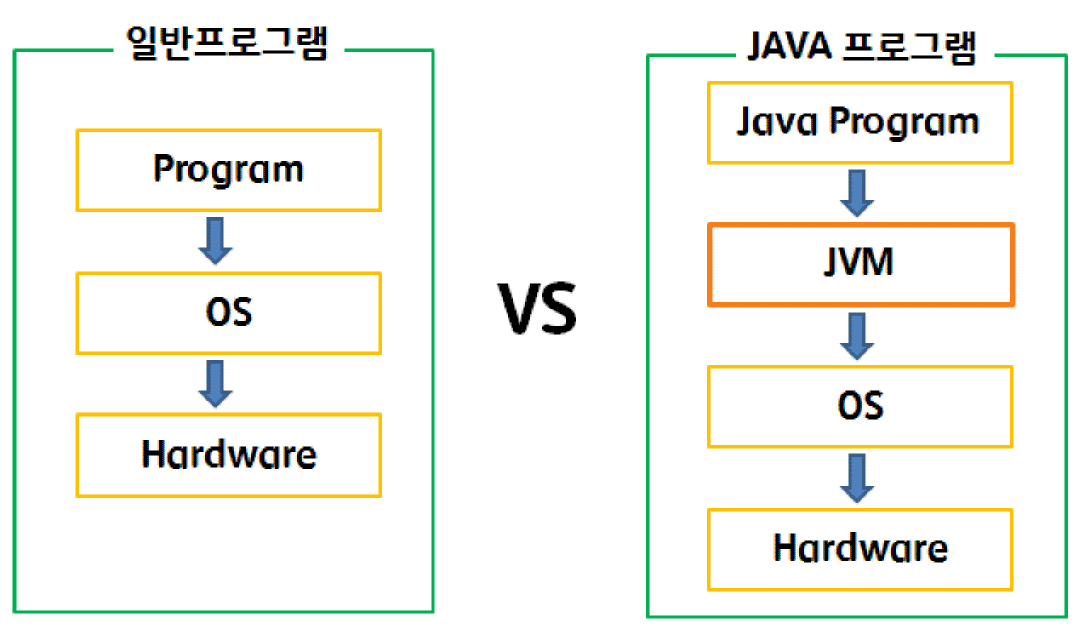
**1. JVM 메모리 구조**

**1.1 JVM 이란?**

-JVM은 하나의 개념, 스팩에 지나지 않는다. 즉 어느 누구도 정확하고 자세한 설계도를 제공하지 않고 단지 표준화된 정의만 존재할 뿐이다.

🡺이러한 표준화된 정의가 나오면서 각 JVM 벤더들(Oracle, IBM 등)은 표준에 맞도록 자신들의 JVM 별도 구현

🡺Java 프로그램의 범주에 들어가는 모든 것을 실행시키는 JVM이라 칭한다.



**-장점 :** 운영체제 독립적이다. JVM만 설치하면 어디에서든 구동가능 하다.

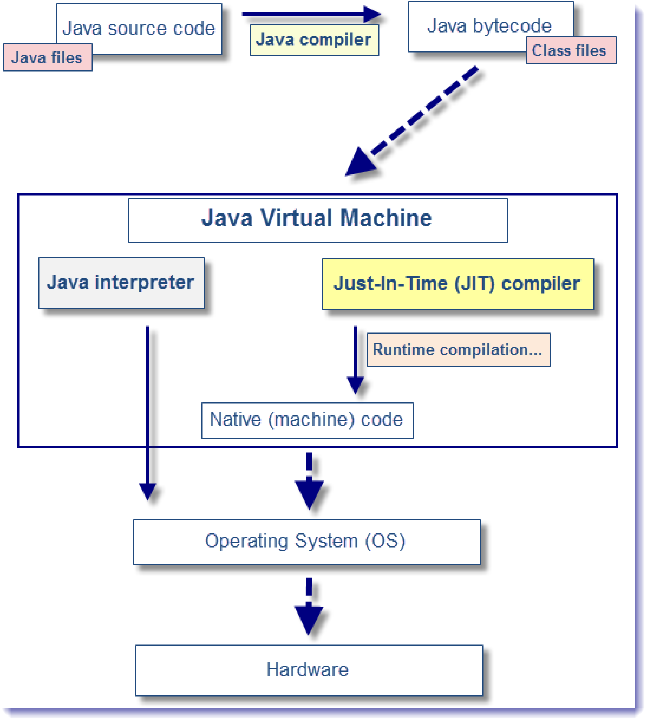
**-단점 :** 속도가 느리다. 🡺 jit를 통해 속도가 개선되었다.

**jit(just-in-time) 란?**

-Java HotSpot VM - Update 를 통해 J2SE 1.2 에서 최초 등장했으며, J2SE 1.3 이후 기본으로 탑재

-실행 시점에서 기계어 코드를 생성하면서 그 코드를 캐싱하여, 같은 함수가 여러 번 불릴 때

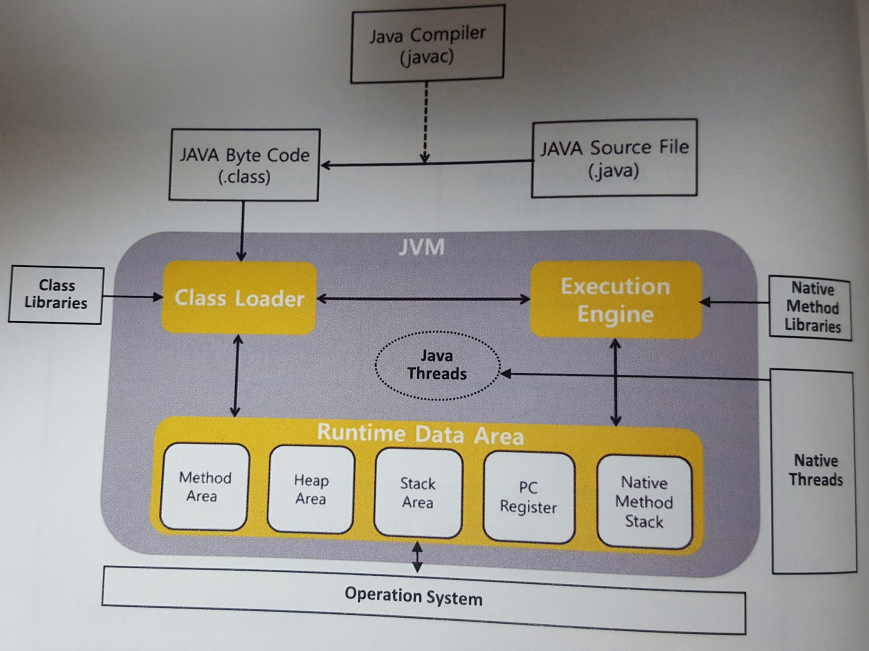
매번 기계어 코드를 생성하는 것을 방지한다.



ex) Java, .Net 등에서 사용

-IBM JVM은 JIT 컴파일러뿐만 아니라 IBM JDK 6부터 AOT(Ahead-Of-Time) 컴파일러라는 기능을 도입했다. 이는 한번 컴파일된 네이티브 코드를 여러 JVM이 공유 캐시를 통해 공유해서 사용하는 것을 의미한다.

**JVM 구조**



-Method Area : 클래스, 변수, Method, static 변수, 상수 정보 (모든 Thread 공유)

-Heap Area : 인스턴스 저장 되는 구역 (GC 대상, 모든 Thread 공유)

-Stack Area : Method 내에 매개변수, 지역변수, 리턴값 등 (각 Thread 별로 하나씩 생성)

-PC Register : 현재 수행중인 JVM 명령 주소값. Native Method 일 경우 Undefined 출력

-Native Method Stack : C, C++ 등의 메소드 호출 (JNI 사용)

**1.2 Java Heap**

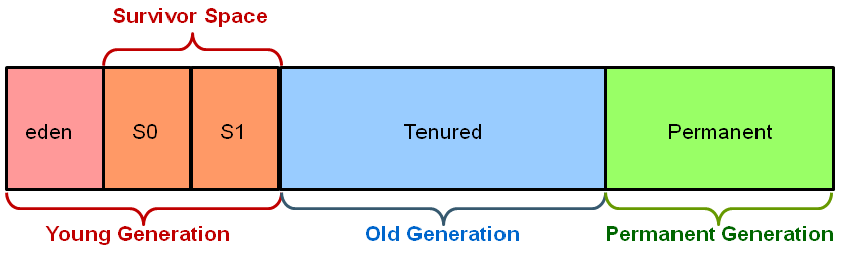
-Heap은 JVM 메모리 구조 중 하나일 뿐이다.

-Java Heap은 단지 Instance와 Array 객체 두가지 종류만 저장되는 공간 일뿐

-C, C++과 다르게 메모리 해제를 위한 Java Code나 Bytecode도 존재하지 않는다.

-Heap과 GC는 전적으로 벤더들에게 위임하고 있다.

**1) Hotspot JVM Heap 구조**



**Young Generation (Minor GC)**

-eden : Object가 Heap에 최초로 할당되는 장소

-Survivor : eden 영역이 꽉 차게 되면 Object 참조 여부를 따져서 참조 중이면 Survivor 영역으로 넘어가면서 eden 영역을 청소 한다.

**Old Generation (Major GC)**

-tenured : Young Generation에서도 계속 참조 중인 Object가 있으면 Old Generation으로 이동하게 된다.

**Permanent Generation**

-Class Meta, Method Meta, Static 변수 상수 등 저장되는 공간.

-JAVA8 부터는 Metaspace로 변경되었다.

**Java7, Java8 JVM 구조 비교**

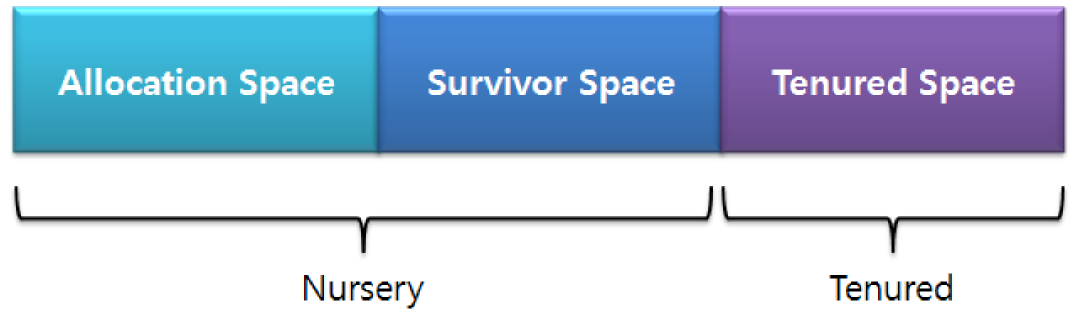


**OutOfMemoryError : PermGen space**



-JAVA8 부터는 PermGen 영역이 사라짐에 따라서 OutOfMemoryError : PermGen space는 못보게 되었다.

**2) IBM JVM의 Heap 구조**



-JAVA5 부터 Hotspot 처럼 Generational Heap을 사용할수 있게 되었다.

**Nursery Generation (Young Generation)**

-Allocate (eden) : Object가 최초로 할당되는 곳

-Survivor : Allocate 영역이 꽉 차면 대피시키는 영역이다.

**Tenured (Old Generation)**

-tenured : Nursery 영역의 객체가 계속 참조되면 이동 하는 곳이다.

**IBM JVM vs Oracle JVM** 참고) <https://www.slideshare.net/jwlee98/ibm-jvm-juwlee-201707v1>

**2. Garbage Collection**

**2.1 GC 소개**

**1) GC 개요**

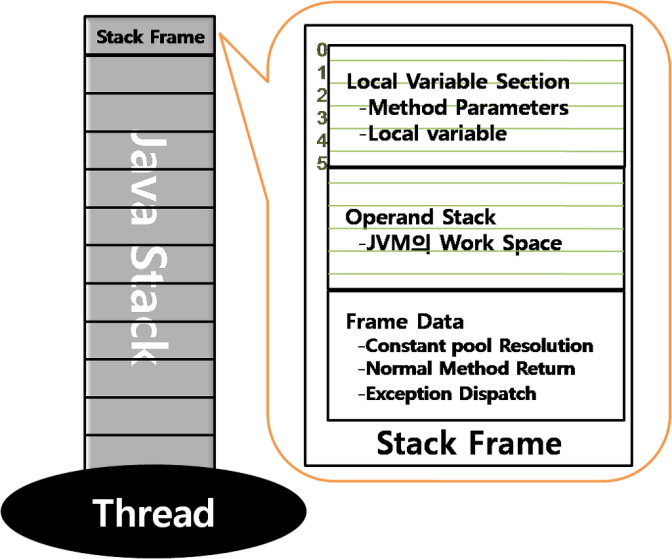
-GC의 기본적인 동작 방식.

**2) GC로 이한 문제점**

-GC는 자동화 메커니즘으로 인해 직접 핸들링 할 필요가 없게 되었고 잘못된 메모리 접근으로 인한 Crash 현상의 소지도 없어 지게 되었다.

-그러나 명시적인 Memory 해제 보다 느리며 GC 순간 발생하는 Suspend Time으로 인해 다양한 문제를 야기 시킨다.

**Stack**



**-Local Variable Section :** 메소드 파라미터, 지역변수

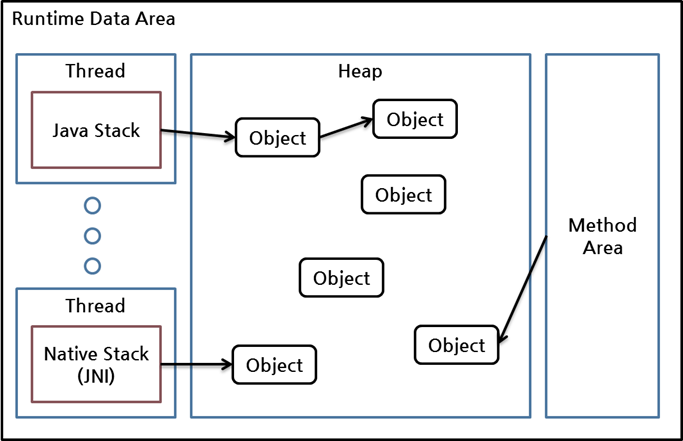
🡺primitive type(원시타입)은 크기를 결정할 수 있으며 Object, Array, String 같은 객체는 reference를 활용해 Heap 메모리를 참조하도록 할 수 있다.

**-Operand Stack :** 연산을 위해 사용되는 데이터 및 결과값 처리 영역

**-Frame Data :** 상수, 정상 종료시 Return, 비정상 종료시 처리할 예외

**3) Root Set과 Garbage**

**힙에 있는 개체들에 대한 참조**



1)Java 스택, 즉 Java 메서드 실행 시에 사용하는 지역 변수와 파라미터들에 의한 참조

2)메서드 영역의 정적 변수에 의한 참조

3)네이티브 스택, 즉 JNI(Java Native Interface)에 의해 생성된 객체에 대한 참조

4)힙 내의 다른 객체에 의한 참조

이들 중 힙 내의 다른 객체에 의한 참조를 제외한 나머지 3개가 **root set**으로, **reachability**를

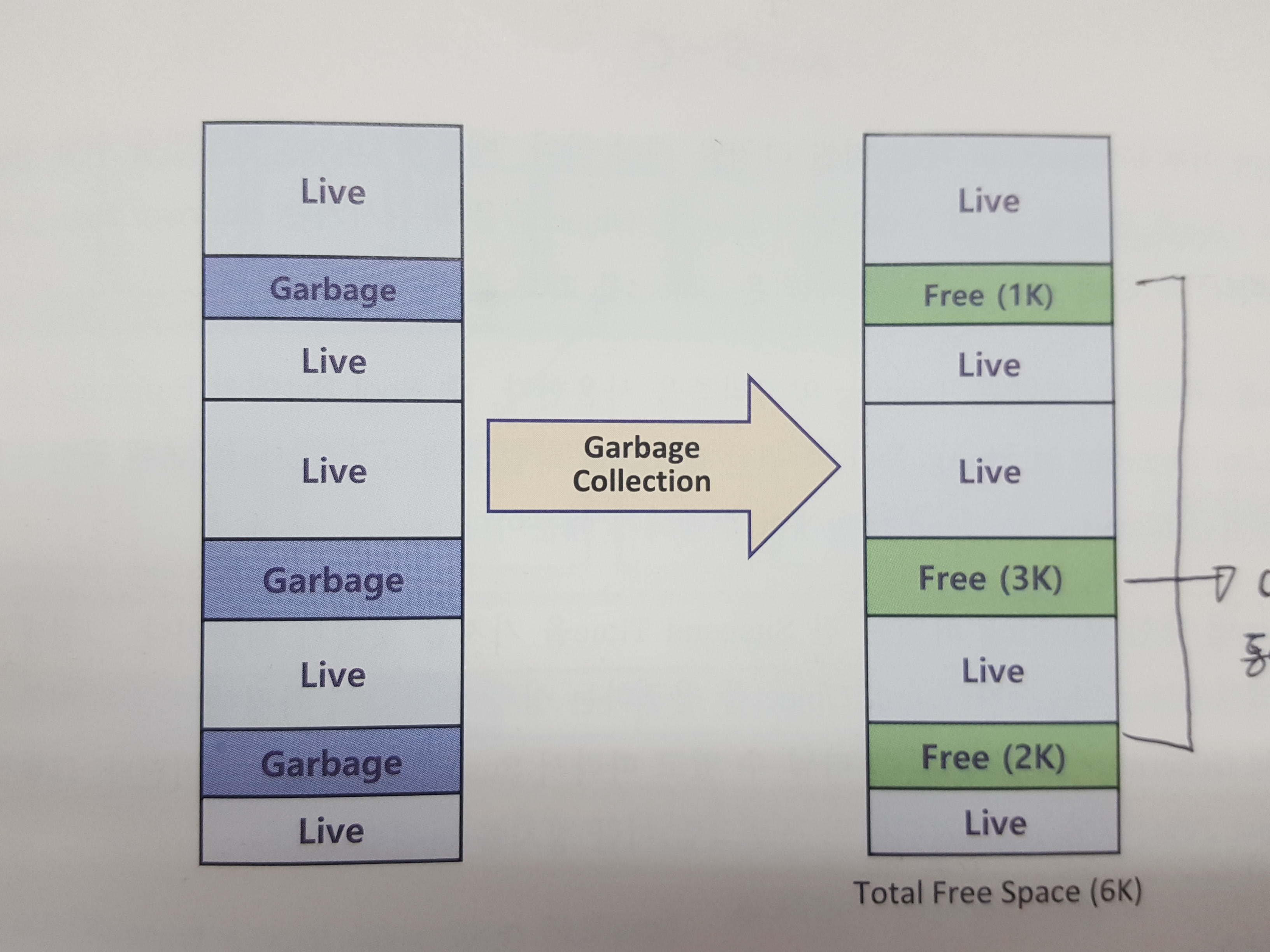
판가름하는 기준이 된다.

**참고 :** <http://d2.naver.com/helloworld/329631>

**4) Garbage Collection 목적**

-Garbage Collection은 새로운 Object의 할당을 위해 한정된 Heap 공간을 재활용 하려는 목적으로 수행

-단편화를 방지하기 위해 Compaction 같은 다양한 알고리즘 사용



**2.2 Hotsport JVM의 Garbage Collection**

**1) 개요**

-Hotspot JVM은 기본적으로 Generational Collection 방식을 사용 (Young, Old)

**CG 메커니즘은 두가지 가설을 두고 있다. (책에서)**

-첫번째 가설 : 대부분의 Object는 생성된 후 금방 Garbage가 된다.

-두번째 가설 : Old Object에서 Young Object를 참조할 일은 드물다.

**첫번째 가설**

-Sweep 작업 (Mark 되지 않은 Object 제거)을 수행 후, 단편화가 발생되면 Compaction을 수행 해야 되는데 비싼 작업 이므로, Eden Area에서 참조 되고 있는 Object를 Survivor Area로 옮기는 작업을 하는 것이다.

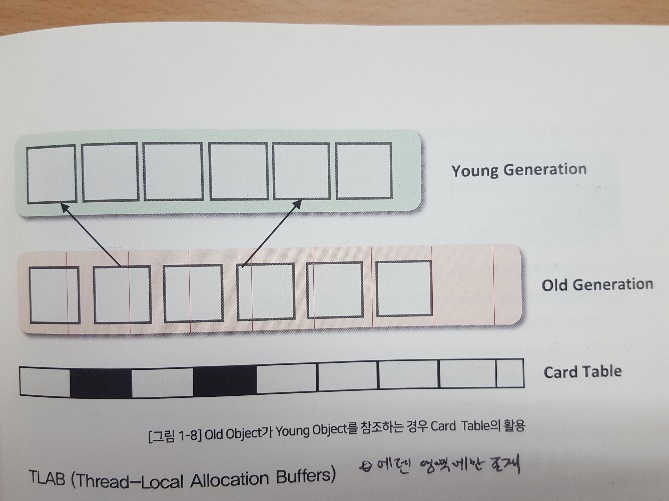
**두번째 가설**

-Card Table을 이용하여 빠른 관계 파악

**Card Table**

-Young Generation의 Object를 참조하는 Old Generation의 Object가 있다면 시작주소에 카드(Flag)를 Dirty로 표시하고 해당 내용을 Card Table에 기록한다.

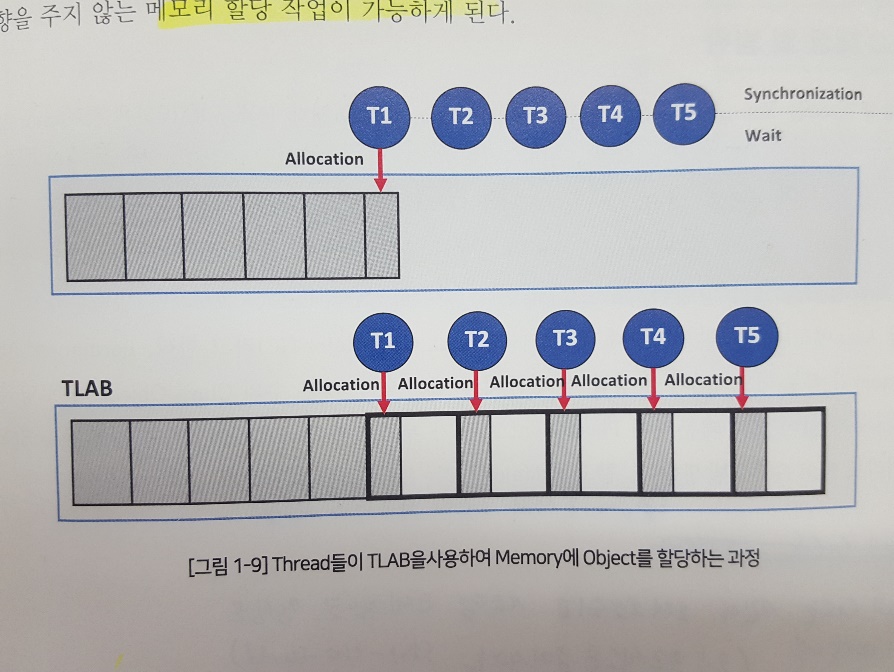
-이후 해당 Reference가 해제하면 표 시한 Dirty Card도 사라지게끔 한다.



**TLAB (Thread-Local Allocation Buffers)**

-Young Area의 Fast Allocation을 위한 것 (Eden Area)

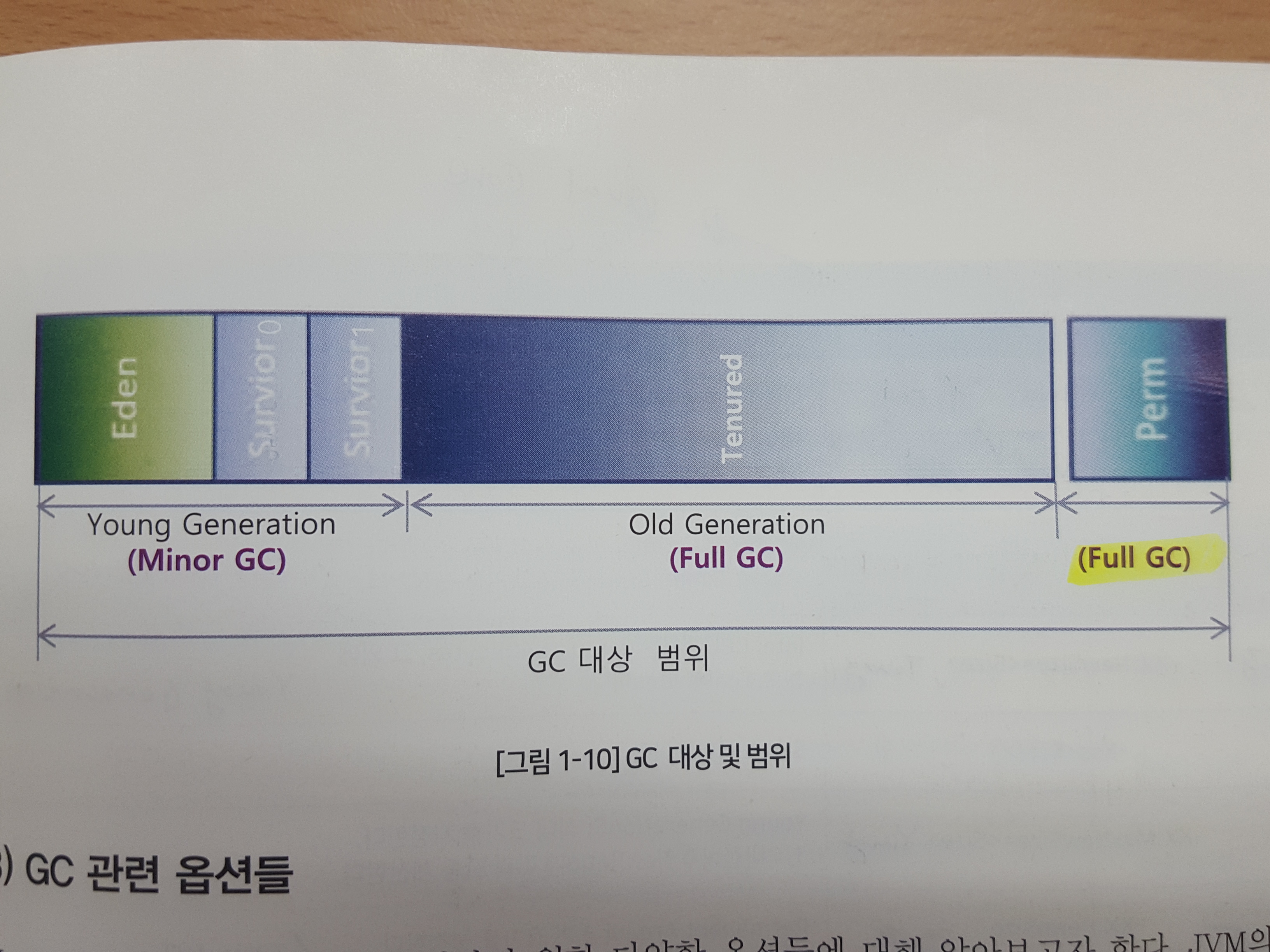
-스레드 로컬 할당 버퍼(TLABs)를 사용하여 각 스레드별 메모리 버퍼를 사용



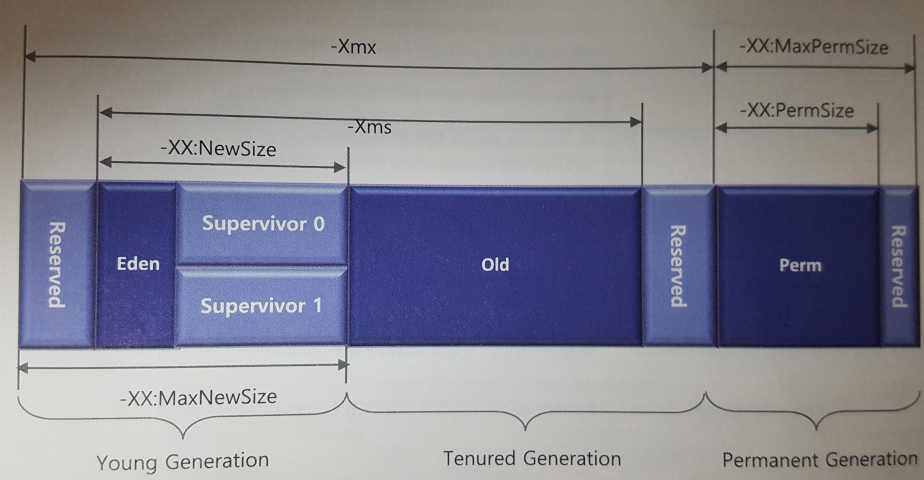
**2) GC 대상 및 범위**

-Young Generation -> Minor GC

-Old Generation -> Full GC



**3) GC 관련 옵션들**



|  |  |
| --- | --- |
| 옵션 | 상세 설명 |
| -Client, -Server | 해당 모드로 구현 |
| -Xms  -Xmx | Total Heap Size |
| -XX:NewSize  -XX:MaxNewSize | Young Generation Heap Size |
| -XX:PermSize  -XX:MaxPermSize | Permanet Area 크기 설정 |
| -XX:NewRatio | Young : OId 비율  -Client 1 : 8  -Server 1 : 2 |
| -XX:MinHeapFreeRatio  -XX:MaxHeapFreeRatio | 가변적으로 메모리 조정 옵션 |

-MinHeapFreeRation와 MaxHeapFreeRation는 –Xms 와 –Xmx로 지정한 Memory 크기가 같으면

아무런 영향이 없다.

- **-X**를 붙이는 것은 보통 Macro한 측면의 제어, **-XX:** 를 붙이는 것은 Micro한 측면의 제어

- -XX:+<옵션> 옵션 ON

- -XX:-<옵션> 옵션 OFF

- -XX:<옵션>=<numeric> 용량과 같은 단위를 포함

- -XX:<옵션>=<string> 특정 파일 또는 Path지정

**4) Garbage Collector 종류**

|  |  |  |  |
| --- | --- | --- | --- |
| **GC** | **Option** | **Young 알고리즘** | **Old 알고리즘** |
| Serial Collector | -XX:+UseSerialGC | Serial | Serial  Mark-Sweep-Compact |
| Parallel Collector | -XX:+UseParallelGC | Parallel Scavenge | Serial  Mark-Sweep-Compact |
| Parallel Compacting  Collector | -XX:+UseParallelOldGC | Parallel Scavenge | Parallel  Mark-Summary-Compact |
| CMS Collector | -XX:+UseConcMarkSweepGC | Parallel | Concurrent  Mark-Sweep |
| G1 Collector | -XX:+UseG1GC | SATB | |

**Parallel Collector**

-다른 CPU가 대기 상태로 남아 있는 것을 최소화 하자.

-Young Area에서 콜렉션을 병렬로 처리한다.

-Old Area은 Mark-Sweep-Compact 알고리즘 사용

**PLAB (Parallel Allocation Buffer) 이란**

-CG Thread가 Promotion시 Thread 마다 Old Generation의 일정부분을 할당하고 다 사용하면 다시 Buffer를 재할당 한다. 단 단편화가 발생할 수 있다.

🡺Memory 공간을 두 Thread가 접근하면 Corruption이 발생할 수 있지만 PLAB을 통해 회피한다.

**CMS Collector**

-힙 메모리 영역의 크기가 클 때 적합하다.

-비교적 자원이 여유 있는 상태에서 GC의 Pause Time을 줄이는 목적이며 Size가 큰 Long Lived Object가 있는 경우에 가장 적합하다.

-Young Area에서는 Parallel Copy 알고리즘 사용

-Old Area에는 Concurrent Mark-Sweep 알고리즘 사용.

**FreeList**

-승격 할당을 할 때 Young Area에서 승격된 Object와 크기가 비슷한 Old Area의 Free Space를 탐색하게 된다. Compaction 작업을 수행하면 Old Area에 그냥 Object를 순서대로 할당하여 Fast Allocation이 가능한데 CMS는 Compation을 하지 않기 때문에, 승격이 빈번하지 않다면 성능상 이득이 있다.

🡺Compation을 하는 시간대신 빈 Old Area 공간을 찾아서 Allocation

**Floating Garbage**

-Initial Mark 단계에서 없었던 Heap에 존재하던 객체만 GC 대상이 되기 때문에 그 이후에 들어 온건 다음번 GC의 Initial Mark 단계에서 Live 여부를 판단하게 되고 이는 잠재적으로 Old Area를 확장 시키게 되는 요인이다.

**Parallel Compaction Collector**

-Parallel Collector에서 Old Area에 새로운 알고리즘이 추가된 개념으로 Multi CPU에서 유리

-Mark Phase : 살아 있는 객체를 식별하여 표시해 놓는 단계

-Summary Phase : 이전에 GC를 수행하여 컴팩션된 영역에 살아 있는 객체의 위치를 조사

-Compact Phase : 컴팩션을 수행하는 단계로 수행 이후에는 컴팩션된 영역과 비어 있는 영역으로 나뉜다.

**Garbage first Collector**

-G1은 물리적 Generation 구분을 없애고 전체 Heap을 1Mbytes 단위로 Region으로 재편한다.

**GC 참고 :**

<http://d2.naver.com/helloworld/1329>

<http://d2.naver.com/helloworld/37111>

GC 튜닝을 진행하기로 결정했다면 GC 방식을 선정하고 메모리의 크기를 지정한다. 이때 서버가 여러 대이면 여러 대의 서버에 GC 옵션을 서로 다르게 지정해서 GC 옵션에 따른 차이를 확인하는 것이 중요하다.

GC 상황을 확인 할 때 Minor GC와 Full GC의 시간만 보면 안 된다는 점이다. GC가 수행되는 횟수도 확인해야 한다. 만약 New 영역의 크기가 너무 작게 잡혀 있다면 Minor GC가 발생하는 빈도도 매우 높을 뿐만 아니라(1초에 한번 이상인 경우도 있음), Old 영역으로 넘어가는 객체의 개수도 증가하게 되어 Full GC 횟수도 증가한다.