07장 상속(Inheritance)

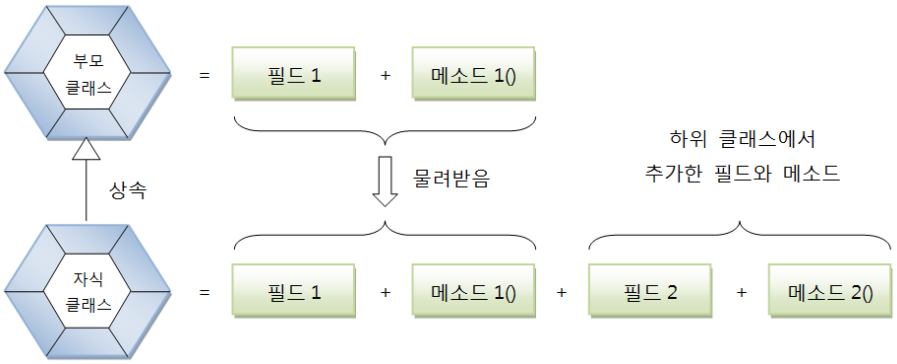
7.1 상속 개념

■ 현실 세계:
- 부모가 자식에게 물려주는 행위
- 부모가 자식을 선택해서 물려줌

■ 객체 지향 프로그램:
- 자식(하위, 파생) 클래스가 부모(상위) 클래스의 멤버를 물려받는 것
- 자식이 부모를 선택해 물려받음
- 상속 대상: 부모의 필드와 메소드

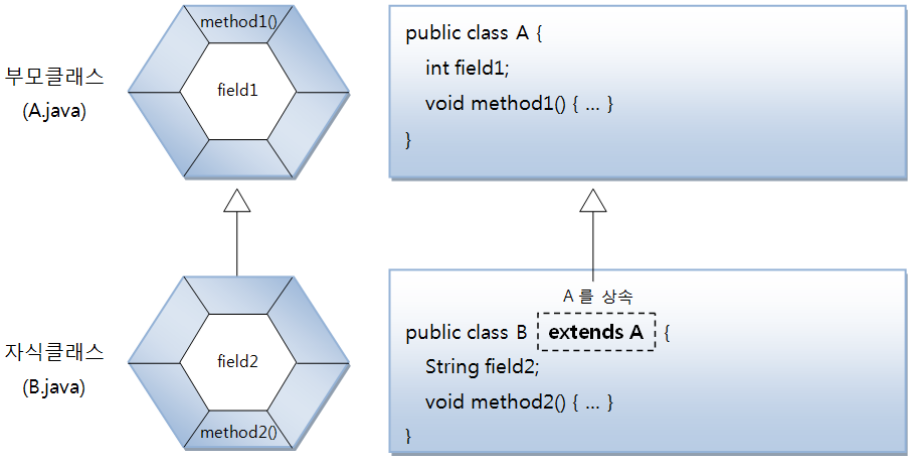
■ 상속의 효과
- 부모 클래스 재사용해 자식 클래스 빨리 개발 가능
- 반복된 코드 중복 줄임
- 유지 보수 편리성 제공
- 객체 다형성 구현 가능

■ 상속 대상 제한
- 부모 클래스의 private 접근 갖는 필드와 메소드 제외
- 부모 클래스가 다른 패키지에 있을 경우, default 접근 갖는 필드와 메소드도 제외



7.2 클래스 상속

■ 자식 클래스를 선언할 때 어떤 부모 클래스를 상속받을 것인지를 결정하고 선택된 부모 클래스는 extends 뒤에 기술한다.



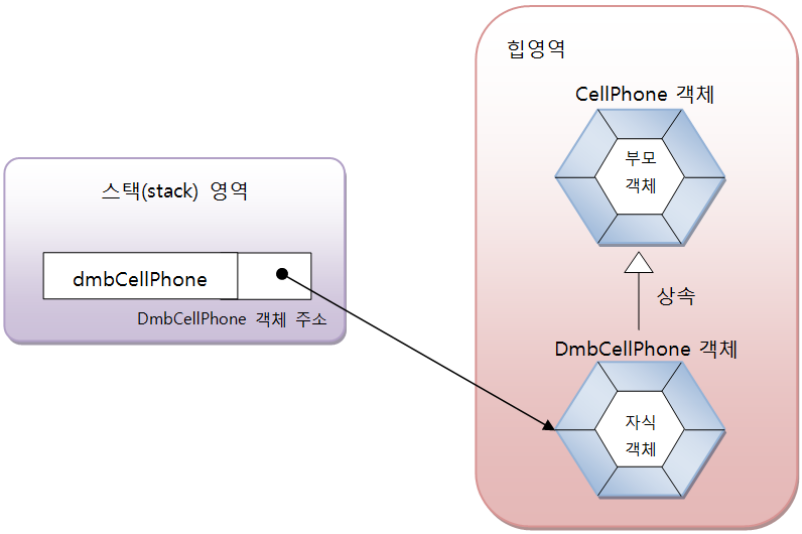
■ 자바는 단일 상속 - 부모 클래스 나열 불가

```
class 자식클래스 extends 부모클래스1, 부모클래스2 {
}
```

7.3 부모 생성자 호출

■ 자식 객체 생성하면 부모 객체도 생성되는가? -> 부모 없는 자식 없음
- 자식 객체 생성할 때는 부모 객체부터 생성 후 자식 객체 생성
- 부모 생성자 호출 완료 후 자식 생성자 호출 완료

```
DmbCellPhone dmbCellPhone = new DmbCellPhone();

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

public CellPhone() {
}
```



■ 명시적인 부모 생성자 호출
- 부모 객체 생성할 때, 부모 생성자 선택해 호출

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

- super(매개값,…): 매개값과 동일한 타입, 개수, 순서 맞는 부모 생성자 호출
- 부모 생성자 없다면 컴파일 오류 발생
- 반드시 자식 생성자의 첫 줄에 위치
- 부모 클래스에 기본(매개변수 없는) 생성자가 없다면 필수 작성

7.4 메소드 재정의

7.4.1 메소드 재정의(Method Overriding)

■ 부모 클래스의 상속 메소드 수정해 자식 클래스에서 재정의하는 것

■ 메소드 재정의 조건
- 부모 클래스의 메소드와 동일한 시그니처(리턴 타입, 메소드 이름, 매개 변수 리스트)를 가져야 한다.
- 접근 제한을 더 강하게 오버라이딩 불가
  - public을 default나 private으로 수정 불가
  - 반대로 default는 public 으로 수정 가능
  - 새로운 예외(Exception) throws 불가

■ @Override 어노테이션
- 컴파일러에게 부모 클래스의 메소드 선언부와 동일한지 검사 지시
- 정확한 메소드 재정의 위해 붙여주면 OK

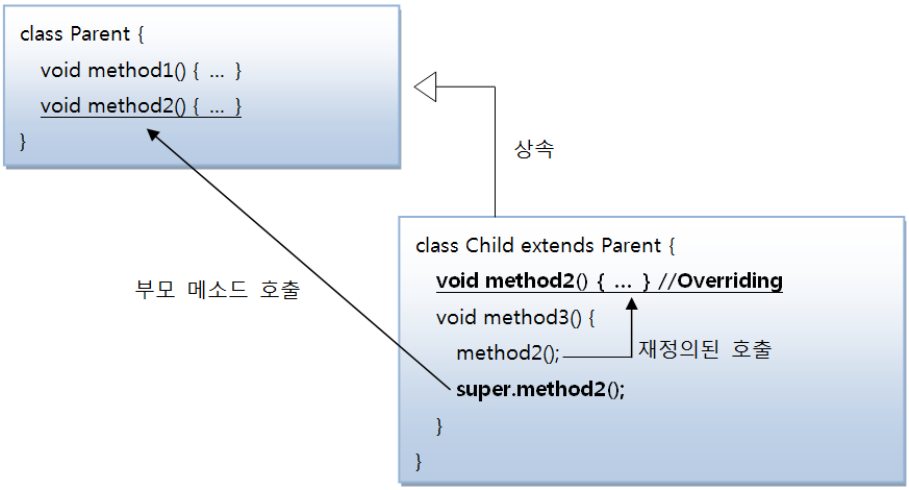
■ 메소드 재정의 효과
- 부모 메소드는 숨겨지는 효과 발생 -> 재정의된 자식 메소드 실행

```
class A{
  public void check(){
     System.out.println("부모 메소드");
  }
}

class B extends A{
   public void check(){  // 메소드 오버라이딩
     System.out.println("자식 메소드");
  }
}
```

7.4.2 부모 메소드 호출(super)

■ 메소드 재정의는 부모 메소드 숨기는 효과 !!
- 자식 클래스에서는 재정의된 메소드만 호출

■ 자식 클래스에서 수정되기 전 부모 메소드 호출 - super 사용
- super는 부모 객체 참조(참고: this는 자신 객체 참조)



7.5 final 클래스와 final 메소드

■ final이 멤버변수에 사용될 경우 -> 상수(값을 수정할 수 없다)

■ final이 메소드에 사용될 경우 -> 메소드 오버라이딩을 허용하지 않는다는 의미

■ final이 클래스에 사용될 경우 -> 상속을 허용하지 않는다는 의미

[FinalTest02.java] final method

```
class FinalMethod {
	String str = "Java ";

	// public void setStr(String s) {
	// final 붙이면 서브 클래스에서 오버라이딩이 불가.
	public final void setStr(String s) {  // method overriding을 허락하지 않는다.
		str = s;
		System.out.println(str);
	}
}

class FinalEx extends FinalMethod {
	int a = 10; // final 붙이면 밑에서 a값 대입 불가.

	public void setA(int a) {
		this.a = a;
	}

	public void setStr(String s) {  // method overriding할 수 없어 에러가 발생한다.
		str += s;
		System.out.println(str);
	}
```

```
}

public class FinalTest02 {
	public static void main(String[] args) {
		FinalEx ft = new FinalEx();
		ft.setA(100);
		ft.setStr("hi");// 슈퍼 클래스의 setStr을 실행.
		FinalMethod ft1 = new FinalMethod();
		ft1.setStr("hi");// 자신의 클래스의 setStr을 실행.
	}
}
```

[FinalTest03.java] final class

```
final class FinalClass {  // 상속을 허락하지 않는다.
	String str = "Java ";

	public void setStr(String s) {
		str = s;
		System.out.println(str);
	}
}

class FinalEx extends FinalClass {  // 상속받을 수 없어 에러 발생한다.
	int a = 10;

	public void setA(int a) {
		this.a = a;
	}

	public void setStr(String s) {
		str += s;
		System.out.println(str);
	}
}

public class FinalTest03 {
	public static void main(String[] args) {
		FinalEx fe = new FinalEx();
	}
}
```

7.6 protected 접근 제한자

접근 제한자	같은 클래스	같은 패키지	자식 클래스	다른 패키지
private	O	X	X	X
생략(default)	O	O	X (O)	X
protected	O	O	O	X (O)
public	O	O	O	O

()는 상속관계인 경우이다.

[SuperSubA.java] 접근 제어자 사용법

```
Import package1.AccessTest;

//AccessTest의 서브 클래스로 SubOne을 설계
class SubOne extends AccessTest {
	void subPrn() {
		System.out.println(a); // [1. Sub] private -X
		System.out.println(b); // [2. Sub] 기본 접근 지정자-X
```

```
                    System.out.println(c); // [3. Sub] protected -O
                    System.out.println(d); // [4. Sub] public -0
          }
}

// AccessTest랑 상속관계가 없는 클래스
class SuperSubA {
          public static void main(String[] args) {
                    AccessTest at = new AccessTest();
                    at.print();
                    System.out.println("main");
                    System.out.println(at.a); // [1. main] private -X
                    System.out.println(at.b); // [2. main] 기본 접근 지정자-X
                    System.out.println(at.c); // [3. main] protected -X
                    System.out.println(at.d); // [4. main] public -O
          }
}
```

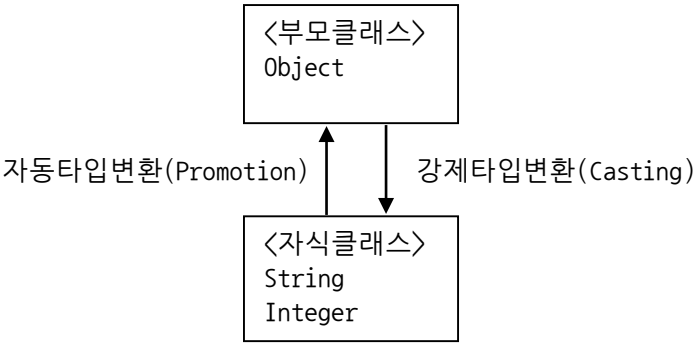
[package1/AccessTest.java] 접근 제어자 사용법

```
package package1;

public class AccessTest { // 다른 패키지에서 가져다 사용할 것임으로 pubic으로
          private int a = 10; // [1] private
          int b = 20; // [2] 기본 접근 지정자
          protected int c = 30; // [3] protected
          public int d = 40; // [4] public

          public void print() {
                    System.out.println("AccessTest의 print");
                    System.out.println(a);
                    System.out.println(b);
                    System.out.println(c);
                    System.out.println(d);
          }
}
```

7.7 타입 변환과 다형성

7.7.1 타입 변환

■ 클래스 타입의 변환은 상속 관계에 있는 클래스 사이에서 발생한다.



(1) 자동 타입 변환 (Promotion)

- 서브클래스에서 슈퍼클래스로 형변환 하는 것
- 참조 가능한 영역이 축소가 된다. (상속한 것만을 참조할 수 있음)
- 컴파일러에 의해서 암시적 형변환(자동 형변환) 된다.

```
Parent p;
Child c = new Child();
p = c;  //업캐스팅(암시적 형변환), 자식의 주소를 부모에게 할당

Parent p = new Child();
Parent p = (Parent) new Child();

//예1
java.util.Calendar
     java.util.GregorianCalendar
Calendar cal = new GregorianCalendar(); //업캐스팅

//예2
java.lang.Object
    java.lang.String
          boolean equals(Object an)     // 매개변수가 부모클래스인 경우

if("java".equals("jsp")){  // 업캐스팅
// Object  an = new String("jsp");  // 업캐스팅
// if("java".equals(an){
}

if(new Integer(30).equals(new Integer(50))  // Object  an = new Integer(50);  // 박싱 + 업캐스팅
if(new Integer(30).equals(50))  // Object an = 50; // 오토박싱 + 업캐스팅
```

(2) 강제 타입 변환 (Casting)

- 슈퍼클래스에서 서브클래스로 형변환 하는것
- 참조 가능한 영역이 확대가 된다.
- 컴파일러에 의해서 암시적 형변환이 되지 않기 때문에 자료형을 생략할 수 없다.(강제 형변환)
- 자식 타입이 부모 타입으로 자동 변환한 후, 다시 자식 타입으로 변환할 때 강제형변환을 사용할 수 있다.

```
Child c;
Parent p = new Child();  // 업캐스팅
c = (Child) p;  // 다운 캐스팅(명시적 형변환), Child c = (Child) new Parent();

Vector v = new Vector();
v.add("java");
v.add("jsp");
for(int i=0; i<v.size(); i++){
    String s = (String) v.get(i);  //다운 캐스팅, String <- Object
    Object s1 = v.get(i);
}
```

(3) 객체 타입 확인 (instanceof)

- instanceof 연산자는 어떤 객체가 어떤 클래스(타입)의 인스턴스인지 확인하기 위해 사용한다.
- 형식: boolean result = 좌항(객체) instanceof 우항(타입)

[InstanceofExample.java] instanceof를 통한 객체 타입 확인

```
public class InstanceofExample {
	public static void method1(Parent parent) {
		if (parent instanceof Child) {
			Child child = (Child) parent;
			System.out.println("method1 - Child로 변환 성공");
		} else {
			System.out.println("method1 - Child로 변환되지 않음");
		}
	}

	public static void method2(Parent parent) {
		Child child = (Child) parent;
		System.out.println("method2 - Child로 변환 성공");
	}

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

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

7.7.2 다형성(polymorphism)

- 다형성은 같은 타입이지만 실행 결과가 다양한 객체를 이용할 수 있는 성질을 말한다. 다형성을 위해 자바는 부모 클래스로 타입 변환을 허용한다. 즉, 부모 타입에 모든 자식 객체가 대입될 수 있다.

(1) 필드의 다형성

■ 필드의 타입은 변함이 없지만 어떤 객체를 필드로 저장하느냐에 따라 실행 결과를 다양화할 수 있다.

[Car.java]

```
package cp07_inheritance.se07_casting.ex02_polymorphism.fields;

public class Car {
	//필드
	Tire frontLeftTire = new Tire("앞왼쪽", 6);
	Tire frontRightTire = new Tire("앞오른쪽", 2);
	Tire backLeftTire = new Tire("뒤왼쪽", 3);
	Tire backRightTire = new Tire("뒤오른쪽", 4);

	//생성자

	//메소드
	int run() { ... }
}
```

[CarExample.java]

```
package cp07_inheritance.se07_casting.ex02_polymorphism.fields;

public class CarExample {
	public static void main(String[] args) {
		Car car = new Car();

		for (int i = 1; i <= 5; i++) {
			int problemLocation = car.run();
			switch (problemLocation) {
			case 1:
				System.out.println("앞왼쪽 HankookTire로 교체");
				car.frontLeftTire = new HankookTire("앞왼쪽", 15);
				// Tire tire = new HankookTire("앞왼쪽", 15);  // 업캐스팅
				// car.frontLeftTire = tire;
```

```
                                        break;
                                case 2:
                                        System.out.println("앞오른쪽 KumhoTire로 교체");
                                        car.frontRightTire = new KumhoTire("앞오른쪽", 13);
                                        break;
                                case 3:
                                        System.out.println("뒤왼쪽 HankookTire로 교체");
                                        car.backLeftTire = new HankookTire("뒤왼쪽", 14);
                                        break;
                                case 4:
                                        System.out.println("뒤오른쪽 KumhoTire로 교체");
                                        car.backRightTire = new KumhoTire("뒤오른쪽", 17);
                                        break;
                                }
                                System.out.println("-----------------------------------------");
                        }
                }
        }
```

(2) 하나의 배열로 객체 관리

■ 동일한 타입의 값들은 배열로 관리하는 것이 유리하다.

[Car.java]

```
01    package sec07.exam04_array_management;
02
03    public class Car {
04            //필드
05            Tire[] tires = {
06                                    new Tire("앞왼쪽", 6),
07                                    new Tire("앞오른쪽", 2),
08                                    new Tire("뒤왼쪽", 3),
09                                    new Tire("뒤오른쪽", 4)
10            };
11
12            //메소드
13            int run() {
14                    System.out.println("[자동차가 달립니다.]");
15                    for(int i=0; i<tires.length; i++) {
16                            if(tires[i].roll()==false) {
17                                    stop();
18                                    return (i+1);
19                            }
20                    }
21                    return 0;
22            }
23
24            void stop() {
25                    System.out.println("[자동차가 멈춥니다.]");
26            }
27    }
```

(3) 매개 변수의 다형성

■ 매소드를 호출할 때에는 매개 변수의 타입과 동일한 매개값을 지정하는 것이 정석이지만, 매개값을 다양화하기 위해 메소드의 매개 변수에 자식 타입 객체를 지정할 수 있다.

[Bus.java]

```
package cp07_inheritance.se07_casting.ex02_polymorphism.methods;

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

[Driver.java]

```
package cp07_inheritance.se07_casting.ex02_polymorphism.methods;

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

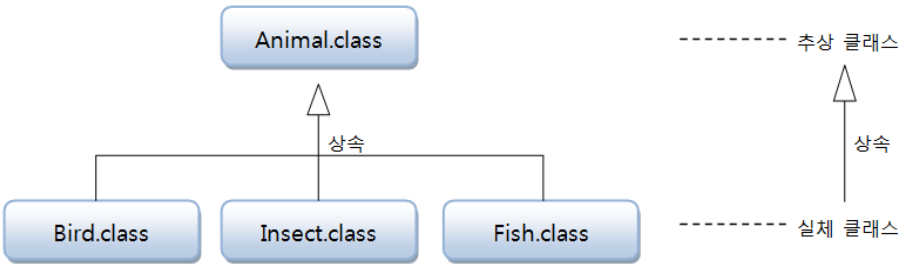
[DriverExample.java]

```
package cp07_inheritance.se07_casting.ex02_polymorphism.methods;

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

		Bus bus = new Bus();
		Taxi taxi = new Taxi();

		driver.drive(bus);
		//Vehicle vehicle = new Bus(); // 자동타입변환, 업캐스팅
		//driver.drive(vehicle);

		driver.drive(taxi);
	}
}
```

7.8 추상 클래스

7.8.1 추상 클래스의 개념

- 실체 클래스들의 공통되는 필드와 메소드를 정의한 클래스
- 추상 클래스는 실체 클래스의 부모 클래스 역할 (단독 객체 X)




```
abstract class Animal {}
Animal animal = new Animal();    //(X) 자체적으로 객체를 생성할 수 없다.
class Bird extends Animal {...}  //추상 클래스를 상속받는 일반 클래스를 생성한다.
Bird bird = new Bird();          // (O)
```

7.8.2 추상 클래스의 용도

- 첫째, 실체 클래스들의 공통된 필드와 메소드의 이름을 통일한 목적
  - 실체 클래스를 설계자가 여러 사람일 경우, 실체 클래스마다 필드와 메소드가 제각기 다른 이름을 가질 수 있음

- 둘째, 실체 클래스를 작성할 때 시간을 절약
  - 실체 클래스는 추가적인 필드와 메소드만 선언

```
public class Telephone {
   public String owner;
   public void turnOn() {...}
}
public class SmartPhone {
   public String user;
   public void powerOn() {...}
}

public abstract class Phone {
   public String owner;
   public Phone(String owner) {  //생성자
      this.owner = owner;
   }
   public void turnOn();
}
public class SmartPhone extends Phone {
   public SmartPhone(String owner) {  //생성자
      super(owner);
   }
   public void internetSearch() {...}  //추가적인 메소드만 선언한다.
}
pulbic class Telephone extends Phone {...}
public class PhoneExample {
   public static void main(String[] args) {
      SmartPhone smartPhone = new SmartPhone("홍길동");
      Telephone telephone = new Telephone("홍길동");
      smartPhone.turnOn();
      telephone.turnOn();
}
```

7.8.3 추상 클래스 선언

- 추상 클래스를 구성하는 요소는 멤버변수, 추상 메소드와 일반메소드가 있다.
- 추상 클래스는 New 연산자로 객체 생성하지 못하고 extends를 이용해서 자식 클래스에서 상속하여 사용된다. 그러나 단일 상속만 가능하다.(클래스의 다중상속을 허용하지 않는다.)

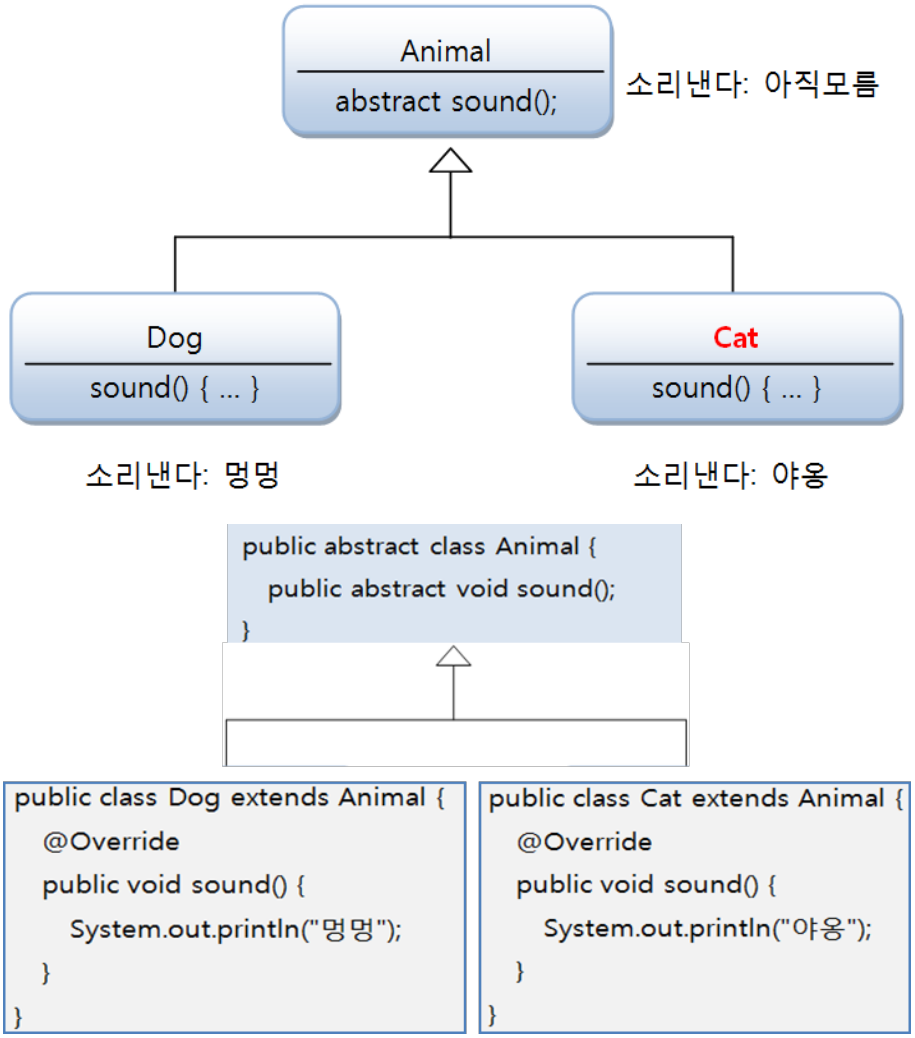
```
// 기본형
public abstract class 클래스 {
	// 필드
	// 생성자
	// 일반 메소드
	// 추상 메소드
}

abstract class AbsClass{                //추상 클래스
	int a=10;                           //멤버변수
	abstract void Method01();           //추상 메소드
	void Method02(){                    //일반메소드
	}
}
```

[Phone.java] 추상 클래스의 선언 방법

```
package cp07_inheritance.se08_abstract.classes;

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.");
	}
}
```

7.8.4 추상 메소드와 오버라이딩

- 추상 클래스를 상속받은 자식 클래스는 추상 클래스 안에 들어있는 추상 메소드를 반드시 Method Overriding해야 된다.



[Animal.java] 추상 메소드 선언

```
public abstract class Animal {
	public String kind;

	public void breathe() {
		System.out.println("숨을 쉽니다.");
	}

	public abstract void sound();
}
```

[Dog.java] 추상 메소드 오버라이딩

```
public class Dog extends Animal {
	public Dog() {
		this.kind = "포유류";
	}

	@Override
	public void sound() {
		System.out.println("멍멍");
	}
}
```

[Cat.java] 추상 메소드 오버라이딩

```
public class Cat extends Animal {
	public Cat() {
		this.kind = "포유류";
	}

	@Override
	public void sound() {
		System.out.println("야옹");
	}
}
```

[AnimalExample.java] 실행 클래스

```
package sec08.exam02_abstract_method;

public class AnimalExample {
	public static void main(String[] args) {
		Dog dog = new Dog();
		Cat cat = new Cat();
		dog.sound();
		cat.sound();
		System.out.println("-----");

		// 변수의 자동 타입 변환
		Animal animal = null;
		animal = new Dog();
		animal.sound();
		animal = new Cat();
		animal.sound();
		System.out.println("-----");

		// 매개변수의 자동 타입 변환
		animalSound(new Dog());
		animalSound(new Cat());
	}

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

[과제] 확인문제

1. 자바의 상속에 대한 설명 중 틀린 것은 무엇입니까?

(1) 자바는 다중 상속을 허용한다.
(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.
(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.
(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 시그너처(리턴 타입, 메소드명, 매개 변수)와 동일해야 한다.
(2) 부모 메소드보다 좁은 접근 제한자를 붙일 수 없다. 예) 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.

5. Parent 클래스를 상속해서 Child 클래스를 다음과 같이 작성했는데, Child 클래스의 생성자에서 컴파일 에러가 발생했습니다. 그 이유를 설명해보세요.

[Parent.java]

```
01    package verify.exam05;
02
03    public class Parent {
04              public String name;
05
06              public Parent(String name) {
07                        this.name = name;
08              }
09    }
```

[Child.java]

```
01    package verify.exam05;
02
03    public class Child extends Parent {
04              private int studentNo;
05
06              public Child(String name, int studentNo) {
07                        this.name = name;
08                        this.studentNo = studentNo;
09              }
10    }
```

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

[Parent.java]

```
01    package verify.exam06;
02
03    public class Parent {
04              public String nation;
05
06              public Parent() {
07                        this("대한민국");
08                        System.out.println("Parent() call");
09              }
10
11              public Parent(String nation) {
12                        this.nation = nation;
13                        System.out.println("Parent(String nation) call");
14              }
15    }
```

[Child.java]

```
01    package verify.exam06;
02
03    public class Child extends Parent {
04              private String name;
05
06              public Child() {
07                        this("홍길동");
08                        System.out.println("Child() call");
09              }
10
11              public Child(String name) {
12                        this.name = name;
13                        System.out.println("Child(String name) call");
14              }
15    }
```

[ChildExample.java]

```
01    package verify.exam06;
02
03    public class ChildExample {
04              public static void main(String[] args) {
05                        Child child = new Child();
06              }
07    }
```

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

[Tire.java]

```
01    package verify.exam07;
02
03    public class Tire {
04              public void run() {
05                        System.out.println("일반 타이어가 굴러갑니다.");
06              }
07    }
```

[SnowTire.java]

```
01    package verify.exam07;
02
03    public class SnowTire extends Tire {
04              @Override
05              public void run() {
06                        System.out.println("스노우 타이어가 굴러갑니다.");
07              }
08    }
```

[SnowTireExample.java]

```
01    package verify.exam07;
02
03    public class SnowTireExample {
04              public static void main(String[] args) {
05                        SnowTire snowTire = new SnowTire();
06                        Tire tire = snowTire;
07
08                        snowTire.run();
09                        tire.run();
10              }
11    }
```