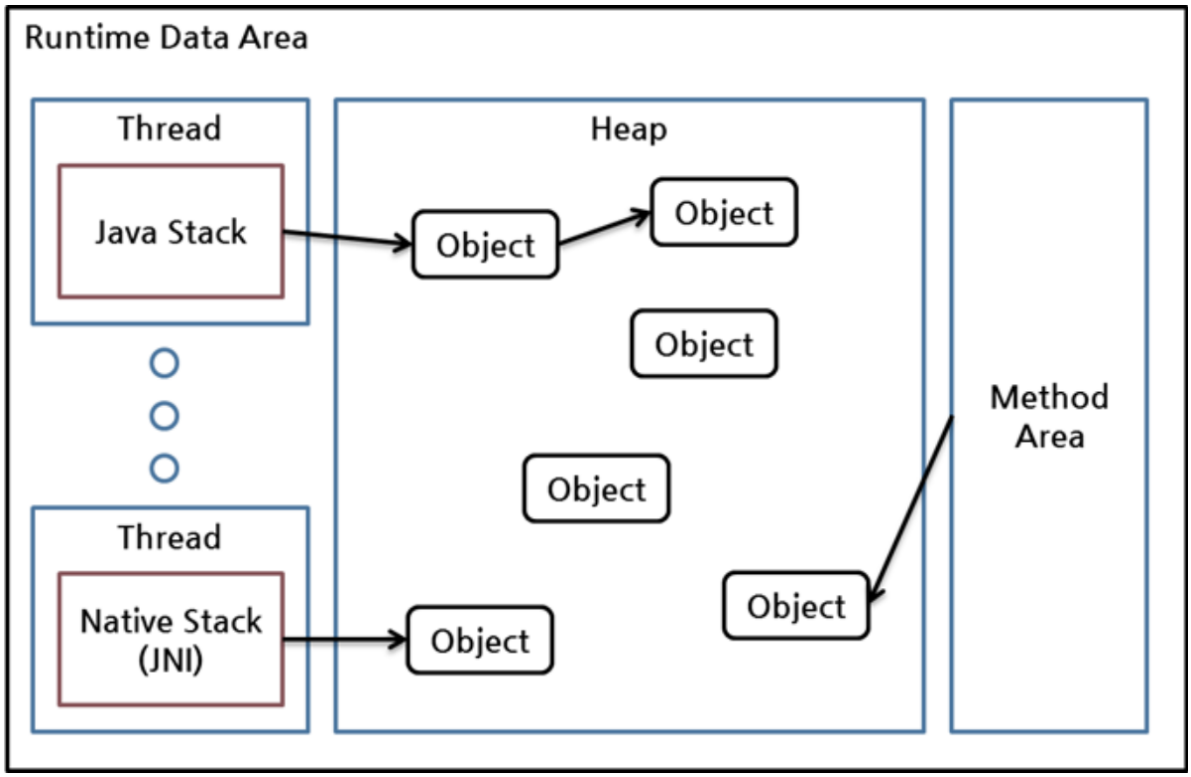
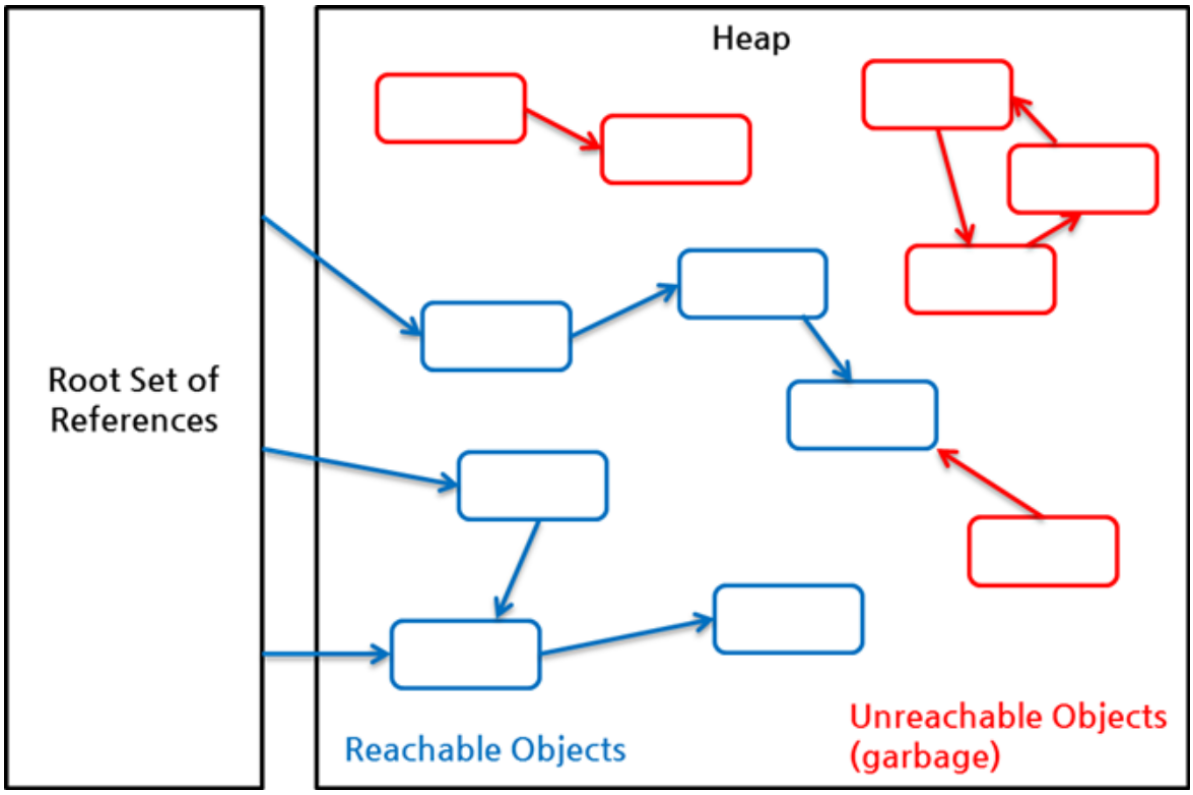
Garbage Collection

Reachability

어떤 객체에 유효한 참조가 있으면 'reachable'로, 없으면 'unreachable'로 구별하고, unreachable 객체를 가비지로 간주해 GC를 수행한다. 한 객체는 여러 다른 객체를 참조하고, 참조된 다른 객체들도 마찬가지로 또 다른 객체들을 참조할 수 있으므로 객체들은 참조 사슬을 이룬다. 이런 상황에서 유효한 참조 여부를 파악하려면 항상 유효한 최초의 참조가 있어야 하는데 이를 객체 참조의 root set이라고 한다.





java.lang.ref는 soft reference와 weak reference, phantom reference를 클래스 형태로 제공한다. 예를 들면, java.lang.ref.WeakReference 클래스는 참조 대상인 객체를 캡슐화(encapsulate)한 WeakReference 객체를 생성한다. 이렇게 생성된 WeakReference 객체는 다른 객체와 달리 Java GC가 특별하게 취급한다

- **strongly reachable:** root set으로부터 시작해서 어떤 reference object도 중간에 끼지 않은 상태로 참조 가능한 객체, 다시 말해, 객체까지 도달하는 여러 참조 사슬 중 reference object가 없는 사슬이 하나라도 있는 객체
- **softly reachable:** strongly reachable 객체가 아닌 객체 중에서 weak reference, phantom reference 없이 soft reference만 통과하는 참조 사슬이 하나라도 있는 객체

  softly reachable 객체, 즉 strong reachable이 아니면서 오직 SoftReferencce 객체로만 참조된 객체는 힙에 남아 있는 메모리의 크기와 해당 객체의 사용 빈도에 따라 GC 여부가 결정된다. 그래서 softly reachable 객체는 weakly reachable 객체와는 달리 GC가 동작할 때마다 회수되지 않으며 자주 사용될수록 더 오래 살아남게 된다. Oracle HotSpot VM에서는 softly reachable 객체의 GC를 조절하기 위해 다음 JVM 옵션을 제공한다.

  -XX:SoftRefLRUPolicyMSPerMB=<N>

  GC여부 결정
  (마지막 strong reference가 GC된 때로부터 지금까지의 시간) > (옵션 설정값 N) * (힙에 남아있는 메모리 크기)

  softly reachable 객체는 힙에 남아 있는 메모리가 많을수록 회수 가능성이 낮기 때문에, 다른 비즈니스 로직 객체들을 위해 어느 정도 비워두어야 할 힙 공간이 softly reachable 객체에 의해 일정 부분 점유된다. 따라서 전체 메모리 사용량이 높아지고 (Q:)GC가 더 자주 일어나며 GC에 걸리는 시간도 상대적으로 길어지는 문제가 있다.

- **weakly reachable:** strongly reachable 객체도 softly reachable 객체도 아닌 객체 중에서, phantom reference 없이 weak reference만 통과하는 참조 사슬이 하나라도 있는 객체

  GC가 동작할 때, unreachable 객체뿐만 아니라 weakly reachable 객체도 가비지 객체로 간주되어 메모리에서 회수된다. root set으로부터 시작된 참조 사슬에 포함되어 있음에도 불구하고 GC가 동작할 때 회수되므로, 참조는 가능하지만 반드시 항상 유효할 필요는 없는 LRU 캐시와 같은 임시 객체들을 저장하는 구조를 쉽게 만들 수 있다.

  GC가 동작하여 어떤 객체를 weakly reachable 객체로 판명하면, GC는 WeakReference 객체에 있는 weakly reachable 객체에 대한 참조를 null로 설정한다. 이에 따라 weakly reachable 객체는 unreachable 객체와 마찬가지 상태가 되고, 가비지로 판명된 다른 객체들과 함께 메모리 회수 대상이 된다.

  LRU 캐시와 같은 애플리케이션에서는 softly reachable 객체보다는 weakly reachable 객체가 유리하므로 LRU 캐시를 구현할 때에는 대체로 WeakReference를 사용한다.

- **phantomly reachable:** strongly reachable 객체, softly reachable 객체, weakly reachable 객체 모두 해당되지 않는 객체. 이 객체는 파이널라이즈(finalize)되었지만 아직 메모리가 회수되지 않은 상태이다.(Q: 해당 객체는 어떤 객체인가)

SoftReference와 WeakReference는 ReferenceQueue를 사용할 수도 있고 사용하지 않을 수도 있다. 이는 이들 클래스의 생성자 중에서 ReferenceQueue를 인자로 받는 생성자를 사용하느냐 아니냐로 결정한다. 그러나 PhantomReference는 반드시 ReferenceQueue를 사용해야만 한다. PhantomReference의 생성자는 단 하나이며 항상 ReferenceQueue를 인자로 받는다.
phantomly reachable로 판명된 객체에 대한 참조를 GC가 자동으로 null로 설정하지 않으므로, 후처리 작업 후에 사용자 코드에서 명시적으로 clear() 메서드를 실행하여 null로 설정해야 메모리 회수가 진행된다.

- **ReferenceQueue**

  SoftReference 객체나 WeakReference 객체가 참조하는 객체가 GC 대상이 되면 SoftReference 객체, WeakReference 객체 내의 참조는 null로 설정되고 SoftReference 객체, WeakReference 객체 자체는 ReferenceQueue에 enqueue된다. ReferenceQueue에 enqueue하는 작업은 GC에 의해 자동으로 수행된다.

  ReferenceQueue의 poll() 메서드나 remove() 메서드를 이용해 ReferenceQueue에 이들 reference object가 enqueue되었는지 확인하면 softly reachable 객체나 weakly reachable 객체가 GC되었는지를 파악할 수 있고, 이에 따라 관련된 리소스나 객체에 대한 후처리 작업을 할 수 있다.어떤 객체가 더 이상 필요 없게 되었을 때 관련된 후처리를 해야 하는 애플리케이션에서 이 ReferenceQueue를 유용하게 사용할 수 있다.

  Java Collections 클래스 중에서 간단한 캐시를 구현하는 용도로 자주 사용되는 WeakHashMap 클래스는 이 ReferenceQueue와 WeakReference를 사용하여 구현되어 있다.

- **unreachable:** root set으로부터 시작되는 참조 사슬로 참조되지 않는 객체

```
1. WeakReference<Sample> wr = new WeakReference<Sample>( new Sample());
2. Sample ex = wr.get();
...
3. ex = null;
```

GC가 실제로 언제 객체를 회수할지는 GC 알고리즘에 따라 모두 다르므로, GC가 수행될 때마다 반드시 메모리까지 회수된다고 보장하지는 않는다.(unreachable도 마찬가지)

1. soft references
2. weak references
3. 파이널라이즈 상태로 만든다.
4. phantom references
5. 메모리 회수

2번

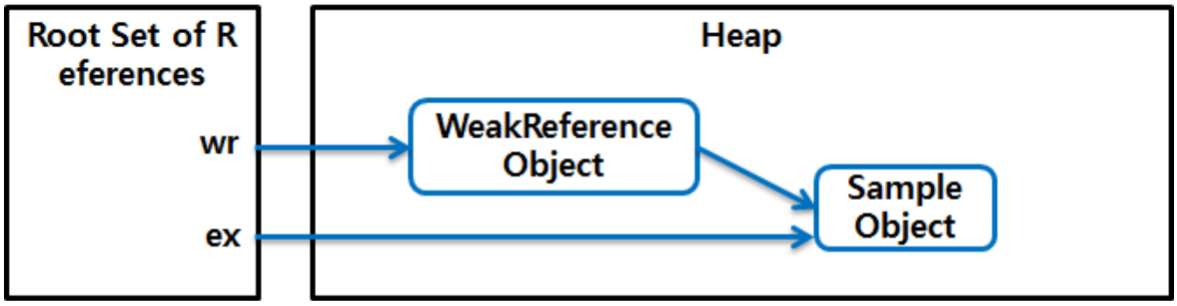


그림 3 Weak Reference 예 1

이 경우 ex 변수가 Sample Object를 가지기 때문에 **Strongly reachable**

3번

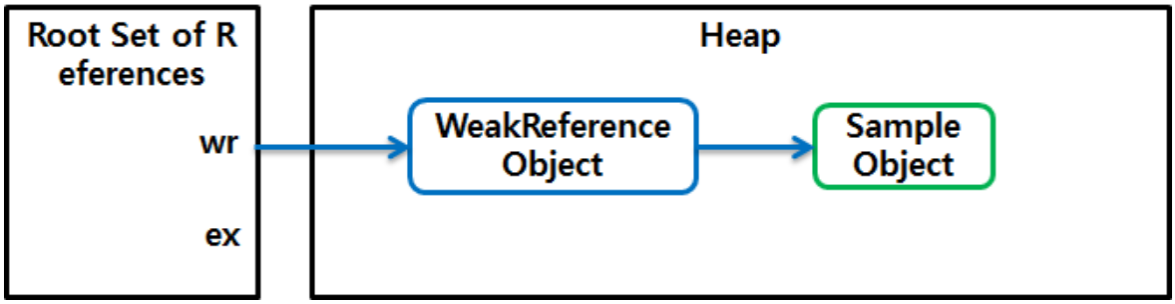


그림 4 Weak Reference 예 2

Java 스펙에서는 SoftReference, WeakReference, PhantomReference 3가지 클래스에 의해 생성된 객체를 "reference object"라고 부른다.

stop-the-world

stop-the-world란, GC을 실행하기 위해 JVM이 애플리케이션 실행을 멈추는 것이다. stop-the-world가 발생하면 GC를 실행하는 쓰레드를 제외한 나머지 쓰레드는 모두 작업을 멈춘다. GC 작업을 완료한 이후에야 중단했던 작업을 다시 시작한다. 어떤 GC 알고리즘을 사용하더라도 stop-the-world는 발생한다. 대개의 경우 GC 튜닝이란 이 stop-the-world 시간을 줄이는 것이다.

가끔 명시적으로 해제하려고 해당 객체를 null로 지정하거나 System.gc() 메서드를 호출하는 개발자가 있다. null로 지정하는 것은 큰 문제가 안 되지만, System.gc() 메서드를 호출하는 것은 시스템의 성능에 매우 큰 영향을 끼치므로 System.gc() 메서드는 절대로 사용하면 안 된다.

GC는 두가지 전제 조건하에 발생한다.

- 대부분의 객체는 금방 접근 불가능 상태(unreachable)가 된다.
- Old 객체에서 Young 객체로의 참조는 아주 적게 존재한다.
    - **Young 영역(Yong Generation 영역):** 새롭게 생성한 객체의 대부분이 여기에 위치한다. 대부분의 객체가 금방 접근 불가능 상태가 되기 때문에 매우 많은 객체가 Young 영역에 생성되었다가 사라진다. 이 영역에서 객체가 사라질때 Minor GC가 발생한다고 말한다.
    - **Old 영역(Old Generation 영역):** 접근 불가능 상태로 되지 않아 Young 영역에서 살아남은 객체가 여기로 복사된다. 대부분 Young 영역보다 크게 할당하며, 크기가 큰 만큼 Young 영역보다 GC는 적게 발생한다. 이 영역에서 객체가 사라질 때 Major GC(혹은 Full GC)가 발생한다고 말한다.
        - Old 영역에 있는 객체가 Young 영역의 객체를 참조하는 경우를 처리하기 위해서 Old 영역에는 512바이트의 덩어리(chunk)로 되어 있는 카드 테이블(card table)이 존재한다.

카드 테이블에는 Old 영역에 있는 객체가 Young 영역의 객체를 참조할 때마다 정보가 표시된다. Young 영역의 GC를 실행할 때에는 Old 영역에 있는 모든 객체의 참조를 확인하지 않고, 이 카드 테이블만 뒤져서 GC 대상인지 식별한다.

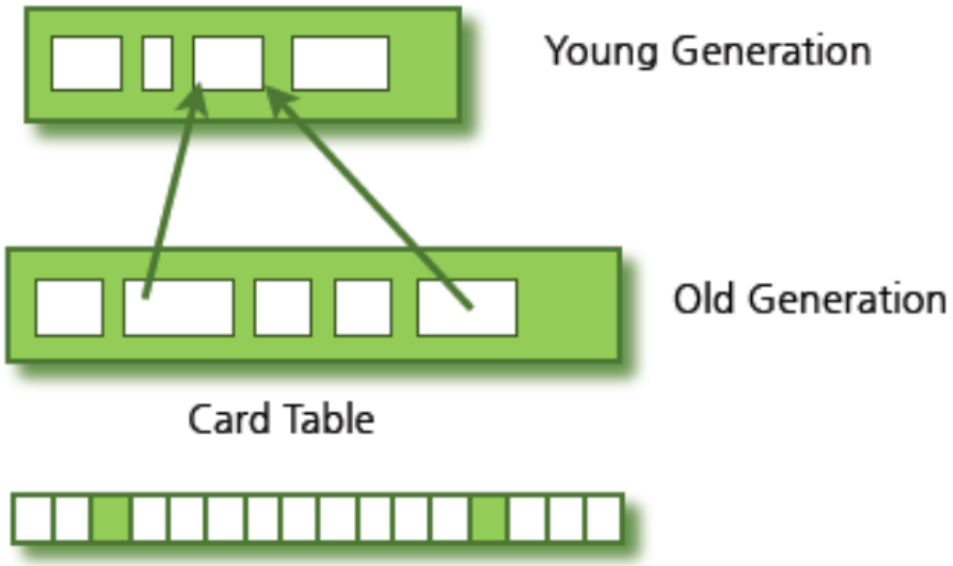


그림 2 카드 테이블 구조

카드 테이블은 write barrier를 사용하여 관리한다. write barrier는 Minor GC를 빠르게 할 수 있도록 하는 장치이다. write barrirer때문에 약간의 오버헤드는 발생하지만 전반적인 GC 시간은 줄어들게 된다.(**Q:** 왜 write barrier가 필요한지, card table은 언제 채우는지)

- **Permanent Generation 영역(Perm 영역):** Method Area라고도 한다. 객체나 억류(intern)된 문자열 정보를 저장하는 곳이며, Old 영역에서 살아남은 객체가 영원히 남아 있는 곳은 절대 아니다. 이 영역에서 GC가 발생할 수도 있는데, 여기서 GC가 발생해도 Major GC의 횟수에 포함된다.

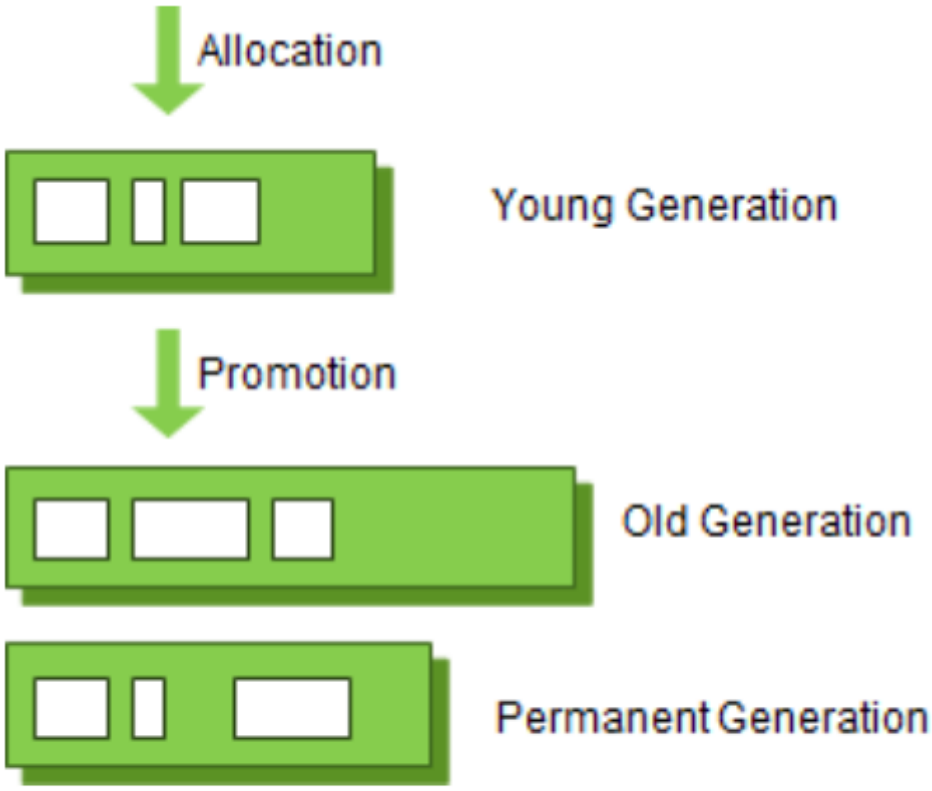


그림 1 GC 영역 및 데이터 흐름도

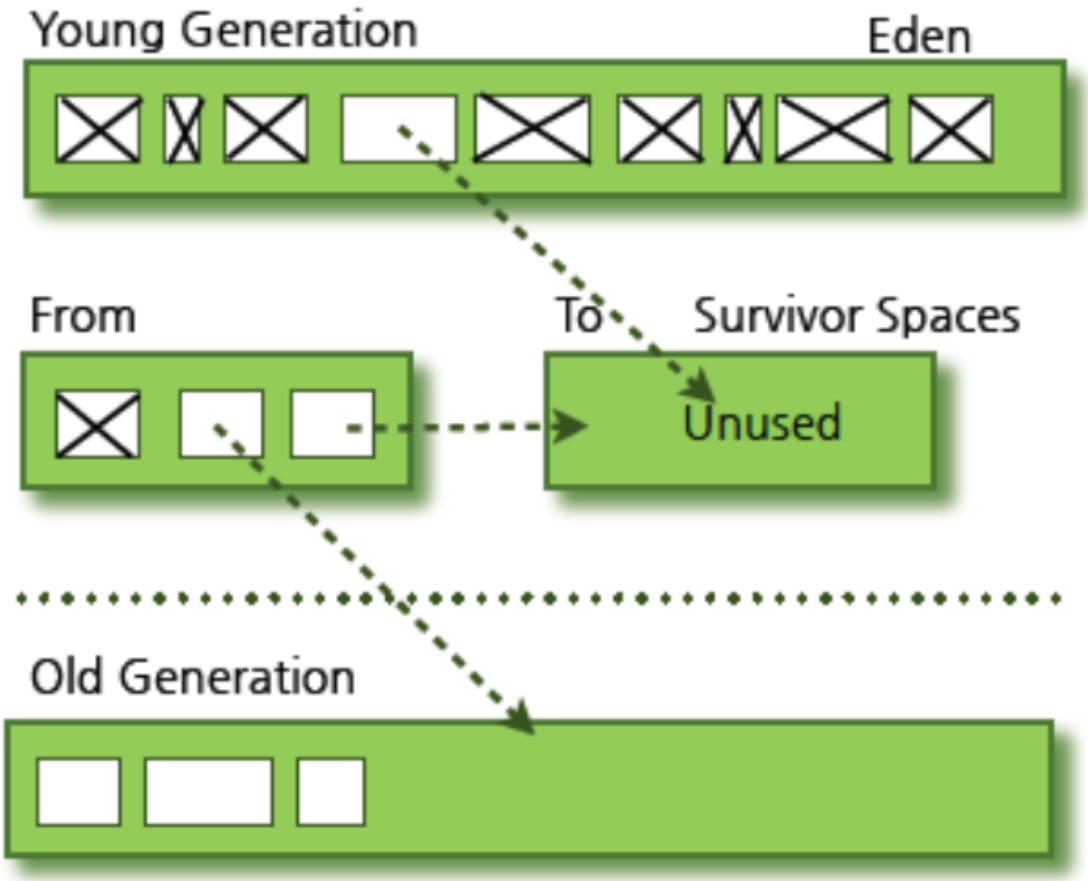
Young 영역

Young 영역은 3개의 영역으로 나뉜다.

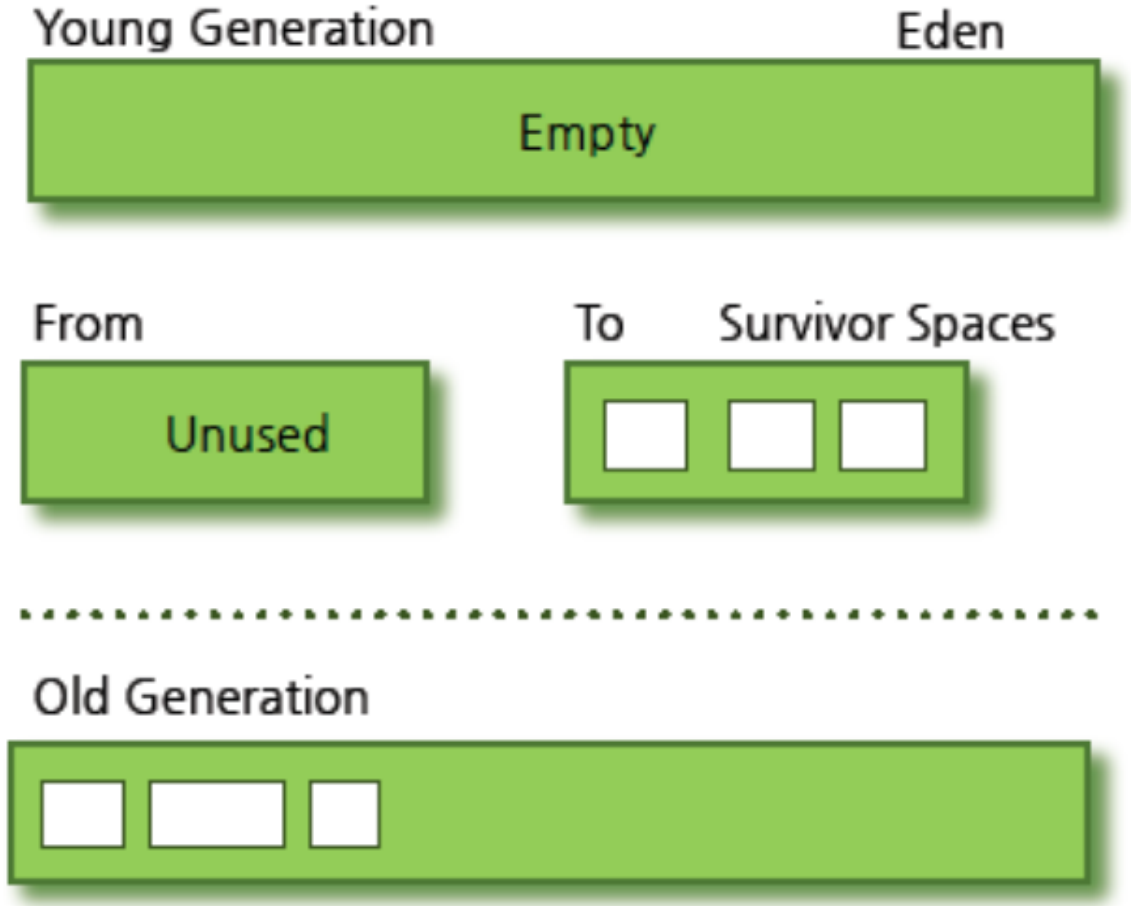
- Eden 영역
- Survivor 영역(2개)

GC 처리

1. 새로 생성한 대부분의 객체는 Eden 영역에 위치한다.
2. Eden 영역에서 GC가 한 번 발생한 후 살아남은 객체는 Survivor 영역 중 하나로 이동된다.
3. Eden 영역에서 GC가 발생하면 이미 살아남은 객체가 존재하는 Survivor 영역으로 객체가 계속 쌓인다.
4. 하나의 Survivor 영역이 가득 차게 되면 그 중에서 살아남은 객체를 다른 Survivor 영역으로 이동한다. 그리고 가득 찬 Survivor 영역은 아무 데이터도 없는 상태로 된다.
5. 이 과정을 반복하다가 계속해서 살아남아 있는 객체는 Old 영역으로 이동하게 된다.



From쪽 servivor space가 가득 찼기 때문에 데이터를 Old 영역과 다른 servivor space로 이동한다.



HotSpot VM에서는 보다 빠른 메모리 할당을 위해서 두 가지 기술을 사용한다. 하나는 **bump-the-pointer**라는 기술이며, 다른 하나는 **TLABs(Thread-Local Allocation Buffers)**라는 기술이다.

bump-the-pointer는 Eden 영역에 할당된 마지막 객체를 추적한다. 마지막 객체는 Eden 영역의 맨 위(top)에 있다. 그리고 그 다음에 생성되는 객체가 있으면, 해당 객체의 크기가 Eden 영역에 넣기 적당한지만 확인한다. 만약 해당 객체의 크기가 적당하다고 판정되면 Eden 영역에 넣게 되고, 새로 생성된 객체가 맨 위에 있게 된다. 따라서, 새로운 객체를 생성할 때 마지막에 추가된 객체만 점검하면 되므로 매우 빠르게 메모리 할당이 이루어진다.

멀티 스레드 환경을 고려하면 이야기가 달라진다. Thread-Safe하기 위해서 만약 여러 스레드에서 사용하는 객체를 Eden 영역에 저장하려면 락(lock)이 발생할 수 밖에 없고, lock-contention 때문에 성능은 매우 떨어지게 될 것이다. HotSpot VM에서 이를 해결한 것이 **TLABs**이다.

각각의 스레드가 각각의 몫에 해당하는 Eden 영역의 작은 덩어리를 가질 수 있도록 하는 것이다. 각 쓰레드에는 자기가 갖고 있는 TLAB에만 접근할 수 있기 때문에, bump-the-pointer라는 기술을 사용하더라도 아무런 락이 없이 메모리 할당이 가능하다.

Old 영역에 대한 GC 알고리즘

- ## Serial GC (-XX:+UseSerialGC)
    - Young 영역에서의 GC는 위에 설명한 방식을 사용한다. Old 영역의 GC는 mark-sweep-compact이라는 알고리즘을 사용한다.
        - **Mark**
            - Old 영역에 살아 있는 객체를 식별
        - **Sweep**
            - 그 다음에는 힙(heap)의 앞 부분부터 확인하여 살아 있는 것만 남김
        - **Compaction**
            - 각 객체들이 연속되게 쌓이도록 힙의 가장 앞 부분부터 채워서 객체가 존재하는 부분과 객체가 없는 부분으로 나눔

      Serial GC는 데스크톱의 CPU 코어가 하나만 있을 때 사용하기 위해서 만든 방식이다. Serial GC를 사용하면 애플리케이션의 성능이 많이 떨어진다.
- ## Parallel GC
    - Parallel GC는 Serial GC와 기본적인 알고리즘은 같다. 그러나 Serial GC는 GC를 처리하는 스레드가 하나인 것에 비해, Parallel GC는 GC를 처리하는 쓰레드가 여러 개이다. 그렇기 때문에 Serial GC보다 빠른게 객체를 처리할 수 있다. Parallel GC는 메모리가 충분하고 코어의 개수가 많을 때 유리하다. Parallel GC는 Throughput GC라고도 부른다.

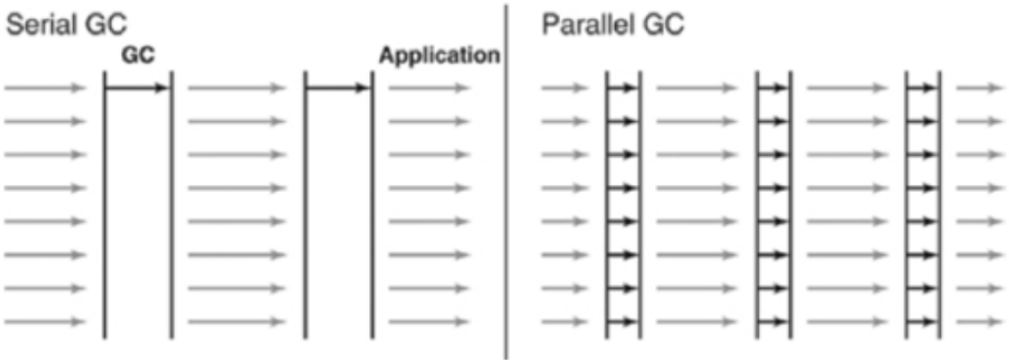


그림 4 Serial GC와 Parallel GC의 차이 (이미지 출처: "Java Performance", p. 86)

- ## Parallel Old GC(Parallel Compacting GC)
    - Parallel Old GC는 JDK 5 update 6부터 제공한 GC 방식이다. 앞서 설명한 Parallel GC와 비교하여 Old 영역의 GC 알고리즘만 다르다. 이 방식은 Mark-Summary-Compaction 단계를 거친다. Summary 단계는 앞서 GC를 수행한 영역에 대해서 별도로 살아 있는 객체를 식별한다는 점에서 Mark-Sweep-Compaction 알고리즘의 Sweep 단계와 다르며, 약간 더 복잡한 단계를 거친다.(MORE!!)
- ## Concurrent Mark & Sweep GC(이하 CMS)
    - 복잡함

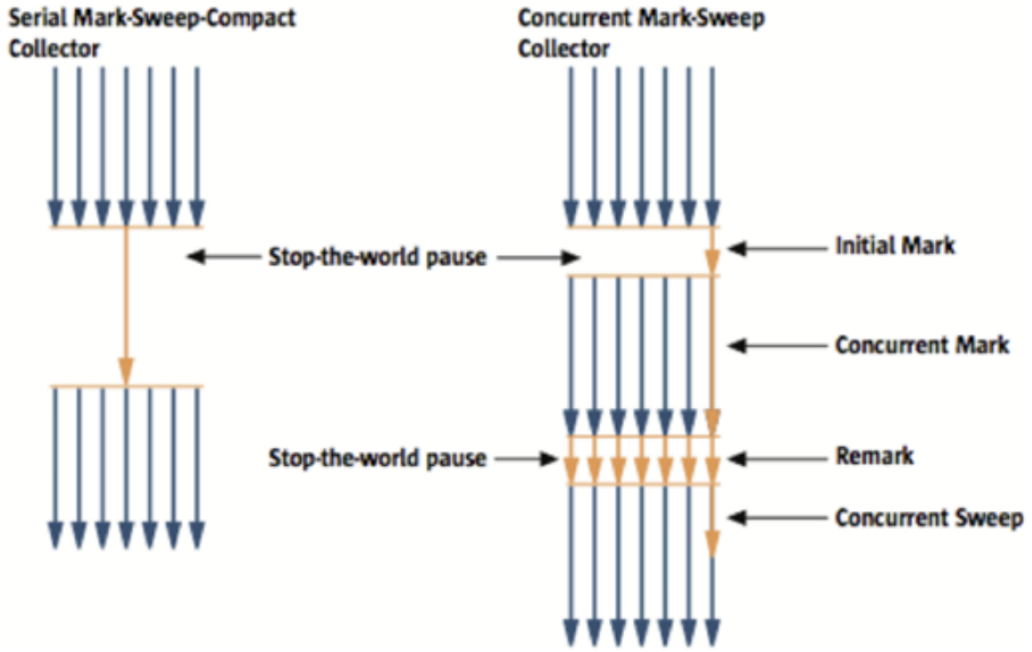


그림 5 Serial GC와 CMS GC(이미지 출처)

    - **Initial Mark**
        - 클래스 로더에서 가장 가까운 객체 중 살아 있는 객체만 찾는 것으로 끝낸다. 따라서, 멈추는 시간은 매우 짧다.
    - **Concurrent Mark**
        - 방금 살아있다고 확인한 객체에서 참조하고 있는 객체들을 따라가면서 확인한다. 이 단계의 특징은 다른 스레드가 실행 중인 상태에서 동시에 진행된다는 것이다.
    - **Remark**
        - Concurrent Mark 단계에서 새로 추가되거나 참조가 끊긴 객체를 확인한다. (**Q:** 굳이 왜 할까?)
    - **Concurrent Sweep**
        - 쓰레기를 정리하는 작업을 실행한다. 이 작업도 다른 스레드가 실행되고 있는 상황에서 진행한다.

이러한 단계로 진행되는 GC 방식이기 때문에 stop-the-world 시간이 매우 짧다. 모든 애플리케이션의 응답 속도가 매우 중요할 때 CMS GC를 사용하며, Low Latency GC라고도 부른다.

단점

- 다른 GC 방식보다 메모리와 CPU를 더 많이 사용한다.
- Compaction 단계가 기본적으로 제공되지 않는다.

메모리의 데이터 파편화가 많이 일어나 Compaction 작업을 실행하면 다른 GC 방식의 stop-the-world 시간보다 stop-the-world 시간이 더 길기 때문에 **Compaction 작업이 얼마나 자주, 오랫동안 수행되는지 확인해야 한다.**

- **G1(Garbage First) GC(MORE!)**
  - Young의 세가지 영역에서 데이터가 Old 영역으로 이동하는 단계가 사라진 GC 방식

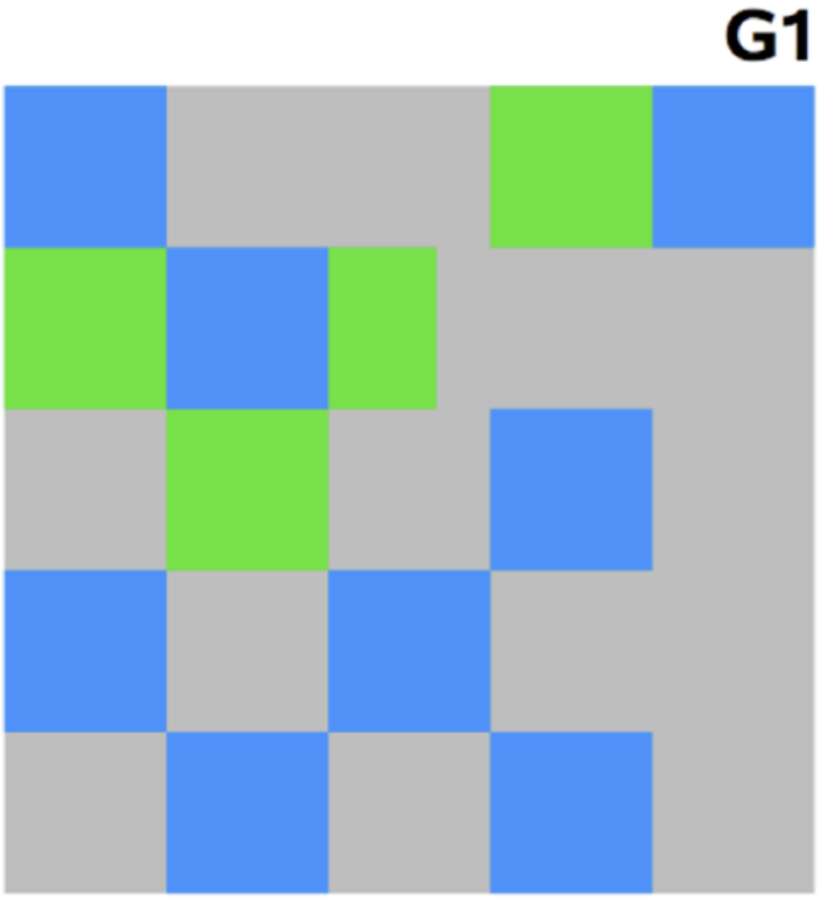


그림 6 G1 GC의 레이아웃(이미지 출처: "The Garbage-First Garbage Collector" (TS-5419), JavaOne 2008, p. 19)

지금까지의 Young 영역과 Old 영역이랑 다른 형태

참조

https://d2.naver.com/helloworld/329631

https://d2.naver.com/helloworld/1329

좀 더 공부할 내용(우선순위)

GC and Statement Pool (https://d2.naver.com/helloworld/4717)

튜닝 (https://d2.naver.com/helloworld/37111)