

finalizer와 cleaner 사용을 피하라

이펙티브 자바 item8

유현우

단점 1. GC알고리즘에 따라서 실행될 수도 안될 수도 있다.

- Finalizer
- Cleaner

단점 2. 성능 문제

- AutoCloseable

단점 3. Finalizer를 사용한 클래스는 finalizer 공격에 노출되어 심각한 보안 문제를 일으킬 수도 있다.

- 해결 방법

결론

단점 1. GC 알고리즘에 따라서 실행될 수도 안될 수도 있다.


- Finalizer



```
public class FinalizerTest {  
  
    @Override  
    protected void finalize() throws Throwable {  
        System.out.println("GC야 호출 해줄거야?");  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        new FinalizerTest();  
        Thread.sleep(1000);  
    }  
}
```

종료 작업을 전혀 수행하지 못한 채 프로그램이 중단될 수 있다.

- Finalizer



```
public static void main(String[] args) throws InterruptedException {  
    new FinalizerTest();  
    System.gc();  
    Thread.sleep(1000);  
}
```

System.gc(), System.runFinalization() 메서드를 호출하더라도 JVM 구현 및 시스템 조건에 따라서 다르다.

java.lang.Object

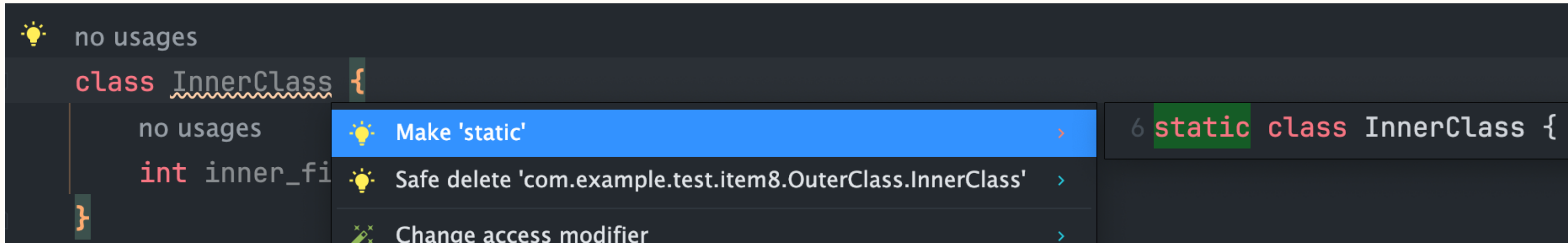
Deprecated The finalization mechanism is inherently problematic. Finalization can lead to performance issues, deadlocks, and hangs. Errors in finalizers can lead to resource leaks; there is no way to cancel finalization if it is no longer necessary; and no ordering is specified among calls to `finalize` methods of different objects. Furthermore, there are no guarantees regarding the timing of finalization. The `finalize` method might be called on a finalizable object only after an indefinite delay, if at all. Classes whose instances hold non-heap resources should provide a method to enable explicit release of those resources, and they should also implement `AutoCloseable` if appropriate. The `java.lang.ref.Cleaner` and `java.lang.ref.PhantomReference` provide more flexible and efficient ways to release resources when an object becomes unreachable.



```
@Deprecated(since="9")  
protected void finalize() throws Throwable { }
```

GC가 `finalize`, `cleaner`를 실행시키는 우선순위가 다른 객체들을 회수하는 것보다 낮아서 계속 밀릴 수 있다.

Inner 클래스는 외부 참조를 한다



```
public class OuterClass {  
    int field = 10;  
  
    class InnerClass {  
        int innerField = 20;  
    }  
}
```

Inner 클래스는 외부 참조를 한다

→ item8 ls

FinalizerTest.java OuterClass.java Room.java

→ item8 javac OuterClass.java

→ item8 ls

FinalizerTest.java OuterClass\$InnerClass.class OuterClass.class OuterClass.java Room.java

→ item8

```
package com.example.test.item8;
```

```
class OuterClass$InnerClass {
```

```
    int inner_field = 20;
```

```
    OuterClass$InnerClass(final OuterClass var1) {
```

```
    }
```

```
}
```

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

```
package com.example.test.item8;
```

```
public class OuterClass {
```

```
    int field = 10;
```

```
    public OuterClass() {
```

```
    }
```

```
    class InnerClass {
```

```
        int inner_field = 20;
```

```
        InnerClass(final OuterClass var1) {
```

```
        }
```

```
    }
```

```
}
```

Inner 클래스의 메모리 누수 현상

```
public class OuterClass {  
    private int[] data;  
  
    class InnerClass {  
    }  
  
    public OuterClass(int size) {  
        this.data = new int[size];  
    }  
  
    InnerClass getInnerInstance() {  
        return new InnerClass();  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        ArrayList<OuterClass.InnerClass> al = new ArrayList<>();  
  
        for (int counter = 0; counter < 50; counter++) {  
            al.add(new OuterClass(100000000).getInnerInstance());  
            System.out.println(counter);  
        }  
    }  
}
```

```
6  
7  
8  
9  
Exception in thread "main" java.lang.OutOfMemoryError: Create breakpoint : Java heap space  
    at com.example.test.item8.OuterClass.<init>(OuterClass.java:14)  
    at com.example.test.item8.Main.main(OuterClass.java:27)
```

원래라면 메소드 호출용도로만 쓰여진 일회용 객체는 바로 GC 수거 대상이 되어 제거되어야 한다.

메모리 누수 해결 방법

3 usages

```
static class InnerClass {  
}
```

□ Main ×

41

42

43

44

45

46

47

48

49

Process finished with exit code 0

static(X)

```
package com.example.test.item8;  
  
class OuterClass$InnerClass {  
    OuterClass$InnerClass(final OuterClass var1) {  
    }  
}
```

static(O)

```
package com.example.test.item8;  
  
class OuterClass$InnerClass {  
    OuterClass$InnerClass() {  
    }  
}
```

단점 1. GC 알고리즘에 따라서 실행될 수도 안될 수도 있다.

- Cleaner (안전망으로 사용)

```
public class Room implements AutoCloseable {
    ...
    private static class State implements Runnable {
        int numJunkPiles;

        State(int numJunkPiles) {
            this.numJunkPiles = numJunkPiles;
        }

        // colse가 호출되거나, GC가 Room을 수거해갈 때 run() 호출
        @Override
        public void run() {
            System.out.println("방 청소");
            numJunkPiles = 0;
        }
    }
}
```

```
public class Room implements AutoCloseable {
    private static final Cleaner cleaner = Cleaner.create();

    private final State state;
    private final Cleaner.Cleanable cleanable;

    public Room(int numJunkPiles) {
        state = new State(numJunkPiles);
        cleanable = cleaner.register(this, state);
    }

    @Override
    public void close() {
        cleanable.clean();
    }
    // Room을 참조하지 말것!!! 순환 참조
    private static class State implements Runnable {
        ...
    }
}
```

Cleaner와 PhantomReference

Cleaner

Registers an object and a cleaning action to run when the object becomes phantom reachable. Refer to the [API Note](#) above for cautions about the behavior of cleaning actions.

Params: obj – the object to monitor
action – a Runnable to invoke when the object becomes phantom reachable

Returns: a Cleanable instance

```
public Cleanable register(Object obj, Runnable action) {
    Objects.requireNonNull(obj, message: "obj");
    Objects.requireNonNull(action, message: "action");
    return new CleanerImpl.PhantomCleanableRef(obj, cleaner: this, action);
}
```

PhantomCleanableRef

```
@Override
protected void performCleanup() {
    action.run();
}
```

PhantomCleanable<T> extends PhantomReference<T> ...

Unregister this PhantomCleanable and invoke `performCleanup()`, ensuring at-most-once semantics.

```
@Override
public final void clean() {
    if (remove()) {
        super.clear();
        performCleanup();
    }
}
```

Reference

Clears this reference object. Invoking this method will not cause this object to be enqueued. This method is invoked only by Java code; when the garbage collector clears references it does so directly, without invoking this method.

```
public void clear() {
    clear0();
}
```

단점 1. GC 알고리즘에 따라서 실행될 수도 안될 수도 있다.



```
public static void main(final String[] args) {  
    new Room(8);  
    System.gc();  
    System.out.println("방 쓰레기 생성~~");  
}
```

try-with-resource



```
public static void main(String[] args) {  
    try (Room room = new Room(8)) {  
    }  
}
```



```
public class Room implements AutoCloseable {  
    private static final Cleaner cleaner = Cleaner.create();  
  
    private final State state;  
    private final Cleaner.Cleanable cleanable;  
  
    public Room(int numJunkPiles) {  
        state = new State(numJunkPiles);  
        cleanable = cleaner.register(this, state);  
    }  
  
    @Override  
    public void close() {  
        cleanable.clean();  
    }  
    // Room을 참조하지 말것!!! 순환 참조  
    private static class State implements Runnable {  
        ...  
    }  
}
```

단점 2. 성능 문제

```
class MyFinalizerWithClose implements AutoCloseable {
    @Override
    public void close() {
        System.out.println("GC야 close 호출?");
    }
}

class MyFinalizer {
    @Override
    protected void finalize() {
        System.out.println("GC야 호출 해줄거야?");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Main main = new Main();
        // 1번
        long start1 = System.nanoTime();
        main.finalizerTest();
        System.gc();
        System.out.println(System.nanoTime() - start1); //4083666

        // 2번
        long start2 = System.nanoTime();
        main.closeableTest();
        System.out.println(System.nanoTime() - start2); //333625
    }

    public void finalizerTest() {
        MyFinalizer test = new MyFinalizer();
    }

    public void closeableTest() {
        try (MyFinalizerWithClose finalizer = new MyFinalizerWithClose()) {
        }
    }
}
```

단점 2. 성능 문제

저자가 작성한 테스트 결과

- AutoCloseable 객체를 생성하고 try-with-resources로 자원을 닫아서 가비지컬렉터가 수거하기까지 12ns가 걸렸다면 finalizer를 사용한 객체를 생성하고 파괴하니 550ns가 걸렸다. (50배)
- finalizer가 가비지 컬렉터의 효율을 떨어지게 한다. 안전망 형태로만 사용하면 66ns가 걸린다. 안전망의 대가로 50배에서 5배로 성능차이를 낼 수 있다.

단점 3. Finalizer를 사용한 클래스는 finalizer 공격에 노출되어 심각한 보안 문제를 일으킬 수도 있다.

```
public class Bank {  
  
    private int money;  
  
    public Bank(final int money) {  
        if (money < 1000) {  
            throw new RuntimeException("1000원 이하로 생성이 불가능해요.");  
        }  
        this.money = money;  
    }  
  
    void transfer(final int money) {  
        this.money -= money;  
        System.out.println(MessageFormat.format("{0}원 입금 완료!!", money));  
    }  
}
```


단점 3. Finalizer를 사용한 클래스는 finalizer 공격에 노출되어 심각한 보안 문제를 일으킬 수도 있다.

```
public class BankAttack extends Bank {  
    public BankAttack(final int money) {  
        super(money);  
    }  
  
    @Override  
    protected void finalize() throws Throwable {  
        super.transfer(1000000000);  
    }  
}
```

```
public class Main {  
    public static void main(final String[] args) throws InterruptedException {  
        Bank bank = null;  
        try {  
            bank = new BankAttack(500);  
            bank.transfer(1000);  
        } catch (Exception e) {  
            System.out.println("예외 터짐");  
        }  
        System.gc();  
        sleep(3000);  
    }  
}
```


final 키워드로 finalizer 공격으로부터 방어할 수 있다.

```
public class BankAttack extends Bank {  
  
    public BankAttack(final int money) {  
        super(money);  
    }  
  
    @Override  
    protected void finalize() throws Throwable {  
        super.transfer(1000000000);  
    }  
}
```

1. 상속 막기 (private 생성자 포함)

```
public final class Bank { ... }
```

2. finalize 메서드 부모 클래스에서 override 막기

```
public class Bank {  
    ...  
    @Override  
    protected final void finalize() throws Throwable {  
    }  
}
```

결론

finalizer 말고 Cleaner를 사용해서 안전망 역할이나 중요하지 않은 네이티브 자원 회수용으로만 사용하자.

- 불확실성과 성능 저하에 주의해야 한다.
- 자원의 소유자가 try-with-resource를 사용하는게 좋다.

Reference

Inner class: <https://inpa.tistory.com/entry/JAVA-%E2%98%95-%EC%9E%90%EB%B0%94%EC%9D%98-%EB%82%B4%EB%B6%80-%ED%81%B4%EB%9E%98%EC%8A%A4%EB%8A%94-static-%EC%9C%BC%EB%A1%9C-%EC%84%A0%EC%96%B8%ED%95%98%EC%9E%90>

Java의 메모리 관리 - Weak, Soft, Phantom reference 예제: <https://tourspace.tistory.com/42>

Java Phantom Reachable, Phantom Reference 칸:
<https://luckydavekim.github.io/development/back-end/java/phantom-reference-in-java/>

이펙티브 자바 item 8: https://github.com/woowacourse-study/2022-effective-java/blob/main/02%EC%9E%A5/%EC%95%84%EC%9D%B4%ED%85%9C_08/finalizer%EC%99%80_cleaner_%EC%82%AC%EC%9A%A9%EC%9D%84%20%ED%94%BC%ED%95%98%EB%9D%BC.md

Thank you

궁금한 점을 물어보세요