2 학기

JAVA Class

'이것이 자바다'

2025.10

Cover Description

made by Lewis

복습 - 클래스 구성 요소

```
class Car {                    // 클래스 이름
    private String modelName;  // 필드
    private int modelYear;     // 필드

    Car(String modelName, int modelYear) { // 생성자
        this.modelName = modelName;
        this.modelYear = modelYear;
    }

    public String getModel() { // 메소드
        return this.modelYear + "년식 " + this.modelName + " " + this.color;
    }
}
```

필드
- 클래스에 포함된 변수(variable)
- 선언된 위치에 따라
  1. 클래스 변수(static variable)
  2. 인스턴스 변수(instance variable)
  3. 지역 변수(local variable)

생성자
- 클래스를 가지고 객체를 생성하면, 해당 객체는 메모리에 즉시 생성
- 객체의 생성과 동시에 인스턴스 변수를 원하는 값으로 초기화할 수 있는 생성자(constructor)라는 메소드를 제공

메소드
- 어떠한 특정 작업을 수행하기 위한 명령문의 집합
- 코드의 반복적인 내용을 하나로 작성 가능
- 모듈화로 가독성이 좋아짐
- 유지보수 시간 단축

복습 – 생성자, 오버로딩

1. 생성자 선언과 호출

- 클래스 변수 = new 클래스(); -> 생성자 호출

```
public class Car {
    //생성자
    Car(String model, String color, int maxSpeed) {
    }
}
```

2. 생성자/메소드 오버로딩

- 메소드의 이름이 같고, 매개변수의 개수나 타입을 다르게 선언하는 것

```
//생성자
Car()
{
    System.out.println("오버로딩1");
}

Car(Integer a)
{
    System.out.println("오버로딩2");
}

Car(String a)
{
    System.out.println("오버로딩3");
}
```

```
// 정사각형의 넓이
double areaRectangle(double width) {
    return width * width;
}

// 직사각형의 넓이
double areaRectangle(double width, double height) {
    return width * height;
}
```

복습 – 접근 제한자

```
public class ClassName {
  //public 접근 제한
  public ClassName(…) { … }

  //protected 접근 제한
  protected ClassName(…) { … }

  //default 접근 제한
  ClassName(…) { … }

  //private 접근 제한
  private ClassName(…) { … }
}
```

public : 모든 패키지에서 아무런 제한 없이 생성자를 호출할 수 있음

protected : 같은 패키지에 속하는 클래스에서 생성자를 호출할 수 있음
다른 패키지에 속한 클래스가 해당 클래스의 자식(child) 클래스라면 생성자를 호출할 수 있음

default : 같은 패키지에서는 아무런 제한 없이 생성자를 호출할 수 있음
다른 패키지에서는 생성자를 호출할 수 없음

private : 클래스 내부에서만 생성자를 호출할 수 있고 객체를 만들 수 있음

01 **상속**

상속이란?

02 **생성자 호출**

기본, 매개변수 생성자

03 **메소드 오버라이딩**

-

04 **접근제한자**

final 클래스, 메소드, protected

05 **Abstract, sealed Class**

-

컴퓨터가 이해하는 코드는
누구라도 작성할 수 있습니다.
뛰어난 프로그래머는 사람이
이해하는 코드를 작성합니다.

상속이란?

- 상속(inheritance)이란 기존의 클래스에 기능을 추가하거나 재정의하여 새로운 클래스를 정의하는 것
- 기존에 정의되어 있는 클래스의 모든 필드와 메소드를 물려받아, 새로운 클래스를 생성할 수 있다.
- 기존에 정의되어 있던 클래스를 부모 클래스(parent class) 또는 상위 클래스(super class), 기초 클래스(base class)라고 하고 상속을 통해 새롭게 작성되는 클래스를 자식 클래스(child class) 또는 하위 클래스(sub class), 파생 클래스(derived class)라고 한다

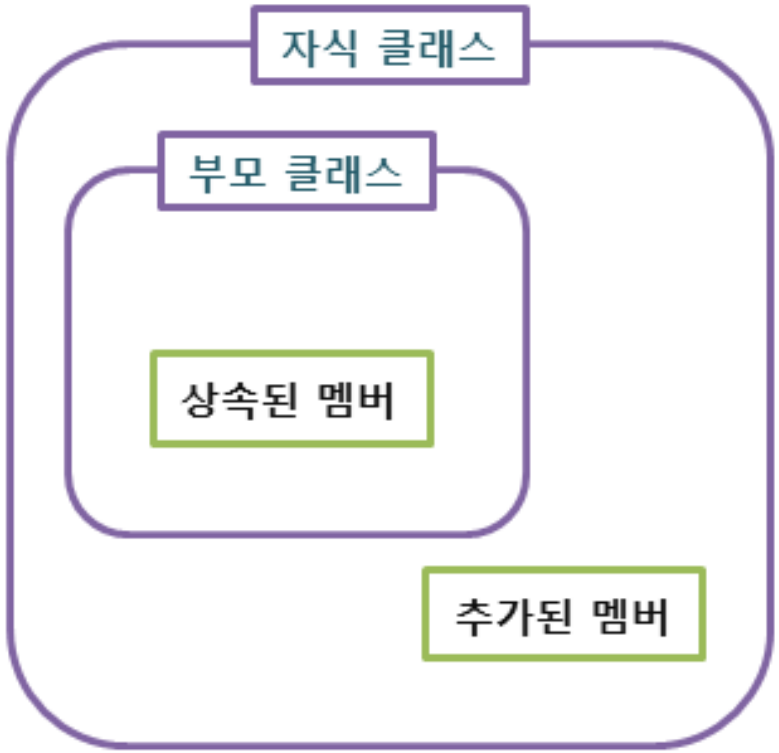
상속의 장점

1. 기존에 작성된 클래스를 재활용할 수 있다.
2. 자식 클래스 설계 시 중복되는 멤버를 미리 부모 클래스에 작성해 놓으면, 자식 클래스에서는 해당 멤버를 작성하지 않아도 된다.
3. 클래스 간의 계층적 관계를 구성함으로써 다형성의 문법적 토대를 마련한다.

클래스 상속

class 자식클래스이름 extend 부모클래스이름 { ... }

➔ 자식 클래스(child class) : 부모 클래스의 모든 특성을 물려받아 새롭게 작성된 클래스를 의미



- 부모 클래스는 자식 클래스에 포함된다.
- 부모 클래스에 새로운 필드를 하나 추가하면, 자식 클래스에도 자동으로 해당 필드가 추가된 것처럼 동작 한다.
- 자식 클래스에는 부모 클래스의 필드와 메소드만이 상속되며, 생성자와 초기화 블록은 상속되지 않는다.

*** 자바는 다중 상속을 허용하지 않는다.**

클래스 상속

```java
package ch07.sec02;

public class Phone {
    // 필드 선언
    public String model;
    public String color;

    // 메소드 선언
    public void bell() {
        System.out.println("벨이 울립니다.");
    }

    public void sendVoice(String message) {
        System.out.println("자기: " + message);
    }

    public void receiveVoice(String message) {
        System.out.println("상대방: " + message);
    }

    public void hangUp() {
        System.out.println("전화를 끊습니다.");
    }
}
```

```java
package ch07.sec02;

public class SmartPhone extends Phone {
    // 필드 선언
    public boolean wifi;

    // 생성자 선언
    public SmartPhone(String model, String color) {
        this.model = model;
        this.color = color;
    }

    // 메소드 선언
    public void setWifi(boolean wifi) {
        this.wifi = wifi;
        System.out.println("와이파이 상태를 변경했습니다.");
    }

    public void internet() {
        System.out.println("인터넷에 연결합니다.");
    }
}
```

```java
package ch07.sec02;

public class SmartPhoneExample {

    public static void main(String[] args) {
        //SmartPhone 객체 생성
        SmartPhone myPhone = new SmartPhone("갤럭시", "은색");

        //Phone으로부터 상속 받은 필드 읽기
        System.out.println("모델: " + myPhone.model);
        System.out.println("색상: " + myPhone.color);

        //SmartPhone의 필드 읽기
        System.out.println("와이파이 상태: " + myPhone.wifi);

        //Phone으로부터 상속 받은 메소드 호출
        myPhone.bell();
        myPhone.sendVoice("여보세요");
        myPhone.receiveVoice("안녕하세요! 저는 홍길동인데요");
        myPhone.sendVoice("아~ 네 반갑습니다.");
        myPhone.hangUp();

        //SmartPhone의 메소드 호출
        myPhone.setWifi(true);
        myPhone.internet();
    }
}
```

부모 생성자 호출

- **생성자 호출**

  인스턴스를 생성할 때 new 연산자를 이용해서 생성자를 호출 한다.
  생성자를 별도로 정의하지 않으면 컴파일러가 자동적으로 기본 생성자를 생성하고 호출 한다.
  매개변수를 갖는 생성자를 개발자가 정의하면 컴파일러는 더 이상 기본 생성자를 만들지 않는다

```
class SuperClass {
    public SuperClass() { // 부모 생성자
        System.out.println("부모 생성자 호출");
    }
}

class SubClass extends SuperClass{
    public SubClass() { // 자식 생성자
        System.out.println("자식 생성자 호출");
    }
}

public class InheritanceConstructorEx01 {
    public static void main(String[] args) {
        SubClass sc = new SubClass(); // 자식 인스턴스 생성
    }
}
```

- ✓ 생성자는 상속이 되지 않는다.
- ✓ 자식 클래스로 인스턴스를 생성할 때 부모 클래스의 기본 생성자(super())를 자동으로 호출

부모 생성자 호출 – 기본 생성자

```java
package ch07.sec03.exam01;

public class Phone {
    //필드 선언
    public String model;
    public String color;

    //기본 생성자를 선언
    public Phone() {
        System.out.println("Phone() 생성자 실행");
    }
}
```

```java
package ch07.sec03.exam01;

public class SmartPhone extends Phone {
    //자식 생성자 선언
    public SmartPhone(String model, String color) {
        super();
        this.model = model;
        this.color = color;
        System.out.println("SmartPhone(String model, String color) 생성자 실행됨");
    }
}
```

```java
package ch07.sec03.exam01;

public class SmartPhoneExample {

    public static void main(String[] args) {
        //SmartPhone 객체 생성
        SmartPhone myPhone = new SmartPhone("갤럭시", "은색");

        //Phone으로부터 상속 받은 필드 읽기
        System.out.println("모델: " + myPhone.model);
        System.out.println("색상: " + myPhone.color);
    }
}
```

✓ 생략가능

```
Phone() 생성자 실행
SmartPhone(String model, String color) 생성자 실행됨
모델: 갤럭시
색상: 은색
```

부모 생성자 호출 – 매개변수 생성자

```java
package ch07.sec03.exam02;

public class Phone {
    //필드 선언
    public String model;
    public String color;

    //매개변수를 갖는 생성자 선언
    public Phone(String model, String color) {
        this.model = model;
        this.color = color;
        System.out.println("Phone(String model, String color) 생성자 실행");
    }
}
```

```java
package ch07.sec03.exam02;

public class SmartPhone extends Phone {
    //자식 생성자 선언
    public SmartPhone(String model, String color) {
        super(model, color);
        System.out.println("SmartPhone(String model, String color) 생성자 실행됨");
    }
}
```

✓ 부모생성자 호출(반드시 작성)

```java
package ch07.sec03.exam01;

public class SmartPhoneExample {

    public static void main(String[] args) {
        //SmartPhone 객체 생성
        SmartPhone myPhone = new SmartPhone("갤럭시", "은색");

        //Phone으로부터 상속 받은 필드 읽기
        System.out.println("모델: " + myPhone.model);
        System.out.println("색상: " + myPhone.color);
    }
}
```

```
Phone(String model, String color) 생성자 실행
SmartPhone(String model, String color) 생성자 실행됨
모델: 갤럭시
색상: 은색
```

부모 생성자 호출

- 부모 클래스의 생성자는 상속되지 않고, 자식 클래스로 인스턴스를 생성할 때 자동적으로 부모의 기본 생성자가 호출된다.
- 부모 생성자가 매개변수를 갖고 있다면 자식 클래스를 인스턴스화할 때 자동으로 호출되지 않고 자식 생성자에서 명시적으로 부모 생성자를 호출해야 한다. ➔ super();
- super()를 사용할 때는 자식 생성자의 첫 줄에 위치하여야 한다.

메소드 오버라이딩

- 상속받은 메서드를 그대로 사용해도 되지만 자식클래스에서 변경 하는 것
- 메서드를 새로 만들게 아니고 내용만을 새로 작성하는 것
- 메서드의 선언부는 부모와 완전히 일치해야 한다.
- 오버라이드 된 메소드는 부모 메소드를 호출하지 않는다.

오버라이딩 사용 조건

1. 자식 클래스의 오버라이딩 하려는 메서드는 부모 클래스의 메서드와 이름, 매개변수 그리고 반환타입이 같아야 한다.
2. 접근 제한자는 부모클래스의 메서드보다 좁은 범위로 변경할 수 없다.
   public -> private 로 변경 불가
3. 부모 클래스의 메서드보다 많은 수의 예외를 선언할 수 없다.
4. 인스턴스 메서드를 static 메서드 또는 그 반대로 변경할 수 없다.

메소드 오버라이딩

```java
package ch07.sec04.exam01;

public class Calculator {
    //메소드 선언
    public double areaCircle(double r) {
        System.out.println("Calculator 객체의 arearCircle() 실행");
        return 3.14159 * r * r;
    }
}
```

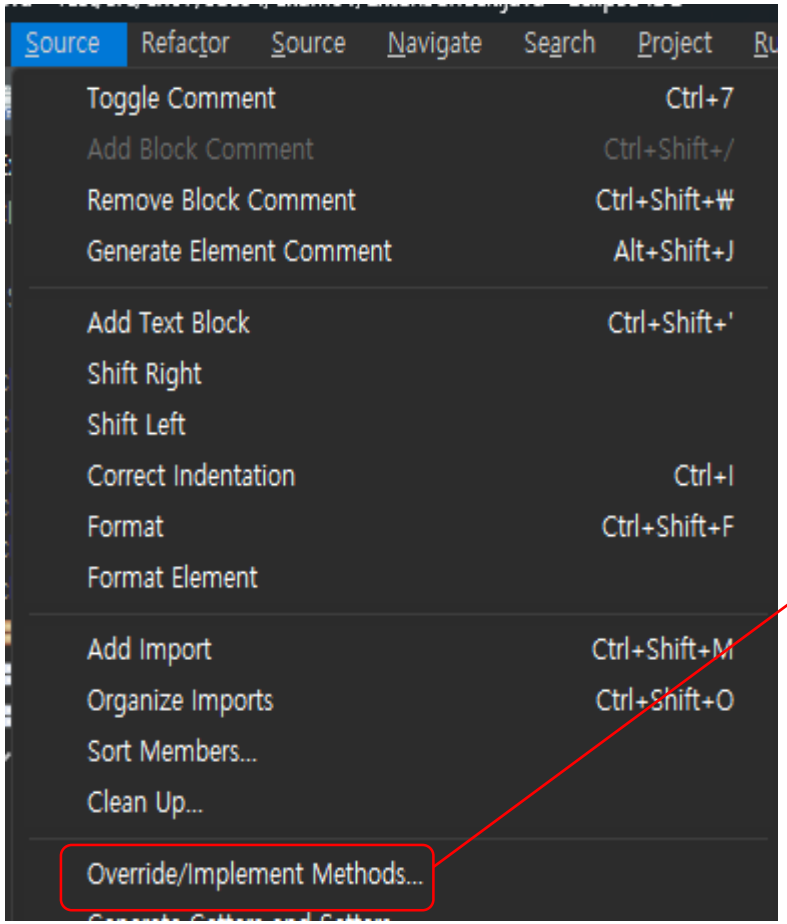
```java
package ch07.sec04.exam01;

public class Computer extends Calculator {
    //메소드 오버라이딩
    @Override
    public double areaCircle(double r) {
        System.out.println("Computer 객체의 arearCircle() 실행");
        return Math.PI * r * r;
    }
}
```

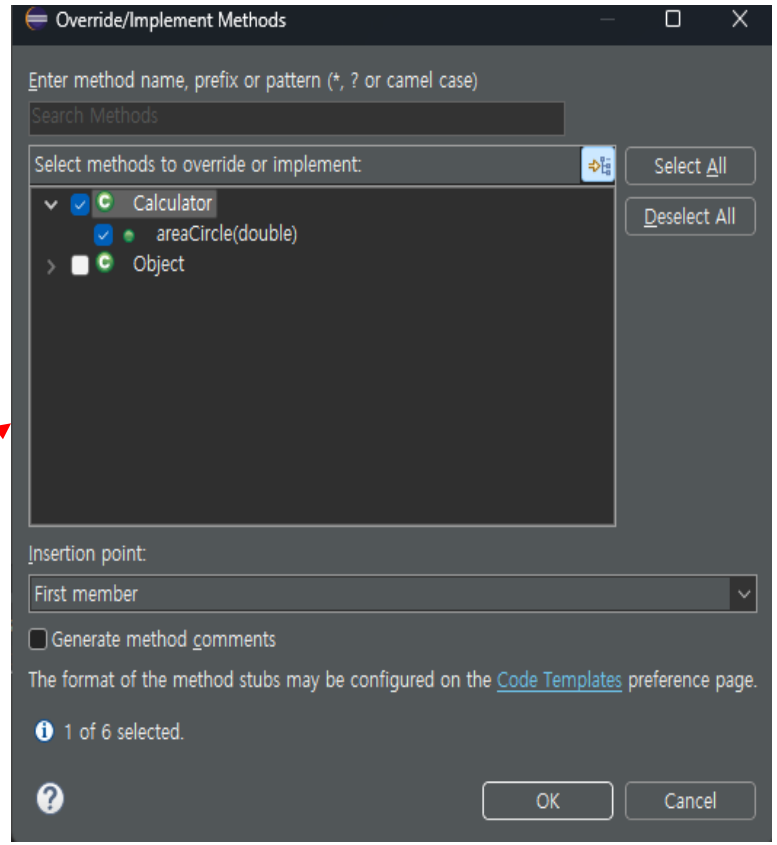
- 컴파일러에게 오버라이드라는 지시
- 정확한 메서드 재정의 표시

```java
package ch07.sec04.exam01;

public class ComputerExample {
    public static void main(String[] args) {
        int r = 10;

        Calculator calculator = new Calculator();
        System.out.println("원면적 : " + calculator.areaCircle(r));
        System.out.println();

        Computer computer = new Computer();
        System.out.println("원면적 : " + computer.areaCircle(r));
    }
}
```

```
<terminated> ComputerExample [Java Appli
Calculator 객체의 arearCircle() 실행
원면적 : 314.159

Computer 객체의 arearCircle() 실행
원면적 : 314.1592653589793
```

메소드 오버라이딩


Source
Refactor
Source
Navigate
Search
Project
Toggle Comment Ctrl+7
Add Block Comment Ctrl+Shift+/
Remove Block Comment Ctrl+Shift+₩
Generate Element Comment Alt+Shift+J
Add Text Block Ctrl+Shift+'
Shift Right
Shift Left
Correct Indentation Ctrl+I
Format Ctrl+Shift+F
Format Element
Add Import Ctrl+Shift+M
Organize Imports Ctrl+Shift+O
Sort Members...
Clean Up...
Override/Implement Methods...


Override/Implement Methods
Enter method name, prefix or pattern (*, ? or camel case)
Search Methods
Select methods to override or implement:
Select All
Deselect All
Calculator
areaCircle(double)
Object
Insertion point:
First member
Generate method comments
The format of the method stubs may be configured on the Code Templates preference page.
1 of 6 selected.
OK
Cancel

메소드 오버라이딩– 부모 메소드 호출

- 메서드를 오버라이딩 시 부모메소드는 호출을 하지 않는다.
- 부모메소드를 호출하기 위해 “super.메소드명()” 를 사용한다.

```
Class A {
    void method1();
    void method1();
}
```

```
Class B extends A
{
    void method1();
    void method1();

    method1();
    super.
method1();
}
```

➔ 부모 메소드 재사용

메소드 오버라이딩– 부모 메소드 호출

```java
package ch07.sec04.exam02;

public class Airplane {
    //메소드 선언
    public void land() {
        System.out.println("착륙합니다.");
    }

    public void fly() {
        System.out.println("일반비행합니다.");
    }

    public void takeOff() {
        System.out.println("이륙합니다.");
    }
}
```

```java
package ch07.sec04.exam02;

public class SupersonicAirplane extends Airplane {
    //상수 선언
    public static final int NORMAL = 1;
    public static final int SUPERSONIC = 2;
    //상태 필드 선언
    public int flyMode = NORMAL;

    //메소드 재정의
    @Override
    public void fly() {
        if(flyMode == SUPERSONIC) {
            System.out.println("초음속비행합니다.");
        } else {
            //Airpanle 객체의 fly() 메소드 호출
            super.fly();
        }
    }
}
```

```java
package ch07.sec04.exam02;

public class SupersonicAirplaneExample {
    public static void main(String[] args) {
        SupersonicAirplane sa = new SupersonicAirplane();
        sa.takeOff();
        sa.fly();
        sa.flyMode = SupersonicAirplane.SUPERSONIC;
        sa.fly();
        sa.flyMode = SupersonicAirplane.NORMAL;
        sa.fly();
        sa.land();
    }
}
```

Final 클래스와 메소드

5.1 Final class

- 클래스 앞에 fianl을 선언하면 그 클래스는 부모 클래스가 될 수 없다.

```
Public final class Member{
}
```

```
Public class Person extends
Member
{

}
```

5.2 Final method

- 메소드 앞에 fianl을 선언하면 그 메소드 는 오버라이딩 할 수 없다.

```
Public class Member{
   public final void SetSSN() {
   }
}
```

```
Public class Person extends
Member{
   @Override
   public void SetSSN() {
   }
}
```

Final 클래스와 메소드

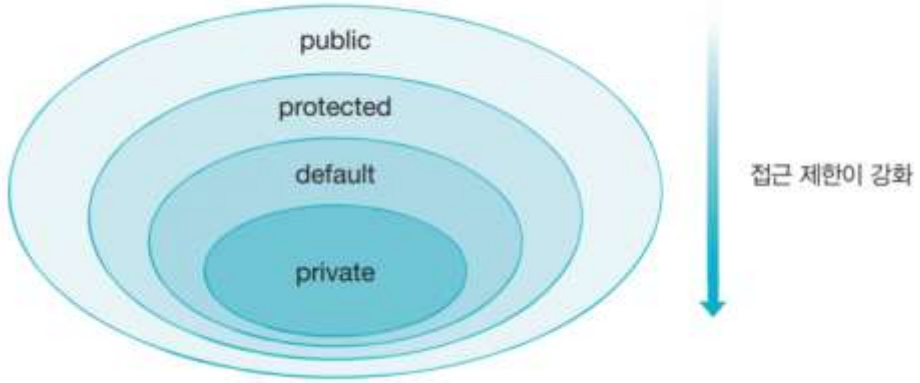
```java
package ch07.sec05.exam02;

public class Car {
    //필드 선언
    public int speed;

    //메소드 선언
    public void speedUp() {
        speed += 1;
    }

    //final 메소드
    public final void stop() {
        System.out.println("차를 멈춤");
        speed = 0;
    }
}
```

```java
package ch07.sec05.exam02;

public class SportsCar extends Car {
    @Override
    public void speedUp() {
        speed += 10;
    }

    // 오버라이딩을 할 수 없음
    /*
    @Override
    public void stop() {
        System.out.println("스포츠카를 멈춤");
        speed = 0;
    }
    */
}
```

protected 접근 제한자

- 동일한 패키지거나, 자식 개체만 사용 가능

접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스



protected 접근 제한자

```java
package ch07.sec06.package1;

public class A {
    //필드 선언
    protected String field;

    //생성자 선언
    protected A() {
    }

    //메소드 선언
    protected void method() {
    }
}
```

```java
package ch07.sec06.package1;

public class B {
    //메소드 선언
    public void method() {
        A a = new A();          //o
        a.field = "value";      //o
        a.method();             //o
    }
}
```

```java
package ch07.sec06.package2;

import ch07.sec06.package1.A;

public class C {
    //메소드 선언
    public void method() {
        //A a = new A();        //x
        //a.field = "value";    //x
        //a.method();           //x
    }
}
```

```java
package ch07.sec06.package2;

import ch07.sec06.package1.A;

public class D extends A {
    //생성자 선언
    public D() {
        //A() 생성자 호출
        super();                //o
    }

    //메소드 선언
    public void method1() {
        //A 필드값 변경
        this.field = "value";   //o
        //A 메소드 호출
        this.method();          //o
    }

    //메소드 선언
    public void method2() {
        //A a = new A();        //x
        //a.field = "value";    //x
        //a.method();           //x
    }
}
```

클래스 형변환

- 부모타입으로 자식객체를 참조하게 되면 부모가 가지고 있는 메소드만 사용할 수 있다.
- 자식객체가 가지고 있는 메소드나 속성을 사용하고 싶다면 형변환을 해야한다.

```
public class Car {
   public void run() {
      System.out.println(" run Method of Car");
   }
}

public class Bus extends Car {
   public void beef() {
      System.out.println("Buuuuus");
   }
}
```

있다

```
public class BusExam {
    public static void main(String args[]) {
        Car car = new Bus(); // 부모타입으로 자식객체를 참조할 수

        car.run();
        //car. beef(); // 컴파일 오류 발생

        Bus bus = (Bus)car;  //부모타입을 자식타입으로 형변환
        bus.run();
        bus. beef();
    }
}
```

1. 상속관계에 있었을 객체들끼리도 형변환이 가능하다.
2. 부모타입으로 자식타입의 객체를 참조할 때는 묵시적으로 형변환이 일어난다.
3. 부모타입 객체를 자식타입으로 참조하게 할때 명시적 형변환을 해줘야 한다.

클래스 형변환

```java
package ch07.sec07.exam02;

public class Parent {
    //메소드 선언
    public void method1() {
        System.out.println("Parent-method1()");
    }

    //메소드 선언
    public void method2() {
        System.out.println("Parent-method2()");
    }
}
```

```java
package ch07.sec07.exam03;

public class Child extends Parent {
    //필드 선언
    public String field2;

    //메소드 선언
    public void method3() {
        System.out.println("Child-method3()");
    }
}
```

```java
package ch07.sec07.exam03;

public class ChildExample {
    public static void main(String[] args) {
        //객체 생성 및 자동 타입 변환
        Parent parent = new Child();

        //Parent 타입으로 필드와 메소드 사용
        parent.field1 = "data1";
        parent.method1();
        parent.method2();
        /*
        parent.field2 = "data2";        //(불가능)
        parent.method3();               //(불가능)
        */

        //강제 타입 변환
        Child child = (Child) parent;

        //Child 타입으로 필드와 메소드 사용
        child.field2 = "data2";         //(가능)
        child.method3();                //(가능)
    }
}
```

```
Parent-method1()
Parent-method2()
Child-method3()
```

다형성

자동 타입 변환 + 메소드 오버라이딩 → 다형성

8.1 다형성 이란?

- 하나의 코드가 여러 자료형으로 구현되어 실행되는 것
- 같은 코드에서 여러 다른 실행 결과가 나옴
- 다형성을 잘 활용하면 유연하고 확장성 있고, 유지보수가 편리한 프로그램을 만들 수 있다.

8.2 다형성을 사용하는 이유

- 상속과 메소드 오버라이딩을 활용하여 확장성 있는 프로그램 개발이 가능해진다.
  →아니면 if 문의 추가 사용으로 코드가 점점 복잡해 진다.
- 상위 클래스에서 공통적인 부분 제공하고 하위클래스에 필요한 기능만 구현한다.
- 여러 클래스를 하나의 타입(부모클래스)으로 처리할 수 있다.

다형성

```java
package ch07.sec08.exam01;

public class Car {
    // 필드 선언
    public Tire tire;

    // 메소드 선언
    public void run() {
        //tire 필드에 대입된 객체의 roll 메소드 호출
        tire.roll();
    }
}
```

```java
package ch07.sec08.exam01;

public class Tire {
    // 메소드 선언
    public void roll() {
        System.out.println("회전합니다.");
    }
}
```

```java
package ch07.sec08.exam01;

public class HankookTire extends Tire {
    // 메소드 재정의(오버라이딩)
    @Override
    public void roll() {
        System.out.println("한국 타이어가 회전합니다.");
    }
}
```

```java
package ch07.sec08.exam01;

public class KumhoTire extends Tire {
    // 메소드 재정의(오버라이딩)
    @Override
    public void roll() {
        System.out.println("금호 타이어가 회전합니다.");
    }
}
```

```java
package ch07.sec08.exam01;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

        //Tire 객체 장착
        myCar.tire = new Tire();
        myCar.run();

        //HankookTire 객체 장착
        myCar.tire = new HankookTire();
        myCar.run();

        //KumhoTire 객체 장착
        myCar.tire = new KumhoTire();
        myCar.run();
    }
}
```

```
회전합니다.
한국 타이어가 회전합니다.
금호 타이어가 회전합니다.
```

다형성- 매개변수

- 참조타입의 매개변수는 메소드에 접근할 때 자신과 같은 타입이거나 또는 자식 타입의 주소를 넘겨준다.
- 부모의 매개변수로 접근할 수 있으면 자손 타입도 접근할 수 있다.
- 부모클래스의 매개변수가 있어서 자식 클래스의 매개변수를 접근하기 때문에 코드가 간단해 진다.

```java
package ch07.sec08.exam02;

public class Vehicle {
    //메소드 선언
    public void run() {
        System.out.println("차량이 달립니다.");
    }
}
```

```java
package ch07.sec08.exam02;

public class Taxi extends Vehicle {
    //메소드 재정의(오버라이딩)
    @Override
    public void run() {
        System.out.println("택시가 달립니다.");
    }
}
```

```java
package ch07.sec08.exam02;

public class DriverExample {
    public static void main(String[] args) {
        //Driver 객체 생성
        Driver driver = new Driver();

        //매개값으로 Bus 객체를 제공하고 driver 메소드 호출
        Bus bus = new Bus();
        driver.drive(bus);

        //매개값으로 Taxi 객체를 제공하고 driver 메소드 호출
        Taxi taxi = new Taxi();
        driver.drive(taxi);
    }
}
```

```java
package ch07.sec08.exam02;

public class Bus extends Vehicle {
    //메소드 재정의(오버라이딩)
    @Override
    public void run() {
        System.out.println("버스가 달립니다.");
    }
}
```

```java
package ch07.sec08.exam02;

public class Driver {
    //메소드 선언(클래스 타입의 매개변수를 가지고 있음)
    public void drive(Vehicle vehicle) {
        vehicle.run();
    }
}
```

객체타입 확인

타입 확인의 필요성

- 객체 지향 언어로 다양한 데이터 타입을 사용하여 개발을 진행하는데 이러한 다양한 타입 중 어떤 타입인지를 확인하는 것은 필수 적인 사항
- 메소드가 여러 데이터 타입을 받을 수 있거나, 객체의 서브 클래스를 리턴하는 경우 데이터 타입 확인은 중요한 사항

타입 확인 방법

- Instanceof 연산자 사용
  - 왼쪽의 객체가 오른쪽에 있는 클래스 또는 인터페이스의 인스턴스인지 확인

```
Object oObj = new String("Object");
if(oObj instanceof String) {
    System.out.println("oObj is a String");
} else {
    System.out.println("oObj is not a String");
}
```

- **getClass() 메소드 사용**
  - 객체의 정확한 클래스를 반환

```
Object oObj = new String("Object");
if(oObj.getClass() == String.class) {
    System.out.println("oObj is a String");
} else {
    System.out.println("oObj is not a String");
}
```

클래스 상속 테스트

1. Vehicle 클래스를 생성하고 brand 라는 필드와 move() 라는 메소드를 생성.
   move() 메소드에 “이동합니다.” 라고 출력
2. Car 클래스를 생성하고 Vehicle 를 상속받고 brand에 “자동차” 등록 후 move()를
   오버라이딩하여 “자동차는 고속도로를 달립니다” 를 출력
3. Motorcycle 클래스를 생성하고 Vehicle 를 상속받고 brand에 “오토바이” 등록 후 move()를
   오버라이딩하여 “오토바이는 일반도로를 달립니다” 를 출력
4. MoveExample 클래스에서 Car, Motorcycle 의 move() 메소드를 호출하는 프로그램을
   작성하시오.

추상클래스

추상 클래스(abstract class)

- A클래스, B클래스, C클래스 가 있다면 추상클래스는 A클래스, B클래스, C클래스들 간에 비슷한 필드와 메서드를 공통적으로 추출해 만들어진 클래스다
- 공통 뼈대 + 부분 구현을 제공하며 직접 생성은 못 하는 클래스(new 인스턴스 화 불가)
- 공통된 필드와 메서드 통일 → 유지보수성을 높이고 통일성 유지
- 실체 클래스 구현 시 시간 절약

추상 클래스 문법 - 클래스 앞에 abstract 키워드를 붙이면 추상클래스

```
public abstract class 클래스명{
    //필드
    //생성자
    //메서드
    //추상메서드
}
```

추상Class	실체 Class	Method	Variable
사람	나, 너, 길동이..	잔다, 걷는다..	얼굴, 눈, 코, 입 …
차	현대차, 기아차 ..	달린다, 정지한다 ….	룸미러, 기어, 라이트 …

추상클래스

추상 클래스(abstract class) 생성 규칙

- 클래스에 abstract 가 붙으면 직접 생성 불가
- 추상 메소드가 하나라도 있으면 클래스는 반드시 abstract
- 클래스 내부에 필드/생성자/구현 메소드/final/static 모두 가질 수 있음
- 다중 상속 불가

객체타입 확인

```java
package ch07.sec10.exam01;

public abstract class Phone {
    //필드 선언
    String owner;

    //생성자 선언
    Phone(String owner) {
        this.owner = owner;
    }

    //메소드 선언
    void turnOn() {
        System.out.println("폰 전원을 켭니다.");
    }
    void turnOff() {
        System.out.println("폰 전원을 끕니다.");
    }
}
```

```java
package ch07.sec10.exam01;

public class SmartPhone extends Phone {
    //생성자 선언
    SmartPhone(String owner) {
        //Phone 생성자 호출
        super(owner);
    }

    //메소드 선언
    void internetSearch() {
        System.out.println("인터넷 검색을 합니다.");
    }
}
```

```java
package ch07.sec10.exam01;

public class PhoneExample {
    public static void main(String[] args) {
        //Phone phone = new Phone();

        SmartPhone smartPhone = new SmartPhone("홍길동");

        smartPhone.turnOn();
        smartPhone.internetSearch();
        smartPhone.turnOff();
    }
}
```

```
폰 전원을 켭니다.
인터넷 검색을 합니다.
폰 전원을 끕니다.
```

```java
Phone phone = new SmartPhone("test");
System.out.println(phone.owner);
```

추상메소드

- 선언은 되어있으나 코드가 구현되지 않은 메소드
- 자식 클래스가 구현해야 하는 메서드의 가이드라인만 제시하기 위한 목적으로 사용
- 작성되어 있지 않은 구현부는 자식 클래스에서 오버라이딩 하여 사용한다.

추상 메소드 문법 - 메소드 앞에 abstract 키워드를 붙이면 추상메소드

```
public abstract void newMethod();
```

→ 선언만 가능하고, 구현부를 위한 블록이 존재하지 않는다.
중괄호{} 대신 문장의 끝을 알리는 세미콜론 (;) 을 적어준다.
추상 클래스를 상속받으면 자식은 부모의 모든 추상 메서드를 재정의 해야한다.

추상메소드

```java
package ch07.sec10.exam02;

public abstract class Animal {
    //메소드 선언
    public void breathe() {
        System.out.println("숨을 쉽니다.");
    }

    //추상 메소드 선언
    public abstract void sound();
}
```

```java
package ch07.sec10.exam02;

public class Dog extends Animal {
    //추상 메소드 재정의
    @Override
    public void sound() {
        System.out.println("멍멍");
    }
}
```

```java
package ch07.sec10.exam02;

public class Cat extends Animal {
    //추상 메소드 재정의
    @Override
    public void sound() {
        System.out.println("야옹");
    }
}
```

```java
package ch07.sec10.exam02;

public class AbstractMethodExample {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.sound();

        Cat cat = new Cat();
        cat.sound();

        //매개변수의 다형성
        animalSound(new Dog());
        animalSound(new Cat());
    }

    public static void animalSound( Animal animal  ) {
        animal.sound();
    }
}
```

```
멍멍
야옹
멍멍
야옹
```

상속 VS 추상클래스

- 코드 재사용이 목적
- 추상클래스는 재사용 + 강제 + 공통 뼈대 제공
- 자식 클래스에서의 일관성을 보장하기위해서 추상 클래스를 사용
- 템플릿 메서드 패턴을 사용하기 위해서 추상 클래스 사용

추상 클래스 ➔ 상속보다 더 안전하고 일관된 확장 시스템 제공

봉인된 클래스

- 무분별한 자식 클래스 생성을 막기 위해 sealed class 사용.
- 상속하거나, 구현할 클래스를 지정해두고 해당 클래스들만 상속 혹은 구현을 허용하는 키워드

추상 메소드 문법 – sealed 를 정의하면 permits 뒤에 상속 가능한 자식 클래스를 지정해야 한다.
public sealed interface CarBrand permits Hyundai, Kia{}

public final class Hyundai implements CarBrand {} → final : 더 이상 상속할 수 없다
Public non-sealed class Kia implements CarBrand {} → non-sealed : 봉인해제 하여 상속될 수 있다.

봉인된 클래스

```java
package ch07.sec11;

public sealed class Person permits Employee, Manager {
    //필드
    public String name;

    //메소드
    public void work() {
        System.out.println("하는 일이 결정되지 않았습니다.");
    }
}
```

```java
package ch07.sec11;

public final class Employee extends Person {
    @Override
    public void work() {
        System.out.println("제품을 생산합니다.");
    }
}
```

```java
package ch07.sec11;

public non-sealed class Manager extends Person {
    @Override
    public void work() {
        System.out.println("생산 관리를 합니다.");
    }
}
```

```java
package ch07.sec11;

public class Director extends Manager {
    @Override
    public void work() {
        System.out.println("제품을 기획합니다.");
    }
}
```

```java
package ch07.sec11;

public class SealedExample {
    public static void main(String[] args) {
        Person p = new Person();
        Employee e = new Employee();
        Manager m = new Manager();
        Director d = new Director();

        p.work();
        e.work();
        m.work();
        d.work();
    }
}
```

```
하는 일이 결정되지 않았습니다.
제품을 생산합니다.
생산 관리를 합니다.
제품을 기획합니다.
```