4. 메서드와 스택 프레임

메서드 정의 및 호출

메서드는 Java의 모든 클래스 내부에서 사용되며, **입력(매개변수)**을 받아 **처리**하고, **출력(반환값)**을 제공하는 **기능 단위 블록**이다.

✅ 1. 메서드란?

> 특정 작업을 수행하는 **코드 블록의 묶음**이며, **클래스 내부에 정의**된다.
> 자주 반복되는 기능을 따로 정의하여 **코드 재사용**, **가독성 향상**, **유지보수성 증가** 목적이 있다.

✅ 2. 메서드 기본 구조

```
[접근제어자] [static] 반환타입 메서드명(매개변수목록) {
    // 실행 코드
    [return 반환값;]
}
```

🔷 구성 요소 설명

항목	설명	예시
접근제어자	외부에서 접근 가능한지 여부 (`public`, `private`, 등)	`public`, `private`
static	클래스 메서드 여부 (정적 메서드)	`static` (생략 가능)
반환타입	메서드가 리턴하는 값의 타입	`int`, `void`, `String` 등
메서드명	소문자로 시작하는 동사형 식별자	`add`, `printMessage()`
매개변수	입력 값 전달 변수들	`(int a, int b)`
return문	반환할 값. `void`면 생략 가능	`return a + b;`

✅ 3. 메서드 정의 예시

🔹 1. 반환값 X, 매개변수 X

```
public void sayHello() {
    System.out.println("Hello!");
}
```

🔹 2. 반환값 X, 매개변수 O

```
public void greet(String name) {
    System.out.println("Hello, " + name);
}
```

🔹 3. 반환값 O, 매개변수 O

```
public int add(int a, int b) {
    return a + b;
}
```

🔹 4. 반환값 O, 매개변수 X

```
public double getPi() {
    return 3.14159;
}
```

✅ 4. 메서드 호출 방법

🔹 클래스 내부에서 호출

```
sayHello();                 // void
greet("Alice");             // void
int result = add(5, 7);     // 반환값 O
```

🔹 클래스 외부에서 호출

```
MyClass obj = new MyClass();
obj.sayHello();
```

> `static` 메서드는 객체 없이 클래스명으로 호출 가능

```
Math.sqrt(9.0);  // Math 클래스의 static 메서드
```

✅ 5. return 문

🔷 역할

- 메서드를 종료하고 값을 반환
- 반환 타입이 `void`인 경우 `return;`만 사용 가능 (생략도 가능)

```java
public int square(int x) {
    return x * x;
}
```

✅ 6. 매개변수 전달 방식

> Java는 **값에 의한 복사 (pass-by-value)** 방식만 지원함

- **기본형**: 값 자체가 복사됨
- **참조형**: 주소(참조값)가 복사됨 (객체 자체가 변경될 수 있음)

🔹 예시: 기본형

```java
void change(int x) {
    x = 100;
}
int a = 10;
change(a);
System.out.println(a); // 10
```

🔹 예시: 참조형

```java
void modify(int[] arr) {
    arr[0] = 99;
}
int[] nums = {1, 2, 3};
modify(nums);
System.out.println(nums[0]); // 99
```

✅ 7. 메서드 오버로딩 (Overloading)

> 같은 이름의 메서드를 **매개변수 타입이나 개수**만 다르게 하여 여러 번 정의 가능

```java
public void print(int x) {
    System.out.println(x);
}

public void print(String s) {
    System.out.println(s);
}
```

- 반환 타입만 다르게 하면 오버로딩 ❌ 불가능

✅ 8. 예제 전체 코드

```java
public class MethodDemo {

    public void sayHello() {
        System.out.println("Hello!");
    }

    public void greet(String name) {
        System.out.println("Hi, " + name);
    }

    public int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        MethodDemo demo = new MethodDemo();

        demo.sayHello();            // void
        demo.greet("Alice");        // void
        int result = demo.add(3, 4); // int
        System.out.println(result); // 7
    }
}
```

✅ 요약 정리

요소	예시	설명
메서드 정의	`int sum(int a, int b)`	반환 타입 + 이름 + 매개변수
반환값	`return result;`	값을 호출자에게 돌려줌
호출 방법	`sum(3, 5)`	정의된 메서드 사용
오버로딩	`print(int)` / `print(String)`	같은 이름, 다른 매개변수
매개변수 전달	값 복사 (기본형), 참조 복사 (참조형)	Java는 항상 pass-by-value

매개변수 전달 (값 전달, 참조 전달)

✅ 1. Java는 "값에 의한 전달"만 지원

> Java는 **C++의 참조 전달(pass-by-reference)**를 **지원하지 않음**
> 대신, **기본형은 값 자체**, **참조형은 참조값(주소)**이 복사되어 전달됨

🔹 의미

- 기본형(Primitive Type) → 값 복사
- 참조형(Reference Type) → **객체 주소** 복사 → 객체 내부는 변경 가능

✅ 2. 기본형 전달 (Primitive Type)

🔹 특징

- 값을 복사하여 전달
- 메서드 안에서 변경해도 **원본에는 영향 없음**

🔹 예제

```
public class PassByValueDemo {
    public static void modify(int x) {
        x = 100;
    }

    public static void main(String[] args) {
        int num = 10;
        modify(num);
        System.out.println(num);  // 출력: 10
    }
}
```

> `modify()` 메서드는 `num`의 복사본을 수정한 것

✅ 3. 참조형 전달 (Reference Type)

🔹 특징

- **참조값(객체 주소)**가 복사되어 전달됨
- 따라서 메서드 내에서 객체의 **필드나 내부 값은 수정 가능**
- 단, **객체 자체를 새로 할당해도 원본 참조에는 반영되지 않음**

예제 1: 배열 값 변경

```
public class ArrayExample {
    public static void modify(int[] arr) {
        arr[0] = 999;
    }

    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        modify(numbers);
        System.out.println(numbers[0]);  // 출력: 999
    }
}
```

> 배열은 객체이므로, 주소값이 복사되어 전달됨 → 내부 변경 가능

예제 2: 객체 필드 변경

```
class Person {
    String name;
}

public class ObjectDemo {
    public static void changeName(Person p) {
        p.name = "Alice";
    }

    public static void main(String[] args) {
        Person person = new Person();
        person.name = "Bob";

        changeName(person);
        System.out.println(person.name);  // 출력: Alice
    }
}
```

예제 3: 객체 자체를 새 객체로 재할당한 경우

```
public class ObjectSwap {
    public static void replace(Person p) {
        p = new Person();        // 새 객체 생성
        p.name = "Charlie";      // 이 객체는 메서드 안에서만 유효
    }

    public static void main(String[] args) {
        Person person = new Person();
        person.name = "Bob";

        replace(person);
```

```
        System.out.println(person.name);  // 출력: Bob (변경 X)
    }
}
```

> `p`는 **복사된 참조값**일 뿐 → 다른 객체로 바꿔도 원본 변수에는 영향 없음

✅ 4. swap 불가능 예시 (참조 자체를 바꾸는 건 불가능)

```
public class SwapDemo {
    public static void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
    }

    public static void main(String[] args) {
        int x = 1, y = 2;
        swap(x, y);
        System.out.println(x + ", " + y);  // 1, 2 (바뀌지 않음)
    }
}
```

✅ 요약 비교

구분	기본형 (Primitive)	참조형 (Reference)
전달 방식	값 자체 복사	참조 주소 복사
메서드 내 변경	원본에 영향 없음	객체 내부는 영향 있음
객체 자체를 재할당	영향 없음	영향 없음
예시 타입	`int`, `double`, `char` 등	`String`, `int[]`, `Object` 등

✅ 핵심 요약

헷갈리는 표현	정확한 표현
Java는 참조로 전달한다?	❌ 모두 값 전달이다
참조형도 참조로 전달된다?	❌ 참조 "값"을 복사하여 전달한다
객체 자체를 바꾸면 바깥에도 반영된다?	❌ 내부 필드만 바뀔 수 있음

반환값 처리

✅ 1. 반환값이란?

> 메서드가 실행을 끝낸 후 **호출한 쪽에 돌려주는 결과 값**

- 메서드는 **0개 또는 1개의 반환값**만 가질 수 있음
- 반환값은 메서드 선언에서 명시한 **반환 타입(return type)**과 반드시 일치해야 함

✅ 2. 기본 문법

```
반환타입 메서드이름(매개변수목록) {
    return 반환값;
}
```

✅ 3. 반환값이 있는 메서드

🔹 예시: 정수 더하기

```
public int add(int a, int b) {
    return a + b;
}
```

- `int` 타입을 반환
- 호출할 때 `int result = add(3, 5);`

🔹 예시: 문자열 반환

```
public String greet(String name) {
    return "Hello, " + name;
}
```

- `String` 타입 반환

✅ 4. 반환값이 없는 메서드 (`void`)

🔹 `void` 키워드

- 아무 값도 반환하지 않겠다는 의미
- `return;`은 생략 가능하거나 단독 사용 가능

```
public void printMessage(String msg) {
    System.out.println(">> " + msg);
}
```

✅ 5. return 문 사용 시 주의점

상황	설명
`return 값;`	반환 타입이 `void`가 아닌 경우 필수
`return;`	반환 타입이 `void`일 경우에만 사용 가능
`return` 생략	`void`인 경우에만 생략 가능
여러 return	조건 분기 내 여러 개의 return 사용 가능

🔹 예시: 조건에 따라 다른 값을 반환

```
public String checkScore(int score) {
    if (score >= 90) return "A";
    else if (score >= 80) return "B";
    else return "F";
}
```

✅ 6. 다양한 반환 타입 예시

반환 타입	예시
기본형	`int`, `double`, `boolean` 등
참조형	`String`, `int[]`, `ArrayList` 등
사용자 정의 타입	`Student`, `Person` 클래스 등
void	반환 없음

✅ 7. 메서드의 반환값을 저장하거나 바로 사용

🔹 변수에 저장

```
int sum = add(3, 4);
```

🔹 바로 출력

```
System.out.println(add(3, 4));
```

- 조건식에 사용

```
if (isEven(10)) {
    System.out.println("짝수입니다.");
}
```

✅ 8. 반환값으로 객체 반환하기

```
public Student createStudent(String name) {
    Student s = new Student();
    s.name = name;
    return s;
}
```

> 호출: `Student s = createStudent("Alice");`

✅ 9. 배열 또는 컬렉션 반환

```
public int[] getArray() {
    return new int[]{1, 2, 3};
}

public List<String> getNames() {
    return Arrays.asList("Alice", "Bob");
}
```

✅ 10. 실습 예제

```
public class ReturnDemo {

    public int square(int x) {
        return x * x;
    }

    public boolean isEven(int num) {
        return num % 2 == 0;
    }

    public void printGreeting() {
        System.out.println("Welcome!");
    }

    public static void main(String[] args) {
        ReturnDemo demo = new ReturnDemo();

        int result = demo.square(5);         // 25
        boolean flag = demo.isEven(6);       // true
```

```
20
21          System.out.println(result);
22          System.out.println(flag);
23
24          demo.printGreeting();                // Welcome!
25      }
26  }
```

✅ 11. 요약 정리

항목	설명	예시
반환 타입	메서드가 돌려주는 값의 타입	`int`, `String`, `void`, ...
반환문	메서드 종료 + 값 반환	`return 값;`
void	반환값 없음	`return;` or 생략 가능
값 사용	변수 저장, 조건식, 직접 출력 등	`int x = sum();`

메서드 오버로딩

✅ 1. 메서드 오버로딩이란?

> **같은 이름의 메서드를 매개변수의 개수나 타입만 다르게** 여러 개 정의하는 것
> → 호출 시 컴파일러가 **인자의 형식에 맞는 메서드를 선택**

🔷 예시

```
1  public void print() {
2      System.out.println("No value");
3  }
4
5  public void print(int x) {
6      System.out.println("Int: " + x);
7  }
8
9  public void print(String s) {
10     System.out.println("String: " + s);
11 }
```

> 호출:

```
1  print();           // No value
2  print(10);         // Int: 10
3  print("Java");     // String: Java
```

✅ 2. 오버로딩이 되는 조건

> 다음 중 **하나라도 다르면 오버로딩 가능**:

조건 항목	예시
매개변수의 **개수**	`print()` vs `print(int x, int y)`
매개변수의 **타입**	`print(int x)` vs `print(double x)`
매개변수의 **순서**	`print(int, String)` vs `print(String, int)`

❌ 주의: 반환 타입만 다르면 오버로딩 불가능

```
public int add(int x) { return x; }
// 아래는 오버로딩 ❌ 오류 발생
public double add(int x) { return (double)x; }
```

> 컴파일러는 **메서드 시그니처(method signature)**만으로 오버로딩 여부를 판단하고,
> **반환 타입은 시그니처에 포함되지 않는다.**

✅ 3. 오버로딩 적용 예시

🔹 다양한 자료형 처리

```
public void print(int n) {
    System.out.println("정수: " + n);
}

public void print(double d) {
    System.out.println("실수: " + d);
}
```

🔹 기본값 대체를 위한 오버로딩

```
public void greet(String name) {
    System.out.println("Hello, " + name);
}

public void greet() {
    greet("Guest");  // 기본값 처리
}
```

🔹 순서만 다른 경우

```
public void log(String message, int level) {
    System.out.println(level + ": " + message);
}

public void log(int level, String message) {
    System.out.println(level + ": " + message);
}
```

> 두 메서드는 매개변수 타입은 같지만 **순서가 다르므로 오버로딩 가능**

✅ 4. 오버로딩과 자동 형변환 (주의!)

```
public void show(int x) {
    System.out.println("int");
}

public void show(long x) {
    System.out.println("long");
}

show(10);   // int → 정확히 일치 → "int"
show(10L);  // long → "long"
```

- 자동 형변환이 가능한 메서드도 오버로딩 대상이 될 수 있음
- `float`을 넣었는데 `double` 오버로딩이 없으면 자동 형변환 발생 가능

✅ 5. 실전 예제

```
public class OverloadDemo {

    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public String add(String a, String b) {
        return a + b;
    }

    public static void main(String[] args) {
        OverloadDemo d = new OverloadDemo();
        System.out.println(d.add(3, 5));            // 8
        System.out.println(d.add(3.2, 1.8));        // 5.0
```

```
19          System.out.println(d.add("Java", "17"));    // Java17
20      }
21  }
```

✅ 6. 오버로딩 vs 오버라이딩 차이

구분	오버로딩 (Overloading)	오버라이딩 (Overriding)
정의	같은 클래스에서 메서드 이름은 같고 시그니처만 다름	상속받은 메서드를 재정의
위치	같은 클래스 내부	부모 클래스 → 자식 클래스
조건	매개변수 타입/개수/순서 다름	시그니처 완전 동일
반환 타입	다를 수 있음 (단독으로는 불가)	부모와 동일하거나 하위 타입

✅ 7. 정리 요약

항목	설명
정의	같은 이름의 메서드를 매개변수 형식만 바꿔 여러 개 정의
가능한 경우	매개변수의 개수, 타입, 순서가 다를 때
불가능한 경우	오직 반환 타입만 다른 경우
목적	코드의 가독성과 유연성 향상
관련 키워드	**다형성(Polymorphism)**의 한 형태

메서드와 스택 프레임

✅ 1. JVM에서 메서드 호출의 본질

Java에서 메서드를 호출하면 JVM은 **스택(stack)**이라는 메모리 구조에 **스택 프레임(Stack Frame)**을 쌓는다.
각 메서드는 실행 시 **고유한 스택 프레임 하나를 생성**하고, **종료 시 해당 프레임이 제거(pop)**된다.

✅ 2. 스택 프레임(Stack Frame)이란?

> JVM이 메서드 실행을 위해 사용하는 **임시 메모리 공간**.
> **호출 시 생성되고, 종료 시 제거되는 구조적 단위**

🔹 주요 구성 요소

구성 요소	설명
지역 변수 배열	`int a = 10;` 등 메서드 내부의 변수들이 저장됨
피연산자 스택	연산 수행에 사용되는 임시 값 저장소 (JVM 명령어의 계산 대상)
프레임 데이터	메서드 호출자 정보, return 주소 등 (JVM 내부용)

🔹 시각적 예시 (단일 호출)

```
public int sum(int a, int b) {
    int result = a + b;
    return result;
}
```

호출 시 JVM 내부 구조:

```
Stack (Top ↓)
┌──────────────────────┐
| result = a + b       | ← 지역 변수: a, b, result
| 피연산자: a, b        | ← JVM 연산자 스택
| return address        |
└──────────────────────┘
```

메서드가 끝나면 이 **스택 프레임 전체가 제거(pop)**됨

✅ 3. 메서드 호출 시 스택 프레임 동작 순서

1. 메서드 호출 → 새로운 스택 프레임 할당
2. 매개변수와 지역 변수 저장
3. 연산 수행 (피연산자 스택 사용)
4. `return` → 반환값 상위 프레임에 전달
5. 현재 프레임 제거(pop)
6. 이전 메서드로 복귀

✅ 4. 중첩 호출 (Call Stack 구조)

```
public static void main(String[] args) {
    greet("Alice");
}

public static void greet(String name) {
    String message = getMessage(name);
    System.out.println(message);
}

public static String getMessage(String name) {
    return "Hello, " + name;
}
```

호출 순서에 따라 스택 구성은 다음과 같음:

```
Stack (Top ↓)
┌──────────────────────┐
| getMessage(name)     |
└──────────────────────┘
┌──────────────────────┐
| greet(name)          |
└──────────────────────┘
┌──────────────────────┐
| main(args)           |
└──────────────────────┘
```

- `getMessage`가 종료되면 프레임 제거 → `greet` → `main`

✅ 5. 재귀 호출 시 스택 누적 구조

```
public int factorial(int n) {
    if (n == 1) return 1;
    return n * factorial(n - 1);
}
```

- `factorial(5)` 호출 시 → **5개의 스택 프레임이 연속적으로 쌓임**
- 재귀 종료 조건 도달 → 역순으로 pop → 반환값 전달

✅ 6. 스택 오버플로우(StackOverflowError)

> 너무 많은 스택 프레임이 쌓이면 → JVM 스택 한계 초과 → `StackOverflowError` 발생

예:

```
public void callMe() {
    callMe();  // 무한 재귀 호출
}
```

✅ 7. 메서드와 스택 프레임 정리 요약

항목	설명
스택 프레임	메서드 호출 시 JVM이 생성하는 실행 환경
지역 변수 배열	변수 저장
피연산자 스택	JVM 바이트코드 연산 처리
호출 시	새로운 프레임 push
종료 시	프레임 pop
중첩 호출	콜 스택에 순서대로 쌓임
재귀 호출	반복적으로 쌓이며 종료 조건 필수
스택 오버플로우	무한 재귀 등으로 발생

☑️ JVM 메모리 구조 내에서의 위치

```
[ Method Area ]
      ↑
[ Heap (객체 저장) ]
      ↑
[ Stack (스레드별, 메서드 호출용) ] ← 오늘의 주제
```