1장(들어가기)

- 자바가 지원하는 타입 : **인터페이스, 클래스, 배열, 기본 타입**

  → 애너테이션 : 인터페이스의 일종

  → 열거 타입 (enum) : 클래스의 일종

- 아무것도 명시하지 않은 접근 수준 : package-private
- API를 사용하는 프로그램 작성자를 그 **API의 사용자**
- API를 사용하는 클래스(코드)는 그 **API의 클라이언트**

- 패키지의 공개 API는 그 패키지의 모든 public 클래스와 인터페이스의 **public** 혹은 **protected** 멤버와 생성자로 구성된다.

  → protected : 동일 패키지 내에 존재하거나 파생 클래스(자식 클래스)에서만 접근 가능

- 자바 9에서는 **모듈 시스템** 개념이 더해짐

  → 모듈 개념을 적용하면 공개 API는 '해당 라이브러리의 모듈 선언에서 공개하겠다고 한' 패키지들의 공개 API만으로 이뤄진다. 즉, **공개할 패키지를 선택할 수 있다.**

2장(객체 생성과 파괴)

▶ 아이템1. 생성자 대신 정적 팩토리 메서드를 고려하라

- 정적 팩토리 메서드가 생성자보다 좋은 **장점**

1. **이름을 가질 수 있다.**
   → 반환될 객체의 특성을 쉽게 묘사할 수 있다.
2. **호출될 때마다 인스턴스를 새로 생성하지 않아도 된다.**
   →
   불변 클래스는 인스턴스를 미리 만들어 놓거나 새로 생성한 인스턴스를 캐싱하여 재활용하는 식으로 불필요한 객체 생성을 피할 수 있다.

   → **플라이웨이트 패턴**도 이와 비슷한 기법임 (자주 변하는 속성(extrinsit)과 변하지 않는 속성(intrinsit)을 분리하여 변하지 않는 속성 캐시하여 재사용)
   → 인스턴스 통제 시, 클래스를
   싱글톤으로 만들 수도, **인스턴스화 불가**로 만들 수도 있다.
3. **반환 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다.**

   → 인터페이스를 정적 팩터리 메서드의 반환 타입으로 사용하는 인터페이스 기반 프레임워크를 만드는 핵심 기술이기도 하다.
4. **입력 매개변수에 따라 매번 다른 클래스의 객체를 반환할 수 있다.**
5. **정적 팩터리 메서드를 작성하는 시점에는 반환할 객체의 클래스가 존재하지 않아도 된다.**

   → 서비스 제공자 프레임 워크(ex. JDBC)를 만드는 근간이 된다.

> 💡 서비스 제공자 프레임 워크는 **3개의 핵심 컴포넌트**로 이뤄짐
>
> **1. 서비스 인터페이스** : 구현체의 동작 정의
> → Connection
>
> **2. 제공자 등록 API** : 제공자가 구현체를 등록할 때 사용
> → DriverManager.registerDriver
>
> **3. 서비스 접근 API** : 클라이언트가 서비스의 인스턴스를 얻을 때 사용
> → DriverManager.getConnection
> +)
> **4. 서비스 제공자 인터페이스** : 서비스 인터페이스의 인스턴스를 생성하는 팩터리 객체
> → Driver

- 정적 팩토리 메서드의 **단점**

1. **상속을 하려면 public이나 protected 생성자가 필요하니 정적 팩터리 메서드만 제공하면 하위 클래스를 만들 수 없다.**
   → 이는 상속보다 컴포지션 사용을 유도하기 때문에 오히려 장점일 수 있다.

2. **정적 팩터리 메서드는 프로그래머가 찾기 어렵다.**
   → 생성자 처럼 API 설명에 명확히 드러나지 않으므로, 널리 알려진 규약을 따라 짓는 식으로정적 팩터리 메서드 이름을 지어서 문제를 완화해야함

▶ 아이템2. 생성자에 매개변수가 많다면 빌더를 고려하라

- **생성자 혹은 정적 팩터리의 구현 방법**

1. **점층적 생성자 패턴(생성자 여러개)**
   → 매개변수 개수가 많아지면 클라이언트 코드를 작성하거나 읽기 어렵다.

2. **자바빈즈 패턴(매개변수 없는 생성자에 setter 메서드)**

   → 객체가 완전히 생성되기 전까지 일관성이 무너진 상태에 놓이게 된다.
   → 클래스를 불변으로 만들 수 없다.

3. **빌더 패턴(필수 매개변수만으로 생성자를 호출해 빌더 객체를 얻은 후, setter 메서드로 선택 매개변수 설정)**

   → 계층적으로 설계된 클래스와 함께 쓰기에 좋다.

▶ 아이템3. private 생성자나 열거 타입으로 싱글턴임을 보증하라

- 싱글턴의 전형적인 예 : 무상태 객체, 시스템 컴포넌트

- 싱글턴을 만드는 방식 2가지

1. **public static 멤버가 final 필드인 방식**

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

        public void leaveTheBuilding() {...}
}
```

- 장점
  1. 해당 클래스가 싱글턴임이 API에 명백히 드러난다.
  2. public static 필드가 final이니 절대로 다른 객체를 참조할 수 없다.
  3. 간결함

2. **정적 팩터리 메서드를 public static 멤버로 제공하는 방식**

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

- 장점
  1. 마음이 바뀌면 API를 바꾸지 않고도 싱글턴이 아니게 변경할 수 있다.
  2. 정적 팩터리를 제네릭 싱글턴 패턴 팩터리로 만들 수 있다.
  3. 정적 팩터리의 메서드 참조를 공급자로 사용할 수 있다.

3. **원소가 하나인 열거 타입 선언**

```
public enum Elvis {
        INSTANCE;
        public void leaveTheBuilding() {...}
}
```

- 장점
  1. 더 간결하고, 추가 노력 없이 직렬화 할 수 있다.
  2. 만들려는 싱글턴이 Enum 클래스만 상속한다면 열거 타입이 싱글턴을 만드는 가장 좋은 방식이다.

▶ 아이템4. 인스턴스화를 막으려거든 private 생성자를 사용하라

- 정적 멤버만 담은 유틸리티 클래스는 인스턴스로 만들어 쓰려고 설계한 게 아니다. 하지만, 생성자를 명시하지 않으면 컴파일러가 자동으로 기본 생성자를 만들어준다.
  츠
- 추상 클래스로 만드는 것은 인스턴스화를 막을 수 없다.

  → 하위 클래스를 만들어 인스턴스화 하면 그만이다.

- **private 생성자를 추가하면 클래스의 인스턴스화를 막을 수 있다.**

  → 하지만, 직관적이지 않으니 주석으로, 기본 생성자 만드는 것을 막는 것을 표시해두자

▶ 아이템5. 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라

- 많은 클래스가 하나 이상의 자원에 의존한다. 가령 맞춤법 검사기는 사전에 의존하는데, 이런 클래스를 정적 유틸리티 클래스로 구현한 모습을 드물지 않게 볼 수 있다. 맞춤법 검사기가 여러 사전을 사용할 수 있게 만들어보자

  → 이 경우, 정적 유틸리티와 싱글턴은 유연하지 않고, 테스트하기 어려우므로 적합하지 않다.

  → 따라서, 인스턴스를 생성할 때, 생성자에 필요한 자원을 넘겨주는 **의존 객체 주입 방식**을 사용하자

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

        public SpellChecker(Lexicon dictionary){
                this.dictionary = Objects.requireNonNull(d
        }
        public boolean inValid(String word) {...}
        public List<String> suggestions(String typo) {...}
}
```

  → 이 패턴의 변형으로, 생성자에 자원 팩터리를 넘겨주는 방식이 있다.
  팩터리란 호출할 때마다 특정 타입의 인스턴스를 반복해서 만들어 주는 객체를 말한다.

▶ 아이템6. 불필요한 객체 생성을 피하라

- **불필요한 객체 생성 예시 1**

```
String s = new String("bikini"); // 절대 사용 금지
```

→ 생성자에 넘겨진 “bikini” 자체가 이 생성자로 만들어내려는 String과 기능적으로 완전히 똑같으므로 쓸데없는 인스턴스가 만들어진 것이다.

→ new 연산자의 경우 Heap 영역에 할당되고, 재사용이 안되므로

```
String s = "bikini";
```

→ 문자열 리터럴 방식의 경우, String Constant Pool 영역에 할당되므로, 객체를 재사용할 수 있다.

이러한 이유로 Boolean(String) 생성자 대신 Boolean.valueOf(String) 팩터리 메서드를 사용하는 것이 좋다.

- **불필요한 객체 생성 예시 2 - 어댑터가 뒷단 객체 하나당 여러개 생성되는 경우**
  어댑터는 실제 작업은 뒷단 객체에 위임하고, 자신은 제 2의 인터페이스 역할을 해주는 객체이다. → 즉, 뒷단 객체 하나당 어댑터 하나씩만 만들어지면 충분하다.

- **불필요한 객체 생성 예시 3 - 오토 박싱**

```
private static long sum() {
        Long sum = 0L;
        for(long i =0; i <= Integer.MAX_VALUE; i++){
                sum  += i;
        }
        return sum;
}
```

  → Long 선언된 타입을 long으로 고쳐주자

  → 박싱된 기본 타입보다는 기본 타입을 사용하고, 의도치 않은 오토박싱이 숨어들지 않도록 주의하자

▶ 아이템7. 다 쓴 객체 참조를 해제하라

- 다 쓴 참조를 계속 가지고 있으면 메모리 누수가 일어난다.

→ 객체 참조 하나를 살려두면 가비지 컬렉터는 그 객체 뿐만 아니라 그 객체가 참조하는 모든 객체를 회수하지 못한다.

- 해당 참조를 다 썼을 때 **null 처리(참조 해제)**하면 된다.

  → 모든 경우, null처리를 하는 것이 아니다. 오히려 그렇게 하면 코드 가독성이 안좋아진다.

  - **null 처리하는 경우**

  1. Stack 클래스의 경우, 자기 메모리를 직접 관리하기 때문에 null 처리를 해준다.
  2. 캐시도 메모리 누수를 일으킬 수 있도록, 시간이 지날수록 엔트리의 가치를 떨어뜨리는 방식을 사용해 유효기간을 정해준다.
  3. 리스너 혹은 콜백의 경우에도, 콜백 등록만 하고 명확히 해제하지 않으면 메모리 누수가 일어날 수 있다.

▶ 아이템8. finalizer와 cleaner의 사용을 피하라

- 자바의 경우 finalizer와 cleaner라는 객체 소멸자를 제공하지만, 예측할 수 없고, 즉시 수행된다는 보장과 수행 여부 모두 보장하지 않는다.

→ 따라서, 자바에서는 try-with-resources와 try-finally를 사용해 해결한다.

- finalizer와 cleaner를 사용하는 경우

  1. 자원의 소유자가 close 메서드를 호출하지 않는 것에 대비한 안정망 역할
  2. 네이티브 피어와 연결된 객체의 경우, 가비지 컬렉터의 역할 대신 사용

▶ 아이템9. try-finally보다는 try-with-resources를 사용하라

- **자바 라이브러리에는 close 메서드를 호출해 직접 닫아줘야 하는 자원이 많다.**

→ ex) InputStream, OutputStream, java.sql.Connection 등

- 자원 닫기는 클라이언트가 놓치기 쉬워서 예측할 수 없는 성능 문제로 이어지기 때문에, 제대로 닫힘을 보장하기 위한 수단으로 try-finally가 사용되었다.
  - **try-finally의 단점**
    1. **자원이 둘 이상일 경우, 코드가 지저분해진다.**
    2. **기기에 물리적인 문제가 생겨 에러가 발생할 경우, 첫번째 try finally 코드 블록 안의 close 메서드가 실패할 뿐 아니라, 스택 추적 내역에 물리적 문제에 대한 정보는 남지 않는다.**

  → 이러한 문제를 해결하기 위해서는 try-with-resources 사용해야 한다.

  → 이 구조를 사용하기 위해서는 해당 자원이 **AutoCloseable 인터페이스**를 구현해야 한다.