Chapter 7. Inheritance(상속)

7.1 상속 개념

: 부모 클래스의 멤버를 자식 클래스에게 물려줄 수 있다. 부모 클래스를 상위 클래스라고 부르고, 자식 클래스를 하위 클래스, 또는 파생 클래스라고 부른다.

- **코드의 중복을 줄여준다. 개발 시간을 절약시켜준다.**

  예시)

```
public class A{
    int field1;
    void method1(){ ... }
}
public class B extends A{   // A를 상속
    String field2;
    void method2(){ ... }
}
```

- **상속을 해도 부모 클래스의 모든 필드와 메소들을 물려받는 것은 아니다.**
  - private 접근 제한을 갖는 필드와 메소드는 상속 대상에서 제외된다.
  - 클래스끼리 다른 패키지에 존재한다면 default 접근 제한을 갖는 필드와 메소드도 상속 대상에서 제외된다.
  - 그 이외는 모두 상속 대상.
- **수정을 최소화시킬 수 있다.**

7.2 클래스 상속

: 자식 클래스를 선언할 때 어떤 부모 클래스를 상속받을 것인지를 결정한다.

- **예시**

```
class 자식클래스 extends 부모클래스{
    // 필드
    // 생성자
    // 메소드
}
```

- **주의할 점**

  : extends 뒤에는 단 하나의 부모 클래스만 와야 한다.

```
class 자식클래스 extends 부모클래스1, 부모클래스2{ // 에러!!
}
```

7.3 부모 생성자 호출

: 자식 객체를 생성하면, 부모 객체가 먼저 생성되고 자식 객체가 그 다음에 생성된다.

- **부모 생성자는 자식 생성자의 맨 첫 줄에서 호출된다.**

  자식 생성자가 명시적으로 선언되지 않을때

```
public DmbCellPhone(){
    super();    // 부모의 기본 생성자를 호출한다.
}
```

직접 자식 생성자를 선언하고 명시적으로 부모 생성자를 호출할때 선언

```
자식클래스(매개변수선언, ...){
    super(매개값, ...);
    ...
}
```

> **super(매개값, ...)**는 매개값의 타입과 일치하는 부모 생성자를 호출. 일치하는 부모 생성자가 없을 경우 컴파일 오류.
>
> > super(매개값, ...)가 생략되면 컴파일러에 의해 super()가 자동적으로 추가되기 때문에 부모의 기본 생성자가 존재해야 한다. 부모 클래스에 기본 생성자가 없고 매개 변수가 있는 생성자만 있다면 자식 생성자에서 **반드시 부모 생성자 호출을 위해 super(매개값, ...)**을 명시적으로 호출해야 한다.

- **예제**

  People.java(부모 클래스)

```
package parent_constructor_call;

public class People {
    public String name;
    public String ssn;

    // 기본 생성자가 없고 두 개의 매개값을 받아 객체를 생성하는 생성자
    public People(String name, String ssn){
        this.name = name;
        this.ssn = ssn;
    }
}
```

  Student.java(자식 클래스)

```java
package parent_constructor_call;

// 부모 생성자가 기본 생성자가 아니기 때문에
// 자식 생성자 맨 윗줄에 super(...)을 써서
// 부모 생성자에 맞게 생성자를 호출해준다.
public class Student extends People{
    public int studentNo;

    public Student(String name, String ssn, int studentNo){
        super(name, ssn);   // 부모 생성자 호출
        this.studentNo = studentNo;
    }
}
```

StudentExample.java(자식 객체 이용)

```java
package parent_constructor_call;

public class StudentExample {
    public static void main(String[] args){
        Student student = new Student("홍길동", "111111-1111111", 1);
        System.out.println("ssn: " + student.ssn);
        System.out.println("studentNo: " + student.studentNo);
    }
}
```

실행 결과

```
ssn: 111111-1111111
studentNo: 1
```

7.4 메소드 재정의

- **메소드 오버라이딩(Overriding)** : 부모 클래스의 메소드가 자식 클래스가 사용하기에 적합하지 않아 상속된 일부 메소드를 자식 클래스에서 수정해서 사용

7.4.1 메소드 재정의(@Override)

: 자식 클래스에서 동일한 메소드를 재정의하는 것을 말한다. 메소드가 오버라이딩되면 부모 객체의 메소드는 숨겨지기 때문에, 자식 객체에서 메소드를 호출하면 **오버라이딩된 자식 메소드가 호출된다.**

- **오버라이딩 규칙**
  - 부모의 메소드와 동일한 **시그니처(리턴 타입, 메소드 이름, 매개 변수 리스트)**를 가져야 한다.
  - 접근 제한을 더 강하게 오버라이딩할 수 없다.

    : 부모 메소드가 public 접근 제한인데 오버라이딩하여 자식 메소드가 default나 private 접근 제한으로 수정할 수 없다.(**더 약하게는 오버라이딩 가능**)
  - 새로운 예외(Exception)를 throws할 수 없다.

- **예제**

  Calculator.java(**부모 클래스**)

```
package method_overriding;

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

  Computer.java(**자식 클래스**)

```
package method_overriding;

public class Computer extends Calculator{
    // 어노테이션을 사용하면 메소드가 정확히 오버라이딩된 것인지
    // 컴파일러가 체크하기 때문에 실수를 줄일 수 있다.
    // @Override
    double areaCircle(double r){
        System.out.println("Computer 객체의 areaCircle() 실행");
        return Math.PI * r * r;
    }
}
```

  ComputerExample.java(**메소드 오버라이딩 테스트**)

```
package method_overriding;

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();

        // 재정의된 자식 클래스 Computer 메소드 호출
        System.out.println("원면적: " + computer.areaCircle(r));
    }
}
```

  실행 결과

```
Calculator 객체의 areaCircle() 실행
원면적: 314.159

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

7.4.2 부모 메소드 호출(super)

: 자식 클래스 내부에서 오버라이딩된 부모 클래스의 메소드를 호출해야 하는 상황이 발생한다면 명시적으로 **super 키워드**를 붙여서 부모 메소드를 호출할 수 있다.

- **예제**

  Airplane.java(**super 변수**)

```
package parent_method_call;

public class Airplane {
    public void land(){
        System.out.println("착륙합니다");
    }
    public void fly(){
        System.out.println("일반 비행");
    }
    public void takeOff(){
        System.out.println("이륙합니다");
    }
}
```

  SupersonicAirplane.java(**super 변수**)

```
package parent_method_call;

public class SupersonicAirplane extends Airplane {
    public static final int NORMAL = 1;
    public static final int SUPERSONIC = 2;

    public int flyMode = NORMAL;

    public void fly(){
        if(flyMode == SUPERSONIC){
            System.out.println("초음속비행");
        } else {
            // Airplane 객체의 fly() 메소드 호출
            super.fly();
        }
    }
}
```

  SupersonicAirplaneExample.java(**super 변수**)

```
package parent_method_call;

public class SupersonicAirpaneExample {
    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();
    }
}
```

실행 결과

```
이륙합니다
일반 비행
초음속비행
일반 비행
착륙합니다
```

7.5 final 클래스와 final 메소드

: final 키워드는 클래스, 필드, 메소드 선언 시에 사용할 수 있다. 클래스와 메소드 선언 시에 final 키워드가 지정되면 상속과 관련이 있다.

7.5.1 상속할 수 없는 final 클래스

: final 키워드를 class 앞에 붙이게 되면 이 클래스는 최종적인 클래스이므로 상속할 수 없는 클래스가 된다. 즉, final 클래스는 부모 클래스가 될 수 없어 자식 클래스를 만들 수 없다.

- **대표적인 예**

```
// 자바 표준 API에서 제공하는 String 클래스는 다음과 같이 선언되어 있다.
public final class String { ... }

public class NewString extends String{ ... } // 컴파일 오류
```

- **예제**

  Member.java(**상속할 수 없는 final 클래스**)

```
public final class Member{
}
```

VeryVeryImportantPerson.java(**상속할 수 없는 final 클래스**)

```
// Member를 상속할 수 없음
public class VeryVeryImportantPerson extends Member{    // 컴파일 오류!!
}
```

7.5.2 오버라이딩할 수 없는 final 메소드

: 메소드를 선언할 때 final 키워드를 붙이게 되면 이 메소드는 최종적인 메소드이므로 오버라이딩(Overriding)할 수 없는 메소드가 된다.

- **예제**

  Car.java(**재정의할 수 없는 final 메소드**)

```
public class Car{
    public int speed;
    public void speedUp(){ speed += 1; }

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

  SportsCar.java(**재정의할 수 없는 final 메소드**)

```
public class SportsCar extends Car {
    // final 메소드가 아니기 때문에 오버라이딩이 가능하다.
    public void speedUp(){ speed += 10; }

    // stop이 부모 클래스에서 final 메소드로 선언되어 있기 때문에
    // 오버라이딩을 할 수 없음
    public void stop(){
        System.out.println("스포츠카를 멈춤");
        speed = 0;
    }
}
```

7.6 protected 접근 제한자

접근 제한(강도)	1	2	3	4
타입	public	protected	default	private

: protected는 같은 패키지에서는 접근 제한이 없지만 다른 패키지에서는 자식 클래스만 접근을 허용한다.

- **예제**

  A.java(**protected 접근 제한자**)

```
package package1;

public class A{
    protected String field;

    protected A(){
    }

    protected void method(){
    }
}
```

  B.java(**protected 접근 제한자 테스트**)

```
package package1;       // A와 같은 패키지

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

> B와 A는 같은 패키지이므로 B가 A의 protected 멤버에 얼마든지 접근이 가능하다.

  C.java(**protected 접근 제한자 테스트**)

```
package package2;
import package1.A;

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

> C는 A와 다른 패키지이기 때문에 A의 protected 멤버에 접근할 수 없다.

  D.java(**protected 접근 제한자**)

```
package package2;
import package1.A;

public class D extends A{
    public D() {
        super();                                // O
        this.field = "value";   // O
        this.method();                  // O
    }
}
```

> D 클래스는 A의 자식 클래스로써 A 클래스의 protected 멤버들에 접근할 수 있다. 단 new 연산자를 사용해서 생성자를 직접 호출할 수는 없고, 자식 생성자에서 **super()로 A 생성자를 호출**할 수 있다.

7.7 타입 변환과 다형성

- **다형성** : 같은 타입이지만 실행 결과가 다양한 객체를 이용할 수 있는 성질. 그리고 다형성은 하나의 타입에 여러 객체를 대입함으로써 다양한 기능을 이용할 수 있도록 해준다
- **다형성을 위해 자바는 부모 클래스로 타입 변환을 허용한다.** 즉 부모 타입에 모든 자식 객체가 대입될 수 있다.
- **예제**

```
public class Car{
    // Tire = 부모 클래스
    // HankookTire, KumhoTire = Tire의 자식 클래스
    Tire t1 = new HankookTire();
    Tire t2 = new KumhoTire();
}
```

> 자식 타입은 부모 타입으로 자동 타입 변환이 가능하다.
>
> HankookTire와 KumhoTire는 Tire을 상속했기 때문에 Tire 변수에 대입할 수 있다.

7.7.1 자동 타입 변환(Promotion)

- 자동 타입 변환 예시

```
부모클래스 변수 = 자식클래스타입;           // 자동 타입 변환이 발생
```

- 코드 예시

```
class Animal{
    ...
}
class Cat extends Animal{
    ...
}

Cat cat = new Cat();
Animal animal = cat;

cat == animal   // true
```

> cat과 animal 변수는 타입만 다를 뿐, 동일한 Cat 객체를 참조하게 된다.

- **예제**

  PromotionExample.java(**자동 타입 변환**)

```
package promotion;

class A{}

class B extends A{}
class C extends A{}

class D extends B{}
class E extends C{}

public class PromotionExample {
    public static void main(String[] args) {
        B b = new B();
        C c = new C();
        D d = new D();
        E e = new E();
        A a = new A();

        // 자식이 부모로 타입 변환은 안된다
        // b = a; 컴파일 에러!!

        A a1 = b;
        A a2 = c;
        A a3 = d;
        A a4 = e;

        B b1 = d;
        C c1 = e;

        // 상속 관계가 아니므로 컴파일 에러!!
        // B b3 = e;
        // C c2 = d;
```

```
    }
}
```

> 부모 타입으로 자동 타입 변환된 이후에는 부모 클래스에 선언된 필드와 메소드만 접근이 가능하다. 비록 변수는 자식 객체를 참조하지만 변수로 접근 가능한 멤버는 부모 클래스 멤버로만 한정된다. 그러나 메소드가 자식 클래스에서 **오버라이딩되었다면** 자식 클래스의 메소드가 대신 호출된다.

- **예제**

  Parent.java(**자동 타입 변환 후의 멤버 접근**)

```
package promotion;

public class Parent {
    public void method1(){
        System.out.println("Parent-method1()");
    }
    public void method2(){
        System.out.println("Parent-method2()");
    }
}
```

  Child.java(**자동 타입 변환 후의 멤버 접근**)

```
package promotion;

public class Child extends Parent {
    public void method2(){      // 메소드 오버라이딩
        System.out.println("Child-method2()");
    }
    public void method3(){
        System.out.println("Child-method3()");
    }
}
```

  ChildExample.java(**자동 타입 변환 후의 멤버 접근**)

```
package promotion;

public class ChildExample {
    public static void main(String[] args) {
        Child child = new Child();

        // 자동 타입 변환
        Parent parent = child;

        parent.method1();
        parent.method2(); // 오버라이딩된 함수 호출
        // parent.method3(); 컴파일 에러(호출 불가능)
```

```
    }
}
```

실행 결과

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

> 자동 타입 변환된 이후에는 부모 클래스에 선언된 필드와 메소드만 접근 가능.
>
> method2는 자식 클래스에 오버라이딩되어있기 때문에 자식 클래스의 메소드로 실행된다.

7.7.2 필드의 다형성

: 필드의 타입은 변함이 없지만, 실행 도중에 어떤 객체를 필드로 저장하느냐에 따라 실행 결과가 달라질 수 있다.

- **예시**

```
class Car{
    // 필드
    Tire frontLeftTire = new Tire();
    Tire frontRightTire = new Tire();
    Tire backLeftTire = new Tire();
    Tire backRightTire = new Tire();

    // 메소드
    void run(){ ... }
}
```

> Car 객체를 생성하면 4개의 Tire 필드에 각각 하나씩 Tire 객체가 들어가게 된다.
>
> 만약 frontRightTire와 backLeftTire를 HankookTire와 KumhoTire로 교체할 필요성이 생겼다면?

```
Car myCar = new Car();
myCar.frontRightTire = new HankookTire();
myCar.backLeftTire = new KumhoTire();
myCar.run();
```

> HankookTire와 KumhoTire는 부모인 Tire의 필드와 메소드를 가지고 있기 때문에 자동 타입 변환이 된다. 그리고 Car 객체에 run()메소드는 각 Tire 객체의 roll() 메소드를 다음과 같이 호출한다.

```
void run(){
    frontLeftTire.roll();
    frontRightTire.roll();
    backLeftTire.roll();
    backRightTire.roll();
}
```

> frontRightTire와 backLeftTire를 교체하기 전에는 Tire 객체의 roll() 메소드가 호출되지만, HankookTire와 KumhoTire로 교체가 된 후에는 HankookTire와 KumhoTire 객체의 roll() 메소드가 호출된다. 만약 HankookTire와 KumhoTire가 roll() 메소드를 **재정의(오버라이딩)했다면 재정의된 roll() 메소드가 호출된다.**

7.7.3 하나의 배열로 객체 관리

: 위의 예시에서 Tire 객체를 4개의 필드로 각각 저장했다. 그렇다면 타이어 객체들도 타이어 배열로 관리하는 것이 코드를 깔끔하게 해준다.

```
class Car{
    Tire frontLeftTire = new Tire();
    Tire frontRightTire = new Tire();
    Tire backLeftTire = new Tire();
    Tire backRightTire = new Tire();
}

class Car{
    Tire[] tires = {
        new Tire("앞왼쪽", 6),
        new Tire("앞오른쪽", 2),
        new Tire("뒤왼쪽",3),
        new Tire("뒤오른쪽", 4)
    };
}

// 타이어를 교체할 때
tires[1] = new KumhoTire("앞오른쪽", 13);

// 전체 타이어의 roll() 메소드를 호출할 때
int run() {
    System.out.println("[자동차가 달립니다.]");
    for(int i=0; i<tires.length; i++){
        if(tires[i].roll()==false){
            stop();
            return (i+1);
        }
    }
    return 0;
}
```

7.7.4 매개 변수의 다형성

: 메소드를 호출할 때에 매개값을 다양화하기 위해 매개 변수에 자식 타입 객체를 지정할 수도 있다.

- **예시**

  : Driver 클래스에는 drive() 메소드가 정의되어 있는데 Vehicle 타입의 매개 변수가 선언되어 있다.

```
class Driver{
    void drive(Vehicle vehicle){
        vehicle.run();
    }
}
```

  drive 메소드를 정상적으로 호출된다면

```
Driver driver = new Driver();
Vehicle vehicle = new Vehicle();
driver.drive(vehicle);
```

  만약 Vehicle의 자식 클래스인 Bus 객체를 drive() 메소드의 매개값으로 넘겨준다면?

  자동 타입 변환이 발생한다.

```
Driver = driver = new Driver();
Bus bus = new Bus();
driver.drive(bus);      // Vehicle vehicle = bus; 와 같이 자동 타입 변환 발생
```

> 매개값으로 어떤 자식 객체가 제공되느냐에 따라 메소드의 실행 결과는 다양해질 수 있다.(자식의 오버라이딩된 메소드를 호출함으로써)

7.7.5 강제 타입 변환(Casting)

: 자식 타입이 부모 타입으로 자동 변환한 후, 다시 자식 타입으로 변환할 때 강제 타입 변환을 사용할 수 있다.

```
// 자식 타입이 부모 타입으로 변환된 상태에서 강제 타입 변환
자식클래스 변수 = (자식클래스) 부모클래스타입;
```

> 자식 타입이 부모 타입으로 자동 변환하면, 부모 타입에 선언된 필드와 메소드만 사용 가능하다. 만약 자식 타입에 선언된 멤버들을 사용해야 한다면 강제 타입 변환을 해야한다.

- **예시**

```
class Parent{
    String field1;
    void method1() { ... }
    void method2() { ... }
}

class Child extends Parent{
    String field2;
    void method3() { ... }
}

class ChildExample{
    public static void main(String[] args) {
        Parent parent = new Child();
        parent.field1 = "xxx";
        parent.method1();
        parent.method2();

        // 자식 클래스의 멤버에 접근
        parent.field2 = "yyy";  // 컴파일 에러!!
        parent.method3();               // 컴파일 에러!!

        // 자식 타입으로 강제 타입 변환
        Child child = (Child) parent;
        child.field2 = "yyy";   // 가능
        child.method3();                // 가능
    }
}
```

7.7.6 객체 타입 확인(instanceof)

: 강제 타입 변환은 자식 타입이 부모 타입으로 변환되어 있는 상태에서만 가능하기 때문에 다음과 같이 부모 타입의 변수가 부모 객체를 참조할 경우 자식 타입으로 변환할 수 없다.

```
Parent parent = new Parent();
Child child = (Child) parent;   // 강제 타입 변환을 할 수 없다.
```

- **instanceof** : 어떤 객체가 어떤 클래스의 인스턴스인지 확인하기 위함

```
// instanceof 연산자의 좌항은 객체, 우항은 타입
// 좌항의 객체가 우항의 인스턴스이면 true, 아니면 false
boolean result = 좌항(객체) instanceof 우항(타입)
```

  - **강제 타입 변환 예시**

```
public void method(Parent parent) {
    // Parent 매개 변수가 참조하는 객체가 Child인지 조사
    if(parent instanceof Child) {
        Child child = (Child) parent;
    }
}
```

- **예제**

  Parent.java(**부모 클래스**)

```
package instanceof_object;

public class Parent {
}
```

  Child.java(**자식 클래스**)

```
package instanceof_object;

public class Parent {
}
```

  InstanceofExample.java(**객체 타입 확인**)

```
package instanceof_object;

public class InstanceofExample {
    public static void method1(Parent parent) {
        // instanceof 연산자로 변환시킬 타입의 객체인지 조사
        if(parent instanceof Child) {
            Child child = (Child) parent;
            System.out.println("method1 - Child로 변환 O");
        } else {
            System.out.println("method1 - Child로 변환 X");
        }
    }

    public static void method2(Parent parent) {
        // 변환시킬 타입의 객체를 조사하지 않음
        Child child = (Child) parent;
        System.out.println("method2 - Child로 변환 O");
    }

    public static void main(String[] args) {
        // Child 객체를 매개값으로 전달
        Parent parentA = new Child();
        method1(parentA);
        method2(parentA);
```

```
        // Parent 객체를 매개값으로 전달
        Parent parentB = new Parent();
        method1(parentB);
        // method2(parentB); 예외 발생!
    }
}
```

실행 결과

```
method1 - Child로 변환 O
method2 - Child로 변환 O
method1 - Child로 변환 X
```

7.8 추상 클래스

7.8.1 추상 클래스의 개념

- **실체 클래스** : 객체를 직접 생성할 수 있는 클래스
- **추상 클래스** : 실체 클래스들의 공통적인 특성을 추출해서 선언한 클래스
  - **실체 클래스(자식) --> 추상 클래스(부모)** : 상속의 관계를 가지고 있다

  - 추상 클래스는 실체 클래스의 공통되는 필드와 메소드를 추출해서 만들었기 때문에 객체를 직접 생성해서 사용할 수 없다. 다시 말해서 **추상 클래스는 new 연산자를 이용해서 인스턴스를 생성시키지 못한다.**

    ```
    Animal animal = new Animal();   // X
    ```

  - 추상 클래스는 새로운 실체 클래스를 만들기 위해 **부모 클래스로만 사용된다.**

    ```
    class Ant extends Animal { ... } // O
    ```

7.8.2 추상 클래스의 용도

1. **실체 클래스들의 공통된 필드와 메소드의 이름을 통일할 목적**
2. **실체 클래스를 작성할 때 시간을 절약**

   : 공통적인 필드와 메소드는 추상 클래스에 선언해 두고, 실체 클래스마다 다른 점만 실체 클래스에 선언하게 되면 실체 클래스를 작성하는데 시간이 절약됨.

7.8.3 추상 클래스 선언

: 클래스 선언에 abstract 키워드를 붙인다. abstract를 붙이게 되면 new 연산자를 이용해서 객체를 만들지 못하고 상속을 통해 자식 클래스만 만들 수 있다.

```
public abstract class 클래스 {
    ...
}
```

- **예제**

  : 추상 클래스는 new 연산자로 직접 생성자를 호출할 수는 없지만 자식 객체가 생성될 때 super(...)를 호출해서 추상 클래스 객체를 생성하므로 **추상 클래스도 생성자가 반드시 있어야 한다.**

  Phone.java(**추상 클래스**)

  ```
  package abstract_class_declare;

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

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

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

  SmartPhone.java(**실체 클래스**)

```
package abstract_class_declare;

public class SmartPhone extends Phone{
    // 생성자
    public SmartPhone(String owner) {
        // Phone의 생성자 호출
        super(owner);
    }
    // 메소드
    public void internetSearch(){
        System.out.println("인터넷 검색을 합니다.");
    }
}
```

PhoneExample.java

```
package abstract_class_declare;

public class PhoneExample {
    public static void main(String[] args) {
        // 추상 클래스는 new로 객체를 만들지 못한다.
        // Phone phone = new Phone(); 컴파일 에러!!

        SmartPhone smartPhone = new SmartPhone("홍길동");

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

실행 결과

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

7.8.4 추상 메소드와 오버라이딩

: 추상 클래스는 실체 클래스의 멤버(필드, 메소드)를 통일화하는데 목적이 있다. 여기서 메소드가 선언만 통일화를 하고, 실행 내용은 실체 클래스마다 달라야 하는 경우가 있을 수 있다.

- **추상 메소드**
  - 추상 클래스에서만 선언 가능
  - 메소드의 선언부만 있고 중괄호 {}가 없는 상태
  - 하위 클래스가 반드시 실행 내용을 채우도록 하고 싶은 메소드가 있을 경우 사용
  - 자식 클래스는 추상 메소드를 재정의(오버라이딩)해서 사용
- **예시**

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

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

> Animal 클래스를 추상 클래스로 선언하고 sound() 메소드를 추상 메소드로 선언.
>
> Dog라는 자식 클래스에서 추상 메소드 오버라이딩.

확인 문제

1. 자바의 상속에 대한 설명 중 틀린 것은 무엇입니까?
    1. **자바는 다중 상속을 허용한다. (X, 허용하지 않는다.)**
    2. 부모의 메소드를 자식 클래스에서 재정의(오버라이딩)할 수 있다.
    3. 부모의 private 접근 제한을 갖는 필드와 메소드는 상속의 대상이 아니다.
    4. final 클래스는 상속할 수 없고, final 메소드는 오버라이딩할 수 없다.

2. 클래스 타입 변환에 대한 설명 중 틀린 것은 무엇입니까?
    1. 자식 객체는 부모 타입으로 자동 타입 변환된다.
    2. **부모 객체는 항상 자식 타입으로 강제 타입 변환된다. (X, 되지 않는 경우도 있다.)**
    3. 자동 타입 변환을 이용해서 필드와 매개 변수의 다형성을 구현한다.
    4. 강제 타입 변환 전에 instanceof 연산자로 변환 가능한지 검사하는 것이 좋다.

3. final 키워드에 대한 설명으로 틀린 것은?
    1. **final 클래스는 부모 클래스로 사용할 수 있다. (X, 상속이 되지 않는다.)**
    2. final 필드는 값이 저장된 후에는 변경할 수 없다.
    3. final 메소드는 재정의(오버라이딩)할 수 없다.
    4. static final 필드는 상수를 말한다.

4. 오버라이딩(Overriding)에 대한 설명으로 틀린 것은?
    1. 부모 메소드의 시그니처(리턴 타입, 메소드명, 매개 변수)와 동일해야 한다.
    2. 부모 메소드보다 좁은 접근 제한자를 붙일 수 없다.

(ex. public(부모)->private(자식))

3. @Override 어노테이션을 사용하면 재정의가 확실한지 컴파일러가 검증한다.
4. **protected 접근 제한을 갖는 메소드는 다른 패키지의 자식 클래스에서 재정의할 수 없다. (X, 할 수 있다.)**

5. Parent 클래스를 상속해서 Child 클래스를 다음과 같이 작성했는데, Child 클래스의 생성자에서 컴파일 에러가 발생했다. 그 이유는?

   Parent.java

```
public class Parent {
    public String name;

    public Parent(String name) {
        this.name = name;
    }
}
```

   Child.java

```
public class Child extends Parent {
    private  int studentNo;

    Child(String name, int studentNo) {
        this.name = name;
        this.studentNo = studentNo;
    }
}
```

> 이유는 자식 클래스인 Child의 생성자에서 부모 클래스의 생성자를 호출해주지 않았기 때문에 에러가 발생한다.
>
> **수정한 코드**
>
> ```
> public class Child extends Parent {
>  private  int studentNo;
>
>  Child(String name, int studentNo) {
>      super(name);       // 부모 생성자를 super로 호출해줘야 한다.
>      this.name = name;
>      this.studentNo = studentNo;
>  }
> }
> ```

6. Parent 클래스를 상속받아 Child 클래스를 다음과 같이 작성했습니다. ChildExample 클래스를 실행했을 때 호출되는 각 클래스의 생성자의 순서를 생각하면서 출력 결과를 작성해보세요.

   Parent.java

```
public class Parent{
    public String nation;

    public Parent() {
        this("대한민국");
        System.out.println("Parent() call");
    }

    public Parent(String nation) {
        this.nation = nation;
        System.out.println("Parent(String nation) call");
    }
}
```

Child.java

```
public class Child extends Parent {
    private String name;

    public Child() {
        this("홍길동");
        System.out.println("Child() call");
    }

    public Child(String name) {
        this.name = name;
        System.out.println("Child(String name) call");
    }
}
```

ChildExample.java

```
public class ChildExample {
    public static void main(String[] args) {
        Child child = new Child();
    }
}
```

실행 결과

```
Parent(String nation) call
Parent() call
Child(String name) call
Child() call
```

1. 부모 기본 생성자
2. 부모 매개 변수 있는 생성자
3. 자식 기본 생성자
4. 자식 매개 변수 있는 생성자

7. Tire 클래스를 상속받아 SnowTire 클래스를 다음과 같이 작성했습니다. SnowTireExample 클래스를 실행했을 때 출력 결과는 무엇일까요?

```
class Tire {
    public void run() {
        System.out.println("일반 타이어가 굴러간다.");
    }
}

class SnowTire extends Tire {
    public void run() {
        System.out.println("스노우 타이어가 굴러간다.");
    }
}

class SnowTireExample {
    public static void main(String[] args) {
        SnowTire snowTire = new SnowTire();
        Tire tire = snowTire;

        snowTire.run();
        tire.run();
    }
}
```

실행 결과

```
스노우 타이어가 굴러간다.
스노우 타이어가 굴러간다.
```

> snowTire.run()은 snowTire 클래스의 run() 메소드를 호출.
>
> tire.run()은 snowTire을 tire로 자동 타입 변환 시켰기 때문에 자식 클래스에 오버라이딩 된 SnowTire의 run() 메소드를 호출한다.

8. A, B, C, D, E, F 클래스가 다음과 같이 상속 관계에 있을 때 다음 빈칸에 들어올 수 없는 코드는?

```
B b = // 빈칸

void method(B b) { ... }
method(// 빈칸)
```

1. new B()
2. **(B) new A() // 강제 타입 변환 (X)**
3. new D() // 자동 타입 변환
4. new E() // 자동 타입 변환