이펙티브 자바 객체 생성과 파괴

item1. 생성자 대신 정적 팩토리 메서드를 고려하라.

- 객체 생성을 할때 생성자가 아닌 정적 팩토리 메서드를 객체 생성 용도로 쓰는 것은 경우에 따라 좋다.

장점 1: 생성자가 이름을 가질 수 있다.

예를 들면 Integer.parseInt()는 정적 팩토리 메서드이다.

이름에서 무엇을 의미하는지 파악하기 쉬움 → 의도가 명확함

장점 2: 매 호출시 인스턴스를 새로 생성할 필요가 없다.

```
public final class Boolean implements java.io.Serializable,
                                      Comparable<Boolean>
{
    /**
     * The {@code Boolean} object corresponding to the primitive
     * value {@code true}.
     */
    public static final Boolean TRUE = new Boolean(true);

    /**
     * The {@code Boolean} object corresponding to the primitive
     * value {@code false}.
     */
    public static final Boolean FALSE = new Boolean(false);
```

```
...
  public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
...
```

Boolean은 true와 false 뿐이고 이는 새로 인스턴스를 만들 필요없이 static한 객체를 반환한다.

이를 플라이웨이트 패턴이라고도 한다.

인스턴스 통제는 싱글톤, 인스턴스화 불가, 1개의 동치 보장에 다양하게 사용됨.

장점 3: 반환 타입의 하위 타입을 반환하는 것도 가능하다.

인터페이스를 반환하는 메서드를 만들고, 매개변수에 따라 구현체등을 반환하는 것이 가능하게 할 수 있다.

- 기본 클래스 생성자는 인터페이스 반환이 불가능함
- 인터페이스를 기반으로 코딩하는 것이 일반적으로 좋은 습관이다. → 유연성 제공

장점 4: 인자에 따라 매번 다른 클래스의 객체를 반환할 수 있다.

인터페이스를 반환하면, 하위 타입 어떤 객체를 반환하든 상관없다.

장점 3번과 비슷한 내용이다.

```
/**
 * Creates an empty enum set with the specified element type
 *
 * @param <E> The class of the elements in the set
 * @param elementType the class object of the element type t
 *     set
 * @return An empty enum set of the specified type.
```

```
     * @throws NullPointerException if <tt>elementType</tt> is n
     */
    public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E>
        Enum<?>[] universe = getUniverse(elementType);
        if (universe == null)
            throw new ClassCastException(elementType + " not an

        if (universe.length <= 64)
            return new RegularEnumSet<>(elementType, universe);
        else
            return new JumboEnumSet<>(elementType, universe);
    }
```

Enumset은 데이터 64개를 기준으로 어떤 객체를 반환할지 결정한다.

장점 5: 정적 팩토리 메서드 작성 시점에는 반환할 객체의 클래스가 존재하지 않아도 된다.

JDBC는 대표적인 서비스 제공자 프레임 워크이다.

서비스 제공자 프레임워크 → 서비스 인터페이스, 제공자 등록 API, 서비스 접근 API

서비스 제공자 인터페이스가 쓰이기도함 → 없다면 리플렉션을 사용함

- 서비스 인터페이스: Connection
- 제공자 등록 API: DriverManager.registerDriver
- 서비스 접근 API: DriverManager.getConnection
- 서비스 제공자 인터페이스: Driver

서비스 접근 API에게 클라이언트는 원하는 구현체의 조건을 명시한다.

JDBC가 어떤 DB를 사용할지, DB접속경로, 아이디, 패스워드 등..

조건을 명시하지 않으면, 기본 구현체를 하나씩 돌아가며 반환함

서비스 제공자 인터페이스가 사용되면 인스턴스를 생성하는 팩토리 객체를 설명해줌.

단점 1: 상속은 protected이상의 생성자가 필요하므로 정적 팩토리 메서드만 제공할 경우, 상속할 수 없음

근데 상속을 지양하는게 나은 방향일 수 있음 → 컴포지션

불변 타입으로 만드는 것이 좋다는 것을 생각하면 단점이 아닐 수 도 있음

단점 2: 정적 팩토리 메서드를 다른 개발자가 찾기 어려움

생성자처럼 API에 명확히 드러나지 않기 때문이다.

이름 규약을 지켜야하는 노력이 필요함

정리

- 단순히 public 생성자를 만들기전에 정적 팩토리 메서드를 생각해보자
- 유연하게 인터페이스를 반환하고 싶을때 사용하자
- 생성자만으로 구분이 어려울때 이름을 주는것이 좋을때

item2. 생성자에 매개변수가 많다면, 빌더 패턴을 고려하라

- 클래스 내부에 멤버 필드가 매우 많다면 각 경우마다 생성자를 만드는 것은 무리이다.
- 이를 자바빈즈 패턴으로 해결가능하긴 하다.
- 자바빈즈 → 빈생성자로 초기화후 필요한 값을 setter로 설정 → 객체가 완전히 생성되기전까지 일관성이 깨짐 → 필드를 불변으로 만들 수 없음 → Threadsafe작업이 추가로 필요함

빌드 패턴을 사용하면?

- 상당히 유연하다
- 빌더 하나로 여러 객체를 순회하며 만들 수도 있음
- 넘기는 매개변수에 따라 다른 객체를 만들 수 있음

단 빌더라는 객체 하나를 더 생성하는 비용 자체가 더 비쌀 수 있음
→ 이를 잘 판단해서 사용해야함

item3. private 생성자나 열거 타입으로 싱글턴임을 보증하라

싱글턴 패턴의 쓰임새 → 무상태 객체, 설계상 유일해야 하는 시스템 컴포넌트

단점

- 테스트가 어려워짐 → mock이 어려워짐

구현법

- 생성자를 private로 감춤
- 인스턴스를 private static final 멤버에 생성해놓고 불러 쓴다.
- 공개된 public static 메서드로 인스턴스를 반환해서 사용함

```
class TS {

    private static final TS instance = new TS();
```

```
    private TS() {
    }

    public static TS getInstance(){
        return instance;
    }
}
```

멤버변수를 public으로 열지 않고, 메서드를 통해 주는 이유는?

- 추후 싱글턴이 아니라 새로운 객체를 반환하고 싶을때, 변경이 용이함
- Supplier로서 사용 가능함

열거 타입 방식의 싱글턴

```
enum TS {
    INSTANCE;
}
```

- 원소가 하나 뿐인 열거 타입이 싱글턴을 만드는 가장 이상적인 방법
- 직렬화, 리플렉션 공격에 완벽함
- Enum 이외의 클래스를 상속해야 한다면 사용 불가함

item4. 인스턴스화를 막으려면 private 생성자를 사용하라

언제 인스턴스화를 막아야할까?

→ java.lang.Math, java.util.Arrays 와 같은 유틸 클래스를 구성하는 경우 해당 클래스가 쓸데없이 인스턴스화되는 것을 막아야 함

1. 특정 인터페이스를 구현하는 객체 생성 팩토리 메서드를 모아놓는다.

```
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {
        return new SynchronizedMap<>(m);
}
```

- Map 인터페이스 타입 객체를 넣으면, SynchronizedMap을 반환해주는 정적 메서드이다.
- 인터페이스에 몰아넣으면 인스턴스화 걱정이 필요없다.

2. 특정 인터페이스를 구현하는 객체 생성 팩토리 메서드를 모아놓는다.

```
public class Collections {
    // Suppresses default constructor, ensuring non-instantiabi
    private Collections() {
    }
...

public static <T extends Comparable<? super T>> void sort(List<T
        list.sort(null);
}
...
```

- 객체를 생성하지는 않지만, 정렬을 해주는 기능을 함
- Collections는 인스턴스화를 위해 사용되지않음 → 애초에 생성자가 막혀있음

3. final 클래스와 관련된 메서드들을 모아 놓는다.

final 클래스를 상속해 하위 클래스에 메서드를 넣는 것은 불가능 하기에 특정 클래스 내부에 몰아넣고 final 클래스와 함께 쓴다.

추상 클래스로는 인스턴스화를 막을 수 없음

→ 상속을 통해 하위 클래스를 만들기 가능 + 추상 클래스를 상속해서 쓰란느 의도로 오해할 수 있기때문에 더 큰 문제가 됨

private 생성자를 통해 인스턴스화를 막을 수 있다.

앞서 살펴본 Collections 클래스도 private로 막혀있다.

상속도 불가하다. → super()를 통해 상위 클래스의 생성자 호출이 불가하기 때문이다.

물론 리플렉션은 막을 수 없다. 하지만 아래와 같은 방식으로 예외를 터트릴 수는 있다.

```
public class Collections {
    // Suppresses default constructor, ensuring non-instantiabi
    private Collections() {
        throw new AssertionsError();
    }
}
```

item5. 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라

정적 유티리티 클래스와 싱글턴의 한계

- 정적 유틸리티 클래스와 싱글턴 패턴은 객체지향의 장점을 버리면서 특수한 목적으로 사용했던 코드 패턴이다. → 남용시 변화 대응 능력이 사라짐

- JDBC와 같은 라이브러리를 만든다고 하면, 내부에 사용할 DB 드라이버를 외부에서 주입 받아야 한다.

```
public class SpellChecker {

  private final Lexicon dictionary;

  public SpellChecker(Lexicon dictionary) {
    this.dictionary = Objects.requireNonNull(dictionary);
  }

  public boolean isValid(Stirng word) { ... }
  public List<String> suggestions(String typo) { ... }
}
```

인스턴스를 생성할 때, 필요한 객체를 넘겨줌으로써 여러가지 상황에 유연하게 대응할 수 있다.

유연성이 높아짐 + 테스트 용이

- 클래스가 하나 이상의 자원에 의존하고, 그 자원이 클래스 동작에 영향을 준다면, 싱글턴 및 정적 유틸리티 클래스 어울리지 않는다.
- 클래스가 해당 자원을 직접 만들게 하지 말고, 주입을 사용하자

item6. 불필요한 객체 생성을 피하라

객체를 매번 생성하고 지우는 것을 반복하면 큰 비용이 든다.

계속 같은 내용의 객체를 사용하는 것이라면 불변 객체를 만들어 놓고 재사용하자.

```
// 1
String s = new String("taesoo");

// 2
String s = "taesoo"

// 3
Boolean bool = new Boolean(true);
```

1번의 경우 JVM 문자열 풀에서 가져오지 않을 이유가 없기에 안티패턴이다.

3번의 경우에도 쓸모없는 코드이다.

정규 표현식에서도 불필요한 객체 생성이 존재한다.

```
static boolean isRomanNumeral(String s) {
  return s.matches("^(?=.)M*(C[MD]|D?C{0,3})"
                 +"(X|[CL]|L?X{0,3}(I[XV]|V?I{0,3}))");
}
```

- String.matchs() 메서드는 정규표현식으로 해당 문자열이 내가 원하는 형태인지 확인하는 가장 쉬운 방법이다.

```
public static Pattern compile(String regex) {
    return new Pattern(regex, 0);
}
```

- String.matchs() 메서드의 구현을 따라가보면, 위처럼 Pattern 인스턴스를 생성하는 부분이 있다.
- 위 과정에서 생성되는 Pattern인스턴스는 한번 쓰고 버려져 가비지컬렉션된다.
- 정규표현식에 해당되는 유한상태머신을 만들기 때문에 인스턴스 생성 비용이 높다.

- 내부적으로 Pattern을 생성하는 로직을 밖으로 빼서 재활용하면 객체 생성을 1번만 하고 성능을 높일 수 있다.

재사용해도 될때 → 객체가 불변인 경우

반대로 불변이 아니라면 재사용을 조심해야한다.

primitive 오토 박싱의 함정

```
public static void unintendedBoxing(Integer number1, Integer nur
  System.out.println(number1 + number2);
}
```

위와 같은 메서드가 있을 때, 일반 int 타입을 넘기더라도 파라미터 타입에 의해 오토 박싱이 일어난다. 이 메서드를 이용해 대량의 작업을 하면 분명히 그냥 int 타입을 활용하는 것보다 훨씬 느리다.

item7. 다 쓴 객체 참조를 해제하라

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }
```

```
    ...
    public Object pop() {
        if (size == 0) {
            throw new EmptyStackException();
        }

        return elements[--size];
    }
        ...
}
```

위와 같은 일반적인 스택 코드가 있다면 메모리 누수가 발생한다.

- pop 부분에서 elements[size] = null; 을 해주지 않으면, 해당 element가 활성 영역에서 벗어나도 JVM은 인식하지 못한다.
- 가비지 컬렌션 언어에서는 메모리 누수를 찾기가 생각보다 까다롭다.
- 객체 참조 하나가 살아있다면 → 참조하는 다른 모든 객체도 살아있기 때문이다.

실제 스택을 구현한 코드는 제대로 처리되고있다.

```
    /**
     * Removes the object at the top of this stack and returns t
     * object as the value of this function.
     *
     * @return  The object at the top of this stack (the last it
     *          of the <tt>Vector</tt> object).
     * @throws  EmptyStackException  if this stack is empty.
     */
    public synchronized E pop() {
        E       obj;
        int     len = size();
```

```
        obj = peek();
        removeElementAt(len - 1);

        return obj;
    }


    public synchronized void removeElementAt(int index) {
        modCount++;
        if (index >= elementCount) {
            throw new ArrayIndexOutOfBoundsException(index + " >
                                                     elementCou
        }
        else if (index < 0) {
            throw new ArrayIndexOutOfBoundsException(index);
        }
        int j = elementCount - index - 1;
        if (j > 0) {
            System.arraycopy(elementData, index + 1, elementDat
        }
        elementCount--;
        elementData[elementCount] = null; /* to let gc do its wo
    }
```

추가로 만약 의도치 않게 활성 영역이 아닌 객체를 참조하면 NullPointerException을 띄워주는 이점도 존재한다.

근데 실제 코딩을하면서 참조값을 null 처리하는 것은 매우 예외적인 경우여야한다.

그러면 어떻게 해결해야하는가? → 최소한의 스코프에서 변수를 이용하고 버려야한다.

사용하지 않는 객체를 참조하는 변수는 자연스레 스코프 밖으로 밀려나 가비지 컬렉터에 의해 처리된다.

캐시도 메모리 누수의 주범이 될 수 있다.

- 보통 캐시를 구현하면, Map 자료구조 형태로 구현하는 일이 흔하다.
- Map의 key가 객체를 참조하고 있다면, 해당 객체와 해당 객체가 참조하는 모든 객체는 계속 살아있는 상태가 될 것이다.
- WeakHashMap에서는 내부적으로 Entry가 WeakReference를 상속하여 구현되기 때문에 key를 null처리 하는 순간 Entry의 key가 가비지 컬렉터에 의해 회수된다.

메모리 누수는 일어나도 별다른 버그나 에러가 없는 경우가 많기에 간과하기 쉽다. 철저한 코드 리뷰나 힙 메모리 분석을 통해서야 마침내 발견되기도 한다. 예방법을 익혀두어 미연에 방지하는 것이 좋다.

item8. finalizer와 cleaner 사용을 피하라

자바에서 2가지 객체 소멸자를 제공하는게 그게 finalizer와 cleaner 이다.

하지만 이 2가지는 지양되는데 그 이유는 가비지 컬렉터에 의해 실행이 결정되고, 즉시 실행된다는 보장이 없기 때문이다.

- ex) 시스템이 동시에 열 수 있는 파일의 갯수는 한정되어 있다.
  - 열었던 파일을 닫아주지 않으면, 더이상 새로운 파일을 열지 못한다.
  - finalizer나 cleaner는 실행 시점이 불명확하기 때문에, 파일이 정말 닫혔는지 알 수 없다.
  - 이로 인해 많은 에러가 발생 가능하다.
  - 자바는 심지어 finalizer와 cleaner의 실행 여부조차 보장해주지 않는다.

finalizer와 cleaner 대신 AutoCloseable 인터페이스를 구현하자.

- 일반적인 리소스 회수는 try-with-resources를 통해 AutoCloseable인터페이스를 구현하는 게 낫다고 배웠다.
- cleaner와 finalizer는 .close() 메서드를 호출하지 않았을 때를 대비해 안전망 역할로 제공할 수 있다.
    - FileInputStream, FileOutputStream, ThreadPoolExecutor가 이러한 방식을 사용하고 있다.
- 네이티브 피어와 연결된 객체를 회수할 때도 cleaner와 finalizer를 사용할 수 있다.
    - 네이티브 피어란 네이티브 메서드를 통해 기능을 위임한 네이티브 객체인데 JVM에 의해 발견되지 않아 자동회수가 되지 않는다.

- cleaner(자바8까지는 finalizer)는 오직 네이티브 자원 회수 용도 혹은 안전망 역할로만 활용하자.
- 대신 AutoClosable을 구현하고, try-with-resources를 적극 활용하자.

item9. try-finally보다는 try-with-resources를 사용하라

- 자바에서는 .close() 메서드를 통해 자원을 닫아줘야 하는 경우가 있다.
    - 외부의 리소스를 사용했을 때
    - 외부의 리소스는 로컬 PC 환경에 존재하는 리소스일 수도 있고, 네트워크로 연결된 다른 컴퓨터에 존재하는 리소스일 수도 있다.
- finalizer를 안전망으로 활용하긴 하지만 딱히 믿을만하진 않다.

try-finally 방식

```
static void copy(String src, String dst) throws IOException {
    InputStream in = new FileInputStream(src);
```

```
    try {
        OutputStream out = new FileOutputStream(src);
        try {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0) {
                out.write(buf, 0, n);
            }
        } finally {
            out.close();
        }
    } finally {
        in.close();
    }
}
```

반환할 자원이 두개가 되면 들여쓰기가 좀 더 늘어나서 조금 더 헷갈리게 보이게 된다.

- 반환할 자원이 여러개일 때, 들여쓰기 때문에 코드가 지저분해진다.
- 만일 예외가 발생하면 try 블록과 finally블록에서 전부 예외가 발생할 수 있는데, finally 블록의 예외가 try 블록의 예외를 잡아먹게 된다.
    - 이 경우 디버깅이 매우 어려워질 수 있다.

try-with-resources 방식

AutoCloseable을 상속받아 close()메서드만 구현해주면 된다.

```
static void copy(String src, String dst) throws IOException {
    try(InputStream in = new FileInputStream(src);
    OutputStream out = new FileOutputStream(src)) {
```

```
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0) {
            out.write(buf, 0, n);
        }
    }
}
```

- 반환할 자원이 몇개더라도, 코드가 지저분해지지 않는다.
- 모든 리소스에 대한 예외 catch가 가능하기 때문에 예외를 묶어서 한번에 처리가능하다.