Garbage Collection

§ Garbage Collection이 호출될 때

과거 많은 프로그래머들은 동적으로 할당한 메모리를 적절하게 해제하지 못해 메모리 누수(memory leak) 문제를 자주 겪었습니다. 이러한 실수는 프로그램의 성능 저하와 시스템 불안정으로 이어질 수 있었고, 메모리 관리는 어려운 과제 중 하나로 남아 있었습니다. 이를 해결하기 위해 자바는 가비지 컬렉션이라는 자동 메모리 관리 기법을 도입했습니다.

가비지 컬렉션은 매번 명령어를 실행할 때마다 발생하는 것이 아닙니다. 마치 음식을 한 입 먹을 때마다 그릇을 씻고 돌아오지 않는 것처럼, 특정 조건 없이 가비지 컬렉션이 즉시 실행되지는 않습니다. 주로 블록을 빠져나가거나 메모리 부족 등의 특정 상황에서 내부 알고리즘에 따라 실행됩니다.

1. 인스턴스가 더 이상 참조되지 않을 때

가비지 컬렉션의 주요 대상은 더 이상 참조되지 않는 인스턴스들입니다. 참조되지 않는 인스턴스는 더 이상 존재 가치가 없어지며, JVM의 가비지 컬렉터에 의해 회수됩니다.

```
public class XCIGcTest1 {

    public static void main(String[] args) {

        String str = new String("Hello");

        str = null; // 이제 "Hello"를 참조하는 인스턴스는 없음

    }

}
```

str이 null로 설정되면, "Hello" 문자열 인스턴스는 더 이상 참조되지 않으므로 가비지 컬렉션 대상이 됩니다.

2. 메모리가 부족할 때

JVM은 힙 메모리가 부족할 때 가비지 컬렉션을 강제로 실행하여 더 많은 메모리를 확보하려고 합니다. 인스턴스들이 많이 생성되었는데 참조되지 않거나 사용되지 않는 인스턴스가 많다면, 가비지 컬렉션이 발생해 이러한 인스턴스들을 제거하여 메모리를 회수합니다.

3. JVM에 의해 주기적으로 실행될 때

JVM은 자체적인 스케줄에 따라 주기적으로 가비지 컬렉션을 실행합니다. 이는 백그라운드에서 이루어지며, 메모리 상태와 JVM 설정에 따라 빈도나 시점이 달라질 수 있습니다.

4. 명시적인 가비지 컬렉션 요청, System.gc()가 호출될 때

System.gc() 메서드를 호출하여 가비지 컬렉션을 요청할 수 있습니다. 그러나 이는 단지 요청일 뿐, JVM이 반드시 즉시 가비지 컬렉션을 수행하는 것은 아닙니다. JVM은 요청을 무시하거나, 적절한 시점에 실행할 수 있습니다.

```
public class XCIGcTest2 {

    public static void main(String[] args) {

        System.gc(); // 가비지 컬렉션 요청

    }

}
```

System.gc() 호출은 일반적으로 권장되지 않으며, JVM의 가비지 컬렉션 스케줄을 신뢰하는 것이 좋습니다.

5. 인스턴스의 생존 시간이 끝났을 때

특정 인스턴스는 메서드 종료와 같이 스코프(scope)가 끝날 때 더 이상 참조되지 않습니다. 예를 들어, 지역 변수가 사용되던 메서드가 종료되면 그 메서드에서 사용하던 인스턴스들이 가비지 컬렉션 대상이 될 수 있습니다.

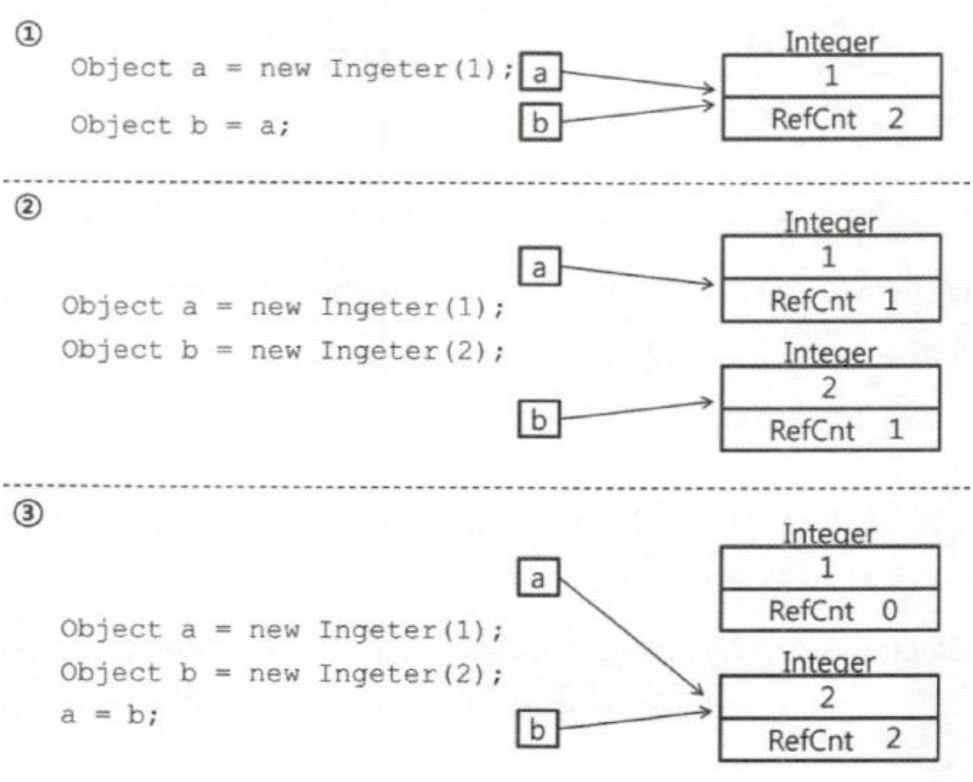
Example.

```
interface A {
}
interface B {
}
class RefMe extends Object implements A,B {
	public RefMe() {
		System.out.println("태어남~");
	}
	@Override
	protected void finalize() throws Throwable {
		System.out.println("낮의 빛이 밤의 어둠의 깊이를 알게 뭐냐");
	}
}

public class XCIGcTest {
	public static void main(String[] args) {
		RefMe r1 = new RefMe();
		RefMe r2 = r1;
		r1 = null;
		System.gc();
		r2 = null;
		System.gc();
		System.out.println("종료");
	}
}
```

Garbage Collection Algorithm

Reference Counting Algorithm

각 Object 마다 Reference Count를 관리하여 Reference Count가 0일 때 Garbage Collection을 수행한다. (자바는 사용하지 않으며 **Mark-and-Sweep, Mark-and-Compaction** 과 같은 **추적 기반 알고리즘**을 사용)



장점

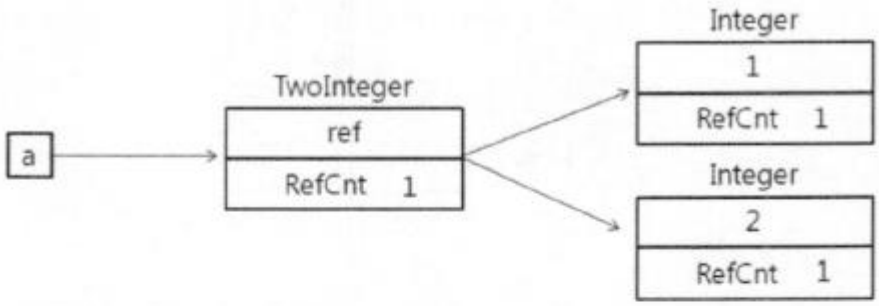
1. Garbage Object를 확인하는 작업이 간단하다.
2. Pause Time이 분산되어 실시간 작업에도 거의 영향을 주지 않는다.

단점

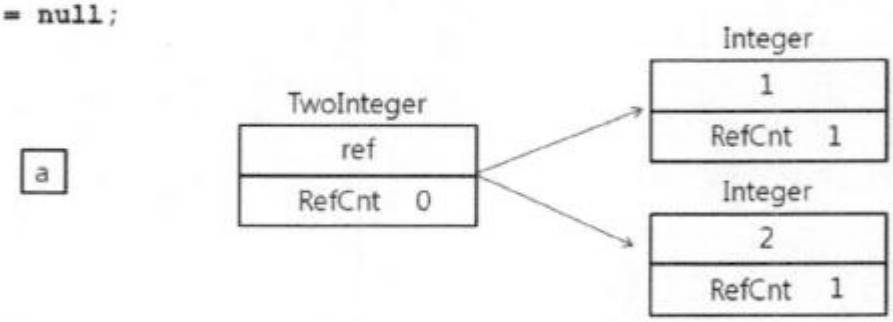
1. Reference의 변경이나 Garbage Collection의 결과에 따라, 각 Object 마다 Reference Count를 변경해 주어야 하기 때문에 관리 비용이 크다.
2. Garbage Collection이 연속적으로 일어날 때 문제가 발생 할 수 있다.

Reference Counting Algorithm의 문제 발생의 예 1.

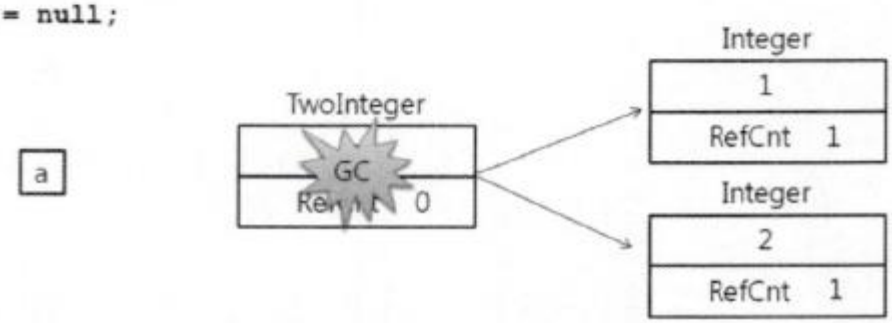
```
Object a = new TwoIngeter(new Integer(1), new Integer(2));
```




```
Object a = new TwoIngeter(new Integer(1), new Integer(2));

a = null;
```

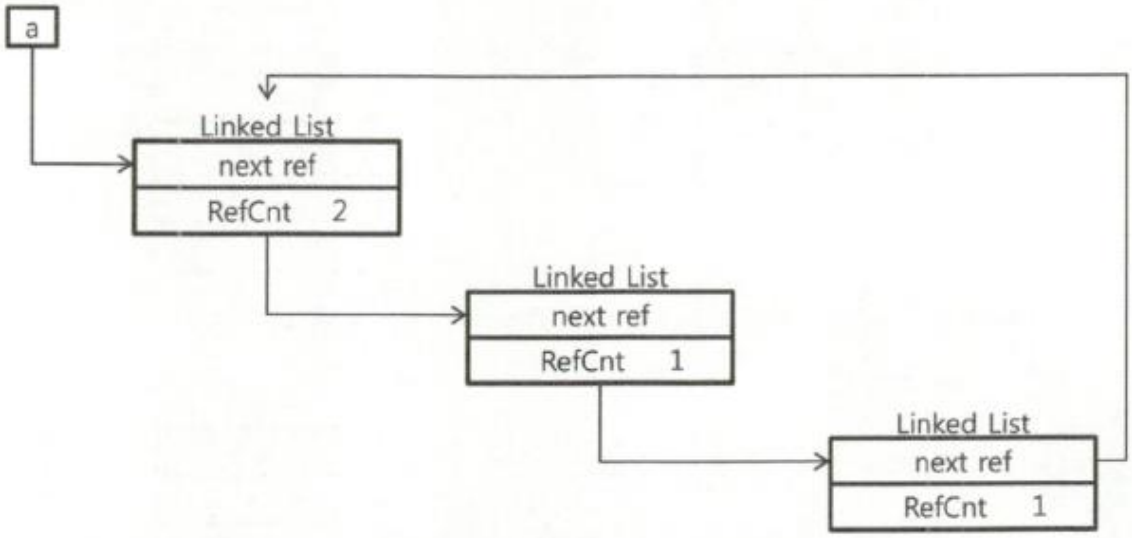


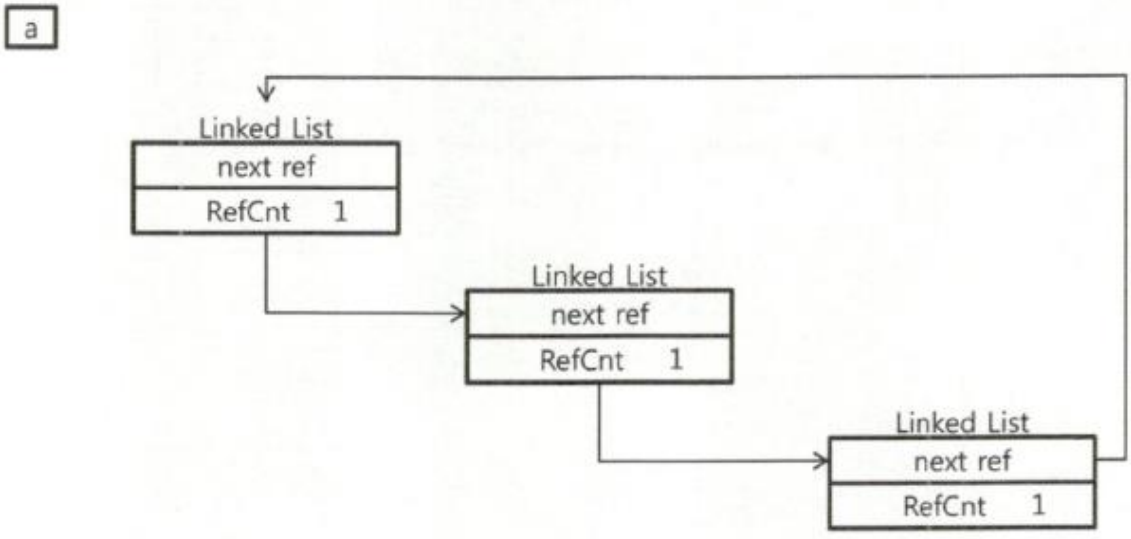

```
Object a = new TwoIngeter(new Integer(1), new Integer(2));

a = null;
```



```
Object a = new TwoIngeter(new Integer(1), new Integer(2));

a = null;
```

Reference Counting Algorithm의 문제 발생의 예 2.





Reference Count가 0이 되어야 Garbage Collection이 수행 되는데 순환 참조 구조에서는

Reference Count가 0이 되지 않을 수 있다. 이 경우 Memory Leak을 유발 할 수 있다.

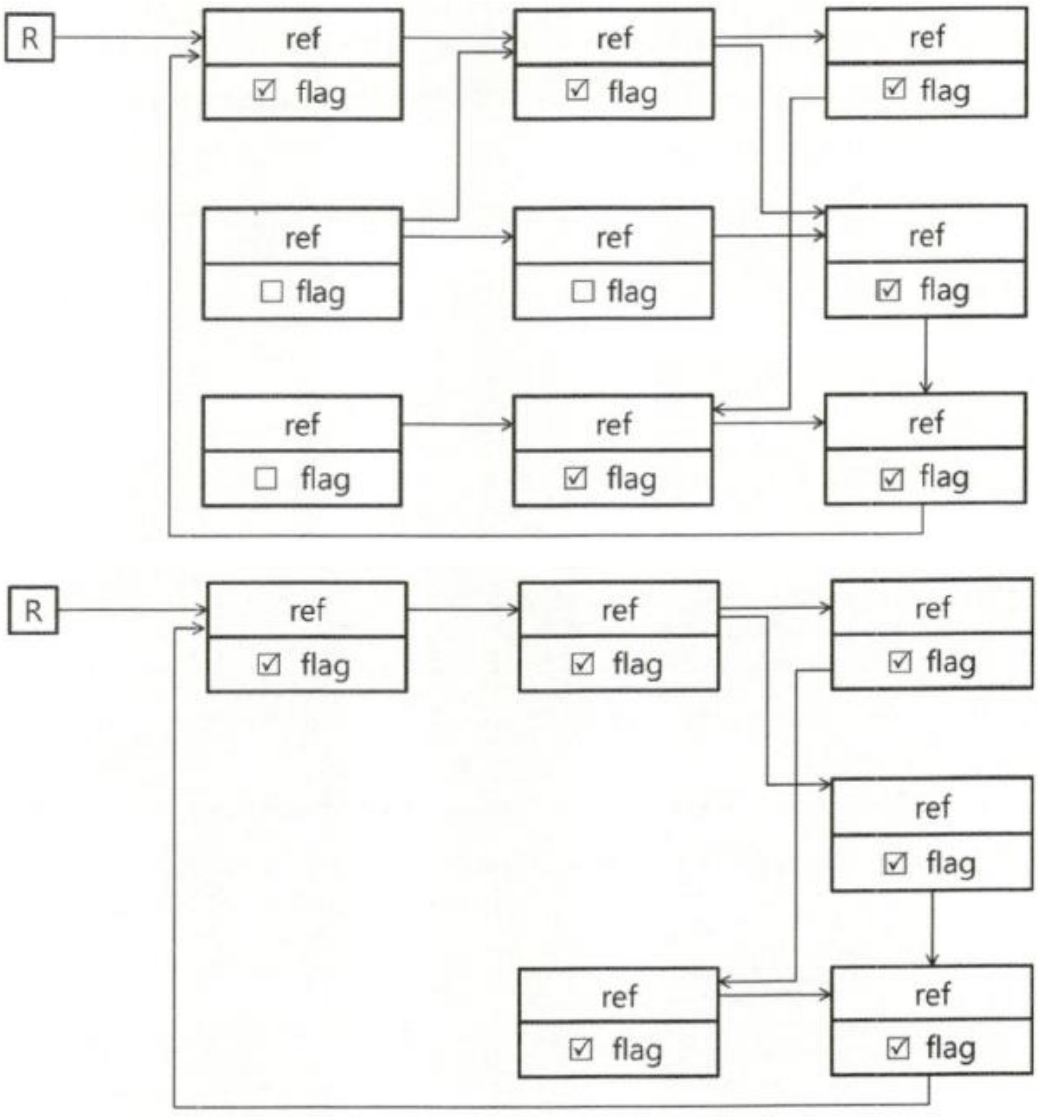
Mark-and-Sweep Algorithm

Tracing Algorithm 이라고도 불리운다.

Reference Counting Algorithm의 단점을 극복하기 위하여 Object 마다 Count를 하는 방식 대신, Root Set 에서 시작하는 Reference의 관계를 추적하는 방식을 사용한다. 이 방식은 Garbage를 찾을 때 효과적이다.

Mark Phase에서 Garbage Object를 구별해 내기 위해 Root Set에서 Reference 관계가 있는 Object에 Marking을 한다.

Sweep Phase에서 Marking이 없는 Object를 삭제한다.



장점

1. Reference 관계가 정확하게 파악된다.
2. Reference 변경시 부가 적업이 없으며 속도가 빠르다.

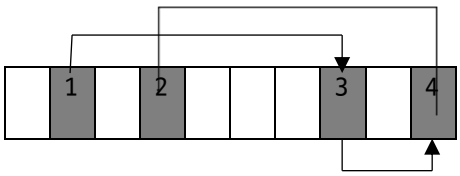
단점

1. Garbage Collection 과정 중에는 Heap의 사용이 제한된다. (Memory Corruption 방지) 이 때 프로그램 Suspend가 발생한다.
2. Fragmentation이 발생 할 수 있다.

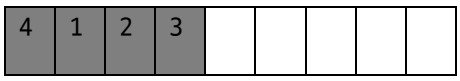
Mark-and-Compaction Algorithm

Flagmentation이 발생하는 단점을 보완하기 위한 알고리즘으로 Mark Phase 이후 Compaction Phase를 수행한다. Compaction Phase란 Live Object를 연속된 메모리 공간에 차곡차곡 적재하는 것을 의미하며 적제후 Sweep Phase를 포함한다. 일반적으로 Arbitrary(임의) 방식, Linear 방식, Sliding 방식 등이 있다.

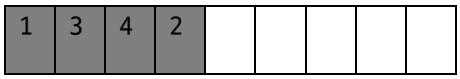
[before Compaction]



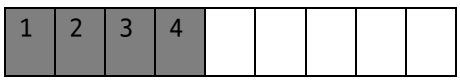
[Arbitrary Compaction]



[Linear Compaction]



[Sliding Compaction]

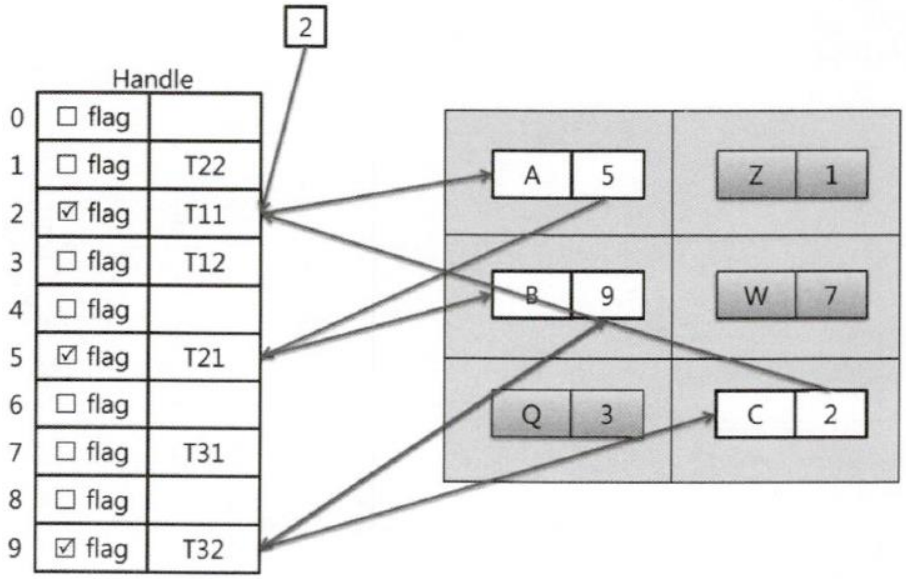


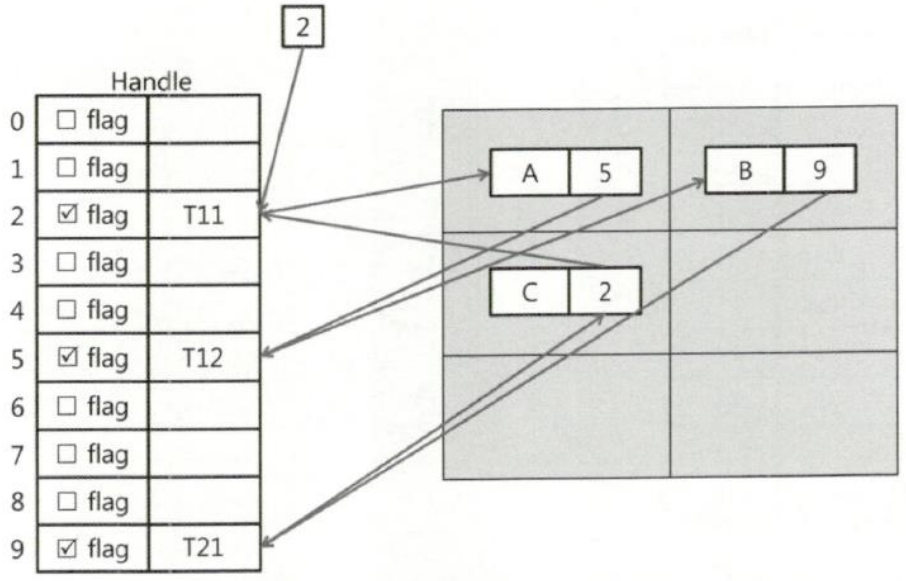
가장 효율적인 방식은 Sliding 방식으로 알려져 있다. Linear 방식은 Reference 순서를 따라가기 위한 Overhead가 발생하기 때문이다. 또한 Object 탐색은 순차적 Access가 아닌 포인터를 기반으로한 Random Access를 수행하기 때문에 인접해 있다고 해서 장점이 되지 못한다.

Mark-and-Compaction Algorithm은 Compaction을 원활하게 하기 위한 Handle과 같은 자료구조를 사용할 수도 있다. Handle에는 객체의 논리적인 주소와 실제 주소가 들어가 있다.

6개의 Heap으로 구성되어 왼쪽 위부터 오른쪽 하단까지 T11,T12,T21,T22,T31,T32의 순서로 주소가 되어 있다고 가정하자. Root Set은 Handle의 2번 주소를 가리키며 A, B, C는 Reference 관계를 갖는다. Q, W, Z도 상호 Reference 관계를 갖지만 Root Set에서 Reference 하지 않고 있다.

Mark Phase 에서는 Live Object인 A, B, C의 Handle에 Marking 하게 된다.





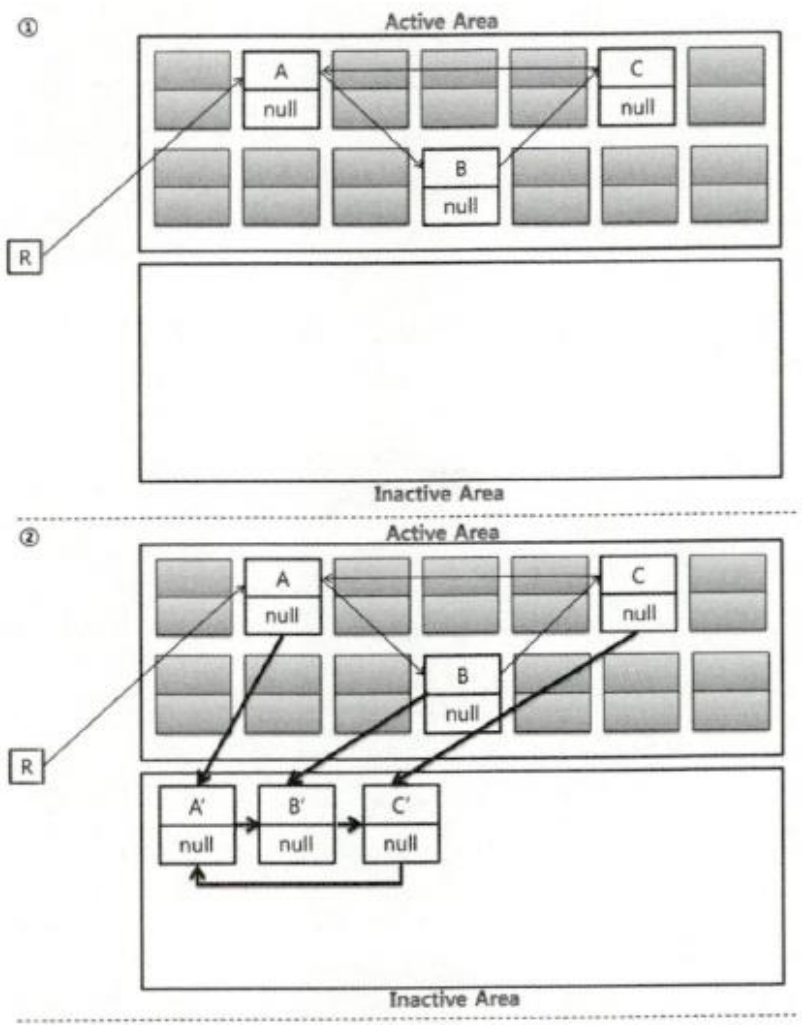
Compaction Phase에서는 Marking 된 정보를 바탕으로 Garbage Object를 Sweep 한 후 Heap의 한 쪽 방향으로 Live Object를 이동 시킨다. 위 그림의 이동하는 방법은 Sliding Compacion의 방법을 사용하였다. 그 이후 이동한 새로운 주소로 모든 Reference를 변경하는 작업을 수행한다.

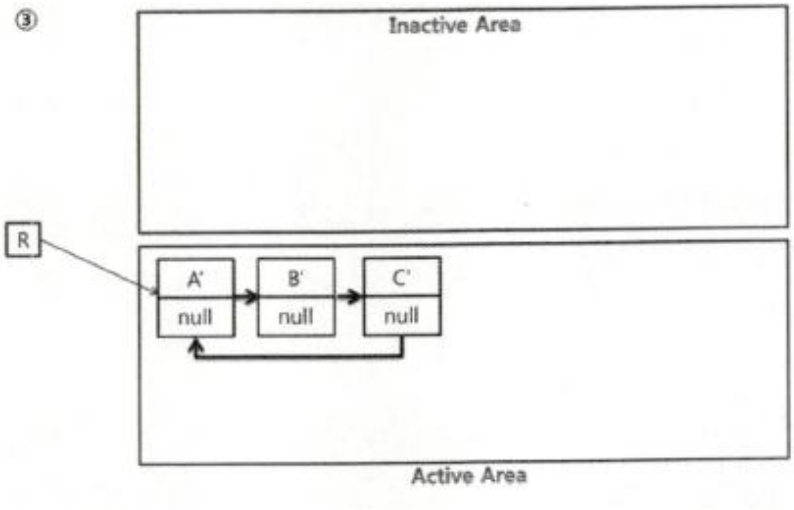
장점 : Fragmentation의 방지에 중점을 둔 Garbage Collection 기법. 메모리 공간의 효율성이 좋다.

단점 : Compaction 작업 이후 모든 Reference를 업데이트 하는 작업은, 경우에 따라서 모든 Object를 Access 하는 등의 부가적인 Overhead를 수반할 수 있으며 Mark Phase와 Compaction Phase 모두 Suspend 현상이 발생한다.

Copying Algorithm

Fragmentation의 문제를 해결하기 위해 제시된 또 다른 방법으로 Heap을 Active 영역과 Inactive 영역으로 나누어 사용한다. Active 영역에 Object를 할당 하고 Active 영역이 꽉 차게 되어 더 이상 Allocation이 불가능하게 되면 Garbage Collection이 수행된다.





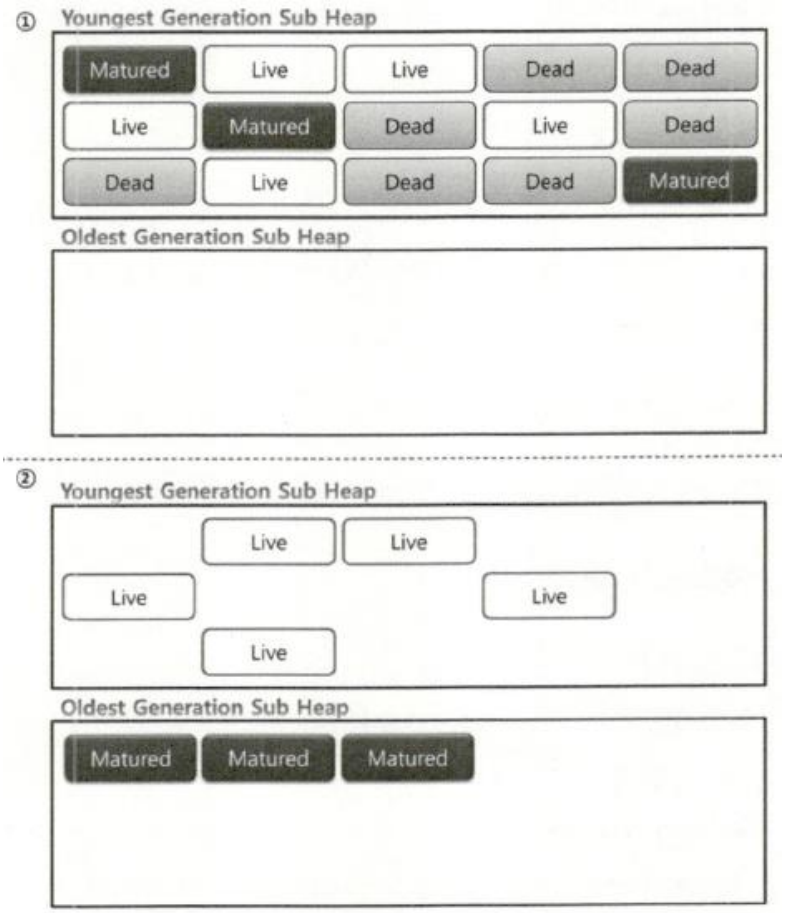
장점 : Fragmentation 방지에 효과적

단점 : 전체 Heap의 절반 정도를 Inactive Area로 사용하므로 활용도가 50%로 떨어지며 Suspend 현상과 Copy에 대한 Overhead가 발생한다.

Generational Algorithm

Hotspot JVM이나 IBM JVM에서 사용하는 대표적인 Heap인 Generational Heap에서 사용하는 Garbage Collection Algorithm.

대다수의 Object는 생성된지 얼마 되지 않아 Garbage가 되는 짧은 수명을 가진다. 때문에 수명이 긴 소수의 Long Lived Object의 경우 Inactive와 Active를 여러번 번갈아가며 Copy 작업을 할 때 Overhead가 발생한하게 된다. 이를 개선하기 위해 Generational Algorithm은 Heap을 Age 별로 몇 개의 Sub Heap으로 나눈다. 그리고 Marking시 Age를 카운트 하여 Age가 일정 수치를 넘게 되면 Long Lived Object로 판단하고 Matured Object가 되어 Promotion을 하고 Garbage Object는 Sweep 한다.



최근의 자바 GC Algorithm

최근의 자바 가비지 컬렉션 알고리즘들은 Generational 및 Concurrent 방식에 기반한 최신 기술로 설계되었습니다. 이들 알고리즘은 메모리 효율성과 애플리케이션 성능을 개선하기 위해 발전해 왔습니다.

1. G1 (Garbage First) GC

G1 GC는 Java 9 이후 기본 가비지 컬렉터로, 메모리를 Region이라는 단위로 나누어 관리합니다. G1 GC는 자주 가비지로 변할 가능성이 높은 영역을 우선적으로 수집하며, 짧은 정지 시간과 높은 처리량을 목표로 설계되었습니다. 이는 메모리의 Eden 영역에서 젊은 객체를 관리하고, 일정 기간 동안 살아남은 객체는 Old Generation으로 이동시켜 관리하는 방식으로 최적화됩니다.

2. ZGC (Z Garbage Collector)

ZGC는 Java 11에서 도입된 저지연(ultra-low latency) 가비지 컬렉터로, 큰 힙 메모리를 관리하는 애플리케이션에서 짧은 일시 정지 시간을 유지하는 것을 목표로 합니다. ZGC는 애플리케이션 스레드와 동시에 작동하는 Concurrent 방식의 수집을 사용하여, 최대 수 밀리초의 짧은 지연 시간만 발생하도록 설계되었습니다. Java 21에서는 Generational ZGC가 도입되어, 젊은 객체와 오래된 객체를 구분하여 더욱 효율적인 메모리 관리를 가능하게 했습니다.

3. Shenandoah GC

Shenandoah GC는 대형 힙 메모리에서 짧은 일시 정지 시간을 제공하기 위해 개발된 또 다른 Concurrent 방식의 가비지 컬렉터입니다. Shenandoah는 애플리케이션 실행 중에 가비지 컬렉션을 병렬로 수행하여 힙의 크기에 상관없이 일관된 성능을 유지하는 것을 목표로 합니다. 이 방식은 특히 짧은 응답 시간이 중요한 대규모 애플리케이션에 적합합니다.

이들 최신 가비지 컬렉션 알고리즘은 지연 시간 최소화, 처리량 최적화, 그리고 대규모 메모리 관리에 중점을 두며, 자바 애플리케이션의 성능을 대폭 향상시킵니다. 각 알고리즘은 특정 요구사항에 맞게 최적화되어 있어, 개발자들은 애플리케이션의 특성에 따라 적절한 GC를 선택해 사용할 수 있습니다.