함께가요 미래로!
Enabling People

# OOP - 1

# 목차

# 객체지향 프로그래밍?

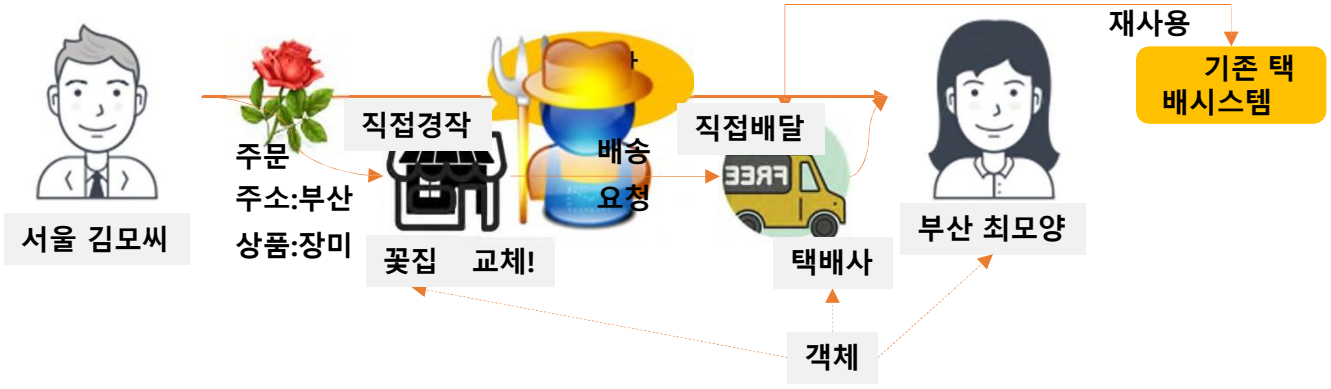
# 객체지향 프로그래밍

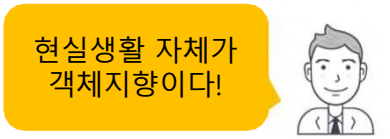
Confidential

## ❖ 객체지향 프로그래밍이란? –Object Oriented Programming

- 객체란?
  - ◆ 客體→ 주체가 아닌 것, 주체가 활용하는 것
    - 우리 주변에 있는 모든 것으로 프로그래밍의 대상 : 사물, 개념, 논리 …
- 객체지향(客體指向) 프로그래밍
  - ◆ 주변의 많은 것들을 객체화 해서 프로그래밍 하는 것
  - ◆ 객체지향은 객체를 많이 만드는 것을 추천한다?

- 서울의 김모씨가 부산의 최모양에게 꽃을 선물하기 위해 필요한 것은?



# 객체지향 프로그래밍



❖ 객체지향 프로그래밍의 장점

- 블록 형태의 모듈화된 프로그래밍
  - ◆ 신뢰성 높은 프로그래밍이 가능하다.
  - ◆ 추가/수정/삭제가 용이하다.
  - ◆ 재 사용성이 높다.
- 객체지향 or not

vs

  - ◆ 날개를 수정하고 싶다면?
  - ◆ 바람을 가르고 하늘을 날고 싶다면?

블록 = 객체

검증된 크기, 색상, 모양
끼우기, 빼기, 움직이기

# Class vs Object

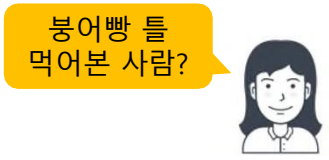


❖ 현실 세계 객체, 클래스, 프로그램의 객체(instance, object)의 관계

- 현실의 객체가 갖는 속성과 기능은 추상화(abstraction) 되어 클래스에 정의된다!
- 클래스는 구체화 되어 프로그램의 객체(instance, object)가 된다.
- 현실의 객체는 우리가 만지고 느낄 수 있는 것 ➔ 실행활에 구체화 되어있는 내용
  - ◆ 이런 객체를 필요할 때마다 매번 처음부터 새로 만들어야 한다면?
  - ◆ 실생활에서는 붕어빵 틀, 설계도(blueprint) 사용
    - 설계도는 제품을 만들기 위해 꼭 필요하지만 이를 사용하지는 않고 설계도로 만든 제품 사용
    - 설계도는 하나의 종류(Type)가 되고 설계도를 통해 나온 제품을 객체라고 부르며 주체가 사용

붕어빵 틀= Type 규정

붕어빵 = 객체

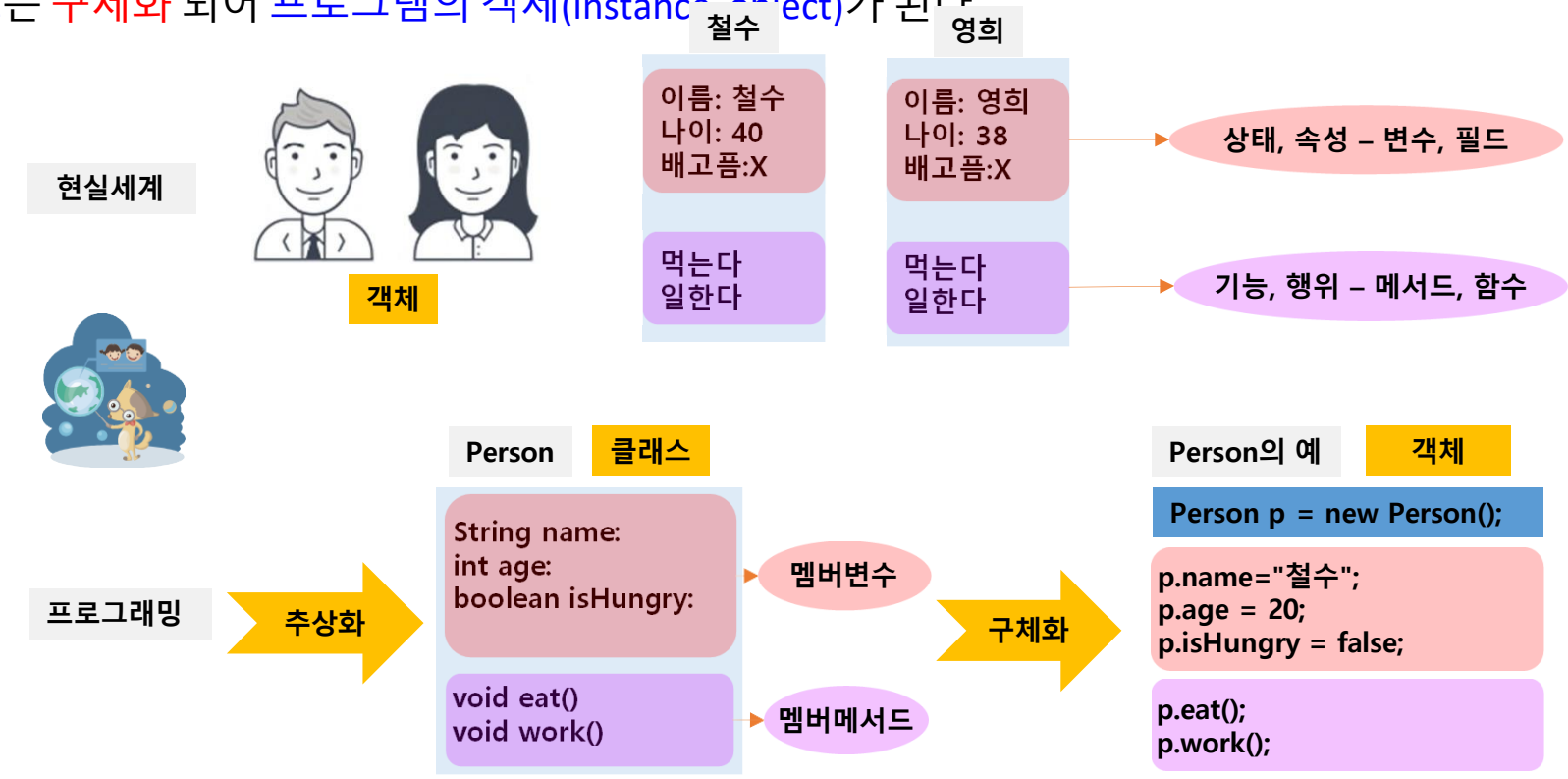


# Class vs Object



❖ 현실 세계 객체, 클래스, 프로그램의 객체(instance, object)의 관계

- 현실의 객체가 갖는 속성과 기능은 추상화(abstraction) 되어 클래스에 정의된다!
- 클래스는 구체화 되어 프로그램의 객체(instance, object)가 된다



# Class vs Object



❖ 현실 세계 객체, 클래스, 프로그램의 객체(instance, object)의 관계

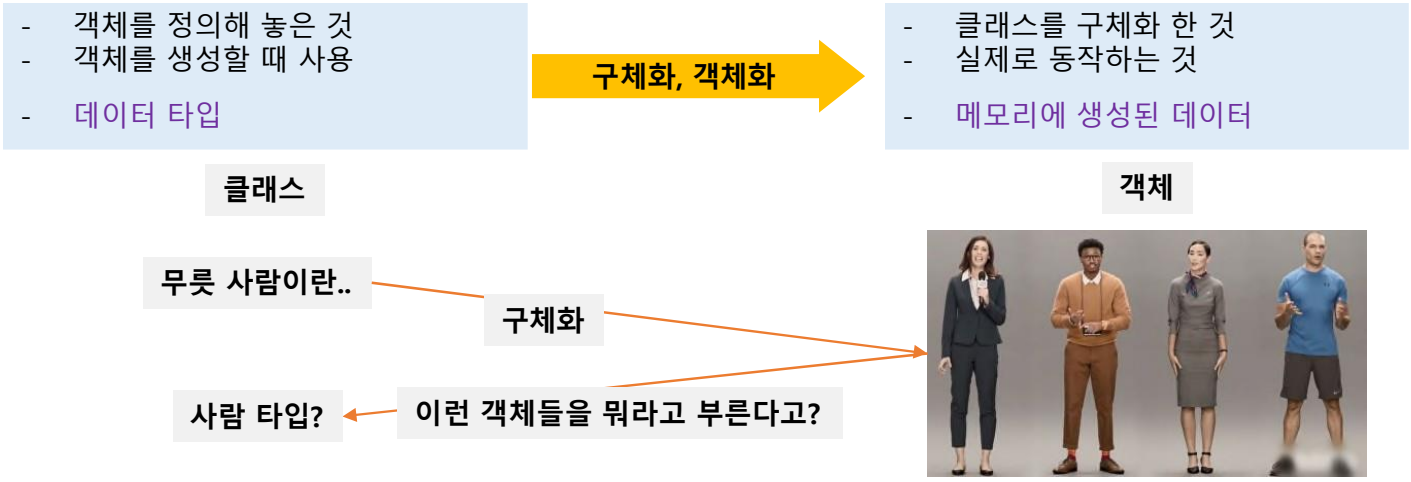
● 프로그램의 클래스와 객체

◆ 클래스

- 객체를 정의해 놓은 것 즉 객체의 설계도, 틀
- 클래스는 직접 사용할 수 없고 직접 사용되는 객체를 만들기 위한 틀을 제공할 뿐

◆ 객체(instance, object)

- 클래스를 데이터 타입으로 메모리에 생성된 것



# Class vs Object



❖ Try Yourself!!

- 추상화로 클래스 만들고 구체화로 객체 사용하기
- 앞서 살펴본 정의를 이용해서 Person 클래스를 작성해보자.
  - Person 타입으로 2개의 객체 생성

◆ 이 객체들은 어떤 관계가 발생할까?

# Class vs Object



❖ Try Yourself!!

- 추상화로 클래스 만들고 구체화로 객체 사용하기

**class = Person 정의**

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

    void eat() {
        System.out.println("냠냠.");
        isHungry = false;
    }
    void work() {
        System.out.println("열심히");
        isHungry = true;
    }
}
```

**Person을 객체화 해서 사용하는 주체**

```
public class PersonTest {
  public static void main(String[] args) {
    Person person1 = new Person();
    person1.name = "홍길동";
    person1.isHungry = true;
    System.out.println(person1.name+" : "+person1.isHungry);
    person1.eat();
    System.out.println(person1.name+" : "+person1.isHungry);

    Person person2 = new Person();
    person2.name = "임꺽정";
    person2.isHungry = true;
    System.out.println(person2.name+" : "+person2.isHungry);
    System.out.println(person1.name+" : "+person1.isHungry);
  }
}
```


- Person 타입으로 2개의 객체 생성
  - ◆ 객체들은 모두 클래스에서 선언된 속성을 가짐
  - ◆ 객체 별로 다른 상태 값을 가짐

# Class vs Object

Confidential

## ❖ 객체 생성과 메모리

- JVM의 메모리 구조

**class area**

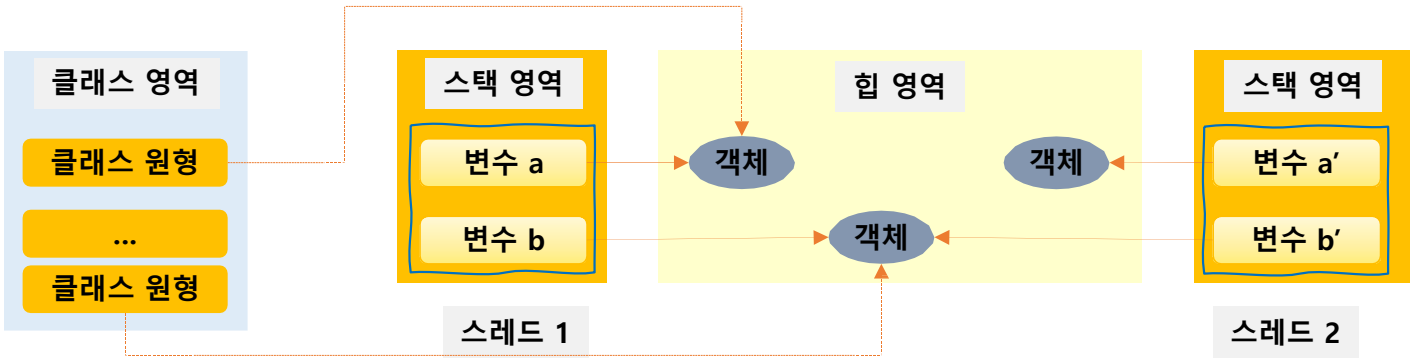
클래스 원형 로딩
- Field 정보
- Method 정보
- 타입 정보
- 상수 풀

**method stack**

메서드들의 실행 공간
- thread 별로 별도 관리
- 메서드 호출 순서대로 쌓이는 구조
- 메서드 프레임에 로컬변수도 쌓이는 구조
- 로컬변수는 선언된 영역을 벗어나면 삭제

**heap**

객체를 저장하기 위한 영역
- thread에 의해 공유
- 생성된 객체는 프로그래머가 삭제할 수 없고 GC만이 제어 가능



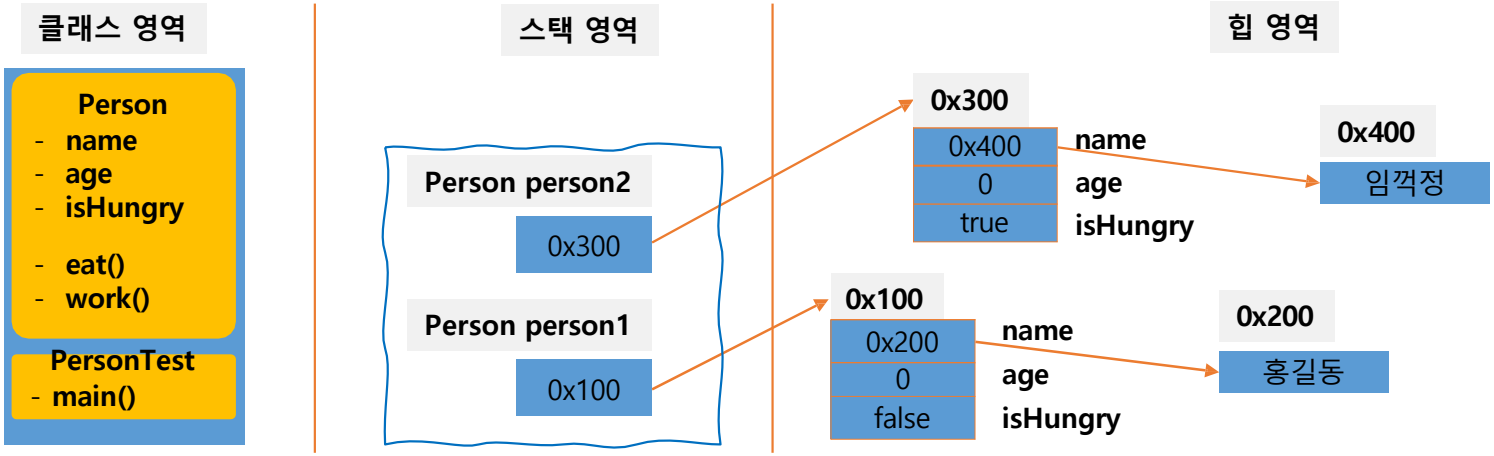
# Class vs Object



## ❖ 객체의 생성과 메모리 할당

```
public static void main(String[] args) {
    Person person1 = new Person();
    person1.name = "홍길동";      person1.isHungry = true;
    System.out.println(person1.name+" : "+person1.isHungry);
    person1.eat();
    System.out.println(person1.name+" : "+person1.isHungry);

    Person person2 = new Person();
    person2.name = "임꺽정";     person2.isHungry = true;
    System.out.println(person2.name+" : "+person2.isHungry);
    System.out.println(person1.name+" : "+person1.isHungry);
}
```



함께가요 미래로!
Enabling People

# 변수

# 위치에 따른 변수의 종



## ❖ 변수의 종류

- 타입에 따른 분류

| 변수 종류 | 특징 | 비고 |
|---|---|---|
| Primitive Type variable | 기본 8가지 type의 변수 | int i, char c, float f... |
| Reference Type variable | 나머지 모든 것(객체 참조) | String s, int [] points , Person p... |

- 선언 위치에 따른 분류

| 종류 | 변수종류 | 선언 위치 |
|---|---|---|
| 멤버 변수 | 클래스 멤버 변수 | 클래스 영역 (static keyword) |
| | 인스턴스 멤버 변수 | 클래스 영역 |
| 지역 변수 | 지역 변수 | 함수 내부 |
| | 파라미터 변수 | 함수 선언부 |

```
public class VariableTypes {

  int instanceVariable; // 인스턴스 멤버 변수
  static int classVariable; // 클래스 멤버 변수

  public static void main(String[] args) {// 파라미터 변수

    int localVariable = 10; // 로컬 변수
    for (int i = 0; i < 100; i++) { // 로컬 변수
      System.out.println(i);
    }
  }
}
```

# 위치에 따른 변수의 종



## ❖ 인스턴스 멤버 변수의 특징

- 선언 위치 : 클래스 {}영역에 선언

```
public class Person {
  static String scientificName = "Homo Sapiens";
  String name;
}
```

- 변수의 생성 : 객체가 만들어질 때 객체 별로 생성됨
  - ◆ 생성 메모리 영역 : heap
- 변수의 초기화 : 타입 별로 default 초기화
- 변수에의 접근 : 객체 생성 후(메모리에 올린 후) 객체 이름(소속)으로 접근
  - ◆ 객체를 만들 때마다 객체 별로 생성 → 객체마다 고유한 상태(변수 값) 유지

```
Person person1 = new Person();
person1.name = "홍길동";

Person person2 = new Person();
person2.name = "임꺽정";
```

- 소멸 시점
  - ◆ Garbage Collector에 의해 객체가 없어질 때, 프로그래머가 명시적으로 소멸시킬 수 없음

# 위치에 따른 변수의 종



❖ 클래스 멤버 변수의 특징

- 선언 위치 : 클래스 {}영역에 선언되며 static 키워드를 붙임

```
public class Person {
  static String scientificName = "Homo Sapiens";
  String name;
}
```

- 변수의 생성 : 클래스 영역에 클래스 로딩 시 메모리 등록
  - ◆ 개별 객체의 생성과 무관
  - ◆ 모든 객체가 공유하게 됨(공유 변수라고도 불림)
- 변수의 초기화 : 타입 별로 default 초기화
- 변수에의 접근 : 객체 생성과 무관하게 클래스 이름(소속)으로 접근
  - ◆ 객체를 생성하고 객체 이름으로 접근도 가능하나 static 에 부합한 표현은 아님

```
Person p = new Person();
p.scientificName = "객체를 통한 변경";
// The static field Person.scientificName should be accessed in a static way
Person.scientificName = "클래스를 통한 변경";
```

- 소멸 시점
  - ◆ 프로그램 종료 시

# 위치에 따른 변수의 종



❖ 지역 변수 & 파라미터 변수

- 선언 위치 : 클래스 영역의 {} 이외의 모든 중괄호 안에 선언되는 변수들
  - ◆ 메서드, 생성자, 초기화 블록

```
                                    // 파라미터 변수
void call(String to){               // 로컬 변수
  String beep = "띠";

                                    // 로컬 변수
for(int i=0; i<3; i++){
  System.out.println(beep);
}


}
```

- 변수의 생성 : 선언된 라인이 실행될 때
  - ◆ 생성 메모리 영역 : thread 별로 생성된 stack 영역
- 변수의 초기화 : 사용하기 전 명시적 초기화 필요
- 변수에의 접근 : 외부에서는 접근이 불가하므로 소속 불필요
  - ◆ 내부에서는 이름에 바로 접근
- 소멸 시점
  - ◆ 선언된 영역인 { }을 벗어날 때

# 위치에 따른 변수의 종



❖ Try Yourself!!

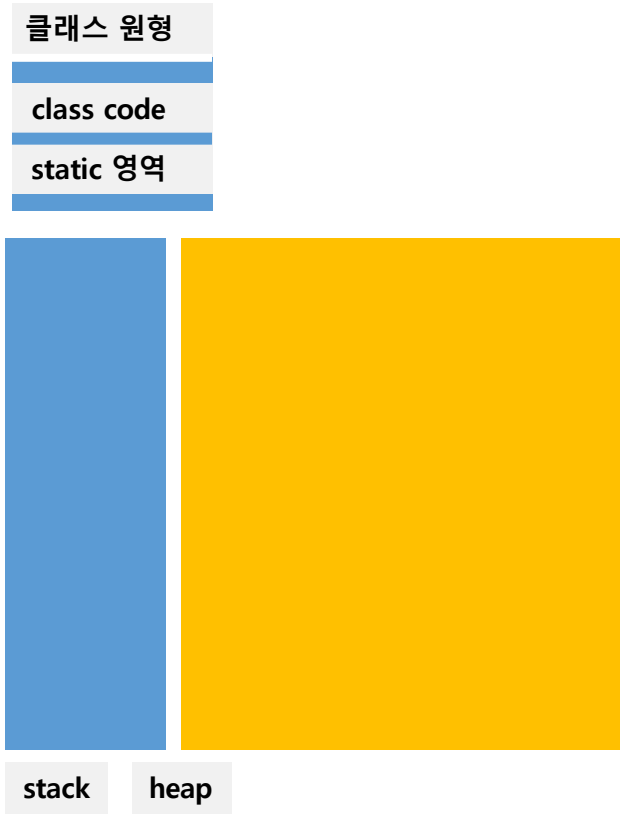
- 다음을 손컴파일링 하고 메모리 동작을 그려보시오.

```
public class SmartPhone {
  static String osName = "iOS";
  String number;
  void call(String to){// 파라미터 변수
    String msg = "띠";// 로컬 변수
    for(int i=0; i<3; i++){// 로컬 변수
      System.out.println(msg);
    }
  }
}

public class SmartPhoneTest {
  public static void main(String[] args) {
    SmartPhone sphone = new SmartPhone();
    sphone.number = "010";
    System.out.println(sphone.osName+" : "+sphone.number);
    sphone.call("011");

    SmartPhone sphone2 = new SmartPhone();
    sphone2.number = "010-111-1111";
    System.out.println(sphone2.osName+" : "+sphone2.number);


    SmartPhone.osName = "android 4.0";
    System.out.println(sphone.osName+" : "+sphone.number);
  }
}
```



함께가요 미래로!
Enabling People

# 메서드

# 메서

Confidential

## ❖ 메서드정의와 필요성

- 메서드란?
  - ◆ 현실의 객체가 하는 동작을 프로그래밍 화
  - ◆ 어떤 작업을 수행하는 명령문의 집합
- 메서드를 작성하는 이유
  - ◆ 반복적으로 사용되는 코드의 중복 방지
    - DRY: Don’t Repeat Yourself!
    - WET: We Enjoy Typing or Write Everything Twice
  - ◆ 코드의 양을 줄일 수 있고 유지 보수가 용이함

# 메서

Confidential

## ❖ 메서드정의와 필요성

```
class Person {
  String name;
  int age=0;
  void printInfo(){
     System.out.println(name+" : "+age);
   }

}
```


● 메서드를 사용하지 않았을 경우

```
Person p = new Person();
System.out.println(p.name+":"+p.age);
Person p2 = new Person();
System.out.println(p2.name+":"+p2.age);
```


● 메서드를 사용했을 경우

```
Person p = new Person();
Person p2 = new Person();
p.printInfo();
p2.printInfo();
```


# 메서

Confidential

## ❖ 메서드정의와 필요성

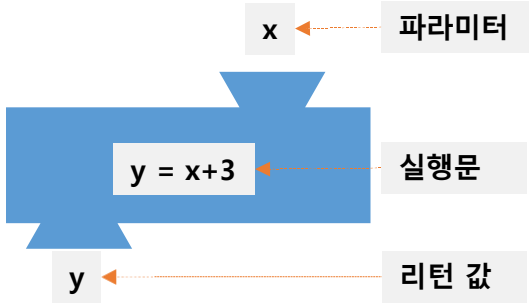
### ● 메서드의 작성 방법

```
제한자  리턴_타입  메서드이름(타입 변수_명, 타입 변수_명...)
{
  // do something
}
```

선언부

구현부

- ◆ 어떤 값을 입력 받아서 → 파라미터(생략 가능)
- ◆ 작업을 진행하고 → 실행문장
- ◆ 결과를 돌려주는 역할 → 리턴 값(생략 가능)



# 메서

Confidential

## ❖ 선언부

- 리턴타입
  - ◆ 호출 결과 호출한 곳으로 반환되는 값의 타입으로 아무것도 리턴하지 않을 경우 void
  - ◆ 결과를 받을 때 묵시적 형 변환 적용

```
public int add(int a, int b) {
  return a + b;
}
int result1 = add(100, 200);
double result2 = add(100, 200);
```

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

//System.out.println(sayHello());
```

  - ◆ 리턴 타입은 하나만 적용 가능

여러 데이터를 넘기려면 어떻게 해야할까?

- 메서드 이름
  - ◆ 메서드가 수행하는 작업을 쉽게 파악하도록 의미 있는 이름 사용
- 파라미터 목록
  - ◆ 메서드 호출 시점에 넘겨줘야 하는 변수들로 넘겨줄 정보가 없을 경우 생략 가능

```
add(10L, 20L);          // (O)
add(100, 200);          // (O)
add(1.1, 2.2);          // (X)
add(100, 200, 300);     // (X)
add(100);               // (X)
```

  - ◆ 파라미터 전달 시 묵시적 형변환 적용

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

# 메서



❖ Variable arguments

- 메서드 선언 시 몇 개의 인자가 들어올 지 예상할 수 없을 경우 (또는 가변적)
  - ◆ 배열 타입을 선언할 수 있으나 → 메서드 호출 전 배열을 생성, 초기화 해야 하는 번거로움
  - ◆ …을 이용해 파라미터를 선언하면 호출 시 넘겨준 값의 개수에 따라 자동으로 배열 생성 및 초기화

```
public static void main(String[] args) {
  VariableTest vt = new VariableTest();
  vt.variableArgs(1,2,3);
  vt.variableArgs(1,2,3,4,5);
  vt.variableArgs(1,2);
}

public void variableArgs(int... params){
  int sum = 0;
  for(int i:params){
    sum+=i;
  }
  System.out.println(sum);
}
```

# 메서



## ❖ 구현부

- 구현부는 중괄호 내에서 처리해야 하는 내용 즉 비즈니스 로직 작성
- 마지막에는 선언된 리턴 타입에 해당하는 값을 return 문장과 함께 반환해야 함
  - ◆ 값 반환 시에는 묵시적 형 변환 적용
  - ◆ 리턴 타입이 void여서 반환할 값이 없을 경우 return 문장 생략 가능

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

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

  - ◆ 메서드 수행 도중 return 문장을 만나거나 마지막 문장을 수행하는 경우 메서드는 종료
    - 조건문을 이용해서 return 할 경우 모든 조건에서 return 필요

```
public double calc(double a, double b, char oper) {
  if (oper == '+') {
    return a + b;
  } else if (oper == '-') {
    return a - b;
  } else {
    return 0;
  }
}
```

```
public double calc2(double a, double b, char oper) {
  double result = 0;
  if (oper == '+') {
    result = a + b;
  } else if (oper == '-') {
    result = a - b;
  }
  return result;
}
```

# 메서

Confidential

## ❖ 메서드 호출

- 메서드를 호출할 때는 메서드의 선언부에 맞춰 호출해야 함
  - ◆ 메서드 이름 : 반드시 동일
  - ◆ 파라미터 : 선언된 파라미터의 개수는 반드시 동일, 타입은 promotion 적용 가능

- 메서드 접근
  - ◆ 멤버 변수와 마찬가지로 static 또는 non static 상태를 구분해서 호출

| 비고 | | static member | non static member(instance member) |
|---|---|---|---|
| 소속 | | 클래스 | 객체 |
| 접근 방법 | 같은 클래스 | 바로 호출 | 바로 호출 |
| | 다른 클래스 | 클래스_이름.멤버_이름 | 객체_이름.멤버_이름 |

  - ◆ 가장 중요한 것은 호출하려는 멤버 메모리에 있는가?
    - 메모리에 있으면 호출 가능
    - 메모리에 없으면 호출 불가 - 먼저 메모리에 로딩 후 사용해야 함

# 메서



❖ class 멤버와 instance 멤버간의 참조와 호출

- 가장 중요한 것은 메모리에 있는가?
  - ◆ 메모리에 있으면 호출 가능
  - ◆ 메모리에 없으면 호출 불가
- static member→ 언제나 메모리에 있음
  - ◆ 클래스 로딩 시 자동 등록
- instance member → 객체 생성 전에는 메모리에 없음
  - ◆ 객체 생성 시 모든 일반 멤버들은 메모리에 생성
  - ◆ 객체 즉 레퍼런스를 통해서 접근

# 메서

Confidential

❖ 클래스 멤버와 일반 멤버간의 참조와 호출

```
  static int cv;              //①
public class First {
  int iv;                     //②

  static void cMethodA() {}//③

  static void cMethodB() {}//④

  void iMethodA() {}          //⑤

  void iMethodB() {}          //⑥

  void iMethodC() {}          //⑦

  void iMethodD(Second s){}//⑧
}



class Second{
  static void cMethod() {} //⑨

  void iMethod() {}           //⑩
}
```

**3에서 1, 4번을 호출하는 코드는?**

**4에서 2번과 5번을 호출하는 코드는?**

**5에서 1번과 3번을 호출하는 코드는?**

**6에서 2번과 5번을 호출하는 코드는?**

**7에서 9번과 10번을 호출하는 코드는?**

**8에서 9번과 10번을 호출하는 코드는?**

ⓐ
```
Second.cMethod();
s.iMethod();
```

ⓑ
```
cv = 100;
cMethodB();
```

ⓒ
```
First f = new First();
f.iv = 10;
f.iMethodA();
```

ⓓ
```
iv = 100;
iMethodA();
```

ⓔ
```
cv = 100;
cMethodA();
```

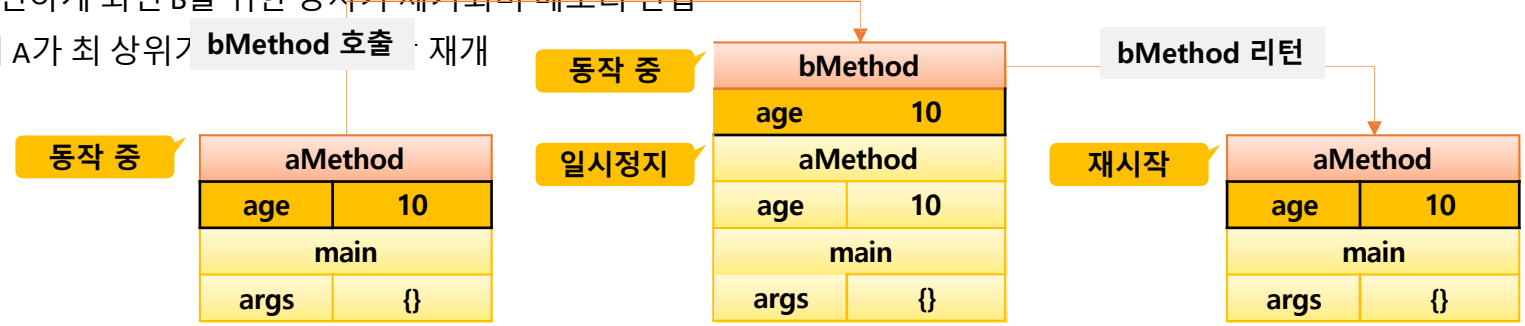
ⓕ
```
Second.cMethod();
Second s = new Second();
s.iMethod();
```

# 메서

Confidential

## ❖ 메서드 호출 스택

- 스택(stack)
  - ◆ First in Last out의 구조
- 메서드 호출 스택
  - ◆ 각각의 메서드 호출 시 마다 메서드 동작을 위한 메모리 상자를 하나씩 할당
    - 상자 내부에 메서드를 위한 파라미터 변수 등 로컬 변수 구성
  - ◆ A 메서드에서 새로운 메서드 B 호출 시 B 실행을 위한 메모리 상자를 쌓음
    - 언제나 맨 위에 있는 메모리 상자(B) 만 활성화
    - 이때 A 메서드는 동작이 끝나지 않고 잠시 정지된 상태
    - B가 리턴하게 되면 B를 위한 상자가 제거되며 메모리 반납
    - 비로서 A가 최 상위기 재개



# 메서

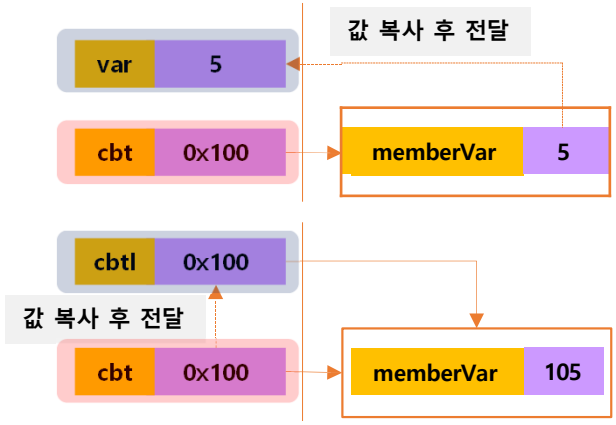
Confidential

## ❖ 기본형 변수와 참조형 변수

- 메서드 호출 시 파라미터로 입력된 값을 복사해서 전달
- Java는 call by value!! Value의 정체는?

```
public class CallByTest {
  int memberVar = 10;
  static void change1(int var) {
    var += 10;
    System.out.printf("change1 : %d%n", var);
  }
  static void change2(CallByTest cbtl) {
    cbtl.memberVar += 100;
    System.out.printf("change2 : %d%n", cbtl.memberVar);
  }

  public static void main(String[] args) {
    CallByTest cbt = new CallByTest();
    cbt.memberVar = 5;
    System.out.printf("change1 호출 전 memberVar: %d%n", cbt.memberVar);
    change1(cbt.memberVar);
    System.out.printf("change1 호출 후 memberVar: %d%n", cbt.memberVar);
    change2(cbt);
    System.out.printf("change2 호출 후 memberVar: %d%n", cbt.memberVar);
  }
}
```



# 메서드 오버로딩



## ❖ 메서드 오버로딩

- overloading: 동일한 기능을 수행하는 메서드의 추가 작성
  - ◆ 일반적으로 메서드 이름은 기능별로 의미 있게 정함
  - ◆ 동일한 기능을 여러 형태로 정의해야 한다면?
  - ◆ eat vs eatUsingChopsticks, eatUsingFork, eatUsingSpoon.?

| eatUsingChopsticks | eatUsingFork | eatUsingSpoon |
|---|---|---|
|  |  |  |
| eat(chopsticks) | eat(fork) | eat(spoon) |

- 입으로 이동하는 부분까지만 다르고 그 이후의 동작은?

# 메서드 오버로딩



## ❖ 메서드 오버로딩

- println 메서드 고찰

```
System.out.println(1);
System.out.println('C');
System.out.println("Hi");
```

**뭐든 출력하면? println**

```
public void println(int x)
public void println(char x)
public void println(String x)
```

**사실은 다른 메서드 들**

```
System.out.printlnInt(1);
System.out.printlnChar('C');
System.out.printlnString("Hi");
```

**매번 이름이 달랐다면 피곤**

  - ◆ 무언가를 출력하는 메서드
    - 출력할 대상은 각각 다르지만 모니터상에 출력하는 방법은?

- 메서드 오버로딩의 장점
  - ◆ 기억해야 할 메서드가 감소하고 중복 코드에 대한 효율적 관리 가능

# 메서드 오버로딩



## ❖ 메서드 오버로딩 방법

- 메서드 이름은 동일
- 파라미터의 개수 또는 순서, 타입이 달라야 할 것
  - ◆ 파라미터가 같으면 중복 선언 오류
- 리턴 타입은 의미 없음

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

**파라미터 이름만 같은 경우**

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

**리턴 타입이 다른 경우**

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

**이름은 같고 파라미터의 타입이 다른 경우**

```
add(3,  5L);
add(5L, 3);
add(3,  4);
```

**실행 결과는?**

# 메서드 오버로딩

Confidential

❖ 메서드 오버로딩의 예

```
void walk(){
  System.out.println("100cm 이동");
}

void walk(int distance){

  System.out.println(distance+"cm 이동");

}

void walk(int distance, String unit){
  switch(unit){
    case "cm":
      break;
    case "inch":
      distance*=2.54;
      break;
    default:
      System.out.println("unknown");
      distance=0;
  }

  System.out.println(distance+"cm 이동");

}
```

하는 일은 같다!

```
void walk() {
  walk(100, "cm");
}

void walk(int distance) {
  walk(distance, "cm");
}

void walk(int distance, String unit) {
  switch (unit) {
    case "cm":
      break;
    case "inch":
      distance *= 2.54;
      break;
    default:
      System.out.println("unknown");
      distance = 0;
  }
  System.out.println(distance + "cm 이동");
}
```

중복 코드의 제거

함께가요 미래로!
Enabling People

# 생성자

# 생성




## ❖ 생성자

- 객체를 생성할 때 호출하는 메서드 비슷한 것
  - ◆ new 키워드와 함께 호출하는 것

```
Person person1 = new Person();
```

  - ◆ 일반 멤버 변수의 초기화나 객체 생성 시 실행돼야 하는 작업 정리

- 작성 규칙
  - ◆ 메서드와 비슷하나 리턴 타입이 없고 이름은 클래스 이름과 동일

```
제한자  클래스_명 (타입 변수_명, 타입 변수_명...)
{
  // 멤버 변수 초기화 작업
}
```

선언부

구현부

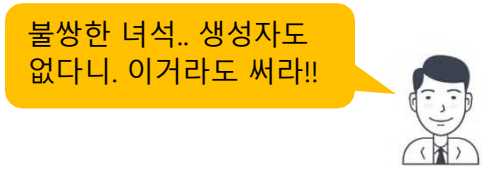
## ❖ 생성자의 종류(1/2)

- 기본 생성자(default constructor)
  - ◆ 그 동안 예제에서는 생성자를 작성하지 않았음
    - 기본 생성자의 형태는 파라미터가 없고 구현부가 비어있는 형태
    - 생성자 코드가 없으면 컴파일러가 기본 생성자 제공

```
public class DefaultPerson {
  String name;
  int age ;
  boolean isHungry;

  //public DefaultPerson() {} -- 생략된 기본 생성자

  public static void main(String[] args) {
    DefaultPerson person = new DefaultPerson();
    person.name = "홍길동";
    person.age = 10;
    person.isHungry = false;
  }
}
```



## ❖ 생성자의 종류(2/2)

- 파라미터가 있는 생성자
  - ◆ 생성자의 목적이 일반 멤버 변수의 초기화→ 생성자 호출 시 값을 넘겨줘서 초기화
  - ◆ 주의! 파라미터가 있는 생성자를 만들면 기본 생성자는 추가되지 않는다.

```
public class ParameterPerson {
  String name;
  int age;
  boolean isHungry;

  // 생성자의 역할 : member 변수의 초기화..
  ParameterPerson(String n, int a, boolean i) {
    name = n;
    age = a;
    isHungry = i;
  }
  public static void main(String[] args) {
    ParameterPerson person = new ParameterPerson("홍길동", 10, true);

    ParameterPerson p2 = new ParameterPerson();
  }
}
```

**오류가 있는 코드는?**

# this의 용법



❖ this.

- 참조 변수로써 객체 자신을 가리킴
  - ◆ 참조변수를 통해 객체의 멤버에 접근했던 것처럼 this를 이용해 자신의 멤버에 접근 가능
- 용도
  - ◆ 로컬 변수와 멤버 변수의 이름이 동일할 경우 멤버 변수임을 명시적으로 나타냄
  - ◆ 명시적으로 멤버임을 나타낼 경우 사용

```
public class Person {
  String name = "아무개";
  int age = 0;
  boolean isHungry = true;;

Person(String name, int age, boolean isHungry){
    this.name = name;
    this.age = age;
    this.isHungry = isHungry;
  }

  void walk(){
    this.isHungry = true;
    System.out.println("뚜벅뚜벅");
  }
}
```

# this의 용법



❖ this.

- this는 객체에 대한 참조
  - ◆ 따라서 static 영역에서 this 사용 불가

```
public static void main(String[] args) {
  ThisPerson person = new ThisPerson("홍길동", 20, true);
  System.out.println(person.getPersonInfo());

  //Cannot use this in a static context
  System.out.println(this.name);
}
```

# this의 용법



❖ this()

- 메서드와 마찬가지로 생성자도 오버로딩 가능
  - ◆ 객체 생성 시 필요한 멤버변수만 초기화 진행 → 생성자 별 코드의 중복 발생
  - ◆ 한 생성자에서 다른 생성자를 호출할 때 사용
- 반드시 첫 줄에서만 호출이 가능

```
public class OverloadConstructorPerson {
  String name = "아무개";
  int age = 0;

  OverloadConstructorPerson(String name, int age) {
    this.name = name;
    this.age = age;
  }

  OverloadConstructorPerson(String name) {
    this.name = name;
  }

  OverloadConstructorPerson() {
    this.name = "홍길동";
    this.age = 100;
  }
}
```

```
public class OverloadConstructorPerson {
  String name = "아무개";
  int age = 0;

  OverloadConstructorPerson(String name, int age) {
    this.name = name;
    this.age = age;
  }

  OverloadConstructorPerson(String name) {
    this(name, 0);
  }

  OverloadConstructorPerson() {
    // 첫 번째 라인에서만 사용 가능
    this("홍길동", 100);
  }
}
```

**중복 코드의 발생과 제거!!**