• 이것도 모르고 넘어가면 안되지
개념은 확실히!!! 짚고 넘어갈 것

민솔이의 코딩 백과사전

HTML5, CSS3, JAVASCRIPT, JAVA

By Jang Minsol

What? Why? How?
그리고 tips or pitfall

JAVA

📘 Chapter 02. 변수

1. 변수의 선언과 초기화

변수란?

✅ what

변수(Variable)는 데이터를 저장하는 상자,

상자에 담긴 데이터는 변할 수 있고 재사용이 가능하다.

✅ why

- 사용자 입력값 저장
- 조건, 반복문에서 값 비교가능
- 계산 결과 저장

✅ how

```
int a; // 데이터타입 변수
int a = 3; // 데이터 타입 변수 = 값
```

2. 데이터 타입(Data type)

✅ what

변수에 어떤 종류의 값이 들어올지 명확히 선언하는 것

✅ why

타입을 정확히 명시해야 컴파일러가 오류를 사전에 잡을 수 있음

잘못된 타입 연산을 막고 코드 안정성 확보

✅ how

자바의 데이터 타입은 크게 2가지로 나뉨

1. 기본형(primitive Type)

→ 값 자체를 저장함

타입	설명	크기	예시
`byte`	정수	1 byte	-128 ~ 127
`short`	정수	2 byte	-32,768~32,767
`int`	정수 (기본)	4 byte	-2^31 ~ 2^31-1
`double`	정수 (더 큰 수)	8 byte	숫자 뒤에 `L` 붙이기
`float`	실수	4 byte	숫자 뒤에 `f` 붙이기
`double`	실수 (기본)	8 byte	소수점 포함
`char`	문자 1개	2 byte	`'A'` , `'민'`
`boolean`	논리	1 bit	`true` , `false`

2. 참조형(Reference Type)

→ 객체의 주소를 저장함 (변수는 그것을 참조함!!)

→ 배열, 클래스 `String` , `Scanner` 등

```
String name = "민솔";
int[] scores = new int[5];
Scanner sc = new Scanner(System.in);
```

⚠️ Tip or Pitfall

✅ 기본형 vs 참조형 차이

항목	기본형	참조형
저장 방식	값 자체 저장	객체 주소 저장
비교 시	`==` 가능	`equals()` 써야함
예시	`int a = 5;`	`String s = "abc";`

✅ 실수형 주의

- `float f = 3.14;` ❌
- `float f = 3.14f;` ✅ (f 붙여야함)

✅ 문자 vs 문자열

- `'A'` → `char`
- `"A"` → `String` (객체)

✅ boolean은 `true` , `false` 만 가능

- `boolean b = 1;` ❌
- `boolean b = true;` ✅

✅ JAVA는 기본적으로 `int` , `double` 을 많이 씀

3. 형 변환

✅ what

형 변환(Type Casting)은

→ 어떤 타입의 데이터를 다른 타입으로 바꾸는 것

예) `double → int` , `int → String` , `char → int` 등

✅ why

- 서로 다른 타입끼리 연산 불가
- 타입을 일치 시켜서 연산, 비교, 출력, 저장 등이 필요할 때 사용

✅ how

1. 자동 형 변환 (묵시적 변환)

- 작은 타입에서 큰 타입으로 갈 때
- 데이터 왜곡 없을 때

```
int a = 10;
double b = a;  // int → double OK (자동 형변환)
System.out.println(b); // 10.0
```

2. 강제 형 변환 (명시적 변환)

- 데이터 왜곡이 생기더라도 해야 할 경우 자동으로 안 해주니까 내가 허락 한 거임~ 하고 쓰는 거

```
double d = 3.14;
int i = (int) d; // 소수점 버림
System.out.println(i); // 3
```

```
char c = (char) 65;  // 65 → 'A'
System.out.println(c); // A
```

3. 문자 ↔ 숫자 형 변환

```
char ch = 'A';
int num = (int) ch;  // 문자 → 아스키코드
System.out.println(num); // 65
```

```
int n = 66;
char ch2 = (char) n; // 숫자 → 문자
System.out.println(ch2); // 'B'
```

4. 문자열 ↔ 숫자 변환

```
int num = Integer.parseInt("123"); // 문자열 → 정수
String str = String.valueOf(456);  // 정수 → 문자열
```

```
double d = Double.parseDouble("3.14");
```

✅ Tip or Pitfall

- `int → double` 은 되지만, `double → int` **는 반드시** `(int)` **붙여야 함**
- 소수점 날아가는 거 주의 (`(int)3.99` → 3)
- `boolean` 은 어떤 타입하고도 형변환 **절대 안 됨**
- 문자열 숫자 변환할 때 `"12a"` 이런 거 들어오면 `NumberFormatException` **터짐**
- `char` 는 숫자와 변환 가능 (ASCII 기반)

🎯 자주 쓰이는 형변환 공식

목적	코드
정수 → 실수	`double d = (double) x;`
실수 → 정수	`int i = (int) d;`
문자 → 숫자	`int code = (int) 'A';`
숫자 → 문자	`char c = (char) 65;`

목적	코드
문자열 → 정수	`Integer.parseInt("123")`
정수 → 문자열	`String.valueOf(123)`

📘 Chapter 03. 연산자

1. 연산자의 종류와 우선순위

✅ what

자바에서 연산자(Operator)는 **값들 간의 계산, 비교, 조합 등을 해주는 기호**

→ 종류에 따라 **우선순위(Priority)** 가 존재

✅ why

- 계산 순서를 정확히 지정하지 않으면 **결과가 달라짐**
- `a + b * c` → 어떤 연산이 먼저냐에 따라 값이 바뀜

✅ How

우선순위 (높음 → 낮음)	예시
괄호	`( )`
단항 연산자	`+`, `-`, `++`, `--`, `!`
산술 연산자	`*`, `/`, `%`, `+`, `-`
비교 연산자	`<`, `>`, `==`, `!=`
논리 연산자	`&&`, `
조건(삼항) 연산자	`조건 ? A : B`
대입 연산자	`=`, `+=`, `-=` 등

✅ Tip or Pitfall

- 항상 **괄호로 우선순위 명확히** 해주는 게 좋음
- `a + b * c` vs `(a + b) * c` 완전 다름
- `&&`는 `&` 아님! `||`는 `|` 아님!

2. 산술 연산자

✅ What

숫자 간의 **기본적인 계산**을 할 때 사용하는 연산자

✅ Why

모든 수학적 계산, 누적 합, 평균, 나머지 계산 등

→ 프로그램의 계산 로직에 반드시 사용됨

✅ How

```
int a = 10, b = 3;
System.out.println(a + b); // 13
System.out.println(a - b); // 7
System.out.println(a * b); // 30
System.out.println(a / b); // 3 (정수 나눗셈)
System.out.println(a % b); // 1 (나머지)
```

✅ Tip or Pitfall

- `int / int`는 **소수점 버려짐**
- 실수로 계산하려면 `double` 사용 or `(double)a / b` 형변환 필요

3. 단항 연산자

1. 부호 연산자(+, -)

✅ What

값의 부호를 바꾸는 단항 연산자 (`+`, `-`)

✅ Why

음수/양수 전환 또는 코드 가독성 위해 사용

✅ How

```
int x = 5;
int y = -x; // y = -5
int z = +x; // z = 5
```

✅ Tip or Pitfall

- `+` 연산자는 아무 의미 없음 (단지 명시적 표현일 뿐)
- `(-x)` → 양수로 돌아옴

2. 논리 부정 연산자(!)

✅ What

`!`는 **boolean 값을 반대로 바꾸는 연산자**

✅ Why

조건을 반전시켜서 제어 흐름을 만들 때 사용

✅ How

```
boolean flag = true;
System.out.println(!flag); // false
```

```
if (!isAvailable)
{
    System.out.println("사용 불가 상태");
}
```

✅ Tip or Pitfall

- `!true` → `false`, `!false` → `true`
- 조건문에서 너무 남발하면 **가독성 떨어짐**, 괄호로 명확히

3. 증감 연산자(++, - -)

✅ What

변수의 값을 **1 증가 또는 감소**시키는 연산자

✅ Why

- 반복문에서 인덱스 조작
- 카운팅 변수 조절
- ++i, i++ 차이 구분도 중요!
-

✅ How

```
int i = 1;
i++;  // 후위 증가 → i = 2
++i;  // 전위 증가 → i = 3
i--;  // 후위 감소 → i = 2
--i;  // 전위 감소 → i = 1
```

```
int a = 5;
System.out.println(a++); // 5 (출력 후 증가)
System.out.println(++a); // 7 (증가 후 출력)
```

✅ Tip or Pitfall

- 후위형(`i++`)은 **먼저 쓰고 나중에 증가**
- 전위형(`++i`)은 **먼저 증가하고 나중에 씀**
- 반복문에서 실수로 `-`나 `++` 빠지면 **무한루프됨**

4. 비교 연산자

1. 대소비교 연산자

✅ What

두 값을 비교하여 **크다/작다 여부를 판단하는 연산자**

✅ Why

조건문, 필터링, 정렬 등 거의 모든 비교 상황에서 사용됨

✅ How

```
int a = 10, b = 20;

System.out.println(a < b);  // true
System.out.println(a >= b); // false
```

✅ Tip or Pitfall

- **boolean 결과가 나온다** (true/false)

- `>=`, `<=`는 **순서 중요** (`=>` ❌)

2. 등가비교 연산자

✅ What

두 값이 같은지(`==`), 다른지(`!=`)를 비교

✅ Why

조건 판별에서 가장 자주 등장하는 연산자

✅ How

```
int x = 3;
int y = 3;

System.out.println(x == y); // true
System.out.println(x != y); // false
```

✅ Tip or Pitfall

- `==`는 **값 비교**
- 문자열은 `==` ❌ → `"abc".equals("abc")`로 비교
- `=`(대입)랑 헷갈리면 죽음! `if (a = 3)` 이런 실수 자주 나옴

5. 논리 연산자

✅ What

논리 조건을 **AND/OR/NOT** 연산으로 결합하는 연산자

✅ Why

복잡한 조건문 만들 때 필수.

→ 여러 조건 동시에 체크할 수 있음

✅ How

```
int age = 25;
boolean isAdult = (age >= 20 && age < 65); // true

boolean condition = (age < 10 || age > 70); // false

boolean notFlag = !isAdult; // false
```

연산자	의미
`&&`	AND (둘 다 true여야 true)
`||`	OR (둘 중 하나만 true 여도 true)
`!`	NOT (true ↔ false 반전)

✅ Tip or Pitfall

- `&&` → 앞이 false면 뒤는 아예 계산 안 함 (short-circuit)
- `||` → 앞이 true면 뒤 생략
- 조건식은 괄호 써서 **명확히 하자**

6. 기타 연산자

1. 삼항 연산자

✅ What

조건식 ? 참일 때 값 : 거짓일 때 값

→ if보다 간결하게 값을 판단해서 리턴할 때 씀

✅ Why

- 한 줄로 조건 분기 가능
- 변수에 값을 조건에 따라 할당할 때 자주 씀

✅ How

```
int score = 85;
String grade = (score >= 90) ? "A" : "B";
System.out.println(grade); // B
```

✅ Tip or Pitfall

- 복잡한 조건은 **삼항보다 if문이 가독성 좋음**
- 중첩 삼항은 ❌ 피하자

2. 대입 연산자

✅ What

변수에 값을 저장하거나, 연산 후 저장하는 연산자

✅ Why

- 반복되는 수식 줄이기
- 누적, 감소, 곱셈 계산할 때 코드 간결해짐

✅ How

```
int x = 10;
x += 5;  // x = x + 5
x *= 2;  // x = x * 2
x -= 3;  // x = x - 3
```

연산자	의미
`=`	대입
`+=`	더해서 대입
`-=`	빼서 대입
`*=`	곱해서 대입
`/=`	나눠서 대입
`%=`	나머지 대입

✅ Tip or Pitfall

- `=`는 **대입**, `==`는 **비교**
- 헷갈리면 조건식이 의도대로 안 먹힘

📘 Chapter 04. 제어문

1. 조건문

1. if문

✅ What

조건이 true일 때만 코드 블록 실행

✅ Why

프로그램 흐름을 분기시키기 위해 사용

✅ How

```
int age = 20;
if (age >= 18)
```

```
{
    System.out.println("성인입니다");
}
```

✅ Tip or Pitfall

- 중괄호 안 써도 한 줄은 되지만, **항상 쓰는 습관 들이자**
- 조건식은 boolean이 나와야 함

2. if-else 문

✅ What

조건이 true면 if 블록, false면 else 블록 실행

✅ Why

상반된 두 상황 중 하나를 선택해야 할 때

✅ How

```
if (age >= 20)
{
    System.out.println("성인");
}
else
{
    System.out.println("미성년자");
}
```

✅ Tip or Pitfall

- `if` 블록 끝에 `else` 붙이면 무조건 양자택일

- `else`는 단독 사용 불가, 항상 `if` 뒤에 붙음

3. if-else if 문

✅ What

여러 조건 중 하나를 선택해서 실행할 수 있는 조건문

✅ Why

조건이 여러 개일 때 하나하나 `if`로 쓰는 대신 **순차적 판단 가능**

✅ How

```
int score = 75;

if (score >= 90)
{
   System.out.println("A");
}
else if (score >= 80)
{
   System.out.println("B");
}
else if (score >= 70)
{
   System.out.println("C");
}
else
{
   System.out.println("F");
}
```

✅ Tip or Pitfall

- 위에서부터 순차적으로 조건 검사, **처음으로 참인 블록만 실행됨**
- 조건 순서 바뀌면 결과 달라짐 → 큰 조건 먼저!

4. 중첩 if 문

✅ What

if문 안에 또 if문을 넣는 구조

✅ Why

두 단계 이상의 조건이 필요할 때 유용

✅ How

```
int age = 25;
boolean isStudent = true;

if (age >= 20)
{
    if (isStudent)
    {
        System.out.println("대학생 성인");
    }
}
```

✅ Tip or Pitfall

- 들여쓰기 안 하면 **가독성 폭망**
- 중첩이 깊으면 → `switch`, `return` 분기 등으로 개선 고려

5. Switch문

✅ What

값이 명확하게 나눠질 때 (`정수`, `문자`, `문자열`) 간단히 분기 가능

✅ Why

많은 `==` 비교를 if-else로 쓰면 **가독성 나빠짐**

→ 이럴 때 `switch` 사용

✅ How

```
int menu = 2;

switch (menu)
{
   case 1:
      System.out.println("한식");
      break;
   case 2:
      System.out.println("중식");
      break;
   case 3:
      System.out.println("양식");
      break;
   default:
      System.out.println("없는 메뉴");
}
```

✅ Tip or Pitfall

- `break` **필수**, 안 쓰면 아래 case까지 연달아 실행됨 (fall-through)
- `default`는 else랑 비슷

2. 반복문

1. for문

✅ What

정해진 횟수만큼 반복 실행하는 대표적인 반복문

✅ Why

배열 순회, 반복 계산, 카운트 등 모든 반복 로직에 핵심

✅ How

```
for (int i = 1; i <= 5; i++)
{
   System.out.println("민솔 최고! " + i);
}
```

구성	설명
초기식	반복 변수 선언 (`int i = 0`)
조건식	반복할 조건 (`i < 5`)
증감식	반복 후 증가/감소 (`i++`)

✅ Tip or Pitfall

- 조건식 생략 or 잘못 설정하면 **무한루프 가능**
- 배열과 함께 쓸 때는 항상 `i < arr.length` 체크!
- `for(;;)` → 무한루프

2. while문

✅ What

조건이 true일 동안 반복하는 루프

✅ Why

반복 횟수가 정해지지 않았거나,

조건 만족할 때까지 계속 돌릴 때 적합

✅ How

```
int i = 1;
while (i <= 5)
{
    System.out.println("민솔 화이팅! " + i);
    i++;
}
```

✅ Tip or Pitfall

- 조건이 false면 **한 번도 실행 안 됨**
- `i++` 빼먹으면 **무한루프**

3. do~while문

✅ What

조건 상관없이 최소 1번은 실행되는 반복문

✅ Why

사용자 입력받기처럼, **일단 한 번은 무조건 처리해야 할 때** 사용

✅ How

```
int i = 10;
do
{
```

```
    System.out.println("무조건 1번 실행됨!");
}
while (i < 5);
```

✅ Tip or Pitfall

- `while`과는 달리, **조건이 false여도 1회 실행됨**
- 자주 쓰진 않지만, 쓰일 땐 꼭 써야 함!

4. break문, continue문

✅ What

- `break` : 반복문 **강제 종료**
- `continue` : 해당 반복 **건너뛰고 다음 반복 진행**

✅ Why

- 조건에 따라 반복 조절 가능
- 무한루프 탈출, 필터링 시 유용

✅ How

```
for (int i = 1; i <= 10; i++)
{
    if (i == 5) break;
    System.out.println(i);
}
// 1~4까지만 출력

for (int i = 1; i <= 5; i++)
{
    if (i == 3) continue;
    System.out.println(i);
```

```
}
// 1 2 4 5 출력 (3 건너뜀)
```

✅ Tip or Pitfall

- 중첩 루프에서 쓸 땐 **어디서 멈추는지** 정확히 체크해야 함
- `while(true)` + `break` → 자주 쓰는 **무한 반복 탈출 패턴**

📘 Chapter 05. 배열

1. 배열

✅ What

같은 타입의 여러 값을 연속으로 저장하는 자료구조

✅ Why

여러 데이터를 한 번에 처리하거나,

인덱스로 값 접근할 수 있는 효율적인 구조

✅ How

```
int[] scores = new int[5];           // 길이 5짜리 빈 배열
int[] nums = {10, 20, 30, 40, 50};   // 초기화와 선언 동시에
System.out.println(nums[2]);         // 30
```

✅ Tip or Pitfall

- 인덱스는 **0부터 시작**

- `ArrayIndexOutOfBoundsException` 주의!
- `.length`는 **속성**이라 괄호 안 씀: `arr.length`

2. 배열의 사용

1. 배열의 선언과 생성

✅ What

배열을 사용하기 위해서는 **먼저 자료형과 크기를 선언하고, 공간을 생성해야 함**

✅ Why

메모리에 **연속된 공간 확보** → 여러 값을 효율적으로 다룰 수 있음

✅ How

```
int[] arr = new int[5];      // 크기 5짜리 int 배열 생성
String[] names = new String[3];  // 문자열 배열
```

또는 선언과 동시에 초기화

```
int[] nums = {1, 2, 3, 4, 5};
```

✅ Tip or Pitfall

- 배열 크기를 정해두면 **수정 불가**
- `arr[0] = 10;`처럼 인덱스 지정해서 할당해야 함
- 배열도 참조형임 (주소 저장)

2. 배열의 길이와 인덱스

✅ What

- **길이**: 배열 안에 몇 개의 요소가 들어가는지
- **인덱스**: 배열의 각 요소 위치 (0부터 시작)

✅ Why

배열은 인덱스를 기준으로 값을 저장하고 꺼내기 때문에

→ **인덱스 계산은 핵심 로직의 기본**

✅ How

```
int[] arr = {10, 20, 30, 40};
System.out.println(arr.length);  // 4
System.out.println(arr[2]);      // 30
```

✅ Tip or Pitfall

- `arr.length`는 속성 (메서드 아님)
- 인덱스 범위 초과 시 → `ArrayIndexOutOfBoundsException`

3. 배열의 초기화와 출력

✅ What

배열 초기화는 **초기값을 설정**하는 과정

→ 출력은 반복문을 통해 각 요소를 꺼내는 방식

✅ Why

초기화 안 하면 기본값으로 채워짐 (`int`는 0, `boolean`은 false)

✅ How

```
int[] arr = new int[3];
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;

for (int i = 0; i < arr.length; i++)
{
   System.out.println(arr[i]);
}
```

3. 다차원 배열

1. 2차원 배열의 선언과 생성

✅ What

행(row)과 열(column)이 있는 **테이블 형태의 배열**

✅ Why

좌표, 행렬, 표 형태의 데이터 저장에 필수

→ 예: 구구단, 게임 맵, 퍼즐판 등

✅ How

```
int[][] matrix = new int[2][3];  // 2행 3열
matrix[0][0] = 1;
matrix[1][2] = 9;
```

✅ Tip or Pitfall

- `int[][] arr = new int[3][];` 가능 (행은 고정, 열은 가변)
- 중첩 for문 필수

```
for (int i = 0; i < matrix.length; i++)
{
  for (int j = 0; j < matrix[i].length; j++)
  {
    System.out.print(matrix[i][j] + " ");
  }
  System.out.println();
}
```

2. 2차원 배열의 인덱스

✅ What

2차원 배열에서 **값에 접근할 때 사용하는 인덱스** → `arr[행][열]` 구조

✅ Why

배열의 각 위치에 접근하려면 반드시 인덱스가 필요

→ 행과 열을 지정해서 정확히 원하는 데이터 꺼냄

✅ How

```
int[][] grid =
{
  {1, 2, 3},
  {4, 5, 6}
};

System.out.println(grid[0][1]); // 2
System.out.println(grid[1][2]); // 6
```

✅ Tip or Pitfall

- 항상 `arr[행][열]` 순서임
- 범위 벗어나면 `ArrayIndexOutOfBoundsException`
- `.length`는 행 기준: `arr.length`

  열은 개별 행마다 다를 수 있음: `arr[i].length`

3. 2차원 배열의 초기화와 출력

✅ What

2차원 배열에 **초기값을 넣고 전체를 출력하는 방법**

✅ Why

표 형태의 데이터 시각화, 디버깅, 반복 처리 등에 자주 사용됨

✅ How

```
int[][] table =
{
  {1, 2, 3},
  {4, 5, 6}
};

for (int i = 0; i < table.length; i++)
{
  for (int j = 0; j < table[i].length; j++)
  {
    System.out.print(table[i][j] + " ");
  }
  System.out.println();
}
```

✅ Tip or Pitfall

- 행이 먼저, 열이 나중! (`i`, `j` 순서)
- `System.out.println()`으로 줄바꿈 필수
- 2차원 배열은 내부적으로 **1차원 배열을 여러 개 갖는 배열**

4. 배열 다루기

1. foreach

✅ What

배열 또는 컬렉션의 **모든 요소를 쉽게 순회**하는 반복문

✅ Why

인덱스 필요 없을 때 → 코드 간결 + 오류 적음

✅ How

```
int[] nums = {10, 20, 30};

for (int num : nums)
{
  System.out.println(num);
}
```

2차원 배열에서도 사용가능

```
for (int[] row : table)
{
  for (int val : row)
  {
    System.out.print(val + " ");
  }
  System.out.println();
}
```

✅ Tip or Pitfall

- **인덱스 접근이 필요하면 사용❌**
- 값 변경하려 해도 실제 배열 값은 바뀌지 않음 (복사본 순회임!)

📘 Chapter 06. 클래스에 대하여

1. 객체지향 언어란?

✅ What

현실 세계를 모델링하여

데이터(속성)와 기능(행동)을 **클래스와 객체 단위로 나누는 프로그래밍 패러다임**

✅ Why

- 코드 재사용성 (클래스 재활용)
- 유지보수성 (기능 분리)
- 확장성 (상속, 다형성 등)

  → 실무, 대형 프로젝트에서 반드시 필요한 구조

✅ 4대 핵심 특징

특징	설명
추상화	복잡한 것을 단순한 모델로 표현
캡슐화	데이터 보호, 접근 제한 (`private` , `getter/setter`)
상속	기존 클래스를 확장해서 새로운 클래스 생성
다형성	동일한 이름의 메서드가 다양한 방식으로 동작 (`오버로딩` , `오버라이딩`)

✅ Tip or Pitfall

- 객체지향은 단순히 클래스 만드는 게 아님
- **잘 나눈다 → 설계력이 좋다**
- **캡슐화/상속/다형성**을 실제 코드에서 써봐야 감이 옴

2. 클래스와 객체

1. 클래스란?

✅ What

- **속성과 동작(메서드)**을 가진 **객체의 설계도**

→ 객체를 찍어내는 틀이라고 보면 됨

✅ Why

- 현실 세계의 개념을 코드로 표현할 수 있음
- 코드 재사용성 / 유지보수성 향상
- 객체 단위로 프로그램 구성 가능

✅ How

```
public class Person
{
  String name;
  int age;

  void sayHello()
  {
    System.out.println("안녕! 나는 " + name);
  }
}
```

✅ Tip or Pitfall

- 클래스명은 **대문자로 시작 (PascalCase)**
- 하나의 자바 파일에 **public 클래스는 하나만 가능**

2. 인스턴스(Instance)

✅ What

클래스를 기반으로 생성한 실제 객체

→ 클래스 = 설계도 / 인스턴스 = 실물

✅ Why

- 설계도(클래스)만 있으면 실행 불가능
- 실제 작업은 인스턴스를 통해 이루어짐

✅ How

```
Person p = new Person();
p.name = "민솔";
p.age = 25;
p.sayHello(); // 안녕! 나는 민솔
```

✅ Tip or Pitfall

- `new 클래스명()` 을 통해 **힙 메모리에 객체 생성**
- 여러 인스턴스를 만들면 **속성은 서로 독립적임**

3. 클래스의 사용

✅ What

클래스는 다음 3단계로 사용해!

1. 클래스 정의
2. 인스턴스 생성
3. 인스턴스를 통해 변수/메서드 사용

✅ Why

클래스를 사용하면 **현실 세계를 그대로 코드로 모델링** 가능

✅ How

```
// 클래스 정의
class Animal
{
  String type;
  void sound()
  {
    System.out.println(type + "가 울어요!");
  }
}

// 클래스 사용
public class Main
{
  public static void main(String[] args)
  {
    Animal dog = new Animal();
    dog.type = "강아지";
    dog.sound(); // 강아지가 울어요!
  }
}
```

✅ Tip or Pitfall

- `main()` 메서드는 항상 **클래스 안에 있어야 함**

- 객체마다 속성값이 다르게 설정 가능 (독립적 인스턴스)

3. 인스턴스 변수와 클래스 변수

1. 인스턴스 변수

✅ What

객체마다 따로 가지고 있는 변수 → `new` 할 때마다 새로 생성됨

✅ Why

각 인스턴스마다 **고유한 상태**를 가지게 하기 위해

✅ How

```
class Car
{
  String color;  // 인스턴스 변수
}

Car car1 = new Car();
car1.color = "red";

Car car2 = new Car();
car2.color = "blue";
```

✅ Tip or Pitfall

- 인스턴스 변수는 **객체가 생성될 때마다 별도로 존재함**
- `static`이 없으면 인스턴스 변수다!

2. 클래스 변수

✅ What

모든 인스턴스가 공유하는 변수, 클래스에 1개만 존재

✅ Why

- 인스턴스와 관계없이 **공통 값**을 저장
- 누적값, 전체 개수 추적 등에서 사용

✅ How

```
class Counter
{
  static int count = 0;

  Counter()
  {
    count++;
  }
}

System.out.println(Counter.count); // 0 → 생성 안 했으니까
Counter c1 = new Counter();
Counter c2 = new Counter();
System.out.println(Counter.count); // 2
```

✅ Tip or Pitfall

- 클래스명으로 직접 접근 가능: `Counter.count`
- 모든 객체가 **공유**하므로 값 조작 주의
- `static`은 객체 생성 없이도 접근 가능