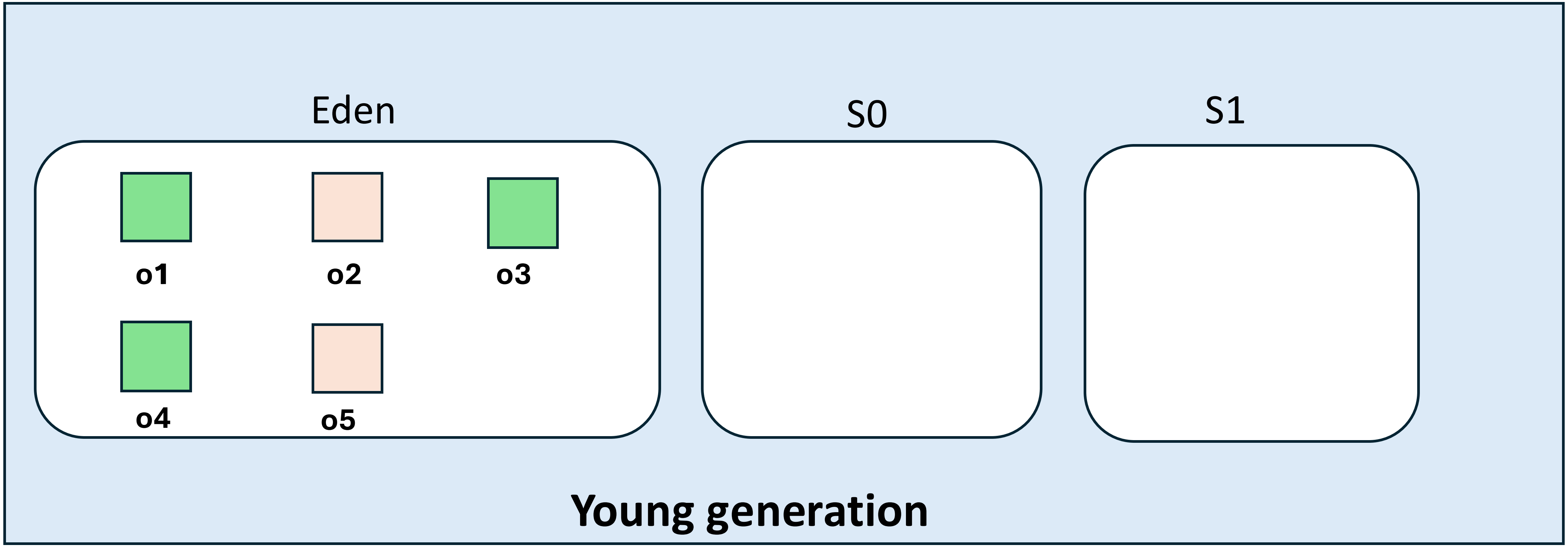
GC algorithm



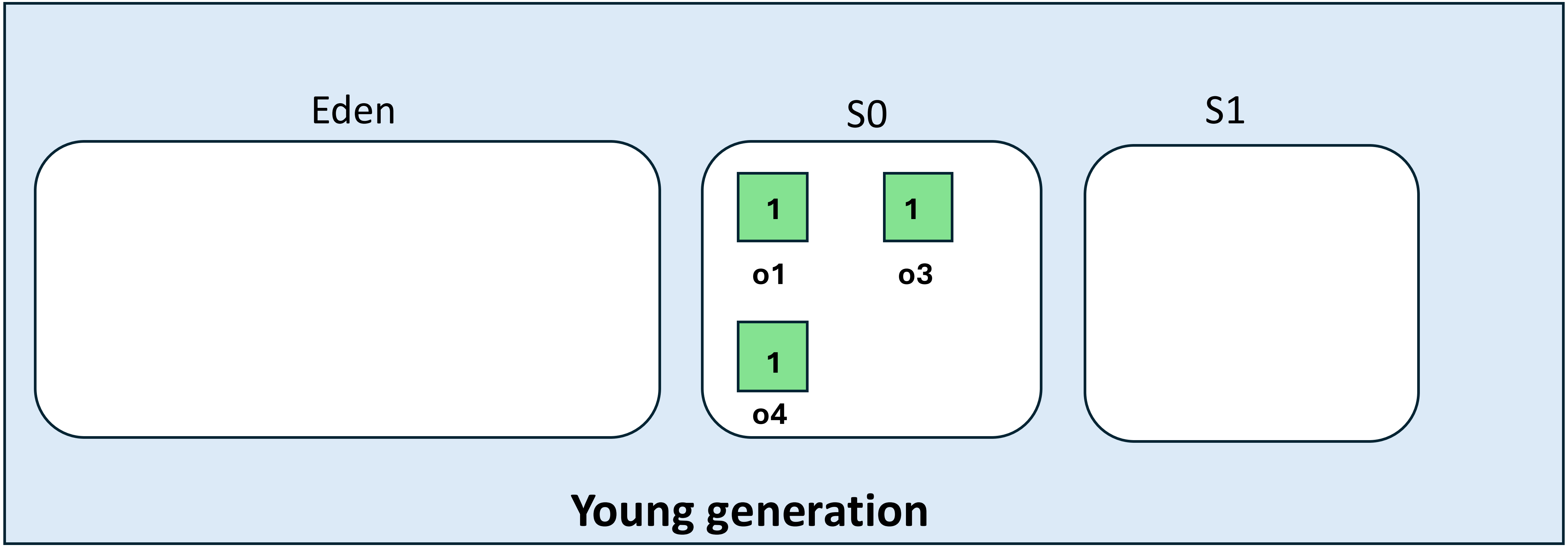
Suppose we create an object
o1, o2, o3, o4, o5



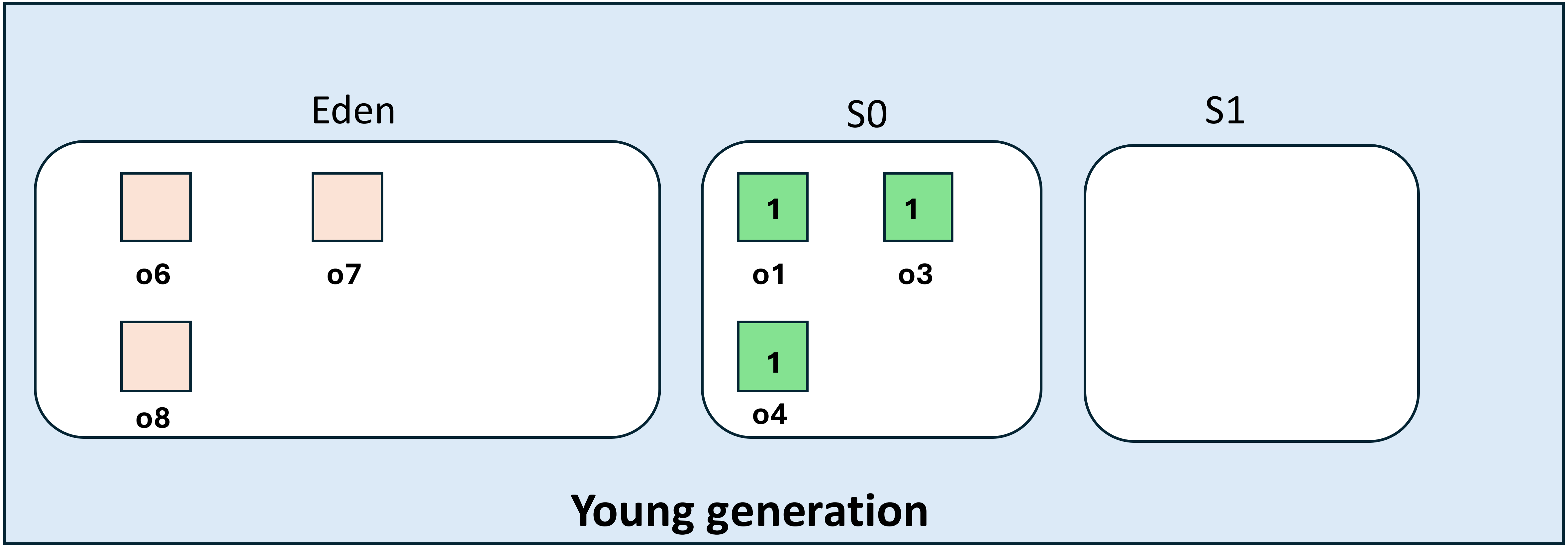
Consider o2 and o5 are unreferenced object



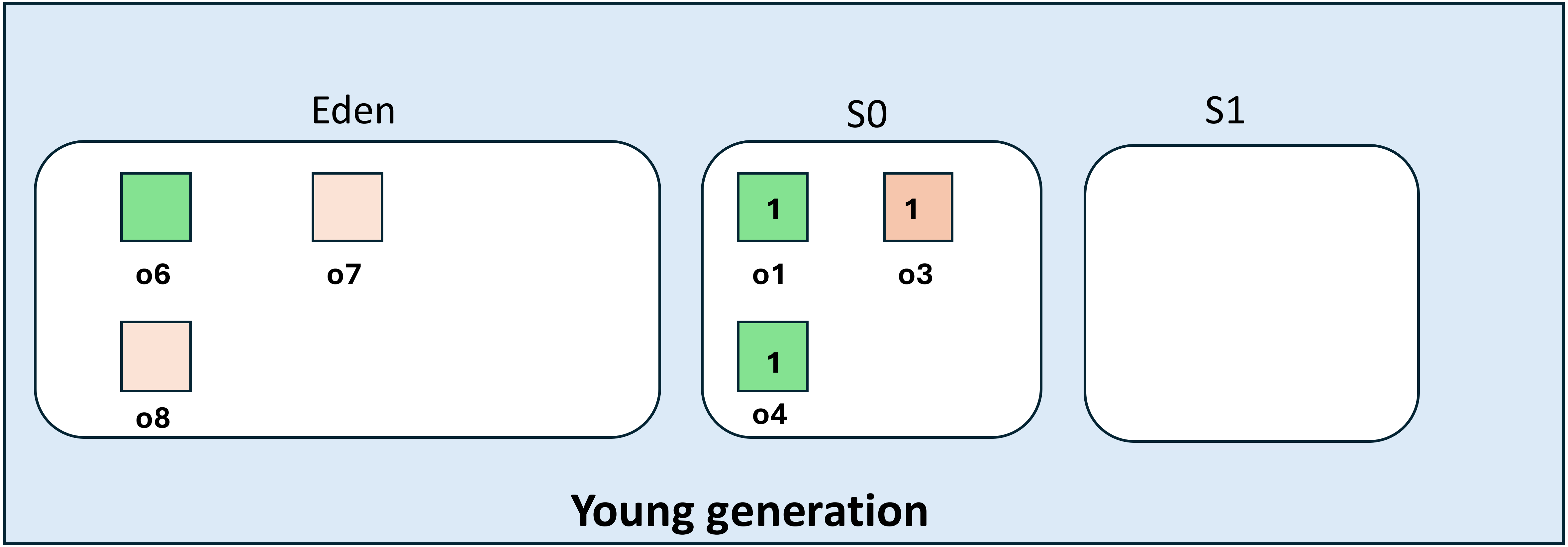
Unreferenced object are removed and referenced object moved to S0



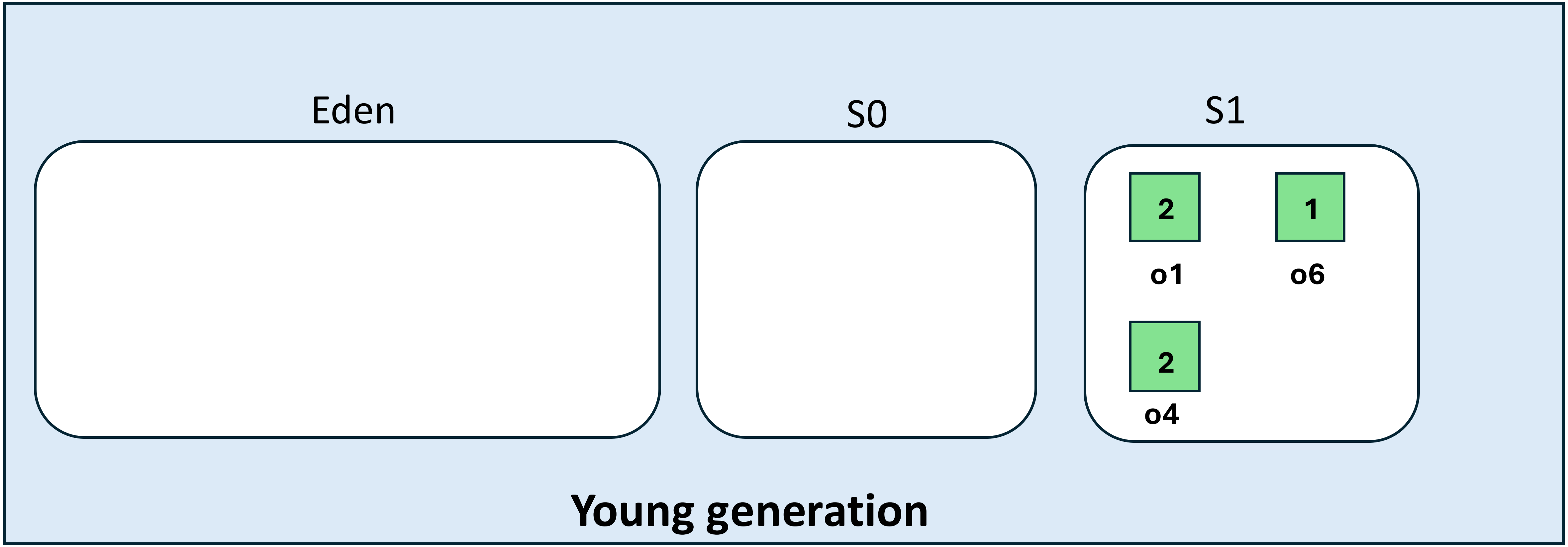
If we again create object o6, o7, o8



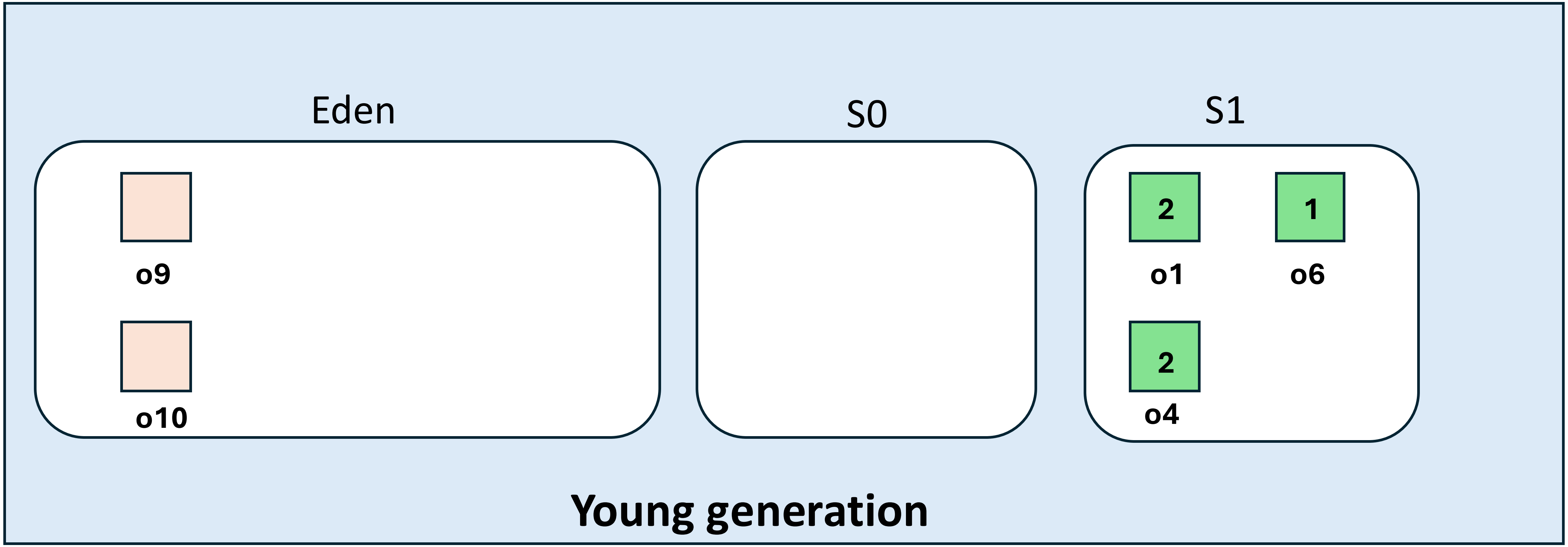
Suppose here o7,o8 and o3 are unreferenced object



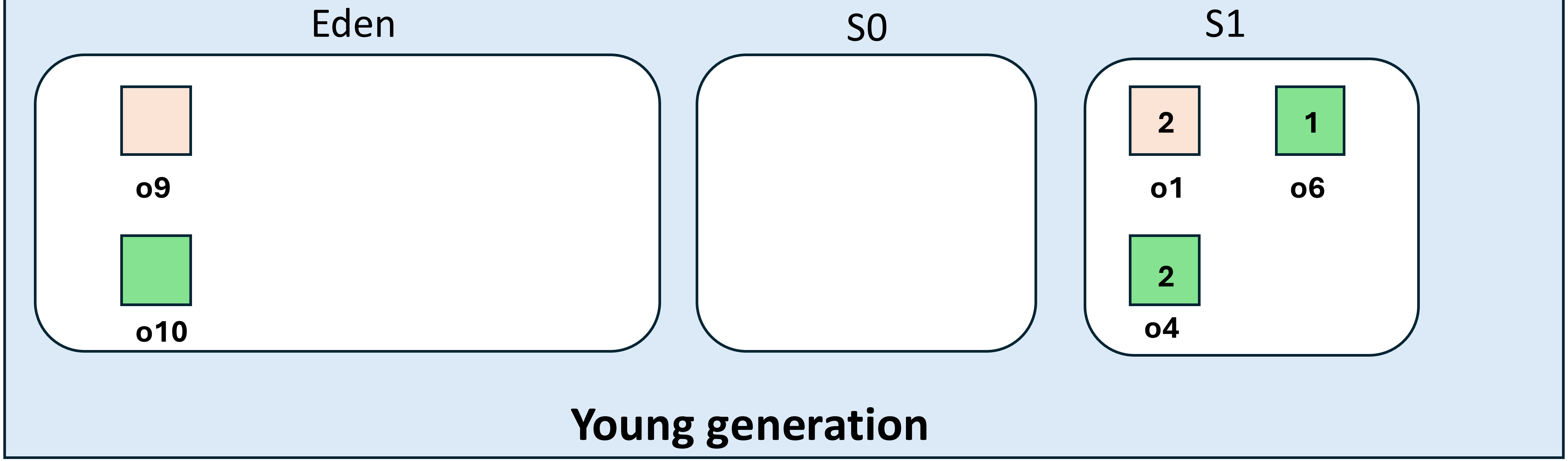
Gc will remove unreferenced object and move all live object in S1



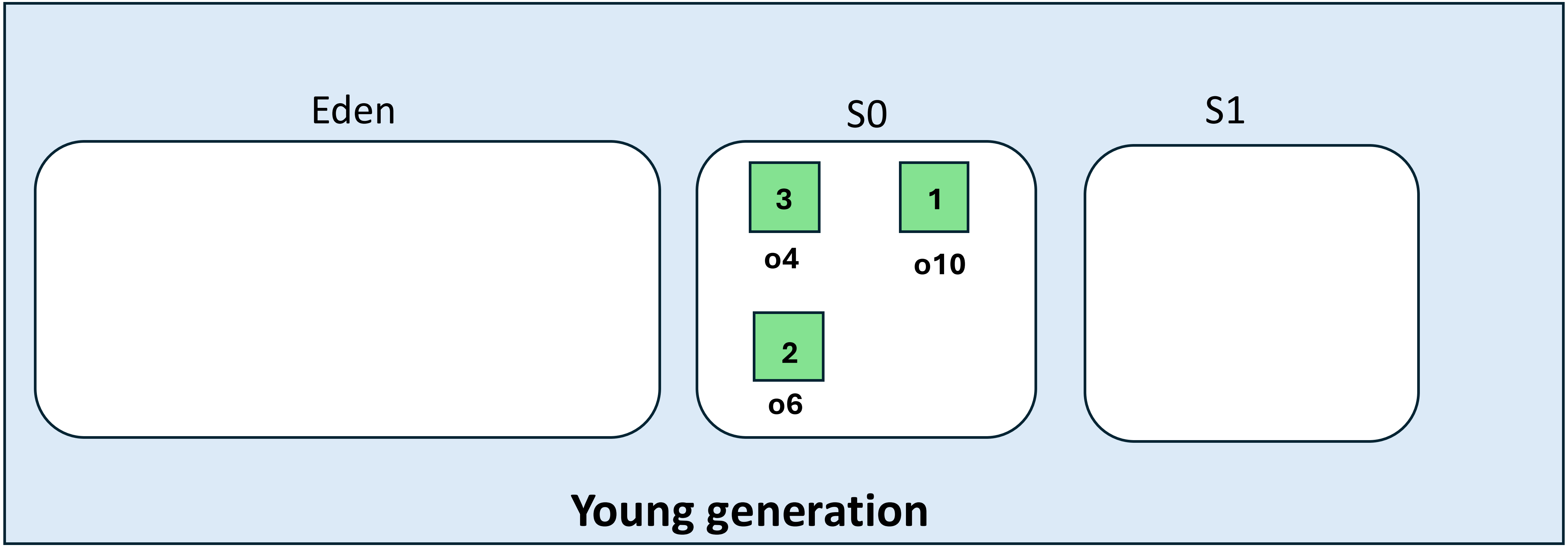
Suppose we again create object of o9 and o10



Suppose o9 and o1 are unreferenced object

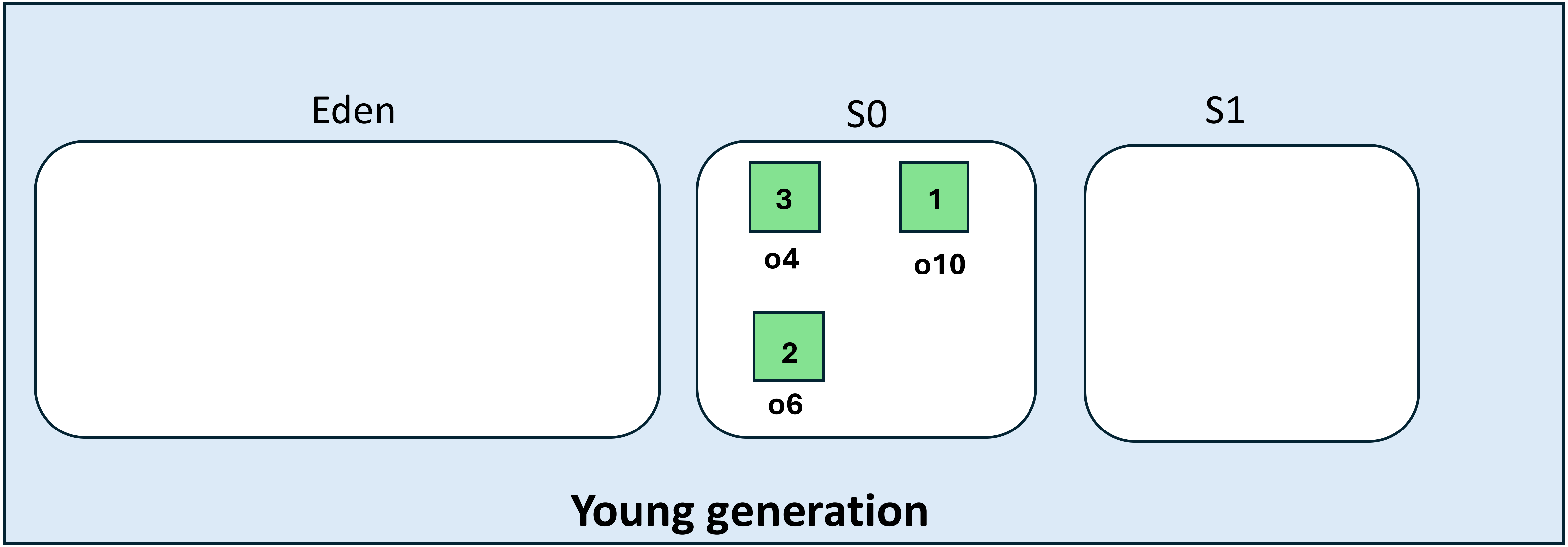


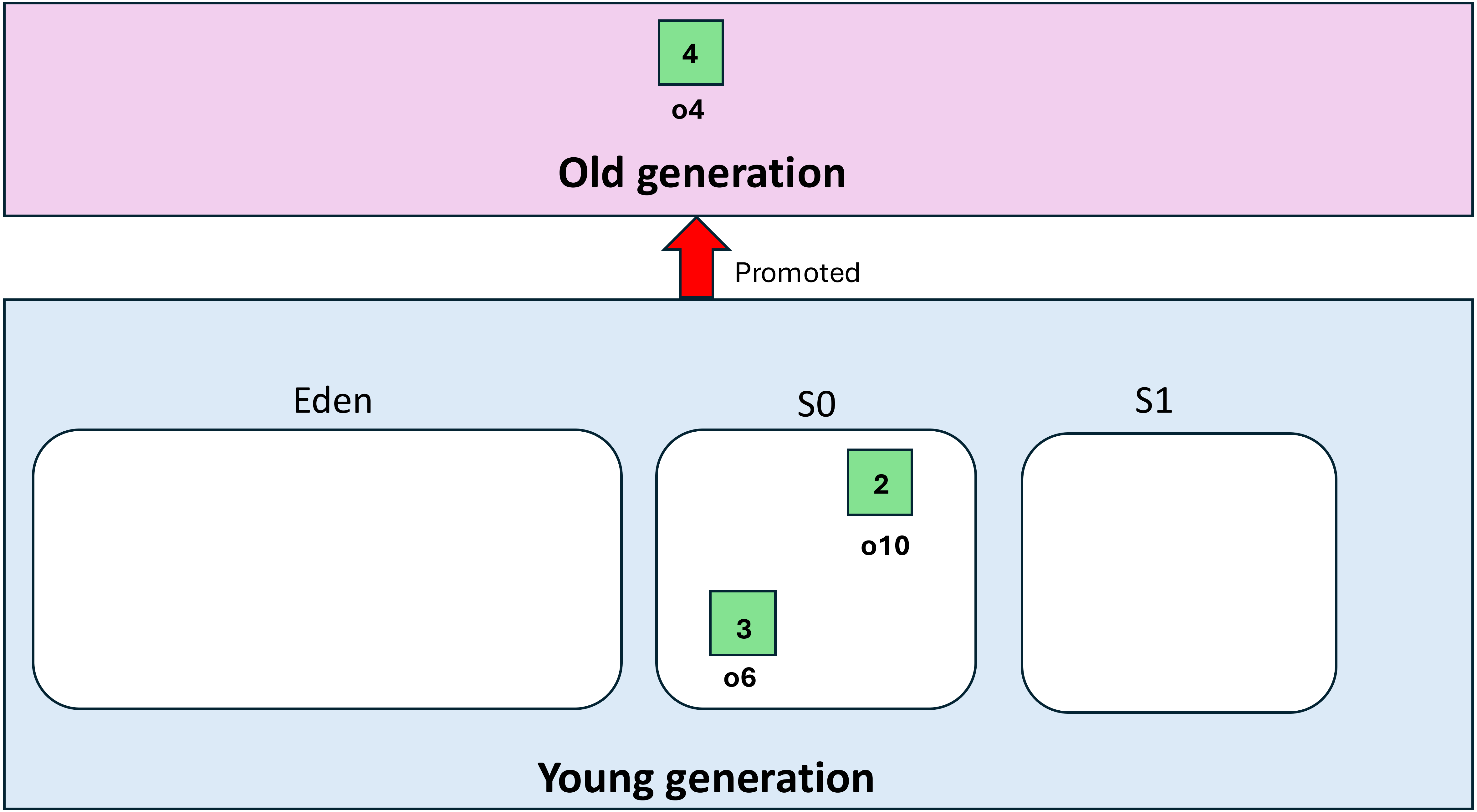
Gc will remove unreferenced object and move referenced object to S0

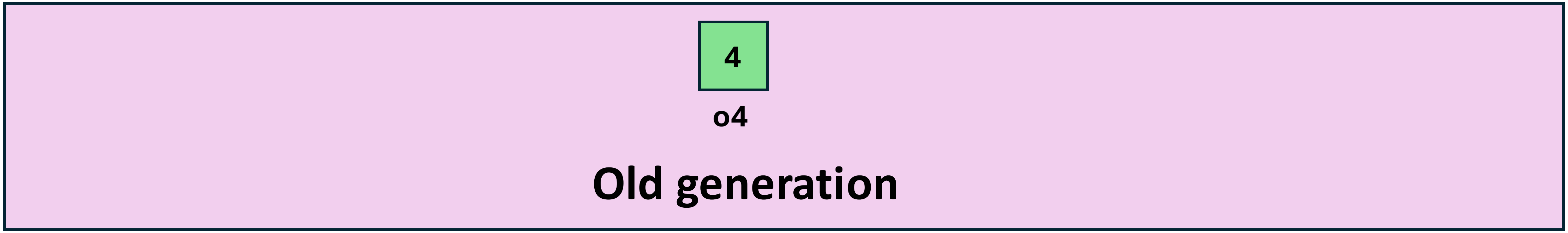


This process is called as **Minor GC.**

Now suppose the threshold of object here is 3.




4
o4
Old generation
Promoted
Eden
S0
2
o10
3
o6
S1
Young generation



When GC runs in old generation it is called as **Major GC**.

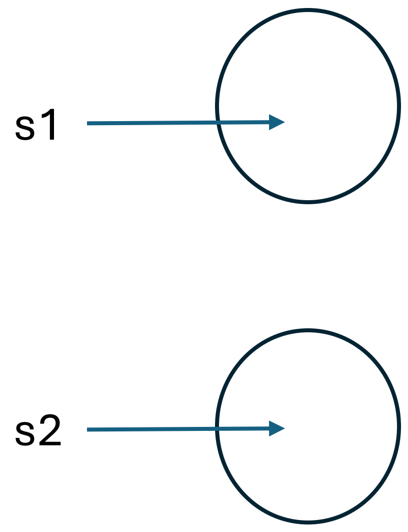
Objects present in old generation are used frequently and alive from too long.

GC in old generation is little bit time taking than young generation.

How can we make objects eligible for GC?

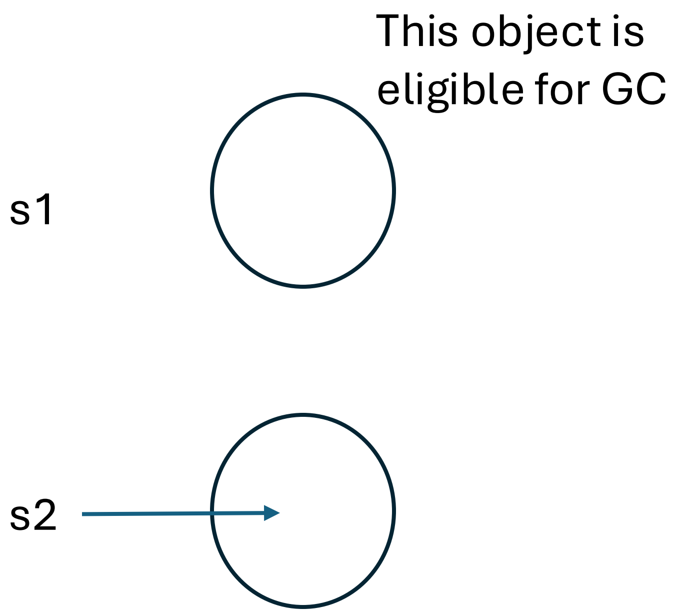
1. Nullifying the reference variable

```
Student s1 = new Student();

Student s2 = new Student();
```



1. Nullifying the reference variable

```
Student s1 = new Student();

Student s2 = new Student();

s1 = null;
```



2. Reassigning the reference variable

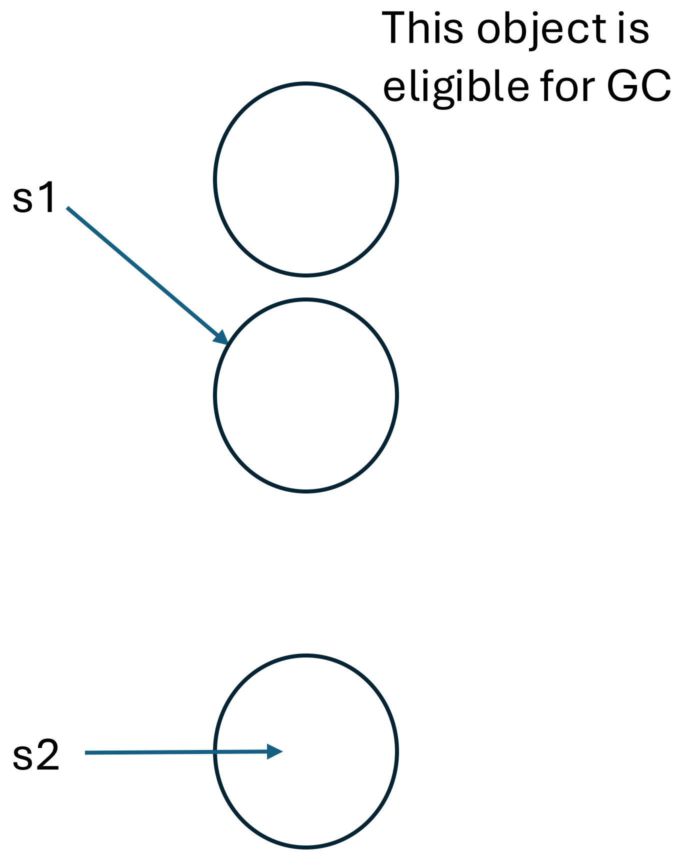
```
Student s1 = new Student();

Student s2 = new Student();
```

2. Reassigning the reference variable

Student s1 = new Student();

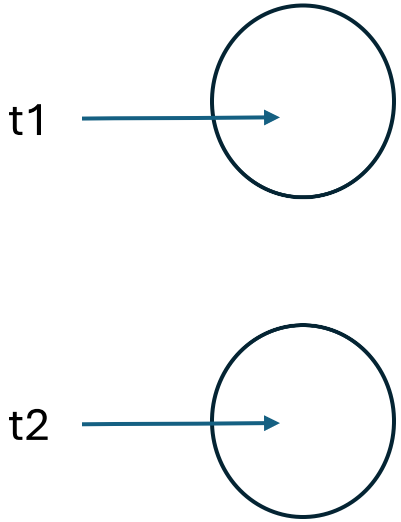
Student s2 = new Student();

s1 = new Student();



3. Object created inside method

```java
class Test{

    public static void main(String[] args) {
        m1();
    }

    public static void m1(){
        Test t1 = new Test();
        Test t2 = new Test();
    }
}
```



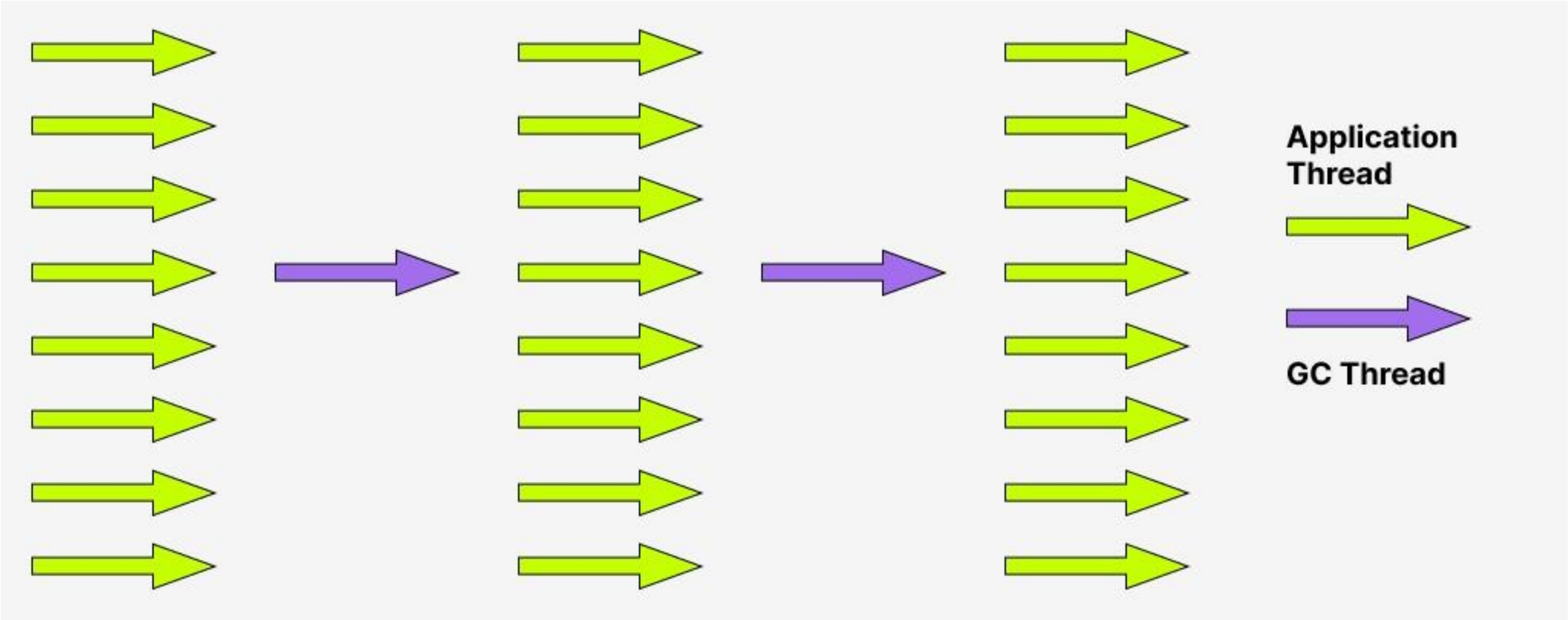
Request JVM to run GC

- System.gc();
- Runtime r = Runtime.getRuntime();

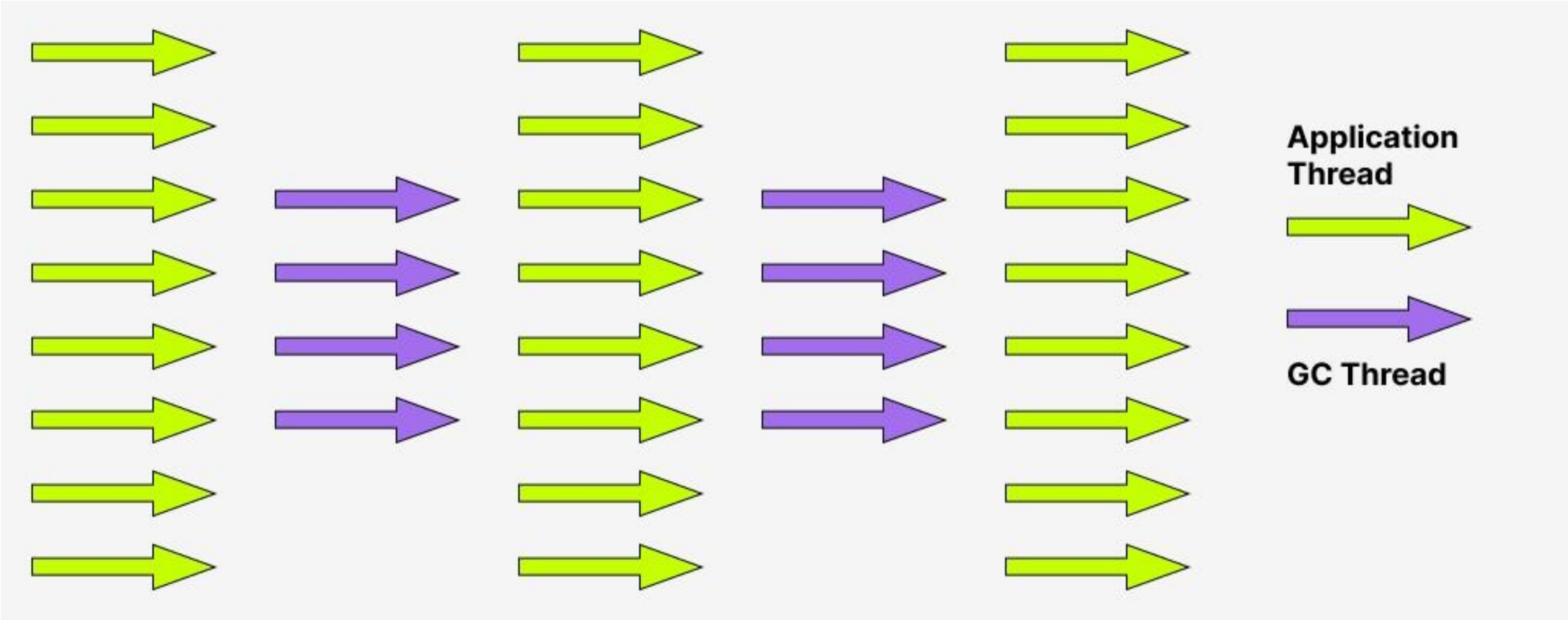
- Even if we write this in our code, JVM has control when to run garbage collector so garbage collector may or may not be activated

Types of Garbage Collector

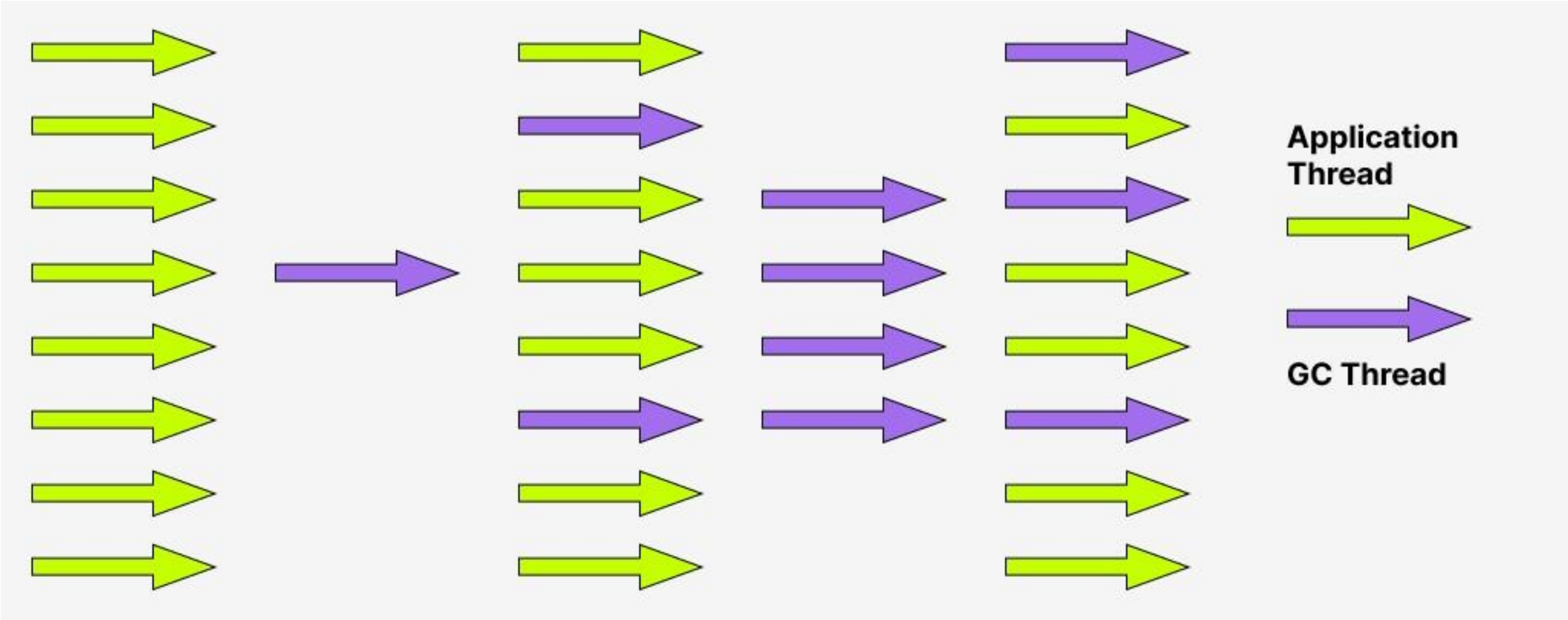
Serial Garbage Collector



Parallel Garbage Collector



Concurrent Mark Sweep GC



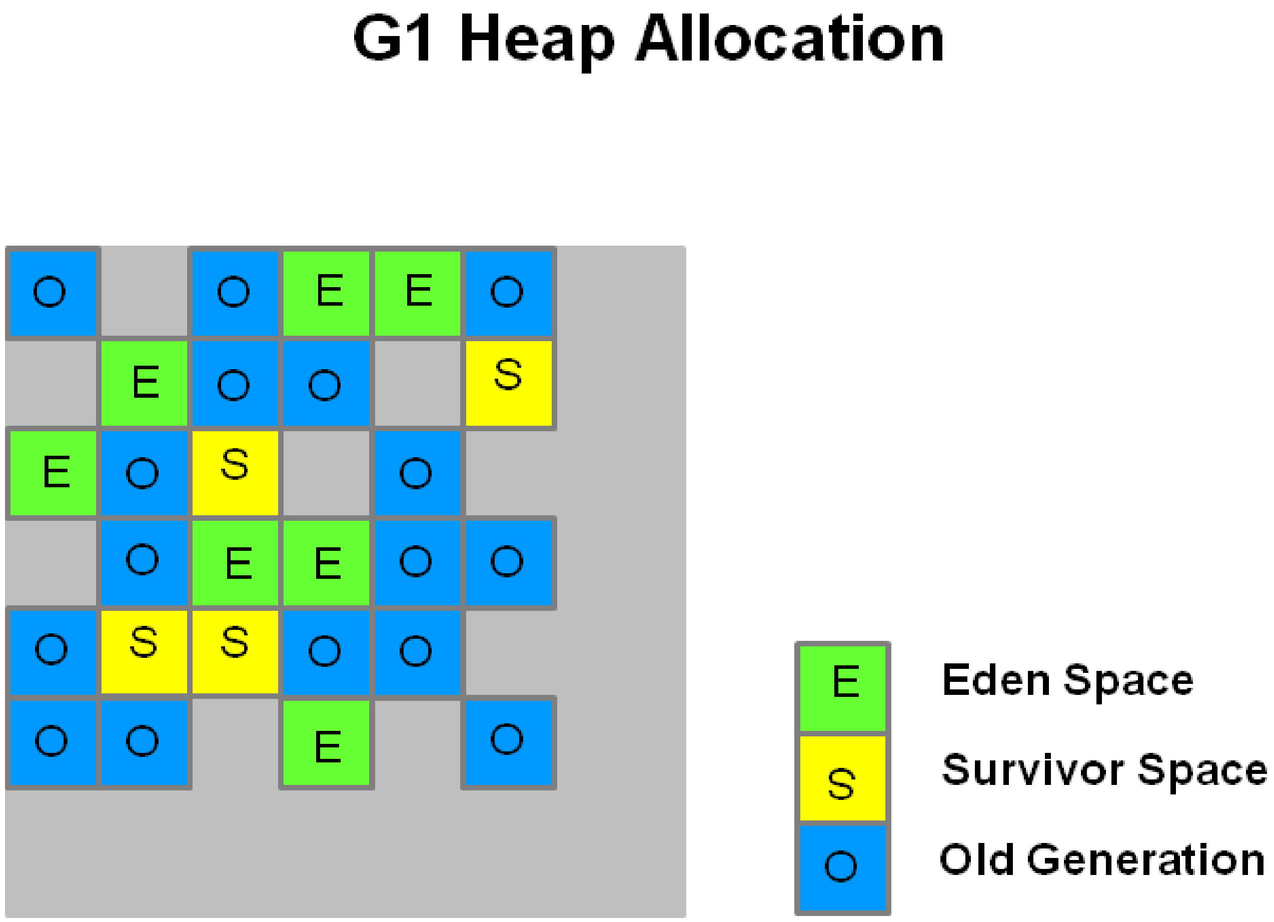
G1GC

G1 Heap Structure

Region size is chosen by JVM at startup.

The JVM generally targets around 2000 regions varying in size from 1Mb to 32 Mb.

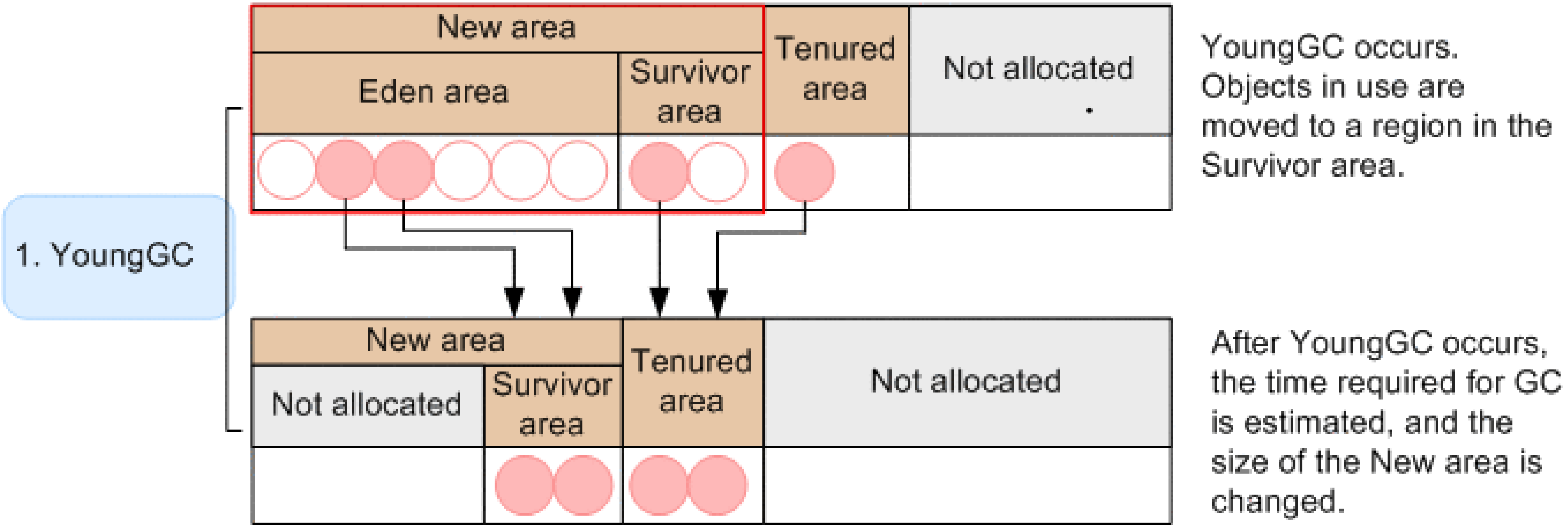
G1GC



Live objects are copied or moved from one region to another.

Regions are designed to be collected in parallel with or without stopping all other application threads.

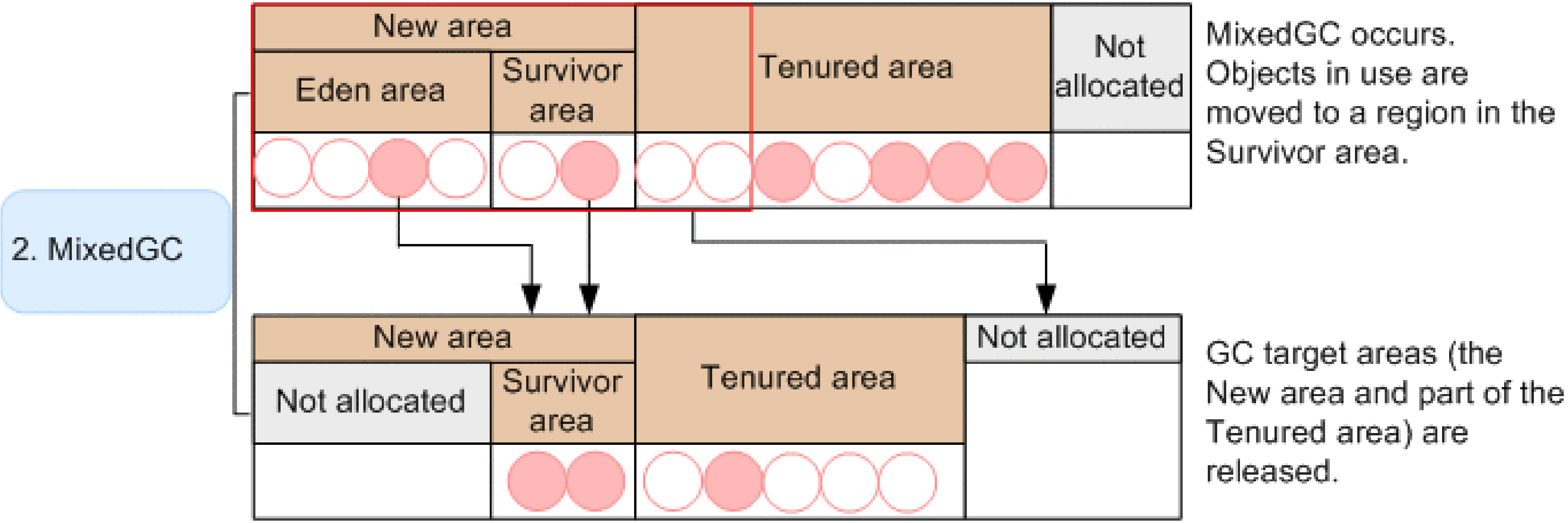
G1GC



Young GC occurs when there is no more space in a region allocated to New area. This is the stop the world pause.

If the GC took longer than expected, G1 may reduce the size of the New area to prevent future delays. Conversely, if the GC was fast, the New area size might increase.

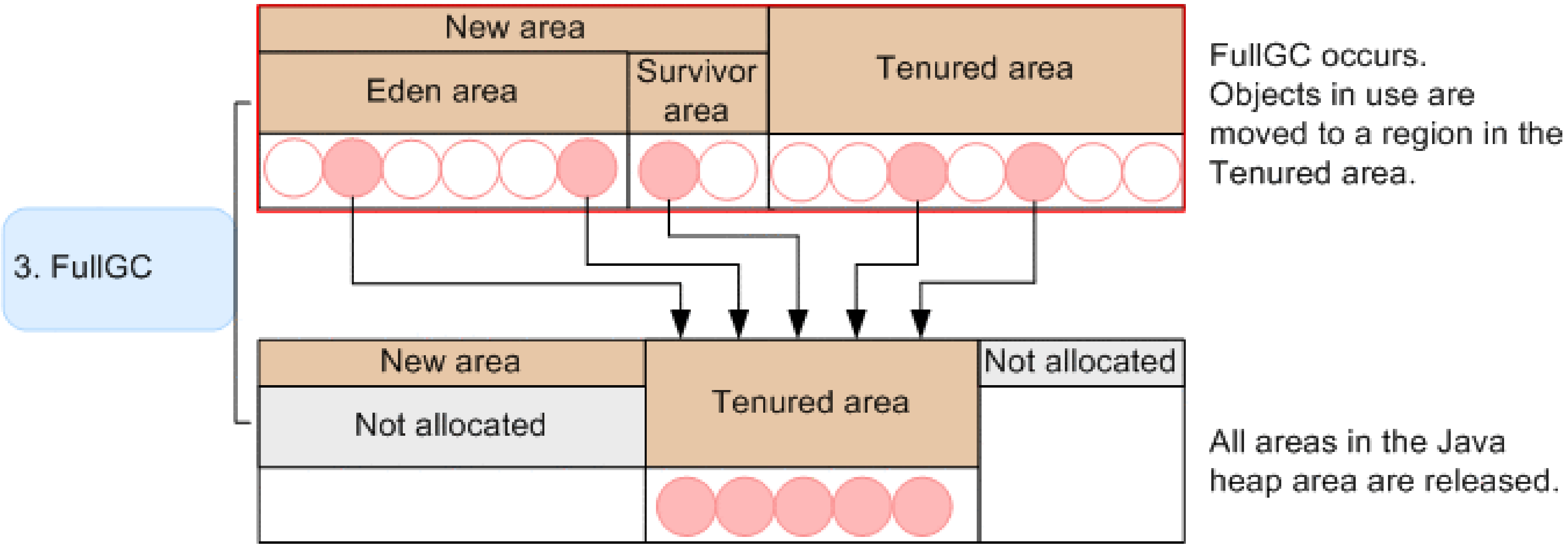
G1GC



Mixed GC occurs when the usage rate of the Tenured area increases.

If the analysis of object information in the Tenured area is insufficient, or if the potential impact of the Mixed GC is minimal, the G1 GC may skip this phase.

G1GC



FullGC occurs for the entire heap area, when there is no space in a region in the heap area. This is stop the world event.

Full GC also includes heap compaction. G1 GC tries to avoid Full GCs as much as possible, but it occurs when memory pressure is high

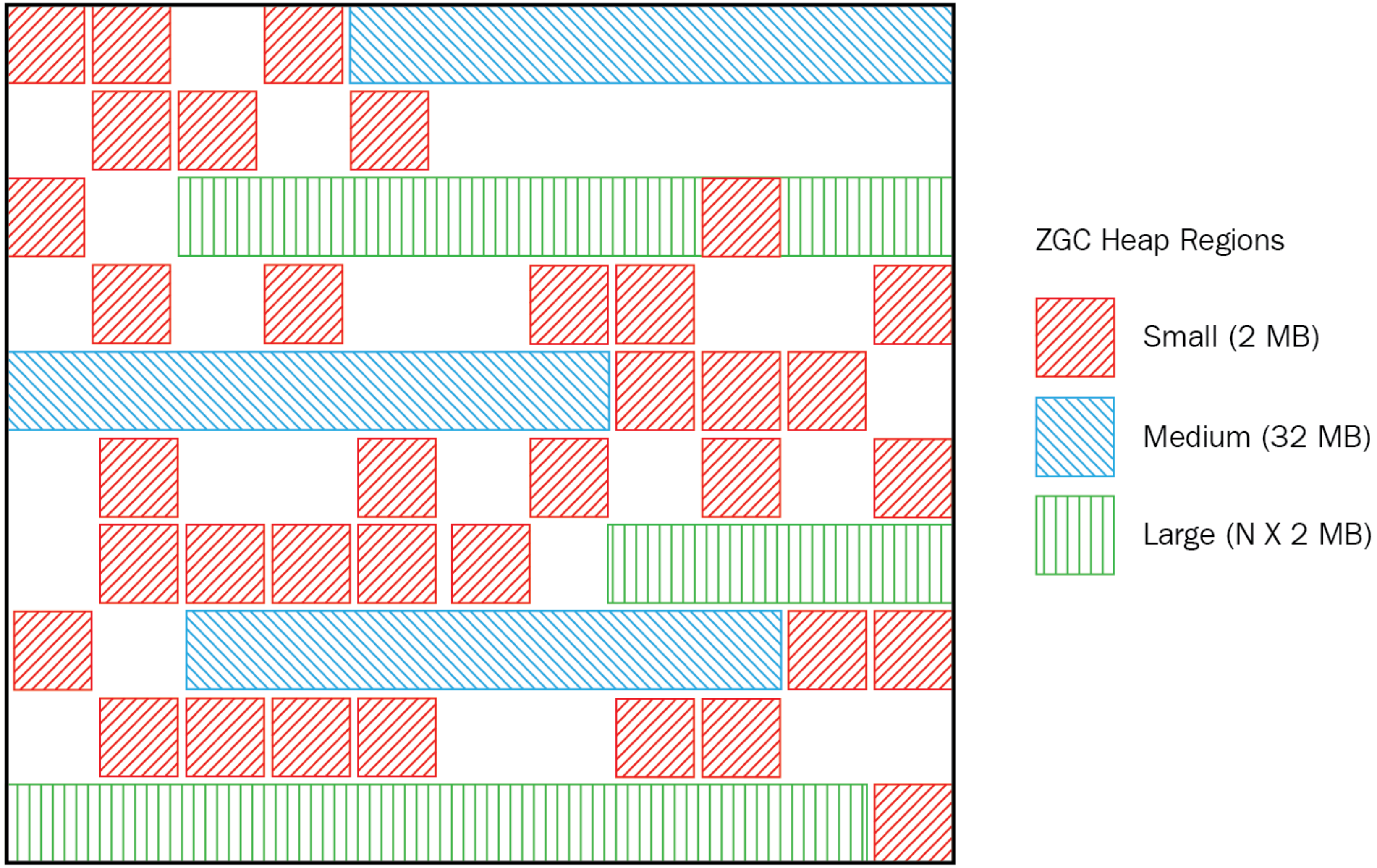
ZGC

ZGC was introduced in JDK 11 and was production ready at JDK 15.

Key features:

- Low latency (Pause time below 10 milliseconds)
- Supports large heap
- Concurrent operations
- No generational concept

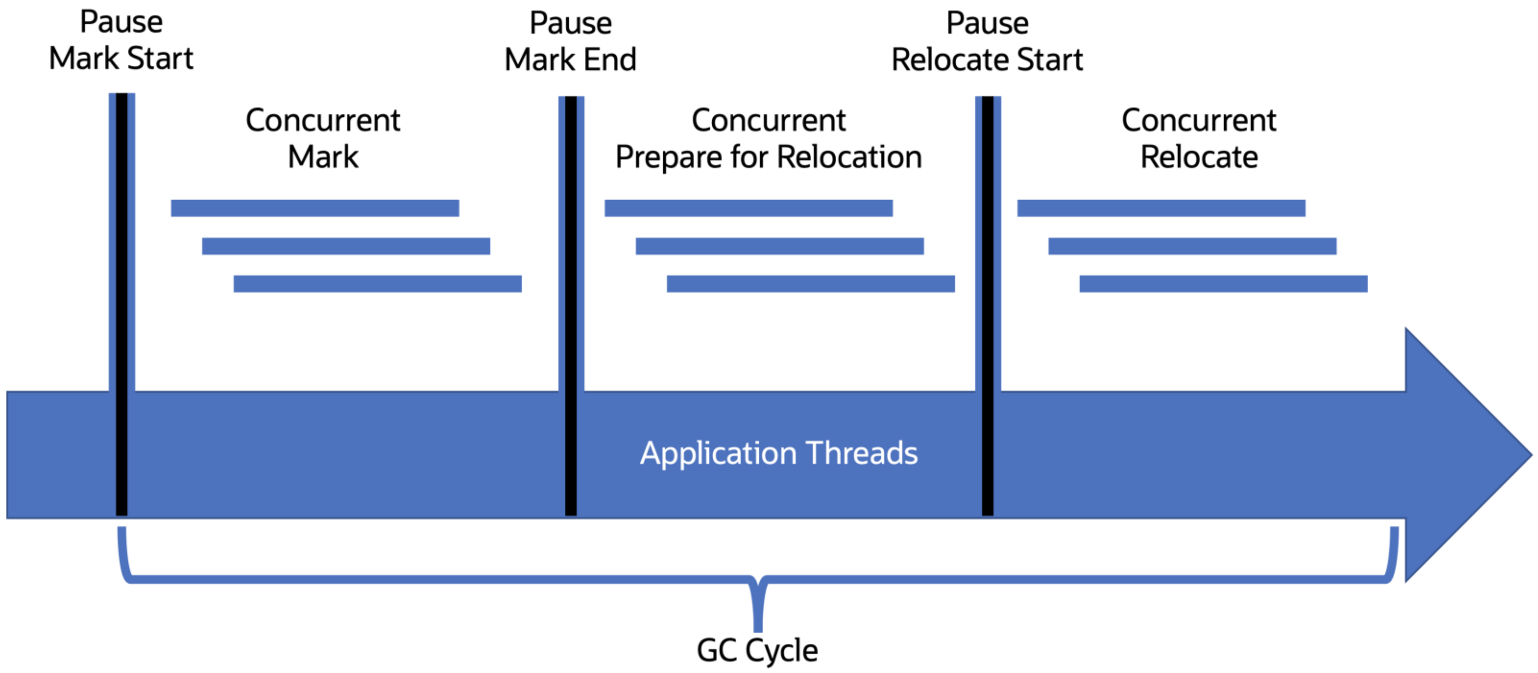
ZGC



The heap in ZGC is divided into a set of regions called **pages**. These pages can be of different sizes (2 MB, 32 MB, or 1 GB).

Unlike G1 GC, ZGC does not divide the heap into generations; all objects are treated uniformly.

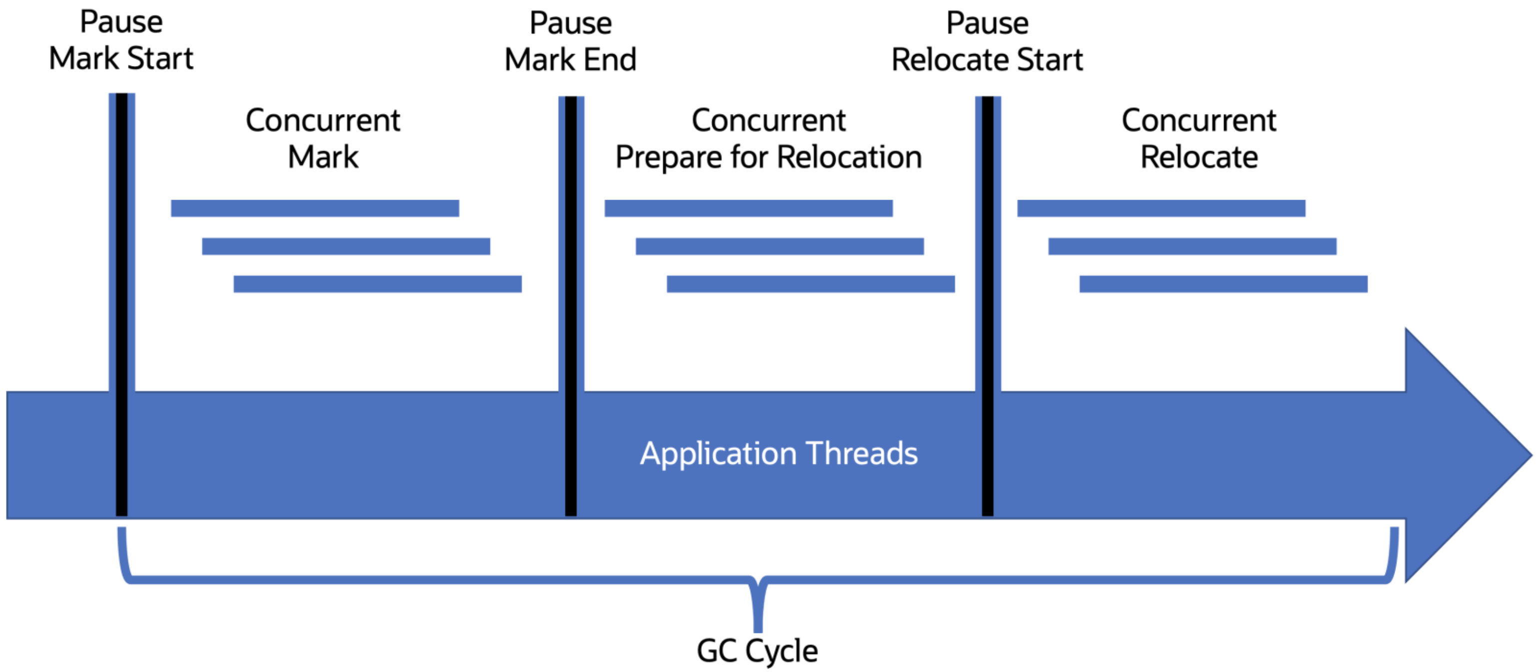
ZGC



Pause Mark Start:

- This is a brief "stop-the-world" (STW) pause where ZGC marks the root objects in the heap. The application threads are paused for a very short time to start the marking phase.

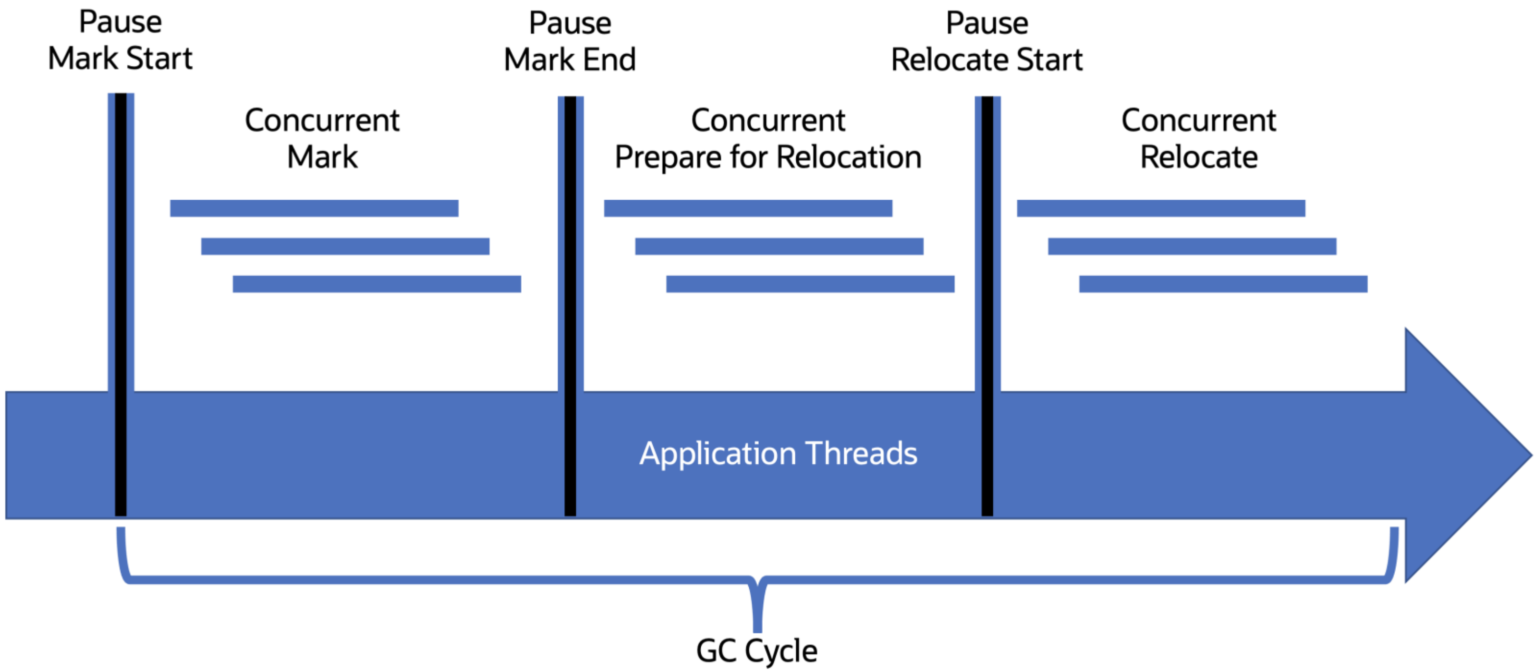
ZGC



Concurrent Mark:

- After the initial pause, the GC marking phase continues in parallel with the application threads. During this phase, ZGC identifies all live (reachable) objects by tracing from the root objects. Since this phase is concurrent, the application continues running while the marking is happening.

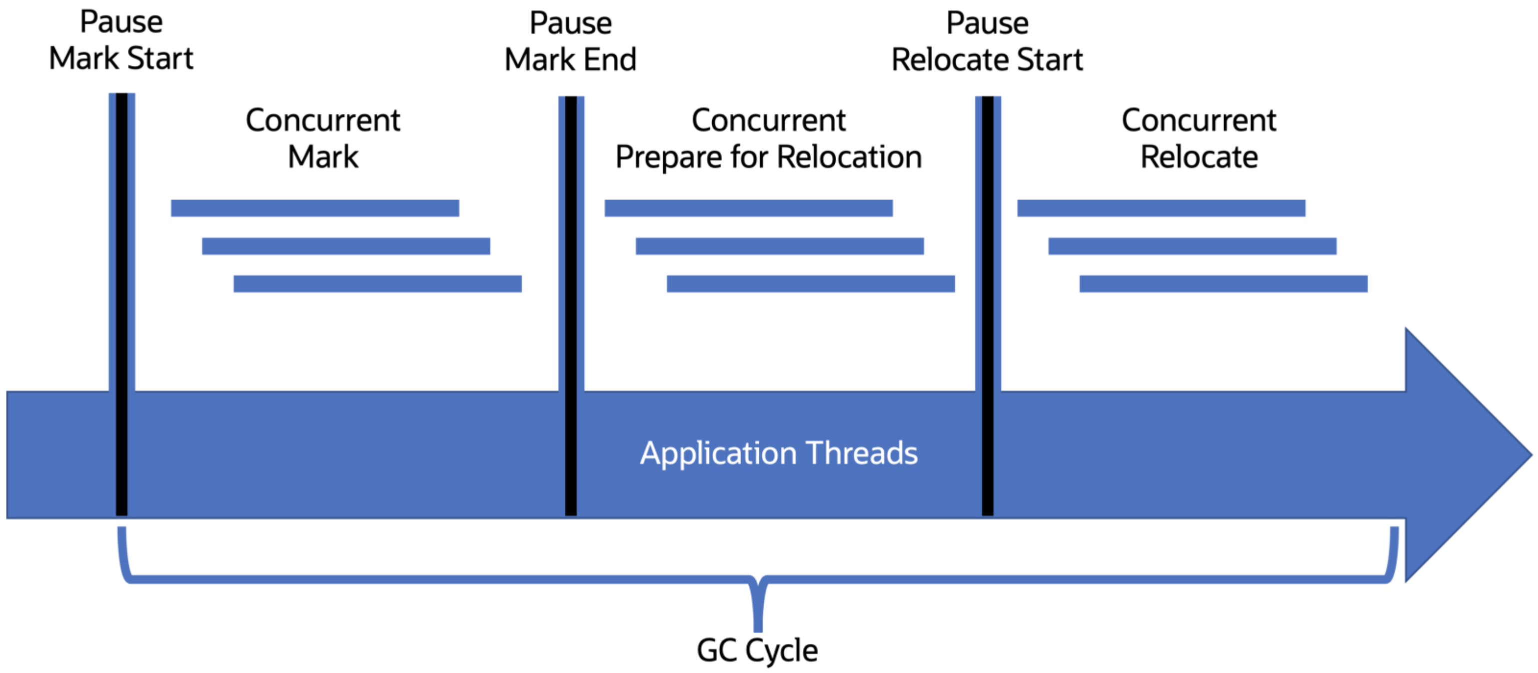
ZGC



Pause Mark End:

- After concurrent marking, there’s another short pause where the GC ensures all live objects have been marked. This marks the end of the marking phase.

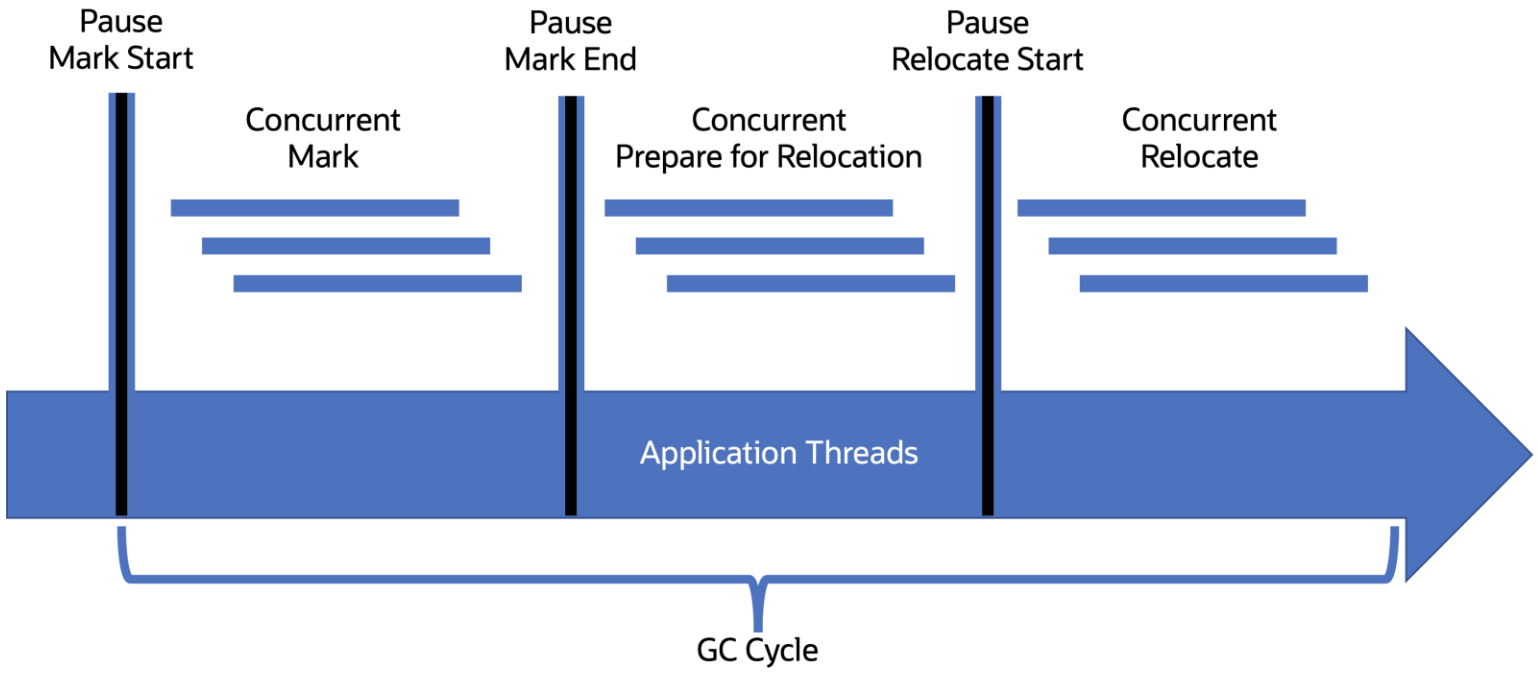
ZGC



Concurrent Prepare for Relocation:

- ZGC then prepares to relocate (move) objects to combat memory fragmentation. This phase is also done concurrently with the application threads. The GC identifies which objects will be moved

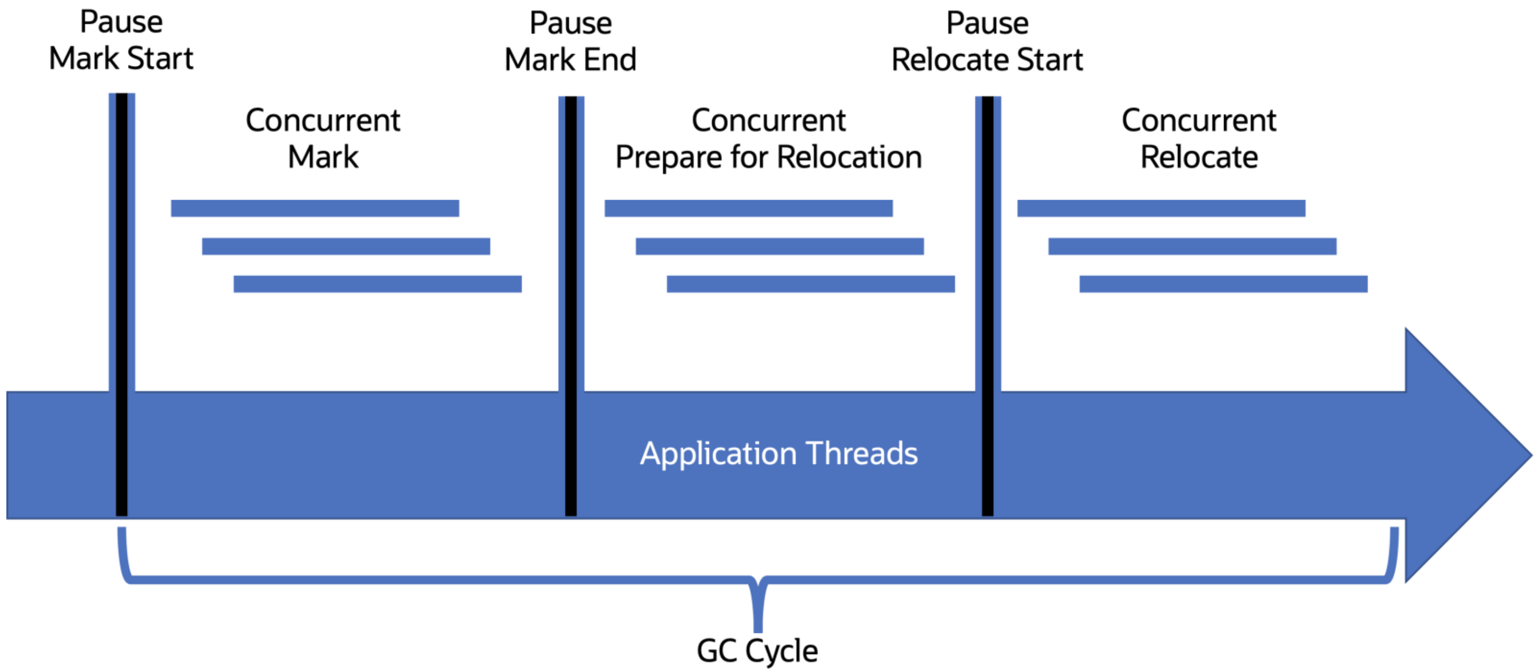
ZGC



Pause Relocate Start:

- Another brief pause occurs where ZGC transitions to the relocation phase. The application threads are paused for a moment to begin the relocation process.

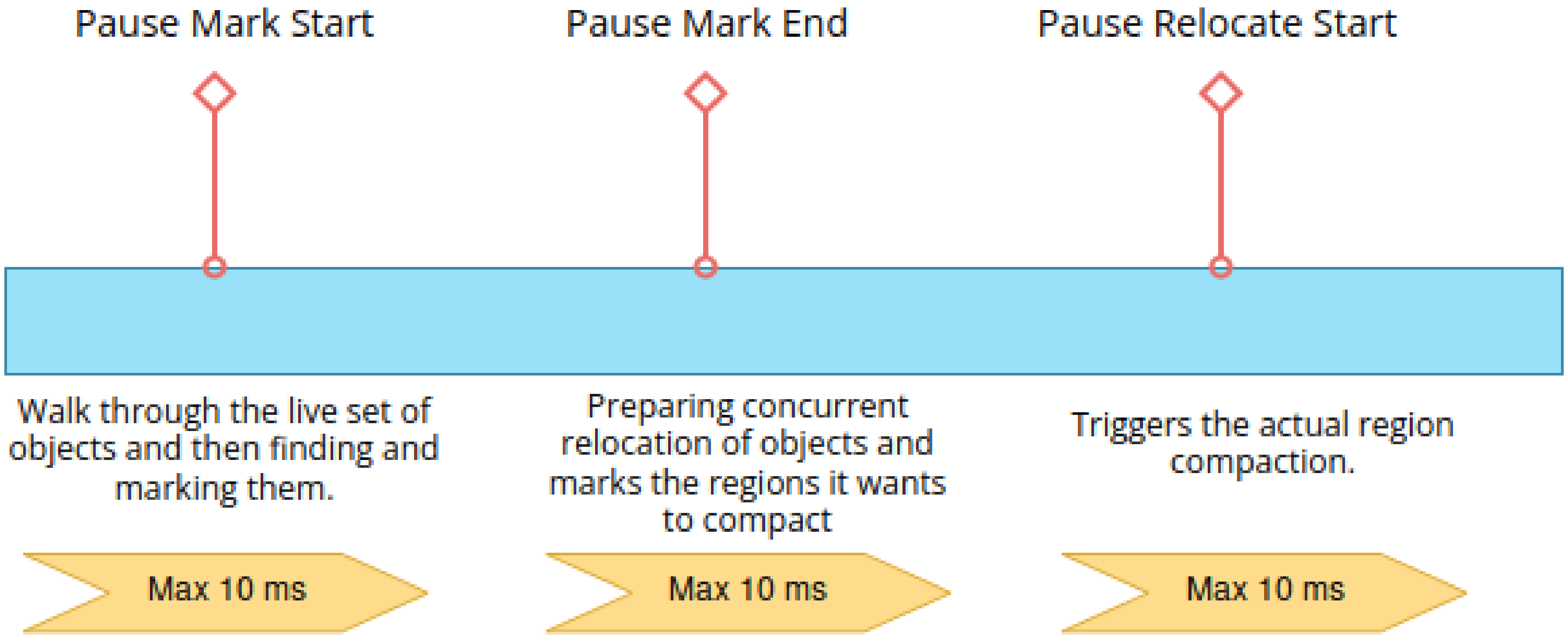
ZGC



Concurrent Relocate:

- Finally, ZGC moves the identified objects to new memory regions. Like other phases, this relocation happens concurrently, allowing the application to keep running with minimal interruption.

ZGC



The entire GC cycle involves marking live objects, preparing for relocation, and relocating them, but these tasks are done in a way that minimizes impact on the application's performance.

Implementation

For ZGC

```
tasks.named('bootRun') {
    jvmArgs = [
        '-XX:+UnlockExperimentalVMOptions',
        '-XX:+UseZGC',
        '-Xlog:gc*:file=gc.log:time,uptime:filecount=5,filesize=10M'
    ]
}
```

For G1GC

```
tasks.named('bootRun') {
    jvmArgs = [
        '-Xlog:gc*:file=gc.log:time,uptime:filecount=5,filesize=10M'
    ]
}
```

Analyzing gc logs

THANK YOU