**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