

23.03.17 10강

어제 과제 피드백

clone()과 Cloneable

clone()은 Cloneable 인터페이스에 있지 않고 Object 클래스 안에 있다.

바로 Override하면 문제 발생

implements Cloneable : 클론을 했다는 뜻 ㅎ

```
@Override
public Student clone() {
    return new Student(같은 요소);
}
```

자바에서 이상하게 만들어 버렸다.

super.clone()

```
@Override
public Student clone() {
    return super.clone();
}
```

생성자 호출 없이 객체 생성이 가능해진다.

reflect ⇒ 느리고 일반적이지 않은 방식

참조문제 발생

배열만 안전하게 복제된다.

결론 super.clone()하지말고 직접 구현하는게 안전하다. 관습상 Cloneable 붙여주자

잘못된 부분 - 단일 책임 원칙

```
public String same(Account o) {
    String msg = "다릅니다!";
    if (this.equals(o)) {
        msg = "같습니다!";
    }
}
```

```
        return msg;
    }
```

객체는 자기가 할 일만 해야함. ⇒ 비교를 계좌가 가지고 있을 필요는 없다.

객체지향에 어긋나는 것이다. ⇒ 계좌의 기능에 다른 계좌와 비교하는 것은 없다.

⇒ ATM기나 은행원등의 객체를 따로 선언하여 그것이 계좌를 비교하게 해야한다.

헛갈린 이유 : equals는 계좌 객체의 기능이 아닌, Object를 Override한 것이라 이를 고려하지 않고 무조건적으로 다뤄줘야하지만, same은 불필요하다.

제네릭, 열거형, 이너클래스

타입이 없을때의 문제점

런타임 에러가 나기 쉽다.

IDE가 컴파일 에러를 미리 찾을 수 없다.

제네릭

타입을 나중에 원하는 형태로 정의할 수 있음

타입 안전(type safe) 효과

```
Class ArrayList<E>
Class HashMap<K,V>
```

E에 Integer, String 심지어 새로 선언한 comparable 클래스까지 다 들어올 수 있다.

제네릭을 사용하지 않는다면?

```
public class Pocket {
    private Object data;
    public void put(Object data) {
        this.data = data;
    }
    public Object get() {
        return this.data;
    }
}
```

Object를 사용하면? : 문법적 오류는 발생하지 않지만, 실행시 치명적 오류가 발생하는 부적절한 코드가 생김. ⇒ 터짐 (인간의 실수)

제네릭 사용법

```
public class Pocket<E> {  
    private E data;  
    public void put(E data) {  
        this.data = data;  
    }  
    public E get() {  
        return this.data;  
    }  
}
```

```
public class Pocket<E extends Character> {  
    private E data;  
    public void put(E data) {  
        this.data = data;  
    }  
    public E get() {  
        return this.data;  
    }  
}
```

열거형

정해 둔 값만 넣어둘 수 있는 타입

```
public class AuthState {  
    public static final int AUTHENTICATED = 1;  
    public static final int UNAUTHENTICATED = 2;  
    public static final int AUTHENTICATING = 3;  
}
```

```
enum AuthState{  
    AUTHENTICATED, UNAUTHENTICATED, AUTHENTICATING;  
}
```

이너클래스 (Inner class)

클래스 안에 정의하는 클래스

static 을 붙이고 안 붙이고의 차이 survivalcoding.com

~~```
class Outer {
 // 생략...

 class Inner {
 // 생략...
 }
}
```~~

```
class Outer {
 // 생략...

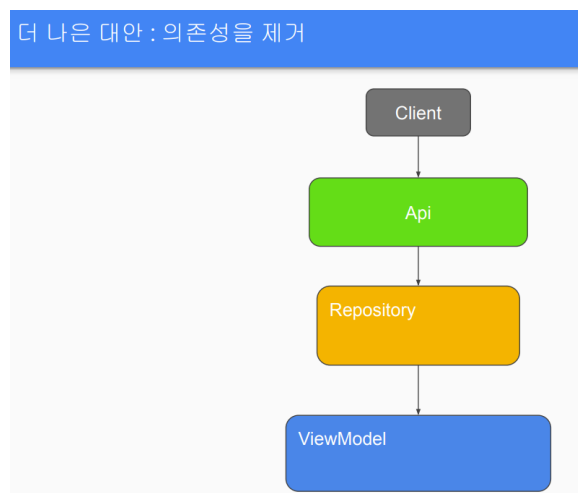
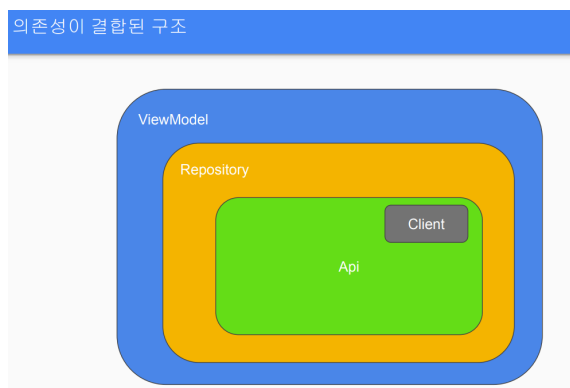
 static class Inner {
 // 생략...
 }
}
```

new Outer.Inner() 의 사용 가능 여부

Inner의 메서드에서 Outer의 필드를 접근하면 좋지 않다. ⇒ 자기 블록내에서만 작동하는 것이 좋다.

```
class Outer {
 i = 0;
 static class Inner{
 i = 10; // 갯수
 }
}
```

## 클래스 의존관계



밖의 블록이 안의 블록을 이용하는 것은 괜찮다.

하지만, 안의 블록에서 밖의 블록을 참조하면(역전), 참조순환이 발생하여 가비지 컬렉터로 인해 두개가 묶여 메모리에 남아있게 됨. ⇒ 전체가 다 묶이면 메모리가 터져서 프로그램이

죽음

⇒ static을 사용하면 의존성을 제거할 수 있다.

## 익명클래스 (anonymous class)

메서드 호출 도중에 갑자기 어떤 클래스가 필요한 경우

→ 이거저거 수정하고 새로 만들고 정신이 없다. 수정이 안 될 수 도 있다.

ex) 이름순 정렬, 이름순 역정렬, 번호순 정렬, 번호순 역정렬 등등 ....

→ String에 compare가 없음 → 이거 또 만들 → 귀찮음

```
익명클래스 (anonymous class)

Pocket<Object> Pocket = new Pocket<>();
Pocket.put(new Object() {
 String field;
 void method() {
 }
});
```

추상클래스 → new 안된다? : 거짓말

유지보수를 위해서는 결국 다 뜯어 고치는게 나음.

단일 책임 원칙 기반, 수정 등 유지보수 좋다

익명클래스는 이해하기 위해 해석 시간이 오래 걸리고 주석도 없어서 손이 많이감

→ 파이썬 협업에서 람다 쓰면 안되는 것과 같은 이유

## 연습문제

## 연습문제 4-1

다음 조건을 만족하는 금고인 **StrongBox** 클래스를 정의하시오.

- 1) 금고 클래스에 담는 인스턴스의 타입은 미정
- 2) 금고에는 1개의 인스턴스를 담을 수 있음
- 3) **put()** 메서드로 인스턴스를 저장하고 **get()** 메서드로 인스턴스를 얻을 있음
- 4) **get()** 으로 얻을 때는 별도의 타입 캐스팅을 사용하지 않아도 됨

```
package com.java14;

public class StrongBox<T> {
 private T item;
 public void put(T item) {
 this.item = item;
 }

 public T get() {
 return this.item;
 }
}
```

## 연습문제 4-2

survivalcoding.c

연습문제 4-1에서 작성한 **StrongBox** 클래스에 열쇠의 종류를 나타내는 열거형 **KeyType**을 정의하고, 다음 내용을 반영하여 **StrongBox** 클래스를 수정하시오.

- 열쇠의 종류를 나타내는 필드 변수
- 열쇠의 종류를 받는 생성자

단, 열쇠의 종류는 다음 4종류로 한정한다. 각 열쇠는 사용횟수의 한도가 정해져 있다.

- 1) PADLOCK (1,024회)
- 2) BUTTON (10,000회)
- 3) DIAL (30,000회)
- 4) FINGER (1,000,000회)

금고에서 **get()** 메서드를 호출할 때 마다 사용횟수를 카운트하고 각 열쇠의 사용횟수에 도달하기 전에는 **null**을 리턴한다.

```
package com.java14;

public class StrongBox<T> implements Comparable<T>{
```

```

private T item;
private KeyType keytype;
private int count;

public enum KeyType {
 PADLOCK("PADLOCK"), BUTTON("BUTTON"), DIAL("DIAL"), FINGER("FINGER");

 private final String type;
 private int limit;

 KeyType(String type) {
 this.type = type;
 switch (type) {
 case "PADLOCK": {
 this.limit = 1_024;
 break;
 }
 case "BUTTON": {
 this.limit = 10_000;
 break;
 }
 case "DIAL": {
 this.limit = 30_000;
 break;
 }
 case "FINGER": {
 this.limit = 1_000_000;
 break;
 }
 default: {
 System.out.println("종류를 올바르게 입력해주세요!!");
 }
 }
 }

 public int getLimit() {
 return this.limit;
 }

 public String getType() {
 return this.type;
 }
}

public StrongBox(KeyType keytype) {
 this.keytype = keytype;
}

public void put(T item) {
 this.item = item;
}

public T get() {
 count++;
 if (count < keytype.getLimit()) {
 return null;
 }
 return this.item;
}

```

```

 }

 @Override
 public String toString() {
 return this.keytype.getType();
 }

 @Override
 public int compareTo(T o) {
 return this.toString().compareToIgnoreCase(o.toString());
 }
}

```

```

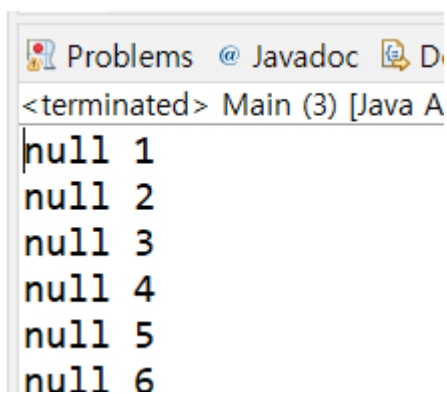
package com.java14;

import java.text.*;
import java.util.*;
import com.java14.StrongBox.KeyType;

public class Main {
 public static void main(String[] args) throws InterruptedException, ParseException {
 StrongBox<String> box = new StrongBox<>(KeyType.PADLOCK);
 box.put("stone");
 int idx = 0;
 while (true) {
 idx++;
 String temp = box.get();
 System.out.println(temp + " " + idx);
 if (temp != null) {
 break;
 }
 }
 }
}

```

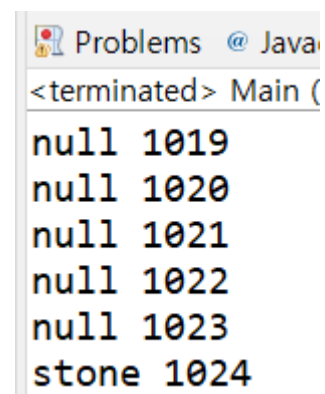
특이사항 : enum으로 생성한 클래스는 직접 인스턴스가 불가능하므로, 클래스.type 형식으로 한다.



```

Problems @ Javadoc
<terminated> Main (3) [Java A
null 1
null 2
null 3
null 4
null 5
null 6

```



```

Problems @ Java
<terminated> Main (
null 1019
null 1020
null 1021
null 1022
null 1023
stone 1024

```

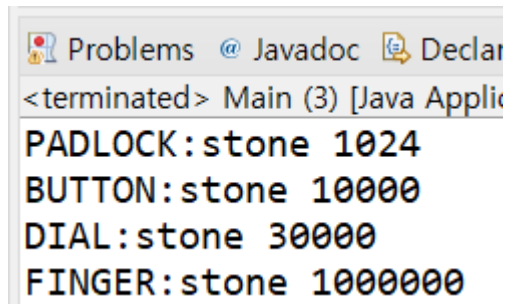


다른 것들도 잘 작동하는지 확인해보자.

```
package com.java14;

import java.text.*;
import java.util.*;
import com.java14.StrongBox.KeyType;

public class Main {
 public static void main(String[] args) throws InterruptedException, ParseException {
 List<StrongBox<String>> boxes = new ArrayList<>();
 boxes.add(new StrongBox<>(KeyType.PADLOCK));
 boxes.add(new StrongBox<>(KeyType.BUTTON));
 boxes.add(new StrongBox<>(KeyType.DIAL));
 boxes.add(new StrongBox<>(KeyType.FINGER));
 for (StrongBox<String> box : boxes) {
 box.put("stone");
 int idx = 0;
 while (true) {
 idx++;
 String temp = box.get();
 if (temp != null) {
 System.out.println(box.toString() + ":" + temp + " " + idx);
 break;
 }
 }
 }
 }
}
```



```
<terminated> Main (3) [Java Application]
PADLOCK:stone 1024
BUTTON:stone 10000
DIAL:stone 30000
FINGER:stone 1000000
```

느낀점 : enum이 헛갈린다. 객체지향에 익숙해질 필요가 있을 듯 하다.