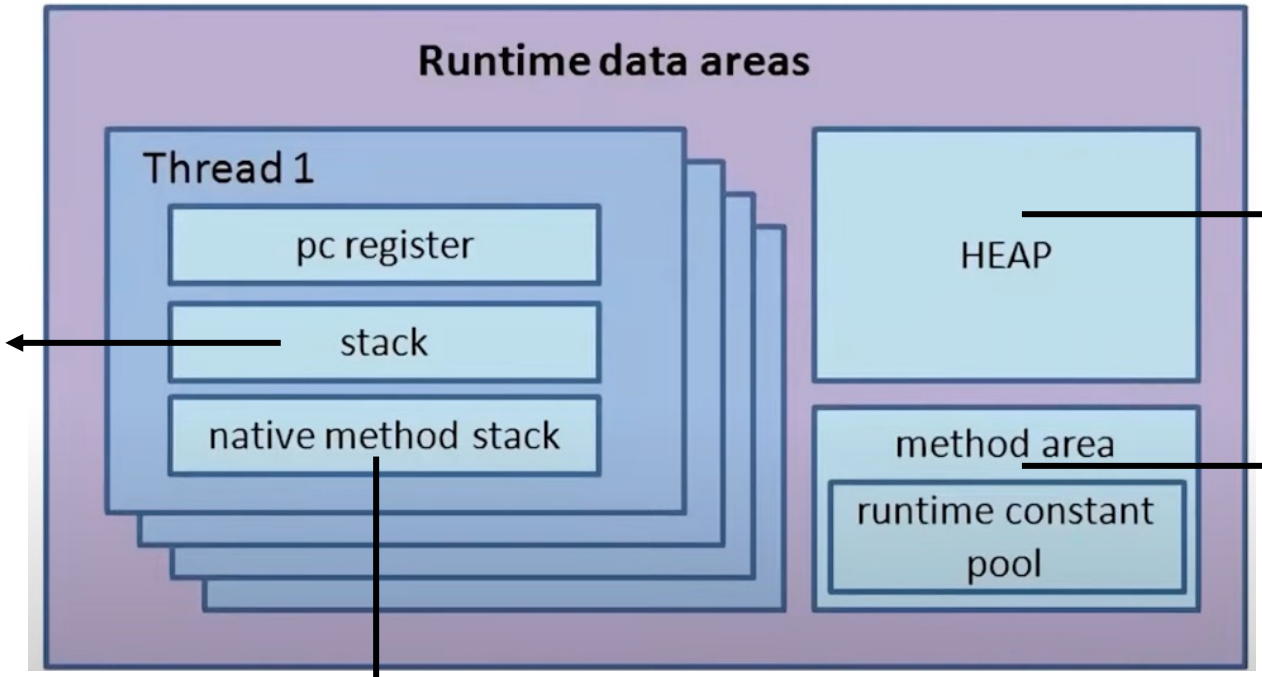
Effective Java

Item 7 : 다 쓴 객체 참조를 해제하라
Item 8 : finalizer와 cleaner 사용을 피하라
Item 9 : try-with-resources를 사용하라

Item 7

다 쓴 객체 참조를 해제하라

JVM 메모리 영역



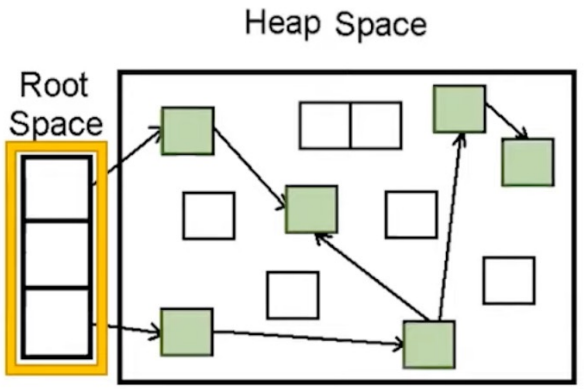
기본 타입 변수
참조 타입 변수

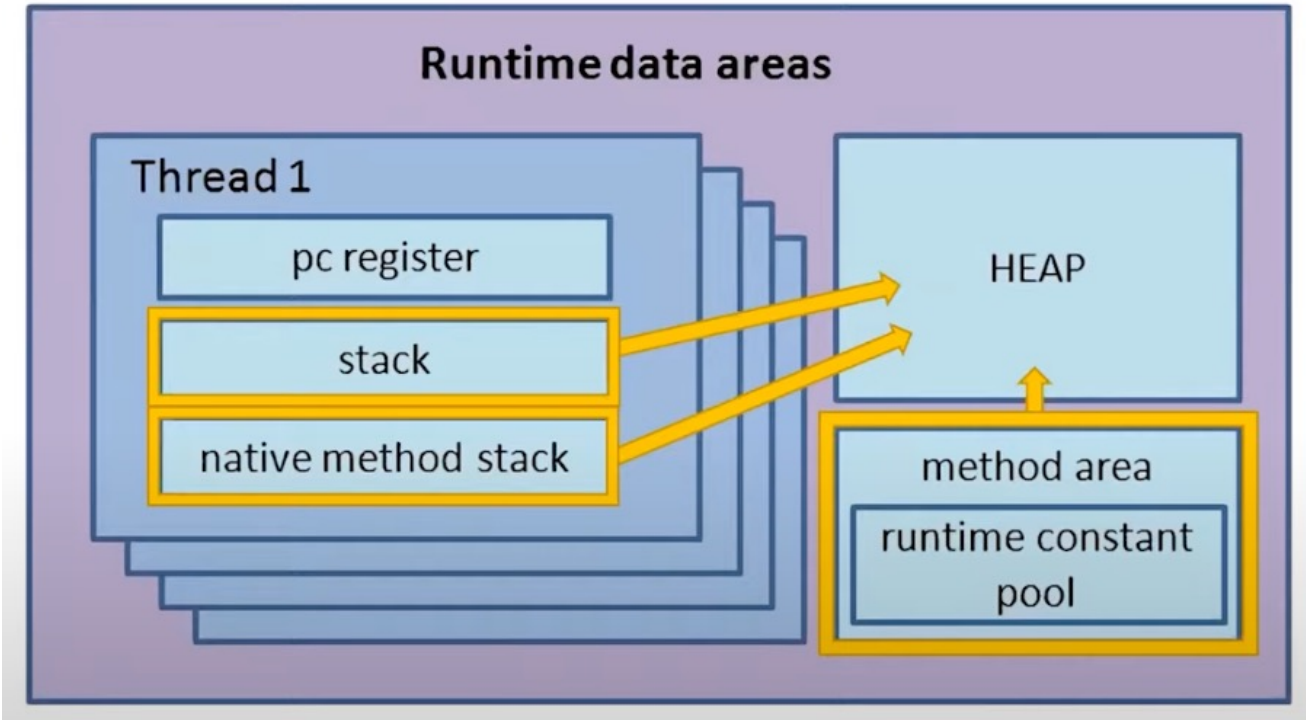
객체와 배열이 생성되는 영역
GC의 대상

런타임 상수풀
필드/메소드 데이터
메소드 코드
생성자 코드

C, C++, 어셈블리 등으로 작성된
네이티브 코드 실행 시 사용

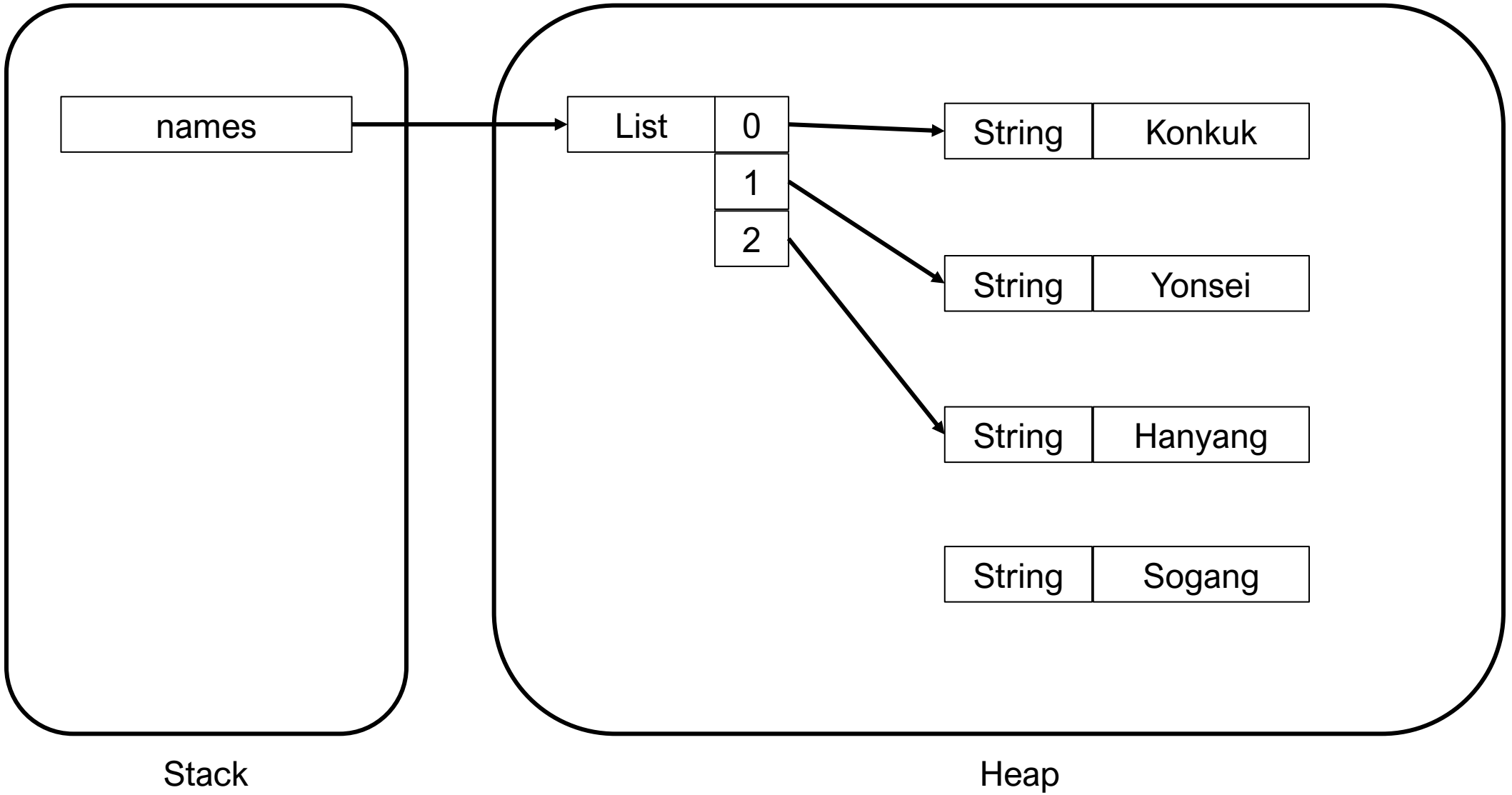
JVM 메모리 영역





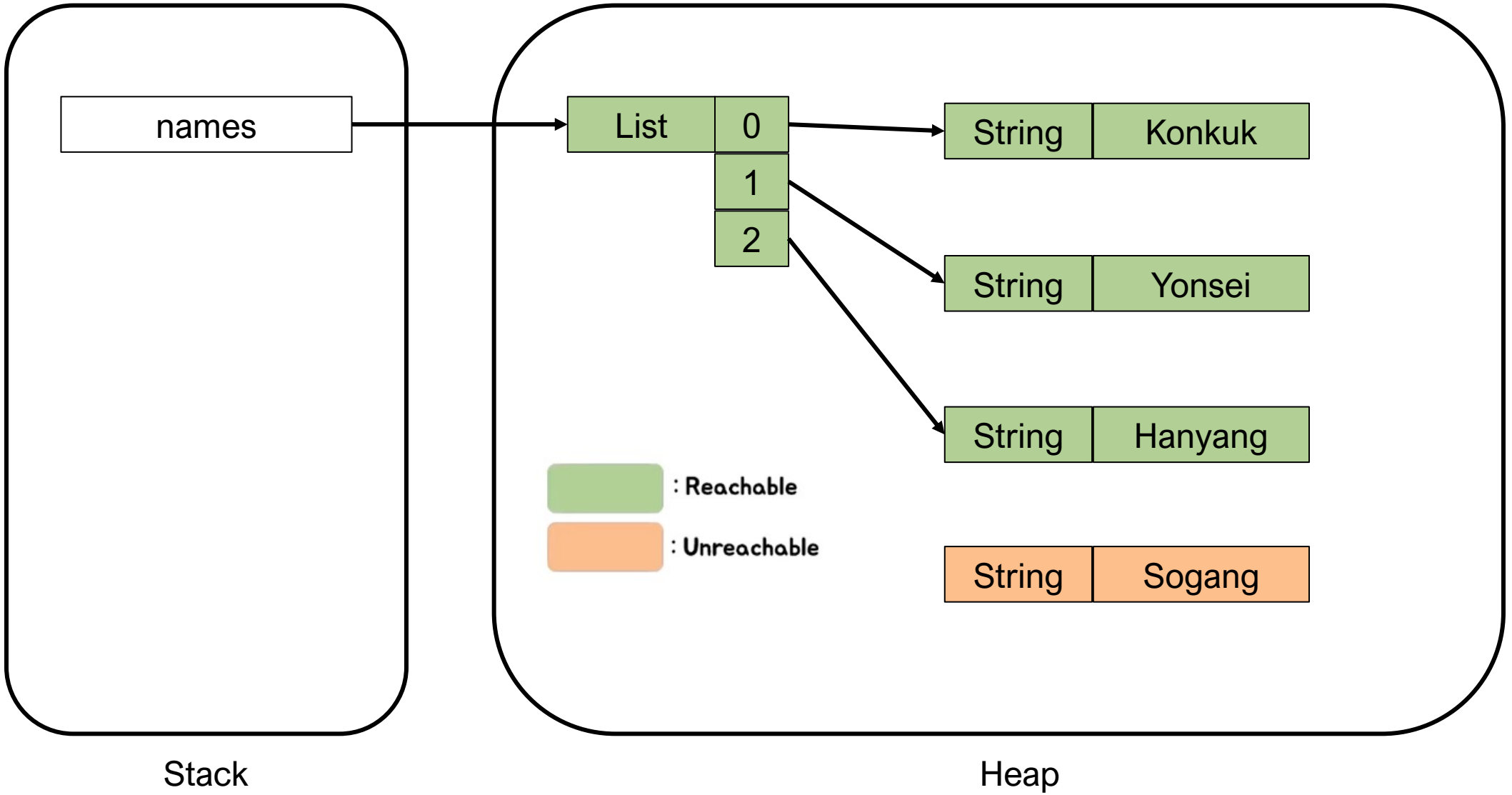
https://www.youtube.com/watch?v=FMUpVA0Vvjw&ab_channel=%EC%9A%B0%EC%95%84%ED%95%9C%ED%85%8C%ED%81%AC

JVM 메모리 영역

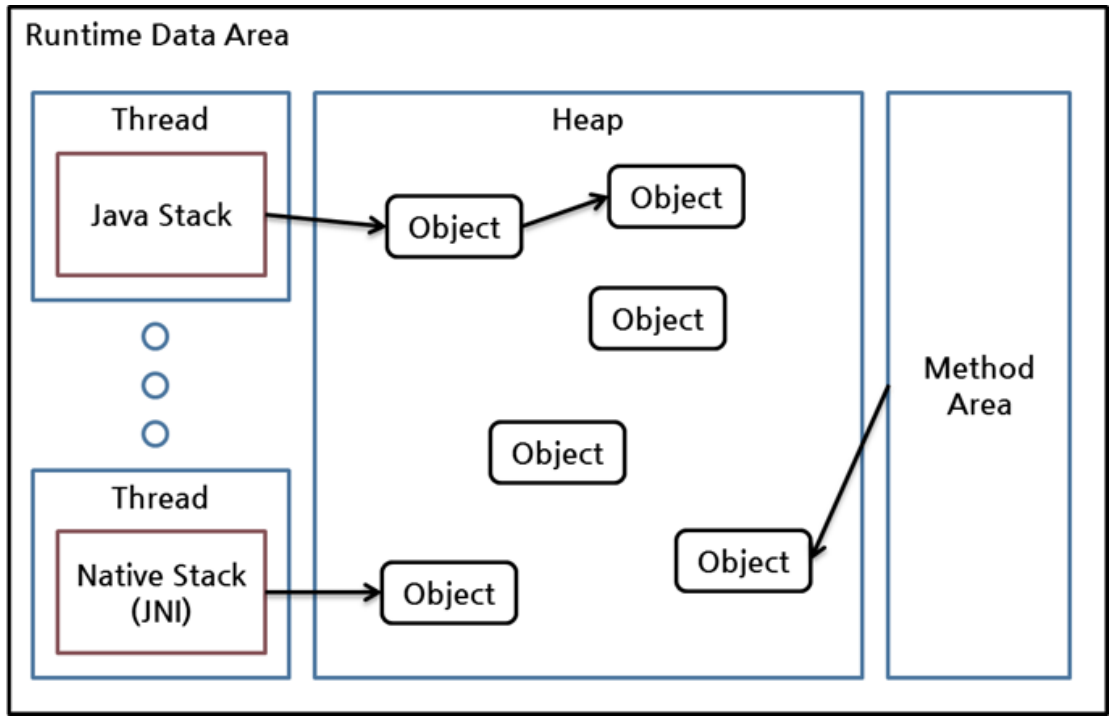


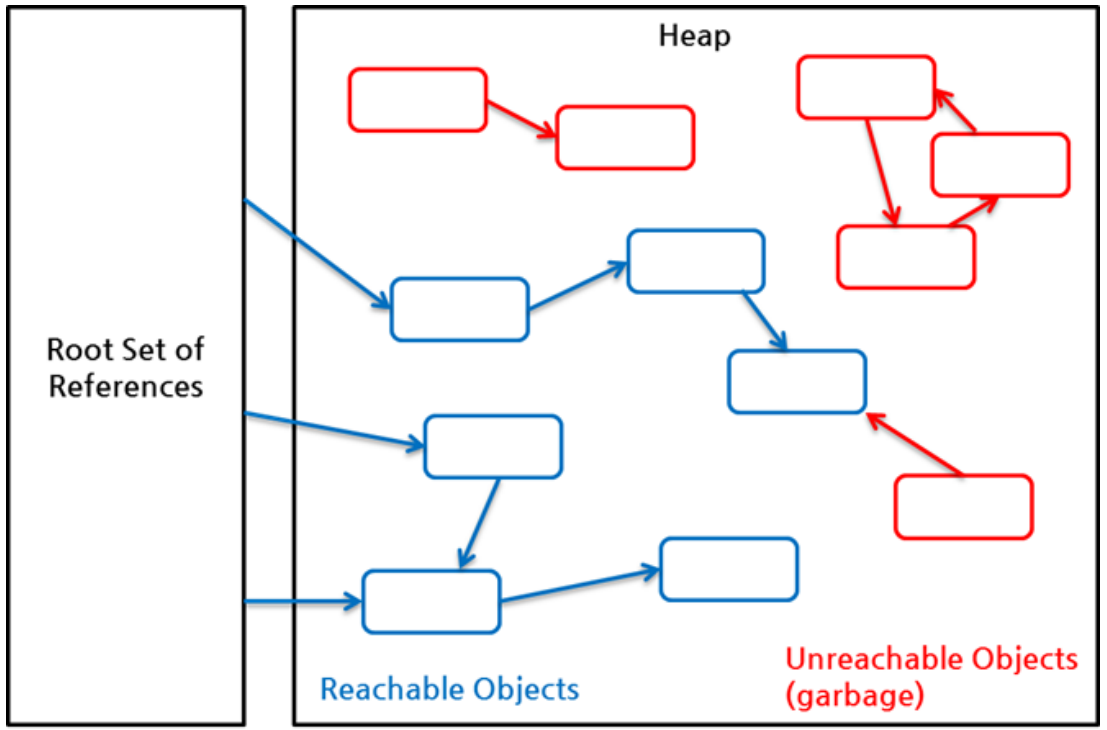
JVM 메모리 영역

Mark and Sweep



JVM 메모리 영역

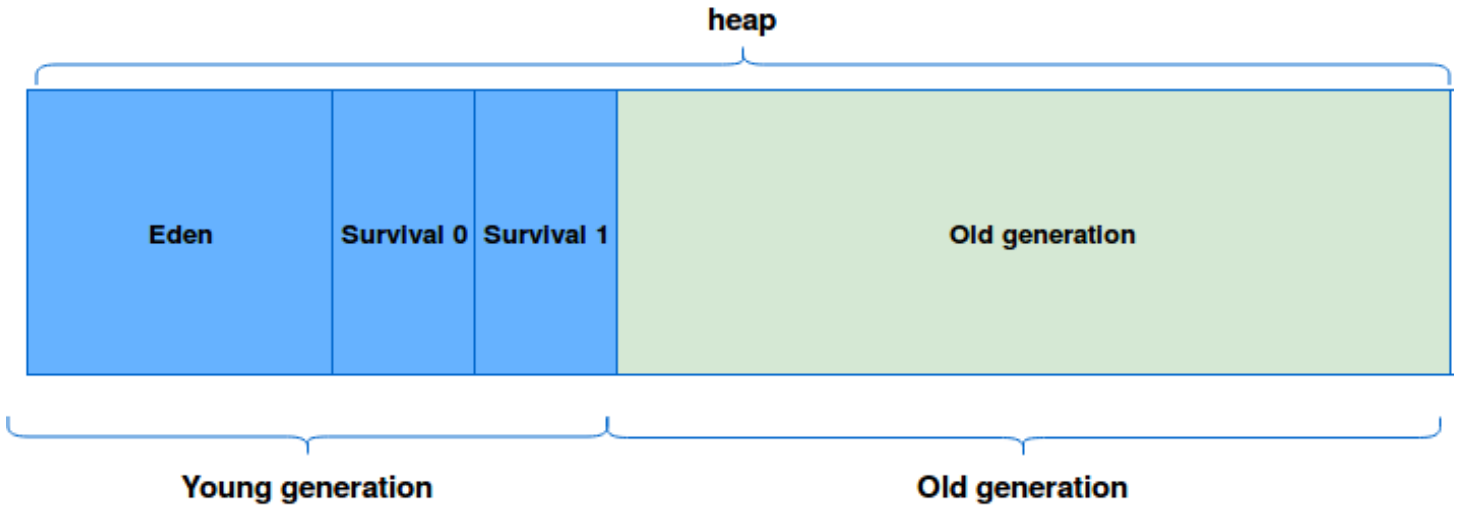




Heap 영역

weak generational hypothesis

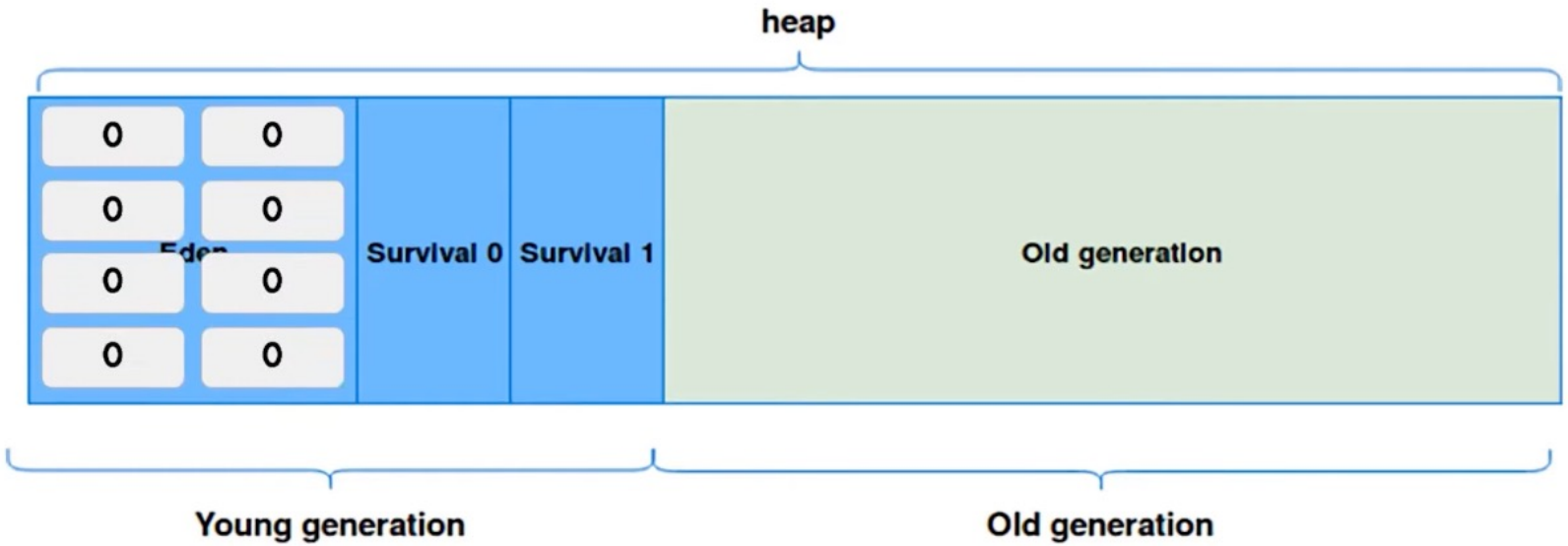
- 대부분의 객체는 금방 접근 불가능 상태(unreachable)가 된다.
- 오래된 객체에서 젊은 객체로의 참조는 아주 적게 존재한다.

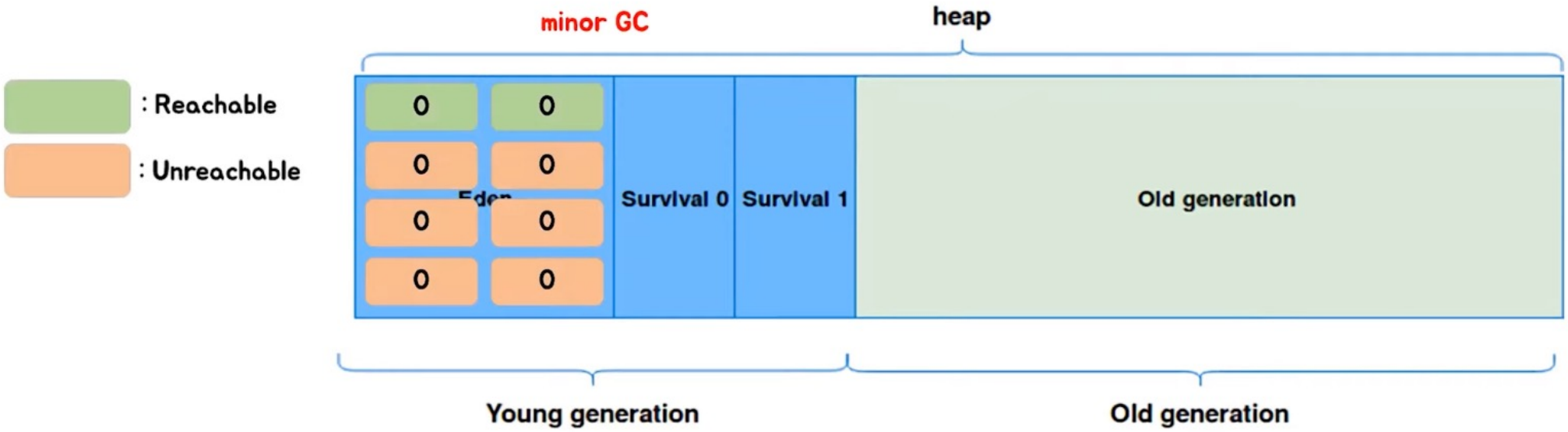


Young generation : 새롭게 생성된 객체가 할당되는 영역 → Minor GC

Old generation : Young 영역에서 오래 살아남은 객체가 저장되는 영역 → Major GC

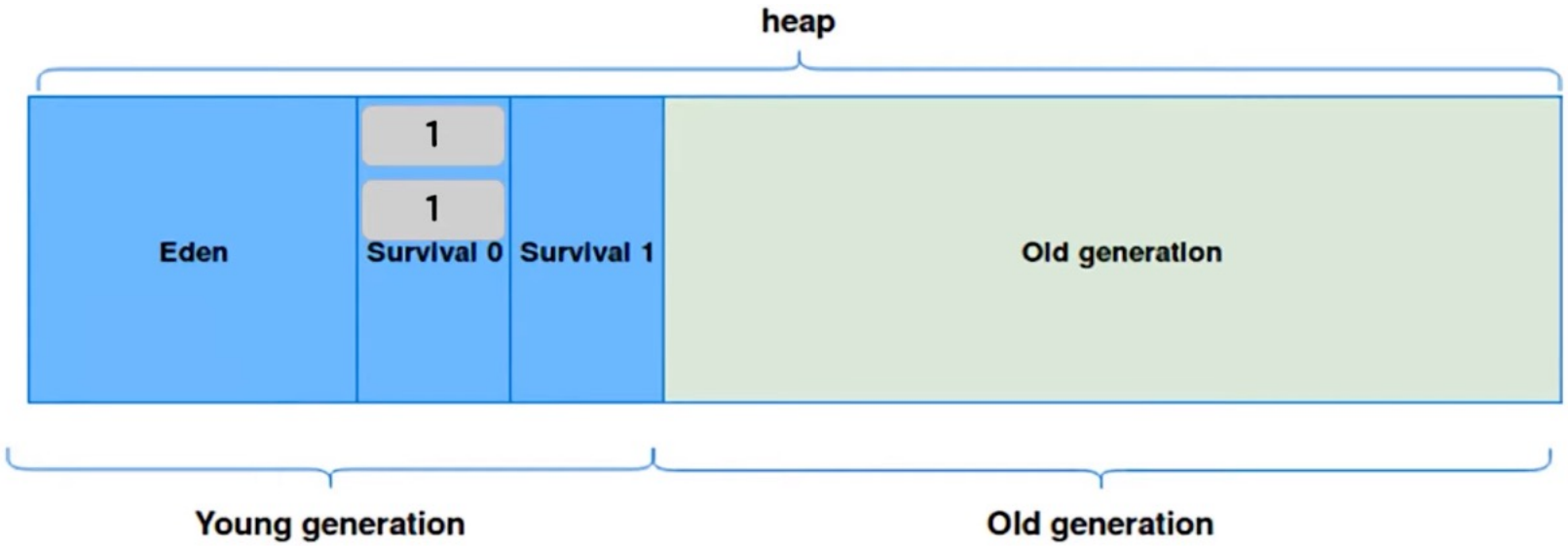
Heap 영역

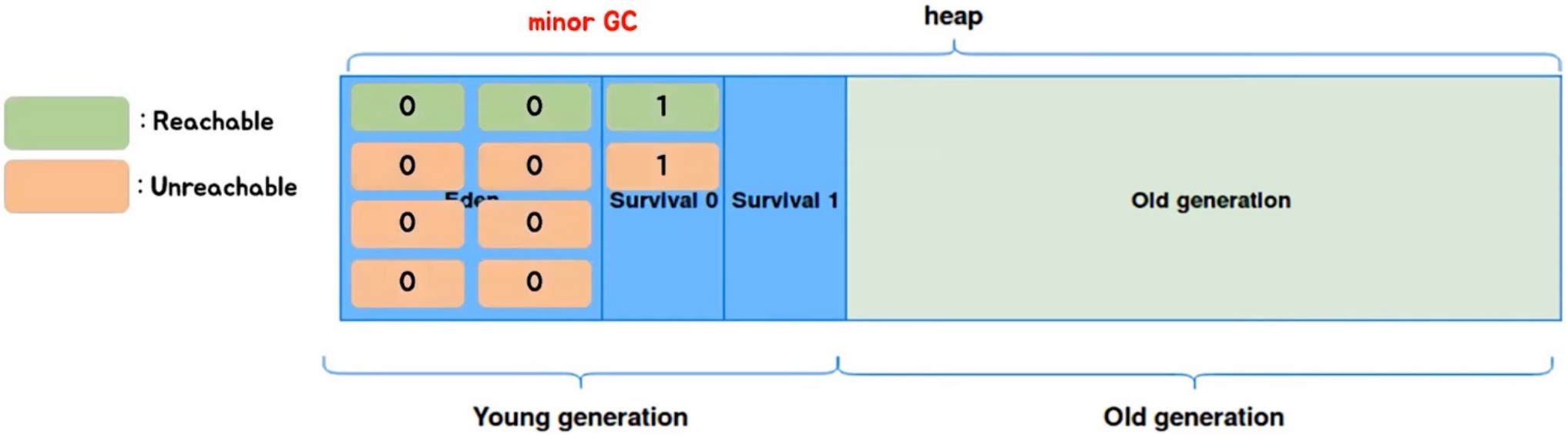




https://www.youtube.com/watch?v=FMUpVA0Vvjw&ab_channel=%EC%9A%B0%EC%95%84%ED%95%9C%ED%85%8C%ED%81%AC

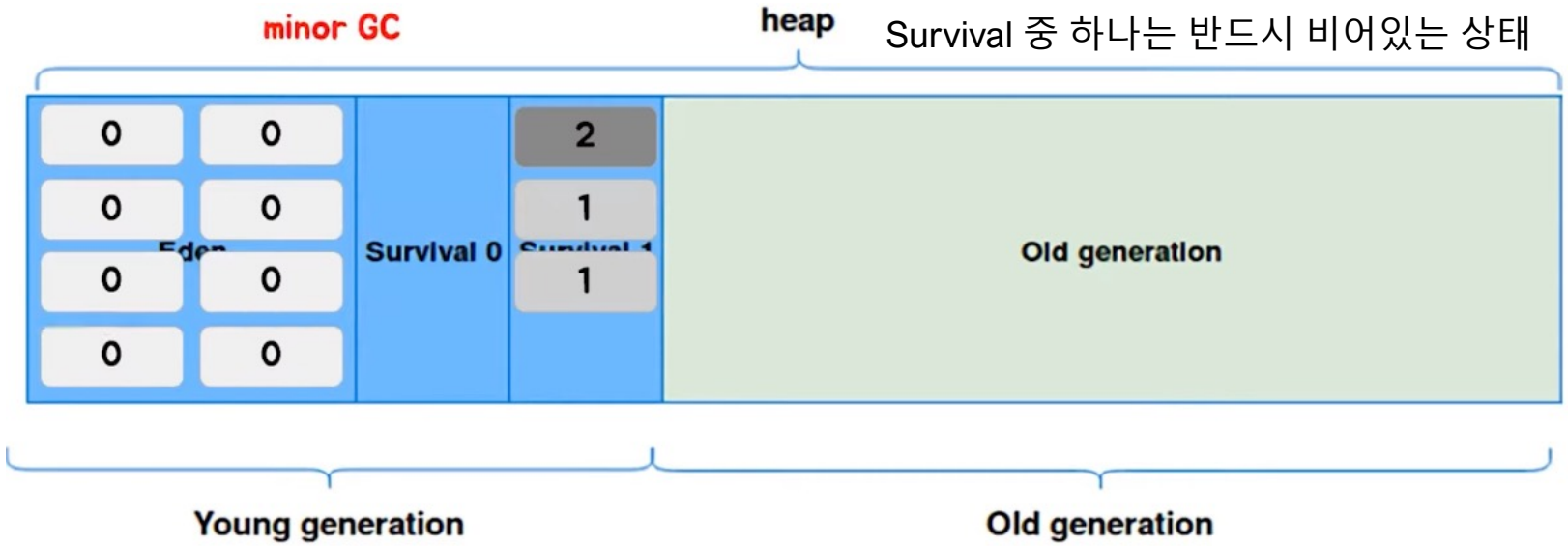
Heap 영역

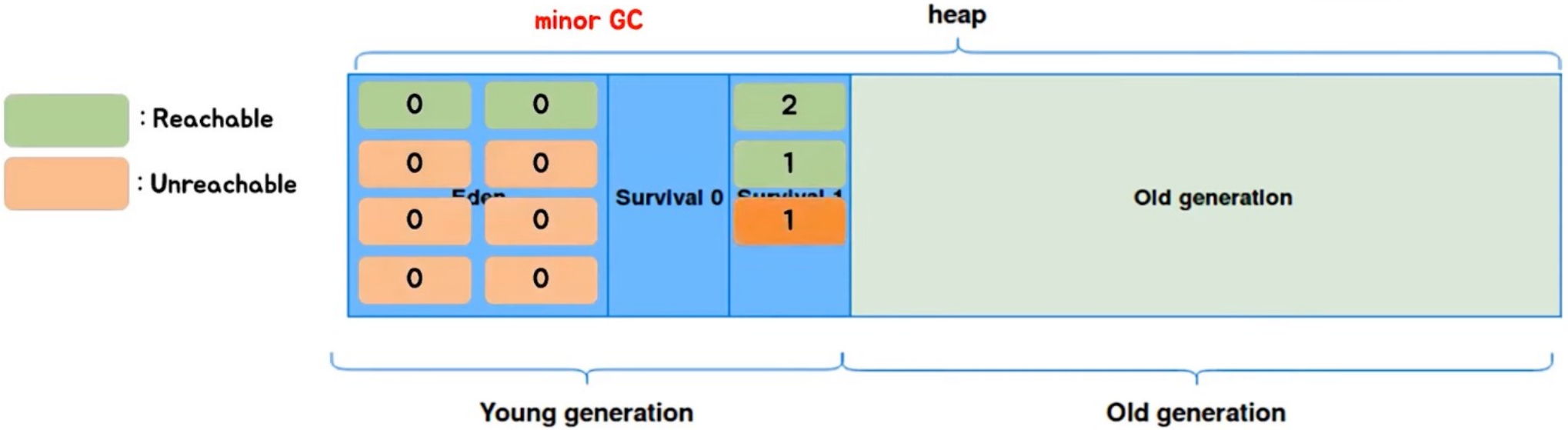




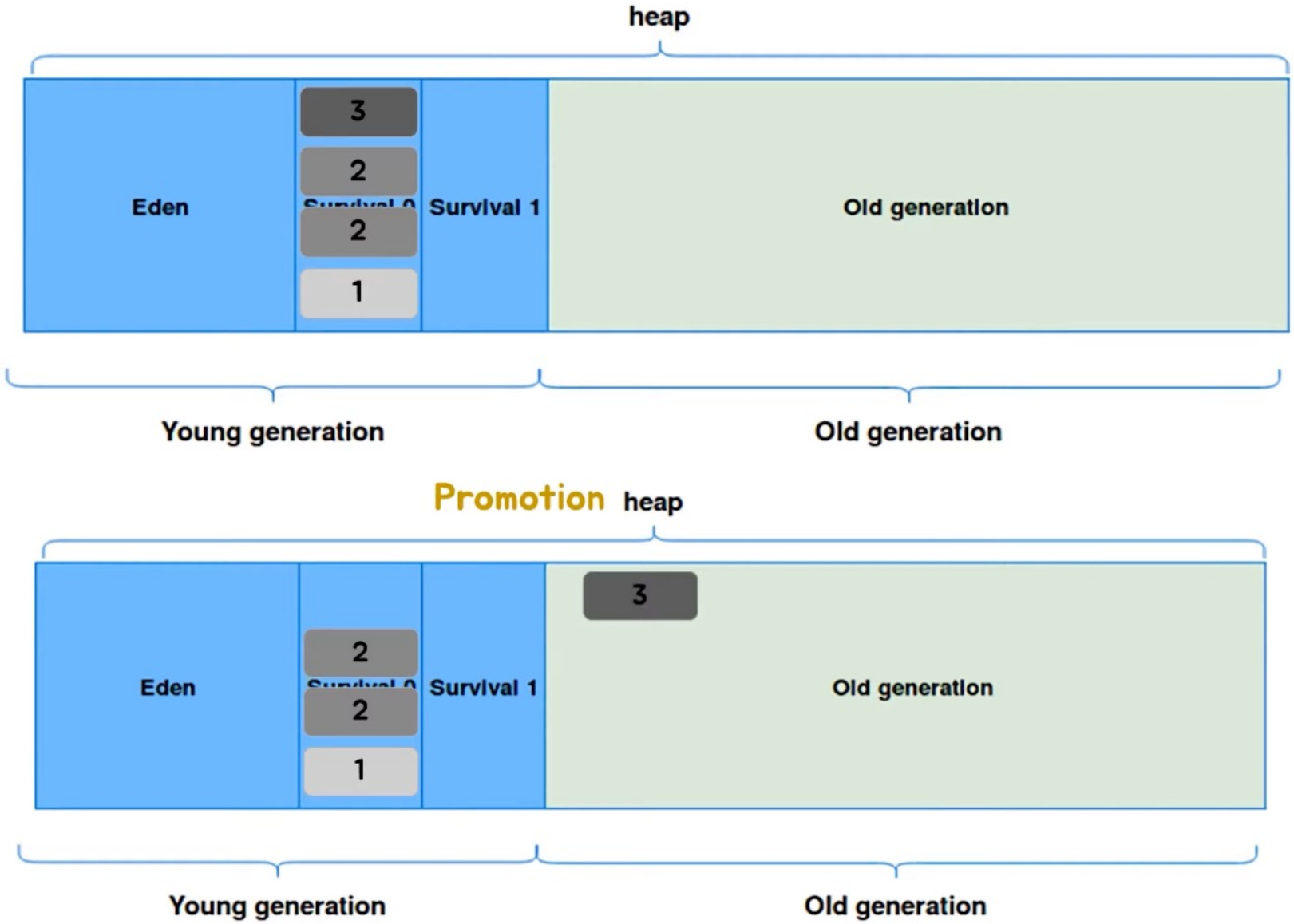
https://www.youtube.com/watch?v=FMUpVA0Vvjw&ab_channel=%EC%9A%B0%EC%95%84%ED%95%9C%ED%85%8C%ED%81%AC

Heap 영역



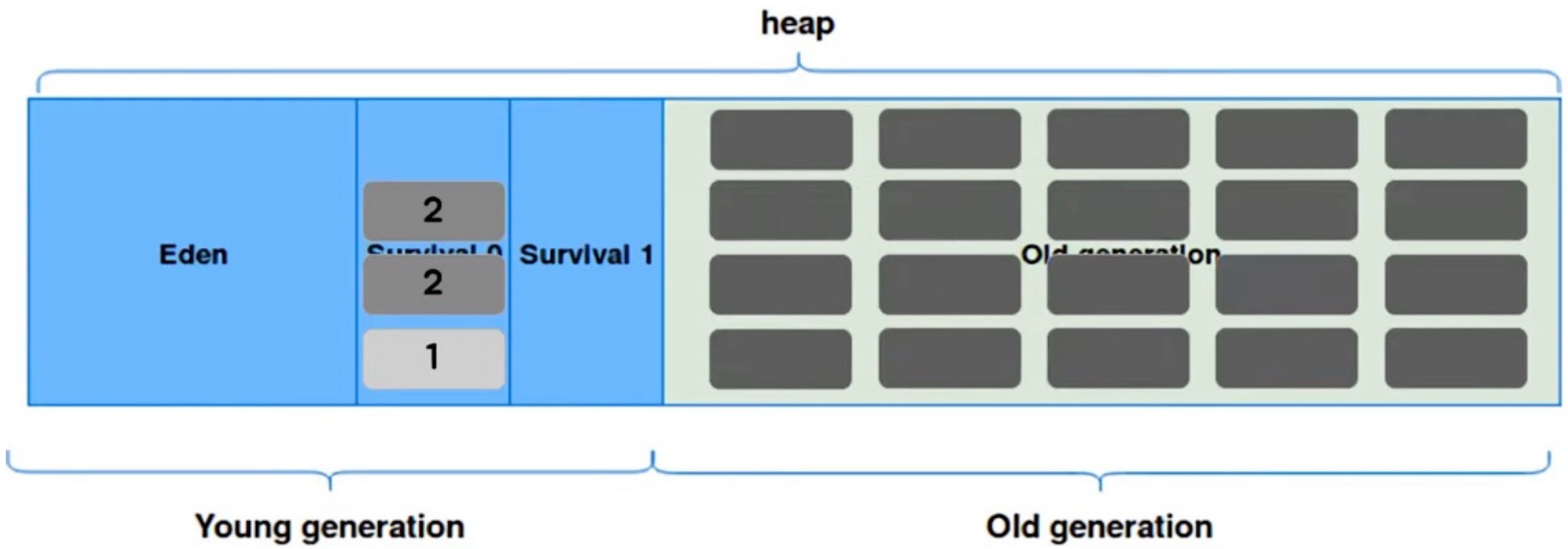


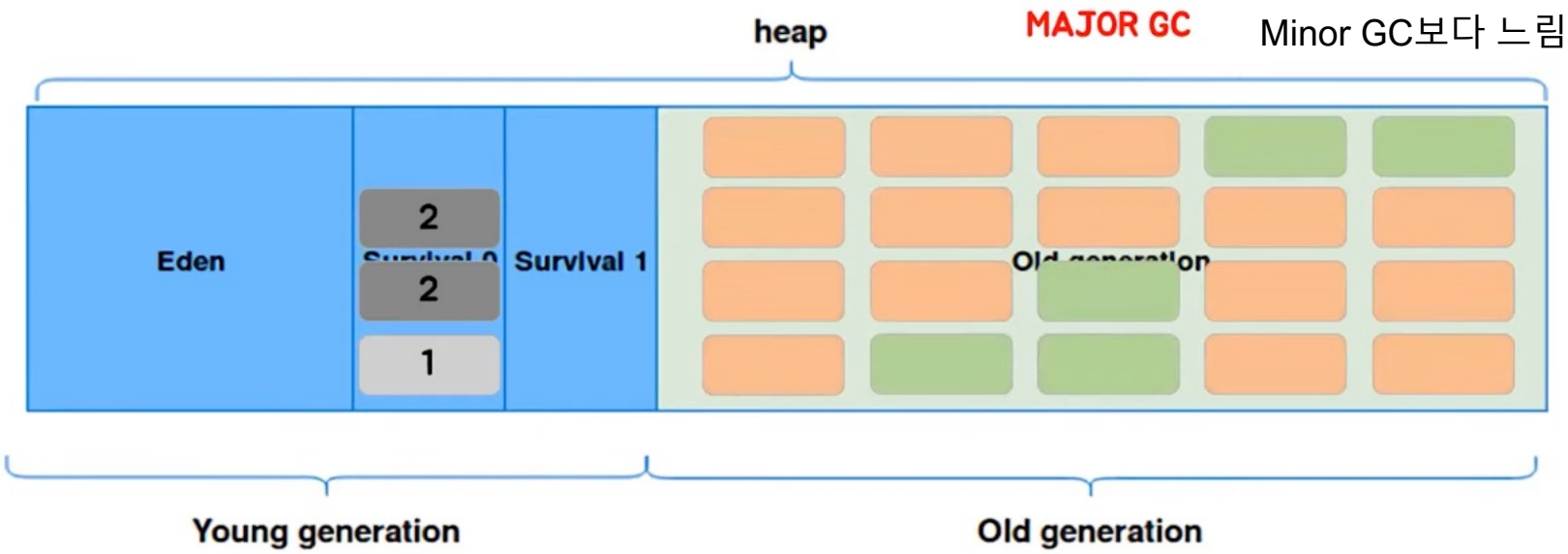
Heap 영역



Java 8 Parallel GC 기준
age가 15이상이면 이동

Heap 영역





Minor GC보다 느림

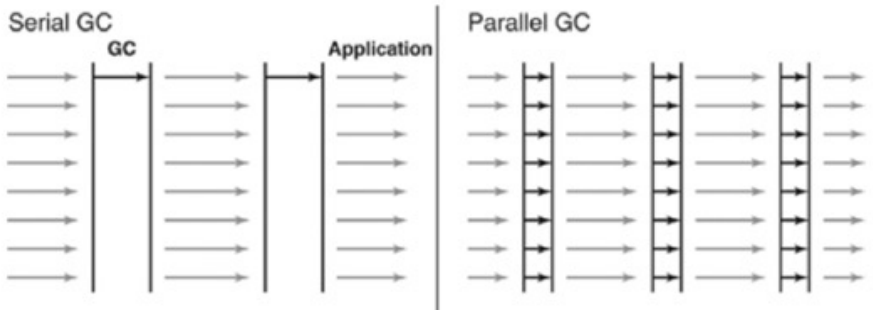
https://www.youtube.com/watch?v=FMUpVA0Vvjw&ab_channel=%EC%9A%B0%EC%95%84%ED%95%9C%ED%85%8C%ED%81%AC

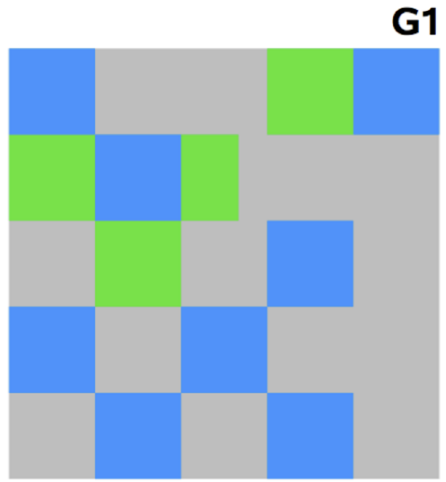
Garbage Collector 종류

Stop the World

GC 실행을 위해 JVM이 애플리케이션의 실행을 멈추는 작업

1. Serial GC : 싱글 코어 컴퓨터를 위한 방식(느림)
2. Parallel GC : Java8까지 기본값
3. Concurrent Mark Sweep GC (CMS GC)
4. G1(Garbage First) GC : Java9부터 기본값





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

Java Reference

1. Strong Reference
일반적으로 new를 통해서 생성한 객체
GC 대상에서 제외된다.

2. Soft Reference
메모리에 충분한 여유가 없다면 GC 대상이 될 수 있다.

new SoftReference<User>(new User())

3. Weak Reference
GC가 발생하면 무조건 수거된다.

new WeakReference<User>(new User())

4. Phantom Reference

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

Weak Reference

GC에 의해 강제로 수집될 수 있는 참조를 나타내는 객체
LRU 캐시 등에 사용

```java
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));
    }
}
```

Weak Reference

```java
public class Example {
    public static void main(String[] args) {
        Map<Object, String> weakMap = new WeakHashMap<>();
        Object key = new Object();
        weakMap.put(key, "value");
        // key 참조 제거
        key = null;
        // garbage collector 강제 실행
        System.gc();
        // weakMap이 비어있는지 확인
        System.out.println("weakMap.size() = " + weakMap.size());
    }
}
```

0

```java
public class Example {
    public static void main(String[] args) {
        Map<Object, String> hashMap = new HashMap<>();
        Object key = new Object();
        hashMap.put(key, "value");
        // key 참조 제거
        key = null;
        // garbage collector 강제 실행
        System.gc();
        // hashMap이 비어있는지 확인
        System.out.println("hashMap.size() = " + hashMap.size());
    }
}
```

1

Item 8

finalizer와 cleaner 사용을 피하라

finalizer와 cleaner를 사용하지 말자

Finalizer은 예측 불가능하고 낮은 성능의 원인이 된다.

Java 9에서 finalizer은 deprecated 되었고 cleaner가 생겼지만 여전히 문제다.

C++에서의 소멸자(desturctor)와 다르다.

단점 1 : 즉시 수행된다는 보장이 없다.

GC 대상이 된 후, 실제로 GC가 실행될 때 finalizer가 실행된다.

언제 실행될 지 모르기 때문에 finalizer에서 타이밍이 중요한 작업은 하면 안된다.

대표적인 예시 : 파일 리소스 반납
- 시스템이 동시에 열 수 있는 파일 개수에 한계가 존재

cleaner는 쓰레드를 제어할 수 있다는 면에서 조금 낫지만 여전히 GC 타이밍은 모름

단점 2 : 아예 실행이 되지 않을 수 있다.

자바 언어 명세는 finalizer나 cleaner의 수행 여부조차 보장하지 않는다.

예를 들어, DB Lock 해제를 finalizer나 cleaner에서 수행한다면 시스템이 멈출 수 있다.

System.gc나 system.runFialization도 실행을 보장해주진 않는다.

System.runFializersOnExit와 Runtime.runFializersOnExit은 deprecated

단점 3 : 동작 중 발생한 예외는 무시된다.

finalizer 실행 중 발생한 예외는 경고조차 출력하지 않고, 그 순간 종료된다.

처리할 작업이 남았더라도 그 순간 종료되기 때문에 수행을 보장하지 않는다.

```java
public class SampleResource {

    @Override
    protected void finalize() {
        System.out.println("finalize");
        throw new RuntimeException("finalize error");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        new SampleResource();
        System.gc();
    }
}
```

```
/Users/hongseungtaeg/.sdkman/c
finalize


종료 코드 0(으)로 완료된 프로세스
```

단점 4 : 심각한 성능 문제

	AutoCloseable + try-with-resources	finalizer	cleaner
exec time	12ns	550ns	500ns
speed up	1	45.8	41.7

(잠시 후 알아볼) 안전망 방식에서는 66ns가 소요 : 5.5배 느려짐

단점 5 : 보안이슈

```java
public class Secure {
    public Secure() {
        throw new RuntimeException();
    }

    public void securityMethod() {
        System.out.println("This is a secure method");
    }
}
```

```java
public class Attack extends Secure {
    static Secure secure;

    public Attack() {
        super();
    }

    public void finalize() {
        secure = this;
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        try {
            new Attack();
        } catch (Exception e) {
        }
        System.gc();
        System.runFinalization();

        Attack.secure.securityMethod();
    }
}
```

문제점

1. **Attack은 GC 대상이 아니다.**
2. **실행되서는 안되는 메서드를 실행할 수 있다.**

단점 5 : 보안이슈

해결방법 : finalize를 final로 선언하자

```
public class Secure {
    public Secure() {
        throw new RuntimeException();
    }

    @Override
    protected final void finalize() throws Throwable {

    }

    public void securityMethod() {
        System.out.println("This is a secure method");
    }
}
```

그럼 언제 사용하는가?

1. 안전망으로 자원 반납하기
2. 네이티브 피어 정리할 때 사용하기
3. Cleaner를 안전망으로 사용하기

자원 반납하기 : AutoCloseable

V1 : try-catch-finally

```
public class SampleResource {
    public void open() {
        System.out.println("open");
    }

    public void close() {
        System.out.println("close");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        SampleResource resource = null;
        try {
            resource = new SampleResource();
            resource.open();
        } catch (Exception e) {
        } finally {
            if (resource != null) {
                resource.close();
            }
        }
    }
}
```

자원 반납하기 : AutoCloseable

V2 : AutoCloseable + try-with-resources

```
public class SampleResource implements AutoCloseable {
    public void open() {
        System.out.println("open");
    }

    public void close() {
        System.out.println("close");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        try (SampleResource resource = new SampleResource()) {
            resource.open();
        } catch (Exception e) {
        }
    }
}
```

자원 반납하기 : AutoCloseable

V3 : 안전망 추가

```
public class SampleResource implements AutoCloseable {
    private boolean closed;

    public void open() {
        System.out.println("open");
    }

    @Override
    public void close() {
        if (closed) {
            throw new IllegalStateException("Already closed");
        }
        closed = true;
        System.out.println("close");
    }

    @Override
    protected void finalize() {
        if (!closed) {
            close();
        }
    }
}
```

native peer와 연결된 객체

네이티브 피어 : 일반 자바 객체가 네이티브 메서드를 통해 기능을 위임한 객체

자바 객체가 아니니 GC에 의해 회수하지 못한다.

하지만 역시 여러 문제가 있기 때문에 (중요한 자원이면) close를 사용하자

cleaner를 안전망으로 사용하기

```
import java.lang.ref.Cleaner;

public class CleanerSample implements AutoCloseable {

    private static final Cleaner CLEANER = Cleaner.create();
    private final Cleaner.Cleanable cleanable;
    private final CleanerRunner cleanerRunner;

    // Cleaner를 수행한 별도의 스레드
    private static class CleanerRunner implements Runnable {
        // TODO 여기에 정리할 리소스 전달
        // 여기에 CleanerSample이 있으면 안된다.(순환참조)
        @Override
        public void run() {
            // 여기서 정리
            System.out.println("close");
        }
    }
```

```
    public CleanerSample() {
        cleanerRunner = new CleanerRunner();
        cleanable = CLEANER.register(this, cleanerRunner);
    }

    @Override
    public void close() {
        cleanable.clean();
    }
}
```

Item 9

try-with-resources를 사용하라

Example

```
public class MyResource implements AutoCloseable {

    public void doSomething() {
        System.out.println("do Something");
        throw new FirstException();
    }

    @Override
    public void close() {
        throw new SecondException();
    }
}
```

Example : V1

```java
public void v1() {
    MyResource resource = null;
    try {
        resource = new MyResource();
        resource.doSomething();
    } finally {
        if (resource != null) {
            resource.close();
        }
    }
}
```

FirstException이 삼켜진다

```
do Something
Exception in thread "main" SecondException
    at MyResource.close(MyResource.java:9)
    at AppRunner.v1(AppRunner.java:14)
    at AppRunner.main(AppRunner.java:4)
```

Example : V2

요구사항 추가 : FirstException이 발생하면 RuntimeException으로 던져라

```
public void v2() {
    MyResource resource = null;
    try {
        resource = new MyResource();
        resource.doSomething();
    } catch (Exception e) {
        e.printStackTrace();
        throw new RuntimeException("에러 발생");
    } finally {
        if (resource != null) {
            resource.close();
        }
    }
}
```

RuntimeException이 먹힌다

```
do Something
FirstException
    at MyResource.doSomething(MyResource.java:5)
    at AppRunner.v2(AppRunner.java:23)
    at AppRunner.main(AppRunner.java:4)
Exception in thread "main" SecondException
    at MyResource.close(MyResource.java:9)
    at AppRunner.v2(AppRunner.java:29)
    at AppRunner.main(AppRunner.java:4)
```

Example : V3

```java
public void v3() {
    MyResource resource = null;
    try {
        resource = new MyResource();
        resource.doSomething();
    } catch (Exception e) {
        e.printStackTrace();
        throw new RuntimeException("에러 발생");
    } finally {
        if (resource != null) {
            try {
                resource.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

성공!

```
do Something
FirstException
    at MyResource.doSomething(MyResource.java:5)
    at AppRunner.v3(AppRunner.java:38)
    at AppRunner.main(AppRunner.java:4)
SecondException
    at MyResource.close(MyResource.java:9)
    at AppRunner.v3(AppRunner.java:45)
    at AppRunner.main(AppRunner.java:4)
Exception in thread "main" java.lang.RuntimeException Create breakpoint : 에러 발생
    at AppRunner.v3(AppRunner.java:41)
    at AppRunner.main(AppRunner.java:4)
```

Example : V4

```java
public void v4() {
    try (MyResource resource = new MyResource()) {
        resource.doSomething();
    } catch (Exception e) {
        e.printStackTrace();
        throw new RuntimeException("에러 발생");
    }
}
```

```
do Something
FirstException
    at MyResource.doSomething(MyResource.java:5)
    at AppRunner.v4(AppRunner.java:55)
    at AppRunner.main(AppRunner.java:4)
    Suppressed: SecondException
        at MyResource.close(MyResource.java:9)
        at AppRunner.v4(AppRunner.java:54)
        ... 1 more
Exception in thread "main" java.lang.RuntimeException Create breakpoint : 에러 발생
    at AppRunner.v4(AppRunner.java:58)
    at AppRunner.main(AppRunner.java:4)
```