2. 기초 문법

식별자와 키워드

✅ 1. 식별자 (Identifier)

🔷 정의

식별자란 변수, 메서드, 클래스, 인터페이스, 패키지 등에서 **사용자가 이름을 붙이는 모든 명칭**을 말한다.

예를 들어 아래 코드에서 식별자는 다음과 같다:

```
public class Student {        // Student ← 클래스 식별자
    int age;                  // age ← 변수 식별자
    public void study() {     // study ← 메서드 식별자
        int score = 100;      // score ← 지역 변수 식별자
    }
}
```

🔷 식별자 명명 규칙

항목	규칙
시작 문자	**문자(~~a~~z, ~~A~~Z), `_`, `$` 만 가능**. 숫자로 시작 ❌
나머지 문자	문자, 숫자, `_`, `$` 모두 허용
대소문자 구분	`Score` ≠ `score`
길이 제한	제한 없음
키워드 사용 금지	`class`, `int`, `for` 등 키워드는 식별자 ❌
Unicode 지원	`이름`, `变量` 같은 유니코드도 가능하긴 하지만 **권장되지 않음**

✅ 식별자 예시

유효한 식별자	설명
`myVar`	일반적인 변수 이름
`_counter`	밑줄로 시작 가능
`$amount`	달러 기호도 가능 (자주 쓰이진 않음)
`Student1`	숫자 포함 OK
`변수명`	유니코드 사용 가능 (한국어, 중국어 등)

유효하지 않은 식별자	이유
`1var`	숫자로 시작 ❌
`int`	키워드 사용 ❌
`my-var`	`-` 기호 불가
`@value`	`@` 불가 (애노테이션에 사용됨)

🔷 식별자 작명 관례 (Java 코딩 스타일)

용도	관례
클래스	**대문자 시작 + CamelCase** → `StudentInfo`
변수, 메서드	**소문자 시작 + CamelCase** → `studentName`, `getScore()`
상수	**전부 대문자 + 언더스코어** → `MAX_SIZE`, `DEFAULT_PORT`
패키지	**소문자 점 구분** → `com.example.app`

✅ 2. 키워드 (Keyword)

🔷 정의

키워드란 Java 언어에서 **특별한 의미를 갖는 예약어**로,
컴파일러가 특정 문법 구조로 해석하기 때문에 **사용자가 식별자로 쓸 수 없음.**

🔷 자바 키워드 목록 (총 50개)

키워드
`abstract`, `assert`, `boolean`, `break`, `byte`, `case`, `catch`, `char`, `class`, `const`, `continue`
`default`, `do`, `double`, `else`, `enum`, `extends`, `final`, `finally`, `float`, `for`
`goto`, `if`, `implements`, `import`, `instanceof`, `int`, `interface`, `long`, `native`, `new`
`package`, `private`, `protected`, `public`, `return`, `short`, `static`, `strictfp`, `super`, `switch`
`synchronized`, `this`, `throw`, `throws`, `transient`, `try`, `void`, `volatile`, `while`

> 🔸 `goto`, `const`는 사용되지 않지만 여전히 예약됨

🔷 분류별 주요 키워드

🔹 제어 흐름 관련

- `if`, `else`, `switch`, `case`, `default`, `while`, `do`, `for`, `break`, `continue`, `return`

🔹 클래스 및 상속 관련

- `class`, `interface`, `extends`, `implements`, `abstract`, `final`

🔹 접근 및 선언 관련

- `public`, `private`, `protected`, `static`, `void`, `this`, `super`

🔹 예외 처리

- `try`, `catch`, `finally`, `throw`, `throws`, `assert`

🔹 기타

- `new`, `import`, `package`, `instanceof`, `native`, `strictfp`, `synchronized`, `transient`, `volatile`

✅ 3. 식별자 vs 키워드 비교

항목	식별자 (Identifier)	키워드 (Keyword)
정의	프로그래머가 정의한 이름	자바 문법에 예약된 단어
사용 가능	변수, 클래스, 메서드 이름 등	코드 작성용으로만 사용 (이름 사용 불가)
예시	`score`, `Student`, `getName`	`class`, `for`, `return`, `public`
중복 사용	가능	불가능

✅ 예제 실습

```
public class Main {         // 'Main'은 식별자
    public static void main(String[] args) {
        int count = 10;     // 'count'는 식별자, 'int'는 키워드
        System.out.println(count);  // 'System', 'out', 'println'도 식별자
    }
}
```

✅ 마무리 요약

용어	정의	사용 예시
식별자	사용자가 이름을 붙이는 모든 명칭	변수명, 메서드명, 클래스명 등

용어	정의	사용 예시
키워드	Java 문법에 예약된 단어	`public`, `class`, `return`, `if`, `for` 등

데이터 타입 (기본형과 참조형)

✅ 1. Java의 데이터 타입 분류

Java의 데이터 타입은 크게 두 가지로 나뉜다:

```
                ┌──────────────┐
                | Data Type |
                └──────┬───────┘
                       |
        ┌──────────────┴──────────────┐
        |                             |
기본형 (Primitive Type)   참조형 (Reference Type)
```

✅ 2. 기본형(Primitive Type)

> 💡 **값 자체를 저장**하는 타입. **메모리(Stack)** 에 직접 값이 저장된다.

🔹 기본형 타입 8가지

분류	타입	크기	예시 값	설명
논리형	`boolean`	1비트	`true`, `false`	조건문에서 사용
문자형	`char`	2바이트	`'A'`, `'가'`	유니코드 문자 하나
정수형	`byte`	1바이트	`-128 ~ 127`	가장 작은 정수
	`short`	2바이트	`-32,768 ~ 32,767`	작은 정수
	`int`	4바이트	`-2^31 ~ 2^31-1`	기본 정수 타입
	`long`	8바이트	`-2^63 ~ 2^63-1`	큰 수 (끝에 `L` 필수)
실수형	`float`	4바이트	`3.14f`	단정도 부동소수점 (`f` 필요)
	`double`	8바이트	`3.141592`	배정도 부동소수점 (기본 실수)

🔹 기본형 변수 선언 예시

```java
boolean isOn = true;
char ch = 'A';
int score = 95;
long population = 8000000000L;
float pi = 3.14f;
double avg = 93.7;
```

> 🔸 `float` 값은 `f`, `long`은 `L`을 반드시 붙여야 함
> 🔸 `char`는 작은 따옴표 `'A'`, 문자열은 큰따옴표 `"ABC"`

✅ 3. 참조형(Reference Type)

> 💡 **값이 아니라 주소(참조)를 저장**하는 타입. **Heap 영역에 실제 객체가 저장되고**, Stack에는 참조값(주소)가 저장된다.

🔹 종류

유형	예시
클래스	`String`, `Scanner`, `Random`, `ArrayList` 등
배열	`int[]`, `String[]`, `char[]` 등
인터페이스	`Runnable`, `Comparable` 등
열거형	`enum`
사용자 정의 클래스	`Student`, `Car`, `Book` 등

🔹 참조형 예시

```java
String name = "Alice";              // 문자열 객체
int[] numbers = {1, 2, 3};          // 정수 배열
Scanner sc = new Scanner(System.in); // 객체 생성
```

- `"Alice"` 문자열은 **Heap 영역**에 저장
- `name`은 해당 객체의 **주소**를 Stack에 저장

🔹 기본형 vs 참조형 비교

항목	기본형 (`int`, `char` 등)	참조형 (`String`, `int[]` 등)
저장 위치	Stack (값 자체 저장)	Stack에 주소 저장 + Heap에 값
비교 방식	`==` 연산으로 값 비교	`==`은 주소 비교, `equals()` 로 내용 비교
크기	고정 크기 (1~8바이트)	객체 크기에 따라 달라짐

항목	기본형 (`int`, `char` 등)	참조형 (`String`, `int[]` 등)
null 사용 가능	❌ (null 없음)	✅ 가능 (`String s = null;`)

🔹 예제 코드: 비교 차이

```
int a = 10;
int b = 10;
System.out.println(a == b);       // true (값 비교)

String s1 = new String("hi");
String s2 = new String("hi");
System.out.println(s1 == s2);     // false (주소 비교)
System.out.println(s1.equals(s2)); // true (내용 비교)
```

✅ 4. null 개념 (참조형에서만 사용)

- `null`은 참조형 변수에만 사용할 수 있음
- 해당 변수에 아무 객체도 참조하지 않음을 의미
- `int a = null;` → ❌ 오류
- `String s = null;` → ✅ 가능

```
String s = null;
if (s != null) {
    System.out.println(s.length());
}
```

✅ 5. 타입 변환 (Type Casting)

🔹 자동 형변환 (Upcasting)

- 작은 → 큰 타입 자동 변환

```
int i = 100;
long l = i;        // 자동 변환
```

🔹 명시적 형변환 (Downcasting)

- 큰 → 작은 타입은 반드시 캐스팅 필요

```
double d = 3.14;
int i = (int)d;    // 소수점 손실됨
```

✅ 6. 요약 비교표

구분	기본형(Primitive)	참조형(Reference)
저장	값 자체	객체 주소
예시 타입	`int`, `char`, `boolean`	`String`, `int[]`, 클래스
null 가능 여부	❌	✅
비교 방식	`==` 값 비교	`==`: 주소, `equals()`: 내용
사용 목적	숫자, 문자, 논리값 등 간단한 값	객체, 배열, 클래스 등 구조적 데이터

변수 선언 및 초기화

✅ 1. 변수란?

- **데이터를 저장할 수 있는 이름이 붙은 공간**
- Java에서는 모든 변수는 **타입을 명시**하고, **선언 후 초기화**해야 사용할 수 있어

✅ 2. 변수 선언 기본 형식

```
타입 변수명;                 // 선언만
타입 변수명 = 초기값;        // 선언과 동시에 초기화
```

🔹 예제

```
int age;             // 변수 선언 (초기화 X)
age = 20;            // 초기화

int score = 95;      // 선언과 동시에 초기화
```

✅ 3. 변수 타입별 선언 예시

타입	선언 예시
`int`	`int num = 10;`
`double`	`double pi = 3.14;`
`boolean`	`boolean flag = true;`
`char`	`char grade = 'A';`
`String`	`String name = "Alice";`

✅ 4. 변수 종류 (선언 위치에 따른 분류)

종류	선언 위치	특징	초기화 필요 여부
지역 변수	메서드 내부	블록이 끝나면 소멸	✅ 반드시 초기화
멤버 변수 (필드)	클래스 내부, 메서드 외부	객체 생성 시 자동 초기화	❌ 생략 가능
정적 변수 (static 필드)	클래스 내부, static 키워드 붙임	클래스 로딩 시 1회 초기화	❌ 생략 가능
매개변수	메서드 정의부의 괄호 내	호출될 때 값 전달	자동 초기화 불가

🔸 예제 코드

```java
public class Example {

    // 멤버 변수
    int instanceVar = 10;
    static int staticVar = 100;

    public void method(int param) { // 매개변수
        int localVar = 5; // 지역 변수
        System.out.println(localVar + param);
    }
}
```

✅ 5. 변수 초기화

🔹 명시적 초기화

```java
int num = 5;
String name = "Java";
```

🔹 지연 초기화 (선언 후 나중에 값 대입)

```java
int age;
age = 30;
```

🔹 디폴트 초기값 (멤버 변수 또는 static 변수일 때만)

타입	기본값
`int`	`0`
`boolean`	`false`
`double`	`0.0`

타입	기본값
`char`	`\u0000` (널 문자)
참조형	`null`

> 지역 변수는 반드시 **직접 초기화**하지 않으면 컴파일 에러 발생!

✅ 6. 여러 변수 한 줄에 선언

```
int a = 1, b = 2, c = 3;
```

> 타입은 하나만 명시해야 하며, 같은 타입일 경우에만 가능

✅ 7. 상수 선언 (`final` 키워드)

```
final int MAX_USERS = 100;  // 상수 선언
```

- 값을 바꿀 수 없는 상수
- **대문자 + 밑줄** 형식 사용이 관례

✅ 8. 변수 이름 규칙 요약

규칙	설명
시작 문자	문자, `_`, `$` 가능 (숫자 ❌)
대소문자	구분함 (`Age` ≠ `age`)
키워드 사용	불가능 (`int`, `class` 등)
관례	camelCase (`userName`, `totalScore`)

✅ 9. 예제 실습 코드

```
public class VariableDemo {

    int instanceCount = 1;          // 멤버 변수
    static String category = "Java"; // 정적 변수

    public void printInfo(String name) { // 매개변수
        int localScore = 100;       // 지역 변수
        System.out.println(name + ": " + localScore);
    }

```

```
    public static void main(String[] args) {
        VariableDemo v = new VariableDemo();
        v.printInfo("Alice");
    }
}
```

✅ 10. 요약 비교표

항목	지역 변수	멤버 변수	정적 변수
선언 위치	메서드 내부	클래스 내부	클래스 내부
메모리 영역	Stack	Heap	Method 영역
기본값 자동 설정	❌	✅	✅
생존 기간	메서드 실행 중	객체가 살아있는 동안	클래스가 메모리에 있는 동안
키워드	없음	없음	`static`

연산자 (산술, 비교, 논리, 비트, instanceof 등)

✅ 1. 연산자 개요

연산자는 다음과 같이 분류된다:

분류	대표 연산자
산술 연산자	`+`, `-`, `*`, `/`, `%`
비교 연산자	`==`, `!=`, `>`, `<`, `>=`, `<=`
논리 연산자	`&&`, `
비트 연산자	`&`, `
대입 연산자	`=`, `+=`, `-=`, `*=`, `/=`, `%=` 등
증감 연산자	`++`, `--`
조건(삼항) 연산자	`? :`
타입 비교 연산자	`instanceof`

✅ 2. 산술 연산자

🔹 개요

숫자 값을 계산하는 기본 연산자

연산자	의미	예시	결과
`+`	더하기	`5 + 2`	`7`
`-`	빼기	`5 - 2`	`3`
`*`	곱하기	`5 * 2`	`10`
`/`	나누기	`5 / 2`	`2` (정수 나눗셈)
`%`	나머지	`5 % 2`	`1`

> 💡 `/` 연산에서 피연산자가 정수면 **몫만 반환**됨
> 💡 `5.0 / 2`은 `2.5` (실수 결과)

✅ 3. 비교 연산자 (관계 연산자)

🔹 개요

두 값을 비교하여 **참/거짓(boolean)** 반환

연산자	의미	예시	결과
`==`	같다	`5 == 5`	`true`
`!=`	다르다	`5 != 3`	`true`
`>`	초과	`5 > 3`	`true`
`<`	미만	`5 < 3`	`false`
`>=`	이상	`5 >= 5`	`true`
`<=`	이하	`5 <= 4`	`false`

✅ 4. 논리 연산자

🔹 개요

조건식들을 연결하거나 부정할 때 사용. **결과는 boolean**

연산자	의미	예시	결과
`&&`	논리 AND	`true && false`	`false`

연산자	의미	예시	결과
`||`	논리 OR		
`!`	논리 NOT	`!true`	`false`

> 💡 `&&`, `||`는 **단축 평가(short-circuit evaluation)** 적용됨
> (앞 조건이 `false`면 `&&` 뒤는 검사 안 함)

✅ 5. 대입 연산자

🔹 개요

변수에 값을 **저장**하거나 **누적**할 때 사용

연산자	예시	의미
`=`	`x = 10`	10을 대입
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 2`	`x = x - 2`
`*=`	`x *= 5`	`x = x * 5`
`/=`	`x /= 4`	`x = x / 4`
`%=`	`x %= 2`	`x = x % 2`

✅ 6. 증감 연산자

🔹 개요

변수 값을 **1씩 증가 또는 감소**시키는 연산자

연산자	의미	예시	설명
`++`	1 증가	`x++`, `++x`	후위, 전위 구분
`--`	1 감소	`x--`, `--x`	후위, 전위 구분

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

✅ 7. 조건 연산자 (삼항 연산자)

```
조건식 ? 값1 : 값2
```

🔹 예시

```
int a = 10;
String result = (a > 5) ? "크다" : "작다";  // "크다"
```

✅ 8. 비트 연산자

🔹 개요

2진수 비트 단위로 연산을 수행함

연산자	설명	예시 (5: 0101, 3: 0011)	결과
`&`	비트 AND	`5 & 3`	`0001` → 1
`|`	비트 OR	`5 | 3`	`0111` → 7
`^`	비트 XOR	`5 ^ 3`	`0110` → 6
`~`	비트 NOT (1의 보수)	`~5`	`...1010` → -6
`<<`	왼쪽 시프트 (n비트 이동)	`5 << 1`	`1010` → 10
`>>`	오른쪽 시프트 (부호 유지)	`5 >> 1`	`0010` → 2
`>>>`	부호 없는 오른쪽 시프트	(음수/양수 다르게 작동)	(부호비트까지 0으로 채움, 예: `-5 >>> 1`)

✅ 9. 타입 비교 연산자 (`instanceof`)

🔹 개요

객체가 특정 클래스의 **인스턴스인지 확인**

```
객체 instanceof 클래스
```

🔹 예시

```
String s = "hello";
System.out.println(s instanceof String);   // true
System.out.println(s instanceof Object);   // true
```

- 자식 클래스 → 부모 클래스 비교도 가능
- null 객체는 항상 `false`

✅ 10. 우선순위 요약 (높을수록 먼저 연산됨)

순서	연산자
1	`()`, `[]`, `.`
2	`++`, `--`, `!`, `~`
3	`*`, `/`, `%`
4	`+`, `-`
5	`<<`, `>>`, `>>>`
6	`<`, `<=`, `>`, `>=`, `instanceof`
7	`==`, `!=`
8	`&`
9	`^`
10	`` ` ``
11	`&&`
12	`` ` ``
13	`? :`
14	`=`, `+=`, `-=`, `*=`, ...

✅ 예제 코드

```
public class OperatorDemo {
    public static void main(String[] args) {
        int a = 10, b = 3;

        System.out.println("a + b = " + (a + b)); // 산술
        System.out.println("a > b? " + (a > b));  // 비교
        System.out.println("a == 10 && b < 5: " + (a == 10 && b < 5)); // 논리
        System.out.println("a & b = " + (a & b)); // 비트
        System.out.println("a instanceof Integer: " + ((Object)a instanceof Integer));
// 타입 비교
    }
}
```

형변환 (암시적, 명시적)

✅ 1. 형변환이란?

> 변수나 값의 **데이터 타입을 다른 타입으로 변환**하는 것

- 타입이 다르면 연산이나 대입이 불가능하므로 형변환이 필요함
- Java는 컴파일 시점에 타입을 체크하는 **정적 타입 언어**

✅ 2. 형변환의 종류

종류	설명
암시적 형변환 (자동 변환)	작은 타입 → 큰 타입으로 자동 변환됨
명시적 형변환 (강제 변환)	큰 타입 → 작은 타입으로 직접 지정 필요

✅ 3. 기본형 간 형변환 (Primitive Type)

🔹 타입 계층 구조 (자동 변환 방향)

```
byte → short → int → long → float → double
                        ↑
                    char (독립)
```

✅ 4. 암시적 형변환 (자동 형변환, Widening Conversion)

> 작은 타입 → 큰 타입으로 JVM이 **자동으로 변환**함
> 데이터 손실 없음

✅ 예시

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

✅ 지원되는 경우

From	To
`byte`	`short`, `int`, `long`, `float`, `double`
`char`	`int`, `long`, `float`, `double`
`int`	`long`, `float`, `double`
`float`	`double`

✅ 5. 명시적 형변환 (강제 형변환, Narrowing Conversion)

> 큰 타입 → 작은 타입으로 변환 시 **데이터 손실 가능성** 때문에
> 반드시 **캐스팅 연산자** `(타입)`을 명시해야 함

✅ 예시

```
java코드 복사double pi = 3.14;
int intPi = (int) pi;   // 소수점 절삭됨
System.out.println(intPi); // 3
```

✅ 데이터 손실 예시

```
int big = 130;
byte small = (byte) big;
System.out.println(small);  // -126 (오버플로우 발생)
```

> byte 범위: -128 ~ 127이므로 130은 초과되어 잘림

✅ 6. 정수 ↔ 실수 간 형변환

🔹 정수 → 실수: 자동 변환

```
int x = 10;
double y = x;  // 10.0
```

🔹 실수 → 정수: 명시적 변환

```
double d = 9.9;
int i = (int)d; // 9 (소수점 제거)
```

✅ 7. char ↔ int 변환

```
char ch = 'A';
int code = ch;         // char → int (자동)
System.out.println(code); // 65

int num = 66;
char ch2 = (char)num;  // int → char (명시적)
System.out.println(ch2); // B
```

✅ 8. 참조형 형변환 (객체 타입)

Java에서는 참조형에서도 형변환이 존재함.
상속 관계 또는 **인터페이스 구현 관계**에서만 가능!

🔹 업캐스팅 (자동)

```
class Animal {}
class Dog extends Animal {}

Animal a = new Dog();  // Dog → Animal (자동)
```

🔹 다운캐스팅 (명시적)

```
Animal a = new Dog();
Dog d = (Dog)a;        // Animal → Dog (명시적)
```

> 잘못된 다운캐스팅은 `ClassCastException` 발생

```
Animal a = new Animal();
Dog d = (Dog)a; // ❌ 런타임 오류
```

→ 안전하게 캐스팅하려면 `instanceof` 연산자 사용

```
if (a instanceof Dog) {
    Dog d = (Dog)a;
}
```

✅ 9. 형변환과 연산

🔹 연산 중 자동 형변환

```
int a = 5;
double b = 2.5;
double result = a + b;  // int → double 자동 변환
```

🔹 char 연산

```
char c = 'A';   // 유니코드 65
System.out.println(c + 1);  // 66
```

✅ 10. 요약 비교표

구분	자동 형변환	명시적 형변환
방향	작은 타입 → 큰 타입	큰 타입 → 작은 타입
예시	`int → double`	`double → int`
데이터 손실	❌ 없음	✅ 있을 수 있음
문법	생략 가능	`(타입)` 명시 필수
객체 변환	업캐스팅	다운캐스팅

✅ 실습 예제 전체 코드

```
public class TypeCastDemo {
    public static void main(String[] args) {
        int i = 100;
        double d = i; // 자동
        System.out.println("자동 형변환: " + d);

        double pi = 3.14;
        int n = (int) pi; // 강제
        System.out.println("명시적 형변환: " + n);

        char ch = 'A';
        int code = ch;
        System.out.println("char → int: " + code);

        code = 66;
        ch = (char) code;
        System.out.println("int → char: " + ch);
    }
}
```

문자열 처리 (`String`, `StringBuilder`, `StringBuffer`)

✅ 1. `String` 클래스 (불변, Immutable)

🔹 기본 개념

```
String str = "Java";
String str2 = new String("Java");
```

- `String` 객체는 **불변(immutable)** → 한 번 생성된 문자열은 **변경 불가**
- 문자열 변경 시 새로운 객체가 생성됨
- Java에서 문자열 리터럴 `"..."`은 내부적으로 **String Pool**에서 관리됨

🔷 주요 메서드

메서드	설명	예시
`length()`	문자열 길이	`"Hello".length()` → `5`
`charAt(i)`	i번째 문자	`"Java".charAt(1)` → `'a'`
`substring(a,b)`	a~b-1까지 자르기	`"Hello".substring(1,4)` → `"ell"`
`equals()`	문자열 내용 비교	`"abc".equals("abc")` → `true`
`equalsIgnoreCase()`	대소문자 무시 비교	`"Java".equalsIgnoreCase("java")` → `true`
`contains()`, `startsWith()`, `endsWith()`	부분 문자열 포함 여부	`"Hello".contains("ll")` → `true``"Hello".startsWith("He")` → `true``"Hello".endsWith("lo")` → `true`
`indexOf()`, `lastIndexOf()`	문자 위치 찾기	`"Hello".indexOf('l')` → `2``"Hello".lastIndexOf('l')` → `3`
`replace()`, `replaceAll()`	치환	`"Hello World".replace("World", "Java")` → `"Hello Java"``"a,b,c".replaceAll(",", "-")` → `"a-b-c"`
`split()`	구분자로 분할	`"a,b,c".split(",")` → `[a, b, c]`
`trim()`	양쪽 공백 제거	`" Hello ".trim()` → `"Hello"`
`toLowerCase()`, `toUpperCase()`	대소문자 변환	`"Java".toLowerCase()` → `"java"``"Java".toUpperCase()` → `"JAVA"`

🔷 예시

```
String name = "Java";
String newName = name.replace("a", "o");

System.out.println(name);     // Java
System.out.println(newName);  // Jovo
```

> 원본 `name`은 **변하지 않음**! → 새 객체 반환됨

✅ 2. `StringBuilder` 클래스 (가변, 비동기)

> 💡 **문자열을 자주 수정(append, delete 등)할 경우** 가장 효율적임
> **단일 스레드 환경**에서 빠름

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); // Hello World
```

🔷 주요 메서드

메서드	설명
`append(str)`	문자열 덧붙이기
`insert(pos, str)`	중간 삽입
`delete(start, end)`	범위 삭제
`reverse()`	역순으로 뒤집기
`replace(start, end, str)`	범위 치환
`setCharAt(index, ch)`	특정 문자 수정
`toString()`	`String` 객체로 변환

🔷 예시

```
StringBuilder sb = new StringBuilder();
sb.append("Java");
sb.append(" is");
sb.append(" fun!");

System.out.println(sb.toString());  // Java is fun!
```

✅ 3. `StringBuffer` 클래스 (가변, 동기화)

> 💡 `StringBuilder`와 거의 동일하지만, **멀티스레드 환경에서 안전(thread-safe)**
> 내부 연산에 **동기화(synchronized)** 적용

```
StringBuffer sb = new StringBuffer("Hi");
sb.append(" there");
System.out.println(sb);  // Hi there
```

🔷 메서드

`StringBuilder`와 동일한 API 사용 가능

✅ 4. 성능/특징 비교

항목	`String`	`StringBuilder`	`StringBuffer`
변경 가능 여부	❌ 불변	✅ 가변	✅ 가변
스레드 안전성	❌ 비동기	❌ 비동기	✅ 동기화(스레드 안전)
성능	느림 (매번 새 객체)	빠름 (단일 스레드)	느림 (동기화 오버헤드)
사용 시점	문자열이 자주 바뀌지 않을 때	문자열 조작이 많을 때	멀티스레드 환경에서 조작할 때

✅ 5. 문자열 연결 성능 비교

```
public class StringPerf {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();

        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 100000; i++) {
            sb.append("x");
        }

        long end = System.currentTimeMillis();
        System.out.println("걸린 시간: " + (end - start) + "ms");
    }
}
```

- `String` : 매우 느림 (매 반복마다 새 객체 생성)
- `StringBuilder` : 훨씬 빠름
- `StringBuffer` : 느리지만 스레드 안전

✅ 6. 문자열 → 숫자 / 숫자 → 문자열 변환

```
// 문자열 → 정수
int n = Integer.parseInt("123");

// 정수 → 문자열
String s = String.valueOf(123);
String s2 = 123 + "";
```

✅ 7. 요약 정리

구분	String	StringBuilder	StringBuffer
변경 가능 여부	❌ 불변	✅ 가능	✅ 가능
스레드 안전	❌ 비동기	❌ 비동기	✅ 동기화
속도	느림	빠름	중간
주요 메서드	`length`, `substring`, `replace`, ...	`append`, `delete`, `insert`, ...	동일
사용 시기	값 변경 거의 없음	변경 많고 단일 스레드	변경 많고 멀티 스레드

배열 (1차원, 다차원)

✅ 1. 배열이란?

> **같은 타입의 데이터를 연속된 메모리 공간에 저장**하는 자료구조
> Java의 배열은 **객체**이며, `heap 영역`에 저장된다.

- 배열의 크기는 **고정 (초기화 시 결정)**
- 배열의 인덱스는 항상 `0`부터 시작
- 배열 자체도 **참조형(Reference Type)**

✅ 2. 배열 선언과 생성

🔹 일반 문법

```
타입[] 변수명 = new 타입[길이];  // 권장 문법
타입 변수명[] = new 타입[길이]; // C 스타일 문법 (허용되지만 잘 안 씀)
```

✅ 3. 1차원 배열

🔹 선언 및 초기화 방법

```
int[] scores = new int[5];   // 길이 5짜리 int 배열 생성

int[] nums = {10, 20, 30};   // 선언과 동시에 초기화

String[] names = new String[]{"Alice", "Bob", "Charlie"};
```

> 초기화 시에는 `new 타입[] {값1, 값2, ...}` 또는 `{값1, 값2}` 형식 사용 가능

🔹 배열 요소 접근 및 수정

```
scores[0] = 100;
System.out.println(scores[0]);  // 100

int len = scores.length;        // 배열 길이
```

🔹 반복문을 통한 순회

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

// 향상된 for문
for (int n : nums) {
    System.out.println(n);
}
```

✅ 4. 다차원 배열

Java에서는 **2차원 배열 이상도 선언 가능**하며, 내부적으로는 **배열의 배열** 구조야.

🔹 2차원 배열 선언

```
int[][] matrix = new int[2][3];  // 2행 3열 배열

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

- 행의 수: `matrix.length`
- 열의 수: `matrix[0].length`

🔹 2차원 배열 요소 접근

```
matrix[0][1] = 99;
System.out.println(matrix[0][1]);  // 99
```

🔹 2중 for문 순회

```
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();
}
```

✅ 5. 가변 배열 (Jagged Array)

> Java의 다차원 배열은 **"배열의 배열"** 구조이기 때문에,
> 각 행마다 열 개수를 다르게 설정할 수 있어.

```
int[][] jagged = new int[3][];
jagged[0] = new int[2];    // 0행은 2열
jagged[1] = new int[3];    // 1행은 3열
jagged[2] = new int[1];    // 2행은 1열
```

✅ 6. 배열의 특징과 제약

항목	설명
타입 고정	배열에 저장되는 모든 값은 동일한 타입이어야 함
크기 고정	생성 시 크기 결정 → 변경 불가
초기값	기본형은 0, false, '\u0000' 등 / 참조형은 null
배열은 객체	`instanceof Object` → true

✅ 7. 배열 복사

🔹 얕은 복사 (주소만 복사)

```
int[] a = {1, 2, 3};
int[] b = a;

b[0] = 100;
System.out.println(a[0]);  // 100
```

🔹 깊은 복사 (`for`, `Arrays.copyOf`, `System.arraycopy`)

```
int[] a = {1, 2, 3};
int[] b = new int[a.length];

for (int i = 0; i < a.length; i++) {
    b[i] = a[i];
}
```

또는

```
int[] b = Arrays.copyOf(a, a.length);
```

✅ 8. 배열 관련 유틸

- `java.util.Arrays` 클래스 활용

```
import java.util.Arrays;

int[] arr = {3, 1, 2};
Arrays.sort(arr);                    // 정렬
System.out.println(Arrays.toString(arr));  // [1, 2, 3]
```

✅ 9. 예제 코드 전체

```
public class ArrayDemo {
    public static void main(String[] args) {
        int[] nums = {10, 20, 30};
        for (int n : nums) {
            System.out.println(n);
        }

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

        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();
        }
    }
}
```

✅ 요약 정리

구분	설명
1차원 배열	`int[] arr = new int[5];`
2차원 배열	`int[][] arr = new int[2][3];`
배열 길이	`arr.length`, `arr[i].length`
순회 방식	for문, 향상된 for문
배열 복사	`System.arraycopy`, `Arrays.copyOf`, 수동 복사
유틸리티	`Arrays.toString()`, `Arrays.sort()` 등