

Java 꺾 잡아!

- JVM부터 GC, 스레드 동기화까지!

**[2-1] GC(Garbage Collection)의 정의와
Java GC 알고리즘에 대해 살펴봅니다.**

[1] 가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

* Memory allocation

Memory allocation examples

Example in C language:

```
//allocate memory  
int* array = (int*)malloc(sizeof(int)*20);  
  
//explicitly deallocate memory  
free(array);
```

Example in Java that employs automatic memory management:

```
//allocate memory for String object  
String s = new String("Hello World");  
  
//no need to explicitly free memory occupied by 's'. It would be released by the  
garbage collector when 's' goes out of scope.
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved. |

7

https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf

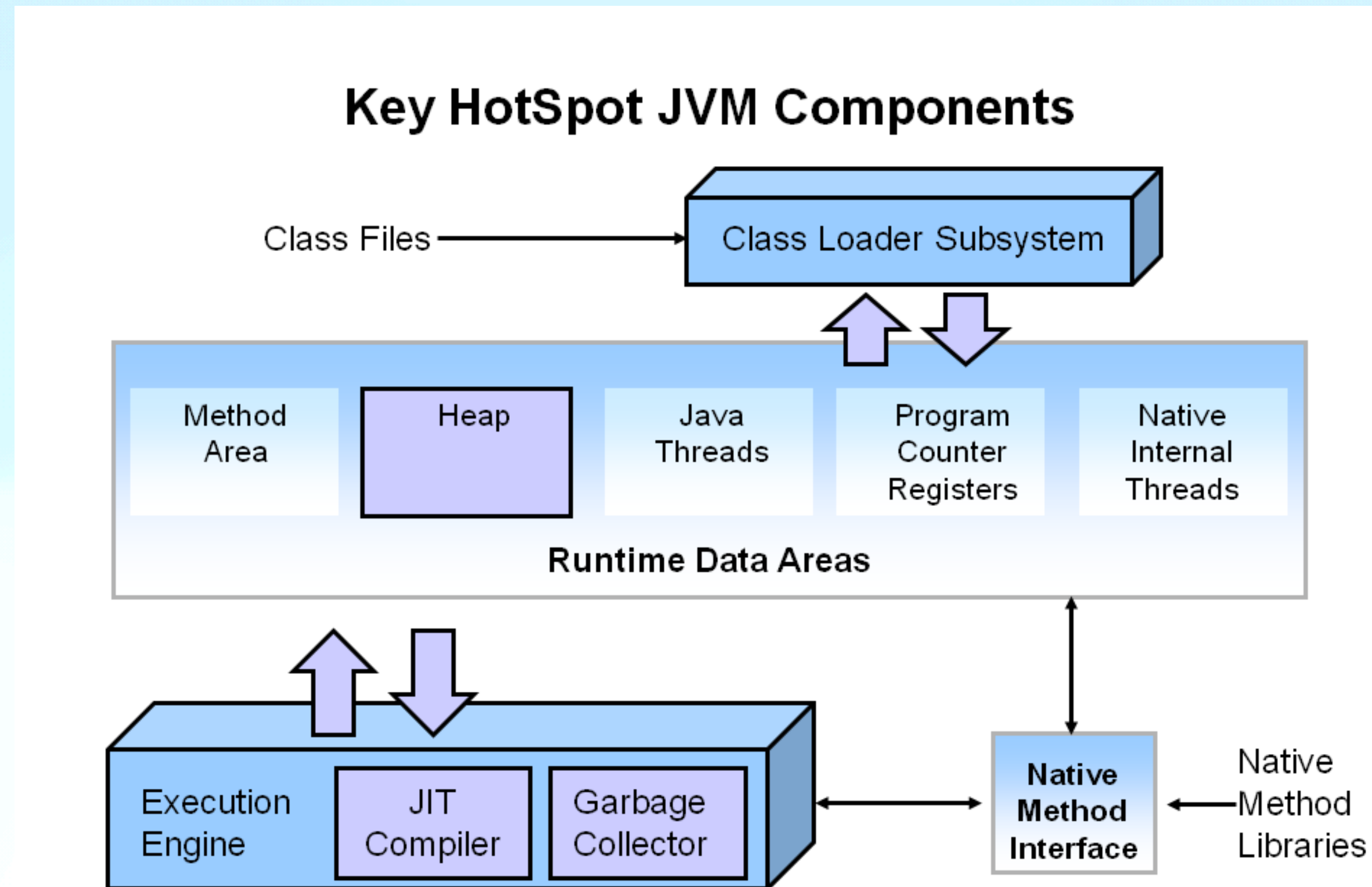
가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Garbage Collection

- Garbage Collection은 자동 메모리 관리의 한 형태이며 결과적으로 성능에 영향을 미침
 - Garbage는 더이상 사용(참조)되지 않는 메모리
- 프로그래머가 메모리 할당/해제 등 수동 관리의 부담을 덜어줌
 - 비슷한 기술로는 스택 할당, 영역 추론, 메모리 소유권 등이 있음
- 일반적으로 네트워크 소켓, DB 핸들, 윈도우, 파일 디스크립터 등과 같은 리소스는 GC 처리되지 않음
보통 이와 같은 것들은 다른 방법들로(destructors) 메모리 할당이 처리됨
- Garbage Collector는 이 작업(자동 메모리 관리)를 수행하는 프로그램

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Garbage Collection



<https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

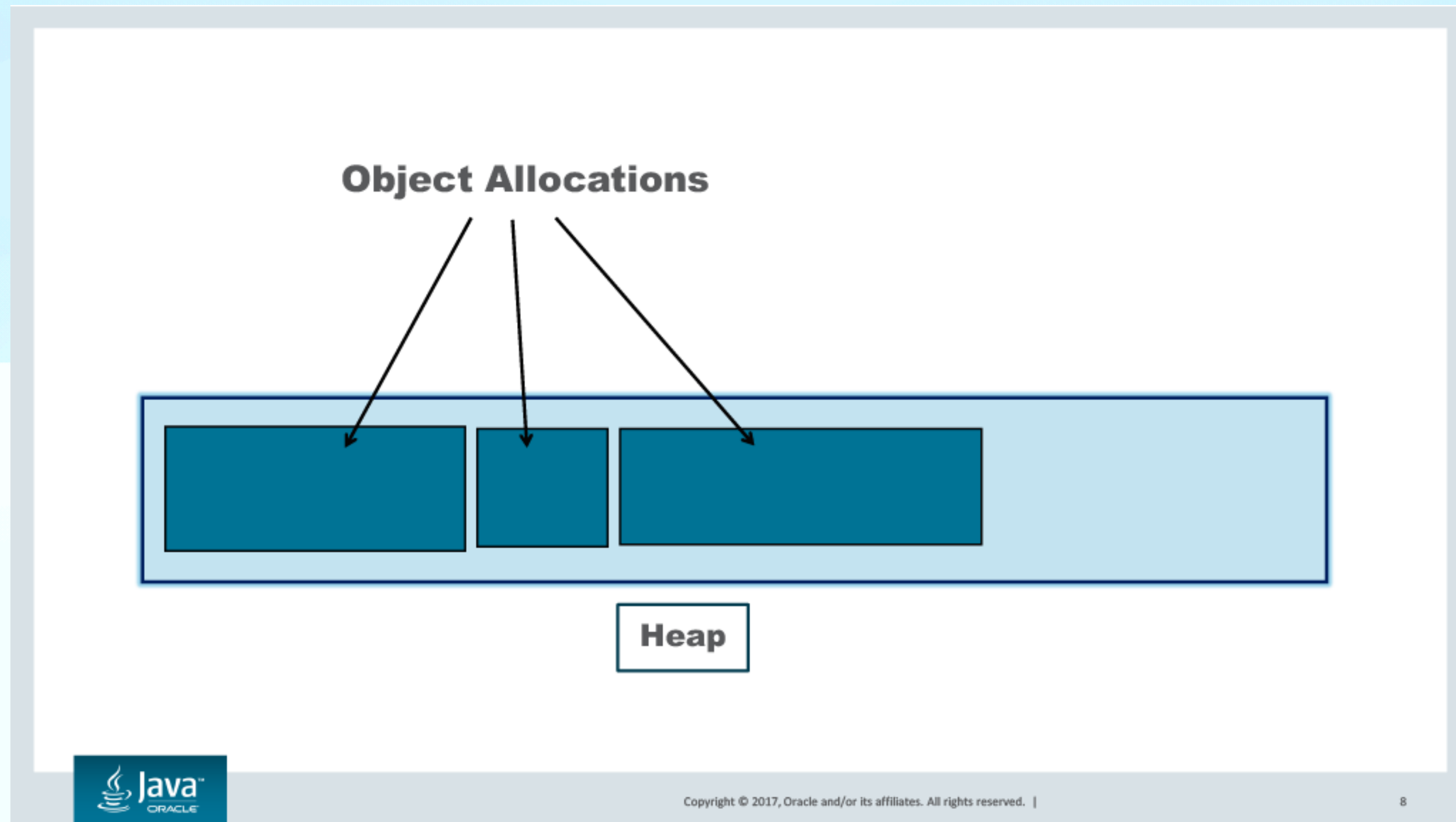
가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Garbage Collectors

- 힙 메모리를 분석해 사용하지 않는 객체를 식별해 삭제하는 프로세스
- 사용 객체는 어디선가 여전히 참조중인(포인터 유지) 객체
반면에 미사용 객체는 어떤 곳에서도 참조(사용)되지 않아 메모리 회수 가능

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

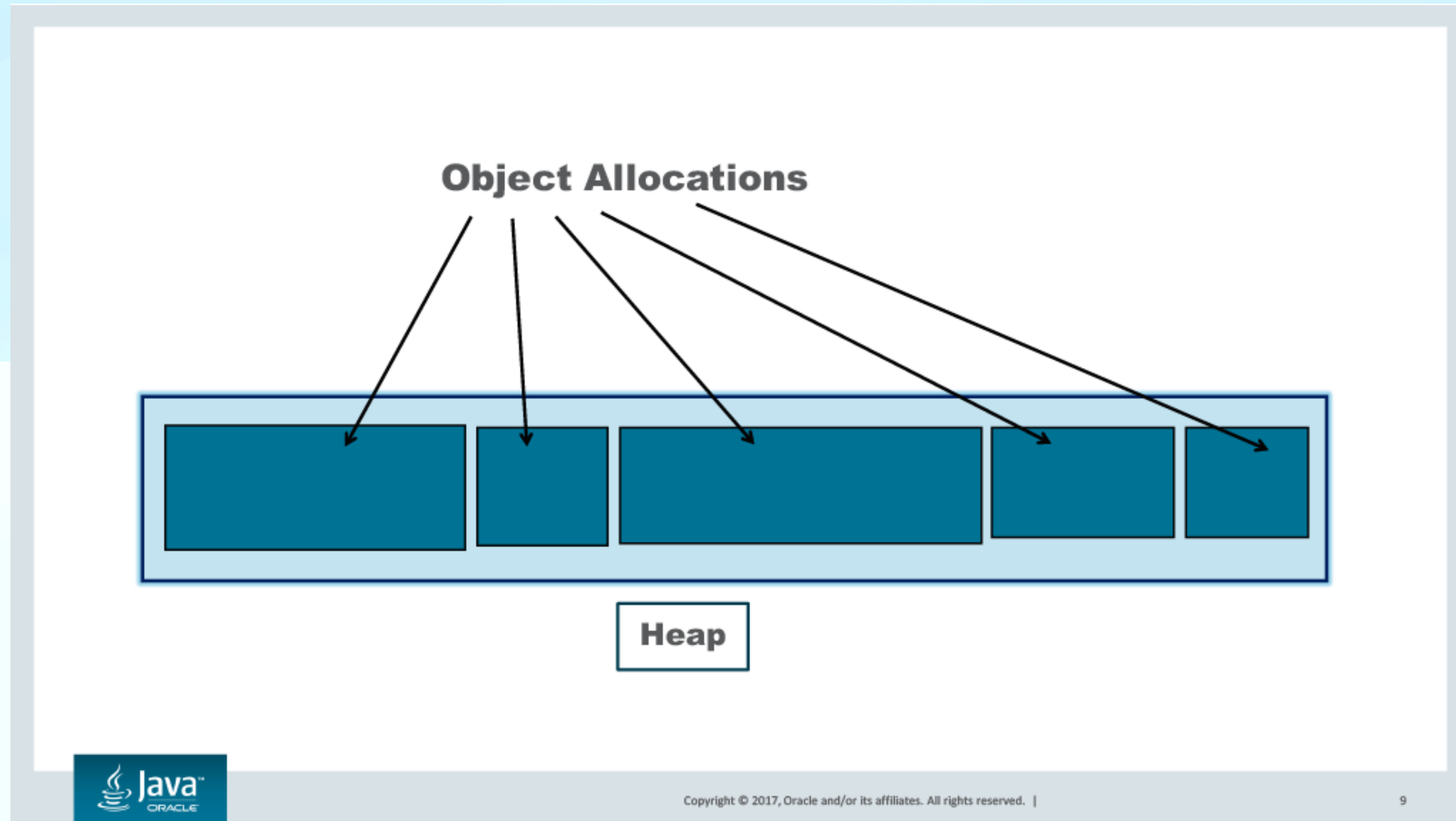
GC Tracing Step



https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

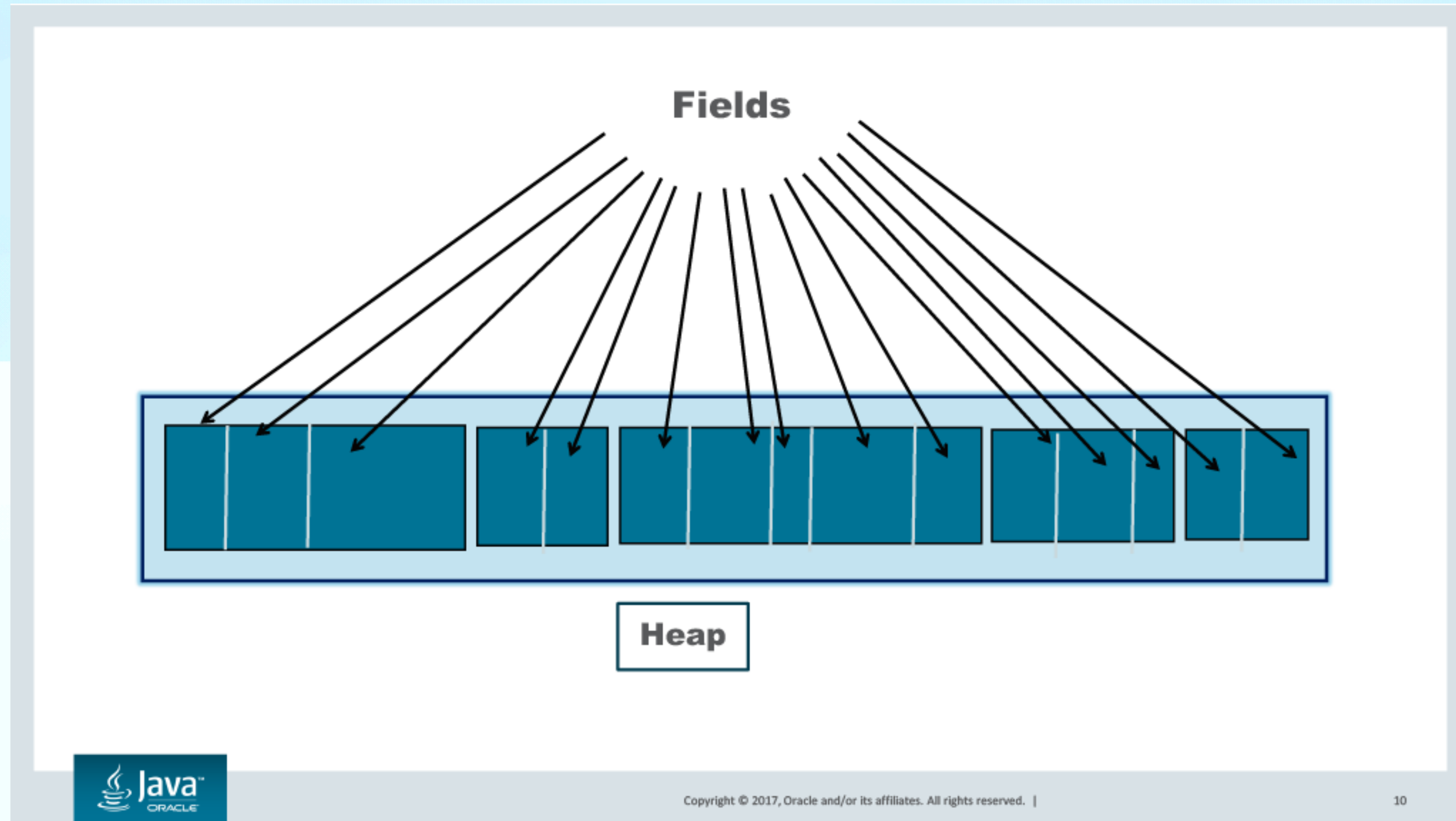
GC Tracing Step



https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

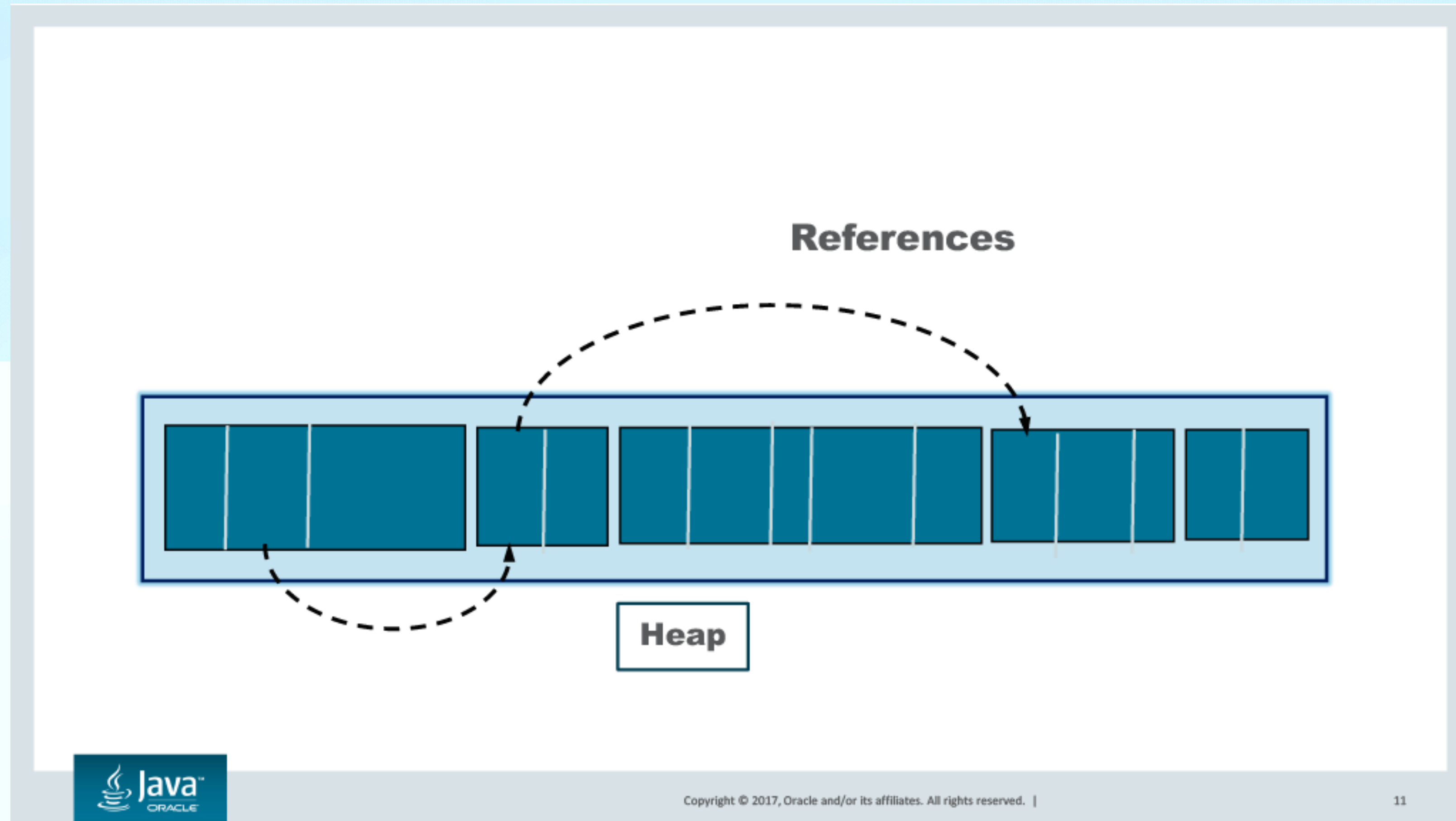
GC Tracing Step



https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

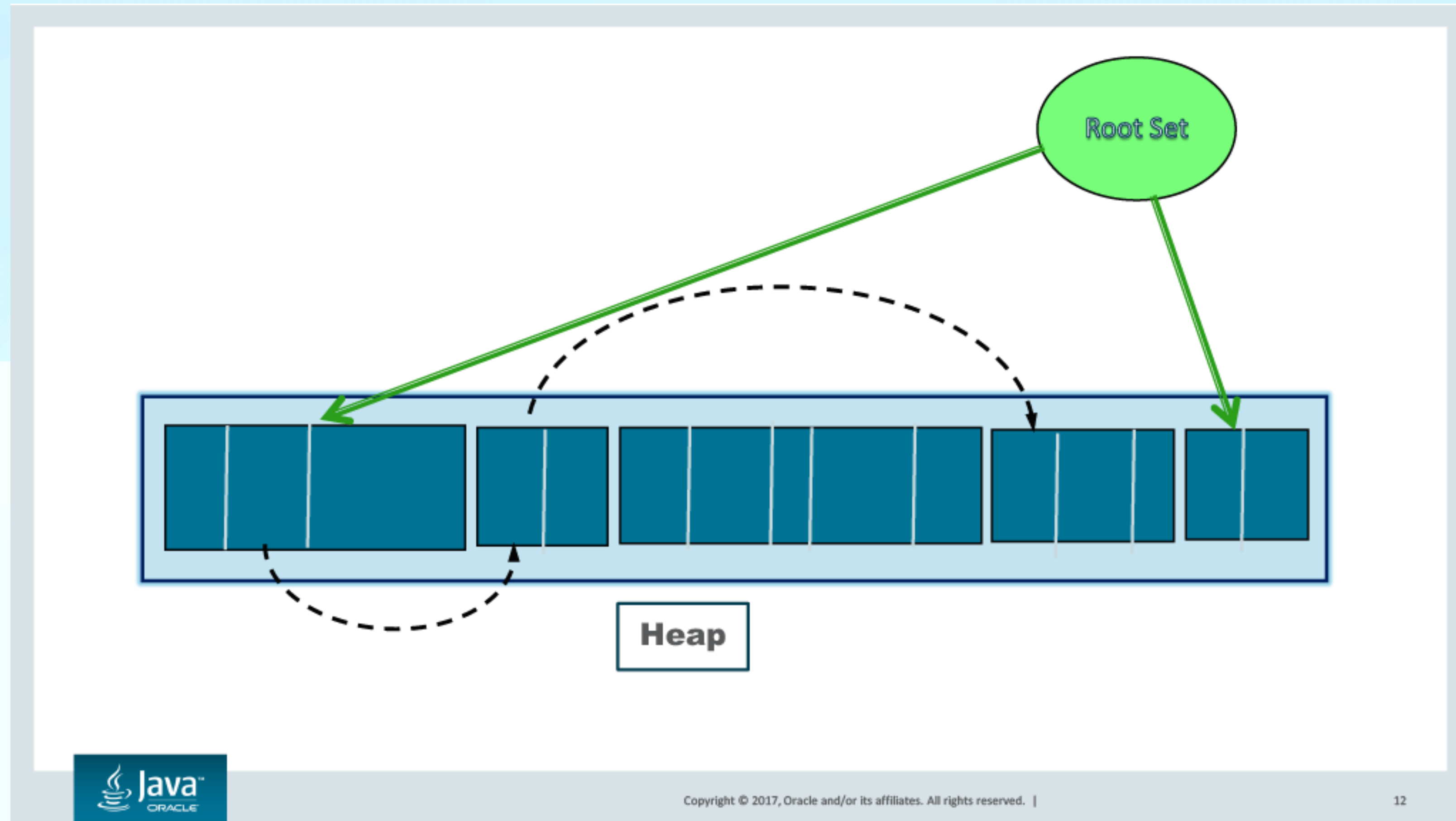
GC Tracing Step



https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf

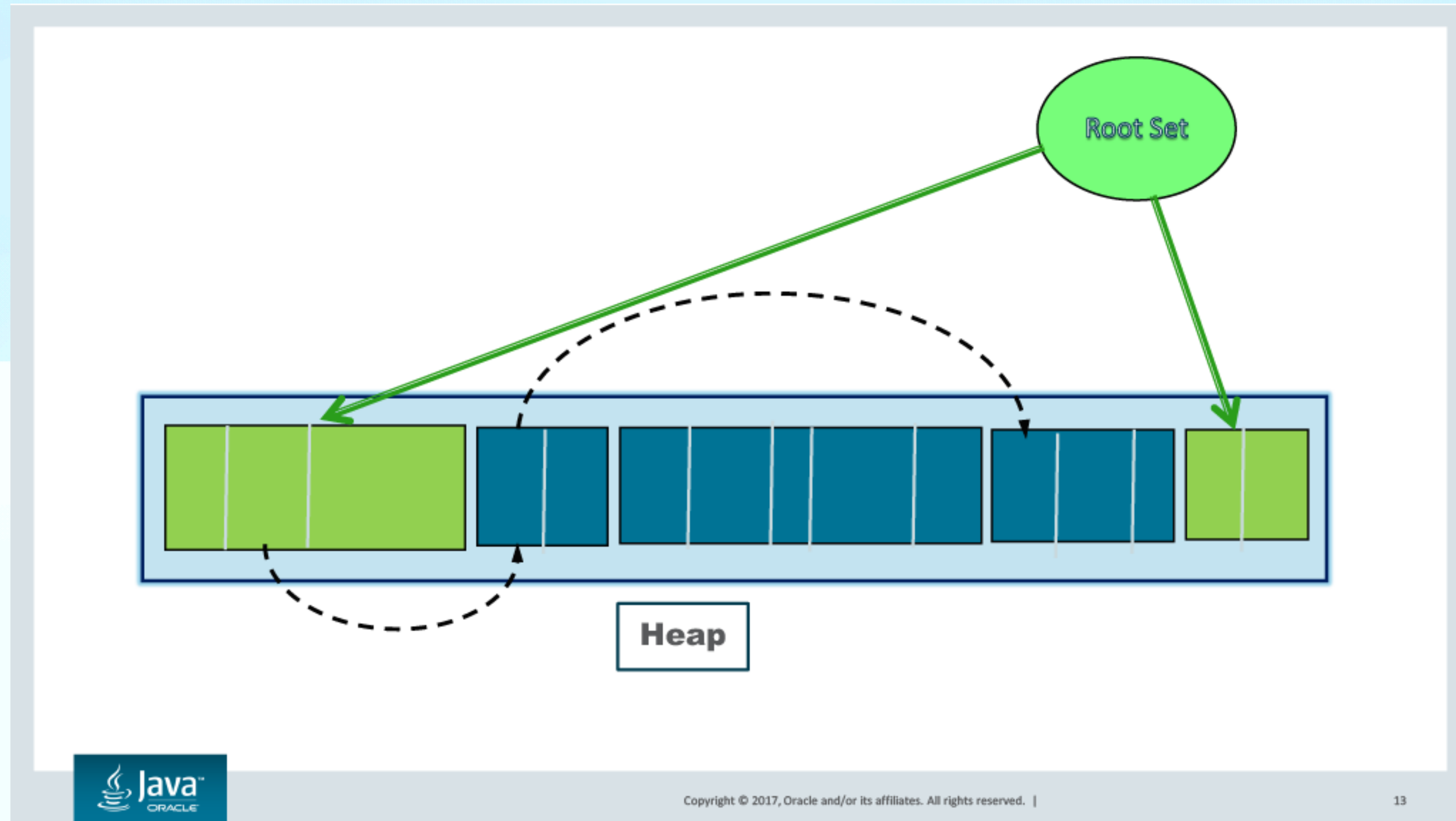
가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC Tracing Step



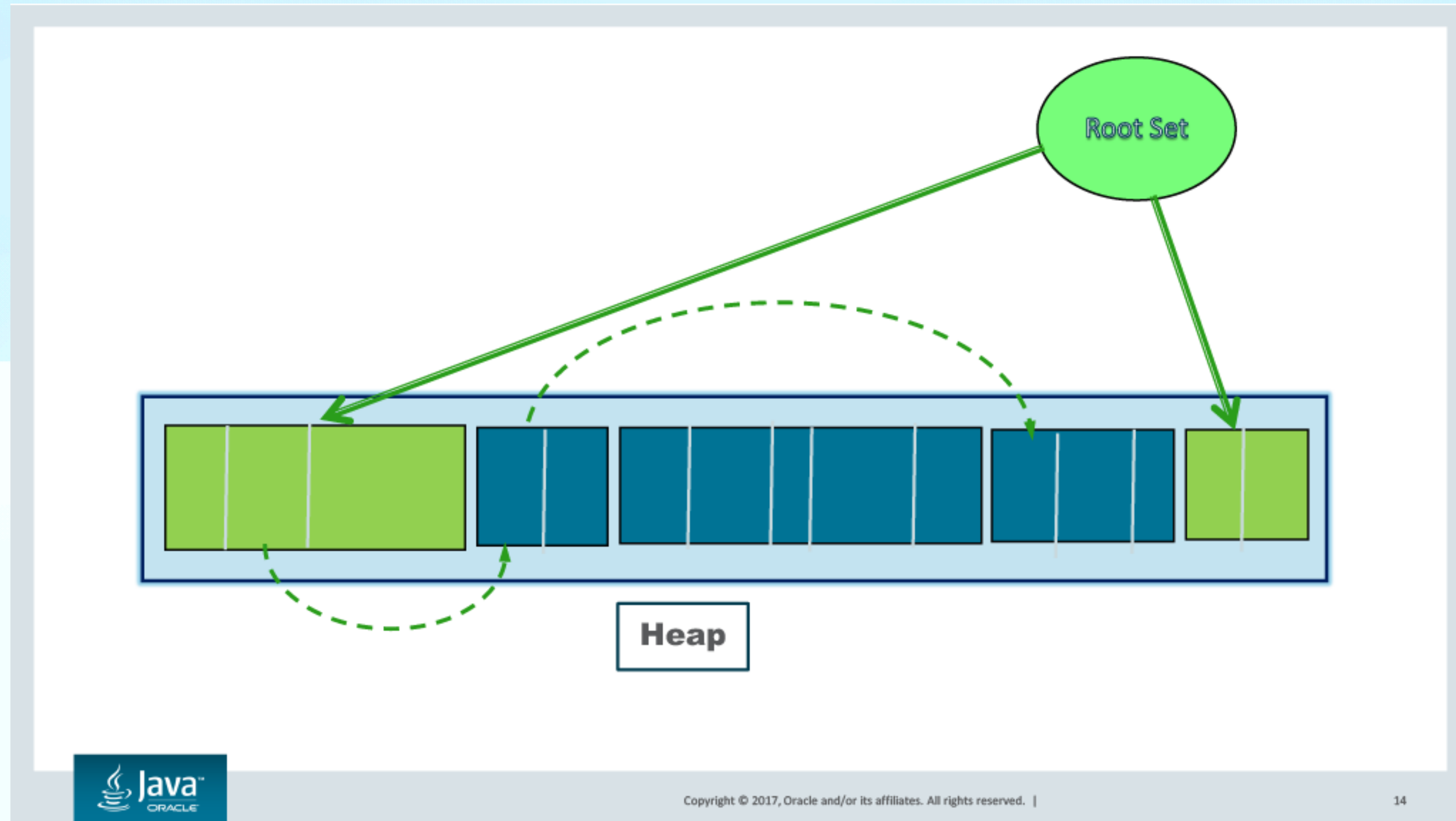
가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC Tracing Step



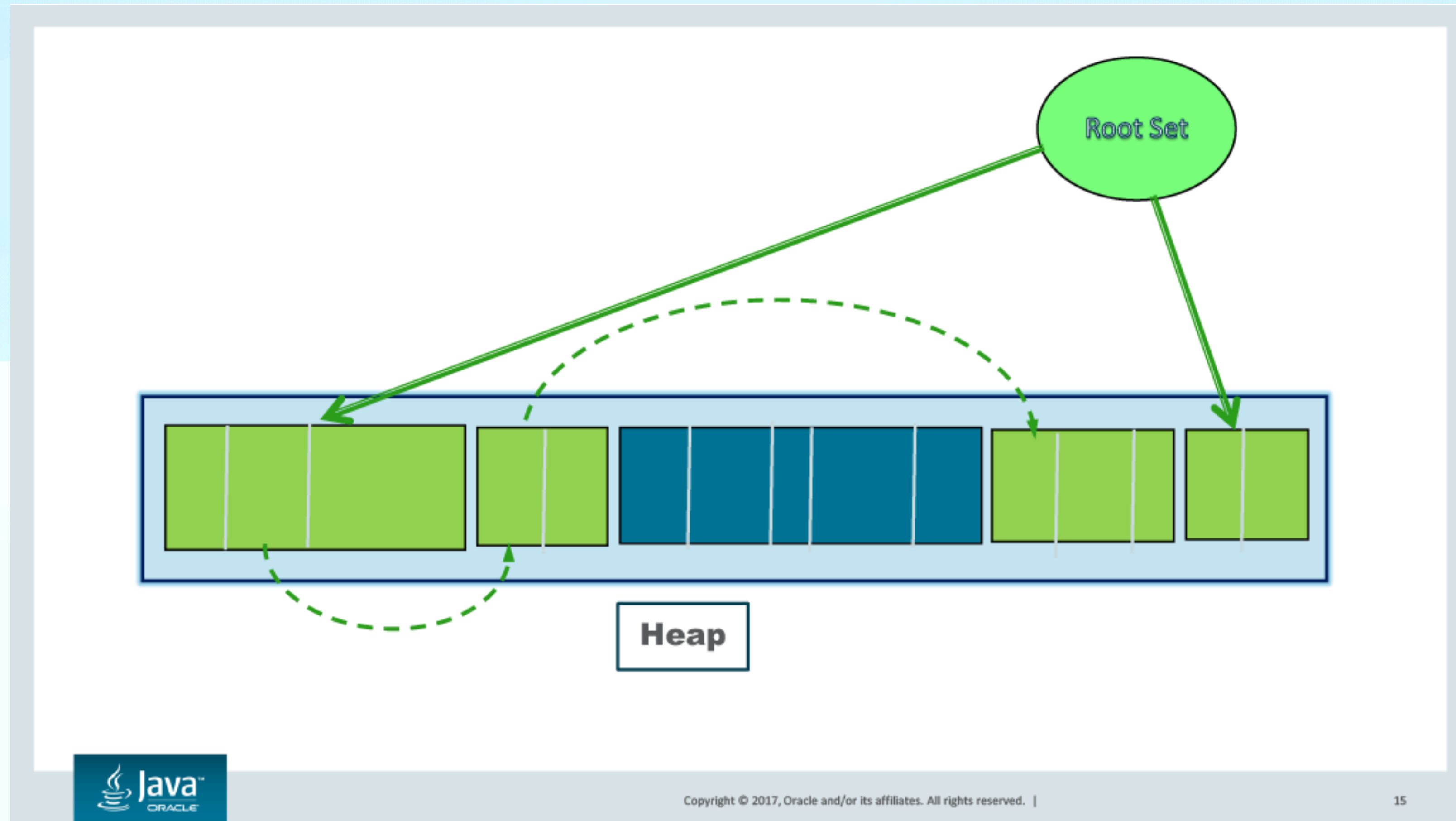
가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC Tracing Step



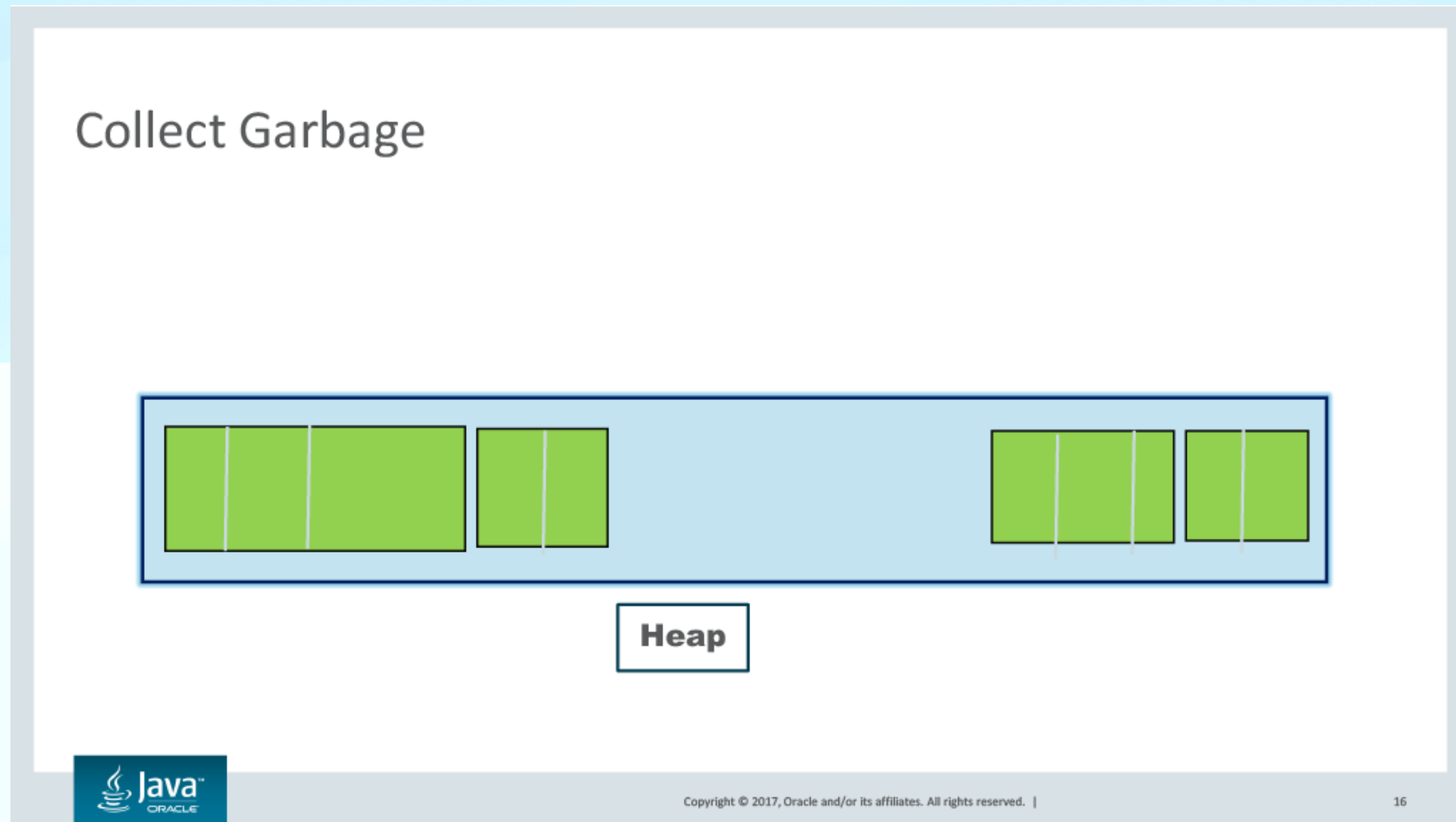
가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC Tracing Step



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

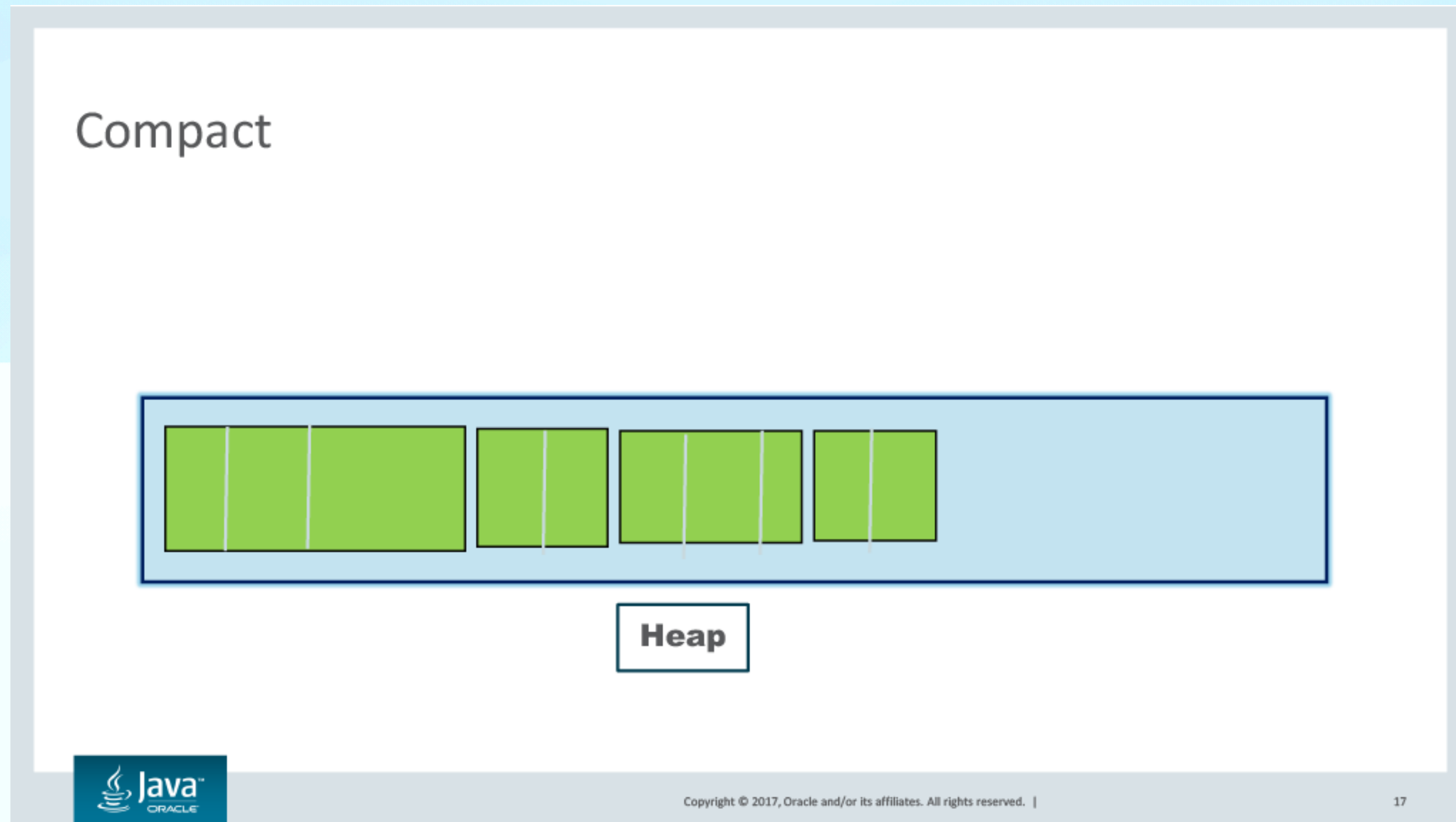
GC Tracing Step



https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

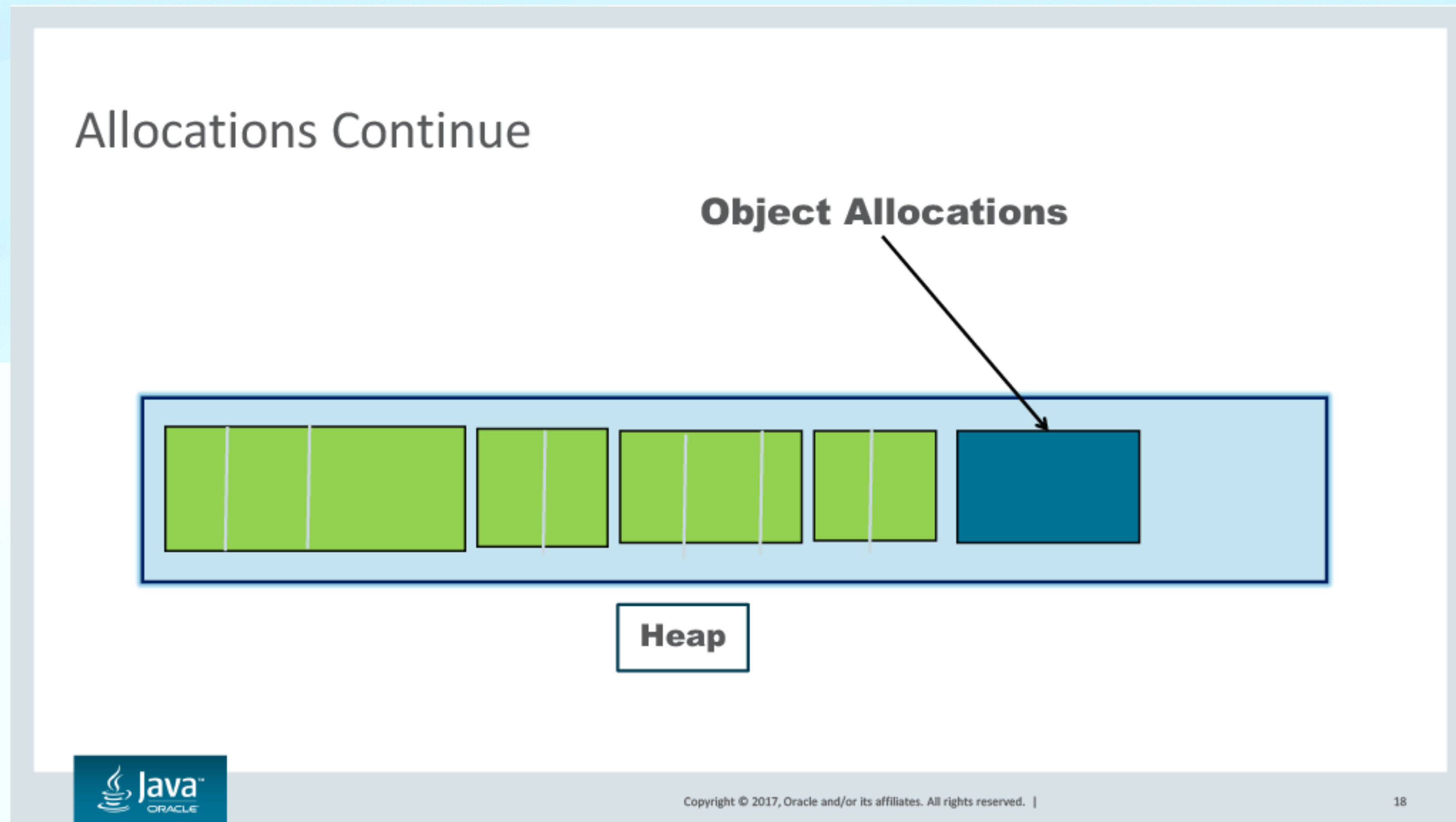
GC Tracing Step



https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf

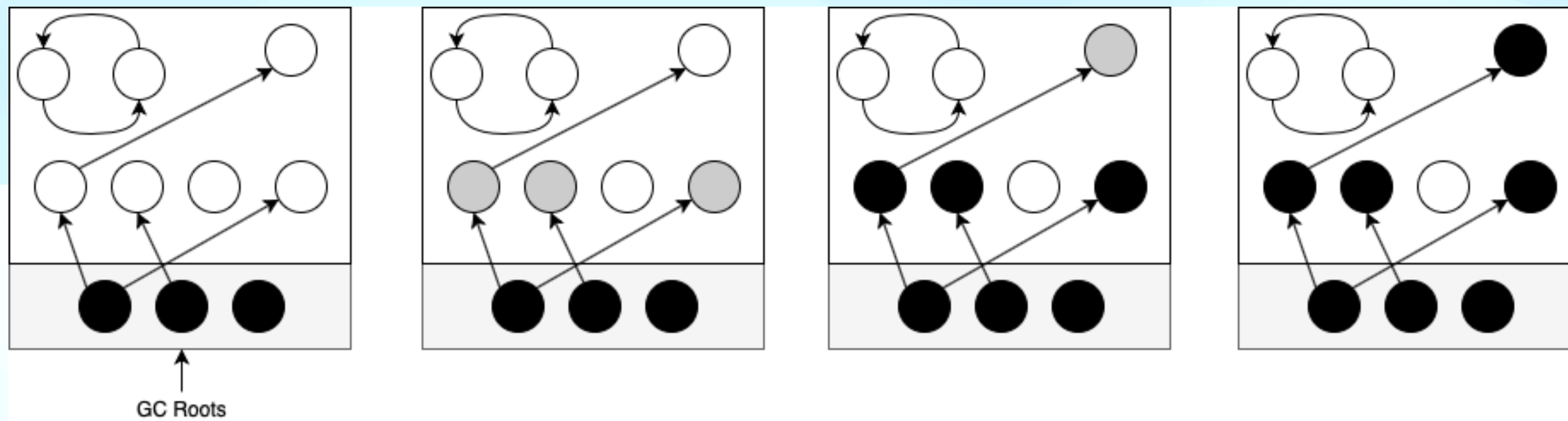
가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC Tracing Step



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC Tracing



<https://www.baeldung.com/java-gc-cyclic-references#tracing-gcs>

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

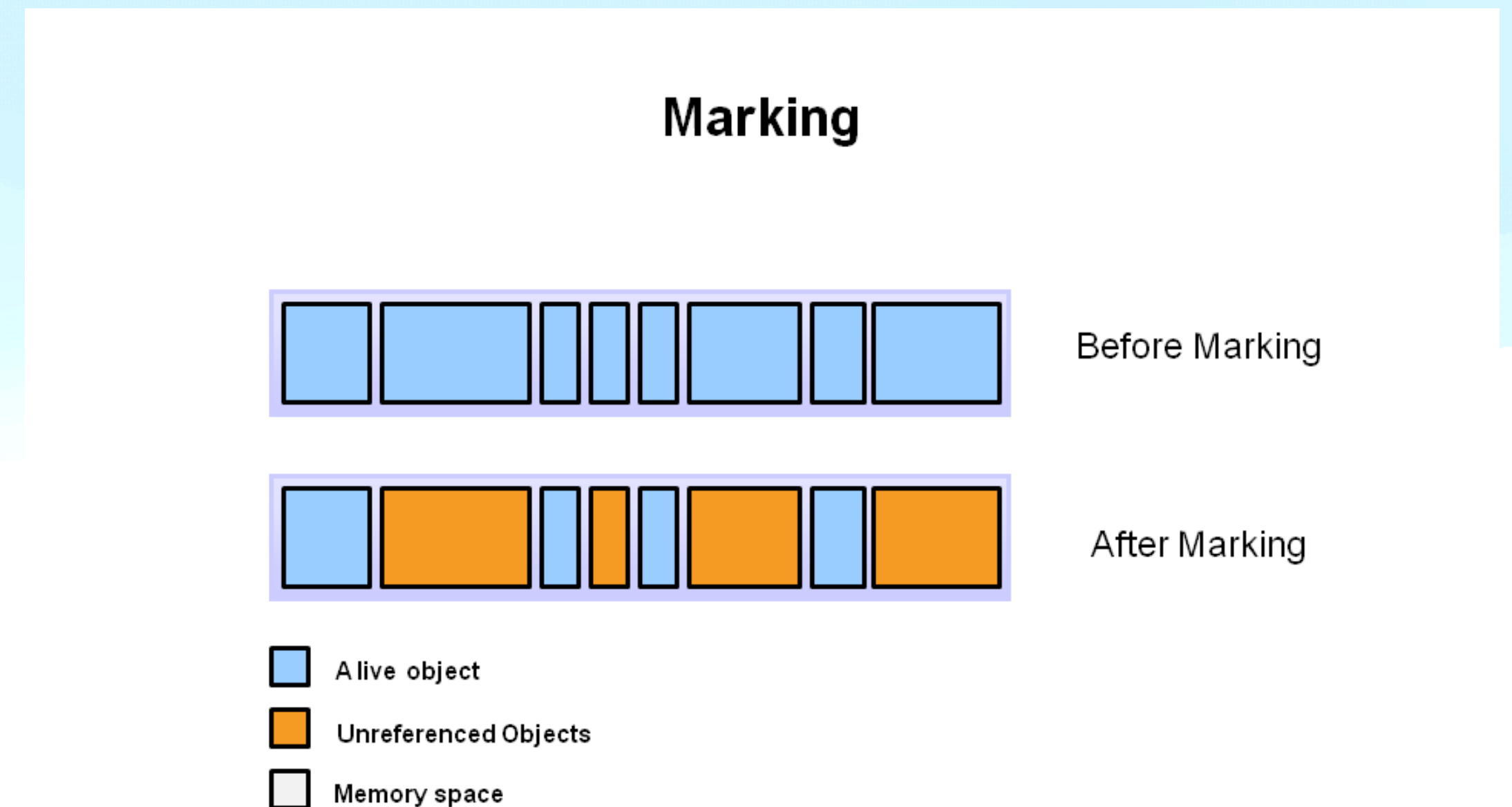
Mark, Sweep, Compact

- Mark
 - 참조 객체와 참조되지 않는 객체를 식별하는 프로세스
- Sweep
 - 마킹된(미사용) 객체를 제거하는 프로세스
- Compact
 - 위 작업들로 인해 생긴 메모리 단편화를 없애는 프로세스

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC의 기본적인 동작 방식 - Marking

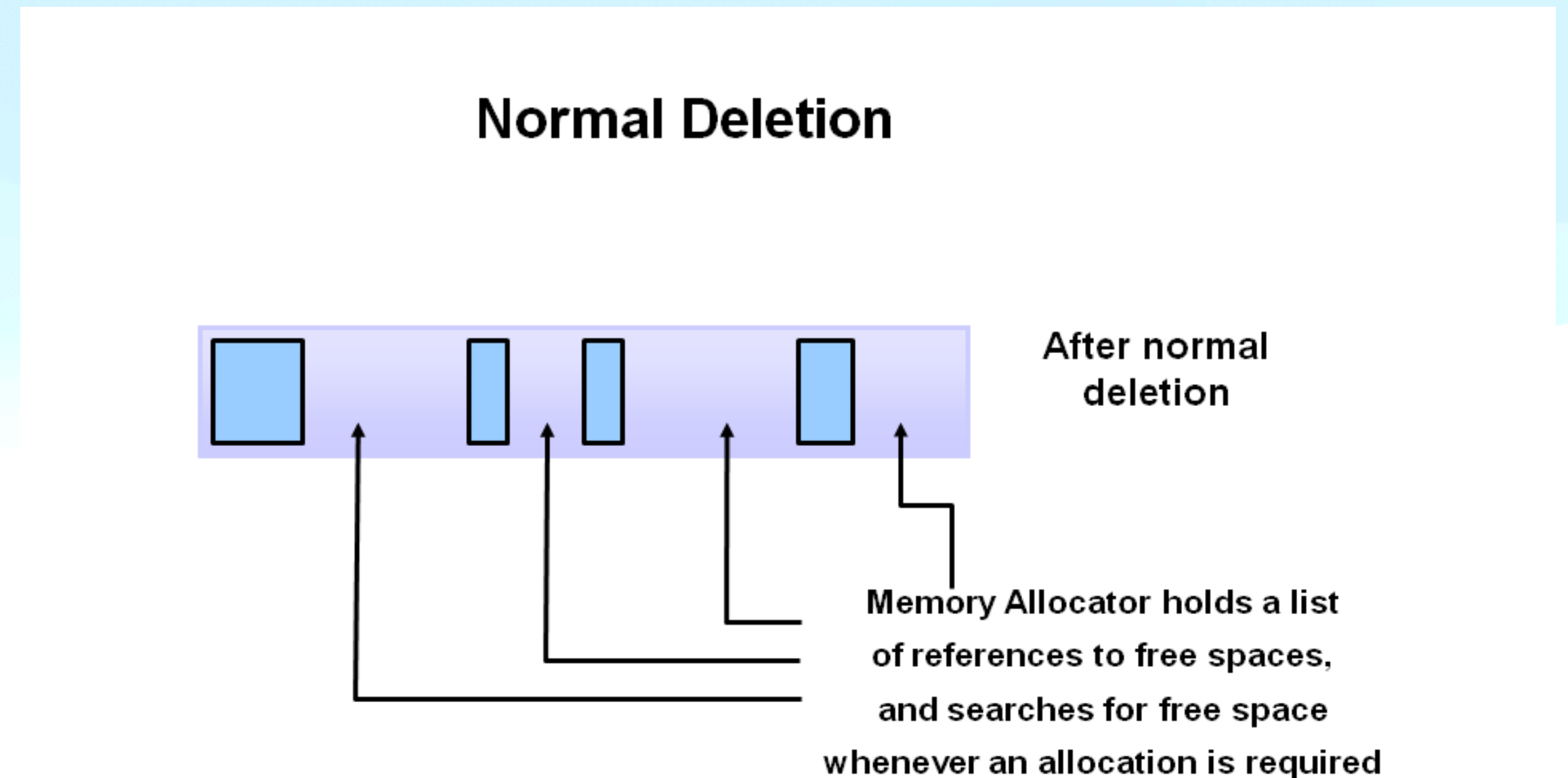
- 사용중인 메모리와 사용하지 않는 메모리 식별
- 모든 객체를 스캔하기 때문에 시간이 많이 걸리는 프로세스가 될 수 있음



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC의 기본적인 동작 방식 - Normal Deletion

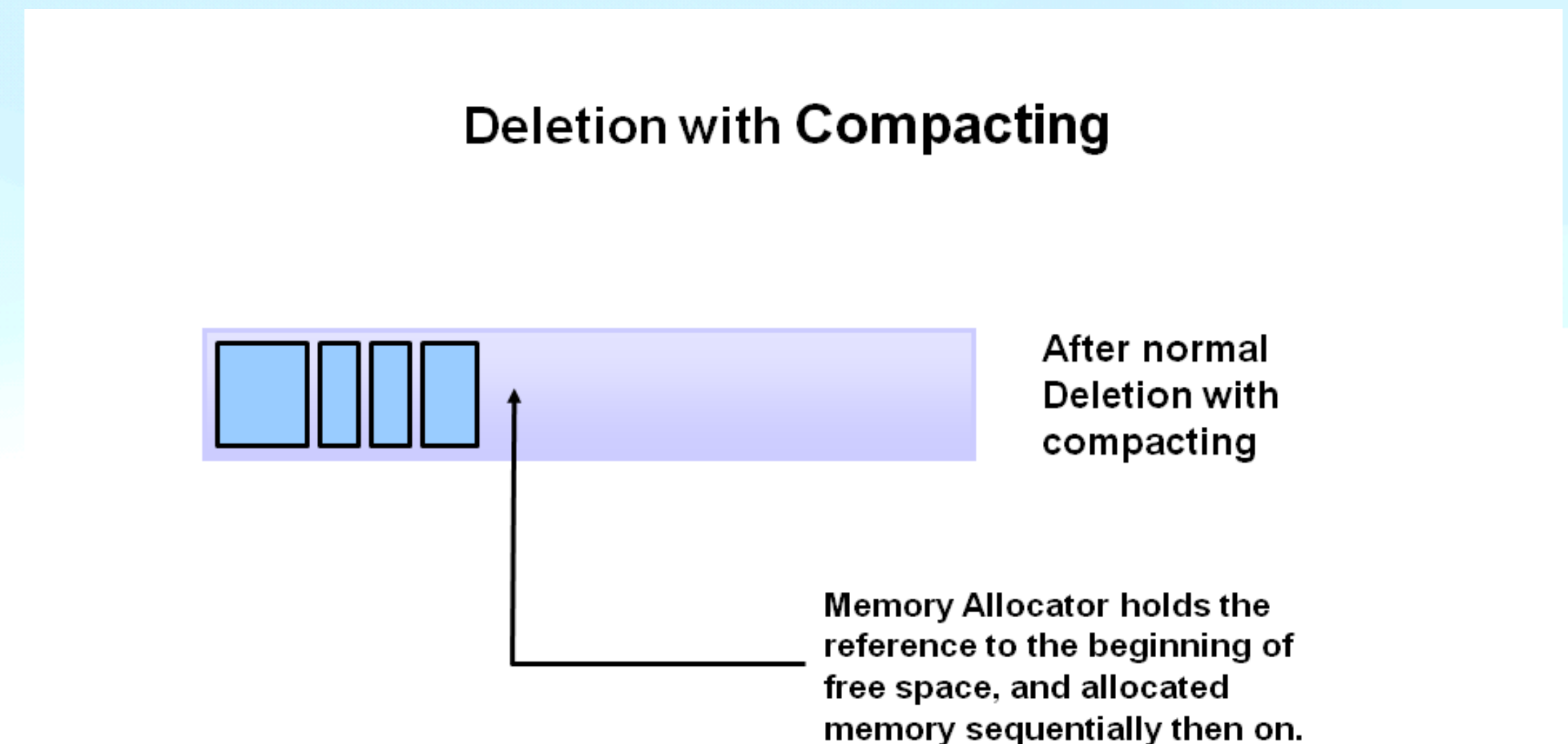
- 일반 삭제는 참조되지 않는 객체를 제거하며 참조된 객체와 여유 공간에 포인터는 제외
- 새 객체를 할당할 여유 공간 블록에 대한 참조 보유



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC의 기본적인 동작 방식 - Compacting

- 성능 향상을 위해 객체 제거 외에도 메모리 압축(compact) 작업을 수행하기도 함
- 남아있는 참조 객체를 이동하여 메모리 압축 작업을 수행하면 새 메모리 할당이 훨씬 빨라짐



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

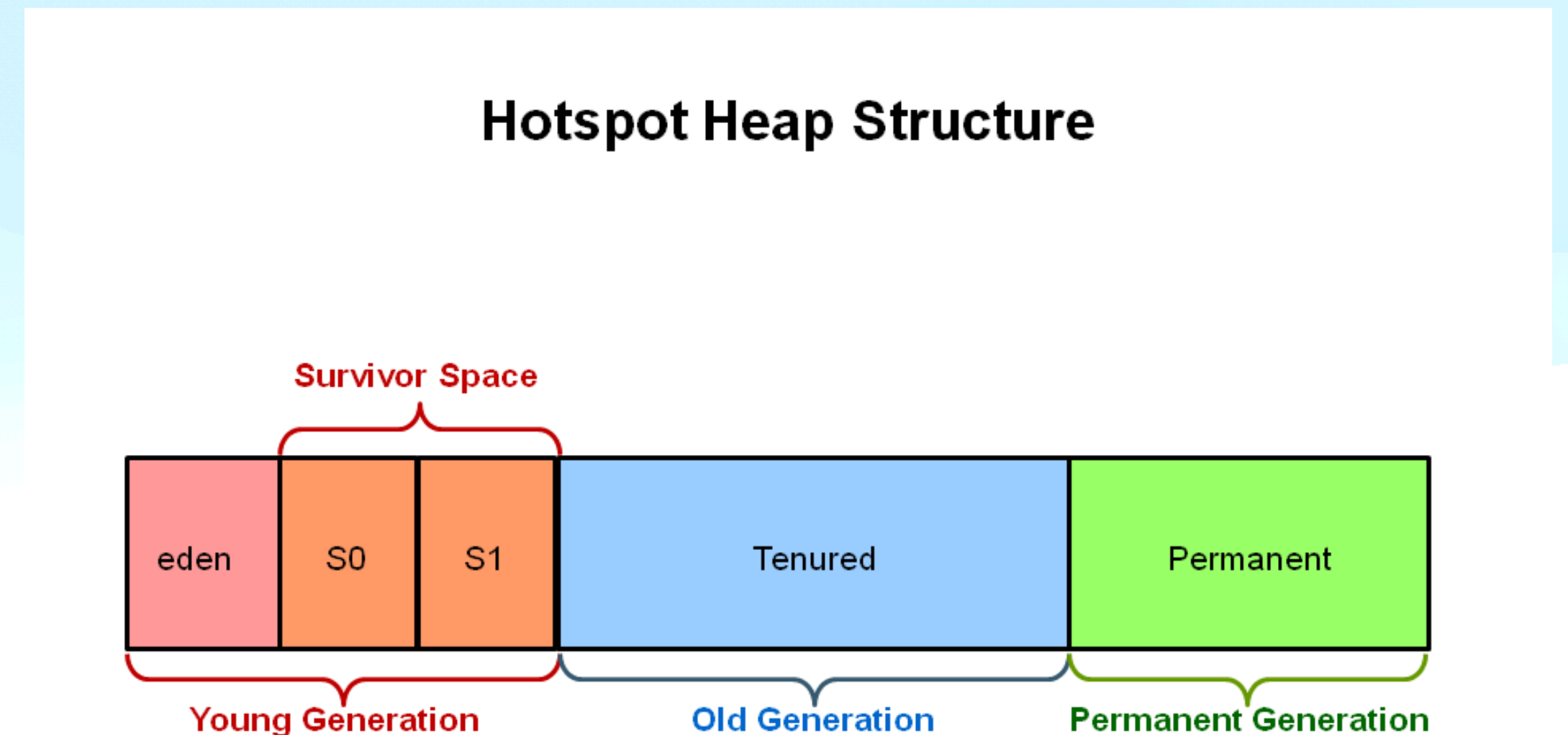
GC의 기본적인 동작 방식의 단점

- JVM의 모든 객체를 스캔해 마킹하고 메모리를 압축하는 작업은 비효율적
- 시간이 경과함에 따라 점점 더 많은 객체가 할당되어 처리해야 할 작업이 많아짐
- 따라서 GC 작업 시간이 점점 더 증가함
- 이를 보완하기 위해 Generations 방식 도입

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

JVM Generations

- 객체 할당 작업에서 학습한 정보를 성능 향상에 활용
- 힙은 더 작은 파트(Generation)로 나뉨
- Generation 종류
 - Young Generation
 - Old(Tenured) Generation
 - Permanent Generation (현재 Metaspace)



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

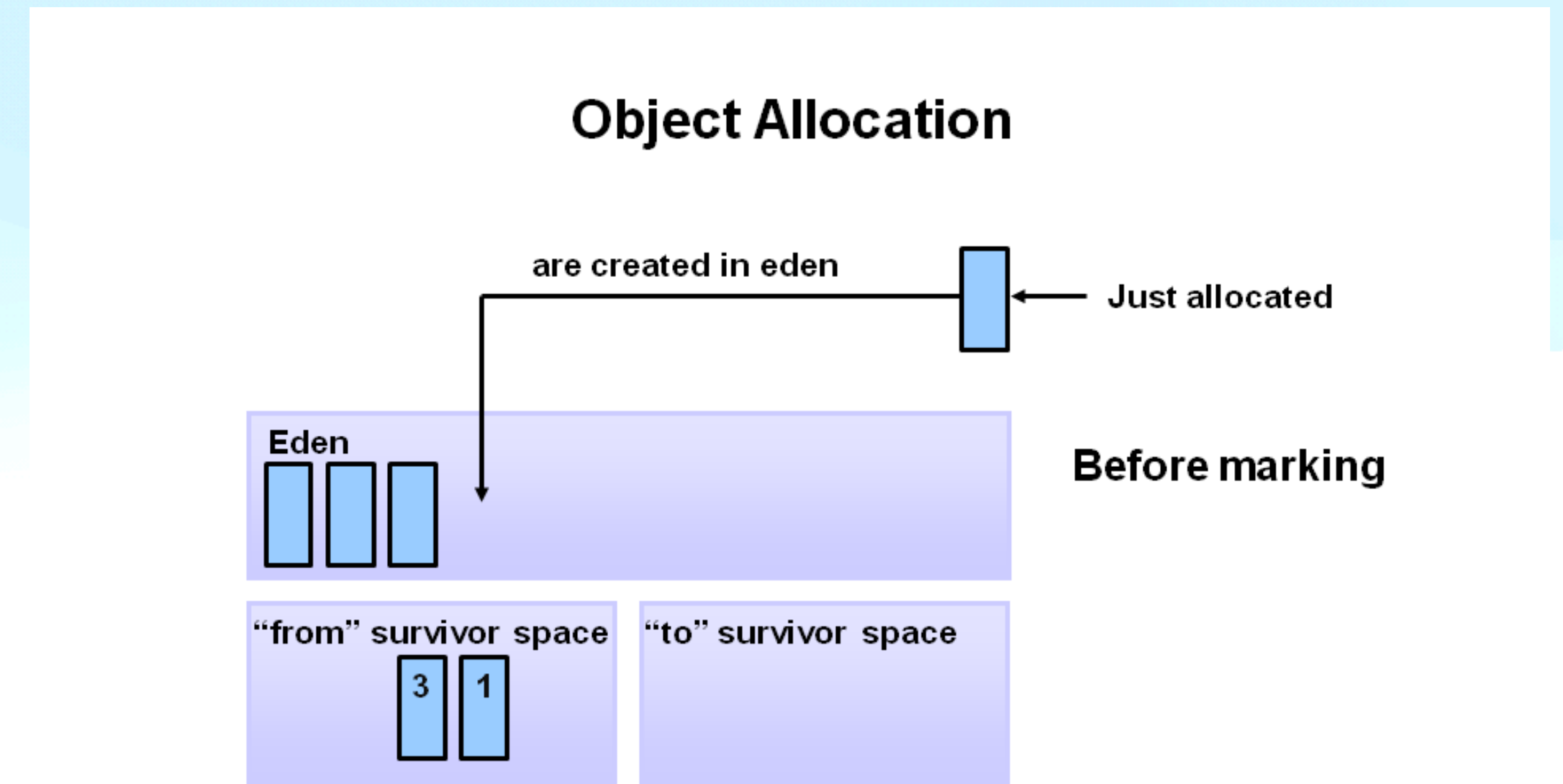
JVM Generations

- Young Generation
모든 새 객체가 할당되는 영역이며 마이너(Minor) GC가 발생
 - 1개의 Eden과 2개의 Survivors로 나뉨
 - 마이너 GC는 신생 객체의 사용 빈도를 가정해 최적화할 수 있음
 - 오래 남은 객체는 Old Generation으로 이동
- Stop-the-world(STW) 이벤트
해당 작업이 완료될 때까지 모든 앱 스레드가 중지되는 행위
 - 마이너 GC는 항상 STW 이벤트
- Old Generation
오랫동안 참조(사용)된 객체가 저장되는 영역
GC에 의해 특정 기준으로 임계값이 설정되고 이를 넘어서면 Young Generation으로부터 이동됨
 - 최종적으로 이 영역도 발생하는 GC 작업을 메이저(Major) GC라고 함
 - 메이저 GC는 간헐적으로 STW 이벤트이며
모든 객체를 확인해야 하기 때문에 종종 느리고 이전 Generation의 영향을 받음
 - 따라서 반응형(리액티브) 앱은 메이저 GC의 빈도를 최소화해야 함
 - 반응형 앱은 빠른 반응, 효율적인 리소스 사용, 탄력성 등의 기준들을 충족하는 앱
- Permanent Generation
클래스와 메서드 등의 메타데이터가 저장되는 영역 (Full GC에 포함됨)
 - 런타임에 앱에서 사용 중인 클래스를 기반으로 저장됨 (SE 라이브러리 등도 포함될 수 있음)
 - 새로운 데이터를 저장할 공간이 필요한 경우 더이상 필요하지 않게된 데이터를 수집/언로딩 할 수 있음

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Generational GC Process

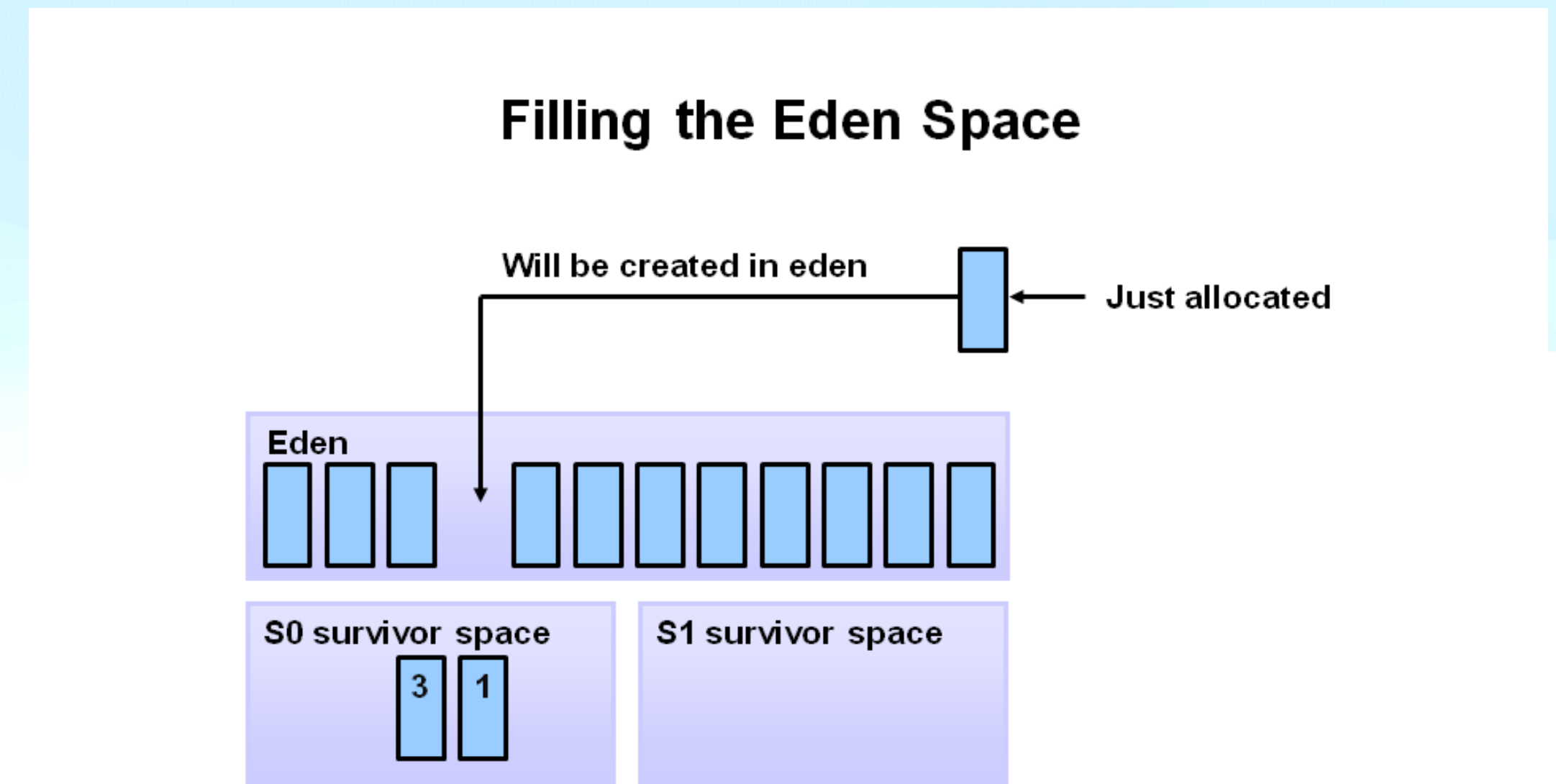
- 새로 생성된 모든 객체가 Eden 영역에 할당됨
- `from`, `to` survivor 영역 모두 비워져 있음



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Generational GC Process

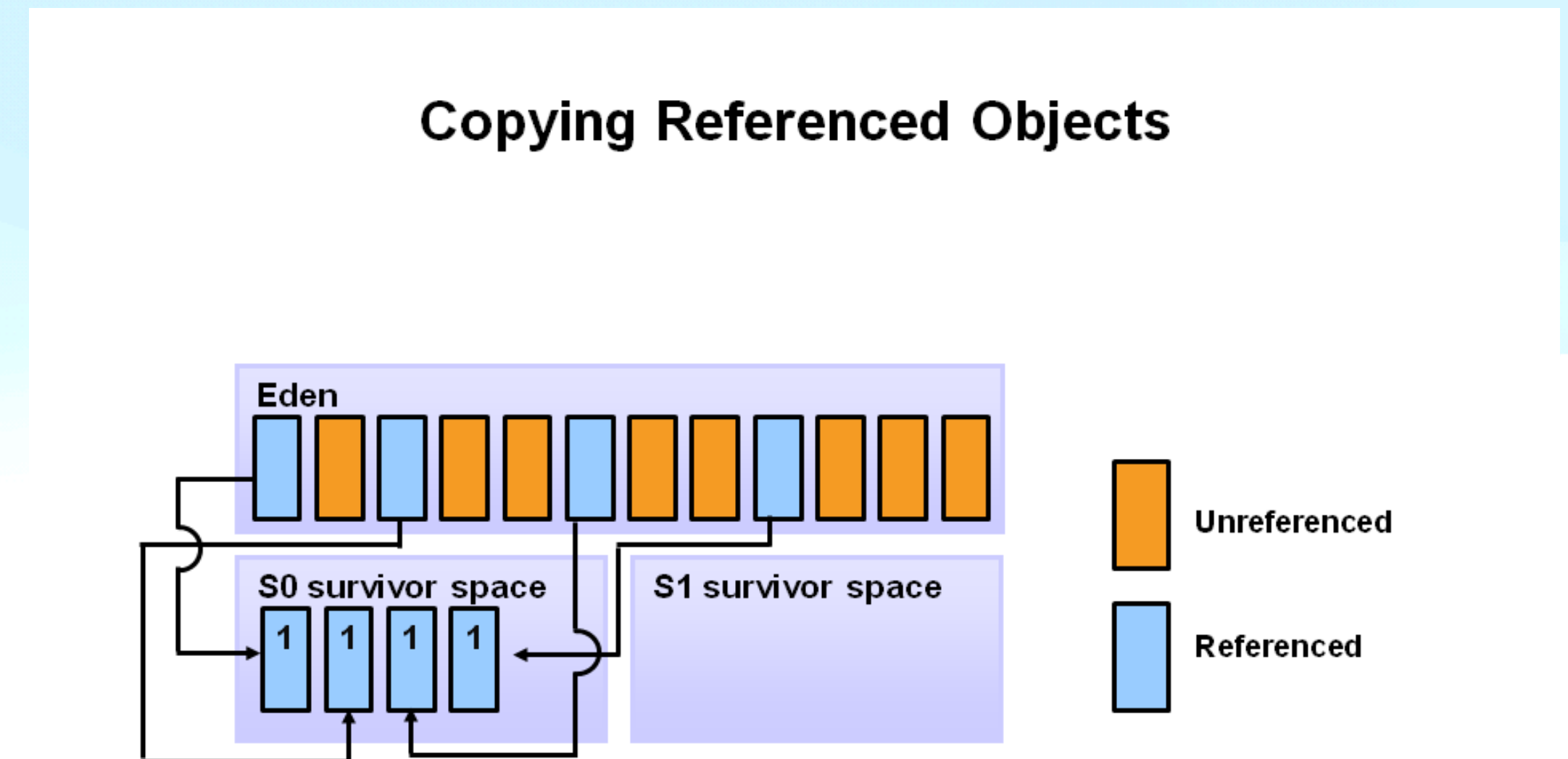
- Eden 영역이 가득차게 되면
마이너 GC가 Eden 영역의 객체들을 수집함



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Generational GC Process

- 참조 객체들을 첫번째 survivor 영역으로 이동시킴
- 미참조 객체들을 Eden 영역이 비워질 때 제거함

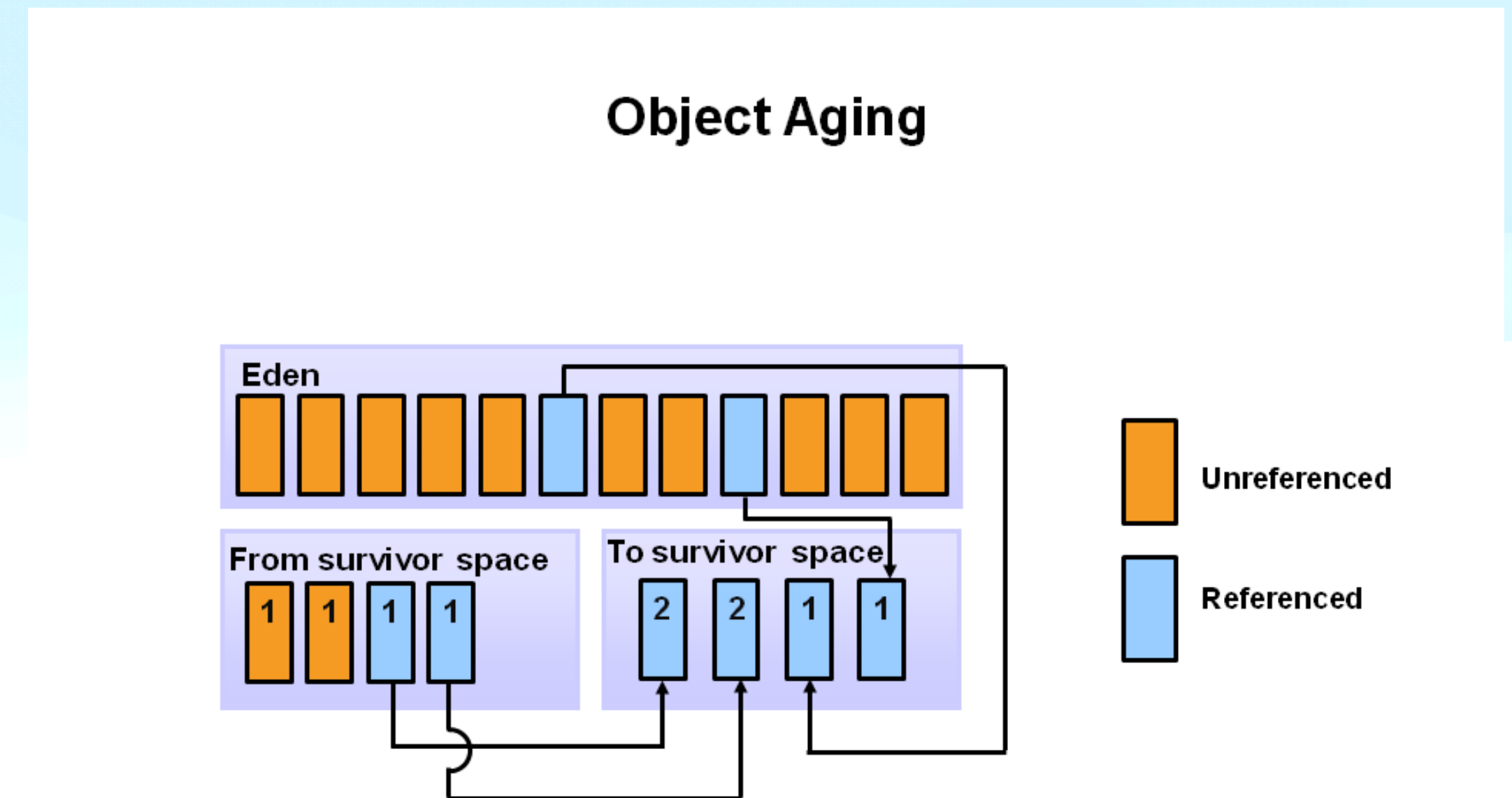


<https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Generational GC Process

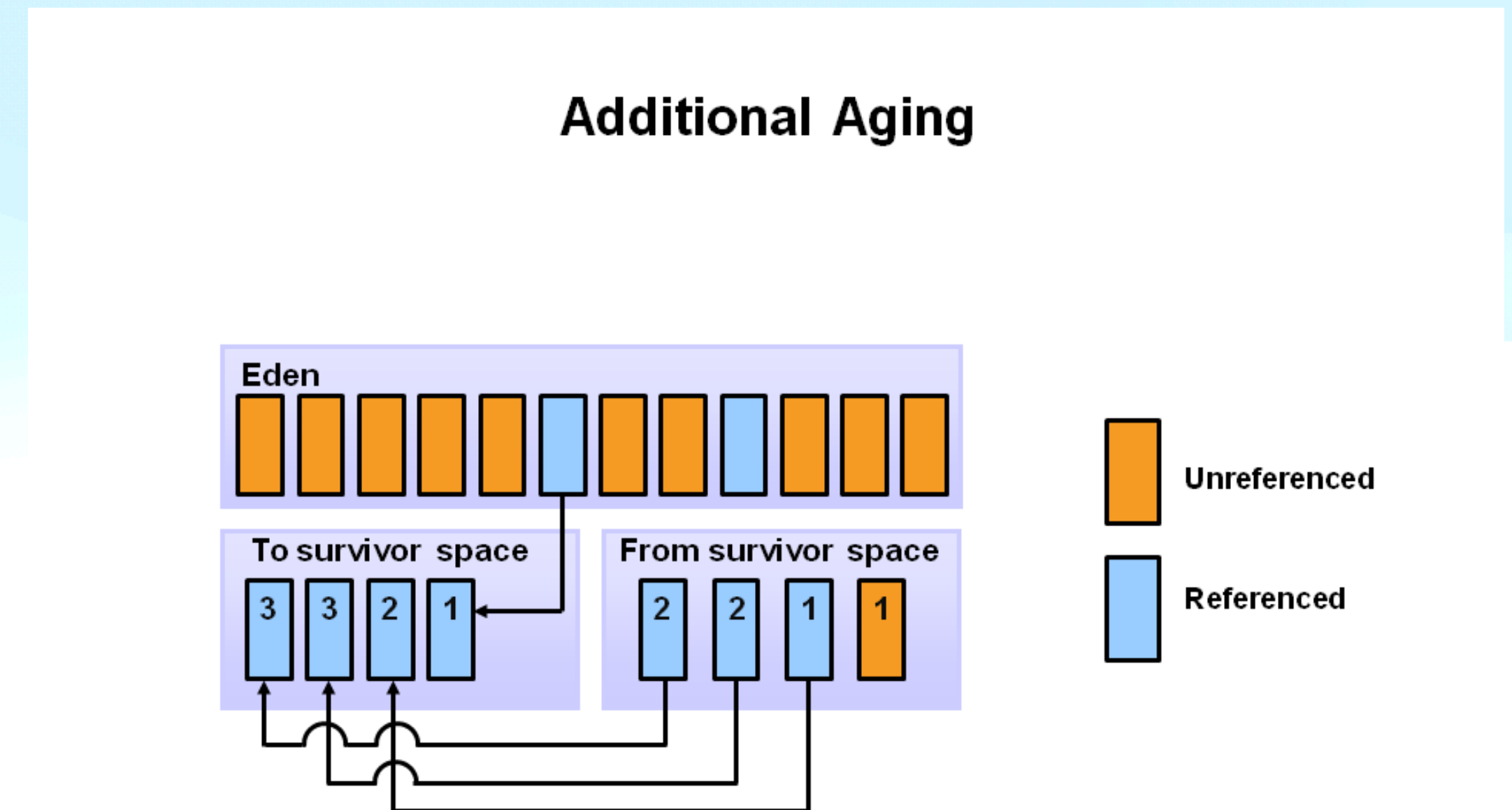
- 다음 마이너 GC 처리 시 동일한 절차를 반복
참조 객체는 survivor 영역으로 이동되며
미참조 객체는 제거됨
- 하지만 이번에는 두번째 survivor 영역으로 이동
- 이어서 첫번째 survivor 영역에 있던 객체들도
Age가 증가하며 두번째 survivor 영역으로 이동
- 위 작업들이 완료되면 `Eden`과 `From` 영역이
모두 비워짐



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Generational GC Process

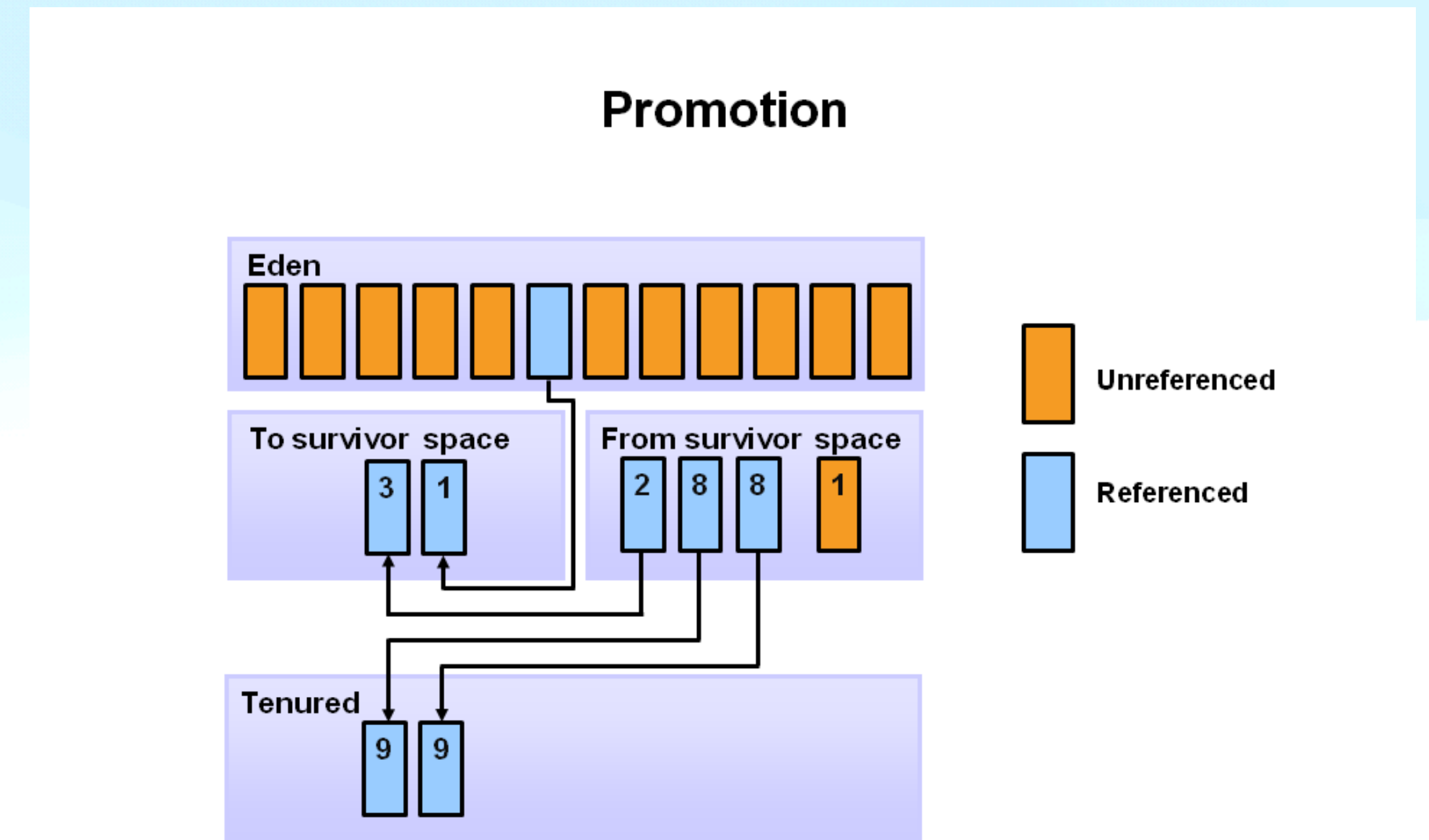
- 다음 마이너 GC 또한 앞선 작업 과정을 반복
- 하지만 이 과정을 반복할 때 마다 교대로 survivor 영역을 바꿔가며 객체를 이동시킴



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Generational GC Process

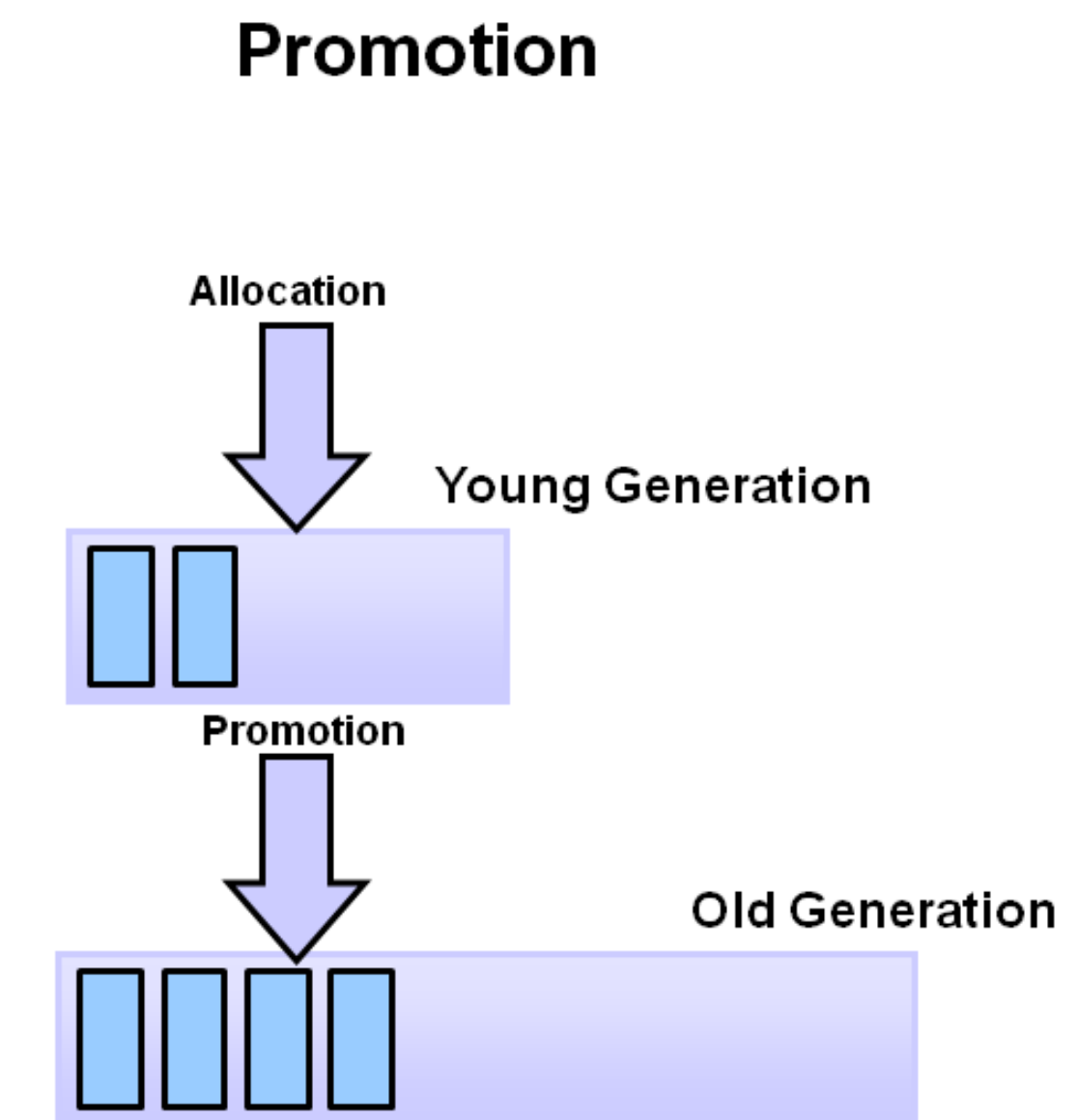
- 반복된 마이너 GC 이후 오래된 객체들이 설정된 임계값을 넘기면 survivor에서 Old(Tenured)으로 이동시킴



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Generational GC Process

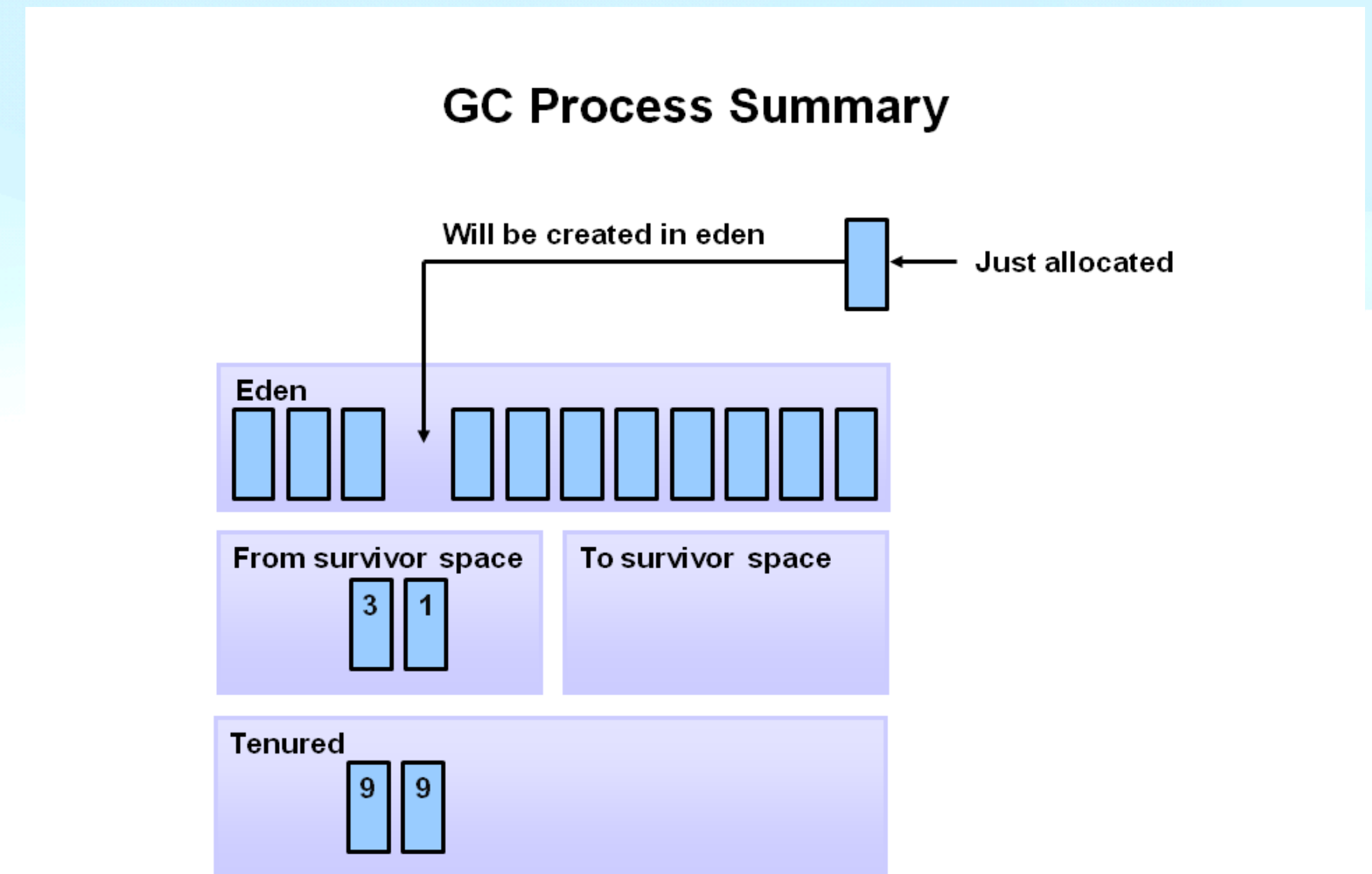
- 이 또한 마이너 GC가 반복되며 오래 참조된 객체들은 계속해서 Old 영역으로 이동하게 됨



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Generational GC Process

- 앞선 많은 과정이 Young 영역에서 발생
- 최종적으로 Old 영역에서 메이저 GC가 일어남



가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC 장단점

- 장점

- GC에 의해 메모리가 자동으로 관리되어 직접 메모리를 할당/해제할 필요가 없음
- 댕글링 포인터(Dangling Pointer) 핸들링으로 인한 오버헤드가 발생하지 않음
 - 유효하지 않는 객체/타입을 가리키는 포인터
- 자동 메모리 누수 관리
 - 완벽하진 않으나 상당 부분 처리 가능

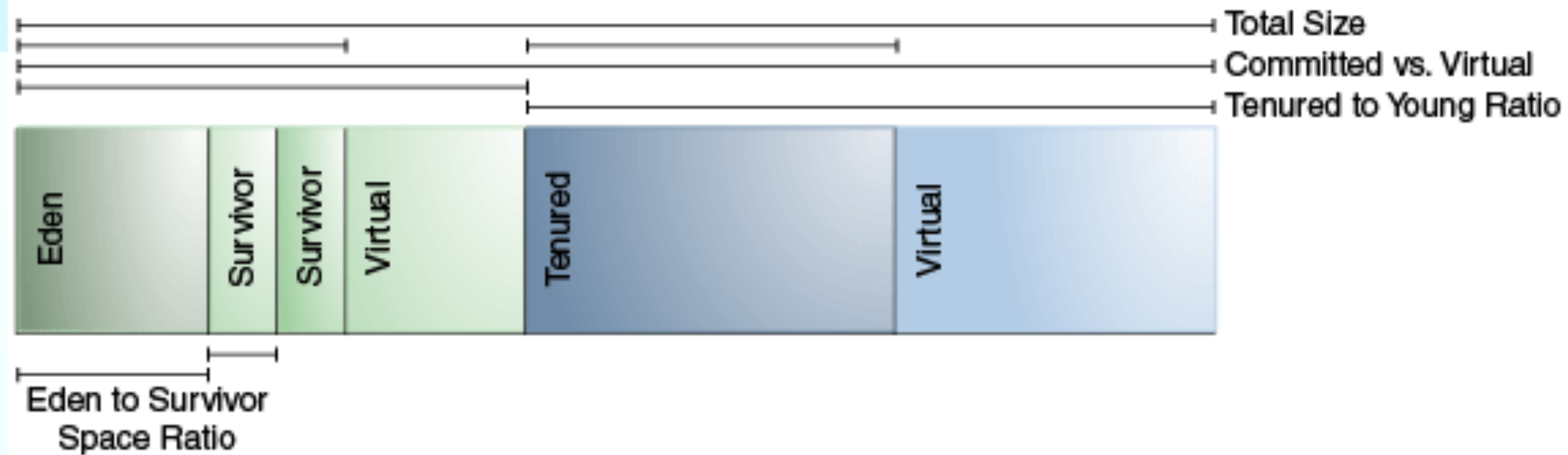
- 단점

- 모든 객체에 대한 생성/삭제를 추적하므로 원래의 앱 사용 리소스(CPU)보다 더 많은 성능이 필요함
대용량 메모리가 필요한 경우 성능에 영향을 줄 수도 있음
- 프로그래머가 사용하지 않는 객체를 해제하는 전용 CPU 스케줄링을 직접 제어할 수 없음
- 일부 GC는 기능적으로 완벽하지 않아 런타임에 중지될 가능성도 있음
- 메모리 관리 자동화는 수동으로 직접 메모리를 관리(할당/해제)하는 것보다 비효율적

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

Heap

- 힙 영역을 아래와 같이 나눠서 관리
- [참고] Virtual 영역은 할당된 최대 메모리를 표현하기 위해 사용된 것으로 보여짐



<https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/sizing.html>

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

GC Roots

- GC 루트는 GC를 위한 특별한 객체로 GC 프로세스의 시작점
GC 루트로부터 직/간접적으로 참조되는 객체는 GC 대상에서 제외됨
- 구현된 Java GC 알고리즘의 대부분(Hotspot VM)은 GC 루트로부터 해당 참조 객체들을 추적하는 형태
 - GC 루트로부터 객체 그래프를 순회, 참조되고 있는 활성화 객체를 식별
 - 활성화 상태라고 인식(마킹)된 객체는 GC 수집(제거) 대상에서 제외됨

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

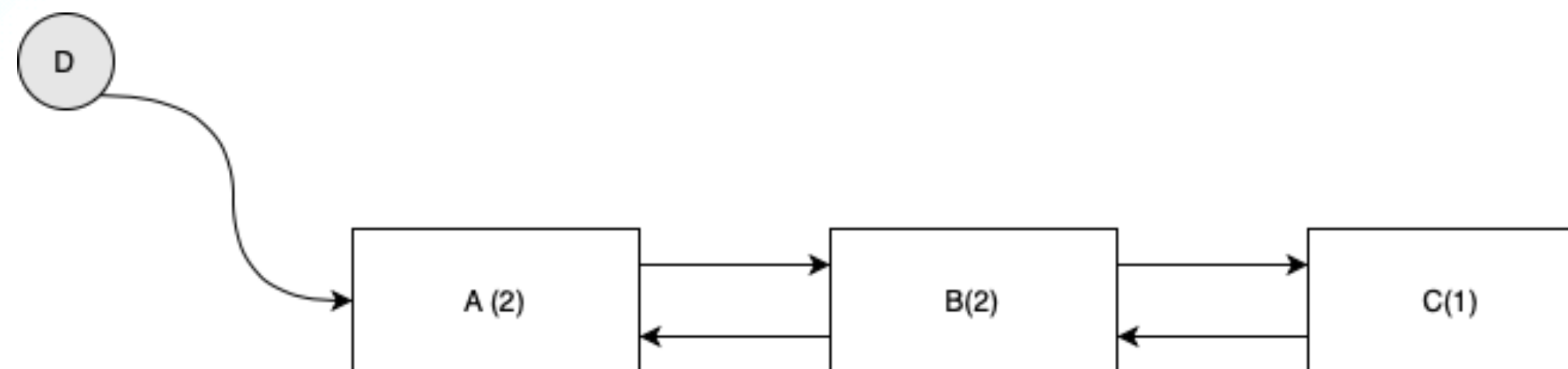
GC Root Types

- 클래스
클래스 로더에 의해 로딩된 클래스, 해당 클래스의 스택 필드의 참조도 포함됨
- JVM 스택의 LVA(Local Variable Array)
LVA의 지역 변수, 매개 변수
- 활성화 상태의 스레드
- JNI 참조
JNI 호출을 위해 생성된 네이티브 코드 Java 객체
지역 변수, 매개 변수, 전역 JNI 참조 등
- 동기화를 위해 모니터로 사용되는 객체
예를 들어 `synchronized` 블록에서 사용(참조)되는 객체
- GC 루트 용도로 JVM 정의/구현한 GC 처리되지 않는 객체
예를 들어 Exception 클래스, system(custom) 클래스 로더 등

가비지 컬렉션의 정의와 가비지 컬렉터가 처리하는 Heap 영역

* GC Reference Counting

- 참조 객체의 참조 횟수를 세어 GC를 처리하는 방법
- 구현이 간단하고 횟수가 0이 되었을 때 즉시 제거할 수 있다는 장점들이 있음
 - 즉 GC가 처리되어야 할 때마다 발생하는 STW 등을 피할 수 있음
- 하지만 순환 참조 문제를 해결하기 어렵고 카운팅을 위한 추가 메모리가 필요
또한 멀티 스레드 환경에서는 카운팅 작업에 대한 동기화 때문에 성능이 저하될 수 있음
- 참고로 JVM의 GC는 현재(JDK 17) 참조 카운팅을 기반으로 하는 알고리즘은 없음



<https://www.baeldung.com/java-gc-cyclic-references#tracing-gcs>

[2] Heap 영역을 제외한 GC 처리 영역

Heap 영역을 제외한 GC 처리 영역

메모리 관리가 필요한 영역

- Heap 외에도 별도의 메모리 관리가 이뤄지는 네이티브 메모리 영역
이 장에서 설명하는 GC가 아닌 OS 또는 JVM이 메모리 관리를 수행하는 영역
- JMM은 Heap 외에도 네이티브 메모리 영역도 관리
- Metaspace (Permanent Generation을 대체)
- 기타
 - CodeCache
 - Native Memory

Heap 영역을 제외한 GC 처리 영역

Metaspace (Permanent)

- JDK 8에서 도입된 영역으로, Hotspot VM의 네이티브 메모리 관리자 (off-heap)
- 클래스가 로딩될 때 할당되는 메타데이터에 대한 메모리 관리를 담당
자동으로 크기가 확장됨
- 일반적으로 해당 클래스의 메타데이터는 로딩을 담당하는 클래스 로더의 생명주기와 관련이 있음
만약에 해당 클래스 로더가 GC에 의해 제거되면 관련된 클래스들의 대량 메타데이터가 제거(소실)됨
- GC는 Metaspace 사용량이 최대치에 도달하면 참조되지 않는 클래스의 수집을 처리(트리거)함
- 해당 영역에 최종 커밋 시 주의사항
 - MaxMetaspaceSize
 - 커밋 가능한 메모리 크기의 최대 상한 값
 - GC Threshold
 - GC 임계값을 Metaspace의 영역이 커지는 것에 영향을 줌 (클래스 언로딩 등)

Heap 영역을 제외한 GC 처리 영역

* Permanent

- 힙과 분리된 특수한 힙 영역으로 JDK 8부터 Metaspace로 대체된 영역
- JVM은 Permanent를 통해 로딩된 클래스 메타데이터를 추적(참조)
- 모든 스택틱 메서드와 스택틱 필드에 대한 참조, 원시 타입 변수 등을 포함
또한 바이트코드에 대한 데이터와 이름, JIT 컴파일에 대한 정보를 포함

Heap 영역을 제외한 GC 처리 영역

* Code Cache

- JIT 컴파일러가 Java 바이트코드를 컴파일한 네이티브 코드를 저장하는 메모리 영역
JIT 컴파일러가 가장 많이 사용하는 영역
- JVM의 `Code Cache sweeper` 스레드가 메모리 관리
 - 코드 캐시 스위퍼가 더이상 사용되지 않는 코드 세그먼트를 제거
- 고정된 크기로 앱이 실행되어 확장이 불가능
따라서 가득차면 JIT 컴파일러가 추가적으로 코드를 컴파일 하지 않음 (성능 부하 야기)

Heap 영역을 제외한 GC 처리 영역

* Native Memory

- Metaspace (Permanent Generation)
- Threads
- Code Cache
- GC (Garbage Collection)
- Symbols
- Native Byte Buffers

[3] Java에서 지원하는 GC 알고리즘

Java에서 지원하는 GC 알고리즘

GC 종류

- Serial GC
- Parallel GC (JDK 7)
- ~~CMS GC~~
 - Remove JDK 14
- G1 GC (JDK 9)
- Shenandoah GC
 - OpenJDK에만 존재
- ZGC
- Epsilon GC

Java에서 지원하는 GC 알고리즘

GC Phase Properties

- a parallel phase
 - 멀티 스레드로 실행 가능
- a serial phase
 - 싱글 스레드로 실행 가능
- a stop-the-world phase
 - 앱 작업과 동시에 실행될 수 없음
- a concurrent phase
 - 백그라운드에서 실행되어 앱 작업과 동시에 실행될 수 있음
- an incremental phase
 - 작업 완료전에 종료하고 나중에 이어서 작업할 수 있음

Java에서 지원하는 GC 알고리즘

Serial GC & Parallel GC

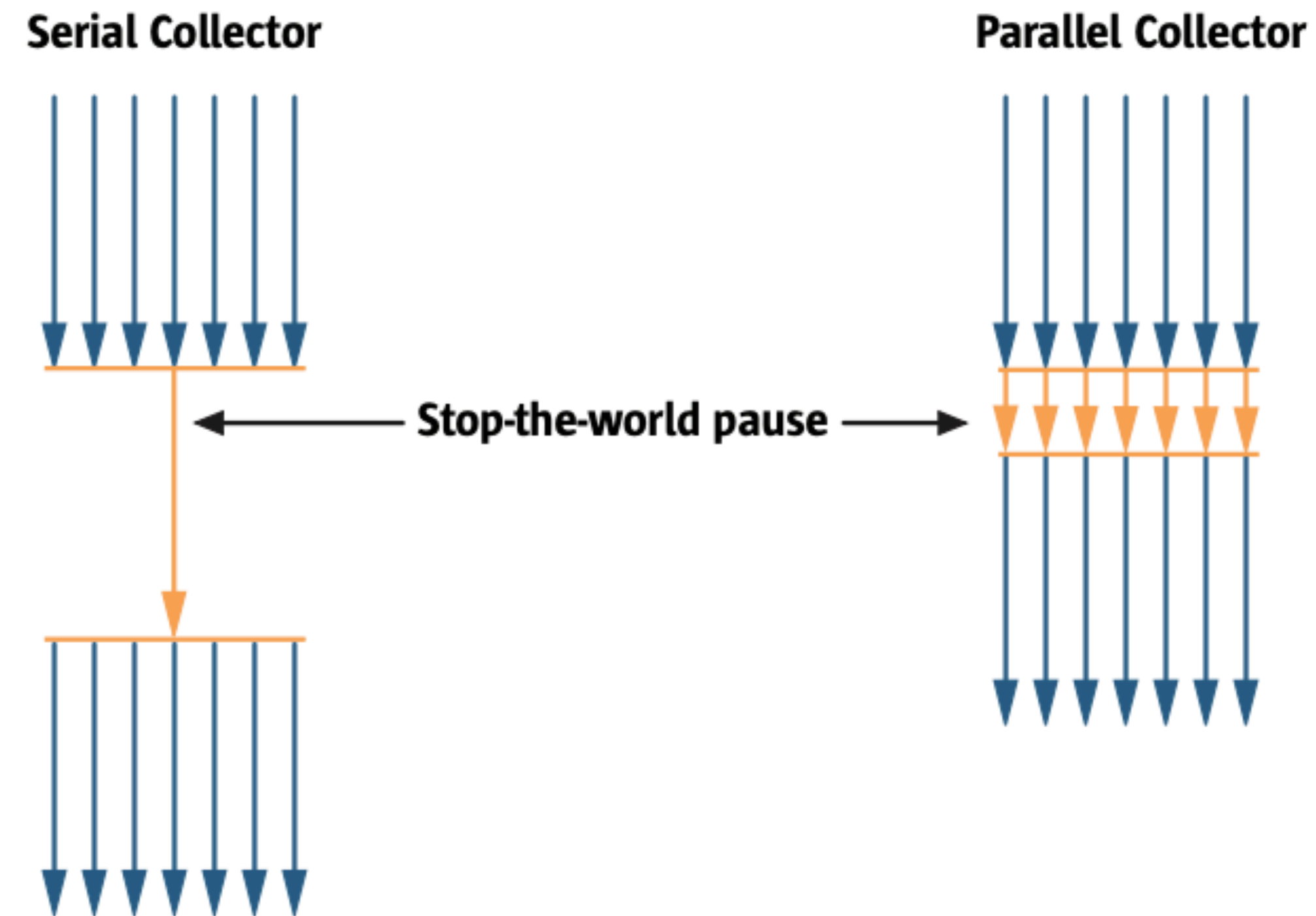


Figure 6. Comparison between serial and parallel young generation collection

Java에서 지원하는 GC 알고리즘

Serial GC

- 싱글 스레드로 동작하는 가장 간단한 GC
- 실행 시 모든 스레드의 STW
따라서 서버 환경에 적합하지 않음
- 클라이언트 앱 또는 일시 중지 시간에 대한 제한이 없는 경우에 대부분 선택하는 GC
- 활성화 옵션
 - ``java -XX:+UseSerialGC -jar <Application.java>``

Java에서 지원하는 GC 알고리즘

Parallel GC

- Serial GC와 다르게 힙 영역 관리를 위해 멀티 스레드를 활용하는 GC 하지만 GC 수행 시에는 똑같이 다른 앱 스레드도 정지 (STW)
- 종종 처리량(Throughput) 수집기라고도 표현
- 이 알고리즘 사용 시 최대 GC 스레드 수와 일시 정지 시간, 처리량, 힙 메모리 사용량 등을 지정할 수 있음
 - 최대 GC 스레드 수 -> ``-XX:ParallelGCThreas=<Number>``
 - 최대 일시 정지 시간 -> ``-XX:MaxGCPauseMillis=<Number>``
 - 최대 목표 처리량 -> ``-XX:GCTimeRatio=<Number>``
처리량은 GC 수집 시간과 GC 수집 시 외부에서 소요된 시간 최대 목표 처리량이라고 함
 - 최대 힙 크기 -> ``-Xmx<Number>``
- 활성화 옵션
 - ``java -XX:+UseParallelGC -jar <Application.java>``

Java에서 지원하는 GC 알고리즘

Serial GC & CMS GC

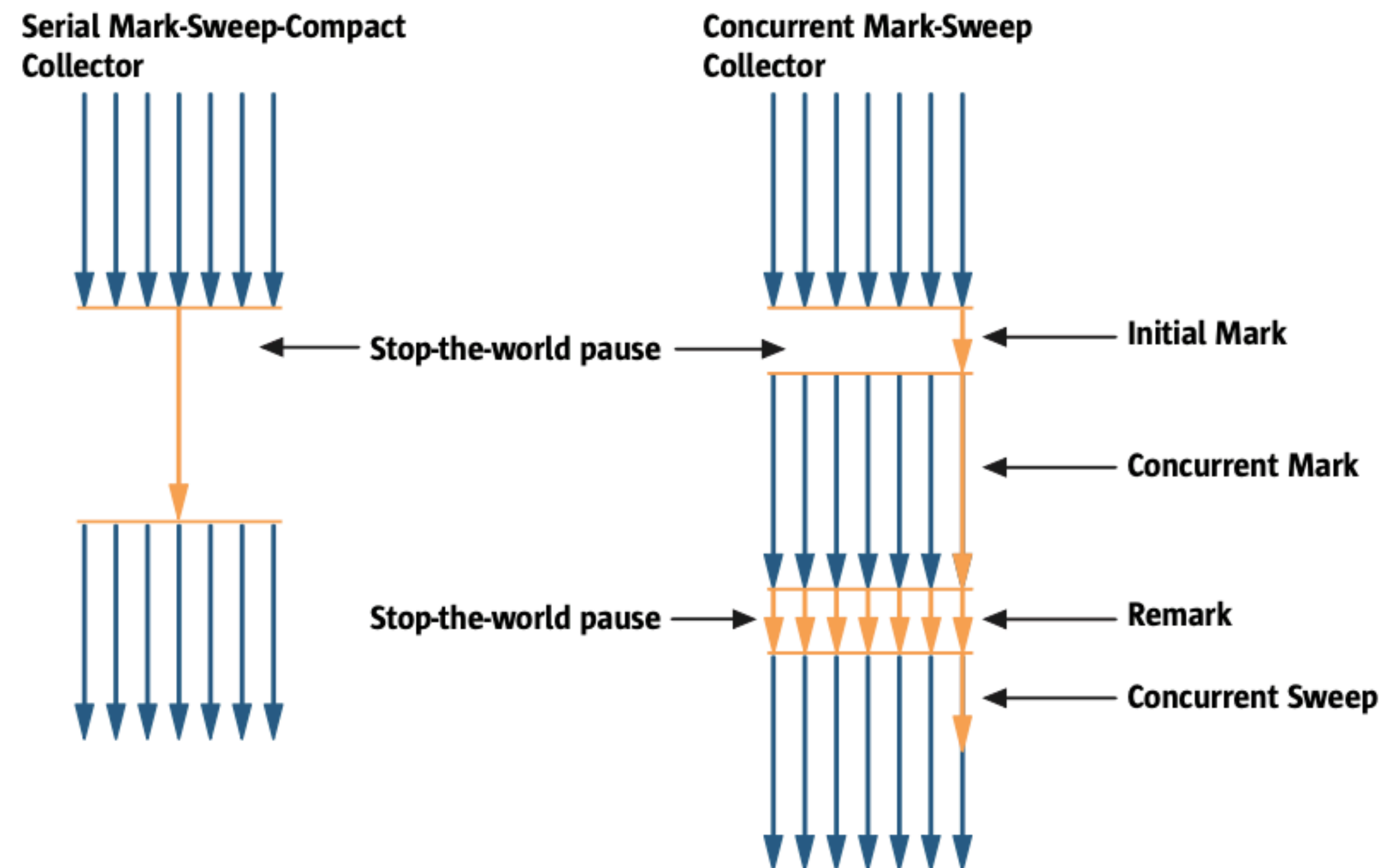


Figure 7. Comparison between serial and CMS old generation collection

Java에서 지원하는 GC 알고리즘

CMS GC

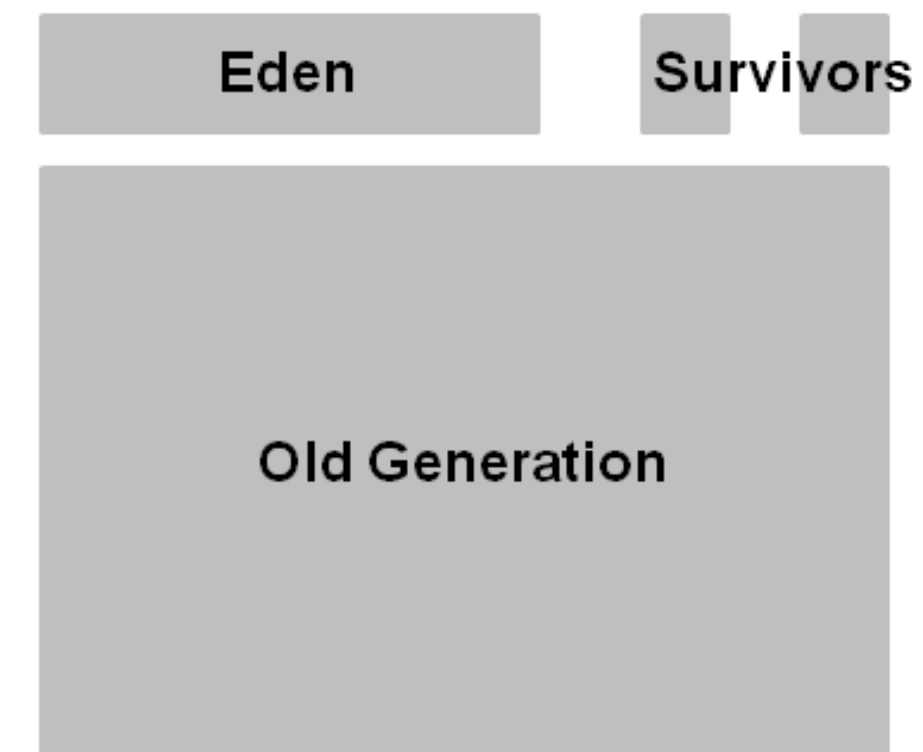
- CMS(Concurrent Mark Sweep)은 멀티 GC 스레드를 활용한 GC (증분모드 포함 14 버전에서 제거됨)
 - 더 짧은 STW를 위해 설계된 GC이며 실행 중에 프로세스 리소스를 GC와 공유할 수 있음
- 평균적으로 다소 느리지만 GC 처리 시 응답 처리 가능
- GC의 동시성 처리를 지원하기 때문에 실행하는 동안 `System.gc()` 처럼 명시적 호출은 동시 처리를 중단 시키거나 실패를 초래할 수 있음
- GC 수집 총 시간의 98% 이상인데 힙 복구가 2% 미만이라면 OutOfMemoryError 발생
 - 위 옵션 비활성화 `-XX:UseGCOverheadLimit`
- 활성화 옵션
 - `java -XX:+UseParNewGC -jar Application.java`

Java에서 지원하는 GC 알고리즘

CMS GC Steps - Heap Structure

- 총 3개의 영역으로 나뉨
 - Young Generation
 - Eden
 - Survivors
 - Old Generation
- 객체 수집(제거)는 그 곳에서 수행되며 Full GC 수행 시에만 컴팩션이 수행됨

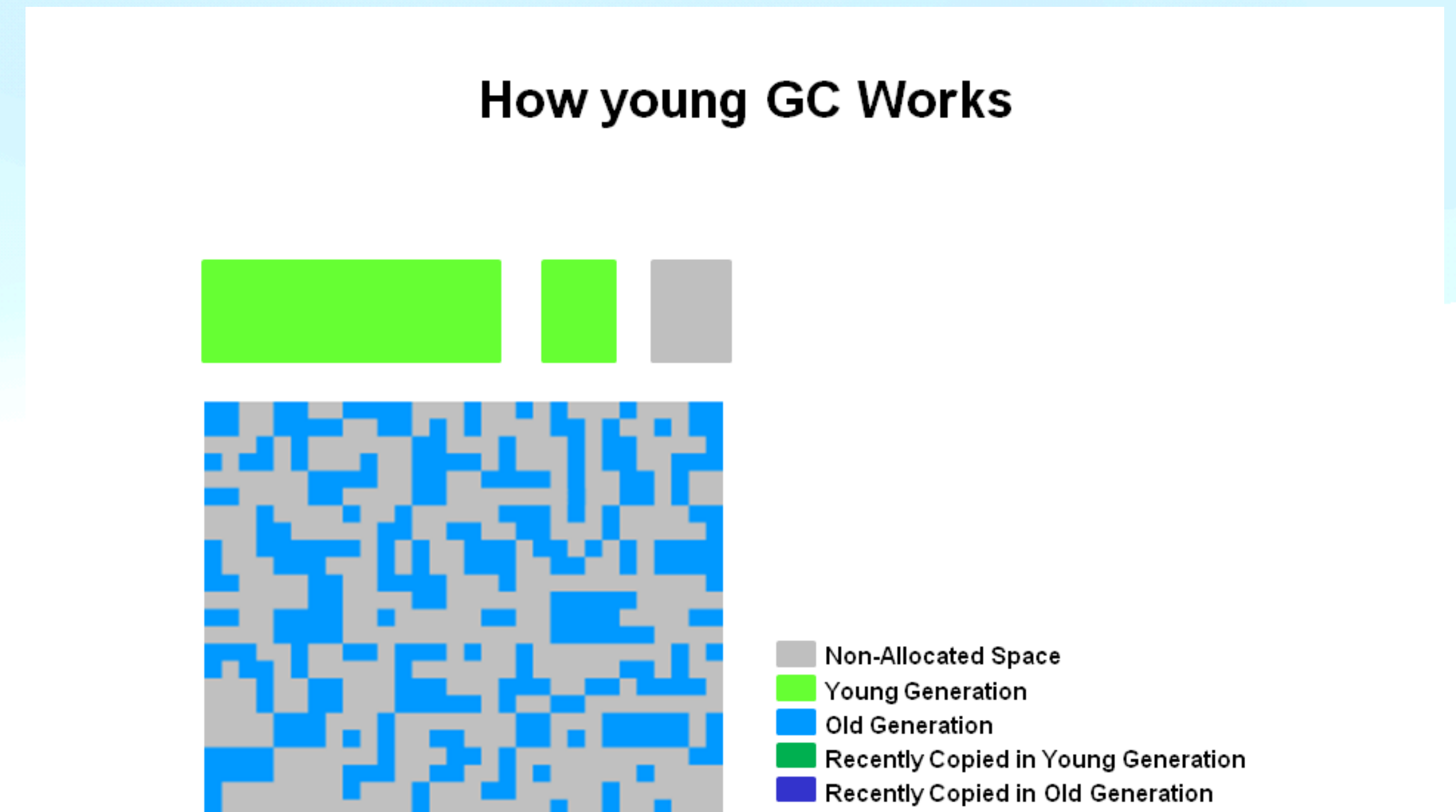
CMS Heap Structure



Java에서 지원하는 GC 알고리즘

CMS GC Steps - Minor GC

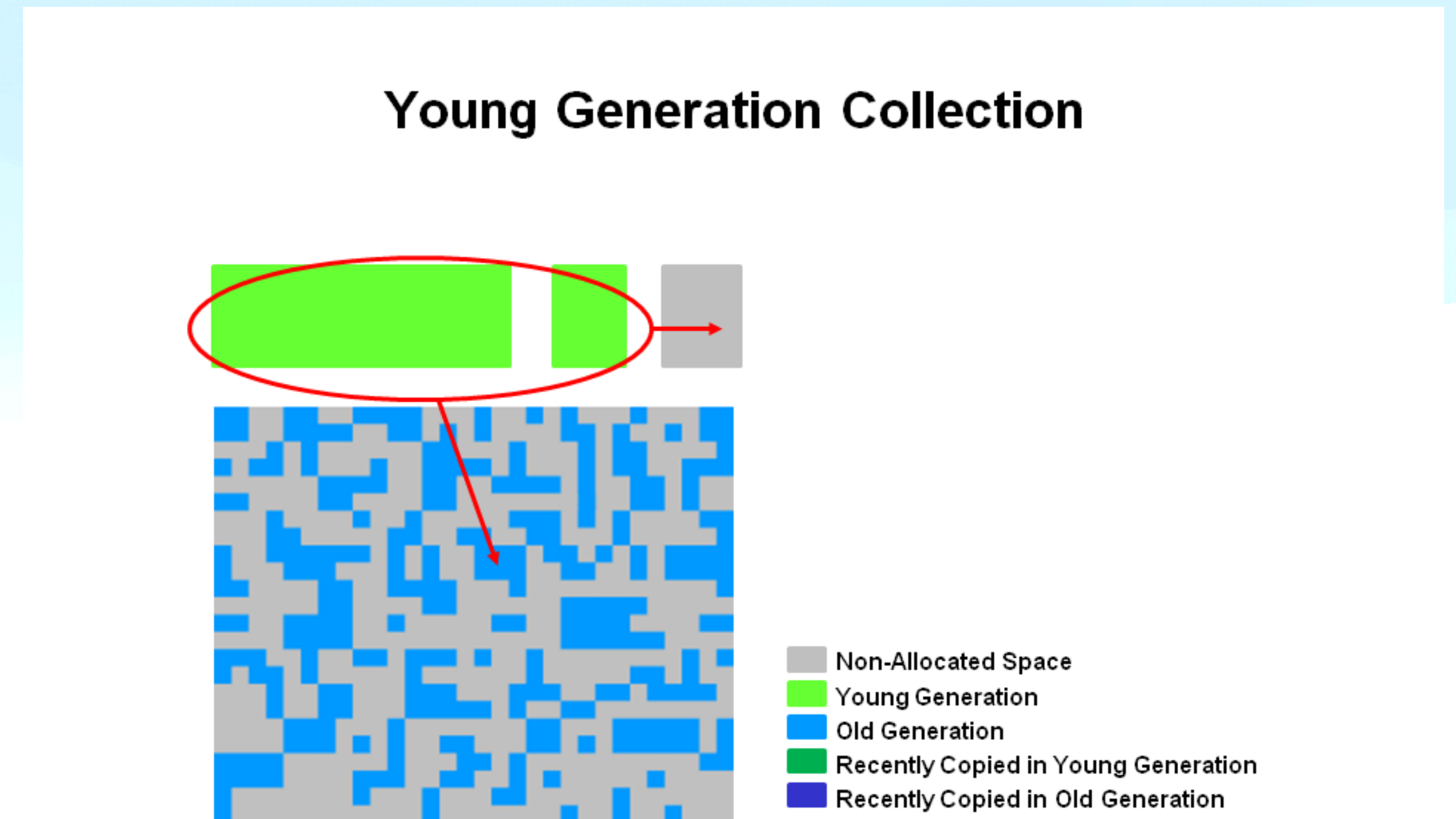
- 앱이 한동안 실행되었을 때를 가정한 그림
- 객체는 Old 제너레이션 영역 주변에 흩어짐
- 앞서 설명한 것처럼 해당 자리에서 할당/해제되며 Full GC 없이 컴팩션 작업이 발생하지 않음



Java에서 지원하는 GC 알고리즘

CMS GC Steps - Minor GC

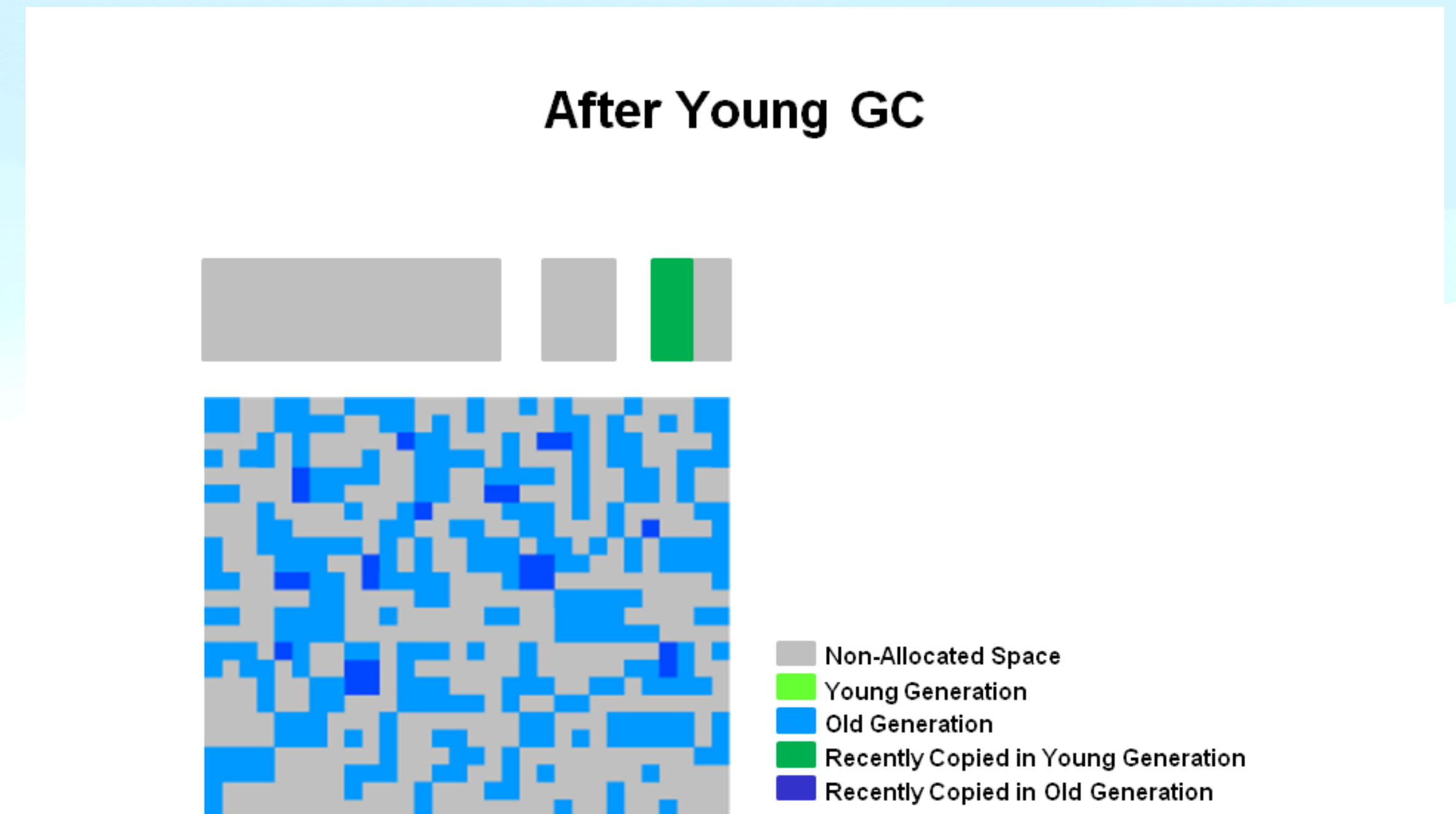
- Live 객체는 Eden과 Survivor 영역에서 다른 Survivor 영역으로 복사(이동)됨
- 또한 특정 기준 임계값을 넘어서 객체들은 Old 제너레이션으로 승격됨



Java에서 지원하는 GC 알고리즘

CMS GC Steps - Minor GC

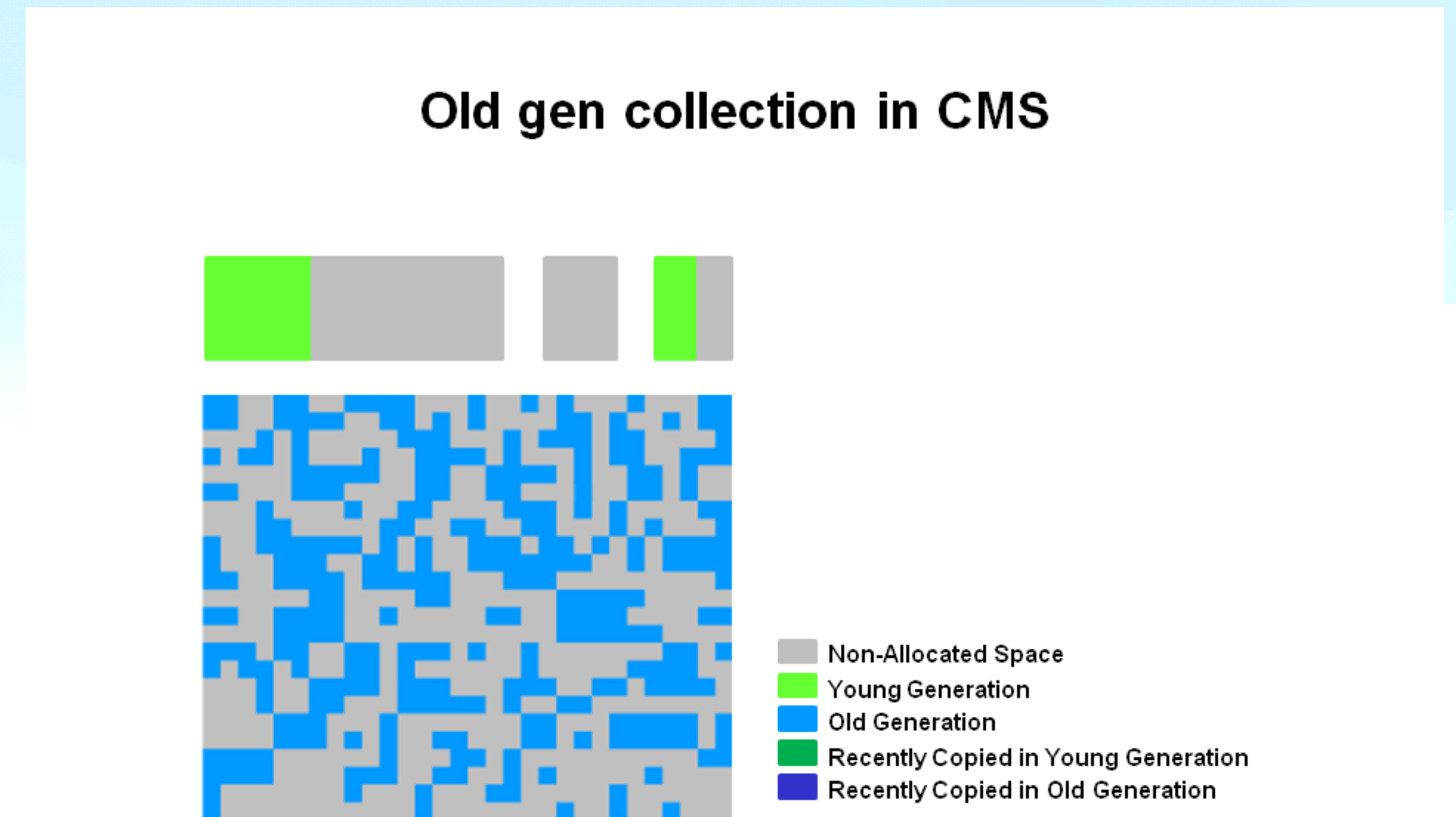
- Minor GC 이후 Eden 영역과 Survivor 영역 중 한 곳이 비워짐



Java에서 지원하는 GC 알고리즘

CMS GC Steps - Full GC

- STW가 두 곳에서 발생
Initial Mark와 Remark
- Old Generation의 점유율이 기준치에 도달하면
CMS가 시작됨
 1. Initial Marking
 2. Concurrent Marking
 3. Remarking

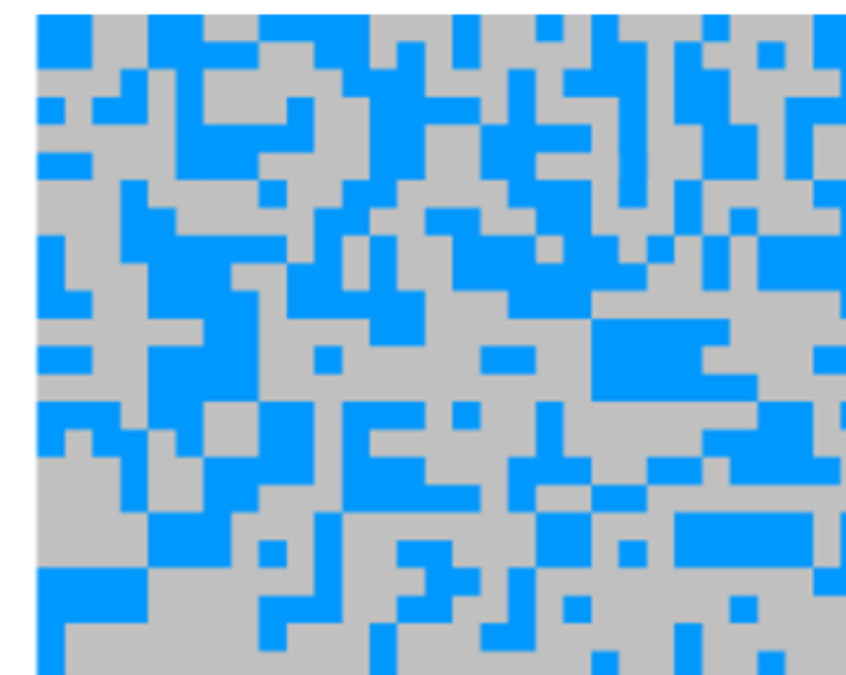


Java에서 지원하는 GC 알고리즘

CMS GC Steps - Full GC

- 마킹되지 않은 객체는 제거(수집)되며 컴팩션 작업은 없음

Old Gen Collection – Concurrent Sweep



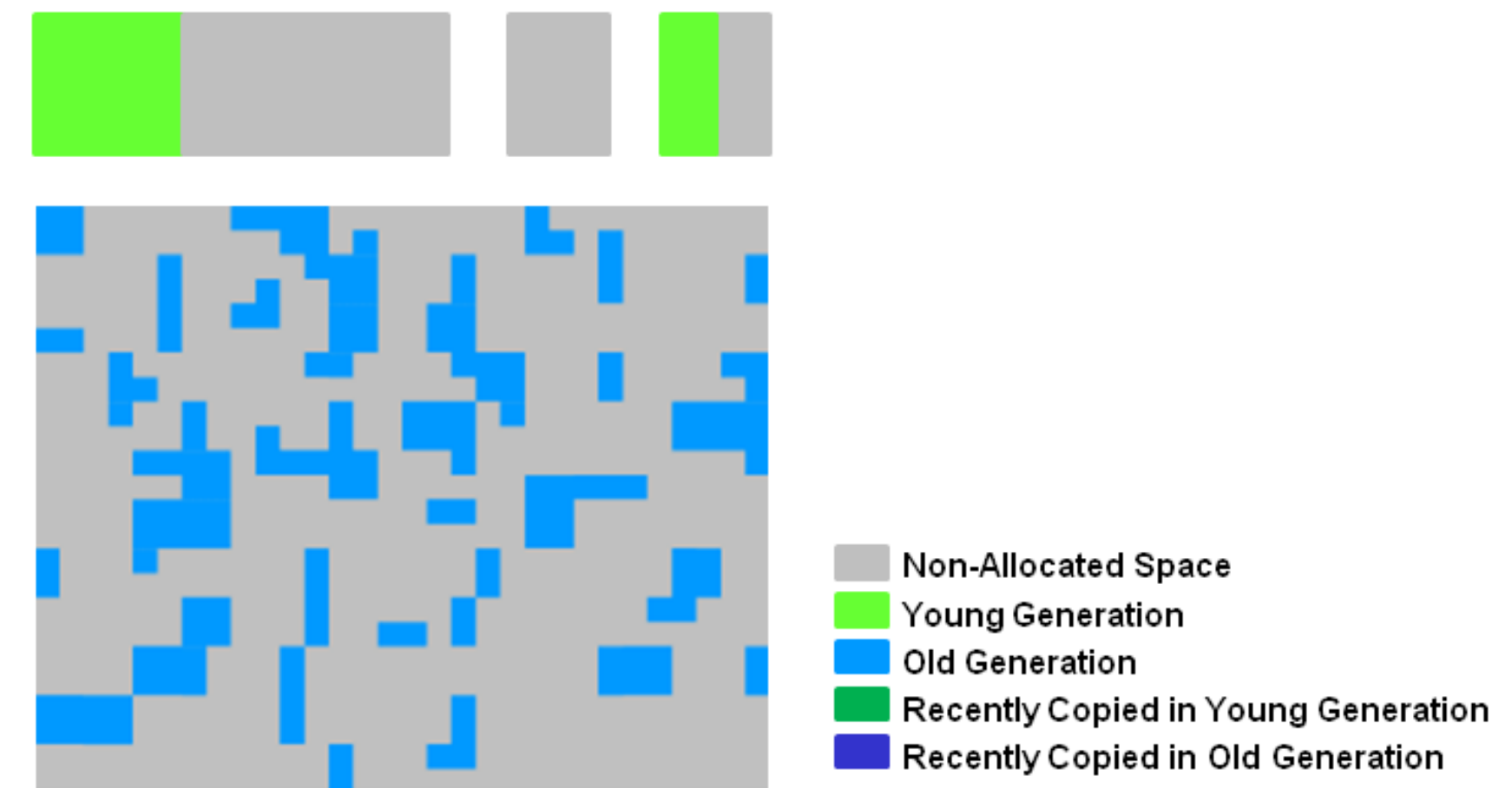
■ Non-Allocated Space
■ Young Generation
■ Old Generation
■ Recently Copied in Young Generation
■ Recently Copied in Old Generation

Java에서 지원하는 GC 알고리즘

CMS GC Steps - Full GC

- 제거(Sweeping) 후 많은 메모리가 확보된 상태
- 마찬가지로 컴팩션 작업은 수행되지 않음
- 메모리 단편화 문제가 발생
이를 극복하기 위해 G1 GC가 등장

Old Gen Collection – After Sweeping



Java에서 지원하는 GC 알고리즘

CMS GC Steps - Summary

1. Initial Mark (STW)

- GC 루트로부터 도달 가능한 객체를 마킹 (Old 제너레이션 내에 객체는 도달 가능한 것으로 간주됨)
일반적인 마이너 GC 중지 시간에 비해 짧음

2. Concurrent Marking

- 앱이 실행되는 동안 Initial Marking 된 객체들에 그래프를 통해 추적
- 뮤테이터(실제 작업을 수행하는 스레드)는 Concurrent Marking, Remark, Resetting 동안 실행되며
이때 생성된 객체들은 모두 Live 객체로 마킹됨

3. Remark (STW)

- 앞 단계에서 마킹이 누락된 객체들을 찾는 작업 수행

4. Concurrent Sweep

- 추적 불가능한 객체들을 수집(제거)하고 추후 할당을 위한 사용 가능한 영역 목록에 해당 영역 추가
이 시점에 죽은 객체들의 병합이 발생할 수도 있음

5. Resetting

- 자료 구조들을 제거하고 다음 GC 작업을 준비

Java에서 지원하는 GC 알고리즘

G1 GC


- G1(Garbage First)은 메모리 공간이 큰 멀티 프로세스 머신에서 실행되는 앱을 기준으로 설계된 GC 성능적으로 CMS보다 효율적이어서 대체됨
- 힙을 동일한 크기의 영역으로 분할하여 관리 (해당 영역들은 가상 메모리의 연속 범위)
- GC 작업 수행 과정
 - GC 수집 시 힙 전체 영역에서 객체 활성(참조)상태를 식별하기 위해 동시 글로벌 마킹 처리
 - 마킹 후 비어있는 영역 확인, 일반적으로 비교적 많은 여유 공간을 생성할 수 있는 영역부터 처리
- 활성화 옵션
 - ``java -XX:+UseG1GC -jar Application.java``

Java에서 지원하는 GC 알고리즘

G1 GC Steps - Heap Structure

- G1 GC에서 힙은 하나의 메모리 영역
- 영역(Region)의 크기는 시작 시 JVM에서 판단하며 일반적으로 하나당 1MB~32MB로 다양한 2,000개 영역을 대상으로 함

G1 Heap Structure



One memory area
split into many fixed
sized regions

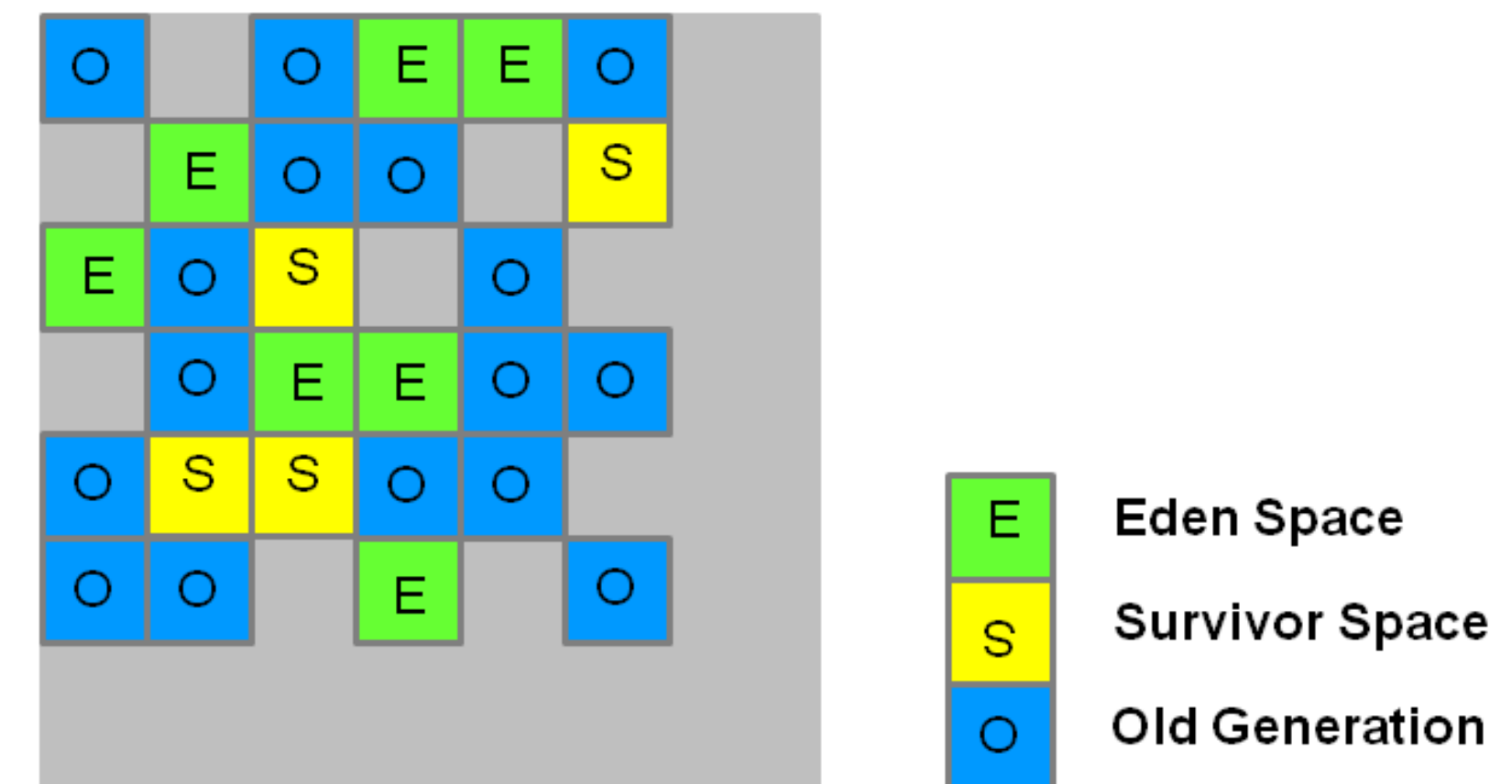
The diagram shows a large gray rectangle representing the heap memory. Inside this rectangle, the text 'One memory area split into many fixed sized regions' is centered, indicating that the heap is divided into many small, fixed-sized regions.

Java에서 지원하는 GC 알고리즘

G1 GC Steps - Heap Allocation

- 이 리전들은 Eden, Survivor, Old 제너레이션 영역의 논리적인 표현으로 매핑됨
- Live 객체는 다른 영역으로 이동(복사)됨
- 리전들은 다른 모든 앱 스레드와 병렬로 실행되거나 중지시키지 않도록 설계됨
- Eden, Survivor, Old 제너레이션 외에도 Humongous이라는 특별한 리전이 존재하고 표준 리전 크기의 50% 이상의 객체를 보관하도록 설계되었으며 인접 리전들의 집합으로 구성됨

G1 Heap Allocation

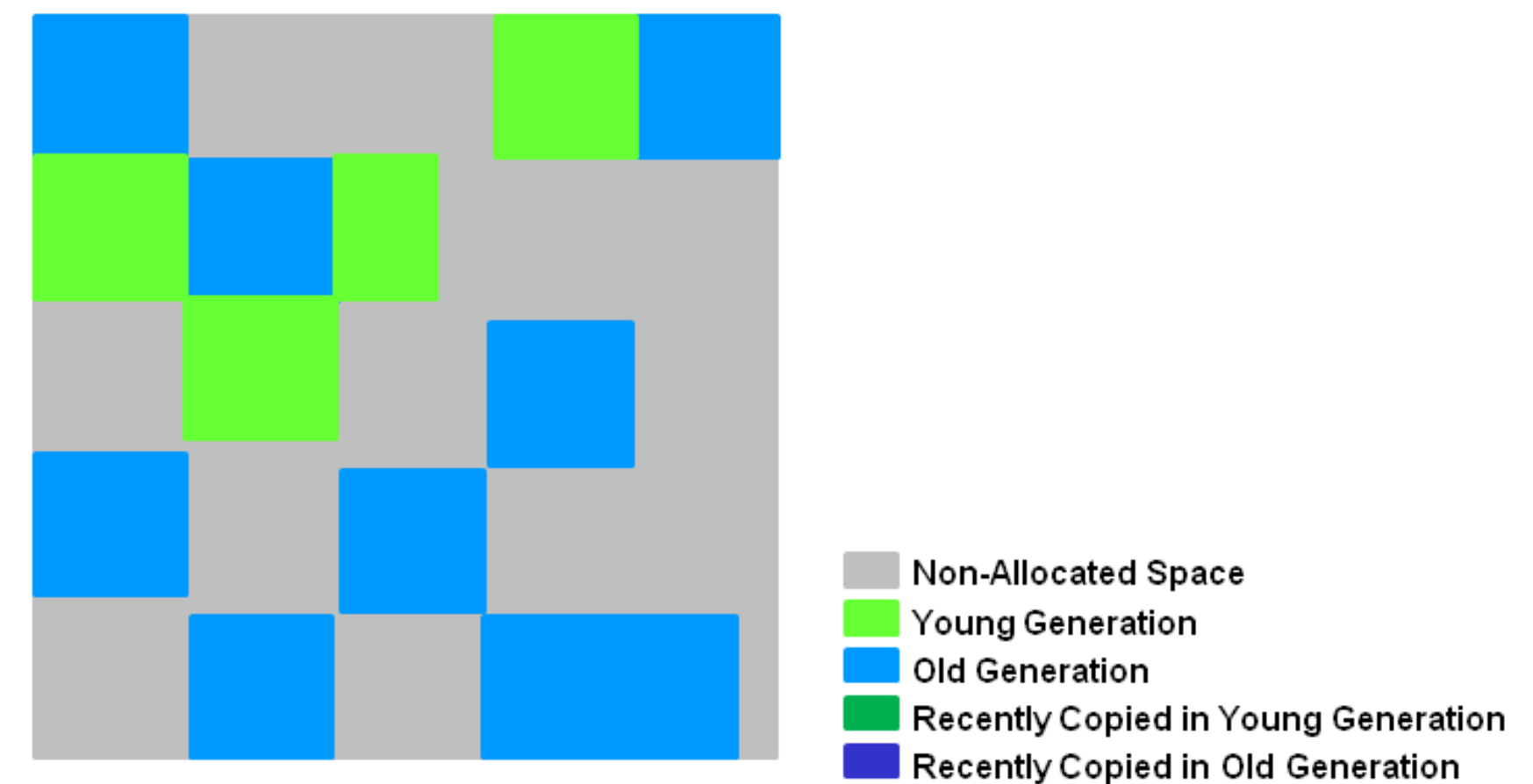


Java에서 지원하는 GC 알고리즘

G1 GC Steps - Region

- 힙은 약 2,000개의 리전으로 분할됨
1MB~32MB로 이전 GC처럼 리전이 연속적일 필요가 없음

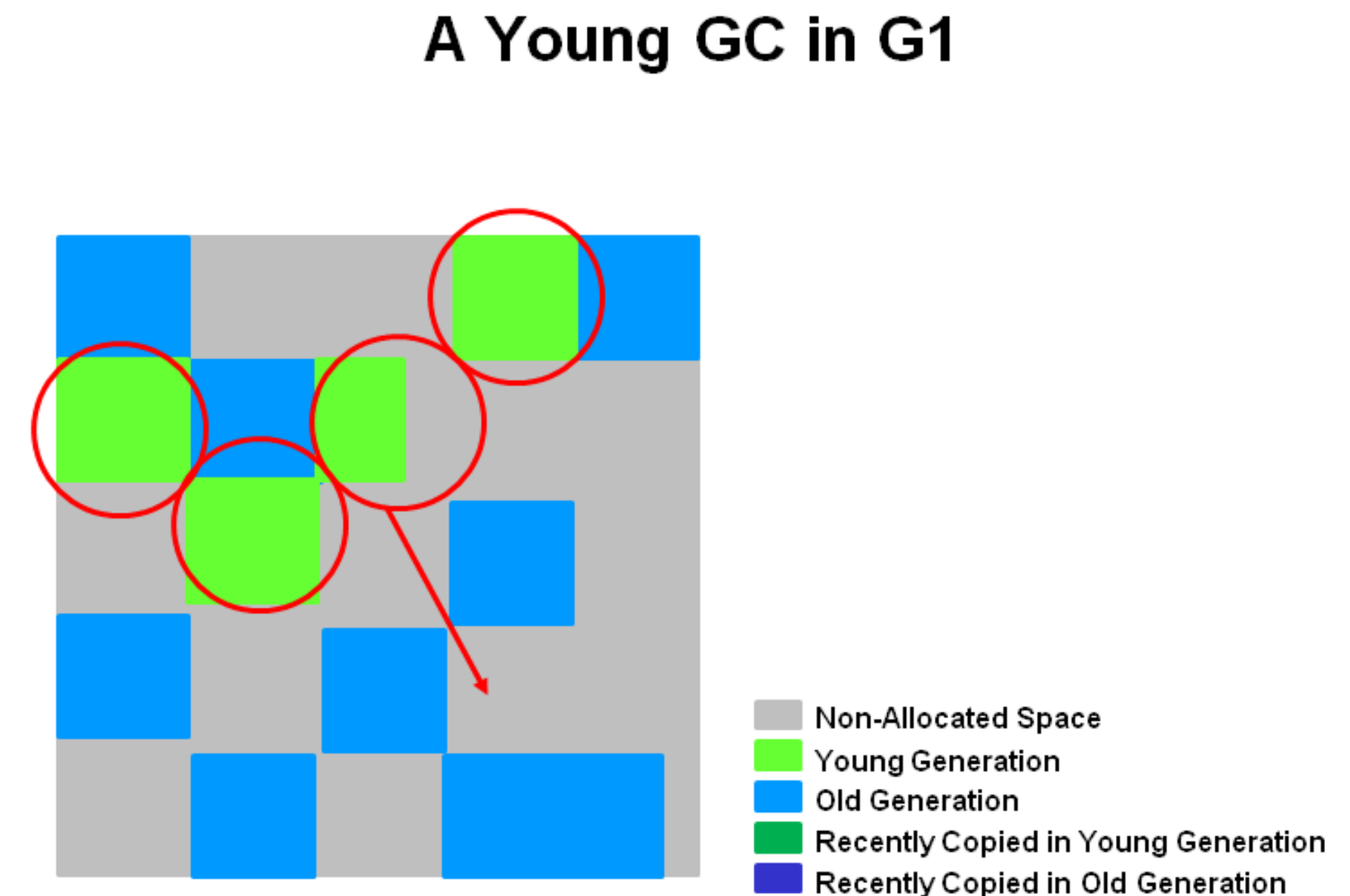
Young Generation in G1



Java에서 지원하는 GC 알고리즘

G1 GC Steps - Young GC

- Live 객체는 하나 이상의 survivor 리전으로 이동하며 객체가 에이징 임계값이 충족되면 Old 제너레이션으로 승격됨
- 이 작업은 STW에 해당하며 Eden과 Survivor의 크기는 다음 Young GC를 위해 계산됨
- 어카운팅 정보는 크기를 계산하는데 활용됨
일시 중지 시간 등이 고려되며 리전 크기를 쉽게 조정하여 필요에 따라 더 크거나 작게 만들 수 있음

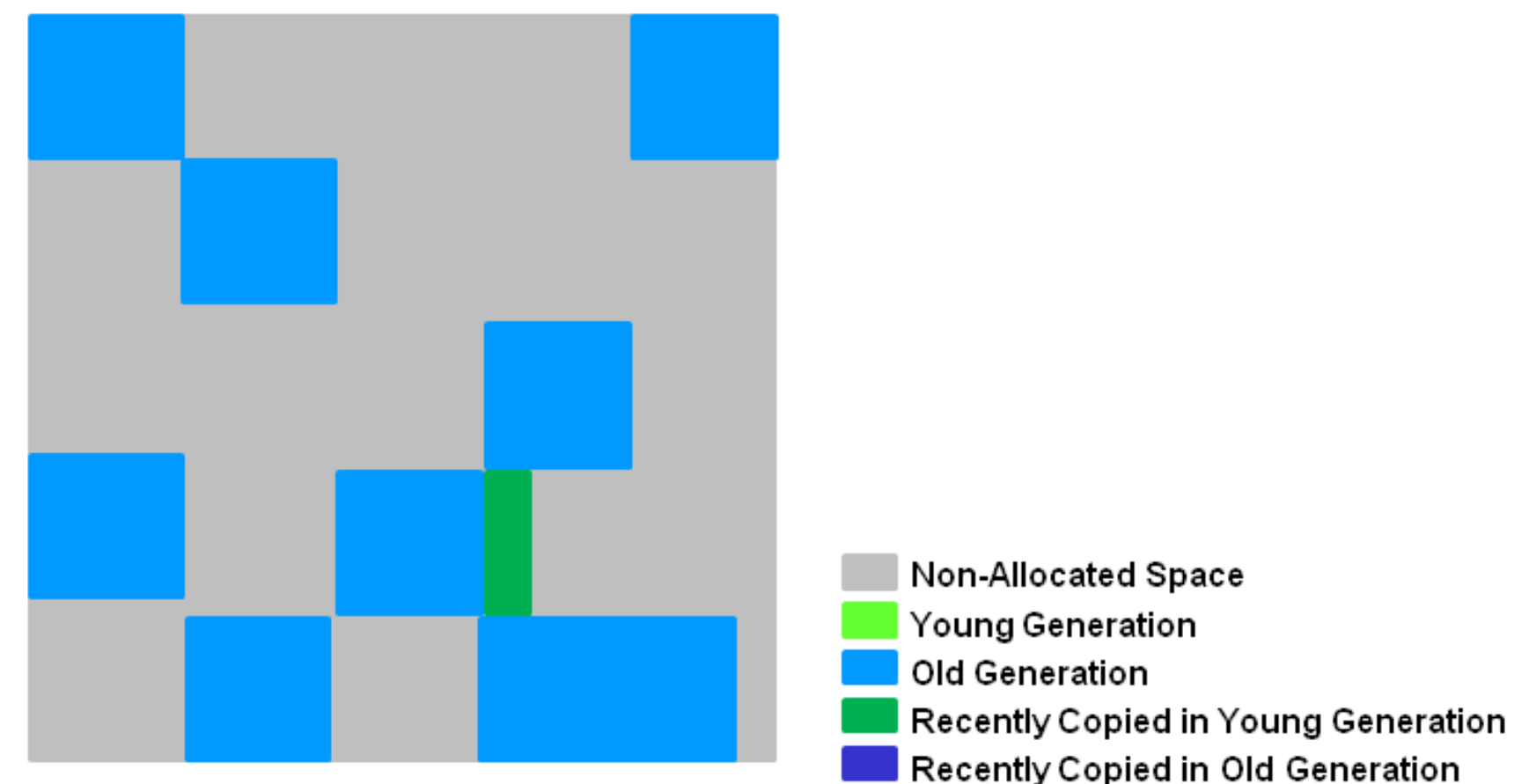


Java에서 지원하는 GC 알고리즘

G1 GC Steps - Young GC

- Live 객체는 Survivor 또는 Old 제너레이션으로 이동됨
- 정리
 - 힙은 리전으로 분할된 단일 메모리 공간
 - Young 제너레이션 메모리는 연속적이지 않은 리전의 집합으로 구성 (쉽게 크기 조정 가능)
 - Young 제너레이션 GC는 STW이며 이 작업을 위해 모든 앱 스레드가 중지됨
 - Young GC는 여러 스레드로 병렬로 수행됨
 - 위에서 설명처럼 Live 객체는 다른 survivor 또는 Old 제너레이션으로 이동됨

End of Young GC with G1

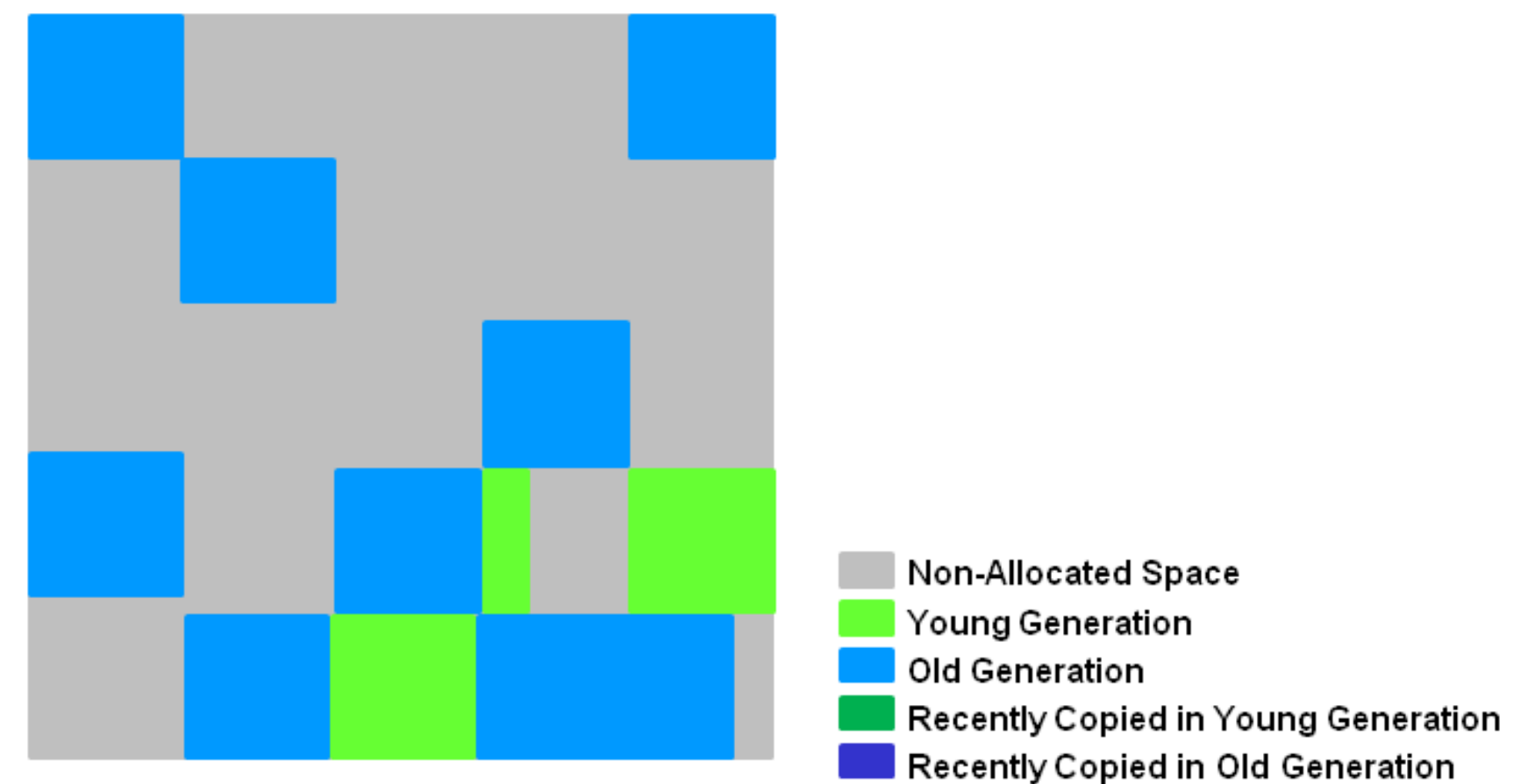


Java에서 지원하는 GC 알고리즘

G1 GC Steps - Full GC

- Live 객체 마킹은 Young GC 작업과 결합하여 수행되며 로그에 `GC pause(young)(initial-mark)`라고 표기됨

Initial Marking Phase

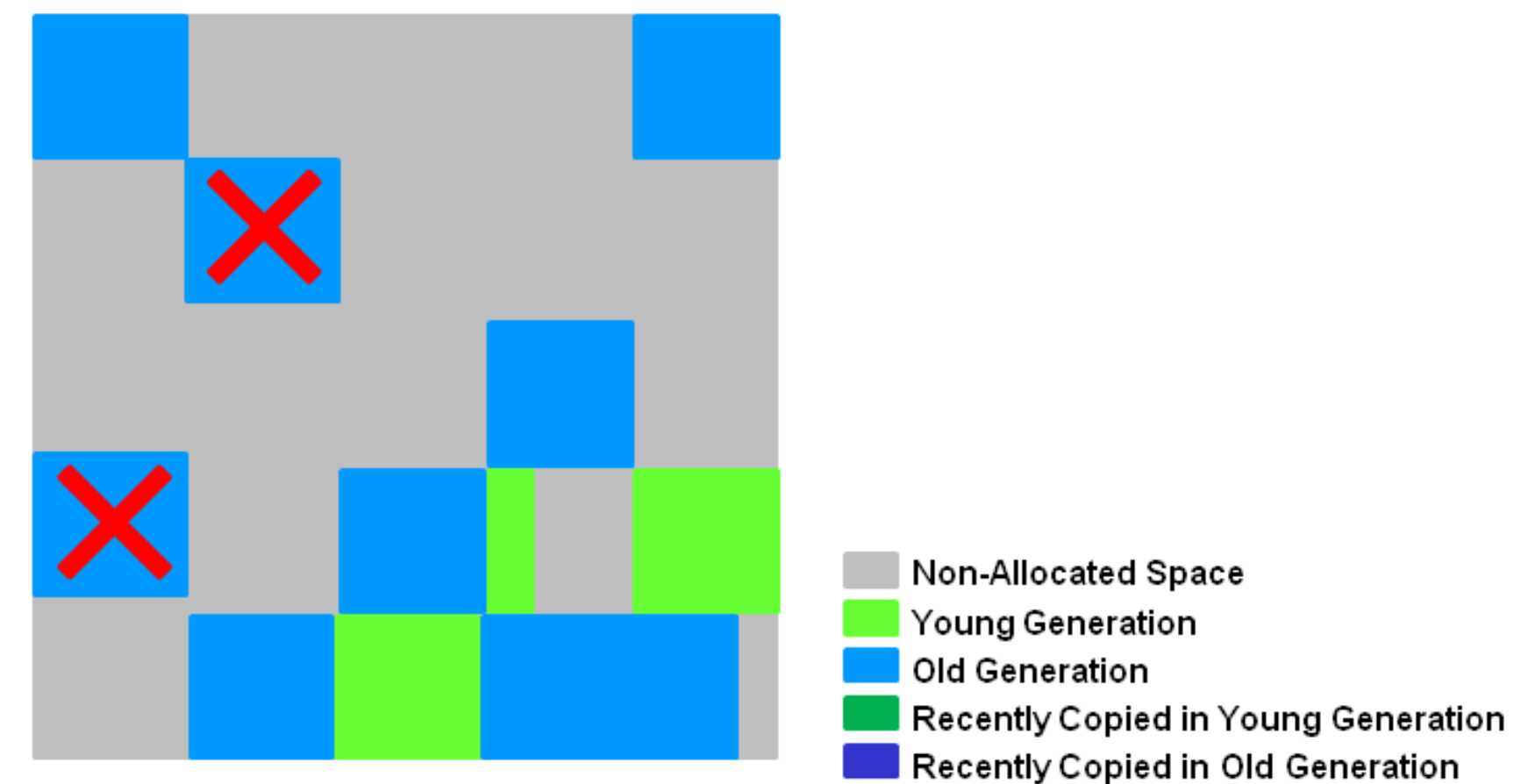


Java에서 지원하는 GC 알고리즘

G1 GC Steps - Full GC

- `X`라고 표기된 빈 리전이 발견되면 Remark 단계에서 즉시 제거되며 활성을 결정하는 어카운팅 정보가 계산됨

Concurrent Marking Phase

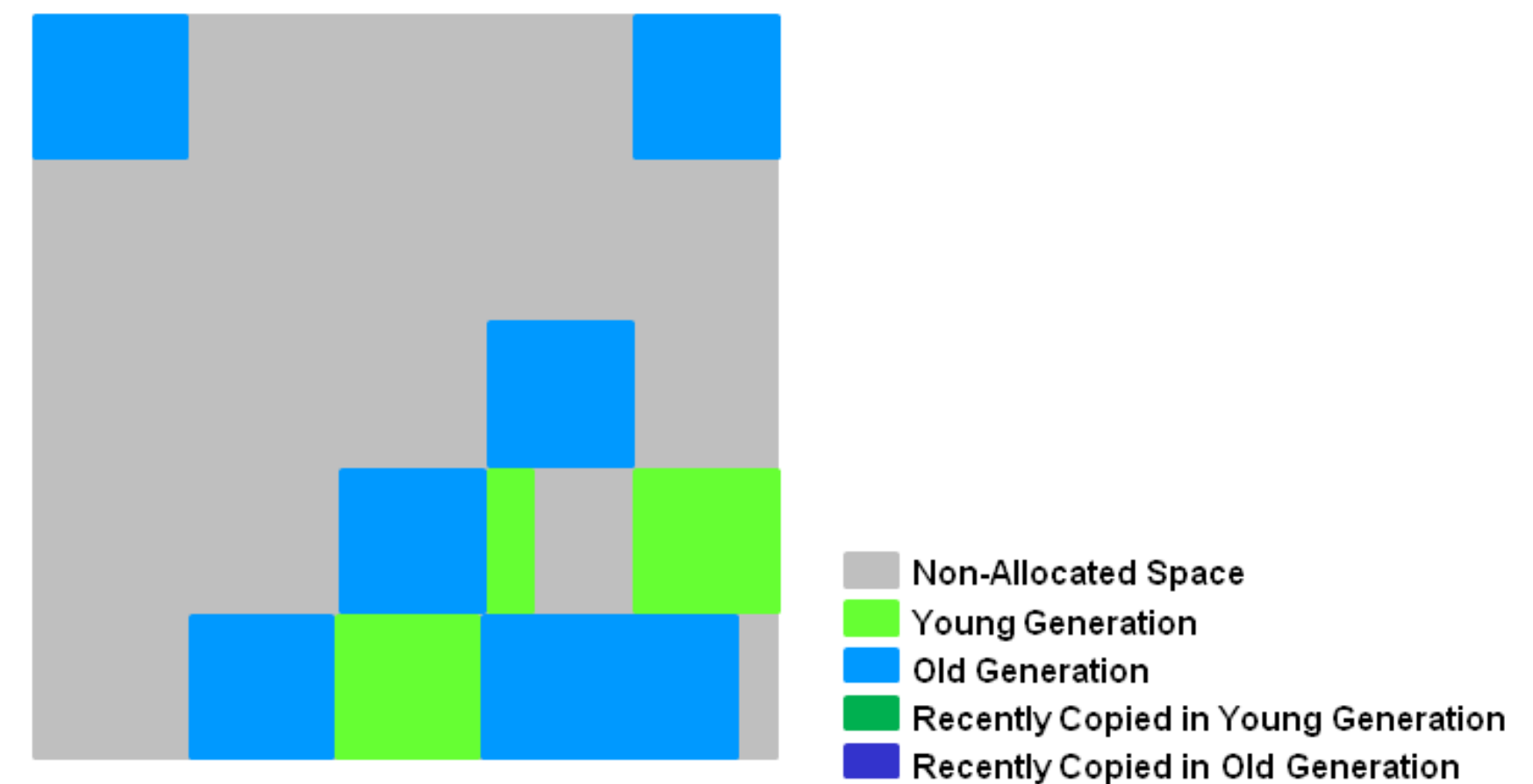


Java에서 지원하는 GC 알고리즘

G1 GC Steps - Full GC

- 빈 리전이 제거되고 회수되며 모든 리전을 위해 리전의 라이브네스가 계산됨

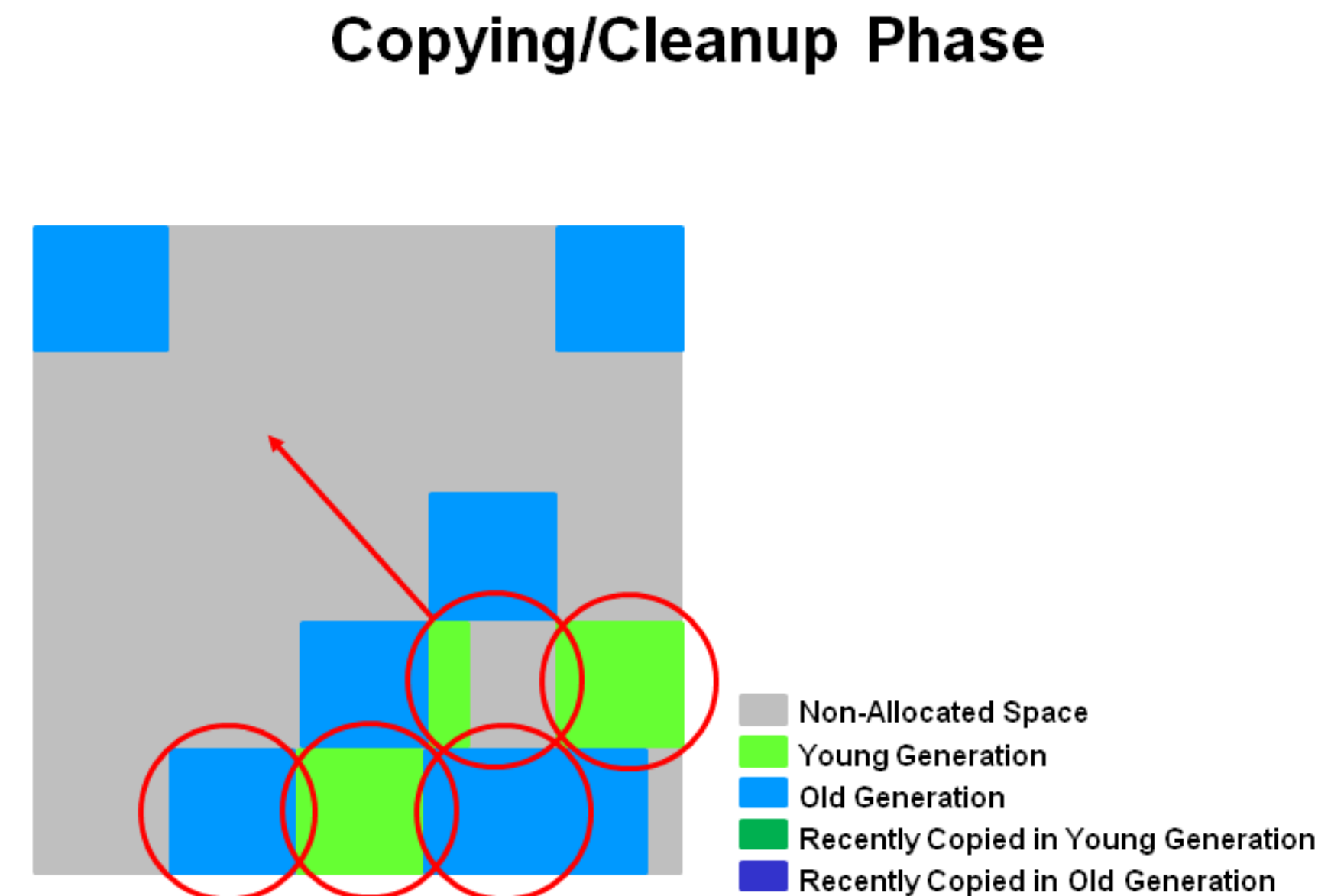
Remark Phase



Java에서 지원하는 GC 알고리즘

G1 GC Steps - Full GC

- G1 GC는 라이브네스가 가장 낮은 리전, 즉 제일 빨리 수집 가능한 지역을 선택하고 이 지역은 Young GC와 동시에 수집되며 로그에 [GC pause (mixed)]로 표기됨 그래서 Young, Old 제너레이션이 동시에 수집됨

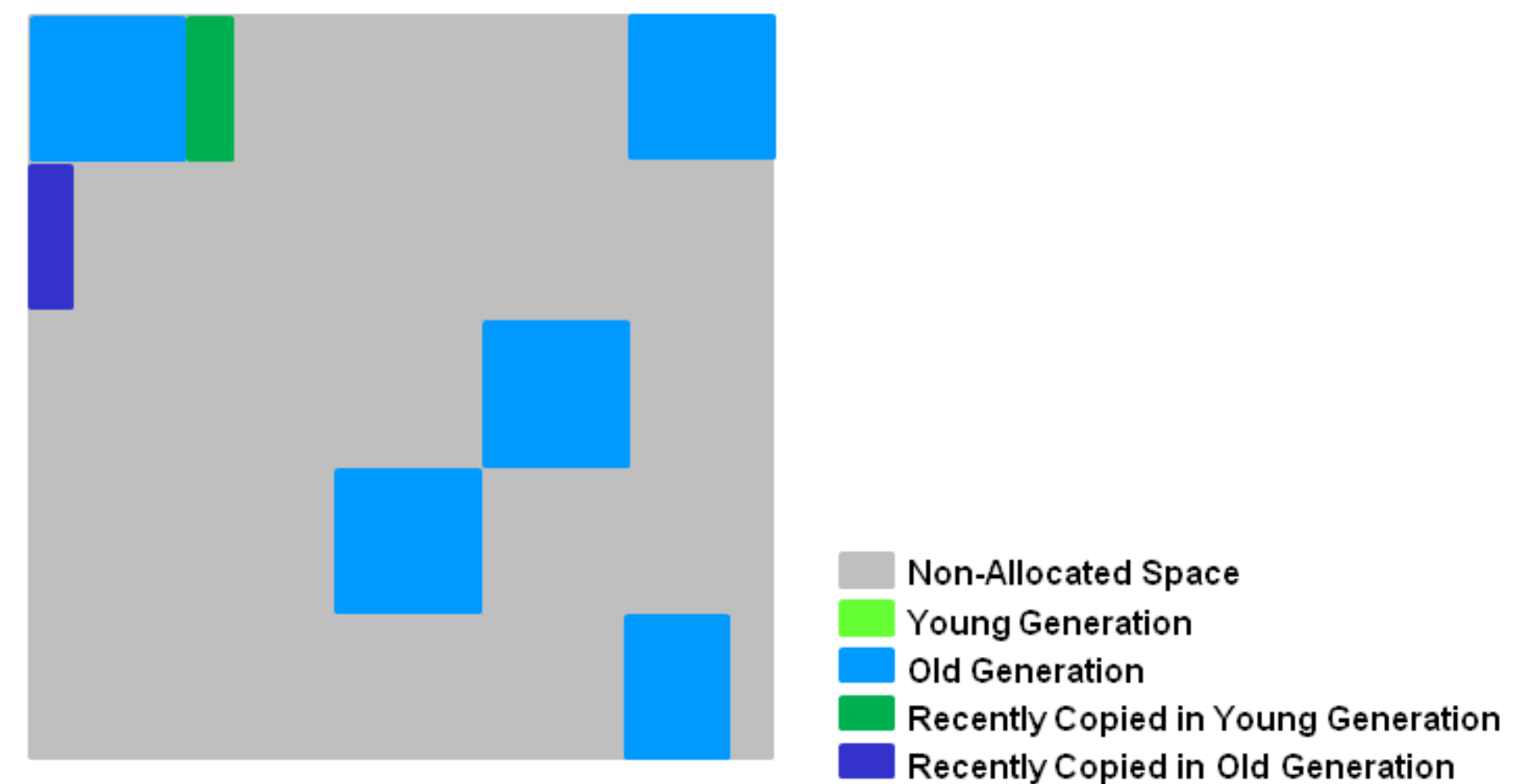


Java에서 지원하는 GC 알고리즘

G1 GC Steps - Full GC

- 선택한 리전이 수집되어 그림과 같이 압축됨

After Copying/Cleanup Phase



Java에서 지원하는 GC 알고리즘

G1 GC Steps - Summary

1. Initial Mark (STW)
 - 일반적인 Young GC이며 STW 이벤트
2. Root Region Scanning
 - 앱이 실행되는 동안 Old 제너레이션을 위해 survivor 리전 스캔
Young GC가 발생되기 전에 완료해야 함
3. Concurrent Marking
 - 앱이 실행되는 동안 전체 힙에서 Live 객체를 찾음
이 작업은 Young GC가 발생되면 중단될 수 있음
4. Remark (STW)
 - 힙에서 Live 객체 마킹을 완료
CMS GC에서 사용된 것보다 훨씬 빠른 SATB 알고리즘 사용
 - SATB(snapshot-at-the-beginning)는 힙의 특정 시점의 스냅샷을 유지하고 그 시점에 Live 객체만을 마킹함
즉 해당 시점에서 참조가 끊긴 객체도 Live 객체라고 판단하여 마킹함
5. Cleanup (STW & Concurrent)
 - Live 객체와 완전하게 할당 가능한 리전을 계산 (STW)
 - 기억된 셋을 스크립 (STW)
 - 빈 리전을 재설정하고, 사용 가능 목록으로 되돌림 (Concurrent)
- [기타] Copying (STW)
 - Live 객체를 사용되지 않는 리전으로 이동하기 위해 발생하는 STW 이벤트
이는 Young 제너레이션에서 수행되거나([GC pause (young)]) Young, Old 제너레이션 모두에서 수행됨([GC pause (mixed)])

Java에서 지원하는 GC 알고리즘

G1 GC Steps - Summary

- Concurrent Marking Phase
 - 라이브네스 정보는 앱이 실행되는 동안 내내 계산됨
 - 라이브네스를 통해 GC 수집 중 가장 좋은 리전을 선별
 - CMS와 같은 스윙핑 단계가 없음
- Remark Phase
 - CMS에서 사용된 것보다 훨씬 빠른 SATB 알고리즘을 사용함
 - 완전히 빈 리전은 회수됨
- Copying/Cleanup Phase
 - Young 제너레이션과 Old 제너레이션에 대한 작업이 동시에 진행됨
 - Old 제너레이션은 라이브네스를 기준으로 선택(식별)됨

Java에서 지원하는 GC 알고리즘

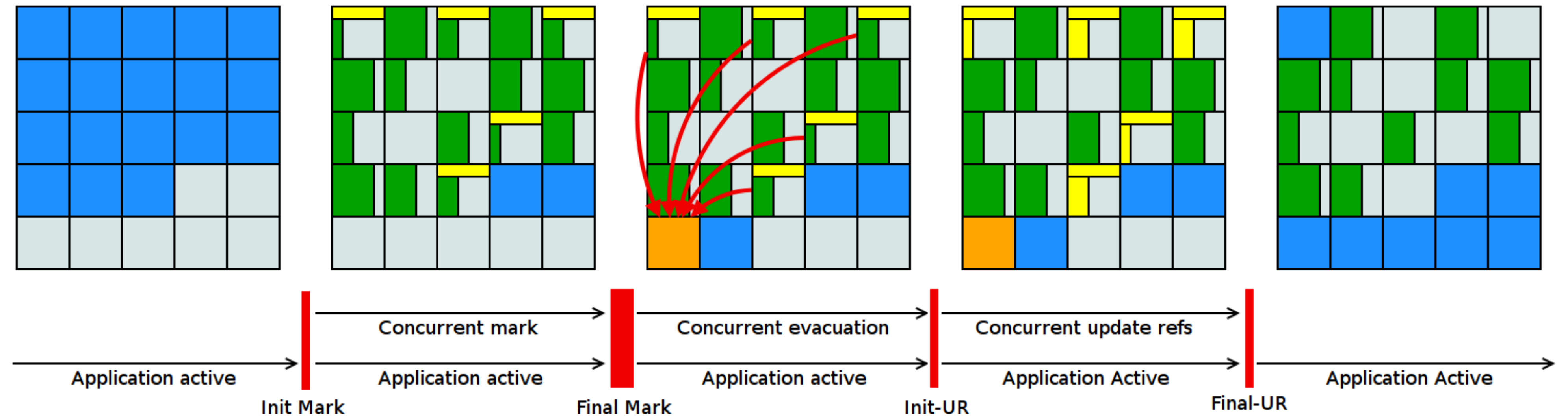
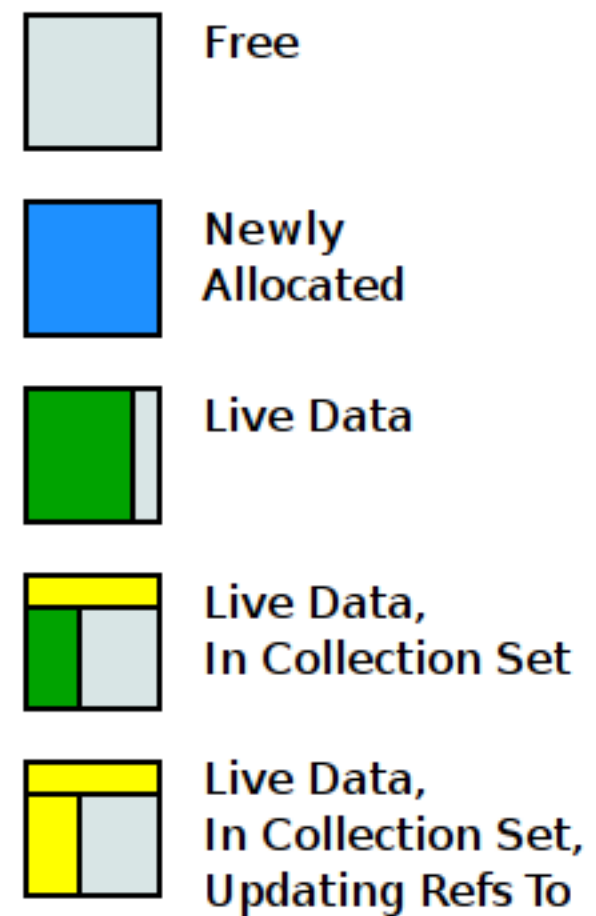
Shenandoah GC

- OpenJDK에서만 존재하는 GC
- G1과 같이 힙 영역이 리전이라는 가상의 공간으로 나뉘짐
하지만 G1을 포함한 다른 GC들과 다르게 힙을 제너레이션으로 나누지 않음
- 앱과 동시에 더 많은 GC 작업을 수행해 일시 중지 시간을 줄인 GC (CMS와 G1 처럼)
세난도아는 동시 압축이 추가된 모델이며 힙 크기와 상관없이 일관된 일시 중지 시간을 가짐
- 응답 시간이 짧고 GC STW가 예측이 가능할 경우 유효한 GC 알고리즘
- 브룩스 포인터를 추가해 참조 재배치 메커니즘을 처리함
브룩스 포인터(Brooks Pointer)는 각 객체의 실제 위치를 가리키는 필드

Java에서 지원하는 GC 알고리즘

Shenandoah GC

Legend:



<https://wiki.openjdk.org/display/shenandoah/Main#Main-SupportOverview>

Java에서 지원하는 GC 알고리즘

Shenandoah GC Steps - Summary

1. Init Mark (STW)

- 동시 마킹 진행하고 이를 위해 힙, 앱 스레드를 준비하고 루트 셋을 스캔하는 단계
- 스캔하는 루트 셋 크기에 따라 STW 시간은 크게 달라짐

2. Concurrent Marking

- 앱 실행과 동시에 힙 내에 객체를 추적하는 단계
- 이 작업의 시간은 Live 객체의 수와 객체 그래프에 따라 달라짐
- 또한 앱은 이 단계에 새 데이터(객체)를 할당 가능 (이 작업이 앱의 실행을 중단시키지 않음)

3. Final Mark (STW)

- 잠시 중단되었던 마킹/업데이트 큐를 비우고 루트 셋을 다시 스캔하여 동시 마킹 작업을 완료하는 단계
- 완료 후 비워져야 할 리전을 파악, 다음 단계를 위해 비우고 초기화 등 작업을 수행
- 대기열을 비우고 루트 셋을 비우는 작업에 따라 STW 시간은 크게 달라짐

4. Concurrent Cleanup

- 즉각적인 GC 영역으로 동시 마킹 작업 후 Live 객체가 없는 리전을 회수하는 단계

5. Concurrent Evacuation

- 컬렉션 셋을 다른 리전으로 `동시에` 이동(복사)하는 단계 (이것이 다른 GC와의 주요 차이점)
- 이 단계는 앱과 함께 다시 실행되어 여유 공간인 리전을 할당할 수 있음
- 선택한 컬렉션 셋의 크기에 따라 작업 시간은 크게 달라짐

Java에서 지원하는 GC 알고리즘

Shenandoah GC Steps - Summary

6. Init Update Refs (STW 중 제일 짧음)

- 이동한 객체들의 참조 업데이트를 초기화하고 모든 GC와 앱 스레드가 제거 작업을 완료했는 지 확인하며 다음 단계를 위해 GC를 준비하는 단계 (이외에는 거의 아무것도 하지 않음)
- 즉 이동 및 제거 등 작업이 완료된지 확인한 후에 다음 단계를 위해 이동한 객체에 대한 포인터 등을 업데이트 작업 등을 초기화(준비)

7. Concurrent Update References

- 힙을 살펴보고, `Concurrent Evacuation` 작업 중 이동된 객체에 대한 참조를 업데이트하는 단계
이것 또한 다른 GC와의 주요 차이점
- 해당 작업은 앱의 실행과 동시에 진행되며 힙의 객체 수에 따라 작업 시간이 달라짐 (하지만 객체 그래프에 영향을 받지 않음)

8. Final Update Refs (STW)

- 기존에 루트 셋을 업데이트하여 참조 업데이트 단계를 완료하는 단계
- 컬렉션 셋에서 리전을 재활용하며 루트 셋의 크기에 따라 STW 시간은 달라짐

9. Concurrent Cleanup

- 이 작업은 참조가 없는 컬렉션 셋 리전을 회수

Java에서 지원하는 GC 알고리즘

Shenandoah GC

- Heap Size
 - 다른 GC와 마찬가지로 힙 크기에 따라 성능이 달라짐
 - 동시 처리 단계에서 충분한 힙 공간이 있다면 더 좋은 성능을 발휘할 수 있어야 함
이 단계들은 LDS(Live Data Set)의 크기와 관련이 있음 (LDS가 클수록 힙크기도 커야 함)
- Pauses
 - 셰난도아의 STW는 주로 루트 셋 작업과 관련되어 있음
루트 셋에는 지역 변수, 생성된 코드에 참조, 내부 String, 클래스 로더 참조, JNI와 JVMTI 참조가 포함
 - 루트 셋이 클 경우 동시 작업의 경우가 아니라면 일반적으로 일시 중지 시간은 더 길 수밖에 없음
 - Final mark 단계의 부가적인 효과 (빈도 옵션 등으로 효율성을 높일 수 있음)
 - 필요한 경우 weak 참조 처리
 - 클래스 언로딩과 JDK 정리
- Throughput
 - 셰난도아는 동시 처리되는 GC이기 때문에 불변성 유지를 위한 GC 작업 사이클 동안에 일종의 배리어를 사용함
 - 이 배리어로 인해 처리량이 줄어 들 수 있음 (손실률은 약 최대 15% 이하)
실제로는 앱의 환경에 따라 성능이 달라지고 루트 셋과 weak 참조 등이 적은 경우 일시 중지는 밀리초 미만이 될 것

Java에서 지원하는 GC 알고리즘

Shenandoah GC vs G1 GC

- Pause Time
 - G1은 STW 시간을 최소화하기 위해 설계되었으나 힙의 크기에 영향을 받음
 - 세난도아는 최대한 힙의 크기와 무관하게 STW 시간 최소화를 목표로 함
- Memory Usage
 - G1은 세난도아에 비해 더 많은 CPU를 사용함으로 메모리 효율성을 더 극대화
대부분의 주요 작업이 STW 상태에서 수행되는데 그동안 멀티코어로 처리됨 (CPU 활용)
 - 세난도아는 짧은 일시 중지 시간을 위해 메모리 사용을 최대한 활용
주요 작업이 앱 실행과 병행되기 때문에 CPU 오버헤드가 G1 보다 낮을 수 있음
- Process
 - G1은 여전히 일부 과정에서 STW가 필요함
 - 세난도아는 GC 동작 중에도 앱의 실행이 중단되지 않도록 설계됨 (재배치 단계 등)

Java에서 지원하는 GC 알고리즘

ZGC

- 낮은 지연속도를 가진 GC(JDK 15부터 정식 기능)이며 대기 시간이 짧은 앱에 적합함
앱 스레드의 실행을 10ms 이상 중단하지 않고 고비용의 작업을 동시에 수행하기 때문
- 컬러 참조, 로드 배리어를 통해 스레드가 실행 시 동시 작업 수행 (힙 사용 추적에도 활용됨)
 - 참조 컬러는 ZGC의 핵심으로 객체의 상태를 표시하기 위해 일부 참조 비트를 사용함
- 8MB ~ 16TB 범위의 힙 크기를 처리하며 또한 힙, 라이브셋, 루트셋 크기 등은 일시 정지 시간에 영향을 주지 않음
- G1과 마찬가지로 힙 영역의 크기가 다를 수 있다는 점을 제외하고 힙을 분할함
- 활성화 옵션
 - ``java -XX:+UseZGC Application.java``
 - ``java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC Application.java`` (JDK 15 이하)

Java에서 지원하는 GC 알고리즘

ZGC Concepts

- STW를 가장 짧게 하는 것이 목표이며 처리 성능과 시간이 힙 크기와 무관하게 설계
일반적으로 빠른 응답시간이 요구되는 큰 힙을 가진 서버 앱에 적합
- 멀티 매핑
 - 가상 메모리와 물리 메모리 간 매핑
- 재배치
 - 메모리 단편화 > 메모리 할당이 어려움 (여유 공간의 크기와 필요 공간이 딱 맞지 않음)
 - 따라서 자주 컴팩션 작업이 일어나는데 그래서 GC들은 모두 재배치 하거나 아예 재배치 하지 않음
- GC
 - 도달 가능한 객체를 제외하고 제거

Java에서 지원하는 GC 알고리즘

ZGC Concepts

1. Marking

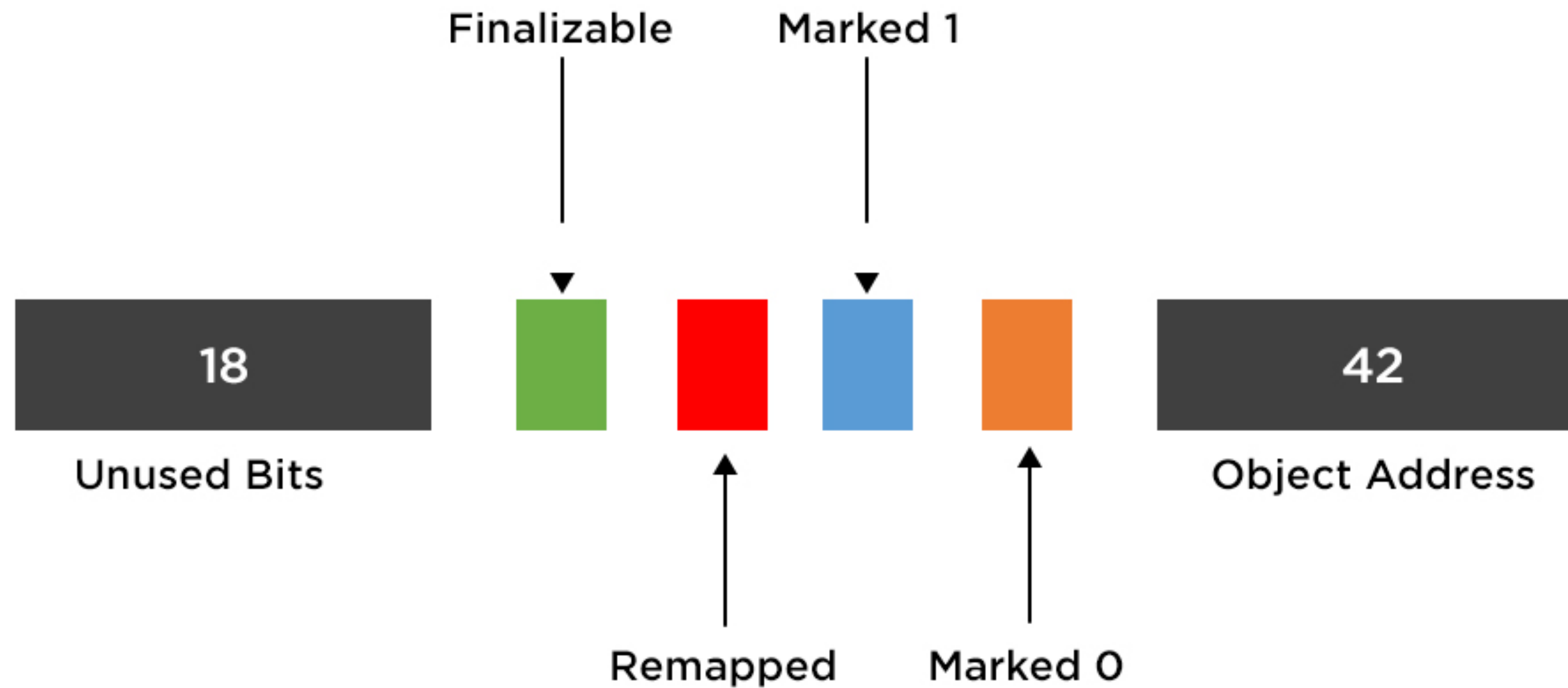
- ZGC는 참조 컬러를 통해 참조 상태를 참조 비트로 저장해서 참조 객체를 식별함
 - 참조 컬러(Reference Coloring 또는 Colored Pointer 등)라고 표현되며 메타데이터 비트를 활용
 - 이외에도 도달 가능한 객체를 식별하기 위해 다양한 방법이 있음
예를 들어 `Map<메모리주소, 객체>` 형태가 있으나 추가 메모리를 필요로함
- 하지만 이렇게 참조 비트를 설정하면 객체 위치에 대한 정보가 없기 때문에 여러 참조가 동일 객체를 가리킬 수 있음
 - 이를 `reference fixup` 메커니즘을 통해 처리 (로드 배리어의 리매핑)

2. Relocation

- 메모리 단편화로 인해 재배치가 필요한데 힙이 클수록 재배치가 느림
- 그래서 재배치를 앱 실행과 동시에 실행하지만 문제 발생
실행 중 재배치되는 객체를 참조하는 경우 (이전 위치 액세스)
- 이를 해결하기 위해 로드 배리어 활용
 - 스레드가 힙에서 객체 참조를 로딩할 때 실행되는 코드 조각 (JIT에 의해)
- 로드배리어는 참조의 메타데이터 비트를 확인, 참조를 얻기 전 일부 처리 수행
- 완전히 다른 참조를 생성하는 것을 리매핑이라고 함

Java에서 지원하는 GC 알고리즘

ZGC Concepts



<https://www.geeksforgeeks.org/z-garbage-collector-in-java/>

Java에서 지원하는 GC 알고리즘

ZGC Steps

1. Marking

- ZGC는 마킹 시 `marked0`, `marked1` 등과 같은 메타데이터 비트를 사용
 1. 루트 참조 객체(지역 변수, 스택 필드 등) 마킹 (STW)
 2. 루트로부터 참조되는 객체 마킹
로드배리어에서 감지한 마킹되지 않은 참조들도 마킹
 3. weak 참조 경우 등 처리 (STW)
Strong 참조 등이 더이상 없는 지 등을 확인해서 마킹 처리
 4. 미참조 클래스 언로딩 재배치 셋 선택 (압축 리전 표시)

2. Reference Coloring

- 참조란 가상 메모리의 바이트 위치를 표시한 것을 의미하는데 이 참조를 위해 반드시 모든 비트를 사용할 필요는 없음
그래서 비트 중 일부를 참조 속성을 나타내기 위해 사용 (참조 컬러)
- 32비트로 4기가 주소를 표현할 수 있는데 32비트로는 참조 컬러를 활용할 수 없음
즉 ZGC의 참조 컬러는 64비트 참조를 사용하기 때문에 64비트 플랫폼에서만 사용 가능
- 42비트를 사용해 주소를 표현, 따라서 4TB의 메모리 공간 처리 가능
- 그리고 메타데이터 비트라고 하는 참조 상태를 저장하는 4비트가 있음 (각각 1비트)
 - `finalizable` 비트 (소멸자를 통해서만 참조가 가능한 상태)
 - `remap` 비트 (최신 참조 상태, 현재 위치를 표현함)
 - `marked0`, `marked1` 비트 (도달 가능한 상태)

Java에서 지원하는 GC 알고리즘

ZGC Steps

3. Relocation

1. 블록을 찾으면서 재배치 셋에 넣으며 재배치 처리 (concurrent phase)
2. 재배치 셋의 모든 루트 참조를 재배치하고 해당 참조의 상태 업데이트 (STW)
3. 남아있는 재배치 셋의 참조를 모두 재배치, 이전/새 주소 매핑을 포워딩 테이블에 저장 (concurrent)
4. 재배치가 되었으나 마킹이 되지않은 참조는 다음 마킹 작업때 새 위치로 업데이트 됨
 - 마킹 작업 때 객체 그래프를 탐색하기 때문에 불필요한 객체 그래프 탐색을 줄일 수 있음
 - 또한 이를 로드 배리어가 처리할 수 있음

4. Remapping & Load Barriers

- Relocation 단계에서 재배치된 참조에 주소를 갱신하지 않아서 원하는 객체를 참조하지 못하거나 가비지에 접근하게 되는 상황이 생길 수 있음
- ZGC는 위 문제를 로드 배리어의 리매핑이라는 기술을 통해 해결
리매핑은 재배치된 객체 참조를 수정
- 앱에서 해당 참조된 객체를 로딩할 때 로드 배리어가 트리거 되어 올바른 참조를 반환
이를 통해 객체에 액세스할 때마다 최신 상태의 참조를 반환하도록 함
- 이로 인해 참조를 로딩할 때마다 성능 저하를 유발할 수 있으나 일시 중지 시간을 줄이기 위한 트레이드 오프

Java에서 지원하는 GC 알고리즘

ZGC Steps - Load Barrier & Remapping

- 앱이 참조를 로딩할 때 로드 배리어는 다음과 같은 절차를 통해 참조를 반환
 1. remap 비트 값이 `1`(최신)이라면 최신 값이기 때문에 안전하게 반환 가능
 2. 그 후 참조된 객체가 재배치 셋에 포함되어 있는 지 확인
포함되어 있지 않다면 이과정을 생략하기 위해 remap 비트 값을 1로 설정, 업데이트된 참조를 반환
 3. 위 단계를 지나오면 해당 참조가 재배치 대상이기 때문에 실제로 재배치가 이뤄졌는지 확인
재배치가 이뤄지지 않았다면 바로 재배치한 후 새 주소값을 저장할 포워드 테이블에 저장
 4. 로드 배리어의 단계들로 인해 해당 참조를 객체의 새 위치로 업데이트한 후에 remap 비트를 설정, 참조를 반환함

Java에서 지원하는 GC 알고리즘

ZGC vs Shenandoah GC

- 앱 실행과 병행되는 참조 이동을 처리하는 메커니즘
 - ZGC는 참조 컬러 (로드바리어-리매핑)
 - 셰난도아는 브룩스 포인터
- 설계 사상
 - ZGC는 STW의 최소화, 힙 용량이 큰 앱에서 더욱 중요
 - 셰난도아도 STW 시간의 최소화를 목표로 하지만 처리량과의 밸런스도 중요함

Java에서 지원하는 GC 알고리즘

Epsilon GC

- 메모리를 할당하지만 실제로 회수하지 않는 GC
사용 가능한 메모리를 모두 사용하면 프로그램이 종료됨
- 메모리 풋프린트와 처리량을 낮추면서 대기시간이 가장 낮은 형태로 최소한의 작업만 수행하는 데 초점을 맞추고 있음
- JVM에 수동 메모리 관리 등은 이 GC의 의도와 맞지 않음

Java에서 지원하는 GC 알고리즘

Epsilon GC Usage Case

- 성능(Performance) 테스트
 - GC의 영향을 받지 않고 성능 측면에서 테스트를 진행할 수 있음
- 메모리 부하(Memory pressure) 테스트
 - 코드 테스트의 경우 할당 메모리의 임계값을 설정하는 것은 메모리 부하를 검증하는데 유용함
- VM 인터페이스 테스트
 - VM 개발 시 GC 옵션 별 최솟값을 이해하기 쉬움
- 수명이 매우 짧은 잡 처리
- 최종 실패 레이턴시 개선
 - 레이턴시에 예민한 앱인 경우 GC 처리 주기로 인한 레이턴시를 격리하여 앱의 레이턴시 확인
 - JVM 리로딩할 때 GC 처리 주기의 영향을 받는 것은 앱 복구를 지연시키기도 함
- 최종 실패 처리량 개선
 - 다른 GC는 실제로 GC가 필요 없는 워크로드의 경우에도 GC 배리어 셋을 선택해야만 함
GC 배리어 셋을 선택한다는 것은 GC에 메모리를 할당해 부가적인 작업을 처리한다는 얘기이며
즉 GC에 리소스가 할당되기 때문에 최종 처리량을 측정하는 데 영향을 준다는 것을 의미

Java에서 지원하는 GC 알고리즘

* Selecting a Collector by Document

- ``-XX:+UseSerialGC``
 - 앱에 소규모의 데이터 셋(최대 100MB)을 핸들링하는 경우
 - 단일 프로세서에서 실행되며 앱의 STW 시간이 크게 상관 없는 경우
- ``-XX:+UseParallelGC``
 - 앱의 피크 성능이 가장 중요하면서 앱의 STW 허용 시간이 1초 이상인 경우
- ``-XX:+UseG1GC``
 - 앱의 응답 시간이 전체 처리량보다 중요하며 GC로 인한 STW 시간을 짧게 유지해야 하는 경우
- ``-XX:+UseZGC``
 - 앱의 응답 시간이 중요한 경우

Java에서 지원하는 GC 알고리즘

* Variables to Consider when choose GC

- Heap Size (OS로부터 할당받은 메모리)
 - 이론적으로 클수록 GC의 시간이 길어지며 작을수록 GC가 빈번하게 일어남
- Application Data Set Size (메모리에 올라갈 데이터의 크기)
 - young 제너레이션에 한 번에 많은 데이터가 로딩되는 것은 최대 힙크기, GC 시간에 영향을 줌
- Number of CPUs
 - GC에 따라 코어 수와 직접적으로 관계가 있음
- Pause Time
 - GC의 메모리 회수를 위해 앱이 잠시 멈추는 시간
- Throughput
 - GC 처리 시간에 비해 앱의 실행 시간이 길수록 앱의 처리량이 높아짐
- Memory Footprint (메모리 사이즈보다 조금 더 추상적인 표현)
 - GC에서 사용되는 메모리 풋프린트 (사용 가능한 메모리의 제한이나 많은 프로세스 등과 같은 환경)
- Promptness (객체가 비참조된 시점과 회수되는 시점의 사이 시간)
 - 이론적으로 힙 크기가 클수록 GC가 돌기까지 오래 걸리기 때문에 신속성이 낮음
- Java Version
 - 버전에 따른 디폴트 GC, 설정으로 시작하여 점차 맞춰가는 것을 권장
- Latency (지연 시간)
 - 앱의 레이턴시를 고려하여 GC의 일시 정지 시간과 조율이 필요함

Java에서 지원하는 GC 알고리즘

* Choose GC - Serial GC

- 싱글 코어(스레드)에서 추천되는 GC
 - 멀티코어 환경에선 이를 제외한 다른 GC를 선택하는 것이 효과적
- 장점
 - 스레드 간 통신 오버헤드가 없다면 비교적 효율적
 - 클라이언트 급 머신, 임베디드 시스템에 적합함
 - 데이터 셋이 작은 경우 적합함
 - 간혹 멀티 프로세서 환경에서도 데이터셋이 작은 경우(최대 100MB) 적합할 수 있음
- 단점
 - 대규모의 데이터셋을 핸들링하는 앱의 경우에는 비효율적
 - 멀티 프로세서 하드웨어를 활용할 수 없음

Java에서 지원하는 GC 알고리즘

* Choose GC - Parallel(Throughput) GC

- 배치 또는 오프라인 작업이나 비 대화형 웹 서버같은 경우에 적합
 - 더 많은 처리량이 요구되며 일시 중지 시간을 크게 신경쓰지 않는 경우
- 장점
 - 멀티 프로세스 하드웨어 활용 가능
 - Serial GC보다 더 큰 데이터셋 핸들링에 효율적
 - 전체적으로 높은 처리량
 - 메모리 풋프린트 최소화 시도
- 단점
 - STW가 비교적 깊
 - 힙 크기에 따라 잘 확장되지 않음

Java에서 지원하는 GC 알고리즘

* Choose GC - G1 GC

- CMS GC처럼 힙을 스캔, 제거
- 증분, 동시 수집 외에도 앱의 동작과 STW를 추적하여 예측 시도하며 대부분 수집될 객체가 있는 영역을 먼저 회수하는 데 초점을 둠
- 거래 플랫폼이나 대화형 GUI 등과 같이 실시간 앱에 적합함
일시 중지 시간이 중요하고 전체 처리량이 적당한 수준인 경우
- 장점
 - 기가 규모 데이터 셋에 매우 효율적
 - 멀티 프로세스 시스템을 최대한 활용
 - 일시 중지 시간을 줄이는 데 가장 효율적
- 단점
 - 목표한 처리량이 굉장히 중요할 때는 최선의 GC가 아닐 수 있음
 - 동시 수집되는 동안 앱이 GC와 리소스(CPU, 메모리 등)를 공유해야 함

Java에서 지원하는 GC 알고리즘

* Choose GC - ZGC

- 확장 가능하며 낮은 지연시간을 가진 GC
- 멀티 테라바이트 힙 규모에도 낮은 STW 시간을 유지함
- 참조 색상, 재배치, 로드 배리어, 리매핑 등의 기술을 활용
- 일반적으로 큰 힙을 사용하고 응답시간이 빨라야 하는 서버 앱의 적합함

실습

아하! 모먼트

- [내가] 생각하는 스터디의 핵심 요소와
선호하는 진행 방식은?

[내가] 생각하는 스터디의 핵심 요소와 선호하는 진행 방식은?

스터디란?

✓ 1. 사전적 의미

영어로 공부하다란 동사거나 서재, 학문이라는 의미의 명사.

1.(책경험 등을 통한) 공부, 학습, 학문 2.학업 3.공부하다, 배우다

미국·영국 ['stʌdi]

복수형 studies

동사형 3인칭 단수: 현재 studies, 과거 studied, 과거분사 studied, 현재분사 studying

파생형: 명사형 studier, 형용사형 studiable / studious

영어사전

스터디하다

(어떤 사람이 다른 사람과 특정한 내용을, 여러 사람이 특정한 내용을) 여럿이 모여 특정한 내용이나 분야를 공부하다.

형태 [+{영어}study-하_다]

활용 <여 불규칙Tip> 스터디하여 스터디해 스터디하니

고려대한국어대사전

[내가] 생각하는 스터디의 핵심 요소와 선호하는 진행 방식은?

스터디의 핵심 요소를 파악하기 위한 질문

- 여러명이 모여서 하는 것과 혼자 하는 것 이 둘은 어떤 차이가 있을까?
- 스터디 멤버는 어디서 어떻게 만나야 할까? 또 몇명이 적당할까?
- 스터디의 목적과 좋은 스터디 주제는 뭘까?
- 어떤 방식으로 진행해야 할까?
- 스터디에 시간을 얼마나 할애할 수 있을까?
- 멤버들의 동기부여를 위한 방법에는 뭐가 있을까?
- 멤버들 간 수준 차이는 어느 정도까지가 적당할까?
- 학습 방식과 구현 방식은 각각 어떤 장단점이 있을까?
- 리더가 필요할까?

[내가] 생각하는 스터디의 핵심 요소와 선호하는 진행 방식은?

스터디 방식

- 미리 준비해오기
 - 장점
 - 시간을 줄일 수 있어 당일에 좀 더 깊은 내용이나 다른 결의 주제를 대화할 수 있음
 - 발표 자료 등을 준비해온다면 더욱 퀄리티가 높은 스터디가 될 수 있음
 - 당일에 리딩할 사람을 뽑을 수 있음
 - 단점
 - 개인 사정으로 준비(스터디 참여)가 힘든 빈도가 너무 많아짐 (야근, 개인 경조사 등)
 - 팀원 성향에 따라 모임 일정 연기가 빈번하게 일어남
- 모임날 한 자리에서 함께 진행하기
 - 장점
 - 스터디를 준비해야 한다는 부담감이 없음
 - 궁금한 점들에 대해 바로바로 이야기해볼 수 있음
 - 단점
 - 모임날 시간을 많이 써야하고, 진도를 빠르게 나갈 수 없는 경우가 대부분
 - 깊은 내용까지 살펴보거나 어떻게 실무에 적용할 지 등을 고민해볼 여유가 없음
 - 각자의 진행 속도를 맞추기가 어려움

[내가] 생각하는 스터디의 핵심 요소와 선호하는 진행 방식은?

스터디 방식

- 북스터디
 - 장점
 - 제일 무난한 방식의 스터디
 - 준비 수준에 대한 편차가 거의 없음
 - 단점
 - 시간이 지날수록 동기부여가 떨어지는 경향이 있음
 - 이론만 습득하고 끝나는 경우가 많음
- 토이 프로젝트
 - 장점
 - 집중하기가 좋음, 준비 과정에서도 페어 프로그래밍, 코드 리뷰 등 추가적인 활동이 가능함
 - 실제 실무에 적용할만한 것들을 시도해보기 때문에 이해도가 높음
 - 단점
 - 할애하는 시간, 분량, 수준, 기여도 등을 맞추기가 어려움
 - 성향이 잘 맞지 않는 경우 오래가지 못함

참고 및 출처

- [https://en.wikipedia.org/wiki/Garbage collection \(computer science\)](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science))
- <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html>
- <https://docs.oracle.com/en/java/javase/17/gctuning/index.html>
- <https://docs.oracle.com/en/java/javase/17/gctuning/garbage-collector-implementation.html#GUID-23844E39-7499-400C-A579-032B68E53073>
- [https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM Troubleshooting/week1/lesson1.pdf](https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf)
- <https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html>
- <https://www.baeldung.com/java-gc-roots>
- <https://www.baeldung.com/java-gc-cyclic-references#tracing-gcs>
- <https://wiki.openjdk.org/display/HotSpot/Metaspace>
- <https://openjdk.org/jeps/387>
- <https://www.baeldung.com/java-permgen-metaspace>
- <https://www.geeksforgeeks.org/metaspaces-in-java-8-with-examples/>

- <https://docs.oracle.com/javase/8/embedded/develop-apps-platforms/codecache.htm>
- <https://bugs.openjdk.org/browse/JDK-8244660>
- <https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr007.html>
- <https://docs.oracle.com/en/java/javase/17/vm/native-memory-tracking.html>
- <https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html>
- <https://docs.oracle.com/en/java/javase/17/gctuning/parallel-collector1.html#GUID-DCDD6E46-0406-41D1-AB49-FB96A50EB9CE>
- <https://docs.oracle.com/en/java/javase/11/gctuning/concurrent-mark-sweep-cms-collector.html>
- <https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf>
- <https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html>
- https://assets.ctfassets.net/oxjq45e8ilak/709UsobBpBGHxaZ0z6MNvH/1d75677b26f1b7c9a71150c372645ad8/100746_367617808_Simone_Bordet_Concurrent_Garbage_collectors_ZGC_Shenandoah.pdf
- <https://www.baeldung.com/jvm-experimental-garbage-collectors>
- https://access.redhat.com/documentation/ko-kr/openjdk/8/html/using_shenandoah_garbage_collector_with_openjdk_8/shenandoah-gc-overview

- <https://wiki.openjdk.org/display/shenandoah/Main#Main-SupportOverview>
- <https://www.baeldung.com/jvm-zgc-garbage-collector>
- <https://www.baeldung.com/java-memory-management-interview-questions>
- <https://www.geeksforgeeks.org/z-garbage-collector-in-java/>
- <https://openjdk.org/jeps/318>
- <https://www.baeldung.com/jvm-epsilon-gc-garbage-collector>
- <https://blogs.oracle.com/javamagazine/post/understanding-the-jdks-new-superfast-garbage-collectors>
- <https://www.baeldung.com/java-choosing-gc-algorithm>
- <https://docs.oracle.com/en/java/javase/17/gctuning/available-collectors.html#GUID-414C9D95-297E-4EE3-B0D9-36F158A83393>