

# Object Orientation First Story

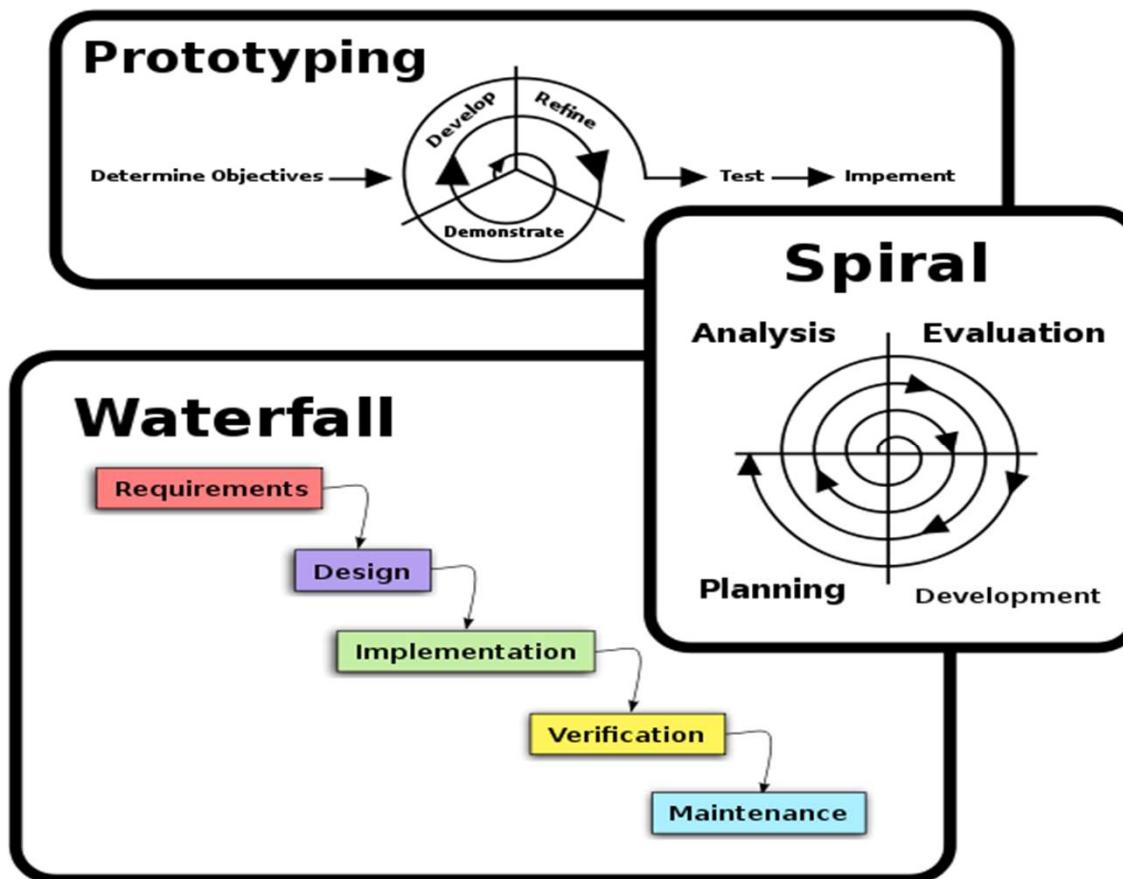
Bok, Jong Soon

[javaexpert@nate.com](mailto:javaexpert@nate.com)

<https://github.com/swacademy/Core-Java>

# Methodologies (Programming Paradigm)

- Is a fundamental style of computer programming.



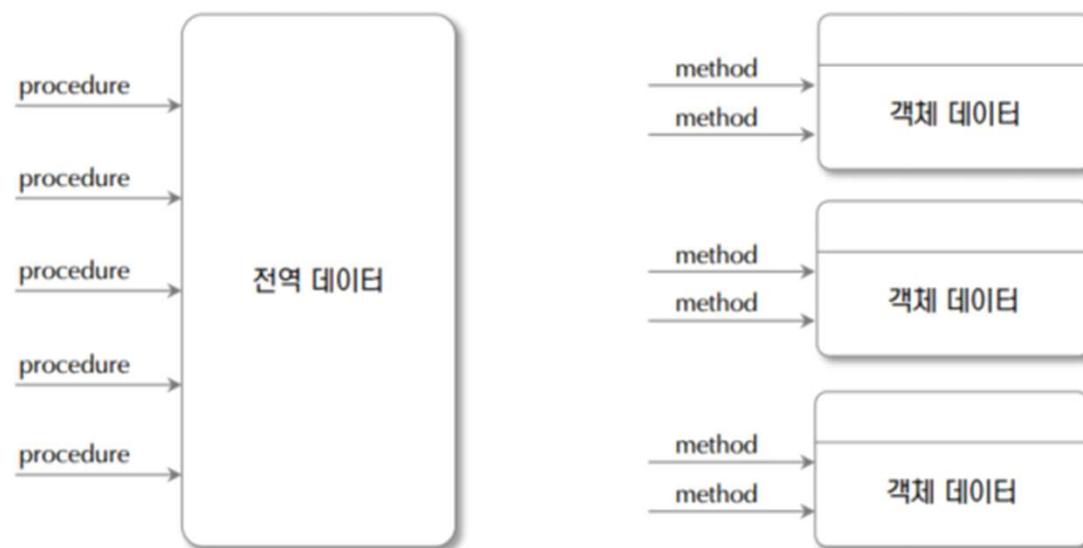
# Methodologies (Programming Paradigm)

## ✓ 절차지향 프로그래밍

- 문제를 기능 또는 구조위주의 관점으로 바라보며, 원하는 기능을 세분화하여 해결책을 찾아가는 프로그래밍 기법입니다.
- Algorithms + Data Structures = Programs (파스칼의 창시자인 Niklaus Wirth의 유명한 저서)

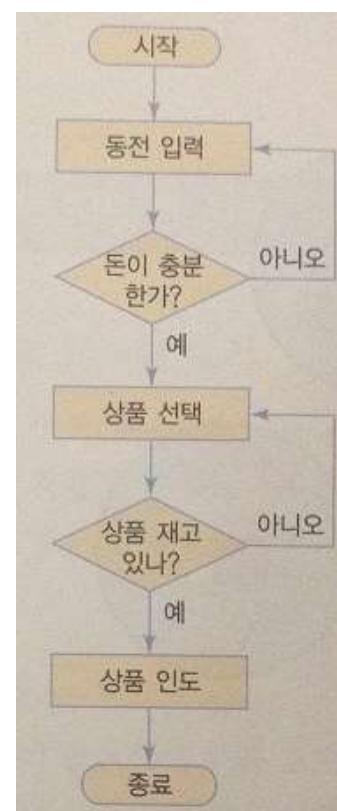
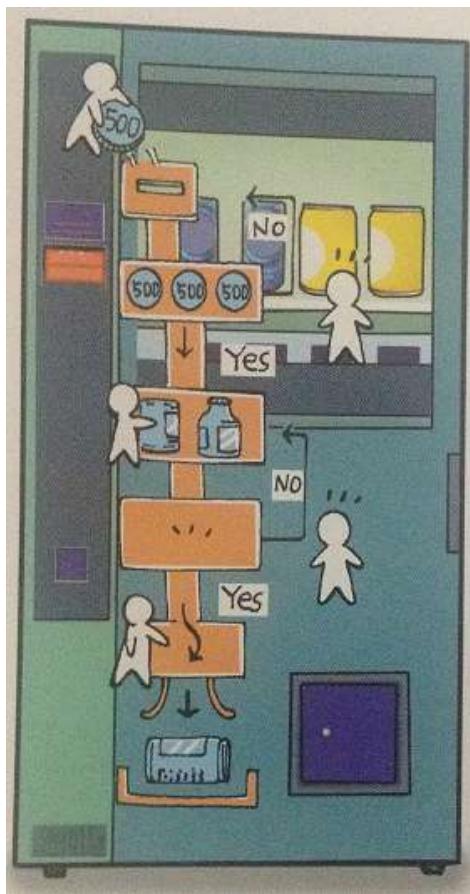
## ✓ 객체지향 프로그래밍

- 문제를 데이터 위주의 관점으로 바라보며, 데이터들의 상호 관계를 정의함으로써 해결책을 찾아가는 프로그래밍 기법입니다.
- 전통적인 절차지향 프로그래밍을 대체하기 위하여 1970년도에 개발되었습니다.



# Methodologies (Programming Paradigm)

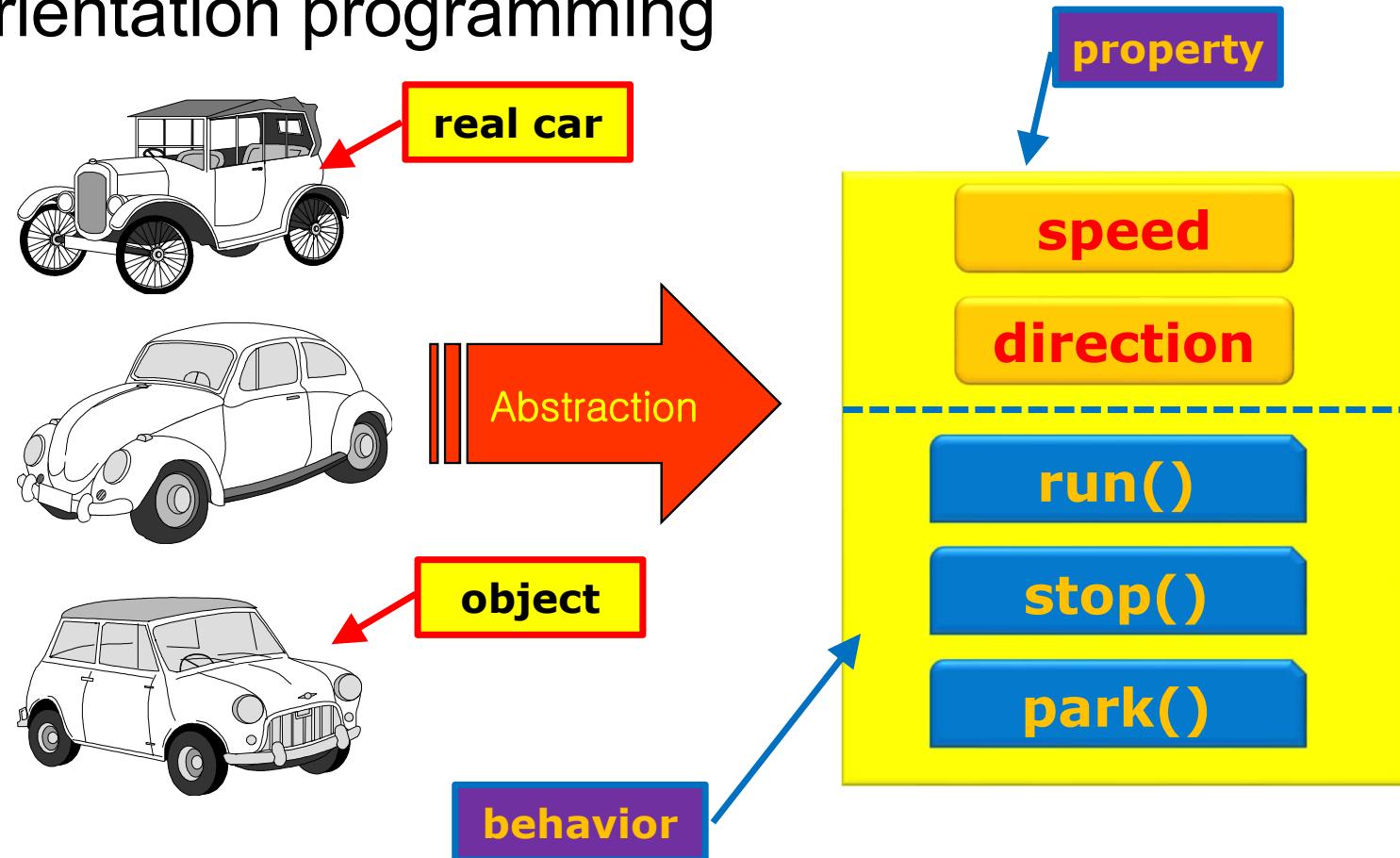
## ■ Procedural programming



황기태/김효수, “명품 JAVA Programming”, (경기 : 생능출판사, 2011), p.166.

# Methodologies (Programming Paradigm)

## ■ Object-orientation programming



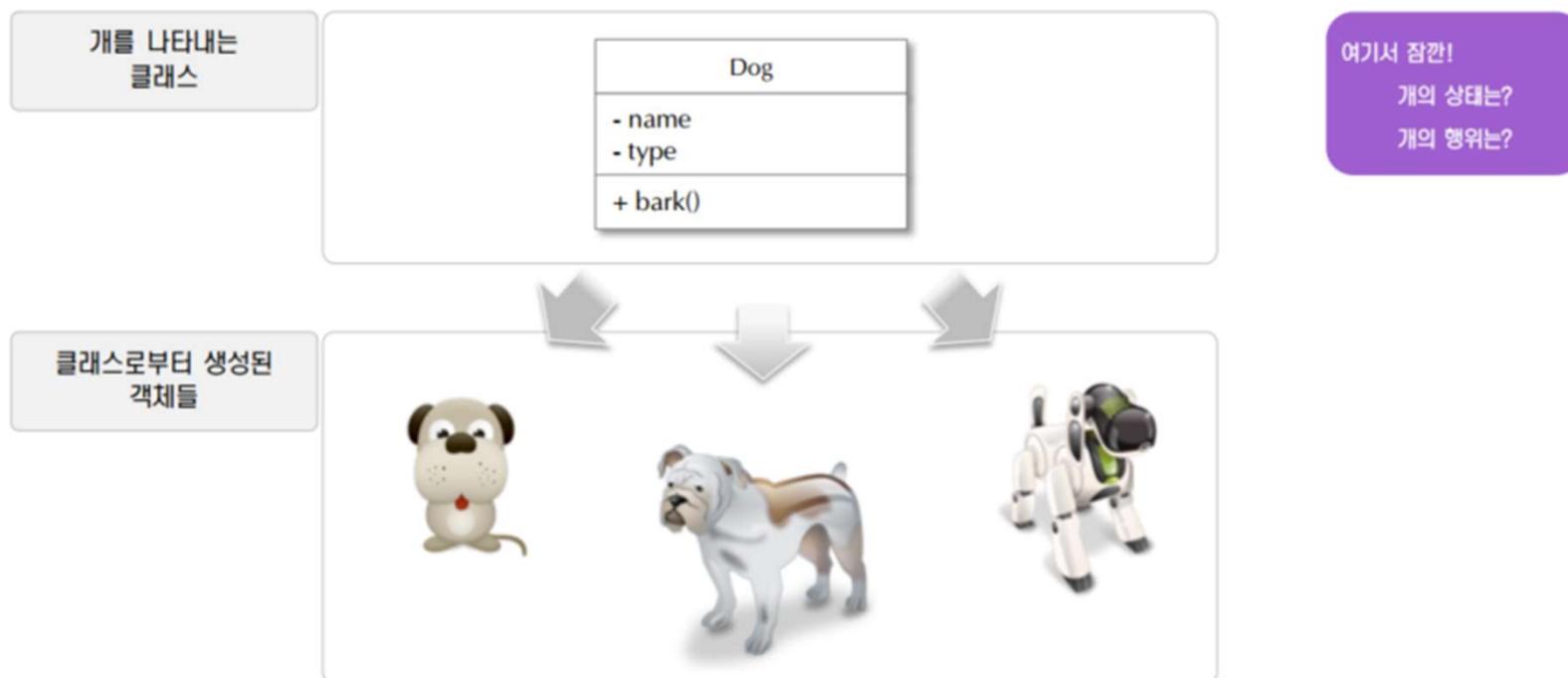
# Object Orientation

- Classes : Source-code templates for objects
- Objects : Runtime instances of classes
- Members : Items put into a class to define the data content (data members) and functionality (member methods) of the objects
  - Data Members - Variables and constants which store the values that model the real-world concept the objects represent
  - Member Methods – Statements grouped into a standalone “module” for accessing or modifying data members within a class

# Object Orientation (Cont.)

✓ 클래스는 객체를 만들기 위한 청사진 또는 템플릿입니다.

- 클래스는 상태(속성)와 행위(메소드)를 가집니다.
- 클래스로부터 생성된 객체를 클래스의 인스턴스라고 합니다.
- 자바에서 모든 코드는 클래스 안에 존재합니다.
- 애플리케이션이 해결하고자 하는 문제영역(problem domain)의 객체를 클래스로 작성합니다.



# Object Orientation (Cont.)

## ✓ 속성 (Attribute)

- 객체가 가지는 변수를 속성이라고 합니다.
- 속성 특정한 값을 가질 수 있으며, 객체의 속성 값은 객체의 상태를 표현합니다.

## ✓ 메소드 (Method)

- 데이터를 조작하는 행위를 하는 부분을 메소드라고 합니다.
- 메소드를 호출하여 객체의 상태를 변경하거나 내부 상태 값을 가져올 수 있습니다.



# Object Orientation (Cont.)

## ■ Object Orientation Methodology's 3R

- Readability
- Reusability
- Reliability

# What Is a Class?

## ■ For the philosopher...

- An artifact of human **classification**!
- **Class**ify based on common behaviour or attributes
- Agree on descriptions and names of useful **classes**
- Create vocabulary; we communicate; we think!

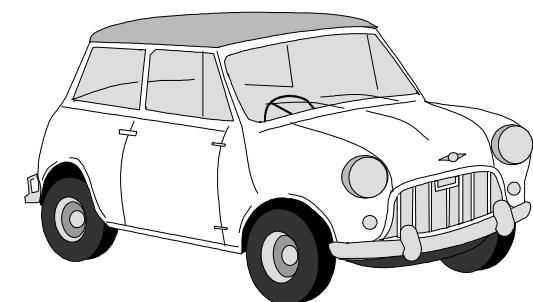
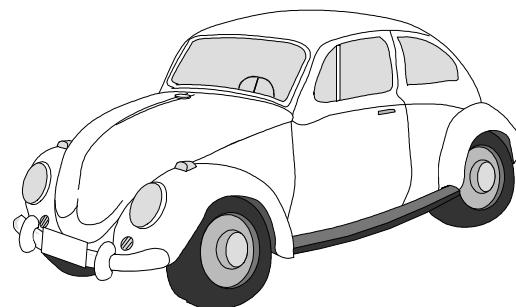
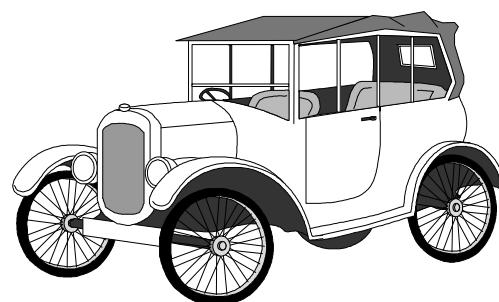
## ■ For the object-oriented programmer...

- A named syntactic construct that describes common behaviour and attributes
- A data structure that includes both data and functions



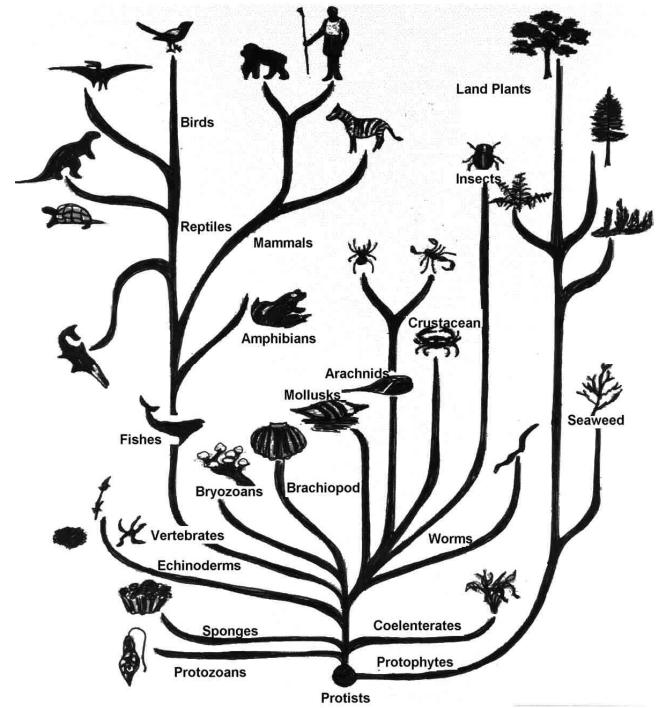
# What Is an Object?

- An object is an instance of a class
- Objects exhibit:
  - Identity: Objects are distinguishable from one another
  - Behaviour(operation, function) : Objects can perform tasks
  - State(property, attribute: Objects store information



# Object-Oriented Key Features

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



# Abstraction

## ■ Abstraction is selective ignorance

- Decide what is important and what is not
- Focus and depend on what is important
- Ignore and do not depend on what is unimportant
- Use encapsulation to enforce an abstraction

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Edsger Dijkstra

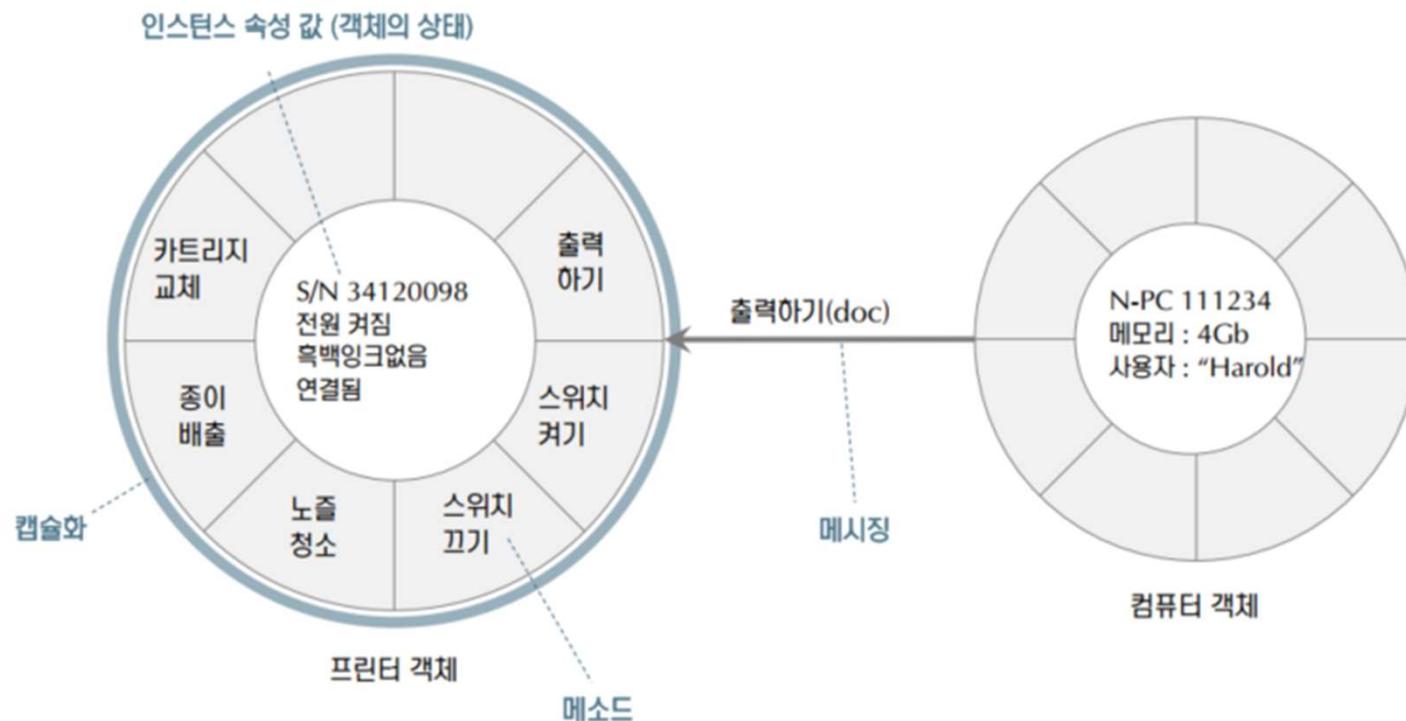
# Encapsulation

- The principle of protecting sensitive parts of your objects from external manipulation.
- Operations and attributes are its *members*.
- Members can be **public** or **private**.
- All variables should be kept **private**.
- Variables are modified by methods of their *own* class.
- Hides the implementation details of a class.
- Forces the user to use an interface to access data.
- Makes the code more maintainable.

# Encapsulation (Cont.)

## ✓ 캡슐화

- 데이터와 행위를 하나의 캡슐처럼 포장하는 것을 캡슐화라고 합니다.
- 객체의 상세한 구현내용은 사용자로부터 숨겨지기 때문에 정보은닉(또는 구현은닉)이라고도 합니다.
- 인스턴스 속성 외부에서 직접 접근할 수 없도록 하고, 메소드를 통해서만 접근할 수 있도록 하는 것이 캡슐화의 핵심입니다.



# Encapsulation Examples

## **Student**

-kor: int  
+display(): void  
+getKor(): int  
+setKor(kor: int): void

## **Product**

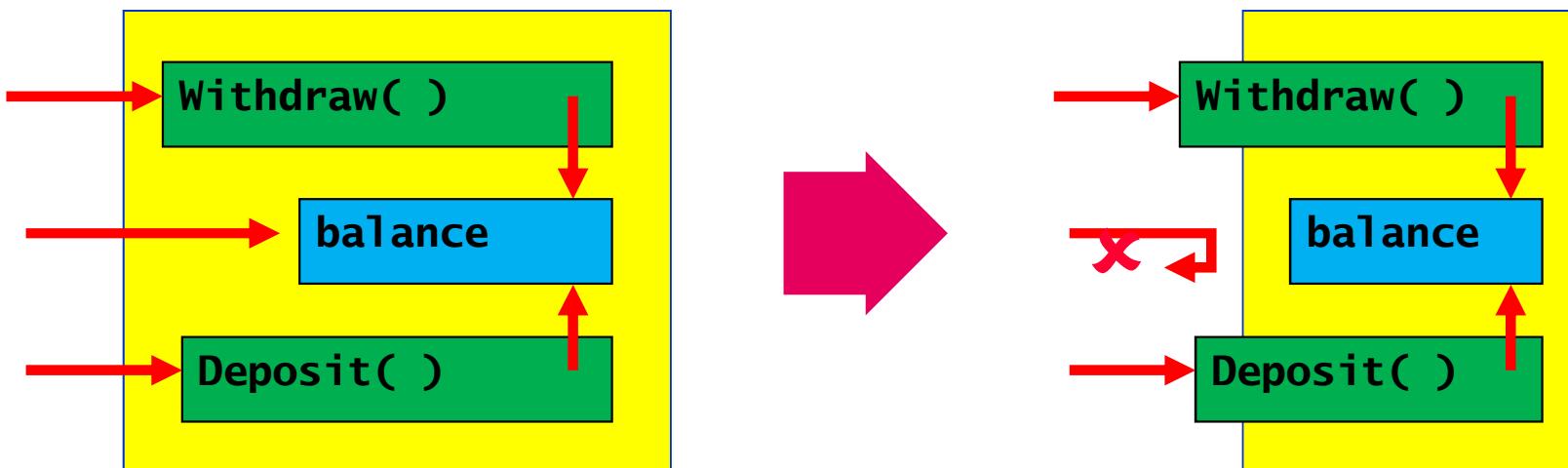
-productName: String  
+getProductName(): String  
+setProductName(name: String): void  
+display(): void

# Restricting Data Access

- Use modifiers to restrict data access:
  - **public**
    - Everyone can use that part of the object.
  - **private**
    - No one outside the object's class can use that part of the object.

# Implementing Encapsulation

- Methods are **public**, accessible from the outside.
- Data is **private**, accessible only from the inside.



## Implementing Encapsulation (Cont.)

- Put **public** or **private** in front of members

```
private int myInt;  
private String name;  
public String getName() {  
    return name;  
}
```



## get Methods and set Methods

- When variables are **private**, they must be accessed by member methods.
- **get** and **set** methods obtain and assign values.

```
class EncapsulatedEmployee{  
    private int employeeNumber;  
    public void setEmployeeNumber(int newValue) {  
        employeeNumber = newValue;  
    }  
    public int getEmployeeNumber() {  
        return employeeNumber;  
    }  
}
```

# The **this** Reference

- The **this** keyword means “*reference to the same object*”.

```
class Example {  
    void method1() {  
        this.method2();  
    }  
    void method2() {  
        //whatever method2 does  
    }  
}
```

## The **this** Reference (Cont.)

```
class Test {  
    private int kor;  
    public void setKor(int kor) {  
        this.kor = kor;  
    }  
}
```

## The **this** Reference (Cont.)

```
class A {  
    int a;  
    B b;  
    public A() {  
        b = new B();  
    }  
    void a(int a){  
        this.a = a;  
        b.doJob(this);  
    }  
}
```

```
class B {  
    public B() {}  
    void doJob(A a) {  
        System.out.println(a.a);  
    }  
}
```

```
public class This{  
    public static void main(String [] args){  
        A a = new A();  
        a.a(10);  
    }  
}
```

# The **this** Reference (Cont.)

- ✓ **this** 는 객체 자신을 나타내는 키워드입니다.
- ✓ 객체 자신의 필드를 참조하거나 해당 클래스의 다른 생성자를 호출할 때 사용합니다.
- ✓ 정적(Static) 메소드에서는 **this**를 사용할 수 없습니다.

```
public class Person {  
  
    private String name;  
    private int age;  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public Person(String name, int age) {  
        this(name);  
        this.age = age;  
    }  
  
    public int getAge() { return age; }  
    public void setAge(int age) { this.age = age; }  
    public String getName() { return name; }  
}
```

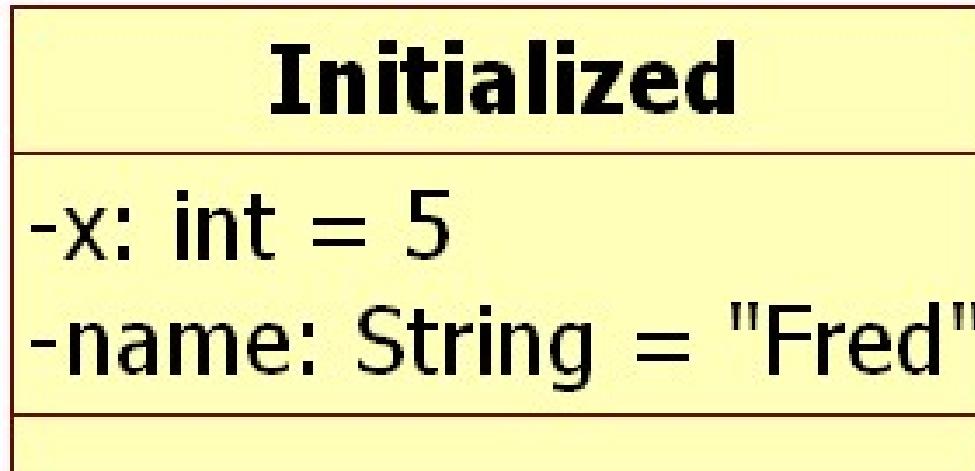
# Creating Objects

- Step 1: Allocating memory
  - Use **new** keyword to allocate memory from the heap
- Step 2: Initializing the object by using a ***constructor***
  - Use the name of the class followed by parentheses

```
Date when = new Date( );
```

# Explicit Member Initialization

```
public class Initialized {  
    private int x = 5;  
    private String name = "Fred";  
}
```



# Constructors

- Are special methods.
- Are called each time you create an object.
- Have no return type.
- Have the same name as the class name.
- Constructors allow you to specify values for objects when you create them.
- You have been using a *default constructor* throughout most of the course.

# Constructors (Cont.)

- ✓ 생성자(Constructor)는 클래스와 이름이 같은 메소드입니다.
- ✓ 생성자는 클래스 초기화와 관련된 작업을 합니다.
- ✓ 생성자를 호출할 때 매개변수를 넘겨서 초기화 작업을 수행할 수 있습니다.
- ✓ 객체를 생성할 때는 생성자 이름 앞에 new 키워드를 붙여서 호출합니다.

```
// Employee 생성자
public Employee(String n, double s, int year, int month, int day) {
    name = n;
    salary = s;
    GregorianCalendar calendar =
        new GregorianCalendar(year, month - 1, day);
    hireDay = calendar.getTime();
}
```

- ✓ 생성자를 이용한 객체 생성

```
Employee employee = new Employee("Harold", 10000, 2008, 1, 2);
```

- 생성된 객체의 인스턴스 필드 값은 아래와 같습니다.
  - name : Harold
  - salary : 10000
  - hireDay : January 2, 2008

# Default Constructors

- All classes must have at least one constructor.
- The compiler provides a default constructor to any class which does not have an explicit constructor.
- Features of a default constructor
  - Public accessibility
  - Same name as the class
  - No return type—not even void
  - Expects no arguments
  - Initializes all fields to zero, false or null

## Default Constructors (Cont.)

- If a class has no constructor, a default constructor is inserted.
- When you use **new** to instantiate an object, **new** automatically calls the class's default constructor.
- The compiler will insert the default constructor.

## Default Constructors (Cont.)

- ✓ 생성자에서 명시적으로 필드에 값을 설정하지 않으면, 디폴트 값으로 초기화 됩니다.
- ✓ 타입 별 디폴트 값은 다음과 같습니다.
  - number → 0, boolean → false, 객체 레퍼런스 → null
- ✓ null로 초기화된 객체의 필드나 메소드를 사용하면 Null Pointer Exception이 발생합니다.

```
class Employee {  
    private String name;  
    private int salary;  
    private Date hireDay;  
    ...  
}
```

```
Employee harry = new Employee();  
Date h = harry.getHireDay();  
calendar.setTime(h); // throws exception if h is null
```

생성자에서 초기화 작업을 하지 않는 경우 salary는 0으로, name과 hireDay는 null로 초기화 됩니다.  
null로 초기화된 hireDay의 필드를 사용하려고 하는 경우, 예외가 발생하게 됩니다.

## Default Constructors (Cont.)

- ✓ 생성자를 정의하지 않으면, 기본적으로 인자가 없는 생성자가 제공됩니다.
- ✓ 인자가 없는 생성자는 디폴트 값으로 필드를 초기화합니다.
- ✓ 디폴트 값이 아닌 다른 값으로 필드를 초기화해야 하는 경우 생성자를 재정의 합니다.

```
public Employee() {  
    name = "";  
    salary = 0;  
    hireDay = new Date();  
}
```

객체 생성 후 name 또는 hireDay가 바로 사용될 때, null로 인한 예외가 발생하는 것을 방지하기 위하여 빈 객체를 생성하여 초기화합니다.

- ✓ 클래스에 생성자를 하나라도 정의하게 되면, 인자가 없는 생성자는 제공되지 않습니다.

```
public Employee(String name, double salary, int y, int m, int d) {  
    ...  
}
```

```
Employee emp = new Employee(); // Compile Error
```

인자가 없는 생성자가 없으므로, 위 소스코드는 컴파일러가 발생합니다. 이런 경우, 인자가 없는 생성자를 추가로 정의해야 합니다.

# Overriding the Default Constructor

- The default constructor might be inappropriate
  - If so, do not use it; write your own!

```
class Date {  
  
    public Date( ) {  
        ccyy = 1970;  
        mm = 1;  
        dd = 1;  
    }  
    private int ccyy, mm, dd;  
}
```

# Overloading Method Revisited

- It can be used as follows:

```
public void println(int i)  
public void println(float f)  
public void println(String s)
```

- Argument lists *must* differ.
- Return types *can* be different.

# Overloading Constructors

## ■ Constructors are methods and can be overloaded

- Same scope, same name, different parameters.
- Allows objects to be initialized in different ways.

## ■ WARNING

- If you write a constructor for a class, the compiler does not create a default constructor.

```
class Date {  
    public Date( ) { ... }  
    public Date(int year, int month, int day) { ... }  
    ...  
}
```

# Overloading Constructors (Cont.)

- ✓ 오버로딩은 메소드의 이름이 같고, 매개 변수만 다른 것을 말합니다.
- ✓ 생성자 오버로딩을 통해, 클래스는 하나 이상의 생성자를 가질 수 있습니다.

```
GregorianCalendar today = new GregorianCalendar();
or
GregorianCalendar deadline = new GregorianCalendar(2099, Calendar.DECEMBER, 31);
```

## ✓ 메소드 오버로딩(Overloading)

- 여러 메소드의 이름이 동일하고, 파라미터 타입이 서로 다른 것을 의미합니다. (파라미터 타입은 메소드 시그니처라고 합니다.)
- 컴파일러는 메소드가 호출되면 파라미터 타입이 일치하는 것을 찾아 메소드를 수행하는데 이러한 처리과정을 'overloading resolution' 이라고 합니다.
- 생성자 또한 메소드이므로 오버로딩할 수 있습니다.
- 리턴 타입은 메소드 시그니처의 일부가 아니므로, 메소드명과 파라미터 타입이 같고 반환형만 다른 것은 정의할 수 없습니다.

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

## Overloading Constructors (Cont.)

- You can use the **this** reference at the first line of a constructor to call another constructor.

```
public class Employee {  
    private String name;  
    private int salary;  
    public Employee (String n, int s) {  
        name = n;  
        salary = s;  
    }  
    public Employee (String n) {  
        this(n,0);  
    }  
    public Employee () {  
        this("Unknown");  
    }  
}
```

## Overloading Constructors (Cont.)

- ✓ 객체 생성자에서는 같은 클래스의 다른 생성자를 호출할 수 있습니다.
- ✓ `this` 키워드로 호출하려는 생성자의 매개변수와 타입을 맞추어 다른 생성자를 호출합니다.
- ✓ 단, `this` 키워드를 이용한 생성자 호출은 생성자의 첫 번째 줄에서만 가능합니다.

```
public Employee(double s) {  
    this("Employee #" + nextId, s); // calls Employee(String, double)  
    nextId++;  
}  
public Employee(String name, double salary) {  
    this.name = name;  
    this.salary = salary;  
}
```

`new Employee(60000)`가 호출되면 `Employee(double)` 생성자는 `Employee(String, double)` 생성자를 호출합니다.

# Overloading Constructors (Cont.)

- ✓ 생성자 파라미터 이름을 정하는 몇 가지 방법입니다.

- 가장 빠르게 사용할 수 있는 방법 → 파라미터 이름만으로는 변수의 의미를 알기 어렵습니다.

```
public Employee(String n, double s) {  
    name = n;  
    salary = s;  
}
```

- 다른 대안입니다. → 매번 a를 붙이는 작업을 하려니, 번거로워 보입니다.

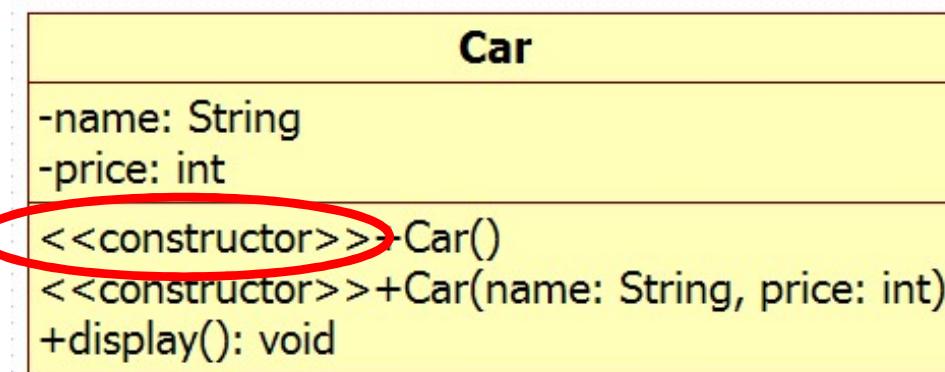
```
public Employee(String aName, double aSalary) {  
    name = aName;  
    salary = aSalary;  
}
```

- 권장하는 방법입니다. (필드명과 동일하게 작성하고, 필드는 this로 접근합니다.)

```
public Employee(String name, double salary) {  
    this.name = name;  
    this.salary = salary;  
}
```

# Overloading Constructors (Cont.)

```
2 public class Car {  
3     private String name; //member variable  
4     private int price; //member variable  
5  
6     public Car(){ //default constructor  
7         System.out.println("Default Constructor");  
8     }  
9     public Car(String name, int price){ //constructor  
10        System.out.println("Constructor");  
11        this.name = name;  
12        this.price = price;  
13    }  
14    public void display(){  
15        System.out.printf("name = %s, price = %,d\n", this.name, this.price);  
16    }  
17 }
```



# Instance Initialization Block

- Are called each time you create an object such *constructor*.
- Is performed prior to *constructor*.

```
2 public class Car {  
3     private String name;    //member variable  
4     private int price;      //member variable  
5  
6     { //Instance Initialization Block  
7         System.out.println("Instance Initialization Block");  
8         this.name = "Matiz";  
9         this.price = 10_000_000;  
10        display();  
11    }  
12  
13    public Car(String name, int price){ //Constructor  
14        System.out.println("Constructor");  
15        this.name = name;  
16        this.price = price;  
17    }  
18    public void display(){  
19        System.out.printf("name = %s, price = %,d\n", this.name, this.price);  
20    }  
}
```

# Instance Initialization Block (Cont.)

✓ 필드를 초기화하는 3가지 방법입니다.

- 생성자를 이용한 방법
- 필드 선언 시 초기값을 할당하는 방법
- 초기화 블록을 사용하는 방법

✓ 초기화 블록은 클래스의 객체가 생성될 때마다 실행됩니다.

```
class Employee {  
    private static int nextId;  
    private int id;  
    private String name;  
    private double salary;  
  
    // 데이터 초기화 블록  
    {  
        id = nextId;  
        nextId++;  
    }  
  
    public Employee(String n, double s) {  
        name = n;  
        salary = s;  
    }  
    public Employee() {  
        name = "";  
        salary = 0;  
    }  
    ...  
}
```

1. 모든 데이터 필드가 디폴트 값으로 초기화됩니다.  
(0, false or null)
2. 클래스에 선언된 순으로 모든 필드 초기화문(initializer) 및 초기화 블록이 실행됩니다.
3. 만약, 생성자의 첫 번째 줄에서 두 번째 생성자를 호출하는 경우,  
두 번째 생성자의 본문(body)이 실행됩니다.
4. 생성자의 본문(body)이 실행됩니다.

# Instance Initialization Block (Cont.)

- ✓ 동일 클래스에 대해서 한번만 수행해야 하는 코드는 정적 초기화 블록을 사용합니다.
- ✓ 정적 초기화 블록은 클래스가 처음 로드될 때 한번만 실행됩니다.
- ✓ 정적 초기화 블록이 여러 개인 경우 선언된 순서대로 실행됩니다. (생성자보다 먼저 실행됩니다.)
- ✓ 정적 필드를 초기화 하기 위하여 정적 초기화 블록을 사용할 수 있습니다.

```
class Employee {  
    private static int nextId;  
    . . .  
  
    // 정적 초기화 블록  
    static  
    {  
        Random generator = new Random();  
        nextId = generator.nextInt(10000);  
    }  
    . . .  
}
```

# Class Structure

- ✓ 하나의 클래스는 하나의 자바 파일로 구성되어 있습니다.
- ✓ 클래스의 내부에는 필드와 메소드 그리고 생성자 등이 있습니다.
- ✓ 생성자는 클래스와 이름이 같은 메소드입니다.

```
class 클래스명 {  
    필드1  
    필드2  
    ...  
  
    생성자1  
    생성자2  
    ...  
  
    메소드1  
    메소드2  
    ...  
}
```

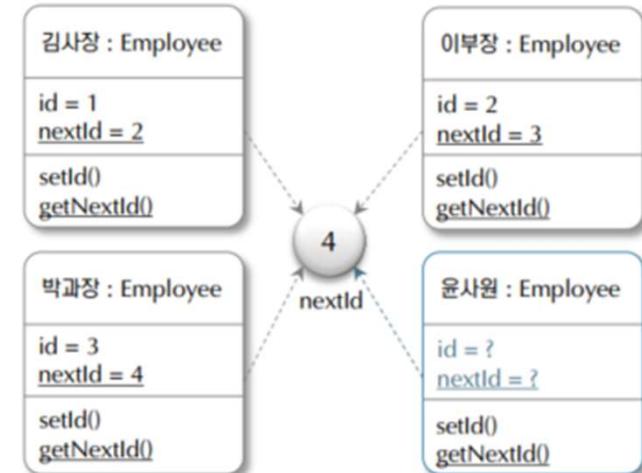
```
class Employee {  
    // 필드  
    private String name;  
    private double salary;  
    private Date hireDay;  
  
    // 생성자  
    public Employee(String n, double s, int year, int month, int day) {  
        name = n;  
        salary = s;  
        GregorianCalendar calendar =  
            new GregorianCalendar(year, month - 1, day);  
        hireDay = calendar.getTime();  
    }  
    // 메소드  
    public String getName() {  
        return name;  
    }  
    ...  
}
```

# Class Structure (Cont.)

- ✓ 인스턴스 변수에 static을 붙이면 정적 필드가 됩니다.
- ✓ 정적 필드는 해당 클래스의 모든 인스턴스들이 공유하는 클래스 변수입니다.
- ✓ 정적 필드는 객체를 생성하지 않아도 필드의 값을 사용할 수 있습니다.
  - 단, 접근제한자가 public인 경우만 가능하며 private인 경우는 메소드를 통해서 접근해야 합니다.

```
public class Employee {  
  
    private static int nextId = 1;  
    private int id;  
  
    public void setId() {  
        id = nextId;  
        nextId++;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public int getNextId() {  
        return nextId;  
    }  
}
```

```
public static void main(String[] args) {  
    Employee kim = new Employee();  
    Employee park = new Employee();  
    Employee lee = new Employee();  
    Employee yoon = new Employee();  
  
    kim.setId();  
    System.out.println(kim.getNextId());  
  
    lee.setId();  
    System.out.println(lee.getNextId());  
  
    park.setId();  
    System.out.println(park.getNextId());  
  
    yoon.setId();  
    System.out.println(yoon.getNextId());  
}  
Employee.java
```



새로운 객체의 실행결과는?

# Class Structure (Cont.)

- ✓ 정적 필드에 final 키워드를 사용하여 값을 변경할 수 없도록 하면 정적 상수가 됩니다.
- ✓ 정적 필드는 객체를 생성하지 않고도 클래스 이름으로 접근해서 사용할 수 있습니다.
- ✓ 정적 상수는 모든 객체가 공유하는 값이 변하지 않는 필드입니다.

예) Math.PI

```
public class Math {  
    . . .  
    public static final double PI = 3.14159265358979323846;  
    . . .  
}
```

예) System.out

```
public class System {  
    . . .  
    public static final PrintStream out = . . .;  
    . . .  
}
```

# Class Structure (Cont.)

✓ 정적 메소드는 static으로 선언된 메소드로써 인스턴스 없이도 호출할 수 있습니다.

- 정적 메소드는 인스턴스 필드에는 접근할 수 없고, 정적 필드에만 접근할 수 있습니다.
- 정적 메소드는 객체를 통해 사용될 수 있지만, 반드시 클래스명과 함께 사용하기 바랍니다.

✓ 정적 메소드 예시

- Employee 클래스의 정적필드에 접근하는 정적 메소드

```
class Employee {  
    private static int nextId = 1;  
    private int id;  
  
    . . .  
    public static int getNextId() {  
        return nextId; // 정적 필드의 값을 리턴합니다.  
    }  
    . . .  
}
```

- Math.pow()

✓ 정적 메소드는 언제 사용할까요?

- 객체의 상태에 접근하지 않고, 필요한 파라미터가 모두 명시적 파라미터인 경우 (예, Math.pow)
- 클래스의 정적 필드에만 접근하는 경우 (예, Employee.getNextId)

## Class Structure (Cont.)

- ✓ main() 메소드는 Java 프로그램을 시작할 때 사용하는 메소드로 어떠한 클래스에도 선언될 수 있습니다.
- ✓ 인스턴스 생성과는 무관한 메소드이기 때문에 static으로 선언됩니다.
  - java 프로그램의 구동 : java Employee
  - 단위 테스트 (어떠한 클래스에도 선언될 수 있는 main 메소드의 특성을 이용)

```
class Employee {  
    public Employee( String n, double s, int year, int month, int day) {  
        name = n; salary = s;  
        GregorianCalendar calendar = new GregorianCalendar(year, month-1, day);  
        hireDay = calendar.getTime();  
    }  
    . . .  
  
    // 단위 테스트  
    public static void main(String[] args) {  
        Employee e = new Employee("Romeo", 50000, 2003, 3, 31);  
        e.raiseSalary(10);  
        System.out.println(e.getName() + " " + e.getSalary());  
    }  
    . . .  
}
```

# Class Structure (Cont.)

## ✓ 소멸자 (Destructor methods)

- C++과 같은 일부 객체지향 프로그래밍 언어에서는 객체가 더 이상 사용되지 않을 때 메모리 공간을 회수하는 코드를 작성할 수 있도록 명시적으로 소멸자를 가지고 있습니다.

## ✓ Java에는 소멸자가 없습니다.

- Java는 자동으로 garbage collection이 동작하여 사용하지 않는 메모리 공간을 회수하기 때문에 소멸자가 제공되지 않습니다.

## ✓ finalize 메소드

- 객체가 메모리가 아닌 파일이나 다른 시스템의 자원을 활용하는 경우가 있습니다.
- 더 이상 자원을 사용하지 않을 때, 자원을 반환하는 처리를 해주기 위하여 finalize 메소드를 추가할 수 있습니다.
- finalize 메소드를 구현해 놓으면, garbage collector가 자동으로 호출합니다. (콜백 메소드)
- 그러나 finalize 메소드는 언제 호출될 지 모르기 때문에, 실무에서는 이 메소드에 의존할 수가 없습니다.

## ✓ 자원을 제공하는 객체는 사용이 끝난 후 자원을 정리할 수 있도록 close() 메소드를 제공합니다.

- 자원을 사용하는 객체는 사용이 끝난 후에 close() 메소드를 호출합니다.