

# Effective Java

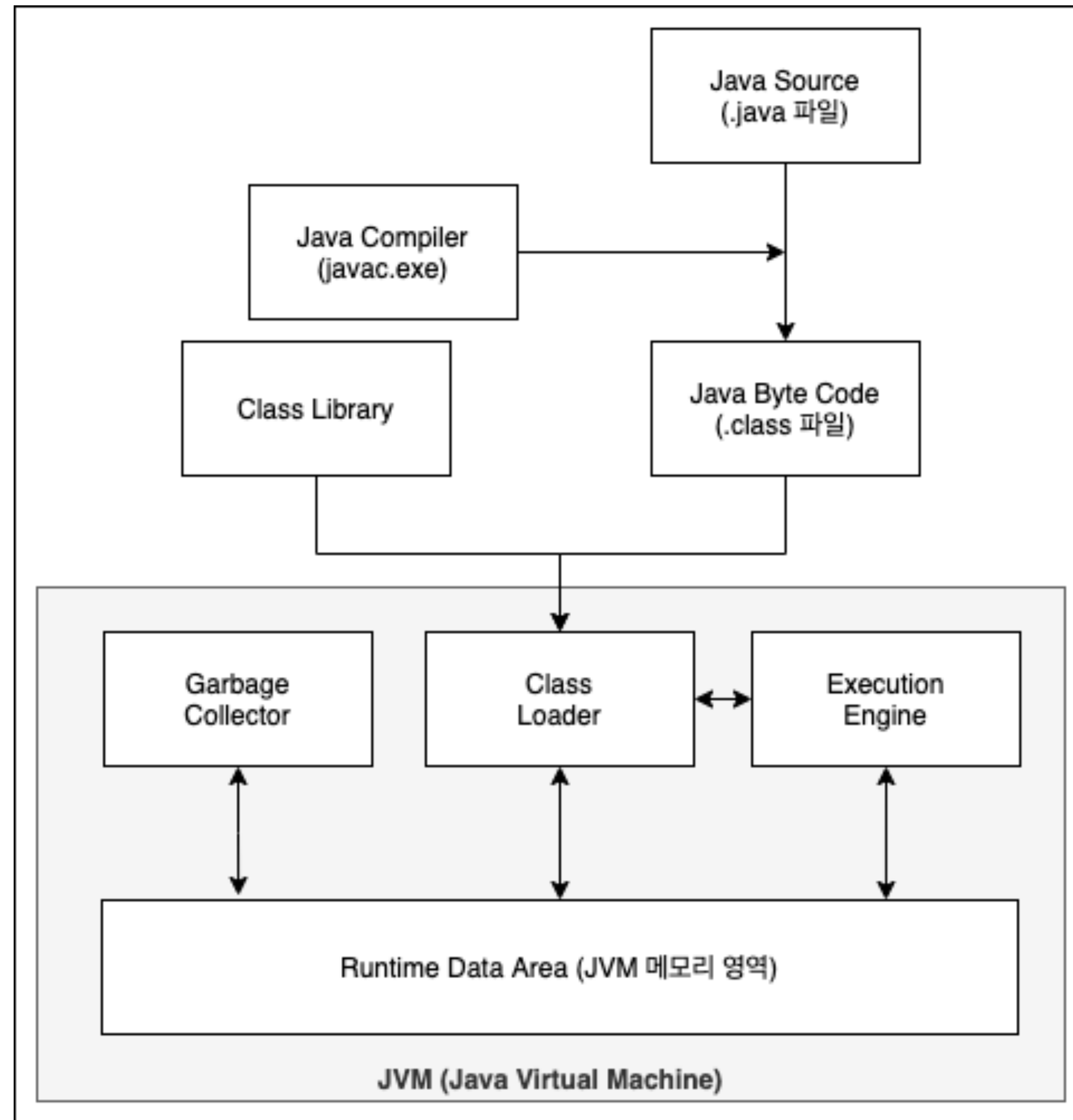
## 아이템 7

**'다 쓴 객체 참조를 해제하라'**

아이템 7. 다 쓴 객체 참조를 해제하라	목차
<div data-bbox="1172 724 1402 771">참고 사이트</div> <ul data-bbox="406 784 2169 1153" style="list-style-type: none"><li>- <u>Understanding the Java Memory Model and Garbage Collection</u></li><li>- <u>마로의 Java(자바) 정리 - 8. 자바 메모리 구조</u></li><li>- <u>JVM(Java Virtual Machine)이란</u></li><li>- <u>Memory Leaks and Java Code</u></li><li>- <u>Demystifying memory management in modern programming languages</u></li><li>- <u>Visualizing memory management in JVM(Java, Kotlin, Scala, Groovy, Clojure)</u></li><li>- <u>Java Memory Leaks in Java</u></li></ul>	1. 메모리 누수에 인한 장애의 징조
	2. 메모리 구조 <ul style="list-style-type: none"><li>- 자바에서 메모리를 점유하는 방법</li></ul>
	3. Garbage Collection
	4. 메모리 누수의 원인
	5. 메모리 모니터링

1. 메모리 누수에 인한 장애의 징조	<div>* 메모리 누수에 인한 장애의 징조</div> <div>- 성능 저하</div> <div>- 간헐적 503 오류 발생</div> <div>- OutOfMemoryException</div>
----------------------	--

## 2. JVM 구조



### \* Java 실행 과정

1. Java Source

2. Java Compiler

3. Java Byte Code

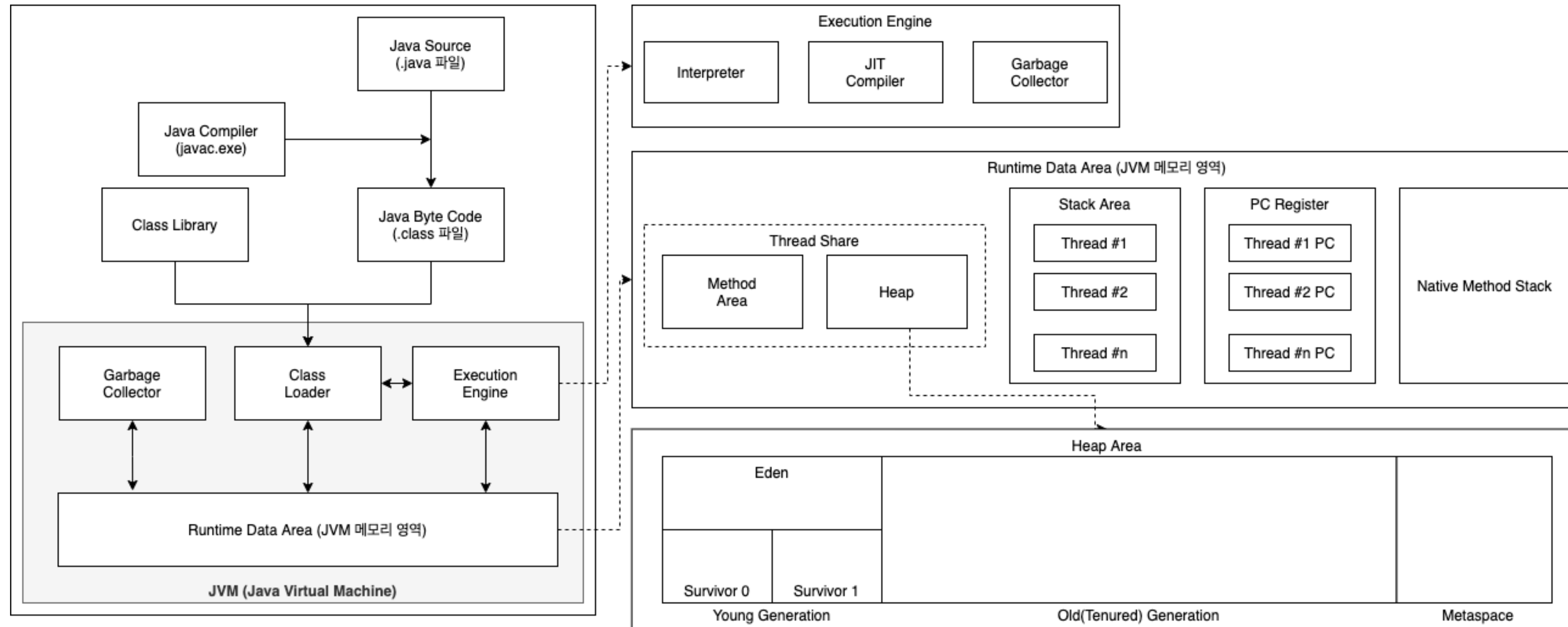
4. Class Loader

5. Runtime Data Area

- GC

- Execution Engine

# Runtime Data Area



## 2. JVM 구조

\* Runtime Data Area

### 1. Method(Static or Class) Area

### 2. Heap Area

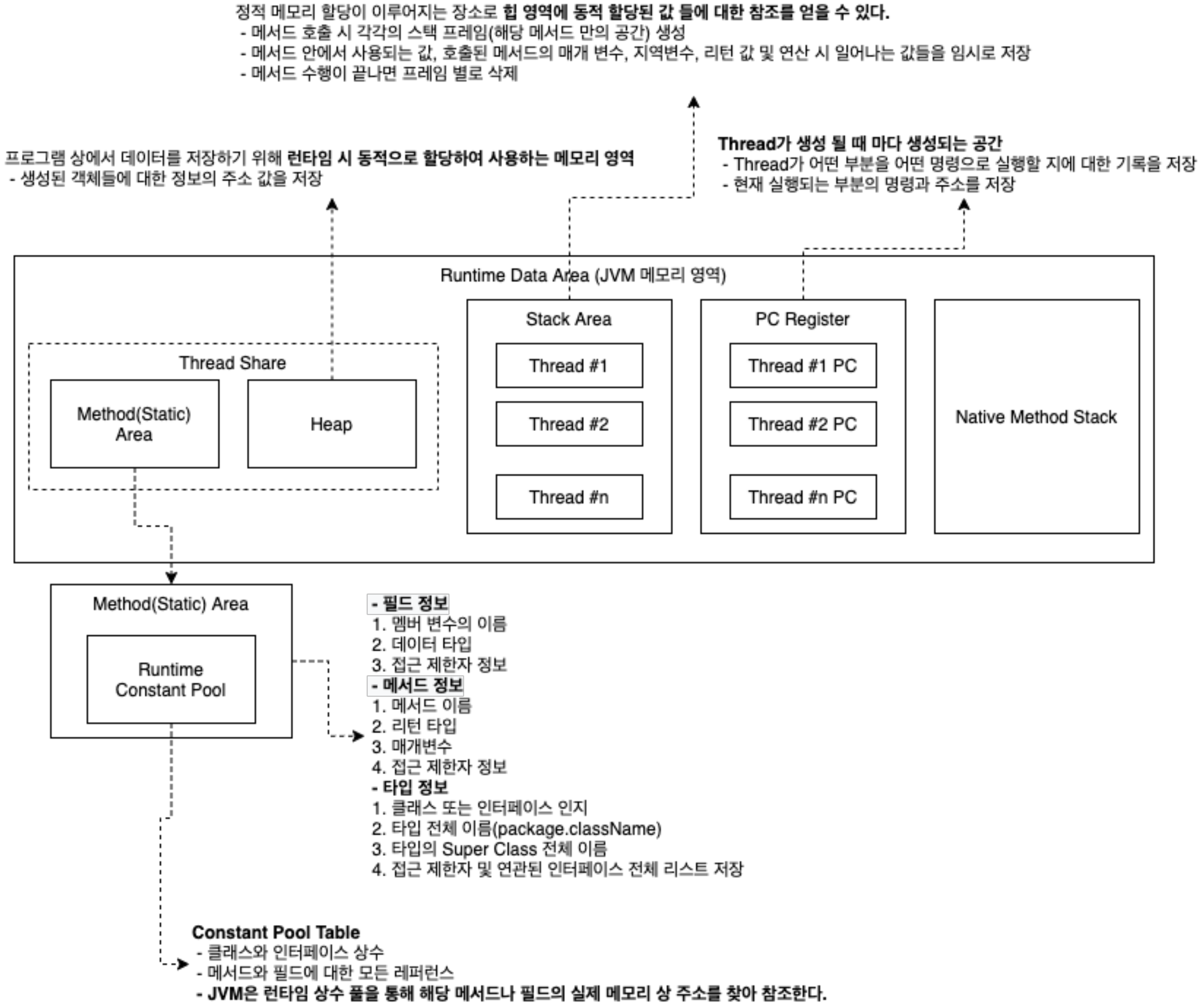
- Young Generation
- Old Generation
- Parmenent Generation

### 3. Stack Area

### 4. PC Register

### 5. Native Method Stack

\* 각 영역에서 관리하는 자원



메모리에 객체를 할당하는 과정

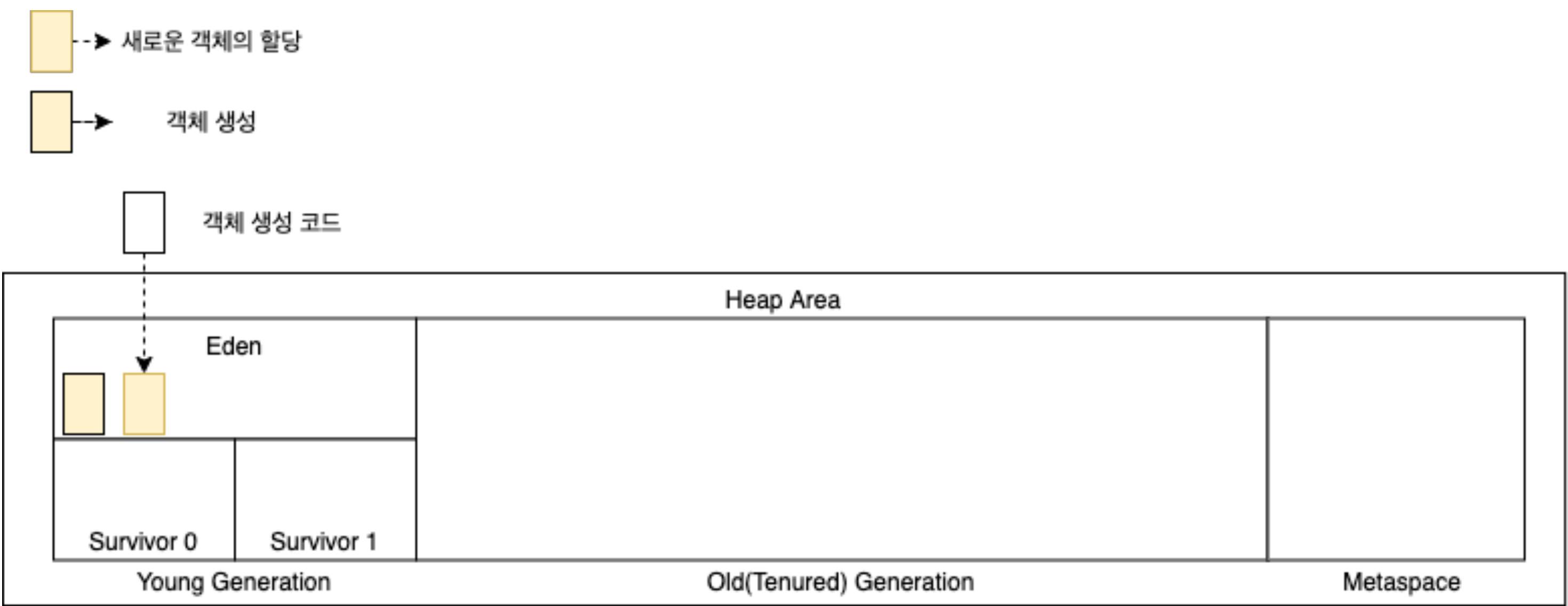
3. Garbage Collection

\* 메모리에 객체를 할당하는 과정

1. 객체 생성 코드

2. Eden 영역에 메모리 할당

3. 새로운 객체 생성




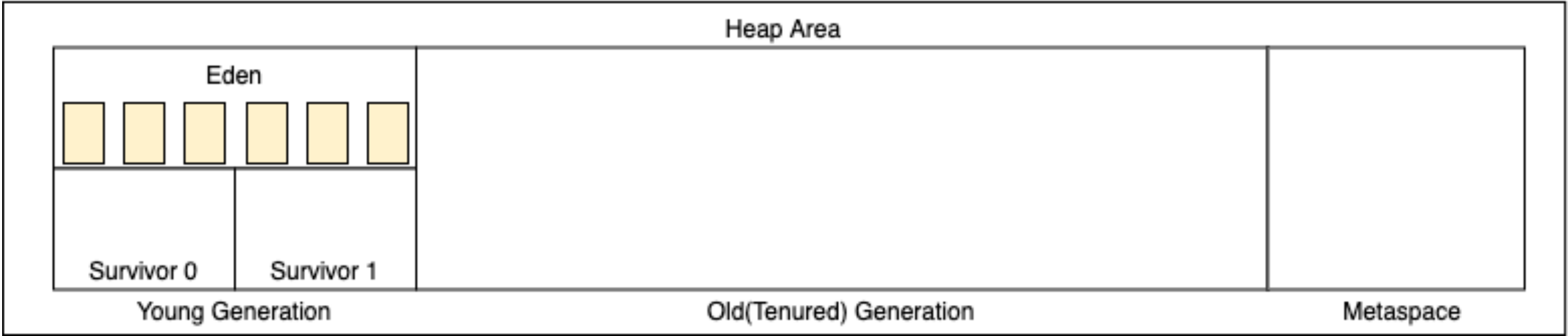
Garbage Collection Trigger

3. Garbage Collection

\* Garbage Collection Trigger

1. Eden 영역의 공간이 부족한 상황 발생

 --> Eden 영역의 공간이 부족하여 할당할 자리가 없는 객체





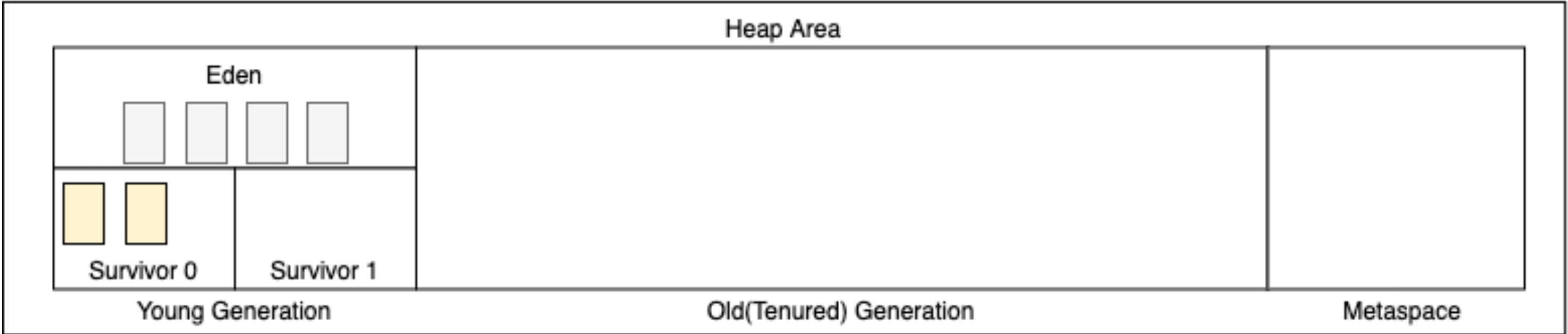
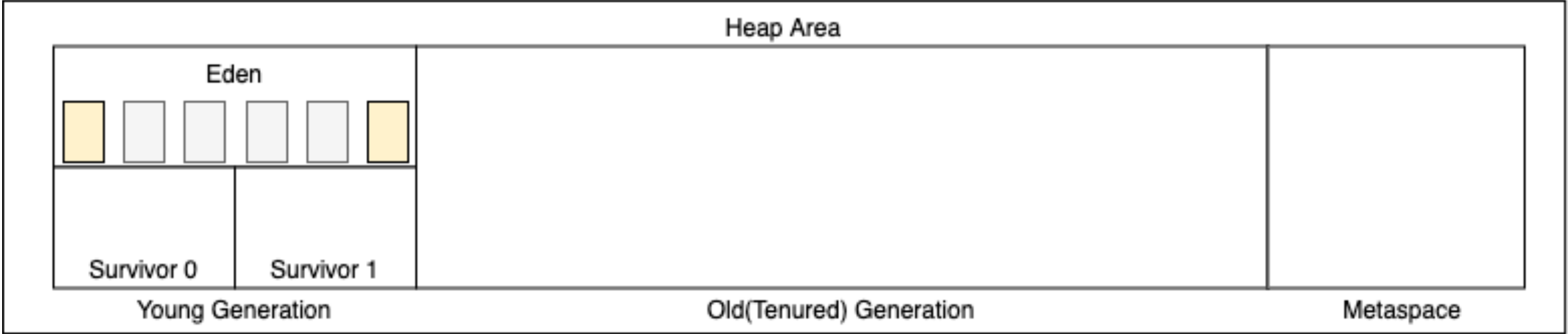
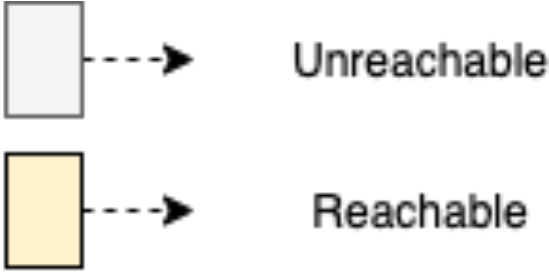
Minor GC

3. Garbage Collection

\* Minor GC

1. **Marking** 작업  
- Reachable or Unreachable

2. Reachable 객체는 보관



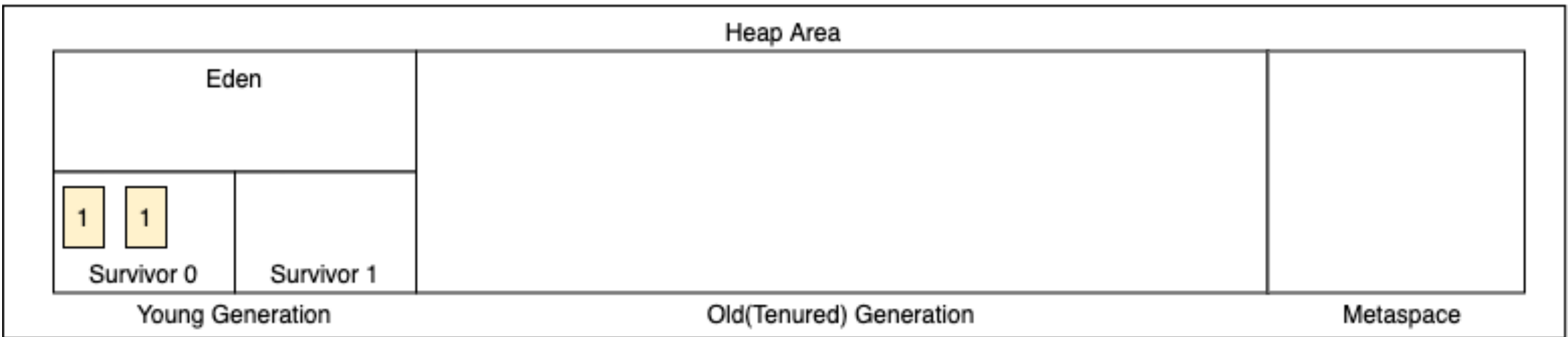
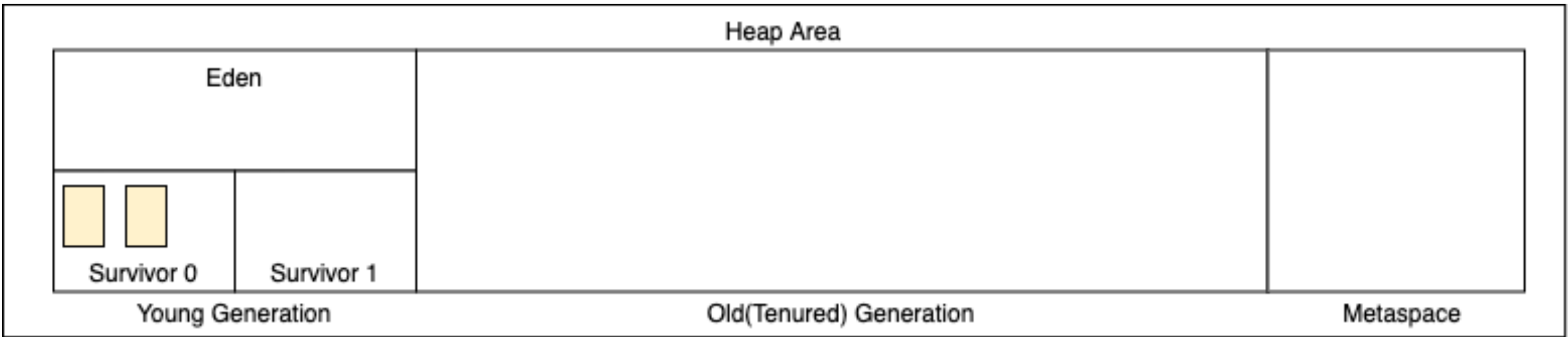
# Minor GC

## 3. Garbage Collection

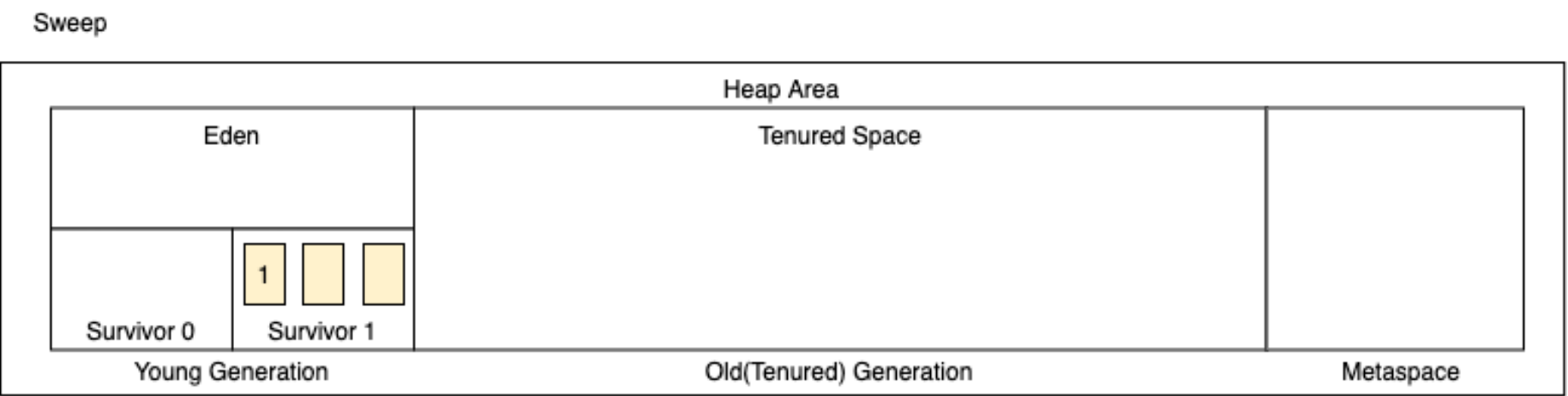
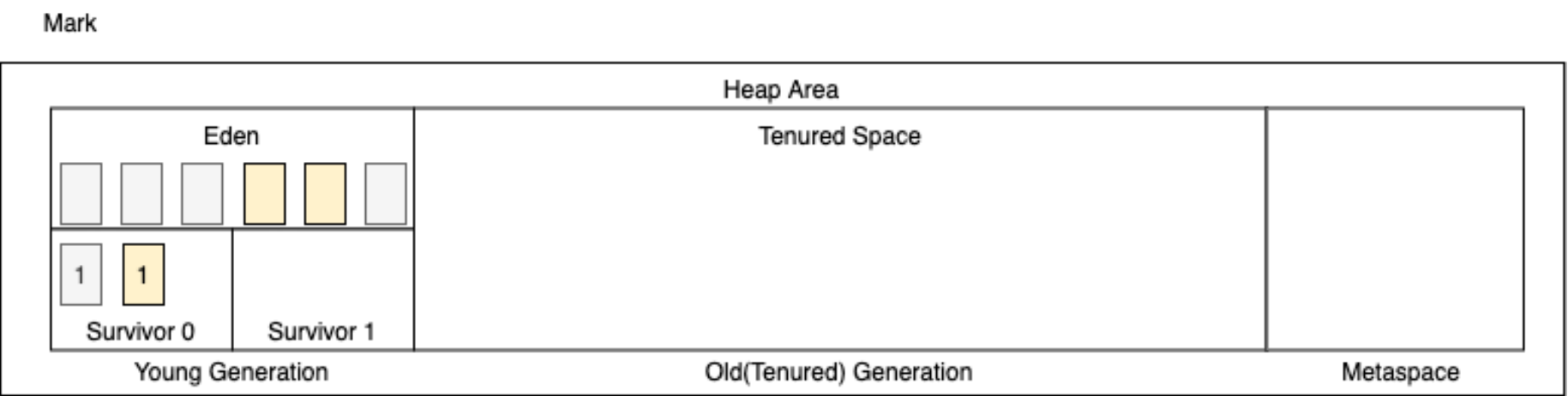
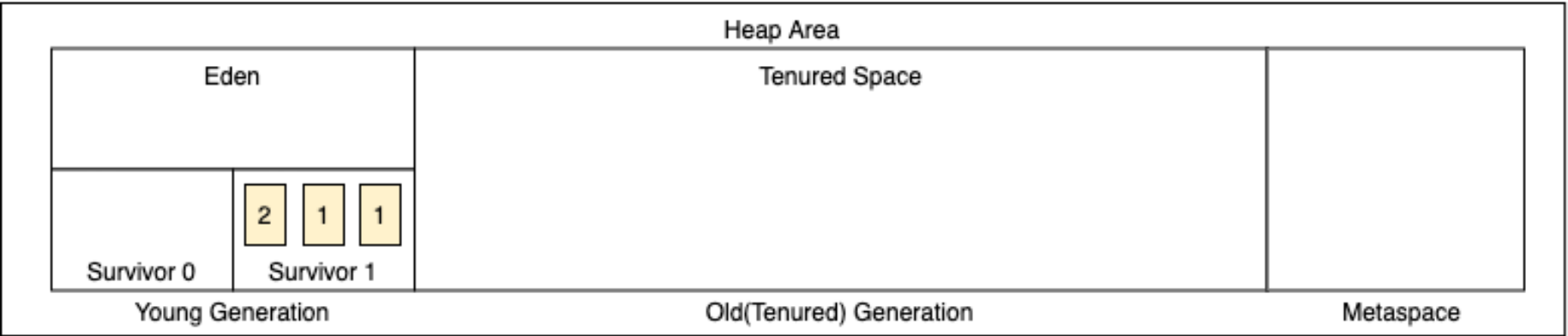
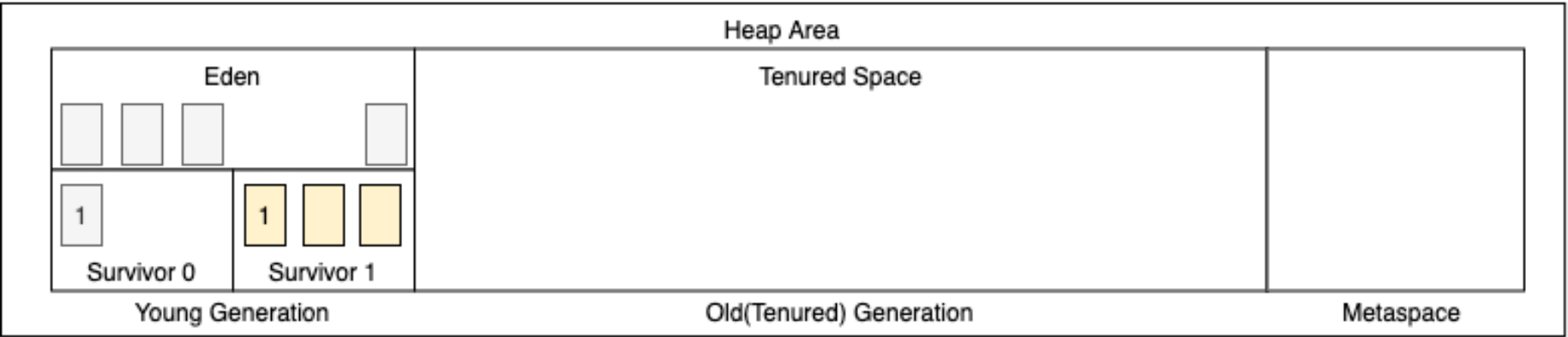
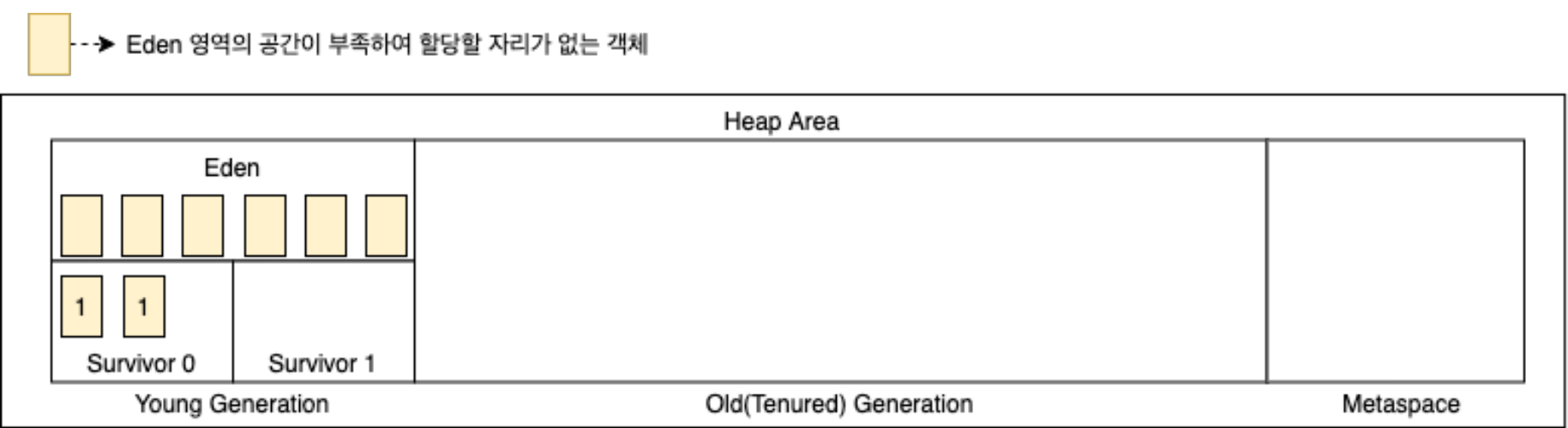
\* Minor GC

1. Unreachable 객체는 GC를 통한 메모리 수거  
- Sweep

2. Survivor0에 보관된 객체의 age를 증가



# Minor GC Repeat



## 3. Garbage Collection

### \* Minor GC Repeat

#### 1. Eden 영역 공간 부족 (상황발생)

#### 2. Mark 작업을 통해 Reachable & UnReachable한 객체를 구분

#### 3. Reachable한 객체를 Survivor1로 이동

#### 4. Sweep 작업을 통해 UnReachable 한 객체의 메모리를 수거

#### 5. Survivor1로 이동한 객체의 age 값을 증가

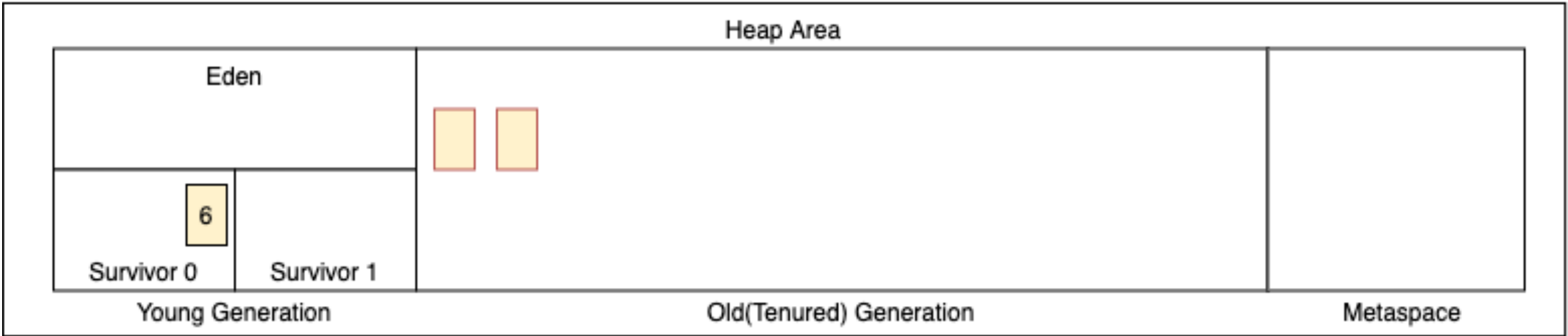
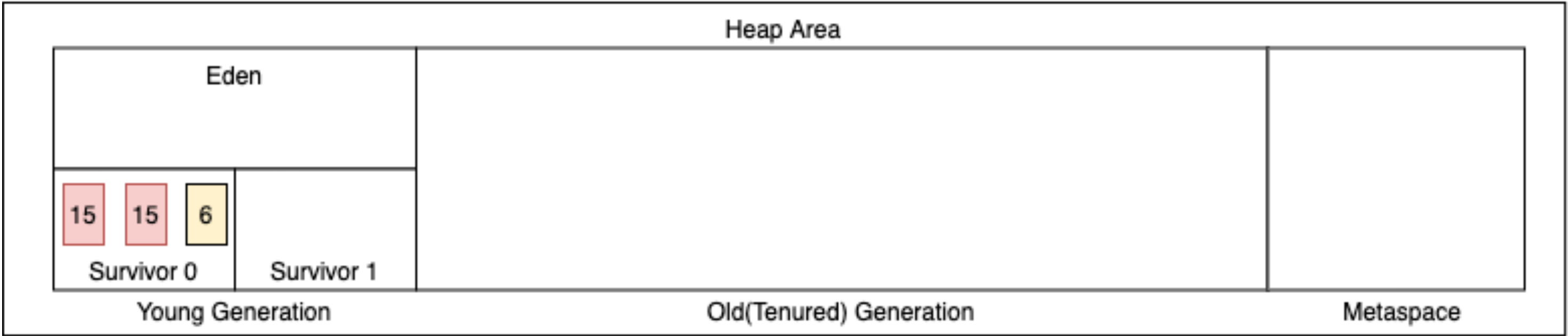
MaxTenuringThreshold

3. Garbage Collection

\* MaxTenuringThreshold

1. Max age threshold에 도달한 메모리는 Old Generation으로 이동

15 -----> Objects reach the max age threshold



# Major GC (Full GC)

## 3. Garbage Collection

\* Major GC (Full GC)

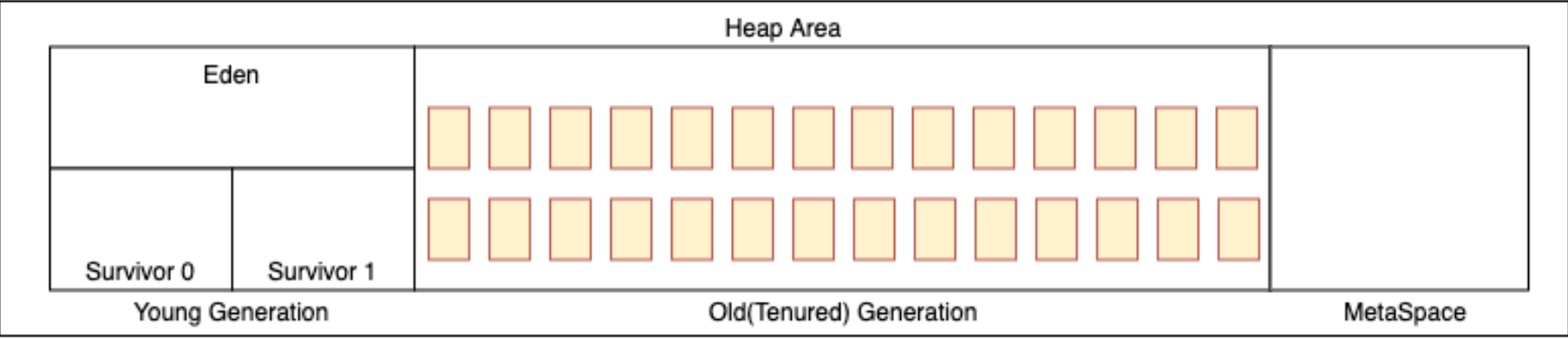
1. Old Generation 영역의 메모리가 부족한 경우 Major GC 발생

2. **Mark** 작업을 통한 구분  
- Reachable or Unreachable

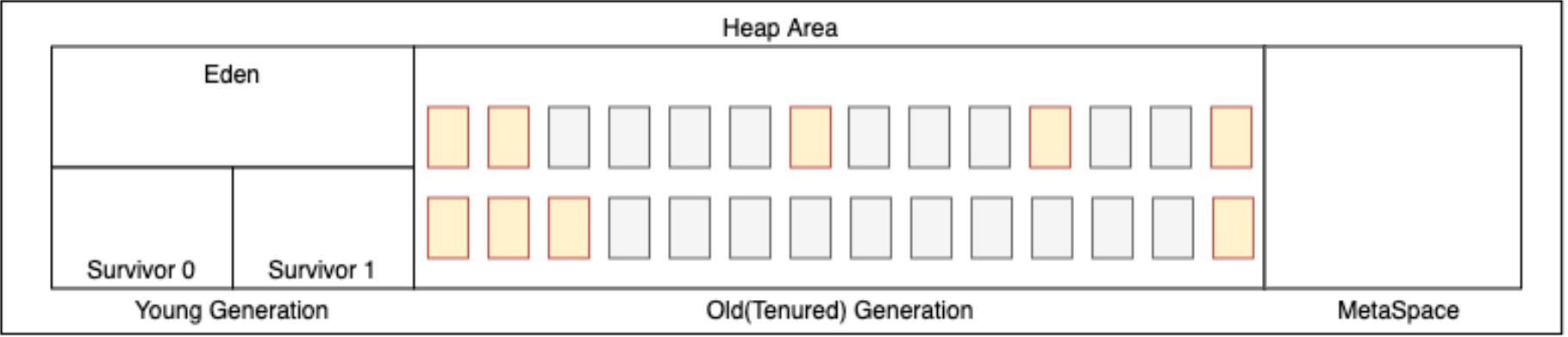
3. **Sweep** 작업을 통한 Unreachable한 객체들의 메모리 수거

4. **Compact** 작업을 통한 메모리 조각 모음

Major GC - 상황 발생



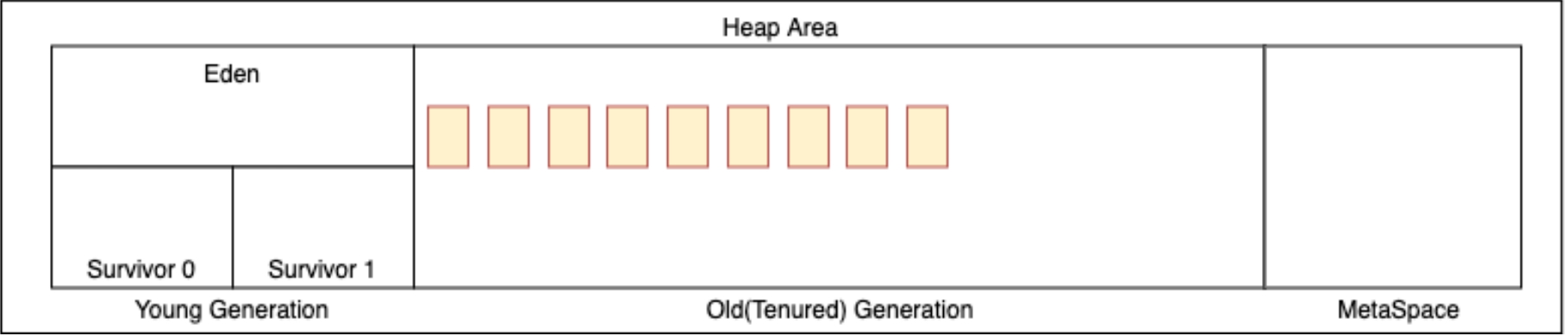
Mark



Sweep



Compact



Garbage Collection Trigger	3. Garbage Collection
<div data-bbox="429 722 2155 1103"><ul style="list-style-type: none"><li>1. System.gc() or Runtime.getRuntime().gc() 실행 시</li><li>2. JVM이 <b>tenured space</b>에 여유 공간이 없다고 판단하는 경우</li><li>3. Minor GC 중 JVM이 <b>eden</b> 또는 <b>survivor</b> 공간에 충분한 공간이 없는 경우</li><li>4. JVM에 <b>MaxMetaspaceSize</b> 옵션을 설정하고 새 클래스를 로드할 공간이 부족한 경우</li></ul></div>	<div data-bbox="2608 165 3268 221"><p>* Garbage Collection Trigger</p></div>

## Garbage Collection 수거 대상

## 3. Garbage Collection

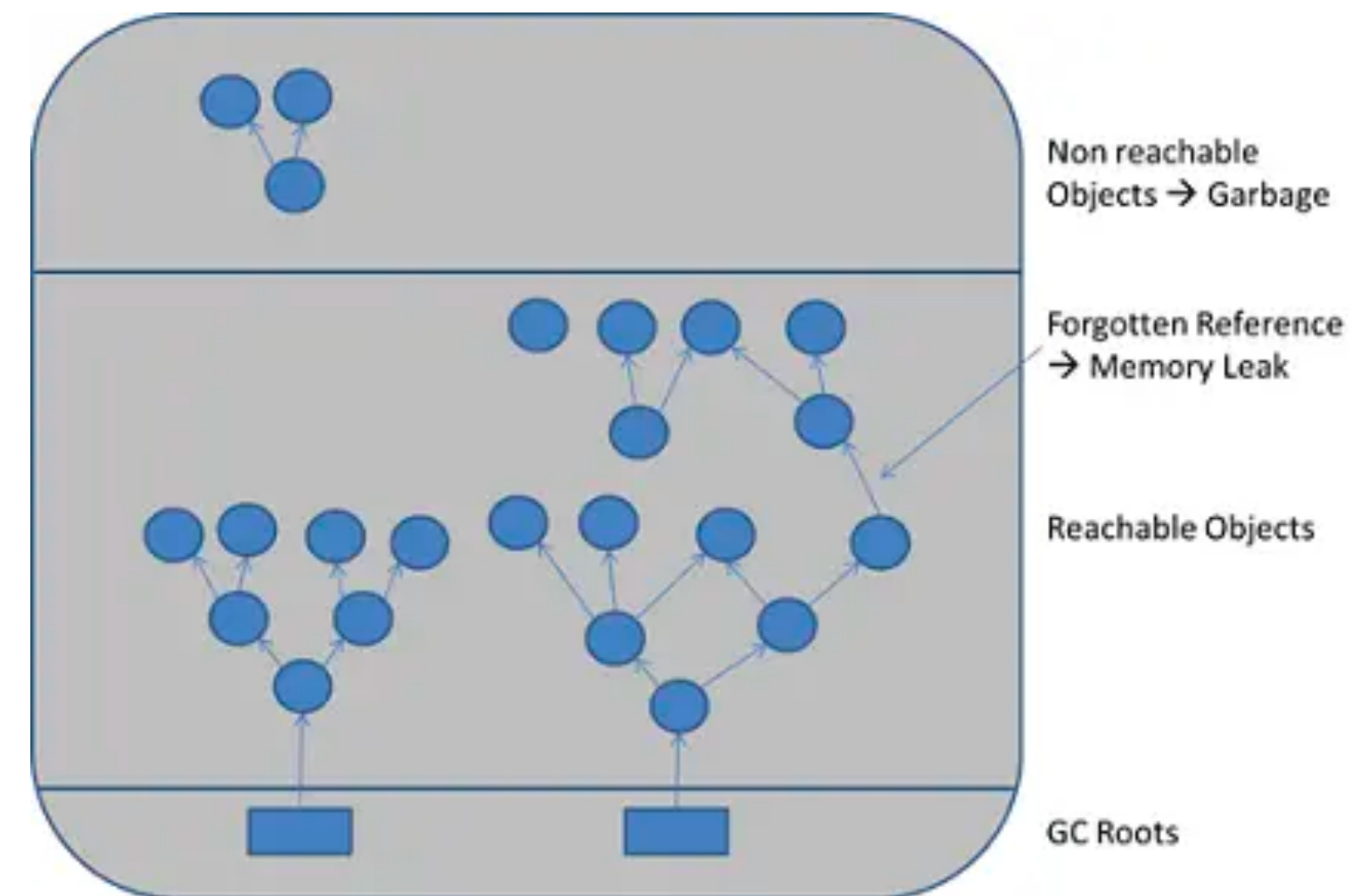
### \* Garbage Collection 수거 대상

- GC Root
- 메인 메서드의 지역변수
- 메인 스레드
- 메인 클래스의 정적 변수

### - GC의 메모리 수거 대상

1. 모든 객체 참조가 **null** 인 경우
2. 객체가 블록 안에서 생성되고 블록이 종료되는 경우(Scope)
3. 부모 객체가 null 이 된 경우, 자식 객체는 자동적으로 GC 대상이 된다.
4. 객체가 Weak 참조만 갖는 경우
5. 객체가 Soft 참조 이지만 메모리가 부족한 경우

Stack or Method(Static) 에서 참조하지 않는 객체



<div data-bbox="906 45 1672 105" data-label="Section-Header"> <h2>아이템 7. 다 쓴 객체 참조를 해제하라</h2> </div>	<div data-bbox="2585 45 2765 105" data-label="Section-Header"> <h3>중간 정리</h3> </div> <div data-bbox="2585 165 3278 697" data-label="List-Group"> <ol style="list-style-type: none"> <li>1. 메모리 누수에 인한 장애의 징조</li> <li>2. 메모리 구조 <ul style="list-style-type: none"> <li>- 자바에서 메모리를 점유하는 방법</li> </ul> </li> <li>3. Garbage Collection</li> <li>4. 메모리 누수의 원인</li> <li>5. 메모리 모니터링</li> </ol> </div>
<div data-bbox="1196 802 1382 855" data-label="Section-Header"> <h3>중간 정리</h3> </div> <div data-bbox="379 859 2195 1073" data-label="List-Group"> <ol style="list-style-type: none"> <li>1. Java는 메모리를 사용하기 위해 <b>Runtime Data Area</b>에 영역별로 저장한다.</li> <li>2. 사용 중인 자원을 일반적으로는 GC가 백그라운드에서 관리를 한다.</li> <li>3. GC는 <b>Minor GC</b>와 <b>Major GC</b>의 동작을 한다.</li> <li>4. GC가 자원을 회수하기 위한 조건은 Stack, Method Area에서 사용하지 않는 자원들이다.</li> </ol> </div>	



<div data-bbox="1062 46 1512 103" data-label="Section-Header"> <h4>4. 메모리 누수의 원인</h4> </div>	<div data-bbox="2578 46 3298 163" data-label="Text"> <p>* 메모리 누수를 발생할 가능성이 있는 경우</p> </div> <div data-bbox="2578 215 3298 1031" data-label="List-Group"> <ol style="list-style-type: none"> <li>1. AutoBoxing</li> <li>2. Cache</li> <li>3. Connection</li> <li>4. CustomKey (객체를 구분)</li> <li>5. Immutable Key</li> <li>6. Internal Data Structure</li> </ol> <p>- 동적 할당이 일어나는 콜 스택</p> <p><u>Example GitHub</u></p> </div>
<div data-bbox="283 853 2292 1022" data-label="Text"> <p>GC는 <b>Unreachable</b> 한 객체는 찾을 수 있지만 <b>Unused</b> 한 객체는 찾을 수 없다.</p> <p><b>Unused</b> 한 객체는 응용 프로그램의 논리에 따라 달라지므로 프로그래머는 비즈니스 코드에 주의해야 한다.</p> </div>	

# AutoBoxing

## 4. 메모리 누수의 원인

### \* AutoBoxing

- Long(Reference Type)

- long(Primitive Type)

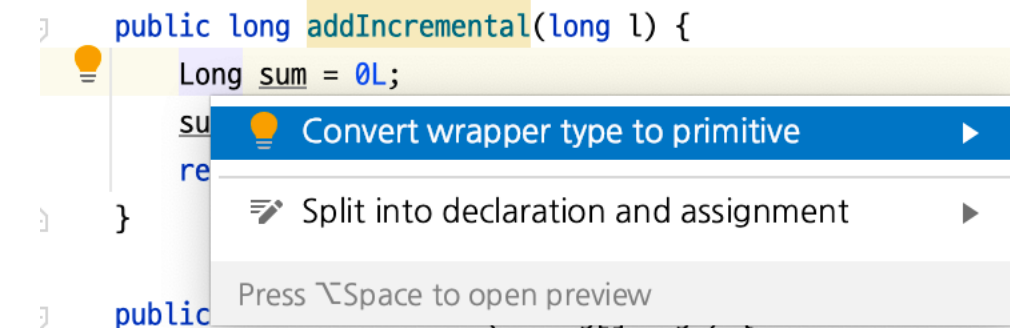
- 문제의 로직

- 100만번 합산하는 로직

- **매번 Long 객체를 생성**

- 해결책

- Primitive Type과 Wrapper Type  
을 혼용해서 사용하지 말 것



```
/**
 * 매 반복 마다 불필요한 AutoBoxing 으로 Memory Leek 발생
 */
public class MemoryLeekAutoBoxing {

    public long addIncremental(long l) {
        Long sum = 0L;
        sum = sum + l;
        return sum;
    }

    public static void main(String[] args) {
        MemoryLeekAutoBoxing autoBoxing = new MemoryLeekAutoBoxing();
        for (long i = 0; i < 1_000_000; i++) {
            autoBoxing.addIncremental(i);
        }
    }
}
```

```
/**
 * 매 반복 마다 불필요한 AutoBoxing 으로 Memory Leek 발생
 */
public class MemoryLeekAutoBoxing {

    public long addIncremental(long l) {
        long sum = 0L;
        sum = sum + l;
        return sum;
    }

    public static void main(String[] args) {
        MemoryLeekAutoBoxing autoBoxing = new MemoryLeekAutoBoxing();
        for (long i = 0; i < 1_000_000; i++) {
            autoBoxing.addIncremental(i);
        }
    }
}
```

```
> Task :java-in-theory:compileJava
> Task :java-in-theory:processResources NO-SOURCE
> Task :java-in-theory:classes

> Task :java-in-theory:MemoryLeekAutoBoxing.main()
[GC (Allocation Failure) 4096K->440K(15872K), 0.0012249 secs]
```

```
> Task :java-in-theory:compileJava
> Task :java-in-theory:processResources NO-SOURCE
> Task :java-in-theory:classes
> Task :java-in-theory:MemoryLeekAutoBoxing.main()
```

# Cache

```
public class MemoryLeekCache {
    private Map<String, String> map = new HashMap<>();

    public void initCache() {
        map.put("Anil", "Work as Engineer");
        map.put("Shamik", "Work as Java Engineer");
        map.put("Ram", "Work as Doctor");
    }

    public Map<String, String> getCache() { return map; }

    public void forEachDisplay() {
        map.keySet().forEach(key -> {
            String val = map.get(key);
            System.out.println(key + " :: " + val);
        });
    }

    public static void main(String[] args) {
        MemoryLeekCache cache = new MemoryLeekCache();
        cache.initCache();
        cache.forEachDisplay();
    }
}
```

## 4. 메모리 누수의 원인

### \* Cache

- 캐시에 저장된 데이터가 필요 없는 경우. 지워야 하지 않을까?

- 문제점
- 전역 변수로 선언된 Map
- 사용할 데이터를 모두 등록
- 사용하지 않아도 이미 등록 되어 있다.

- 해결책
- WeakHashMap

Connection	4. 메모리 누수의 원인
<div>외부 자원에 접근 하기 위한 라이브러리<ul style="list-style-type: none"><li>- <u>URLConnection</u></li><li>- <u>FileOutputStream</u></li><li>- <u>JDBCConnection</u></li></ul></div>	<div>* Connection</div> <div>- 외부 디바이스에 연결하기 위한 Connection을 사용하는 경우</div> <div>- 문제점<ul style="list-style-type: none"><li>- 필요한 경우 자원에 API를 통한 연결</li><li>- 작업 종료 시 자원의 연결을 끊지 않음</li></ul></div> <div>- 해결책<ul style="list-style-type: none"><li>- 자원의 연결 이후 연결을 끊는 로직을 호출</li><li>- try-with-resources 를 사용</li></ul></div>

# CustomKey

```
public class MemoryLeekCustomKey {

    private final String name;

    public MemoryLeekCustomKey(String name) {
        this.name = name;
    }

    public static void main(String[] args) {
        Map<MemoryLeekCustomKey, String> map = new HashMap<>();

        // 신규 이름 등록
        MemoryLeekCustomKey name = new MemoryLeekCustomKey( name: "Shamik");
        map.put(name, "Shamik Mitra");

        // 같은 이름으로 조회
        MemoryLeekCustomKey sameName = new MemoryLeekCustomKey( name: "Shamik");
        String value = map.get(sameName);

        // 해당 객체의 hashCode가 동일하지 않아 찾을 수 없음
        System.out.println("Missing equals and hascode so value is not accessible from Map " + value);
    }
}
```

```
public class MemoryLeekCustomKey {

    private final String name;

    public MemoryLeekCustomKey(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        MemoryLeekCustomKey that = (MemoryLeekCustomKey) o;
        return Objects.equals(name, that.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }

    public static void main(String[] args) {
        Map<MemoryLeekCustomKey, String> map = new HashMap<>();

        // 신규 이름 등록
        MemoryLeekCustomKey name = new MemoryLeekCustomKey( name: "Shamik");
        map.put(name, "Shamik Mitra");

        // 같은 이름으로 조회
        MemoryLeekCustomKey sameName = new MemoryLeekCustomKey( name: "Shamik");
        String value = map.get(sameName);

        // 해당 객체의 hashCode가 동일하지 않아 찾을 수 없음
        System.out.println("Missing equals and hascode so value is not accessible from Map " + value);
    }
}
```

# 4. 메모리 누수의 원인

## \* CustomKey

- 사용자 정의 Key값을 사용하는 경우

- 문제점

- 특정 Key 값으로 Map 내에 데이터를 조회 하는 경우 Key 의 hashCode가 일치하지 않아 조회가 되지 않는 문제

- 해결책

- equals() and hashCode() 를 통한 객체를 비교



# Immutable Key

```
public class MemoryLeekMutableCustomKey {

    private String name;

    public MemoryLeekMutableCustomKey(String name) { this.name = name; }

    public String getName() { return name; }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object o) {...}

    @Override
    public int hashCode() { return Objects.hash(getName()); }

    public static void main(String[] args) {
        MemoryLeekMutableCustomKey key = new MemoryLeekMutableCustomKey( name: "Shamik");

        Map<MemoryLeekMutableCustomKey, String> map = new HashMap<>();
        map.put(key, "Shamik Mitra");

        MemoryLeekMutableCustomKey refKey = new MemoryLeekMutableCustomKey( name: "Shamik");
        String val = map.get(refKey);
        System.out.println("Value Found " + val);
        key.setName("Bubun");

        String val1 = map.get(refKey);
        System.out.println("Due to MutableKey value not found " + val1);
    }
}
```

# 4. 메모리 누수의 원인

## \* Immutable Key

- Consistency Key를 사용해야 하는 이유

- 문제점  
- 사용자 정의된 Key값을 이용해 데이터  
를 관리하는 상황에서 이미 저장된 데이터  
의 Key 값을 변경하는 경우 기존 Key 값  
으로 검색이 불가능해지는 문제

- 해결책  
- Key: Value 값으로 저장되는 데이터  
구조는 이미 저장된 데이터의 Key값을 변  
경할 수 없어야 한다.  
- Key 값을 수정할 수 있는 인터페이스  
를 제공하지 않는다.

## Internal Data Structure

### 4. 메모리 누수의 원인

#### \* Internal Data Structure

- 클래스 자체에서 배열의 메모리를 관리하는 경우 메모리 누수가 발생 가능

- 책에서 예시로 제공하는 내용

```
For positive values of minimumCapacity, this method behaves like ensureCapacity, however it is never synchronized. If minimumCapacity is non positive due to numeric overflow, this method throws OutOfMemoryError.

private void ensureCapacityInternal(int minimumCapacity) {
    // overflow-conscious code
    if (minimumCapacity - value.length > 0) {
        value = Arrays.copyOf(value,
            newCapacity(minimumCapacity));
    }
}

The maximum size of array to allocate (unless necessary). Some VMs reserve some header words in an array. Attempts to allocate larger arrays may result in OutOfMemoryError: Requested array size exceeds VM limit

private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

Returns a capacity at least as large as the given minimum capacity. Returns the current capacity increased by the same amount + 2 if that suffices. Will not return a capacity greater than MAX_ARRAY_SIZE unless the given minimum capacity is greater than that.
Params: minCapacity - the desired minimum capacity
Throws: OutOfMemoryError - if minCapacity is less than zero or greater than Integer.MAX_VALUE

private int newCapacity(int minCapacity) {...}

private int hugeCapacity(int minCapacity) {...}

Attempts to reduce storage used for the character sequence. If the buffer is larger than necessary to hold its current sequence of characters, then it may be resized to become more space efficient. Calling this method may, but is not required to, affect the value returned by a subsequent call to the capacity() method.

public void trimToSize() {
    if (count < value.length) {
        value = Arrays.copyOf(value, count);
    }
}
```

- 클래스 내에서 자료구조를 배열로 사용하고 데이터의 추가와 삭제 시 배열 사이즈에 대한 크기를 조절하지 않는 경우에 발생하는 문제

<div data-bbox="906 45 1672 103" data-label="Section-Header"><h2>아이템 7. 다 쓴 객체 참조를 해제하라</h2></div>	<div data-bbox="2578 45 2765 103" data-label="Section-Header"><h3>중간 정리</h3></div> <div data-bbox="2578 159 3275 703" data-label="List-Group"><ol style="list-style-type: none"><li>1. 메모리 누수에 인한 장애의 징조</li><li>2. 메모리 구조<ul style="list-style-type: none"><li>- 자바에서 메모리를 점유하는 방법</li></ul></li><li>3. Garbage Collection</li><li>4. 메모리 누수의 원인</li><li>5. 메모리 모니터링</li></ol></div>
<div data-bbox="1176 774 1402 832" data-label="Section-Header"><h3>마지막 정리</h3></div> <div data-bbox="226 832 2349 1099" data-label="List-Group"><ol style="list-style-type: none"><li>1. Java는 메모리를 사용하기 위해 Runtime Data Area에 영역별로 저장한다.</li><li>2. 사용 중인 자원을 일반적으로는 GC가 백그라운드에서 관리를 한다.</li><li>3. GC는 Minor GC와 Major GC의 동작을 한다.</li><li>4. GC가 자원을 회수하기 위한 조건은 Stack, Method Area에서 사용하지 않는 자원들이다.</li><li>5. 메모리의 누수는 특정 시점에 발생하는 것이 아니라 평소에 관리할 수 있는 방법은 코딩 방법과 모니터링이다.</li></ol></div>	



5. 메모리 모니터링

\* 모니터링 도구

1. jstat

- YGC: Minor GC 발생 횟수
- YGCT: Minor GC 누적 시간
- FGC: Major GC 발생 횟수
- FGCT: Major GC 누적 시간

2. VisualVM + Visual GC

3. 그 외

... 4 more						jstat -gcutil 93527 2s						
S0	S1	E	O	M	CCS	YGC	YGCT	FGC	FGCT	CGC	CGCT	GCT
0.00	3.12	33.47	7.28	89.90	80.61	11	0.025	0	0.000	-	-	0.025
5.51	0.00	43.27	7.28	89.90	80.61	12	0.027	0	0.000	-	-	0.027
0.00	6.70	59.04	7.28	89.90	80.61	13	0.029	0	0.000	-	-	0.029
3.12	0.00	68.18	7.28	89.90	80.61	14	0.030	0	0.000	-	-	0.030
0.00	8.63	83.33	7.28	89.90	80.61	15	0.031	0	0.000	-	-	0.031
6.70	0.00	93.42	7.35	89.90	80.61	16	0.036	0	0.000	-	-	0.036
6.70	0.00	7.54	7.35	89.90	80.61	18	0.039	0	0.000	-	-	0.039
0.00	3.12	16.33	7.35	89.90	80.61	19	0.040	0	0.000	-	-	0.040
6.70	0.00	30.69	7.35	89.90	80.61	20	0.042	0	0.000	-	-	0.042
0.00	3.12	39.65	7.35	89.90	80.61	21	0.045	0	0.000	-	-	0.045
6.70	0.00	47.79	7.35	89.90	80.61	22	0.046	0	0.000	-	-	0.046
0.00	3.12	61.58	7.35	89.90	80.61	23	0.047	0	0.000	-	-	0.047
6.70	0.00	76.30	7.35	89.90	80.61	24	0.049	0	0.000	-	-	0.049
0.00	3.12	84.64	7.35	89.90	80.61	25	0.050	0	0.000	-	-	0.050
13.40	0.00	93.41	7.35	89.90	80.61	26	0.052	0	0.000	-	-	0.052
13.40	0.00	7.54	7.35	89.90	80.61	28	0.054	0	0.000	-	-	0.054
0.00	6.25	16.33	7.35	89.90	80.61	29	0.058	0	0.000	-	-	0.058
13.40	0.00	24.98	7.35	89.90	80.61	30	0.060	0	0.000	-	-	0.060
0.00	6.25	39.65	7.35	89.90	80.61	31	0.064	0	0.000	-	-	0.064
13.40	0.00	53.50	7.35	89.90	80.61	32	0.066	0	0.000	-	-	0.066
0.00	6.25	61.58	7.35	89.90	80.61	33	0.067	0	0.000	-	-	0.067
13.40	0.00	76.30	7.35	90.03	80.61	34	0.070	0	0.000	-	-	0.070

