GC (Garbage Collection)

가비지 컬렉션

GC (Garbage Collection)

쓰레기 수집

: 더 이상 참조되지 않는 객체를 수거해서 메모리를 확보

GC의 필요성

- 메모리 할당을 해야하는데 메모리가 없다면? → OutOfMemoryError 발생
- C와 Cpp 같은 경우는 개발자가 직접 메모리를 관리해야한다.
- 그러나 Java, Python, C#, js의 경우는 GC가 메모리를 관리해주기 때문에 개발에만 집중 가능

GC의 대상이 되는 객체

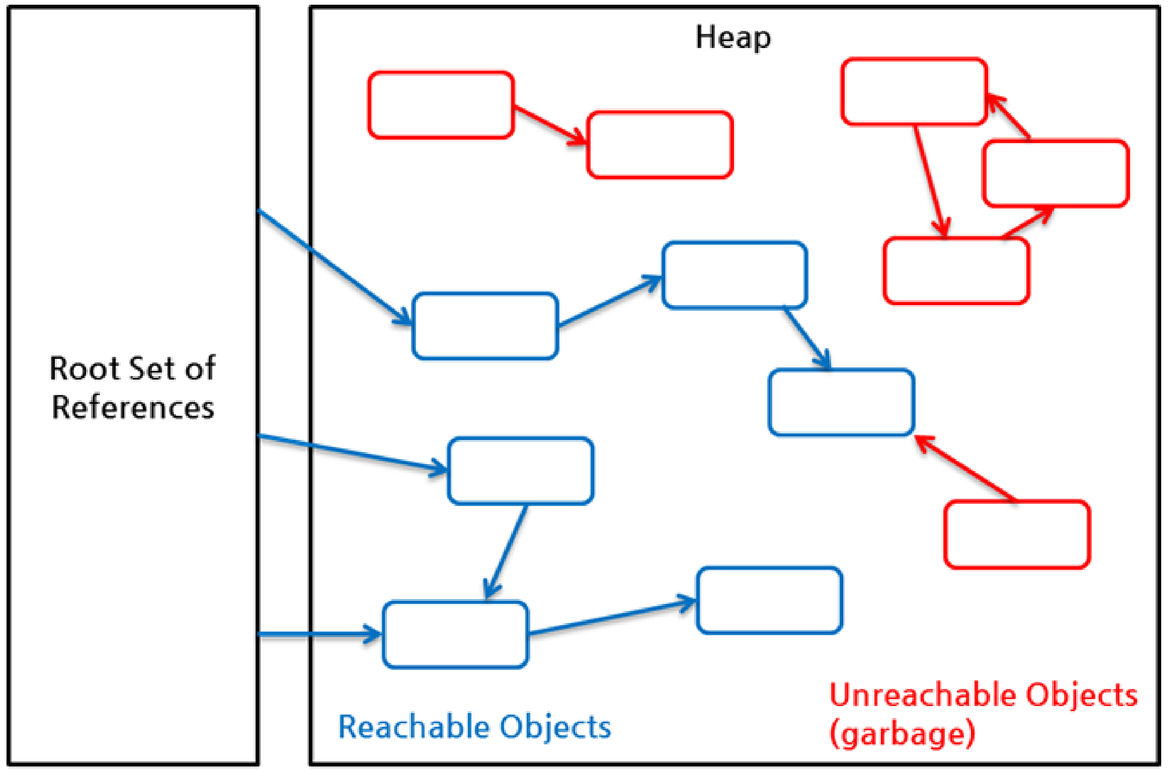
Reference Counting

1. 객체가 참조되고 있는 숫자를 카운팅, 0이 되면 수거해간다.

Mark & Sweep

1. Reachable : 객체가 참조되고 있는 상태
2. UnReachable : 객체가 참조되지 않고 있는 상태 (GC의 대상이 됨)

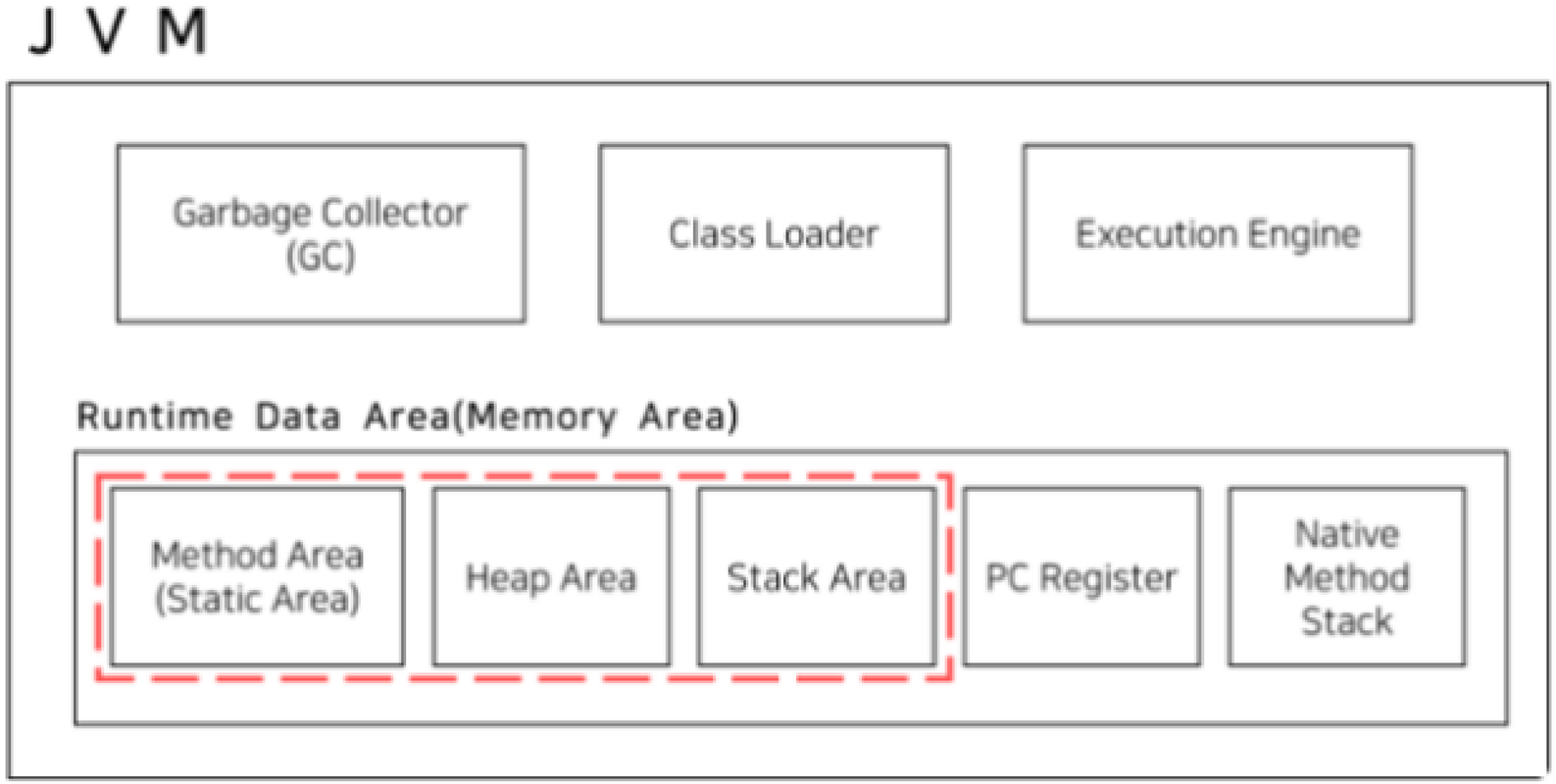
참조 & 비참조


Heap
Root Set of References
Reachable Objects
Unreachable Objects (garbage)

UnReachable 판단

1. 모든 객체 참조가 Null인 경우
2. 객체가 블록 안에서 생성되고 블록이 종료된 경우
3. 부모 객체가 Null인 경우, 자식 객체는 자동으로 GC 대상
4. 객체가 약한 참조만 가지고 있는 경우
5. 객체가 Soft 참조이지만 메모리 부족이 발생한 경우

강한 참조? 약한 참조? Soft참조?

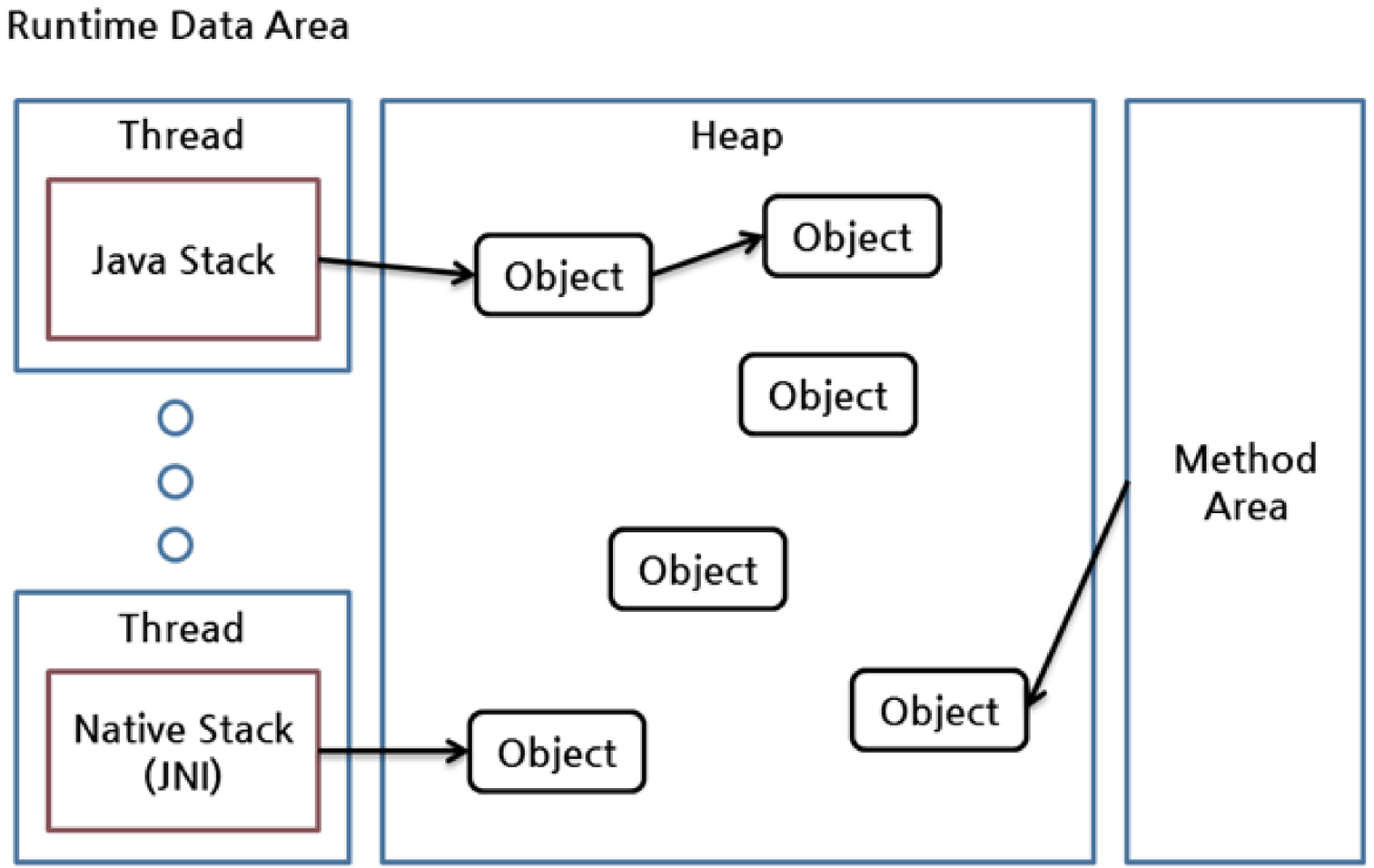


Heap Area

- 동적으로 생성된 객체가 저장되는 공간

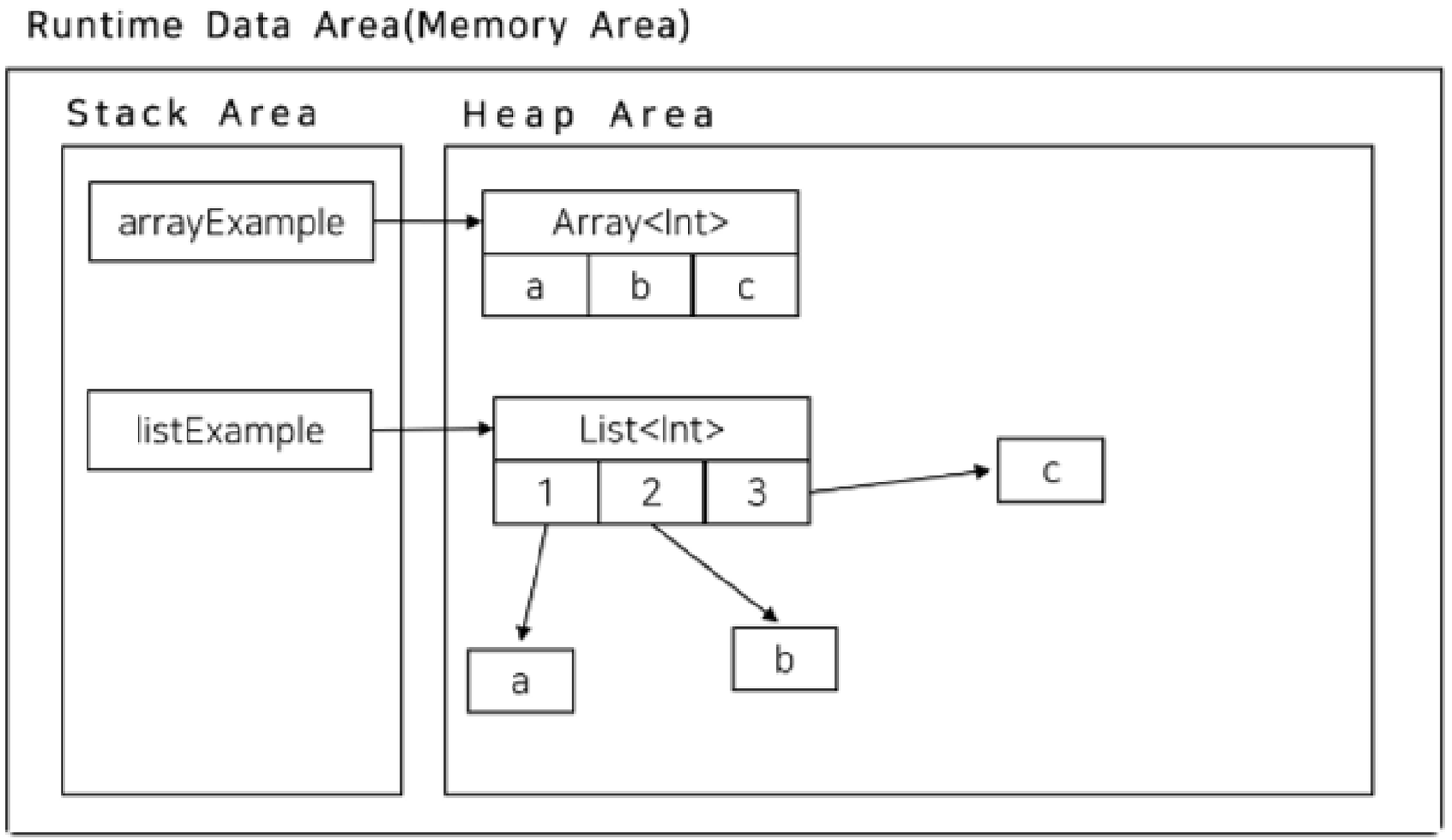
- 메서드가 실행되면서 Stack영역에는 참조값만을 저장해놓고 Heap Area에 객체 데이터를 저장해 놓는다. 메서드가 실행되면 Stack영역에는 참조값만을 저장해놓고 Heap Area에 객체 데이터를 저장한다.

객체 참조


Runtime Data Area
Thread
Java Stack
Thread
Native Stack
(JNI)
Heap
Object
Object
Object
Object
Object
Object
Method
Area

객체 참조

```
List<String> list = List.of("a", "b", "c");
String[] arr = Arrays.of("a", "b", "c");
```



강한 참조

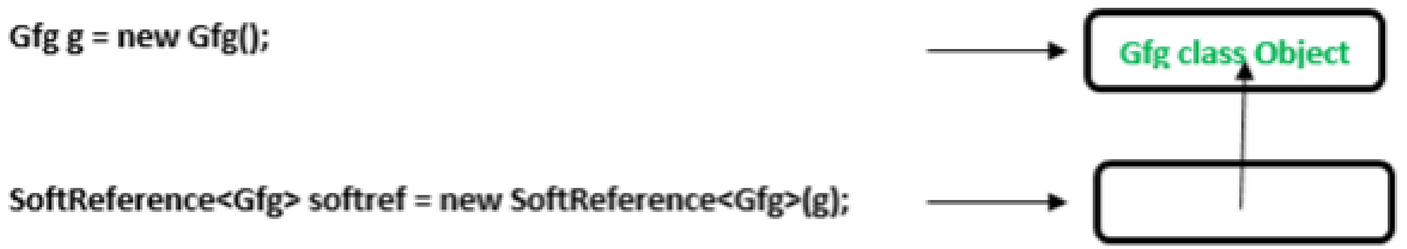
```
MyClass obj = new MyClass(); // GC 대상 X

obj = null; // GC 대상 O
```

- gc의 제거대상 X

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class Main {
    static class MyClass {
        public MyClass() {
        }
    }
    public static void main(String[] args) throws IOException {
        MyClass myClass = new MyClass();

        List<MyClass> softRef = new ArrayList<MyClass>(List.of(myClass));
        System.out.println(myClass); // Main$MyClass@56cbfb61

        myClass = null;
        System.out.println(myClass); // null
    }
}
```

Soft 참조

```
MyClass ref = new MyClass();

SoftReference<MyClass> softRef = new SoftReference<MyClass>(ref);
```



Soft 참조

JVM 메모리가

부족하지 않다면

굳이 제거하지는 않음

```
MyClass ref = new MyClass();

SoftReference<MyClass> softRef = new SoftReference<MyClass>(ref);
```

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class Main {
    static class MyClass {
        public MyClass() {
        }
    }
    public static void main(String[] args) throws IOException {
        MyClass myClass = new MyClass();

        SoftReference<MyClass> softRef = new SoftReference<MyClass>(myClass);
        System.out.println(myClass); // Main$MyClass@1134affc
        System.out.println(softRef); // Main$MyClass@1134affc

        myClass = null;
        System.out.println(myClass); // null
        System.out.println(softRef); // Main$MyClass@1134affc
    }
}
```

Soft 참조

```java
import java.io.IOException;
import java.lang.ref.SoftReference;
import java.lang.ref.WeakReference;

public class Main {
    static class MyClass {
        String name;
        public MyClass(String name) {
            this.name = name;
        }

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

        public String getName() {
            return this.name;
        }
    }
    public static void main(String[] args) throws IOException {
        MyClass myClass = new MyClass("권민우");

        SoftReference<MyClass> newClass = new SoftReference<MyClass>(myClass);

        myClass.update("건미누");

        System.out.println(myClass.getName()); // 건미누

        myClass = null;

        System.gc();
        System.out.println(newClass.get().getName()); // 건미누
    }
}
```

Weak 참조

```
MyClass ref = new MyClass();

WeakReference<MyClass> weakRef = new WeakReference<MyClass>(ref));
```

무조건적인 수거 대상

```
    public static void main(String[] args) throws IOException {
        MyClass myClass = new MyClass("권민우");

        WeakReference<MyClass> newClass = new WeakReference<MyClass>(myClass);

        myClass.update("건미누");

        System.out.println(myClass.getName()); // 건미누

        myClass = null;

        System.gc();
        System.out.println(newClass.get().getName()); // NullPointExceptionError
    }
}
```

캐시에서 사용되는 WeakReference

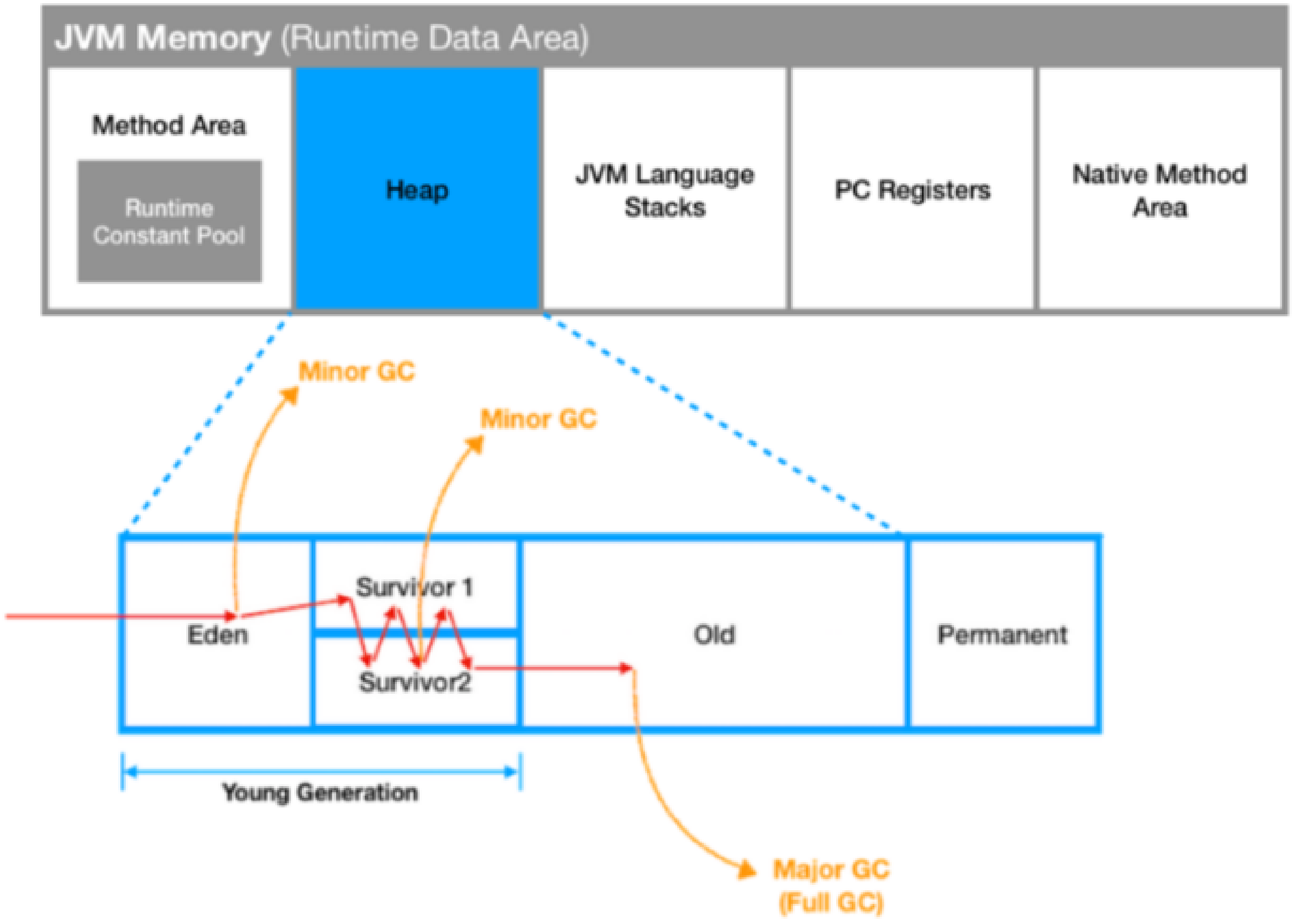
```java
import java.lang.ref.WeakReference;
import java.util.HashMap;
import java.util.Map;

public class Cache<T> {
    private final Map<String, WeakReference<T>> cacheMap;

    public Cache() {
        cacheMap = new HashMap<>();
    }

    public T get(String key) {
        T value = null;
        WeakReference<T> weakRef = cacheMap.get(key);
        if (weakRef != null) {
            value = weakRef.get();
            if (value == null) {
                cacheMap.remove(key); // 참조된 객체가 garbage collector에 의해 수집됨
            }
        }
        return value;
    }

    public void put(String key, T value) {
        cacheMap.put(key, new WeakReference<>(value));
    }
}
```

UnReachable 판단

1. 모든 객체 참조가 Null인 경우
2. 객체가 블록 안에서 생성되고 블록이 종료된 경우
3. 부모 객체가 Null인 경우, 자식 객체는 자동으로 GC 대상
4. 객체가 약한 참조만 가지고 있는 경우
5. 객체가 Soft 참조이지만 메모리 부족이 발생한 경우

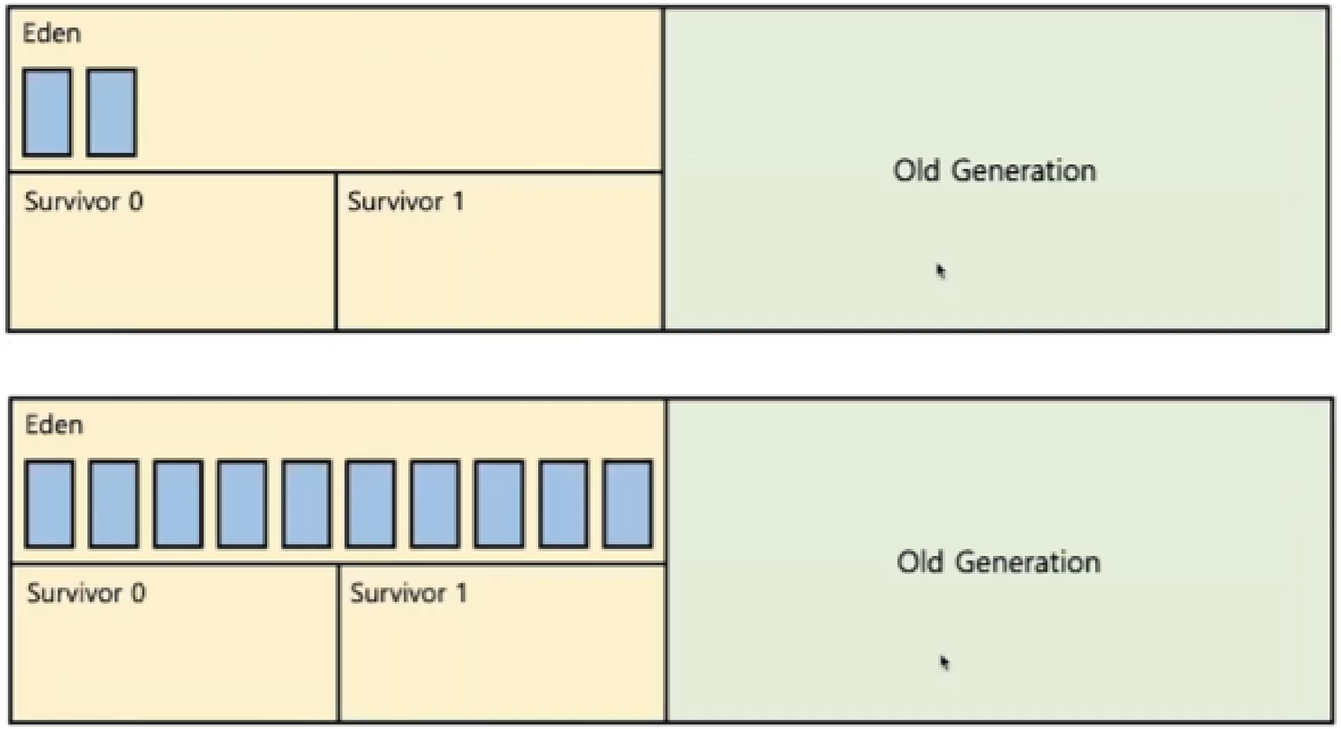
GC 동작과정

java 8



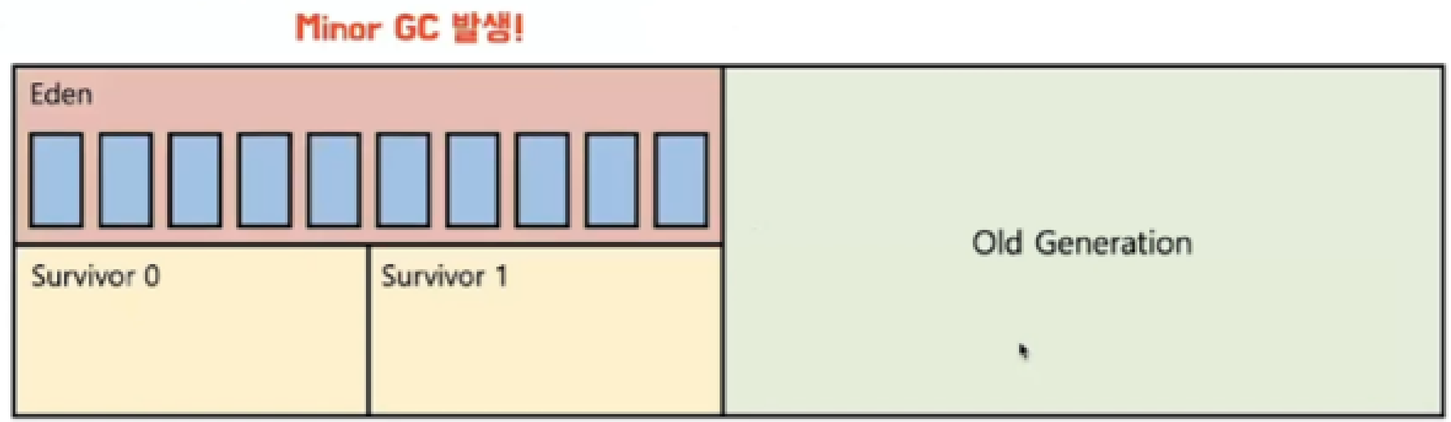
minor GC 동작과정

1. 새로 객체가 생성되며 Eden이 가득차게 된다.



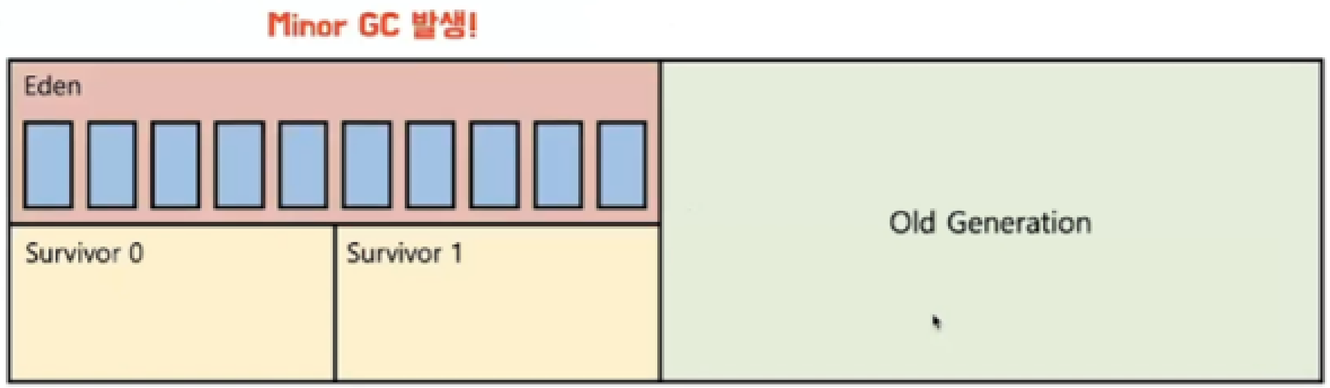
minor GC 동작과정

2. Eden이 가득차게 되면 Minor GC 실행



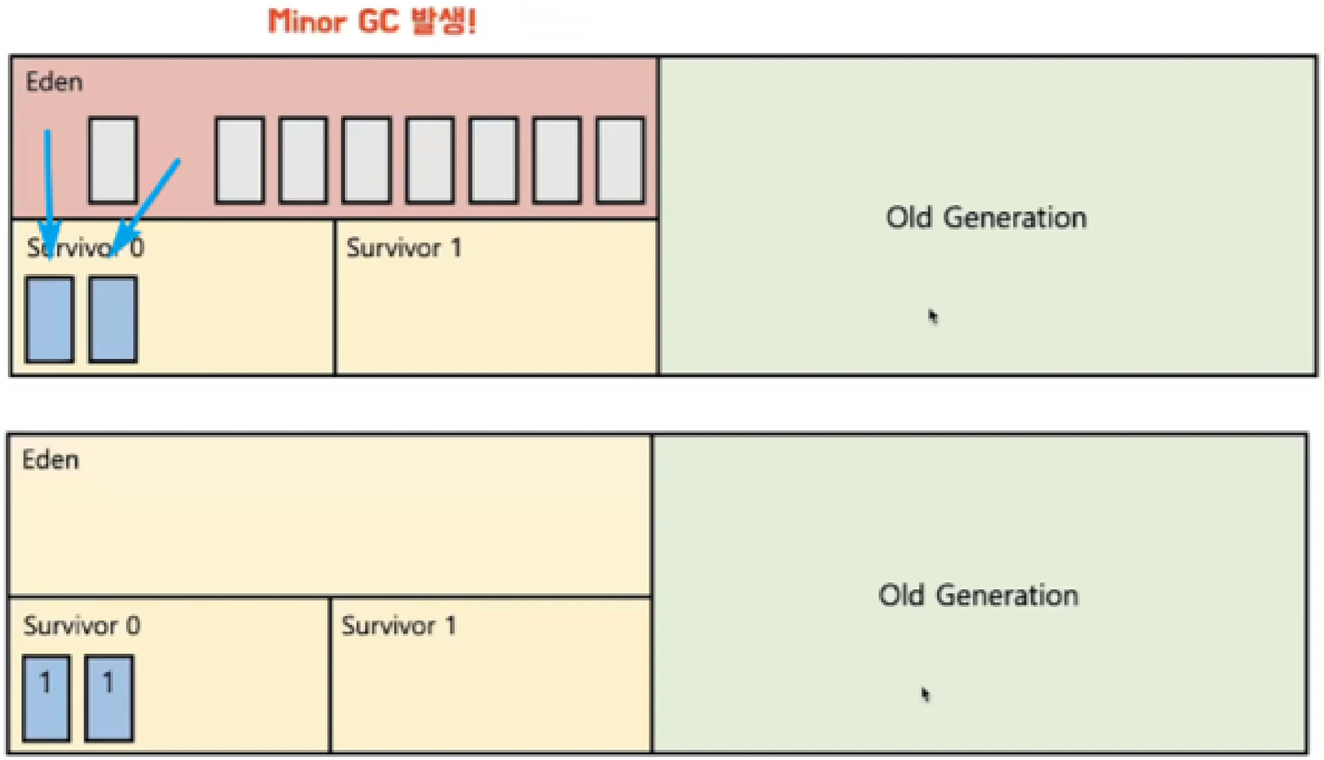
minor GC 동작과정

2. Eden이 가득차게 되면 Minor GC 실행



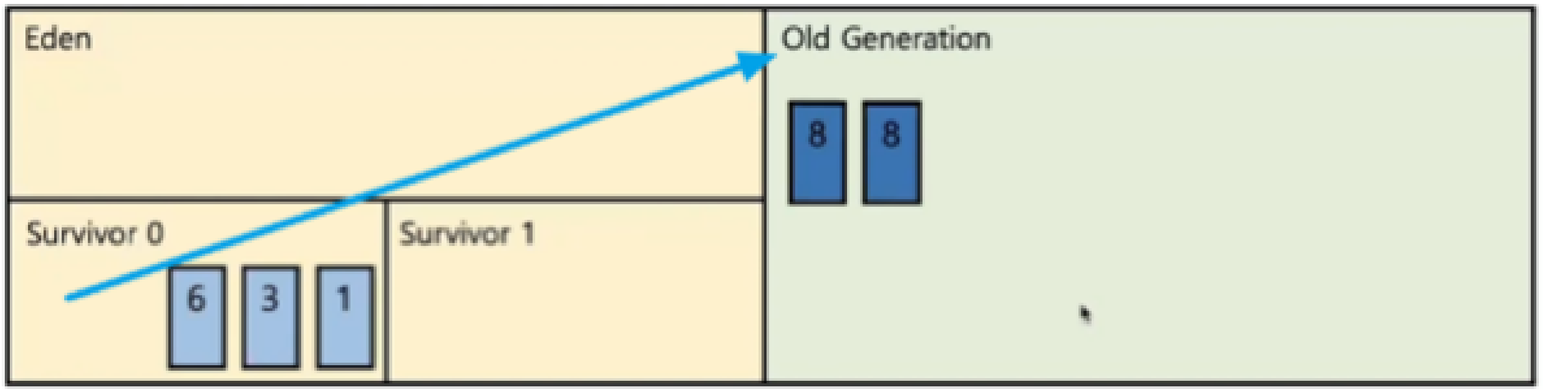
minor GC 동작과정

3. Mark 동작으로 reachable 객체 탐색 후 살아남은 객체는 Survivor영역으로 이동.
   임계값 증가



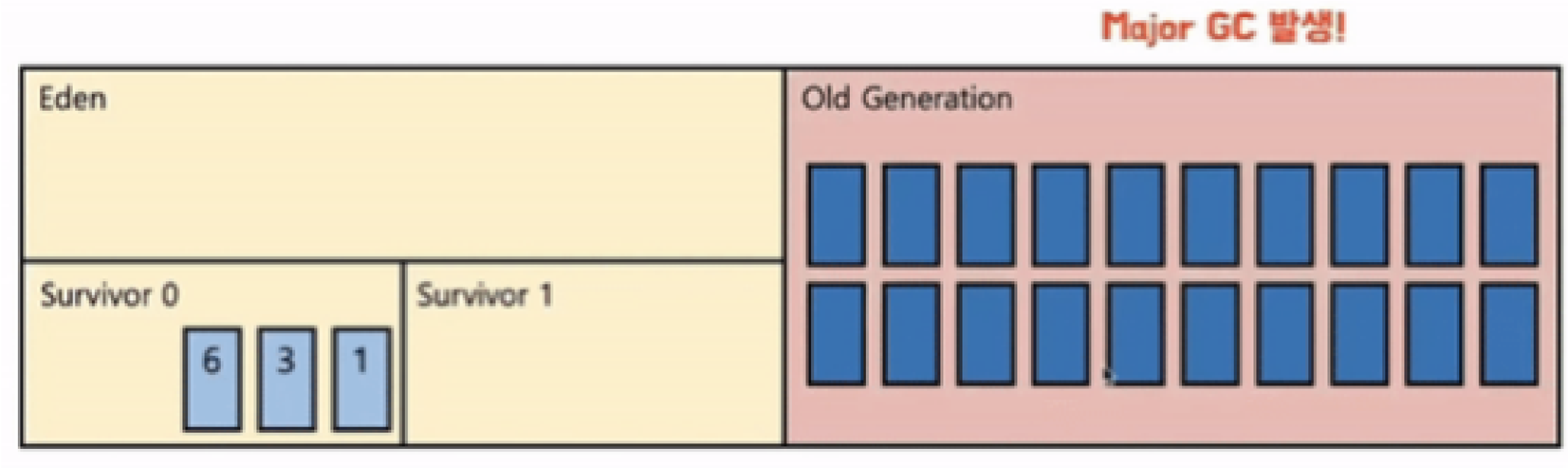
major GC 동작과정

1. 설정한 임계값에 다다르면 Old Generation 구역으로 이동



major GC 동작과정

2. 꽉 차게 되면 Major GC 발생



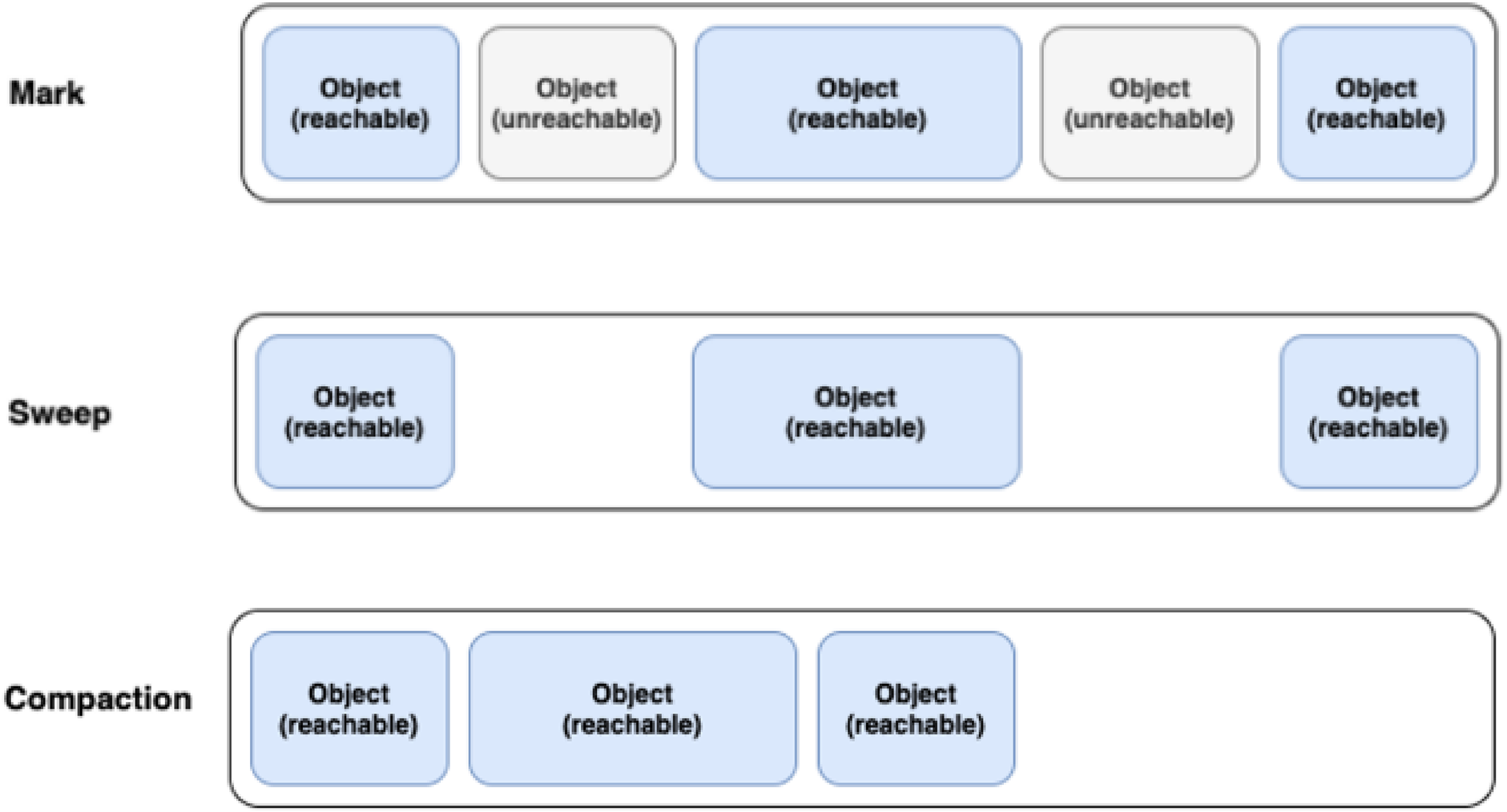
GC의 청소방식

세 가지 프로세스

Mark

Sweep

Compaction


Mark
Object (reachable)
Object (unreachable)
Object (reachable)
Object (unreachable)
Object (reachable)
Sweep
Object (reachable)
Object (reachable)
Object (reachable)
Compaction
Object (reachable)
Object (reachable)
Object (reachable)

메모리를 자동으로 청소해주는 고마운 GC.

단점은 없는걸까?

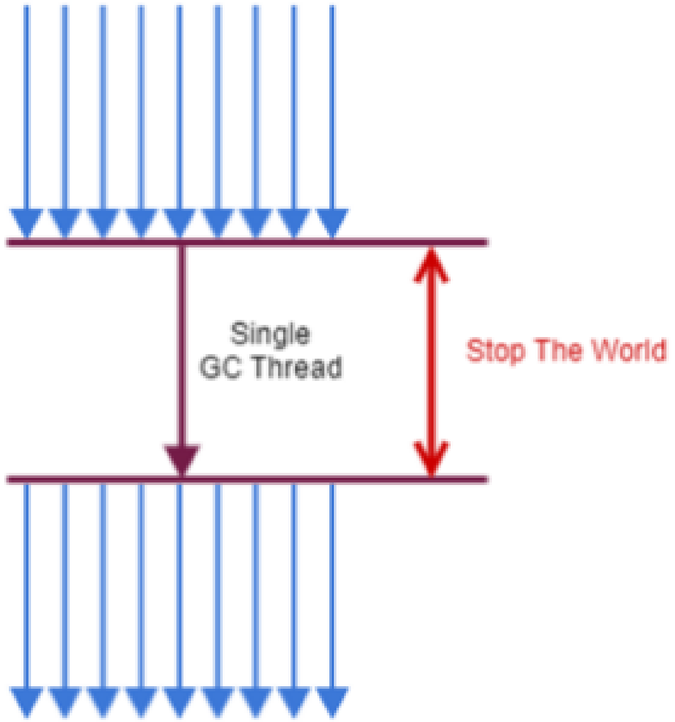
GC의 단점

1. 메모리 해제의 정확한 타이밍을 알 수 없음

2. Stop-The-World : GC가 동작하는 동안 다른 동작은 모두 멈춤. (오버헤드 발생)

Stop-the-world

Serial GC



GC의 종류

Serial GC : 가장 태초의 GC, 싱글 스레드 처리

Parallel GC : 기본 동작은 같으나, YG영역에서 멀티스레드로 GC

Parallel Old GC : 기본 동작은 같으나, 전 영역에서 멀티스레드로 GC

CMS GC : 스레드를 유지하면서 마크를 진행, Compact과정이 X

G1 GC : 영역의 구분을 없애고, Region이라는 메모리 위치 개념 도입

ZGC : CMS의 단점을 개선, 메모리 구조 Region, 64bit 운영체제에서만 사용가능

끗