💚

아이템 8) finalizer 와 cleaner 사용을 피하라

- 자바의 2가지 객체 소멸자 finalizer / cleaner 에 대해 알아보자
    - 자바 9 부터는 finalizer 가 deprecated 되었고, cleaner 를 그 대안으로 소개한다

```
// Object 클래스의 메소드로 정의된 finalize()
@Deprecated(since="9")
protected void finalize() throws Throwable { }
```

GC 와 finalize 메소드와의 관계 및 수행 과정

1. **객체가 더 이상 참조되지 않음**
    - GC는 객체의 생존 여부를 판단하기 위해 **더 이상 참조되지 않는 객체를 식별하고** 가비지로 표시
2. **finalize 큐에 등록됨**
    - finalize 큐 : finalize() 가 호출될 객체를 별도로 추적하고 관리하는 데 사용
    - finalize() 메소드를 오버라이딩하지 않은 객체는 해당 단계 수행 X / 가비지 표시된 채로 남음
    - GC 는 finalize() 메소드를 오버라이딩한 객체들을 **finalize 큐에 등록**
3. **finalize() 호출**
    - finalize 큐에 있는 객체들은, 별도의 스레드에서 처리되는데, finalize 큐에서 객체를 가져와 finalize() 를 호출하고 객체 자원 정리함
    - finalize() 메소드가 호출되면 **객체는 아직 메모리에서 살아 있지만 이미 가비지로 표시된 상태**
4. **객체의 메모리 해제**
    - finalize() 메소드가 실행된 후, 객체는 메모리에서 제거됨
- 이 때 중요한 점은 finalize() 메소드 실행 시점이 불확실하다는 것 뿐만 아니라 finalize 큐에 등록되는 시점 또한 예측하기 어려움
- finalize() 메소드를 오버라이드하지 않은 객체는 가비지로 표시된 이후에 더이상 참조 되지 않으면 즉시 회수될 수 있지만, finalize() 메소드를 오버라이드한 객체는 가비지로 표시된 이후에도, finalize() 가 호출될때까지 추가적인 생명주기를 가지니까 수명이 한사이클 정도 길다고 볼 수 있음

finalizer 와 cleaner 는 *즉시 수행된다는 보장이 없어*, 제 때 실행되어야 하는 작업은 절대 할 수 없다

- finalizer, cleaner 를 `얼마나 신속히 수행할 지` 는 전적으로 GC 알고리즘에 달림
- 클래스에서 finalizer 를 달아두면 인스턴스의 자원 회수가 제멋대로 지연될 수 있다 (ex. finalizer 스레드가 다른 애플리케이션 스레드보다 우선순위가 낮아서 실행될 기회가 없음)
- **어떤 스레드가 finalizer 를 수행할 지 명시하지 않으니** 이 문제를 예방할 방법이 없었음
- cleaner 는, **자신을 수행할 스레드를 제어할 수 있다**는 점에서 조금 더 나은 방법이다

finalizer 와 cleaner 는 수행시점 뿐만 아니라, *수행 여부조차 보장하지 않는다*

- **따라서 상태를 영구적으로 수정하는 작업에서는 절대 finalizer, cleaner 에 의존해서는 안됨**
- finalizer 와 cleaner 가 실행될 가능성을 높여줄 수 있는 2가지가 메소드가 존재하지만, 실행을 보장하지는 않는다
- System.gc() : GC 수행을 명령하는 메소드 (GC 발생 시 소멸 대상이 되는 인스턴스는 결정되지만, 이것이 실제 소멸로는 이어지지 않음)
    - 예시로 실제 System.gc() 를 강제로 발생시키는 코드를 보면 약 5,000배 이상의 성능 차이가 발생
      → GC 방식이 무엇이든 관계없이 GC를 수행하는 동안 다른 애플리케이션의 성능에 영향을 미치게 되어, finalizer 와 cleaner 메소드의 실행 가능성을 높여줄 수 있다는 이유만으로 강제적으로 GC 를 발생시키는 것은 좋지 않다

```
<%
long mainTime = System.nanoTime();
for(int outLoop=0; outLoop<10; outLoop++) {
    String aValue = "abcdefghijklmnopqrstuvwxyz"
    for(int loop=0; loop<10; loop++) {
        aValue += aValue;
    }
    System.gc();
}
double mainTimeElapsed = (System.nanoTime() - mainTime) / 1000000.000;
out.println("<BR><B>"+mainTimeElapsed+"</B><BR><BR>");
%>
```

구분	응답 시간
System.gc() 메서드 포함	750ms ~ 850ms
System.gc() 메서드 미포함	0.13ms ~ 0.16ms

- System.runFinalization() : GC 에 의해 소멸이 결정된 인스턴스를 즉시 소멸시키는 메소드
- finalizer 동작 중 발생한 예외는 무시되고, 처리할 작업이 남은 경우에도 그 순간 종료됨(Oracle 문서에도 명시되어 있듯이 finalize() 메소드 실행 도중 발생한 예외는 아무도 처리할 수 없음)
  - finalize() 메소드를 실행하는 스레드는 `유저 애플리케이션 컨텍스트가 없기 때문`
  - finalize() 를 실행하는 스레드는 또 별도로 생성되기 때문에 이 오버헤드 또한 감수해야함

→ 이 말이 무슨 뜻이냐하면…. **finalize() 메소드가 실행되는 환경이 별도의 컨텍스트에서 동작**하며, 사용자 애플리케이션의 **일반적인 예외처리 메커니즘을 사용할 수 없다** (사용자 애플리케이션 레벨에서 처리하거나 로깅하는 것이 불가능) 는 것을 의미함!

→ finalize() 메소드는 GC 내부 작업 중 하나로 실행되기 때문에, **사용자 애플리케이션의 일반적인 실행흐름과는 분리된 컨텍스트에서 동작**하며, **사용자 애플리케이션 코드와 직접 상호작용 할 수 없기 때문에 코드 단으로 예외정보가 전달되지 않아**, 예외처리나 로깅이 불가능한 것

  - 예외를 무시할 수 없다면, finalize() 메소드 대신 AutoCloseable 인터페이스를 구현하거나, try-with-resources 블록 활용하는 것이 더 안전하고 예측가능한 방법임!
  - cleaner 를 사용하는 경우에는 **자신의 스레드를 통제하기 때문에** (=사용자 어플리케이션의 컨텍스트에서 실행되므로, 스레드간 동기화 문제와 예외처리 제약사항이 크게 완화됨) 이러한 문제는 발생하지 않음
    - 보통 추천되는 방법 (AutoCloseable을 구현해서 try-with-resource 와 같이 사용한다. (이 편이 추천된다)

finalizer / cleaner 의 심각한 성능 문제가 동반된다

- 안전망 방식으로 했을 때는 AutoCloseable 을 사용하는 것보다 5배 느림
- finalizer 를 사용한 객체를 생성/파괴한 경우 AutoCloseable 을 사용하는 것보다 50배 느림
  - finalizer 가 GC 의 효율을 떨어뜨리기 때문 → 왜 효율을 떨어뜨릴까?
    1. **호출 불확실성** : GC 는 JVM 스케줄에 따라 수행되고, 객체의 finalize 메소드 실행하는 스레드 또한 GC 와 별도의 스레드여서 이로 인해 finalize 메소드의 실행 시점 예측이 어려움
    2. **추가 오버헤드 발생** : finalize() 메소드는 모든 객체에 대해 호출될 필요는 없지만, GC가 이를 결정하기 위해 추가 오버헤드를 발생 / finalize() 메소드 자체가 객체의 정리 작업을 수행하므로 이 작업이 느릴 경우 가비지 컬렉션 프로세스 전체가 느려질 수 있음
       - **분명 finalize 메소드 수행하는 스레드랑 GC 스레드가 별개로 실행된다고 했는데 왜 finalize() 작업 수행이 GC 성능에 영향을 미칠까?**
         1. finalizer 큐 관리가 GC 에 의해 수행되므로
         2. finalize() 메소드에 의한 객체 클린업 작업은 GC 튜닝을 어렵게 만듦
    3. **리소스 누출 가능성** : finalize() 내에서 예외가 발생하면 해당 예외는 무시되며, 객체의 정리 작업이 중단
    4. **잠재적인 데드락** : 두 객체가 서로를 참조하고 finalize() 메소드에서 서로를 정리하려고 시도하는 경우 데드락이 발생

finalizer 공격에 노출되어 심각한 보안 문제가 발생할 수도 있다

- finalizer 공격 : 객체의 finalize() 메소드를 악용하여 객체를 살리려고 시도하는 공격

1. 생성자나 직렬화 과정에서 예외 발생 시, 생성되다 만 객체에서 **악의적인 하위 클래스의 finalizer 가 수행될 수 있게 됨**

- 생성자나 직렬화 과정에서 예외발생하는 경우 코드 예제를 찾고 싶었으나 미숙한 검색능력으로 찾지 못하여 하위 클래스에서 악의적인 예외를 발생시키는 예제로 대체합니다...

```
class MaliciousFinalizer extends Zombie {
    public void finalize() { // 3. Zombie 클래스의 finalize() 메소드를 오버라이드
        try {
            // 4. 악의적인 동작: 부모 클래스의 finalize() 메소드 호출
            super.finalize();
        } catch (Throwable t) {
            // 6. 악의적인 동작: 상위 클래스에서 발생한 예외 악용
            System.out.println("부모 클래스의 finalize() 예외를 악용: " + t.getMessage());
        }
    }
}

public class Zombie {
    static Zombie zombie;

    public void finalize() {
        zombie = this;

        // 5. 악의적인 예외 발생
        int x = 1 / 0;
    }

    public static void main(String[] args) {
        MaliciousFinalizer malicious = new MaliciousFinalizer();

        // 1. 명시적인 null 할당을 통해 객체에 대한 참조 끊고
        malicious = null;

        // 2. GC 실행 요청
        System.gc();
    }
}
```

2. **정적 필드에 자신의 참조를 할당**하여, GC 가 수집하지 못하게 막을 수도 있다
    - finalizer 메서드가 호출되면 zombie static 변수에 this가 저장된다.
    - 그러므로 finalize() 내에서 zombie 필드가 현재 객체를 참조하게 되므로, GC 가 Zombie 객체를 수거할 때 zombie 필드가 여전히 해당 객체를 참조하고 있으므로 해당 객체는 GC 에서 제거되지 않음

```
public class Zombie {
  static Zombie zombie;

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

- final 이 아닌 클래스를 finalizer 공격으로부터 방어하려면, **어떤 작업도 하지 않는 finalize 메소드를 만들고 final 로 선언하자**
    - 클래스를 final 로 : 상속 불가능하게 만들어서, 하위클래스에서 악의적인 수행을 막음
    - 어떤 작업도 하지 않는 finalize 메소드 만들기 : `finalize()` 메소드를 오버라이드하더라도 아무 작업도 수행하지 않도록 하여 보안 문제 예방

```
public final class MyFinalClass { // 1. 클래스를 final 로 만들기

    // 2. finalize 메소드를 무작위로 호출하는 Finalizer 공격 방어
    @Override
    protected final void finalize() throws Throwable {
        // 아무 작업도 수행하지 않음
        super.finalize();
    }
}
```

finalizer 나 cleaner 를 대신해 줄 방법 : AutoCloseable 구현 후 클라이언트에서 close 메소드 호출하기

```
class Resource implements AutoCloseable {
    public Resource() {
        // 자원 관리를 위한 초기화 작업 수행
    }

    @Override
    public void close() {
        // 클라이언트에서 close() 메소드를 호출할 때 실행될 작업
    }

    public void doSomething() {
        // 다른 동작을 수행하는 메소드
    }
}

public class Client {
    public static void main(String[] args) {
        try (Resource resource = new Resource()) {
            resource.doSomething(); // 클라이언트는 자원을 사용
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- 클라이언트 코드에서 `Resource` 객체를 try-with-resources 블록으로 래핑하면, `try` 블록을 벗어날 때 자동으로 `close()` 메소드가 호출됨
- 클라이언트는 `Resource` 객체를 사용한 후 명시적으로 `close()`를 호출하지 않아도 됨
- 각 인스턴스는 자신이 닫혔는지 추적하는 것이 좋음
  - close 메소드에서 이 객체가 더이상 유효하지 않음을 필드에 기록하고, 다른 메소드가 이 필드를 검사해서 객체가 닫힌 후에 불렸다면 `IllegalStateException` 을 던지는 것

finalizer / cleaner 의 적절한 쓰임새 2가지

1. 자원의 소유자가 close 메소드를 호출하지 않는 것에 대비한 **안전망** 역할
   - 즉시 호출되리라는 보장은 없지만, 클라이언트가 하지 않은 **자원 회수를 늦게라도 해주는 것이 아예 안해주는 것보단 낫기 때문** !
   - 자바 라이브러리 일부 클래스에서 안전망 역할의 finalizer 제공하는 경우 : FileInputStream, FileOutPutStream, ThreadPoolExecutor
     - `FileInputStream`, `FileOutputStream`, 및 `ThreadPoolExecutor`는 Java 7 이후 `finalize()` 메소드를 사용하지 않고 있으며, 대신 자원 관리에 다른 메커니즘을 사용
     - 예시) FileInputStream 은 finalize() 대신 try-with-resources 구문 사용 (AutoCloseable 인터페이스를 구현한 클래스의 인스턴스를 자동으로 닫아주는 구문)

     ```
     try (FileInputStream fis = new FileInputStream("example.txt")) {
         // 파일 읽기 작업 수행
     } catch (IOException e) {
         // 예외 처리
     }
     ```

2. 네이티브 피어와 연결된 객체에서 활용될 수 있다.
   - 네이티브 피어 : C/C++이나 어셈블리 프로그램을 컴파일한 기계어 프로그램
     - 주로 하드웨어 접근이나, 기존의 C/C++ 코드의 통합과 같이, **자바로 직접 처리하기 어려운 작업을 수행할 때 사용됨**
   - 자바 피어가 로딩될 때 정적으로 System.loadLibrary() 메소드를 호출해 네이티브 피어를 로딩하고 네이티브 메소드는 native 키워드를 사용해 호출하는 방식으로 사용

```
public class NativePeerExample {
    // 네이티브 메소드를 선언
    public native void callNativeMethod();

    // 정적 초기화 블록에서 네이티브 라이브러리 로딩
    static {
```

```
        System.loadLibrary("NativeLibrary");
    }

    public static void main(String[] args) {
        NativePeerExample example = new NativePeerExample();
        example.callNativeMethod();
    }
}
```

```
// 네이티브 메소드를 구현한 C/C++ 코드
#include <jni.h>

JNIEXPORT void JNICALL Java_NativePeerExample_callNativeMethod(JNIEnv *env, jobject obj) {
    printf("%s", "Hello World!\n");
}
```

- 네이티브 피어 ≠ 자바 객체이므로 GC 가 그 존재를 알지 못함
- 그 결과 자바 피어를 회수할 때 네이티브 객체까지 회수 못함
  → finalizer / cleaner 가 나서서 처리하기에 적당한 작업 (성능 저하 감당할 수 있고, 네이티브 피어가 심각한 자원을 가지고 있지 않을 때만 해당됨)

✅ finalizer 와 달리 cleaner 는 클래스의 public API 에 나타나지 않는다 라는 말이 뭘까 ?!

- finalize() 는 객체의 Object 클래스에서 상속된 메소드이며, 이 메소드는 모든 Java 클래스에서 사용 가능
- 반면에 Cleaner를 사용하는 클래스는 내부적으로 Cleaner.Cleanable 객체를 생성하여 객체의 자원 관리와 정리를 위임함
  - Cleaner와 관련된 코드는 클래스의 public API에 직접 노출되지 않음
  - `Cleanable` 객체는 가비지 컬렉션에 의해 클린업 (정리) 작업이 수행되어야 하는 객체를 추적하고 해당 객체에 대한 클린업 작업 실행

Cleaner 를 안전망으로 활용하는 AutoCloseable 클래스 예제

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

    // 청소가 필요한 자원. 절대 Room을 참조해서는 안 된다!
    private static class State implements Runnable {
        int numJunkPiles;

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

        // Room 클래스의 close 메서드, 또는 Cleaner가 호출
        @Override public void run() {
            System.out.println("Cleaning room");
            numJunkPiles = 0;
        }
    }

    // 방의 상태. cleanable과 공유한다.
    private final State state;

    // cleanable 객체. 수거 대상이 되면 방을 청소한다.
    private final Cleaner.Cleanable cleanable;

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

    public void doSomething() { ... }

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

public class Client {
    public static void main(String[] args) {
        try (Room room = new Room(5)) {
```

```
            room.doSomething();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

1. Room 생성자에서 State 객체 생성 / Cleaner.Cleanable 객체를 생성하고 해당 객체를 GC 에 의해 클린업 되어야 하는 대상 객체와 연결

    ```
    public Cleaner.Cleanable register(Object target, Runnable cleanupAction);
    ```

    - `target` : 클린업 작업이 필요한 대상 (이 예제에서는 Room)
    - `cleanupAction` : 클린업 작업을 정의하는 Runnable 객체 (클린업 작업을 수행할 내용을 정의)
2. try-with-resources 문에서 try 블록을 벗어날 때 Room 클래스의 close() 호출
3. close() 에서 cleanable 의 clean() 호출
4. clean() 안에서 run() 호출 (Runnable 을 구현하는 State 의 run() 호출)

- GC 가 Room 회수할 때까지 클라이언트가 close() 호출하지 않는다면, cleaner가 State 의 run() 을 호출해줄 것이다.
- State 인스턴스가 Room 인스턴스를 참조할 경우 순환참조가 생겨 GC 가 Room 인스턴스를 회수해 갈 기회조차 오지 않는다
    - 정적이 아닌 중첩 클래스는 자동으로 바깥 객체의 참조를 갖게 되기 때문에, 순환 참조가 생기지 않으려면 State 를 **정적 중첩 클래스로 선언**해야 한다

결론

- cleaner 는 자원 회수를 늦게라도 해주는 게 아예 안해주는 것보단 낫다는 안전망 역할을 하거나, 중요하지 않은 네이티브 자원 회수용으로만 사용하자!