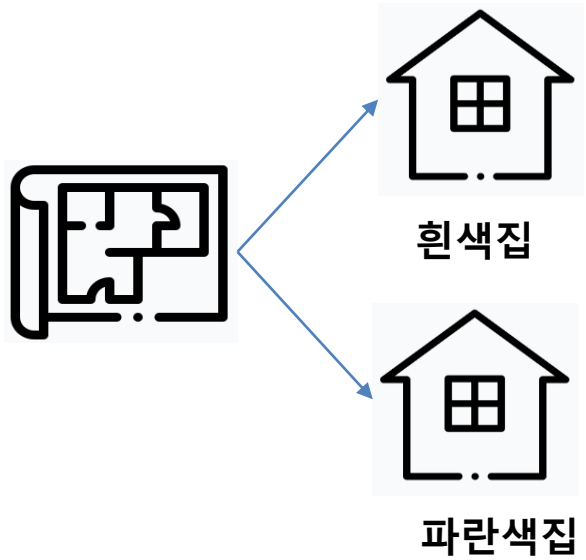
객체지향 프로그래밍

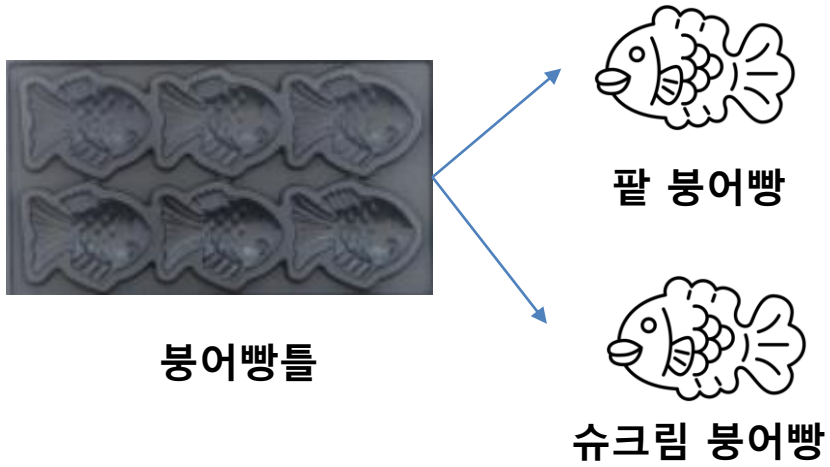
한림대학교 SW중심대학

객체

객체와 클래스

- 클래스
  - 객체를 만들어 내기 위한 설계도 혹은 틀
  - 연관되어 있는 필드와 메서드의 집합
- 객체
  - 설계도나 틀을 통해 실체화 것
  - 클래스에 선언한 필드 , 메소드를 생성화한 실체





객체지향 프로그래밍

```
public class BluePrint {

}
```

```
public class ObjectMain {

    public static void main(String[] args) {

        BluePrint whiteHouse = new BluePrint();
        BluePrint blueHouse = new BluePrint();
    }
}
```

객체지향 프로그래밍

```
public class BluePrint {
    public String wallColor;
    public String roofColor;

    public String getWallColor() {
        return wallColor;
    }
    public void setWallColor(String col) {
        this. wallColor = col;
    }
    public String getRoofColor() {
        return roofColor;
    }
    public void setRoofColor(String col) {
        this.roofColor = col;
    }
}
```

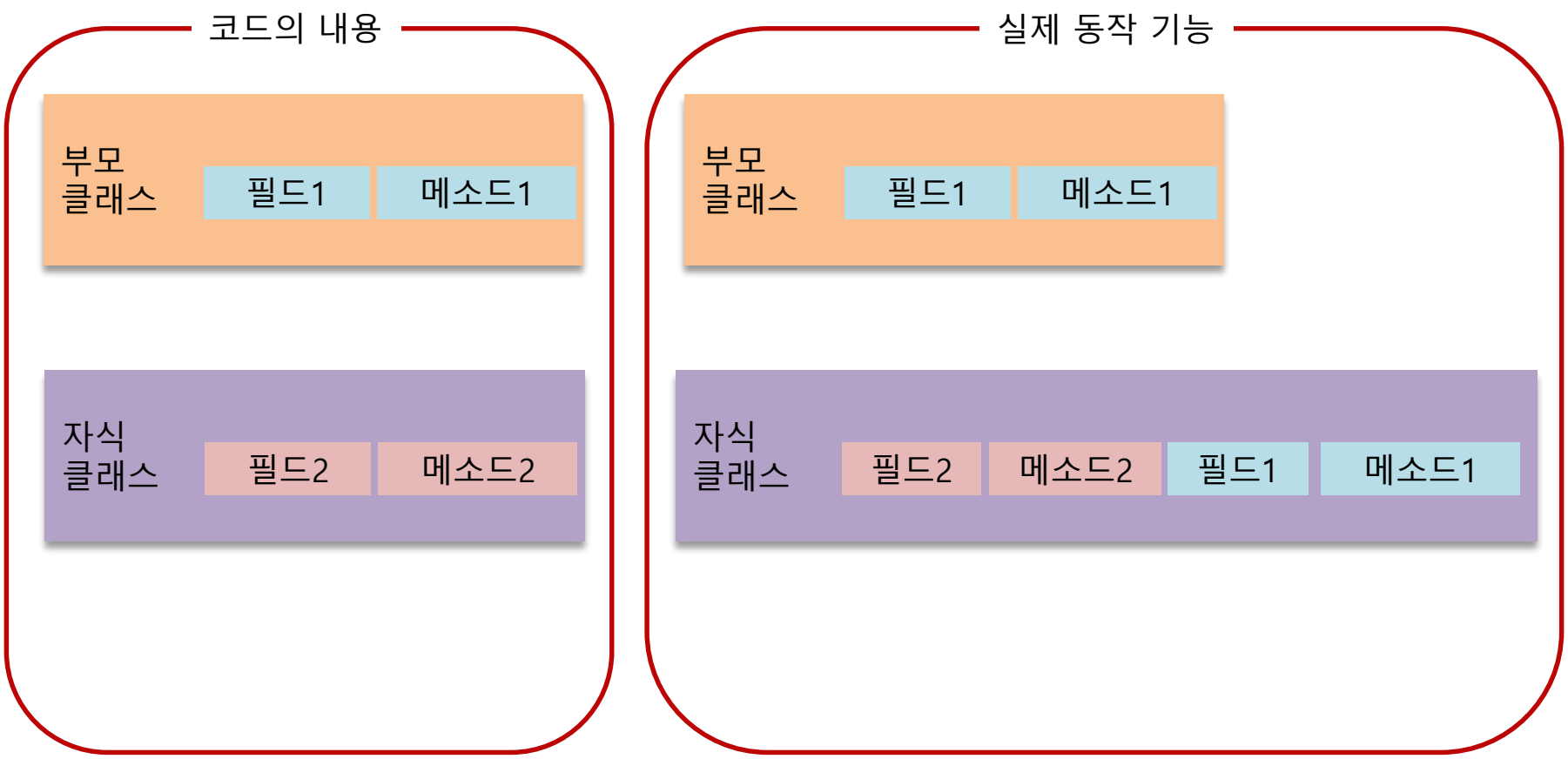
```
public class ObjectMain {

    public static void main(String[ ] args) {

        BluePrint whiteHouse = new BluePrint();
        whiteHouse.setWallColor("white");
        String wallCol=whiteHouse.getWallColor();
        whiteHouse.setRoofColor("brown");
        String roofCol=whiteHouse.getRoofColor();
        System.out.println(wallCol+","+roofCol);

        BluePrint blueHouse = new BluePrint();
        blueHouse.setWallColor("blue");
        wallCol=blueHouse.getWallColor();
        blueHouse.setRoofColor("black");
        roofCol=blueHouse.getRoofColor();
        System.out.println(wallCol+", "+roofCol);
    }
}
```

한림대학교 SW중심대학

상속

상속

- 상속
  - 상위(부모) 객체의 필드와 메소드를 하위(자식) 객체에게 물려주는 행위



상속

java.lang

Class Object

java.lang.Object

```
public class Object
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

See Also:

`Class`

java.lang

Class String

java.lang.Object
  java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

상속

슈퍼 클래스

```
public class BasicCalculator {

    public int add(int x, int y) {
        return x+y;
    }
    public int sub(int x, int y) {
        return x-y;
    }
    public int div(int x, int y) {
        return x/y;
    }
    public int mul(int x, int y) {
        return x*y;
    }

}
```

서브 클래스

```
public class EngineerCalculator extends BasicCalculator{

    public double convertToLog10(double num) {
        return Math.log(num);
    }

}
```

```
public class InheritanceMain {

    public static void main(String[] args) {
        EngineerCalculator ec = new EngineerCalculator();
        int  result =ec.add(1, 2);
        System.out.println(result);

    }

}
```

한림대학교 SW중심대학

캡슐화

캡슐화

- 객체의 필드, 메소드를 하나로 묶고, 외부에서 바로 접근하지 못하게 은닉하는것
  - 객체가 제공하는 필드나 메소드로만 접근 가능
- 필드와 메소드를 캡슐화하여 보호하는 이유는 외부의 잘못된 사용으로 인해 객체가 손상되지 않도록 하기 위함
- 접근 제한자를 통해 이루어짐

접근제한자	동일클래스	동일패키지	다른 패키지의 자식 클래스	다른 패키지
public	O	O	O	O
protected	O	O	O	
default	O	O		
private	O			

캡슐화

잘못된 경우

```
public class UserInvalid {

    public String id="orange";
    public String password="banana";

}
```

```
public class CapsuleMain {

    public static void main(String[ ] args) {
        UserInvalid uiv = new UserInvalid();
        uiv.password = "apple";
        System.out.println(uiv.password);
    }

}
```

올바른 경우

```
public class User {
    private String id;
    private String password;
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

```
public class CapsuleMain {

    public static void main(String[ ] args) {
        User uv = new User ();
        uv.setPassword("apple");
        System.out.println(uv.getPassword());
    }
}
```

다형성

다형성

- 하나의 참조 변수로 여러 타입의 객체를 참조 할 수 있도록 하는것
- 객체를 부품화 하여 유지보수 용이 및 재활용성의 증가
- ex) 상속을 통한 업캐스팅, 메소드 오버라이딩

```
public class User {

    public boolean login(String id, String pw) {
        boolean result=false;
        //보안 신뢰성 낮은  DB 사용 구현
        useHash();
        return result;
    }
    public boolean useHash() {
        boolean result=true;
        return true;
    }

}
```

```
public class Admin  extends User{

    public boolean login(String id, String pw) {
        boolean result=false;
        //보안 신뢰성 높은  DB 사용 구현
        useSSL();
        return result;
    }
    public boolean useSSL() {
        boolean result=true;
        return true;
    }

}
```

```
public class MainClass {
    public static void main(String[] args) {
        User ur = new Admin();
        boolean res=ur.login("apple","orange");
    }
}
```

추상클래스

Abstract class

- **형식**
  - class 앞에 abstract 예약어가 붙음
  - 반드시 클래스 안에 abstract method 가 있어야 함
  - 객체 생성 사용시 반드시 상속 클래스에서 abstract method를 구현해야 하며, 상속 클래스를 이용해 객체를 생성해야 함.

class

```
class AnimalClass {
    public void animalSound() {
        System.out.println("sound");
    }
    public void sleep() {
        System.out.println("Zzz");
    }
}
```

```
public class MainClass {
    public static void main(String[] args) {
        AnimalClass ani= new AnimalClass();
        ani.animalSound();
        ani.sleep();
    }
}
```

Abstract class

```
abstract class Animal {
    public abstract void animalSound();
    public void sleep() {
        System.out.println("Zzz");
    }
}
```

```
class Pig extends Animal {
    public void animalSound() {
        System.out.println("wee wee");
    }
}
```

```
public class AbstractMain {
    public static void main(String[] args) {
        Animal pg= new Pig();
        pg.animalSound();
        pg.sleep();
    }
}
```

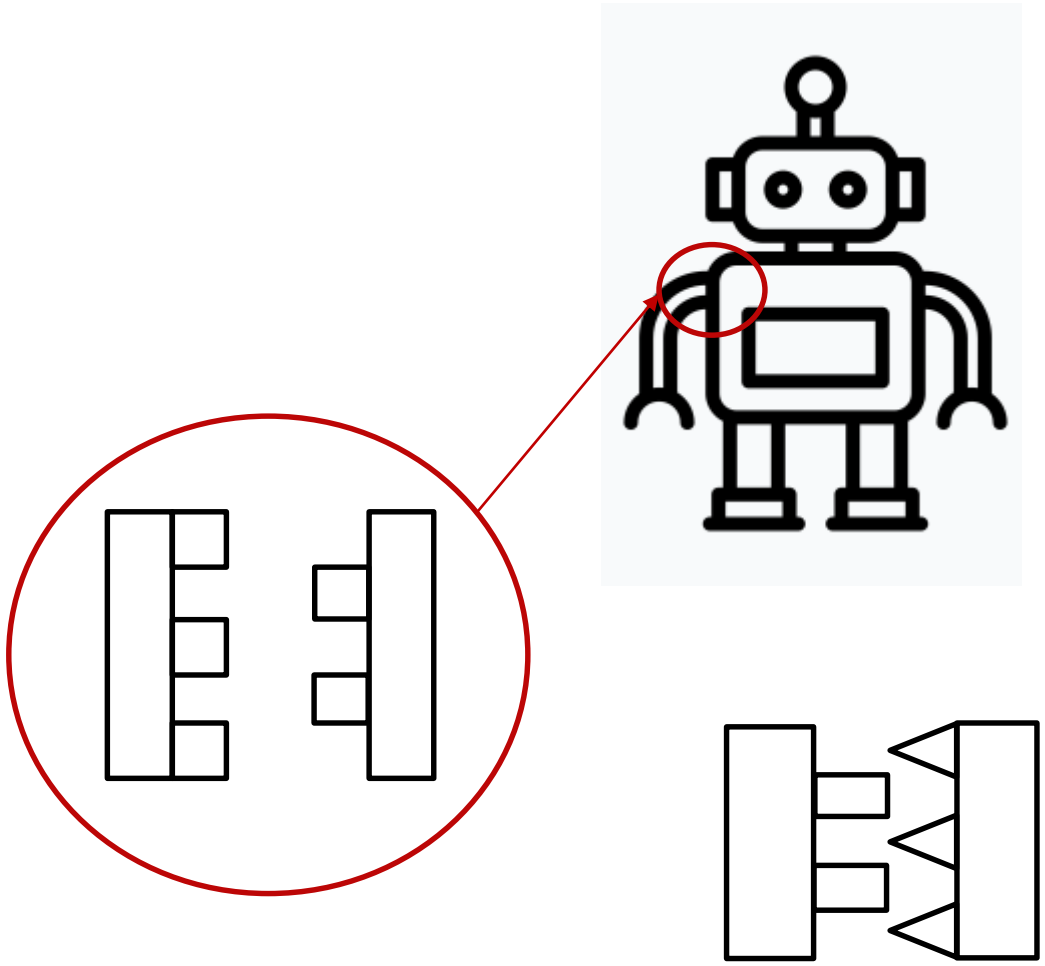
Abstract class

- 구현 클래스 설계 규격을 만들고자 할 때
  - 구현 클래스가 가져야 할 필드와 메소드를 추상 클래스에 미리 정의
  - 구현 클래스의 공통된 필드와 메소드의 이름 통일할 목적
- 전체 기능중 일부 기능이 달라질수 있을 경우

인터페이스

인터페이스

- 프로그램의 접합부의 클래스의 설계도 역할
- 장점
  - 개발시간 단축
  - 결합도를 낮춘다
  - 표준화 가능



인터페이스

형식

```
접근제한자 class 클래스이름{
        필드;
        접근제한자 리턴타입 메소드이름(매개변수들){
        }

}
```

```
public class Calculator {

    public String calName;

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

```
[public] interface 인터페이스이름 {
        [public static final] 필드;
        [public abstract] 리턴타입 메소드이름 (매개변수들);
}
```

public static final으로 선언된 상수만 가능

abstract 메소드만 가능

```
interface Calculator {
    int DecimalUnit = 4;
    int add(int x, int y) ;

}
```

- **사용법**
  - 인터페이스를 구현한 클래스 생성
  - 업캐스팅 형식으로 구현클래스의 객체 생성

```
public interface RemoteControl {

    void turnOn();
    void turnOff();
    void setVolume(int volume);

}
```

```
public class SKRemoteControlImpl implements RemoteControl {

    private int volume;

    public void turnOn() {
        System.out.println("SK turnOn");
    }
    public void turnOff() {
        System.out.println("SK turnOff");
    }
    public void setVolume(int volume) {

        System.out.println("SK volume: " + volume);
    }
}
```

```
public class LGRemoteControlImpl implements RemoteControl {

    private int volume;

    public void turnOn() {
        System.out.println("LG turnOn");
    }
    public void turnOff() {
        System.out.println("LG turnOff");
    }
    public void setVolume(int volume) {

        System.out.println("LG volume: " + volume);
    }
}
```

```
public class RemoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc = new LGRemoteControlImpl();
//      RemoteControl rc = new SKRemoteControlImpl();
        rc.turnOn();
        rc.turnOff();
        rc.setVolume(1);
    }
}
```