6. 스코프, 형변환

#1.인강/0.자바/1.자바-입문

- /스코프1 - 지역 변수와 스코프
- /스코프2 - 스코프 존재 이유
- /형변환1 - 자동 형변환
- /형변환2 - 명시적 형변환
- /계산과 형변환
- /정리

스코프1 - 지역 변수와 스코프

변수는 선언한 위치에 따라 지역 변수, 멤버 변수(클래스 변수, 인스턴스 변수)와 같이 분류된다.
우리가 지금까지 학습한 변수들은 모두 영어로 로컬 변수(Local Variable) 한글로 지역 변수라 한다. 나머지 변수들은 뒤에서 학습한다 지금은 지역 변수만 기억하자.

지역 변수는 이름 그대로 특정 지역에서만 사용할 수 있는 변수라는 뜻이다. 그 특정 지역을 벗어나면 사용할 수 없다. 여기서 말하는 지역이 바로 변수가 선언된 코드 블록(`{}`)이다. 지역 변수는 자신이 선언된 코드 블록(`{}`) 안에서만 생존하고, 자신이 선언된 코드 블록을 벗어나면 제거된다. 따라서 이후에는 접근할 수 없다.

예제 코드로 확인해보자

Scope1

```
package scope;

public class Scope1 {
    public static void main(String[] args) {
        int m = 10; //m 생존 시작
        if (true) {
            int x = 20; //x 생존 시작
            System.out.println("if m = " + m); //블록 내부에서 블록 외부는 접근 가능
            System.out.println("if x = " + x);
        } //x 생존 종료

        //System.out.println("main x = " + x); //오류, 변수 x에 접근 불가
        System.out.println("main m = " + m);
    } //m 생존 종료
```

```
}
```

- `int m`
  - `int m`은 `main{}`의 코드 블록안에서 선언되었다. 따라서 변수를 선언한 시점부터 `main{}`의 코드 블록이 종료될 때 까지 생존한다.
  - `if{}`블록 내부에서도 외부 블록에서 선언된 `m`에 접근할 수 있다. 쉽게 이야기해서 생존 범위만 맞으면 다 접근할 수 있다.
- `int x`
  - `int x`는 `if{}`블록안에서 선언되었다. 따라서 변수를 선언한 시점부터 `if{}`의 코드 블록이 종료될 때 까지 생존한다.
  - `if{}` 내부에서는 자신의 범위에서 선언한 `x`에 당연히 접근할 수 있다.
  - `if{}` 코드 블록이 끝나버리면 `x`는 제거된다. 따라서 더는 `x`에 접근할 수 없다. 따라서 이후에 접근하면 `cannot find symbol`이라는 변수 이름을 찾을 수 없다는 컴파일 오류가 발생한다.

정리하면 지역 변수는 본인의 코드 블록 안에서만 생존한다. 그리고 자신의 코드 블록 안에서는 얼마든지 접근할 수 있다. 하지만 자신의 코드 블록을 벗어나면 제거되기 때문에 접근할 수 없다.

이렇게 변수의 접근 가능한 범위를 스코프(Scope)라 한다. 참고로 Scope를 번역하면 범위라는 뜻이다.
`int m`은 `main{}`전체에서 접근할 수 있기 때문에 스코프가 넓고, `int x`는 `if{}` 코드 블록 안에서만 접근할 수 있기 때문에 스코프가 좁다.

이번에는 `if{}` 대신에 `for{}`를 사용하는 예제를 보자

Scope2

```
package scope;

public class Scope2 {
    public static void main(String[] args) {
        int m = 10;
        for (int i = 0; i < 2; i++) { //블록 내부, for문 내
            System.out.println("for m = " + m); //블록 내부에서 외부는 접근 가능
            System.out.println("for i = " + i);
        } //i 생존 종료

        //System.out.println("main i = " + i); //오류, i에 접근 불가
        System.out.println("main m = " + m);
    }
}
```

`for`문으로 바뀐 것을 제외하면 앞의 예제와 비슷한 예제이다.
`for`문의 경우 `for(int i=0;..)`과 같이 `for`문 안에서 초기식에 직접 변수를 선언할 수 있다. 그리고 이렇게 선

언한 변수는 `for` 문 코드 블록 안에서만 사용할 수 있다.

스코프2 - 스코프 존재 이유

변수를 선언한 시점부터 변수를 계속 사용할 수 있게 해도 되지 않을까? 왜 복잡하게 접근 범위(스코프)라는 개념을 만들었을까?
이해를 위해 다음 코드를 보자

Scope3_1

```
package scope;

public class Scope3_1 {
    public static void main(String[] args) {
        int m = 10;
        int temp = 0;
        if (m > 0) {
            temp = m * 2;
            System.out.println("temp = " + temp);
        }
        System.out.println("m = " + m);
    }
}
```

조건이 맞으면 변수 `m`의 값을 2배 증가해서 출력하는 코드이다. 여기서 2배 증가한 값을 저장해두기 위해 임시 변수 `temp`를 사용했다. 그런데 이 코드는 좋은 코드라고 보기는 어렵다. 왜냐하면 임시 변수 `temp`는 `if` 조건이 만족할 때 임시로 잠깐 사용하는 변수이다. 그런데 임시 변수 `temp` `main()` 코드 블록에 선언되어 있다. 이렇게 되면 다음과 같은 문제가 발생한다.

- **비효율적인 메모리 사용**: `temp`는 `if`코드 블록에서만 필요하지만, `main()`코드 블록이 종료될 때 까지 메모리에 유지된다. 따라서 불필요한 메모리가 낭비된다. 만약 `if` 코드 블록 안에 `temp`를 선언했다면 자바를 구현하는 곳에서 `if` 코드 블록의 종료 시점에 이 변수를 메모리에서 제거해서 더 효율적으로 메모리를 사용할 수 있다.
- **코드 복잡성 증가**: 좋은 코드는 군더더기 없는 단순한 코드이다. `temp`는 `if`코드 블록에서만 필요하고, 여기서만 사용하면 된다. 만약 `if` 코드 블록 안에 `temp`를 선언했다면 `if`가 끝나고 나면 `temp`를 전혀 생각하지 않아도 된다. 머리속에서 생각할 변수를 하나 줄일 수 있다. 그런데 지금 작성한 코드는 `if` 코드 블록이 끝나도 `main()` 어디서나 `temp`를 여전히 접근할 수 있다. 누군가 이 코드를 유지보수 할 때 `m`은 물론이고 `temp`까지 계속 신경써야 한다. 스코프가 불필요하게 넓은 것이다. 지금은 코드가 매우 단순해서 이해하는데 어려움이 없겠지만 실무에서는 코드가 매우 복잡한 경우가 많다.

`temp`의 스코프를 꼭 필요한 곳으로 한정해보자

Scope3_2

```
package scope;

public class Scope3_2 {
    public static void main(String[] args) {
        int m = 10;
        if (m > 0) {
            int temp = m * 2;
            System.out.println("temp = " + temp);
        }
        System.out.println("m = " + m);
    }
}
```

`temp`를 `if` 코드 블록 안에서 선언했다. 이제 `temp`는 `if` 코드 블록 안으로 스코프가 줄어든다. 덕분에 `temp` 메모리를 빨리 제거해서 메모리를 효율적으로 사용하고, `temp` 변수를 생각해야 하는 범위를 줄여서 더 유지보수 하기 좋은 코드를 만들었다.

while문 vs for문 - 스코프 관점

이제 스코프 관점에서 `while`문과 `for`문을 비교해보자

다음 코드들은 기존에 반복문에서 학습했던 코드이다.

While

```
package loop;

public class While2_3 {
    public static void main(String[] args) {
        int sum = 0;
        int i = 1;
        int endNum = 3;

        while (i <= endNum) {
            sum = sum + i;
            System.out.println("i=" + i + " sum=" + sum);
```

```
            i++;
        }
        //... 아래에 더 많은 코드들이 있다고 가정
    }
}
```

For

```
package loop;

public class For2 {
    public static void main(String[] args) {
        int sum = 0;
        int endNum = 3;

        for (int i = 1; i <= endNum; i++) {
            sum = sum + i;
            System.out.println("i=" + i + " sum=" + sum);
        }
        //... 아래에 더 많은 코드들이 있다고 가정
    }
}
```

변수의 스코프 관점에서 카운터 변수 `i`를 비교해보자.

- `while`문의 경우 변수`i`의 스코프가 `main()` 메서드 전체가 된다. 반면에 `for`문의 경우 변수 `i`의 스코프가 for문 안으로 한정된다.
- 따라서 변수 `i`와 같이 `for`문 안에서만 사용되는 카운터 변수가 있다면 `while`문 보다는 `for`문을 사용해서 스코프의 범위를 제한하는 것이 메모리 사용과 유지보수 관점에서 더 좋다.

정리

- 변수는 꼭 필요한 범위로 한정해서 사용하는 것이 좋다. 변수의 스코프는 꼭 필요한 곳으로 한정해서 사용하자. 메모리를 효율적으로 사용하고 더 유지보수하기 좋은 코드를 만들 수 있다.
- **좋은 프로그램은 무한한 자유가 있는 프로그램이 아니라 적절한 제약이 있는 프로그램이다.**

형변환1 - 자동 형변환

형변환

- 작은 범위에서 큰 범위로는 당연히 값을 넣을 수 있다.
  - 예) `int` ➜ `long` ➜ `double`
- 큰 범위에서 작은 범위는 다음과 같은 문제가 발생할 수 있다.
  - 소수점 버림
  - 오버플로우

작은 범위에서 큰 범위로 대입은 허용한다

자바에서 숫자를 표현할 수 있는 범위는 다음과 같다.

`int < long < double`

`int`보다는 `long`이, `long`보다는 `double`이 더 큰 범위를 표현할 수 있다.

작은 범위에서 큰 범위에 값을 대입하는 다음 코드를 실행하면 특별한 문제없이 잘 수행된다.

Casting1

```
package casting;

public class Casting1 {

    public static void main(String[] args) {
        int intValue = 10;
        long longValue;
        double doubleValue;

        longValue = intValue; // int -> long
        System.out.println("longValue = " + longValue); //longValue = 10

        doubleValue = intValue; // int -> double
        System.out.println("doubleValue1 = " + doubleValue); //doubleValue1 =
10.0

        doubleValue = 20L; // long -> double
        System.out.println("doubleValue2 = " + doubleValue); //doubleValue2 =
20.0
    }
}
```

실행 결과

```
longValue = 10
doubleValue1 = 10.0
doubleValue2 = 20.0
```

- 자바는 기본적으로 같은 타입에 값을 대입할 수 있다. 그런데 다른 타입에 값을 대입하면 어떻게 될까?
- `int` ➜ `long`을 비교해보면 `long`이 `int`보다 더 큰 숫자 범위를 표현한다. 작은 범위 숫자 타입에서 큰 범위 숫자 타입에 대입을 하면 문제가 되지 않는다. 만약 이런 경우까지 오류가 발생한다면 개발이 너무 불편할 것이다.
- `long` ➜ `double`의 경우에도 `double`은 부동 소수점을 사용하기 때문에 더 큰 숫자 범위를 표현한다. 따라서 대입할 수 있다.
- 정리하면 작은 범위에서 큰 범위로의 대입은 자바 언어에서 허용한다. 쉽게 이야기하면 큰 그릇은 작은 그릇에 담긴 내용물을 담을 수 있다.

자동 형변환

하지만 결국 대입하는 형(타입)을 맞추어야 하기 때문에 개념적으로는 다음과 같이 동작한다.

```
//intValue = 10
doubleValue = intValue
doubleValue = (double) intValue //형 맞추기
doubleValue = (double) 10 //변수 값 읽기
doubleValue = 10.0 //형변환
```

이렇게 앞에 `(double)`과 같이 적어주면 `int`형이 `double`형으로 형이 변한다. 이렇게 형이 변경되는 것을 형변환이라 한다.

작은 범위 숫자 타입에서 큰 범위 숫자 타입으로의 대입은 개발자가 이렇게 직접 형변환을 하지 않아도 된다. 이런 과정이 자동으로 일어나기 때문에 **자동 형변환**, 또는 **묵시적 형변환**이라 한다.

형변환2 - 명시적 형변환

이번에는 반대로 큰 범위에서 작은 범위로 대입해보자.

큰 범위에서 작은 범위 대입은 명시적 형변환이 필요하다

`double`은 실수를 표현할 수 있다. 따라서 `1.5`가 가능하다. 그런데 `int`는 실수를 표현할 수 없다. 이 경우 `double` ➜ `int`로 대입하면 어떻게 될까?

Casting2

```
package casting;

public class Casting2 {

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

```
        double doubleValue = 1.5;
        int intValue = 0;

        //intValue = doubleValue; //컴파일 오류 발생
        intValue = (int) doubleValue; //형변환
        System.out.println(intValue); //출력:1
    }
}
```

다음 코드의 앞부분에 있는 주석을 풀면(주석을 제거하면) 컴파일 오류가 발생한다.

```
intValue = doubleValue  //컴파일 오류 발생
```

```
java: incompatible types: possible lossy conversion from double to int
//java: 호환되지 않는 유형: double에서 int로의 가능한 손실 변환
```

`int`형은 `double`형보다 숫자의 표현 범위가 작다. 그리고 실수를 표현할 수도 없다. 따라서 이 경우 숫자가 손실되는 문제가 발생할 수 있다. 쉽게 이야기해서 큰 컵에 담긴 물을 작은 컵에 옮겨 담으려고 하니, 손실이 발생할 수 있다는 것이다.

이런 문제는 매우 큰 버그를 유발할 수 있다. 예를 들어서 은행 프로그램이 고객에게 은행 이자를 계산해서 입금해야 하는데 만약 이런 코드가 아무런 오류 없이 수행된다면 끔찍한 문제를 만들 수 있다. 그래서 자바는 이런 경우 컴파일 오류를 발생시킨다. 항상 강조하지만 컴파일 오류는 문제를 가장 빨리 발견할 수 있는 좋은 오류이다.

형변환

하지만 만약 이런 위험을 개발자가 직접 감수하고도 값을 대입하고 싶다면 데이터 타입을 강제로 변경할 수 있다.

예를 들어서 대략적인 결과를 보고 싶은데, 이때 소수점을 버리고 정수로만 보고 싶을 수 있다.

형변환은 다음과 같이 변경하고 싶은 데이터 타입을 `(int)`와 같이 괄호를 사용해서 명시적으로 입력하면 된다.

```
intValue = (int) doubleValue; //형변환
```

이것을 형(타입)을 바꾼다고 해서 형변환이라 한다. 영어로는 캐스팅이라 한다. 그리고 개발자가 직접 형변환 코드를 입력한다고 해서 **명시적 형변환**이라 한다.

캐스팅 용어

"캐스팅"은 영어 단어 "cast"에서 유래되었다. "cast"는 금속이나 다른 물질을 녹여서 특정한 형태나 모양으로 만드는 과정을 의미한다.

명시적 형변환 과정

```
//doubleValue = 1.5
```

```
intValue = (int) doubleValue;
intValue = (int) 1.5; //doubleValue에 있는 값을 읽는다.
intValue = 1; //(int)로 형변환 한다. intValue에 int형인 숫자 1을 대입한다.
```

형변환 후 출력해보면 숫자 `1`이 출력되는 것을 확인할 수 있다.

참고로 형변환을 한다고 해서 `doubleValue` 자체의 타입이 변경되거나 그 안에 있는 값이 변경되는 것은 아니다. `doubleValue`에서 읽은 값을 형변환 하는 것이다. `doubleValue` 안에 들어있는 값은 `1.5`로 그대로 유지된다. 참고로 변수의 값은 대입연산자(`=`)를 사용해서 직접 대입할 때만 변경된다.

형변환과 오버플로우

형변환을 할 때 만약 작은 숫자가 표현할 수 있는 범위를 넘어서면 어떻게 될까?

Casting3

```
package casting;

public class Casting3 {

    public static void main(String[] args) {
        long maxIntValue = 2147483647; //int 최고값
        long maxIntOver = 2147483648L; //int 최고값 + 1(초과)
        int intValue = 0;

        intValue = (int) maxIntValue; //형변환
        System.out.println("maxIntValue casting=" + intValue); //출력:2147483647

        intValue = (int) maxIntOver; //형변환
        System.out.println("maxIntOver  casting=" + intValue); //출력:-2147483648
    }
}
```

출력 결과

```
maxIntValue casting=2147483647
maxIntOver  casting=-2147483648
```

정상 범위

`long maxIntValue = 2147483647`를 보면 `int`로 표현할 수 있는 가장 큰 숫자인 `2147483647`를 입력했다. 이 경우 `int`로 표현할 수 있는 범위에 포함되기 때문에 다음과 같이 `long` ➜ `int`로 형변환을 해도 아무런 문제가 없다.

```
intValue = (int) maxIntValue; //형변환
```

어떻게 수행되는지 확인해보자.

```
maxIntValue = 2147483647; //int 최고값
intValue = (int) maxIntValue; //변수 값 읽기
intValue = (int) 2147483647L; //형변환
intValue = 2147483647;
```

초과 범위

다음으로 `long maxIntOver = 2147483648L` 를 보면 `int` 로 표현할 수 있는 가장 큰 숫자인 `2147483647` 보다 1큰 숫자를 입력했다. 이 숫자는 리터럴은 `int` 범위를 넘어가기 때문에 마지막에 `L` 을 붙여서 `long` 형을 사용해야 한다.

이 경우 `int` 로 표현할 수 있는 범위를 넘기 때문에 다음과 같이 `long` ➜ `int` 로 형변환 하면 문제가 발생한다.

```
intValue = (int) maxIntOver; //형변환
```

어떻게 수행되는지 확인해보자.

```
maxIntOver = 2147483648L; //int 최고값 + 1
intValue = (int) maxIntOver; //변수 값 읽기
intValue = (int) 2147483648L; //형변환 시도
intValue = -2147483648;
```

- 결과를 보면 `-2147483648` 이라는 전혀 다른 숫자가 보인다. `int` 형은 `2147483648L` 를 표현할 수 있는 방법이 없다. 이렇게 기존 범위를 초과해서 표현하게 되면 전혀 다른 숫자가 표현되는데, 이런 현상을 오버플로우라 한다.
- 보통 오버플로우가 발생하면 마치 시계가 한바퀴 돈 것 처럼 다시 처음부터 시작한다. 참고로 `-2147483648` 숫자는 `int` 의 가장 작은 숫자이다.
- 중요한 것은 오버플로우가 발생하는 것 자체가 문제라는 점이다! 오버플로우가 발생했을 때 결과가 어떻게 되는지 계산하는데 시간을 낭비하면 안된다! 오버플로우 자체가 발생하지 않도록 막아야 한다. 이 경우 단순히 대입하는 변수(`intValue`)의 타입을 `int` ➜ `long` 으로 변경해서 사이즈를 늘리면 오버플로우 문제가 해결된다.

계산과 형변환

형변환은 대입 뿐만 아니라, 계산을 할 때도 발생한다.

Casting4

```
package casting;

public class Casting4 {

    public static void main(String[] args) {
        int div1 = 3 / 2;
        System.out.println("div1 = " + div1); //1

        double div2 = 3 / 2;
        System.out.println("div2 = " + div2); //1.0

        double div3 = 3.0 / 2;
        System.out.println("div3 = " + div3); //1.5

        double div4 = (double) 3 / 2;
        System.out.println("div4 = " + div4); //1.5

        int a = 3;
        int b = 2;
        double result = (double) a / b;
        System.out.println("result = " + result); //1.5
    }
}
```

출력 결과

```
div1 = 1
div2 = 1.0
div3 = 1.5
div4 = 1.5
result = 1.5
```

자바에서 계산은 다음 2가지를 기억하자.

1. **같은 타입끼리의 계산은 같은 타입의 결과를 낸다.**
    - `int` + `int` 는 `int` 를, `double` + `double` 은 `double` 의 결과가 나온다.
2. **서로 다른 타입의 계산은 큰 범위로 자동 형변환이 일어난다.**
    - `int` + `long` 은 `long` + `long` 으로 자동 형변환이 일어난다.
    - `int` + `double` 은 `double` + `double` 로 자동 형변환이 일어난다.

다양한 타입별로 더 자세히 들어가면 약간 차이가 있지만 이 기준으로 이해하면 충분하다.

예시를 통해서 자세히 이해해보자.

```
int div1 = 3 / 2; //int / int
int div1 = 1; //int / int이므로 int타입으로 결과가 나온다.
```

```
double div2 = 3 / 2; //int / int
double div2 = 1; //int / int이므로 int타입으로 결과가 나온다.
double div2 = (double) 1; //int -> double에 대입해야 한다. 자동 형변환 발생
double div2 = 1.0; // 1(int) -> 1.0(double)로 형변환 되었다.
```

```
double div3 = 3.0 / 2; //double / int
double div3 = 3.0 / (double) 2; //double / int이므로, double / double로 형변환이 발생한다.
double div3 = 3.0 / 2.0; //double / double -> double이 된다.
double div3 = 1.5;
```

```
double div4 = (double) 3 / 2; //명시적 형변환을 사용했다. (double) int / int
double div4 = (double) 3 / (double) 2; //double / int이므로, double / double로 형변환이 발생한다.
double div4 = 3.0 / 2.0; //double / double -> double이 된다.
double div4 = 1.5;
```

`3 / 2`와 같이 `int`형끼리 나눗셈을 해서 소수까지 구하고 싶다면 `div4`의 예제처럼 명시적 형변환을 사용하면 된다.

물론 변수를 사용하는 경우에도 다음과 같이 형변환을 할 수 있다.

```
int a = 3;
int b = 2;
double result = (double) a / b;
```

처리 과정

```
double result = (double) a / b; //(double) int / int
double result = (double) 3 / 2; //변수 값 읽기
double result = (double) 3 / (double) 2; //double + int 이므로 더 큰 범위로 형변환
double result = 3.0 / 2.0; //(double / double) -> double이 된다.
double result = 1.5;
```

정리

형변환

`int` ➜ `long` ➜ `double`

- 작은 범위에서 큰 범위로는 대입할 수 있다.
  - 이것을 묵시적 형변환 또는 자동 형변환이라 한다.
- 큰 범위에서 작은 범위의 대입은 다음과 같은 문제가 발생할 수 있다. 이때는 명시적 형변환을 사용해야 한다.
  - 소수점 버림
  - 오버플로우
- 연산과 형변환
  - 같은 타입은 같은 결과를 낸다.
  - 서로 다른 타입의 계산은 큰 범위로 자동 형변환이 일어난다.