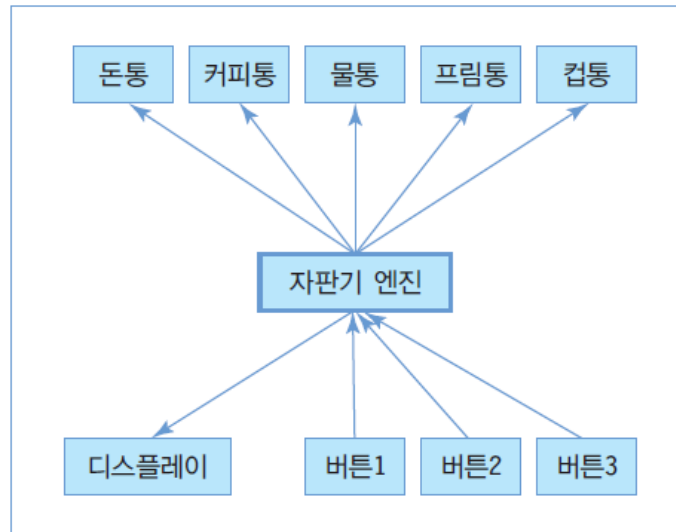


클래스

- <https://github.com/hallymsw>
  - -> [2021\\_major\\_program](#) -> 08\_object

## ■ 객체 지향 프로그래밍

- OOP: Object Oriented Programming
- 부품 객체를 먼저 만들고 이것들을 하나씩 조립해 완성된 프로그램을 만드는 기법

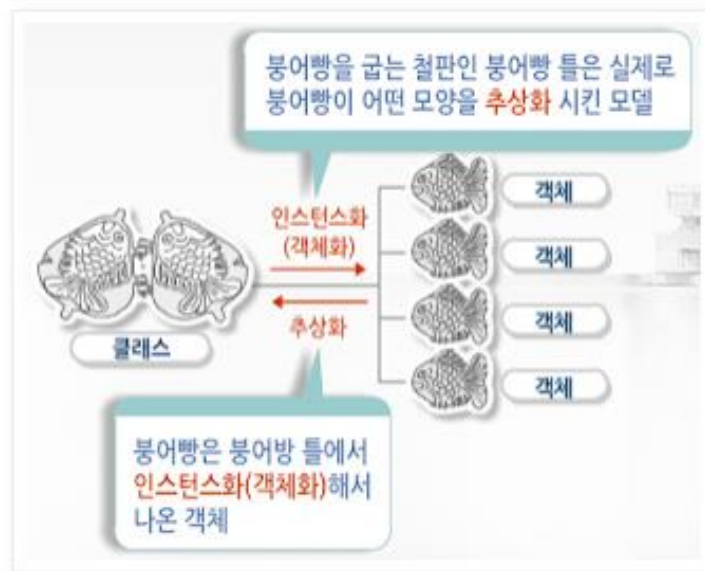


## ■ 클래스

- 객체의 속성(state)과 행위(behavior) 선언
- 객체의 설계도 혹은 틀

## ■ 객체

- 클래스의 틀로 찍어낸 실체
  - 프로그램 실행 중에 생성되는 실체
  - 메모리 공간을 갖는 구체적인 실체
  - 인스턴스(instance)라고도 부름



## ■ 사례

- 클래스: 소나타자동차,      객체: 출고된 실제 소나타 100대
- 클래스: 벽시계,            객체: 우리집 벽에 걸린 벽시계들
- 클래스: 책상,              객체: 우리가 사용중인 실제 책상들

- 클래스의 이름
  - 자바 식별자 작성 규칙에 따라야

번호	작성 규칙	예
1	하나 이상의 문자로 이루어져야 한다.	Car, SportsCar
2	첫 번째 글자는 숫자가 올 수 없다.	Car, 3Car(x)
3	'\$', '_', ' ' 외의 특수 문자는 사용할 수 없다.	\$Car, _Car, @Car(x), #Car(x)
4	자바 키워드는 사용할 수 없다.	int(x), for(x)

- 한글 이름도 가능하나, 영어 이름으로 작성
- 알파벳 대소문자는 서로 다른 문자로 인식
- 첫 글자와 연결된 다른 단어의 첫 글자는 대문자로 작성하는 것이 관례

Calculator, Car, Member, ChatClient, ChatServer, Web\_Browser

## ■ 형식

```
public class 클래스명 {  
  
}
```



```
public class Student {  
  
}
```

- 소스 파일당 하나의 클래스를 선언하는 것이 관례
  - 두 개 이상의 클래스도 선언 가능
  - 소스 파일 이름과 동일한 클래스만 public으로 선언 가능

1. 클래스 선언방법
2. 객체 생성 방법 : new 이용

```
public class Student {  
  
}
```

```
public class StudentExample {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        System.out.println("s1 변수가 Student 객체를 참조합니다.");  
  
        Student s2 = new Student();  
        System.out.println("s2 변수가 또 다른 Student 객체를 참조합니다.");  
    }  
}
```

- 클래스의 구성 멤버
  - 필드(Field)
  - 생성자(Constructor)
  - 메소드(Method)

- 필드(Field) ————— 객체의 데이터가 저장되는 곳
  - `int fieldName;`
- 생성자(Constructor) ————— 객체 생성시 초기화 역할 담당
  - `ClassName() { ... }`
- 메소드(Method) ————— 객체의 동작에 해당하는 실행 블록
  - `void methodName() { ... }`

접근지정자

클래스이름

```
public class ClassName {
```

```
//필드
```

```
int fieldName;
```

```
//생성자
```

```
ClassName() { ... }
```

```
//메소드
```

```
void methodName() { ... }
```

```
}
```



- 클래스내에 선언된 변수
- 필드 선언

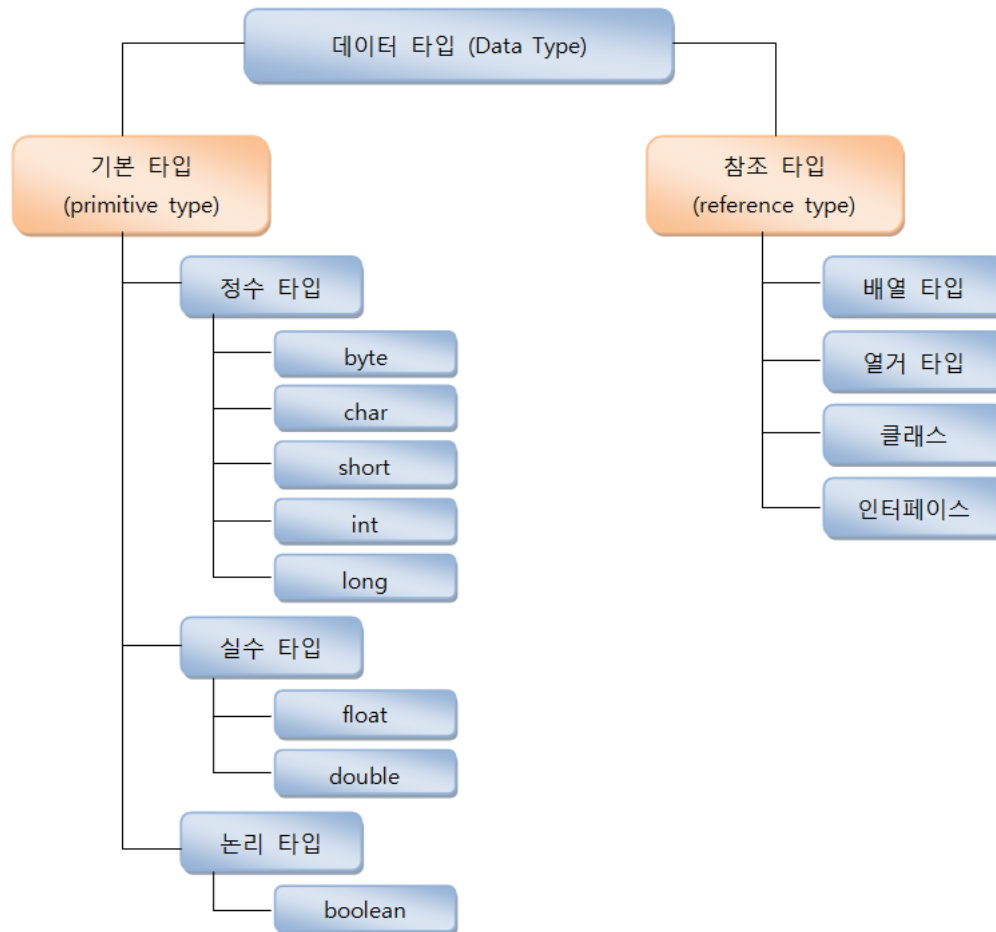
타입 필드 [= 초기값] ;

```
String company = "현대자동차";  
String model = "그랜저";  
int maxSpeed = 300;  
int productionYear;  
int currentSpeed;  
boolean engineStart;
```

- 필드의 기본 초기값
  - 초기값 지정되지 않은 필드
    - 객체 생성시 자동으로 기본값으로 초기화

분류		데이터 타입	초기값
기본 타입	정수 타입	byte	0
		char	₩u0000 (빈 공백)
		short	0
		int	0
		long	0L
	실수 타입	float	0.0F
		double	0.0
	논리 타입	boolean	false
참조 타입		배열	null
		클래스(String 포함)	null
		인터페이스	null

## ■ 데이터 타입 분류



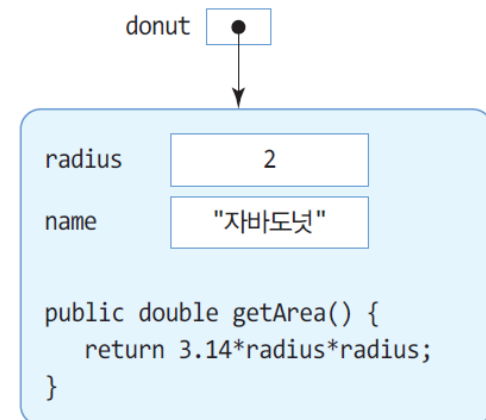
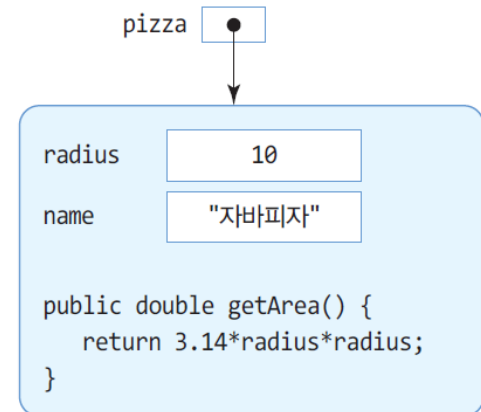
- 필드 사용

- 필드 값을 읽고, 변경하는 작업을 말한다.
- 필드 사용 위치
  - 선언된 클래스 내부: “**필드이름**” 으로 바로 접근
  - 선언된 클래스 외부: “**객체.필드이름**” 으로 접근

# 필드(field)

```
public class Circle {  
    int radius;           // 원의 반지름을 저장하는 멤버 변수  
    String name;          // 원의 이름을 저장하는 멤버 변수  
  
    public double getArea() { // 멤버 메소드  
        double area = 3.14 * radius * radius;  
        return area;  
    }  
}
```

```
public class CircleExample {  
    public static void main(String[] args) {  
        Circle pizza;  
        pizza = new Circle();           // Circle 객체 생성  
        pizza.radius = 10;              // 피자 반지름을 10으로 설정  
        pizza.name = "자바피자";        // 피자 이름 설정  
        double area = pizza.getArea();   // 피자 면적 알아보기  
        System.out.println(pizza.name + "의 면적은 " + area);  
  
        Circle donut = new Circle();     // Circle 객체 생성  
        donut.radius = 2;                // 도넛 반지름을 2로 설정  
        donut.name = "자바도넛";         // 도넛 이름 설정  
        area = donut.getArea();          // 도넛 면적 알아보기  
        System.out.println(donut.name + "의 면적은 " + area);  
    }  
}
```



# 객체 생성과 활용

## 1. 레퍼런스 변수 선언

```
Circle pizza;
```

## 2. 객체 생성

- new 연산자 이용

```
pizza = new Circle();
```

## 3. 객체 멤버 접근

- 점(.) 연산자 이용

```
pizza.radius = 10;
```

```
area = pizza.getArea();
```

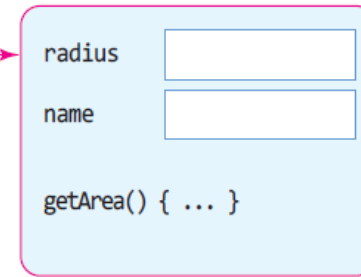
(1) Circle pizza;

pizza

(2) pizza = new Circle();

pizza

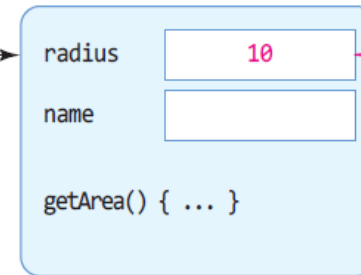
Circle 타입의 객체



객체 메모리  
할당 및  
객체 생성

(3) pizza.radius = 10;

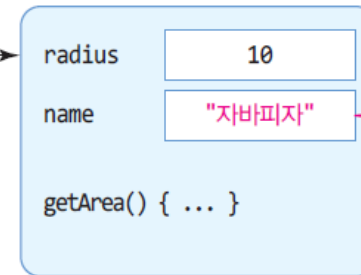
pizza



radius 값 변경

(4) pizza.name = "자바피자";

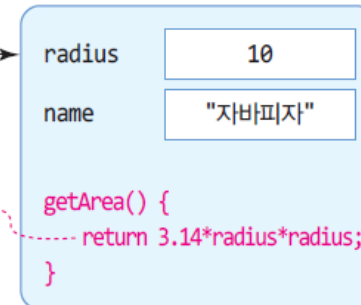
pizza



name 값 변경

(5) double area = pizza.getArea();

pizza



area 314.0

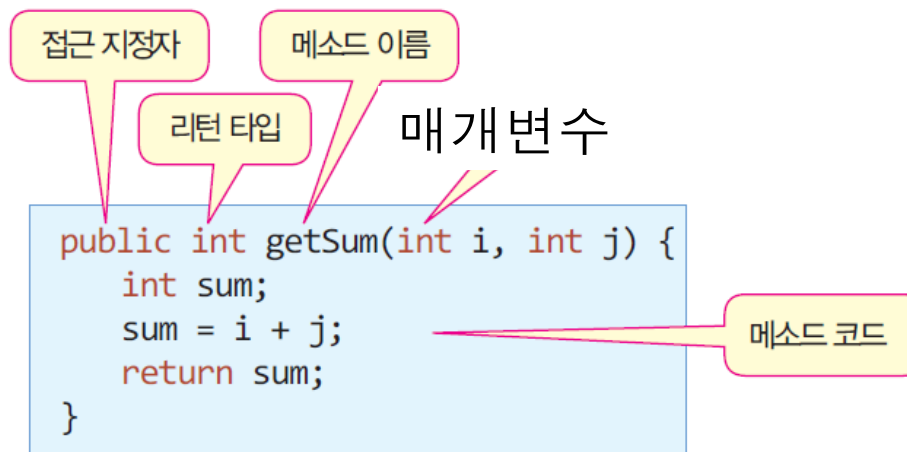
getArea()  
메소드 실행

# 7절. 메소드(method)

## ■ 메소드란?

- 객체의 동작(기능)
- 호출해서 실행할 수 있는 중괄호 { } 블록
- 메소드 호출하면 중괄호 { } 블록에 있는 모든 코드들이 일괄 실행

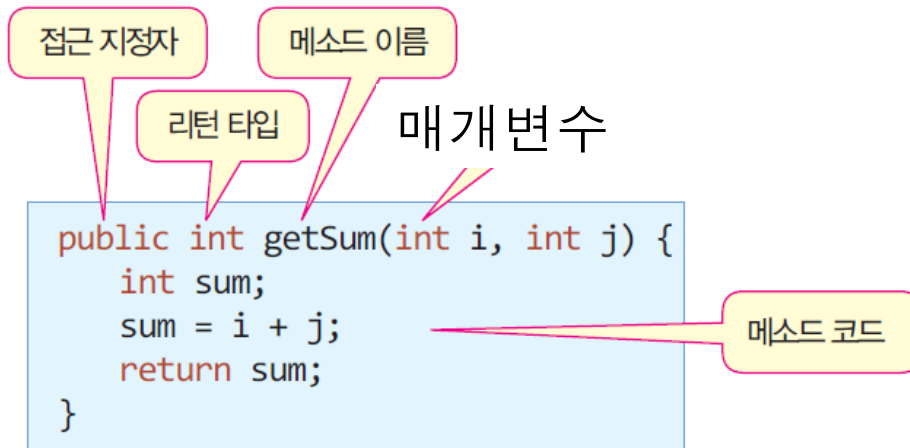
## ■ 메소드 선언



## ■ 접근 지정자

- 다른 클래스에서 메소드를 접근할 수 있는지 여부 선언
- `public`, `private`, `protected`, 디폴트(접근 지정자 생략)

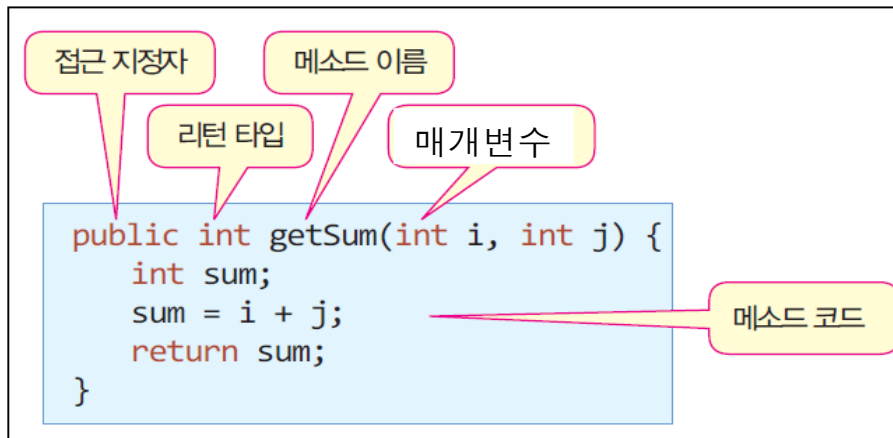
# 7절. 메소드(method)





# 7절. 메소드(method)

- 메소드 리턴 타입
  - 메소드 실행된 후 리턴하는 값의 타입
  - 메소드는 리턴값이 있을 수도 있고 없을 수도 있음



## [메소드 선언]

```
void powerOn() { ... }  
double divide(int x, int y) { ... }
```

## [메소드 호출]

```
powerOn();  
double result = divide( 10, 20 );
```

- 메소드 이름
  - 자바 식별자 규칙에 맞게 작성

- 메소드 매개변수 선언
  - 매개변수는 메소드를 실행할 때 필요한 데이터를 외부에서 받기 위해 사용
  - 매개변수도 필요 없을 수 있음

[메소드 선언]

```
void powerOn() { ... }  
double divide(int x, int y) { ... }
```

[메소드 호출]

```
powerOn();  
double result = divide( 10, 20 );
```

```
byte b1 = 10;  
byte b2 = 20;  
double result = divide(b1, b2);
```

- 리턴(return) 문
  - 메소드 실행을 중지하고 리턴값 지정하는 역할
  - 리턴값이 있는 메소드
    - 반드시 리턴(return)문 사용해 리턴값 지정해야

```
int plus(int x, int y) {  
    int result = x + y;  
    return result;  
}
```

- return 문 뒤에 실행문 올 수 없음
- 리턴값이 없는 메소드
  - 메소드 실행을 강제 종료 시키는 역할

```
boolean isLeftGas() {  
    if(gas==0) {  
        System.out.println("gas 가 없습니다.");  
        return false;  
    }  
    System.out.println("gas 가 있습니다.");  
    return true;  
}
```

# 메소드(method)

리턴타입이 있다면 세개의  
타입이 모두 일치 해야 함

```
class Calculator {  
    int radius;  
    String name;  
    int getSum(int x, int y)  
    {  
        int sum = x+y;  
        printSum(sum);  
        return sum;  
    }  
    void printSum(int s)  
    {  
        System.out.println(s);  
    }  
}  
public class MainClass  
{  
    public static void main(String[] args) {  
        int ret=0, x=1, y=2;  
        Calculator cal = new Calculator ();  
        ret=cal.getSum(x,y);  
    }  
}
```

매개변수가 있다면 호출시  
매개변수의 개수 타입이  
모두 일치 해야 함

## ■ 메소드 호출 방법

### ■ 같은 클래스 내에서 호출

- 리턴값이 있는 경우
- 형식 : 변수 = 메소드명(매개변수)
- 리턴값이 없는 경우
- 형식 : 메소드명(매개변수);

```
class Calculator {  
    int radius;  
    String name;  
    int getSum(int x, int y)  
    {  
        int sum = x+y;  
        printSum(sum);  
        return sum;  
    }  
    void printSum(int s)  
    {  
        System.out.println(s);  
    }  
}  
public class MainClass  
{  
    public static void main(String[] args) {  
        int x=1, y=2, ret=0;  
        Calculator cal = new Calculator ();  
        ret=cal.getSum(x,y);  
    }  
}
```

### ■ 다른 클래스 에서 호출

- 1. 객체 생성
- 2. 메소드 호출
- 리턴값이 있는 경우
- 형식 : 변수 = 객체.메소드명(매개변수);
- 리턴값이 없는 경우
- 형식 : 객체.메소드명(매개변수);

## ■ JVM이 사용하는 메모리 영역 메소드 영역

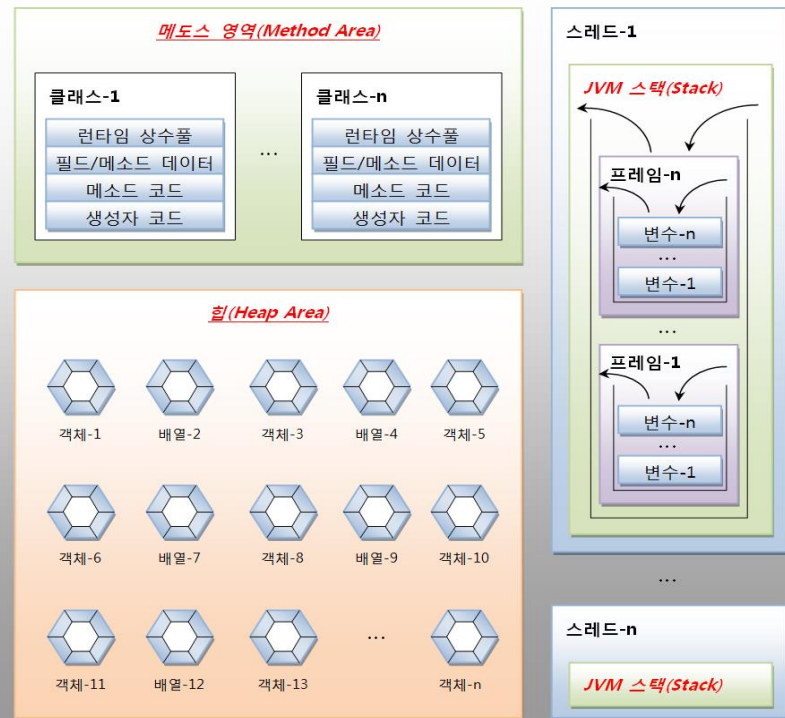
### ■ 힙 영역

- **JVM** 시작할 때 생성
- 참조타입의 데이터 저장(객체/배열)
- 사용되지 않는 객체는 **Garbage Collector** 가 자동 제거

### ■ JVM 스택

- 지역 변수 저장
- 스레드 별 생성

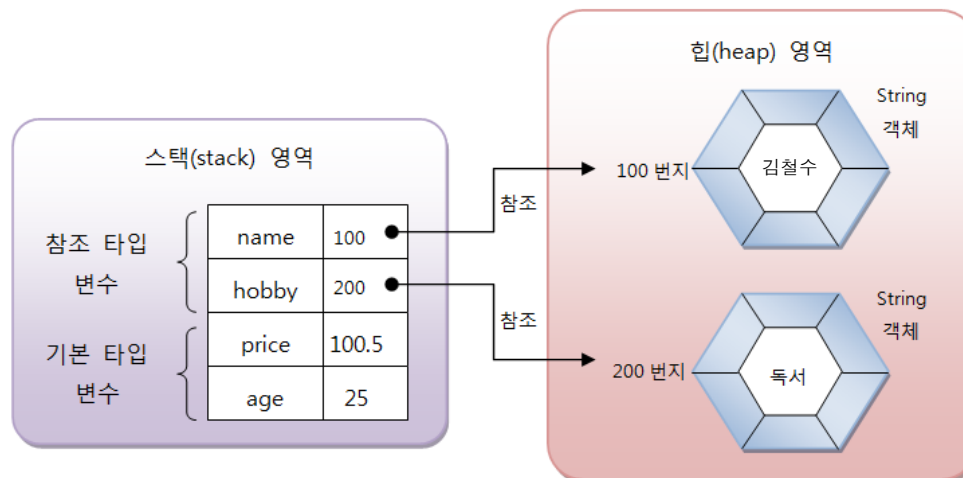
Runtime Data Area



- 변수의 메모리 사용
  - 기본 타입 변수 – 실제 값을 변수 안에 저장
  - 참조 타입 변수 – 주소를 통해 객체 참조

[기본 타입 변수]  
int age = 25;  
double price = 100.5;

[참조 타입 변수]  
String name = "김철수";  
String hobby = "독서";



# 메모리 사용 영역

```
class Calculator {  
    int radius;  
    double area  
    int getSum(int px, int py)  
    {  
        int sum = px+py;  
        printSum(sum);  
        return sum;  
    }  
    void printSum(int s)  
    {  
        System.out.println(s);  
    }  
}  
  
public class MainClass  
{  
    public static void main(String[] args) {  
        int x=1, y=2, ret=0;  
        Calculator cal = new Calculator ();  
        ret=cal.getSum(x,y);  
        cal.radius=3;  
        cal.area=cal.radius*radius;  
        Calculator pc_cal = cal;  
    }  
}
```

heap

radius

area

stack

s

px

py

sum

x

ret

cal

pc\_cal

y



# 참조 변수의 ==, != 연산

- 변수의 값이 같은지 다른지 비교
  - 기본 타입: **byte, char, short, int, long, float, double, boolean**
    - 의미 : 변수의 값이 같은지 다른지 조사
  - 참조 타입: 배열, 열거, 클래스, 인터페이스
    - 의미 : 동일한 객체를 참조하는지 다른 객체를 참조하는지 조사  
→ **==, !=** 이용

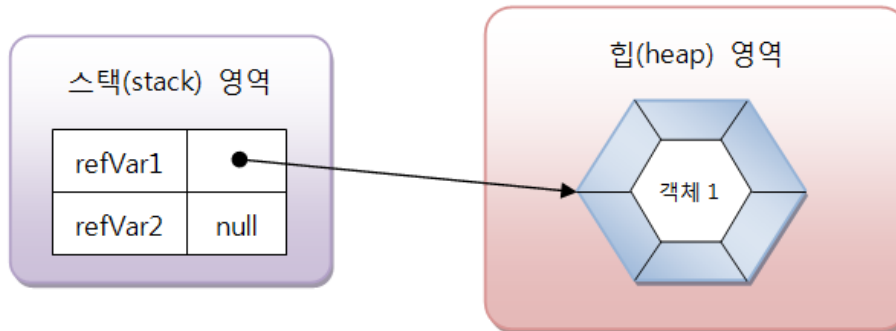


```
if( refVar2 == refVar3 ) { ... }
```

# null과 NullPointerException

## ■ null(널)

- 변수가 참조하는 객체가 없을 경우 초기값으로 사용 가능
- 참조 타입의 변수에만 저장가능
- **null**로 초기화된 참조 변수는 스택 영역 생성



그림에서 refVar1 은 힙 영역의 객체를 참조하므로 연산의 결과는 다음과 같다.

refVar1 == null	결과: false
refVar1 != null	결과: true

refVar2 는 null 값을 가지므로 연산의 결과는 다음과 같다.

refVar2 == null	결과: true
refVar2 != null	결과: false

## ■ NullPointerException의 의미

### ■ 예외(Exception)

- 사용자의 잘못된 조작 이나 잘못된 코딩으로 인해 발생하는 프로그램 오류

### ■ NullPointerException

- 참조 변수가 **null** 값을 가지고 있을 때
  - 객체의 필드나 메소드를 사용하려고 했을 때 발생

```
int[] intArray = null;  
intArray[0] = 10;           //NullPointerException
```

```
String str = null;  
System.out.println("총 문자수: " + str.length()); //NullPointerException
```

# 생성자(Constructor)

## ■ 생성자

- new 연산자에 의해 호출되어 객체의 초기화 담당(호출시기)

```
Car c = new Car();
```

- 생성자 이름은 클래스 이름과 동일
- 생성자는 여러 개 작성 가능(생성자 중복)
- 목적 : 객체 생성 시 필드의 초기화 및 초기화 관련 메소드 호출

## ■ 기본 생성자(default constructor)

- 매개 변수 없고, 아무 작업 없이 단순 리턴하는 생성자
- 디폴트 생성자라고도 불림
- 생성자 선언을 생략하면 컴파일러는 다음과 같은 기본 생성자 추가

소스 파일(Car.java)

```
public class Car {  
  
}
```

→

바이트 코드 파일(Car.class)

```
public class Car {  
    public Car() { } //자동 추가  
}  
    기본 생성자
```

```
Car myCar = new Car();
```

기본 생성자

# 생성자(Constructor)

```
public class Car {  
    //생성자  
    Car() {  
    }  
}
```

```
public class CarExample {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
    }  
}
```

# 생성자(Constructor)

- 생성자 선언 : 메소드와의 차이점 -> 리턴값이 없음
  - 디폴트 생성자 대신 개발자가 직접 선언
  - 개발자 선언한 생성자 존재 시 컴파일러는 기본 생성자 추가하지 않음
  - 생성자 선언 형식

```
클래스명(매개변수들){  
}
```

- 메소드 선언 형식

```
리턴타입 메소드명(매개변수들){  
}
```

```
public class Car {  
    //생성자  
    Car() {  
    }  
}
```



```
public class CarExample {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        //Car myCar = new Car(); (x)  
    }  
}
```

```
public class Car {  
    String color = "red";  
    int cc = 1100;  
  
    public Car()  
    {  
    }  
  
    public Car(String col, int c)  
    {  
        color = col;  
        cc=c;  
    }  
}
```



```
public class CarExample {  
    public static void main(String[] args)  
    {  
        Car cno = new Car();  
        System.out.println(cno.color);  
        System.out.println(cno.cc);  
        Car c = new Car("black", 2000);  
        System.out.println(c.color);  
        System.out.println(c.cc);  
    }  
}
```

## ■ this

- 객체 내부에서 인스턴스 멤버임을 명확히 하기 위해 this. 사용
- 필드
  - 객체내부에서 인스턴스 필드에 접근시 사용(객체 생성 없이 this를 이용 바로 사용 가능)

```
public class Car {  
    String color="red";  
    int cc=1100;  
    public Car(String color, int cc) {  
        this.color=color;  
        this.cc=cc;  
    }  
}
```

## ■ 메소드

- 객체내부에서 인스턴스 메소드에 접근시 사용(객체 생성 없이 this를 이용 바로 사용 가능)



```
public class Car {  
    String color="red";  
    int cc=1100;  
    public Car() {  
  
    }  
    public Car(String color, int cc) {  
        this.color=color;  
        this.cc=cc;  
    }  
    public String getColor() {  
        return this.color;  
    }  
}
```

```
public class CarExample {  
    public static void main(String[] args)  
    {  
  
        Car c = new Car("black",2000);  
        System.out.println(c.color);  
        System.out.println(c.cc);  
    }  
}
```



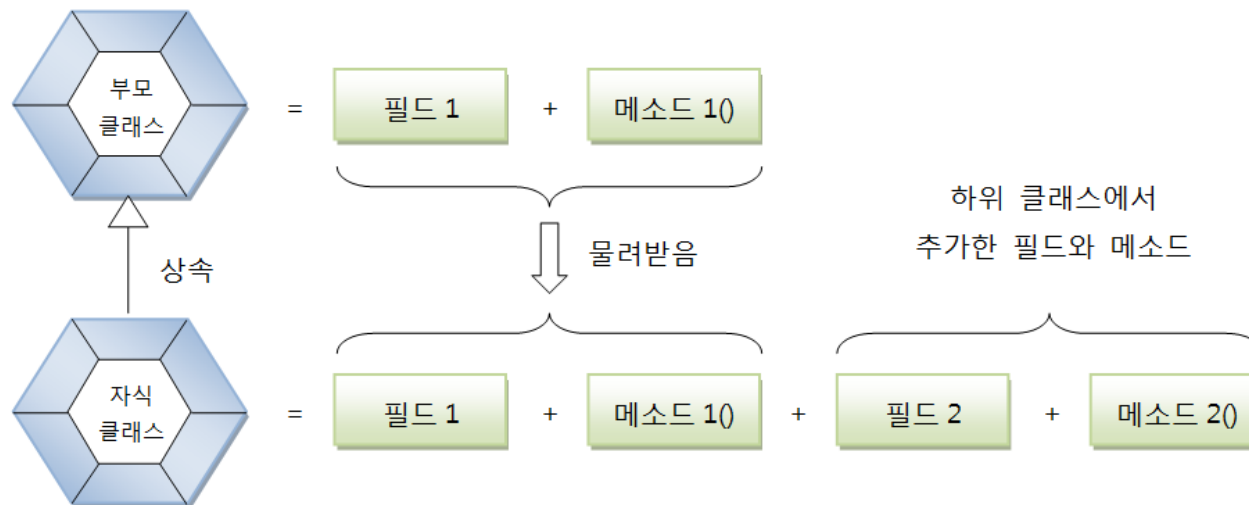
한림대학교 SW중심대학

상속

## ■ 상속(Inheritance)이란?

### ■ 객체 지향 프로그램:

- 자식(하위, 파생) 클래스가 부모(상위) 클래스의 멤버를 물려받는 것
- 자식이 부모를 선택해 물려받음
- 상속 대상: 부모의 필드와 메소드



- **상속(Inheritance) 개념의 활용**
  - **상속의 효과**
    - 부모 클래스 재사용해 자식 클래스 빨리 개발 가능
    - 반복된 코드 중복 줄임
    - 유지 보수 편리성 제공
  - **상속 대상 제한**
    - 부모 클래스의 **private** 접근 갖는 필드와 메소드 제외
    - 부모 클래스가 다른 패키지에 있을 경우, **default** 접근 갖는 필드와 메소드도 제외

멤버에 접근하는 클래스	멤버의 접근 지정자			
	private	디폴트 접근 지정	protected	public
같은 패키지의 클래스	×	○	○	○
다른 패키지의 클래스	×	×	×	○
접근 가능 영역	클래스 내	동일 패키지 내	동일 패키지와 자식 클래스	모든 클래스

## ■ 상속 선언

- extends 키워드로 선언
  - 부모 클래스를 물려받아 확장한다는 의미
- 부모 클래스 -> 슈퍼 클래스(super class)
- 자식 클래스 -> 서브 클래스(sub class)

```
class BaiscCalculator {  
    public String calName = "pCal";  
    ...  
}  
  
// BaiscCalculator 를 상속받는 EnginneringCalculator 클래스 선언  
class EnginneringCalculator extends BaiscCalculator {  
    ...  
}
```

- 자바는 단일 상속 - 부모 클래스 나열 불가

```
public class BaiscCalculator
{
    public String calName = "pCal";
    public int add(int x, int y)
    {
        System.out.println("add()");
        int resultAdd=x+y;
        return resultAdd;
    }
    public int sub(int x, int y)
    {
        System.out.println("sub");
        int resultAdd=x-y;
        return resultAdd;
    }
    public int mul(int x, int y)
    {
        System.out.println("mul()");
        int resultAdd=x*y;
        return resultAdd;
    }
    public double div(int x, int y)
    {
        double ret=0;
        ret=x/y;
        return ret;
    }
}
```

```
public class EnginneringCalculator extends
BaiscCalculator
{
}
```

```
public class MainApp
{
    public static void main(String[] args)
    {
        EnginneringCalculator ch = new
EnginneringCalculator();
        int result = ch.add(1, 2);
        String name = ch.calName;
        System.out.println(result);
        System.out.println(name);

    }
}
```

```
class Point {
    int x;
    int y;
    void set(int a, int b) {
        x = a;
        y = b;
    }
    void showPoint() {
        System.out.println(x + ", " + y);
    }
}

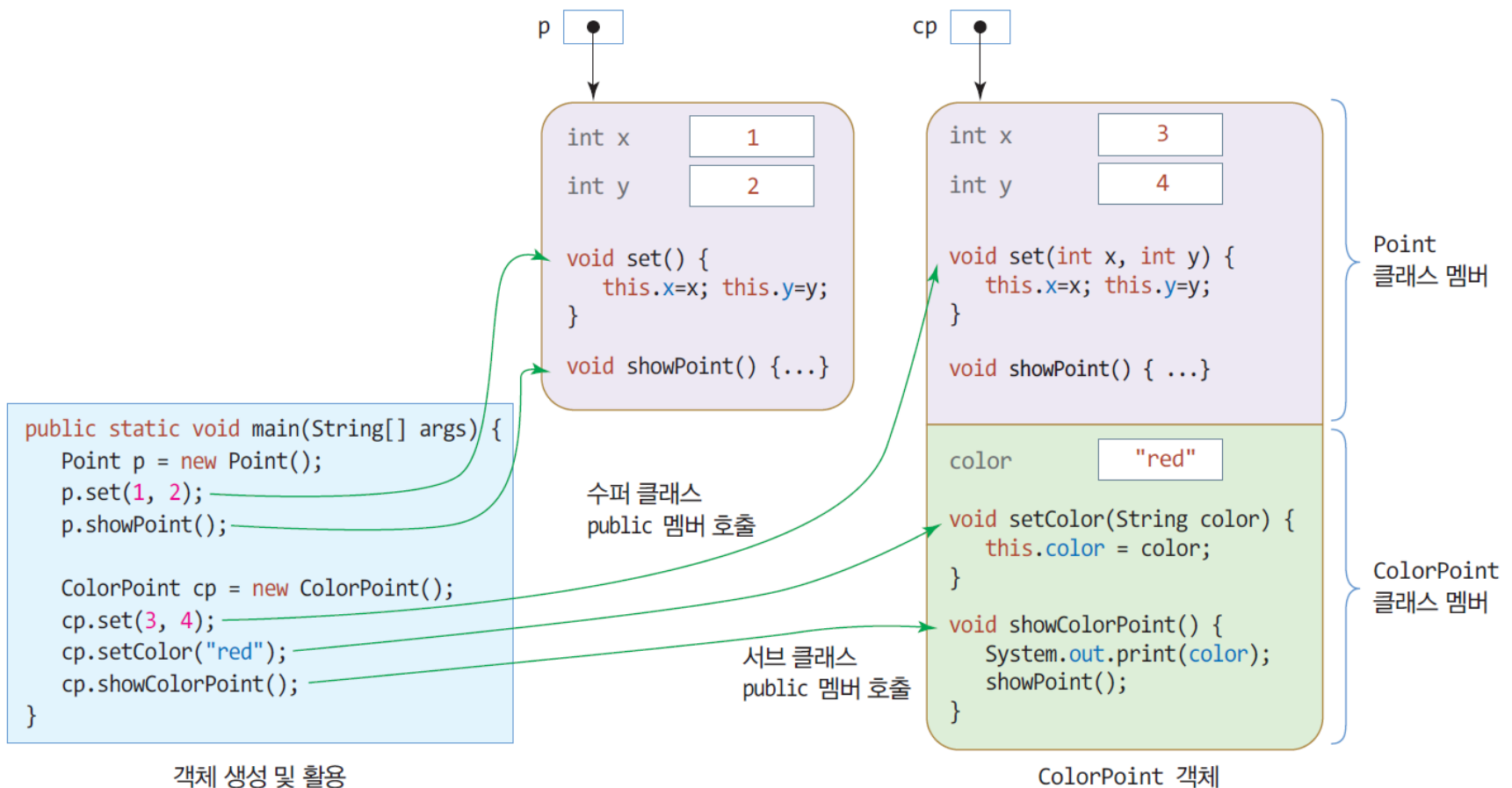
// Point를 상속받은 ColorPoint 선언
class ColorPoint extends Point {
    String color;
    void setColor(String c) {
        this.color = c;
    }
    void showColorPoint() {
        System.out.print(color);
        showPoint(); // Point의 showPoint() 호출
    }
}
```

```
public class ColorPointEx {
    public static void main(String [] args) {
        Point p = new Point();
        p.set(1, 2);
        p.showPoint();

        ColorPoint cp = new ColorPoint();
        cp.set(3, 4);
        cp.setColor("red");
        cp.showColorPoint();
    }
}
```

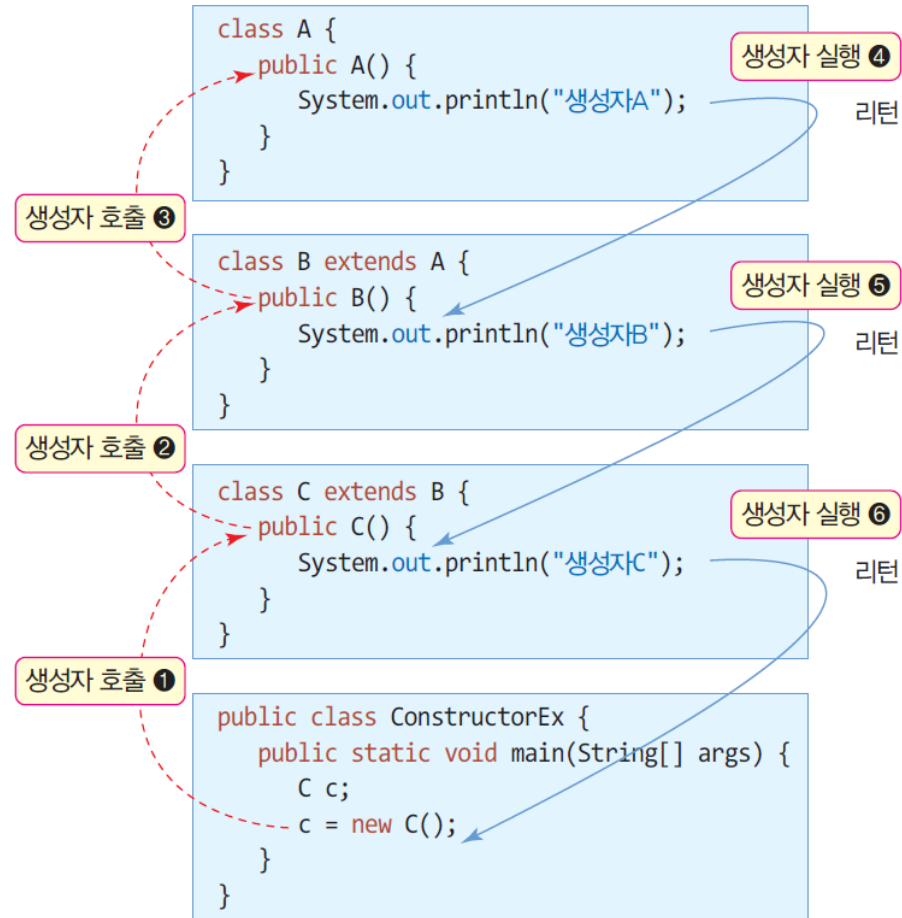
# 서브 클래스/슈퍼 클래스의 생성자 호출과 실행

- 슈퍼 클래스 객체와 서브 클래스의 객체는 별개
- 서브 클래스 객체는 슈퍼 클래스 멤버 포함





- 서브 클래스의 객체가 생성
  - 슈퍼클래스 생성자와 서브 클리
  - 실행 순서
    - 슈퍼 클래스의 생성자가 먼저 실행



결과  
생성자A  
생성자B  
생성자C

- **super()**
  - 서브 클래스에서 명시적으로 슈퍼 클래스의 생성자 선택 호출
  - 사용 방식
    - `super(parameter);`
    - 인자를 이용하여 슈퍼 클래스의 적당한 생성자 호출
    - 반드시 **서브 클래스 생성자 코드의 제일 첫 라인에 와야 함**

```
class Point {  
    private int x, y; // 한 점을 구성하는 x, y 좌표  
    Point() {  
        this.x = this.y = 0;  
    }  
    Point(int x, int y) {  
        this.x = x; this.y = y;  
    }  
    void showPoint() { // 점의 좌표 출력  
        System.out.println("(" + x + ", " + y + ")");  
    }  
}  
  
class ColorPoint extends Point {  
    private String color; // 점의 색  
    ColorPoint(int x, int y, String color) {  
        super(x, y); // Point의 생성자 Point(x, y) 호출  
        this.color = color;  
    }  
    void showColorPoint() { // 컬러 점의 좌표 출력  
        System.out.print(color);  
        showPoint(); // Point 클래스의 showPoint() 호출  
    }  
}
```

x=5,  
y=6

```
public class SuperEx {  
    public static void main(String[] args) {  
        ColorPoint cp = new ColorPoint(5, 6, "blue");  
        cp.showColorPoint();  
    }  
}
```

blue(5,6)

x=5, y=6,  
color = "blue" 전달

- 개발자가 서브 클래스의 생성자에 대해 슈퍼 클래스의 생성자를 명시적으로 선택하지 않은 경우

서브 클래스의  
기본 생성자에 대해  
컴파일러는 자동으로  
슈퍼 클래스의  
기본 생성자와 짝을 맺음

```
class A {  
    ➔ public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        .....  
    }  
}
```

```
class B extends A {  
    ➔ public B() {  
        System.out.println("생성자B");  
    }  
}
```

```
public class ConstructorEx2 {  
    public static void main(String[] args) {  
        B b;  
        b = new B(); // 생성자 호출  
    }  
}
```

생성자A  
생성자B

# 슈퍼 클래스에 기본 생성자가 없어 오류 난 경우

```
class A {  
    public A(int x) {  
        System.out.println("생성자A");  
    }  
}  
  
class B extends A {  
    public B() { // 오류 발생 오류  
        System.out.println("생성자B");  
    }  
}  
  
public class ConstructorEx2 {  
    public static void main(String[] args) {  
        B b;  
        b = new B();  
    }  
}
```



```
class A {  
    public A() {  
        System.out.println("생성자A");  
    }  
    public A(int x) {  
        System.out.println("매개변수생성자A");  
    }  
}  
  
class B extends A {  
    public B() {  
        System.out.println("생성자B");  
    }  
    public B(int x) {  
        System.out.println("매개변수생성자B");  
    }  
}  
  
public class ConstructorEx3 {  
    public static void main(String[] args) {  
        B b;  
        b = new B(5);  
    }  
}
```