**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)

**ServiceLoader**

public final class ServiceLoader<S> implements Iterable<S>  
{

public static <S> ServiceLoader<S> load(Class<S> service) {  
 ClassLoader cl = Thread.*currentThread*().getContextClassLoader();  
 return new ServiceLoader<>(Reflection.*getCallerClass*(), service, cl);  
 }

}

**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)

**참고로 자바9부터 Integer, Double 생성자 Deprecated**

@Deprecated(since="9")  
public Integer(int value) {  
 this.value = value;  
}

**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 ( telescopring )**

public IncrementalConstructor(String name, String age) {  
 this(name, age, 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;  
}

**6) Lombok**

전체 Class Lombok 금지

@Builder  
public class UserProfile {  
 private final int userSeq;  
 private String userName;  
 private String email;  
}

문제점 : 빌더의 장점이 깨짐

UserProfile userProfile = new UserProfile(1, "simjunbo", "sjb@tmon.com");

해결 : private 생성자에 Lombok 사용

public class UserProfile {  
 private final int userSeq;  
 private String userName;  
 private String email;  
  
 @Builder  
 private UserProfile(int userSeq, String userName, String email) {  
 this.userSeq = userSeq;  
 this.userName = userName;  
 this.email = email;  
 }  
}

맹목적인 @Data 금지

@Data  
public class UserProfile {  
 private int userSeq;  
 private String userName;  
 private String email;  
  
 @Builder  
 private UserProfile(int userSeq, String userName, String email) {  
 this.userSeq = userSeq;  
 this.userName = userName;  
 this.email = email;  
 }  
}

문제점 : 불변 깨짐

UserProfile userProfile = UserProfile.*builder*()  
 .userName("심준보")  
 .userSeq(1)  
 .email("simjunbo@tmon.co.kr")  
 .build();  
  
  
userProfile.setUserName("장영운");  
userProfile.setEmail("jylock");

해결 : 명시적으로 @Getter사용하거나 final 필드 사용

@Getter  
@ToString  
public class UserProfile {  
 private final int userSeq;  
 private final String userName;  
 private final String email;  
  
 @Builder  
 private UserProfile(int userSeq, String userName, String email) {  
 this.userSeq = userSeq;  
 this.userName = userName;  
 this.email = email;  
 }  
}

**Scala Self Type (클래스 안에서 사용하는 모든 this의 타입이 어떤 것인지 가정하는 것)**

trait A { def foo = "foo" }  
  
trait B { this: A => def foobar = foo + "bar" }

**공변환 타이핑 (서브 클래스 타입으로 오버라이딩 가능)**

class Parent {  
 Parent get() {  
 return this;  
 }  
}  
  
class Child extends Parent {  
 @Override  
 Child get() {  
 return this;  
 }  
  
 void message() {  
 System.*out*.println("HI");  
 }  
}

**빌더 패턴 장점**

생성자 패턴 (불변) + 자바빈즈(가독성)

**빌더 패턴 단점**

빌더부터 만들어야 한다. (코드가 장황해 진다.)

**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**

**정적 메서드, 정적 필드 클래스**

-유틸성 : java.lang,Math, java.util.Arrays

-동반 클래스 : java.util.Collections (자바8부터 필요 없음)

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

**🡺private 생성자를 통해 객체 생성 방지**

**ITEM5 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라**

사용하는 자원에 따라 동작이 달라지는 클래스에는 정적 유틸리티 클래스나 싱글턴 방식이 적합하지 않다,

**의존 객체 주입 패턴**

public class SpellChecker {  
 // 불변  
 private final Lexicon dictionary;  
  
 public SpellChecker(Lexicon dictionary) {  
 this.dictionary = Objects.*requireNonNull*(dictionary);  
 }  
  
 public boolean isValid(String word) {  
 return dictionary.isValid(word);  
 }  
  
 public static void main(String[] args) {  
 SpellChecker checker = new SpellChecker(new Donga());  
 boolean result = checker.isValid("자바");  
 System.*out*.println(result);  
  
 checker = new SpellChecker(new Doosan());  
 result = checker.isValid("자바");  
 System.*out*.println(result);  
  
 }  
}

public interface Lexicon {  
 boolean isValid(String word);  
}  
  
class Doosan implements Lexicon {  
 @Override  
 public boolean isValid(String word) {  
 if ("자바".equals(word)) {  
 return true;  
 }  
  
 return false;  
 }  
}  
  
class Donga implements Lexicon {  
 @Override  
 public boolean isValid(String word) {  
 return false;  
 }  
}

**Supplier**

@FunctionalInterface  
public interface Supplier<T> {T get();  
}

**ITEM6 불필요한 객체 생성을 피하라**

불변 객체는 언제든 재사용할 수 있다.

1) 정적 팩터리 메서드를 사용하여 불필요한 객체 생성 제거 (캐시)

**Boolean**

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

public static Boolean valueOf(String s) {  
 return *parseBoolean*(s) ? *TRUE* : *FALSE*;  
}

**Integer**

private static class IntegerCache {  
 static final int *low* = -128;  
 static final int *high*;  
 static final Integer *cache*[];  
  
 static {  
 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);  
}

**유한 상태 머신**

-유한 상태 머신은 자신이 취할 수 있는 유한한 개수의 상태를 가진다. (정규식)

-그 중 반드시 하나의 상태만 취한다. (조건)

ex) 전구 ON/OFF

**slow**

static boolean isRomanNumeralSlow(String s) {  
 return s.matches("^(?=.)M\*(C[MD]|D?C{0,3})"  
 + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");  
}

// Pattern.java

public static boolean matches(String regex, CharSequence input) {  
 Pattern p = Pattern.*compile*(regex);  
 Matcher m = p.matcher(input);  
 return m.matches();  
}

**fast**

// Pattern.compile 자체를 캐싱

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})$");  
  
static boolean isRomanNumeralFast(String s) {  
 return ROMAN.matcher(s).matches();  
}

🡺성능은 좋지만, 만약 한번도 호출되지 않는다면 쓸데 없이 초기화 된다.

**어댑터 패턴**

[](https://user-images.githubusercontent.com/7076334/46731987-2a989100-ccc7-11e8-9775-70b30c4927d2.png)

-실제 작업은 Adaptee

-Adaptor는 Ataptee에게 위임(인터페이스 역할)

-어떨 때 사용할까? 기존 클래스(Adaptee)를 수정하지 않고 인터페이스(API)를 맞출 때

2) 오토박싱

**천만번 반복**

primitive (2m/s)

public static long iterativeSum(long n) {  
 long result = 0;  
 for (long i = 1L; i <= n; i++) {  
 result += i;  
 }  
 return result;  
}

Stream.iterate (언박싱 – 92m/s)

public static long sequentialSum(long n) {  
 return Stream.*iterate*(1L, i -> i + 1) // 박싱됨  
 .limit(n)  
 .reduce(0L, Long::*sum*);  
}

LongStream (박싱X - 4m/s)

public static long rangedSum(long n) {  
 return LongStream.*rangeClosed*(1, n)  
 .reduce(0L, Long::*sum*);  
}

**C# vs Java 벤치마킹 비교**

<http://bangjunyoung.blogspot.kr/2014/06/java-arraylist-vs-csharp-list-quicksort-benchmark.html>

🡺엄청난 성능 희생

Generic은 런타임 시점에서 Type을 삭제 한다 (Type Erasure)

Generic Type은 Object가 된다.

public class Container<T> {  
  
 private T data;  
  
 public T getData() {  
 return data;  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println("test");  
 }  
}

🡺런타임

public class Container {  
  
 private Object data;  
  
 public Object getData() {  
 return data;  
 }  
}

🡺primitive 형태는 Object가 될 수 없음… 결국 Generic에 primitive 사용 못함…

**하지만!! Java10 Vahalla (발할라 인큐베이터)에서 이를 극복하기 위한 노력 중**

그러면 Wrapper Class는 언제 사용할까?

🡺캐싱, 유틸(valueOf), DTO (null 반환)

3) 단순히 객체 생성을 위한 객체 풀 생성 금지

**ITEM7 다 쓴 객체 참조를 해제하라**

public Object pop() {  
 if (size == 0)  
 throw new EmptyStackException();  
 return elements[--size];  
}

스택이 커졌다가 줄어들 때 스택에서 꺼내진 객체들은 가비지 컬렉터가 회수하지 않는다.

참조 해제 해야 된다.

public Object pop() {  
 if (size == 0)  
 throw new EmptyStackException();  
 Object result = elements[--size];  
 elements[size] = null; // 다 쓴 참조 해제  
 return result;  
}

🡺null 처리 하면 이점은 참조할 경우 프로그램은 NullPointException을 던지며 종료

**메모리 릭**

1) 스택, 리스트 잘못된 해제

🡺보통 pop(stack)이나 remove(list) 사용하면 된다.

2) 로컬 캐시

🡺WeakHashMap 사용

3) 리스너, 콜백

🡺WeakHashMap 사용

**Java Reference**

-종류 : strong(new), soft, weak, phantom(referenceQueue)

-reachable: 유효한 참조가 있는 상태

-unreachable : 유효한 참조가 없는 상태 (GC 대상)



-root set : Stack, Native, MethodArea (이 3개가 GC대상인가 판별)



그 외…

**1) static**

private static Random *random* = new Random();  
public static final ArrayList<Double> *list* = new ArrayList<Double>(1000000);  
  
public static void main(String[] args) throws InterruptedException {  
 for (int i = 0; i < 10000000; i++) {  
 *list*.add(*random*.nextDouble());  
 }  
  
 System.*gc*();  
 Thread.*sleep*(10000); // to allow GC do its job  
}

🡺정적 사용에 주의

**2) unclosed stream**

static String firstLineOfFile(String path) throws IOException {  
 BufferedReader br = BufferedReader(new FileReader(path);  
  
 String str = null;  
 while (br.readLine() != null) {  
 str += br.readLine();  
 }  
 return str;  
}

🡺try-with-resources

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

**3) hashCode, equals 누락**

public class Sample {  
 public static void main(String[] args) {  
 Map<Object, Object> map = new HashMap<>();  
 while (true) {  
 map.put(new Key("key"), "value");  
 }  
 }  
}  
  
class Key {  
 public String key;  
  
 public Key(String key) {  
 this.key = key;  
 }  
}

🡺hashcode, equals 재정의

**ITEM8 finalizer와 cleaner 사용을 피하라**

-finalizer와 cleaner는 제때 실행되어야 하는 작업은 할 수 없다.

**finalize의 Exception은 무시됨**

protected void finalize() {  
 System.*out*.println("Good bye cruel world");  
 throw new RuntimeException("무시됨");  
}

**AutoCloseable**

public class FileInputStream implements AutoCloseable {  
 private String file;  
  
 public FileInputStream(String file) {  
 this.file = file;  
 }  
  
 public void read() {  
 System.*out*.println(file + "을 읽습니다.");  
 }  
  
 @Override  
 public void close() throws Exception {  
 System.*out*.println(file + "을 닫습니다.");  
 }  
}

public class Execute {  
 public static void main(String[] args) {

// try-with-resources  
 try (FileInputStream fi = new FileInputStream("sample.json")) {  
 fi.read();  
 } catch (Exception e) {  
 System.*out*.println(e.getStackTrace());  
 }  
 }  
}

**using (C#) == try-with-resources**

using (SqlConnection connection = new SqlConnection(connectionString))  
{  
 SqlCommand command = new SqlCommand(queryString, connection);  
 command.Connection.Open();  
 command.ExecuteNonQuery();  
}

**cleaner와 finalizer 를 그럼 언제 쓰나?**

1) 늦게라도 회수 하는게 안하는 것보다 낫다.

🡺9부터 finalizer는 deprecated 되었다. 그래도 쓸꺼면 cleaner를 사용하자.

**finalizer (9부터 Depreacted..)**

public class Object {

@Deprecated(since="9")  
 protected void finalize() throws Throwable { }  
}

Public class FileInputStream extends InputStream  
{

@Deprecated(since="9")  
 protected void finalize() throws IOException {  
 if ((fd != null) && (fd != FileDescriptor.*in*)) {

...  
 close();  
 }  
 }  
}

2) 네이티브 객체 해제 (쓸만 할 듯)

**ITEM9 try-finally 보다는 try-with-resources를 사용하라**

**전통적 해제 방법**

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

**전통적 해제 방법 (자원 여러 개)**

static void copy(String src, String dst) throws IOException {  
 InputStream in = new FileInputStream(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();  
 }  
}

**AutoCloseable interface**

public interface AutoCloseable {  
void close() throws Exception;  
}

**try-with-resources**

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

**다중 try-with-resources**

static void copy2(String src, String dst) throws IOException {  
 try (InputStream in = new FileInputStream(src);  
 OutputStream out = new FileOutputStream(src)) {  
 byte[] buf = new byte[*BUFFER\_SIZE*];  
 int n;  
 while ((n = in.read(buf)) >= 0) {  
 out.write(buf, 0, n);  
 }  
 }  
}

**CHAPTER3**

**ITEM10 equals는 일반 규약을 지켜 재정의하라**

다음중 하나라도 해당하면 재정의하지 않는다.

1) 각 인스턴스가 본질적으로 고유하다. (값이 아닌 개체)

2) 인스턴스의 ‘논리적 동치성’을 검사할 일이 없다.

3) 상위 클래스에서 재정의한 euqals가 하위 클래스에도 딱 들어 맞는다.

**4) 클래스가 private 이거나 package-private이고 equals 메서드를 호출할 일이 없다.**

5) 싱글턴 객체일 때

**equals 규약**

1) 반사성(reflexivity) : null이 아닌 모든 참조 값 x에 대해, x.equals(x)는 true 이다.

Vehicle x = new Vehicle("Focus", 2002, "Ford");

// 반사성

if (x.equals(x)) {  
 System.*out*.println("It is reflexive!");  
}

2) 대칭성(symmetry) : null이 아닌 모든 참조 값 x, y에 대해, x.equals(y)가 true면 y.equals(x)도 true 이다.

Vehicle x = new Vehicle("Focus", 2002, "Ford");  
Vehicle y = new Vehicle("Focus", 2002, "Ford");

// 대칭성  
if (x.equals(y) && y.equals(x)) {  
 System.*out*.println("It is symmetric!");  
}

3) 추이성(transitivity) : null이 아닌 모든 참조 값 x, y, z에 대해, x.equals(y),가 true이고 y.equals(z)도 true면 x.equals(z)도 true이다.

Vehicle x = new Vehicle("Focus", 2002, "Ford");  
Vehicle y = new Vehicle("Focus", 2002, "Ford");  
Vehicle z = new Vehicle("Focus", 2002, "Ford");  
  
// 추이성  
if (x.equals(y) && y.equals(z) && z.equals(x)) {  
 System.*out*.println("It is transitive!");  
}

4) 일관성(consistency) : null이 아닌 모든 참조 값 x, y에 대해 x.equals(y)를 반복해서 호출하면 항상 true를 반환하거나 항상 false를 반환한다.

Vehicle x = new Vehicle("Focus", 2002, "Ford");  
Vehicle y = new Vehicle("Focus", 2002, "Ford");  
  
// 일관성  
for (int i = 0; i < 5; i++) {  
 if (x.equals(y)) {  
 System.*out*.println("It is consistency!");  
 }

}

5) null아님 : null이 아닌 모든 참조 값 x에 대해 x.equals(null)은 false 이다.

Vehicle x = new Vehicle("Focus", 2002, "Ford");  
  
// null 아님  
if (x.equals(null) == false) {  
 System.*out*.println("Nothing equals null");  
}

**동치관계**

집합을 서로 같은 원소들로 이뤄진 부분집합으로 나누는 연산이다.

**대칭성 위반**

@Override  
public boolean equals(Object o) {  
 if (o instanceof CaseInsensitiveString)  
 return s.equalsIgnoreCase(((CaseInsensitiveString) o).s);  
  
 if (o instanceof String)  
 return s.equalsIgnoreCase((String) o);  
  
 return false;  
}  
  
public static void main(String[] args) {  
 CaseInsensitiveString cis = new CaseInsensitiveString("Polish");  
 String s = "polish";  
  
 if (cis.equals(s)) {  
 System.*out*.println("cis.equals(s) 동일");  
 }  
  
 // 대칭성 위배  
 if (s.equals(cis)) {  
 System.*out*.println("s.equals(cis) 동일");  
 }  
}

**추이성 위반**

public class Point {  
 private final int x;  
 private final int y;  
  
 public Point(int x, int y) {  
 this.x = x;  
 this.y = y;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (!(o instanceof Point))  
 return false;  
  
 Point p = (Point) o;  
 return p.x == x && p.y == y;  
 }  
  
 public static void main(String[] args) {  
 ColorPoint p1 = new ColorPoint(1, 2, Color.*RED*);  
 Point p2 = new Point(1, 2);  
 ColorPoint p3 = new ColorPoint(1, 2, Color.*BLUE*);  
  
 if (p1.equals(p2)) {  
 System.*out*.println("p1과 p2는 같다.");  
 }  
  
 if (p2.equals(p3)) {  
 System.*out*.println("p2와 p3는 같다.");  
 }  
  
 // 추이성 문제  
 if (p3.equals(p1)) {  
 System.*out*.println("p3와 p1은 같다.");  
 }  
 }  
}  
  
class ColorPoint extends Point {  
 private final Color color;  
  
 public ColorPoint(int x, int y, Color color) {  
 super(x, y);  
 this.color = color;  
 }  
  
 // 추이성이 깨진다.  
 @Override  
 public boolean equals(Object o) {  
 if (!(o instanceof Point))  
 return false;  
  
 // o가 Point면 색상을 무시하고 비교  
 if (!(o instanceof ColorPoint))  
 return o.equals(this);  
  
 return super.equals(o) && ((ColorPoint) o).color == color;  
 }  
}

객체 지향 언어의 동치관계에서 나타나는 근본적인 문제

🡺구체 클래스를 확장해 새로운 값을 추가하면서 equals 규약을 만족시킬 방법은 존재하지 않는다.

**상속 대신 컴포지션**

class ColorPoint {  
 private final Point point;  
 private final Color color;  
  
 public ColorPoint(int x, int y, Color color) {  
 point = new Point(x, y);  
 this.color = Objects.*requireNonNull*(color);  
 }  
public Point asPoint() {  
 return point;  
 }  
  
 @Override  
 public boolean equals(Object o) {  
 if (!(o instanceof ColorPoint))  
 return false;  
  
 ColorPoint cp = (ColorPoint) o;  
 return cp.point.equals(point) && cp.color.equals(color);  
 }  
  
 @Override  
 public int hashCode() {  
 return 31 \* point.hashCode() + color.hashCode();  
 }  
}

**잘못 설계된1) Date, Timestamp**

public class DateSample {  
 public static void main(String[] args) {  
 Date date = new Date();  
 Date stamp = new Timestamp(date.getTime());  
  
 System.*out*.println(date.equals(stamp));  
 // 대칭성 위배된다.  
 System.*out*.println(stamp.equals(date));  
 System.*out*.println(date.compareTo(stamp) == 0);  
 System.*out*.println(stamp.compareTo(date) == 0);  
 }  
}

🡺compareTo 사용하자

**일관성 위반**

**잘못된 설계2) URL equals**

protected boolean sameFile(URL u1, URL u2) {  
 // Compare the protocols.  
 if (!((u1.getProtocol() == u2.getProtocol()) ||  
 (u1.getProtocol() != null &&  
 u1.getProtocol().equalsIgnoreCase(u2.getProtocol()))))  
 return false;  
  
 // Compare the files.  
 if (!(u1.getFile() == u2.getFile() ||  
 (u1.getFile() != null && u1.getFile().equals(u2.getFile()))))  
 return false;  
  
 // Compare the ports.  
 int port1, port2;  
 port1 = (u1.getPort() != -1) ? u1.getPort() : u1.handler.getDefaultPort();  
 port2 = (u2.getPort() != -1) ? u2.getPort() : u2.handler.getDefaultPort();  
 if (port1 != port2)  
 return false;  
  
 // Compare the hosts.  
 if (!hostsEqual(u1, u2))  
 return false;  
  
 return true;  
}

🡺항시 메모리에 존재하는 객체만을 사용한 결정적 계산만 수행하자.

**Null 아님 위반**

@Override  
public boolean equals(Object o) {  
 if (!(o instanceof Point))  
 return false;

...  
}

instanceof를 사용하면 묵시적 null 검사를 할 수 있다.

**양질의 equals 메서드 구현 방법**

1. == 연산자를 사용해 입력이 자기 자신의 참조인지 확인한다.

2. instanceof 연산자로 입력이 올바른 타입인지 확인한다.

3. 입력을 올바른 타입으로 형변환 한다.

4. 입력 객체와 자신 자신의 대응되는 ‘핵심’ 필드들이 모두 일치

**ex) AbstractList**

public boolean equals(Object o) {  
 if (o == this) // 자기 자신의 참조 확인  
 return true;  
 if (!(o instanceof List)) // instanceof 연산자로 타입 확인  
 return false;  
  
 ListIterator<E> e1 = listIterator();  
 ListIterator<?> e2 = ((List<?>) o).listIterator(); // 올바른 타입으로 형변환  
 while (e1.hasNext() && e2.hasNext()) {  
 E o1 = e1.next();  
 Object o2 = e2.next();  
 if (!(o1==null ? o2==null : o1.equals(o2)))  
 return false;  
 }  
 return !(e1.hasNext() || e2.hasNext());  
}

부동소수 좀 찾아 보자

Float.Nan (Not A Number), -0.0f (float)

**주의사항**

-equals 재정의 할 땐 hashCode도 반드시 재정의 하자

-너무 복잡하게 해결하려 들지 말자.

-Object 외의 타입을 매개변수로 받는 equals 메서드를 선언하지 말자

**lombok, autovalue, immutables (hard boiler plate code)]**

**ITEM11 equals를 재정의하려거든 hashCode도 재정의하라**

**좋은 hashCode를 작성하는 간단한 요령**

1. int 변수 result를 선언 후 값 c로 초기화. 이때 c는 해당 객체의 첫 번째 핵심 필드

(핵심필드란 equals 비교에 사용되는 필드)

2-1. 나머지 핵심 필드 f 각각에 대해 다음 작업을 수행

- 기본 타입 필드라면 Type.hashCode(f)

- 참조 타입 필드라면 클래스의 equals 메서드가 이 필드의 equals를 재귀적으로 호출 시, 이

필드의 hashCode 재귀적으로 호출. 아니면 이 필드의 표준형을 만들어 사용

- 필드가 배열이라면, 핵심 원소 각각을 별도 필드처럼 다룬다.

핵심 원소가 하나도 없다면 단순히 상수(0 추천)를 사용한다.

모든 원소가 핵심 원소라면 Arrays.hashCode를 사용한다.

2-2. 단계 2-1에서 계산한 해시코드 c로 result를 갱신한다.

- result = 31 \* result + c;

3. result를 반환한다.

public class Member {  
 private String name;  
 private int age;  
 private String nickname;  
  
 @Override  
 public int hashCode() {  
 int result = ((name == null) ? 0 : name.hashCode()); // 참조 타입 필드라면  
 result = 31 \* result + Integer.*hashCode*(age); // 기본 타입 필드라면 Type.hashCode(f)  
 return result;  
 }  
}

**31을 사용 이유?** 소수 중, 적당한 크기의 소수인 31이 사용.

짝수 사용 시, 비트의 오른쪽이 0으로 가득 찬다.

**변경될 수 있는 객체는 equals, hashCode를 따로 구현하지 않는 것이 낫다.**

**Objects.hash 사용**

@Override  
public int hashCode() {  
 return Objects.*hash*(name, age);  
}

🡺사용하기 간편하나, 인수를 담기 위한 배열이 만들어 지고, 기본 타입이 있다면 박싱과 언박싱도 거쳐야 하기 때문에 성능은 좀 더 느리다.

**Open Addressing과 Separate Chanining**



🡺해시코드가 많이 겹칠수록 O(n)의 시간이 걸린다.

**hashCode가 반환하는 값의 생성 규칙을 API 사용자에게 공표하지 말자**

🡺이 값에 의지하지 않게 되고, 추후에 계산 방식을 바꿀 수도 있다.

**캐시 (pre loading)**

public class Member {  
  
 // 캐시  
 private static int *hashCode*;  
  
 private String name;  
 private int age;  
  
 public Member(String name, int age) {  
 this.name = name;  
 this.age = age;  
 this.*hashCode* = getHashCode(); // 초기화시 캐싱  
 }  
  
 // 캐시 (pre loading)  
 private int getHashCode() {  
 int result = ((name == null) ? 0 : name.hashCode()); // 참조 타입 필드라면  
 result = 31 \* result + Integer.*hashCode*(age); // 기본 타입 필드라면 Type.hashCode(f)  
 return result;  
 }  
  
 @Override  
 public int hashCode() {  
 return *hashCode*;  
 }  
}

**캐시 (lazy loading)**

public class MemberCache {  
 // 캐시  
 private int hashCode;  
  
 private String name;  
 private int age;  
  
 public MemberCache(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 // 캐시 (lazy loading)  
 @Override  
 public int hashCode() {  
 int result = hashCode;  
 if (result == 0) {  
 result = ((name == null) ? 0 : name.hashCode()); // 참조 타입 필드라면  
 result = 31 \* result + Integer.*hashCode*(age); // 기본 타입 필드라면   
 hashCode = result;  
 }  
 return result;  
 }  
}

**ITEM12 toString을 항상 재정의하라**

toString의 규약은 “모든 하위 클래스에서 이 메서드를 재정의하라”고 한다.

toString은 객체를 println, printf, 문자열 연결 연산자(+), assert, 디버거가 객체 출력할 때 자동으로 호출

포멧을 명시하든 아니든 의도는 명확히 밝혀야 한다.

**포멧이 있는 경우**

*/\*\*  
 \* 이 전화번호의 문자열 표현을 반환한다.  
 \* 이 문자열은 "XXX-YYY-ZZZZ" 형태의 12글자로 구성된다.  
 \* XXX는 지역 코드, YYYY는 프리픽스, ZZZZ는 가입자 번호다.  
 \* 각각의 대문자는 10진수 숫자 하나를 나타낸다.  
 \*  
 \* 전화번호의 각 부분의 값이 너무 작아서 자릿수를 채울 수 없다면,  
 \* 앞에서부터 0으로 채워나간다. 예컨대 가입자 번호가 123이라면  
 \* 전화번호의 맞미ㅏㄱ 네문자는 "0123"이 된다.  
 \*/*@Override  
public String toString() {  
 return String.*format*("%03d-%03d-%04d", areaCode, prefix, lineNum);  
}

**포멧이 없는 경우**

*/\*\*  
 \* 이 약물에 관한 대략적인 설명을 반환한다.  
 \* 다음은 이 설명의 일반적인 형태이나,  
 \* 상세 형식은 정해지지 않았으며 향후 변경될 수 있다.  
 \*  
 \* "[약물 #9: 유형=사랑, 냄새=테레빈유, 겉모습=먹물]"  
 \*/*@Override  
public String toString() {  
 return String.*format*("%03d-%03d-%04d", areaCode, prefix, lineNum);  
}

**toString의 자동생성이 적합하지 않는 경우**

@Override  
public String toString() {  
 return String.*format*("%03d-%03d-%04d", areaCode, prefix, lineNum);  
}

🡺자동 생성 시, 클래스의 의미를 모른다.

🡺그럼에도 불구하고 Object의 toString보다는 자동 생성된 toString (모든 정보가 포함되어 있는)

이 낫다.

**우리팀에서 자주 사용하는 방법**

@Override  
public String toString() {  
 return ToStringBuilder.*reflectionToString*(this, ToStringStyle.*JSON\_STYLE*);  
}

🡺결과

{"hashCode":1537519681,"name":"심준보","age":33,"nickname":"준보"}

**ITEM13 clone 재정의는 주의해서 진행하라**

**1) Object의 Cloneable은 protected 이다.**

public class A implements Cloneable {  
 private String name;  
 private int age;  
  
 public A(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
}

class B extends A {  
 public B(String name, int age) {  
 super(name, age);  
 }  
  
 public static void main(String[] args) throws CloneNotSupportedException {  
 B b = new B("심준보", 33);  
 B b2 = (B) b.clone(); // 접근 가능  
 }  
}

**2) 그러면 외부 객체는?**

public class C {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 A a = new A("심준보", 33);  
 A a2 = (A) a.clone(); // Object의 clone이 protected 이기 때문에 에러난다.  
 }  
}

에러남 ㅠㅠ

**3) 그럼 외부 객체에서 접근 하려면?**

public class A implements Cloneable {  
 . . . (생략)

@Override  
 public Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
}

접근 제한자 public으로 오버라이딩 하면 됨

**Cloneable 인터페이스는 Object의 protected 메서드인 clone의 동작 방식을 결정한다.**

Cloneable을 구현하지 않은 상태에서 clone을 호출하면 CloneNotSupportedException 에러남



**clone 메서드의 일반 규약은 허술하다**

**clone 메서드가 super.clone이 아닌, 생성자를 호출 했을 경우**

public class A implements Cloneable {  
 . . . (생략)  
 @Override  
 public Object clone() throws CloneNotSupportedException {  
 return new A();  
 }  
}

🡺컴파일러는 불평하지 않는다.

**하지만 하위 클래스에서 clone을 호출하면?**

class B extends A {  
 public static void main(String[] args) throws CloneNotSupportedException {  
 B b = new B();  
 B b2 = (B) b.clone();  
 }  
}

🡺 java.lang.ClassCastException 에러가 난다. A 객체가 생성되기 때문이다.

public class A implements Cloneable {  
. . . (생략)  
 @Override  
 public Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
}

🡺이런경우 super.clone을 연쇄적으로 호출하도록 구현하면 상위 클래스 객체가 만들어진다.

(생성자 연쇄와 살짝 비슷함)

**중요) 쓸데없는 복사를 지양한다는 관점에서 보면 불변 클래스는 굳이 clone 메서드를 제공하지 않는게 좋다.**

배열의 clone은 런타임 타입과 컴파일타임 타입 모두가 원본 배열과 똑 같은 배열을 반환한다.

private Object[] elements;

...(생략)

Stack result = (Stack) super.clone();  
result.elements = elements.clone();

상속용 클래스는 Cloneable을 구현해서는 안 된다.

Cloneable을 사용하는 것보다, 생성자와 팩터리를 이용하는게 낫다.

단 배열만은 clone 메서드 방식이 가장 깔끔하다.

**ITEM14 Comparable을 구현할지 고려하라**

**compareTo의 성격은 두가지만 빼면 Object의 equals와 같다.**

1) 순서까지 비교할 수 있다.

2) 제네릭 하다.

public interface Comparable<T> {public int compareTo(T o);  
}

**🡺파라미터가 제네릭으로 되어 있다.**

**compareTo 메서드의 일반 규약은 equals의 규약과 비슷한다.**

-객체가 주어진 객체보다 작으면 음의 정수, 같으면 0, 크면 양의 정수를 반환

-비교할 수 없는 타입객체가 주어지면 **ClassCastException**

**-부호함수는 수의 부호를 판별하는 함수다.**



1) sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) 여야 한다.

2) (x.compareTo(y) > 0 && y.compareTo(z) > 0) 이면 x.compareTo(z) > 0 이다.

3) x.compareTo(y) == 0이면 sgn(x.compareTo(z)) == sgn(y.compareTo(z)) 다.

4)(x.compareTo(y) == 0) == (x.equals(y)) (필수는 아님)

Collection, set, map 같은 인터페이스들은 equals 메서드 규약을 따른다고 되어 있지만,

정렬된 컬렉션들은 동치성을 비교할 때 equals 대신 compareTo를 사용한다.

**HashSet - put**

final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  
 boolean evict) {  
 Node<K,V>[] tab; Node<K,V> p; int n, i;  
 if ((tab = table) == null || (n = tab.length) == 0)  
 n = (tab = resize()).length;  
 if ((p = tab[i = (n - 1) & hash]) == null)  
 tab[i] = newNode(hash, key, value, null);  
 else {  
 Node<K,V> e; K k;  
 if (p.hash == hash &&  
 ((k = p.key) == key || (key != null && key.equals(k))))  
 e = p;  
 else if (p instanceof TreeNode)  
 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  
 else {  
 for (int binCount = 0; ; ++binCount) {  
 if ((e = p.next) == null) {  
 p.next = newNode(hash, key, value, null);  
 if (binCount >= *TREEIFY\_THRESHOLD* - 1) // -1 for 1st  
 treeifyBin(tab, hash);  
 break;  
 }  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 break;  
 p = e;  
 }  
 }  
 if (e != null) { // existing mapping for key  
 V oldValue = e.value;  
 if (!onlyIfAbsent || oldValue == null)  
 e.value = value;  
 afterNodeAccess(e);  
 return oldValue;  
 }  
 }  
 ++modCount;  
 if (++size > threshold)  
 resize();  
 afterNodeInsertion(evict);  
 return null;  
}

**TreeSet - put**

public V put(K key, V value) {

. . .(생략)  
 // split comparator and comparable paths  
 Comparator<? super K> cpr = comparator;  
 if (cpr != null) {  
 do {  
 parent = t;  
 cmp = cpr.compare(key, t.key);  
 if (cmp < 0)  
 t = t.left;  
 else if (cmp > 0)  
 t = t.right;  
 else  
 return t.setValue(value);  
 } while (t != null);  
 }  
 else {  
 if (key == null)  
 throw new NullPointerException();  
 @SuppressWarnings("unchecked")  
 Comparable<? super K> k = (Comparable<? super K>) key;  
 do {  
 parent = t;  
 cmp = k.compareTo(t.key);  
 if (cmp < 0)  
 t = t.left;  
 else if (cmp > 0)  
 t = t.right;  
 else  
 return t.setValue(value);  
 } while (t != null);  
 }

. . .(생략)  
}

public static void main(String[] args) {  
 BigDecimal bd1 = new BigDecimal("1.0");  
 BigDecimal bd2 = new BigDecimal("1.00");  
 Set<BigDecimal> hashSet = new HashSet<>();  
 Set<BigDecimal> treeSet = new TreeSet<>();  
  
 hashSet.add(bd1);  
 hashSet.add(bd2);  
 System.*out*.println(hashSet.size()); // size : 2  
  
 treeSet.add(bd1);  
 treeSet.add(bd2);  
 System.*out*.println(treeSet.size()); // size : 1  
}

🡺 equals와 compareTo의 결과가 달르기 때문에 HashSet과 TreeSet의 결과가 다르다

**막간> Comparable, Comparator 차이**

기본형 타입이 아닌 Object 내에 특정값을 비교할 경우에는

Comparable를 implements 해서 compareTo를 구현하면 Collections.sort() , Arrays.sort() 등을 사용할 수 있음.

ex)

public class Book implements Comparable<Book>{  
 private int price;  
 public Book(int price){  
 this.price = price;  
 }  
 public int getPrice(){  
 return this.price;  
 }  
  
 @Override  
 public int compareTo(Book b) {  
 return this.price - b.price; // 자신이 앞에 있는게 ascending order  
 }  
}

Comparator는 compareTo 에서 구현한 기본값 외에 별도의 기준으로 구현하고 싶을 때 다음과 같이 Comparator 구현체 또는 익명 Comparator 구현체를 이용하여 나타낼 수 있다.

**익명 Comparator**

Collections.sort(myBookList, new Comparator<Book>() {  
 @Override  
 public int compare(Book b1, Book b2) {  
 return b1.getPrice() - b2.getPrice();  
 }  
});

**결론 :** Comparable과 Comparator은 상호보완 차이

더 간단하게 람다를 사용하면 다음과 같은 형태로도 가능

Collections.sort(myBookList,

(Book b1, Book b2)-> b1.getPrice() - b2.getPrice());

or

list.sort(Comparator.*comparing*(Book::getPrice)); // 오름차순  
list.sort(Comparator.*comparing*(Book::getPrice).reversed()); // 내림차순

기본 타입 필드가 여럿일 때의 비교자

@Override  
public int compareTo(PhoneNumber pn) {  
 int result = Short.*compare*(areaCode, pn.areaCode); // 가장 중요한 필드  
 if (result == 0) {  
 result = Short.*compare*(prefix, pn.prefix); // 두 번째로 중요한 필드  
 if (result == 0) {  
 result = Short.*compare*(lineNum, pn.lineNum); // 세 번째로 중요한 필드  
 }  
 }  
 return result;  
}

**비교자 생성 메서드를 활용한 비교자 (약간의 성능저하가 발생)**

private static final Comparator<PhoneNumber> *COMPARATOR* =  
 Comparator.*comparingInt*((PhoneNumber pn) -> pn.areaCode)  
 .thenComparingInt(pn -> pn.prefix)  
 .thenComparingInt(pn -> pn.lineNum);  
@Override  
public int compareTo(PhoneNumber pn) {  
 return *COMPARATOR*.compare(this, pn);  
}

**객체 참조용 비교자 생성 메서드**

public static <T, U> Comparator<T> comparing(  
 Function<? super T, ? extends U> keyExtractor,  
 Comparator<? super U> keyComparator)  
{  
 Objects.*requireNonNull*(keyExtractor);  
 Objects.*requireNonNull*(keyComparator);  
 return (Comparator<T> & Serializable)  
 (c1, c2) -> keyComparator.compare(keyExtractor.apply(c1),  
 keyExtractor.apply(c2));  
}

**compareTo 메서드에서 필드의 값을 비교할 때 <와 > 연산자는 쓰지 말자.**

**대신 박싱된 기본 타입 클래스가 제공하는 정적 compare 메서드나**

**Comparator 인터페이스가 제공하는 비교자 생성 메서드 사용 하자**

**CHAPTER4**

**ITEM15 클래스와 멤버의 접근 권한을 최소화하라**

**잘 설계된 컴포넌트의 가장 큰 차이는 바로 클래스 내부 데이터와 구현 정보를 외부 컴포넌트로부터 얼마나 잘 숨겼느냐다.**

**정보은닉의 장점 (결국 독립적을 말하는 건가)**

-시스템 개발 속도를 높인다. 여러 컴포넌트를 병렬로 개발가능

-시스템 관리 비용을 낮춘다.

-정보 은닉 자체가 성능을 높여주지는 않지만, 성능 최적화에 도움을 준다. 다른 컴포넌트에 영향을 주지 않고 해당 컴포넌트만 최적화

-소프트웨어 재사용성을 높인다.

-큰 시스템을 제작하는 난이도를 낮춰준다.

**접근 제한자를 제대로 활용하는 것이 정보 은닉의 핵심이다.**

**모듈 : 가장 상위에 위치하는 구현 단위 (실질적으로 구현이 된 단위)**

**컴포넌트 : 런타임 엔티티를 참조하는 단위**

ex) 1개의 서버에게 서비스를 제공받는 100개의 클라이언트

🡺모듈은 서버 + 클라이언트 = 총 2개

🡺컴포넌트는 서버(1) + 클라이언트(100) = 101

접근 제한자 (좁은 것부터)

-private : 멤버를 선언한 톱레벨 클래스에서만 접근 가능

-package-private (default) : 멤버가 소속된 패키지 안의 모든 클래스에서 접근

-protected : package-private(default)의 접근 범위를 포함하여, 멤버를 선언한 클래스의 하위 클래스에서도 접근할 수 있다.

-public : 모든 곳에서 접근 가능

**Serializable을 구현한 클래스에서는 그 필드들도 의도치 않게 공개 API가 될 수 있다.**

**🡺설계할 때, 주의하자**

**테스트 목적일 경우 적당한 수준까지 넓혀도 괜찮다.**

🡺public클래스의 private 멤버를 package-private 까지 풀어주는 것까지는 허용 가능하다.

**public 클래스의 인스턴스 필드는 되도록 public이 아니어야 한다.**

🡺가별 필드를 갖는 클래스는 일반적으로 스레드에 안전하지 않다.

🡺final이면서 불변 객체를 참조하면, 내부 구현을 바꾸고 싶어도 그 public 필드를 없애는 방식으로는 리팩터링 할 수 없게 된다.

**예외로 필요한 구성요소로써의 상수라면 public static final 필드를 공개해도 된다.**

**클래스에서 public static final 배열 필드를 두거나 이 필드를 반환하는 접근자 메서드를 제공하면 안된다.**

public static final Thing[] *VALUES* = {...};

🡺클라이언트에서 배열을 수정할 수 있게 된다. 보안 허점이 존재

**해결책**

**1) 불변 리스트 추가**

private static final Thing[] *PRIVATE\_VALUES* = {new Thing()};  
public static final List<Thing> *VALUES* = Collections.*unmodifiableList*(Arrays.*asList*(*PRIVATE\_VALUES*));

**2) 배열을 private으로 만들고 그 복사본을 반환하는 public메서드 추가 (방어적 복사)**

private static final Thing[] *PRIVATE\_VALUES* = {new Thing()};  
public static final Thing[] values() {  
 return *PRIVATE\_VALUES*.clone();  
}

**자바9에서는 모듈 시스템이라는 개념이 도입되면서 두 가지 암묵적 접근 수준 추가**

-모듈 : 패키지의 묶음이다. exports로 공개할 것을 선언

module com.sjb {  
 requires java.base;  
 exports chapter4.item15;  
}

🡺protected 혹은 public 멤버라도 해당 패키지를 공개하지 않았다면 외부에서 접근 불가

**ITEM16 public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라**

**Public 클래스 필드를 직접 노출하지 말라는 규칙을 어긴 사례 (java.awt.Dimenstion)**

public class Dimension extends Dimension2D implements java.io.Serializable {public int width;public int height;  
}

public final class Time {  
 private static final int *HOURS\_PER\_DAY* = 24;  
 private static final int *MINUTES\_PER\_HOUR* = 60;  
  
 public final int hour;  
 public final int minute;  
  
 public Time(int hour, int minute) {  
 if (hour < 0 || hour >= *HOURS\_PER\_DAY*) {  
 throw new IllegalArgumentException("시간 : " + hour);  
 }  
  
 if (minute < 0 || minute >= *MINUTES\_PER\_HOUR*) {  
 throw new IllegalArgumentException("분 : " + minute);  
 }  
  
 this.hour = hour;  
 this.minute = minute;  
 }  
}

🡺 public 클래스의 필드가 불변이라면 직접 노출의 단점이 조금 줄어 들지만, 여전히 좋지 않다.

필드가 public 으로 노출되기 때문에 수정할 수 없다.

**ITEM17 변경 가능성을 최소화하라**

**불변으로 만들경우 다섯가지 규칙**

1) 객체의 상태를 변경하는 메서드를 제공하지 않는다. ex) setter

2) 클래스를 확장할 수 없도록 한다. ex) final class, private 생성자

3) 모든 필드를 final로 선언한다.

4) 모든 필드를 private으로 선언한다.

5) 자신 외에는 내부의 가변 컴포넌트에 접근할 수 없도록 한다.

불변 객체는 단순하다. 불변 객체는 생성된 시점의 상태를 파괴될 때까지 그대로 간직한다.

불변 객체는 근복적으로 스레드 안전하여 따로 동기화할 필요 없다.

🡺 한번 만든 인스턴스를 최대한 재활용하기를 권한다.

**ex) 자주 사용되는 인스턴스 캐싱**

public static final Complex *ZERO* = new Complex(0, 0);  
public static final Complex *ONE* = new Complex(1, 0);  
public static final Complex *I* = new Complex(0, 1);

@HotSpotIntrinsicCandidate  
public String(String original) {  
 this.value = original.value;  
 this.coder = original.coder;  
 this.hash = original.hash;  
}

🡺String 클래스의 복사 생성자는 초창기때 만들어진 것으로, 되도록 사용하지 말자

**불변 객체는 자유롭게 공유할 수 있음은 물론, 불변 객체끼리는 내부 데이터를 공유할 수 있다.**

BigInteger(int[] magnitude, int signum) {  
 this.signum = (magnitude.length == 0 ? 0 : signum);  
 this.mag = magnitude; // magnitude 공유  
 if (mag.length >= *MAX\_MAG\_LENGTH*) {  
 checkRange();  
 }  
}

**객체를 만들 때 다른 불변 객체들을 구성요소로 사용하면 이점이 많다**

ex) 맵의 키, Set의 원소

**불변 객체는 그 자체로 실패 원자성을 제공한다.**

**불변 클래스 단점**

🡺값이 다르면 객체 새로 만들어야 한다. 리소스 낭비가 심할 수 있음

**객체를 완성하기까지 단계가 많은 경우 (성능 문제)**

1)다단계 연산 (내부적으로 가변 동반 클래스)

ex) default BigInteger, StringBuilder

2)클래스를 public 으로 제공 (가변)

**생성자 대신 정적 팩터리를 사용한 불변**

public class Complex2 {  
 private final double re;  
 private final double im;  
  
 private Complex2(double re, double im) {  
 this.re = re;  
 this.im = im;  
 }  
  
 public static Complex2 valueOf(double re, double im) {  
 return new Complex2(re, im);  
 }  
}

🡺생성자가 private 이라 상속 못함

**모든 클래스를 불변으로 만들 수 없지만, 불변으로 만들 수 없는 클래스라도 변경할 수 있는 부분을 최소한으로 줄이자**

**다른 합당한 이유가 없다면 모든 필드는 private final 이어야 한다.**

**생성자는 불변식 설정이 모두 완료된, 초기화가 완벽히 끝난 상태의 객체를 생성해야 한다.**

🡺생성자와 정적 팩터리 외에는 그 어떤 초기화 메서드도 public으로 제공하면 안된다.

(특히 setter)

**ITEM18 상속보다는 컴포지션을 사용하라**

같은 패키지안에서 상속은 안전하지만, 다른 패키지로 넘어가면 위험하다. (인터페이스 상속 무관)

public class InstrumentedHashSet<E> extends HashSet<E> {  
 // 추가된 원소의 수  
 private int addCount = 0;  
  
 public InstrumentedHashSet() {  
 }  
  
 public InstrumentedHashSet(int initCap, float loadFactor) {  
 super(initCap, loadFactor);  
 }  
  
 @Override  
 public boolean add(E e) {  
 addCount++;  
 return super.add(e);  
 }  
  
 // 해당 메서드를 재정의하지 않으면 문제를 고칠 수 있다.  
 // 하지만 HashSet의 addAll이 add 메서드를 이용해 구현했음을 가정한 해법이라는 한계를 지닌다.  
 @Override  
 public boolean addAll(Collection<? extends E> c) {  
 addCount += c.size();  
 return super.addAll(c);  
 }  
  
 // 요구 조건 : 생성 이후, 원소가 몇 개 더해졌는지 알 수 있어야 한다.  
 public int getAddCount() {  
 return addCount;  
 }  
}

**컴포지션 :** 기존 클래스가 새로운 클래스의 구성 요소로 쓰인다는 뜻

**위임 :** 컴포지션과 전달의 조합은 넓은 의미로 위임(delegation)이라고 부른다.

단, 엄밀히 따지면 래퍼 객체가 내부 객체에 자기 자신의 참조를 넘기는 경우만 위임에 해당

**래퍼 클래스는 단점이 거의 없다.**

🡺콜백 프레임워크와는 어울리지 않는다는 점만 주의하면 된다.

🡺콜백 시 자신의 참조를 넘기기 때문에 레퍼 클래스가 아닌 내부 객체를 호출하게 된다.

**ex) 구아바의 전달 메서드**

<https://github.com/google/guava/wiki/CollectionHelpersExplained>

A is-a B (A는 B이다) : 강아지는 동물이다.

A has-a B (A가 B를 가지고 있음) : 새는 날개를 가지고 있다.

**is-a의 잘못된 예)**

Stack – Vector

Properties – Hashtable

**is-a 관계라고 해도, 하위 클래스의 패키지가 상위 클래스와 다르고, 상위 클래스가 확장을**

**고려해 설계되지 않았다면 문제가 될 수 있다.**

public static void main(String[] args) {  
 User user = new User("simjunbo", 33);  
  
 Properties properties = new Properties();  
 properties.put("simjunbo", user);  
  
 User result = (User) properties.get("simjunbo");  
 System.*out*.println(result);  
}

🡺 properties의 첫 의도는 키와 값으로 문자열만 허용하도록 설계하려고 했었다.

**ITEM19 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라.**

**상속을 고려한 설계와 문서화란?**

상속용 클래스는 재정의할 수 있는 메서드들을 내부적으로 어떻게 이용하는지(self-use)

문서로 남겨야 한다.

**@implSpec 내부 동작 방식 설명**

*// AbstractCollection*

*/\*\*  
 \** ***@implSpec***

***\* 생 략...*** *\*/*public boolean remove(Object o) {  
 Iterator<E> it = iterator();  
 if (o==null) {  
 while (it.hasNext()) {  
 if (it.next()==null) {  
 it.remove();  
 return true;  
 }  
 }  
 } else {  
 while (it.hasNext()) {  
 if (o.equals(it.next())) {  
 it.remove();  
 return true;  
 }  
 }  
 }  
 return false;  
}

🡺설명을 읽어 보면, iterator 메서드를 재정의하면 remove 메서드의 동작에 영향을 준다.

🡺내부 구현 방식을 설명해야 되는 이유는 상속이 캡슐화를 해치기 때문에 일어나는 안타까운

현실이다.

// AbstractList

protected void removeRange(int fromIndex, int toIndex) {  
 ListIterator<E> it = listIterator(fromIndex);  
 for (int i=0, n=toIndex-fromIndex; i<n; i++) {  
 it.next();  
 it.remove();  
 }  
}

🡺이 메서드를 재정의하면 이 리스트와 부분리스트의 clear 연산 성능을 크게 개선할 수 있다.

그렇다면 상속용 클래스를 설계할 때 어떤 메서드를 protected로 노출해야 할까?

🡺답은 없다. 심사숙고해서 잘 예측해본 다음, 실제 하위 클래스를 만들어 시험해보는 것이 최선이다.

🡺책에 나온 거는 하위 클래스 3개 정도… 그리고 이 중 하나 이상은 제 3자가 작성

API 설명과 상속용 설명을 구분해 주는 것이 좋다.

상속용 클래스의 생성자는 직접적으로든 간접적으로든 재정의 가능 메서드를 호출해서는 안 된다.

🡺어기면 프로그램 오작동 한다.

public class Super {

// 잘못된 예 - 생성자가 재정의 가능 메서드를 호출한다.  
 public Super() {  
 overrideMe();  
 }  
  
 public void overrideMe() {  
  
 }  
}

public final class Sub extends Super {  
 private final Instant instant;  
 Sub() {  
 instant = Instant.*now*();  
 }

// 상위 클래스 생성자에서 호출되기 때문에 처음에는 초기화가 되지 않는다.  
 @Override

public void overrideMe() {  
 System.*out*.println(instant);  
 }  
  
 public static void main(String[] args) {  
 Sub sub = new Sub();  
 sub.overrideMe();  
 }  
}

반대로

private, final, static 메서드는 재정의가 불가능하니 생성자에서 안심하고 호출해도 된다.

**Cloneable**과 **Serializable** 둘 중 하나라도 구현한 클래스를 상속할 수 있게 설계하는 것은 일반적으로 좋지 않은 생각이다.

**생성자와 비슷하게 clone과 readObject 모두 직접적으로든 간접적으로든 재정의 가능 메서드를 호출해서는 안 된다.**

-readObject의 경우 하위 클래스의 상태가 미처다 역직렬화되기 전에 재정의한 메서드부터 호출

-clone의 경우 복제본의 상태를(올바른 상태로) 수정하기 전에 재정의한 메서드를 호출

**그렇다면 그 외의 일반적인 구체 클래스는? (상속용이 아닌)**

🡺상속용으로 설계되거나 문서화되지도 않았다.

🡺클래스에 변화가 생길 때마다 하위 클래스를 오동작하게 만들 수 있기 때문이다.

🡺해결법은 상속용으로 설계하지 않은 클래스는 상속을 금지하는 것이다.

**상속금지 방법**

-클래스를 final로 선언

-모든 생성자를 private이나 default로 선언하고 public 정적 팩터리를 만들어 준다.

**만약 그럼에도 불구하고 구체 클래스를 상속하고 싶으면?**

-내부에서 재정의 가능 메서드 사용 금지.

-이 사실을 문서로 남기기

**도우미 메서드는 무엇일까?**

🡺실제 사용되는 class 내부에서 선언 될 때는 private 하게 관리된다.

유틸 클래스와 다른점은 유틸 클래스는 static 메서드만 가지고 있고 상태는 없다.

그리고 인스턴스화 시킬 수 없지만, 도우미 클래스는 인스턴스화 가능하며, static 메서드 뿐만 아니라 상태도 가질 수 있다.

**ITEM20 추상클래스보다는 인터페이스를 우선하라**

기존 클래스 위에 새로운 추상 클래스를 끼워넣기는 어렵다.

🡺두 클래스가 같은 추상 클래스를 확장하기 원한다면, 공통 조상이어야 한다. 이 방식은 클래스 계층구조에 커다란 혼란을 일으킨다.

🡺추상 클래스를 **상속** 받으면 반드시 하위 클래스가 되어야 한다.

🡺인터페이스는 상속과 무관하다. (**구현**)

public class A {  
 public void print() {  
 System.*out*.println("프린트");  
  
 }  
}  
  
class B extends A {  
  
}  
  
class C extends B {  
 public void subPrint() {  
 print();  
 }  
}

🡺모든 자식에서 abstract 기능을 상속 받고 있다.

🡺하지만 자바8의 interface도 default method를 가지고 있는데???

**인터페이스는 믹스인(mixin) 정의에 안성맞춤이다.**

믹스인이란 클래스가 구현할 수 있는 타입으로, ‘주된 타입’ 외에도 특정 선택적 행위를 제공한다.

ex) Cloneable, Comparable

**인터페이스로는 계층구조가 없는 타입 프레임워크를 만들 수 있다.**

**래퍼 클래스 관용구(컴포지션)와 함께 사용하면 인터페이스는 기능을 향상 시키는 안전하고 강력한 수단이 된다.**

**인터페이스의 디폴트 메서드를 제공할 때는 상속하려는 사람을 위한 설명을 @implSpec 자바독 태그를 붙여 문서화해야 한다.**

public interface A {  
 boolean equals(Object o);  
  
 int hashCode();  
}  
  
class B implements A {  
  
}

🡺인터페이스에 equals랑 hashCode가 선언되어 있어도, 클래스에서 구현 하지 않아도 된다.

왜일까? Interface를 구현하는 클래스는 Object에서 파생되었기 때문이 아닐까?

public interface A {  
 boolean equals(Object o);  
  
 int hashCode();  
}  
  
class B implements A {  
  
 @Override  
 public boolean equals(Object o) {  
 return true;  
 }  
  
 @Override  
 public int hashCode() {  
 return 0;  
 }  
}

🡺Override로 구현을 해도 Object 메서드를 구현하는 것이다.

그러면 왜 인터페이스에 equals랑 hashCode를 선언하는 것일까?

강제보다는 구현을 권장하는 의미가 아닐까?

관례상 인터페이스 이름이 Interface라면 그 골격 구현 클래스의 이름은 AbstractInterface로 짖는다.

ex) AbstractCollection, AbstractSet, AbstractList, AbstractMap

**1. 메서드 방식의 구체 클래스**

public class Sample {  
 */\*\*  
 \* 골격 구현을 사용해 완성한 구체 클래스  
 \*/* static List<Integer> intArrayAsList(int[] a) {  
 Objects.*requireNonNull*(a);  
  
 return new AbstractList<>() {  
 @Override public Integer get(int i) {  
 return a[i];  
 }  
  
 @Override public Integer set(int i, Integer val) {  
 int oldVal = a[i];  
 a[i] = val;  
 return oldVal;  
 }  
  
 @Override public int size() {  
 return a.length;  
 }  
 };  
 }  
  
 public static void main(String[] args) {  
 int[] a = { 1, 2, 3, 4, 5 };  
 List<Integer> result = *intArrayAsList*(a);  
 Integer get = result.get(0);  
 }  
}

제대로 설계된 골격 구현 클래스는 그 인터페이스로 나름의 구현을 만들려는 프로그래머의 일을 상당히 덜어준다.

골격 구현 클래스의 아름다움은 추상 클래스처럼 구현을 도와주는 동시에, 추상 클래스로 타입을 정의할 때 따라오는 심각한 제약에서 자유롭다. (익명 클래스)

**2. 내부 클래스 방식** (시뮬레이트한 다중 상속)

*/\*\*  
 \* 내부 클래스 방식 (래퍼 클래스랑 비슷)  
 \*/*public class Sample2 {  
 private InnerClass innerClass;  
  
 Sample2() {  
 int a[] = { 1, 2, 3, 4, 5, 6 };  
 innerClass = new InnerClass(a);  
 }  
  
 public Integer get(int i) {  
 return innerClass.get(i);  
 }  
  
 public int size() {  
 return innerClass.size();  
 }  
  
 private static class InnerClass extends AbstractList {  
  
 int[] a;  
  
 InnerClass(int[] a) {  
 this.a = a;  
 }  
  
 @Override public Integer get(int i) {  
 return a[i];  
 }  
  
 @Override public int size() {  
 return a.length;  
 }  
 }  
  
 public static void main(String[] args) {  
 Sample2 sample = new Sample2();  
 System.*out*.println(sample.get(0));  
 System.*out*.println(sample.size());  
 }  
}

다중 상속의 많은 장점을 제공하는 동시에 단점은 피한다.

**3. 골격 구현 클래스**

기반 메서드나 디폴트 메서드로 만들지 못한 메서드가 남아 있다면, 이 인터페이스를 구현하는 골격 구현 클래스를 하나 만들어 남은 메서드들을 작성해 넣는다.

ex) AbstractMapEntry

**4. 단순 구현**

단순 구현도 골격 구현과 같이 상속을 위해 인터페이스를 구현한 것이지만, 추상 클래스가 아니란 점이 다르다.

ex) AbstractMap.SimpleEntry

**ITEM21 인터페이스는 구현하는 쪽을 생각해 설계해라**

**Collection.java**

default boolean removeIf(Predicate<? super E> filter) {  
 Objects.*requireNonNull*(filter);  
 boolean removed = false;  
 final Iterator<E> each = iterator();  
 while (each.hasNext()) {  
 if (filter.test(each.next())) {  
 each.remove();  
 removed = true;  
 }  
 }  
 return removed;  
}

🡺Predicate를 이용해 true를 반환 하는 모든 원소 제거

**디폴트 메서드의 잘못된 예**

**Apache SynchronizedCollection.java**

public class SynchronizedCollection<E> implements Collection<E>, Serializable {

@Override  
 public boolean isEmpty() {  
 synchronized (lock) {  
 return decorated().isEmpty();  
 }  
 }

}

Apache SynchronizedCollection에서는 모든 메서드의 lock을 걸어서 동기화 시켜 주는데

removeIf의 디폴트 구현을 물려 받게 되어서, lock을 걸 수 없게 된다.

🡺멀티 스레드 환경에서 ConcurrentModificationException이 발생할 수도 있다.

**자바 플랫폼 라이브러리 내에 해결**

static class SynchronizedCollection<E> implements Collection<E>, Serializable {  
 @Override  
 public boolean removeIf(Predicate<? super E> filter) {  
 synchronized (mutex) {return c.removeIf(filter);}  
 }  
}

인터페이스의 디폴트 메서드를 재정의하고, 다른 메서드에서는 디폴트 메서드를 호출하기 전에 필요한 작업을 수행하도록 했다.

🡺하지만 제 3의 구현체들은 미리 알지 못하기 때문에 수정이 늦거나, 안되어 있다.

**디폴트 메서드는 컴파일에 성공하더라도 기존 구현체에 런타임 오류를 일으킬 수 있다.**

**기존 인터페이스는 디폴트 메서드 추가 시 주의!**

**신규는 상관없음 (유용함)**

인터페이스를 릴리스한 후, 결함을 수정하는 것은 매우 어렵기 때문에 배포전에

여러 테스트를 거쳐야 한다.

**ITEM22 인터페이스는 타입을 정의하는 용도로만 사용하라**

클래스가 어떤 인터페이스를 구현한다는 것은 자신의 인스턴스로 무엇을 할 수 있는지를 클라이언트에 얘기해주는 것이다. (명세서)

**상수 인터페이스** : 메서드 없이, 상수를 뜻하는 static final 필드로만 가득 찬 인터페이스

*/\*\*  
 \* 사용 금지  
 \*/*public interface PhysicalConstants {  
 static final double *AVOGADROS\_NUMBER* = 6.022\_140\_857e23;  
 static final double *BOLTZMANN\_CONSTANT* = 1.380\_648\_52e-23;  
 static final double *ELECTRON\_MASS* = 9.109\_383\_56e-31;  
}

🡺인터페이스를 잘못 사용한 예

🡺클라이언트 코드에 혼란을 준다. 이 상수들에 종속되면 바이너리 호환성을 위해 여전히 상수 인터페이스를 구현하고 있어야 한다.

🡺잘못된 예 java.io.ObjectStreamConstants

**참고) instance에 생략된 내용들**

// instance에 모든 멤버 변수는 public static final 이다.  
final double *PI* = 3.14; // public static 생략  
static int *HEART* = 10; // public final 생략  
  
// instance에 모든 메서드는 public abstract 이다.  
String getCard(); // public abstract 생략

**상수 유틸리티 클래스 (상수 인터페이스 대신 사용한다.)**

*/\*\*  
 \* 상수 유틸리티 클래스  
 \*/*public class PhysicalConstants {  
 // 인스턴스 방지  
 private PhysicalConstants() {  
 }  
  
 public static final double *AVOGADROS\_NUMBER* = 6.022\_140\_857e23;  
 public static final double *BOLTZMANN\_CONSTANT* = 1.380\_648\_52e-23;  
 public static final double *ELECTRON\_MASS* = 9.109\_383\_56e-31;  
}

정적 import를 사용해서 상수 이름으로만 사용하기

import static chapter4.item22.abst.PhysicalConstants.*AVOGADROS\_NUMBER*;  
  
public class Test {  
 double atoms(double mols) {  
 return *AVOGADROS\_NUMBER*;  
 }  
}

**ITEM23 태그 달린 클래스보다는 클래스 계층구조를 활용하라**

**태그(주석) 달린 클래스의 단점**

1) 열거 타입 선언, 태그 필드, switch 문등 쓸데 없는 코드가 많다.

2) 여러 구현이 한 클래스에 혼합돼 있어서 가독성도 나쁘다.

3) 다른 의미를 위한 코드도 언제나 함께 하니 메모리도 많이 사용한다.

4) 필드를 final로 선언하려면 해당 의미에 쓰이지 않는 필드들가지 생성자에서 초기화 해야 한다.

🡺태그 달린 클래스는 장황하고, 오류를 내기 쉽고, 비효율적이다.

내 생각 : 태그는 설명을 하기 위해 달리는 것이기 때문에 여러 기능들이 혼합되어 있을 가능성이 크다.

**클래스 태그 필드 => 계층구조 리팩토링**

**ITEM24 멤버 클래스는 되도록 static으로 만들라**

**중첩 클래스(nested class)** 란 다른 클래스 안에 정의된 클래스

public class OuterClass {  
 private static class InnterClass {  
  
 }  
}

**톱 클래스**

public class OuterClass {  
  
}  
  
class TopClass {  
   
}

**중첩 클래스 종류**

중첩 클래스를 사용하면 불필요한 관계 클래스를 감춤으로써 코드의 복잡성을 줄일 수 있다.

**1) 멤버 클래스**

Outer class 자원들 다 접근 가능

*/\*\*  
 \* 멤버 클래스(비정적)  
 \*/*public class Outer {  
 class Inner {  
  
 }  
}

**2) 지역 클래스**

Inner 클래스의 객체생성은 outer 메서드내에서 한다.

*/\*\*  
 \* 로컬 클래스  
 \* 메서드 안에 정의되어 있기 때문에 로컬 변수 처럼 인식된다.  
 \* 메서드가 호출될 때 생성되며 메서드가 종료될 때 삭제된다.  
 \*/*

public class Outer {  
 public void outerMethod() {  
 class Inner {  
 public void print() {  
 outerPrint();  
 System.*out*.println(a + " " + b + " " + *c*);  
 }  
 }  
  
 // Inner 클래스의 객체생성은 outer 메서드내에서 한다.  
 Inner i = new Inner();  
 i.print();  
 }  
}

**3) 정적 멤버 클래스**

Outer 클래스의 static 변수만 접근 가능

*/\*\*  
 \* 정적 멤버 클래스  
 \*/*public class Outer {  
 static class Inner {  
  
 }  
}

**4) 익명 클래스**

클래스명을 가지지 않으며 객체 생성과 메서드 선언만을 정의한다.

public interface Anonymous {  
 void print();  
}

public class Sample {  
 public static void main(String[] args) {  
 Anonymous anonymous = new Anonymous() {  
 @Override  
 public void print() {  
 System.*out*.println("anonymous class");  
 }  
 };  
 anonymous.print();  
 }  
}

**중첩 클래스 용도**

메서드 안에 정의하기엔 너무 길면 멤버 클래스로 만든다.

멤버 클래스의 인스턴스 각각이 바깥 인스턴스를 참조한다면 **비정적** 으로,

그렇지 않으면 **정적** 으로 만든다.

중첩 클래스가 한 메서드 안에서만 쓰이면서 인스턴스를 생성하는 지점이 단 한곳이고 해당 타입으로 쓰기에 적합한 클래스나 인터페이스가 이미 있다면 **익명 클래스** 로,

그렇지 않으면 **지역 클래스** 로 만들자.

**ITEM25 톱레벨 클래스는 한 파일에 하나만 담으라**

**Dessert.java**

// 톱 레벨 클래스  
class Utensil {  
 static final String *NAME* = "pot";  
}  
  
class Dessert {  
 static final String *NAME* = "pie";  
}

**Utensil.java**

// 톱 레벨 클래스  
class Utensil {  
 static final String *NAME* = "pan";  
}  
  
class Dessert {  
 static final String *NAME* = "cake";  
}

컴파일러에 어느 소스 파일을 먼저 건네느냐 따라서 동작이 달라진다.

**potpie** 가 출력될 수도 있고 **pancake** 가 출력될 수도 있다.

🡺이슈다.

해결책으로 톱레벨 클래스들(Utensil과 Dessert)을 서로 다른 소스 파일로 분리하면 된다.

(패키지의 분리를 의미하는 것인가?)

굳이 톱레벨 클래스를 한파일에 담고 싶으면 정적 멤버 클래스를 사용하자

public class Test {  
 public static void main(String[] args) {  
 System.*out*.println(Utensil.*NAME* + Dessert.*NAME*);  
 }  
  
 private static class Utensil {  
 static final String *NAME* = "pan";  
 }  
  
 private static class Dessert {  
 static final String *NAME* = "cake";  
 }  
}

🡺정적 멤버 클래스가 읽기도 좋고, private으로 선언하면 접근 범위도 최소로 관리할 수 있기 때문이다.

**CHAPTER5**

**ITEM26 로 타입은 사용하지 말라**

클래스와 인터페이스 선언에 타입 매개변수가 쓰이면, 이를 제네릭 클래스 혹은 제네릭 인터페이스라고 한다.

🡺이를 통틀어 제네릭 타입 이라고 한다.

로타입이란 제네릭 타입에서 타입 매개변수를 전혀 사용하지 않을 때를 말한다.

🡺목적은 제네릭이 도래하기 전 코드와 호환되도록 하기 위한 궁여지책

public class Sample {  
 private final Collection stamps = ...;  
}

**타입 안전성 확보**

public class Sample {  
 private final Collection<Stamp> stamps = ...;  
}

제네릭 타입을 쓰고 싶지만 실제 타입 매개변수가 무엇인지 신경 쓰고 싶지 않다면 와일드카드(?)를 사용하자. (비한정적 와일드 카드)

static int numElementsInCommon(Set<?> s1, Set<?> s2) { . . . }

**Set<?>과 로 타입 Set의 차이?**

와일드카드 타입은 안전하고, 로 타입은 안전하지 않다.

List<?> list = new ArrayList<>();  
list.add("verboten"); // 에러 발생

🡺불변 식을 훼손하지 못하게 막는다.

**예외**

**1) class 리터럴에는 로 타입을 써야 한다.**

ex) List.class, String[].class, int.class (O), List<String>.class, List<?>.class (X)

**2) instanceof 연산자 (런타임에는 제네릭 타임 정보가 지워짐)**

private static void instanceOf(List<String> list) {  
 if (list instanceof List<?>) {  
 System.*out*.println("비한정 와일드카드 적용");  
 }  
  
 if (list instanceof List) {  
 System.*out*.println("로 타입 적용");  
 }  
  
 // 컴파일 에러  
 if (list instanceof List<String>) {  
  
 }  
}

**🡺로 타입이든 비한정 와일드카드 이든 동일하게 동작한다.**

Set<Object>는 어떤 타입의 객체도 저장할 수 있는 매개변수화 타입이고, Set<?>는 모종의 타입 객체만 저장할 수 있는 와일드카드 타입이다. (안전함)

**ITEM27 비검사 경고를 제거하라**

public class Sample {  
 public static void main(String[] args) {  
 @SuppressWarnings("unchecked")  
 Set<String> set = new HashSet();  
 }

경고를 제거할 수는 없지만 타입이 안전하다고 확신할 수 있다면

**@SuppressWarnings(“unchecked”)** 애너테이션을 달아 경고를 숨기자

@Target({*TYPE*, *FIELD*, *METHOD*, *PARAMETER*, *CONSTRUCTOR*, *LOCAL\_VARIABLE*, *MODULE*})  
@Retention(RetentionPolicy.*SOURCE*)  
public @interface SuppressWarnings {String[] value();  
}

**@SuppressWarnings** 은 개별 지역변수 선언부터 클래스 전체까지 어떤 선언에도 달 수 있다.

하지만 가능한 좁은 범위에 적용하자

**@SuppressWarnings** 를 사용할 때면 그 경고를 무시해도 안전한 이유를 항상 주석으로 남겨야 한다.

**ITEM28 배열보다 리스트를 사용하라**

배열 vs 제네릭

**1) 배열은 공변(함께 변한다)이다. 제네릭은 불공변 이다.**

public class Sample {  
 public static void main(String[] args) {  
 // 공변성  
 A[] aArray = new A[10];  
 B[] bArray = new B[10];  
 aArray = bArray;  
  
 // 불공변성  
 List<A> aList = new ArrayList<>();  
 List<B> bList = new ArrayList<>();  
 aList = bList; // 에러남  
 }  
}  
  
class A {  
  
}  
  
class B extends A {  
  
}

배열은 타입이 다르면 런타임 시점에서 알 수 있지만, 리스트는 컴파일 시점에 확인할 수 있다.

Object[] objectArray = new Long[1];  
objectArray[0] = "타입이 다름";  
  
List<Object> ol = new ArrayList<Long>();  
ol.add("test");

**2) 배열은 실체화 된다.**

배열은 런타임에도 자신의 원소타입을 인지함. 제네릭은 런타임 시점에 소거

제네릭 배열을 만들지 못하게 막은 이유는 타입에 안전하지 않기 때문이다. (ClassCastException)이 발생할 수 있다.

E, List<E>, List<String> 같은 타입을 실체화 불가 타입이라고 한다. 런타임에 정보가 없다.

매개변수화 타입 가운데 실체화 될 수 있는 타입은 **List<?>, Map<?, ?>** 같은 비한정적

와일드카드 타입 뿐이다.

제네릭으로 변경

**1) 일반 형태**

public class Chooser {  
 private final Object[] choiceArray;  
  
 public Chooser(Collection choices) {  
 choiceArray = choices.toArray();  
 }  
  
 public Object choose() {  
 Random rnd = ThreadLocalRandom.*current*();  
 return choiceArray[rnd.nextInt(choiceArray.length)];  
 }  
}

🡺런타임 시점에 형변환 오류가 날 수 있다. 제네릭 타입으로 변경해 보자.

**2) 1차 제네릭 형태**

public class Chooser<T> {  
 private final T[] choiceArray;  
  
 // 형변환은 되지만 컴파일 경고(unchecked)가 뜬다.  
 public Chooser(Collection<T> choices) {  
 choiceArray = (T[]) choices.toArray();  
 }  
  
 public Object choose() {  
 Random rnd = ThreadLocalRandom.*current*();  
 return choiceArray[rnd.nextInt(choiceArray.length)];  
 }  
}

**3) 2차 비검사 형변환 경고를 제거하기 위한 배열 대신 리스트**

public class Chooser<T> {  
 private final List<T> choiceList;  
  
 public Chooser(Collection<T> choices) {  
 choiceList = new ArrayList<>(choices);  
 }  
  
 public T choose() {  
 Random rnd = ThreadLocalRandom.*current*();  
 return choiceList.get(rnd.nextInt(choiceList.size()));  
 }  
}

**ITEM29 이왕이면 제네릭 타입으로 만들라**

private E[] elements;  
  
public Stack() {  
 elements = new E[*DEFAULT\_INITIAL\_CAPACITY*];  
}

다음과 E와 같은 실체화 불가 타입으로는 배열을 만들 수 없다.

배열을 사용하는 코드를 제네릭으로 만들려 할때는 이 문제가 항상 발목을 잡을 것이다.

**해결책**

**1) 제네릭 배열 생성을 금지하는 제약을 대놓고 우회**

*/\*\*  
 \* private 필드에 저장되어 있고 다른 메서드에 전달되는 일이 전혀 없기 때문에 안전하다.  
 \*/*@SuppressWarnings("unchecked")  
public Stack() {  
 elements = (E[]) new Object[*DEFAULT\_INITIAL\_CAPACITY*];  
}

🡺장점으로는 가독성이 좋다. 초기 한번만 선언해 주면 된다.

🡺단점으로는 E가 Object가 아닌 한 배열의 런타임 타입이 컴파일타임 타입과 달라 힙 오염을 일으킨다.

**2) elements 필드의 타입을 E[]에서 Object[]로 변경**

private Object[] elements;  
  
public Stack() {  
 elements = new Object[*DEFAULT\_INITIAL\_CAPACITY*];  
}  
public E pop() {  
 if (size == 0)  
 throw new EmptyStackException();  
 */\*\*  
 \* push에서 E 타입만 허용하기 때문에 이 형변화는 안전하다.  
 \*/*  
 @SuppressWarnings("unchecked")  
 E result = (E) elements[--size];  
 elements[size] = null;  
 return result;  
}

🡺장점으로는 힙 오염이 없다.

🡺단점으로 원소를 읽을 때마다 형변환 해줘야 한다.

**bounded type parameter (<E extends Number)**

**vs bounded wildcard type (List<? extends Number>**

**-한정적 와일드 카드 타입**

와일드 카드는 단 하나의 경계를 가질 수 있다.

public void process(List<? extends A> elements) {  
  
}

상한과 하한을 가질 수 있다.

public void process(List<? extends A> elements) {  
  
}  
  
public void process2(List<? super A> elements) {  
  
}

**-한정적 타입 매개변수**

한정적 타입 매개변수는 여러 경계를 가질 수 있다. (multi)

**multi (클래스가 앞에 나와야 된다)**

public class D<E extends A & B> {  
}  
  
class A {  
}  
  
interface B {  
}

상한만 있다. (super 없음)

**-공통점 (한정적 와일드카드, 한정적 타입 매개변수)**

둘 다 API의 유연성이 향상된다.

**-결 론 (내가 생각하는)**

매개변수가 여러 개 필요할 경우에는 한정적 매개변수 사용하자

동일상황이면 와일드 카드를 사용하자 (명확하고 간결함)

**PECS(Producer Extends Consumer Super)**

**ITEM30 이왕이면 제네릭 메서드로 만들라**

*/\*\*  
 \* 제네릭 메서드  
 \* 타입 매개변수 목록은 메서드의 제한자와 반환 타입 사이에 온다.  
 \*/*public static <E> Set<E> union2(Set<E> s1, Set<E> s2) {  
 Set<E> result = new HashSet<>(s1);  
 result.addAll(s2);  
 return result;  
}

🡺제네릭은 런타임에 타입 정보가 소거되므로 하나의 객체를 어떤 타입으로든 매개변수화할 수 있다.

@SuppressWarnings("unchecked")  
public static <T> Comparator<T> reverseOrder() {  
 return (Comparator<T>) ReverseComparator.*REVERSE\_ORDER*;  
}

🡺제네릭 싱글턴 팩터리

**재귀적 타입 한정** (자기 자신이 들어간 표현식을 사용하여 타입 매개변수의 허용 범위를 한정할 수 있다.)

// 모든 타입 E는 자신과 비교할 수 있다.  
public static <E extends Comparable<E>> E max(Collection<E> c) throws IllegalAccessException {  
 if (c.isEmpty()) {  
 throw new IllegalAccessException("컬렉션이 비어 있습니다.");  
 }  
  
 E result = null;  
 for (E e : c) {  
 if (result == null || e.compareTo(result) > 0) {  
 result = Objects.*requireNonNull*(e);  
 }  
 }  
 return result;  
}

결론 : 명시적 형변환해야 하는 메서드가 있으면 제네릭 메서드 사용하자

**ITEM31 한정적 와일드카드를 사용해 API 유연성을 높이라**

유연성을 극대화하려면 원소의 생산자나 소비자용 입력 매개변수에 와일드카드 타입을 사용하라.

**펙스(PECS) : producer-extends, consumer-super**

매개변수화 타입 T가 생산자라면 <? extends T>를 사용하고, 소비자라면 <? super T>를 사용하라.

ex) Stack에서 pushAll의 매개변수는 Stack이 사용할 인스턴스를 생성하므로 producer,

popAll은 Stack으로부터 E 인스턴스를 소비하므로 consumer

**주의!)** 반환 타입에는 한정적 와일드카드 타입을 사용하면 안 된다. 유연성을 높여주기는커녕 클라이언트 코드에서도 와일드카드 타입을 써야 하기 때문이다.

public static <E> **Set<? extends E>** union(Set<? extends E> s1, Set<? extends E> s2) {  
 Set<E> result = new HashSet<>(s1);  
 result.addAll(s2);  
 return result;  
}

**매개변수(parameter)와 인수(argument)의 차이**

매개변수는 메서드 선언에 정의한 변수이고, 인수는 메서드 호출 시 넘기는 ‘실젯값’

**예)**

void add(int value) 🡺 parameter

add(10) 🡺argument

**도우미 메서드**

public static void swap2(List<?> list, int i, int j) {  
 *swapHelper*(list, i, j);  
}  
  
private static <E> void swapHelper(List<E> list, int i, int j) {  
 list.set(i, list.set(j, list.get(i)));  
}

**ITEM32 제네릭과 가변인수를 함께 쓸 때는 신중하라**

매개변수화 타입의 변수가 타입이 다른 객체를 참조하면 힙 오염이 발생한다.

static void dangerous(List<String>... stringLists) {  
 List<Integer> intList = List.*of*(42);  
 Object[] objects = stringLists;  
 objects[0] = intList; // 힙 오염 발생... List<String>에 List<Integer>를 넣었음  
 String s = stringLists[0].get(0); // ClassCastException  
}

**제네릭 varargs 배열 매개변수에 값을 저장하는 것은 안전하지 않다.**

해당 배열 생성은 허용하지 않지만 제네릭 varargs 매개변수를 받는 메서드는 선언 가능하다.

왜그럴까?

List<String>[] stringLists = new List<String>[1]; // 컴파일 오류  
List<Integer> intList = List.*of*(42);  
Object[] objects = stringLists;  
objects[0] = intList;  
String s = stringLists[0].get(0);

**답)** 매개변수화 타입의 varargs 매개변수르 ㄹ받는 메서드가 실무에서 매우 유용하기 때문에 이 모순을 수용하기로 했다.

**@SafeVarargs 애너테이션은 메서드 작성자가 그 메서드가 타입 안전함을 보장하는 장치다.**

컴파일 시점에 해당 경고가 뜨면 안전하지 않다. 안전할 때만 @SafeVarargs를 붙이자



**다음 두 조건을 모두 만족하는 제네릭 varargs 메서드는 안전하다.**

-varargs 매개변수 배열에 아무것도 저장하지 않는다.

-그 배열(혹은 복제본)을 신뢰할 수 없는 코드에 노출하지 않는다.

**제네릭 varargs 매개변수를 List로 대체**

static <T> List<T> flatten2(List<List<? extends T>> lists) {  
 List<T> result = new ArrayList<>();  
 for (List<? extends T> list : lists) {  
 result.addAll(list);  
 }  
 return result;  
}

**ITEM33 타입 안전 이종 컨테이너를 고려하라 (다양한 타입을 지원하기 위해)**

타입 안전 이종(heterogeneous) 컨테이너

*/\*\*  
 \* 타입 안전 이종 컨테이너 패턴  
 \*/*public class Favorites {  
 private Map<Class<?>, Object> favorites = new HashMap<>();  
  
 public <T> void putFavorite(Class<T> type, T instance) {  
 favorites.put(Objects.*requireNonNull*(type), instance);  
 }  
  
 public <T> T getFavorite(Class<T> type) {  
 return type.cast(favorites.get(type));  
 }  
  
 public static void main(String[] args) {  
 Favorites f = new Favorites();  
 f.putFavorite(String.class, "Java");  
 f.putFavorite(Integer.class, 0xcafebabe);  
 f.putFavorite(Class.class, Favorites.class);  
  
 String favoriteString = f.getFavorite(String.class);  
 int favoriteInteger = f.getFavorite(Integer.class);  
 Class<?> favoriteClass = f.getFavorite(Class.class);  
  
 System.*out*.printf("%s %x %s%n", favoriteString, favoriteInteger, favoriteClass.getName());  
 }  
}

**비한정 와일드카드 (에러)**

List<?> list = new ArrayList<>();  
list.add("verboten");

**비한정 와일드 카드 (정상)**

Map<Class<?>, Object> favorites = new HashMap<>();  
favorites.put(String.class, "test");

🡺와일드카드 타입이 중척(nested)되어 있기 때문에 맵이 아니라 키가 와일드카드 타입인 것이다.

**Favorites 클래스의 제약사항 두가지**

1) 악의적인 클라이언트가 Class 객체를 (제네릭이 아닌)로 타입으로 넘기는 경우

Favorites f = new Favorites();  
f.putFavorite((Class) Integer.class, "Intger의 인스턴스가 아닙니다.");  
int favoriteInteger2 = f.getFavorite(Integer.class);

🡺 ClassCastException이 발생한다.



🡺이런 경우 컴파일할 때 비검사 경고도 뜬다.



이를 해결하기 위해 put 할경우 type.cast(동적 형변환) 를 통해 검사를 한다. (하지만 타입이 다르면 에러남)

public <T> void putFavorite(Class<T> type, T instance) {  
 favorites.put(Objects.*requireNonNull*(type), type.cast(instance));  
}

ex) Collections의 checkedSet, checkedList, checkedMap 등

public static <E> List<E> checkedList(List<E> list, Class<E> type) {  
 return (list instanceof RandomAccess ?  
 new CheckedRandomAccessList<>(list, type) :  
 new CheckedList<>(list, type));  
}

2) 실체화 불가 타입에서는 사용 불가

String, String[] 이 아닌 List<String> 같은 형태(실체화 불가)

🡺슈퍼타입 토큰 사용 (하지만 완벽한 것은 아니다)

🡺Spring에서는 ParameterizedTypeReference 사용

때로는 메서드들이 허용하는 타입을 제한하고 싶을 때, 한정적 타입 토큰을 활용하면 된다.

**asSubClass**를 사용해 한정적 타입 토큰을 안전하게 형변환한다.

static Annotation getAnnotation(AnnotatedElement element, String annotationTypeName) {  
 Class<?> annotationType = null; // 비한정적 타입 토큰  
  
 try {  
 annotationType = Class.*forName*(annotationTypeName);  
 } catch (Exception ex) {  
 throw new IllegalArgumentException(ex);  
 }  
 return element.getAnnotation(annotationType.asSubclass(Annotation.class);  
}

**용어**

탑입 안전 이종 컨테이너는 Class를 키로 쓰며, 이런식으로 쓰이는 Class 객체를 **타입 토큰** 이라고 한다.

**TypeToken과 ParameterizedTypeReference 차이**

<http://blog.woniper.net/320?category=506090>

**CHAPTER6**

**ITEM34 int 상수 대신 열거 타입을 사용하라**

**정수 열거 패턴 (상당이 취약함)**

public static final int *APPLE\_FUJI* = 0;  
public static final int *APPLE\_PIPPIN* = 1;  
public static final int *APPLE\_GRANNY\_SMITH* = 2;  
  
public static final int *ORANGE\_NAVEL* = 0;  
public static final int *ORANGE\_TEMPLE* = 1;  
public static final int *ORANGE\_BLOOD* = 2;

🡺별도의 접두어 (APPLE\_, ORANGE\_)를 써서 이름 충돌 방지하는 것이다.

**단점들**

1) ELEMENT\_MERCURY, PLANET\_MERCURY 접두어를 사용하여 구분

2) 상수의 값이 변경되면 클라이언트도 반드시 다시 컴파일 (가장 큰 단점 인 듯)

3) 로그나 디버깅 할 때 숫자로 보여서 의미파악하기 힘듬 (표현력 좋지 않음)

4) 상수가 몇 개인지도 알 수 없음.

**enum은 간단하게 length 이용**

public class Sample {  
 public static void main(String[] args) {  
 int i = (*APPLE\_FUJI* - *ORANGE\_TEMPLE*) / *APPLE\_PIPPIN*;  
  
 System.*out*.println(Apple.*values*().length);  
 }  
}  
  
enum Apple {*FUJI*, *PIPPIN*, *GRANNY\_SMITH*}

5) 문자열 열거 패턴은 상수 이름 대신 문자열 값을 그대로 하드코딩 (오타가 있어도 컴파일러는 확인 못함)

**C# vs Java enum 비교**

**C#**

class Program  
{  
 static void Main(string[] args)  
 {  
 Console.WriteLine(Apple.*FUJI*); // FUJI

Console.WriteLine((int)Apple.*FUJI*); // 1  
 }  
}  
  
enum Apple { *FUJI* = 1, PIPPIN = 2, GRANY\_SMITH = 3 }

**Java**

public class Sample {  
 public static void main(String[] args) {  
 System.*out*.println(Apple.*FUJI*); // FUJI  
 System.*out*.println(Apple.*FUJI*.getValue()); // 1  
 }  
}  
  
enum Apple {  
 *FUJI*(1), *PIPPIN*(2), *GRANNY\_SMITH*(3);  
 Apple(int value) {  
 this.value = value;  
 }

private final int value;  
  
 public int getValue() {  
 return value;  
 }  
}

**Enum을 컴파일 하면???**

**-디컴파일 Decompiler (IntelliJ class) :** class파일을 java파일로 다시 변환 (고급언어)

**-디스어셈블(javap) :** class 파일을 사람이 읽을 수 있게 변환 (상수, 함수 간단 목록)

**-디스어셈블-bytecode(javap –c) :** bytecode의 모든 명령어 표시

**Enum Orange**

enum Orange {*NAVEL*, *TEMPLE*, *BLOOD*}

**javap Orange**

final class chapter6.item34.Orange extends java.lang.Enum<chapter6.item34.Orange>{  
 public static final chapter6.item34.Orange NAVEL;  
 public static final chapter6.item34.Orange TEMPLE;  
 public static final chapter6.item34.Orange BLOOD;  
 public static chapter6.item34.Orange[]values();  
 public static chapter6.item34.Orange valueOf(java.lang.String);  
 static {};  
}

**javap –c Orange (bytecode)**

final class chapter6.item34.Orange extends java.lang.Enum<chapter6.item34.Orange> {  
 public static final chapter6.item34.Orange NAVEL;  
 public static final chapter6.item34.Orange TEMPLE;  
 public static final chapter6.item34.Orange BLOOD;  
 public static chapter6.item34.Orange[] values();  
 Code:  
 0: getstatic #1 // Field $VALUES:[Lchapter6/item34/Orange;  
 3: invokevirtual #2 // Method "[Lchapter6/item34/Orange;".clone:()Ljava/lang/Object;  
 6: checkcast #3 // class "[Lchapter6/item34/Orange;"  
 9: areturn  
  
 public static chapter6.item34.Orange valueOf(java.lang.String);  
 Code:  
 0: ldc #4 // class chapter6/item34/Orange  
 2: aload\_0  
 3: invokestatic #5 // Method java/lang/Enum.valueOf:(Ljava/lang/Class;Ljava/lang/String;)Ljava/lang/Enum;  
 6: checkcast #4 // class chapter6/item34/Orange  
 9: areturn

. . .생략

**열거 타입 특징**

**1) 타입 안정성**

public static void main(String[] args) {  
 Sample sample = new Sample();  
 sample.enumTest(Apple.*PIPPIN*);  
}

*/\*\*  
 \* 타입 안정성  
 \*/*private void enumTest(Apple apple) {

. . .  
}

**2) 같은 이름 공존**

Apple apple = Apple.*PIPPIN*;  
Orange orange = Orange.*PIPPIN*;

**3) 컴파일 필요 없음**

**정수 열거 패턴**

public class Foo {  
 public static final int *SOMETHING* = 100;  
}  
  
public class Bar {  
 public static void main(String[] args) {  
 System.*out*.println(Foo.*SOMETHING*);  
 }  
}

🡺이 상태에서 Foo의 SOMETHING 값을 **200**으로 변경 후, Foo만 재빌드하고 Bar를 실행시키면?



🡺이미 Bar에 BIPUSH로 값이 박혀 있기 때문에 값이 변경 안 된다.

**enum**

public enum Foo {  
 *FUJI*(1), *PIPPIN*(2), *GRANNY\_SMITH*(3);  
  
 private final int value;  
  
 Foo(int value) {  
 this.value = value;  
 }  
  
 public int getValue() {  
 return value;  
 }  
}

public class Bar {  
 public static void main(String[] args) {  
 System.*out*.println(Foo.*FUJI*.getValue());  
 }  
}

🡺이 상태에서 Foo의 FUJI 값을 **100**으로 변경 후, Foo만 재빌드하고 Bar를 실행시키면? 100출력

🡺이 상태에서 Foo의 enum을 하나더 추가 시키면? **length 4개가 됨.**



🡺Bar의 컴파일 없이 변경된 Foo의 값을 가지고 옴

**4) toString**

@Override  
public String toString() {  
 return super.toString() + ":" + String.*valueOf*(value);  
}

**enum 메서드, 필드**

public enum Planet {  
 *MERCURY*(3.302e+23, 2.439e6),  
 *VENUS*(4.869e+24, 6.052e6),  
 *EARTH*(5.975e+24, 6.378e6),

. . .  
  
 private final double mass; // 질량  
 private final double radius; // 반지름  
 private final double surfaceGravity; // 표면중력  
  
 private static final double *G* = 6.67300E-11;  
  
 Planet(double mass, double radius) {  
 this.mass = mass;  
 this.radius = radius;  
 surfaceGravity = *G* \* mass / (radius \* radius);  
 }  
  
 public double mass() {  
 return mass;  
 }  
  
 public double radius() {  
 return radius;  
 }  
  
 public double surfaceGravity() {  
 return surfaceGravity;  
 }  
}

**열거 타입에서 상수를 하나 제거하면?**

🡺기존 클라이언트에서 해당 상수를 참조하지 않으면 영향이 없다.

🡺참조하는 경우는 클라이언트에서 다시 컴파일 하면 컴파일 오류가 발생한다.



컴파일 하지 않아도, 제거 된, 상수를 사용 할 때 런타임 에러가 발생한다.

****

**널리 쓰이는 열거 타입은 톱레벨 클래스로, 톱레벨 클래스에서만 쓰인다면 멤버 클래스로**

**-enum의 좀 더 유용한 기능들…**

**switch문을 이용하여 상수에 따라 분기**

public enum Operation {  
 *PLUS*, *MINUS*, *TIMES*, *DIVIDE*;  
  
 public double apply(double x, double y) {  
 switch (this) {  
 case *PLUS*:  
 return x + y;  
 case *MINUS*:  
 return x - y;  
 case *TIMES*:  
 return x \* y;  
 case *DIVIDE*:  
 return x / y;  
 }  
 throw new AssertionError("알 수 없는 연산: " + this);  
 }  
  
 public static void main(String[] args) {  
 System.*out*.println(Operation.*PLUS*.apply(3, 4));  
 }  
}

단점 :

🡺불필요한 throw (정상적인 경우, 도달하지 않지만 컴파일 때문에 추가)

🡺상수 추가되면 case도 변경해야 된다. (깨지기 쉬움)

**상수별 메서드 구현을 활용**

public enum Operation2 {  
 *PLUS* {public double apply(double x, double y) {return x + y;}},  
 *MINUS* {public double apply(double x, double y) {return x - y;}},  
 *TIMES* {public double apply(double x, double y) {return x \* y;}},  
 *DIVIDE* {public double apply(double x, double y) {return x / y;}};  
  
 public abstract double apply(double x, double y);  
}

apply가 추상 메서드 이므로 재정의하지 않으면 컴파일 때 오류

final class chapter6.item34.Orange extends java.lang.Enum<chapter6.item34.Orange>{  
 public static final chapter6.item34.Orange NAVEL;  
 public static final chapter6.item34.Orange TEMPLE;  
 public static final chapter6.item34.Orange BLOOD;  
 public static chapter6.item34.Orange[]values();  
 public static chapter6.item34.Orange valueOf(java.lang.String);  
 static {};  
}

🡺values, valueOf 가 기본 생성

**특정 필드에 따른 enum 반환**

private static final Map<String, Operation4> *stringToEnum* =  
 Stream.*of*(*values*()).collect(  
 *toMap*(Object::toString, e -> e));  
  
// 지정한 symbol에 해당하는 Operation이 존재하면 반환  
public static Optional<Operation4> fromString(String symbol) {  
 return Optional.*ofNullable*(*stringToEnum*.get(symbol));  
}

**ex) 프로젝트에서 사용하고 있는 코드**

public static ErrorMessages fromErrorCode(String errorCode) {  
 List<ErrorMessages> list = Arrays.*asList*(ErrorMessages.*values*());  
 return list.stream().filter(m -> m.errorCode.equals(errorCode)).findAny().orElse(null);  
}

**주의점**

enum의 생성자에서 자신의 인스턴스를 맵에 추가하려고 하면

Operation(String symbol) {  
 this.symbol = symbol;  
 Map<String, Operation4> stringToEnum = Stream.*of*(*values*()).collect(  
 *toMap*(Object::toString, e -> e));  
}

🡺NullPointerException이 발생한다. (enum의 생성자는 오직 상수 변수만 접근할 수 있다.)

****

상수별 메서드 구현에는 열거 타입 상수끼리 코드를 공유하기 어렵다.

**값에 따라 분기하여 코드를 공유하는 열거 타입 (별로 좋지 않다.)**

public enum PayrollDay {  
 *MONDAY*, *TUESDAY*, *WEDNESDAY*, *THURSDAY*, *FRIDAY*, *SATURDAY*, *SUNDAY*;  
  
 private static final int *MINS\_PER\_SHIFT* = 8 \* 60;  
  
 int pay(int minutesWorked, int payRate) {  
 int basePay = minutesWorked \* payRate;  
  
 int overtimePay;  
 switch (this) {  
 case *SATURDAY*:  
 case *SUNDAY*:  
 overtimePay = basePay / 2;  
 break;  
 default:  
 overtimePay = minutesWorked <= *MINS\_PER\_SHIFT* ?  
 0 : (minutesWorked - *MINS\_PER\_SHIFT*) \* payRate / 2;  
 }  
 return basePay + overtimePay;  
 }  
}

🡺휴가와 같은 새로운 값을 열거 타입에 추가하려면 그 값을 처리하는 case문을 잊지 말고 쌍으로 넣어줘야 한다. (앞에 switch를 이용한 분기랑 비슷한 문제점 발생)

전략 열거 타입 패턴을 사용하면 swtich문 보다 복잡하지만 더 안전하고 유연하다.

**열거 타입을 언제 쓰면 될까?**

🡺필요한 원소를 컴파일타임에 다 알 수 있는 상수 집합이라면 항상 열거 타입을 사용하자.

**ITEM35 ordinal 메서드 대신 인스턴스 필드를 사용하라**

열거 타입 상수는 자연스럽게 하나의 정수값에 대응된다. 몇 번째 위치인지 반환하는 ordinal이라는 메서드 제공

*/\*\*  
 \* ordinal을 잘못 사용한 예  
 \*/*

public enum Ensemble {  
 *SOLO*, *DUET*, *TRIO*, *QUARTET*, *QUINTET*, *SEXTET*, *SEPTET*, *OCTET*, *NONET*, *DECTET*;  
  
 public int numberOfMusicians() {  
 return ordinal() + 1;  
 }  
}

🡺동작은 하지만, 상수 선언 순서를 바꾸는 순간 numberOfMusicians가 오동작하며, 이미 사용중인 정수와 값이 같은 상수는 추가할 방법이 없다.

*/\*\*  
 \* 열거 타입 상수에 연결된 값은 ordinal 메서드로 얻지 말고, 인스턴스 필드에 저장하자  
 \*/*

public enum Ensemble {  
 *SOLO*(1), *DUET*(2), *TRIO*(3), *QUARTET*(4), *QUINTET*(5), *SEXTET*(6), *SEPTET*(7), *OCTET*(8),  
 *DOUBLE\_QUARTER*(8), *NONET*(9), *DECTET*(10), *TRIPLE\_QUARTET*(12);  
  
 private final int numberOfMusicians;  
  
 Ensemble(int size) {  
 this.numberOfMusicians = size;  
 }  
  
 public int numberOfMusicians() {  
 return numberOfMusicians;  
 }  
}

🡺인스턴스 필드를 사용하면 값이 같은 정수를 사용할 수 있고, 중간에 값을 비워둘 수도 있다.

**결론 : ordinal 메서드는 실용성이 떨어진다. 사용하지 말자.**

**ITEM36 비트 필드 대신 EnumSet을 사용하라**

**비트 필드 열거 상수 (구닥다리)**

public class Text {  
 public static final int *STYLE\_BOLD* = 1 << 0; // 1  
 public static final int *STYLE\_ITALIC* = 1 << 1; // 2  
 public static final int *STYLE\_UNDERLINE* = 1 << 2; // 4  
 public static final int *STYLE\_STRIKETHROUGH* = 1 << 3; // 8  
  
 public void applyStyles(int styles) {  
 System.*out*.println(styles);  
 }  
  
 public static void main(String[] args) {  
 Text text = new Text();  
 int result = *STYLE\_BOLD* | *STYLE\_ITALIC* | *STYLE\_UNDERLINE* | *STYLE\_STRIKETHROUGH*;  
 text.applyStyles(result);  
 }  
}

**장점** : 비트 연산을 사용해 합집합과 교집합 같은 집합 연산을 효율적으로 수행할 수 있다.

**단점**

1) 기존 정수 열거 상수의 단점을 그대로 지니고 있음

2) 해석하기 어렵다. (값이 섞이기 때문에)

3) 원소를 순회하기도 까다롭다.

4) 최대 비트 수에 따른 타입을 선택해야 한다. (int, long)

**EnumSet**

public class Text {  
 public enum Style {*BOLD*, *ITALIC*, *UNDERLINE*, *STETHROUGH*}  
  
 // 어떤 Set을 넘겨도 되나, EnumSet이 가장 좋다.  
 public void applyStyles(Set<Style> styles) {  
 System.*out*.printf("Applying styles %s to text%n",  
 Objects.*requireNonNull*(styles));  
 }

public static void main(String[] args) {  
 Text text = new Text();  
 text.applyStyles(EnumSet.*of*(Style.*BOLD*, Style.*ITALIC*));  
 }  
}

🡺EnumSet을 사용함녀 짧고 깔끔하고 안전하다.

**참고**

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);  
}

EnumSet 내부적으로 원소가 64개 이하면 **RegularEnumSet** (long),

65개 이상이면 **JumboEnumSet** (long[]) 사용. (ITEM1)

/// <summary>  
/// bidding status flag example   
/// </summary>

[Flags]  
public enum Status  
{  
 *None* = 0,  
 Live = 1,  
 ScheduleOver = 2,  
 DailyBudgetOver = 4,  
 TotalBudgetOver = 8,  
 DailyImpOver = 16,  
 TotalImpOver = 32,  
 DailyClickOver = 64,  
 TotalClickOver = 128,  
 DailyConvOver = 256,  
 TotalConvOver = 512,  
 Deleted = 1024,  
}

**ITEM37 ordinal 인덱싱 대신 EnumMap을 사용하라**

**배열 인덱스로 사용**

// 코드 37-1 ordinal()을 배열 인덱스로 사용 - 따라 하지 말 것!  
Set<Plant>[] plantsByLifeCycleArr =  
 (Set<Plant>[]) new Set[Plant.LifeCycle.*values*().length];  
for (int i = 0; i < plantsByLifeCycleArr.length; i++) {  
 plantsByLifeCycleArr[i] = new HashSet<>();  
}  
  
for (Plant p : garden) {  
 plantsByLifeCycleArr[p.lifeCycle.ordinal()].add(p);  
}  
  
// 결과 출력  
for (int i = 0; i < plantsByLifeCycleArr.length; i++) {  
 System.*out*.printf("%s: %s%n",  
 Plant.LifeCycle.*values*()[i], plantsByLifeCycleArr[i]);  
}

**단점**

1) 배열은 제네릭과 호환되지 않으니 비검사 형변환을 수행

2) 각 인덱스의 의미를 모름

3) 잘못된 값을 사용하면 잘못된 로직 수행

**EnumMap 사용**

// EnumMap을 사용해 데이터와 열거 타입을 매핑한다.  
Map<LifeCycle, Set<Plant>> plantsByLifeCycle = new EnumMap<>(Plant.LifeCycle.class);  
  
for (Plant.LifeCycle lc : Plant.LifeCycle.*values*()) {  
 plantsByLifeCycle.put(lc, new HashSet<>());  
}  
  
for (Plant p : garden) {  
 plantsByLifeCycle.get(p.lifeCycle).add(p);  
}  
System.*out*.println(plantsByLifeCycle);

**스트림 사용 (Enum 사용 X)**

// 스트림을 사용한 코드 (EnumMap을 사용하지 않는다.)  
System.*out*.println(Arrays.*stream*(garden)  
 .collect(Collectors.*groupingBy*(p -> p.lifeCycle)));

**스트림 사용 (Enum 사용 O)**

// 스트림을 사용한 코드 (EnumMap을 사용한다.)  
System.*out*.println(Arrays.*stream*(garden)  
 .collect(Collectors.*groupingBy*(p -> p.lifeCycle,  
 () -> new EnumMap<>(LifeCycle.class), Collectors.*toSet*())));

**ITEM38 확장할 수 있는 열거 타입이 필요하면 인터페이스를 사용하라**

타입 안전 열거 패턴은 확장할 수 있으나 열거 타입은 그럴 수 없다. (열거 패턴이 좋은 예외)

**인터페이스를 이용해 확장 가능 열거 타입을 흉내**

public interface Operation {  
 double apply(double x, double y);  
}

public enum BasicOperation implements Operation {  
 *PLUS*("+") {  
 public double apply(double x, double y) {  
 return x + y;  
 }  
 },  
 *MINUS*("-") {  
 public double apply(double x, double y) {  
 return x - y;  
 }  
 },  
 *TIMES*("\*") {  
 public double apply(double x, double y) {  
 return x \* y;  
 }  
 },  
 *DIVIDE*("/") {  
 public double apply(double x, double y) {  
 return x / y;  
 }  
 };

}

**확장 가능 열거 타입 (동일하게 Operation을 구현하고 있음)**

public enum ExtendedOperation implements Operation {  
 *EXP*("^") {  
 @Override  
 public double apply(double x, double y) {  
 return 0;  
 }  
 },  
 *REMAINDER*("%") {  
 @Override  
 public double apply(double x, double y) {  
 return 0;  
 }  
 };

}

🡺상수별 메서드 구현과 다른점은 apply가 인터페이스에 선언되어 있으니 열거 타입에 따로 추상 메서드로 선언하지 않아도 된다.

**사용법**

public static void main(String[] args) {  
 double x = 4;  
 double y = 2;  
  
 *test*(ExtendedOperation.class, x, y);  
 *test*(Arrays.*asList*(ExtendedOperation.*values*()), x, y);  
}  
  
*/\*\*  
 \* case1) 한정적 타입 토큰 역할  
 \*/*private static <T extends Enum<T> & Operation> void test(Class<T> opEnumType, double x, double y) {  
 for (Operation op : opEnumType.getEnumConstants()) {  
 System.*out*.printf("%f %s %f = %f%n", x, op, y, op.apply(x, y));  
 }  
}  
  
*/\*\*  
 \* case2) 한정적 와일드카드 타입  
 \*/*private static void test(Collection<? extends Operation> opSet, double x, double y) {  
 for (Operation op : opSet) {  
 System.*out*.printf("%f %s %f = %f%n", x, op, y, op.apply(x, y));  
 }  
}

**이 방식의 사소 한문제점**

🡺열거 타입끼리 구현을 상속할 수 없다.

🡺공유하는 기능이 많다면, 그 부분을 별도의 도우미 클래스나 정적 도우미 메서드로 분리

**도우미 클래스?**

utility class와 helper class는 비슷하다고 생각할 수도 있다.

하지만 utility class는 모든 메소드가 정적 메소드이지만, 일반적으로 helper class는

모든 메소드가 정적 메소드이지 않으며, 여러 개의 helper class의 인스턴스가 있을 수 있다.

또한, helper class는 private로 선언하여 다른 곳에서의 접근을 막도록 하는 것이 좋다.

이는 특정 클래스를 도와주기 위해 helper class를 만든 것이지, 외부에서 접근하여 사용할 목적으로 만든 것이 아니기 때문이다.

**ITEM39 명명 패턴보다 애너테이션을 사용하라**

**명명패턴 단점**

1) 오타가 나면 안된다. (컴파일로 구분 안됨)

ex) tsetSafety

2) 올바른 프로그램 요소에서만 사용되리라 보증할 방법이 없다.

ex) TestSafetyMechanisms.class

3) 프로그램 요소를 매개변수로 전달할 마땅한 방법이 없다.

**마커 애너테이션 타입 선언**

*/\*\*  
 \* 테스트 메서드임을 선언하는 애너테이션  
 \* 매개변수 없는 정적 메서드 전용   
 \*/*@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
public @interface Test {  
}

@Retention은 @Test가 런타임에도 유지되어야 한다는 표시다.

@Target은 @test가 반드시 메서드 선언에서만 사용돼야 한다고 알려준다.

**배열 매개변수를 받는 애너테이션 타입**

@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
public @interface ExceptionTest {  
 Class<? extends Throwable>[] value();  
}

**반복 가능한 애너테이션 타입 (Java8 부터)**

*/\*\*  
 \* 컨테이너 애너테이션  
 \*/*@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
public @interface ExceptionTestContainer {  
 ExceptionTest[] value();  
}

@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
@Repeatable(ExceptionTestContainer.class)  
public @interface ExceptionTest {  
 Class<? extends Throwable> value();  
}

@ExceptionTest(IndexOutOfBoundsException.class)  
@ExceptionTest(NullPointerException.class)  
public static void doublyBad() {  
 List<String> list = new ArrayList<>();  
 list.addAll(5, null);  
}

애너테이션으로 할 수 있는 일을 명명 패턴으로 처리할 이유는 없다.

**ITEM40 @Override 애너테이션을 일관되게 사용하라**

@Override 애너테이션을 사용하면, 컴파일 시점에 잘못된 선언을 찾을 수 있다.

상위 클래스의 메서드를 재정의하려는 모든 메서드에 @Override 애너테이션을 달자.

**ITEM41 정의하려는 것이 타입이라면 마커 인터페이스를 사용하라**

**마커 인터페이스는 마커 애너테이션보다 두가지 면에서 낫다.**

1)마커 인터페이스는 이를 구현한 클래스의 인스턴스들을 구분하는 타입으로 쓸 수 있으나, 마커 애너테이션은 그렇지 않다.

2)적용 대상을 더 정밀하게 지정할 수 있다.

**마커 애너테이션이 마커 인터페이스보다 나은 점으로는?**

거대한 애너테이션 시스템의 지원을 받는다. (프레임워크)

**그러면 어떨 때 뭘 써야 되나?**

- 클래스와 인터페이스 외의 프로그램 요소(모듈, 패키지, 필드, 지역변수 등)

- 클래스나 인터페이스에 적용 해야된다면? 매개변수로 받는 메서드를 작성할 일이 있나 여부에 따라서 결정( 있으면 인터페이스 없으면 애노테이션)

- 매개변수로 받는 경우가 있어도 프레임워크에서 사용하려는 마커라면 애너테이션이 낫다.

**Colelction.java**

public interface Collection<E> extends Iterable<E> {int size();boolean isEmpty();boolean contains(Object o);Iterator<E> iterator();Object[] toArray();<T> T[] toArray(T[] a);boolean add(E e);boolean remove(Object o);boolean containsAll(Collection<?> c);boolean addAll(Collection<? extends E> c);boolean removeAll(Collection<?> c);default boolean removeIf(Predicate<? super E> filter) {  
 Objects.*requireNonNull*(filter);  
 boolean removed = false;  
 final Iterator<E> each = iterator();  
 while (each.hasNext()) {  
 if (filter.test(each.next())) {  
 each.remove();  
 removed = true;  
 }  
 }  
 return removed;  
 }boolean retainAll(Collection<?> c);void clear();boolean equals(Object o);int hashCode();  
@Override  
 default Spliterator<E> spliterator() {  
 return Spliterators.*spliterator*(this, 0);  
 }default Stream<E> stream() {  
 return StreamSupport.*stream*(spliterator(), false);  
 }default Stream<E> parallelStream() {  
 return StreamSupport.*stream*(spliterator(), true);  
 }  
}

**Set.java**

public interface Set<E> extends Collection<E> {int size();boolean isEmpty();boolean contains(Object o);Iterator<E> iterator();Object[] toArray();<T> T[] toArray(T[] a);boolean add(E e);boolean remove(Object o);boolean containsAll(Collection<?> c);boolean addAll(Collection<? extends E> c);boolean retainAll(Collection<?> c);boolean removeAll(Collection<?> c);void clear();boolean equals(Object o);int hashCode();@Override  
 default Spliterator<E> spliterator() {  
 return Spliterators.*spliterator*(this, Spliterator.*DISTINCT*);  
 }  
  
 */\*\*  
 \** ***@since*** *9  
 \*/* static <E> Set<E> of() {  
 return ImmutableCollections.Set0.*instance*();  
 }static <E> Set<E> of(E e1) {  
 return new ImmutableCollections.Set1<>(e1);  
 }static <E> Set<E> of(E e1, E e2) {  
 return new ImmutableCollections.Set2<>(e1, e2);  
 }static <E> Set<E> of(E e1, E e2, E e3) {  
 return new ImmutableCollections.SetN<>(e1, e2, e3);  
 }static <E> Set<E> of(E e1, E e2, E e3, E e4) {  
 return new ImmutableCollections.SetN<>(e1, e2, e3, e4);  
 }static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5) {  
 return new ImmutableCollections.SetN<>(e1, e2, e3, e4, e5);  
 }static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6) {  
 return new ImmutableCollections.SetN<>(e1, e2, e3, e4, e5,  
 e6);  
 }static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7) {  
 return new ImmutableCollections.SetN<>(e1, e2, e3, e4, e5,  
 e6, e7);  
 }static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8) {  
 return new ImmutableCollections.SetN<>(e1, e2, e3, e4, e5,  
 e6, e7, e8);  
 }

static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9) {  
 return new ImmutableCollections.SetN<>(e1, e2, e3, e4, e5,  
 e6, e7, e8, e9);  
 }static <E> Set<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10) {  
 return new ImmutableCollections.SetN<>(e1, e2, e3, e4, e5,  
 e6, e7, e8, e9, e10);  
 }@SafeVarargs  
 @SuppressWarnings("varargs")  
 static <E> Set<E> of(E... elements) {  
 switch (elements.length) { // implicit null check of elements  
 case 0:  
 return ImmutableCollections.Set0.*instance*();  
 case 1:  
 return new ImmutableCollections.Set1<>(elements[0]);  
 case 2:  
 return new ImmutableCollections.Set2<>(elements[0], elements[1]);  
 default:  
 return new ImmutableCollections.SetN<>(elements);  
 }  
 }  
}

**CHAPTER7**

**ITEM42 익명 클래스보다는 람다를 사용하라**

List<String> words = Arrays.*asList*("틱", "택", "톡");  
  
// 익명클래스로 처리한 낡은 기법  
Collections.*sort*(words,  
 new Comparator<String>() {  
 @Override  
 public int compare(String s1, String s2) {  
 return Integer.*compare*(s1.length(), s2.length());  
 }  
 });  
// 람다식을 함수 객체로 사용 (익명 클래스 대체)  
Comparator<String> comparator = (String s1, String s2) -> Integer.*compare*(s1.length(), s2.length());  
Collections.*sort*(words, comparator);  
Collections.*sort*(words,  
 (s1, s2) -> Integer.*compare*(s1.length(), s2.length()));  
  
// 람다 코드 좀더 간결하게  
Collections.*sort*(words, Comparator.*comparingInt*(String::length));  
  
// List의 sort  
words.sort(Comparator.*comparingInt*(String::length));

타입을 명시해야 코드가 더 명확할 때만 제외하고는, 람다의 모든 매개변수 타입은 생략하자

람다는 이름이 없고 문서화도 못 한다. 따라서 코드 자체로 동작이 명확히 설명되지 않거나 코드 줄 수가 많아지면 람다를 쓰지 말자 (세 줄 안에 끝내는게 좋다)

**익명 클래스를 람다로 대체할 수 없는 부분**

-추상 클래스의 인스턴스를 만들 때

-추상 메서드가 여러 개인 인터페이스의 인스턴스를 만들 때 (익명 클래스로)

-람다의 this는 바깥 인스턴스, 익명 클래스의 this는 인스턴스 자신이기 때문에

함수 객체가 자신을 참조해야 할 경우는 익명 클래스

람다를 직렬화하는 일은 삼가자(가상머신 별로 다를 수 있다.)

**ITEM43 람다보다는 메서드 참조를 사용하라**

보통 람다보다는 메서드 참조를 사용하는 것이 좋지만, 람다가 더 짧은 경우는 람다를 사용하자.

ex)

**메서드 참조**

service.execute(GoshThisClassNAmeIsHumongous::action);

**람다**

service.execute(() -> action());

**메서드 참조 유형**

1) 정적 메서드

2) 수신객체를 특정하는 한정적 인스턴스

🡺함수 객체가 받는 인수와 참조되는 메서드가 받는 인수가 똑같다.

3) 수신객체를 특정하지 않는 비한정적 인스턴스

🡺함수 객체를 적용하는 시점에 수신 객체를 알려준다.

이를 위해 수신 객체 전달용 매개변수가 매개변수 목록의 첫 번째로 추가되며, 그 뒤로는 참조되는 메서드 선언에 정으된 매개변수들이 뒤따른다.

4) 클래스 생성자

5) 배열 생성자

람다로 할 수 없는 일이라면 메서드 참조로도 할 수 없다.

**예외는 제네릭 함수 구현이다.**

**ITEM44 표준 함수형 인터페이스를 사용하라**

필요한 용도에 맞는게 있다면, 직접 구현하지 말고 표준 함수형 인터페이스를 활용하라.

(java.util.function)

|  |  |
| --- | --- |
| 인터페이스 | 함수 시그니처 |
| UnaryOperator<T> | T apply(T t) |
| Binary Operator<T> | T apply(T t1, T t2) |
| Predicate<T> | boolean test(T t) |
| Function<T, R> | R apply(T t) |
| Supplier<T> | T get() |
| Consumer<T> | void accept(T t) |

기본 함수형 인터페이스 중 3개는 인수를 2개씩 받는 변형이 있다.

BiPredicate<T, U> / boolean test(T t, U u)

BiFunction<T,U,R> / R apply(T t, U u)

BiConsumer<T, U> / void accept(T t, U u)

// UnaryOperator  
UnaryOperator<String> operator = u -> u.toUpperCase(); // return  
System.*out*.println(operator.apply("java2s.com"));  
  
// BinaryOperator  
BinaryOperator<Integer> adder = (n1, n2) -> n1 + n2; // return  
System.*out*.println(adder.apply(3, 4));  
  
// Predicate  
Predicate<String> p = s -> s.length() > 5; // return  
System.*out*.println(p.test("java2s.com "));  
  
// Function  
Function<Integer, String> converter = f -> Integer.*toString*(f); // return  
System.*out*.println(converter.apply(3).length());  
System.*out*.println(converter.apply(30).length());  
  
// Supplier  
Supplier<String> i = () -> "java2s.com"; // return  
System.*out*.println(i.get());  
  
// Consumer  
Consumer<String> consumer = c -> System.*out*.println(c.toLowerCase());  
consumer.accept("Java2s.com");

**기본 함수형 인터페이스에 박싱된 기본 타입을 넣어서 사용하지 말자**

🡺성능이 처참히 느려질 수 있음

**다음중 하나 이상을 만족하면 커스텀 함수형 인터페이스의 구현을 고민해보자**

1) 자주 쓰이며, 이름 자체가 용도를 명확히 설명해준다.

2) 반드시 따라야 하는 규약이 있다.

3) 유용한 디폴트 메서드를 제공할 수 있다.

**ITEM45 스트림은 주의해서 사용하라**

**computeIfAbsent**

이 메서드는 맵 안에 키가 있는지 찾은 다음, 있으면 단순히 그 키에 매핑된 값을 반환한다.

키가 없으면 건네진 함수 객체를 키에 적용하여 값을 계산해낸 다음 그 키와 값을 매핑해 놓고 계산된 값을 반환한다.

**함수 객체(람다, 메서드 참조)로는 못하고 코드 블록으로는 할 수 있는 일들**

-코드 블록에서는 범위 안의 지역변수를 읽고 수정할 수 있다. 하지만 람다에서는 final이거나 사실상 final인 변수만 읽을 수 있고, 지역변수를 수정하는 건 불가능하다.

-코드 블록에서는 return 문을 사용해 메서드에서 빠져나가거나, break나 continue 문으로 블록 바깥의 반복문을 종료하거나 반복을 한 번 건너 띌 수 있다.

또한 메서드 선언에 명시된 검사 예외를 던질 수 있다.

**스트림과 안성맞춤**

-원소들의 시퀀스를 일관되게 변환

-원소들의 시퀀스를 필터링한다.

-원소들의 시퀀스를 하나의 연산을 사용해 결합한다(더하기, 연결하기, 최솟값 구하기 등)

-원소들의 시퀀스를 컬렉션에 모은다

-원소들의 시퀀스에서 특정 조건을 만족하는 원소를 찾는다.

**스트림으로 처리하기 어려운일**

-한 데이터가 파이프라인의 여러 단계를 통과할 때 이 데이터의 각 단계에서의 값들에 동시에 접근하기는 어려운 경우

**ITEM46 스트림에서는 부작용 없는 함수를 사용하라**

순수 함수란 오직 입력만이 결과에 영향을 주는 함수를 말한다.

**ITEM47 반환 타입으로는 스트림보다 컬렉션이 낫다**

// Iterable<E>를 Stram<E>로 중개해주는 어댑터  
public static <E> Stream<E> streamOf(Iterable<E> iterable) {  
 return StreamSupport.*stream*(iterable.spliterator(), false);  
}

🡺스트림 파이프라인에서만 쓰인다면 스트림 반환을 사용

// Stream<E>를 Iterable<E>로 중개해주는 어댑터  
public static <E> Iterable<E> iterableOf(Stream<E> stream) {  
 return stream::iterator;  
}

🡺객체들이 반복문에서만 쓰일 걸 안다면 Iterable을 반환하자

Collection인터페이스는 Iterable의 하위 타입이고 stream 메서드도 제공하니 반복과 스트림을 동시에 지원한다.

따라서 원소 시퀀스를 반환하는 공개 API의 반환 타입에는 Collection이나 그 하위 타입을 쓰는게 일반적으로 최선이다.

**ITEM48 스트림 병렬화는 주의해서 적용하라**

데이터 소스가 Stream.iterate거나 중간 연산으로 limit를 쓰면 파이프라인 병렬화로는 성능 개선을 기대할 수 없다.

대체로 스트림의 소스가 ArrayList, HashMap, HashSet, ConcurrentHashMap의 인스턴스거나 배열, int 범위, long 범위일 때 병렬화의 효과가 가장 좋다.

참조지역성 : 이웃한 원소의 참조들이 메모리에 연속해서 저장되어 있다는 뜻 (배열이 가장 뛰어남)

조건이 잘 갖춰지면 parallel 메서드 호출 하나로 거의 프로세서 코어 수에 비례하는 성능 향상을 만끽할 수 있다.

**CHAPTER8**

**ITEM49 매개변수가 유효한지 검사하라**

매개변수 검사를 제대로 하지 못하면 몇 가지 문제가 생길 수 있다.

-메서드가 수행되는 중간에 모호한 예외를 던지며 실패할 수 있다. (원자성을 어길 수 있다.)

*\* <p>All methods and constructors in this class throw  
\* {****@code*** *NullPointerException} when passed  
\* a null object reference for any input parameter.*public class BigInteger extends Number implements Comparable<BigInteger> {

...

}

🡺BigInteger의 mod에 NullPointException에 대한 정의가 없는 이유는 클래스 수준에서 기술했기 때문이다.

strategy = Objects.*requireNonNull*(strategy, "전략");

requireNonNull는 검사를 수동으로 하지 않아도 되고, 원하는 예외 메시지도 지정할 수 있다.

private static void sort(long a[], int offset, int length) {  
 assert a != null;  
 assert offset >= 0 && offset <= a.length;  
 assert length >= 0 && length <= a.length - offset;  
}

🡺 VM options에 –ea를 줘야 된다.

🡺실패 시, 다음과 같은 에러가 난다.



Objects.*requireNonNull*(null);

🡺null 검사

if(ArrayUtils.*isEmpty*(a)) {  
 System.*out*.println("비었음");  
}

🡺ArrayUtils를 이용

**ITEM50 적시에 방어적 복사본을 만들라**

클라이언트가 불변식을 깨뜨리려 혈안이 되어 있다고 가정하고 방어적 프로그래밍해야 한다.

public class Period {  
 private final Date start;  
 private final Date end;  
  
 public Period(Date start, Date end) {  
 if (start.compareTo(end) > 0) {  
 throw new IllegalArgumentException(start + " after " + end);  
 }  
 this.start = start;  
 this.end = end;  
 }  
  
 public Date start() {  
 return start;  
 }  
  
 public Date end() {  
 return end;  
 }  
  
 // 클라이언트에서 어택!  
 public static void main(String[] args) {  
 Date start = new Date();  
 Date end = new Date();  
 Period p = new Period(start, end);  
 end.setYear(78);  
 }  
}

🡺자바 8 이후로는 Date 대신 불변 Instant를 사용하면 된다(LocalDateTime이나 ZonedDateTime도 가능).

Date는 낡은 API이니 새로운 코드를 작성할 때는 더 이상 사용하면 안 된다.

// 매개변수의 방어적 복사본을 만든다.  
public Period(Date start, Date end) {  
 this.start = new Date(start.getTime());  
 this.end = new Date(end.getTime());  
  
 if (this.start.compareTo(this.end) > 0) {  
 throw new IllegalArgumentException(start + " after " + end);  
 }  
}

🡺방어적 복사

**ITEM51 메서드 시그니처를 신중히 설계하라**

-메서드 이름을 신중히 짓자

-편의 메서드를 너무 많이 만들지 말자

-매개변수 목록은 짧게 유지하자 (4개 이하)

🡺긴 매개변수 목록을 짧게 줄여주는 기술 세 가지

1) 여러 메서드로 쪼갠다.

2) 매개변수 여러 개를 묶어주는 도우미 클래스를 만든다.

3) 앞서 두 기법을 혼합한 것으로, 객체 생성에 사용한 빌더 패턴을 메서드 호출에 응용한 것이다.

-매개변수의 타입으로는 클래스보다 인터페이스가 더 낫다.

-boolean보다는 원소 2개짜리 열거 타입이 낫다.

**직교 :** 각 요소들이 서로 독립적임. 원자적으로 쪼개다 보면, 자연스럽게 중복이 줄고 결합성이 낮아져 코드를 수정하기 수월해진다. 테스트하기 쉬워짐은 물론이다.

**ITEM52 다중정의는 신중히 사용하라**

public class CollectionClassifier {  
 public static String classify(Set<?> s) {  
 return "집합";  
 }  
  
 public static String classify(List<?> lst) {  
 return "리스트";  
 }  
  
 public static String classify(Collection<?> c) {  
 return "그 외";  
 }  
  
 public static void main(String[] args) {  
 Collection<?>[] collections = {  
 new HashSet<String>(),  
 new ArrayList<BigInteger>(),  
 new HashMap<String, String>().values()  
 };  
  
 for (Collection<?> c : collections) {  
 System.*out*.println(*classify*(c));  
 }  
 }  
}

🡺”그 외” 만 세 번 연달아 출력한다. 그 이유는 다중정의(overloading)는 어떤 메서드를 호출할지 컴파일시점에 정해지기 때문이다.

재정의한 메서드(overriding) 는 동적으로 선택되고, 다중정의한 메서드(overloading)는 정적으로 선택된다.

안전하고 보수적으로 가려면 매개변수 수가 같은 다중정의는 만들지 말자.

매개변수 수가 같은 다중정의 메서드가 많더라도, 그 중 어느 것이 주어진 매개변수 집합을 처리할지가 명확히 구분된다면 헷갈릴 일은 없을 것이다. (즉 매개변수 중 하나 이상 근본적으로 다르다 - 두 타입의 값을 서로 어느쪽으로든 형변환 할 수 없는 경우)

public ArrayList(int initialCapacity) {  
 . . .  
}  
public ArrayList(Collection<? extends E> c) {  
 . . .  
}

🡺int를 받는 생성자와 Collection을 받는 생성자가 있는데, 어떤 상황에서든 헷갈릴 일은 없다.

(두 타입이 어느쪽으로든 형변환 할 수 없다.)

@FunctionalInterface  
public interface Runnable {public abstract void run();  
}

@FunctionalInterface  
public interface Callable<V> {V call() throws Exception;  
}

// 1번 Thread의 생성자 호출  
new Thread(System.*out*::println).start();  
  
// 2번 ExecutorService의 submit 메서드 호출  
ExecutorService exec = Executors.*newCachedThreadPool*();  
exec.submit(System.*out*::println);

🡺2번의 submit의 호출은 실패 하는데 Callable, Runnable 다중 정의가 되어 있기 때문이다.

하지만 Runnable은 void고 Callable은 반환값이 있기 때문에 자동으로 찾을 수 있다고 생각하지만 다중정의 해소(적절한 다중정의 메서드를 찾는 알고리즘)는 이렇게 동작하지 않는다.

다중정의된 메서드들이 함수형 인터페이스를 인수로 받을 때, 비록 서로 다른 함수형 인터페이스라도 인수 위치가 같으면 혼란이 생긴다.

**ITEM53 가변인수는 신중히 사용하라**

인수 개수가 일정하지 않은 메서드를 정의해야 한다면 가변인수가 반드시 필요하다.

메서드를 정의할 때 필수 매개변수는 가변인수 앞에 두고, 가변인수를 사용할 때는 성능 문제까지 고려하자.

**ITEM54 null이 아닌, 빈 컬렉션이나 배열을 반환하라**

// 컬렉션이 비었으면 null을 반환한다. - 따라 하지 말것  
public List<String> getCheeses() {  
 return cheesesInStock.isEmpty() ? null : new ArrayList<>(cheesesInStock);  
}

빈 컨테이너를 할당하는 데도 비용이 드니 null을 반환하는 쪽이 낫다는 주장이 있다.

하지만 두가지 측면에서 틀렸다.

1) 성능 차이는 신경 쓸 수준이 못 된다.

2) 빈 컬렉션과 배열은 굳이 새로 할당하지 않고도 반환할 수 있다.

🡺

// 빈 컬렉션을 반환하는 올바른 예  
// 반환 값 - private static final Object[] EMPTY\_ELEMENTDATA = {};  
public List<String> getCheeses() {  
 return new ArrayList<>(cheesesInStock);  
}

위 방법은 가능성은 작지만, 사용 패턴에 따라 빈 컬렉션 할당이 성능을 눈에 띄게 떨어 뜨릴 수 있다. 해법은 매번 똑 같은 빈 ’불변’ 컬렉션을 반환하는 것이다.

🡺

// 빈 컬렉션을 매번 새로 할당하지 않도록 함  
public List<String> getCheeses() {  
 return cheesesInStock.isEmpty() ? Collections.*emptyList*() : new ArrayList<>(cheesesInStock);  
}

배열을 쓸때도 절대 null을 반환하지 말고 길이가 0인 배열을 반환하라

🡺

// 길이가 0일 수도 있는 배열을 반환하는 올바른 방법  
public String[] getCheeses() {  
 return cheesesInStock.toArray(new String[0]);  
}

결론 : null이 아닌, 빈 배열이나 컬렉션을 반환하라. Null을 반환하는 API는 사용하기 어렵고 오류 처리 코드도 늘어난다. 그렇다고 성능이 좋은 것도 아니다.