EJ

이펙티브 자바 CP.1

작성 일시	@2022년 12월 30일 오후 7:23
최종 편집 일시	@2023년 1월 27일 오후 6:42
유형	이펙티브 자바
작성자	
참석자	

1️⃣ 객체 생성과 파괴

▼ 1. 생성자 대신 정적 팩토리 메소드를 고려해라

- public ClassName(), static getWhat()
- 클래스는 `public` 생성자 대신 정적 팩토리 메소드를 제공할 수 있다.
- 정적 팩토리 메소드의 장점
  1. 이름을 가질 수 있다.

     명확화 할 수 있다.
  2. 호출할 때마다 인스턴스를 새로 생성하지 않아도 된다.

     lib 같이 자주 쓰는 파일에 활용 가능해보인다.
  3. 반환 타입을 하위 타입 객체로 반환할 수 있는 능력이 있다.
  4. 입력 매개변수에 따라 매번 다른 클래스의 객체를 반환할 수 있다.

     3, 4 항목 안에서

     이건 단점이 아닌가 생각 했지만, Map 을 예로 생각해보면 편하다. `TreeMap`, `LinkedMap`, `HashMap` 등을 입력 변수에 따라 변경할 수 있다는건, 유연한 개발이 가능 할 것으로 보인다.
  5. 정적 팩토리 메소드는 작성하는 시점에는 반환할 객체의 클래스가 존재하지 않아도 된다.

```
// Order 라는 interface 안에 구현체가 없어도 반환 가능하다.
public static List<Order> getOrders() {
  return new ArrayList<>();
}

interface Order {
}
```

6. 서비스 제공자 프레임워크를 만드는 배경이된다. (ex. JDBC)

- 정적 팩토리 메소드의 단점

1. 상속을 하려면 `public` 이나 `protected` 생성자가 필요하니, 정적 팩터리 메소드만 제공하면 하위 클래스를 만들 수 없다.

   어쩌면, 이 제약은 상속보다 컴포지션(아이템 18)을 사용한다면, 오히려 장점일 수 있다.

2. 정적 패터리 메소드는 프로그래머가 찾기 어렵다.

   흔히 사용하는 명명 방식

```
// from 매개수를 하나 받아서 해당 타입의 인스턴스를 반환하는 형변환 메소드
Data d = Date.from(instant);

// of 여러 매개변수를 받아 적합한 타입의 인스턴스를 반환하는 집계 메소드
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);

// instance 혹은 getInstance 매개변수로 명시한 인스턴스를 반환하지만, 같은 인스턴스임을 보장하지 않음
StackWalker luke = StackWalker.getInstance(option);

// create 혹은 newInstance, 위와 같지만, 매번 새로운 인스턴스를 반환함을 보장함
Object newArray = Array.newInstance(classObject, arrayLen);

// getType getInstance 와 같으나, 생성할 클래스가 아닌 다른 클래스에 팩터리 메소드를 정의할 때 사용한다.
FileStore fs = Files.getFileStore(path);

// newType newInstance 와 같으나, 위와 같음
BufferedReader br = Files.newBufferedReader(paht);

// type 위 getType, newType의 간결한 버전
List<Complaint> litany = Collections.list(legacyLitany);
```

▼ 2. 생성자에 매개 변수가 많다면 빌더를 고려해라

정적 팩토리 메소드든, 생성자든 선택적 매개 변수가 많다면, 점층적 생성자 패턴, 자비빈즈 패턴, 빌더 패턴 가 있다.

1. 점층적 생성자 패턴

```
public class Unit {
  private final int hp;
  private final int mp;
  private final int moveSpeed;
  private final int damage;
  private final int armor;
  private final int shield;
```

```
  public Unit(int hp, int moveSpeed, int damage, int armor) {
    Unit(hp, 0, moveSpeed, damage, armor, 0);
  }

  public Unit(int hp, int mp, int moveSpeed, int damage, int armor) {
    Unit(hp, mp, moveSpeed, damage, armor, 0);
  }

  public Unit(int hp, int moveSpeed, int damage, int armor) {
    Unit(hp, 0, moveSpeed, damage, armor, 0);
  }

  public Unit(int hp, int mp, int moveSpeed, int damage, int armor, int shield) {
    this.hp = hp;
    this.mp = mp;
    this.moveSpeed = moveSpeed;
    this.damage = damage;
    this.armor = armor;
    this.shield = shield;
  }
}
```

- 이 클래스의 인스턴스를 만들려면 원하는 매개변수를 모두 포함한 생성자 중 가장 짧은 것을 골라 호출하면 된다. 여기서 몇몇 오버로딩된 생성자들은 0이라는 값을 넘기는 매개 변수들이 있다. 지금은 매개 변수 수가 많이 없어서 괜찮게? 보일 수 있지만 수가 100개 이상이라면 해당 코드를 변경하기 어려울 것이다.
- 요약, 사용은 할 수 있겠으나, 매개변수 개수가 많아지면 생성자를 늘려가면서 코드를 작성해야하고, 해당 코드의 변경, 읽기 어려워 진다.

2. 자바빈즈 패턴

```
public class Unit {
  private final int hp;
  private final int mp;
  private final int moveSpeed;
  private final int damage;
  private final int armor;
  private final int shield;

  public Unit () {}
  public void setHp(int hp) { this.hp = hp; }
  public void setMp(int mp) { this.mp = mp; }
  public void setMoveSpeed(int moveSpeed) { this.moveSpeed = moveSpeed; }
  public void setDamage(int damage) { this.damage = damage; }
  public void setArmor(int armor) { this.armor = armor; }
  public void setShield(int shield) { this.shield = shield; }
}

Unit zealot = new Unit();
zealot.setHp(100);
zealot.setMoveSpeed(8);
zealot.setDamage(16);
zealot.setArmor(0;
zealot.setShield(100);
```

- 매개변수가 없는 생성자로 객체를 생성 후 setter 메소드를 통해 매개변수의 값을 설정하는 방식이다.

- 자바빈즈 패턴으로는 생성자 수를 늘리지 않아도 되어 점층적 생성자 패턴의 단점이 더이상 보이지 않는다. 하지만, 단점으로 객체 하나를 만들기 위해 메소드를 여러 개 호출해야 하고, 객체가 완전히 생성되기 전까지는 **일관성(멀티 스레드 환경이라고 가정)**이 무너진 상태이다. 즉, 클래스를 불변(아이템17)으로 만들 수 없으며, 스레드 안정성을 얻기 위해선, 추가 작업을 해야한다.
- 요약, 점층적 생성자 패턴의 단점을 극복할 수 있겠으나, 다른 단점들이 크다.

3. 빌더 패턴

```
public class Unit {
  private final int hp;
  private final int mp;
  private final int moveSpeed;
  private final int damage;
  private final int armor;
  private final int shield;

  public unit(Bulider builder) {
    hp = builder.hp;
    mp = builder.mp;
    moveSpeed = builder.moveSpeed;
    damage = builder.damage;
    armor = builder.armor;
    shield = builder.shield;
  }

  public static class Builder {
    // 필수 매개변수
    private final int hp;
    private final int moveSpeed;
    private final int damage;
    private final int armor;

    // 선택 매개 변수 기본값으로 초기화
    private int mp = 0;
    private int shield = 0;

    public Builder(int hp, int moveSpeed, int damage, int armor) {
      this.hp = hp;
      this.moveSpeed = moveSpeed;
      this.damage = damage;
      this.armor = armor;
    }
    public Builder mp(int val) {
      this.mp = val;
      return this;
    }
    public Builder shield(int val) {
      this.shield = val;
      return this;
    }

    public Unit build() {
      return new Unit(this);
    }

  }
}

Unit unit = new Unit.Builder(100, 6, 16, 0).shield(100).build();
```

- Unit 클래스는 불변(어떤 변경도 허용하지 않겠다는 뜻 - `private`, set 메소드 존재 X)이며, 모든 매개 변수의 기본값을 한 곳에 모아 둔다. 빌더의 setter 메서드는 빌더 자신을 반환하기 때문에, 연쇄적 호출이 가능하다. (aka. 플루언트 API, 면세드 연쇄)
- 빌더 패턴은 계층적으로 설계된 클래스와 함께 사용하기 좋다.
    - 각 계층의 클래스에 관련 빌더를 멤버로 정의하고 추상 클래스는 추상 빌더를, 구체 클래스는 구체 빌더를 갖게 한다.

```
public abstract class Unit {

    public enum Skill { CLOCKING, STORM, BURROW, LOCKDOWN }
    final Set<Skill> skills;
    final int hp;
    final int moveSpeed;
    final int damage;
    final int armor;

    abstract static class Builder<T extends Builder<T>> {
        EnumSet<Skill> skills = EnumSet.noneOf(Skill.class);
        public T addSkill(Skill skill) {
            skills.add(Objects.requireNonNull(skill));
            return self();
        }

        abstract Unit build();

        // 반드시, 하위 클래스는 이 메서드를 오버라이딩 하여 this 를 반환하도록 해야한다.
        protected abstract T self();

    }

    Unit(Builder<?> builder, int hp, int moveSpeed, int damage, int armor) {
        skills = builder.skills.clone(); // item 50 에서 다뤄볼 내용
        this.hp = hp;
        this.moveSpeed = moveSpeed;
        this.damage = damage;
        this.armor = armor;
    }
}

public class Ghost extends Unit {
    private final int mp;

    public static class Builder extends Unit.Builder<Builder> {
        private final int mp;

        public Builder(int mp) {
            this.mp = mp;
        }

        // 오버라이딩에서 반환 타입을 해당 하는 하위 클래스로 반환하도록 한다.
        // aka. 공변 반환 타이핑 - 객채 생성시 캐스팅하지 않아도 된다.
        @Override
        public Ghost build() {
            return new Ghost(this);
        }

        @Override
        protected Builder self() {
            return this;
        }
    }
```

```
    private Ghost(Builder builder) {
        super(builder, 45, 5, 10, 0);
        mp = builder.mp;
    }
}

Ghost ghost = new Ghost.Builder(50)
               .addSkill(Unit.Skill.CLOCKING)
               .addSkill(Unit.Skill.LOCKDOWN)
               .build();
```

- 더 나가아가기, 유효성 검사의 타이밍은 빌더 생성자와 메서드에서 입력 매개변수를 검사하고, build 메서드가 호출하는 생성자에서 여러 매개변수에 걸친 불변식을 검사한다.

▼ 3. private 생정자나 열거 타입으로 싱글턴임을 보증해라

- **싱글턴**이란 인스턴스를 오직 하나만 생성 할 수 있는 클래스를 뜻 함

  주로 stateless 객체나 설계상 유일해야 하는 시스템 컴포넌트

  단점으로는, 이를 사용하는 클라이언트 코드를 테스트하기 어렵다.

- 싱글턴을 만드는 방식

  1. `public` `static` 멤버 변수가 `final` 인 경우
     - 생성자는 `private` 로 감쳐두고, 유일한 인스턴스를 접근할 수단 `public static final` 멤버 변수를 하나 둔다.
     - `private` 생성자는 Singleton.INSTANCE 를 초기화할 때, 딱 한 번 호출된다.
     - `public` 이나 `protected` 생성자가 없음으로 클래스가 초기화 될 때 만들어진 인스턴스가 전체 시스템에서 하나뿐이 보장된다.
     - 장점
       - 해당 클래스가 싱글턴임이 API에 명백히 드러난다. `public static` 필드가 `final`이니 절대로 다른 객체를 참조할 수 없다.
       - 간결하다.

     ```
     public class Singleton {
       public static final Singleton INSTANCE = new Singleton();
       private Singleton() {...}
     }
     ```

  2. 정적 팩터리 메소드를 `public static` 멤버 변수를 제공하는 경우
     - Singleton.getInstance 는 항상 같은 객체의 참조를 반환함으로, 인스턴스를 결코 만들어지지 않는다.
     - 장점
       - 언제든 싱글턴에서 여러 인스턴스를 반환하는 방식으로 변경 가능하다.
       - 원한다면 정적 팩터리를 제네릭 싱글턴 팩터리로 만들 수 있다.

- 정적 팩터리의 메서드 참조를 공급자(supplier)로 사용할 수 있다.

```
Supplier<Singleton> singletonSupplier = Singleton::getInstance;
Singleton singleton = singletonSupplier.get();
```

```
public class Singleton {
  private static final Singleton INSTANCE = new Singleton();
  private Singleton() {...}
  public static Singleton getInstance() { return INSTANCE; }
}
```

- 위 두 방식 모두의 단점
  - 권한이 있는 클라이언트에서 리플렉션 API 인 `AccessibleObject.setAccessible` 을 이용해 `private` 생성자를 호출 할 수 있다.
    - 해결방안 - `private` 생성자에 해당 코드를 추가한다.

```
private Singleton() {
  if(INSTANCE != null){
    throw new RuntimeException("생성자 호출이 불가능합니다.");
  }
}
```

  - 직렬화된 인스턴스를 역직렬화할 때 새로운 인스턴스를 만들어서 반환한다. 역직렬화는 기본 생성자를 호출하지 않고 값을 복사해서 새로운 인스턴스를 반환한다. 그때 사용되는 것이 `readResolve` 라는 메소드이다.
    - 해결방안 - 아래 코드를 추가한다.

```
private Object readResolve() {
  return INSTANCE;
}
```

- 번외 (lazy initialization - 지연 초기화). (백기선님의 싱글톤 강의)
  - 만약, 위 싱글톤 내용(이른 초기화)들이 무거운 객체들이고 언제 사용될지 모를 경우 비효율 적인 객체를 생성해서 보관 할 수 있다. 이를 방지해 맨 처음 사용될 때, 사용하는 방식이 있다.
    - double checked locking - 복잡한 방식

```
public class Singleton {
  public static volatile Singleton INSTANCE;
  private Singleton() {}
  public static Singleton getInstance() {
    if (INSTANCE == null) {
      // 동시 접근시 막기 위함.
      synchronized (Singleton.class) {
        if (INSTANCE == null) {
          INSTANCE = new Singleton();
        }
```

```
      }
    }
    return INSTANCE;
  }
}
```

- static inner class - 비교적 간단한 방식

```
public class Singleton {
  private Singleton() {}

  private static class SingletonHolder {
    private static final Singleton INSTANCE = new Singleton();
  }

  public static Singleton getInstance() {
    return SingletonHolder.INSTANCE;
  }
}
```

▼ 4. 인스턴스화를 막으려거든 private 생성자를 사용해라

- 정적 메소드와, 정적 필드만을 담고 있는 클래스를 만들고 싶을 때가 있을 것이다. 객체 지향적 프로그래밍에 적합하지는 않지만, 분명 쓰임새가 있다. 해당 클래스를 추상 클래스로 만드는 것으로는 인스턴스화를 막을 수는 없다. (상속해서 인스턴스화를 사용하라는 것으로 오해 할 수 있다.) 해당 기능을 하는 클래스를 만들고 싶으면, `private` **생성자**를 사용해서 인스턴스화를 막으면된다.
- 아래는 실제로 사용하는 util 클래스이다.

```
public class DateTimeConvertTimeZoneUtil {
    static final LOCAL_ZONEID = ZoneId.systemDefault();
    static final UTC_ZONEID = ZoneId.of("UTC");

    private DateTimeConvertTimeZoneUtil ()
    {
        throw new IllegalStateException("Utility class");
    }

    public static LocalDateTime localDateTimeToUTCDateTime(LocalDateTime localDateTime) {
        return localDateTime.atZone(LOCAL_ZONEID)
            .withZoneSameInstant(UTC_ZONEID).toLocalDateTime();
    }

    public static LocalDateTime UTCDateTimeToLocalDateTime(LocalDateTime utcDateTime) {
        return utcDateTime.atZone(UTC_ZONEID)
            .withZoneSameInstant(LOCAL_ZONEID).toLocalDateTime();
    }
}
```

▼ 5. 자원을 직접 명시하지 말고 의존 객체를 주입해라

- 사용하는 자원에 따라 동작이 달라지는 클래스는 정적 클래스나 싱글턴 방식이 적합하지 않다. 대신 클래스가 여러 자원 인스턴스를 지원해야 하며, 클라이언트가 원하는 자원을 사용해야 한다. **인스턴스를 생성시 생성자에 필요한 자원을 넘겨 주면 된다.** (의존성 주입)
- 자원이 몇 개든, 의존 관계가 어떻든 상관없이 잘 동작한다.

- 또한, `final` 키워드로 불변을 보장해, 같은 자원을 사용하는 클라이언트가 의존 객체들을 안심하고 공유 할 수 있다.
- 의존 객체 주입은 생성자, 정적 팩터리, 빌더에 모두 같이 응용이 가능하다.
- 의존 객체 주입은 클래스의 유연성, 재사용성, 테스트 용이성을 개선해준다.

```
public class SpellChecker {
  private final Lexicon dictionary;

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

  public boolean isVaild(String word) {...}
  public List<String> suggestion(String word) {...}
}
```

▼ 6. 불 필요한 객체 생성을 피해라

- 똑같은 기능의 객체를 매번 생성하기 보다는 객체 하나를 재사용하는 편이 좋을 때가 많다. 재사용은 빠르고 가시성이 더 좋다. 특히 불변 객체는 언제든 다시 사용 가능하다.
  - String 예시
    - String s1 = new String(”abc”);
      String s2 = new String(”abc”);
      - 생성자 방식으로 String 인스턴스가 두 개가 만들어진다.
    - String s1 = “abc”;
      String s2 = “abc”;
      - s1은 String 리터럴을 사용해 String 상수 풀 (String Constant Pool) 이라는 곳에 “abc” 있는지 체크 후 없으면 값을 저장합니다.
        s2는 String Constant Pool 에 해당 값이 있음으로, 값을 추가하지 않고 해당 상수 풀 주소를 참조하게 됩니다.

```
String gs1 = new String("abc");
String gs2 = new String("abc");
String cs1 = "abc";
String cs2 = "abc";

// false
gs1 == gs2
// false
gs1 == cs1
// true (주소값 참조)
cs1 == cs2
```

    - String Constant Pool 참조
      - https://softworking.tistory.com/497
      - https://jiwondev.tistory.com/114

        - Constant Pool 방식을 사용한다면 같은 JVM 안에서 똑같은 리터럴을 사용하는 모든 코드가 같은 객체를 재사용함이 보장된다.
- 생성자 대신 정적 팩터리 메소드를 제공하는 불변 클래스에서 는 불필요한 객체 생성을 피할 수 있다.
    - `Boolean` 예시
        - `new Boolean(String)` 보다 `Boolean.valueOf(String)` 하는 것이 더 좋다
          `Boolean` 값은 true, false 로 만 구성되는데, 매번 인스턴스를 만드는 것은 메모리 낭비임으로, `Boolean` 클래스의 정적 팩터리 메소드를 사용하는 게 좋다.

```
/**
 * Allocates a {@codeBoolean} object representing the value
 * {@codetrue} if the string argument is not {@codenull}
 * and is equal, ignoring case, to the string {@code"true"}.
 * Otherwise, allocates a {@codeBoolean} object representing the
 * value {@codefalse}.
 *
 *@paramsthe string to be converted to a {@codeBoolean}.
 *
 *@deprecated
* It is rarely appropriate to use this constructor.
 * Use {@link#parseBoolean(String)} to convert a string to a
 * {@codeboolean} primitive, or use {@link#valueOf(String)}
 * to convert a string to a {@codeBoolean} object.
 */
@Deprecated(since="9")
public Boolean(String s) {
    this(parseBoolean(s));
}
```

- 생성 비용이 아주 비싼 객체도 반복해서 필요하다면, 캐싱해서 재사용을 권한다.
    - `String.match` 예시 - 로마 숫자인지 확인하는 메소드
        - 아래에서 로마 숫자가 사용되었는지 확인하는 정규 표현식 형태로 가장 쉬운 방법이지만, 반복 사용에는 적합하지 않다. matched의 파라미터 내용인 `Pattern` 인스턴스는 한 번 쓰고 버려져 바로 GC 대상이 된다. 때문에, Pattern 인스턴스가 될 데이터를 직접 생성해 캐쉬해두고, 나중에 `isRomanNumberal` 이 호출될 때 재사용하면 된다.

```
public class RomanNumerals {
  public 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})$");
  }
}
```

```
public class RomanNumerals {
    private static final Pattern ROMAN = Pattern.compile(
        "^(?=.)M*(C[MD]|D?C{0,3})(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

    public static boolean isRomanNumeral(String s) {
        return ROMAN.matcher(s).matches();
    }
}
```

- 위 예시들은 다 객체가 불변이였다. 이유는 재사용에 있어서 안정성이 보장되기 때문이다.

  하지만 불변이 보장되지 않는 상황도 있는데, 어댑터 패턴을 생각해보면 실제 객체를 연결해주는 제 2의 인터페이스 역할을 하는 어댑터같은 경우 사용자는 이 어댑터를 사용할 때 뒷 단에서 매번 인스턴스를 생성해 반환될 지, 동일한 내용에 대해서 동일한 인스턴스를 반환해줄지 알 수 없다. (즉, 가변일지 불변일지 모르는 상황이 올 수 있다.)

  - `Map interface` 의 `keytSet` 메서드 예시
    - `keySet`을 호출할 때마다 새로운 Set 인스턴스가 만들어질 수도 있겠지만, 사실은 매번 같은 `Set` 인스턴스를 반환 할지도 모른다. 반환된 `Set` 인스턴스가 일반적으로 가변이라도 반환된 인스턴스는 기능적으로 모두 똑같다.
    - `Set` 인스턴스는 가변일지라도 수행하는 기능이 모두 동일하고, 모든 `Set` 인스턴스가 `Map` 인스턴스를 대변하기 때문이다.
    - 실제 `HashMap` 의 `keySet` 구현

      ```
      public Set<K> keySet() {
      //keySet은 HashMap 의 abstract class 인 AbstractMap 의 필드로 지정되어있다.
          Set<K> ks = keySet;
          if (ks == null) {
              ks = new KeySet();
              keySet = ks;
          }
          return ks;
      }
      ```

- 오토박싱 - 기본 타입과 boxing 된 기본 타입을 섞어 쓸 때, 자동으로 상호 변환해주는 기술.
  - 오토박싱은 기본 타입과 그에 대응하는 기본 타입의 구분을 흐려 주지만, 완전히 없애주는게 아니다. (`double` 은 null 이 못 들어감, `Double` 은 가능함 - 이 내용을 잘 생각해보면, 오토박싱 기능은 성능에서 좋지 않은 점을 가져다 줄 것이다.)
  - 왠만해서는 박싱되지 않은 타입보다 기본 타입을 사용하고 의도치 않은 오토박싱이 있는지 주의하자.
- 오해 요소가 있는 부분
  - 요즘 JVM 에서는 작은 객체를 생성하고 회수하는 일에 크게 부담되지 않는다. (GC 발전)
    - 프로그램의 명확성, 간결성, 기능을 위해서 객체를 추가로 생성하는 건 일반적으론 좋다.
    - 반대로, 단순 객체 생성을 피하고자 **객체 풀**을 만들지는 말자. 물론 DB connect 처럼 객체 생성 비용이 명확히 아는 경우에는 객체 풀을 사용해 재사용하는 편이 낫다.
- 이번 내용은 방어적 복사 (아이템 50)과 대조적이다.
  방어적 복사가 필요한 상황에서 객체를 재사용할 때의 피해가, 필요 없는 객체를 반복 생성했을 때의 피해보다 훨씬 크다는 사실을 기억하자. 반복 생성의 부작용은 코드 형태와 성능에만 영향을 주지만, 방어적 복사가 실패하면 버그와 보안 문제로 이어진다.

▼ 7. 다 쓴 객체 참조를 해제해라

- 책의 적힌 **문제가 있는** `Stack` 코드

```
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 void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() throws Exception {
        if (size == 0) {
            throw new IllegalStateException("stack 사이즈가 0일 경우 오류가 발생합니다");
        }
        return elements[--size];
    }

    private void ensureCapacity() {
        if (elements.length == size) {
            elements = Arrays.copyOf(elements, 2 * size + 1);
        }
    }
}
```

- 별 문제는 없어 보이는 코드이지만, 프로그램을 오래 사용하다 보면 GC 활동과 메모리 사용량이 늘어나면서, 디스크 페이징이나 OutOfMemoryError 를 발생 시킬 것이다.
- 위 코드에서 문제점은, `pop` 메소드 호출시 만들어진 객체들을 GC 가 회수 하지 않아서 문제가 발생된다. 프로그램에서 더 이상 해당 객체를 사용하지 않아도 말이다. 이 `Stack` 이 객체들의 다 쓴 참조(앞으로 다시 쓰지 않을 참조)를 여전히 갖고 있기 때문이다.
- GC 언어에서는 메모리 누수를 찾기가 아주 까다롭다. 객체 참조를 하나 살려두면, GC 그 객체 뿐만 아니라, 그 객체가 참조하는 모든 객체를 회수 하지 못한다… 그래서 단 몇 개 의 객체로 매우 많은 객체를 회수를 못 할 수 있고, 잠재적으로 성능에 악영향을 준다.
- 해법은 간단하다. 해당 참조를 다 썼을 때 `null` 처리(참조 해제)하면 된다.

```
  public Object pop() throws Exception {
    if (size == 0) {
      throw new IllegalStateException("stack 사이즈가 0일 경우 오류가 발생합니다");
    }
    Object result = elements[--size];
    element[size] = null;
    return result;
  }
```

  - 다 쓴 참조를 `null` 처리하면 다른 이점도 따라온다. 만약 `null` 처리한 참조를 실수로 사용하는 경우 프로그램은 바로 `NullPoinerException` 을 던지며 종료된다. (프로그램 오류는 조기에 발견되는게 좋다.)
- 실제 `Stack pop` 메소드 안에 `removeElementAt` 메소드를 호출한다. 해당 메소드는 `Stack` 상위 클래스 `Vector` 에 담긴 내용이다

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

- 객체 참조를 `null` 처리하는 일은 예외적인 경우여야 한다.
  - 다 쓴 참조를 해제하는 **가장 좋은 방법**은 그 참조를 담은 변수를 **유효 범위(scope)** 밖으로 밀어내는 것이다.
  - `null` 처리를 하는 경우는 자기 메모리를 직접 관리하는 경우이다.
    - **자기 메모리를 직접 관리하는 클래스**라면 프로그래머는 항시 메소드 누수에 주의해야한다.
    - `Stack` 클래스로는 `element` 배열로 저장소 풀을 만들어 원소들을 관리한다. 배열 활성 영역에 속한 원소들이 사용되고 비활성 영역은 쓰이지 않는 것은 GC 는 이 사실을 알 수 없다. GC 입장으로는 비활성 영역에서 참조하는 객체도 똑같이 유효한 객체이다. 비활성 영역에 쓸 모 없다는건 프로그래머가 아는 사실임으로 null 처리를 해서 해당 객체를 더는 사용하지 않을 것을 GC 알려야한다.
- 캐시 역시 메모리 누수를 일으키는 주범이다.
  - 객체의 레퍼런스를 캐시에 넣어 놓고, 캐시를 비우는 것을 잊기 쉽다. 여러 가지 해결책이 있지만, **캐시의 키**에 대한 레퍼런스가 캐시 밖에서 필요 없어지면 해당 엔트리를 캐시에서 자동으로 비워주는 `Weak Hash Map`을 쓸 수 있다.
- 리스너 혹은 콜백도 메모리 누수를 일으킨다.
  - 클라이언트가 콜백을 등록만 하고 명확히 해지하지 않는다면, 뭔가 조치를 하지 않는 한 콜백은 계속 쌓여갈 것이다. 이럴 때 콜백이 약한 참조(weak reference)로 지정하면 GC 가 즉시 수거해간다. (ex WeakHashMap 에 키로 저장)
- **Java Reference 와 GC**

▼ 8. finalizer, cleaner 사용을 피해라

- 자바는 두 가지 객체 소멸자를 제공한다. 하지만 대부분 사용해서는 안된다.
  - `finazilzer` 는 예측 할 수 없고, 상황에 따라 위험할 수도 있어 일반적으로 불필요하다. 오동작, 낮은 성능, 이식성 문제의 원인이기도 함.
  - `cleaner` 는 `finazilzer` 만큼 위험하진 않지만, 여전히 예측할 수 없고, 일반적으로 불필요하다.

    - 단점 1. 둘 다 즉시 수행된다는 보장이 없다. 객체에 접근 할수 없게 된 후 `finalizer`나 `cleaner`가 실행되기까지 얼마나 걸릴지 알 수 없다. 즉 `finalizer`와 `cleaner`로는 제때 실행되어야 하는 작업은 절대 할 수 없다.
    - 단점 2. 인스턴스의 자원 회수가 제멋대로 지연될 수 있다. `finalizer` 스레드는 다른 애플리케이션 스레드보다 우선 순위가 낮아 실행될 기회를 제대로 얻지 못할 수 있다. `cleaner`는 자신을 수행할 스레드를 제어할 수는 있지만, 역시나 가비지 컬렉터에 의존하므로 사용하지 않는다.
    - 단점 3. 둘 다 수행 시점뿐만 아니라 수행 여부조차 보장하지 않는다. 접근할수 없는 일부 객체에 딸린 종료 작업을 전혀 수행하지 못한 채 프로그램이 종료될 수도 있다는 이야기이다. 따라서 프로그램 생애주기와 상관없는, 상태를 영구적으로 수정하는 작업에서는 **절대로** 사용하면 안된다. 예를 들어 DB 같은 공유 자원의 영구 lock 해제를 `finalizer` 나 `cleaner`에 맡겨 놓으면 분산 시스템 전체가 서서히 멈출 것이다.
        - `System.gc` 나 `System.runFinalization` 메소드에 현혹되지 말자. `finalizer`와 `cleaner`가 실행될 가능성을 높일 수는 있으나 보장해주진 못한다. (이전에 `System.runFinalizersOnExit`, `Runtime.runFinalizersOnExit` 가 있었으나, **ThreadStop** 결함으로 비난 받았다.)
    - 단점 4.
        - `finalizer` 동작 중 발생된 예외를 무시하며, 처리할 작업이 남았어도 그 순간 종료된다. 잡지 못한 예외 때문에 해당 객체는 자칫 마무리가 덜 된 상태로 남을 수 있다. 그리고 다른 스레드가 이처럼 훼손된 객체를 사용하려 한다면 어떻게 동작할지 예측할 수 없다. 보통의 경우엔 잡지 못한 예외가 스레드를 중단시키고 스택 추적 내역을 보여주지만, `finalizer` 에서는 경고 조차 보여주지 않는다.
        - `cleaner`는 자신의 스레드를 통제하기 때문에 위의 문제는 발생하지 않는다.
    - 단점 5. 심각한 성능 문제
        - GC 의 효율을 떨어 트린다.
    - 단점 6. `finalizer` 를 사용한 클래스는 `finalizer `공격에 노출되어 심각한 보안 문제를 일으킬 수도 있다.
        - finalizer 공격
- 정상적으로 자원 반납을 하는 방법으로는 `AutoCloseable` 을 구현 한다.
    - 클라이언트에서 인스턴스를 다 쓰고 나면 close 메소드를 호출 해주면 된다. (일반적으로 예외가 발생되어도 제대로 종료되도록 **try-with-resources** 를 사용한다.)
        - 각 인스턴스는 자신이 닫혔는지를 추적하는 것이 좋다. 다시 말해, `close` 메서드에서 이 객체는 더 이상 유효하지 않음을 필드에 기록하고, 다른 메서드는 이 필드를 검사해서 객체가 닫힌 후 불렸다면 `IllegalStateException` 을 던진다.
- 그럼 `finalizer`, `cleaner` 는 언제 쓰이나? ~~(그냥 쓰지말자...)~~
    1. 자원의 소유자가 `close()` 메소드를 호출하지 않는 것에 대비한 안전망 역할이다.
    클라이언트가 하지 않은 자원 회수를 늦게라도 하는게 더 나으니까 그렇다..

2. native peer 와 연결된 객체. native peer란 일반 자바 객체가 네이티브 메소드를 통해 기능을 위임한 네이티브 객체를 뜻한다. 네이티브 피어는 자가 객체가 아니니 GC에 대상이 아니다. 그 결과 자바 피어를 회수 할 때, 네이티브 객체까지 회수 하지 못한다. cleaner 나 finalizer 가 처리하기 좋다. 단, 성능 저하를 감당하고 네이티브 피어가 심각한 자원을 가지고 있지 않을 때만 해당된다.

▼ 9. try-finally 보다는 try-with-resources를 사용해라

- 자바 라이브러리에는 close 메소드를 호출해 직접 닫아줘야 하는 자원(`InputStream`, `OutputStream`, `sql.Connection` 등)이 많다. 자원 닫기는 클라이언트가 놓치기 쉬워 예측 할 수 없는 성능 문제로 이어지기도 한다. (안전망으로 `finalizer`, `cleaner` 으로 되어있긴 하지만, 믿을 수없다 아이템8)
- 전통적으로 자원 닫힘을 보장하는 수단으로, `try-finally` 로 쓰였다.

```
static String firstLineOfFile(String path) throw IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        br.close();
    }
}
```

  - 나쁘지 않지만, 자원을 두 개 이상 사용한다면 어떨까?

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

    - 위와 같은 코드에는 문제점이 있다.
      - 예외는 `try`, `finally` 문에 모두 발생할 수 있다.
        - `firstLineOfFile` 메소드 안의 `readLine` 메서드가 예외를 던지고, 같은 이유로 `close` 메소드도 실패 할 것이다. 이런 상황이라면, 두 번째 예외가 첫 번째 예외를 완전히 집어 삼켜 버려, 스택 추적 내역에 첫 번째 예외 정보가 나오지 않게 되어, 실제 시스템에서의 디버깅을 어렵게 한다.
      - 가독성이 좋지 않다.
- 위 문제들은 자바 7 `try-with-resources` 덕에 모두 해결되었다. `AutoCloseable` 인터페이스를 구현해야한다.

- `try-with-resources` 로 적용된 코드

```
static String firstLineOfFile(String path) throw IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

```
static void copy(String src, String dst) throw IOException {
    try (InputStream in = new FilrInputStream(src);
        OutputStream out = new FileOutputStream(dst)) {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

- `close` 메소드를 호출하지 않아도 자동으로 블럭 이후 `close` 메소드가 호출된다.
- `Suppressed`를 통해 단계별로 발생된 모든 `Exception`을 확인할 수 있다.
- `catch` 로 인해 다수의 예외를 처리 할 수 있다.
- `try-finally` 보다 가독성이 좋다.