

---

# 7장. 상속

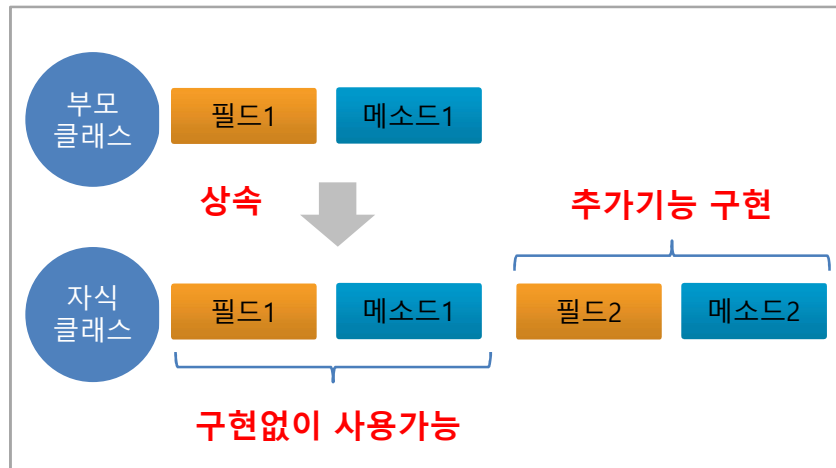
## 7.1. 상속개념

- 상속은 부모클래스의 기능을 자식클래스가 바로 사용할 수 있는 것을 의미함
- 상속은 extends 키워드를 통해서 구현 가능(주의사항 : 2개 이상의 클래스는 안됨)

### 상속 개념 및 사용법

#### 상속 개념

❖ 부모가 가지고 있는 기능을 자식에게 물려줌



❖ 상속의 효과

- 부모의 클래스를 재사용함으로써 개발의 속도 증가
- 반복된 코드의 중복을 줄여줌
- 유지보수의 편의성 제공
- 객체의 다형성을 구현 가능

#### 상속의 사용법

❖ extends 키워드로 상속가능

```

public class Car {
    private int currentSpeed;

    public Car (int currentSpeed) {
        this.currentSpeed = currentSpeed;
    }
    int speedUp() {
        // 속도 1증가 시킴
        return (this.currentSpeed + 1);
    }
}
  
```

Car class

```

public class SportsCar extends Car {
    public SportsCar (int currentSpeed) {
        this.currentSpeed = currentSpeed; // 부모의 필드 사용
    }
    void turboSpeedUp() {
        // 속도 10증가 시킴
        return (this.currentSpeed + 10);
    }
}
  
```

SportsCar class

외부 class

```

SportsCar sportscar = new SportsCar(100); // 스포츠카 객체 생성
sportscar.speedUp(); // 부모클래스(Car)의 메서드 실행
// 속도 1증가하여 currentSpeed = 101이 됨
sportscar.turboSpeedUp(); // 본인클래스의 메서드 실행
// 속도 10증가하여 currentSpeed = 111이 됨
  
```

## 7.2. 부모생성자의 호출

- JAVA의 모든 클래스는 생성자를 통해 객체가 생성되어야 함.
- 부모생성자는 생략가능하나 컴파일시 **super** 키워드를 강제 생성하여 부모 생성자를 생성함.

### 부모 생성자의 호출

#### 부모생성자 호출 예

```
public class Car {
    private int currentSpeed;
    Car () {}
    Car (int currentSpeed) {
        this.currentSpeed = currentSpeed;
    }
    int speedUp() {           // 속도 1증가 시킴
        return this.currentSpeed + 1;
    }
}
```

Car class

```
public class SportsCar extends Car{
    SportsCar (int currentSpeed) {
        this.currentSpeed = currentSpeed; // 부모의 필드 사용
    }
    int turboSpeedUp() {           // 속도 10증가 시킴
        return this.currentSpeed + 10;
    }
}
```

SportsCar class

#### 외부 class

```
SportsCar sportscar = new SportsCar(100); // 스포츠카 객체 생성
sportscar.speedUp();                     // 부모클래스(Car)의 메서드 실행
// 속도 1증가하여 currentSpeed = 101이 됨
sportscar.turboSpeedUp();                 // 본인클래스의 메서드 실행
// 속도 10증가하여 currentSpeed = 111이 됨
```

- ❖ 부모생성자는 자식생성자 생성시 자동으로 호출됨.  
부모생성자가 생성됨으로써 자식객체에서  
부모객체의 멤버변수와 메서드 호출이 가능해짐

```
public class SportsCar extends Car{
    SportsCar (int currentSpeed) {
        super(int currentSpeed);
        // 부모생성자 호출. 부모 객체의 생성
        this.currentSpeed = currentSpeed;
    }
    int turboSpeedUp() {
        return this.currentSpeed + 10;
    }
}
```

SportsCar class

## 7.3. 메서드 재정의 (Overriding)

- 부모클래스의 메서드가 자식에게 맞지 않을 경우 재정의의 가능함.
- 이렇게 메서드를 재정의 하는 것을 오버라이딩(Overriding)이라 함.

### 메서드 overriding

#### 메서드 재정의

- ❖ 상속받은 자식클래스 입장에서 부모클래스의 메서드를 수정하여 사용해야 할 경우 메서드 재정의의 가능함.

```
public class Car {
    private int currentSpeed;

    Car (int currentSpeed) {
        this. currentSpeed = currentSpeed;
    }
    int speedUp() {
        // 속도 1증가 시킴
        return this. currentSpeed + 1;
    }
}
```

Car class

```
public class SportsCar extends Car{
    SportsCar (int currentSpeed) {
        this. currentSpeed = currentSpeed; // 부모의 필드 사용
    }
    int SpeedUp() {
        return this. currentSpeed + 10;
        // 메서드 재정의 (스포츠카는 10단위로 속도 증가함)
    }
}
```

SportsCar class

외부 class

```
SportsCar sportscar = new SportsCar(100); // 스포츠카 객체 생성
sportscar.speedUp(); // 재정의 된 메서드 실행 (속도 110됨)
```

재정의된  
메서드 실행

## 7.4. 상속과 관련한 접근제한자 (Protected)

- **protected** 접근제한자는 상속된 클래스에 한해서는 어디에서나 접근 가능함
- **protected** 접근제한자는 생성자, 필드, 메서드에 적용됨

### protected 접근 제한자

#### 사용문법

접근제한	설 명
public	- 모든 package에서 생성자 생성 가능
<b>protected</b>	- 동일 package에서 생성자 생성 가능 - 단, 다른 package라도 동일 package 안의 클래스를 상속받았을 때는 생성자 생성 가능
default	- 동일 package에서 생성자 생성 가능 - 접근제한자 생략한 형태로 사용함
private	- 동일 class 내에서만 접근 가능. - 외부에서는 접근 불가

#### A Package

##### class A

- **protected** 멤버

##### class B

- A 클래스의 멤버 접근 가능  
(동일 package이므로)

#### B Package

##### class C

- A 클래스의 멤버 접근 불가(X)  
(다른 package이므로)

##### class D extends A

- A 클래스의 멤버 접근 가능(O)  
\* 다른 package이지만  
상속으로 인해 접근 가능

#### 사용사례 및 주의사항

```
package A;
public class A {
    protected int field ;      // protected로 변수 선언
    protected void method() { ... } // protected로 메서드 정의
}
```

```
package A;      // 접근하려는 클래스와 동일 package
public class B {
    A a = new A();
    a.field = 10;      // (O) 동일 package 이므로
    a.method();        // (O) 동일 package 이므로
}
```

```
package B;      // 접근하려는 클래스와 다른 package
public class C {
    A a = new A();      // (X) 에러발생.
    a.field = 10;      // (X) 접근 불가. 다른 패키지.
    a.method();        // (X) 접근 불가. 다른 패키지.
}
```

```
package B;      // 접근하려는 클래스와 다른 package
public class D extends A { // A 상속받음
    public D() { this.field = 100; }
    D d = new D();
    d.method();
}
```

## 7.5. 타입변환 / 다형성

- 상속관계에 있는 객체는 객체간의 타입의 변환이 가능함
- 타입변환은 자동타입변환과 강제타입변환이 있음.

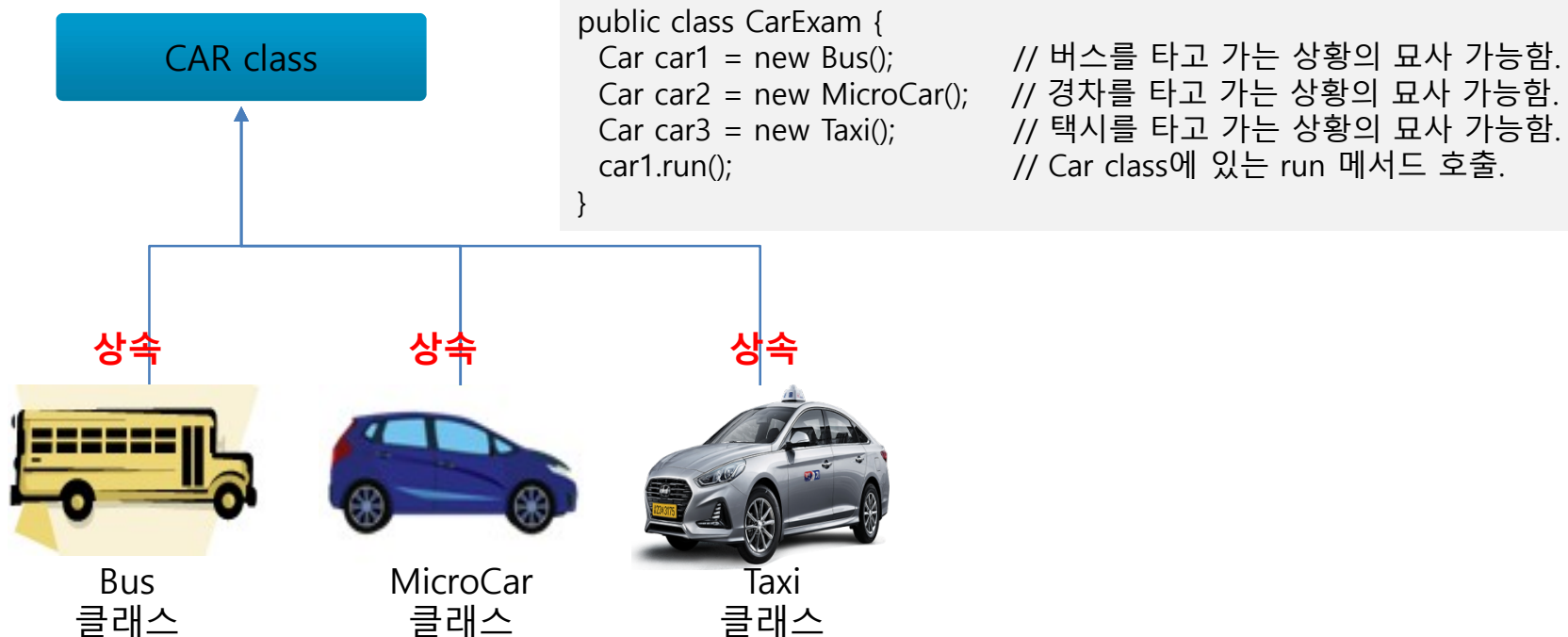
### 타입변환의 개념

```
public class CarExam {
    Bus bus1= new Bus();           // 버스를
    MicroCar() mcar1 = new MicroCar();

    bus1.run();                     // Car cl
    mcar1.run();
}
```

❖ 출근하는 상황을 프로그램하는 것을 가정해 봄.

어떤 Car를 타고 가는지에 따라서 거리 및 시간, 그리고 요금이 다르게 나오는 상황인데  
그 때마다 다르게 프로그램하는 것이 아니라 타입의 변환을 활용하면 구현이 쉬워짐.



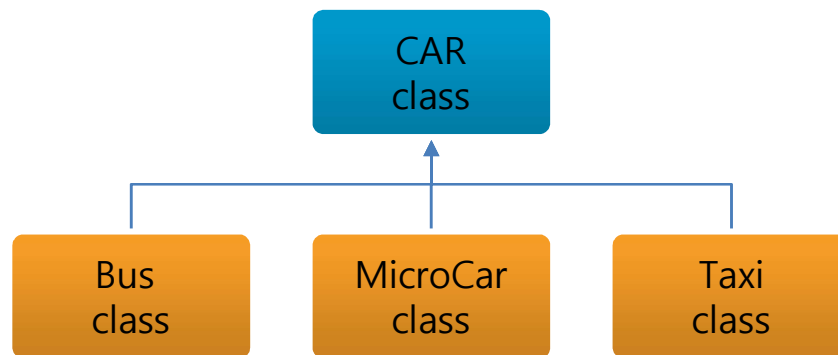
## 7.5.1. 자동타입변환

- 자동타입변환은 프로그램 실행 도중 자동적으로 타입변환이 일어나는 것을 의미함
- 자식클래스의 객체를 부모클래스 객체에 저장할 때 부모의 타입으로 변환되는 것을 의미

### 자동 타입 변환

#### 자동타입변환 형식

❖ 부모클래스 변수에 자식클래스 타입이 대입될 때 발생.



대입시 자동타입변환 일어남

❖ 문법 : 부모클래스 변수 = 자식클래스 타입

❖ 예시

- Car car = new Bus();

❖ 상황이 바뀌면 new 연산자 부분의 자식클래스만 변경하면 됨.

#### 사용예제

❖ 자식클래스의 메서드 오버라이딩시 자식메서드 호출

```
public class Parent {
    void method 1( ){ }
    void method 2( ){ }
}
```

Parent class

```
public class Child extends Parent {
    void method 2( ){ } // 메서드 오버라이드
    void method 3( ){ } // 자식객체만의 메서드
}
```

Child class

```
Parent parent = new Child( ); // 자동타입변환
parent.method1( ) // 호출가능
parent.method2( ) // override된 Child의 method 실행됨
parent.method3( ) // 호출불가. Parent에 없는 메서드
```

외부 class

## 7.5.2. 자동타입변환 다형성 응용

- 다형성 응용은 메서드를 호출시 오버라이딩 된 메서드를 활용하는 상황
- 동일한 메서드를 호출하지만 생성된 객체별로 다른 형태의 동작을 수행함.

### 다형성 응용

#### 버스를 타고 출근할 때

```
public class Car {
    int getCharge() {
        System.out.println ("기본요금 : 1600원" );
        return 1600;
    }
}
```

Parent class

```
public class Bus extends Car {
    int getCharge() { // 오버라이딩
        System.out.println ("버스요금 : 1450원" );
        return 1450;
    }
}
```

Child class

```
int money;
Car car = new Bus( ); // Bus 객체에서 타입변환
money = car.getCharge( ); // Bus 클래스 메서드 호출
// money에는 1450이 입력됨.
```

외부 class

#### 택시를 타고 출근할 때

❖ 택시클래스 추가 및 객체 변경만으로 프로그램 완성가능.

```
public class Car {
    int getCharge() {
        System.out.println ("기본요금 : 1200원" );
        return 1200;
    }
}
```

Parent class

```
public class Taxi extends Car {
    int getCharge() { // 오버라이딩
        System.out.println ("택시요금 : 8000원" );
        return 8000;
    }
}
```

Child class

**Taxi 클래스 추가**

**Taxi 객체로 변경**

```
int money;
Car car = new Taxi( ); // Taxi 객체에서 타입변환
money = car.getCharge( ); // Bus 클래스 메서드 호출
// money에는 8000이 입력됨.
```

외부 class



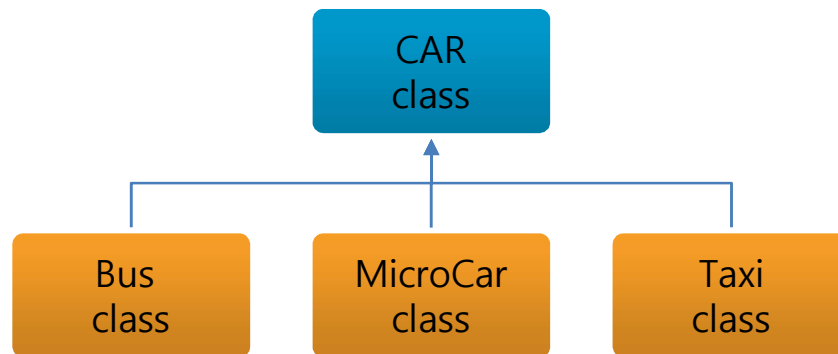
### 7.5.3. 강제타입변환 (Casting)

- 강제 타입변환은 부모타입의 객체를 자식클래스의 객체에 대입하는 것을 의미함.
- 자동 타입변환된 객체에 한해서만 사용 가능함.

#### 강제타입변환 (Casting)

##### 강제 타입변환 형식

- ❖ 자식클래스 변수에 부모클래스 타입이 대입될 때 발생.



##### ❖ 문법

- 자식클래스 변수 = (자식클래스타입) 부모클래스 타입

##### ❖ 예시

```

Car car = new Bus( );    // 자동타입변환
Bus bus = (Bus) car;    // 강제타입변환.
  
```

##### ❖ 주의사항

- 자동타입변환이 일어난 객체에 한해서만 Casting 가능

##### 사용예제

- ❖ 자식클래스의 메서드 오버라이딩시 자식메서드 호출

```

public class Parent {
    void method 1( );
    void method 2( );
}
  
```

Parent class

```

public class Child extends Parent {
    void method 2( ); // 메서드 오버라이드
    void method 3( ); // 자식객체만의 메서드
}
  
```

Child class

외부 class

```

Parent parent = new Child( ); // 자동타입변환
parent.method1( ); // 호출가능
parent.method2( ); // override된 Child method2 실행됨
parent.method3( ); // 호출불가. Parent에 없는 메서드
  
```

```

Child child = (Child) parent; // 강제타입변환
child.method1( ); // 부모클래스 메서드 호출
child.method2( ); // 자식클래스 메서드 호출
child.method3( ); // 자식클래스 메서드 호출
  
```

## 7.5.4. 강제타입변환 활용 (instanceof 키워드)

- 강제 타입변환을 사용할 때는 제한이 있기 때문에 instanceof 를 활용하여 체크로직 활용 가능
- 강제 타입변환이 안되는 상황에서는 ClassCastException 예외 발생함.


### 강제타입변환 활용

#### 객체 타입 체크

- ❖ 부모타입이면 모두 자식타입이 될 수 없음

```
Parent parent = new Parent( );
Child child = (Child) parent;
// 자동타입변환이 일어나지 않은 객체이므로
// 강제타입변환 불가.
```

- ❖ 먼저 자식타입인지 확인 후 강제타입 해야 함.
- parent 객체가 Child 클래스 타입인지 확인함.



```
public void method (Parent parent) {
    if (parent instanceof (Child)) {
        Child child = (Child) parent;
    }
}
```

#### 객체타입체크 예제

- ❖ 택시클래스 추가 및 객체 변경만으로 프로그램 완성가능.

```
public class Car {
    int getCharge() {
        return 1200;
    }
}
```

Parent class

```
public class Taxi extends Car {
    int getCharge() {
        return 8000;
    }
    void print ( )
        System.out.println ("택시를 타고 출근합니다.");
}
```

Child class

```
int money;
Car car = new Taxi( );
money = car.getCharge( );
if (car instanceof Taxi) {
    Taxi t = (Taxi) car;
}
```

외부 class

// Taxi 객체에서 타입변환  
// Bus 클래스 메서드 호출  
// 자식객체여부 판단.

## Exam\_06 - 1 실습

## Buyer

다음은 물건을 구입하는 사람을 정의한 Buyer 클래스이다. 이 클래스는 멤버변수로 돈(money)과 장바구니(cart)를 가지고 있다. 제품을 구입하는 기능의 buy메서드와 장바구니에 구입한 물건을 추가하는 add메서드, 구입한 물건의 목록과 사용금액, 그리고 남은 금액을 출력하는 summary메서드를 완성하시오.

### 1. 메서드명 : buy

기능 : 지정된 물건을 구입한다. 가진 돈(money)에서 물건의 가격을 빼고, 장바구니(cart)에 담는다.  
만일 가진 돈이 물건의 가격보다 적다면 바로 종료한다.

반환타입 : 없음

매개변수 : Product p - 구입할 물건

마지막에 add 메서드 호출

### 2. 메서드명 : add

기능 : 지정된 물건을 장바구니에 담는다.

반환타입 : 없음

매개변수 : Product p - 구입할 물건

### 3. 메서드명 : summary

기능 : 구입한 물건의 목록과 사용금액, 남은 금액을 출력한다.

반환타입 : 없음

매개변수 : 없음

반복문을 통한  
구입물건과 가격 표시  
그리고, 현재잔액 표시

변수

money = 1000 // 잔액  
Product [ ] cart = new Product 10  
Index = 0; // cart index

## Product

Super호출하여  
변수에 저장.

Int price  
String name

TV

Int price = 100  
Name = "TV"

Computer

Int price = 200  
Name =  
"COMPUTER"

Audio

Int price = 50  
Name="AUDIO"

## BuyerExam

```
public static void main(String[] args) {
    int tvPrice = 100;
    int compPrice = 200;
    int audioPrice = 50;
    Buyer buyer1 = new Buyer(1000);

    Tv tv = new Tv(tvPrice, "TV");
    Computer comp = new Computer(compPrice, "COMPUTER");
    Audio audio = new Audio(audioPrice, "AUDIO");

    buyer1.buy(tv);
    buyer1.buy(comp);
    buyer1.buy(tv);
    buyer1.buy(audio);
    buyer1.buy(comp);
    buyer1.buy(comp);
    buyer1.buy(comp);
    buyer1.summary();
}
```

Cart[1] : TV  
Cart[2] : COMPUTER  
Cart[3] : TV  
Cart[4] : AUDIO  
Cart[5] : COMPUTER  
Cart[6] : COMPUTER  
현재 잔액이 150이기 때문에 제품(200)을 살수없습니다.  
구입한 물건 : TV, COMPUTER, TV, AUDIO, COMPUTER, CO  
중 사용금액 : 850  
현재잔액 : 150

## 07. 상속은 언제 사용 할까?

### IS-A 관계(is a relationship : inheritance)

- 일반적인(general) 개념과 구체적인(specific) 개념과의 관계
- 상위 클래스 : 하위 클래스보다 일반적인 개념 ( 예: Employee )
- 하위 클래스 : 상위 클래스보다 구체적인 개념들이 더해짐 ( 예: Engineer, Manager...)
- 상속은 클래스간의 결합도가 높은 설계
- 상위 클래스의 수정이 많은 하위 클래스에 영향을 미칠 수 있음
- 계층구조가 복잡하거나 hierarchy가 높으면 좋지 않음

하위로 정규직, 임시직으로 구분하여 사용가능.

또는 하위로 엔지니어, 관리자 임원 등으로 구분가능.

상위클래스의 변경이 많을 수 있을 경우 주의 요함.

### HAS-A 관계(composition)

- 클래스가 다른 클래스를 포함하는 관계 ( 변수로 선언 )
- 코드 재사용의 가장 일반적인 방법
- Student가 Subject를 포함하는
- Library를 구현할 때 ArrayList 생성하여 사용
- 상속하지 않음

부품과 완성품간의 관계는 조심해서 사용해야 함.  
자동차 클래스와 타이어 클래스는 상속관계보다는  
자동차 클래스내의 Tire를 멤버변수로 가지는 것이 좋음.  
(단, 타이어클래스 내의 특정회사로 구분할 때는 상속관계 좋음)

코드 재사용의 일반적인 방법이긴하나.

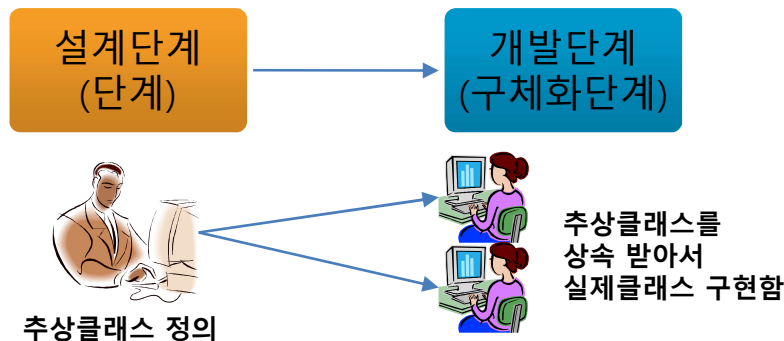
## 7.6.1. 추상클래스 (abstract class) ↔ Concrete Class (구체화된 클래스)

- 추상클래스는 실체들간에 공통된 특성을 추출한 Class를 의미함
- 아직 구체화한 것이 아니므로 부모클래스로써만 사용가능. (직접 객체생성 불가)

### 추상클래스

#### 추상클래스 사용이유

- ❖ 실체클래스의 공통된 필드와 메서드의 이름을 통일할 목적으로 사용됨.
  - 설계자가 클래스 설계시 필드와 메서드 정의함. 클래스가 필수적으로 가져야 할 내용을 정의함
  - 개발자는 실제 구현할 때 정의된 필드와 메서드를 기본적으로 사용해야 함



- ❖ 실체클래스 작성시 시간 절약 가능
  - 실체클래스는 추가적인 필드, 메서드만 정의하면 됨

#### 추상클래스 사용예제

```
public abstract class Car {
    int currentSpeed = 100;
    public void driving ( ) {
        System.out.println ("달리다");
    }
    public void stop ( ) {
        System.out.println ("브레이크를 밟다");
    }
}
```

추상 class

자동차의  
공통특징만 정의함

```
public class SportsCar extends Car{
    public int speedUp( ) {
        return ++currentSpeed;
    }
}
```

실제 class

. Car를 상속받아 사용  
. 스포츠카의 추가적인  
기능만 구현

외부 class

```
Car car = new Car ( ); //(X)추상클래스는 객체 생성 안됨
SportsCar sc = new SportsCar( );
sc.driving( );           // 추상클래스의 메서드 실행
speed = sc.speedUp( );   // 실제클래스의 메서드 실행
sc.stop( );              // 추상클래스의 메서드 실행
```

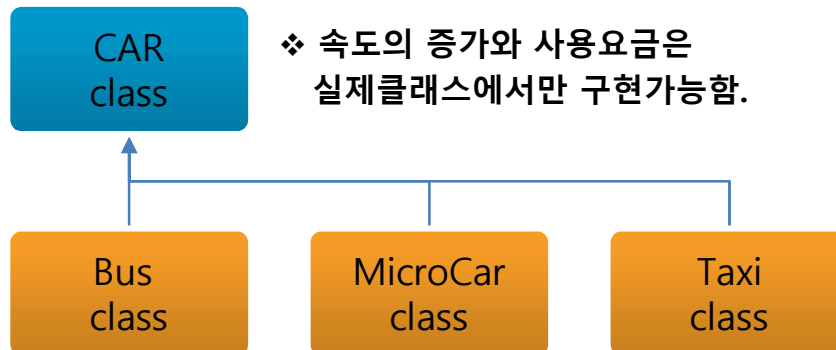
## 7.6.2. 추상메서드 및 오버라이드

- 추상메서드가 있으면 추상클래스로 선언해야 함.
- 추상메서드는 실제클래스에서 꼭 구현해야 함.

### 추상메서드

#### 추상메서드 개념

- ❖ 자식클래스에서 메서드의 이름은 동일하나 실행내용이 달라야 하는 경우가 있음



- ❖ 속도의 증가와 사용요금은 실제클래스에서만 구현가능함.

- ❖ 예시)  
설계단계에서 Car 클래스에 speedUp(속도증가), getCharge (사용요금) 를 정의해놓으면 실제클래스에서 목적에 맞게 구현 가능함.

```

public abstract class Car {
    int currentSpeed = 100;
    public void driving ( )
        System.out.println ("달리다");
    public void stop ( )
        System.out.println ("브레이크를 밟다");
    public abstract int speedUp( ); // 추상메서드의 정의
    public abstract int getCharge( ); // 추상메서드의 정의
}
  
```

추상 class

자동차의  
공통특징만 정의함

```

public class Bus extends Car{
    public int speedUp( ) {
        return ++currentSpeed;
    }
    public int getCharge( ) {
        return 1450;
    }
}
  
```

실제 class

추상메서드를  
실제 구현함.

```

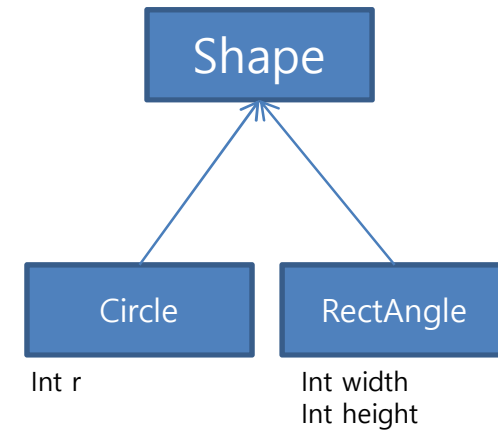
Bus bus = new Bus( );
bus.driving( ); // 추상클래스 메서드 실행
bus.speedUp( ); // 실제클래스 메서드 실행
money = bus.getCharge( ); // 실제클래스 메서드 실행
bus.stop(); // 추상클래스 메서드 실행
  
```

외부 class

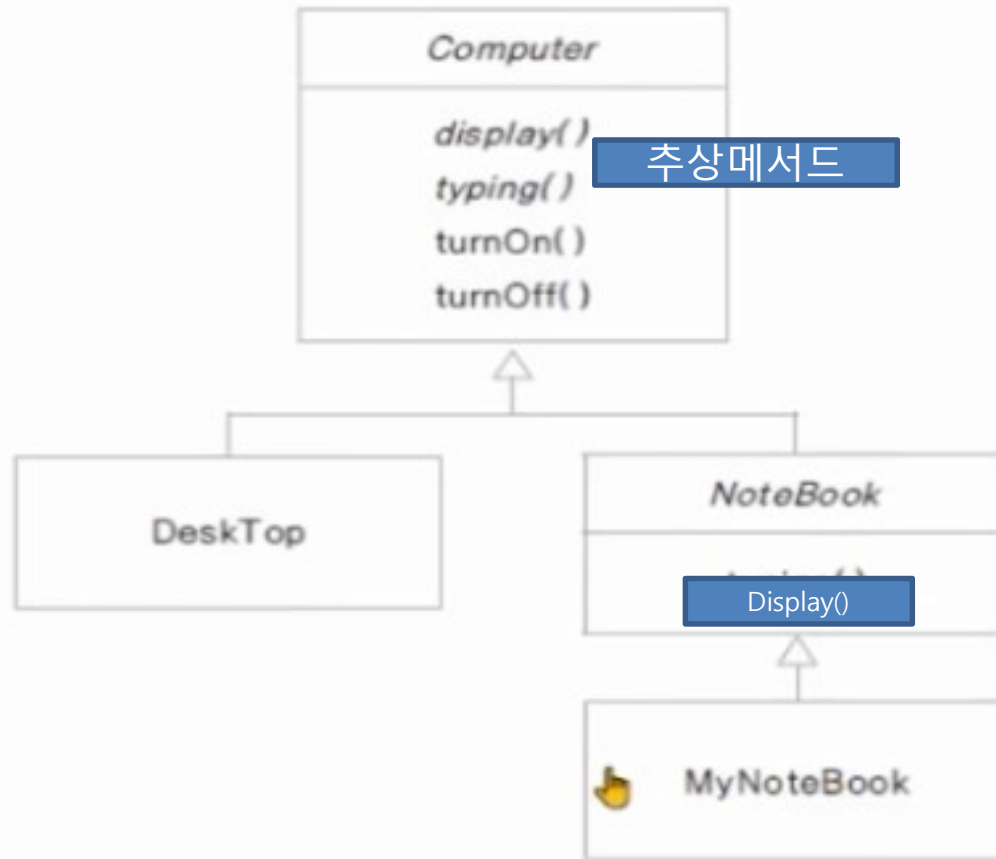
## Exam\_06 – 2 실습

아래는 도형을 정의한 Shape클래스이다. 이 클래스를 조상으로 하는 Circle클래스와 Rectangle클래스를 작성하시오. 이 때, 생성자도 각 클래스에 맞게 적절히 추가해야 한다.

- (1) 클래스명 : Circle  
조상클래스 : Shape  
멤버변수 : double r - 반지름
- (2) 클래스명 : Rectangle  
조상클래스 : Shape  
멤버변수 : double width - 폭  
double height - 높이



```
public class ShapeExam {  
  
    public static void main(String[] args) {  
        Shape circle = new Circle(5.0);  
        System.out.printf ("원의 넓이는 %f입니다. \n", circle.calcArea());  
  
        Shape rect = new Rectangle(3,4);  
        System.out.printf ("사각형의 넓이는 %f입니다. \n", rect.calcArea());  
  
    }  
  
}
```



추상메서드



Typing 미구현

Typing 구현

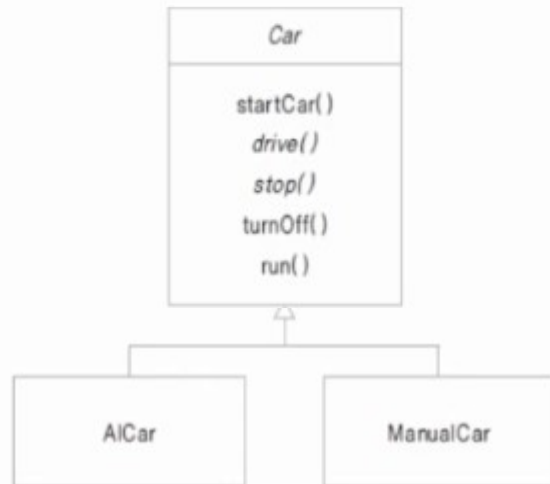


## 10. 추상 클래스의 응용 - 템플릿 메서드 패턴

### 템플릿 메서드

- 추상 메서드나 구현된 메서드를 활용하여 코드의 흐름(시나리오)을 정의하는 메서드
- `final`로 선언하여 하위 클래스에서 재정의 할 수 없게 함
- 프레임워크에서 많이 사용되는 설계 패턴
- 추상 클래스로 선언된 상위 클래스에서 템플릿 메서드를 활용하여 전체적인 흐름을 정의 하고 하위 클래스에서 다르게 구현되어야 하는 부분은 추상 메서드로 선언하여 하위 클래스에서 구현 하도록 함

### 템플릿 메서드 예제



<https://blog.naver.com/hke3255/222271163735>

## 2.4 HelloWorld 앱 코드 분석

### 2.4.2. MainActivity.java

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```