고급객체지향 프로그래밍
강의노트 #01

객체 지향 프로그래밍

조용주
ycho@smu.ac.kr

SW 왜 문제인가?

- 소프트웨어 위기 (SW Crisis)
  - 1968 년 NATO 소프트웨어공학 학회에서 처음 사용됨
  - 위기의 원인?
    - 소프트웨어 개발 프로세스의 복잡성
    - 소프트웨어 공학의 미성숙
    - SW 규모가 커짐
      - 계획된 SW 납기를 맞추지 못함
      - 계획된 SW 인력보다 많이 필요해서 밤샘하며 일함
      - SW 품질을 맞추지 못함
      - 유지 보수가 어려워짐

SW 품질

- 건축과 비교
  - 집을 짓는다고 가정하면, 어떻게 품질을 판단하는가?
  - 방을 계획보다 1 개 적게 만들면? → 설계에 어긋나는 중대한 결함
  - 설계를 준수해서 만들어도
    - 입주 일자를 못 맞춘다면?
    - 계획한 예산보다 많은 비용이 들어간다면?
  - 설계 준수도 중요하지만, 납기 및 비용도 중요함

SW 품질

- SW
  - 품질 문제는 정해진 시간에, 정해진 노력으로, 정해진 요구 사항을 해결하지 못함
  - 정해진 시간
    - 시간 초과되지 말아야 함
  - 정해진 노력
    - man-month 로 측정
    - 시간과 동일하면서 비용이기도 함
  - 정해진 요구 사항
    - 사용자가 원하는 기능
    - 개발자가 원하는 기능이 아님
  - SW 는 보이지 않기 때문에 품질 판단이 더욱 어려움

SW 심각성

- SW 가 대형화 되면서 더욱 위기가 심각해짐
  - 대기업 프로그램을 10,000,000 LOC 라고 가정하면
    - 1 페이지에 50 줄이 담길 수 있다면, 약 20 만 페이지
    - 책 한 권이 300 페이지라면 약 700 권 정도
  - 개발 비용이 1 LOC 에 3 만원이라면 3 천억 원
    - 한 줄의 프로그램이 아니라, 요구 분석, 설계, 테스트, 문서화, 검수 등이 모두 포함된 비용
- SW 대형화로 인한 문제
  - SW 대형화로 인한 협업의 문제
    - 설계의 중요성
    - 대형 SW 를 분할, 나누어 개발하고 통합
  - SW 대형화로 인한 요구 사항의 문제
    - 요구 사항은 계속해서 늘어남

SW 심각성

- SW 오류의 심각성
  - 눈에 보이지 않아서 오류의 심각성을 중요하게 생각하지 않는 경향이 있음
  - SW 오류로 인해 의료 / 국방 분야에서는 생명의 손실이 발생할 수 있음
  - 금융 / 재무 분야에서는 금전적인 경제 손실 발생 가능
  - 테슬라 자율 주행 사고

차량의 자동주행센서가 밝게 빛나는 하늘과 트럭의 흰색 면을 미처 구분하지 못한 것으로 테슬라 측은 파악하고 있다.

https://www.mk.co.kr/news/special-edition/view/2016/07/473465/

좋은 SW 란 ?

- 좋은 SW 는 사람 중심
  - 기능성, 효율성, 유지보수성, 재사용성, 가독성 등이 중요
  - 사용자가 원하는 기능을 다하는 것도 중요하지만, **유지보수**가 쉽고 **재사용**을 하기 쉽게 하고, **읽기 좋게** 만들어야 함
    - 읽기 쉽게 만들지 않으면 인건비 및 유지보수 비용이 늘어남
- 좋은 SW 는 만들어 놓으면
  - 중복 없고 이해하기 쉬워서 수정이 쉬움
  - 다른 사람이 가져다 쓰기 편리함

객체 지향 프로그래밍

- SW 위기를 해결하는 방안으로 패러다임이 변화
  - SW 비용을 줄이기 위해 개발자가 빠르게, 보다 잘 할 수 있게 변화 ◊ 객체 지향으로의 패러다임 변화
- 객체 지향의 특징을 이해해야 함
  - 비객체 지향과 비교해서 무엇이 다른지 ?
    - 어떤 점이 프로그래밍을 향상시키는지 이해
  - 공통 부분을 재사용하는 방법으로 상속을 이해해야 함
    - 사용할 때와 사용하지 않을 때를 구분

객체 지향 프로그래밍

- 객체 지향 프로그래밍 (object-oriented programming) 이란 ?
  - 프로그래밍하는 스타일 중 한 가지
  - 절차적 (procedural) 프로그래밍 또는 구조화 (structured) 프로그래밍의 문제점을 개선한 프로그래밍 방법
  - 프로그램은 두 가지 요소 (데이터와 코드) 로 구성됨
    - 데이터
      - 입출력에 사용되는 값으로 코드가 실행되면서 사용됨
    - 코드
      - 컴퓨터가 실행하는 명령
      - 데이터를 이용해서 문제를 해결하고 결과 도출

절차 중심 프로그래밍 방법의 문제점

- 객체 지향 프로그래밍은 절차 중심 프로그래밍 방법을 개선
- 절차 중심 프로그래밍의 문제
  - 절차와 데이터가 분리되어 있음
  - 예 : 자동차가 움직이는 과정을 코드로 표현한다고 가정
    - 차가 움직이려면 자동차 엔진에 기름이 들어가고 이를 연소시킨다. 이 과정에서 발생하는 에너지가 바퀴에 전달되어 차를 움직임
    - 이 과정을 move() 함수라고 만들면 차를 움직일 때에는 move() 함수를 호출하기만 하면 됨
    - move() 함수는 연료라는 데이터가 필요한데 함수 안에 넣을 수는 없음
      - 해결 방안은 매개 변수로 전달

절차 중심 프로그래밍 방법의 문제점

- 자바 코드로 작성

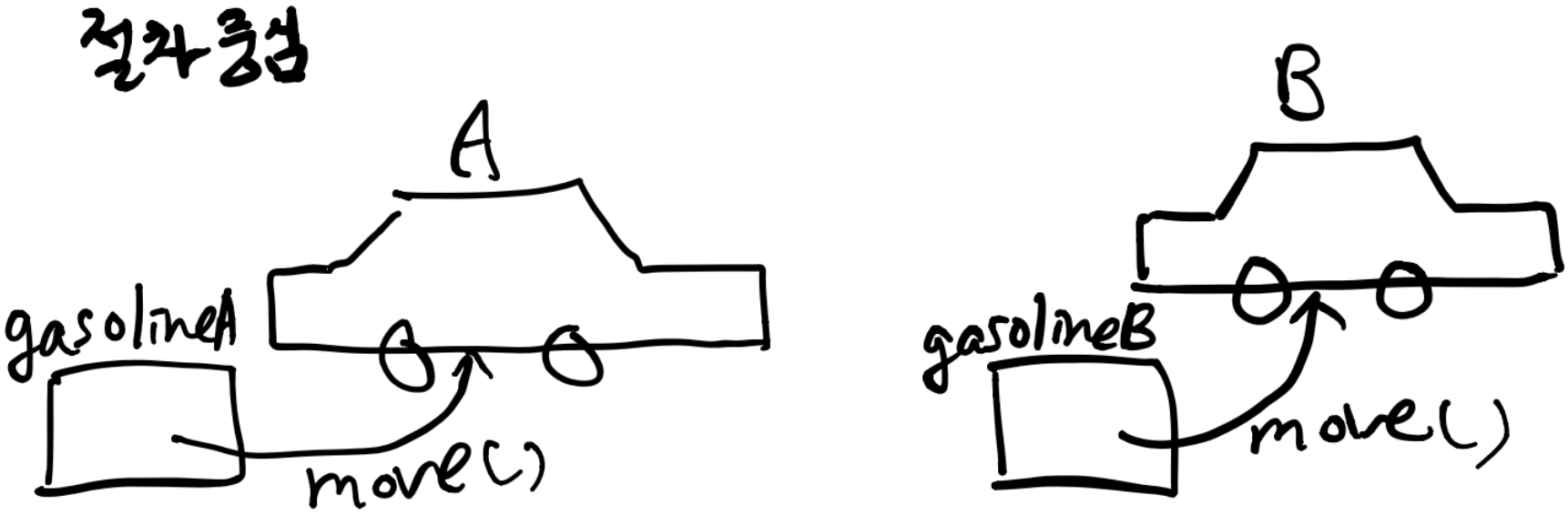
```
void move(double gas) {
    //  여기에는 주어진 기름을 이용해서 엔진에서
    //  연소시키고 바퀴에 동력을 전달해서 차를
    //  움직이는 코드가 있다고 가정
    ...
}

double gasoline = 20.0;
move(gasoline);
```

절차 중심 프로그래밍 방법의 문제점

- 만약 차 가 두 대라면 ? A 와 B 자동차가 있음

```
double gasolineA = 20.0;
double gasolineB = 20.0;
move(gasolineA);
move(gasolineB);
```

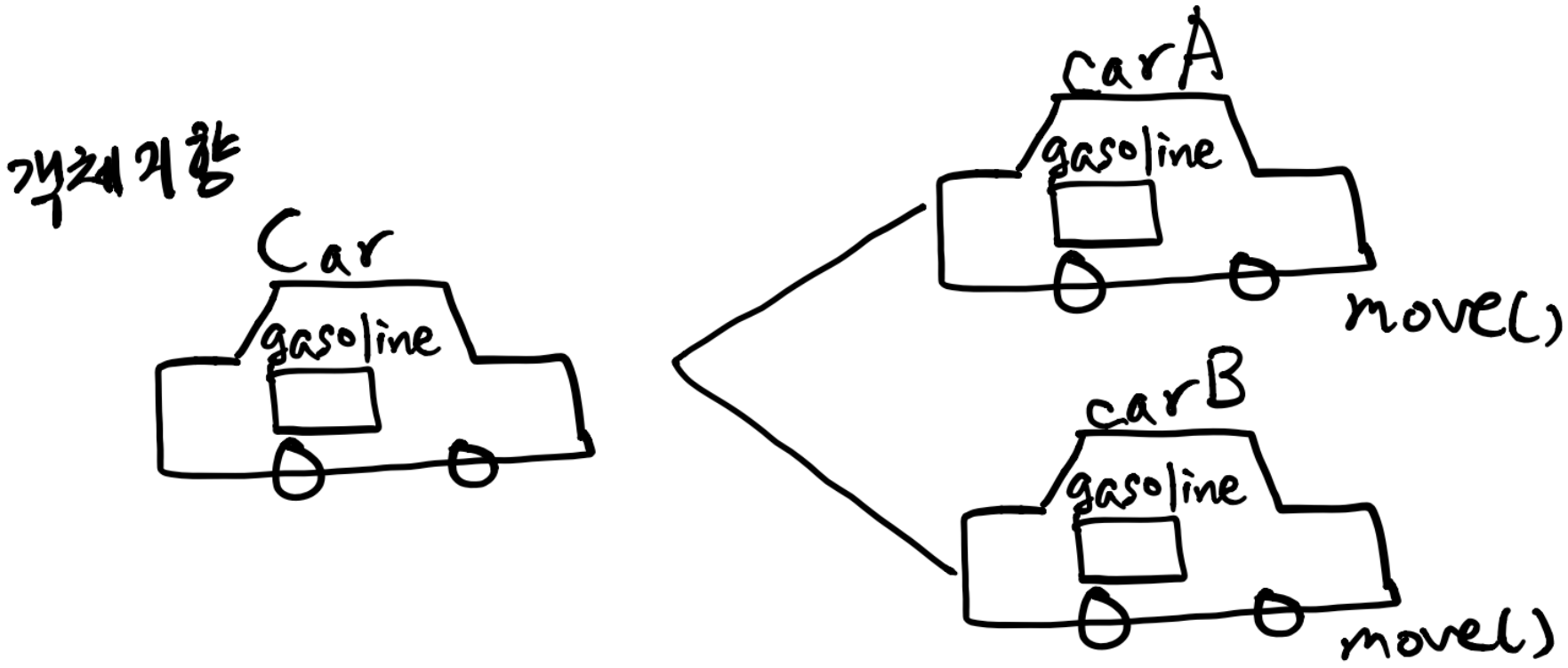


객체 지향 프로그래밍

- 객체 지향 프로그래밍에서는 클래스를 이용해서 데이터와 절차를 묶어서 한 개의 자료형으로 취급할 수 있음

```
class Car {
    double gasoline;
    void move() {
        …
    }
}
Car carA = new Car();
Car carB = new Car();
carA.move();
carB.move();
```

객체 지향 프로그래밍



- 코드의 재사용성을 생각해도 객체 지향 프로그래밍이 유리함

클래스와 객체

- 객체 지향 프로그래밍에서 코드는 **객체로 구성**
  - 객체 없이 프로그램을 만들 수 없음
- 객체 지향은 우리가 사는 세상에서 실제로 일어나는 일을 흉내 내어 프로그래밍 함
  - 주변에 있는 모든 것이 클래스가 될 수 있고 프로그래밍 될 수 있음
  - 명사로 표현될 수 있는 것들은 모두 객체로 표현할 수 있다고 봄
- 클래스 (class)
  - 객체의 **속성과 함수를 정의**
  - 객체의 속성과 기능을 설명하는 자료
  - 제품의 설계도 또는 템플릿에 해당됨

클래스와 객체

- 객체 (object) 또는 인스턴스 (instance)
  - 클래스로부터 만들어지는 실체
  - 설계도로부터 생성된 제품

객체 지향 프로그래밍의 특성

- 추상화 (abstraction)
- 캡슐화 (encapsulation)
- 상속 (inheritance)
- 다형성 (polymorphism)

추상화

- 추상화 (abstraction)
  - 추상적인 것으로 만듦
  - 추상 – 여러 가지 사물이나 개념에서 공통되는 특성이나 속성 따위를 추출하여 파악하는 작용 (표준국어대사전)
  - 복잡한 사물 / 문제로부터 핵심적인 속성 또는 기능들을 추출하는 작업
    - 문제로부터 해결 방법을 설계하는 것
    - 클래스의 속성과 절차를 결정하는 것
    - 클래스 간의 연관 관계를 결정하는 것
  - 새로운 내용은 아님. 절차적 프로그래밍에서도 해오던 일

캡슐화

- 캡슐화 (encapsulation, 은닉)
  - 데이터와 함수들을 클래스 내에 담고 사용자가 사용할 수 있는 부분만 보이도록 하고 나머지는 내부에 감추는 것
  - 인터페이스만 보여줌으로써 쉽게 사용할 수 있게 함
  - 내부 코드와 데이터를 보호
  - 내부 코드의 복잡함을 감출 수 있음

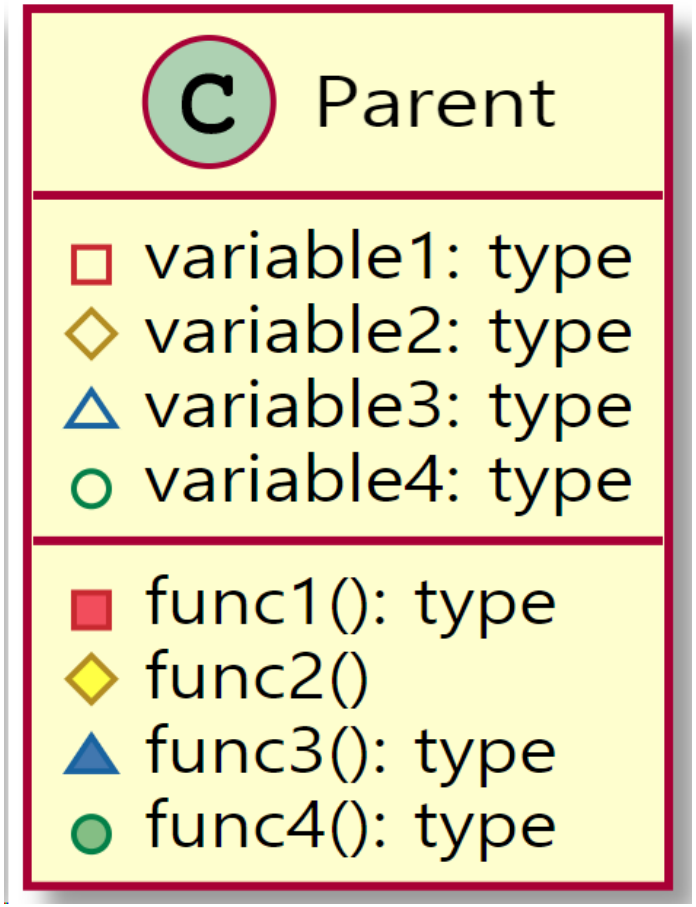
접근자 (getter) 와 설정자 (setter) 메소드

- 접근자
  - 주로 멤버 변수의 값을 반환하는 용도로 사용
  - 일반적으로 함수 이름 앞에 get 을 붙임
  - 예 : getAge()

- 설정자
  - 멤버 변수의 값을 수정하는 용도로 사용
  - 일반적으로 함수 이름 앞에 set 을 붙임
  - 새로운 값을 인자로 전달 받음
  - 예 : setAge(30)

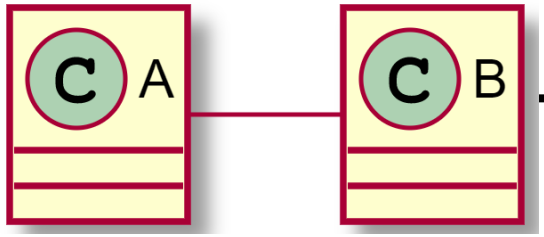
UML 클래스 다이어그램 (Class Diagram)

- 세 개 영역(클래스 이름, 멤버 변수, 멤버 함수)으로 나누어서 표기

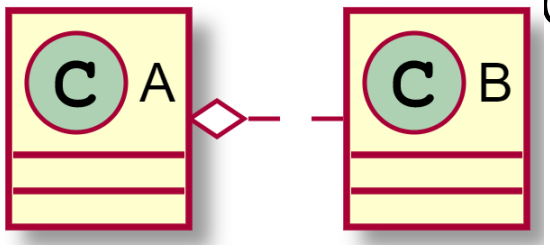


Character	Icon for field	Icon for method	Visibility
-	□	■	private
#	◇	◆	protected
~	△	▲	package private
+	○	●	public

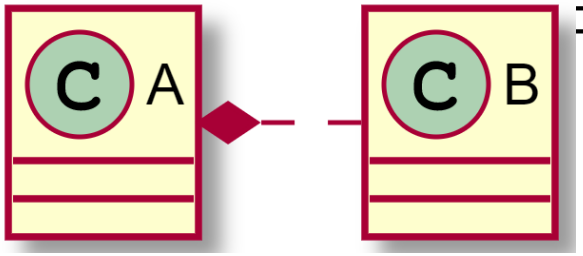
UML 클래스 다이어그램



- Association 연관
  - A, B 두 개 클래스가 참조 관계에 놓여 있음
  - Aggregation 과 Composition 은 Association 은 특수한 경우
- Aggregation 집합
  - A 가 B 를 포함하고 있을 때, B 가 A 없이도 존재할 수 있다면 집합 관계
  - 자주 쓰이지 않음

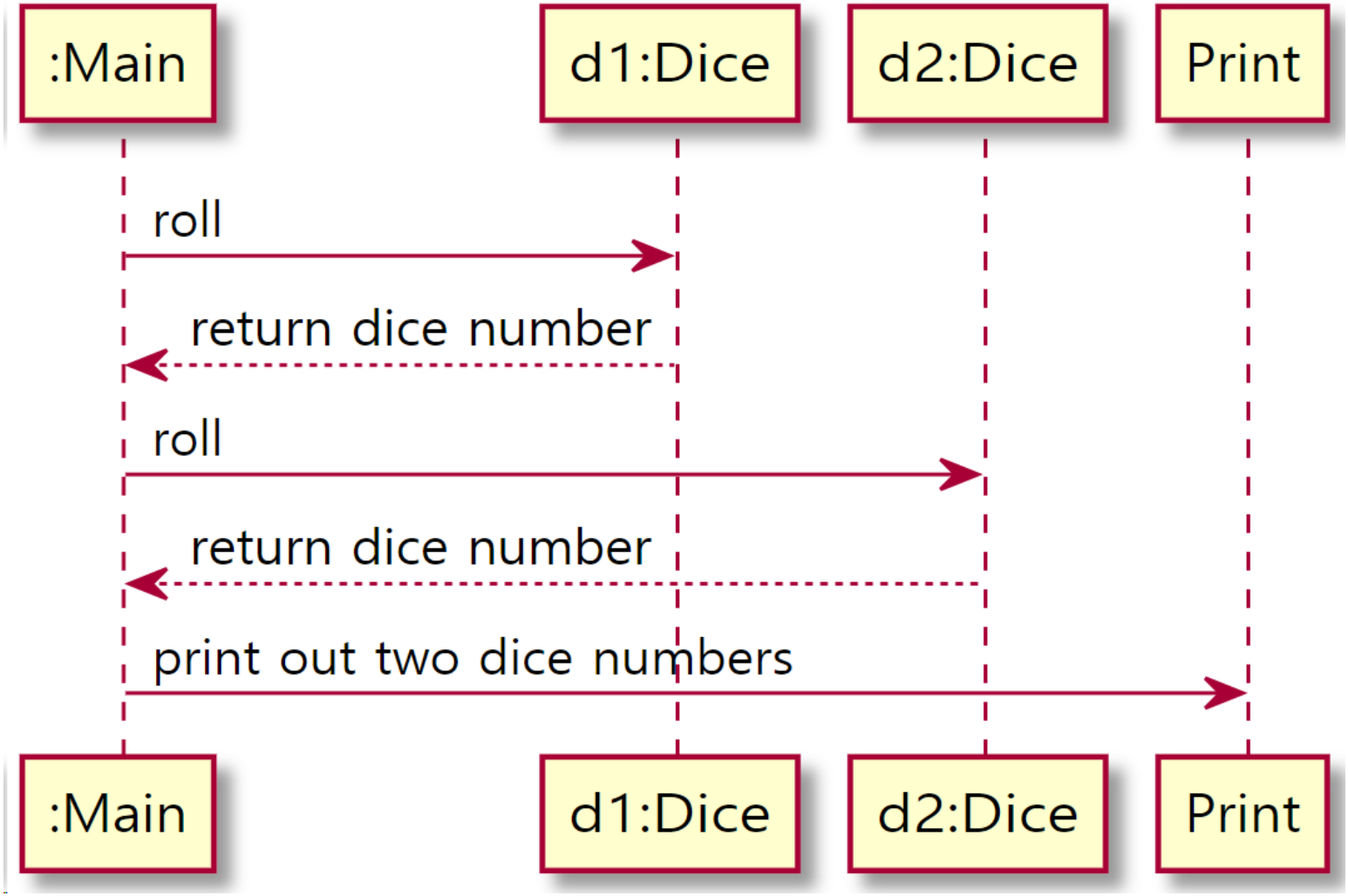


- Composition 합성
  - A 가 B 를 포함하고 있을 때, B 가 A 없이는 존재할 수 없다면 합성 관계



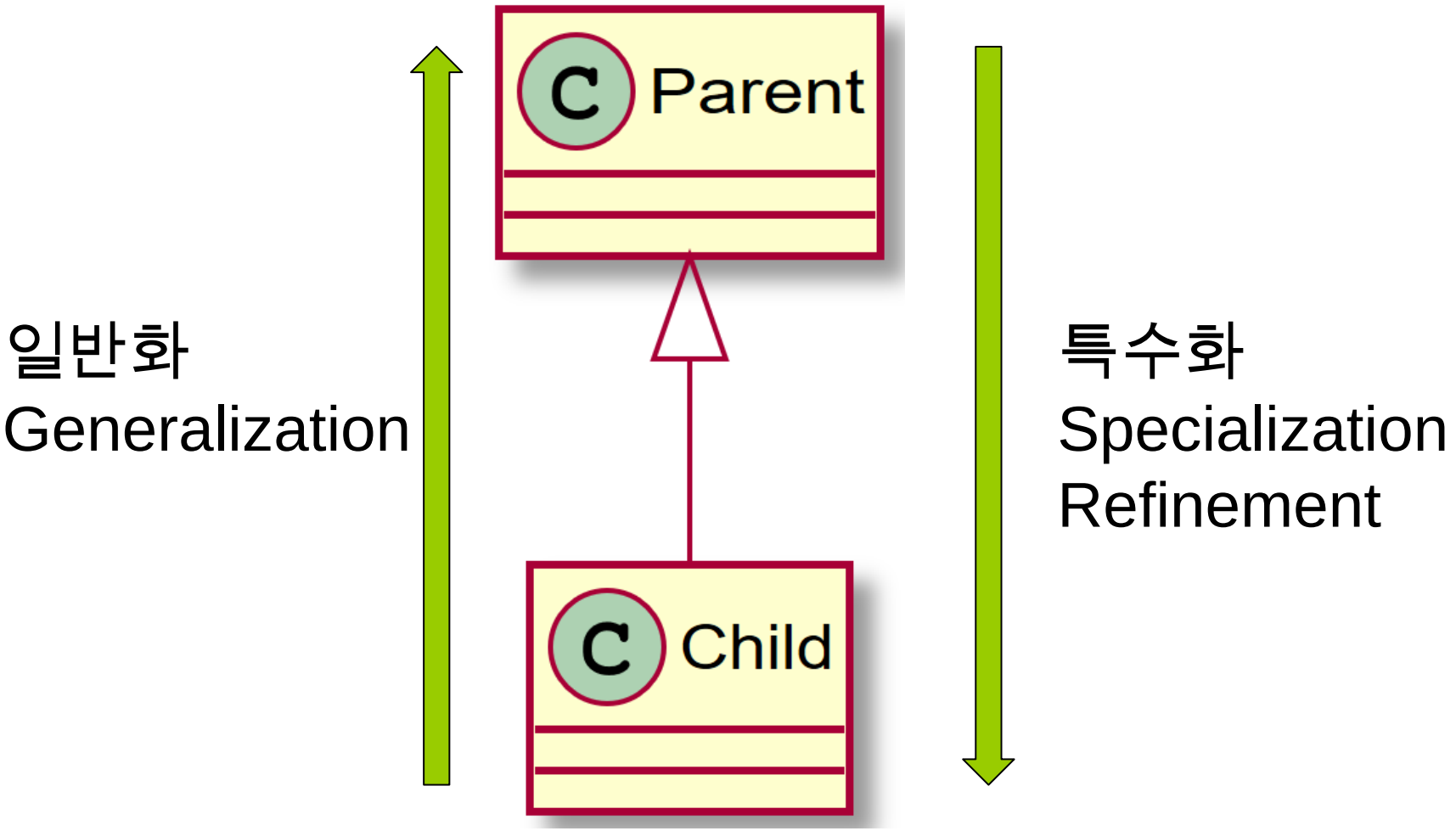
UML 시퀀스 다이어그램 (Sequence Diagram)

- 시퀀스 다이어그램



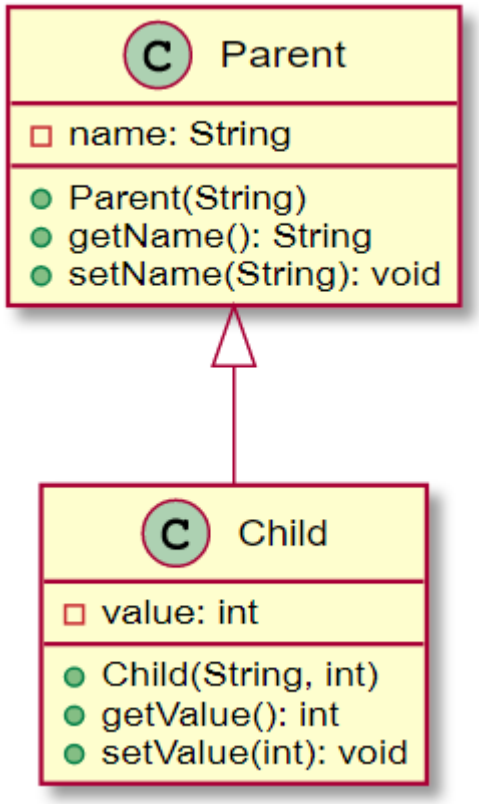
상속

- 상속 (inheritance, 확장 , 특수화)



상속

상속



- 왼쪽 그림은 Child 가 Parent 로부터 상속받은 것을 보임
- 생성자는 상속 안됨

Parent 의 메모리 구조

String name
Parent (String n)
String getName()
void setName(String n)

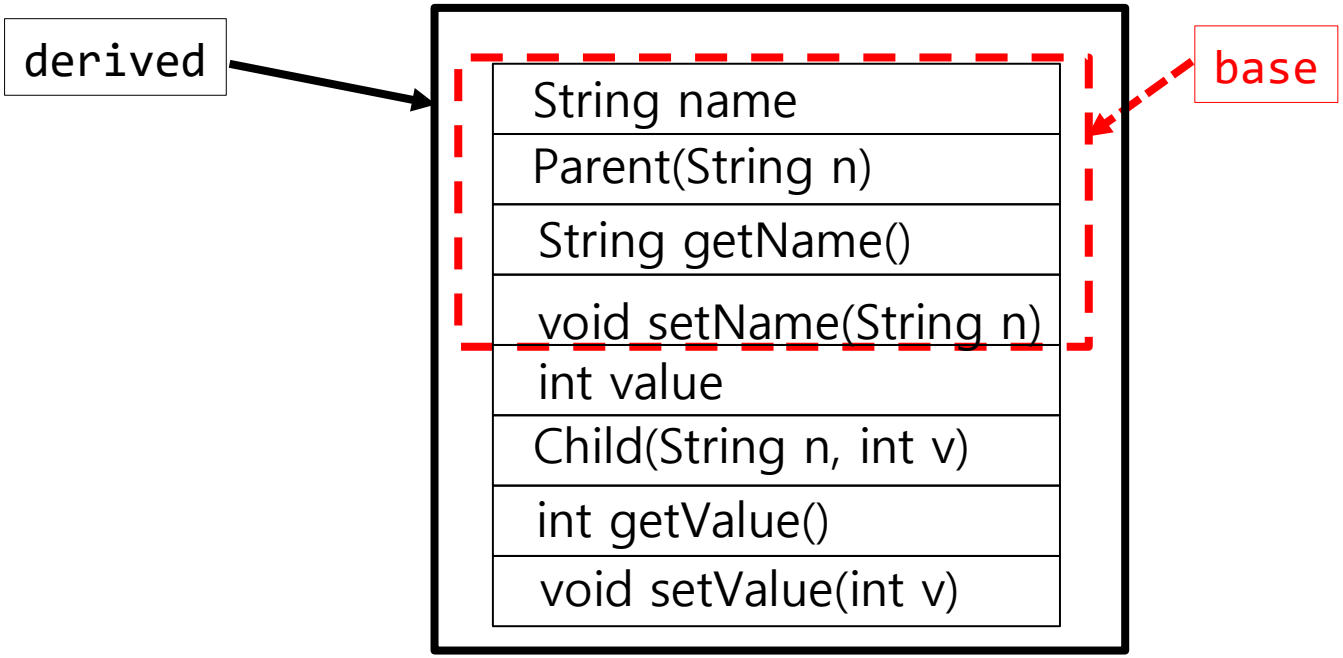
Child 의 메모리 구조

String name
Parent(String n)
String getName()
void setName(String n)
int value
Child(String n, int v)
int getValue()
void setValue(int v)

상속

- 자식 객체는 부모 변수에 저장 가능 (업캐스트 upcast)
- 부모 변수에 자식 객체를 저장했을 때 부모 객체에 있는 내용만 사용 가능

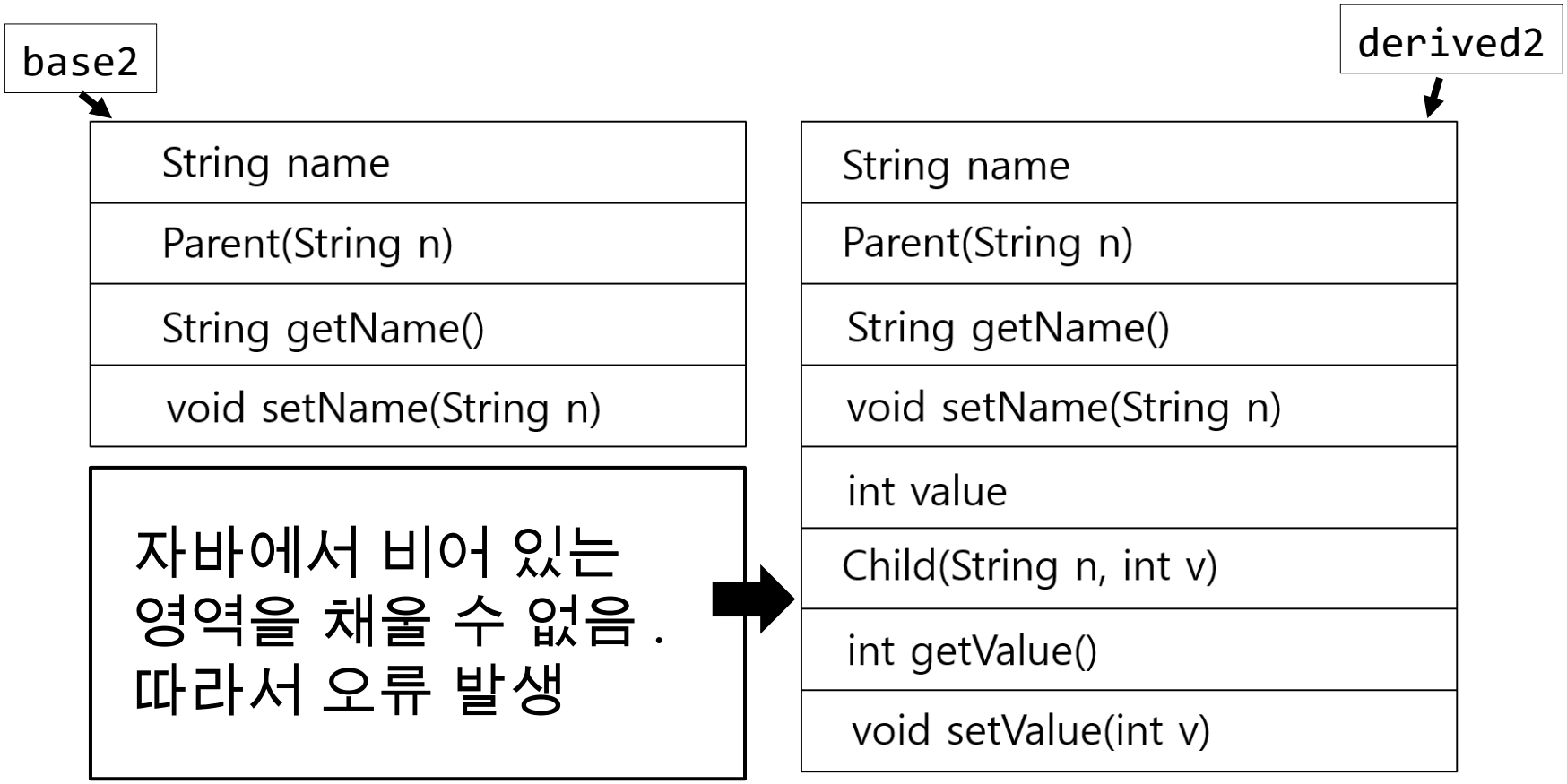
```
Parent base;
Child derived = new Child("cho", 2019);
base = derived;
```



상속

- 부모 객체를 자식 변수에 저장 못함

```
Parent base2 = derived;
Child derived2 = base2; //  오류 발생
```



상속

- 형 변환 (type casting) 을 사용하면 전체 사용 가능 (다운캐스트 downcast)
  - 단 원본 객체가 자식이어야 함

```
Child derived2 = (Child) base2;
System.out.println(derived2->getValue());
Child derived3 = (Child) base; //  오류 발생
```

- instanceof 연산자
  - 좌측 (참조값) 이 우측 (클래스 또는 하위) 의 객체이면 true, 아니면 false 가 반환됨
  - 상속 관계에 있는 경우 자식 클래스 객체는 부모 클래스의 객체로도 판별되기도 함
  - derived2 instanceof Child → true
  - derived2 instanceof Parent → true

상속

- 오버로딩과 오버라이딩
  - 오버로딩 (overloading)
    - 함수의 매개 변수의 개수나 종류가 다름
    - 반환값의 종류 (return type) 는 의미 없음
    - 같은 클래스 또는 상속 관계의 클래스에서 유효함

```
void print() { … }
void print(String s) { … }
void print(int n) { … }
void print(String s, int n) { … }
int print(int n) { … } //  오류
```

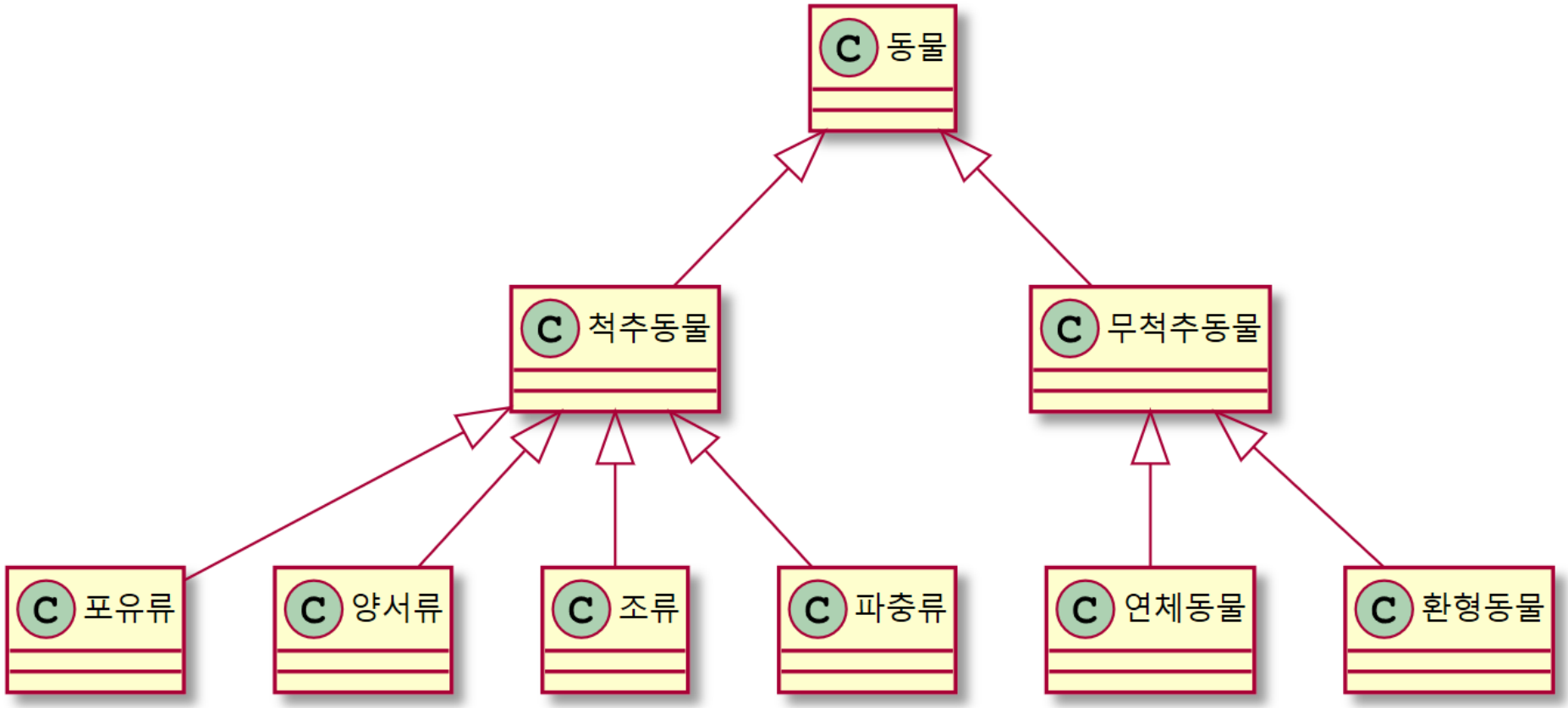
상속

- 오버라이딩 (overriding)
  - 함수의 시그니처가 같음
  - 상속 관계에서만 의미 있음

```
class Parent {
    void print() { … }
    void print(String s, int n) { … }
}

class Child extends Parent {
    void print() { … }
    void print(String s, int n) { … }
}
```

상속



상속 구현 예제

```
// Parent.java
class Parent {
    private String name;
    public Parent (String n) {
        name = n;
    }
    public String getName() { return name; }
    public void setName(String n) {
        name = n;
    }
}
```

상속 구현 예제

```
// Child.java
class Child extends Parent {
    private int value;
    public Child (String s, int n) {
        super(s);
        value = n;
    }
    public int getValue() { return value; }
    public void setValue(int n) {
        value = n;
    }
}
```

상속 구현 예제

```
class Main {
    public static void main(String[] args) {
        Parent base = new Parent("ycho");
        Child derived = new Child("cho", 2019);
        System.out.println(base.getName());
        System.out.println(derived.getValue());
        Parent base2 = derived;
        System.out.println(base2.getName());
        System.out.println(base2.getValue());//  오류 발생
        Child derived2 = base2; //  오류 발생
    }
}
```

상속 구현 예제

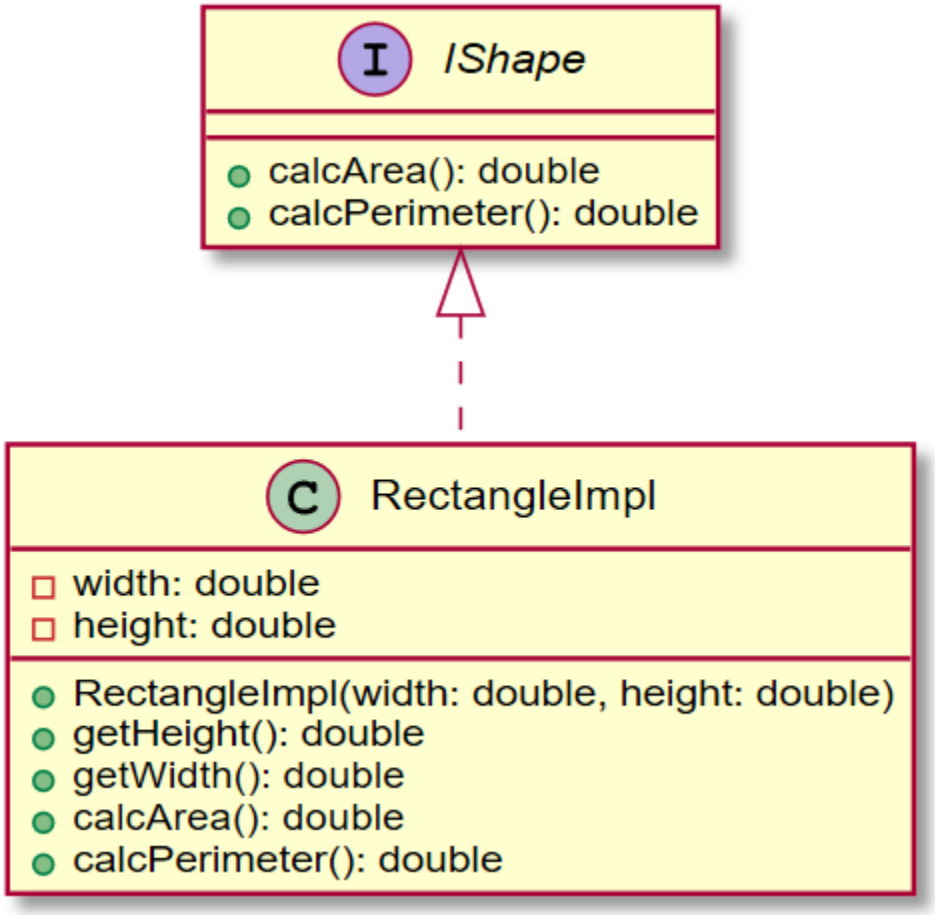
```
class ChildPlus1 extends Child {
    public ChildPlus1(String s, int n) {
        super(s, n);
    }
    @Override
    public int getValue() {
        return super.getValue() + 1;
    }
}
```

인터페이스 (Interface) 와 추상 클래스 (Abstract class)

- 인터페이스
  - 구현할 클래스의 함수 시그니처 (signature) 만 정해 놓음
  - 자바 8 부터는 디폴트 메소드 (미리 구현된 함수) 추가 가능
  - 인터페이스를 구현하는 클래스에서는 함수 시그니처만 있는 것을 구현해야 함
  - 인터페이스의 모든 함수는 public 접근 제어자로 지정됨

- 추상 클래스
  - 멤버 변수 포함 가능
  - 미리 구현된 함수 포함 가능
  - 한 개 이상의 abstract 함수 포함 (함수의 시그니처만 포함)

인터페이스 구현 예제


IShape
calcArea(): double
calcPerimeter(): double
RectangleImpl
width: double
height: double
RectangleImpl(width: double, height: double)
getHeight(): double
getWidth(): double
calcArea(): double
calcPerimeter(): double

인터페이스 구현 예제

```
// IShape.java
interface IShape {
    public double calcArea();
    public double calcPerimeter();
}

// RectangleImpl.java
class RectangleImpl implements IShape {
    private double width, height;
    public RectangleImpl(double width,
                         double height) {
        this.width = width;
        this.height = height;
    }
```

인터페이스 구현 예제

```
    @Override
    public double calcArea() {
        return width * height;
    }
    @Override
    public double calcPerimeter() {
        return 2 * (width + height);
    }
    public double getHeight() { return height; }
    public double getWidth() { return width; }
}
```

인터페이스 구현 예제

```
// RectangleMain.java
class RectangleMain {
    public static void main(String[] args) {
        IShape r = new RectangleImpl(10., 20.);
        System.out.println(r.calcArea());
    }
}
```

인터페이스 구현 예제 – 디폴트 메소드

```
//IValue.java
interface IValue {
    default public int getValue() { return 0; }
}

// ValueImpl1.java
class ValueImpl1 implements IValue {
    private String name = "ValueImpl1";
    ValueImpl(String s) { name = s; }
    public String getName() { return name; }
    public void setName(String s) {
        name = s;
    }
}
```

인터페이스 구현 예제 – 디폴트 메소드

```
// ValueImpl2.java
class ValueImpl2 implements IValue {
    private String name;
    ValueImpl2() {
        name = "ValueImpl2";
    }
    public String getName() { return name; }
    public void setName(String s) {
        name = s;
    }
    public int getValue() { return 1; }
}
```

인터페이스 구현 예제 – 디폴트 메소드

```
// ValueMain.java
class ValueMain {
    public static void main(String[] args) {
        ValueImpl1 v1 = new ValueImpl1("ValueImpl1");
        ValueImpl2 v2 = new ValueImpl2();
        System.out.println(v1.getName());
        System.out.println(v2.getName());
        IValue i1 = v1;
        IValue i2 = v2;
        System.out.println(i1.getValue()); // 0
        System.out.println(i2.getValue()); // 1
    }
}
```

추상 클래스 구현 예제

```
// Shape.java
abstract class Shape {
    public abstract double calcArea();
    public abstract double calcPerimeter();
}

// Rectangle.java
class Rectangle extends Shape {
    private double width, height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
```

추상 클래스 구현 예제

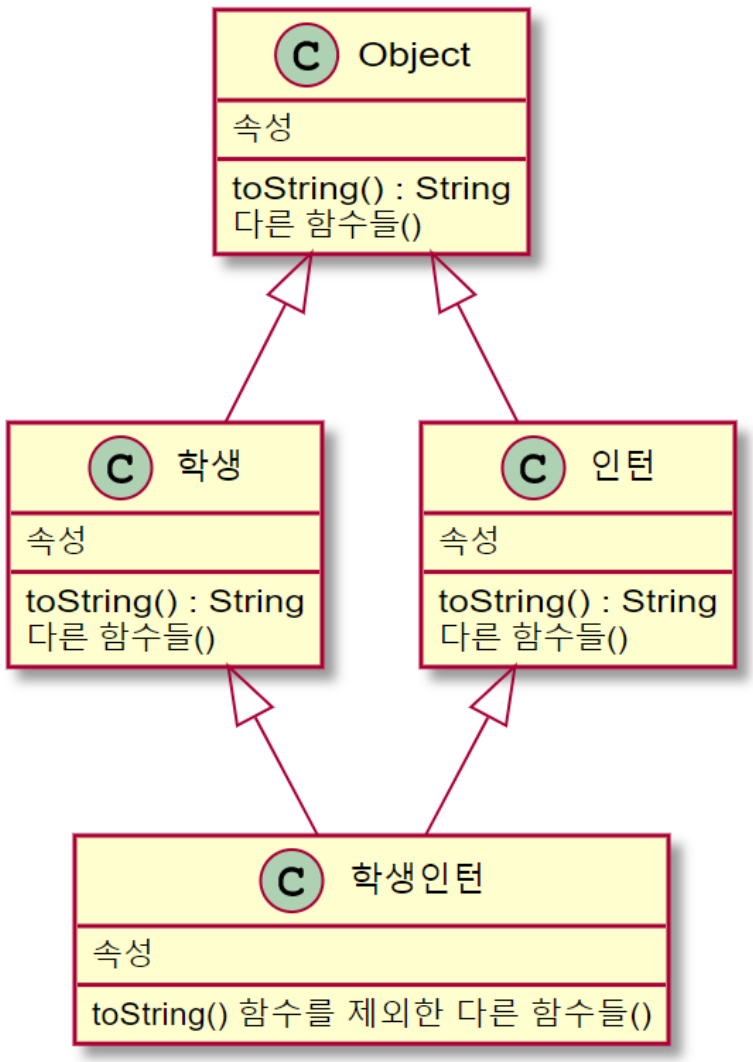
```
    @Override
    public double calcArea() {
        return width * height;
    }
    @Override
    public double calcPerimeter() {
        return 2 * (width + height);
    }
    public double getHeight() { return height; }
    public double getWidth() { return width; }
}
```

```
public class Circle extends Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double calcArea() {
        return Math.PI * radius * radius;
    }
    @Override
    public double calcPerimeter() {
        return 2 * Math.PI * radius;
    }
    public double getRadius() { return radius; }
}
```

추상 클래스 구현 예제

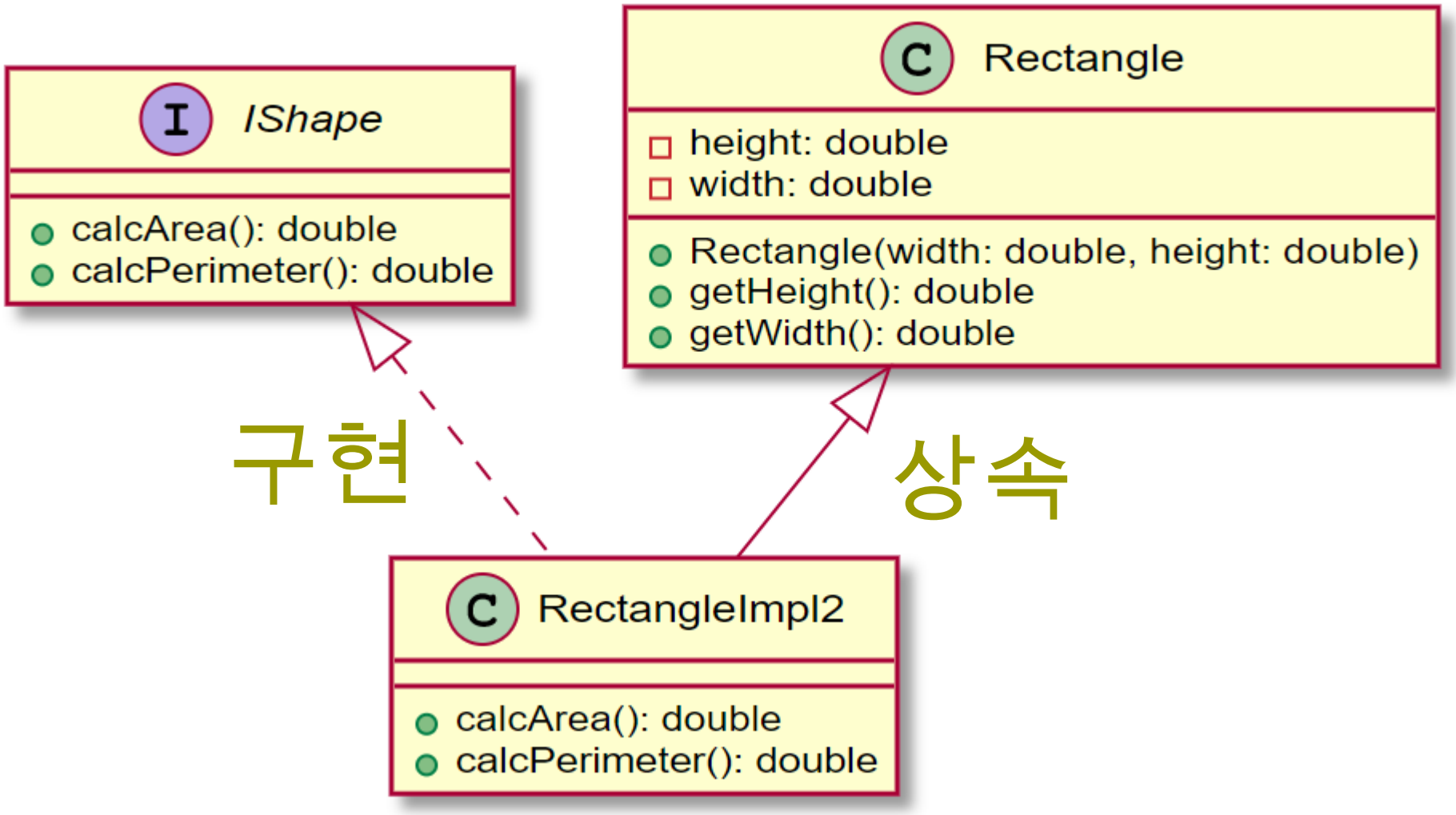
```
// AbstractShapeMain.java
class AbstractShapeMain {
    public static void main(String[] args) {
        Rectangle r = new Rectangle(20.0, 10.0);
        Circle c = new Circle(10);
        System.out.printf(" 사각형의 면적 : %.2f\
n",
                          r.calcArea());
        System.out.printf(" 원의 둘레 : %.2f\n",
                          c.calcPerimeter());
    }
}
```

다중 상속



- 다이아몬드 문제 발생 가능
- 자바는 다중 상속 지원 안함
- 대신 다중 상속 효과를 인터페이스를 이용해서 구현 가능

다중 상속


IShape
calcArea(): double
calcPerimeter(): double
Rectangle
height: double
width: double
Rectangle(width: double, height: double)
getHeight(): double
getWidth(): double
구현
상속
RectangleImpl2
calcArea(): double
calcPerimeter(): double

다중 상속

```
interface IShape {
    public double calcArea();
    public double calcPerimeter();
}
class Rectangle {
    private double width, height;
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    public double getHeight() { return height;  }
    public double getWidth() { return width;  }
}
```

```
// RectangleImpl2.java
class RectangleImpl2 extends Rectangle
                                implements IShape {
    public RectangleImpl2(double width,
                          double height) {
        super(width, height);
    }
    @Override
    public double calcArea() {
         return getWidth() * getHeight();
    }
    @Override
    public double calcPerimeter() {
        return 2 * (getWidth() + getHeight());
    }
}
```

인터페이스 구현 예제 – 디폴트 메소드

```
interface IValue {
    public int getValue() { return 0; }
}

class ValueImpl {
    private String name = "ValueImpl";
    ValueImpl() { }
    ValueImpl(String s) { name = s; }
    public String getName() { return name; }
    public void setName(String s) {
        name = s;
    }
}
```

인터페이스 구현 예제 – 디폴트 메소드

```
// ValueImpl1.java
class ValueImpl1 extends ValueImpl implements IValue {
    ValueImpl1(String s) {
        super(s);
    }
}
// ValueImpl2.java
class ValueImpl2 extends ValueImpl implements IValue {
    ValueImpl2() {
        super();
        setName("ValueImpl2");
    }
    public int getValue() { return 1; }
}
```

인터페이스 구현 예제 – 디폴트 메소드

```
// ValueMain.java
class ValueMain {
    public static void main(String[] args) {
        ValueImpl v1 = new ValueImpl1("ValueImpl1");
        ValueImpl v2 = new ValueImpl2();
        System.out.println(v1.getName());
        System.out.println(v2.getName());
        IValue i1 = v1;
        IValue i2 = v2;
        System.out.println(i1.getValue()); // 0
        System.out.println(i2.getValue()); // 1
    }
}
```

객체 지향 프로그래밍의 특성

- 다형성 (polymorphism)
  - 동적 함수 호출
  - '컴파일 시점'에 '선언'된 클래스와 다른 객체를 '실행 시점'에 사용하는 것
  - 부모 클래스로 선언된 변수에서 자식 클래스에서 오버라이딩 된 함수 호출 가능

```
public class ShapeTag {
   private String tag;
   public ShapeTag(String tag) {
       this.tag = tag;
   }
   public String toString() { return "#" + tag; }
   }
```

```
public class RectangleTag extends ShapeTag {
    private String rectangleTag;
    public RectangleTag(String tag,
                        String rectangleTag) {
        super(tag);
        this.rectangleTag = rectangleTag;
    }
    @Override
    public String toString() {
        return "#" + rectangleTag + " "
               + super.toString();
    }
    public String getRectangleTag() {
        return rectangleTag;
    }
}
```

```
public class CircleTag extends ShapeTag {
    private String circleTag;
    public CircleTag(String tag,
                     String circleTag) {
        super(tag);
        this.circleTag = circleTag;
    }
    @Override
    public String toString() {
        return "#" + circleTag + " "
               + super.toString();
    }
    public String getCircleTag() {
        return circleTag;
    }
}
```

```
public class CircleTag extends ShapeTag {
    private String circleTag;
    public CircleTag(String tag,
                     String circleTag) {
        super(tag);
        this.circleTag = circleTag;
    }
    @Override
    public String toString() {
        return "#" + circleTag + " "
               + super.toString();
    }
    public String getCircleTag() {
        return circleTag;
    }
}
```

다형성

```
ShapeTag s1 = new ShapeTag("shape1");
ShapeTag s2 = new ShapeTag("shape2");
RectangleTag r =
        new RectangleTag("shape", "rectangle");
CircleTag c = new CircleTag("shape", "circle");
System.out.println("Shape1 Tag: " + s1);
System.out.println("Shape2 Tag: " + s2);
System.out.println("Rectangle Tags: " + r);
System.out.println("Circle Tags: " + c);
```

```
Shape1 Tag: #shape1
Shape2 Tag: #shape2
Rectangle Tags: #rectangle #shape
Circle Tags: #circle #shape
```

다형성

```
s1 = r;
s2 = c;

System.out.println("Rectangle Tags: " + s1);
System.out.println("Circle Tags: " + s2);
```

```
Rectangle Tags: #rectangle #shape
Circle Tags: #circle #shape
```

도형 면적을 다형성을 이용해서 계산

- 도형의 종류에 따라 면적 계산 방법이 다르다 . 다형성을 이용한 객체 지향 방식으로 프로그래밍 한 것과 아닌 방식을 비교
  - 부모클래스 Shape[] 에 자식클래스를 저장하고 (예 : Circle, Rectangle)
  - 부모클래스인 Shape.calcArea() 를 호출하면 자식클래스 Rectangle.calcArea() 또는 Circle.calcArea() 가 실행되는 것
- 버전 1: instanceof 연산자를 이용해서 클래스를 구분하고 , 이를 이용해서 calcArea() 함수를 호출
- 버전 2: 다형성을 이용해서 호출

```
Rectangle r = new Rectangle(3, 4);
Circle c = new Circle(5);
Shape[] shapes = new Shape[2];
shapes[0] = r;
shapes[1] = c;
```

- 버전 1 – if 문으로 구별

```
for (Shape shape : shapes) {
    if (shape instanceof Rectangle) {
        System.out.println(shape.calcArea());
    }
    else if (shape instanceof Circle) {
        System.out.println(shape.calcArea());
    }
}
```

도형 면적을 다형성을 이용해서 계산

- 만약 Triangle 이란 클래스가 새로 추가된다면 ?

```
for (Shape shape : shapes) {
    if (shape instanceof Rectangle) {
        System.out.println(shape.calcArea());
    }
    else if (shape instanceof Circle) {
        System.out.println(shape.calcArea());
    }
    else if (shape instanceof Triangle) {
        System.out.println(shape.calcArea());
    }
}
```

도형 면적을 다형성을 이용해서 계산

- 버전 2 – 다형성 이용

```
for (Shape shape : shapes)
    System.out.println(shape.calcArea());
```

- 만약 Triangle 이란 클래스가 새로 추가된다면 ?

제네릭스 (Generics)

- ArrayList 자료구조 사용
  - 배열과 비슷하게 인덱스를 이용해서 요소에 접근할 수 있는 자료구조
  - 배열과는 달리 저장할 수 있는 요소의 개수가 가변적
  - 배열을 기본형과 객체를 모두 저장할 수 있지만 , ArrayList 는 객체만 저장할 수 있음
    - Object 클래스 또는 그 후손 클래스 객체만 저장 가능

```
ArrayList list = new ArrayList();
list.add("Seoul");
list.add(new String("Tokyo"));
list.add(new Integer(3));
list.add(5); // boxing
```

제네릭스 (Generics)

- 기존 ArrayList 의 문제점
  - 어떤 종류의 객체든 저장 가능
  - 객체를 사용할 때 형 변환 (type cast) 필요 – 다운 캐스팅
  - 어디에 어떤 클래스형 자료가 있는지 정확하게 기억해야 함
  - 대부분은 단일 자료형만 저장

```
String s1 = list.get(0); //  오류 발생
String s2 = (String) list.get(1);
String s3 = (String) list.get(2); //  예외 발생
Integer i1 = (Integer) list.get(2);
int i2 = (Integer) list.get(3);
```

제네릭스 (Generics)

- ArrayList 에 단일 자료형을 저장할 때 효율적으로 사용하는 방법이 제네릭스를 사용하는 것
- 제네릭스 사용법
  - 자료구조 < 자료형 >

```
ArrayList<String> list = new ArrayList<String>();
// ArrayList<String> list = new ArrayList<>();
list.add("Seoul");
list.add(new String("Tokyo"));
list.add(new Integer(3)); //  오류 발생
String s = list.get(0); //  형 변환 필요 없음
```

제네릭스 (Generics)

- 기존 ArrayList 를 이용해서 제네릭스 ArrayList 만들기

```
class MyArrayList<E> {
    ArrayList list;
    public MyArrayList() {
        list = new ArrayList();
    }
    public void add(E e) {
        list.add(e);
    }
    public E get(int i) {
        return (E) list.get(i);
    }
}
```

제네릭스 (Generics)

```
MyArrayList<String> l = new MyArrayList<String>();
l.add("temp");
l.add("add");
String s = l.get(0);
```

키워드

- SW 위기와 좋은 소프트웨어
- 절차 중심과 객체 지향 프로그래밍 차이
- 클래스와 객체
- 객체 지향 프로그래밍의 특성
  - 추상화
  - 캡슐화
  - 상속
    - 추상클래스
    - 인터페이스
  - 다형성
- 제네릭스