JAVA 입문 : 이론과 실습

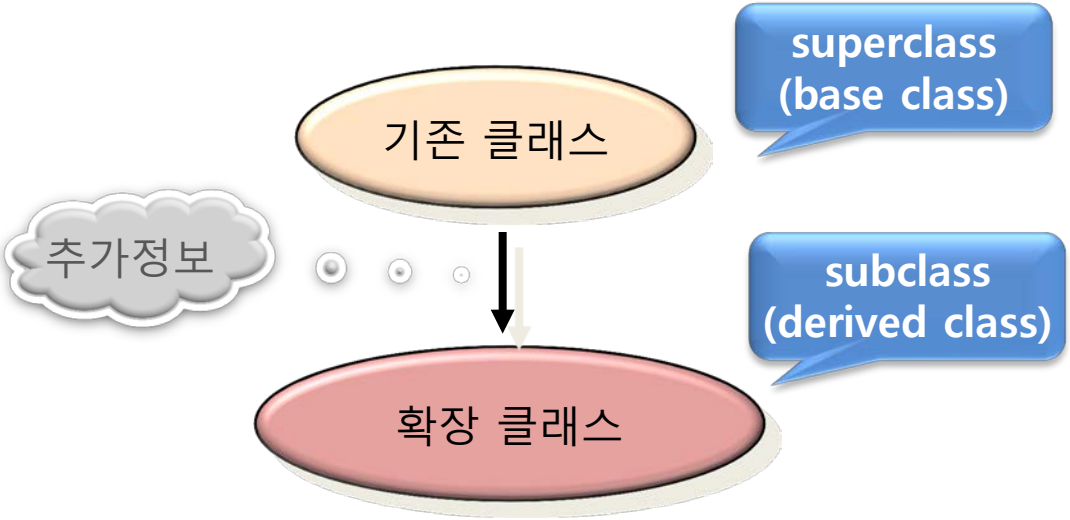
제 6장
확장 클래스와 인터페이스

목차

- 확장 클래스
- 인터페이스
- 클래스형 변환
- 클래스 설계

확장 클래스 [1/3]

- 확장 클래스의 개념



- 상속(Inheritance)
  - superclass의 모든 속성이 subclass로 전달되는 기능
  - 클래스의 재사용성을 증가 시킨다.

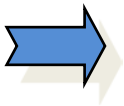
확장 클래스 [2/3]

- 확장 클래스 정의 형태

```
class SubClassName extends SuperClassName {
        // 필드 선언
        // 메소드 정의
}
```

- 확장 클래스의 예 :

```
class SuperClass {
        int a;
        void methodA {
                // ...
        }
}
```

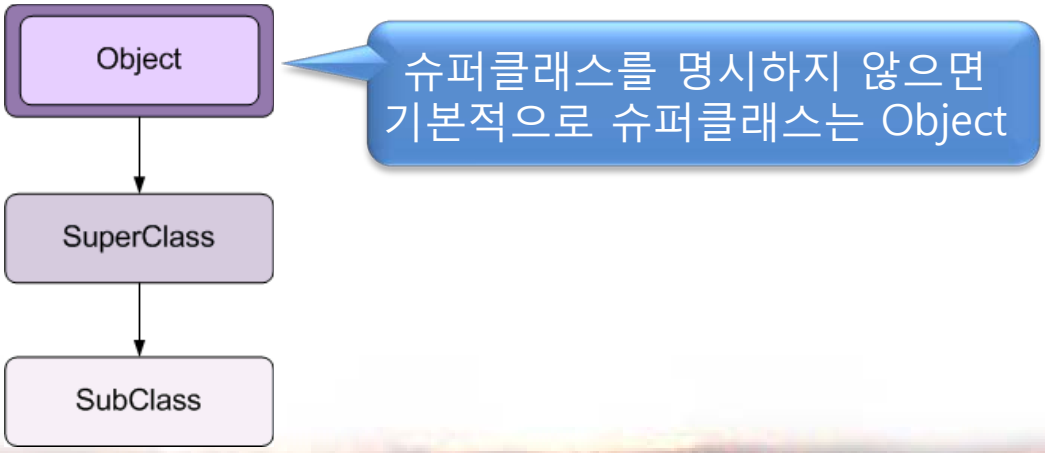


```
class SubClass extends SuperClass {
        int b;
        void methodB {
                // ...
        }
}
```

확장 클래스 [3/3]

- 단일 상속(single inheritance)
  - 오직 한 개의 슈퍼클래스만 가질 수 있다.
  - 다중 상속(multiple inheritance)

- Object 클래스
  - 모든 클래스의 슈퍼클래스
  - 객체에 적용할 수 있는 기본 메소드가 정의되어 있음



확장 클래스의 필드 [1/2]

- 다른 이름
  - 그대로 상속된다.
- 동일한 이름
  - 숨겨진다.
  - super --- 숨겨진 슈퍼클래스에 있는 필드를 참조할 때 사용.

```
class SuperClass {
        int a = 1;
        int b = 1;
}
class SubClass extends SuperClass {
        int a = 2;
        int b = super.a;
        // . . .
}
```

super.a
super.b
a
b

확장 클래스의 필드 [2/2]

[예제 6.1 – NameConflict.java]

```
class SuperClass {
   int a = 1;
   int b = 1;
}
class SubClass  extends  SuperClass {
   int a = 2;
   double b = 2.0;
   void output() {
       System.out.println("Base class: a = " + super.a + ",
                          Extended class: a = " + a);
       System.out.println("Base class: b = " + super.b + ",
                          Extended class: b = " + b);
   }
}
public class NameConflict {
   public static void main(String[] args) {
      SubClass obj = new SubClass();
      obj.output();
   }
}
```

실행 결과 :

Base class: a = 1, Extended class: a = 2
Base class: b = 1, Extended class: b = 2.0

확장 클래스의 생성자 [1/4]

- 형태와 의미는 모두 클래스 생성자와 같다.
  - 단, 슈퍼클래스를 초기화하기 위해 먼저 슈퍼클래스 생성자를 호출한다.
  - super()
    - 슈퍼클래스의 생성자를 명시적으로 호출
  - 명시적으로 호출하지 않으면, 기본 생성자가 컴파일러에 의해 자동적으로 호출

- 실행과정

1) 슈퍼클래스의 생성자를 호출한다.
2) 필드를 초기화하는 부분이 실행된다.
3) 생성자의 몸체 부분을 수행한다.

예제 6.2 테스트

확장 클래스의 생성자 [2/4]

[예제 6.2 - SubConstructor.java]

```
class SuperClass {
    SuperClass() {
        System.out.println("SuperClass Constructor ...");
    }
}
class SubClass extends SuperClass {
    SubClass() {
        System.out.println("SubClass Constructor ...");
    }
}
public class SubConstructor {
    public static void main(String[] args) {
        SubClass obj = new SubClass();
        System.out.println("in main ...");
    }
}
```

실행 결과 :

```
SuperClass Constructor ...
SubClass Constructor ...
in main ...
```

확장 클래스의 생성자 [3/4]

[예제 6.3 - SuperCallTest.java]

```
class SuperClass {
    int a, b;
    SuperClass() {
        a = 1; b = 1;
    }
    SuperClass(int a, int b) {
        this.a = a; this.b = b;
    }
}
class SubClass extends SuperClass {
    int c;
    SubClass() {
        c = 1;
    }
    SubClass(int a, int b, int c) {
        super(a, b);
        this.c = c;
    }
}
```

[Next Page]

확장 클래스의 생성자 [4/4]

- 슈퍼 클래스의 중복된 생성자 사용 예

[예제 6.3 - SuperCallTest.java]

```
public class SuperCallTest {
   public static void main(String[] args) {
      SubClass obj1 = new SubClass();
      SubClass obj2 = new SubClass(1, 2, 3);
      System.out.println("a = " + obj1.a + ", b = " + obj1.b +
                         ", c = " + obj1.c);
      System.out.println("a = " + obj2.a + ", b = " + obj2.b +
                         ", c = " + obj2.c);
   }
}
```

실행 결과 :

```
a = 1, b = 1, c = 1
a = 1, b = 2, c = 3
```

메소드 재정의 [1/4]

- 메소드 재정의(**method overriding**)
  - 슈퍼클래스에 정의된 메소드의 의미를 서브클래스에서 변경
    - 매개변수 개수와 형이 같음 --- 같은 메소드
    - 서브 클래스의 메소드로 대체
  - 메소드 중복(**method overloading**)
    - **매개변수 개수와 형이 다름** --- 다른 메소드

- 메소드 재정의를 할 수 없는 경우
  - 정적(static) 메소드
  - 최종(final) 메소드
    - 최종 클래스 내에 있는 모든 메소드는 묵시적으로 최종 메소드

예제 6.6 테스트

메소드 재정의 [2/4]

[예제 6.4 - OverridingAndOverloading.java]

```
class SuperClass {
   void methodA() {
      System.out.println(“In SuperClass …");
   }
}
class SubClass extends SuperClass {
   void methodA() {                  // Overriding
      System.out.println("In SubClass … Overriding !!!");
   }
   void methodA(int i) {             // Overloading
      System.out.println("In SubClass … Overloading !!!");
   }
}
public class OveridingAndOverloading {
   public static void main(String[] args) {
      SuperClass obj1 = new SuperClass ();
      SubClass obj2 = new SubClass();
      obj1.methodA();
      obj2.methodA();
      obj2.methodA(1);
   }
}
```

실행 결과 :

```
In SuperClass …
In SubClass … Overriding !!!
In SubClass … Overloading !!!
```

메소드 재정의 [3/4]

[예제 6.5 - HiddenMethod.java]

```
class SuperClass {
   static String greeting() { return "Good Bye"; }
   String name() { return "Oak"; }
}
class SubClass extends SuperClass {
   static String greeting() { return "Hello"; }
   String name() { return "Java"; }
}
public class HiddenMethod {
   public static void main(String[] args) {
      SuperClass s = new SubClass();
      System.out.println(s.greeting() + ", " + s.name());
   }
}
```

실행 결과 :

Good Bye, Java

메소드 재정의 [4/4]

[예제 6.6 - Addendum.java]

```
class SuperClass {
    void methodA() {
        System.out.println("do SuperClass Task.");
    }
}
class SubClass extends SuperClass {
    void methodA() {
        super.methodA();
        System.out.println("do SubClass Task.");
    }
}
public class Addendum {
    public static void main(String[] args) {
        SubClass obj = new SubClass();
        obj.methodA();
    }
}
```

실행 결과 :

```
do SuperClass Task.
do SubClass Task.
```

추상 클래스(abstract class) [1/3]

- **추상 메소드(abstract method)**를 갖고 있는 클래스
  - **추상 메소드** :
    - 실질적인 구현을 갖지 않고 **메소드 선언**만 있는 경우

- 선언 방법

```
abstract class AbstractClass {
        public abstract void methodA();
        void methodB() {
                // ...
        }
}
```

추상 클래스(abstract class) [2/3]

- 구현되지 않고, 단지 외형만을 제공
  - 추상 클래스는 객체를 가질 수 없음
  - 다른 외부 클래스에서 메소드를 사용할 때 일관성 있게 다루기 위한 방법을 제공
- 다른 클래스에 의해 상속 후 사용 가능
  - 서브클래스에서 모든 추상 메소드를 구현한 후에 객체 생성 가능

[예제 6.7] 테스트

추상 메소드를 서브클래스에서 구현할 때 접근 수정자는 항상 일치

추상 클래스(abstract class) [3/3]

[예제 6.7 - AbstractClassTest.java]

```
abstract class AbstractClass {
    public abstract void methodA();
    void methodB() {
        System.out.println("Implementation of methodB()");
    }
}
class ImpClass extends AbstractClass {
    public void methodA () {
        System.out.println("Implementation of methodA()");
    }
}
public class AbstractClassTest {
    public static void main(String[] args) {
        ImpClass obj = new ImpClass();
        obj.methodA();
        obj.methodB();
    }
}
```

실행 결과 :

```
Implementation of methodA()
Implementation of methodB()
```

무명 클래스(Anonymous Class) [1/2]

- 중첩 클래스와 같이 클래스 내부에서 사용하고자하는 객체를 정의할 수 있는 방법을 제공하며
- 객체를 생성하고자 하는 코드 부분에서 직접 객체 생성루틴과 함께 클래스를 정의
  - 클래스의 이름이 없다.
  - 한번만 사용이 가능

[예제 6.8]

무명 클래스를 사용하여 필요한 클래스를 따로 정의하지 않고, methodA() 메소드의 매개변수로 직접 정의하고 객체를 생성하여 사용

무명 클래스(Anonymous Class) [2/2]

[예제 6.8 - AnonymousExample.java]

```
class AnonymousClass {
   public void print() {
      System.out.println("This is AnonymTest Class.");
   }
}

public class AnonymousExample {
   public static void methodA(AnonymousClass obj) {
      obj.print();
   }
   public static void main(String[] args) {
      methodA(new AnonymousClass() {       // 무명 클래스
         public void print() {
            System.out.println("This is redefined by Anonymous Example.");
         }
      });
   }
}
```

실행 결과 :

This is redefined by Anonymous Example.

인터페이스란

- 사용자 접속을 기술하는 방법으로 메소드들을 선언하고 필요한 상수들을 정의한 프로그래밍 단위
- 상수와 구현되지 않은 메소드들만을 갖고 있는 순수한 설계의 표현
- 다중상속(multiple inheritance)이 가능
  - 단일 상속(single inheritance) - 클래스

인터페이스 선언 [1/4]

- 선언 형태

```
[public] interface interfaceName [extends ListOfSuperInterfaces]
        // constant definitions
        // method declarations
}
```

- 사용 예

```
interface BaseColors {
        int RED = 1;
        int GREEN = 2;
        int BLUE = 4;

        void setColor(int color);
        int getColor();
}
```

내부적으로 상수를 의미하는 public, static, final의 속성

인터페이스 선언 [2/4]

- 인터페이스의 필드
  - 내부적으로 상수를 의미하는 public, static, final의 속성
  - 선언된 모든 필드는 반드시 초기화

```
interface BaseColors {
        int RED = 1;
        int GREEN = RED + 1;
        int BLUE = GREEN + 2;

        void setColor(int color);
        int getColor();
}
```

C 언어의 #define문 사용과 유사

인터페이스 선언 [3/4]

- 인터페이스의 메소드
  - 추상(abstract) 속성
    - 내부적으로 추상 메소드가 됨
  - 인터페이스의 모든 메소드를 구현하지 않는 클래스는 추상 클래스로 선언
  - 내부적으로 모두 public 메소드
  - static이 올 수 없음
  - 생성자를 갖지 않음

인터페이스 선언 [4/4]

- 인터페이스 몸체에서 사용할 수 없는 선언 수정자
  - **private, protected, synchronized, volatile**

- 일단, 인터페이스가 선언되면 인터페이스를 구현하는 클래스가 있어야 한다. 인터페이스는 객체를 가질 수 없다.
  - 7.3절에서 설명

```
class className implements InterfaceName {
        // . . .
}
```

인터페이스 확장 [1/5]

- 확장 형태

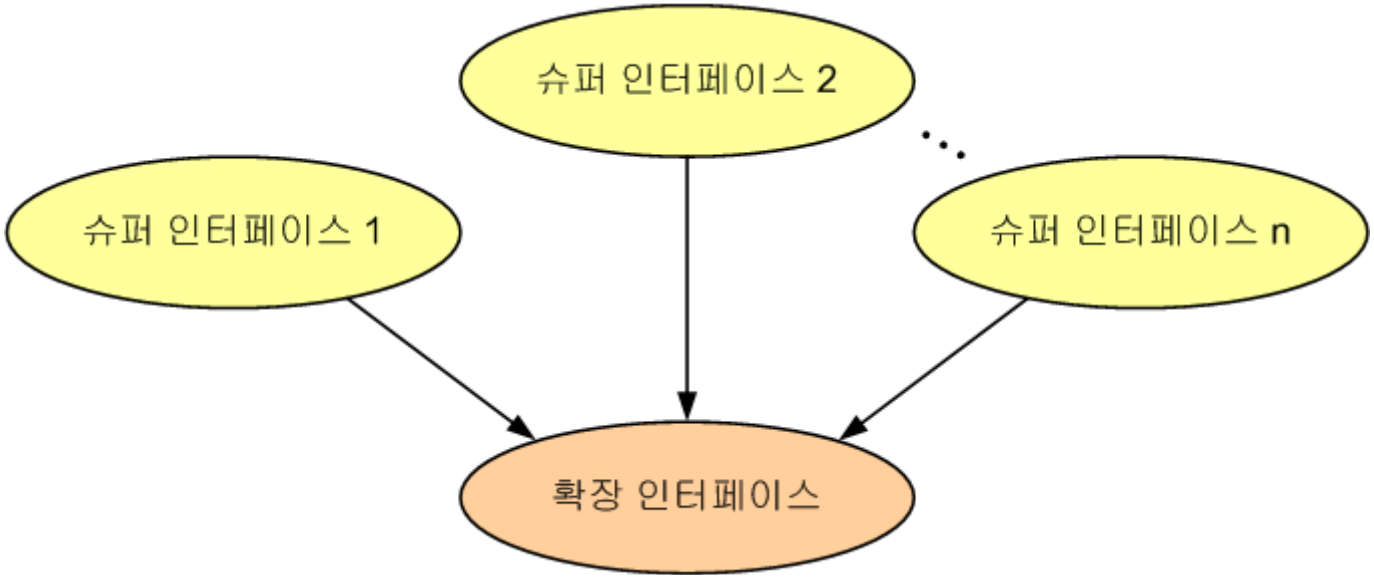
```
[public] interface interfaceName extends ListOfSuperInterfaces {
        //  constant definitions
        //  method declarations
}
```

- 확장 예

```
interface RainbowColors extends BaseColors {
        int YELLOW = 3;
        int ORANGE = 5;
        int INDIGO = 6;
        int VIOLET = 7;
        void printColor(int color);
}
```

인터페이스 확장 [2/5]

- 인터페이스의 다중 상속



인터페이스 확장 [3/5]

- 다중 상속 예

```
interface ManyColors extends RainbowColors, PrintColors {
        int VERMILION = 3;
        int CHARTUSE = RED + 90;
  }
```

- 다중 상속시 동일한 이름의 상수를 상속
  - 단순 명은 ambiguous하기 때문에 에러
  - RainbowColors.YELLOW, PrintColors.YELLOW

인터페이스 확장 [4/5]

- 메소드 상속
  - **overloading, overriding**
  - 그러나, 시그네춰가 같고 복귀형이 다르면 에러

```
interface BaseColors {
    // ...
    void setColor(int color);
    int getColor();
}
```

- 메소드 상속 예

```
interface ActionColors extends BaseColors {
        void setColor(int color) ;                      // overriding
        void setColor(int red, int green, int blue) ;  // overloading
        // Color getColor();                            // 에러
}
```

인터페이스 확장 [5/5]

[예제 6.9 - InterfaceNameConflict.java]

```
interface BaseColors {
    int RED = 1;
    int GREEN = 2;
    int BLUE = 4;
}
interface ExtendedColors extends BaseColors {
    int RED = 1;
    int BLUE = 3;
    int YELLOW = 5;
}
public class InterfaceNameConflict implements ExtendedColors {
    public static void main(String[] args) {
        System.out.println("Red                    = " + RED);
        System.out.println("GREEN                  = " + GREEN);
        System.out.println("BLUE                   = " + BLUE);
        System.out.println("BaseColors.BLUE        = " + BaseColors.BLUE);
        System.out.println("ExtendedColors.BLUE = " + ExtendedColors.BLUE);
        System.out.println("YELLOW                 = " + YELLOW);
    }
}
```

실행 결과 :

```
Red                    = 1
GREEN                  = 2
BLUE                   = 3
BaseColors.BLUE        = 4
ExtendedColors.BLUE = 3
YELLOW                 = 5
```

인터페이스 구현 [1/9]

- 클래스를 통하여 구현된 후 객체를 가짐
- 구현 형태

```
class ClassName implements InterfaceName {
        // fields
        // methods
}
```

인터페이스 구현 [2/9]

- 구현 예

```
class Color implements BaseColors {
        // fields
        public void setColor(int color) {
                // …
        }
        public int getColor() {
                // …
        }
}
```

```
interface BaseColors {
        int RED = 1;
        int GREEN = RED + 1;
        int BLUE = GREEN + 2;

        void setColor(int color);
        int getColor();
}
```

- 구현시 public 명시

인터페이스 구현 [3/9]

- 인터페이스에 있는 모든 메소드들을 구현하지 않으면 그 클래스는 추상 클래스가 된다.

```
abstract class SetColor implements BaseColors {
        // fields
        public void setColor(int color) {
                // …
        }
}
```

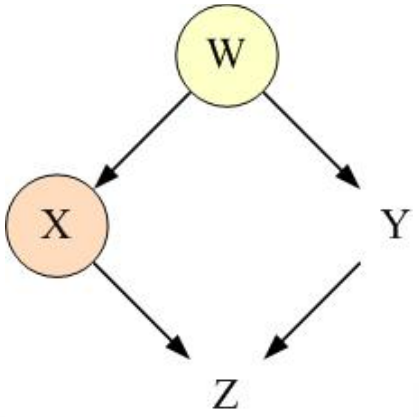
인터페이스 구현 [4/9]

- 클래스 확장과 동시에 인터페이스 구현

```
class ClassName extends SuperClass implements ListOfInterfaces {
        // fields
        // methods
}
```

- 다이아몬드 상속  [예제 6.10] 테스트

```
interface W   {   }
interface X extends W  {   }
class Y implements W  {   }
class Z extends Y implements X  {   }
```



인터페이스 구현 [5/9]

[예제 6.10 - ImplementingInterface.java]

```
interface BaseColors {
    int RED = 1, GREEN = 2, BLUE = 4;
    void setColor(int color);
    int getColor();
}
abstract class SetColor implements BaseColors {
    protected int color;
    public void setColor(int color) {
        this.color = color;
        System.out.println("in the setColor method ...");
    }
}
class Color extends SetColor {
    public int getColor() {
        System.out.println("in the getColor method ...");
        return color;
    }
}
```

[Next Page]

인터페이스 구현 [6/9]

[예제 6.10 - ImplementingInterface.java](cont.)

```
public class ImplementingInterface {
   public static void main(String[] args) {
      Color c = new Color();
      int i;

      c.setColor(10);
      i = c.getColor();
      System.out.println("in the main method ...");
   }
}
```

실행 결과 :

```
in the setColor method ...
in the getColor method ...
in the main method ...
```

인터페이스 구현 [7/9]

[예제 6.11 - DiamondInheritance.java]

```
interface W {
    int w = 1;
}
interface X extends W {
    int x = 2;
}
class Y implements W {
    private int y;
    void setY(int i) { y = w + I; }
    int getY() { return y; }
}
class Z extends Y implements X {
    private int z;
    void setZ(int i) { z = x + i; }
    int getZ() { return z; }
}
```

[Next Page]

인터페이스 구현 [8/9]

[예제 6.10 - DiamondInheritance.java](cont.)

```
public class DiamondInheritance {
   public static void main(String[] args) {
      Z z = new Z();

      z.setY(10);
      z.setZ(10);
      System.out.println("y = " + z.getY() + ", z = " + z.getZ());
   }
}
```

실행 결과 :

y = 11, z = 12

인터페이스 구현 [9/9]

- 인터페이스 형의 매개 변수 선언

```
void dummy(InterfaceA interfaceRef) {
        Object obj = interfaceRef;
        // ...
}
```

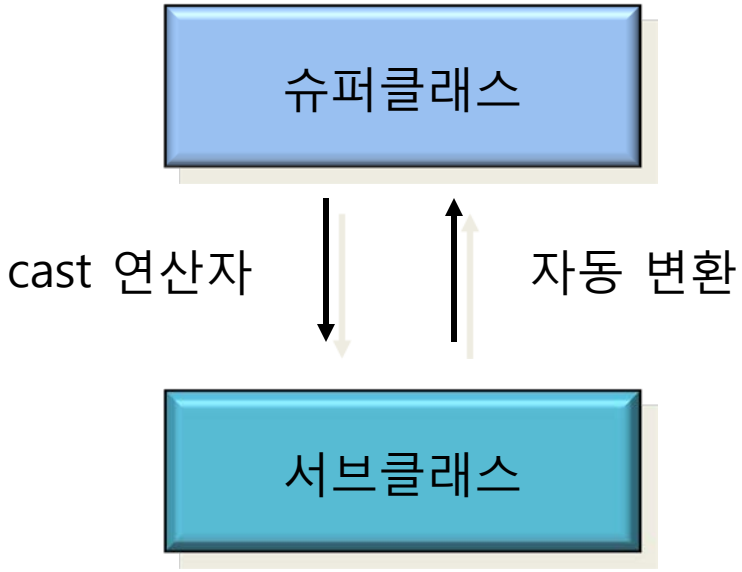
  - dummy()의 실 매개 변수 : InterfaceA 형을 구현한 클래스의 객체

인터페이스 vs. 추상 클래스

- 인터페이스
  - 다중 상속을 지원
  - 메소드 선언만 가능 - 메소드를 정의할 수 없다.

- 추상 클래스
  - 단일 상속
  - 메소드의 부분적인 구현이 가능

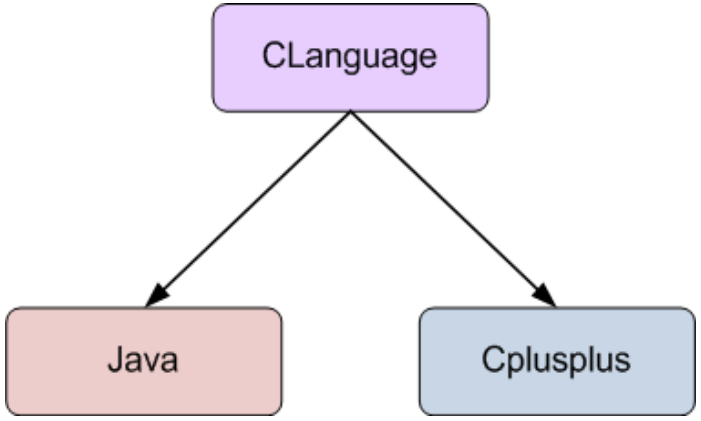
클래스형 변환 [1/7]



♣ casting up : valid type conversion

♣ casting down : invalid type conversion

클래스형 변환 [2/7]




```
void dummy(CLanguage obj) {
            // …
}
// ...
Java j = new Java();
dummy(j);        // OK
```

```
void dummy(Java  obj) {
            // …
}
// ...
Clanguage c = new CLanguage();
dummy(c);        // 에러
```

dummy((Java)c); // 예외발생

클래스형 변환 [3/7]

- **Polymorphism**
  - 적용하는 객체에 따라 메소드의 의미가 달라지는 것

c의 형은 CLanguage이지만
Java 클래스의 객체를 가리킴

```
CLanguage  c = new Java();
c.print();
```

클래스형 변환 [4/7]

[예제 6.12 - ClassConversion.java]

```
class SuperClass {
    public boolean equal(Object obj) {
        if (obj instanceof SuperClass) return true;
        else return false;
    }
}
class SubClass extends SuperClass {
    public boolean equal(Object obj) {
        if (obj instanceof SubClass) return true;
        else return false;
    }
}
public class ClassConversion {
    public static void main(String[] args) {
        SuperClass sup = new SuperClass();
        SubClass sub = new SubClass();
```

[Next Page]

클래스형 변환 [5/7]

[예제 6.12 - ClassConversion.java](cont.)

```
        if (sup.equals(sub)) System.out.println("casting up is valid");
          else System.out.println("casting up is not valid");
        if (sub.equals(sup)) System.out.println("casting down is valid");
          else System.out.println("casting down is not valid"); }
    }
}
```

실행 결과 :

```
casting up is valid
casting down is not valid
```

클래스형 변환 [6/7]

[예제 6.13 - Polymorphism.java]

```
class SuperClass {
    int value;
    SuperClass() {
        value = 0;
    }
    SuperClass(int i) {
        value = i;
    }
    void output() {
        System.out.println("SuperClass : " + value);
    }
}
class SubClass extends SuperClass {
    int value;
    SubClass (int i) {
        value = i;
    }
    void output() {
        System.out.println("SubClass : " + value);
    }
}
```

클래스형 변환 [7/7]

[예제 6.13 - Polymorphism.java](cont.)

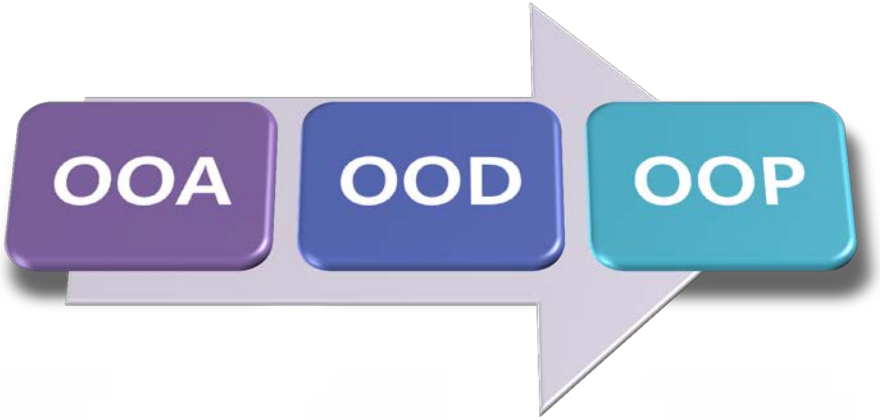
```
public class Polymorphism {
   public static void main(String[] args) {
      SuperClass obj1 = new SuperClass(1);
      SubClass obj2 = new SubClass(1);

      print(obj1);
      print(obj2);
   }
}
```

실행 결과 :

```
SuperClass : 1
SubClass : 1
```

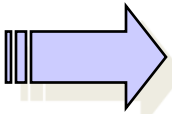
클래스 설계 [1/2]

- 클래스를 정의할 때 상속성을 이용하여 계층적 구조를 갖도록 설계하는 일은 매우 중요
- 공통적으로 갖는 필드와 메소드들은 모두 슈퍼클래스에 선언
- 객체지향 방법론의 접근 순서



클래스 설계 [2/2]

```
class AnsiC {
        int declarations;
        int operators;
        int statements;
        void functions() {
                // ...
        }
}
class Java extends AnsiC {
        int classes;
        int exceptions;
        int threads;
}
class Cplusplus Extends AnsiC {
        int classes;
        int exceptions;
        int operatorOverloadings;
}
```



```
class Oopl extends AnsiC {
        int classes;
        int exceptions;
}

class Java extends Oopl {
        int threads;
}

class Cplusplus extends Oopl {
        int operatorOverloadings;
}
```

단원 요약 [1/2]

- 확장 클래스
  - 상속을 통한 superclass의 속성 사용과 추가적인 기능을 정의한 클래스
    - 클래스의 재사용성을 증가 시킴
  - 자바에서는 단일 상속만 가능
    - 인터페이스의 제공
  - Object 클래스
    - 모든 클래스의 슈퍼클래스로 객체에 적용할 수 있는 기본 메소드가 정의되어 있음

- 메소드 재정의
  - 상속 과정에서 발생하며, superclass에 정의된 메소드의 의미를 subclass에서 변경하는 경우로 매개변수 개수와 형이 같음

 메소드 중복

단원 요약 [2/2]

- 추상 클래스
  - 추상 메소드를 갖고 있는 클래스로 단일 상속만 가능하며, 메소드의 부분적인 구현이 가능함
- 인터페이스
  - 상수와 구현되지 않은 메소드들만을 갖고 있는 순수한 설계의 표현으로 다중 상속이 가능함