

## 메소드에 대한 이해와 메소드의 정의

## main 메소드에 대해서 우리가 아는 것과 모르는 것

메소드의 이름이 `main`이고 중괄호 내 문장들이 **순차적으로 실행**된다는 사실은 안다.

```
public static void main(String[] args) {  
    int num1 = 5;  
    int num2 = 7;  
    System.out.println("5 + 7 = " + (num1 + num2));  
}
```

`public`, `static`, `void` 선언이 의미하는 바는?

메소드 이름이 `main`인 이유는? **자바에서 정한 규칙: 프로그램의 시작은 `main`에서부터!**

`main` 옆에 있는 `(String[] args)`의 의미는?

```
public static void main(String[] args) {  
    System.out.println("프로그램의 시작");  
    hiEveryone(12); // 12를 전달하며 hiEveryone 호출  
    hiEveryone(13); // 13을 전달하며 hiEveryone 호출  
    System.out.println("프로그램의 끝");  
}  
                                     매개변수가 하나인 메소드의 정의  
public static void hiEveryone(int age) {  
    System.out.println("좋은 아침입니다.");  
    System.out.println("제 나이는 " + age + "세 입니다.");  
}
```

## 메소드의 정의와 호출



```
C:\#JavaStudy>java MethodDef  
프로그램의 시작  
좋은 아침입니다.  
제 나이는 12세 입니다.  
좋은 아침입니다.  
제 나이는 13세 입니다.  
프로그램의 끝  
C:\#JavaStudy>
```

```
public static void main(String[ ] args) {  
    System.out.println("프로그램의 시작");  
    hiEveryone(12);  
    hiEveryone(13);  
    System.out.println("프로그램의 끝");  
}  
  
public static void hiEveryone(int age) {  
    System.out.println("좋은 아침입니다.");  
    System.out.println("제 나이는 .... ");  
}
```

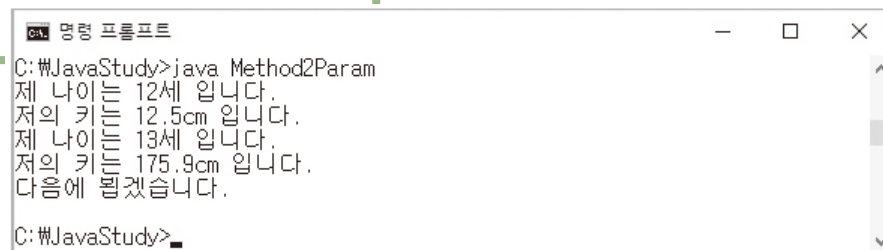
변수 age로 12 전달

①  
②  
③

## 메소드의 호출

```
public static void main(String[] args) {  
    double myHeight = 175.9;  
    hiEveryone(12, 12.5);  
    hiEveryone(13, myHeight);  
    byEveryone();  
}  
    매개변수가 둘인 메소드의 정의  
public static void hiEveryone(int age, double height) {  
    System.out.println("제 나이는 " + age + "세 입니다.");  
    System.out.println("저의 키는 " + height + "cm 입니다.");  
}  
    매개변수가 없는 메소드의 정의  
public static void byEveryone() {  
    System.out.println("다음에 뵙겠습니다.");  
}  
}
```

매개변수 0개, 2개인 메소드



```
명령 프롬프트  
C:\JavaStudy>java Method2Param  
제 나이는 12세 입니다.  
저의 키는 12.5cm 입니다.  
제 나이는 13세 입니다.  
저의 키는 175.9cm 입니다.  
다음에 뵙겠습니다.  
C:\JavaStudy>
```

void: 값을 반환하지 않음을 의미

```
public static void main(String[] args) {  
    int result;  
    result = adder(4, 5);    // adder가 반환하는 값을 result에 저장  
    System.out.println("4 + 5 = " + result);  
    System.out.println("3.5 x 3.5 = " + square(3.5));  
}
```

```
public static int adder(int num1, int num2) {  
    int addResult = num1 + num2;  
    return addResult;    // addResult의 값을 반환  
}
```

return: 값의 반환을 명령

```
public static double square(double num) {  
    return num * num;    // num * num의 결과를 반환  
}
```

## 값을 반환하는 메소드

명령 프롬프트


```
C:\JavaStudy>java MethodReturns  
4 + 5 = 9  
3.5 x 3.5 = 12.25  
  
C:\JavaStudy>
```

```
public static void main(String[] args) {  
    divide(4, 2);  
    divide(6, 2);  
    divide(9, 0);  
}
```

메소드를 호출한 영역으로 값을 반환  
메소드의 종료

```
public static void divide(int num1, int num2) {  
    if(num2 == 0) {  
        System.out.println("0으로 나눌 수 없습니다.");  
        return; // 값의 반환 없이 메소드만 종료  
    }  
    System.out.println("나눗셈 결과: " + (num1 / num2));  
}
```

return의 두 가지 의미



```
명령 프롬프트  
C:\JavaStudy>java OnlyExitReturn  
나눗셈 결과: 2  
나눗셈 결과: 3  
0으로 나눌 수 없습니다.  
C:\JavaStudy>
```

# 변수의 스코프



가시성: 여기서는 저 변수가 보여요.

```
if(...) {  
    int num = 5;  
    .... 지역변수 num  
}
```

```
매개변수 num  
public static void myFunc(int num) {  
    ....  
} 지역변수의 범주에 포함되는 매개변수
```

```
for(int num = 1; num < 5; num++) {  
    ....  
} for문 내에서 유효한  
지역변수 num
```

```
{ 속한 영역을 벗어나면 지역변수 소멸  
    int num2 = 33;  
    num2++;  
    System.out.println(num2);  
}
```

```
public static void main(String[] args) {  
    boolean ste = true;  
    int num1 = 11;
```

영역 1

```
    if(ste) {  
        // int num1 = 22;    // 주석 해제하면 컴파일 오류 발생  
        num1++;  
        System.out.println(num1);  
    }
```

영역 2

```
    {  
        int num2 = 33;  
        num2++;  
        System.out.println(num2);  
    }
```

영역 3

```
    // System.out.println(num2);    // 주석 해제하면 컴파일 오류 발생  
}
```

같은 영역 내에서 동일 이름의 변수 선언 불가

## 지역변수 선언의 예

# 메소드의 재귀 호출

## 수학적 측면에서의 재귀적인 사고

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

$$4! = 4 \times 3 \times 2 \times 1$$

$$3! = 3 \times 2 \times 1$$

$$2! = 2 \times 1$$

$$1! = 1$$



$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1$$



$$n! = n \times (n-1)!$$

이 문장을 코드로 그대로 옮기도록  
돕는 것이 재귀 메소드의 정의

## 재귀의 함수식 정의

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1$$

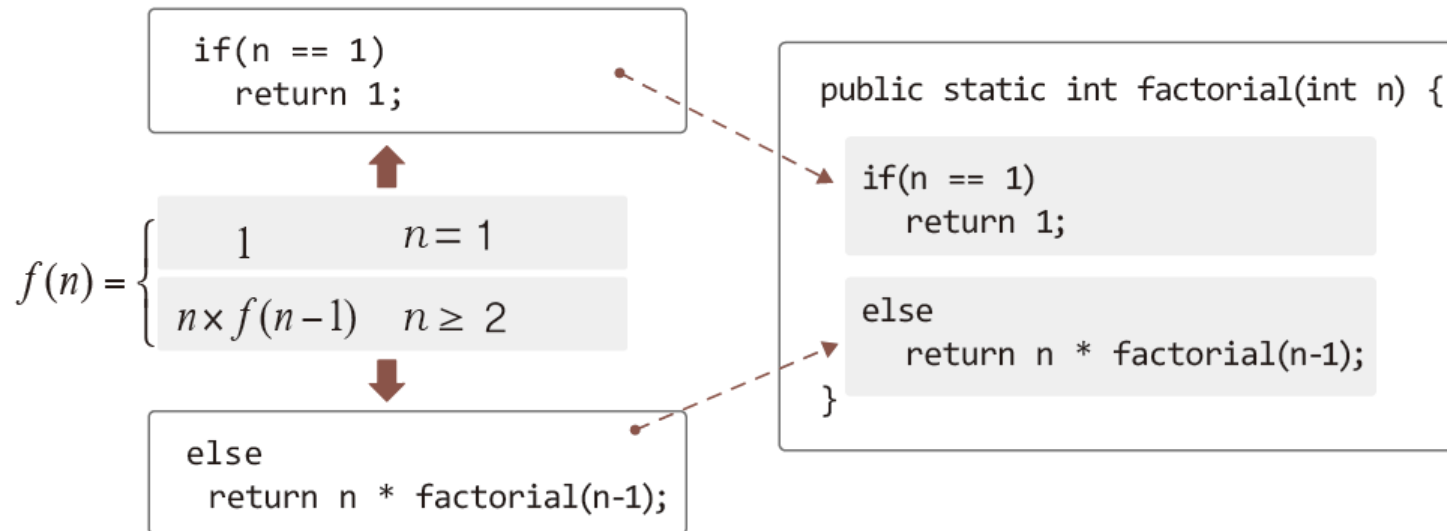


$$f(n) = \begin{cases} n \times f(n-1) & n \geq 2 \\ 1 & n = 1 \end{cases}$$

함수 f의 실행

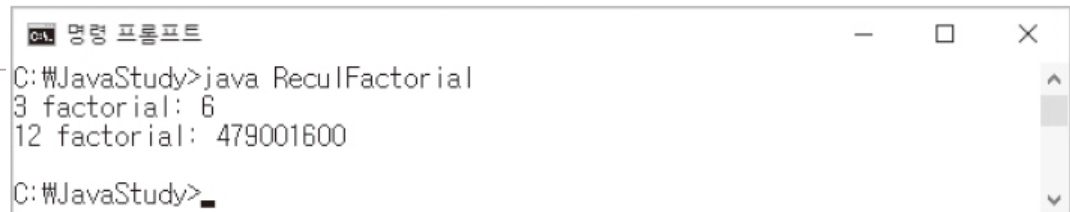
함수 f의 정의

## 함수식 기반의 메소드 정의



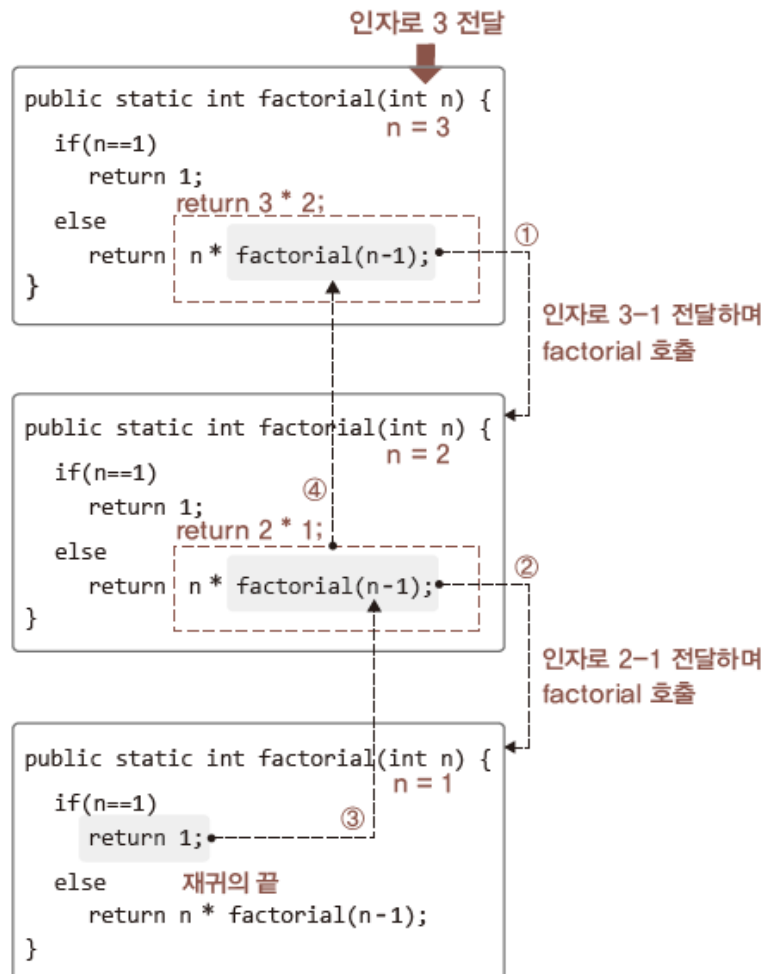
## ◆ ReculFactorial.java

```
1. class ReculFactorial {  
2.     public static void main(String[] args) {  
3.         System.out.println("3 factorial: " + factorial(3));  
4.         System.out.println("12 factorial: " + factorial(12));  
5.     }  
6.  
7.     public static int factorial(int n) {  
8.         if(n == 1)  
9.             return 1;  
10.        else  
11.            return n * factorial(n-1);  
12.    }  
13. }
```



```
C:\WJavaStudy>java ReculFactorial  
3 factorial: 6  
12 factorial: 479001600  
C:\WJavaStudy>
```

팩토리얼 구현의 예





# 클래스의 정의와 인스턴스 생성

# 프로그램의 기본 구성

## 데이터

프로그램상에서 유지하고 관리해야 할 데이터

## 기능

데이터를 처리하고 조작하는 기능

```
class BankAccountPO {  
    static int balance = 0;    // 예금 잔액  
    public static void main(String[] args) {  
        deposit(10000);    // 입금 진행  
        checkMyBalance();    // 잔액 확인  
        withdraw(3000);    // 출금 진행  
        checkMyBalance();    // 잔액 확인  
    }  
    public static int deposit(int amount) {    // 입금  
        balance += amount;  
        return balance;  
    }  
    public static int withdraw(int amount) {    // 출금  
        balance -= amount;  
        return balance;  
    }  
    public static int checkMyBalance() {    // 예금 조회  
        System.out.println("잔액 : " + balance);  
        return balance;  
    }  
}
```

# 클래스 = 데이터 + 기능

## 인스턴스 변수

클래스 내에 선언된 변수

## 인스턴스 메소드

클래스 내에 정의된 메소드

```
class BankAccount {  
    // 인스턴스 변수  
    int balance = 0;  
  
    // 인스턴스 메소드  
    public int deposit(int amount) {...}  
    public int withdraw(int amount) {...}  
    public int checkMyBalance() {...}  
}
```



new BankAccount(); // BankAccount 인스턴스 1

new BankAccount(); // BankAccount 인스턴스 2

## 인스턴스와 참조변수

```
BankAccount myAcnt1;    // 참조변수 myAcnt1 선언
```

```
BankAccount myAcnt2;    // 참조변수 myAcnt2 선언
```

```
myAcnt1 = new BankAccount();    // myAcnt1이 새로 생성되는 인스턴스를 가리킴
```

```
myAcnt2 = new BankAccount();    // myAcnt2가 새로 생성되는 인스턴스를 가리킴
```

```
myAcnt1.deposit(1000);    // myAcnt1이 참조하는 인스턴스의 deposit 호출
```

```
myAcnt2.deposit(2000);    // myAcnt2가 참조하는 인스턴스의 deposit 호출
```

```

class BankAccount {
    int balance = 0;    // 예금 잔액

    public int deposit(int amount) {
        balance += amount;
        return balance;
    }
    public int withdraw(int amount) {
        balance -= amount;
        return balance;
    }
    public int checkMyBalance() {
        System.out.println("잔액 : " + balance);
        return balance;
    }
}

```

## 클래스, 인스턴스 관련 예제

```

class BankAccount00 {
    public static void main(String[] args) {
        // 두 개의 인스턴스 생성
        BankAccount yoon = new BankAccount();
        BankAccount park = new BankAccount();

        // 각 인스턴스를 대상으로 예금을 진행
        yoon.deposit(5000);
        park.deposit(3000);

        // 각 인스턴스를 대상으로 출금을 진행
        yoon.withdraw(2000);
        park.withdraw(2000);

        // 각 인스턴스를 대상으로 잔액을 조회
        yoon.checkMyBalance();
        park.checkMyBalance();
    }
}

```

## 참조변수의 특성

1. `BankAccount yoon = new BankAccount();`
2. ....
3. `yoon = new BankAccount();`      `// yoon이 새 인스턴스를 참조한다.`
4. ....

1. `BankAccount ref1 = new BankAccount();`
2. `BankAccount ref2 = ref1;`      `// 같은 인스턴스 참조`
3. ....

```
class BankAccount {
    int balance = 0;

    public int deposit(int amount) {
        balance += amount;
        return balance;
    }
    public int withdraw(int amount) {
        balance -= amount;
        return balance;
    }
    public int checkMyBalance() {
        System.out.println("잔액 : " + balance);
        return balance;
    }
}
```

```
class DupRef {
    public static void main(String[] args) {
        BankAccount ref1 = new BankAccount();
        BankAccount ref2 = ref1;

        ref1.deposit(3000);
        ref2.deposit(2000);
        ref1.withdraw(400);
        ref2.withdraw(300);
        ref1.checkMyBalance();
        ref2.checkMyBalance();
    }
}
```

## 참조변수 관련 예제

```
class BankAccount { . . . }

class PassingRef {
    public static void main(String[] args) {
        BankAccount ref = new BankAccount();
        ref.deposit(3000);
        ref.withdraw(300);
        check(ref);    // '참조 값'의 전달
    }

    public static void check(BankAccount acc) {
        acc.checkMyBalance();
    }
}
```

## 참조변수의 매개변수 선언



## 참조변수에 null 대입

1. `BankAccount ref = new BankAccount();`
2. ....
3. `ref = null;`    `// ref가 참조하는 인스턴스와의 관계를 끊음`

1. `BankAccount ref = null;`
2. ....
3. `if(ref == null)`    `// ref가 참조하는 인스턴스가 없다면`
4. ....    `null 저장 유무에 대한 비교 연산 가능!`

# 생성자와 String 클래스의 소개

## String 클래스에 대한 첫 소개

---

```
public static void main(String[] args) {  
    String str1 = "Happy";  
    String str2 = "Birthday";  
    System.out.println(str1 + " " + str2);  
  
    printString(str1);  
    printString(str2);  
}
```

코드상에서의 문자열 표현은  
String 인스턴스의 생성으로 이어진다.

```
public static void printString(String str) {  
    System.out.print(str);  
}
```

문자열을 메소드의 인자로 전달할 수 있다.

매개변수로 String형 참조변수를 선언하여 문자열을 인자로 전달받을 수 있다.

## 클래스 정의 모델: 인스턴스 구분에 필요한 정보를 갖게 하자.

---

```
class BankAccount {  
    int balance = 0;    // 예금 잔액  
  
    public int deposit(int amount) {...}  
    public int withdraw(int amount) {...}  
    public int checkMyBalance() {...}  
}
```

문제 있는 클래스 정의

```
class BankAccount {  
    String accNumber;    // 계좌번호  
    String ssNumber;    // 주민번호  
    int balance = 0;    // 예금 잔액  
  
    public int deposit(int amount) {...}  
    public int withdraw(int amount) {...}  
    public int checkMyBalance() {...}  
}
```

좋은 클래스 정의 후보!

## 좋은 클래스 정의 후보를 위한 초기화 메소드!

---

```
class BankAccount {  
    String accNumber;    // 계좌번호  
    String ssNumber;     // 주민번호  
    int balance = 0;     // 예금 잔액
```

초기화를 위한 메소드

```
    public void initAccount(String acc, String ss, int bal) {  
        accNumber = acc;  
        ssNumber = ss;  
        balance = bal; // 계좌 개설 시 예금액으로 초기화  
    }
```

...

```
}  
  
    public static void main(String[] args) {  
        BankAccount yoon = new BankAccount();    // 계좌 생성  
        yoon.initAccount("12-34-89", "990990-9090990", 10000);    // 초기화  
        ...  
    }
```

## 초기화 메소드를 대신하는 생성자

---

```
class BankAccount {  
    String accNumber; // 계좌번호  
    String ssNumber; // 주민번호  
    int balance; // 예금 잔액
```

생성자의 이름은 클래스의 이름과 동일해야 한다.

생성자는 값을 반환하지 않고 반환형도 표시하지 않는다.

```
    public BankAccount(String acc, String ss, int bal) { // 생성자  
        accNumber = acc;  
        ssNumber = ss;  
        balance = bal;  
    }
```

초기화를 위한 생성자

```
    . . .  
}
```

```
    public static void main(String[] args) {  
        BankAccount yoon = new BankAccount("12-34-89", "990990-9090990", 10000);  
        . . .  
    }
```

## 디폴트 생성자

---

```
class BankAccount {  
    int balance;  
    public BankAccount() { // 컴파일러에 의해 자동 삽입되는 '디폴트 생성자'  
        // empty  
    }  
  
    public int deposit(int amount) {...}  
    public int withdraw(int amount) {...}  
    public int checkMyBalance() {...}  
}
```

이렇듯 모든 클래스의 인스턴스 생성은 생성자 호출을 동반한다.

# 메소드 오버로딩



# 메소드 오버로딩

---

호출된 메소드를 찾을 때 참조하게 되는 두 가지 정보

- 메소드의 이름
- 메소드의 매개변수 정보

따라서 이 둘 중 하나의 형태가 다른 메소드를 정의하는 것이 가능하다.

```
class MyHome {  
    void mySimpleRoom(int n) {...}  
    void mySimpleRoom(int n1, int n2) {...}  
    void mySimpleRoom(double d1, double d2) {...}  
}
```

} 메소드 오버로딩

## 메소드 오버로딩의 예

---

```
void simpleMethod(int n) {...}  
void simpleMethod(int n1, int n2) {...}
```

매개변수의 수가 다르므로 성립!

```
void simpleMethod(int n) {...}  
void simpleMethod(double d) {...}
```

매개변수의 형이 다르므로 성립!

```
int simpleMethod() {...}  
double simpleMethod() {...}
```

반환형은 메소드 오버로딩의 조건 아님!

## 오버로딩 관련 피해야할 애매한 상황

---

```
class AAA {  
    void simple(int p1, int p2) {...}  
    void simple(int p1, double p2) {...}  
}
```

다음과 같이 모호한 상황을 연출하지 않는 것이 좋다!

```
AAA inst = new AAA();  
inst.simple(7, 'K');    // 어떤 메소드가 호출될 것인가?
```

## 생성자의 오버로딩

```
class Person {  
    private int regiNum;    // 주민등록 번호  
    private int passNum;    // 여권 번호  
  
    Person(int rnum, int pnum) {  
        regiNum = rnum;  
        passNum = pnum;  
    }  
  
    Person(int rnum) {  
        regiNum = rnum;  
        passNum = 0;  
    }  
  
    void showPersonalInfo() {...}  
}
```

```
public static void main(String[] args) {  
    // 여권 있는 사람의 정보를 담은 인스턴스 생성  
    Person jung = new Person(335577, 112233);  
  
    // 여권 없는 사람의 정보를 담은 인스턴스 생성  
    Person hong = new Person(775544);  
  
    jung.showPersonalInfo();  
    hong.showPersonalInfo();  
}
```


생성자의 오버로딩을 통해 생성되는 인스턴스의 유형을 구분할 수 있다.

ex) 여권이 있는 사람과 없는 사람

ex) 운전 면허증을 보유한 사람과 보유하지 않은 사람

## 키워드 `this`를 이용한 다른 생성자의 호출

```
class Person {  
    private int regiNum;    // 주민등록 번호  
    private int passNum;    // 여권 번호  
  
    Person(int rnum, int pnum) {  
        regiNum = rnum;  
        passNum = pnum;  
    }  
  
    Person(int rnum) {  
        regiNum = rnum;  
        passNum = 0;  
    }  
  
    void showPersonalInfo() {...}  
}
```



```
Person(int rnum) {  
    this(rnum, 0);  
}
```

`rnum`과 `0`을 인자로 받는 오버로딩 된 다른 생성자 호출,  
중복된 코드를 줄이는 효과!

## 키워드 this를 이용한 인스턴스 변수의 접근

---

```
class SimpleBox {  
    private int data;  
  
    SimpleBox(int data) {  
        this.data = data;  
    }  
}
```

} this.data는 어느 위치에서 건 인스턴스 변수 data를 의미함