9. 메서드

#1.인강/0.자바/1.자바-입문

메서드 시작

두 숫자를 입력 받아서 더하고 출력하는 단순한 기능을 개발해보자.
먼저 `1 + 2`를 수행하고, 그 다음으로 `10 + 20`을 수행할 것이다.

Method1

```
package method;

public class Method1 {

    public static void main(String[] args) {
        //계산1
        int a = 1;
        int b = 2;
        System.out.println(a + "+" + b + " 연산 수행");
        int sum1 = a + b;
        System.out.println("결과1 출력:" + sum1);

        //계산2
        int x = 10;
        int y = 20;
        System.out.println(x + "+" + y + " 연산 수행");
```

```
        int sum2 = x + y;
        System.out.println("결과2 출력:" + sum2);
    }

}
```

- 같은 연산을 두 번 수행한다.
- 코드를 잘 보면 계산1 부분과, 계산2 부분이 거의 같다.

계산1

```
int a = 1;
int b = 2;
System.out.println(a + "+" + b + " 연산 수행");
int sum1 = a + b;
```

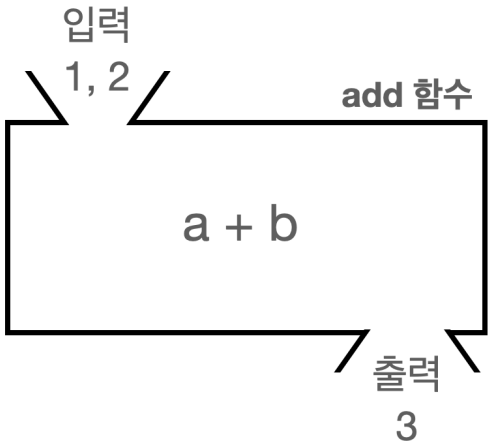
계산2

```
int x = 10;
int y = 20;
System.out.println(x + "+" + y + " 연산 수행");
int sum2 = x + y;
```

계산1, 계산2 둘 다 변수를 두 개 선언하고, 어떤 연산을 수행하는지 출력하고, 두 변수를 더해서 결과를 구한다.
만약 프로그램의 여러 곳에서 이와 같은 계산을 반복해야 한다면? 같은 코드를 여러번 반복해서 작성해야 할 것이다.
더 나아가서 어떤 연산을 수행하는지 출력하는 부분을 변경하거나 또는 제거하고 싶다면 해당 코드를 다 찾아다니면서 모두 수정해야 할 것이다.
이런 문제를 어떻게 깔끔하게 해결할 수 있을까?

잠깐 아주 간단하게 수학의 함수를 알아보자.

함수(function)



수학 용어가 나왔다고 전혀 어렵게 생각할 것이 없다! 숫자를 2개 입력하면 해당 숫자를 더한 다음에 그 결과를 출력하는 아주 단순한 함수이다. 이 함수의 이름은 `add` 이다.

함수 정의

```
add(a, b) = a + b
```

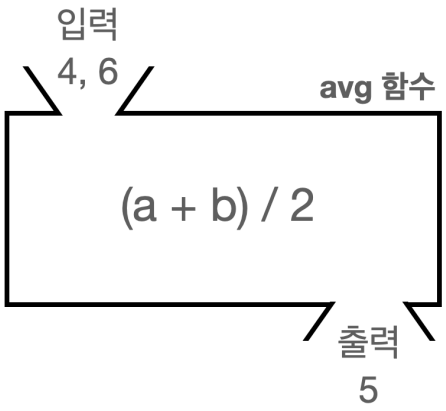
- 이름이 `add` 이고 `a`, `b` 라는 두 값을 받는 함수이다. 그리고 이 함수는 `a + b` 연산을 수행한다.

함수 사용

```
add(1,2) -> 결과:3
add(5,6) -> 결과:11
add(3,5) -> 결과:8
```

- 함수에 값을 입력하면, 함수가 가진 연산을 처리한 다음 결과를 출력한다. 여기서는 단순히 `a+b` 라는 연산을 수행한다.
- 여러번 같은 계산을 해야 한다면 지금처럼 함수를 만들어두고(정의), 필요한 입력 값을 넣어서 해당 함수를 호출하면 된다. 그러면 계산된 결과가 나온다.
- 함수는 마치 블랙박스와 같다. 함수를 호출할 때는 외부에서는 필요한 값만 입력하면 된다. 그러면 계산된 결과가 출력된다.
- 같은 함수를 다른 입력 값으로 여러번 호출할 수 있다.
- 여기서 핵심은 함수를 한번 정의해두면 계속해서 재사용할 수 있다는 점이다!

평균 함수



만약 두 수의 평균을 구해야 한다면 매번 (a + b) / 2라는 공식을 사용해야 할 것이다.
이것을 함수로 만들어두면 다음과 같이 사용하면 된다.

함수 정의

```
avg(a, b) = (a + b) / 2
```

함수 사용

```
avg(4,6) -> 결과:5
avg(10,20) -> 결과:15
avg(100,200) -> 결과:150
```

수학의 함수의 개념을 프로그래밍에 가지고 온다면 어떨까? 필요한 기능을 미리 정의해두고 필요할 때 마다 호출해서 사용할 수 있기 때문에 앞서 고민한 문제들을 해결할 수 있을 것 같다.
프로그램 언어들은 오래 전 부터 이런 문제를 해결하기 위해 수학의 함수라는 개념을 차용해서 사용한다.

메서드 사용

자바에서는 함수를 메서드(Method)라 한다.
메서드도 함수의 한 종류라고 생각하면 된다. 지금은 둘을 구분하지 않고, 이정도만 알아두자.

메서드를 사용하면 앞서 고민한 문제를 한번에 해결할 수 있다.
메서드에 대한 자세한 설명보다는 우선 메서드를 사용해서 코드를 작성해보자. 참고로 앞에서 작성한 코드와 완전히 동일하게 작동하는 코드이다.

Method1Ref

```
package method;

public class Method1Ref {

    public static void main(String[] args) {
        int sum1 = add(5, 10);
        System.out.println("결과1 출력:" + sum1);

        int sum2 = add(15, 20);
        System.out.println("결과2 출력:" + sum2);
    }

    //add 메서드
    public static int add(int a, int b) {
        System.out.println(a + "+" + b + " 연산 수행");
        int sum = a + b;
        return sum;
    }
```

```
}
```

실행 결과

```
5+10 연산 수행
결과1 출력:15
15+20 연산 수행
결과2 출력:35
```

중복이 제거되고, 코드가 상당히 깔끔해진 것을 느낄 수 있을 것이다.

메서드 정의

```
public static int add(int a, int b) {
    System.out.println(a + "+" + b + " 연산 수행");
    int sum = a + b;
    return sum;
}
```

이 부분이 바로 메서드이다. 이것을 함수를 정의하는 것과 같이, 메서드를 정의한다고 표현한다.
메서드는 수학의 함수와 유사하게 생겼다. 함수에 값을 입력하면, 어떤 연산을 처리한 다음에 결과를 반환한다.
(수학에 너무 집중하지는 말자, 단순히 무언가 정의해두고 필요할 때 불러서 사용한다는 개념으로 이해하면 충분하다)

메서드는 크게 **메서드 선언**과 **메서드 본문**으로 나눌 수 있다.

메서드 선언(Method Declaration)

`public static int add(int a, int b)`

메서드의 선언 부분으로, 메서드 이름, 반환 타입, 매개변수(파라미터) 목록을 포함한다.
이름 그대로 이런 메서드가 있다고 선언하는 것이다. 메서드 선언 정보를 통해 다른 곳에서 해당 메서드를 호출할 수 있다.

- `public static`
  - `public` : 다른 클래스에서 호출할 수 있는 메서드라는 뜻이다. 접근 제어에서 학습한다.
  - `static` : 객체를 생성하지 않고 호출할 수 있는 정적 메서드라는 뜻이다. 자세한 내용은 뒤에서 다룬다.
  - 두 키워드의 자세한 내용은 뒤에서 다룬다. 지금은 단순하게 메서드를 만들 때 둘을 사용해야 한다고 생각하자.
- `int add(int a, int b)`
  - `int` : 반환 타입을 정의한다. 메서드의 실행 결과를 반환할 때 사용할 반환 타입을 지정한다.
  - `add` : 메서드에 이름을 부여한다. 이 이름으로 메서드를 호출할 수 있다.
  - `(int a, int b)` : 메서드를 호출할 때 전달하는 입력 값을 정의한다. 이 변수들은 해당 메서드 안에서만

사용된다. 이렇게 메서드 선언에 사용되는 변수를 영어로 파라미터(parameter), 한글로 매개변수라 한다.

메서드 본문(Method Body)

```
{
    System.out.println(a + "+" + b + " 연산 수행");
    int sum = a + b;
    return sum;
}
```

- 메서드가 수행해야 하는 코드 블록이다.
- 메서드를 호출하면 메서드 본문이 순서대로 실행된다.
- 메서드 본문은 블랙박스이다. **메서드를 호출하는 곳에서는 메서드 선언은 알지만 메서드 본문은 모른다.**
- 메서드의 실행 결과를 반환하려면 `return` 문을 사용해야 한다. `return` 문 다음에 반환할 결과를 적어주면 된다.
  - `return sum`: `sum` 변수에 들어있는 값을 반환한다.

메서드 호출

앞서 정의한 메서드를 호출해서 실행하려면 메서드 이름에 입력 값을 전달하면 된다. 보통 메서드를 호출한다고 표현한다.

```
int sum1 = add(5, 10);
int sum2 = add(15, 20);
```

메서드를 호출하면 어떻게 실행되는지 순서대로 확인해보자

```
int sum1 = add(5, 10); //add라는 메서드를 숫자 5,10을 전달하면서 호출한다.
int sum1 = 15; //add(5, 10)이 실행된다. 실행 결과는 반환 값은 15이다.
//sum1에 15 값이 저장된다.
```

메서드를 호출하면 메서드는 계산을 끝내고 결과를 반환한다. 쉽게 이야기하자면, 메서드 호출이 끝나면 해당 메서드가 반환한 결과 값으로 치환된다.

조금 더 자세히 알아보자. 메서드의 코드는 일부 축약했다.

```
//1: 메서드 호출
int sum1 = add(5, 10);

//2: 파라미터 변수 a=5, b=10이 전달되면서 메서드가 수행된다.
public static int add(int a=5, int b=10) {
    int sum = a + b;
    return sum;
}

//3: 메서드가 수행된다.
```

```
public static int add(int a=5, int b=10) {
    int sum = a(5) + b(10);
    return sum;
}

//4: return을 사용해서 메서드 실행의 결과인 sum을 반환한다. sum에는 값 15가 들어있으므로 값 15가 반환된다.
public static int add(int a=5, int b=10) {
    int sum = 15;
    return sum(15);
}

//5: 메서드 호출 결과로 메서드에서 반환한 값 15가 나온다. 이 값을 sum1에 대입했다.
int sum1 = 15;
```

메서드 호출이 끝나면 더 이상 해당 메서드가 사용한 메모리를 낭비할 이유가 없다. 메서드 호출이 끝나면 메서드 정의에 사용한 파라미터 변수인 `int a`, `int b`는 물론이고, 그 안에서 정의한 `int sum`도 모두 제거된다.

메서드 호출과 용어정리

메서드를 호출할 때는 다음과 같이 메서드에 넘기는 값과 매개변수(파라미터)의 타입이 맞아야 한다. 물론 넘기는 값과 매개변수(파라미터)의 순서와 갯수도 맞아야 한다.

```
호출: call("hello", 20)
메서드 정의: int call(String str, int age)
```

인수(Argument)

여기서 `"hello"`, `20`처럼 넘기는 값을 영어로 Argument(아규먼트), 한글로 인수 또는 인자라 한다.
실무에서는 아규먼트, 인수, 인자라는 용어를 모두 사용한다.

매개변수(Parameter)

메서드를 정의할 때 선언한 변수인 `String str`, `int age`를 매개변수, 파라미터라 한다.
메서드를 호출할 때 인수를 넘기면, 그 인수가 매개변수에 대입된다.
실무에서는 매개변수, 파라미터 용어를 모두 사용한다.

용어정리

- **인수**라는 용어는 '인'과 '수'의 합성어로, '들어가는 수'라는 의미를 가진다. 즉, 메서드 내부로 들어가는 값을 의미한다. 인자도 같은 의미이다.
- **매개변수, parameter**는 '매개'와 '변수'의 합성어로, '중간에서 전달하는 변수'라는 의미를 가진다. 즉, 메서드 호

출부와 메서드 내부 사이에서 값을 전달하는 역할을 하는 변수라는 뜻이다.

메서드 정의

메서드는 다음과 같이 정의한다.

```
public static int add(int a, int b) {
  //메서드 본문, 실행 코드
}

제어자 반환타입 메서드이름(매개변수 목록) {
  메서드 본문
}
```

- **제어자(Modifier)**: `public`, `static`과 같은 부분이다. 제어자는 뒤에서 설명한다. 지금은 항상 `public static` 키워드를 입력하자.
- **반환 타입(Return Type)**: 메서드가 실행 된 후 반환하는 데이터의 타입을 지정한다. 메서드가 값을 반환하지 않는 경우, 없다는 뜻의 `void`를 사용해야 한다. 예) `void print(String str)`
- **메서드 이름(Method Name)**: 메서드의 이름이다. 이 이름은 메서드를 호출하는 데 사용된다.
- **매개변수(Parameter)**: 입력 값으로, 메서드 내부에서 사용할 수 있는 변수이다. 매개변수는 옵션이다. 입력값이 필요 없는 메서드는 매개변수를 지정하지 않아도 된다. 예) `add()`
- **메서드 본문(Method Body)**: 실제 메서드의 코드가 위치한다. 중괄호 `{}` 사이에 코드를 작성한다.

매개변수가 없거나 반환 타입이 없는 경우

매개변수가 없고, 반환 타입도 없는 메서드를 확인해보자.

```
package method;

public class Method2 {

    public static void main(String[] args) {
        printHeader();
        System.out.println("프로그램이 동작합니다.");
        printFooter();
    }

    public static void printHeader() {
```

```
        System.out.println("= 프로그램을 시작합니다 =");
        return; //void의 경우 생략 가능
    }

    public static void printFooter() {
        System.out.println("= 프로그램을 종료합니다 =");
    }

}
```

실행 결과

```
= 프로그램을 시작합니다 =
프로그램이 동작합니다.
= 프로그램을 종료합니다 =
```

`printHeader()`, `printFooter()` 메서드는 매개변수가 없고, 반환 타입도 없다.

- 매개변수가 없는 경우
  - 선언: `public static void printHeader()` 와 같이 매개변수를 비워두고 정의하면 된다.
  - 호출: `printHeader();` 와 같이 인수를 비워두고 호출하면 된다.
- 반환 타입이 없는 경우
  - 선언: `public static void printHeader()` 와 같이 반환 타입을 `void` 로 정의하면 된다.
  - 호출: `printHeader();` 와 같이 반환 타입이 없으므로 메서드만 호출하고 반환 값을 받지 않으면 된다.
    - `String str = printHeader();` 반환 타입이 `void` 이기 때문에 이렇게 반환 값을 받으면 컴파일 오류가 발생한다.

void와 return 생략

모든 메서드는 항상 `return` 을 호출해야 한다. 그런데 반환 타입 `void` 의 경우에는 예외로 `printFooter()` 와 같이 생략해도 된다. 자바가 반환 타입이 없는 경우에는 `return` 을 마지막줄에 넣어준다. 참고로 `return` 을 만나면 해당 메서드는 종료된다.

반환 타입

반환 타입이 있으면 반드시 값을 반환해야 한다.

반환 타입이 있는 메서드는 반드시 `return` 을 사용해서 값을 반환해야 한다.

이 부분은 특히 조건문과 함께 사용할 때 주의해야 한다.

MethodReturn1

```
package method;

public class MethodReturn1 {

    public static void main(String[] args) {
        boolean result = odd(2);
        System.out.println(result);
    }

    public static boolean odd(int i) {
        if (i % 2 == 1) {
            return true;
        }
    }
}
```

이 코드에서 `if` 조건이 만족할 때는 `true`가 반환되지만, 조건을 만족하지 않으면 어떻게 될까? 조건을 만족하지 않은 경우에는 `return`문이 실행되지 않는다. 따라서 이 코드를 실행하면 `return` 문을 누락했다는 다음과 같은 컴파일 오류가 발생한다.

컴파일 오류

```
java: missing return statement
```

MethodReturn1 - 수정 코드

```
package method;

public class MethodReturn1 {

    public static void main(String[] args) {
        boolean result = odd(2);
        System.out.println(result);
    }

    public static boolean odd(int i) {
        if (i % 2 == 1) {
            return true;
        } else {
            return false;
        }
```

```
    }
}
```

이렇게 수정하면 `if` 조건을 만족하지 않아도 `else`를 통해 `return`문이 실행된다.

return 문을 만나면 그 즉시 메서드를 빠져나간다.

`return` 문을 만나면 그 즉시 해당 메서드를 빠져나간다.

다음 로직을 수행하는 메서드를 만들어보자.

- 18살 미만의 경우: 미성년자는 출입이 불가합니다.
- 18살 이상의 경우: 입장하세요.

MethodReturn2

```
package method;

public class MethodReturn2 {

    public static void main(String[] args) {
        checkAge(10);
        checkAge(20);
    }

    public static void checkAge(int age) {
        if (age < 18) {
            System.out.println(age + "살, 미성년자는 출입이 불가능합니다.");
            return;
        }

        System.out.println(age + "살, 입장하세요.");
    }
}
```

- 18세 미만의 경우 "미성년자는 출입이 불가능합니다"를 출력하고 바로 `return`문이 수행된다. 따라서 다음 로직을 수행하지 않고, 해당 메서드를 빠져나온다.
- 18세 이상의 경우 "입장하세요"를 출력하고, 메서드가 종료된다. 참고로 반환 타입이 없는 `void`형이기 때문에 마지막 줄의 `return`은 생략할 수 있다.

반환 값 무시

반환 타입이 있는 메서드를 호출했는데 만약 반환 값이 필요없다면 사용하지 않아도 된다.

예시1: `int sum = add(1,2)` //반환된 값을 받아서 `sum`에 저장했다.
예시2: `add(1,2)` //반환된 값을 사용하지 않고 버린다. 여기서는 예시1과 같이 호출 결과를 변수에 담지 않았다. 단순히 메서드만 호출했다.

메서드 호출과 값 전달1

지금부터 자바에서 아주 중요한 대원칙 하나를 이야기하겠다. 밑줄 100번 긋자!

자바는 항상 변수의 값을 복사해서 대입한다.

이 대원칙은 반드시 이해해야 한다. 그러면 아무리 복잡한 상황에도 코드를 단순하게 이해할 수 있다.

변수와 값 복사

다음 코드를 보고 어떤 결과가 나올지 먼저 생각해보자.

MethodValue0

```
package method;

public class MethodValue0 {

    public static void main(String[] args) {
        int num1 = 5;
        int num2 = num1;
        num2 = 10;
        System.out.println("num1=" + num1);
        System.out.println("num2=" + num2);
    }

}
```

실행 결과

```
num1=5
num2=10
```

실행 과정

```
int num2 = num1; //num1의 값은 5이다. num1(5)
int num2 = 5; //num2 변수에 대입하기 전에 num1의 값 5를 읽는다. 결과: num1(5), num2(5)
num2 = 10; // num2에 10을 대입한다. 결과: num1(5), num2(10)
```

여기서 값을 복사해서 대입한다는 부분이 바로 이 부분이다.

```
int num2 = num1;
```

- 이 부분은 생각해보면 `num1`에 있는 값 `5`를 복사해서 `num2`에 넣는 것이다.
  - 복사한다고 표현한 이유는 `num1`의 값을 읽어도 `num1`에 있는 기존 값이 유지되고, 새로운 값이 `num2`에 들어가기 때문이다. 마치 `num1`의 값이 `num2`에 복사가 된 것 같다.
  - `num1`이라는 변수 자체가 `num2`에 들어가는 것이 아니다. `num1`에 들어있는 값을 읽고 복사해서 `num2`에 넣는 것이다.
  - 간단하게 `num1`에 있는 값을 `num2`에 대입한다고 표현한다. 하지만 실제로는 그 값을 복사해서 대입하는 것이다.

너무 당연한 이야기를 왜 이렇게 장황하게 풀어서 하지? 라고 생각한다면 이제 진짜 문제를 만나보자.

메서드 호출과 값 복사

다음은 숫자를 2배 곱하는 메서드이다. 다음 코드를 보고 어떤 결과가 나올지 먼저 생각해보자.

MethodValue1

```
package method;

public class MethodValue1 {

    public static void main(String[] args) {
        int num1 = 5;
        System.out.println("1. changeNumber 호출 전, num1: " + num1);
        changeNumber(num1);
        System.out.println("4. changeNumber 호출 후, num1: " + num1);
    }

    public static void changeNumber(int num2) {
        System.out.println("2. changeNumber 변경 전, num2: " + num2);
        num2 = num2 * 2;
        System.out.println("3. changeNumber 변경 후, num2: " + num2);
    }
}
```

실행 결과를 먼저 예측해보고 그 다음에 실행 결과를 확인해보자.

혹시라도 실행 결과의 마지막이 10이라고 생각했다면 대원칙을 떠올려보자

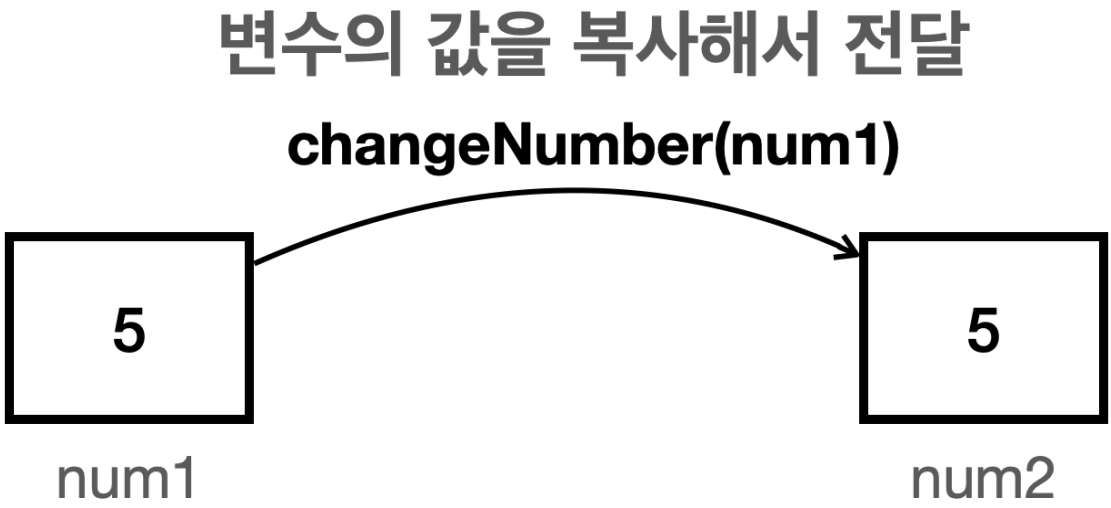
실행 결과

```
1. changeNumber 호출 전, num1: 5
2. changeNumber 변경 전, num2: 5
3. changeNumber 변경 후, num2: 10
4. changeNumber 호출 후, num1: 5
```

다음 대원칙을 따라간다면 문제를 정확하게 풀 수 있다.

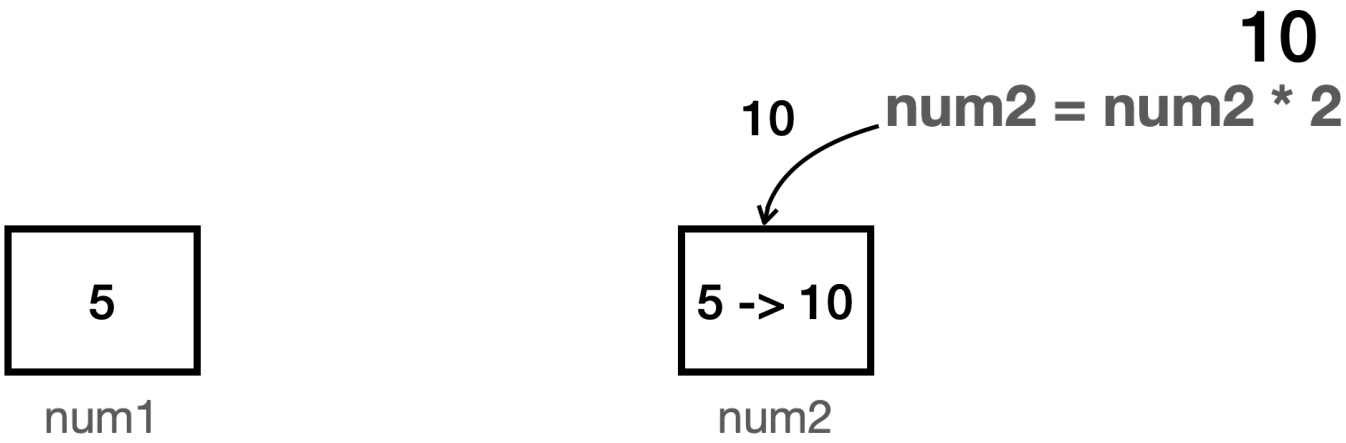
자바는 항상 변수의 값을 복사해서 대입한다.

실행 과정 그림



changeNumber(num1) 호출 시점

- `num1` 의 값 5를 읽고 복사해서 `num2` 에 전달 ➜ 이 부분이 핵심



changeNumber메서드 실행 중

- `num2` 의 변경은 `num1` 에 영향을 주지 않는다. 왜냐하면 앞서 값을 복사해서 전달했기 때문이다.

최종 결과

실행 과정 코드

```
changeNumber(num1); //changeNumber를 호출한다. num1(5)
changeNumber(5); //num1의 값을 읽는다.

void changeNumber(int num2=5) //num1의 값 5가 num2에 복사된다. 결과: num1(5), num2(5)
num2 = num2 * 2; //num2에 2를 곱한다. 결과: num1(5), num2(5)
num2 = 5 * 2; //num2의 값을 읽어서 2를 곱한다. 결과: num1(5), num2(5)
num2 = 10; //num2에 계산 결과인 값 10을 대입한다. 결과: num1(5), num2(10)

num2를 출력한다: num2의 값인 10이 출력된다.
num1을 출력한다: num1의 값인 5가 출력된다.
```

결과적으로 매개변수 `num2`의 값만 10으로 변경되고 `num1`의 값은 변경되지 않고 기존 값인 5로 유지된다. 자바는 항상 값을 복사해서 전달하기 때문에 `num2`의 값을 바꾸더라도 `num1`에는 영향을 주지 않는다.

메서드 호출과 값 전달2

메서드 호출과 이름이 같은 변수

같은 문제를 호출자의 변수 이름과 매개변수의 이름을 같게 해서 한번 더 풀어보자

MethodValue2

```
package method;

public class MethodValue2 {

    public static void main(String[] args) {
        int number = 5;
        System.out.println("1. changeNumber 호출 전, number: " + number); // 출력:
```

```
5
        changeNumber(number);
        System.out.println("4. changeNumber 호출 후, number: " + number); // 출력:
5
    }

    public static void changeNumber(int number) {
        System.out.println("2. changeNumber 변경 전, number: " + number); // 출력:
5
        number = number * 2;
        System.out.println("3. changeNumber 변경 후, number: " + number); // 출력:
10
    }
}
```

이번에는 `main()`에 정의한 변수와 메서드의 매개변수(파라미터)의 이름이 둘다 `number`로 같다.

실행 결과

```
1. changeNumber 호출 전, number: 5
2. changeNumber 변경 전, number: 5
3. changeNumber 변경 후, number: 10
4. changeNumber 호출 후, number: 5
```

`main()`도 사실은 메서드이다. 각각의 메서드 안에서 사용하는 변수는 서로 완전히 분리된 다른 변수이다. 물론 이름이 같아도 완전히 다른 변수다. 따라서 `main()`의 `number`와 `changeNumber()`의 `number`는 서로 다른 변수이다.

실행 과정

```
changeNumber(number); //changeNumber를 호출한다. main의number(5)
changeNumber(5); //number의 값을 읽는다.

//main의 number값 5가 changeNumber의 number에 복사된다.
//결과: main의 number(5), changeNumber의 number(5)
void changeNumber(int number=5)

//changeNumber의 number에 값 10을 대입한다.
//결과: main의 number(5), changeNumber의 number(10)
number = number * 2;
main의 number을 출력한다: main의 number의 값인 5가 출력된다.
```

메서드 호출과 값 반환받기

그렇다면 메서드를 사용해서 값을 변경하려면 어떻게 해야할까?

메서드의 호출 결과를 반환 받아서 사용하면 된다.

MethodValue3

```
package method;

public class MethodValue3 {

    public static void main(String[] args) {
        int num1 = 5;
        System.out.println("changeNumber 호출 전, num1: " + num1); // 출력: 5
        num1 = changeNumber(num1);
        System.out.println("changeNumber 호출 후, num1: " + num1); // 출력: 10
    }

    public static int changeNumber(int num2) {
        num2 = num2 * 2;
        return num2;
    }
}
```

실행 결과

```
changeNumber 호출 전, num1: 5
changeNumber 호출 후, num1: 10
```

실행 과정

```
num1 = changeNumber(num1); //num1(5)
num1 = changeNumber(5);

//호출 시작:changeNumber()
//num1의 값 5가 num2에 대입된다. num1의 값을 num2에 복사한다. num1(5), num2(5)
int changeNumber(int num2=5)
num2 = num2 * 2; //계산 결과: num1(5), num2(10)
return num2; // num2의 값은 10이다.
return 10;
//호출 끝: changeNumber()

num1 = changeNumber(5);//반환 결과가 10이다.
num1 = 10;//결과: num1(10)
```

정리

꼭 기억하자! **자바는 항상 변수의 값을 복사해서 대입한다.**

(참고로 뒤에서 참조형이라는 것을 학습하는데, 이때도 똑같다. 결국 참조형 변수에 있는 값인 **참조값을 복사**하는 것이다! 이것은 나중에 알아본다.)

메서드와 형변환

메서드를 호출할 때도 형변환이 적용된다. 메서드 호출과 명시적 형변환, 자동 형변환에 대해 알아보자.

명시적 형변환

메서드를 호출하는데 인자와 매개변수의 타입이 맞지 않다면 어떻게 해야할까?

다음 예제 코드를 보자

MethodCasting1

```
package method;

public class MethodCasting1 {

    public static void main(String[] args) {
        double number = 1.5;
        //printNumber(number); // double을 int형에 대입하므로 컴파일 오류
        printNumber((int) number); // 명시적 형변환을 사용해 double을 int로 변환
    }

    public static void printNumber(int n) {
        System.out.println("숫자: " + n);
    }
}
```

먼저 주석으로 처리해둔 부분의 주석을 풀고 실행해보자.

```
printNumber(number) // double을 int형에 대입하므로 컴파일 오류
```

실행 결과 - 컴파일 오류

```
java: incompatible types: possible lossy conversion from double to int
```

다음과 같은 이유로 컴파일 오류가 발생한다.

```
printNumber(number) //number는 1.5 실수
printNumber(1.5) //메서드를 호출하기 전에 number 변수의 값을 읽음
void printNumber(int n=1.5) //int형 매개변수 n에 double형 실수인 1.5를 대입 시도, 컴파일 오류
```

이 경우 메서드 호출이 꼭 필요하다면 다음과 같이 명시적 형변환을 사용해야 한다.

```
printNumber((int) number); // 명시적 형변환을 사용해 double을 int로 변환
printNumber(1); // (double) 1.5 -> (int) 1로 변환
void printNumber(int n=1) //int형 파라미터 변수 n에 int형 1을 대입
```

실행 결과

```
숫자: 1
```

자동 형변환

`int < long < double`

메서드를 호출할 때 매개변수에 값을 전달하는 것도 결국 변수에 값을 대입하는 것이다. 따라서 앞서 배운 자동 형변환이 그대로 적용된다.

```
package method;

public class MethodCasting2 {

    public static void main(String[] args) {
        int number = 100;
        printNumber(number); // int에서 double로 자동 형변환
    }

    public static void printNumber(double n) {
        System.out.println("숫자: " + n);
    }
}
```

- `double`형 매개변수(파라미터)에 `int`형 인수를 전달하는데 문제없이 잘 동작한다.

실행 결과

```
숫자: 100.0
```

다음과 같이 자동 형변환이 동작한다.

```
printNumber(number); // number는 int형 100
printNumber(100); //메서드를 호출하기 전에 number 변수의 값을 읽음

void printNumber(double n=100) //double형 파라미터 변수 n에 int형 값 100을 대입
void printNumber(double n=(double) 100) //double이 더 큰 숫자 범위이므로 자동 형변환 적용
void printNumber(double n=100.0) //자동 형변환 완료
```

정리

메서드를 호출할 때는 전달하는 인수의 타입과 매개변수의 타입이 맞아야 한다. 단 타입이 달라도 자동 형변환이 가능한 경우에는 호출할 수 있다.

메서드 오버로딩

다음과 같은 메서드를 만들고 싶다.

- 두 수를 더하는 메서드
- 세 수를 더하는 메서드

이 경우 둘다 더하는 메서드이기 때문에 가급적 같은 이름인 `add`를 사용하고 싶다.
자바는 메서드의 이름 뿐만 아니라 매개변수 정보를 함께 사용해서 메서드를 구분한다.
따라서 다음과 같이 이름이 같고, 매개변수가 다른 메서드를 정의할 수 있다.

오버로딩 성공

```
add(int a, int b)
add(int a, int b, int c)
add(double a, double b)
```

이렇게 이름이 같고 매개변수가 다른 메서드를 여러개 정의하는 것을 메서드 오버로딩(Overloading)이라 한다.
오버로딩은 번역하면 과적인데, 과하게 물건을 담았다는 뜻이다. 따라서 같은 이름의 메서드를 여러개 정의했다고 이해하면 된다.

오버로딩 규칙

메서드의 이름이 같아도 매개변수의 타입 및 순서가 다르면 오버로딩을 할 수 있다. 참고로 반환 타입은 인정하지 않는다.

다음 케이스는 메서드 이름과 매개변수의 타입이 같으므로 컴파일 오류가 발생한다. 반환 타입은 인정하지 않는다.

오버로딩 실패

```
int add(int a, int b)
double add(int a, int b)
```

용어: 메서드 시그니처(method signature)

`메서드 시그니처 = 메서드 이름 + 매개변수 타입(순서)`

메서드 시그니처는 자바에서 메서드를 구분할 수 있는 고유한 식별자나 서명을 뜻한다. 메서드 시그니처는 메서드의 이름과 매개변수 타입(순서 포함)으로 구성되어 있다. 쉽게 이야기해서 메서드를 구분할 수 있는 기준이다. 자바 입장에서는 각각의 메서드를 고유하게 구분할 수 있어야한다. 그래야 어떤 메서드를 호출 할 지 결정할 수 있다.

따라서 메서드 오버로딩에서 설명한 것 처럼 메서드 이름이 같아도 메서드 시그니처가 다르면 다른 메서드로 간주한다. 반환 타입은 시그니처에 포함되지 않는다. 방금 오버로딩이 실패한 두 메서드를 보자. 두 메서드는 `add(int a, int b)` 로 메서드 시그니처가 같다. 따라서 메서드의 구분이 불가능하므로 컴파일 오류가 발생한다.

다양한 예제를 통해서 메서드 오버로딩을 알아보자.

먼저 매개변수의 갯수가 다른 오버로딩 예제를 보자

Overloading1

```
package overloading;

public class Overloading1 {

    public static void main(String[] args) {
        System.out.println("1: " + add(1, 2));
        System.out.println("2: " + add(1, 2, 3));
    }

    // 첫 번째 add 메서드: 두 정수를 받아서 합을 반환한다.
    public static int add(int a, int b) {
        System.out.println("1번 호출");
        return a + b;
    }

    // 두 번째 add 메서드: 세 정수를 받아서 합을 반환한다.
    // 첫 번째 메서드와 이름은 같지만, 매개변수 목록이 다르다.
    public static int add(int a, int b, int c) {
        System.out.println("2번 호출");
        return a + b + c;
    }
```

```
}
```

1: 정수 1,2를 호출했으므로 `add(int a, int b)`가 호출된다.
2: 정수 1,2,3을 호출했으므로 `add(int a, int b, int c)`가 호출된다.

실행 결과

```
1번 호출
1: 3
2번 호출
2: 6
```

이번에는 매개변수의 타입이 다른 오버로딩 예제를 보자

Overloading2

```
package overloading;

public class Overloading2 {

    public static void main(String[] args) {
        myMethod(1, 1.2);
        myMethod(1.2, 2);
    }

    public static void myMethod(int a, double b) {
        System.out.println("int a, double b");
    }

    public static void myMethod(double a, int b) {
        System.out.println("double a, int b");
    }
}
```

1: 정수1, 실수 1.2를 호출했으므로 `myMethod(int a, double b)`가 호출된다.
2: 실수 1.2, 정수 2를 호출했으므로 `myMethod(double a, int b)`가 호출된다.

실행 결과

```
int a, double b
double a, int b
```

마지막으로 매개변수의 타입이 다른 경우를 추가로 확인해보자.

Overloading3

```
package overloading;

public class Overloading3 {

    public static void main(String[] args) {
        System.out.println("1: " + add(1, 2));
        System.out.println("2: " + add(1.2, 1.5));
    }

    // 첫 번째 add 메서드: 두 정수를 받아서 합을 반환한다.
    public static int add(int a, int b) {
        System.out.println("1번 호출");
        return a + b;
    }

    // 두 번째 add 메서드: 두 실수를 받아서 합을 반환한다.
    // 첫 번째 메서드와 이름은 같지만, 매개변수의 유형이 다르다.
    public static double add(double a, double b) {
        System.out.println("2번 호출");
        return a + b;
    }

}
```

1: 정수1, 정수 2를 호출했으므로 `add(int a, int b)`가 호출된다.

2: 실수 1.2, 실수 1.5를 호출했으므로 `add(double a, double b)` 가 호출된다.

실행 결과

```
1번 호출
1: 3
2번 호출
2: 2.7
```

여기서 만약 다음 첫 번째 메서드를 삭제하면 어떻게 될까?

```
public static int add(int a, int b) {
    System.out.println("1번 호출");
    return a + b;
}
```

1: `int` 형 정수 1, `int` 형 정수 2를 호출했으므로 자동 형변환이 발생해서 `add(double a, double b)`가 호출된다.
2: 실수 1.2, 실수 1.5를 호출했으므로 `add(double a, double b)` 가 호출된다.

실행 결과

```
2번 호출
1: 3.0
2번 호출
2: 2.7
```

정리하면 먼저 본인의 타입에 최대한 맞는 메서드를 찾아서 실행하고, 그래도 없으면 형 변환 가능한 타입의 메서드를 찾아서 실행한다.

문제와 풀이1

> **코딩이 처음이라면 필독!**
> 프로그래밍이 처음이라면 아직 코딩 자체가 익숙하지 않기 때문에 문제와 풀이에 상당히 많은 시간을 쓰게 될 수 있다. 강의를 들을 때는 다 이해가 되는 것 같았는데, 막상 혼자 생각해서 코딩을 하려니 잘 안되는 것이다. 이것은 아직 코딩이 익숙하지 않기 때문인데, 처음 코딩을 하는 사람이라면 누구나 겪는 자연스러운 현상이다.
> 문제를 스스로 풀기 어려운 경우, 너무 고민하기 보다는 먼저 **강의 영상의 문제 풀이 과정을 코드로 따라하면서 이해하자. 반드시 코드로 따라해야 한다.** 그래야 코딩하는 것에 조금씩 익숙해질 수 있다. 그런 다음에 정답을 지우고 스스로 문제를 풀어보면 된다. 참고로 강의를 듣는 시간만큼 문제와 풀이에도 많은 시간을 들어야 제대로 성장할 수 있다!

문제 - 평균값 리펙토링

메서드를 잘 이해하고 있는지 확인하기 위해 다음 코드를 메서드를 사용하도록 리펙토링해보자.

MethodEx1

```
package method.ex;

public class MethodEx1 {
    public static void main(String[] args) {
        int a = 1;
```

```
        int b = 2;
        int c = 3;

        int sum = a + b + c;
        double average = sum / 3.0;
        System.out.println("평균값: " + average);

        int x = 15;
        int y = 25;
        int z = 35;

        sum = x + y + z;
        average = sum / 3.0;
        System.out.println("평균값: " + average);
    }
}
```

실행 결과

```
평균값: 2.0
평균값: 25.0
```

정답

```
package method.ex;

public class MethodEx1Ref {

    public static void main(String[] args) {
        System.out.println("평균값: " + average(1, 2, 3));
        System.out.println("평균값: " + average(15, 25, 35));
    }

    public static double average(int a, int b, int c) {
        int sum = a + b + c;
        return sum / 3.0;
    }
}
```

문제 - 반복 출력 리펙토링

다음은 특정 숫자만큼 같은 메시지를 반복 출력하는 기능이다.

메서드를 사용해서 리펙토링해보자.

MethodEx2

```
package method.ex;

public class MethodEx2 {
    public static void main(String[] args) {
        String message = "Hello, world!";

        for (int i = 0; i < 3; i++) {
            System.out.println(message);
        }

        for (int i = 0; i < 5; i++) {
            System.out.println(message);
        }

        for (int i = 0; i < 7; i++) {
            System.out.println(message);
        }
    }
}
```

실행 결과

```
Hello, world!
Hello, world!
... //여러번 반복
```

정답

```
package method.ex;

public class MethodEx2Ref {
    public static void main(String[] args) {
        printMessage("Hello, world!", 3);
        printMessage("Hello, world!", 5);
        printMessage("Hello, world!", 7);
    }

    public static void printMessage(String message, int times) {
        for (int i = 0; i < times; i++) {
            System.out.println(message);
        }
```

```
    }
}
```

문제 - 입출금 리펙토링

다음은 입금, 출금을 나타내는 코드이다.

입금(deposit)과, 출금(withdraw)을 메서드로 만들어서 리펙토링 해보자.

```
package method.ex;

public class MethodEx3 {
    public static void main(String[] args) {
        int balance = 10000;

        // 입금 1000
        int depositAmount = 1000;
        balance += depositAmount;
        System.out.println(depositAmount + "원을 입금하였습니다. 현재 잔액: " + balance
+ "원");

        // 출금 2000
        int withdrawAmount = 2000;
        if (balance >= withdrawAmount) {
            balance -= withdrawAmount;
            System.out.println(withdrawAmount + "원을 출금하였습니다. 현재 잔액: " +
balance + "원");
        } else {
            System.out.println(withdrawAmount + "원을 출금하려 했으나 잔액이 부족합니
다.");
        }

        System.out.println("최종 잔액: " + balance + "원");
    }
}
```

실행 결과

```
1000원을 입금하였습니다. 현재 잔액: 11000원
2000원을 출금하였습니다. 현재 잔액: 9000원
최종 잔액: 9000원
```

정답

```java
package method.ex;

public class MethodEx3Ref {
    public static void main(String[] args) {
        int balance = 10000;

        balance = deposit(balance, 1000);
        balance = withdraw(balance, 2000);

        System.out.println("최종 잔액: " + balance + "원");
    }

    public static int deposit(int balance, int amount) {
        balance += amount;
        System.out.println(amount + "원을 입금하였습니다. 현재 잔액: " + balance +
"원");
        return balance;
    }

    public static int withdraw(int balance, int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println(amount + "원을 출금하였습니다. 현재 잔액: " + balance +
"원");
        } else {
            System.out.println(amount + "원을 출금하려 했으나 잔액이 부족합니다.");
        }
        return balance;
    }
}
```

리펙토링 결과를 보면 `main()`은 세세한 코드가 아니라 전체 구조를 한눈에 볼 수 있게 되었다. 쉽게 이야기해서 책의 목차를 보는 것 같다. 더 자세히 알고 싶으면 해당 메서드를 찾아서 들어가면 된다. 그리고 입금과 출금 부분이 메서드로 명확하게 분리되었기 때문에 이후에 변경 사항이 발생하면 관련된 메서드만 수정하면 된다. 특정 메서드로 수정 범위가 한정되기 때문에 더 유지보수 하기 좋다.

이런 리펙토링을 메서드 추출(Extract Method)이라 한다. 메서드를 재사용하는 목적이 아니어도 괜찮다. 메서드를 적절하게 사용해서 분류하면 구조적으로 읽기 쉽고 유지보수 하기 좋은 코드를 만들 수 있다.

그리고 메서드의 이름 덕분에 프로그램을 더 읽기 좋게 만들 수 있다.

문제와 풀이2

문제 - 은행 계좌 입출금

- 다음 실행 예시를 참고해서, 사용자로부터 계속 입력을 받아 입금과 출금을 반복 수행하는 프로그램을 작성하자. 또한 간단한 메뉴를 표시하여 어떤 동작을 수행해야 할지 선택할 수 있게 하자
- 출금시 잔액이 부족하다면 "x원을 출금하려 했으나 잔액이 부족합니다."라고 출력해야 한다.

실행 예시

```
---------------------------------
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료
---------------------------------
선택: 1
입금액을 입력하세요: 10000
10000원을 입금하였습니다. 현재 잔액: 10000원
---------------------------------
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료
---------------------------------
선택: 2
출금액을 입력하세요: 8000
8000원을 출금하였습니다. 현재 잔액: 2000원
---------------------------------
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료
---------------------------------
선택: 2
출금액을 입력하세요: 3000
3000원을 출금하려 했으나 잔액이 부족합니다.
---------------------------------
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료
---------------------------------
선택: 3
현재 잔액: 2000원
---------------------------------
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료
---------------------------------
선택: 4
시스템을 종료합니다.
```

정답

```
package method.ex;
```

```java
import java.util.Scanner;

public class MethodEx4 {
    public static void main(String[] args) {
        int balance = 0;
        Scanner scanner = new Scanner(System.in);

        while (true) {
            System.out.println("---------------------------------");
            System.out.println("1.입금 | 2.출금 | 3.잔액 확인 | 4.종료");
            System.out.println("---------------------------------");
            System.out.print("선택: ");

            int choice = scanner.nextInt();
            int amount;

            switch (choice) {
                case 1:
                    System.out.print("입금액을 입력하세요: ");
                    amount = scanner.nextInt();
                    balance = deposit(balance, amount);
                    break;

                case 2:
                    System.out.print("출금액을 입력하세요: ");
                    amount = scanner.nextInt();
                    balance = withdraw(balance, amount);
                    break;

                case 3:
                    System.out.println("현재 잔액: " + balance + "원");
                    break;

                case 4:
                    System.out.println("시스템을 종료합니다.");
                    return;

                default:
                    System.out.println("올바른 선택이 아닙니다. 다시 선택해주세요.");
            }
        }
    }
```

```
    public static int deposit(int balance, int amount) {
        balance += amount;
        System.out.println(amount + "원을 입금하였습니다. 현재 잔액: " + balance +
"원");
        return balance;
    }

    public static int withdraw(int balance, int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println(amount + "원을 출금하였습니다. 현재 잔액: " + balance +
"원");
        } else {
            System.out.println(amount + "원을 출금하려 했으나 잔액이 부족합니다.");
        }
        return balance;
    }
}
```

정리

변수명 vs 메서드명

변수 이름은 일반적으로 명사를 사용한다. 한편 메서드는 무언가 동작하는데 사용하기 때문에 일반적으로 동사로 시작한다.

이런 차이점 외에는 변수 이름과 메서드 이름에 대한 규칙은 둘다 같다.

- 변수명 예): `customerName`, `totalSum`, `employeeCount`, `isAvailable`
- 메서드명 예): `printReport()`, `calculateSum()`, `addCustomer()`, `getEmployeeCount()`, `setEmployeeName()`

메서드 사용의 장점

- **코드 재사용**: 메서드는 특정 기능을 캡슐화하므로, 필요할 때마다 그 기능을 다시 작성할 필요 없이 해당 메서드를 호출함으로써 코드를 재사용할 수 있다.
- **코드의 가독성**: 이름이 부여된 메서드는 코드가 수행하는 작업을 명확하게 나타내므로, 코드를 읽는 사람에게 추가적인 문맥을 제공한다.

- **모듈성**: 큰 프로그램을 작은, 관리 가능한 부분으로 나눌 수 있다. 이는 코드의 가독성을 향상시키고 디버깅을 쉽게 만든다.
- **코드 유지 관리**: 메서드를 사용하면, 코드의 특정 부분에서 문제가 발생하거나 업데이트가 필요한 경우 해당 메서드만 수정하면 된다. 이렇게 하면 전체 코드 베이스에 영향을 주지 않고 변경 사항을 적용할 수 있다.
- **재사용성과 확장성**: 잘 설계된 메서드는 다른 프로그램이나 프로젝트에서도 재사용할 수 있으며, 새로운 기능을 추가하거나 기존 기능을 확장하는 데 유용하다.
- **추상화**: 메서드를 사용하는 곳에서는 메서드의 구현을 몰라도 된다. 프로그램의 다른 부분에서는 복잡한 내부 작업에 대해 알 필요 없이 메서드를 사용할 수 있다.
- **테스트와 디버깅 용이성**: 개별 메서드는 독립적으로 테스트하고 디버그할 수 있다. 이는 코드의 문제를 신속하게 찾고 수정하는 데 도움이 된다.

따라서, 메서드는 효율적이고 유지 보수가 가능한 코드를 작성하는 데 매우 중요한 도구이다.