2장 객체 생성과 파괴

아이템1. 생성자 대신 정적 팩터리 메서드를 고려하라.

1-1. 정적 팩터리 메서드의 장점

- 이름을 가질 수 있다.

  - 하나의 시그니처로 하나의 생성자를 만들 수 있지만 매개변수의 수나 순서를 다르게 하는 방법 등으로 생성자 제한을 피하는 것은 좋지 않은 방법이다.(어떤 역할을 하는지 정확히 알기 힘들다.)

  - 여러개가 필요할 것 같으면 정적 팩터리 메서드를 사용하고, 이름을 잘 짓자.

- 호출될 때마다 인스턴스를 새로 생성하지는 않아도 된다.

  - 불변 클래스는 인스턴스를 미리 만들어 놓거나 새로 생성한 인스턴스를 캐싱(저장)하여 재활용하는 방식으로 불필요한 객체 생성을 피할 수 있다.

  - 대표적인 예로 `Boolean.valueOf(boolean)`메서드는 객체를 아예 생성하지 않는다.

- 반환 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다.

  - 반환 객체의 클래스를 자유롭게 선택할 수 있는 유연성을 갖는다.

  - API를 만들 때 구현 클래스를 공개하지 않고도 객체를 반환할 수 있어 API를 작게 유지할 수 있다.

- 입력 매개변수에 따라 매번 다른 클래스의 객체를 반환할 수 있다.

  - 반환 타입의 하위타입이긴만 하면 어떤 클래스의 객체를 반환하든 상관없다.

- 정적 팩터리 메서드를 작성하는 시점에는 반활할 객체의 클래스가 존재하지 않아도 된다.

  - 반환값이 인터페이스여도 된다.

  정적 팩터리의 메서드 변경 없이 구현체를 바꿔 끼울 수 있다. → 유연한 시스템

아이템 2. 생성자에 매개변수가 많다면 빌더를 고려하라.

여러 생성자 패턴

- **심층적 생성자 패턴**: 다양한 매개변수 좋바을 가진 생성자를 생성해서 쓰는 패턴. 가독성이 좋지 않으며, 모든 조합을 작성하지 않는 한 불필요한 값을 넣어야할 수 있다.
- **자바빈즈 패턴:** 매개변수 없는 생성자 + 세터 조합이며, 하나의 객체를 생성하기 위해 세터 메서드를 여러번 호출해야하고, 그 때 까지 일관성이 유지되지 않음.
  freeze 메서드를 사용할 수 있지만 컴파일러가 보증하지 못해 런타임 에러에 취약하다.

빌더패턴

- 필수 매개변수만 호출한다. build 메서드를 호출해 최종 객체를 얻는다. 일반적으로 정적 멤버 클래스로 만들며, 빌더 패턴은 유효성검사를 build메서드나 메서드에서 호출하는 생성자에서 검사를 하면 된다.
- 검사 후 잘못된 점을 발견하면 어떤 매개변수가 잘못되었는지를 자세히 알려주는 메시지를 담아 IllegalArgumentExpection을 던지면 된다.
- 빌더 생성 비용이 있기 때문에 매개변수가 4개 이상인 경우만, 하지만 API는 시간이 지날수록 매개변수가 많아지는 경향이 있기 때문에 잘 고려해야한다.

아이템 3. private 생성자나 열거 타입으로 싱글턴임을 보증하라.

싱글턴 : 인스턴스를 오직 하나만 생성할 수 있는 클래스

ex) 함수

- 싱글턴을 만드는 방법

  1.싱글턴 인스턴스를 public static final 필드로 만들고 생성자를 private으로 한다.

  - 간결하고 API에 싱글턴임이 명백하게 드러난다.
  - 단점: 리플렉션 API를 통해 두번째 객체 생성 가능 → 예외처리를 통해 방어가능

2.정적 팩터리 메서드를 public static 멤버로 제공

- 장점: 팩터리 메서드만 수정하면 언제든지 싱글턴이 아니게 바꿀 수 있으며, 제네릭 생글턴 팩트로리로 만듦으로써 타입에 유연하게 대처가 가능하다. 또한 공급자(Supplier는 매개변수를 받지 않고 단순히 반환하는 추상메서드가 존재한다.) 로도 사용가능하다.

- **1,2 번의 문제점**
  - 각 클래스를 직렬화한 후 역직렬화할 때 새로운 인스턴스를 만들어서 반환한다. 이를 방지하기 위해 readResolve()에서 싱글턴 인스턴스를 반환하고 모든 필드에 transient(일시적) 키워드를 넣는다.
  - 직렬화: 객체들의 데이터를 연속적인 데이터로 변형하여 전송 가능한 형태로 만드는 것(데이터→전송가능한 형태로 만듦)
  - 역 직렬화: 직렬화된 데이터를 다시 객체의 형태로 만드는 것

3. **원소가 하나인 enum 타입 선언**
   - 대부분의 상황에서 가장 좋은 방식 → 기본적으로 직렬화가 되어있다.

아이템 4. 인스턴스화를 막으려거든 private 생성자를 사용하라.

- 인스턴스화를 할 수 없는 클래스
  - 정적 멤버만 담은 클래스(Collections, Array)
- 생성자를 명시하지 않으면 자동으로 퍼블릭 기본 생성자가 생성되기 때문에 인스턴스화를 원하지 않으면 직접 private 기본 생성자를 작성해야 한다.
- 확실히 하고 싶다면 private 생성자에 Expection을 thorw하는 코드를 적성하자.
- private 생성자는 인스턴스화를 막을 뿐 아니라 상속을 막는 효과도 있다.

아이템 5. 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라.

- 클래스가 하나 이상의 자원에 의존하고 그 자원이 클래스의 동작에 영향을 준다면 정적 유틸리티 클래스나 싱글턴 방식은 적합하지 않다.

ex) 사전 객체에 의존하는 맞춤법 검사기 클래스를 정적 유틸리티 클래스나 싱글턴 클래스로 만들려는 경우

→ 사전객체의 종류가 바뀔 수 있다는 판단을 하지 못한 것

- 해결 방법: 인스턴스를 생성할 때 생성자에 필요한 자원을 넘겨줘야한다.

DI패턴: 객체가 실행시간에 자신의 의존성을 외부로부터 받아들이도록 설계하는 방법

- 생성자에 자원 팩터리를 넘겨줘야한다. 이때, 한정적 와일드카드 타입을 통해 Supplier 타입을 제한해야한다.

아이템 6. 불필요한 객체생성을 피하라.

객체 생성이 잦은데 그 비용이 크다면 불필요한 객체 생성을 피하면서 자원을 절약해야한다.

1. 같은 값임에도 다른 레퍼런스인 경우

- 같은 값을 가지고 동시에 불변객체라면 같은 레퍼런스를 가져야한다. 같은 값을 가지는 인스턴스가 두 개 이상 만들어 지는 것은 불필요한 객체 생성이다.
- 대표적 불변 클래스인 String은 같은 문자열이라면 같은 레퍼런스를 가진다. 하지만 새로운 객체를 생성한다면 같은 문자열임에도 다른 레퍼런스를 가지게 된다.

  레퍼런스: Object 변수나 함수가 저장된 곳

2. 재사용 빈도가 높고 생성비용이 비싼 경우

- 생성비용이 비싼 객체라면 캐싱을 고려하자.

3. 같은 인스턴스를 대변하는 여러개의 인스턴스를 생성하지 않도록 하자.

4. 오토 박싱

- 불필요한 박싱/언박싱은 불필요한 메모리 할당, 재할당을 반복할 수 있다. 꼭 박싱 타입이 필요한 경우가 아니라면 기본 타입을 사용하도록 하자.

아이템 7. 다 쓴 객체 참조를 해제해라.

자바를 사용할 때 'GC'에 의존하여 메모리 자원 관리에 소홀해서는 안된다. GC의 손이 닿지 않는 자원이 생기지 않도록 주의하자.

메모리 누수 case

1. **자기 자신의 자원을 관리하는 클래스**
    - ex) 스택이 자원을 삭제하기만 한다고 가비지 컬렉션의 대상이 되지는 않는다. 일반적으로 스택 size -1을 해줄 뿐이다.
    - 해결방법: null 처리를 할 수 있으나 무분별한 처리는 코드를 복잡하게 만들 수 있다.
2. **캐시**
    - 캐시에 담아놓기만 하고 정리를 하지 않는다면 쌓이고 쌓여 결국 제 역할을 하지 하게 된다.
    - 해결 방법: weakHashMap
        - 캐시에 넣어놓고 더이상 쓰지 않아도 들어있을 수 있다. 일반적인 HashMap은 사용 여부에 관계없이 Key-Value를 삭제하지 않지만 WeakHashMap은 더 이상 사용되지 않는다고 판단되면 삭제해준다.

      HashMap: 데이터를 저장할 때 key와 value가 짝으로 저장된다. 데이터를 저장할 때는 키값으로 해시함수를 실행한 결과를 통해 저장 위치를 결정한다. 특정 데이터의 저장위치를 해시함수를 통해 바로알 수 있기 때문에 검색이 빠르다는 장점을 지님, 키값은 중복 x, 밸류는 키값이 다르다면 중복이 가능

        - 캐시의 경우 시간이 지남에 따라 사용되지 않으면 그 가치를 떨어뜨리는 방법을 사용한다.

3. **리스너, 콜백**
    - 리스너와 콜백을 등록만 하고 해지를 하지 않으면 메모리 낭비이다.
        - 해결방법: 약한 참조로 넣어서 가비지 컬렉터의 소거 대상이 되도록 하자.

        - 약한 참조(Weak Reference): 객체에 대한 참조를 유지하되, 가비지 컬렉터가 해당 객체를 수거하는 것을 방해하지 않는 참조 방식

아이템 8. finalizer 와 cleaner사용을 피하라.

- **사용시기와 사용 유무가 모두 불확실하고 느리다.**
    - System.gc, System.runFinalization으로 GC 실행의 가능성을 높여줄 수는 있지만 보장하지는 않는다.
- System.runFinalizersOnExit, Runtime.runFinalizersOnExit은 finalizer 실행을 보장하지만 쓰레드가 멈출 수 있다.
- finalizer 동작 중 예외 발생 시 종료될 수 있다.
- 공격에 취약하다.

해결 방법

- 아무것도 하지 않는 finalize 메서드를 만들고 final로 선언하자.

- **올바른 방법**
    - AutoCloseable 인터페이스 구현
    - close() 메서드
    - try-with-resources
    - try 블록이 끝날 close 호출
- **finalizer와 cleaner를 사용해야할 때**
    - try-with-resources 를 사용하지 않을 때를 대비하여 안정망 용도로 사용
    - 중요한 리소스가 아닌 네이티브 피어와 연결된 객체

      네이티브 피어: 일반 자바객체가 네이티브 메서드(네이티브 언어로 작성한 메서드)를 통해 기능을 위임한 네이티브 객체

아이템 9. try-finally 보다는 try-with-resources를 사용하라.

try-finally의 문제점

- try 문과 finally문 모두에서 Exception이 발생한다면 finally에서 발생한 예외만 보이기 때문에 디버깅하기 어렵다.

→ try-with-resources를 사용하자!

close문에서 예외 발생 시 close에서 발생한 예외는 숨겨지고 try문에서의 예외가 기록된다. 숨겨진 예외들도 스택 추적 내역에 숨겨졌다는 표시를 달고 출력된다.