The Object-Oriented Thought Process

Chapter 03

Advanced Object-Oriented Concepts

Contents

Constructors

Constructors are used to initialize objects.

- In Java and C++, C#, constructors are methods that share the same name as the class.
- Objective-C uses the *init keyword.*

- Constructor: special methods - initialization
  - have no return type
- a constructor for the Cabbie class would look like this:

```
public Cabbie(){
  /* code to construct the object */
}
```

- The compiler will recognize that the method name is identical to the class name and consider the method a constructor.

- Class Cabbie

```
public class Cabbie {
        public Cabbie(){
          /* code to construct the object */
        }
}
```

  - No return type
  - Same as class name

```
public int Cabbie(){
  /*  the compiler will not consider this a constructor */
}
```

When a Constructor is Called

When a new object is created, one of the first things that happens is that the constructor is called.

- ***new*** creates a new instance of the class,
  - thus allocating the required memory.
- Then the constructor itself is called, passing the arguments in the parameter list.
  - Providing the opportunity to attend to the appropriate initialization.

```
Cabbie myCabbie = new Cabbie();
```

- The new keyword creates a new instance of the Cabbie class, thus allocating the required memory.
- Then the constructor itself is called, passing the arguments in the parameter list.
- The constructor provides the developer the opportunity to attend to the appropriate initialization.

What's Inside a Constructor

Perhaps the most important function of a constructor is to initialize the memory allocated.

- In short, code included inside a constructor should set the newly created object to its initial, stable, safe state.
- Ex) **Count** object having the *count* attribute
  - set *count* to zero in the constructor:

```
count = 0;
```

The Default Constructor

If the class provides no explicit constructor, a default constructor will be provided.

- It is important to understand that at least one constructor always exists, regardless of whether you write a constructor yourself.
- If you do not provide a constructor, the system will provide a default constructor for you.

- the only action that a default constructor takes is to <u>call the constructor of its superclass</u>.

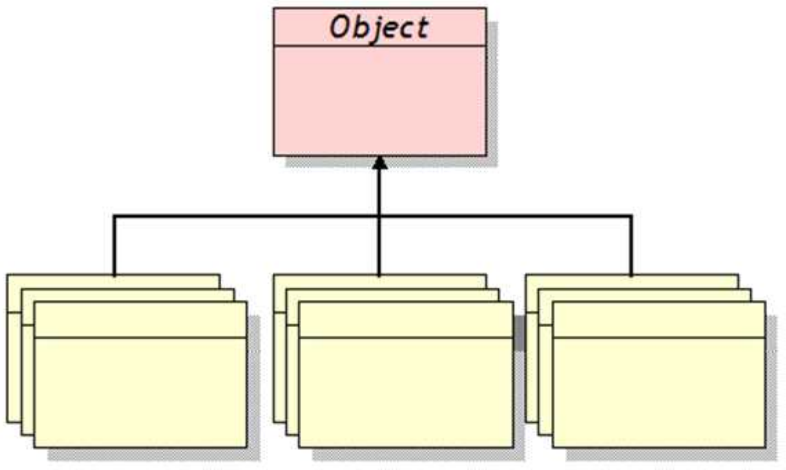
```
public Cabbie() {
    super();
}
```

수퍼클래스
객체를
가리킨다.

  – In JAVA, if Cabbie does not explicitly inherit from another class, the Object class will be the parent class.

[참고] JAVA 최상위 클래스 Object

- Object 클래스는 자바 상속계층의 가장 위에 있다.
- 모든 자바 클래스는 내부적으로 Object 클래스를 상속
  - 자바 클래스가 아무것도 상속하지 않으면 java.lang 패키지의 Object 클래스를 자동으로 상속한다
  - 자바의 모든 객체는 Object 클래스에 정의된 메소드 호출 가능



[redacted]

[redacted]

[참고] Object 클래스 주요 메소드

메소드	설명
protected Object clone()	객체 자신의 복사본을 생성하여 반환한다.
public boolean equals(Object obj)	Obj가 이 객체와 내용이 같은지를 나타낸다.
protected void finalize()	가비지 콜렉터에 의하여 호출된다.
public final Class getClass()	객체의 실행 클래스를 반환한다.
public int hashCode()	객체에 대한 해쉬 코드를 반환한다.
public String toString()	객체의 문자열 표현을 반환한다.

Using Multiple Constructors

In many cases, an object can be constructed in more than one way.

- To accommodate this situation, you need to provide more than one constructor.
- This is called *overloading a method* *(overloading pertains to all methods, not just constructors).*
  - Most OO languages provide functionality for overloading a method.

- To define multiple methods with the same name (but different "signature")

```
public class Count {
        int count;
        public Count(){
                count = 0;
        }
}
```

```
public Count (int number){
        count = number;
}
```

Overloading Methods

Overloading(중복 정의) allows a programmer to use the same method name over and over.

- As long as the signature of the method is different each time.
- The signature consists of the method name and a parameter list

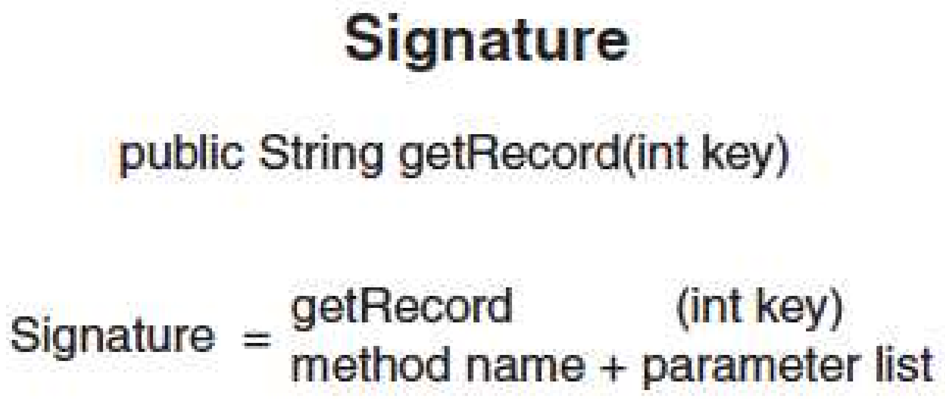

Signature
public String getRecord(int key)
Signature =
getRecord
(int key)
method name + parameter list

Figure 3.1 The components of a signature.

Methods all have different signatures:

```
public void getCab();
// different parameter list
public void getCab (String cabbieName);
// different parameter list
public void getCab (int numberOfPassengers);
```

Signatures

- Depending on the language, the signature may or may not include the return type.
- In Java and C#, the return type is not part of the signature.
- For example, the following methods would conflict even though the return types are different:

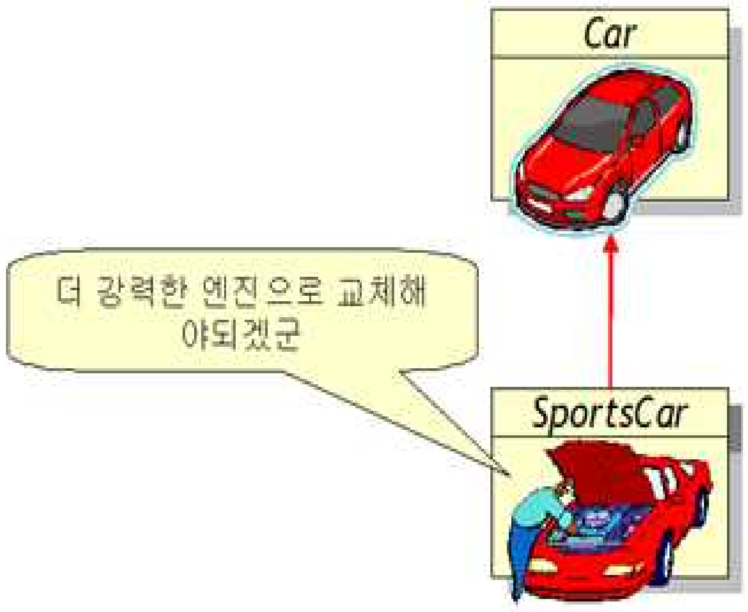
```
public void getCab (String cabbieName);
public int getCab (String cabbieName);
```

[참고] Overriding Methods

- 서브클래스가 필요에 따라 상속된 메소드를 재정의하는 것
  - **주의 사항** : 메소드 이름, 반환형, 매개변수의 개수와 데이터타입이 수퍼클래스에 있는 메소드와 일치해야 함.

```
class Car {
    ...
    public double speedUp(int upSpeed)
     {
        speed += upSpeed;
        if (speed > 120) speed = 120;
     }
}
```

```
class SportsCar extends Car {
    ...
    public double speedUp(int upSpeed)
   {
        speed += upSpeed;
        if (speed > 250) speed = 250;
     }
}
```



메소드 재정의가 잘못된 예

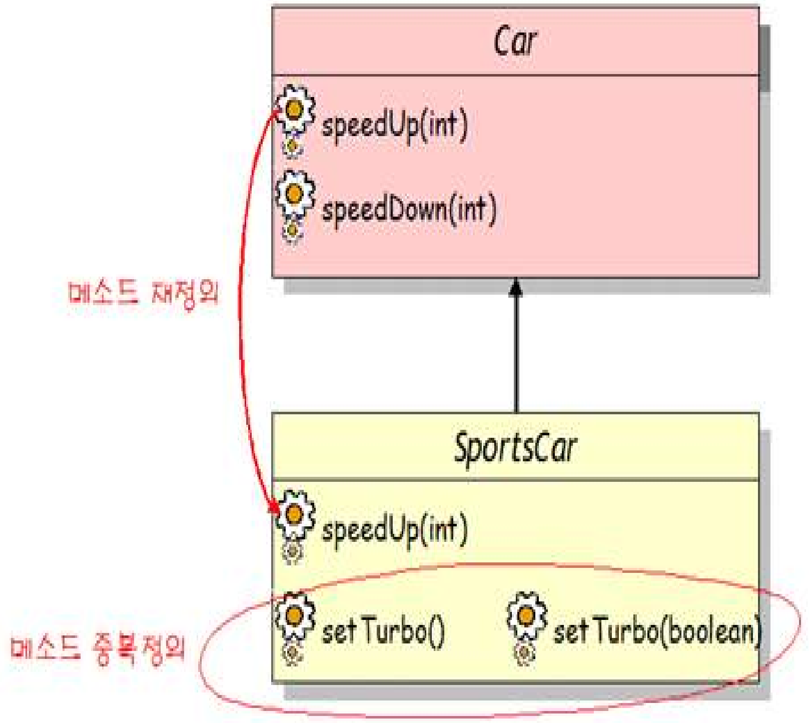
```
public class Animal {
		public void makeSound()
		{
		}
};
```

메소드 재정의가 아님(반환형이 다름)

```
public class Dog extends Animal {
		public int makeSound()
		{
		}
};
```

Overriding vs. Overloading

- 메소드 재정의(overriding)
  - 수퍼클래스로부터 상속받은 메소드를 서브클래스에서 자신의 용도에 맞게 다시 정의하는 것
- 메소드 중복정의(overloading)
  - 한 클래스 내에서 이름은 같으나, 매개변수의 개수, 타입, 순서 등이 다른 메소드를 2개 이상 정의하는 것



The Superclass

When using inheritance, you must know how the parent class is constructed.

- Inside the constructor, the constructor of the class’s superclass is called.
- Each class attribute of the object is initialized.
- The rest of the code in the constructor executes.

Designing Constructors

It is good practice to initialize all the attributes.

- In some languages, the compiler provides some sort of initialization.
- As always, don't count on the compiler to initialize attributes!
- Constructors are used to ensure that the application is in a stable (or safe) state.

Using UML to Model Classes

E.g. DatabaseReader Constructor:

- Pass the name of the database and position the cursor at the beginning of the database.
- Pass the name of the database and the position within the database where we want the cursor to position itself.

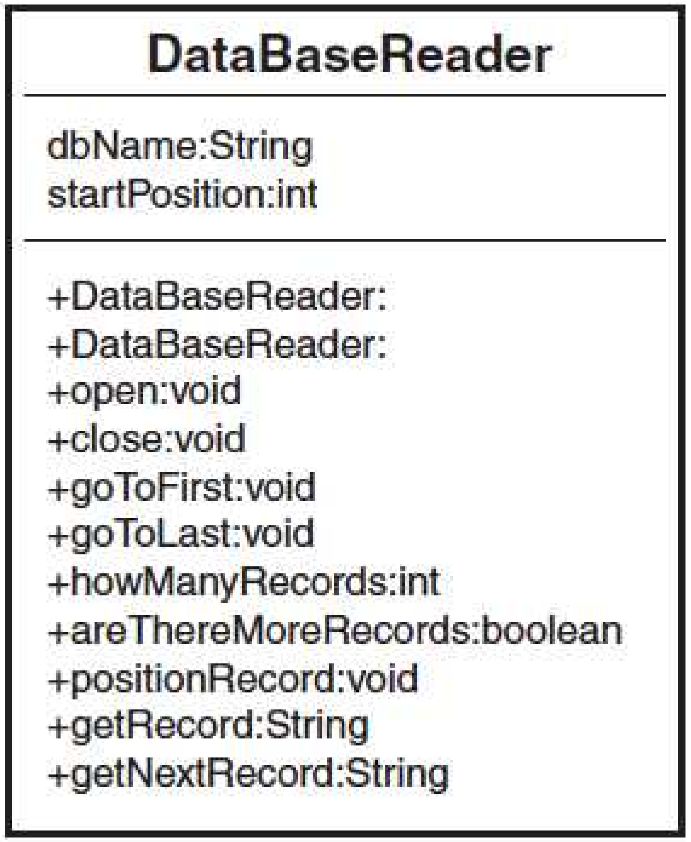


Figure 3.2 The **DataBaseReader** class diagram.

```
public class DataBaseReader {
        String dbName;
        int startPosition;

        // initialize just the name
        public DataBaseReader (String name){
                dbName = name;
                startPosition = 0;
        };

        // initialize the name and the position
        public DataBaseReader (String name, int pos){
                dbName = name;
                startPosition = pos;
        };
  .. // rest of class
}
```

How the Superclass Is Constructed

- When you use inheritance, you are inheriting everything about the parent.

Constructing an Object

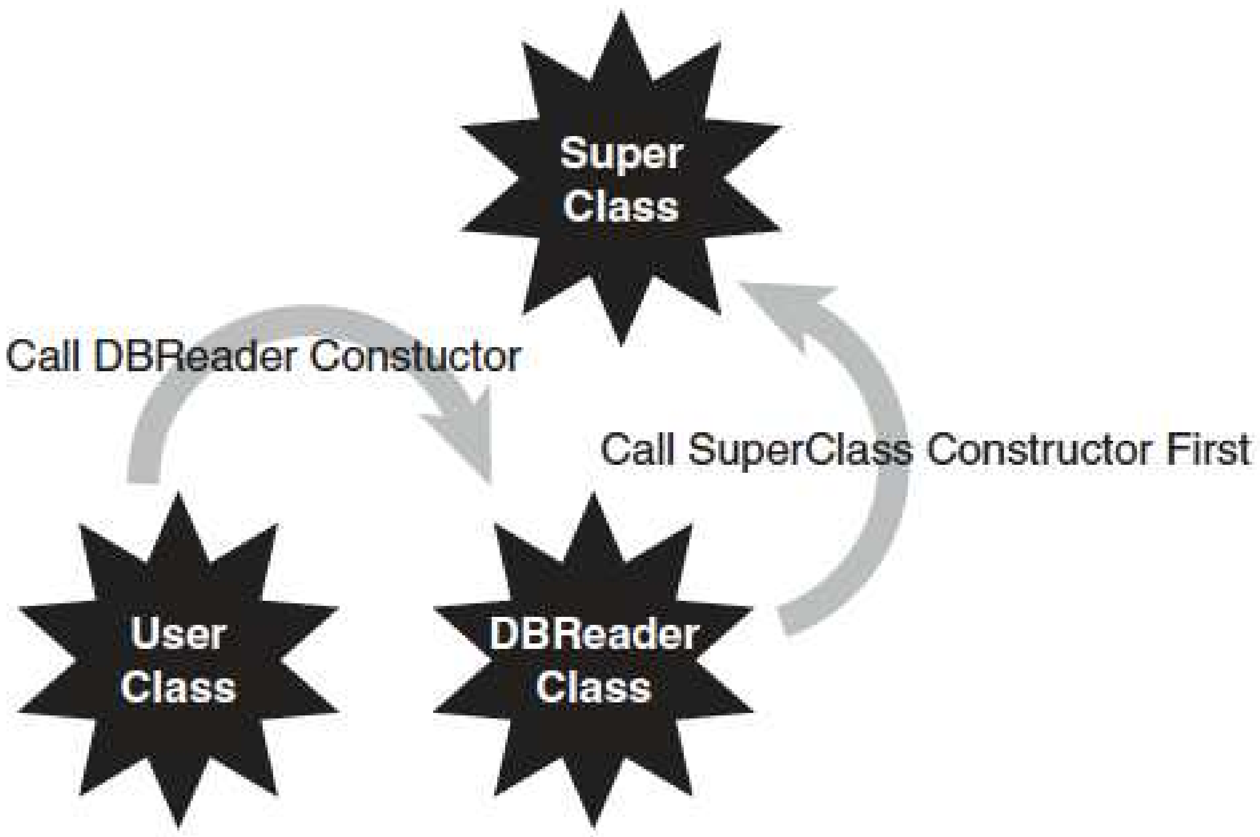
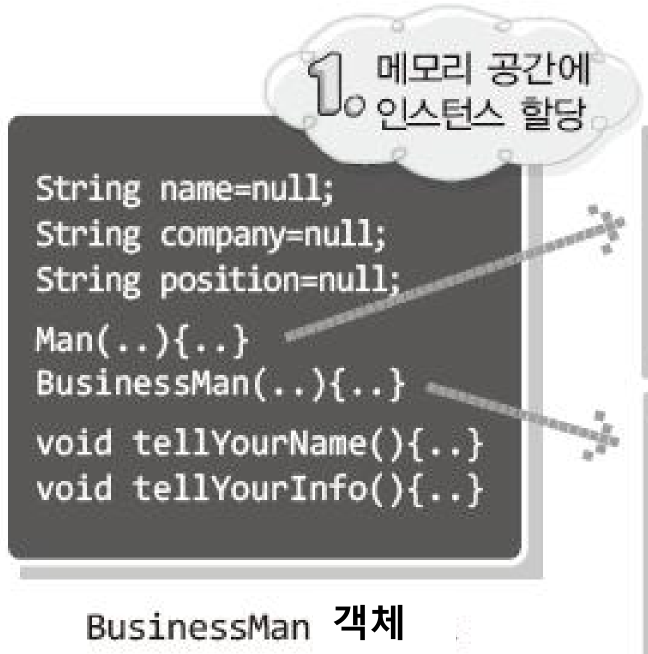


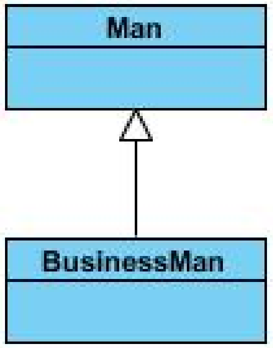
Figure 3.4 Constructing an object.

[참고]상속관계에 있는 객체 생성 과정1



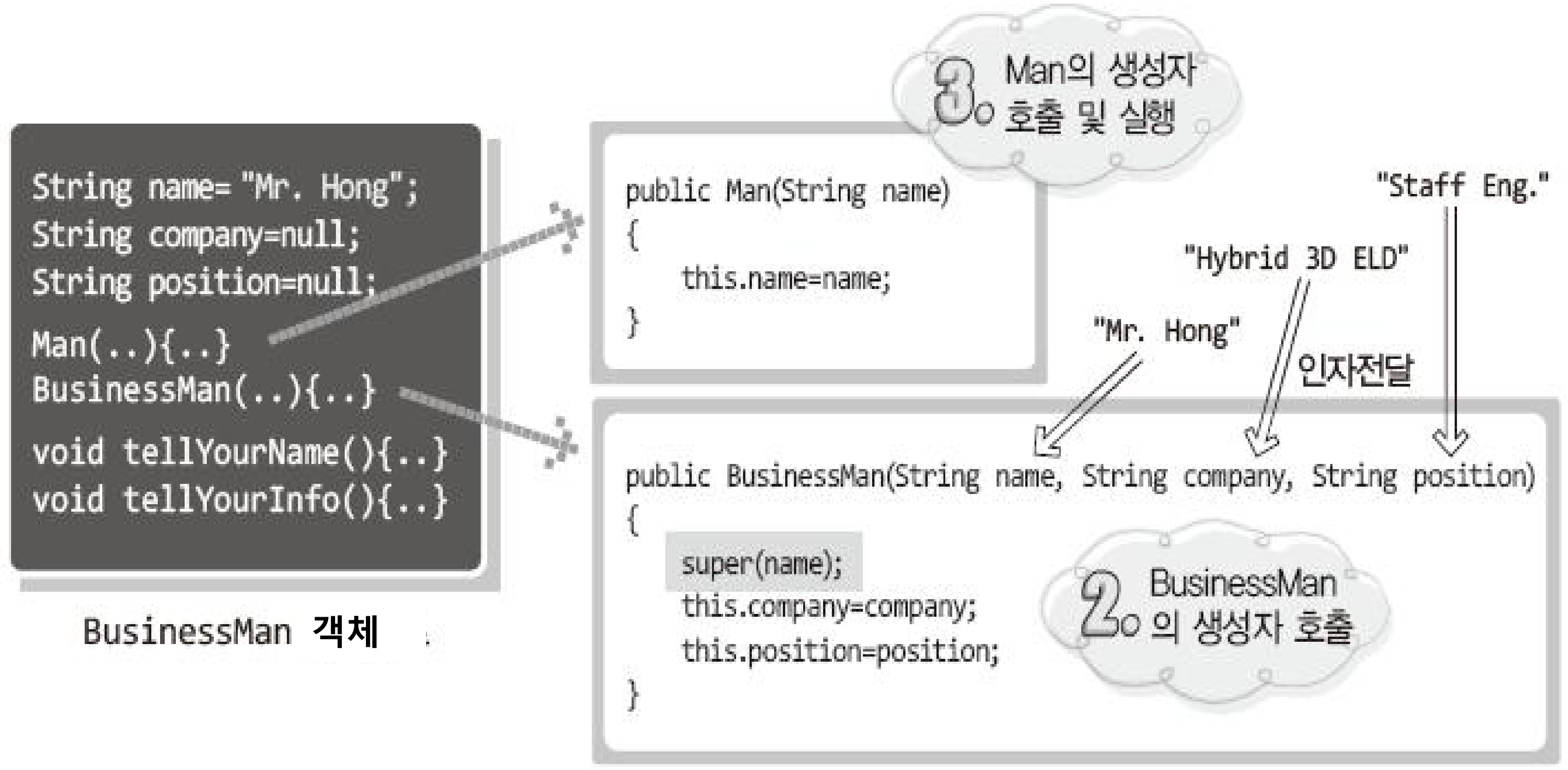
```
public Man(String name)
{
     this.name=name;
}
```

```
public BusinessMan(String name, String company, String position)
{
     super(name);
     this.company=company;
     this.position=position;
}
```



```
public static void main(String[] args) {
     BusinessMan man1
          = new BusinessMan("Mr. Hong", "Hybrid 3D ELD", "Staff Eng.");
     …
}
```

[참고] 상속관계에 있는 객체 생성 과정2~3



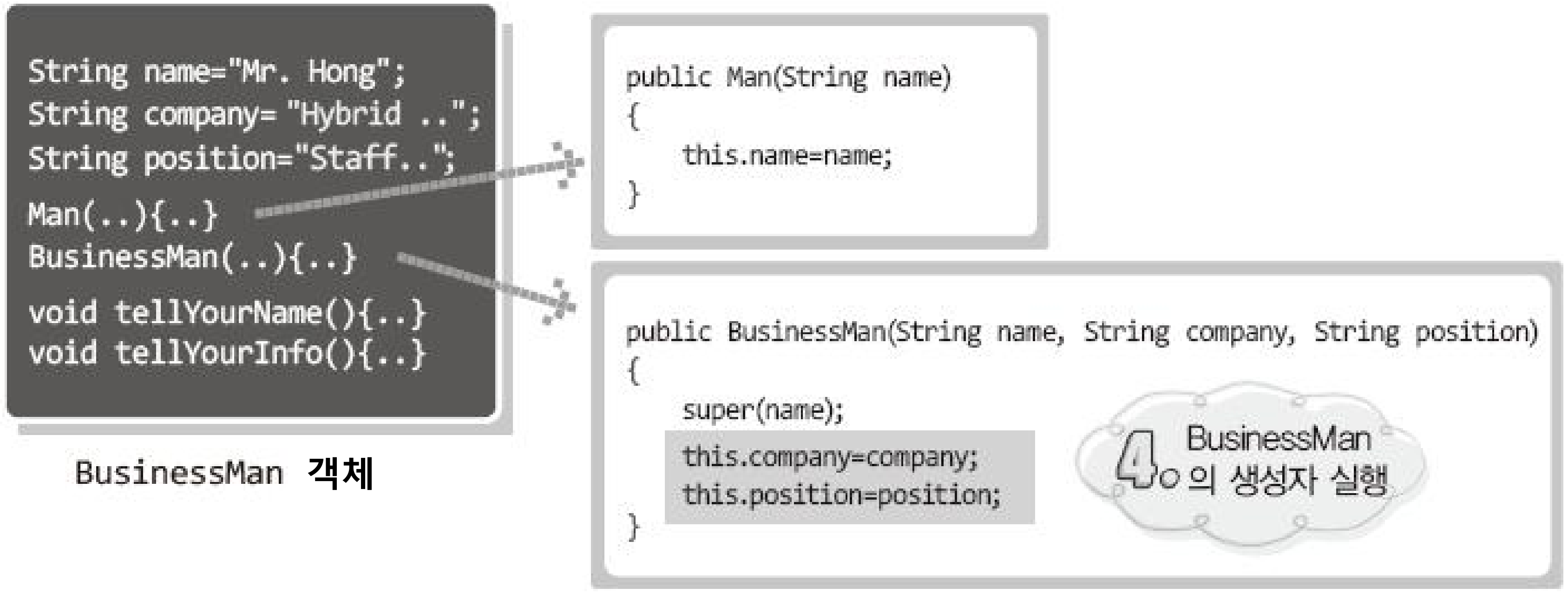
```
String name= "Mr. Hong";
String company=null;
String position=null;
Man(..){..}
BusinessMan(..){..}
void tellYourName(){..}
void tellYourInfo(){..}
```

BusinessMan 객체

```
public Man(String name)
{
    this.name=name;
}
```

```
public BusinessMan(String name, String company, String position)
{
    super(name);
    this.company=company;
    this.position=position;
}
```

[참고] 상속관계에 있는 객체 생성 과정4



```
String name="Mr. Hong";
String company= "Hybrid ..";
String position="Staff..";
Man(..){..}
BusinessMan(..){..}
void tellYourName(){..}
void tellYourInfo(){..}
```

BusinessMan 객체

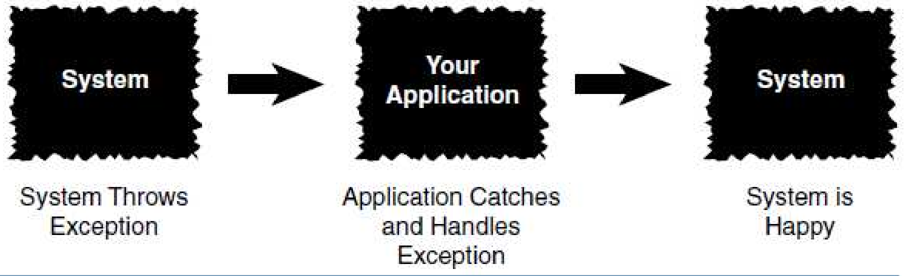
```
public Man(String name)
{
    this.name=name;
}
```

```
public BusinessMan(String name, String company, String position)
{
    super(name);
    this.company=company;
    this.position=position;
}
```

Error Handling

Assuming that your code has the capability to detect and trap an error condition, you can handle the error in several ways:

- Ignore the problem—not a good idea!
- Check for potential problems and abort the program when you find a problem.
- Check for potential problems, catch the mistake, and attempt to fix the problem.
- Throw an exception.



Ignore the problem

- No way!!!

Check for problems and abort the program

When a problem is detected, the application can display a message indicating that there is a problem.

- This does NOT allow the system to clean up things and put itself in a more stable state, such as closing files.

Checking for problems and attempting to recovery

```
c = b/a;
```

- Exception with a==0

```
if (a == 0) a=1;
c = b/a;
```

- No crash, but not proper solution

Throw an exception

- Exceptions provide a way to detect problems and then handle them.
  - In Java, C# and C++, exceptions are handled by the keywords catch and throw.
- Here is the structure for a try/catch block:

```
try {
    // possible nasty code
} catch(Exception e) {
    // code to handle the exception
}
```

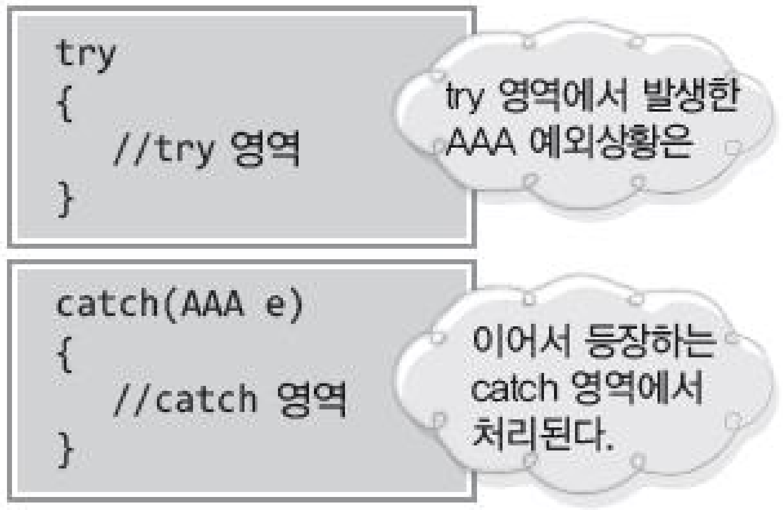
If an exception is thrown within the try block, the catch block will handle it. When an exception is thrown while the block is executing, the following occurs:

- 1. The execution of the try block is terminated.
- 2. The catch clauses are checked to determine whether an appropriate catch block for the offending exception was included. (There might be more than one catch clause per try block.)

- 3. If none of the catch clauses handle the offending exception, it is passed to the next higher-level try block. (If the exception is not caught in the code, the system ultimately catches it, and the results are unpredictable, i.e., an application crash.)
- 4. If a catch clause is matched (the first match encountered), the statements in the catch clause are executed.
- 5. Execution then resumes with the statement following the try block.

try/catch 블록

- 예외를 처리할 것임을 알려주기 위한 용도로 쓰이는 구문
  - try - 예외발생의 감지 대상을 감싸는 목적으로 사용
  - catch - 발생한 예외상황의 처리를 위한 목적으로 사용
  - try 블록과 catch 블록은 독립된 블록 – try 블록에서 정의된 변수는 catch 블록에서 사용될 수 없음

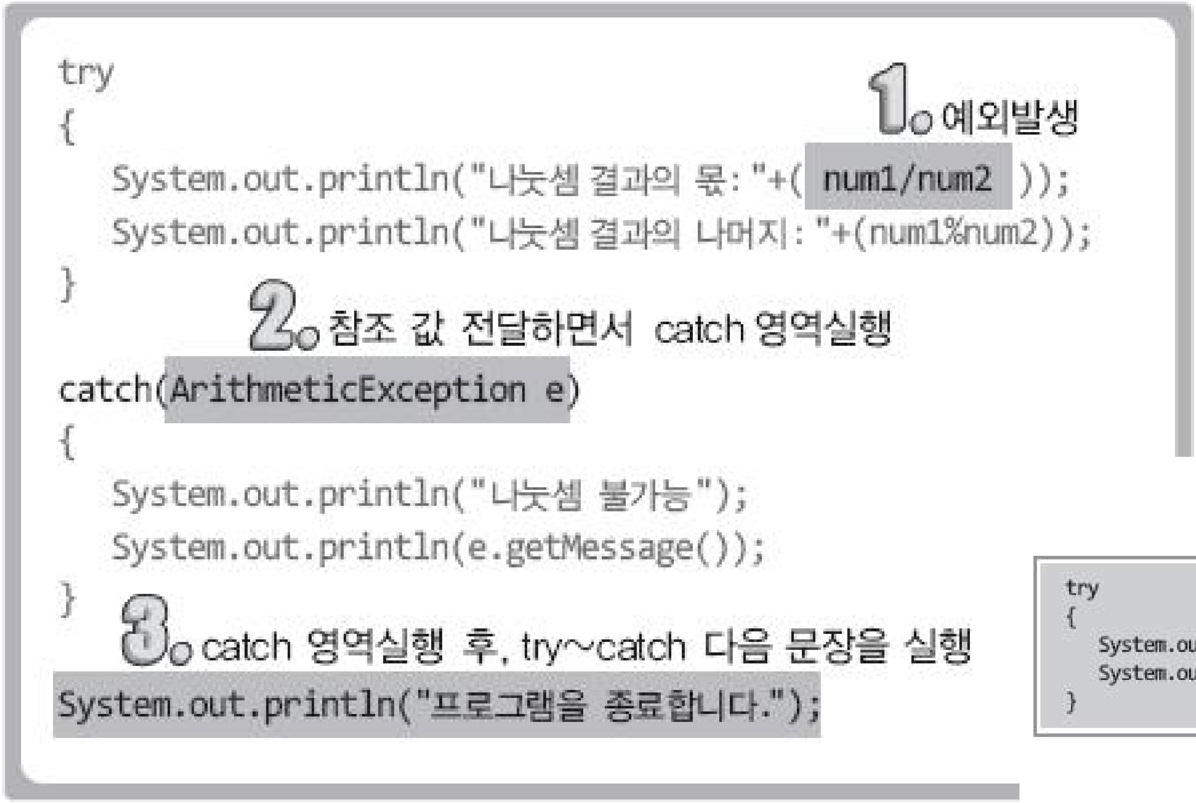


```
try
{
    //try 영역
}
```

```
catch(AAA e)
{
    //catch 영역
}
```

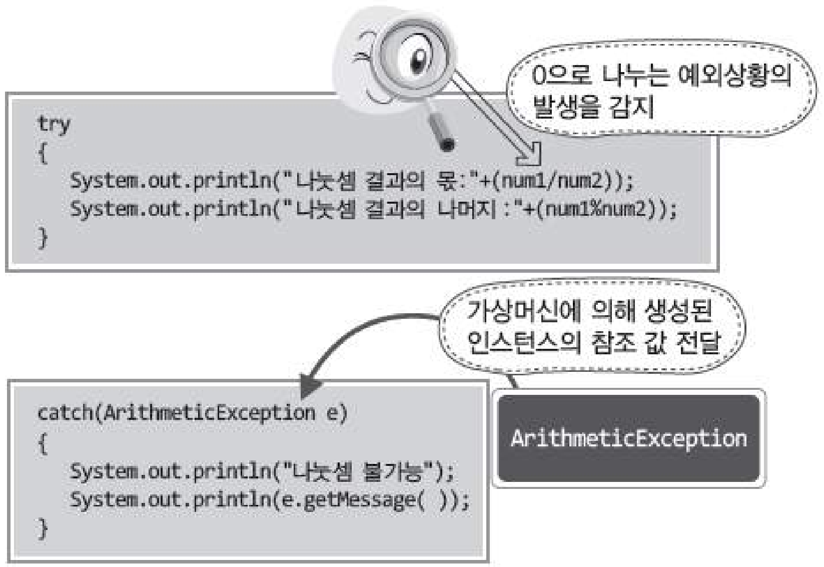
try 영역에서 발생한 AAA 예외상황은
이어서 등장하는 catch 영역에서 처리된다.

try/catch 블록의 예외처리 과정



```
try
{
    System.out.println("나눗셈 결과의 몫: "+( num1/num2 ));
    System.out.println("나눗셈 결과의 나머지: "+(num1%num2));
}
catch(ArithmeticException e)
{
    System.out.println("나눗셈 불가능");
    System.out.println(e.getMessage());
}
System.out.println("프로그램을 종료합니다.");
```

1. 예외발생
2. 참조 값 전달하면서 catch 영역실행
3. catch 영역실행 후, try~catch 다음 문장을 실행



```
try
{
    System.out.println("나눗셈 결과의 몫:"+(num1/num2));
    System.out.println("나눗셈 결과의 나머지 :"+(num1%num2));
}
```

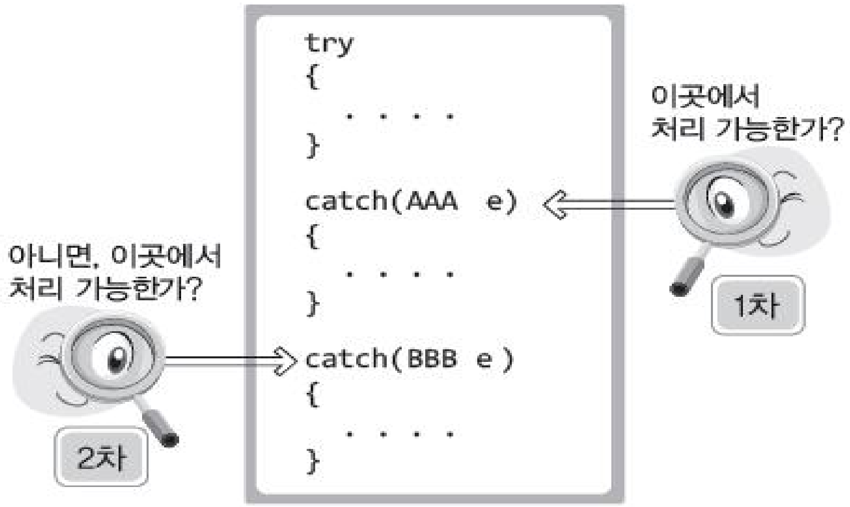
```
catch(ArithmeticException e)
{
    System.out.println("나눗셈 불가능");
    System.out.println(e.getMessage( ));
}
```

0으로 나누는 예외상황의 발생을 감지
가상머신에 의해 생성된 인스턴스의 참조 값 전달
ArithmeticException

catch 블록이 여러 개인 경우

- 예외의 종류에 따라 여러 개의 catch 블록 정의 가능
- 발생한 예외의 종류와 일치하는 catch 블록만 실행됨
- 일치되는 catch 블록이 없으면 발생된 예외는 처리되지 않음

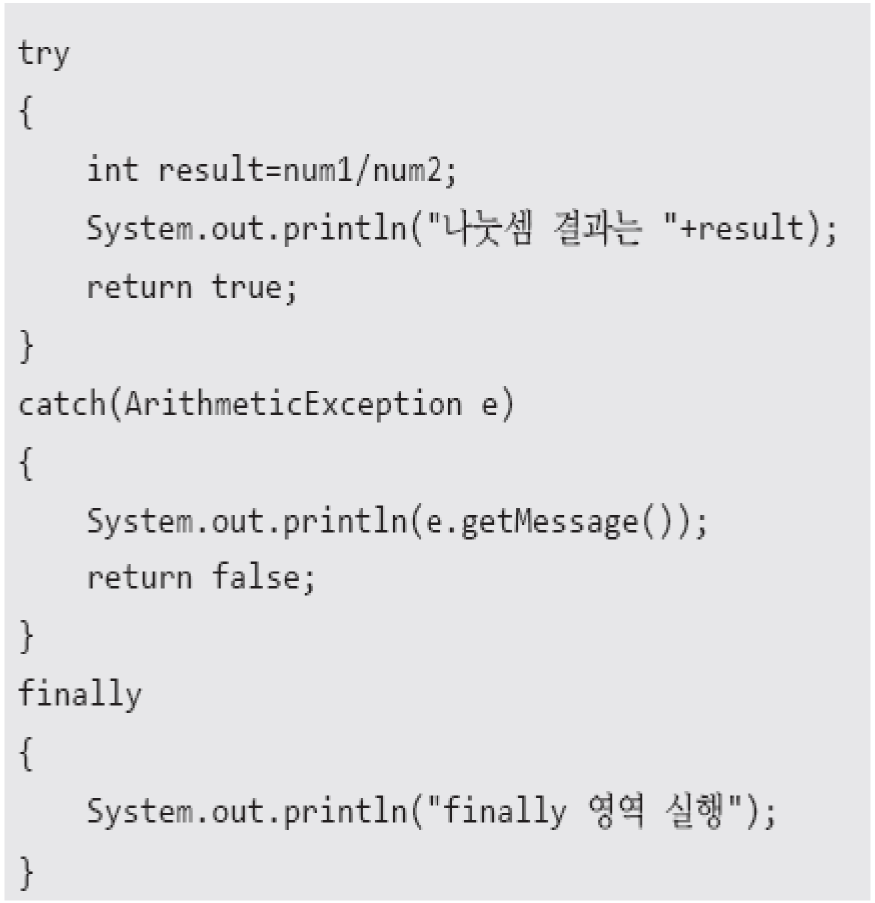
```
try
{
    . . . .
}
catch(Throwable e)
{
    . . . .
}
catch(ArithmeticException e)
{
    . . . .
}
```



Finally 블록

- finally와 연결되어 있는 try 블록으로 일단 진입을 하면, 무조건 실행되는 영역

- 중간에 return 문을 실행하더라도 finally 블록이 실행된 다음에 메소드를 빠져 나감



```
try
{
    int result=num1/num2;
    System.out.println("나눗셈 결과는 "+result);
    return true;
}
catch(ArithmeticException e)
{
    System.out.println(e.getMessage());
    return false;
}
finally
{
    System.out.println("finally 영역 실행");
}
```

```
try {
      // possible nasty code
      count = 0;
      count = 5/count;
} catch(ArithmeticException e) {
      // code to handle the exception
      System.out.println(e.getMessage());
      count = 1;
}
System.out.println("The exception is handled.");
```

The Concept of Scope

- Multiple objects can be instantiated from a single class.
  - Each of these objects has a unique identity and state.
  - Each object is constructed separately and is allocated its own separate memory.

Types of Scope

- Methods represent the behaviors of an object; the state of the object is represented by attributes.
  - Local attributes
  - Object attributes
  - Class attributes

Local Attributes

Local attributes are owned by a specific method

- Local variables are accessible only inside a specific method.
- In Java, C#, C++ and Objective-C, scope is delineated by curly braces ({ }).

```
public method() {
    int count;
}
```

```
public class Number {
        public method1() {
                int count;
        }
        public method2() {
        }
}
```

- The method method1 contains a local variable called count. This integer is accessible only inside method1.

```
public class Number {
        public method1() {
                int count;
        }
        public method2() {
                int count;
        }
}
```

```
method1.count;
method2.count;
```

- In this example, there are two copies of an integer count in this class.
- Differentiated attribute – method1 and method2 each has its own scope.

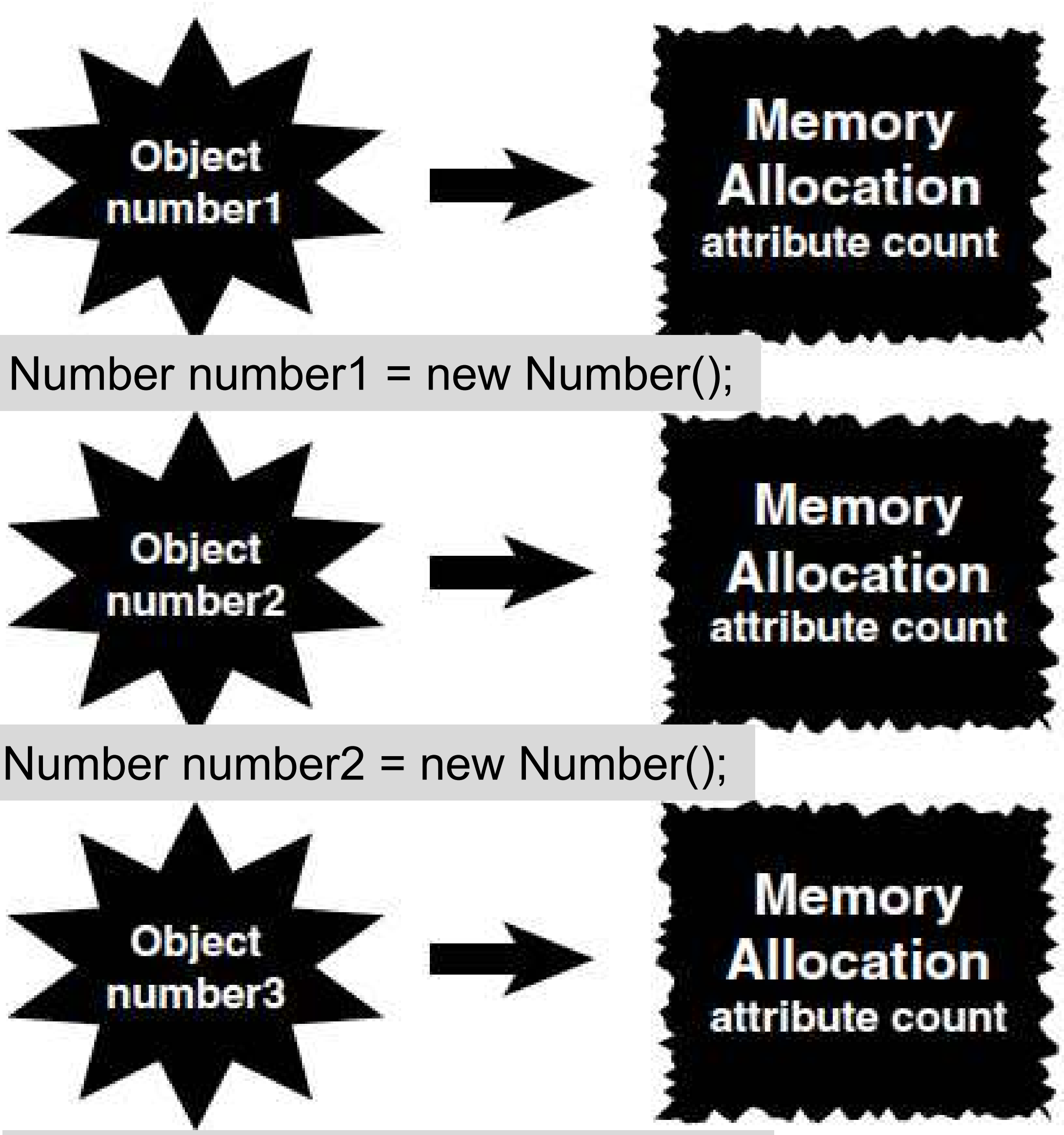
Object Attributes

In many design situations, an attribute must be shared by several methods within the same object.

```
public class Number {
    int count; // available to both method1 and method2
    public method1() {
        count = 1;
    }
    public method2() {
        count = 2;
    }
}
```

- The attribute `count` is declared outside the scope of both `method1` and `method2`.
- However, it is within the scope of the class.

Object Attributes


Object number1
Memory Allocation attribute count
Object number2
Memory Allocation attribute count
Object number3
Memory Allocation attribute count

```
Number number1 = new Number();
Number number2 = new Number();
Number number3 = new Number();
```

Figure 3.6 Object attributes.

- To create three copies of the Number class:

```
Number number1 = new Number();
Number number2 = new Number();
Number number3 = new Number();
```

- Each of these objects—number1 , number2 , and number3 —is constructed separately and is allocated its own resources.

```
public method1() {
        int count; // local variable
        this.count = 1; // object variable
}
```

- The use of the this keyword directs the compiler to access the object variable count and not the local variables within the method bodies.
- The keyword this is a reference to the current object.

Class Attributes

It is possible for two or more objects to share attributes. In Java, C#, C++ and Objective-C, you do this by making the attribute *static:*

```
public class Number {
    static int count;
    public method1() {
    }
}
```

- For class attributes, you must be aware of potential synchronization problems.

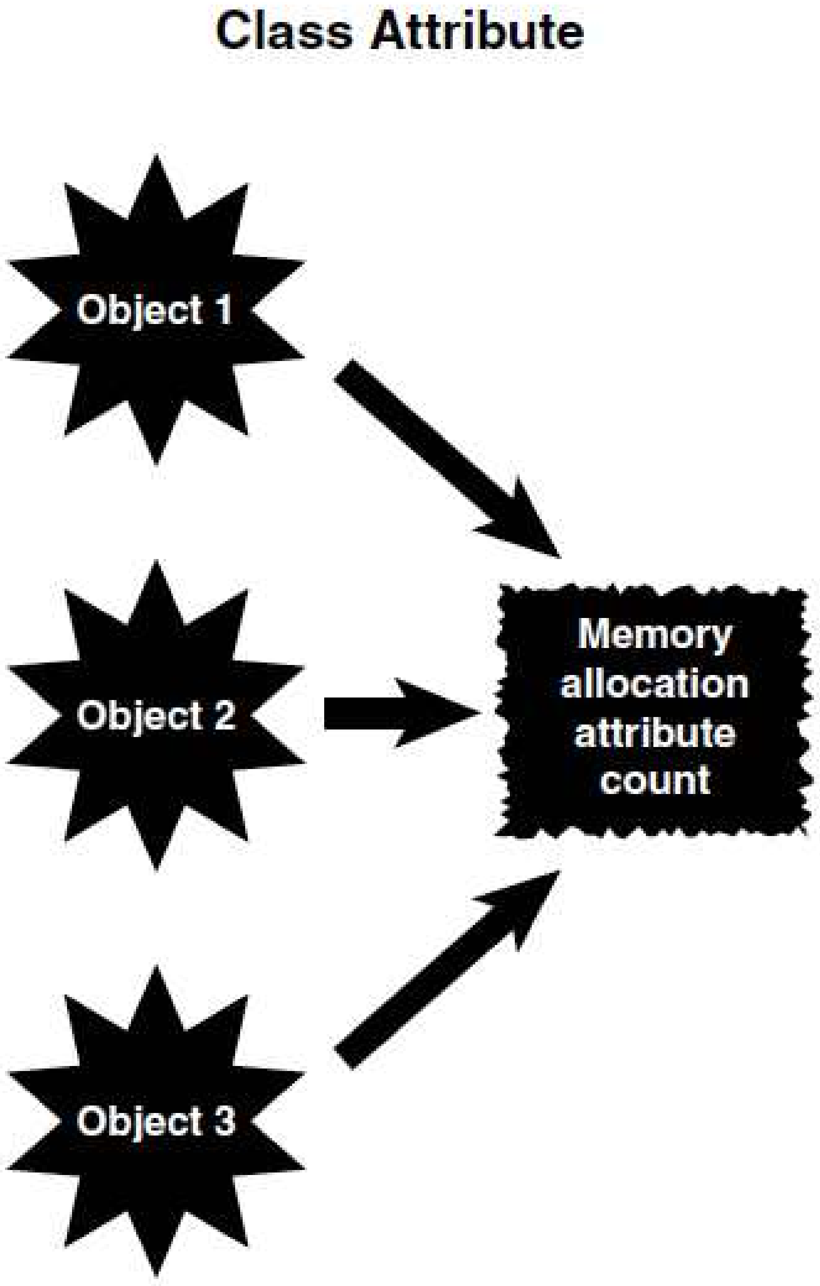
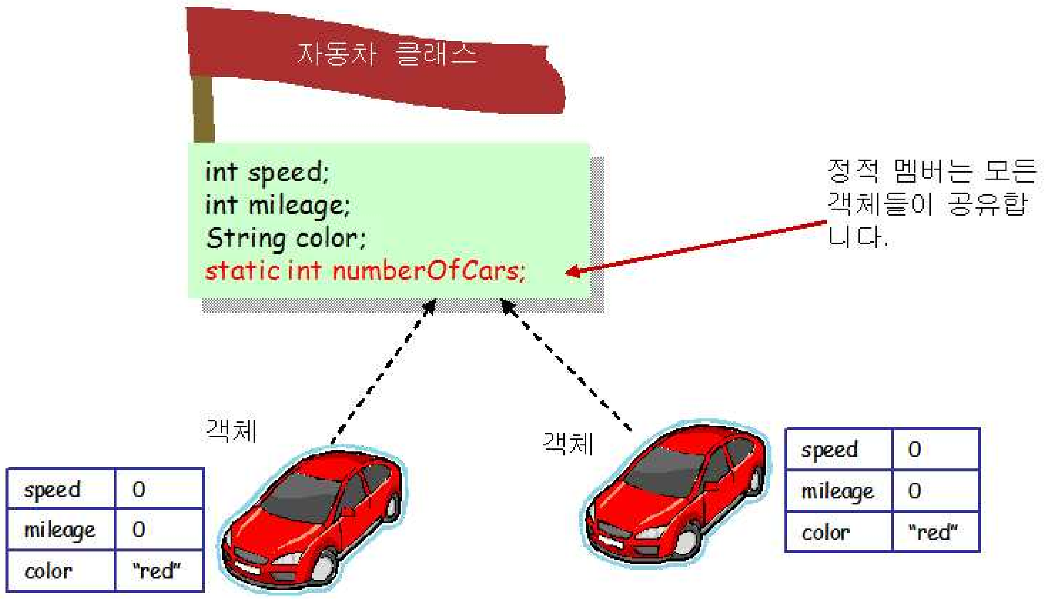


Figure 3.7 Class attributes.

[참고] 정적 변수(Static Variable)

- 인스턴스 변수(instance variable)
  - 객체마다 하나씩 있는 변수(object attribute)
- 정적 변수(static variable) – 클래스 변수(class attribute)
  - 클래스에 속하는 모든 객체를 통틀어서 하나만 있는 변수
  - 클래스에 속하는 모든 객체가 공유해서 사용하는 변수
  - 객체가 아니라 클래스 내에 static 변수를 위한 기억공간이 생성됨.



[참고] 정적 변수의 예

Car.java

```java
public class Car {
        private int speed;
        private int mileage;
        private String color;

        // 자동차의 시리얼 번호
        private int id;

        // 실체화된 Car 객체의 개수를 위한 정적 변수
        private static int numberOfCars = 0;

        public Car(int s, int m, String c) {
                speed = s;
                mileage = m;
                color = c;

                // 자동차의 개수를 증가하고 id 번호를 할당한다.
                id = ++numberOfCars;
        }
}
```

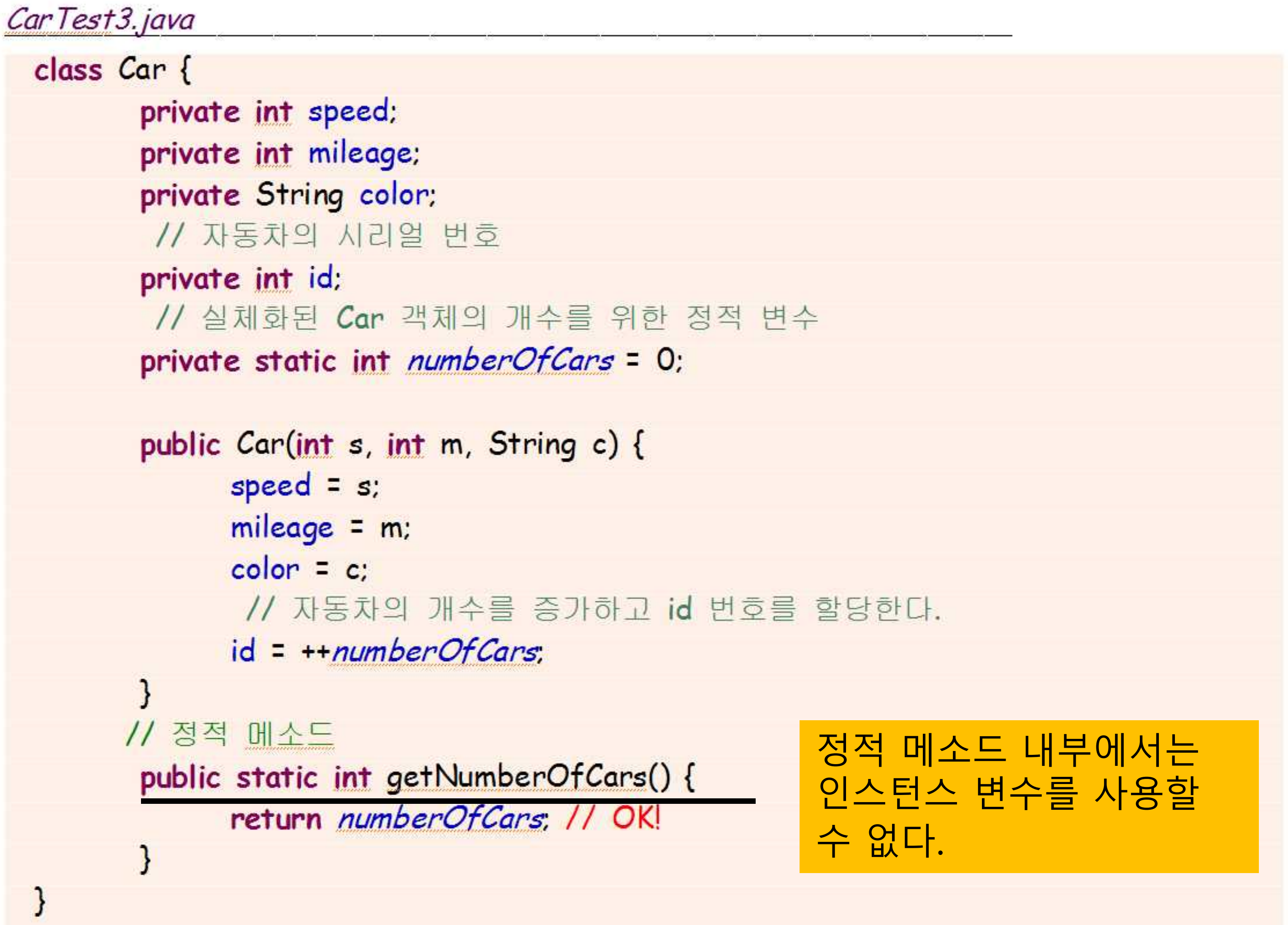
PEARSON

[참고] 정적 메소드 (Static Method)

- 정적 메소드(static method): 객체를 생성하지 않고 사용할 수 있는 메소드 – 클래스 메소드
  - (예) Math 클래스에 들어 있는 각종 수학 메소드 들

```
double value = Math.sqrt(9.0);
```

[참고] 정적 메소드의 예



CarTest3.java

```
class Car {
        private int speed;
        private int mileage;
        private String color;
         // 자동차의 시리얼 번호
        private int id;
         // 실체화된 Car 객체의 개수를 위한 정적 변수
        private static int numberOfCars = 0;

        public Car(int s, int m, String c) {
                speed = s;
                mileage = m;
                color = c;
                 // 자동차의 개수를 증가하고 id 번호를 할당한다.
                id = ++numberOfCars;
        }
       // 정적 메소드
        public static int getNumberOfCars() {
                return numberOfCars; // OK!
        }
}
```

정적 메소드 내부에서는 인스턴스 변수를 사용할 수 없다.

PEARSON

[참고] 정적 메소드의 예

```java
public class CarTest3 {
        public static void main(String args[]) {
                Car c1 = new Car(100, 0, "blue");          // 첫 번째 생성자 호출
                Car c2 = new Car(0, 0, "white");  // 첫 번째 생성자 호출
                int n = Car.getNumberOfCars();   // 정적 메소드 호출
                System.out.println("지금까지 생성된 자동차 수 = " + n);
        }
}
```

실행결과

```
지금까지 생성된 자동차 수 = 2
```

Operator Overloading

Some OO languages allow you to overload an operator.

- C++ is an example of one such language. Operator overloading allows you to change the meaning of an operator.
- More recent OO languages like Java, .NET, and Objective-C do not allow operator overloading.
    - Due to “confusion”

- E.g., plus sign (+)

```
x = 5 + 6;

Matrix a, b, c;
c = a + b;
```

C++에서 a.operator+(b) 로 바뀌어서 처리됨.
Matrix 클래스에 operator+(Matrix m) 정의 필요

```
Matrix operator+(Matrix m) {
     Matrix result;
    // 두 Matrix 를 더해서 result에 저장하는 코드
    return result;
}
```

```
String firstName = "Joe", lastName = "Smith";
String Name = firstName + " " + lastName;
```

  - Arithmetic addition – integer & Matrix
  - String concatenation – two separate strings are combined to create a new, single string.

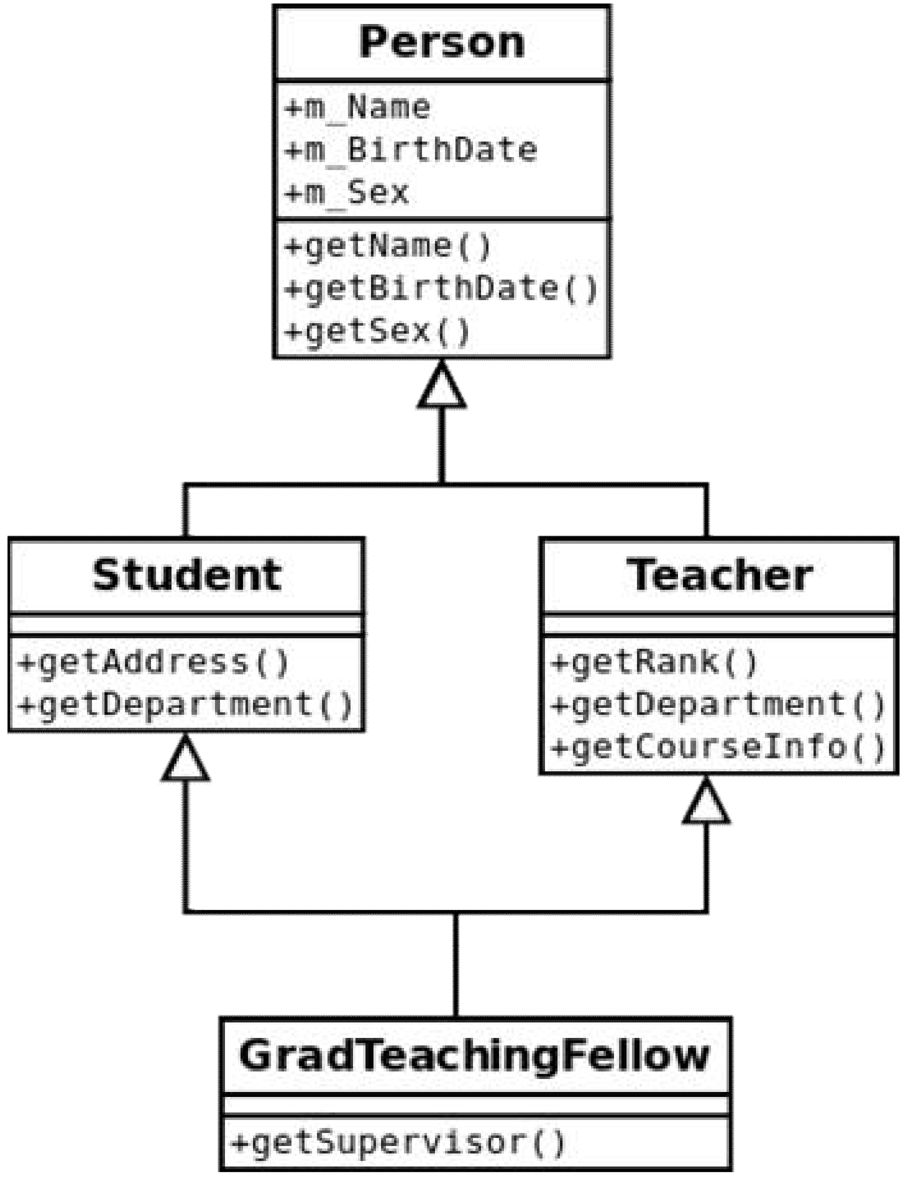
Multiple Inheritance

Multiple inheritance allows a class to inherit from more than one class.

- Multiple inheritance can significantly increase the complexity of a system,
- Java, .NET, and Objective-C do not support multiple inheritance (C++ does).
- In some ways interfaces compensates for this.

Class Person, Student, Teacher, and GradTeachingFellow

- GradTeachingFellow is inherited from both attributes of Student and Teacher.



Object Operations

Comparing primitive data types is quite straightforward.

- Copying and comparing objects is not quite as simple.
- The problem with complex data structures and objects is that they might contain references.

- Simply making a copy of the reference does not copy the data structures or the object that it references.

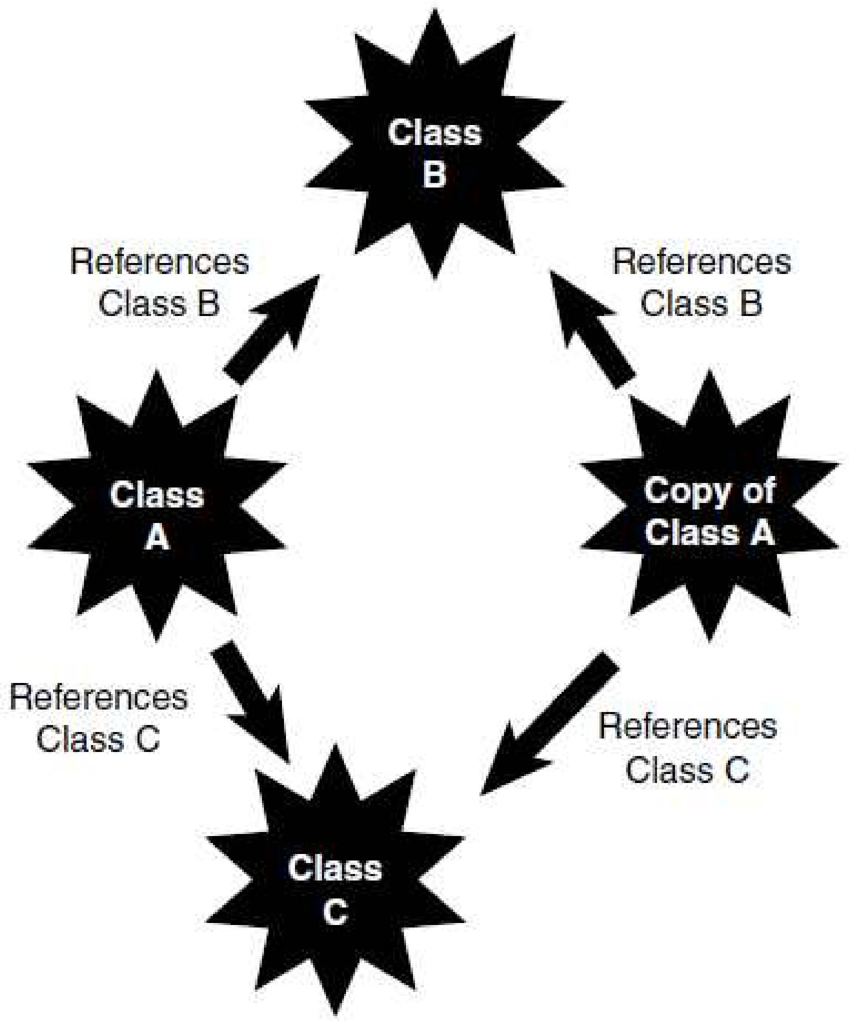
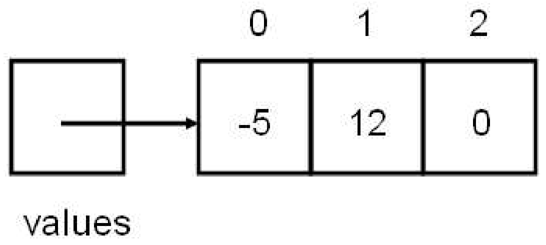


Figure 3.8 Following object references.

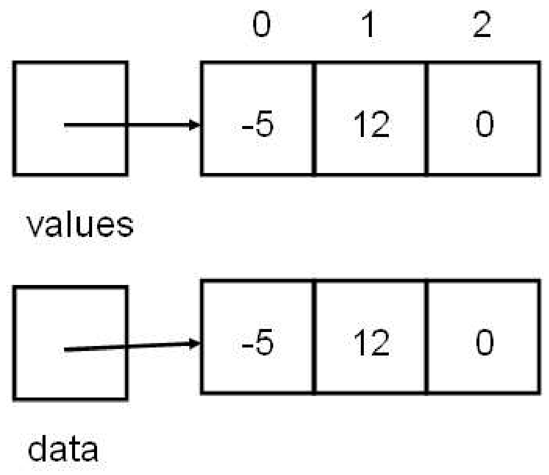
Deep Versus Shallow Copies

- A deep copy is when all the references are followed and new copies are created for all referenced objects.
- A shallow copy would simply copy the reference and not follow the levels.

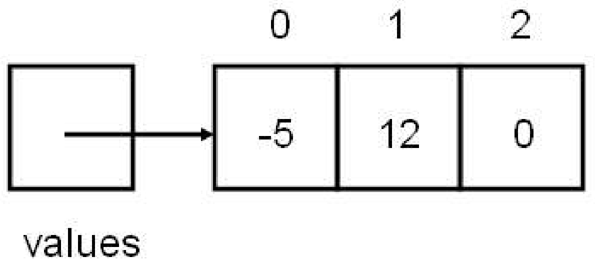
Deep copy



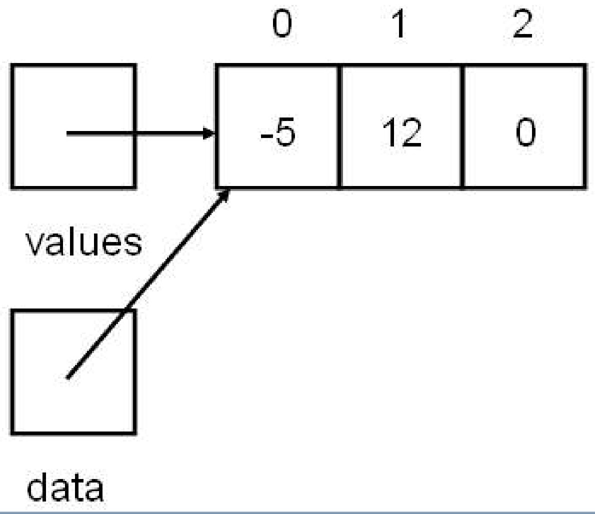
```
//code for deep copy
```



Shallow copy



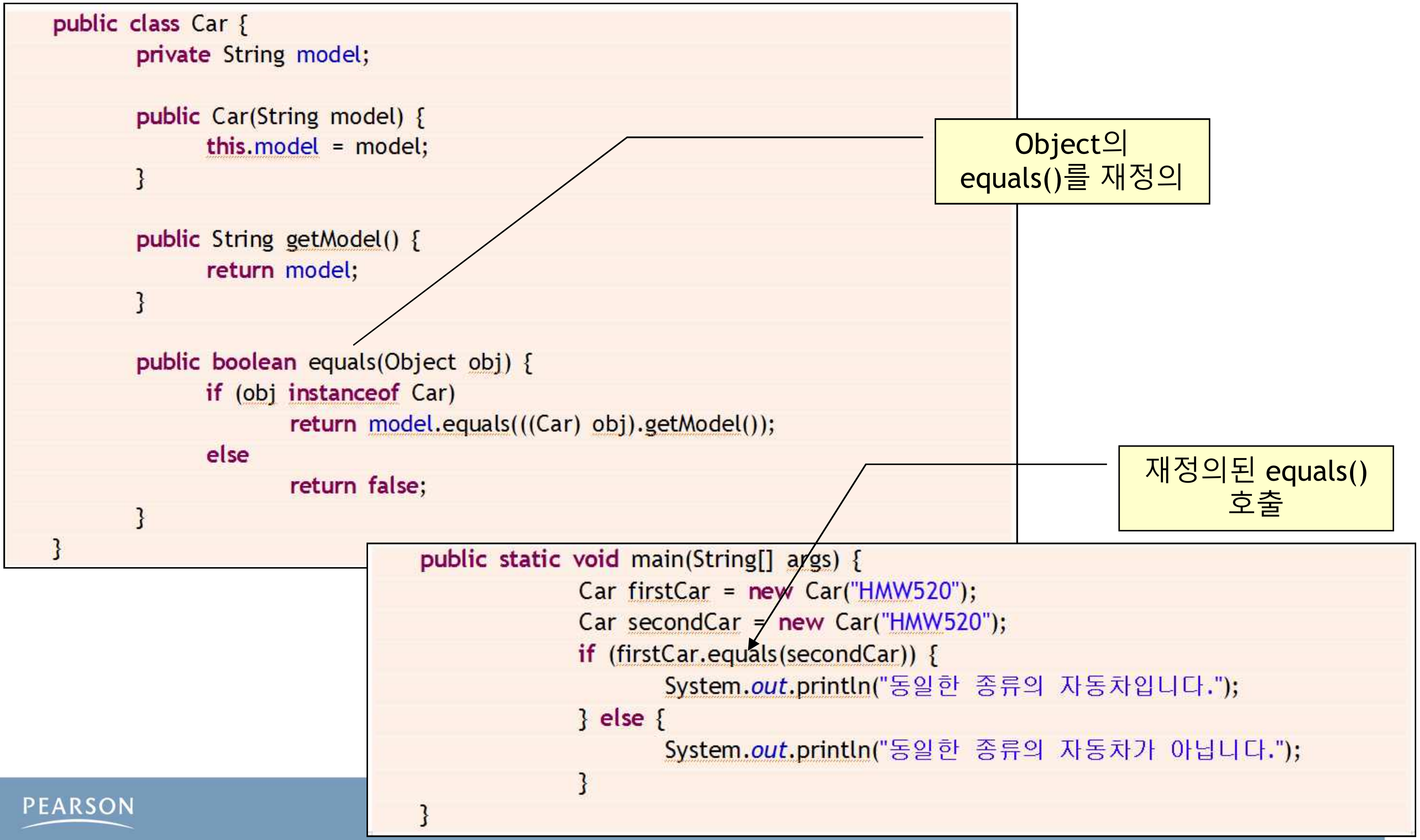
```
data = values;
```



Comparing Objects : equals()

- 자바에서 == 연산자는 참조값 비교를 함.
  - 기초 자료형의 경우에는 올바른 결과를 생성
  - 객체에 대해서는 객체 참조값이 같은지를 검사 ⇒ 비교되는 객체가 동일한 객체인지 검사
- 객체 간 내용 비교를 위해서는 내용 비교 기능의 메소드가 필요.
- 자바에서는 인스턴스간의 내용 비교를 목적으로 Object 클래스에 equals 메소드를 정의해 놓음.
- 따라서 새로 정의되는 클래스의 내용 비교가 가능하도록 이 메소드를 재정의 하는 것이 좋다!

equals() 메소드 재정의 예



```
public class Car {
        private String model;

        public Car(String model) {
                this.model = model;
        }

        public String getModel() {
                return model;
        }

        public boolean equals(Object obj) {
                if (obj instanceof Car)
                        return model.equals(((Car) obj).getModel());
                else
                        return false;
        }
}
```

```
public static void main(String[] args) {
                Car firstCar = new Car("HMW520");
                Car secondCar = new Car("HMW520");
                if (firstCar.equals(secondCar)) {
                        System.out.println("동일한 종류의 자동차입니다.");
                } else {
                        System.out.println("동일한 종류의 자동차가 아닙니다.");
                }
}
```