

*Luke's Language*로 파싱된 이펙티브 자바

# 생성자 Yithability

제작. 홍성혁

# 들어가며

“완벽한 언어는 없지만, 훌륭한 언어는 좀 있다.” - *Joshua Bloch*, 졸트상 수상자

# 들어가며

“완벽한 언어는 없지만, 훌륭한 언어는 좀 있다.” - *Joshua Bloch*, *쥘트상* 수상자  
Effective Java를 관통하는 주제.

# 들어가며

“완벽한 언어는 없지만, 훌륭한 언어는 좀 있다.” - *Joshua Bloch*, 졸트상 수상자

Effective Java를 관통하는 주제.

명료성 (Clarity)과 단순성 (Simplicity)

# 들어가며

“완벽한 언어는 없지만, 훌륭한 언어는 좀 있다.” - *Joshua Bloch*, 졸트상 수상자  
Effective Java를 관통하는 주제.

명료성 (Clarity)과 단순성 (Simplicity)

- 컴포넌트는 사용자를 놀라게 하는 동작을 해서는 절대 안 된다.

# 들어가며

“완벽한 언어는 없지만, 훌륭한 언어는 좀 있다.” - *Joshua Bloch*, 졸트상 수상자

Effective Java를 관통하는 주제.

## 명료성 (Clarity)과 단순성 (Simplicity)

- 컴포넌트는 사용자를 놀라게 하는 동작을 해서는 절대 안 된다.  
(있어빌리티 챙겨라)

# 자바의 기본 지원 타입

- interface
- class
- array
- primitive

# 자바의 기본 지원 타입

- interface                      annotation?
- class
- array
- primitive





# 자바의 기본 지원 타입

- interface — annotation
- class
- array
- primitive

# 자바의 기본 지원 타입


- interface — annotation
- class
- array enum?
- primitive

# 자바의 기본 지원 타입

- interface      annotation
- class      enum
- array
- primitive

# 자바의 기본 지원 타입

- interface
- class
- array
- primitive



reference  
type

# 자바의 기본 지원 타입

- interface
- class
- array
- primitive



Object !

# 자바의 기본 지원 타입 - class

## class 구성 요소

- field
- method
- member class
- member interface


# 자바의 기본 지원 타입 - class

class 구성 요소

- field
- method
- member class
- member interface

# 자바의 기본 지원 타입 - class

## class 구성 요소

- field
  - method
  - member class
  - member interface
- method Signature
- 




# 자바의 기본 지원 타입 - class

## class 구성 요소

- field
  - method
  - member class
  - member interface
- method Signature
- method name
- 

# 자바의 기본 지원 타입 - class

## class 구성 요소

- field
  - method
  - member class
  - member interface
- method Signature
- method name
  - parameter
- 

# 자바의 기본 지원 타입 - class

## class 구성 요소

- field
  - method
  - member class
  - member interface
- method Signature
- method name
  - parameter
  - parameter type
- 

# 자바의 기본 지원 타입 - class

## class 구성 요소

- field
  - method
  - member class
  - member interface
- method Signature
- method name
  - parameter
  - parameter type

- *return type*은 메서드 시그니처에 속하지 않는다.

# API

## API 란?

- exported API, Application Programming Interface
- 프로그래머가 클래스, 인터페이스, 패키지를 통해 접근 가능한 클래스, 인터페이스, 생성자, 멤버, 직렬화된 형태 **serialized form** 총칭
- 자바 9 모듈시스템이 더해지며,  
공개 API는 **module declaration**에서 공개 선언한 패키지들로 이루어진다.

# Item1. Static Factory Method

정적 메서드 패턴

- 그 클래스의 인스턴스를 반환하는 단순한 정적 메서드

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

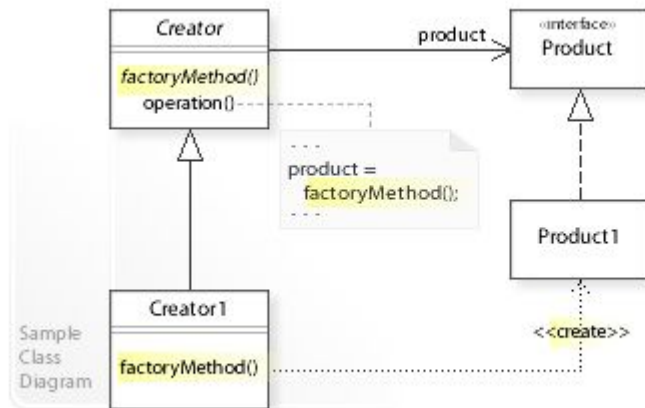
# Item1. Static Factory Method

## 정적 메서드 패턴

- 그 클래스의 인스턴스를 반환하는 단순한 정적 메서드

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

***vs Factory Method***



# Item1. Static Factory Method vs *Factory Method*

## Factory method pattern

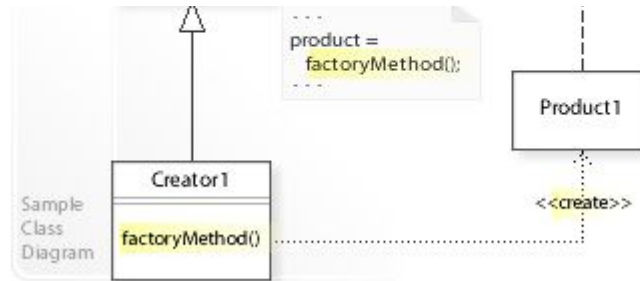
27 languages

Article Talk

Read Edit View history Tools

From Wikipedia, the free encyclopedia

In [object-oriented programming](#), the **factory method pattern** is a [design pattern](#) that uses factory methods to deal with the problem of [creating objects](#) without having to specify their exact [classes](#). Rather than by calling a [constructor](#), this is accomplished by invoking a factory method to create an object. Factory methods can be specified in an [interface](#) and implemented by subclasses or implemented in a base class and optionally [overridden](#) by subclasses. It is one of the 23 classic design patterns described in the book *Design Patterns* (often referred to as the "Gang of Four" or simply "GoF") and is subcategorized as a [creational pattern](#).<sup>[1]</sup>





# Item1. Static Factory Method vs *Factory Method*

## Factory method pattern

🌐 27 languages

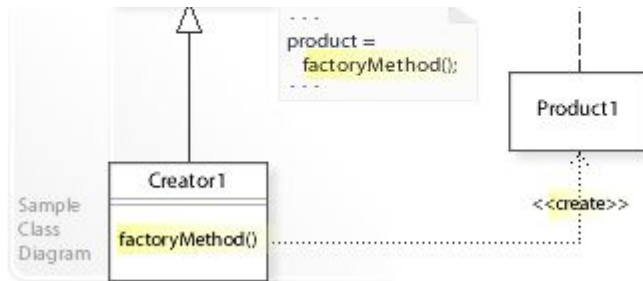
Article Talk

Read Edit View history Tools

From Wikipedia, the free encyclopedia

객체 지향 프로그래밍에서 팩토리 메서드 패턴은 팩토리 메서드를 사용하여 정확한 클래스를 지정하지 않고도 객체를 생성하는 문제를 처리하는 디자인 패턴입니다. 생성자를 호출하는 대신 팩토리 메서드를 호출하여 객체를 생성합니다. 팩토리 메서드는 인터페이스에서 지정하여 하위 클래스에서 구현하거나 기본 클래스에서 구현하여 선택적으로 하위 클래스에서 재정의할 수 있습니다. 이는 *Design Patterns* (종종 "Gang of Four" 또는 간단히 "GoF"라고 함) 책에 설명된 23가지 클래식 디자인 패턴 중 하나이며 생성 패턴으로 하위 분류됩니다. <sup>[1]</sup>

subcategorized as a *creational pattern*.<sup>[1]</sup>



# Item1. Static Factory Method vs *Factory Method*

## Definition [\[ edit \]](#)

According to *Design Patterns: Elements of Reusable Object-Oriented Software*: "Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses."<sup>[2]</sup>

Creating an object often requires complex processes not appropriate to include within a composing object. The object's creation may lead to a significant duplication of code, may require information inaccessible to the composing object, may not provide a sufficient level of abstraction or may otherwise not be included in the composing object's [concerns](#). The factory method design pattern handles these problems by defining a separate [method](#) for creating the objects, which subclasses can then override to specify the [derived type](#) of product that will be created.

The factory method pattern relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects.<sup>[3]</sup> The pattern can also rely on the implementation of an [interface](#).

# Item1. Static Factory Method vs *Factory Method*

## Definition [edit]

*디자인 패턴: 재사용 가능한 객체 지향 소프트웨어의 요소에* 따르면 "객체를 생성하기 위한 인터페이스를 정의하지만 하위 클래스가 인스턴스화할 클래스를 결정하도록 합니다. 팩토리 메서드를 사용하면 클래스가 사용하는 인스턴스화를 하위 클래스에 연기할 수 있습니다." <sup>[2]</sup>

객체를 생성하려면 종종 구성 객체에 포함하기에 적합하지 않은 복잡한 프로세스가 필요합니다. 객체를 생성하면 상당한 코드 중복이 발생할 수 있으며, 구성 객체에서 액세스할 수 없는 정보가 필요할 수 있으며, 충분한 수준의 추상화를 제공하지 못할 수 있거나, 그렇지 않으면 구성 객체의 관심사에 포함되지 않을 수 있습니다. 팩토리 메서드 디자인 패턴은 객체를 생성하기 위한 별도의 메서드를 정의하여 이러한 문제를 처리합니다. 그런 다음 하위 클래스에서 이를 재정의하여 생성될 파생된 제품 유형을 지정할 수 있습니다.

팩토리 메서드 패턴은 객체 생성이 객체를 생성하기 위한 팩토리 메서드를 구현하는 하위 클래스에 위임되므로 상속에 의존합니다. <sup>[3]</sup> 이 패턴은 인터페이스 구현에도 의존할 수 있습니다.

# Item1. Static Factory Method

생성자의 역할이란?

- 2가지

# Item1. Static Factory Method

생성자의 역할이란?

- 2가지

1. 생성

# Item1. Static Factory Method

생성자의 역할이란?

- 2가지
  1. 생성
  2. 메서드

# Item1. Static Factory Method

생성자의 역할이란?

- 2가지
- 1. 생성
- 2. 메서드

## method Signature

- method name
- parameter
- parameter type
- *return type*은 메서드 시그니처에 속하지 않는다.

# Item1. Static Factory Method

생성자의 역할이란?

- 2가지
- 1. 생성
- 2. 메서드

## method Signature

- method name
- parameter
- parameter type
- *return type은 메서드 시그니처에 속하지 않는다.*

파라미터의 값과, 타입을 받는다.

=> 타입 검증!



# Item1. Static Factory Method

- from
- of
- valueOf
- instance
- create
- getType
- newType
- type

```
public interface Tbl extends DataObject, Iterable<TblRow> { 2개 구현  🧑 lshh

    static Tbl empty() { return TblImplement.empty(); }
    static Tbl empty(TblSchema schema) { return TblImplement.empty(schema); }
    static Tbl from(List<?> objectList) { return TblImplement.from(objectList); }
    static Tbl fromObjects(List<?> objectList) { return TblImplement.fromObjects(objectList); }
    static Tbl of(TblSchema schema, List<List<Object>> data) { return TblImplement.of(schema, data); }
    static Tbl fromRowMapList(List<Map<String, Object>> rowMapList) { return TblImplement.fromRowMapList(rowMapList); }
    static Tbl fromColumnListMap(Map<String, List<Object>> columnListMap) {...}

    static TblBuilder builder(TblSchema schema) { return TblBuilder.forRowSet(schema); }
```

## Item1. Static Fa

- from
- of
- valueOf
- instance
- create
- getType
- newType
- type

```
@Test
void testToRowMapListMethod() {
    List<TblColumn> schemas = List.of(
        TblColumn.of(columnName: "column1", String.class),
        TblColumn.of(columnName: "column2", String.class),
        TblColumn.of(columnName: "column3", String.class)
    );
    List<List<Object>> inputData = List.of(
        List.of("A", "B", "C"),
        List.of("D", "E", "F"),
        List.of("G", "H", "I")
    );
    TblSchema schema = TblSchema.from(schemas);
    Tbl tblInstance = Tbl.of(schema, inputData);

    List<Map<String, Object>> result = tblInstance.toRowMapList();

    // Assertions
    assertEquals(expected: 3, result.size());
    assertEquals(expected: "A", result.get(0).get("column1"));
    assertEquals(expected: "B", result.get(0).get("column2"));
    assertEquals(expected: "C", result.get(0).get("column3"));
}
```

# Item1. Static Factory Method - 장점

## 1. 이름을 가진다.

- from
- of
- valueOf
- instance
- create
- getType
- newType
- type

# Item1. Static Fa

## 1. 이름을 가진다.

- from
- of
- valueOf
- instance
- create
- getType
- newType
- type

```
@Test
void testToRowMapListMethod() {
    List<TblColumn> schemas = List.of(
        TblColumn.of(columnName: "column1", String.class),
        TblColumn.of(columnName: "column2", String.class),
        TblColumn.of(columnName: "column3", String.class)
    );
    List<List<Object>> inputData = List.of(
        List.of("A", "B", "C"),
        List.of("D", "E", "F"),
        List.of("G", "H", "I")
    );
    TblSchema schema = TblSchema.from(schemas);
    Tbl tblInstance = Tbl.of(schema, inputData);

    List<Map<String, Object>> result = tblInstance.toRowMapList();

    // Assertions
    assertEquals(expected: 3, result.size());
    assertEquals(expected: "A", result.get(0).get("column1"));
    assertEquals(expected: "B", result.get(0).get("column2"));
    assertEquals(expected: "C", result.get(0).get("column3"));
}
```

Column List에서 유래된 TblSchema

# Item1. Static Fa

## 1. 이름을 가진다.

- from
- of
- valueOf
- instance
- create
- getType
- newType
- type

```
@Test
void testToRowMapListMethod() {
    List<TblColumn> schemas = List.of(
        TblColumn.of(columnName: "column1", String.class),
        TblColumn.of(columnName: "column2", String.class),
        TblColumn.of(columnName: "column3", String.class)
    );
    List<List<Object>> inputData = List.of(
        List.of("A", "B", "C"),
        List.of("D", "E", "F"),
        List.of("G", "H", "I")
    );
    TblSchema schema = TblSchema.from(schemas);
    Tbl tblInstance = Tbl.of(schema, inputData);

    List<Map<String, Object>> result = tblInstance.toRowMapList();

    // Assertions
    assertEquals(expected: 3, result.size());
    assertEquals(expected: "A", result.get(0).get("column1"));
    assertEquals(expected: "B", result.get(0).get("column2"));
    assertEquals(expected: "C", result.get(0).get("column3"));
}
```

schema, inputData로 이루어진 Tbl

# Item1. Static Factory Method - 장점

2. 호출될 때마다 인스턴스 생성이 반드시 필요하지 않다.

```
@SafeVarargs
/varargs/
static <E> List<E> of(E... elements) {
    switch (elements.length) { // implicit null check of elements
        case 0:
            /unchecked/
            var list = (List<E>) ImmutableCollections.EMPTY_LIST;
            return list;
        case 1:
            return new ImmutableCollections.List12<>(elements[0]);
        case 2:
            return new ImmutableCollections.List12<>(elements[0], elements[1]);
        default:
            return ImmutableCollections.listFromArray(elements);
    }
}
```

# Item1. Static Factory Method - 장점

2. 호출될 때마다 인스턴스 생성이 반드시 필요하지 않다.

```
@SafeVarargs
/varargs/
static <E> List<E> of(E... elements) {
    switch (elements.length) { // implicit null check of elements
        case 0:
            /unchecked/
            var list = (List<E>) ImmutableCollections.EMPTY_LIST;
            return list;
        case 1:
            return new ImmutableCollections.List12<>(elements[0]);
        case 2:
            return new ImmutableCollections.List12<>(elements[0], elements[1]);
        default:
            return ImmutableCollections.listFromArray(elements);
    }
}
```



# Item1. Static Factory Method - 장점

## 3. 리턴 타입의 구현 객체를 줄 수 있다.

```
public interface Tbl extends DataObject, Iterable<TblRow> { 2개 구현  🡕 lshh

    static Tbl empty() { return TblImplement.empty(); }
    static Tbl empty(TblSchema schema) { return TblImplement.empty(schema); }
    static Tbl from(List<?> objectList) { return TblImplement.from(objectList); }
    static Tbl fromObjects(List<?> objectList) { return TblImplement.fromObjects(objectList); }
    static Tbl of(TblSchema schema, List<List<Object>> data) { return TblImplement.of(schema, data); }
    static Tbl fromRowMapList(List<Map<String, Object>> rowMapList) { return TblImplement.fromRowMapList(rowMapList); }
    static Tbl fromColumnListMap(Map<String, List<Object>> columnListMap) {...}

    static TblBuilder builder(TblSchema schema) { return TblBuilder.forRowSet(schema); }
```



# Item1. Static Factory Method - 장점

## 3. 리턴 타입의 구현 객체를 줄 수 있다.

```
public interface Tbl extends DataObject, Iterable<TblRow> { 2개 구현  🧑 lshh

    static Tbl empty() { return TblImplement.empty(); }
    static Tbl empty(TblSchema schema) { return TblImplement.empty(schema); }
    static Tbl from(List<?> objectList) { return TblImplement.from(objectList); }
    static Tbl fromObjects(List<?> objectList) { return TblImplement.fromObjects(objectList); }
    static Tbl of(TblSchema schema, List<List<Object>> data) { return TblImplement.of(schema, data); }
    static Tbl fromRowMapList(List<Map<String, Object>> rowMapList) { return TblImplement.fromRowMapList(rowMapList); }
    static Tbl fromColumnListMap(Map<String, List<Object>> columnListMap) {...}

    static TblBuilder builder(TblSchema schema) { return TblBuilder.forRowSet(schema); }
```

## Item1. Static Factory Method - 장점

4. 입력 매개변수에 따라 상태 패턴 가능!

# Item1. Sta

## 4. 입력 매개변

```
public interface SurveyResponseItemValue <T>{ 32개 사용 위치 4개 구현 🧑 Luke SungHyuk Hong
    static SurveyResponseItemValue<?> of(SurveyItemFormType type, Object value) { 🧑 Luke
        return switch (type) {
            case TEXT -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextValue.of(args);
            }
            case TEXTAREA -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextareaValue.of(args);
            }
            case RADIO -> {
                if(!(value instanceof Long optionId)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemRadioValue.of(optionId);
            }
            case CHECKBOX -> {
                if(!(value instanceof Long[] optionIds)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemCheckboxValue.of(optionIds);
            }
        }
    }
};
}
```

# Item1. Sta

## 4. 입력 매개변

```
public interface SurveyResponseItemValue <T>{ 32개 사용 위치 4개 구현 🧑 Luke SungHyuk Hong
    static SurveyResponseItemValue<?> of(SurveyItemFormType type, Object value) { 🧑 Luke
        return switch (type) {
            case TEXT -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextValue.of(args);
            }
            case TEXTAREA -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextareaValue.of(args);
            }
            case RADIO -> {
                if(!(value instanceof Long optionId)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemRadioValue.of(optionId);
            }
            case CHECKBOX -> {
                if(!(value instanceof Long[] optionIds)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemCheckboxValue.of(optionIds);
            }
        };
    }
}
```

# Item1. Sta

## 4. 입력 매개변

```
public interface SurveyResponseItemValue <T>{ 32개 사용 위치 4개 구현 🧑 Luke SungHyuk Hong
    static SurveyResponseItemValue<?> of(SurveyItemFormType type, Object value) { 🧑 Luke
        return switch (type) {
            case TEXT -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextValue.of(args);
            }
            case TEXTAREA -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextareaValue.of(args);
            }
            case RADIO -> {
                if(!(value instanceof Long optionId)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemRadioValue.of(optionId);
            }
            case CHECKBOX -> {
                if(!(value instanceof Long[] optionIds)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemCheckboxValue.of(optionIds);
            }
        };
    }
}
```

# Item1. Sta

## 4. 입력 매개변

```
public interface SurveyResponseItemValue <T>{ 32개 사용 위치 4개 구현 🧑 Luke SungHyuk Hong
    static SurveyResponseItemValue<?> of(SurveyItemFormType type, Object value) { 🧑 Luke
        return switch (type) {
            case TEXT -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextValue.of(args);
            }
            case TEXTAREA -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextareaValue.of(args);
            }
            case RADIO -> {
                if(!(value instanceof Long optionId)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemRadioValue.of(optionId);
            }
            case CHECKBOX -> {
                if(!(value instanceof Long[] optionIds)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemCheckboxValue.of(optionIds);
            }
        };
    }
}
```

# Item1. Sta

## 4. 입력 매개변

```
public interface SurveyResponseItemValue <T>{ 32개 사용 위치 4개 구현 🧑 Luke SungHyuk Hong
    static SurveyResponseItemValue<?> of(SurveyItemFormType type, Object value) { 🧑 Luke
        return switch (type) {
            case TEXT -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextValue.of(args);
            }
            case TEXTAREA -> {
                if(!(value instanceof String args)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemTextareaValue.of(args);
            }
            case RADIO -> {
                if(!(value instanceof Long optionId)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemRadioValue.of(optionId);
            }
            case CHECKBOX -> {
                if(!(value instanceof Long[] optionIds)){
                    throw new IllegalArgumentException("Invalid value: " + value);
                }
                yield SurveyResponseItemCheckboxValue.of(optionIds);
            }
        };
    }
}
```

# Item1. Static Factory Method - 장점

5. 지금 당장 만들어놓지 않아도 됨

대표 사례. JDBC

SPI(Service Provider Interface)라는 것



# Item1. Static Factory Method - 장점

## 5. 지금 당장 만들어놓지 않아도 됨

대표

SPI

### Service provider interface

🌐 5 languages ▼

Article Talk

Read Edit View history Tools ▼

From Wikipedia, the free encyclopedia

**Service provider interface (SPI)** is an [API](#) intended to be implemented or extended by a third party. It can be used to enable framework extension and replaceable components.<sup>[1][2][3]</sup>

#### Details [edit]

From Java documentation:

A service is a well-known set of interfaces and (usually abstract) classes. A service provider is a specific implementation of a service. The classes in a provider typically implement the interfaces and subclass the classes defined in the service itself. Service providers can be installed in an implementation of the Java platform in the form of extensions, that is, jar files placed into any of the usual extension directories. Providers can also be made available by adding them to the application's class path or by some other platform-specific means.<sup>[4]</sup>

# Item1. Static Factory Method - 장점

## 5. 지금 당장 만들어놓지 않아도 됨

대표

SPI

### Service provider interface

🌐 5 languages ▼

Article Talk

Read Edit View history Tools ▼

From Wikipedia, the free encyclopedia

**Service provider interface (SPI)** is an [API](#) intended to be implemented or extended by a third party. It can be used to enable framework extension and replaceable components.<sup>[1][2][3]</sup>

#### Details [\[edit\]](#)

From Java documentation:

서비스는 잘 알려진 인터페이스와 (일반적으로 추상적인) 클래스의 집합입니다. 서비스 제공자는 서비스의 특정 구현입니다. 제공자의 클래스는 일반적으로 인터페이스를 구현하고 서비스 자체에 정의된 클래스를 하위 클래스화합니다. 서비스 제공자는 확장의 형태로 Java 플랫폼 구현에 설치할 수 있습니다. 즉, 일반적인 확장 디렉토리에 배치된 jar 파일입니다. 제공자는 애플리케이션의 클래스 경로에 추가하거나 다른 플랫폼별 수단을 통해 제공자를 사용할 수 있습니다. <sup>[4]</sup>

## Item1. Static Factory Method - 장점

1. 이름을 가진다.
2. 호출될 때마다 인스턴스 생성이 반드시 필요하지 않다.
3. 리턴 타입의 구현 객체를 줄 수 있다.
4. 입력 매개변수에 따라 상태 패턴 가능!
5. Lazy한 생성도 가능!

## Item1. Static Factory Method - 단점

1. 정적 팩터리 메서드만 있으면, 상속하는 하위 클래스는 만들 수 없다.
  - 다만, 컴포지션을 쓰게 강제할 수 있으니, SOLID 관점에서는 **개이득???**
2. 정적 팩터리 메서드는 부채다.
  - 바로 예상 가능한, 관습적 약속이 필요해!

## Item2. Builder

정적 팩토리 메소드 좋다.

근데,

매개변수가 많으면 어떻게 할래?

+ 그게 선택지도 많다면?

## Item2. Builder - 해결책. 점진적 생성자 패턴 ?

Telescoping Constructor Pattern

## Item2. Builder - 해결책. 점

### Telescoping Constructor Pattern

```
public class NutritionFacts {  
    private final int servingSize; // (ml, 1회 제공량)    필수  
    private final int servings; // (회, 총 n회 제공량)    필수  
    private final int calories; // (1회 제공량당)        선택  
    private final int fat; // (g/1회 제공량)            선택  
    private final int sodium; // (mg/1회 제공량)        선택  
    private final int carbohydrate; // (g/1회 제공량)    선택
```

```
    public NutritionFacts(int servingSize, int servings) {  
        this(servingSize, servings, 0);  
    }  
  
    public NutritionFacts(int servingSize, int servings, int calories) {  
        this(servingSize, servings, calories, 0);  
    }  
    public NutritionFacts(int servingSize, int servings, int calories,  
        int fat) {  
        this(servingSize, servings, calories, fat, 0);  
    }  
  
    public NutritionFacts(int servingSize, int servings, int calories,  
        int fat, int sodium) {  
        this(servingSize, servings, calories, fat, sodium, 0);  
    }  
  
    public NutritionFacts(int servingSize, int servings, int calories,  
        int fat, int sodium, int carbohydrate) {  
        this.servingSize = servingSize;  
        this.servings = servings;  
        this.calories = calories;  
        this.fat = fat;  
        this.sodium = sodium;  
        this.carbohydrate = carbohydrate;  
    }  
}
```

## Item2. Builder - 해결책. 점

### Telescoping Constructor Pattern

- 오버로딩 지옥...
- 코드가 너무 길어!
- 읽기 귀찮아!

```
public class NutritionFacts {  
    private final int servingSize; // (ml, 1회 제공량)      필수  
    private final int servings;    // (회, 총 n회 제공량)   필수  
    private final int calories;    // (1회 제공량당)        선택  
    private final int fat;         // (g/1회 제공량)        선택  
    private final int sodium;      // (mg/1회 제공량)       선택  
    private final int carbohydrate; // (g/1회 제공량)       선택
```

```
    public NutritionFacts(int servingSize, int servings) {  
        this(servingSize, servings, 0);  
    }  
  
    public NutritionFacts(int servingSize, int servings, int calories) {  
        this(servingSize, servings, calories, 0);  
    }  
    public NutritionFacts(int servingSize, int servings, int calories,  
                           int fat) {  
        this(servingSize, servings, calories, fat, 0);  
    }  
  
    public NutritionFacts(int servingSize, int servings, int calories,  
                           int fat, int sodium) {  
        this(servingSize, servings, calories, fat, sodium, 0);  
    }  
  
    public NutritionFacts(int servingSize, int servings, int calories,  
                           int fat, int sodium, int carbohydrate) {  
        this.servingSize = servingSize;  
        this.servings     = servings;  
        this.calories     = calories;  
        this.fat          = fat;  
        this.sodium       = sodium;  
        this.carbohydrate = carbohydrate;  
    }  
}
```



## Item2. Builder - 해결책. 점

### Telescoping Constructor Pattern

- 오버로딩 지옥...
- 코드가 너무 길어!
- 읽기 귀찮아!

이것은... 부채?

```
public class NutritionFacts {  
    private final int servingSize; // (ml, 1회 제공량)      필수  
    private final int servings;    // (회, 총 n회 제공량)   필수  
    private final int calories;    // (1회 제공량당)        선택  
    private final int fat;         // (g/1회 제공량)        선택  
    private final int sodium;      // (mg/1회 제공량)       선택  
    private final int carbohydrate; // (g/1회 제공량)       선택  
}
```

```
public NutritionFacts(int servingSize, int servings) {  
    this(servingSize, servings, 0);  
}  
  
public NutritionFacts(int servingSize, int servings, int calories) {  
    this(servingSize, servings, calories, 0);  
}  
  
public NutritionFacts(int servingSize, int servings, int calories,  
    int fat) {  
    this(servingSize, servings, calories, fat, 0);  
}  
  
public NutritionFacts(int servingSize, int servings, int calories,  
    int fat, int sodium) {  
    this(servingSize, servings, calories, fat, sodium, 0);  
}  
  
public NutritionFacts(int servingSize, int servings, int calories,  
    int fat, int sodium, int carbohydrate) {  
    this.servingSize = servingSize;  
    this.servings    = servings;  
    this.calories    = calories;  
    this.fat         = fat;  
    this.sodium      = sodium;  
    this.carbohydrate = carbohydrate;  
}
```

## Item2. Builder - 해결책. 자바빈즈 ?

Java Beans

# Item2. Builder - 해결책. 자바빈즈 ?

## Java Beans

```
public class NutritionFacts {  
    // 매개변수들은 (기본값이 있다면) 기본값으로 초기화된다.  
    private int servingSize = -1; // 필수; 기본값 없음  
    private int servings    = -1; // 필수; 기본값 없음  
    private int calories    = 0;  
    private int fat         = 0;  
    private int sodium      = 0;  
    private int carbohydrate = 0;  
  
    public NutritionFacts() { }  
    // setter 메서드들  
    public void setServingSize(int val) { servingSize = val; }  
    public void setServings(int val)    { servings = val; }  
    public void setCalories(int val)    { calories = val; }  
    public void setFat(int val)         { fat = val; }  
    public void setSodium(int val)      { sodium = val; }  
    public void setCarbohydrate(int val) { carbohydrate = val; }  
}
```

## Item2. Builder - 해결책. 자바빈즈 ?

### Java Beans

- 일관성 Consistency **無 !**
- 찾아볼 수 없는 불변성

```
public class NutritionFacts {  
    // 매개변수들은 (기본값이 있다면) 기본값으로 초기화된다.  
    private int servingSize = -1; // 필수; 기본값 없음  
    private int servings    = -1; // 필수; 기본값 없음  
    private int calories    = 0;  
    private int fat         = 0;  
    private int sodium      = 0;  
    private int carbohydrate = 0;  
  
    public NutritionFacts() { }  
    // 세터 메서드들  
    public void setServingSize(int val) { servingSize = val; }  
    public void setServings(int val)    { servings = val; }  
    public void setCalories(int val)    { calories = val; }  
    public void setFat(int val)         { fat = val; }  
    public void setSodium(int val)      { sodium = val; }  
    public void setCarbohydrate(int val){ carbohydrate = val; }  
}
```

## Item2. Builder - 해결책. 자바빈즈 ?

### Java Beans

- 일관성 Consistency **無 !**
- 찾아볼 수 없는 불변성

“당신의 코드엔 스파게티 소스가 어울리겠어요!”

```
public class NutritionFacts {  
    // 매개변수들은 (기본값이 있다면) 기본값으로 초기화된다.  
    private int servingSize = -1; // 필수; 기본값 없음  
    private int servings    = -1; // 필수; 기본값 없음  
    private int calories    = 0;  
    private int fat         = 0;  
    private int sodium      = 0;  
    private int carbohydrate = 0;  
  
    public NutritionFacts() { }  
    // setter 메서드들  
    public void setServingSize(int val) { servingSize = val; }  
    public void setServings(int val)    { servings = val; }  
    public void setCalories(int val)    { calories = val; }  
    public void setFat(int val)         { fat = val; }  
    public void setSodium(int val)      { sodium = val; }  
    public void setCarbohydrate(int val) { carbohydrate = val; }  
}
```

## Item2. Builder

- 믿음직한 불변함 Immutable
- 유려한 fluent API. Method Chaining 패턴

```
public class NutritionFacts {  
    private final int servingSize;  
    private final int servings;  
    private final int calories;  
    private final int fat;  
    private final int sodium;  
    private final int carbohydrate;  
}
```

```
public static class Builder {  
    // 필수 매개변수  
    private final int servingSize;  
    private final int servings;  
  
    // 선택 매개변수 - 기본값으로 초기화한다.  
    private int calories = 0;  
    private int fat = 0;  
    private int sodium = 0;  
    private int carbohydrate = 0;  
  
    public Builder(int servingSize, int servings) {  
        this.servingSize = servingSize;  
        this.servings = servings;  
    }  
  
    public Builder calories(int val)  
    { calories = val; return this; }  
    public Builder fat(int val)  
    { fat = val; return this; }  
    public Builder sodium(int val)  
    { sodium = val; return this; }  
    public Builder carbohydrate(int val)  
    { carbohydrate = val; return this; }  
  
    public NutritionFacts build() {  
        return new NutritionFacts(this);  
    }  
}
```

```
private NutritionFacts(Builder builder) {  
    servingSize = builder.servingSize;  
    servings = builder.servings;  
    calories = builder.calories;  
    fat = builder.fat;  
    sodium = builder.sodium;  
    carbohydrate = builder.carbohydrate;  
}
```

## Item2. Builder

- 하위 계층 반환 패턴,
- 공변 반환 타이핑 **Convariant Return Typing**

```
public abstract class Pizza {
    public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }
    final Set<Topping> toppings;

    abstract static class Builder<T extends Builder<T>> {
        EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);
        public T addTopping(Topping topping) {
            toppings.add(Objects.requireNonNull(topping));
            return self();
        }

        abstract Pizza build();

        // 하위 클래스는 이 메서드를 재정의(overriding)하여
        // "this"를 반환하도록 해야 한다.
        protected abstract T self();
    }

    Pizza(Builder<?> builder) {
        toppings = builder.toppings.clone(); // 아이템 50 참조
    }
}
```

```
public class Calzone extends Pizza {
    private final boolean sauceInside;

    public static class Builder extends Pizza.Builder<Builder> {
        private boolean sauceInside = false; // 기본값

        public Builder sauceInside() {
            sauceInside = true;
            return this;
        }

        @Override public Calzone build() {
            return new Calzone(this);
        }

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

    private Calzone(Builder builder) {
        super(builder);
        sauceInside = builder.sauceInside;
    }
}
```

## Item2. Builder

- 하위 계층 반환 패턴,
- 공변 반환 타이핑 **Convariant Return Typing**

```
public abstract class Pizza {
    public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }
    final Set<Topping> toppings;

    abstract static class Builder<T extends Builder<T>> {
        EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);
        public T addTopping(Topping topping) {
            toppings.add(Objects.requireNonNull(topping));
            return self();
        }

        abstract Pizza build();

        // 하위 클래스는 이 메서드를 재정의(overriding)하여
        // "this"를 반환하도록 해야 한다.
        protected abstract T self();
    }

    Pizza(Builder<?> builder) {
        toppings = builder.toppings.clone(); // 아이템 50 참조
    }
}
```

```
public class Calzone extends Pizza {
    private final boolean sauceInside;

    public static class Builder extends Pizza.Builder<Builder> {
        private boolean sauceInside = false; // 기본값

        public Builder sauceInside() {
            sauceInside = true;
            return this;
        }

        @Override public Calzone build() {
            return new Calzone(this);
        }
    }
}
```

```
public class NyPizza extends Pizza {
    public enum Size { SMALL, MEDIUM, LARGE }
    private final Size size;

    public static class Builder extends Pizza.Builder<Builder> {
        private final Size size;

        public Builder(Size size) {
            this.size = Objects.requireNonNull(size);
        }

        @Override public NyPizza build() {
            return new NyPizza(this);
        }

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

    private NyPizza(Builder builder) {
        super(builder);
        size = builder.size;
    }
}
```



## Item2. Builder

- 하위 계층 반환 패턴,
- 공변 반환 타이핑 Covariant Return Typing

```
public abstract class Pizza {  
    public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }  
    final Set<Topping> toppings;
```

```
    abstract static class Builder {  
        EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);  
        public T addTopping(Topping topping) {  
            toppings.add(topping);  
            return self();  
        }  
    }
```

```
    abstract Pizza build();
```

```
    // 하위 클래스는 이 메서드를 재정의(overriding)하여  
    // "this"를 반환하도록 해야 한다.
```

```
    protected abstract T self();
```

```
}
```

```
Pizza(Builder<?> builder) {  
    toppings = builder.toppings.clone(); // 아이템 50 참조  
}
```

```
}
```

```
public class Calzone extends Pizza {  
    private final boolean sauceInside;  
  
    public static class Builder extends Pizza.Builder<Builder> {  
        private boolean sauceInside = false; // 기본값  
  
        public Builder sauceInside() {  
            sauceInside = true;  
            return this;  
        }  
  
        @Override public Calzone build() {  
            return new Calzone(this);  
        }  
    }  
}
```

```
public class NyPizza extends Pizza {  
    public enum Size { SMALL, MEDIUM, LARGE }
```

```
NyPizza pizza = new NyPizza.Builder(SMALL)  
    .addTopping(SAUSAGE).addTopping(ONION).build();  
Calzone calzone = new Calzone.Builder()  
    .addTopping(HAM).sauceInside().build();
```

```
    @Override public NyPizza build() {  
        return new NyPizza(this);  
    }
```

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

```
    private NyPizza(Builder builder) {  
        super(builder);  
        size = builder.size;  
    }  
}
```

## Item3. 싱글톤 써먹기

Singleton?

- 프로세스에서 하나만 있는 객체
- 생성자 감추고, `public static` 멤버로만 접근 가능

## Item3. 싱글톤 써먹기 - 방법1.

public static final 필드

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

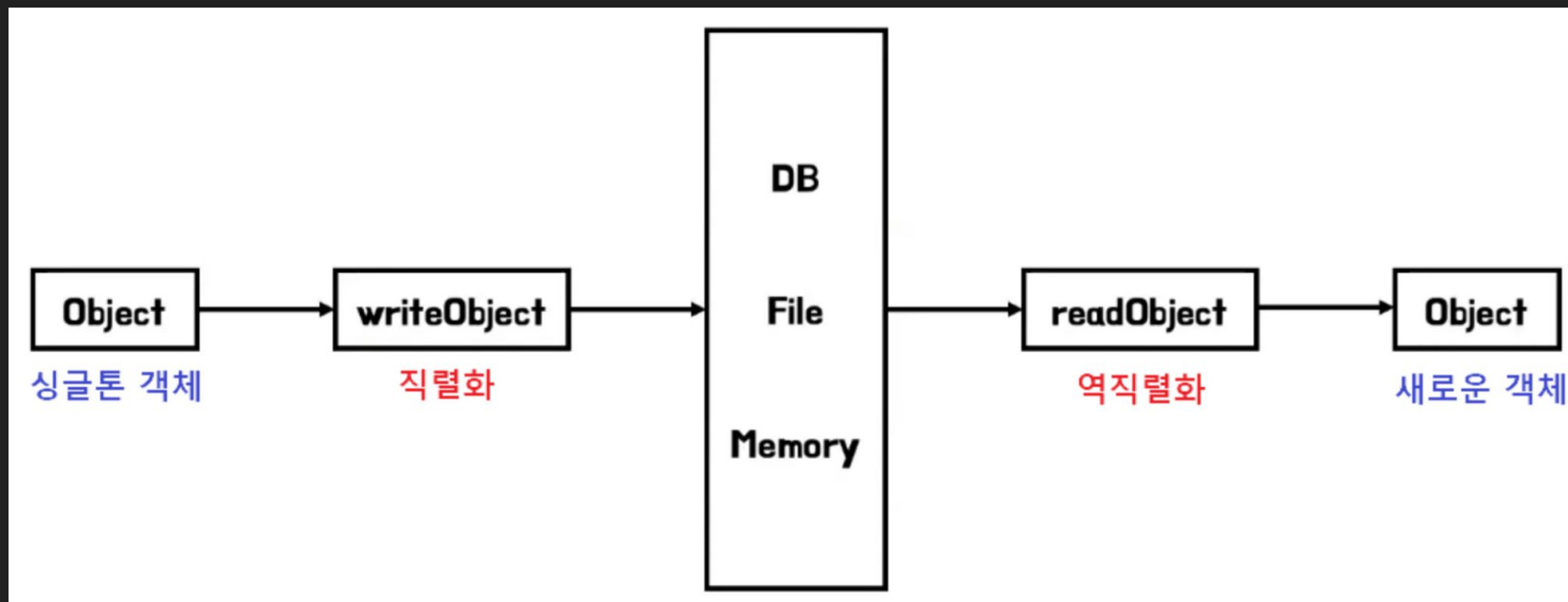
## Item3. 싱글톤 써먹기 - 방법2.

정적 팩터리

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

=> 역직렬화때 새로 만들어짐

<https://techblog.woowahan.com/2550/>



=> 역직렬화때 새로 만들어짐

<https://techblog.woowahan.com/2550/>

```
// 싱글톤 + 직렬화
class Singleton implements Serializable {

    private Singleton() {}

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

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

readObject

역직렬화

Object

새로운 객체

=> 역직렬화때 새로 만들어짐

<https://techblog.woowahan.com/2550/>

```
public static void main(String[] args) throws IOException, ClassNotFoundException { 0개의 사용위치 신규 *
    Singleton singleton1 = Singleton.getInstance();

    String fileName = "singleton.obj";

    // 직렬화
    ObjectOutputStream out = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(fileName)));
    out.writeObject(singleton1);
    out.close();

    // 역직렬화
    ObjectInputStream in = new ObjectInputStream(new BufferedInputStream(new FileInputStream(fileName)));
    Singleton singleton2 = (Singleton) in.readObject();
    in.close();

    System.out.println("singleton1 == singleton2 : " + (singleton1 == singleton2));
    System.out.println(singleton1);
    System.out.println(singleton2);
}
```

```
public static void main(String[] args) throws IOException, ClassNotFoundException { 0개의 사용위치 신규 *
    Singleton singleton1 = Singleton.getInstance();
```

```
    String fileName = "singleton.obj";
```

```
    // 직렬화
```

```
    ObjectOutputStream out = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(fileName)));
    out.writeObject(singleton1);
    out.close();
```

```
    // 역직렬화
```

```
    ObjectInputStream in = new ObjectInputStream(new BufferedInputStream(new FileInputStream(fileName)));
    Singleton singleton2 = (Singleton) in.readObject();
    in.close();
```

```
    System.out.println("singleton1 == singleton2 : " + (singleton1 == singleton2));
    System.out.println(singleton1);
    System.out.println(singleton2);
```

```
}
```



```
public static void main(String[] args) throws IOException, ClassNotFoundException { 0개의 사용위치 신규 *
    Singleton singleton1 = Singleton.getInstance();

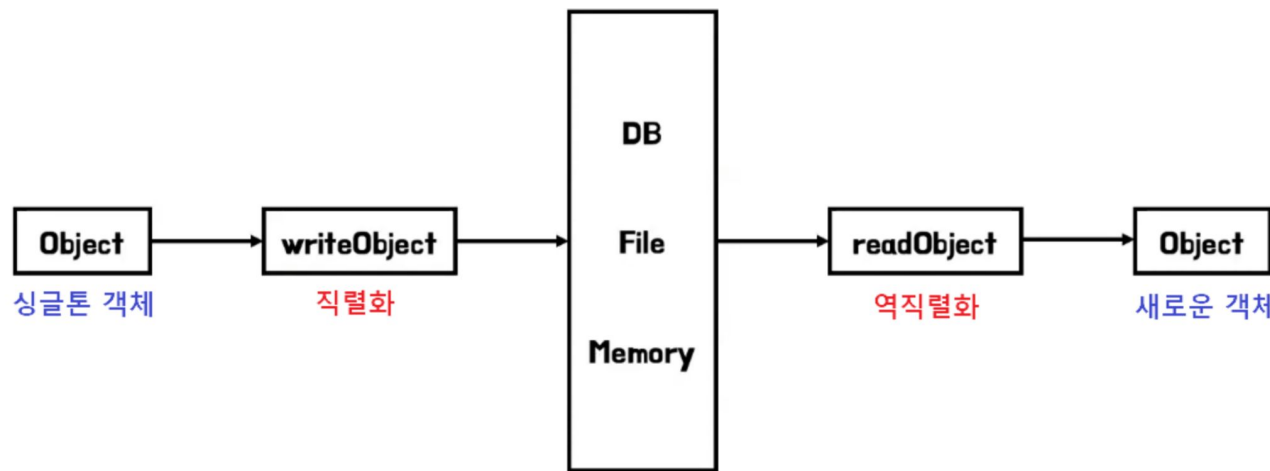
    String fileName = "singleton.obj";

    // 직렬화
    ObjectOutputStream out = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(fileName)));
    out.writeObject(singleton1);
    out.close();

    // 역직렬화
    ObjectInputStream in = new ObjectInputStream(new BufferedInputStream(new FileInputStream(fileName)));
    Singleton singleton2 = (Singleton) in.readObject();
    in.close();

    System.out.println("singleton1 == singleton2 : " + (singleton1 == singleton2));
    System.out.println(singleton1);
    System.out.println(singleton2);
}
```

pub



{ 0개의 사용위치 신규 \*

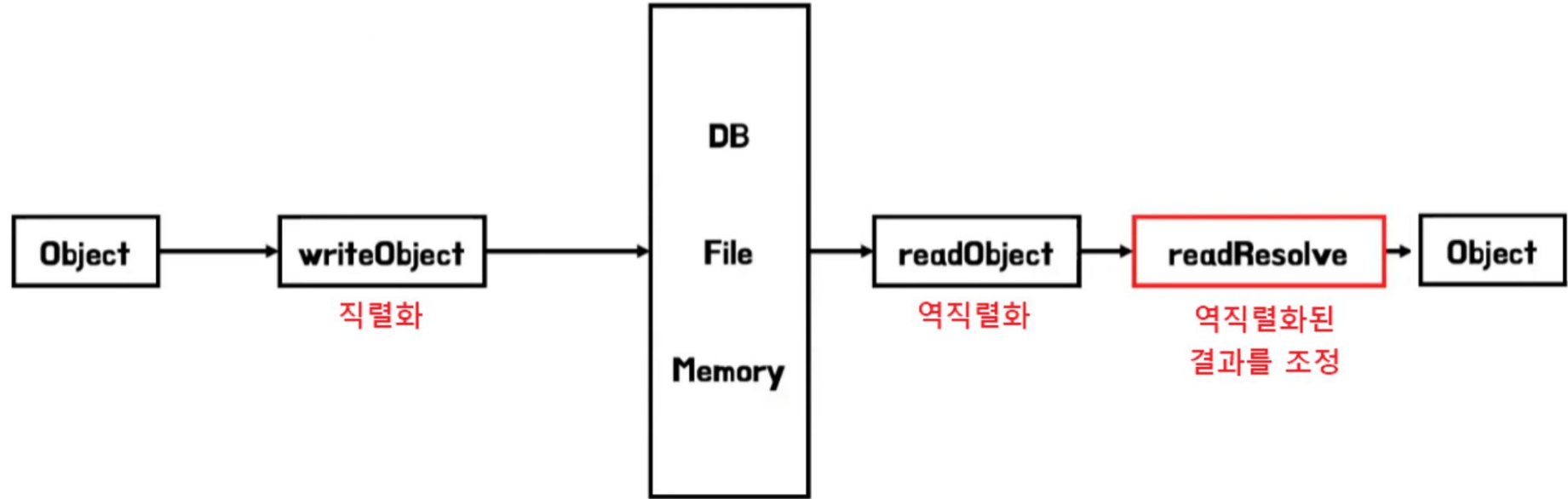
fileOutputStream(fileName));

// 역직렬화

```
ObjectInputStream in = new ObjectInputStream(new BufferedInputStream(new FileInputStream(fileName)));  
Singleton singleton2 = (Singleton) in.readObject();  
in.close();
```

```
System.out.println("singleton1 == singleton2 : " + (singleton1 == singleton2));  
System.out.println(singleton1);  
System.out.println(singleton2);
```

}



=> 방지가 필요

```

class Singleton implements Serializable { 7개 사용 위치 신규 *

    private Singleton() {} 1개 사용 위치 신규 *

    private static class SettingsHolder { 2개 사용 위치 신규 *
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() { 1개 사용 위치 신규 *
        return SettingsHolder.INSTANCE;
    }

    private Object readResolve() { 0개의 사용 위치 신규 *
        return SettingsHolder.INSTANCE;
    }
}

```

=> 방지가 필요



## Item3. 싱글톤 써먹기 - 접근 제한자는 리플렉션에 뚫림

```
public static void main(String[] args) 0개의 사용위치 신규 *  
    throws NoSuchMethodException, InstantiationException, IllegalAccessException, InvocationTargetException {  
    /* Reflection API */  
  
    // 1. Singleton의 생성자 가져오기  
    Constructor<Singleton> consructor = Singleton.class.getDeclaredConstructor();  
  
    // 2. 생성자 private 접근을 허용하기  
    consructor.setAccessible(true);  
  
    // 3. 생성자로 객체 생성  
    Singleton singleton1 = consructor.newInstance();  
    Singleton singleton2 = consructor.newInstance();  
  
    System.out.println("singleton1 == singleton2 : " + (singleton1 == singleton2));  
    System.out.println(singleton1);  
    System.out.println(singleton2);  
}
```

=> 리플렉션에 뚫림

## Item3. 싱글톤 써먹기 - 방법3.

enum

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

# enum?

그런데 어떤 규칙들은 너무 강력한 나머지  
그 규칙을 지키지 않으면 그저 혼쭐이 나버리고



규칙에 대해 의문을 제기하는 것 자체도 어려울 때가 있다

# enum?

## Enumerated type

🌐 15 languages ▼

Article Talk

Read Edit View history Tools ▼

From Wikipedia, the free encyclopedia

In [computer programming](#), an **enumerated type** (also called **enumeration**, **enum**, or **factor** in the [R programming language](#), and a [categorical variable](#) in statistics) is a [data type](#) consisting of a set of named [values](#) called *elements*, *members*, *enumerals*, or *enumerators* of the type. The enumerator names are usually [identifiers](#) that behave as [constants](#) in the language. An enumerated type can be seen as a degenerate [tagged union](#) of [unit type](#). A [variable](#) that has been [declared](#) as having an enumerated type can be assigned any of the enumerators as a value. In other words, an enumerated type has values that are different from each other, and that can be compared and assigned, but are not specified by the programmer as having any particular concrete representation in the computer's memory; compilers and interpreters can represent them arbitrarily.



규칙에 대해 의문을 제기하는 것 자체도 어려울 때가 있다

@waterglasstoon



# enum?

## Enumerated type

🌐 15 languages ▼

Article Talk

Read Edit View history Tools ▼

From Wikipedia, the free encyclopedia

컴퓨터 프로그래밍에서 열거형 (R 프로그래밍 언어에서는 열거형, 열거형 또는 요인 이라고도 하며 통계에서는 범주형 변수 라고도 함)은 요소의 집합, 멤버, 열거형 또는 열거자로 구성된 데이터 유형입니다. 열거자 이름은 일반적으로 언어에서 상수로 작동하는 식별자입니다. 열거형은 단위 유형의 퇴화 태그 합집합으로 볼 수 있습니다. 열거형으로 선언된 변수에는 열거자 중 하나를 값으로 할당할 수 있습니다. 즉, 열거형은 서로 다른 값을 가지며 비교 및 할당할 수 있지만 프로그래머가 컴퓨터 메모리에 특정한 구체적인 표현이 있다고 지정하지 않습니다. 컴파일러와 인터프리터는 이를 임의로 표현할 수 있습니다.



규칙에 대해 의문을 제기하는 것 자체도 어려울 때가 있다

# enum in Java

Java [\[edit\]](#)

The J2SE version 5.0 of the [Java programming language](#) added enumerated types whose declaration syntax is similar to that of C:

```
enum Cardsuit { CLUBS, DIAMONDS, SPADES, HEARTS };  
...  
Cardsuit trump;
```



The Java type system, however, treats enumerations as a type separate from integers, and intermixing of enum and integer values is not allowed. In fact, an enum type in Java is actually a special compiler-generated [class](#) rather than an arithmetic type, and enum values behave as global pre-generated instances of that class. Enum types can have instance methods and a constructor (the arguments of which can be specified separately for each enum value). All enum types implicitly extend the [Enum](#) [abstract class](#). An enum type cannot be instantiated directly.<sup>[12]</sup>

Internally, each enum value contains an integer, corresponding to the order in which they are declared in the source code, starting from 0. The programmer cannot set a custom integer for an enum value directly, but one can define [overloaded constructors](#) that can then assign arbitrary values to self-defined members of the enum class. Defining getters allows then access to those self-defined members. The internal integer can be obtained from an enum value using the [ordinal\(\)](#) [method](#), and the list of enum values of an enumeration type can be obtained in order using the [values\(\)](#) [method](#). It is generally discouraged for programmers to convert enums to integers and vice versa.<sup>[13]</sup> Enumerated types are [Comparable](#), using the internal integer; as a result, they can be sorted.

The Java standard library provides utility classes to use with enumerations. The [EnumSet](#) [class](#) implements a [Set](#) of enum values; it is implemented as a [bit array](#), which makes it very compact and as efficient as explicit bit manipulation, but safer. The [EnumMap](#) [class](#) implements a [Map](#) of enum values to object. It is implemented as an array, with the integer value of the enum value serving as the index.

# enum in Java

Java [edit]

The J2SE version 5.0 of the [Java programming language](#) added enumerated types whose declaration syntax is similar to that of C:

```
enum Cardsuit { CLUBS, DIAMONDS, SPADES, HEARTS };  
...  
Cardsuit trump;
```



그러나 Java 유형 시스템은 열거형을 정수와 별개의 유형으로 취급하며 열거형과 정수 값을 섞는 것은 허용되지 않습니다. 사실 Java의 열거형은 **산술 유형이 아니라 컴파일러에서 생성한 특수 클래스**이며 열거형 값은 해당 클래스의 전역 사전 생성 인스턴스처럼 작동합니다. 열거형 유형은 인스턴스 메서드와 생성자(각 열거형 값에 대해 별도로 지정할 수 있는 인수)를 가질 수 있습니다. 모든 열거형 유형은 [Enum](#) 추상 클래스를 암묵적으로 확장합니다. **열거형 유형은 직접 인스턴스화할 수 없습니다.** <sup>[12]</sup>

내부적으로 각 열거형 값은 소스 코드에서 선언된 순서에 따라 0부터 시작하는 정수를 포함합니다. 프로그래머는 열거형 값에 대한 사용자 정의 정수를 직접 설정할 수 없지만, 열거형 클래스의 자체 정의 멤버에 임의의 값을 할당할 수 있는 **오버로드된 생성자**를 정의할 수 있습니다. 게터를 정의하면 해당 자체 정의 멤버에 액세스할 수 있습니다. 내부 정수는 메서드를 사용하여 열거형 값에서 가져올 수 있으며 [ordinal\(\)](#), 열거형 유형의 열거형 값 목록은 메서드를 사용하여 순서대로 가져올 수 있습니다 [values\(\)](#). 일반적으로 프로그래머가 열거형을 정수로 변환하거나 그 반대로 변환하는 것은 권장되지 않습니다. <sup>[13]</sup> 열거형은 [Comparable](#) 내부 정수를 사용하여 implements. 결과적으로 정렬할 수 있습니다.

Java 표준 라이브러리는 열거형과 함께 사용할 유틸리티 클래스를 제공합니다. 이 [EnumSet](#) 클래스는 Set 열거형 값의 a를 구현합니다. **비트 배열**로 구현되어 매우 간결하고 명시적 비트 조작만큼 효율적이지만 더 안전합니다. 이 [EnumMap](#) 클래스는 객체에 대한 열거형 값의 a를 구현합니다 [Map](#). 배열로 구현되며 열거형 값의 정수 값이 인덱스 역할을 합니다.

E.O.D.