정리

#형변환

byte - short - int - long - || float - double

8 - 16 - 32 - 64 - || 32 - 64실수, 소수점까지 포함

double - float - long - int - short - byte
형변환 해줘야함. 범위가 더 큰 타입에 들어가는건 가능하지만,
반대의 경우에.

** 형변환을 해야하는 이유 : 작은 데이터타입은 작은 용량을 가져 메모리를 적게 먹는다. 비용절감.

그래서, 필요할때만 형변환을 해서 처리해주는게 이득일때 사용

```
short a1 =10;
byte a2 = a1;   --->  (byte) a1;
```

큰 그릇의 타입 안에 담겨있는 수를 작은 그릇에서 수용가능하고, 사용하기 위해서는 형변환 ! 조건은 말한대로 큰 그릇에 담겨있던 값이라고 하더라도 형변환 하려는 작은 그릇의 타입 범위 안에 들어가야함.

```
integer.parseInt()

String a1 = "10";
int i = (int)a1;  // ---> x 틀림
int i = Integer.parseInt(a1);
```

으로 해야 형변환 가능. 스트링 문자에서 인트로 바꾸려면.
텍스트를 읽고 데이터를 파싱. 사실, 문자열과 숫자 타입은 호환불가.

float b = 10.0f;--> 부동소수점
int i = (int)b;

형변환 시 같은 괄호로 묶어준다.

```
int a = 25;
float b = 42.159f;
int value = (int)(a*a + 2*(a*b) + b*b);
```

이렇게 형변환이 필요한 추가 이유

= 데이터 타입이 변수가 저장되기도 하지만, 메모리에서 차지하는 공간을 지정하기 때문에. 메모리 할당에 직접적인 영향을 미침. 그릇이 담길 곳의 크기인 메모리에 얼만큼의 공간을 요구하는지 이게 정함.

왜 float num = 11.0f;
라고 끝에 f를 처리해주나?

📌 자바의 실수 리터럴 기본 규칙

- `11.0` → 기본적으로 **double 타입** (8바이트, 64비트 부동소수점)
- `11.0f` → **float 타입** (4바이트, 32비트 부동소수점)
- `11.0d` → **double 타입** (사실 기본이 double이기 때문에 `d`는 생략 가능)

즉, 자바에서 **소수점이 있는 숫자(실수 리터럴)는 기본적으로 double로 처리**돼.

📌 왜 오류가 나는가?

예를 들어:

`float f = 11.0; // 오류 발생`

👉 이유: `11.0`은 double인데, float은 double보다 **더 작은 범위**라서 자동 변환이 안 돼. (손실 가능성 때문에 자바는 엄격하게 막음)

따라서 이렇게 해야 함:

```
float f = 11.0f; // OK
```

📌 double은 왜 괜찮나?

```
double d1 = 11.0; // OK (기본 double이니까) double d2 = 11.0d; // OK (명시적으로 double 표시, 사실 안 붙여도 됨)
```

- `11.0` 자체가 원래 double 타입이므로 에러 없음.
- `d` 접미사는 있어도 되고 없어도 됨 (가독성 때문에 붙이는 경우 있음).

📌 정리

- `11.0` → double (기본값)
- `11.0d` → double (명시적 표기, 생략 가능)
- `11.0f` → float (안 붙이면 에러)

📌 관련 키워드

- **primitive type**: `float`(4byte), `double`(8byte)
- **precision(정밀도)**: float ≈ 7자리, double ≈ 15~16자리
- **type casting**: `(float) 11.0`처럼 강제 변환 가능
- **literal suffix**: `f`, `d`, `l` (long), `u`(C 언어), 등

👉 요약: **자바에서 실수 리터럴은 기본 double이라서, float에 넣으려면 `f`를 붙여야 하고, double은 원래 기본값이니까 `d`는 안 붙여도 되지만 붙일 수도 있다.**

#for문

for문으로 작업할 때,

```
for(i = 0; i <= 10 ; i++){}
```

이런 식을

```
int i = 0;
for(;  ; i++){
    if(i >= 10){}
    break;
}
```

이렇게 밖으로 조건식을 빼내면,

i <= 10; 과는 다르게 if(i >= 10)으로 바꿔줘야 하는데,
조건문에서는 i가 해당 조건 이전까지 실행해야 되는거고,
if문에서는 i가 해당 조건이 되면 멈춰야 하기 때문.

풀어보면, i가 11 이 되는 순간 루프를 멈추고 나가. 그러니 10 까지 숫자가 오르며 카운트

if에서는 i가 10보다 커지면 나가야 하니

**해당 조건이 될 때까지 진행하다 멈춰 / 해당 조건이 되면 break로 나가

왜 부등호 방향이 바뀌었을까?

- **for문의 조건식(`i <= 10`)**은 **"반복을 계속할 조건"**을 뜻해.
  즉, `i <= 10`일 동안만 루프가 돈다는 뜻이지.
- 그런데 **if문 안으로 조건을 옮기면** 의미가 반대가 돼:
  - for는 **조건이 true면 실행 계속**

- if는 **조건이 true면 실행 중단하는 로직(break)**으로 짜야 하니까
- 따라서 `i <= 10` → `i >= 10` 으로 “부정”이 되는 거야.

즉, **루프 유지 조건**을 **루프 종료 조건**으로 바꿨기 때문에 부등호 방향이 뒤집힌 거지.

✅ 결론:
부등호가 바뀐 게 "이상한 현상"이 아니라,
for문의 조건(반복 유지) ↔ if문의 조건(반복 종료)로 옮기면서 **논리 방향이 반전된 것**이야.

조건이 맞으면 들어와라 vs 조건이 맞으면 나가라
(10보다 작으면) (10보다 커지면)

for (i = 0 ; i <= 10 ; i++;)

에서 i++를 ++i로 바꿔도, 0부터 시작하는데
이유는, for문에서 조건에 맞으면 통과 통과 시키고, 3번째 조건은 전체 문장이 1번 통과한 후 부터 적용되는 시스템.
가장 초기조건인 i=0 역시 만족하며 로그를 남김.

#switchcase

```
package toyProject;

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

        int x = 2;

        switch (x){
            case 1 :
                System.out.println("X is 1");
            case 2 :
                System.out.println("X is 2");
                break;
            case 3 :
                System.out.println("X is 3");
            case 4 :
                System.out.println("X is 4");
        }

        System.out.println();
    }
}
```

여기서 인터프리터 형식, break가 없으면, x의 값이 2면, 2만 선택하는것이 아닌, 2 이후의 모든 것을 타고 내려온다. 따라서, break가 존재.

Lamda 표기법

```
    public static void main(String[] args) {
        int x = 1;

       switch (x) {
           case 1 -> System.out.println("X is 1");
           case 2 -> System.out.println("X is 2");
           case 3 -> System.out.println("X is 3");
           default -> System.out.println("X is other than 1,2,3");
       }
    }
```

-> 가 람다 표기법. 더 깔끔하고, 기존의 케이스 문법보다 가독성 있다.
복잡성을 줄이고, break의 사용을 줄일 수 있다.

```
    public static void main(String[] args) {
        char x = 'b';

       switch (x) {
           case 'a' -> System.out.println("X is a");
           case 'b' -> System.out.println("X is b");
           case 'c' -> System.out.println("X is c");
           default -> System.out.println("X is other than a,b,c");
       }
    }
```

JDK 14 이상

```java
 package toyProject;

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

        char x = B;

        switch (x){
            case 'a' :
                System.out.println("X is a");
                break;
            case 'b' :
                System.out.println("X is b");
                break;
            case 'c' :
                System.out.println("X is c");
                break;
            case 'd' :
                System.out.println("X is d");
                break;
        }

        System.out.println();
    }
}
```

여기서 문제가 생긴다. java는 변수의 대문자와 소문자를 구분하기 때문에,
b와 B는 다르다.

```java
 package toyProject;

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

```
        char x = B;

        switch (x){
            case 'a' :
            case 'A' :
                System.out.println("X is a");
                break;
            case 'b' :
            case 'B' :
                System.out.println("X is b");
                break;
            case 'c' :
            case 'C' :
                System.out.println("X is c");
                break;
            case 'd' :
            case 'D' :
                System.out.println("X is d");
                break;
        }

        System.out.println();
    }
}
```

이런식으로 해야하지만, 가독성이 매우 떨어진다. 따라서,
java.lang에서 toLowerCase()함수 호출

```
    public static void main(String[] args) {
        String x = "B";

        switch (x.toLowerCase()) {
            case "a" -> System.out.println("X is a");
```

```
            case "b" -> System.out.println("X is b");
            case "c" -> System.out.println("X is c");
            default -> System.out.println("X is other than a,b,c");
        }
    }
```

대문자를 소문자로 바꿔주는 케이스. 이 때, String으로 해주면, 변수와 매개변수를 쓰는 함수를 호출할 때 더 편리하다. 단, "도 ""로 바꿔줘야 한다.

맞아, **char랑 String은 타입이 다르니까 → 대소문자 변환 같은 함수 호출 방식도 다름**.

📌 char의 경우

- `char`는 **원시 타입(primitive type)** → 객체가 아님.
- 그래서 `x.toLowerCase()` 같은 메서드 직접 호출은 불가능.
- 대신 **Character 클래스(Wrapper 클래스)**의 static 메서드를 써야 함. `char x = 'B'; char y = Character.toLowerCase(x); // 'B' → 'b'`

📌 String의 경우

- `String`은 **클래스(객체)** → 인스턴스 메서드를 바로 호출 가능. `String x = "B"; String y = x.toLowerCase(); // "B" → "b"`

📌 차이 정리

타입	리터럴 표기	대소문자 변환 방법	switch에서 case
`char`	`'a'`, `'B'`	`Character.toLowerCase(x)`	`'a'` `'b'` `'c'`
`String`	`"a"`, `"B"`	`x.toLowerCase()`	`"a"` `"b"` `"c"`

👉 즉,

- `char`는 **함수 호출 = Character 클래스의 static 메서드**
- `String`은 **함수 호출 = 인스턴스 메서드**

✅ 결론:

함수 호출 방식이 다르다

(char는 static 메서드 / String은 객체 메서드)

int, char, String 사용 가능

#뒤에f를붙이는이유

```java
public class Main {
    public static void main(String[] args) {
        int number = 11;
        float value = 11.0f;
        System.out.println("Number: " + number);
    }
}
```

f를 붙여서 float타입으로 인식. 기본적으로 소수점이 있는 리터럴을 double로 인식해서 float이라면, 뒤에 f로 마무리.