10-13 [Java - 래퍼 클래스]

소유자	종수 김
태그	

래퍼 클래스

래퍼 클래스 - 기본형의 한계

자바는 객체지향 연어이나, 객체가 아닌 것이 존재.
바로 Primitive Type 객체가 아니기 때문에 다음과 같은 한계가 있음.

- 객체가 아님 : 기본형 데이터는 객체가 아니기 때문에, 객체 지향 프로그래밍의 장점을 살릴 수 없음
  즉, 기본형은 메서드를 제공할 수 없음을 의미.
- null 값을 가질 수 없음 : 객체는 때로는 데이터가 없음(존재하지 않음)이라는 상태를 나타내야 할 필요가 있으나, 기본 형은 항상 값을 가지기에 이런 표현을 할 수 없음.

```
package wrapper;

public class MyIntegerMethodMain0 {
    public static void main(String[] args) {
        int value = 10;
        int i1 = compareTo(value, 5);
        int i2 = compareTo(value, 10);
        int i3 = compareTo(value, 20);
        System.out.println("i1 = " + i1);
        System.out.println("i2 = " + i2);
        System.out.println("i3 = " + i3);
    }

    public static int compareTo(int value, int target) {
        if (value < target) {
            return -1;
        } else if (value > target) {
            return 1;
        } else {
```

```
            return 0;
        }
    }
}
```

위와 같은 코드의 경우 value를 기준으로 나머지 값을 비교하는 것이므로 int라는 객체 내부에 compareTo 메서드를 통해 value.compareTo(5)와 같이 객체 스스로 자기 자신의 값과 다른 값을 비교하는 메서드를 만드는 것이 더 유용할 것 처럼 보임.

직접 만든 래퍼 클래스

int를 클래스로 만들려면, int는 클래스가 아니지만 int 값을 가지고 클래스를 만들 수 있음.

```
package wrapper;

public class MyInteger {
    private final int value;

    public MyInteger(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    public int compareTo(int target) {
        if (value < target) {
            return -1;
        } else if (value > target) {
            return 1;
        } else {
            return 0;
        }
    }

    @Override
    public String toString() {
```

```
        return String.valueOf(value);
    }
}
```

- int를 멤버 변수로 받으며, 객체 형태로 감싸는 MyInteger 객체 선언.
- 기본형 변수를 사용하기 편하게 메서드를 제공.
- compareTo라는 메서드를 통해 숫자 간의 비교를 할 수 있으며 캡슐화 또한 가능.

래퍼 클래스 - 기본형의 한계2

기본형은 항상 값을 가져야함. 하지만, 때로는 데이터가 존재하지 않음(Null)이 필요함.

```
package wrapper;

public class MyIntegerNullMain {
    public static void main(String[] args) {
        int[] intArr = {-1, 0, 1, 2, 3};
        System.out.println(findValue(intArr, -1));
        System.out.println(findValue(intArr, 0));
        System.out.println(findValue(intArr, 1));
        System.out.println(findValue(intArr, 100)); // return
    }

    private static int findValue(int[] intArr, int target) {
        for(int value : intArr) {
            if (value == target) {
                return value;
            }
        }
        return -1;
    }
}
```

- findValue는 기본형인 int를 반환하므로 값이 항상 있어야 하지만, 해당 메서드는 값이 있어서 -1을 return하는 것인지, -1을 findValue한 것인지 명확하지 않음.
- 기본형(Primitive Type)은 무조건 저 한계를 극복할 수 없음.

```
package wrapper;

public class MyIntegerNullMain1 {
    public static void main(String[] args) {
        MyInteger [] intArr = {new MyInteger(-1), new MyInteg
        System.out.println(findValue(intArr, -1));
        System.out.println(findValue(intArr, 0));
        System.out.println(findValue(intArr, 1));
        System.out.println(findValue(intArr, 100)); // return
    }

    private static MyInteger findValue(MyInteger[] intArr, in
        for(MyInteger value : intArr) {
            if (value.getValue() == target) {
                return new MyInteger(value.getValue());
            }
        }
        return null;
    }
}
```

- MyInteger라는 객체를 통해 없는 경우 null을 반환할 수 있음.
- 이렇게 기본형을 객체 형태로 감싼 객체를 래퍼(wrapper)클래스라고 명명함.
- 대신 NPE(Null Pointer Exception)을 조심해야 함.

래퍼 클래스 - 자바 래퍼 클래스

기본형을 객체로 감싸서 더 편리하게 사용하도록 도와줄 수 있음.
즉, 래퍼 클래스는 기본형의 객체 버전.

기본형은 첫 문자가 소문자, 래퍼 클래스는 대문자.

- byte → Byte
- short → Short
- int → Integer
- long → Long
- float → Float

- double → Double
- char → Character
- boolean → Boolean

자바가 제공하는 기본 래퍼 클래스는 모두 '불변'이며, equals 비교해야 함.
String과 동일하다고 생각하면 됨.
(Integer.valueOf도 String Constant Pool 처럼 동작하므로 == 비교시 true)

래퍼 클래스 생성 - 박싱(Boxing)

기본형을 래퍼 클래스로 변경하는 것을 마치 박스에 물건을 넣는 것 같다고 해서 박싱(Boxing)이라고 함.

- `new Integer(10)`은 직접 사용하면 안된다. 작동은 하지만, 향후 자바에서 제거될 예정이다.
- 대신에 `Integer.valueOf(10)`를 사용하면 된다.
  - 내부에서 `new Integer(10)`을 사용해서 객체를 생성하고 돌려준다.
- 추가로 `Integer.valueOf()`에는 성능 최적화 기능이 있다. 개발자들이 일반적으로 자주 사용하는 -128 ~ 127 범위의 `Integer` 클래스를 미리 생성해준다. 해당 범위의 값을 조회하면 미리 생성된 `Integer` 객체를 반환한다. 해당 범위의 값이 없으면 `new Integer()`를 호출한다.
  - 마치 문자열 풀과 비슷하게 자주 사용하는 숫자를 미리 생성해두고 재사용한다.
  - 참고로 이런 최적화 방식은 미래에 더 나은 방식으로 변경될 수 있다.

intValue() - 언박싱(Unboxing)

- 래퍼 클래스에 들어있는 기본형 값을 다시 꺼내는 메서드
- 박스에 들어있는 물건을 꺼내는 것 같다 해서 언박싱.

비교는 equals() 사용

- 래퍼 클래스는 객체이기 때문에 `==` 비교를 하면 인스턴스의 참조값을 비교한다.
- 래퍼 클래스는 내부의 값을 비교하도록 `equals()`를 재정의 해두었다. 따라서 값을 비교하려면 `equals()`를 사용해야 한다.

참고로 래퍼 클래스는 객체를 그대로 출력해도 내부에 있는 값을 문자로 출력하도록 `toString()`을 재정의했다.

래퍼 클래스 - 오토 박싱

오토 박싱 - Autoboxing

자바에서 int를 Integer → Integer.valueOf();
Integer를 int → 변수.intValue();

> 개발자들이 오랜기간 개발을 하다 보니 기본형을 래퍼 클래스로 변환하거나 또는 래퍼 클래스를 기본형으로 변환하는 일이 자주 발생했다. 그래서 많은 개발자들이 불편함을 호소했다.
> 자바는 이런 문제를 해결하기 위해 자바 1.5부터 오토 박싱(Auto-boxing), 오토 언박싱(Auto-Unboxing)을 지원한다.

```
package wrapper;

public class AutoBoxingMain1 {
    public static void main(String[] args) {
        int value = 7;
        Integer boxedValue = value; // 오토 박싱 (Auto-Boxing)

        int unboxedValue = boxedValue; // 오토 언박싱(Auto-Unbo

        System.out.println("boxedValue = " + boxedValue);
        System.out.println("unboxedValue = " + unboxedValue);

    }
}
```

오토 박싱과 오토 언박싱은 컴파일러가 개발자 대신 valueOf, xxxValue()등의 코드를 추가해주는 기능.

래퍼 클래스 - 주요 메서드와 성능

```
package wrapper;

public class WrapperUtilsMain {
    public static void main(String[] args) {
        Integer i1 = Integer.valueOf(10); // 숫자, 래퍼 객체 변환
        Integer.valueOf("10");  // 문자열, 래퍼 객체 변환
        Integer.parseInt("10"); // 문자열 전용, 기본형 변환

        //비교
```

```
        int compareResult = i1.compareTo(20);
        System.out.println("compareResult = " + compareResult

        //산술 연산
        System.out.println("sum : " + Integer.sum(10, 20));
        System.out.println("sum : " + Integer.min(10, 20));
        System.out.println("sum : " + Integer.max(10, 20));


    }
}
```

- pasreInt() vs valueOf()

  기본형 반환 vs 래퍼 타입 반환.

래퍼 클래스와 성능

래퍼 클래스는 객체이기 때문에, 기본형 보다 다양한 기능을 제공
하지만, 기본형이 래퍼클래스에 비해 나은 성능을 제공

```
package wrapper;

public class WrapperVsPrimitive {
    public static void main(String[] args) {
        int iterations = 1_000_000_000; // 반복 횟수 설정, 10억
        long startTime, endTime;

        // 기본형 long 사용
        long sumPrimitive = 0;
        startTime = System.currentTimeMillis();
        for (int i = 0; i < iterations; i++) {
            sumPrimitive += i;
        }
        endTime = System.currentTimeMillis();
        System.out.println("sumPrimitive = " + sumPrimitive);
        System.out.println("기본 자료형 long 실행 시간 " + (endTi
```

```
        // 래퍼 클래스 Long 사용
        Long sumWrapper = 0L;
        startTime = System.currentTimeMillis();
        for (int i = 0; i < iterations; i++) {
            sumWrapper += i; // 오토 박싱 발생
        }
        endTime = System.currentTimeMillis();
        System.out.println("sumWrapper = " + sumWrapper);
        System.out.println("기본 자료형 long 실행 시간 " + (endTi


    }
}
```

실행 결과 - M2 맥북 기준

```
sumPrimitive = 499999999500000000
기본 자료형 long 실행 시간: 318ms
sumWrapper = 499999999500000000
래퍼 클래스 Long 실행 시간: 1454ms
```

- 기본형 연산이 래퍼 클래스보다 대략 5배 정도 빠른 것을 확인할 수 있다. 참고로 계산 결과는 시스템 마다 다르다.
- 기본형은 메모리에서 단순히 그 크기만큼의 공간을 차지한다. 예를 들어 `int`는 보통 4바이트의 메모리를 사용한다.
- 래퍼 클래스의 인스턴스는 내부에 필드로 가지고 있는 기본형의 값 뿐만 아니라 자바에서 객체 자체를 다루는데 필요한 객체 메타데이터를 포함하므로 더 많은 메모리를 사용한다. 자바 버전과 시스템마다 다르지만 대략 8~16 바이트의 메모리를 추가로 사용한다.

기본형, 래퍼 클래스 어떤 것을 사용?

- 이 연산은 10억 번의 연산을 수행했을 때 0.3초와, 1.5초의 차이다.
- 기본형이든 래퍼 클래스든 이것을 1회로 환산하면 둘다 매우 빠르게 연산이 수행된다.
  - 0.3초 나누기 10억, 1.5초 나누기 10억이다.
- 일반적인 애플리케이션을 만드는 관점에서 보면 이런 부분을 최적화해도 사막의 모래알 하나 정도의 차이가 날 뿐이다.
- CPU 연산을 아주 많이 수행하는 특수한 경우이거나, 수만~ 수십만 이상 연속해서 연산을 수행해야 하는 경우라면 기본형을 사용해서 최적화를 고려하자.
- 그렇지 않은 일반적인 경우라면 코드를 유지보수하기 더 나은 것을 선택하면 된다.

유지보수 vs 최적화

유지보수 vs 최적화를 고려해야 하는 상황이라면 유지보수하기 좋은 코드를 먼저 고민해야 한다. 특히 최신 컴퓨터는 매우 빠르기 때문에 메모리 상에서 발생하는 연산을 몇 번 줄인다고해도 실질적인 도움이 되지 않는 경우가 많다.

- 코드 변경 없이 성능 최적화를 하면 가장 좋겠지만, 성능 최적화는 대부분 단순함 보다는 복잡함을 요구하고, 더 많은 코드들을 추가로 만들어야 한다. 최적화를 위해 유지보수 해야 하는 코드가 더 늘어나는 것이다. 그런데 진짜 문제는 최적화를 한다고 했지만 전체 애플리케이션의 성능 관점에서 보면 불필요한 최적화를 할 가능성이 있다.
- 특히 웹 애플리케이션의 경우 메모리 안에서 발생하는 연산 하나보다 네트워크 호출 한 번이 많게는 수십만배 더 오래 걸린다. 자바 메모리 내부에서 발생하는 연산을 수천번에서 한 번으로 줄이는 것 보다, 네트워크 호출 한 번을 더 줄이는 것이 더 효과적인 경우가 많다.
- 권장하는 방법은 개발 이후에 성능 테스트를 해보고 정말 문제가 되는 부분을 찾아서 최적화 하는 것이다.

Class 클래스

클래스의 정보(메타데이터)를 다루는데 사용.
Class 클래스를 통해 개발자는 실행 중인 자바 애플리케이션 내에서 필요한 클래스의 속성과 메서드에 대한 정보를 조회하고 조작할 수 있음.

- 타입 정보 얻기
  - 클래스의 이름, 슈퍼클래스, 인터페이스, 접근 제한자 등과 같은 정보를 조회 가능.
- 리플렉션
  - 클래스에 정의된 메서드 필드, 생성자 등을 조회하고 이들을 통해 객체 인스턴스를 생성하거나 메서드를 호출하는 등의 작업 가능.
- 동적 로딩과 생성
  - Class.forName() 메서드를 사용하여 클래스를 동적으로 로드하고, newInstance() 메서드를 통해 새로운 인스턴스를 생성

- 애노테이션 처리
    - 클래스에 적용된 애노테이션을 조회하고 처리하는 기능을 제공

```
package lang.clazz;

import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ClassMetaMain {
    public static void main(String[] args) throws Exception {
        // Class 조회의 3가지 방법
        Class clazz = String.class; // 1. 클래스에서 조회
//        Class clazz = new String().getClass(); // 2. 인스턴스
//        Class clazz = Class.forName("java.lang.String"); //

        // 모든 필드 출력
        Field[] fields = clazz.getDeclaredFields();
        for (Field field : fields) {
            System.out.println("field = " + field.getType() +
        }

        // 모든 메서드 출력
        Method[] methods = clazz.getDeclaredMethods();
        for (Method method : methods) {
            System.out.println("Method : " + method);
        }

        // 상위 클래스 정보 출력
        System.out.println("Superclass = " + clazz.getSupercl

        // 인터페이스 정보 출력
        Class[] interfaces = clazz.getInterfaces();
        for (Class i : interfaces) {
            System.out.println("Interface : " + i);
        }
    }
}
```

클래스 생성하기

동적으로 클래스를 생성

```
package lang.clazz;

public interface Test {
    public String hello();
}

package lang.clazz;

public class Hello implements Test{
    @Override
    public String hello() {
        return "hello";
    }
}

package lang.clazz;

public class Bye implements Test{
    @Override
    public String hello(){
        return "bye";
    }
}

package lang.clazz;

import java.util.Scanner;

public class ClassCreateMain {
    public static void main(String[] args) throws Exception {
        // 동적으로 객체를 만드는 것이 가능, 즉 사용자의 입력에 맞추어 ㅅ
//        Class helloClass = Hello.class;
        Scanner sc = new Scanner(System.in);
        //lang.clazz.Hello / lang.clazz.Bye
```

```
        Class helloClass = Class.forName(sc.nextLine());

        Test object = (Test) helloClass.getDeclaredConstructo
        System.out.println(object.hello());
    }
}
```

리플렉션 - relection

`Class`를 사용하면 클래스의 메타 정보를 기반으로 클래스에 정의된 메서드, 필드, 생성자 등을 조회하고, 이들을 통해 객체 인스턴스를 생성하거나 메서드를 호출하는 작업을 할 수 있다. 이런 작업을 리플렉션이라 한다. 추가로 애노테이션 정보를 읽어서 특별한 기능을 수행할 수 도 있다. 최신 프레임워크들은 이런 기능을 적극 활용한다.
지금은 `Class`가 뭔지, 그리고 대략 어떤 기능들을 제공하는지만 알아두면 충분하다. 지금은 리플렉션을 학습하는 것보다 훨씬 더 중요한 기본기들을 학습해야 한다. 리플렉션은 이후에 별도로 다룬다.

System 클래스

시스템과 관련된 기본 기능을 제공

```
package lang.system;

import java.util.Arrays;

public class SystemMain {
    public static void main(String[] args) {
        // 현재 시간(밀리초)를 가져옴
        long currentTime = System.currentTimeMillis();
        System.out.println("currentTime = " + currentTime);

        //현재 시간(나노초)를 가져옴
        long currentNanoTime = System.nanoTime();
        System.out.println("currentNanoTime = " + currentNano

        //환경 변수
        System.out.println("env  = " + System.getenv());

        // 시스템 속성
        System.out.println("properties = " + System.getProper
```

```
        System.out.println("Java version = " + System.getProp

        // 배열 고속 복사
        char[] originalArray = new char[]{'h', 'e', 'l', 'l',
        char[] copiedArray = new char[originalArray.length];
        System.arraycopy(originalArray, 0, copiedArray, 0, or
        System.out.println("originalArray = " + Arrays.toStri
        copiedArray[0] = 'J';
        System.out.println("copied Array = " + Arrays.toStrin

        // 프로그램 종료
        System.exit(0);
    }
}
```

- **표준 입력, 출력, 오류 스트림**: `System.in`, `System.out`, `System.err`은 각각 표준 입력, 표준 출력, 표준 오류 스트림을 나타낸다.
- **시간 측정**: `System.currentTimeMillis()`와 `System.nanoTime()`은 현재 시간을 밀리초 또는 나노초 단위로 제공한다.
- **환경 변수**: `System.getenv()` 메서드를 사용하여 OS에서 설정한 환경 변수의 값을 얻을 수 있다.
- **시스템 속성**: `System.getProperties()`를 사용해 현재 시스템 속성을 얻거나 `System.getProperty(String key)`로 특정 속성을 얻을 수 있다. 시스템 속성은 자바에서 사용하는 설정 값이다.
- **시스템 종료**: `System.exit(int status)` 메서드는 프로그램을 종료하고, OS에 프로그램 종료의 상태 코드를 전달한다.
  - 상태 코드 `0`: 정상 종료
  - 상태 코드 `0`이 아님: 오류나 예외적인 종료
- **배열 고속 복사**: `System.arraycopy`는 시스템 레벨에서 최적화된 메모리 복사 연산을 사용한다. 직접 반복문을 사용해서 배열을 복사할 때 보다 수 배 이상 빠른 성능을 제공한다.

Math, Random 클래스

Math

수학과 관련된 문제를 해결해주는 클래스.

```
package lang.math;

public class MathMain {
```

```
    public static void main(String[] args) {
        // 기본 연산 메서드
        System.out.println("max(10, 20) " + Math.max(10, 20))
        System.out.println("min(10, 20) " + Math.min(10, 20))
        System.out.println("abs(-10) " + Math.abs(-10)); // 절

        // 반올림 및 정밀도 메서드
        System.out.println("ceil(2.1) " + Math.ceil(2.1)); //
        System.out.println("floor(2.1) " + Math.floor(2.1));
        System.out.println("round(2.5) " + Math.round(2.5));

        // 기타 유용한 메서드
        System.out.println("sqrt(4) " + Math.sqrt(4)); // 제곱
        System.out.println("random() " + Math.random()); // 0


    }
}
```

1. 기본 연산 메서드

- `abs(x)` : 절대값
- `max(a, b)` : 최대값
- `min(a, b)` : 최소값

2. 지수 및 로그 연산 메서드

- `exp(x)` : e^x 계산
- `log(x)` : 자연 로그
- `log10(x)` : 로그 10
- `pow(a, b)` : a의 b 제곱

3. 반올림 및 정밀도 메서드

- `ceil(x)` : 올림
- `floor(x)` : 내림
- `rint(x)` : 가장 가까운 정수로 반올림
- `round(x)` : 반올림

4. 삼각 함수 메서드

- `sin(x)` : 사인
- `cos(x)` : 코사인
- `tan(x)` : 탄젠트

5. 기타 유용한 메서드

- `sqrt(x)` : 제곱근
- `cbrt(x)` : 세제곱근
- `random()` : 0.0과 1.0 사이의 무작위 값 생성

Random

```
package lang.math;

import java.util.Random;

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

```
        Random random = new Random();

        int randomInt = random.nextInt();
        System.out.println("randomInt = " + randomInt);

        double randomDouble = random.nextDouble();
        System.out.println("randomDouble = " + randomDouble);

        boolean randomBoolean = random.nextBoolean();
        System.out.println("randomBoolean = " + randomBoolean

        // 범위 조회
        int randomRange1 = random.nextInt(10); // 0 ~ 9
        System.out.println("randomRange1 = " + randomRange1);

        int randomRange2 = random.nextInt(10) + 1;
        System.out.println("randomRange2 = " + randomRange2);
    }
}
```

Seed

랜덤은 내부에서 Seed값을 사용해서 랜덤 값을 구하는데, 해당 Seed 값이 같으면 항상 같은 값을 출력.

```
package lang.math;

import java.util.Random;

public class RandomMain {
    public static void main(String[] args) {
        Random random = new Random(1); // seed가 같으면 Random으

        int randomInt = random.nextInt();
        System.out.println("randomInt = " + randomInt);
    }
}
```

```
// 하위 결과 값 고정. (Seed가 같으면 같은 랜덤 값을 출력)
// randomInt = -1155869325
```

- 결과가 고정되므로, 테스트 코드 같은 곳에서 같은 결과를 검증할 수 있음.
  Ex) 마인크래프트 : 게임 시작 시 지형을 랜덤으로 생성하나, Seed값이 같으면 같은 지형을 생성.