**CHAPTER2**

**ITEM1 생성자 대신 정적 팩터리 메서드를 고려하라**

**팩토리 메서드 (하위 클래스에서 인스턴스 작성하기)**

[](https://user-images.githubusercontent.com/7076334/46743237-95f05c00-cce3-11e8-91e6-f7687d4f5f85.png)

-서브 클래스로 생성을 위임하기 때문에 효율적인 코드 제어를 할 수 있고 의존성을 제거한다.

**정적 팩터리 장점**

**1) 이름을 가질 수 있다.**

ex) BigInteger(int, int, Random) vs BigInteger.probablePrime

-valueOf, getInstance, newInstance, getType, newType 등

**시그니처가 같은 생성자**

public static Blog newInstanceWithId(String id);

public static Blog newInstanceWithName(String name);

**2) 호출될 때마다 인스턴스를 새로 생성하지는 않아도 된다.**

2-1) 미리 static으로 선언해 놓는 경우

ex) Boolean.valueOf(Boolean)

public static final Boolean *TRUE* = new Boolean(true);  
public static final Boolean *FALSE* = new Boolean(false);

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

2-2) 미리 특정 값까지 캐싱해 놓는 경우

ex) Integer.valueOf(int i), Long.valueOf(long l) -127 ~ 128

public static void main(String[] args) {  
 // -127 ~ 128 까지 autoboxing(unboxing) 해서 캐싱 (기본값 변경 가능)  
 // -XX:AutoBoxCacheMax= or -Djava.lang.Integer.IntegerCache.high=2000  
 Integer a = Integer.*valueOf*(10);  
 Integer b = Integer.*valueOf*(10);  
  
 if (a == b) {  
 System.*out*.println("a와 b가 같음");  
 }  
  
 Integer d = Integer.*valueOf*(200);  
 Integer e = Integer.*valueOf*(200);  
  
 if (d == e) {  
 System.*out*.println("d와 e가 같음");  
 }  
}

**결과 “a와 b가 같음”**

private static class IntegerCache {  
 static final int *low* = -128;  
 static final int *high*;  
 static final Integer *cache*[];  
  
 static {  
 // high value may be configured by property  
 int h = 127;

...

...

}

public static Integer valueOf(int i) {  
 if (i >= IntegerCache.*low* && i <= IntegerCache.*high*)  
 return IntegerCache.*cache*[i + (-IntegerCache.*low*)];  
 return new Integer(i);  
}

🡺플라이웨이트 패턴 (메모리 캐시)

**3) 반환 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다. (유연성)**

**동반 클래스 (scala)**

class Dog(name: String) {  
 def bark = println("bark! bark!")  
 def getName = name  
}  
  
object Dog {  
 def apply(name: String) = new Dog(name)  
}

object Test extends App{  
 val dog = Dog("dog1")  
}

-스칼라는 static 키워드가 없어서 비슷한 역할의 object를 만들 수 있다.

**하위 타입 객체 반환**

public interface ParentClass {  
 void print();  
  
 static ParentClass create(String gender) {  
 if ("남자".equals(gender)) {  
 return new Boy();  
 } else {  
 return new Girl();  
 }  
 }  
}

**4) 네 번째, 입력 매개변수에 따라 매번 다른 클래스의 객체를 반환할 수 있다.**

**Enumset (파라미터 개수에 따라)**

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);  
}

**5) 정적 팩터리 메서드를 작성하는 시점에는 반환할 객체의 클래스가 존재하지 않아도 된다.**

서비스 제공자 프레임워크는 3개의 핵심 컴포넌트로 이루어진다.

1. 서비스 인터페이스 : 구현체의 동작을 정의

2. 제공자 등록 API : 제공자가 구현체를 등록할 때 사용

3. 서비스 접근 API : 클라이언트가 서비스의 인스턴스를 얻을 때 사용

+4. 서비스 제공자 인터페이스

**브릿지 패턴 (기능 계층과 구현 계층 분리하기)**

**-기능의 클래스 계층**

**-구현의 클래스 계층**

[](https://user-images.githubusercontent.com/7076334/46918131-07444d80-d009-11e8-9e81-6ebb02bcb3b1.png)

**정적 팩터리 단점**

**1) 상속을 하려면 public이나 protected 생성자가 필요하니 정적 펙터리 메서드만 제공하면 하위 클래스를 만들 수 없다.**

-상속보다 컴포지션 이용

-불변 타입 용이(접근자 제한, 상속 불가)

**2) 정적 팩터리 메서드는 프로그래머가 찾기 어렵다.**

-일반 static method 와 API doc에서 구분하지 않는다.

**ITEM2 생성자에 매개변수가 많다면 빌더를 고려하라**

**1) 다중 생성자**

public Constructor(String name, String age, String tel, String address) {  
 this.name = name;  
 this.age = age;  
 this.tel = tel;  
 this.address = address;  
}  
  
public Constructor(String name, String age, String tel) {  
 this.name = name;  
 this.age = age;  
 this.tel = tel;  
}

**2) 점층적 생성자 패턴**

public IncrementalConstructor(String name, String age) {  
 this(name, age, null, null);  
}  
  
public IncrementalConstructor(String name, String age, String tel) {  
 this(name, age, tel, null);  
}  
  
public IncrementalConstructor(String name, String age, String tel, String address) {  
 this.name = name;  
 this.age = age;  
 this.tel = tel;  
 this.address = address;  
}

public static void main(String[] args) {  
 IncrementalConstructor info = new IncrementalConstructor("심준보", 33);  
}

**3) 자바빈즈**

public void setName(String name) {  
 this.name = name;  
}  
  
public void setAge(Integer age) {  
 this.age = age;  
}  
  
public void setTel(String tel) {  
 this.tel = tel;  
}

public void setAddress(String address) {  
 this.address = address;  
}

public static void main(String[] args) {  
 JavaBeans info = new JavaBeans();  
 info.setName("심준보");  
 info.setAge(18);  
 info.setTel("010-1234-1234");  
}

****

**4) 얼린 자바빈즈 (잘 사용안함)**

public synchronized void setName(String name) {  
 checkNotFrozen();  
 this.name = name;  
}

...

public boolean isFrozen() {  
 return frozen;  
}  
  
public synchronized void freeze() {  
 frozen = true;  
}  
  
private void checkNotFrozen() {  
 if (frozen) throw new RuntimeException();  
}  
  
public static void main(String[] args) {  
 FreezeJavaBaens info = new FreezeJavaBaens();  
 info.setName("심준보");  
 info.freeze();

if (info.isFrozen()) {  
 // 얼린 후에 사용  
 }  
}

**5) Builder 패턴**

public class JavaBuilder {  
 private final String name;  
 private final String address;  
  
 private JavaBuilder(Builder builder) {  
 name = builder.name;  
 address = builder.address;  
 }  
  
 public static class Builder {  
 // 필수 매개변수  
 private final String name;  
  
 // 선택 매개변수  
 private String address = "";  
  
 public Builder(String name) {  
 this.name = name;  
 }

public Builder address(String address) {  
 this.address = address;  
 return this;  
 }  
  
 public JavaBuilder build() {  
 return new JavaBuilder(this);  
 }  
 }  
   
 public static void main(String[] args) {  
 JavaBuilder info = new Builder("심준보”)  
 .address("서울시")  
 .build();  
 }  
}

**build 에서 매개변수 검사하기**

public JavaBuilder build() {  
 JavaBuilder builder = new JavaBuilder(this);  
  
 if (builder.age < 0) {  
 throw new IllegalArgumentException("나이는 -가 될 수 없습니다.");  
 }  
  
 return builder;  
}

**제네릭**

|  |  |
| --- | --- |
| 공변성(covariant) | T’가 T의 서브타입이면, C<T’>는 C<T>의 서브타입이다. |
| 반공변성(contravariant) | T’가 T의 서브타입이면, C<T>는 C<T’>의 서브타입이다. |
| 무변성(invariant) | C<T>와 C<T’>는 아무 관계가 없다. |

**무변성 (자기 자신만 허용)**

List<String> cities = new ArrayList<>();  
cities.add("Changwon");  
cities.add("Seoul");  
cities.add("Suwon");  
cities.add("Yongin");  
*printCollectionGen*(cities); // 에러 List<String>은 Collection<Object>와 아무 관계가 없다.

public static void printCollectionGen(Collection<Object> collection) {  
 for (Object e : collection) {  
 System.*out*.println(e);  
 }  
}

List<String>과 Collection<Object>는 아무 관계가 없다.

**공변성 (List<? extends Number>)**

List<Integer> integers = new ArrayList<>();  
integers.add(1);  
  
List<? extends Number> numbers = integers;  
System.*out*.println(numbers); // [1]

**반공변성 (List<? super B>)**

// 반 공변성  
List<Number> numbers = new ArrayList<>();  
numbers.add(1);  
  
List<? super Integer> integers = numbers;  
System.*out*.println(integers); // [1]

**빌더 패턴 장점**

생성자 패턴 (불변) + 자바빈즈(가독성)

**빌더 패턴 단점**

빌더부터 만들어야 한다. (코드가 장황해 진다.)

**ITEM3 private 생성자나 열거 타입으로 싱글턴임을 보증하라**

**1) 고전방식 (잘못된 방식)**

public class Singleton {  
 private static Singleton *instance*;  
 private Singleton() {}  
 public static Singleton getInstance() {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 return *instance*;  
 }  
}

🡺멀티 쓰레드인 경우, 인스턴스 여러 번 생성

**2) Synchronized 방식 (잘못된 방식)**

public class Singleton {  
 private static Singleton *instance*;  
 private Singleton() {}  
 public static synchronized Singleton getInstance() {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 return *instance*;  
 }  
}

🡺Lock이 걸려서 비용낭비 심함

**3) DCL(Double-Checked-Locking) (잘못된 방식)**

public class Singleton {  
 private static Singleton *instance*;  
 private Singleton() {}  
 public static Singleton getInstance() {  
 if (*instance* == null) {   
 synchronized (Singleton.class) {  
 if (*instance* == null) {  
 *instance* = new Singleton();  
 }  
 }  
 }  
 return *instance*;  
 }  
}

🡺인스턴스가 없는 경우만 Lock을 잡지만, 재배치(reordering) 과정에서 이슈 생길 수 있음.

**해결점**

**1) public static 멤버가 final 필드 방식**

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

}

public static void main(String[] args) {  
 Elvis elvis = Elvis.*INSTANCE*;  
 elvis.leaveTheBuilding();  
}

🡺간결함

**🡺Private 생성자가 있기 때문에 Reflection으로 접근 가능**

**2) static factory method를 public static 멤버로 제공**

public class Elvis {  
 private static final Elvis *INSTANCE* = new Elvis();  
  
 private Elvis() {  
 if (*INSTANCE* != null) {  
 throw new RuntimeException("중복 생성");  
 }  
 }  
  
 public static Elvis getInstance() {  
 return *INSTANCE*;  
 }

/\*  
 public static Elvis getInstance() {  
 return new Elvis();  
 }  
 \*/

public void leaveTheBuilding() {  
 }

public static void main(String[] args) {  
 Elvis elvis = Elvis.*getInstance*();  
 elvis.leaveTheBuilding();  
 }

}

🡺API 변경 없이 싱글턴 ON/OFF

🡺제네릭 메서드로 전환 가능

🡺제네릭 싱글턴 팩터리로 전환 (하위 타입 객체 반환)

🡺메서드 레퍼런스로 사용가능

**🡺Private 생성자가 있기 때문에 Reflection으로 접근 가능**

**3) enum 사용**

public enum Elvis {  
 *INSTANCE*;  
  
 public void leaveTheBuilding() {  
  
 }  
}

public static void main(String[] args) {  
 Elvis elvis = Elvis.*INSTANCE*;  
 elvis.leaveTheBuilding();  
}

🡺 Private 생성자가 없기 때문에 Reflection으로 접근 불가능

**🡺ENUM 외에 상속 해야 한다면 이 방법은 사용못함**

**ITEM4 인스턴스화를 막으려거든 private 생성자를 사용하라**

public class UtilityClass {  
 private UtilityClass() {  
 throw new AssertionError();  
 }  
}