

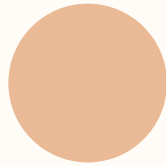
이펙티브 자바

Item1

생성자 대신 정적 팩터리
메서드를 고려하라

유현우

객체를 생성하는 방법



public 생성자

```
1 | public 클래스 이름 () {  
2 |  
3 | }
```

정적 팩터리 메서드 (Boolean에서 사용하는 정적 팩터리 메서드)

- 디자인 패턴에서의 팩터리 메서드와 다르다.

```
1 | public static Boolean valueOf(boolean b) {  
2 |     return (b ? TRUE : FALSE);  
3 | }
```

생성자 말고 정적 팩터리 메서드는 왜 사용할까?

1. 이름을 가질 수 있다.



생성자, 정적 팩터리 메서드 중에서 “값이 소수인 BigInteger를 반환한다.”라는 뜻을 명확하게 표현하는지 생각해 보면 이름의 장점을 얻을 수 있다.


```
1 | public BigInteger(int bitLength, int certainty, Random rnd)
2 | public static BigInteger probablePrime(int bitLength, Random rnd)
```

생성자의 문제점

```
public class User {  
    private String name;  
    private String address;  
    private int money;  
  
    public User(String name, int money) { //1 (3)번이 추가되면 같이 컴파일 에러  
        this.name = name;  
        this.money = money;  
    }  
  
    public User(int money, String address) { //2  
        this.address = address;  
        this.money = money;  
    }  
  
    public User(String address, int money) { //3 메서드 시그니처 중복으로 컴파일 에러  
        this.address = address;  
        this.money = money;  
    }  
}
```

매개변수들의 순서를 다르게 한 생성자를 새로 추가하는 식으로 사용하다 보면 API를 사용하는 개발자 입장에서 어떤 역할을 하는지 정확하게 기억하기 힘들다.

정적 팩터리 메서드는 오버로딩의 한계가 없다.



```
public class User {  
    private String name;  
    private String address;  
    private int money;  
  
    private User() {  
    }  
  
    public static User withNameAndMoney(String name, int money) {  
        User user = new User();  
        user.name = name;  
        user.money = money;  
        return user;  
    }  
  
    public static User withAddressAndMoney(String address, int money) {  
        User user = new User();  
        user.address = address;  
        user.money = money;  
        return user;  
    }  
}
```

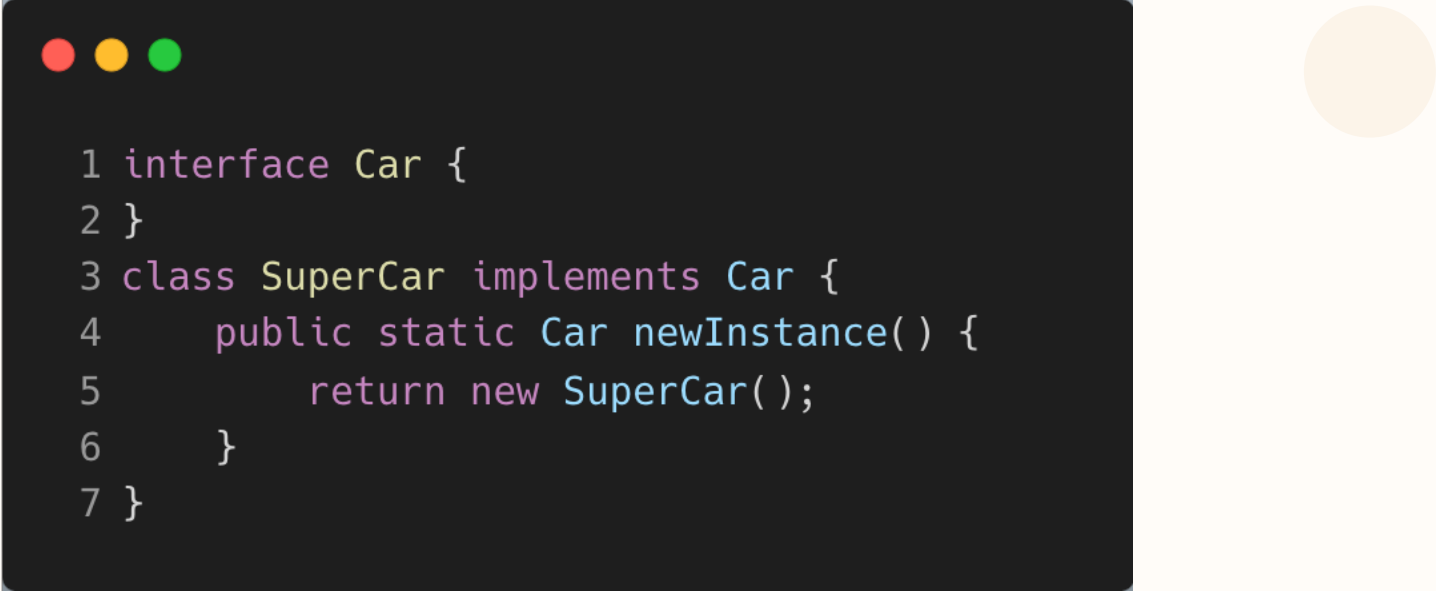
한 클래스에 시그니처가 같은 생성자가 여러 개 필요하면 생성자를 정적 팩터리 메서드로 바꾸고 각각의 차이를 잘 드러내는 이름을 지어주자

2. 호출될 때마다 인스턴스를 새로 생성하지 않을 수 있다.

```
1 public class MyContext {
2     private static Map<String, Object> MY_CONTEXT_CACHE = new HashMap<>();
3     static {
4         MY_CONTEXT_CACHE.put("myService", new MyService());
5         MY_CONTEXT_CACHE.put("myRepository", new MyRepository());
6     }
7     public static Object getContext(String name) {
8         return MY_CONTEXT_CACHE.get(name);
9     }
10
11     public static void main(String[] args) {
12         MyService myService1 = (MyService) MyContext.getContext("myService");
13         MyService myService2 = (MyService) MyContext.getContext("myService");
14         System.out.println(myService1.equals(myService2)); //true
15     }
16 }
17 class MyService {
18     public MyService() {
19         System.out.println("hihi"); // MY_CONTEXT_CACHE가 초기화 될 때 한번만 호출
20     }
21 }
22 class MyRepository {}
```

생성 비용이 큰 객체를 자주 요청되는 상황이라면 캐싱하여 인스턴스를 재사용하므로 성능을 올려 줄 수 있다.

3. 반환 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다.



```
1 interface Car {  
2 }  
3 class SuperCar implements Car {  
4     public static Car newInstance() {  
5         return new SuperCar();  
6     }  
7 }
```

인터페이스를 통해서 구현 클래스를 공개하지 않고도 그 객체를 반환할 수 있어 API를 작게 유지할 수 있다.

명시한 인터페이스대로 동작하는 객체를 얻을 것임을 알기 때문에 별도 문서를 찾아가며 구현클래스를 확인하지 않아도 된다.

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

인터페이스

```
1 public interface MessageService {  
2     /**  
3     *  
4     * @return 언어에 맞는 메시지를 반환한다.  
5     */  
6     String getMessage();  
7 }
```

주석은 생략

구현 클래스

```
1 class KoreanMessageService implements  
    MessageService{  
2     @Override  
3     public String getMessage() {  
4         return "안녕";  
5     }  
6 }  
7  
8 class EnglishMessageService implements  
    MessageService{  
9     @Override  
10    public String getMessage() {  
11        return "hi";  
12    }  
13 }
```

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

Factory class



```
1 class MessageFactory {  
2     public static MessageService from(String location) {  
3         if ("ko".equals(location)) {  
4             return new KoreanMessageService();  
5         }  
6         return new EnglishMessageService();  
7     }  
8 }
```

매개변수에 따라 다른 클래스 객체를 반환할 수 있다.

EnumSet 클래스

Creates an empty enum set with the specified element type.

Params: `elementType` - the class object of the element type for this enum set

Returns: An empty enum set of the specified type.

Throws: `NullPointerException` - if `elementType` is null

```
public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) {
    Enum<?>[] universe = getUniverse(elementType);
    if (universe == null)
        throw new ClassCastException(elementType + " not an enum");

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

Creates an enum set containing all of the elements in the specified element type.

Params: `elementType` - the class object of the element type for this enum set

Returns: An enum set containing all the elements in the specified type.

Throws: `NullPointerException` - if `elementType` is null

```
public static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType) {
    EnumSet<E> result = noneOf(elementType);
    result.addAll();
    return result;
}
```

EnumSet 클래스는 생성자 없이 public static 메서드로 `allOf()`, `of()` 등을 제공한다.

리턴하는 객체의 타입이 **enum 타입의 개수(64개)**에 따라서 **RegularEnumSet** 또는 **JumboEnumSet**으로 달라진다.

EnumSet 클래스

Creates an empty enum set with the specified element type.

Params: elementType - the class object of the element type for this enum set

Re

Th

pub

HashMap 은 hash 값을 계산하여 table 을 제어하는 형태로 데이터를 관리합니다. 이에 비해 EnumMap 은 열거형 상수가 정의된 순서를 가지고, 배열의 index 만 가져오면 되기 때문에 상대적으로 대부분의 경우에 성능이 더 좋다고 할 수 있습니다.

EnumMap 은 HashMap 과 같이 thread safe 하지 않습니다. 따라서 멀티쓰레드 환경에서 여러 스레드가 접근할 수 있다면, 동기화를 위한 처리를 해주어야 합니다. 이를 위해 SynchronizedMap 을 사용할 수 있습니다.

Creates an enum set containing all of the elements in the specified element type.

Params: elementType - the class object of the element type for this enum set

Returns: An enum set containing all the elements in the specified type.

Throws: **NullPointerException** - if elementType is null

```
public static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType) {  
    EnumSet<E> result = noneOf(elementType);  
    result.addAll();  
    return result;  
}
```

자바 8부터는 public static 메서드를 인터페이스 추가 가능하다.

interface

```
1 public interface MessageService {
2     /**
3      * @return 언어에 맞는 메시지를 반환한다.
4      */
5     public static String getMessage(String location) {
6         if ("ko".equals(location)) {
7             return "안녕";
8         }
9         return "hi";
10    };
11 }
```

java.util.Collections 같은 유틸성 클래스들은 필요 없어진다.

java 8

인터페이스는 public 정적 멤버만 가능하다.

java 9

private 정적 메서드는 가능하지만 정적 멤버 클래스는 public만 가능하다.

5. 정적 팩터리 메서드를 작성하는 시점에는 변환할 객체의 클래스가 존재하지 않아도 된다.

```
1 public class Car {  
2  
3     // 상황에 따라 유연하게 반환될 인스턴스가 결정됨  
4     public static Car getCar() {  
5         Car car = new Car();  
6  
7         // 특정 텍스트 파일에서 Car 구현체의 FQCN(Full Qualified Class Name)을 가져온다  
8         // FQCN에 해당하는 인스턴스 생성  
9         // car가 FQCN에 해당하는 인스턴스를 가리키도록 한다  
10  
11         return car;  
12     }  
13 }
```

자바 6부터는 `java.util.ServiceLoader`라는 범용 서비스 제공자 프레임워크를 통해서 jar 파일을 바꿔끼는 것만으로 내가 원하는 적절한 구현체를 사용하도록 바꿀 수 있다.

서비스 제공자 프레임워크

서비스 제공자 프레임워크 또는 서비스 제공자 인터페이스 패턴이라 불리는 것은 개념적인 이야기다.

- 다양한 구현 방법과 변형이 존재할 수 있다.
- 목적이 중요한 것 이지 구현 형태가 중요한게 아니다.
- 목적은 확장 가능한 애플리케이션 을 만드는 방법을 제공하는 것이다.
 - 확장이 가능하다는 건 코드는 그대로 유지되면서 외적인 요인을 변경했을 때 애플리케이션의 동작을 다르게 동작할 수 있게 만들 수 있는 것 을 말한다.

대표적인 서비스 제공자 프레임워크 중 하나인 JDBC

JDBC는 자바 6 전에 등장한 개념이라 ServiceLoader를 사용하지 않는다.

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

서비스 인터페이스(service interface): 구현체의 동작을 정의한다.

제공자 등록 API(provider registration API): 제공자가 구현체를 등록할 때 사용한다.

서비스 접근 API(service access API): 클라이언트가 서비스의 인스턴스를 얻을 때 사용한다.

서비스 제공자 인터페이스(service provider interface): 서비스 인터페이스의 인스턴스를 생성하는 팩터리 객체를 설명해준다.

서비스 제공자 프레임워크 - 스프링 느낌

서비스 제공자 인터페이스 (SPI)와 서비스 제공자 (서비스 구현체)

```
1 public interface ServiceInterface {
2 }
3
4 class Service1 implements ServiceInterface {
5 }
6
7 class Service2 implements ServiceInterface {
8     public static String ofLocation(String location) {
9         if ("ko".equals(location)) {
10             return "안녕하세요";
11         }
12         return "hello";
13     }
14 }
```

서비스 제공자 등록 API (서비스 인터페이스의 구현체를 등록하는 방법)

```
1 public class ProviderRegistrationConfig {
2     public List<ServiceInterface> getServiceInstance() {
3         return List.of(
4             new Service1(),
5             new Service2()
6         );
7     }
8 }
```

서비스 접근 API (서비스의 클라이언트가 서비스 인터페이스의 인스턴스를 가져올 때 사용하는 API)

```
1 public static void main(String[] args) throws Exception {
2     MyApplicationContext context = new MyApplicationContext(ProviderRegistration.class);
3     Service2 service2 = (Service2) context.getBean(Service2.class);
4     System.out.println(service2.ofLocation("ko"));
5 }
```

정적 팩터리 메서드의 단점은?

상속을 하려면 public이나 protected 생성자가 필요하니 정적 팩터리 메서드만 제공하면 하위 클래스를 만들 수 없다.



```
83  
84     public class Collections {  
85         // Suppresses default constructor, ensuring non-instantiability.  
86         private Collections() {  
87         }  
88     }
```

java.util.Collections는 상속할 수 없다.

우회해서 사용하는 방법

collections에게 위임하면서 확장할 수 있다.

```
no usages
public class MyCollections {
    ⚡ no usages
    Collections collections;
}
```

정적 팩토리를 제공하면서 생성자를 제공

```
@NotNull
@SafeVarargs
/varargs/
public static <T> List<T> asList( @NotNull T... a) {
    return new ArrayList<>(a);
}
```

정적 팩터리 메서드는 프로그래머가 찾기 어렵다.

생성자처럼 API 설명에 명확히 드러나지 않으니 사용자는 정적 팩터리 메서드 방식 클래스를 인스턴스화할 방법을 알아내야 한다.

- 문서 정리에 좀 더 신경을 써서 사용자가 한번에 찾아볼 수 있도록 작성해야한다.
- 헛갈리지 않도록 일종의 명명규칙을 사용하도록 하자.

컨벤션

- **from**: 매개 변수 하나를 받아서 해당 타입의 인스턴스를 반환하는 형변환 메서드

```
Date d = Date.from(instant);
```

- **of**: 여러 매개변수를 받아 적합한 타입의 인스턴스를 반환하는 집계 메서드

```
Set faceCards = EnumSet.of(JACK, QUEEN, KING);
```

- **valueOf**: from 과 of의 더 자세한 버전

```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

- **instance** 혹은 **getInstance**: (매개변수를 받는다면) 매개변수로 명시한 인스턴스를 반환하지만, 같은 인스턴스임을 보장하지는 않는다.

```
StackWalker luke = StackWalker.getInstance(options);
```

- **create** 혹은 **newInstance**: instance 혹은 getInstance 같지만, 매번 새로운 인스턴스를 생성해 반환함을 보장한다.

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

- **getType**: getInstance 와 같으나, 생성할 클래스가 아닌 다른 클래스에 팩터리 메서드를 정의할 때 쓴다. "Type" 은 팩터리 메서드가 반환할 객체의 타입이다.

```
FileStore fs = Files.getFileStore(path);
```

- **newType**: newInstance 와 같으나, 생성할 클래스가 아닌 다른 클래스에 팩터리 메서드를 정의할 때 쓴다. "Type" 은 팩터리 메서드가 반환할 객체의 타입이다.

```
BufferedReader br = Files.newBufferedReader(path);
```

- **type**: getType과 newType의 간결한 버전

```
List litany = Collections.list(legacyLitany)
```

Reference

ServiceLoader: <https://alkhwa-113.tistory.com/entry/ServiceLoader>

EnumMap 적용하기: <https://uhanuu.tistory.com/entry/EnumMap-%EC%A0%81%EC%9A%A9%ED%95%98%EA%B8%B0>

이펙티브 자바 Item1: <https://pro-dev.tistory.com/96>

이펙티브 자바 Item1: https://github.com/woowacourse-study/2022-effective-java/blob/main/02%EC%9E%A5/%EC%95%84%EC%9D%B4%ED%85%9C_01/%EC%83%9D%EC%84%B1%EC%9E%90_%EB%8C%80%EC%8B%A0_%EC%A0%95%EC%A0%81_%ED%8C%A9%ED%84%B0%EB%A6%AC_%EB%A9%94%EC%84%9C%EB%93%9C%EB%A5%BC_%EA%B3%A0%EB%A0%A4%ED%95%98%EB%9D%BC.md

Thank you

궁금한 점을 물어보세요