Java™

ORACLE®

Understanding Java Virtual Machine

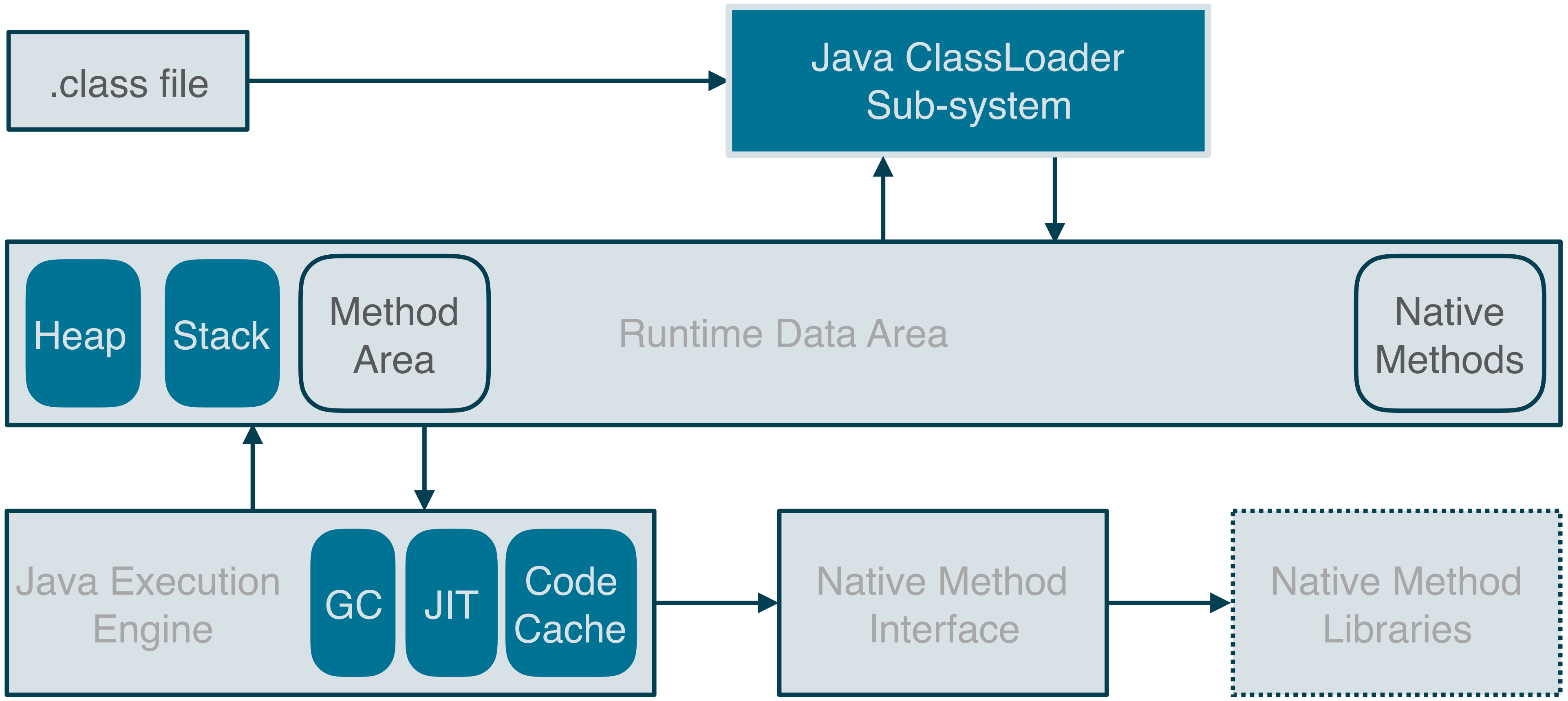
Vaibhav Choudhary (@vaibhav_c)
Java Platforms Team, Oracle
https://blogs.oracle.com/vaibhav

Java Your Next (Cloud)

Safe Harbor Statement

Agenda of the day …

Java Virtual Machine - Architectural View



Beginning of Java ClassLoader Sub-system

Class And Data

- Class represents code. Data represents state.
- State often change, code mostly not.
- Class + State = Instance
- Once a class is loaded in JVM, the same class (?) will not be loaded.
- Same class concept
  - (C1, P1, K1) != (C1, P1, K2)

Class loader types

- Bootstrap class loader (primordial class loader)
  - Special class loader
  - VM runs without verification
  - Access to the repository of “trusted classes”
  - Native implementation
  - **<JAVA_HOME>/jre/lib (rt.jar, i18n.jar)**

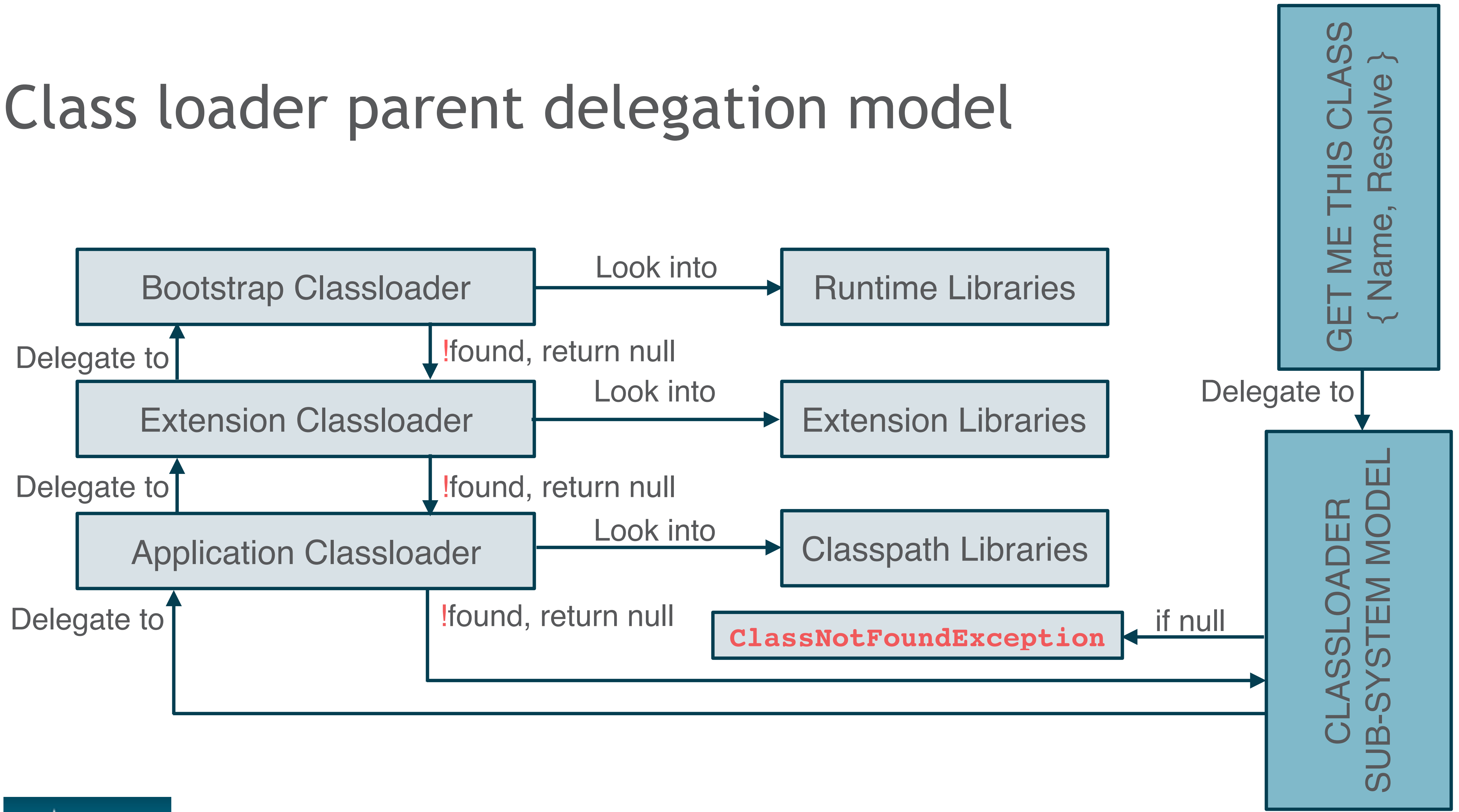
Class loader types

- Extension class loader
  - Loads the code in the extension directories
  - <JAVA_HOME>/jre/lib/ext
  - -Djava.ext.dir
  - Java implementation - sun.misc.Launcher$ExtClassLoader

Class loader types

- Application class loader
  - Loads the code found in java.class.path which maps to CLASSPATH.
  - Java implementation - sun.misc.Launcher$AppClassLoader

Class loader parent delegation model



In short, this is how it works

```
protected synchronized Class<?> loadClass
    (String name, boolean resolved)
    throws ClassNotFoundException {

    // First check if the class is already loaded
    Class c = findLoadedClass(name);
    if (c == null) {
        try {
            if (parent != null) {
                c = parent.loadClass(name, false);  // parent delegation
            } else {
                  c = findBootstrapClass0(name);
            }
        } catch (ClassNotFoundException e) {
            c = findClass(name);                    // if still not found, invoke findClass
        }
    }
    if(resolved) {
          resolveClass(c);
    }
    return c;
}
```

Parallel Class loading

- JDK7 made an important enhancement to Multi-threading custom class loader.
- Previously, certain class loader were prone to deadlock.
- JDK7 modifies the lock mechanism.
- Previously as well, it was not an issue which adhere the acyclic delegation model.
- http://openjdk.java.net/groups/core-libs/ClassLoaderProposal.html
- `protected static boolean registerAsParallelCapable()`

Deadlock Scenario

```
Class Hierarchy:
  class A extends B
  class C extends D

ClassLoader Delegation Hierarchy:

Custom Classloader CL1:
  directly loads class A
  delegates to custom ClassLoader CL2 for class B

Custom Classloader CL2:
  directly loads class C
  delegates to custom ClassLoader CL1 for class D
```

```
Thread 1:
  Use CL1 to load class A (locks CL1)
    defineClass A triggers
      loadClass B (try to lock CL2)

Thread 2:
  Use CL2 to load class C (locks CL2)
    defineClass C triggers
      loadClass D (try to lock CL1)
```

JDK7 - Resolution

```
Class Hierarchy:
  class A extends B
  class C extends D

ClassLoader Delegation Hierarchy:

Custom Classloader CL1:
  directly loads class A
  delegates to custom ClassLoader CL2 for class B

Custom Classloader CL2:
  directly loads class C
  delegates to custom ClassLoader CL1 for class D
```

```
Thread 1:
  <if parallel capable>
  Use CL1 to load class A (locks CL1 + A)
    defineClass A triggers
      loadClass B (try to lock CL2 + B)

Thread 2:
  <if parallel capable>
  Use CL2 to load class C (locks CL2 + C)
    defineClass C triggers
      loadClass D (try to lock CL1 + D)
```

End of Java ClassLoader Sub-system

Beginning of Java Runtime Compilers (JIT)

Why we need Runtime Compilers ?

- Interpreter are slow in nature. How much ?
- We always have scope of optimization of runtime code.
- Can we be biased towards a path which is 99.9 percent times right ?
- With more run, we get profiled data and optimizations can be done as per the information.
- Just-In-Time (JIT) Compiler comes with 2 flavors to address all these :-
    - C1 - Less optimized, fast startup - perfect for UI (Client) app.
    - C2 - More optimized, slow startup - perfect for Server app.

In working, where I see JIT ?

- Whenever you use any of these flags :-

  - -client [C1]
  - -server [C2]
  - -XX:+TieredCompilation [Mix of C1, C2]
  - -Xint [Only interpreter]
  - -Xcomp [Only compiler]
  - -Xmixed [Mix of interpreter and compiler]

Optimizations_1 (Dead code removal + Expression Opt)

```
void someCalculation(int x1, int x2, int x3) {
int res1 = x1+x2;
int res2 = x1-x2;
int res3 = x1+x3;
(res1+res2)/2;
}
```

↓

```
void someCalculation(int x1, int x2, int x3) {
x1;
}
```

Optimizations_2 (Monomorphic Dispatch)

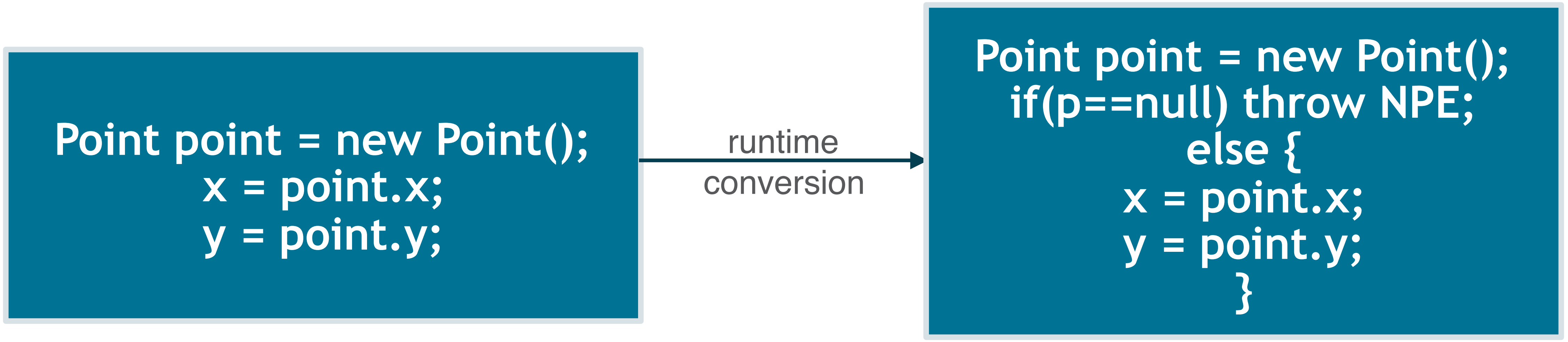
```
public class Animal {
private String color;
public String getColor() {
return color; } }

Animal a = new Animal();
a.getColor();
```

If not overridden

```
Animal a = new Animal();
a.color;
```

Optimizations_3 (Null Check Optimization)

- How Java handles Null Pointer ?



```
Point point = new Point();
x = point.x;
y = point.y;
```

```
Point point = new Point();
if(p==null) throw NPE;
else {
x = point.x;
y = point.y;
}
```

- if profiled information says that, p is never going for null
  - remove the if-else condition
  - insert UnCommonTrap [path to de-optimization]

New VM Based Languages (Kotlin Example)

```
fun main(args : Array<String>) {
    var a: String = "abc"
    a = null // compilation error

    var b: String? = "abc"
    b = null // ok

    val l = a.length //ok

    val l = b.length // error: variable 'b' can be null
    val l = if (b != null) b.length else -1
}
```

Optimizations_3_4 (Inlining and Caching)

- Inlining is one of the most powerful optimization. Method call replaced by the method body if function is pure.
- Inline is method is < 35 KB
- Inlining in Java is more powerful than languages like C++

- Caching is also a good optimization technique
  - p.x = (p1.x + p2.x) / 2 ; p.y = (p1.y + p2.y) / 2

Optimizations_5_6 (Thread and Loop Optimization)

- Eliminate lock if not reachable from other thread.
- Join sync blocks on the same object, if possible.
- BaisedLocking

- Loop combining - if possible
- Replacing while from do-while
- Tiling Loops - Fitting into the cache exactly

Real life challenges and Future

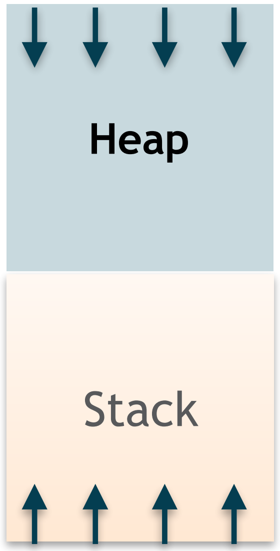
- Higher start-up time - Not a good idea in low latency systems.
- Different techniques.

- Future may be better (AOT, AppCDS …)

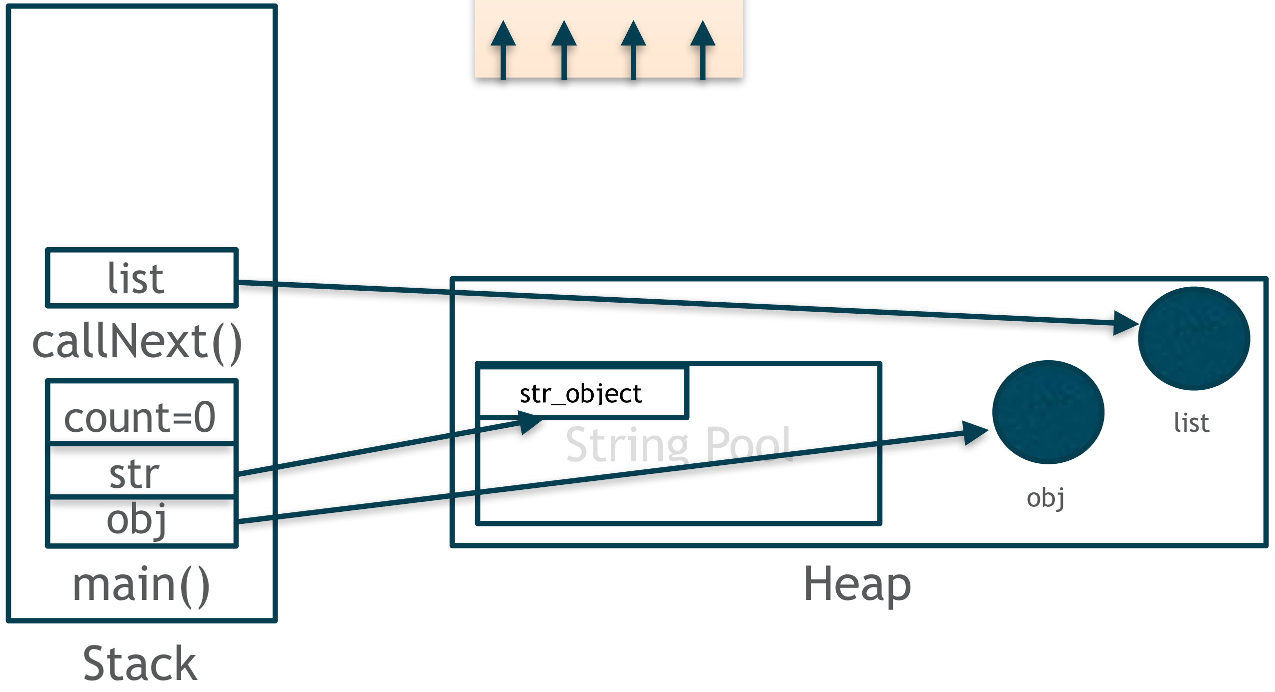
End of Java Runtime Compilers (JIT)

Beginning of Java Garbage Collectors

Stack and Heap in Java

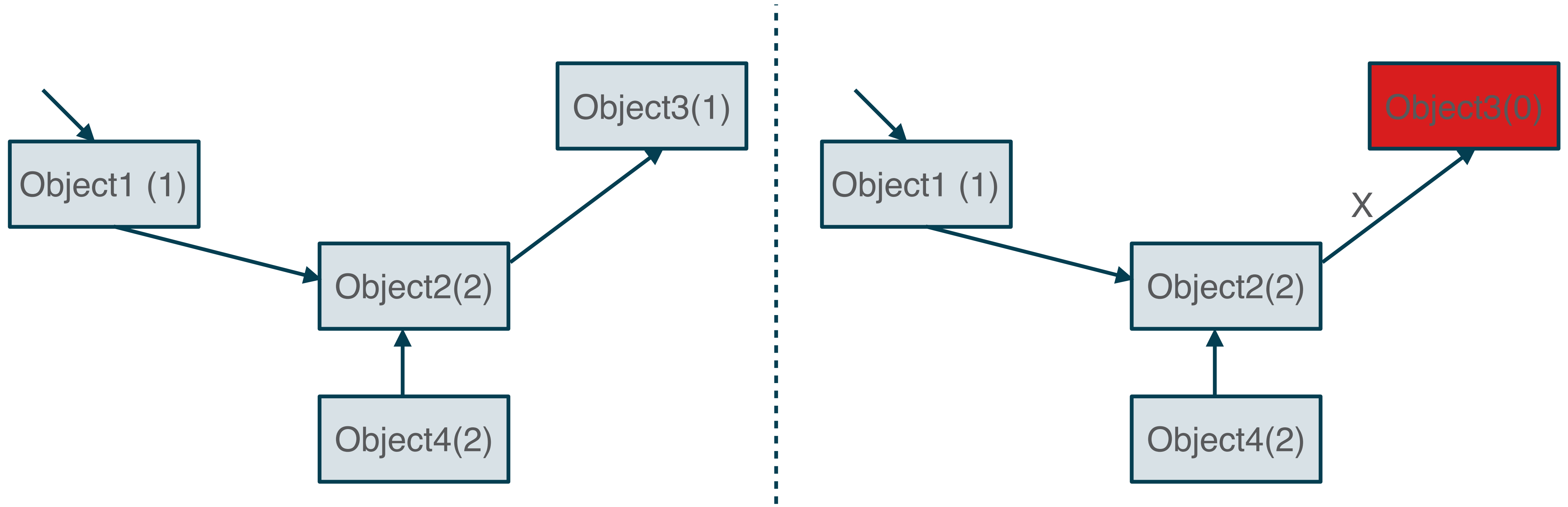


```
public static void main(String[] args) {

     Object obj = new Object();
     String str = new String();
     int count = 0;
     // Do some work here
     callNext();

}

public void callNext() {

     List list = new ArrayList();
     // some work
}
```



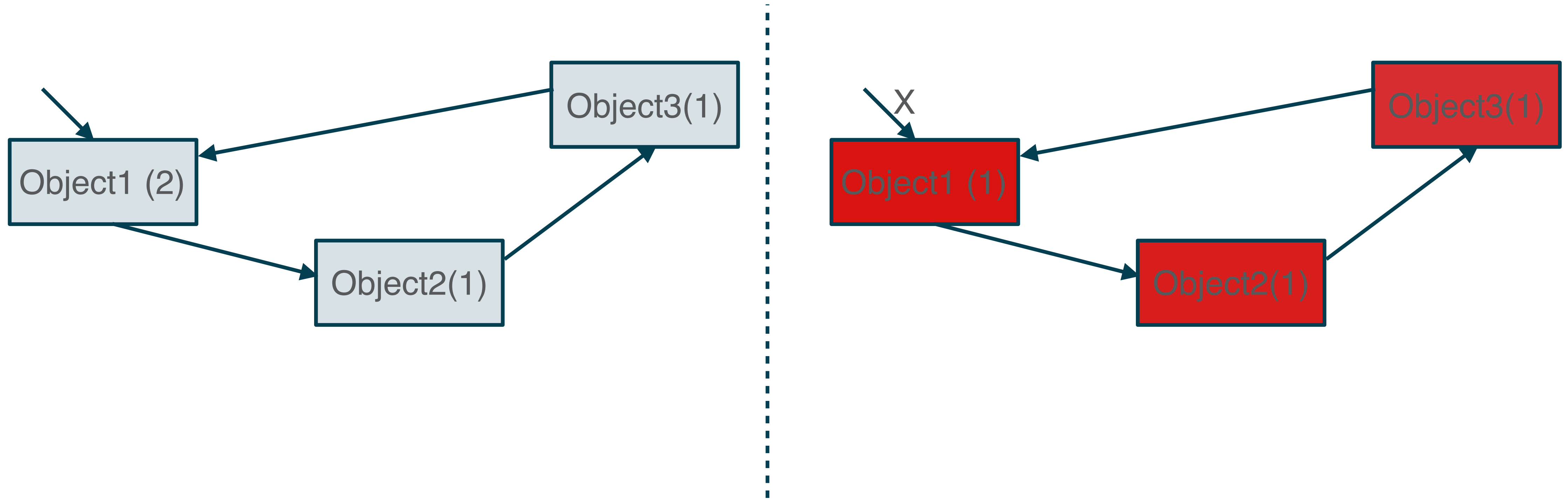
Initial Thoughts

- Reference Count for all the “Object” Instance.
- If refCount == 0, should be eligible for GC.



Initial Thoughts

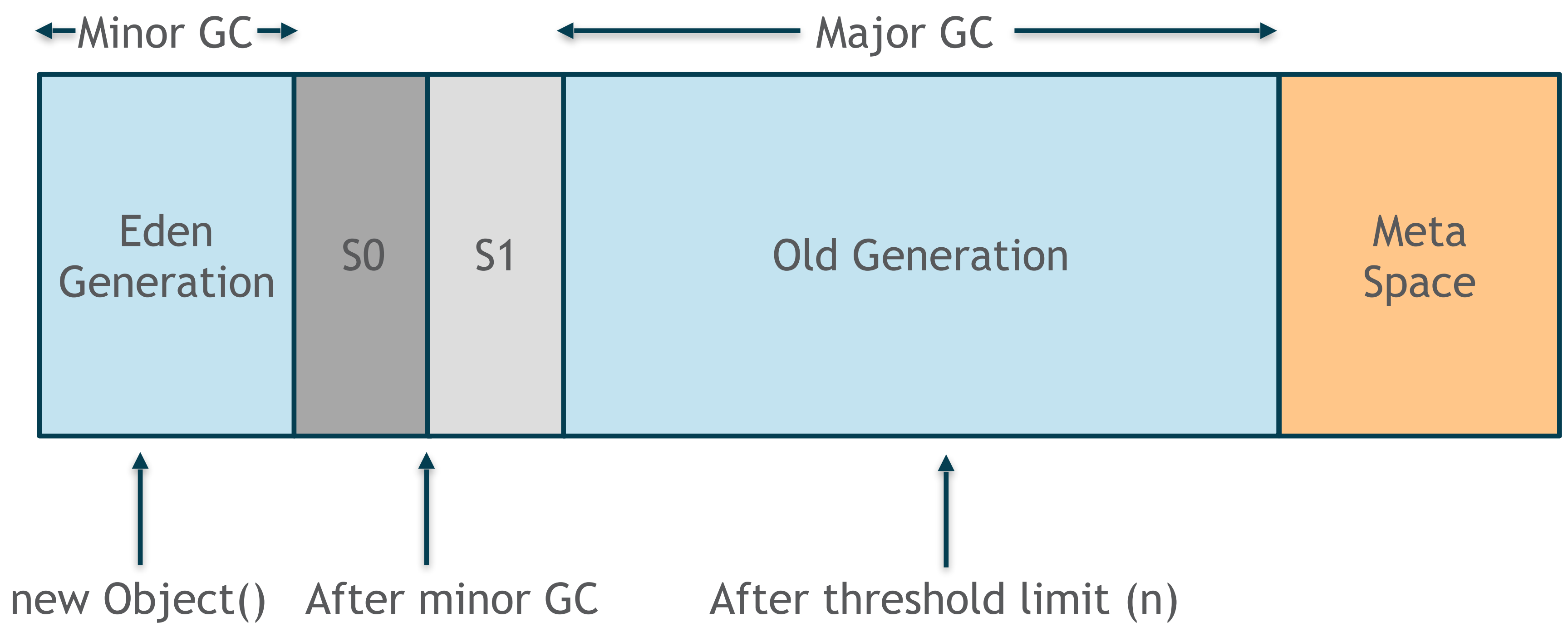
- But tend to create “Achilles Heel”
- If refCount != 0, should be eligible for GC ?



Definition

- **Mark**
  - ➤ find and mark all accessible objects
- **Sweep**
  - ➤ scans through the heap and reclaims all the unmarked objects
- **Copy**
  - ➤ copies all live objects from one part of the heap to another empty part
- **Compaction**
  - ➤ moving all the live objects into contiguous memory locations

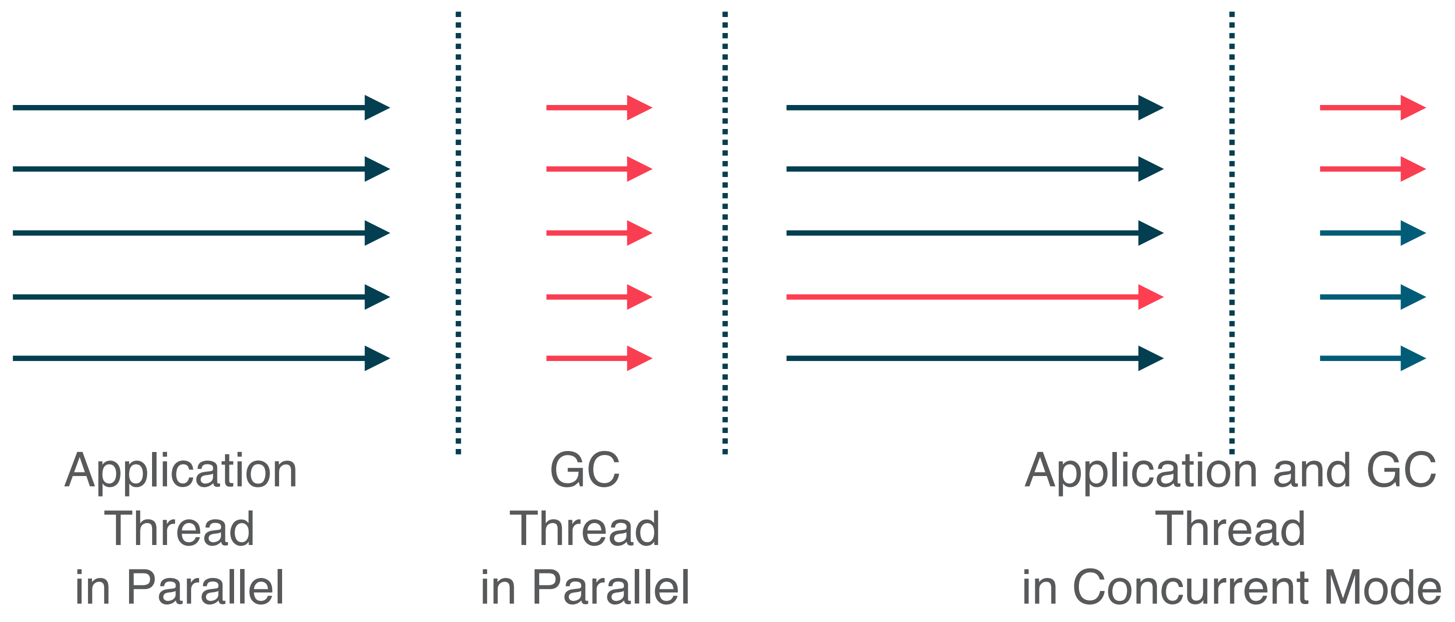
Java Heap Structure



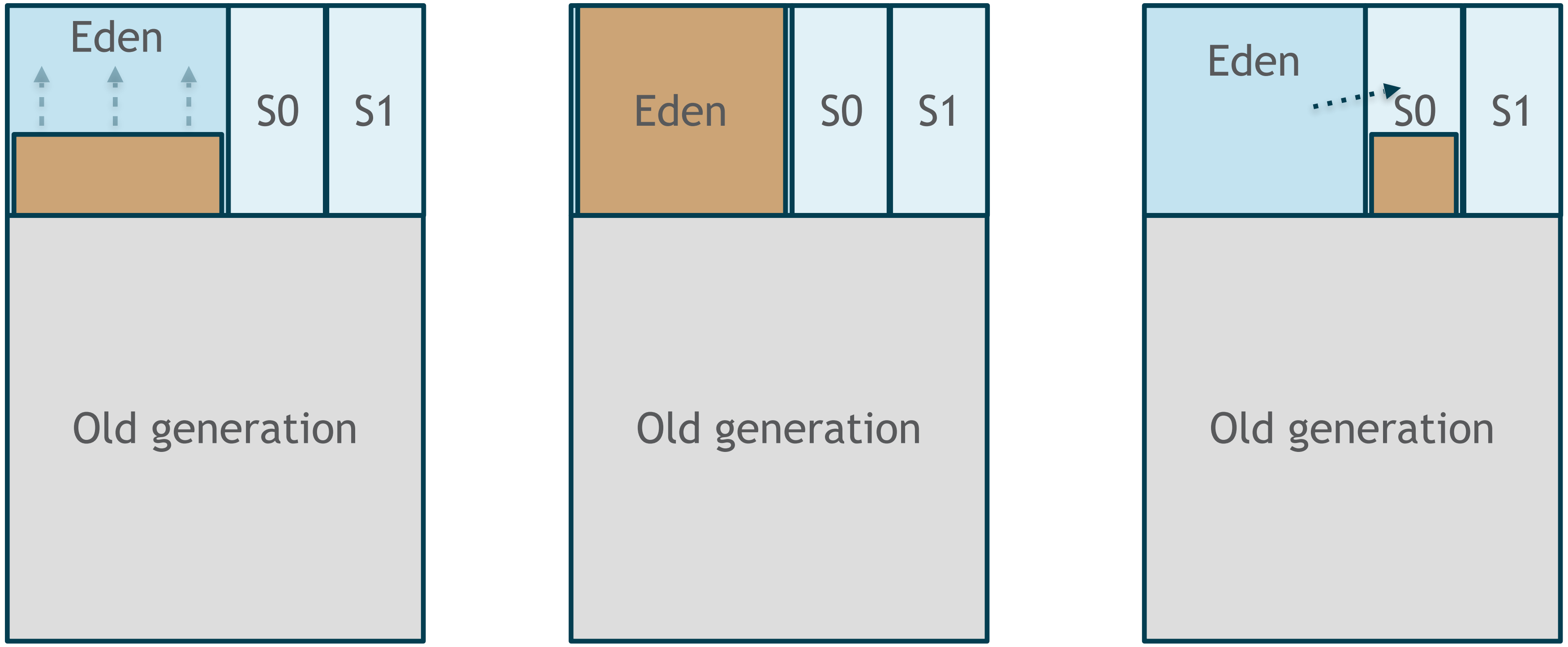
Weak Generational Hypothesis :

- Most of the object die young.
- There are very few old to young reference.

Parallel And Concurrent

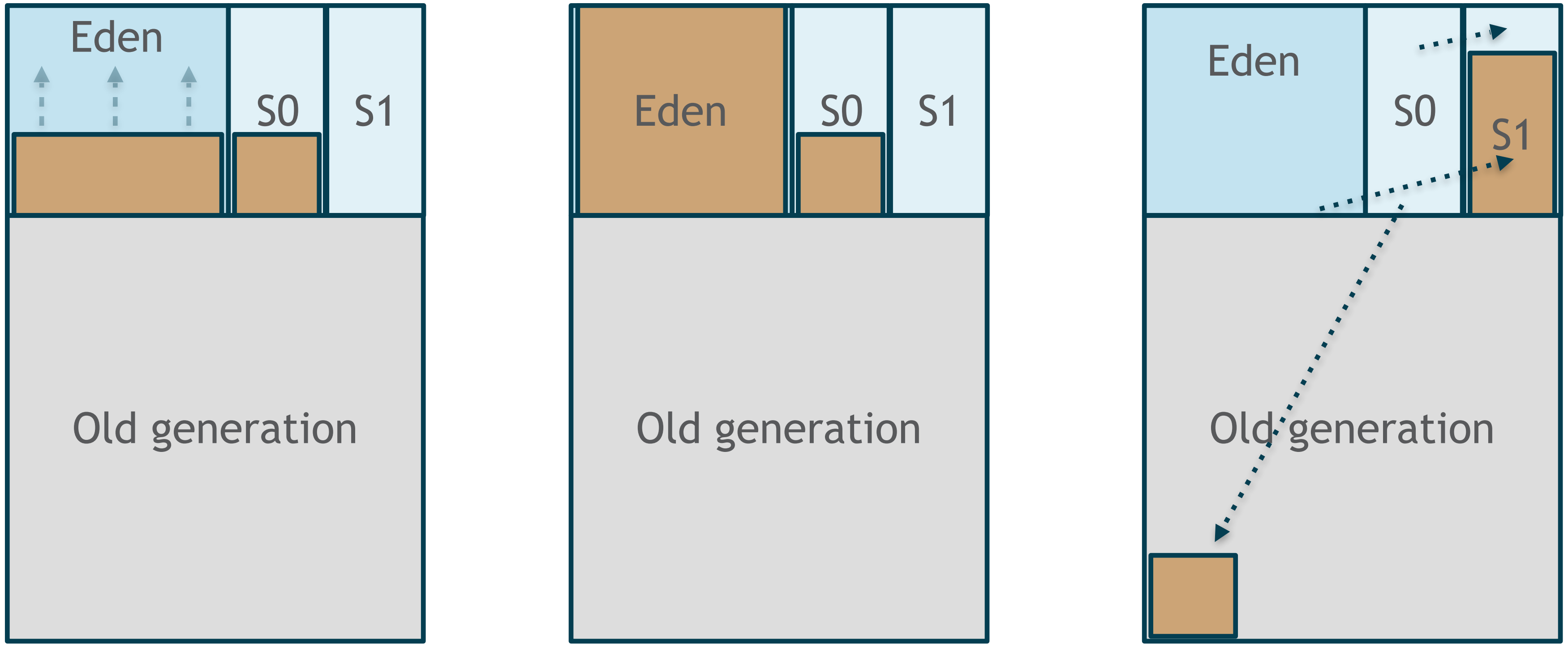


CMS Simplified (In Young generation)…



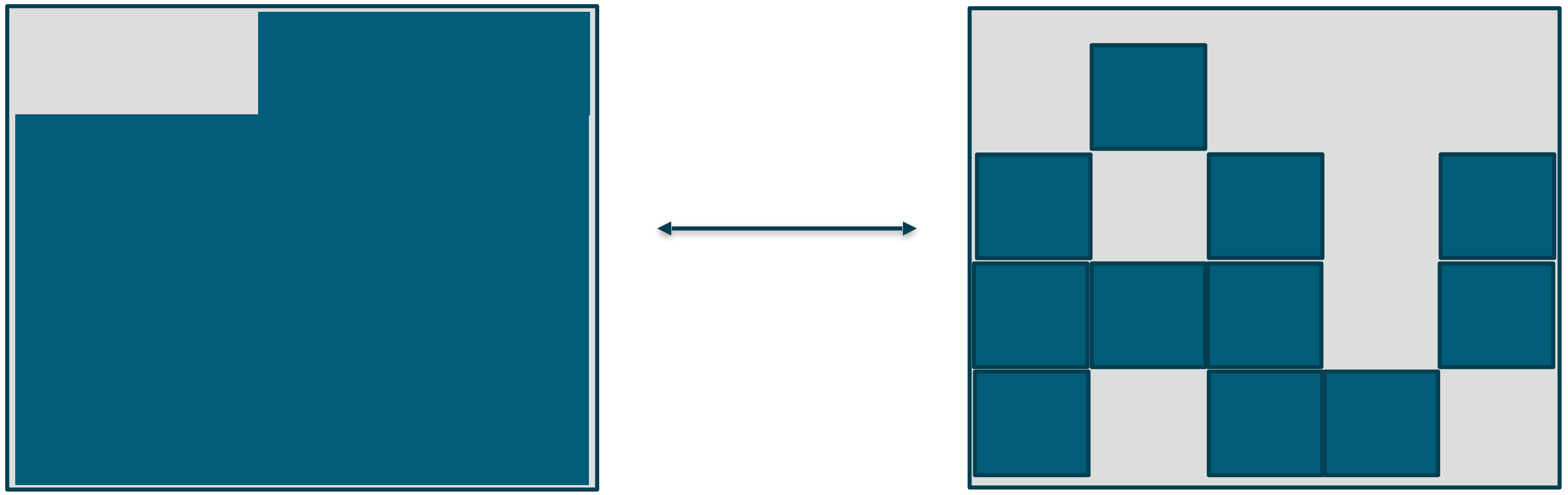
When eden space will be full. Minor GC will be triggered.
Usages of S0 and S1.

CMS Simplified (In Young generation)…



Empty Eden and one of the Survivors after Minor GC completion.
Stop the world process.
Some objects can be promoted to Old generation.

CMS Simplified (In Old generation)…



May be the first snapshot of Old generation

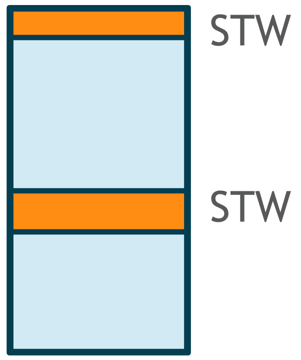
Old generation will mostly look like this

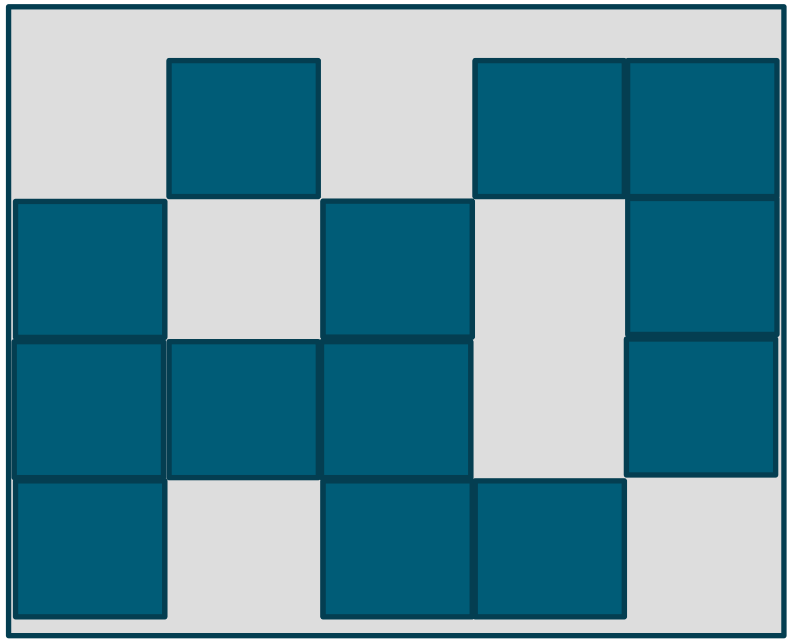
No compaction in CMS
Mostly concurrent

CMS Simplified (In Old generation)…

CMS phases :-

1. Initial marking phase.
2. Concurrent Marking phase.
3. Concurrent Pre-cleaning Phase.
4. Remarking phase.
5. Concurrent Sweep Phase.
6. Concurrent Reset Phase.





CMS Simplified (In Old generation)…

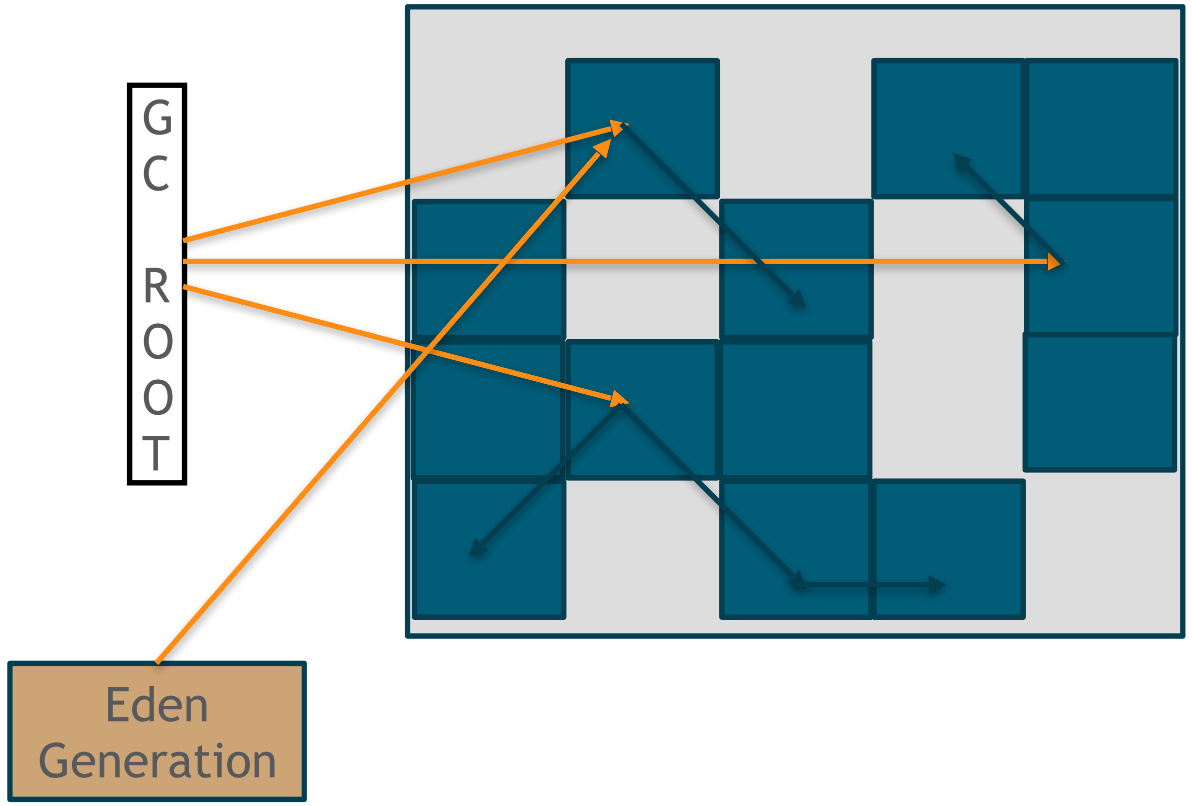
Initial Marking Phase :-

1. Scan the object directly reachable from root or from Eden Generation.

2. All the mutator thread need to be stopped (STW).

3. STW - We don't want to mess up with this information.

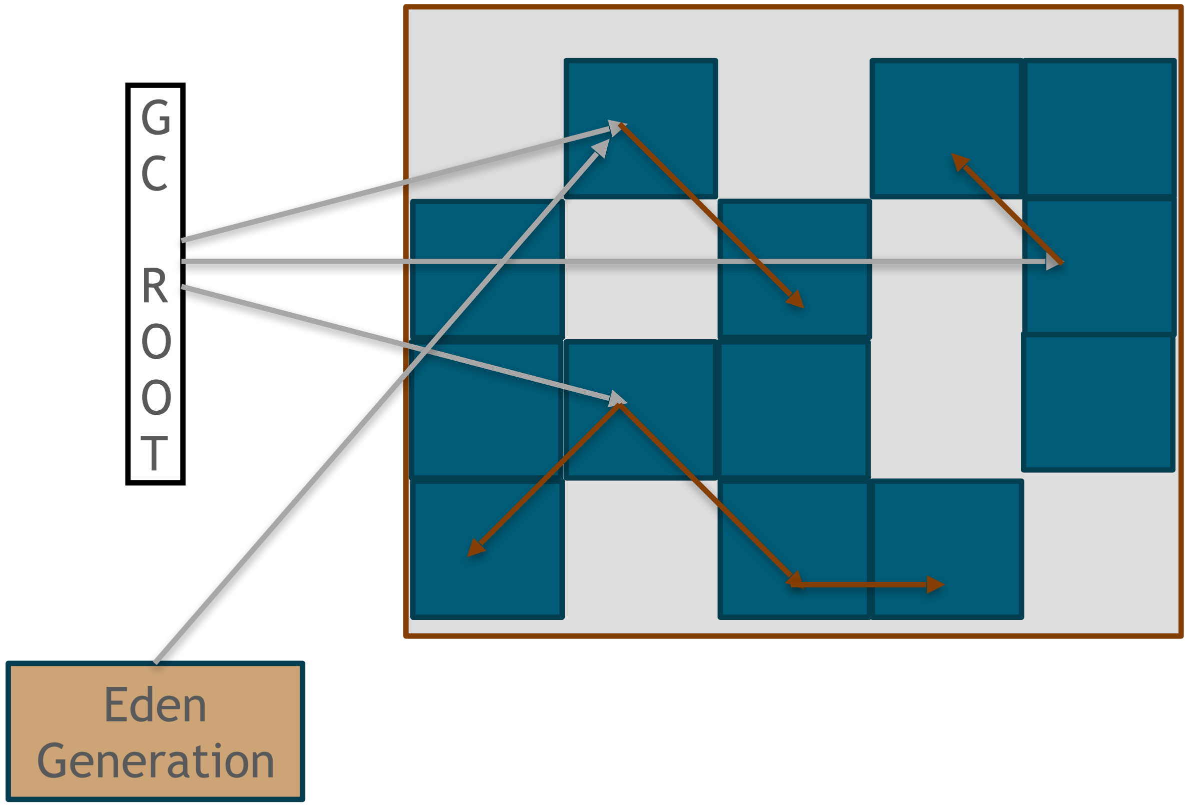
4. Generally small pause in nature.



CMS Simplified (In Old generation)…

Concurrent Marking Phase :-

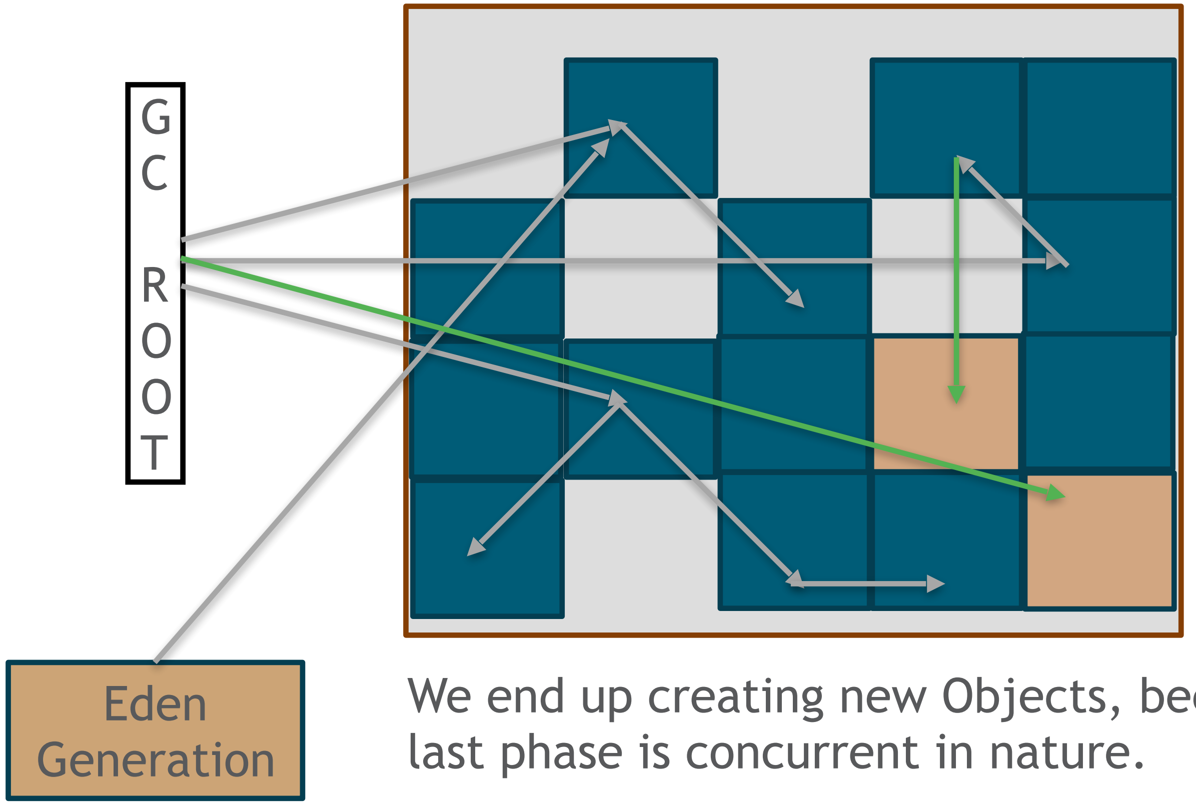
1. Scan **almost** all live objects from the object marked from phase 1.
2. This will be done in concurrent fashion.
3. And this will lead to a remarking phase.
4. It will take time but it's concurrent.



CMS Simplified (In Old generation)…

Remarking Phase :-

1. Scan the newly created objects.
2. We need to do STW.
3. Generally, small pause in nature.



We end up creating new Objects, because last phase is concurrent in nature.

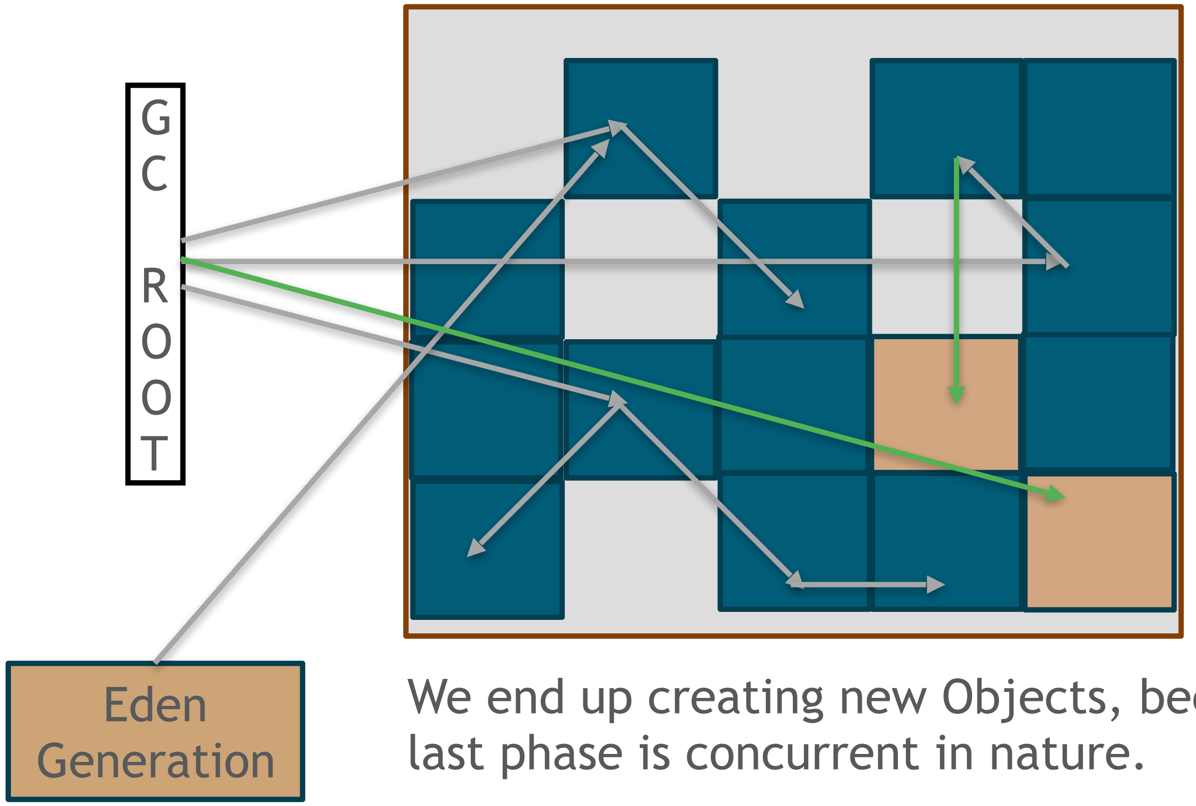
CMS Simplified (In Old generation)…

Remarking Phase :-

1. Scan the newly created objects.

2. We need to do STW.

3. Generally, small pause in nature.



We end up creating new Objects, because last phase is concurrent in nature.

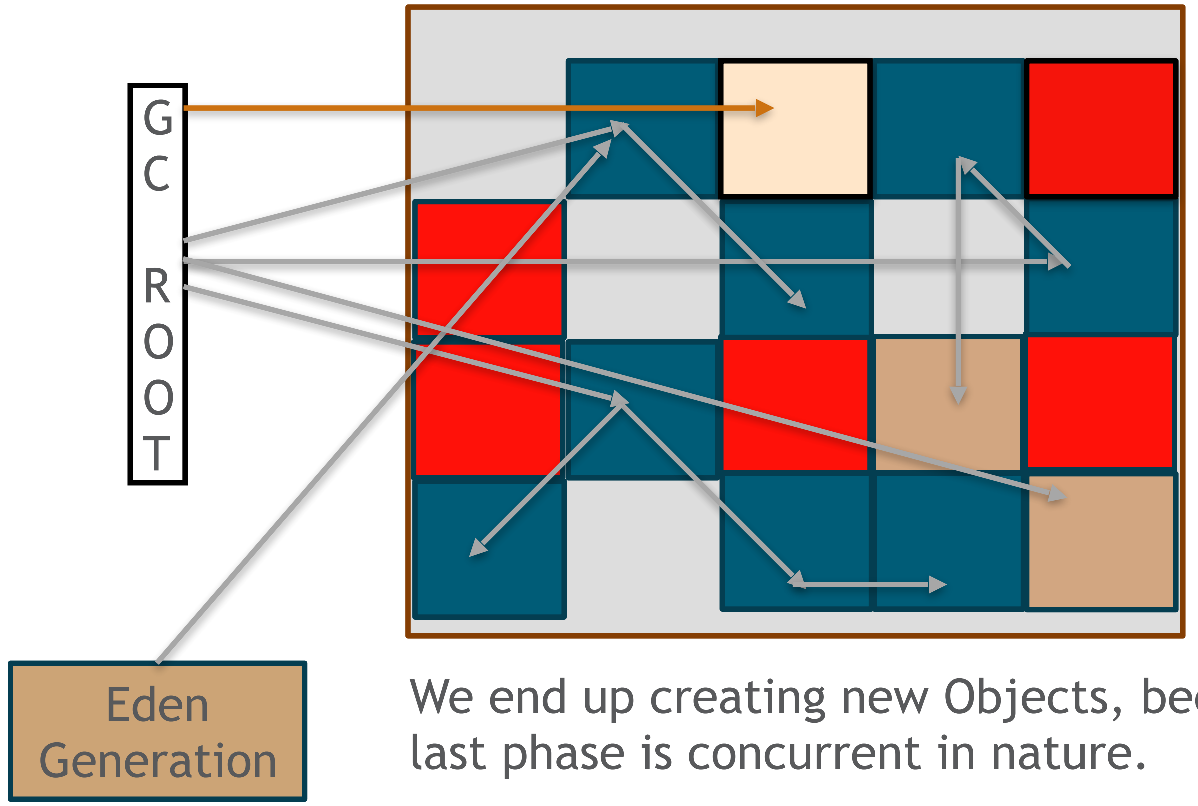
CMS Simplified (In Old generation)…

Concurrent Sweep Phase :-

1. Start sweeping all the non-marked Object.

2. Concurrent in nature.

3. Red boxes can be claimed by GC.



We end up creating new Objects, because last phase is concurrent in nature.

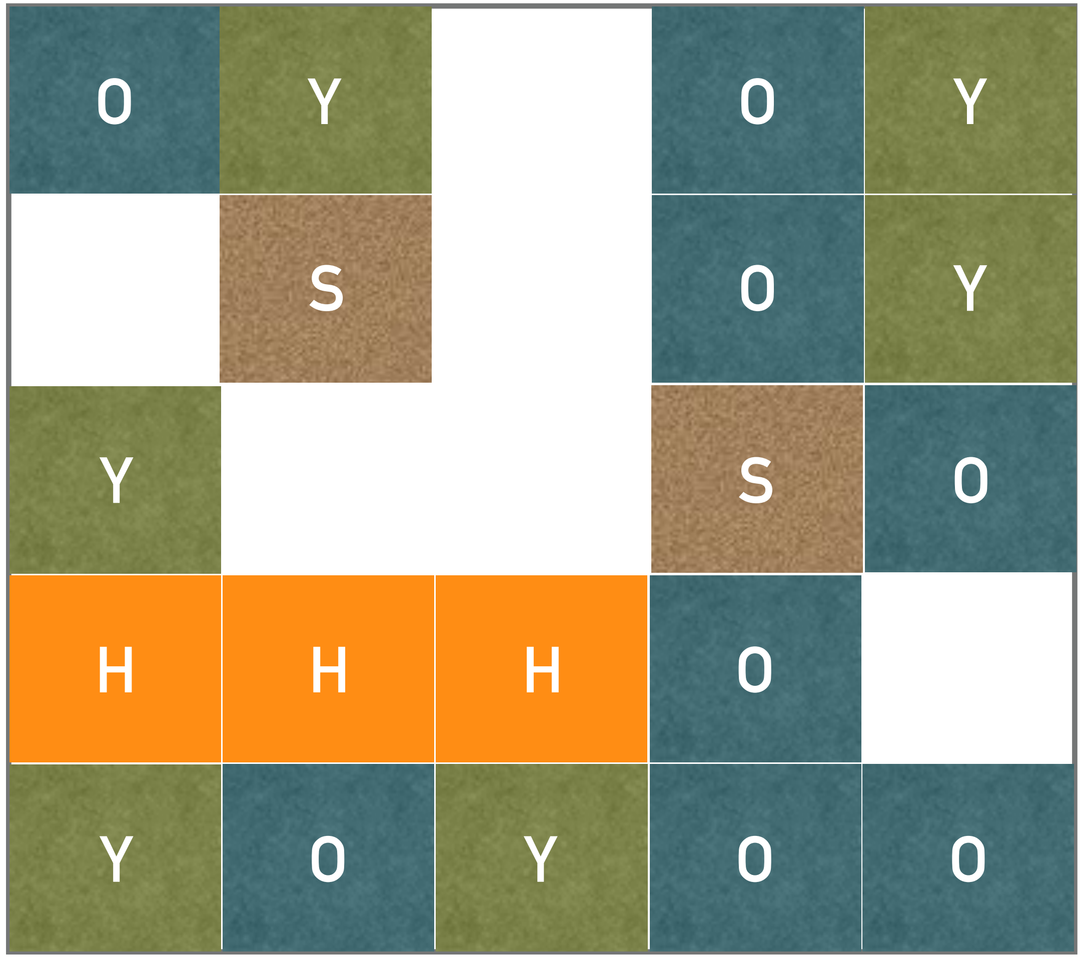
Final thoughts for CMS

- It’s almost concurrent other than some small STW’s.
- Memory Fragmentation - so can’t fit large object.
- Less predictable.
- Can Lead to **promotion failure -** Young to old promotion is not happening because object is too big or old space is almost full.
- Can Lead to **Concurrent mode failure** - Old generation not finished the collection work and Old generation is full. STW and then run a different GC algorithm, probably Mark-Sweep-Compact.
- If application is working fine with CMS, let it work !

Welcome to Garbage First (G1)

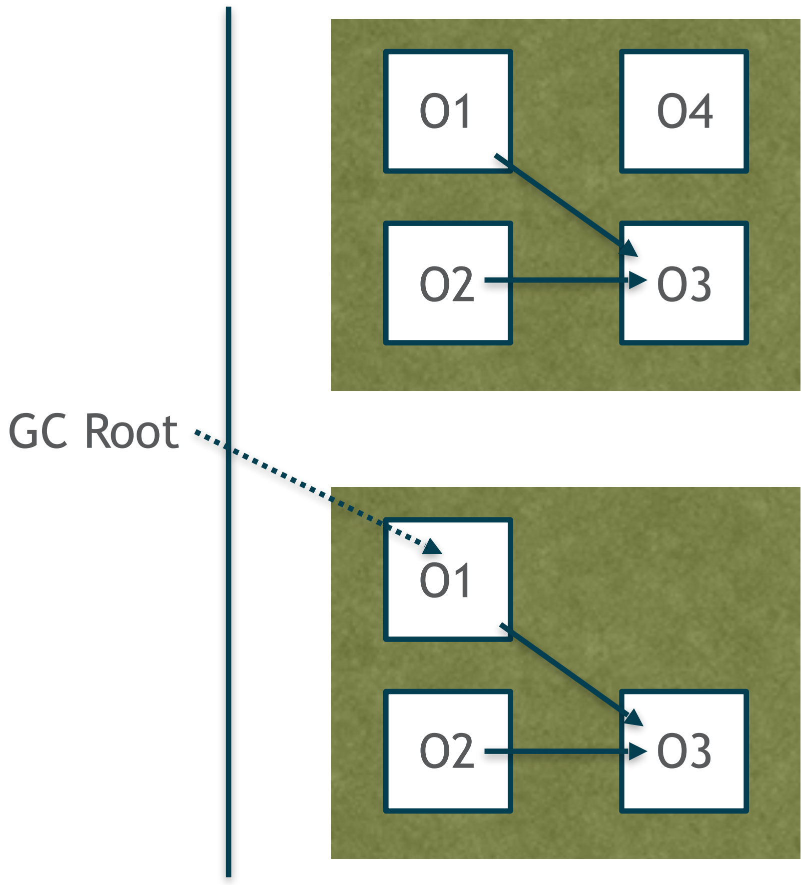
- Default GC algorithm for JDK9.
- G1 Goals
  - Low latency
  - Predictable (Can’t be 100 percent)
  - Easy to use (Less parameter settings)
- Concurrent, Parallel and better Compacting.
- If you are not on JDK9, use -XX:+UseG1GC.
- Careful with your greedy throughput desires.

Garbage First (G1) - Memory layout



- Memory is divided into small regions
- More than 2000 regions
- More flexible boundaries
- Use -XX:+G1HeapRegionSize
- Different regions :
  - Young
  - Survivor
  - Old
  - Humongous

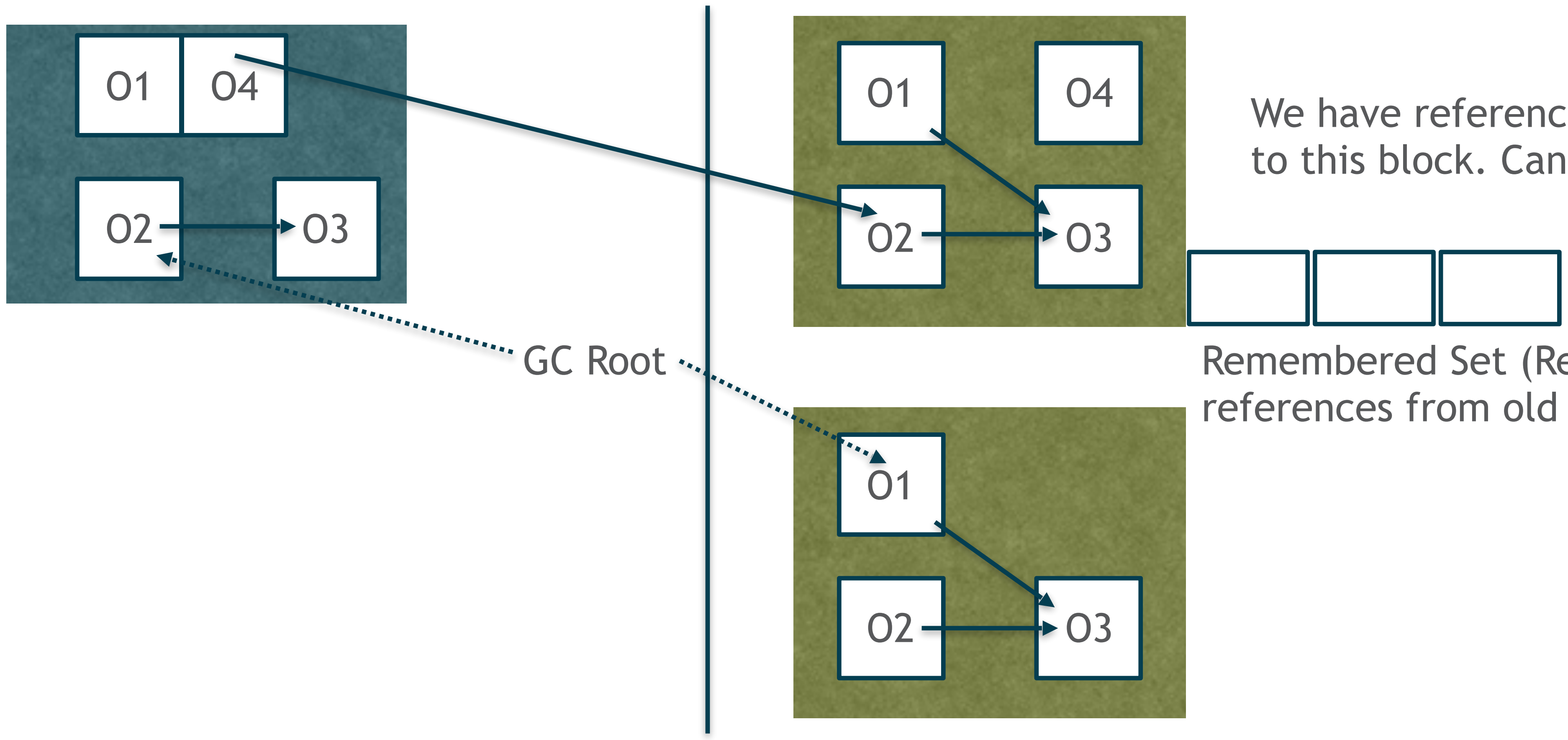
G1 - Young region View



None of the object is reachable from GC root in first block. Looks like the complete region can be collected. **(No you can't)**

GC root is pointing to some object. Can't collect this region.

G1 - Young region View



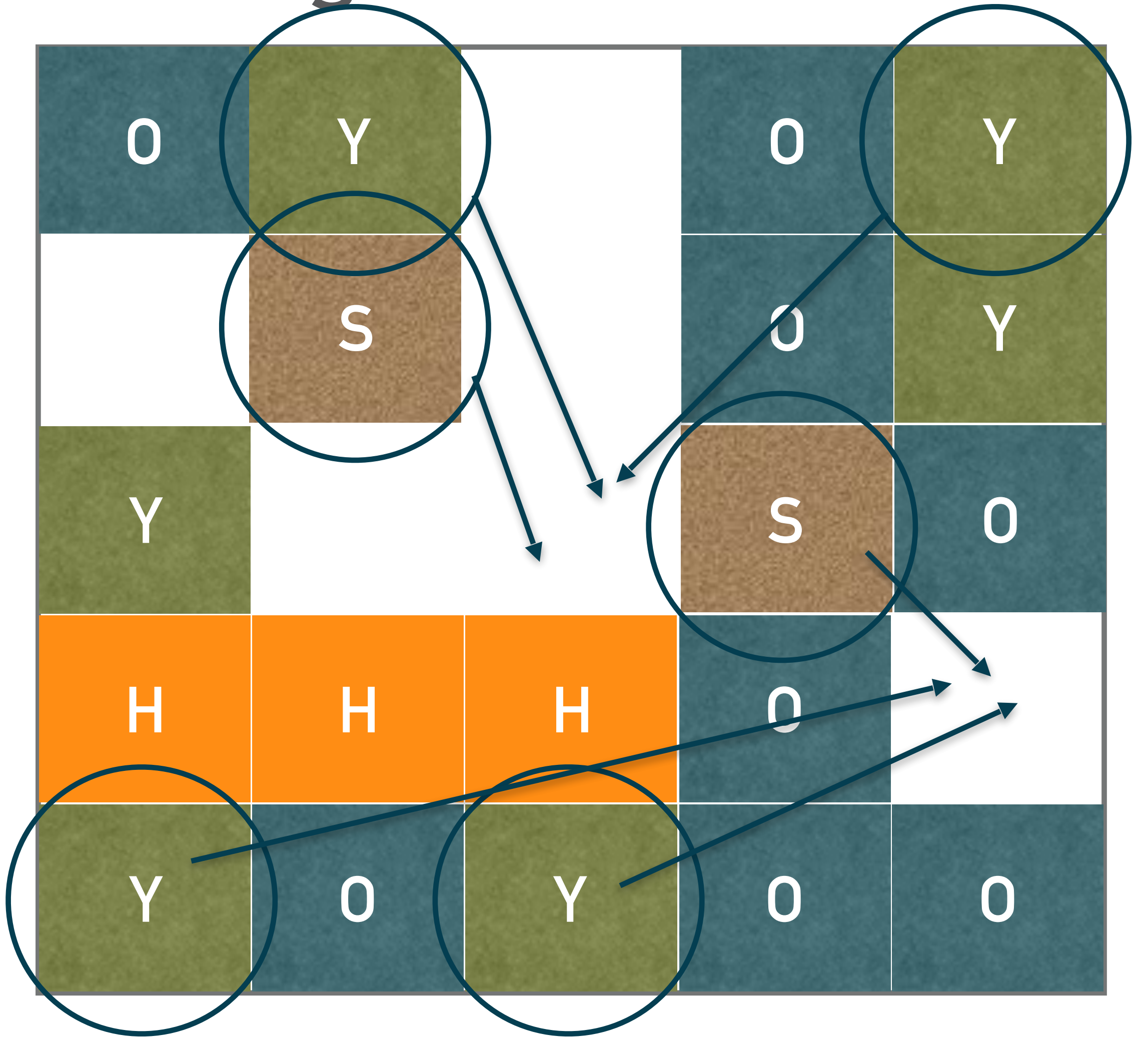
We have reference from Old region to this block. Can’t be collected.

Remembered Set (Remembers the references from old region)

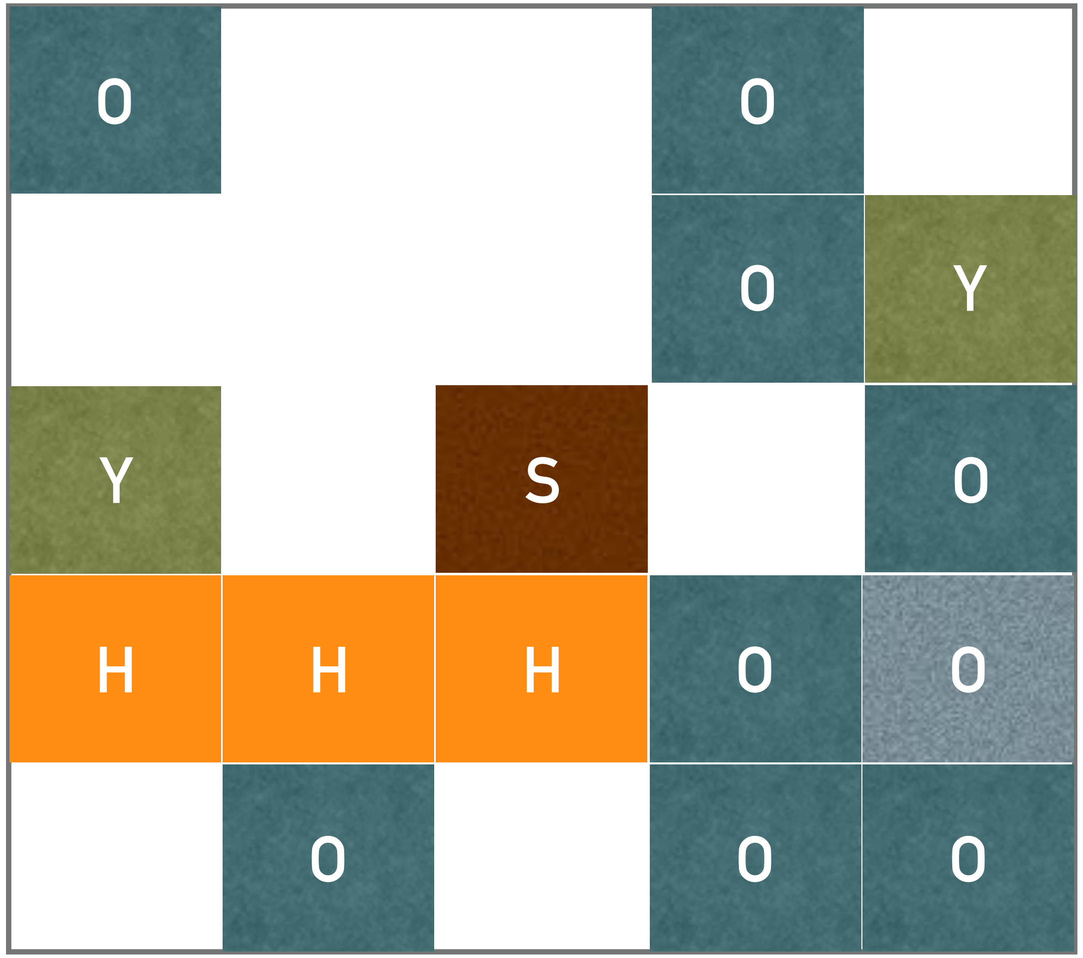
G1 - Ease to use

- java -Xmx50m -XX:+UseG1GC -XX:+MaxGCPauseMillis=200 -jar Java2DDemo.jar
- MaxGCPauseMillis=200 - A soft goal. G1 will try to respect as much as possible, but can’t guarantee you.
- Important to know
  - -XX:+InitiatingHeapOccupancyPercentage=45

G1 - Young GC View



G1 - Young GC View



G1 - Best Practices

- Avoid setting up too much of parameter.
  - You will end up G1 behaving abruptly.
- Avoid full GC. Look at the adaptive policy.
- Avoid allocation Failure
  - Very similar to Concurrent Mark Failure in CMS.
  - It will call STW and a full GC
  - Issue with heap or with the code
- Careful with Humongous Regions. Change region size, if it is more.

End of Java Garbage Collectors

Understood Java Virtual Machine

Vaibhav Choudhary (@vaibhav_c)
Java Platforms Team, Oracle
https://blogs.oracle.com/vaibhav

For any query, mail at
vaibhav.x.choudhary@oracle.com

Java
Your
Next
(Cloud)