4. 래퍼, Class 클래스

#1.인강/0.자바/3.자바-중급1편

- /래퍼 클래스 - 기본형의 한계1
- /래퍼 클래스 - 기본형의 한계2
- /래퍼 클래스 - 자바 래퍼 클래스
- /래퍼 클래스 - 오토 박싱
- /래퍼 클래스 - 주요 메서드와 성능
- /Class 클래스
- /System 클래스
- /Math, Random 클래스
- /문제와 풀이1
- /문제와 풀이2
- /정리

래퍼 클래스 - 기본형의 한계1

기본형의 한계

자바는 객체 지향 언어이다. 그런데 자바 안에 객체가 아닌 것이 있다. 바로 `int`, `double` 같은 기본형(Primitive Type)이다.

기본형은 객체가 아니기 때문에 다음과 같은 한계가 있다.

- **객체가 아님**: 기본형 데이터는 객체가 아니기 때문에, 객체 지향 프로그래밍의 장점을 살릴 수 없다. 예를 들어 객체는 유용한 메서드를 제공할 수 있는데, 기본형은 객체가 아니므로 메서드를 제공할 수 없다.
  - 추가로 객체 참조가 필요한 컬렉션 프레임워크를 사용할 수 없다. 그리고 제네릭도 사용할 수 없다. (이 부분은 뒤에서 설명한다.)
- **null 값을 가질 수 없음**: 기본형 데이터 타입은 `null` 값을 가질 수 없다. 때로는 데이터가 `없음` 이라는 상태를 나타내야 할 필요가 있는데, 기본형은 항상 값을 가지기 때문에 이런 표현을 할 수 없다.

기본형의 한계를 이해하기 위해, 두 값을 비교해서 다음과 같은 결과를 출력하는 간단한 코드를 작성해보자.

- 왼쪽의 값이 더 작다 `-1`
- 두 값이 같다 `0`
- 왼쪽의 값이 더 크다 `1`

```
package lang.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;
        }
    }

}
```

실행 결과

```
i1 = 1
i2 = 0
i3 = -1
```

여기서는 `value`와 비교 대상 값을 `compareTo()`라는 외부 메서드를 사용해서 비교한다. 그런데 자기 자신인 `value`와 다른 값을 연산하는 것이기 때문에 항상 자기 자신의 값인 `value`가 사용된다. 이런 경우 만약 `value`가 객체라면 `value` 객체 스스로 자기 자신의 값과 다른 값을 비교하는 메서드를 만드는 것이 더 유용할 것이다.

직접 만든 래퍼 클래스

`int`를 클래스로 만들어보자. `int`는 클래스가 아니지만, `int` 값을 가지고 클래스를 만들면 된다. 다음 코드는 마치 `int`를 클래스로 감싸서 만드는 것 처럼 보인다. 이렇게 특정 기본형을 감싸서(Wrap) 만드는 클래스를 래퍼 클래스(Wrapper class)라 한다.

```
package lang.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); //숫자를 문자로 변경
    }
}
```

- `MyInteger`는 `int value`라는 단순한 기본형 변수를 하나 가지고 있다.
- 그리고 이 기본형 변수를 편리하게 사용하도록 다양한 메서드를 제공한다.
- 앞에서 본 `compareTo()` 메서드를 클래스 내부로 캡슐화 했다.
- 이 클래스는 불변으로 설계했다.

`MyInteger` 클래스는 단순한 데이터 조각인 `int`를 내부에 품고, 메서드를 통해 다양한 기능을 추가했다. 덕분에 데이터 조각에 불과한 `int`를 `MyInteger`를 통해 객체로 다룰 수 있게 되었다.

```
package lang.wrapper;

public class MyIntegerMethodMain1 {

    public static void main(String[] args) {
        MyInteger myInteger = new MyInteger(10);
```

```
        int i1 = myInteger.compareTo(5);
        int i2 = myInteger.compareTo(10);
        int i3 = myInteger.compareTo(20);
        System.out.println("i1 = " + i1);
        System.out.println("i2 = " + i2);
        System.out.println("i3 = " + i3);
    }
}
```

실행 결과

```
i1 = 1
i2 = 0
i3 = -1
```

- `myInteger.compareTo()`는 자기 자신의 값을 외부의 값과 비교한다.
- `MyInteger`는 객체이므로 자신이 가진 메서드를 편리하게 호출할 수 있다.
- 참고로 `int`는 기본형이기 때문에 스스로 메서드를 가질 수 없다.

래퍼 클래스 - 기본형의 한계2

기본형과 null

기본형은 항상 값을 가져야 한다. 하지만 때로는 데이터가 '없음'이라는 상태가 필요할 수 있다.

다음 코드를 작성해보자.

```
package lang.wrapper;

public class MyIntegerNullMain0 {

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

    private static int findValue(int[] intArr, int target) {
```

```
        for (int value : intArr) {
            if (value == target) {
                return value;
            }
        }
        return -1;
    }
}
```

- `findValue()`는 배열에 찾는 값이 있으면 해당 값을 반환하고, **찾는 값이 없으면 `-1`을 반환**한다.
- `findValue()`는 결과로 `int`를 반환한다. `int`와 같은 기본형은 항상 값이 있어야 한다. 여기서도 값을 반환할 때 값을 찾지 못하면 숫자 중에 하나를 반환해야 하는데 보통 `-1` 또는 `0`을 사용한다.

실행 결과

```
-1
0
1
-1
```

실행 결과를 보면 입력값이 `-1`일 때 `-1`을 반환한다. 그런데 배열에 없는 값인 `100`을 입력해도 같은 `-1`을 반환한다. 입력값이 `-1`일 때를 생각해보면, 배열에 `-1` 값이 있어서 `-1`을 반환한 것인지, 아니면 `-1` 값이 없어서 `-1`을 반환한 것인지 명확하지 않다.

객체의 경우 데이터가 없다는 `null`이라는 명확한 값이 존재한다.
다음 코드를 작성해보자.

```
package lang.wrapper;

public class MyIntegerNullMain1 {

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

    private static MyInteger findValue(MyInteger[] intArr, int target) {
        for (MyInteger myInteger : intArr) {
            if (myInteger.getValue() == target) {
                return myInteger;
            }
```

```
        }
        return null;
    }
}
```

실행 결과

```
-1
0
1
null
```

- 앞서 만든 `MyInteger` 래퍼 클래스를 사용했다.
- 실행 결과를 보면 `-1`을 입력했을 때는 `-1`을 반환한다.
- `100`을 입력했을 때는 값이 없다는 `null`을 반환한다.

기본형은 항상 값이 존재해야 한다. 숫자의 경우 `0`, `-1` 같은 값이라도 항상 존재해야 한다. 반면에 객체인 참조형은 값이 없다는 `null`을 사용할 수 있다. 물론 `null` 값을 반환하는 경우 잘못하면 `NullPointerException`이 발생할 수 있기 때문에 주의해서 사용해야 한다.

래퍼 클래스 - 자바 래퍼 클래스

지금까지 설명한 래퍼 클래스는 기본형을 객체로 감싸서 더 편리하게 사용하도록 도와주기 때문에 상당히 유용하다.
쉽게 이야기해서 래퍼 클래스는 기본형의 객체 버전이다.

자바는 기본형에 대응하는 래퍼 클래스를 기본으로 제공한다.

- `byte` ➜ `Byte`
- `short` ➜ `Short`
- `int` ➜ `Integer`
- `long` ➜ `Long`
- `float` ➜ `Float`
- `double` ➜ `Double`
- `char` ➜ `Character`
- `boolean` ➜ `Boolean`

그리고 자바가 제공하는 기본 래퍼 클래스는 다음과 같은 특징을 가지고 있다.

- 불변이다.
- `equals`로 비교해야 한다.

자바가 제공하는 래퍼 클래스의 사용법을 알아보자.

```
package lang.wrapper;

public class WrapperClassMain {

    public static void main(String[] args) {
        Integer newInteger = new Integer(10); //미래에 삭제 예정, 대신에 valueOf() 사용
        Integer integerObj = Integer.valueOf(10); //-128 ~ 127 자주 사용하는 숫자 값
재사용, 불변
        Long longObj = Long.valueOf(100);
        Double doubleObj = Double.valueOf(10.5);

        System.out.println("newInteger = " + newInteger);
        System.out.println("integerObj = " + integerObj);
        System.out.println("longObj = " + longObj);
        System.out.println("doubleObj = " + doubleObj);

        System.out.println("내부 값 읽기");
        int intValue = integerObj.intValue();
        System.out.println("intValue = " + intValue);
        long longValue = longObj.longValue();
        System.out.println("longObj = " + longValue);

        System.out.println("비교");
        System.out.println("==: " + (newInteger == integerObj));
        System.out.println("equals: " + newInteger.equals(integerObj));
    }
}
```

실행 결과

```
newInteger = 10
integerObj = 10
longObj = 100
doubleObj = 10.5

내부 값 읽기
intValue = 10
longObj = 100
```

```
비교
==: false
equals: true
```

래퍼 클래스 생성 - 박싱(Boxing)

- 기본형을 래퍼 클래스로 변경하는 것을 마치 박스에 물건을 넣은 것 같다고 해서 **박싱(Boxing)**이라 한다.
- `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 메서드이다.
- 박스에 들어있는 물건을 꺼내는 것 같다고 해서 **언박싱(Unboxing)**이라 한다.

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 `int` 를 `Integer` 로 변환하거나, `Integer` 를 `int` 로 변환하는 부분을 정리해보자.
다음과 같이 `valueOf()` , `intValue()` 메서드를 사용하면 된다.

```
package lang.wrapper;
```

```
public class AutoboxingMain1 {
    public static void main(String[] args) {
        // Primitive -> Wrapper
        int value = 7;
        Integer boxedValue = Integer.valueOf(value);

        // Wrapper -> Primitive
        int unboxedValue = boxedValue.intValue();

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

실행 결과

```
boxedValue = 7
unboxedValue = 7
```

- 박싱: `valueOf()`
- 언박싱: `xxxValue()` (예: `intValue()`, `doubleValue()`)

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.

오토 박싱, 오토 언박싱

```
package lang.wrapper;

public class AutoboxingMain2 {
    public static void main(String[] args) {
        // Primitive -> Wrapper
        int value = 7;
        Integer boxedValue = value; // 오토 박싱(Auto-boxing)

        // Wrapper -> Primitive
        int unboxedValue = boxedValue; // 오토 언박싱(Auto-Unboxing)

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

```
    }
}
```

실행 결과

```
boxedValue = 7
unboxedValue = 7
```

오토 박싱과 오토 언박싱은 컴파일러가 개발자 대신 `valueOf`, `xxxValue()` 등의 코드를 추가해주는 기능이다.
덕분에 기본형과 래퍼형을 서로 편리하게 변환할 수 있다.
따라서 `AutoboxingMain1` 과 `AutoboxingMain2` 는 동일하게 작동한다.

```
Integer boxedValue = value; //오토 박싱(Auto-boxing)
Integer boxedValue = Integer.valueOf(value); //컴파일 단계에서 추가

int unboxedValue = boxedValue; //오토 언박싱(Auto-Unboxing)
int unboxedValue = boxedValue.intValue(); //컴파일 단계에서 추가
```

래퍼 클래스 - 주요 메서드와 성능

래퍼 클래스 - 주요 메서드

레퍼 클래스가 제공하는 주요 메서드를 알아보자.

```
package lang.wrapper;

public class WrapperUtilsMain {

    public static void main(String[] args) {
        Integer i1 = Integer.valueOf(10);//숫자, 래퍼 객체 반환
        Integer i2 = Integer.valueOf("10");//문자열, 래퍼 객체 반환
        int intValue = Integer.parseInt("10");//문자열 전용, 기본형 반환

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

        //산술 연산
```

```
        System.out.println("sum: " + Integer.sum(10, 20));
        System.out.println("min: " + Integer.min(10, 20));
        System.out.println("max: " + Integer.max(10, 20));
    }
}
```

실행 결과

```
compareResult = -1
sum: 30
min: 10
max: 20
```

- `valueOf()` : 래퍼 타입을 반환한다. 숫자, 문자열을 모두 지원한다.
- `parseInt()` : 문자열을 기본형으로 변환한다.
- `compareTo()` : 내 값과 인수로 넘어온 값을 비교한다. 내 값이 크면 `1`, 같으면 `0`, 내 값이 작으면 `-1`을 반환한다.
- `Integer.sum()`, `Integer.min()`, `Integer.max()` : `static` 메서드이다. 간단한 덧셈, 작은 값, 큰 값 연산을 수행한다.

parseInt() vs valueOf()

원하는 타입에 맞는 메서드를 사용하면 된다.

- `valueOf("10")` 는 래퍼 타입을 반환한다.
- `parseInt("10")` 는 기본형을 반환한다.
  - `Long.parseLong()` 처럼 각 타입에 `parseXxx()`가 존재한다.

래퍼 클래스와 성능

래퍼 클래스는 객체이기 때문에 기본형보다 다양한 기능을 제공한다.

그렇다면 더 좋은 래퍼 클래스만 제공하면 되지 기본형을 제공하는 이유는 무엇일까?

다음 코드를 실행해서 기본형과, 래퍼 클래스의 성능 차이를 비교해보자.

```
package lang.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: " + (endTime - startTime) +
"ms");

        // 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("래퍼 클래스 Long 실행 시간: " + (endTime - startTime) +
"ms");
    }
}
```

- 단순히 값을 반복해서 10억 번 더한다.
- 기본형 `long`에 더하는 것과 래퍼 클래스 `Long`에 더하는 부분으로 나누어 테스트 한다. 결과 값은 같다.

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

자바에서 `Class` 클래스는 클래스의 정보(메타데이터)를 다루는데 사용된다. `Class` 클래스를 통해 개발자는 실행 중인 자바 애플리케이션 내에서 필요한 클래스의 속성과 메서드에 대한 정보를 조회하고 조작할 수 있다.

`Class` 클래스의 주요 기능은 다음과 같다.

- **타입 정보 얻기**: 클래스의 이름, 슈퍼클래스, 인터페이스, 접근 제한자 등과 같은 정보를 조회할 수 있다.
- **리플렉션**: 클래스에 정의된 메서드, 필드, 생성자 등을 조회하고, 이들을 통해 객체 인스턴스를 생성하거나 메서드를 호출하는 등의 작업을 할 수 있다.
- **동적 로딩과 생성**: `Class.forName()` 메서드를 사용하여 클래스를 동적으로 로드하고, `newInstance()` 메서드를 통해 새로운 인스턴스를 생성할 수 있다.
- **애노테이션 처리**: 클래스에 적용된 애노테이션(annotation)을 조회하고 처리하는 기능을 제공한다.

예를 들어, `String.class`는 `String` 클래스에 대한 `Class` 객체를 나타내며, 이를 통해 `String` 클래스에 대한

메타데이터를 조회하거나 조작할 수 있다.

다음 코드를 실행해보자.

```
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 조회
        Class clazz = String.class; // 1.클래스에서 조회
        //Class clazz = new String().getClass();// 2.인스턴스에서 조회
        //Class clazz = Class.forName("java.lang.String"); // 3.문자열로 조회

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

        // 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.getSuperclass().getName());

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

class vs clazz - class는 자바의 예약어다. 따라서 패키지명, 변수명으로 사용할 수 없다.
이런 이유로 자바 개발자들은 `class` 대신 `clazz`라는 이름을 관행으로 사용한다. `clazz`는 `class`와 유사하게 들리고, 이 단어가 `class`를 의미한다는 것을 쉽게 알 수 있다.

주의!

`main()` 옆에 `throws Exception`이 추가된 부분에 주의하자. 이 코드가 없으면 컴파일 오류가 발생한다. 자세한 내용은 예외 처리에서 학습한다.

실행 결과

```
Field: class [B value
...
Method: public boolean java.lang.String.equals(java.lang.Object)
Method: public int java.lang.String.length()
...
Superclass: java.lang.Object
Interface: java.io.Serializable
Interface: java.lang.Comparable
...
```

`Class` 클래스는 다음과 같이 3가지 방법으로 조회할 수 있다.

```
Class clazz = String.class; // 1.클래스에서 조회
Class clazz = new String().getClass();// 2.인스턴스에서 조회
Class clazz = Class.forName("java.lang.String"); // 3.문자열로 조회
```

Class 클래스의 주요 기능

- **getDeclaredFields()**: 클래스의 모든 필드를 조회한다.
- **getDeclaredMethods()**: 클래스의 모든 메서드를 조회한다.
- **getSuperclass()**: 클래스의 부모 클래스를 조회한다.
- **getInterfaces()**: 클래스의 인터페이스들을 조회한다.

실행 결과를 보면 `String` 클래스의 다양한 정보를 확인할 수 있다.

클래스 생성하기

`Class` 클래스에는 클래스의 모든 정보가 들어있다. 이 정보를 기반으로 인스턴스를 생성하거나, 메서드를 호출하고, 필드의 값도 변경할 수 있다. 여기서는 간단하게 인스턴스를 생성해보자.

```
package lang.clazz;

public class Hello {
    public String hello() {
```

```
        return "hello!";
    }
}
```

```
package lang.clazz;

public class ClassCreateMain {
    public static void main(String[] args) throws Exception {
        //Class helloClass = Hello.class;
        Class helloClass = Class.forName("lang.clazz.Hello");
        Hello hello = (Hello) helloClass.getDeclaredConstructor().newInstance();
        String result = hello.hello();
        System.out.println("result = " + result);
    }
}
```

실행 결과

```
result = hello!
```

getDeclaredConstructor().newInstance()

- `getDeclaredConstructor()` : 생성자를 선택한다.
- `newInstance()` : 선택된 생성자를 기반으로 인스턴스를 생성한다.

리플렉션 - reflection

`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

`System` 클래스는 시스템과 관련된 기본 기능들을 제공한다.

```java
package lang.system;

import java.util.Arrays;

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

        // 현재 시간(나노초)를 가져온다.
        long currentTimeNano = System.nanoTime();
        System.out.println("currentTimeNano: " + currentTimeNano);

        // 환경 변수를 읽는다.
        System.out.println("getenv = " + System.getenv());

        // 시스템 속성을 읽는다.
        System.out.println("properties = " + System.getProperties());
        System.out.println("Java version: " +
System.getProperty("java.version"));

        // 배열을 고속으로 복사한다.
        char[] originalArray = new char[]{'h', 'e', 'l', 'l', 'o'};
        char[] copiedArray = new char[5];
        System.arraycopy(originalArray, 0, copiedArray, 0,
originalArray.length);

        // 배열 출력
        System.out.println("copiedArray = " + copiedArray);
        System.out.println("Arrays.toString = " + Arrays.toString(copiedArray));

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

실행 결과

```
currentTimeMillis: 1703570732276
currentTimeNano: 286372106104583

getenv = {IDEA_INITIAL_DIRECTORY=/, COMMAND_MODE=unix2003, LC_CTYPE=ko_KR.UTF-8,
SHELL=/bin/zsh, HOME=/Users/yh, PATH=/opt/homebrew/bin:/usr/local/bin: ...}
```

```
properties = {java.specification.version=21, java.version=21.0.1,
sun.jnu.encoding=UTF-8, os.name=Mac OS X, file.encoding=UTF-8 ...}

Java version: 21.0.1

copiedArray = [C@77459877
Arrays.toString = [h, e, l, l, o]
```

- **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 클래스

`Math`는 수 많은 수학 문제를 해결해주는 클래스이다. 너무 많은 기능을 제공하기 때문에 대략 이런 것이 있구나 하는 정도면 충분하다. 실제 필요할 때 검색하거나 API 문서를 찾아 보자.

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

`Math` 에서 자주 사용하는 기능들을 예제로 만들어서 실행해보자.

```
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)); //최대값
        System.out.println("min(10, 20): " + Math.min(10, 20)); //최소값
        System.out.println("abs(-10): " + Math.abs(-10)); //절대값

        // 반올림 및 정밀도 메서드
        System.out.println("ceil(2.1): " + Math.ceil(2.1)); //올림
        System.out.println("floor(2.7): " + Math.floor(2.7)); //내림
        System.out.println("round(2.5): " + Math.round(2.5)); //반올림
```

```
        // 기타 유용한 메서드
        System.out.println("sqrt(4): " + Math.sqrt(4)); //제곱근
        System.out.println("random(): " + Math.random()); //0.0 ~ 1.0 사이의
double 값
    }
}
```

실행 결과

```
max(10, 20): 20
min(10, 20): 10
abs(-10): 10
ceil(2.1): 3.0
floor(2.7): 2.0
round(2.5): 3
sqrt(4): 2.0
random(): 0.006347084592260965
```

> **참고**: 아주 정밀한 숫자와 반올림 계산이 필요하다면 `BigDecimal`을 검색해보자.

Random 클래스

랜덤의 경우 `Math.random()`을 사용해도 되지만 `Random` 클래스를 사용하면 더욱 다양한 랜덤값을 구할 수 있다.
참고로 `Math.random()`도 내부에서는 `Random` 클래스를 사용한다.

참고로 `Random` 클래스는 `java.util` 패키지 소속이다.

```
package lang.math;

import java.util.Random;

public class RandomMain {

    public static void main(String[] args) {
        Random random = new Random();
        //Random random = new Random(1); //seed가 같으면 Random의 결과가 같다.

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

        double randomDouble = random.nextDouble(); //0.0d ~ 1.0d
```

```
        System.out.println("randomDouble: " + randomDouble);

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

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

        int randomRange2 = random.nextInt(10) + 1; //1 ~ 10까지 출력
        System.out.println("1 ~ 10: " + randomRange2);
    }
}
```

실행 결과

실행 결과는 항상 다르다.

```
randomInt: -1316070581
randomDouble: 0.37735342193577215
randomBoolean: false
0 ~ 9: 5
1 ~ 10: 7
```

- `random.nextInt()` : 랜덤 `int` 값을 반환한다.
- `nextDouble()` : `0.0d` ~ `1.0d` 사이의 랜덤 `double` 값을 반환한다.
- `nextBoolean()` : 랜덤 `boolean` 값을 반환한다.
- `nextInt(int bound)` : `0` ~ `bound` 미만의 숫자를 랜덤으로 반환한다. 예를 들어서 3을 입력하면 `0, 1, 2`를 반환한다.

1부터 특정 숫자의 `int` 범위를 구하는 경우 `nextInt(int bound)`의 결과에 +1을 하면 된다.

씨드 - Seed

랜덤은 내부에서 씨드(Seed)값을 사용해서 랜덤 값을 구한다. 그런데 이 씨드 값이 같으면 항상 같은 결과가 출력된다.

```
//Random random = new Random();
Random random = new Random(1); //seed가 같으면 Random의 결과가 같다.
```

실행 결과

`Seed`가 같으면 실행 결과는 반복 실행해도 같다.

```
randomInt: -1155869325
randomDouble: 0.10047321632624884
randomBoolean: false
0 ~ 9: 4
1 ~ 10: 5
```

- `new Random()` : 생성자를 비워두면 내부에서 `System.nanoTime()` 에 여러가지 복잡한 알고리즘을 섞어서 씨드값을 생성한다. 따라서 반복 실행해도 결과가 항상 달라진다.
- `new Random(int seed)` : 생성자에 씨드 값을 직접 전달할 수 있다. 씨드 값이 같으면 여러번 반복 실행해도 실행 결과가 같다. 이렇게 씨드 값을 직접 사용하면 결과 값이 항상 같기 때문에 결과가 달라지는 랜덤 값을 구할 수 없다. 하지만 결과가 고정되기 때문에 테스트 코드 같은 곳에서 같은 결과를 검증할 수 있다. 참고로 마인크래프트 같은 게임은 게임을 시작할 때 지형을 랜덤으로 생성하는데, 같은 씨드값을 설정하면 같은 지형을 생성할 수 있다.

문제와 풀이1

문제1 - parseInt()

문제 설명

- 문자로 입력된 `str1` , `str2` 두 수의 합을 구하자.

```
package lang.wrapper.test;

public class WrapperTest1 {
    public static void main(String[] args) {
        String str1 = "10";
        String str2 = "20";

        // 코드 작성
    }
}
```

실행 결과

```
두 수의 합: 30
```

정답

```java
package lang.wrapper.test;

public class WrapperTest1 {
    public static void main(String[] args) {
        String str1 = "10";
        String str2 = "20";

        int num1 = Integer.parseInt(str1);
        int num2 = Integer.parseInt(str2);
        int sum = num1 + num2;
        System.out.println("두 수의 합: " + sum);
    }
}
```

문제2 - parseDouble()

문제 설명

- 배열에 입력된 모든 숫자의 합을 구하자. 숫자는 `double` 형이 입력될 수 있다.

```java
package lang.wrapper.test;

public class WrapperTest2 {
    public static void main(String[] args) {
        String[] array = {"1.5", "2.5", "3.0"};

        // 코드 작성
    }
}
```

실행 결과

```
sum = 7.0
```

정답

```java
package lang.wrapper.test;

public class WrapperTest2 {
    public static void main(String[] args) {
        String[] array = {"1.5", "2.5", "3.0"};
```

```
        double sum = 0;
        for (String s : array) {
            double i = Double.parseDouble(s);
            sum += i;
        }
        System.out.println("sum = " + sum);
    }
}
```

문제3 - 박싱, 언박싱

문제 설명

- `String str`을 `Integer`로 변환해서 출력해라.
- `Integer`를 `int`로 변환해서 출력해라.
- `int`를 `Integer`로 변환해서 출력해라.
- 오토 박싱, 오토 언박싱을 사용하지 말고 직접 변환해야 한다.

```
package lang.wrapper.test;

public class WrapperTest3 {
    public static void main(String[] args) {
        String str = "100";

        // 코드 작성
    }
}
```

실행 결과

```
integer1 = 100
intValue = 100
integer2 = 100
```

정답

```
package lang.wrapper.test;

public class WrapperTest3 {
    public static void main(String[] args) {
        String str = "100";
```

```
        //String -> Integer
        Integer integer1 = Integer.valueOf(str);
        System.out.println("integer1 = " + integer1);

        //Integer -> int
        int intValue = integer1.intValue();
        System.out.println("intValue = " + intValue);

        //int -> Integer
        Integer integer2 = Integer.valueOf(intValue);
        System.out.println("integer2 = " + integer2);
    }
}
```

문제4 - 오토 박싱, 오토 언박싱

문제 설명

- `String str`을 `Integer`로 변환해서 출력해라.
- `Integer`를 `int`로 변환해서 출력해라.
- `int`를 `Integer`로 변환해서 출력해라.
- 오토 박싱, 오토 언박싱을 사용해서 변환해야 한다.

```
package lang.wrapper.test;

public class WrapperTest4 {
    public static void main(String[] args) {
        String str = "100";

        // 코드 작성
    }
}
```

실행 결과

```
integer1 = 100
intValue = 100
integer2 = 100
```

정답

```
package lang.wrapper.test;

public class WrapperTest4 {
    public static void main(String[] args) {
        String str = "100";

        //String -> Integer
        Integer integer1 = Integer.valueOf(str);
        System.out.println("integer1 = " + integer1);

        //Integer -> int
        int intValue = integer1;
        System.out.println("intValue = " + intValue);

        //int -> Integer
        Integer integer2 = intValue;
        System.out.println("integer2 = " + integer2);
    }
}
```

문제와 풀이2

문제 - 로또 번호 자동 생성기

문제 설명

- 로또 번호를 자동으로 만들어주는 자동 생성기를 만들자
- 로또 번호는 1~45 사이의 숫자를 6개 뽑아야 한다.
- 각 숫자는 중복되면 안된다.
- 실행할 때 마다 결과가 달라야 한다.

실행 결과

```
로또 번호: 11 19 21 35 29 16
```

정답

```
package lang.math.test;
```

```java
import java.util.Random;

public class LottoGenerator {
    private final Random random = new Random();
    private int[] lottoNumbers;
    private int count;

    public int[] generate() {
        lottoNumbers = new int[6];
        count = 0;

        while (count < 6) {
            // 1부터 45 사이의 숫자 생성
            int number = random.nextInt(45) + 1;
            // 중복되지 않는 경우에만 배열에 추가
            if (isUnique(number)) {
                lottoNumbers[count] = number;
                count++;
            }
        }
        return lottoNumbers;
    }

    // 이미 생성된 번호와 중복되는지 검사
    private boolean isUnique(int number) {
        for (int i = 0; i < count; i++) {
            if (lottoNumbers[i] == number) {
                return false;
            }
        }
        return true;
    }

}
```

```java
package lang.math.test;

public class LottoGeneratorMain {
    public static void main(String[] args) {
        LottoGenerator generator = new LottoGenerator();
        int[] lottoNumbers = generator.generate();
```

```
        // 생성된 로또 번호 출력
        System.out.print("로또 번호: ");
        for (int lottoNumber : lottoNumbers) {
            System.out.print(lottoNumber + " ");
        }
    }
}
```

정리