KB금융그룹 | 국민의 평생 금융파트너

2025년 상반기 K-디지털 트레이닝

상속

[KB] IT's Your Life

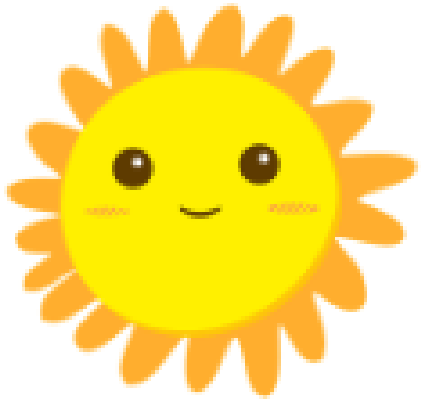
상속
(재사용)
extends

상속시
생성자

오버라이드
(재정의)

클래스 형변환
(다형성)

KB국민은행

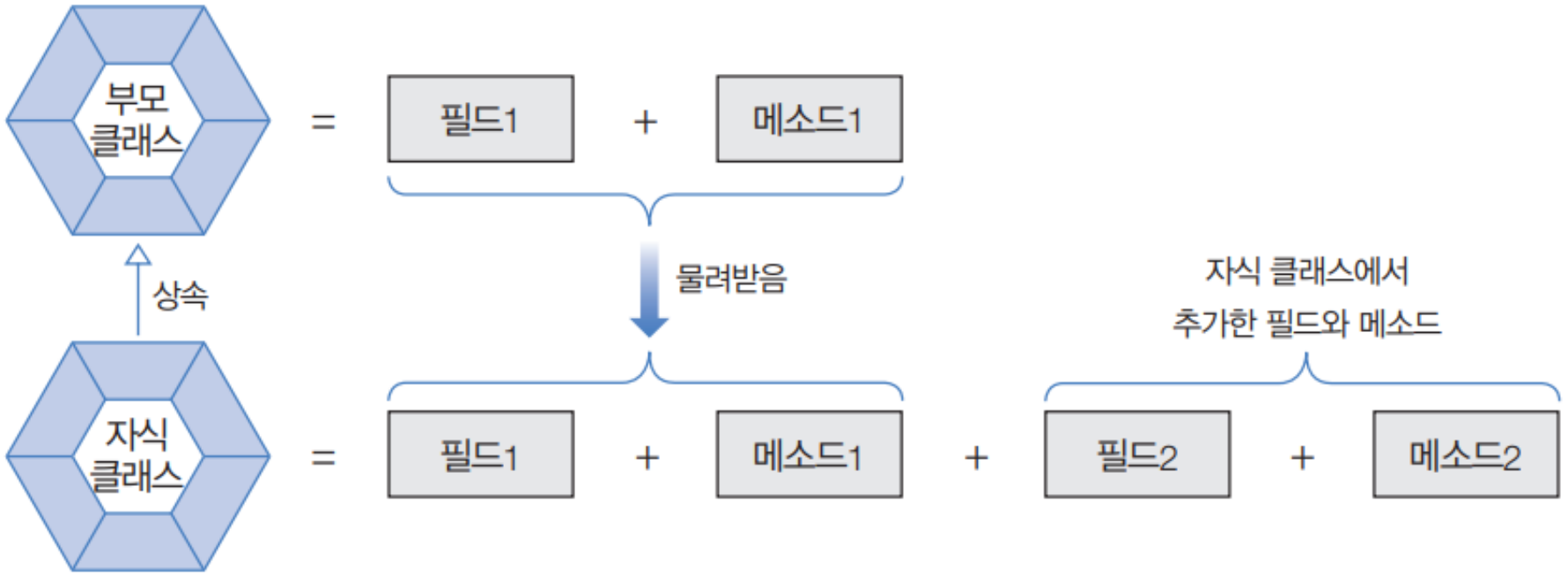


상속(재사용) 개념

extends

상속

- 클래스를 만들 때 기존에 있던 클래스를 이용해 확장해 만드는 것
- 기존 클래스를 재사용
- 부모 클래스의 필드와 메소드를 자식 클래스에게 물려줄 수 있음



상속


```
달리다(){
  //바퀴가 지면위에서 앞으로 가게 하다.
}
```

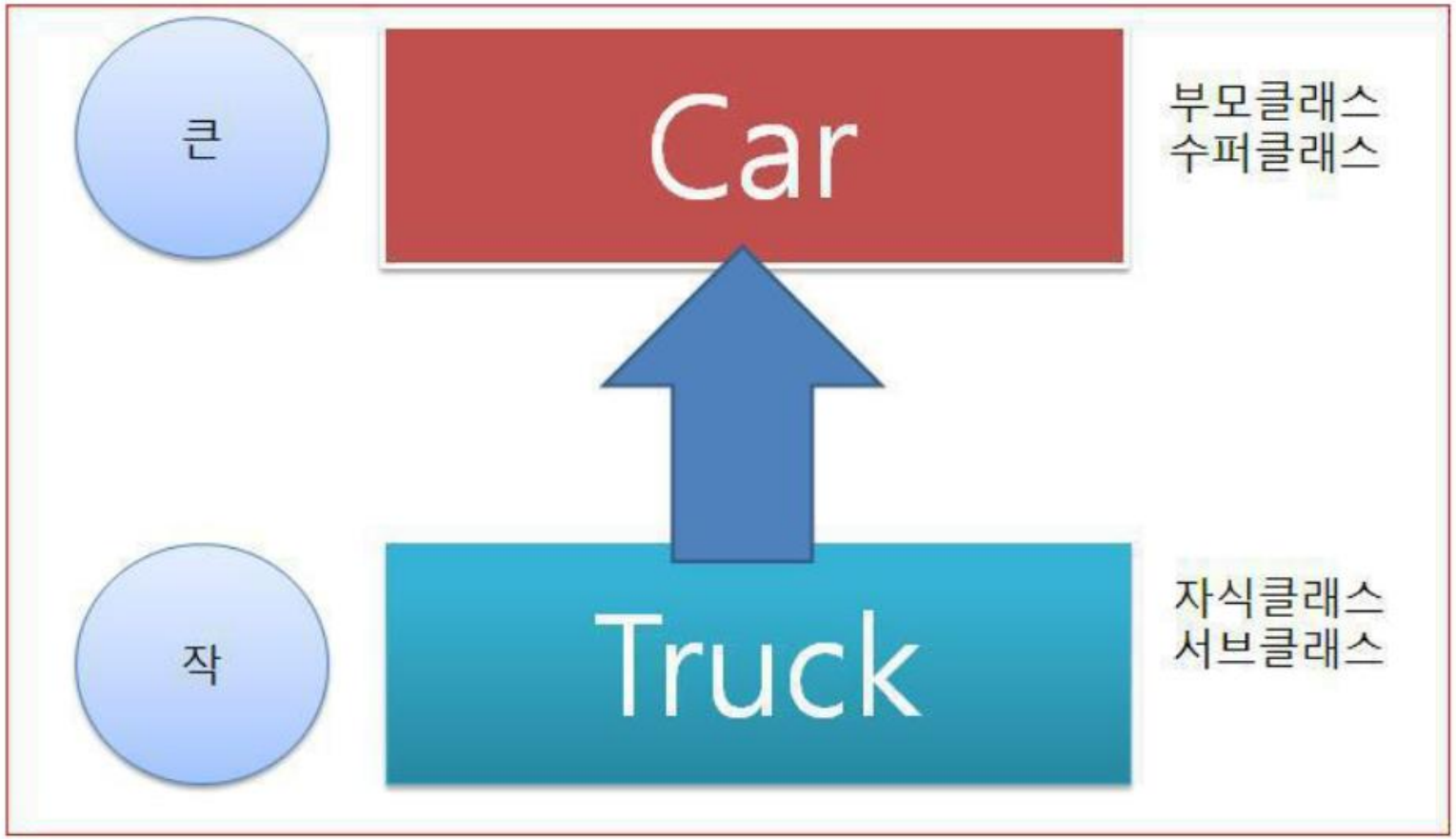
기능의 재정의(오버라이드)

```
달리다(){
  //공중에서 앞으로 나아가다.
}
```

단일상속O

다중상속X

개념의 크기(is-a관계. 자식이 부모이다.)

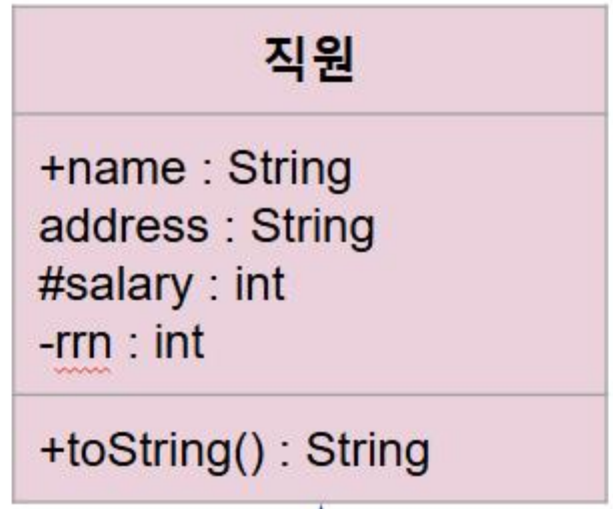


부모클래스/상위클래스

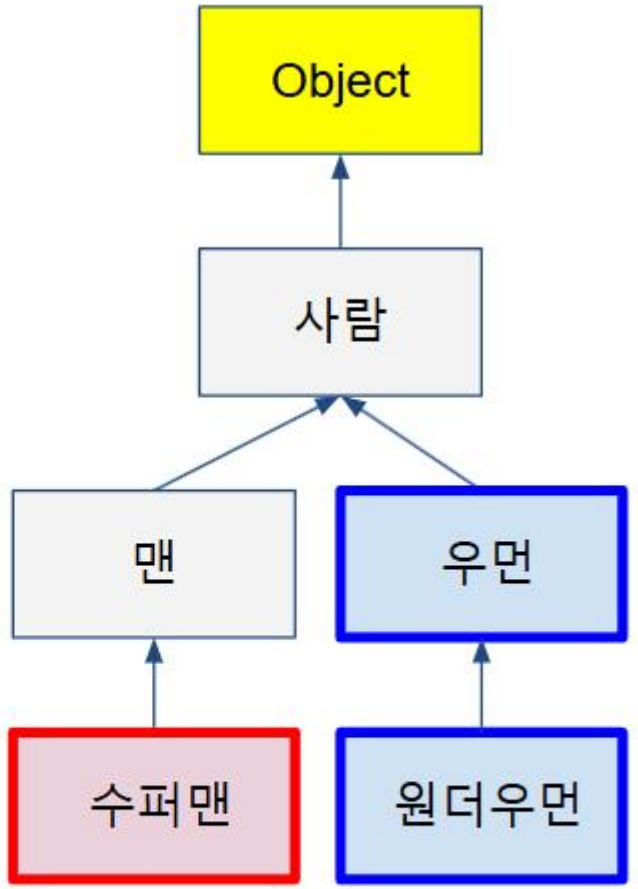
Parent Class
Super Class

자식클래스/하위클래스

Child Class
Sub Class



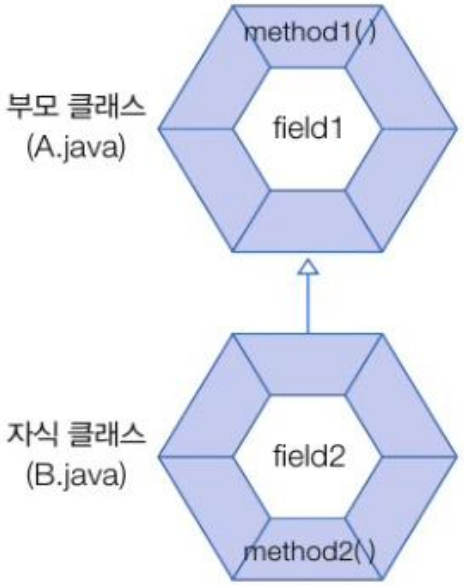
```
사람        - 성별, 이름
            - 잠자다, 먹다
맨= 사람 + 맨 특징
            - 힘크기
            - 달리다
수퍼맨= 맨 + 수퍼맨의 특징
            - 날아다니다.
스파이더맨= 맨 + 스파이더맨의 특징
            - 벽을 탄다
우먼사용!! 변수+메소드 호출 확인
```



클래스 다이어그램

상속의 이점

- 이미 개발된 클래스를 재사용하므로 중복 코드를 줄임
- 클래스 수정을 최소화




```
public class A {
  int field1;
  void method1() { … }
}
```

A를 상속

```
public class B extends A {
  String field2;
  void method2() { … }
}
```

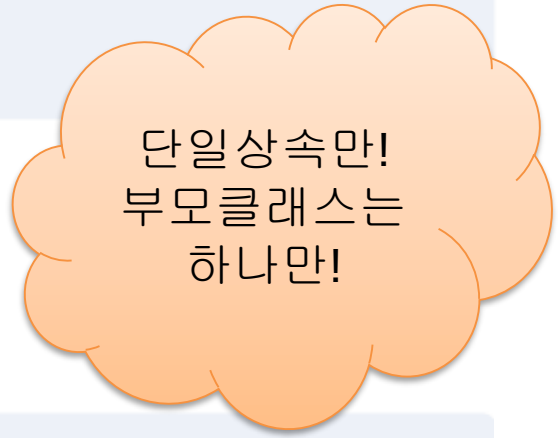
```
B b = new B();
b.field1 = 10;
b.method1();
```
} A로부터 물려받은 필드와 메소드

```
b.field2 = "홍길동";
method2();
```
} B가 추가한 필드와 메소드

클래스 상속

- 자식 클래스를 선언할 때 어떤 부모로부터 상속받을 것인지를 결정하고, 부모 클래스를 다음과 같이 extends 뒤에 기술

```
public class 자식클래스 extends 부모클래스 {
}
```

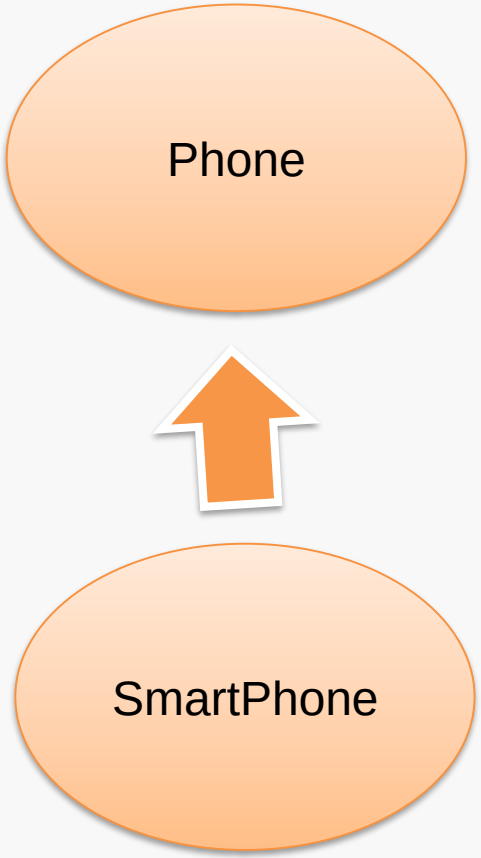


- 다중 상속 허용하지 않음. extends 뒤에 하나의 부모 클래스만 상속

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

Phone.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.");
		}
}
```

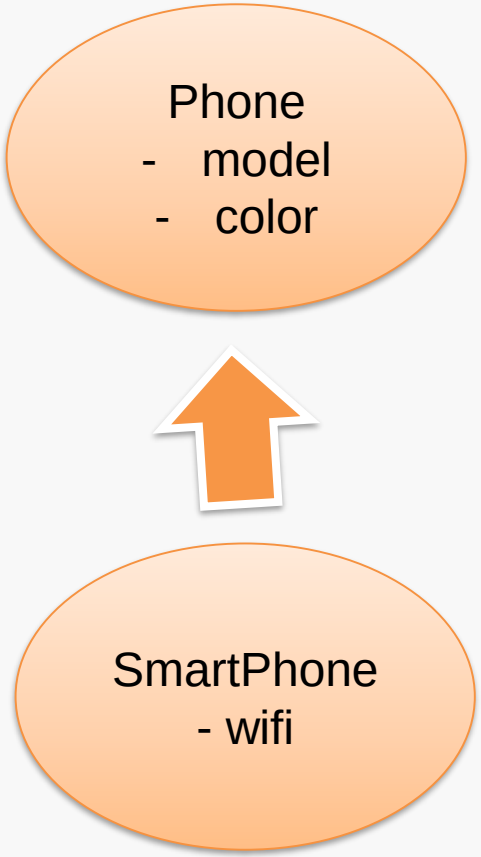


SmartPhone.java

```
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.");
	}
}
```

SmartPhoneExample.java

```
package ch07.sec02;

public class SmartPhoneExample {

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

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



SmartPhoneExample.java

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

```
모델: 갤럭시
색상: 은색
와이파이 상태: false
벨이 울립니다.
자기: 여보세요.
상대방: 안녕하세요! 저는 홍길동인데요.
자기: 아~ 네, 반갑습니다.
전화를 끊습니다.
와이파이 상태를 변경했습니다.
인터넷에 연결합니다.
```

상속시 생성자

- **부모객체부터** 생성자를 호출해서 **먼저 만들어** 놓아야 자식을 만들 때 문제가 없다.

부모 생성자 호출

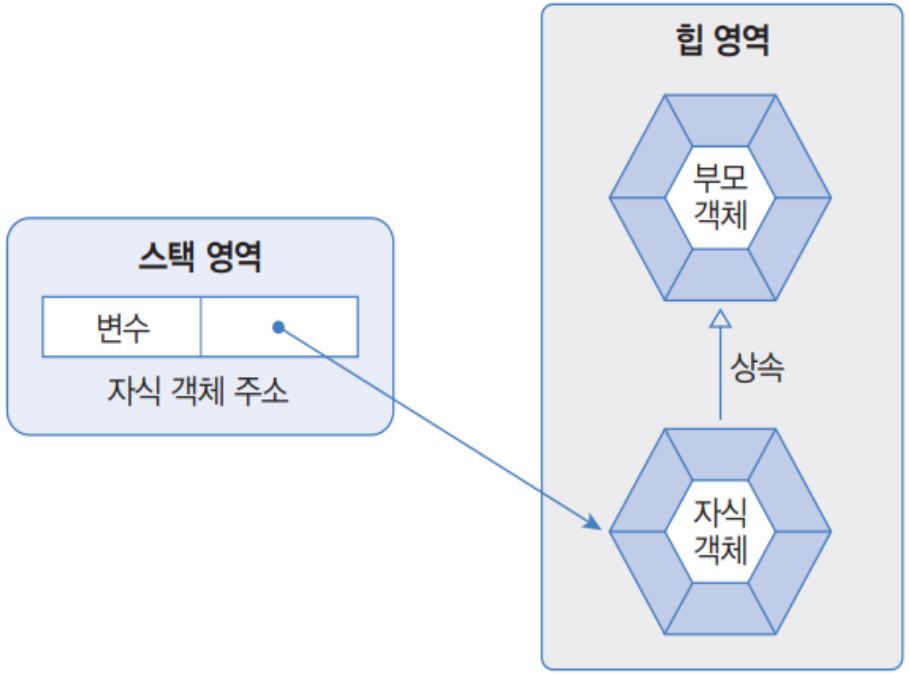
- 자식 객체를 생성하면 부모 객체가 먼저 생성된 다음에 자식 객체가 생성해야함.
- 부모의 생성자를 먼저 호출하게 함. super();, super(필드, 필드);

```
자식클래스 변수 = new 자식클래스( );
```

- 부모 생성자는 자식 생성자의 맨 첫 줄에 숨겨져 있는 super()에 의해 호출

```
//자식 생성자 선언
public 자식클래스(…) {
  super();
  …
}
```

```
//자식 생성자 선언
public 자식클래스(…) {
  super(매개값, … );
  …
}
```



Phone.java

```
package ch07.sec03.exam01;

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

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

SmartPhone.java

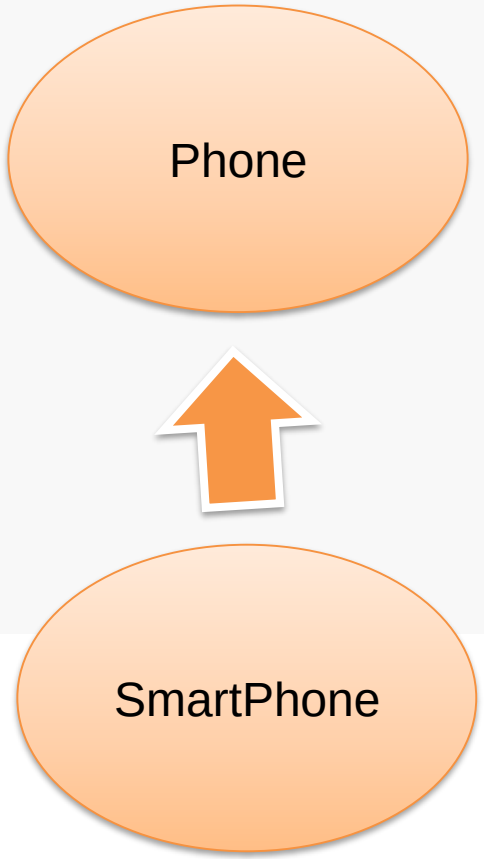
자식클래스

SmartPhone() 실행 전에 Phone()이 먼저 만들어져야 함.

```
package ch07.sec03.exam01;

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

SmartPhoneExample.java

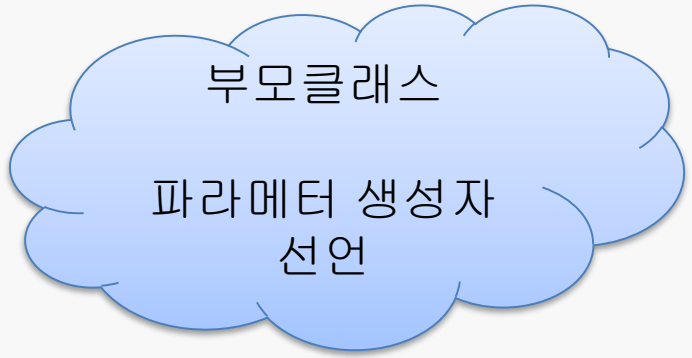
```
package ch07.sec03.exam01;

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

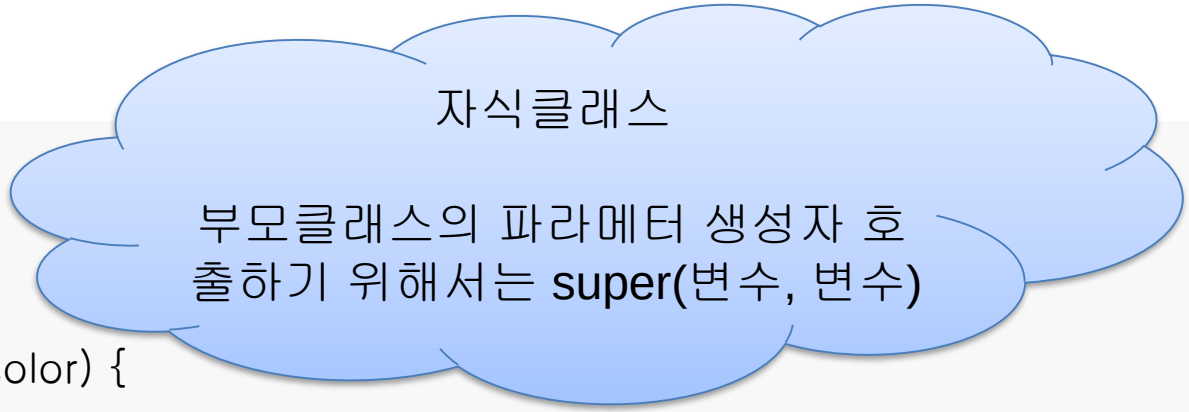
Phone.java

```
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");
        }
}
```



SmartPhone.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");
	}
}
```



SmartPhoneExample.java

```
package ch07.sec03.exam02;

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
```

오버라이드

- 부모 클래스의 필드와 메서드를 자식 클래스에서 **재정의**해 사용 가능하다.

오버라이드(재정의)

상속

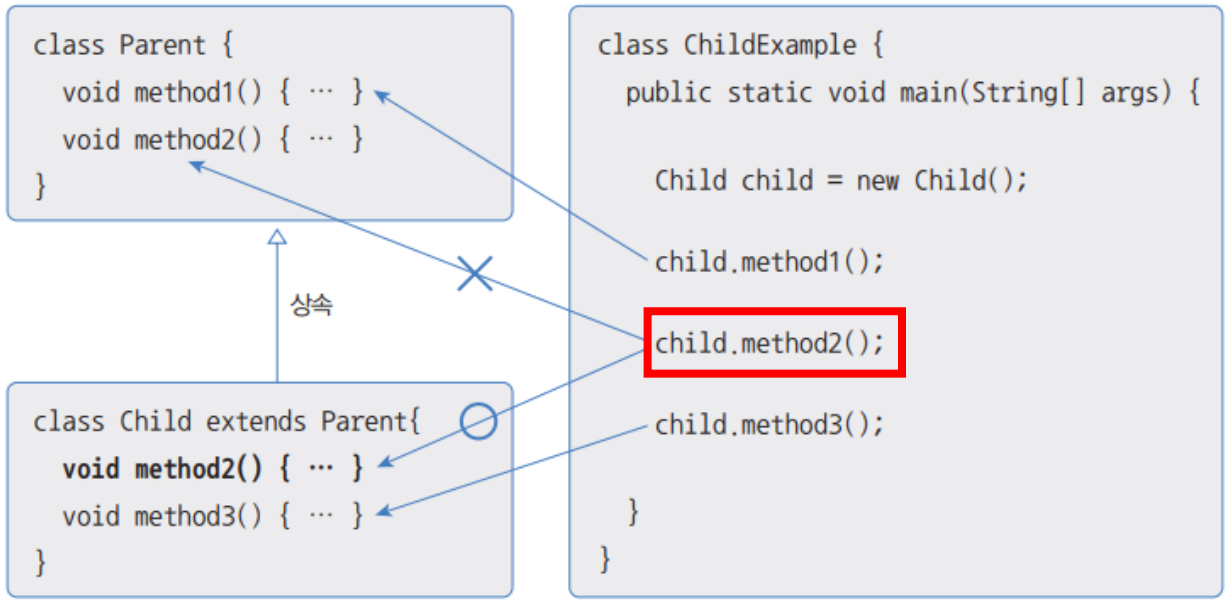

```
달리다(){
  //바퀴가 지면위에서 앞으로 가게 하다.
}
```

기능의 재정의(오버라이드)

```
달리다(){
  //공중에서 앞으로 나아가다.
}
```

메소드 오버라이딩

- 상속된 메소드를 자식 클래스에서 **재정의**하는 것.
- 해당 부모 메소드는 숨겨지고, 자식 메소드가 우선적으로 사용



method2()는 Parent, Child class에 있게됨.
자식객체를 만들고 method2()를 호출하면 자식의 메서드가 호출됨.

- 부모 메소드의 선언부(**리턴 타입, 메소드 이름, 매개변수**)와 **동일해야** 함 → 완전히 같아야함.
- 접근 제한을 더 강하게 오버라이딩할 수 없음(public → private으로 변경 불가)
- 새로운 예외를 throws할 수 없음

Calculator.java

```
package ch07.sec04.exam01;

public class Calculator {
	//메소드 선언
	public double areaCircle(double r) {
		System.out.println("Calculator 객체의 areaCircle() 실행");
		return 3.1459 * r * r;
	}
}
```

Computer.java

```
package ch07.sec04.exam01;

public class Computer extends Calculator {
	//메소드 오버라이딩
	@Override		// 컴파일 시 정확히 오버라이딩이 되었는지 체크 해줌
	public double areaCircle(double r) {
		System.out.println("Computer 객체의 areaCircle() 실행");
		return Math.PI * r * r;
	}
}
```

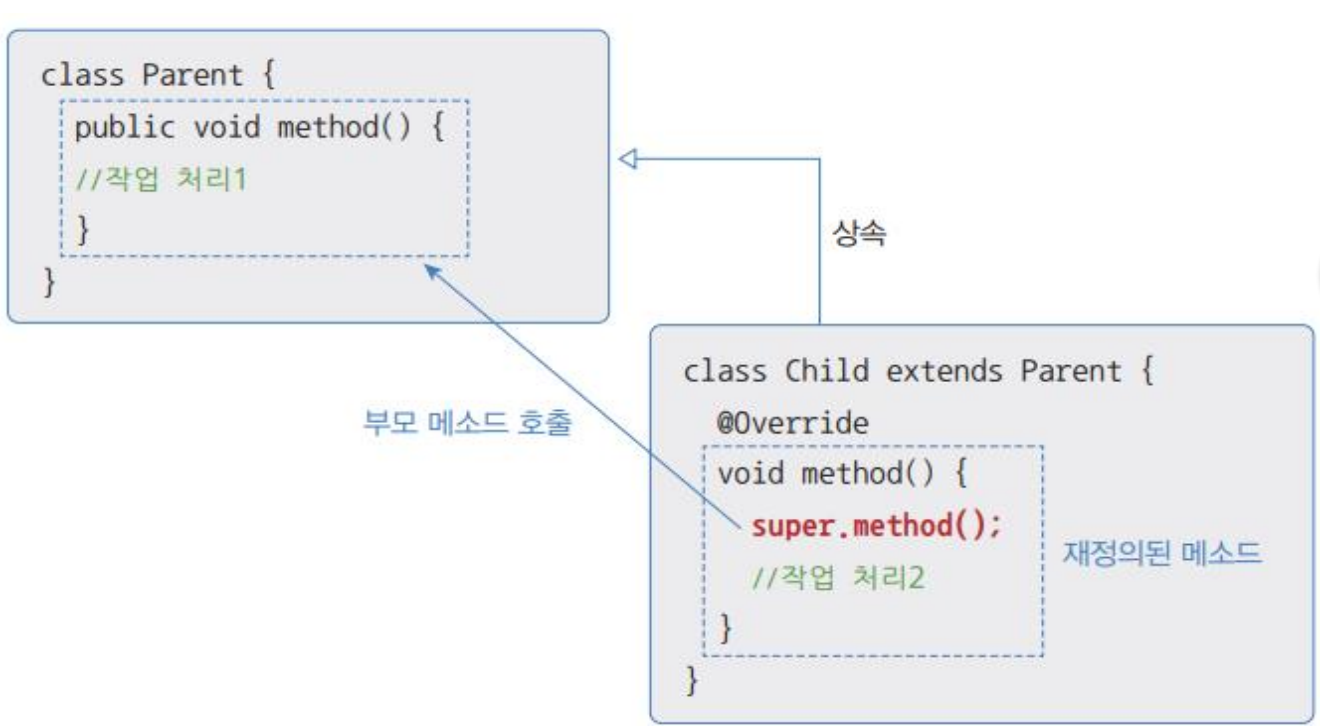
ComputerExample.java

```
package ch07.sec04.exam01;

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

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

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

```
Calculator 객체의 areaCircle() 실행
원 면적: 314.159

Computer 객체의 areaCircle() 실행
원 면적: 314.1592653589793
```

부모 메소드 호출

- 자식 메소드 내에서 super 키워드와 도트(.) 연산자를 사용하면 숨겨진 부모 메소드를 호출
- 부모 메소드를 재사용함으로써 자식 메소드의 중복 작업 내용을 없애는 효과



부모의 method()를 사용하고 싶은 경우 super.method2()를 호출함.

Airplane.java

```
package ch07.sec04.exam02;

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

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

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

SupersonicAirplane.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("초음속 비행합니다.");
		} else {
			//Airplane 객체의 fly() 메소드 호출
			super.fly();
		}
	}
}
```

ComputerExample.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();
	}
}
```

```
이륙합니다.
일반 비행합니다.
초음속 비행합니다.
일반 비행합니다.
착륙합니다.
```

final 클래스

- final 클래스는 부모 클래스가 될 수 없어 자식 클래스를 만들 수 없음

```
public final class 클래스 { … }
```

- final class : 상속 불가능한 클래스
- final method : 재정의(오버라이드) 불가능 메서드

final 메소드

- 메소드를 선언할 때 final 키워드를 붙이면 오버라이딩할 수 없음
- 부모 클래스를 상속해서 자식 클래스를 선언할 때,
  부모 클래스에 선언된 final 메소드는 자식 클래스에서 재정의할 수 없음

```
public final 리턴타입 메소드( 매개변수, … ) { … }
```

Member.java

```
package ch07.sec05.exam01;

public final class Member {
}
```

VeryImportantPerson.java

```
package ch07.sec05.exam01;

public class VeryImportantPerson extends Member {                    // 컴파일 에러 상속 불가
}
```

Car.java

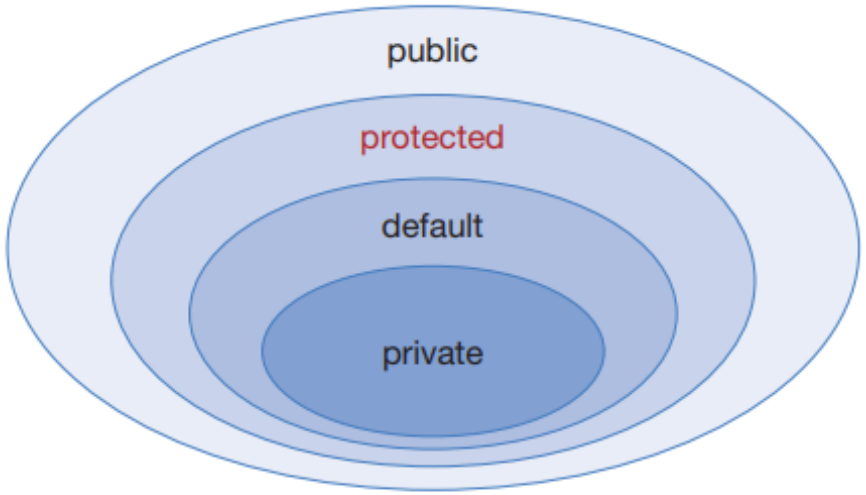
```
package ch07.sec05.exam02;

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;
	}
}
```

Member.java

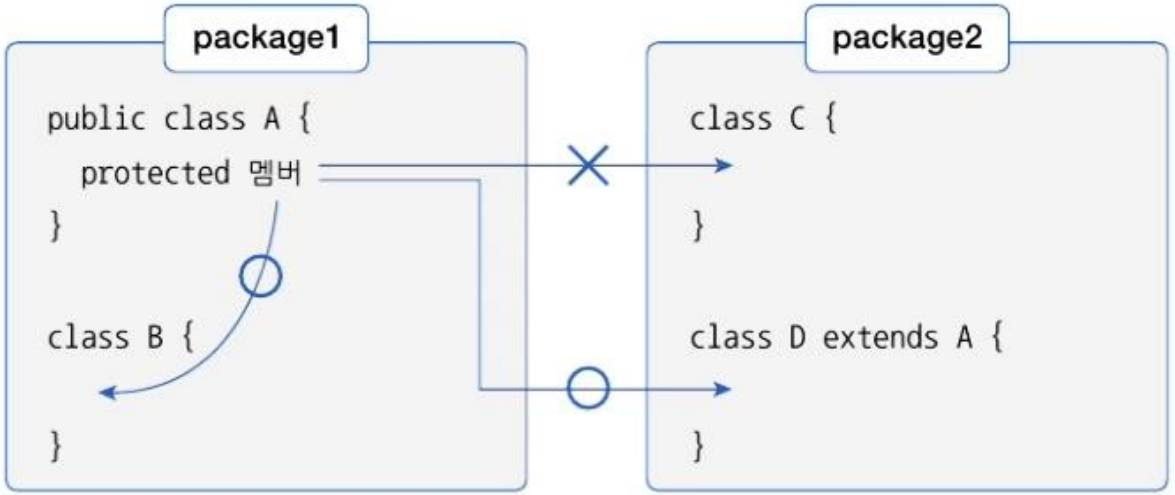
```
package ch07.sec05.exam02;

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

	// 컴파일 에러, final 메서드는 오버라이딩을 할 수 없음
	@Override
	public void stop() {
		System.out.println("스포츠카를 멈춤");
		speed = 0;
	}

}
```

protected 접근 제한자

- protected는 상속과 관련이 있고, public과 default의 중간쯤에 해당하는 접근 제한
- protected는 같은 패키지에서는 default처럼 접근이 가능하나,
  다른 패키지에서는 자식 클래스만 접근을 허용



접근 제한이 강화



NOTE ▸ default는 접근 제한자가 아니라 접근 제한자가 붙지 않은 상태를 말한다.

접근 제한자	제한 대상	제한 범위
protected	필드, 생성자, 메소드	같은 패키지이거나, 자식 객체만 사용 가능

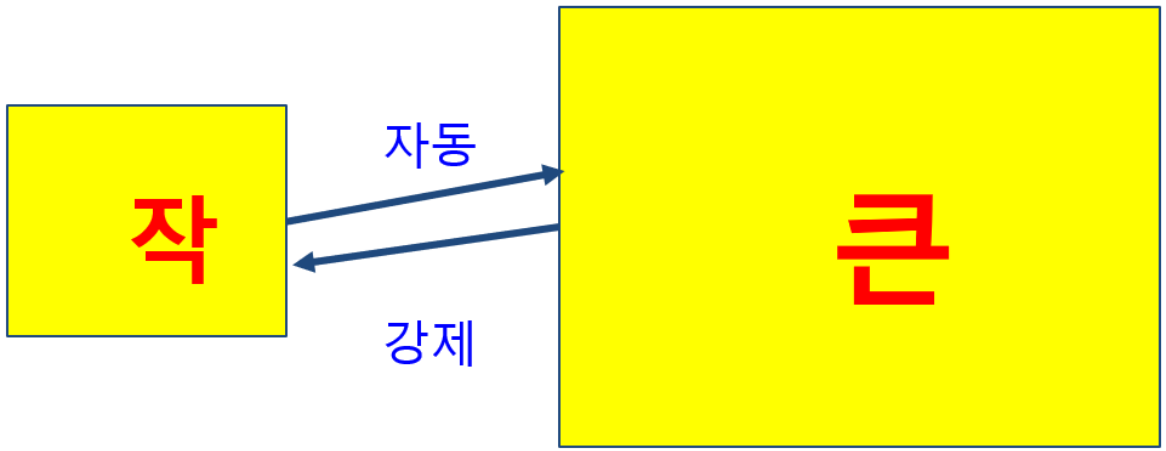
클래스 형변환(다형성)

변수에 들어있는 주소를 다른 변수에 넣을 때 타입을 변환해서 할당

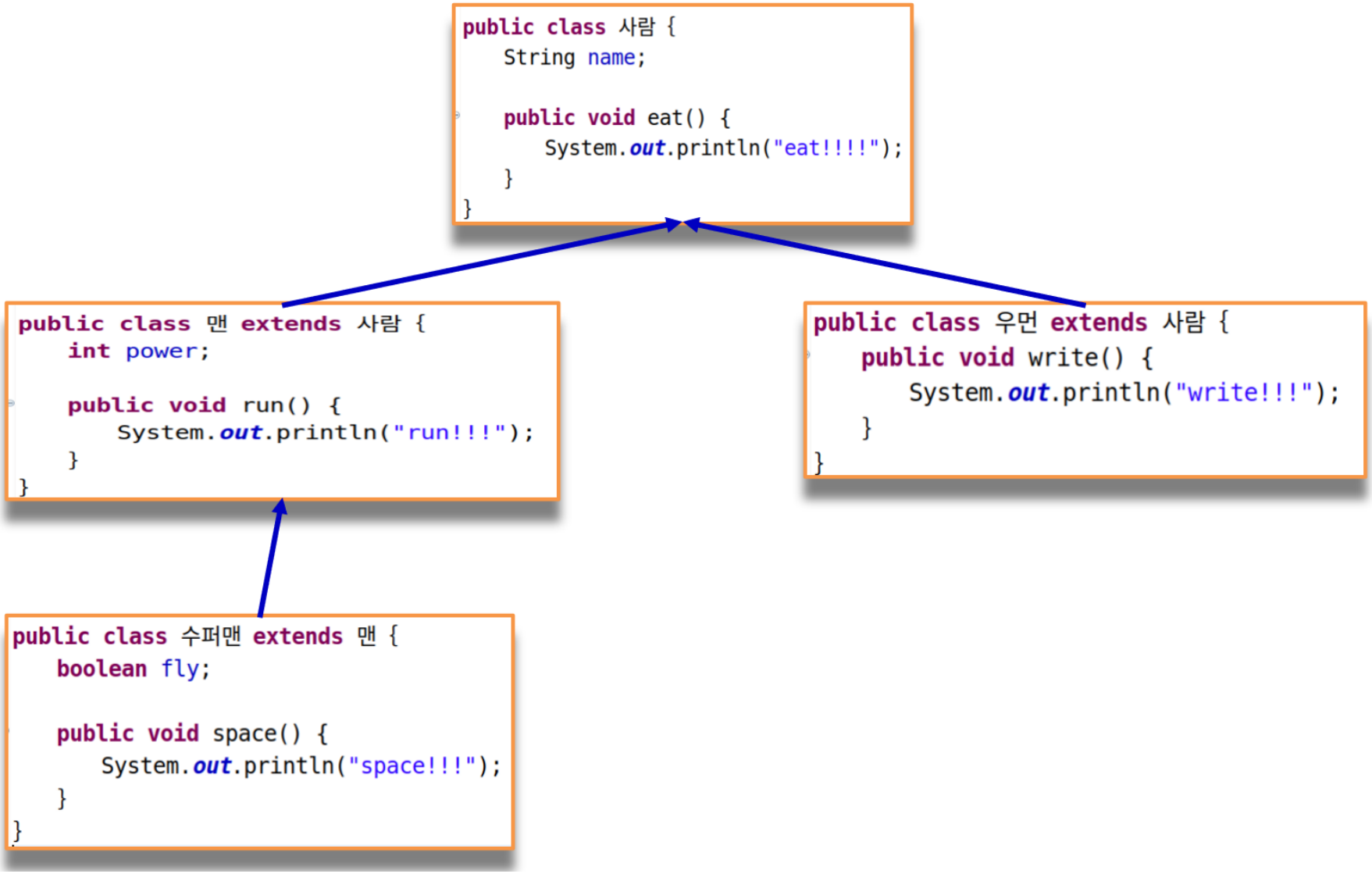
```java
public class CastingPrimitive {

    public static void main(String[] args) {
        // 기본형 형변환(정수, 실수, 문자, 논리)
        // 형변환(데이터 type변환)
        byte b1 = 100; //-128-127
        int i1 = 200;

        i1 = b1; //가능 int(큰)<----byte(작)
        //자동으로 byte에 들어있던 100데이터가 int로 변환되어 저장됨.
        //자동 데이터 변환(자동형변환)

        b1 = (byte)i1; //불가능 int(큰) ----> byte(작)
        //강제로 int에 들어있던 100데이터가 byte로 변환되어 저장됨.
        //강제 데이터 변환(강제형변환)

        int i2 = 3000;
        byte b2 = (byte)i2;
        System.out.println(b2);
        //강제형변환이 가능한 경우는 범위의 값만 가능!
```

기본형 형변환

- 크기 기준으로(**큰지 작은지만 판단**하면 개념은 명확해짐)
  - **자동형변환**
    작은 곳에 있던 데이터가 큰 곳으로 복사되는 경우, 자동으로 형변환되어 저장
  - **강제형변환**
    큰 곳에 있던 데이터가 작은 곳으로 복사되는 경우, 강제로 형변환하여 저장



	크기 기준	큰	작
기본형	변수의 크기	큰 크기의 변수	작은 크기의 변수
참조형	**상속관계만!** 개념의 크기	부모클래스	자식클래스


public class 사람 {
String name;
public void eat() {
System.out.println("eat!!!!");
}
}
public class 맨 extends 사람 {
int power;
public void run() {
System.out.println("run!!!");
}
}
public class 우먼 extends 사람 {
public void write() {
System.out.println("write!!!");
}
}
public class 수퍼맨 extends 맨 {
boolean fly;
public void space() {
System.out.println("space!!!");
}
}

```java
public class 참조형형변환1 {

    public static void main(String[] args) {
        사람 p = new 사람();
        수퍼맨 s = new 수퍼맨();
        맨 m = new 맨();
        우먼 w = new 우먼();

        //참조형은 상속관계에 있을 때만 된다.
        //부모자식간의 관계만 형변환이 가능해서 대입할 수 있다.
        p = s; //큰 <-- 작(자동 형변환)
        p = w; //큰 <-- 작(자동 형변환, up casting)
        //m = w; //형제간의 관계는 형변환하여 대입할 수 없다.
        m = s; //큰 <-- 작(자동 형변환)
        m = (맨)p; //작 <-- 큰(강제 형변환) => (형변환하고 싶은 타입)변수
```

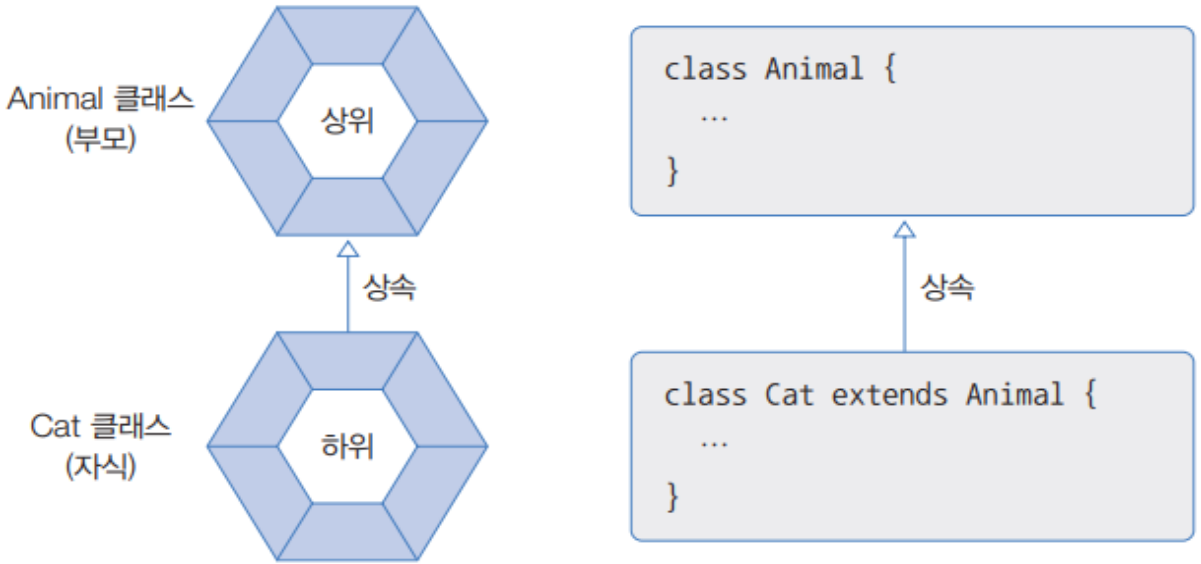
자동 타입 변환

- 자동적으로 타입 변환이 일어나는 것

```
부모타입 변수 = 자식타입객체;
```
(자동 타입 변환)

참조형 형변환
클래스는 상속관계에
있을 때만 가능
→ 상속만 대소관계가
생김.

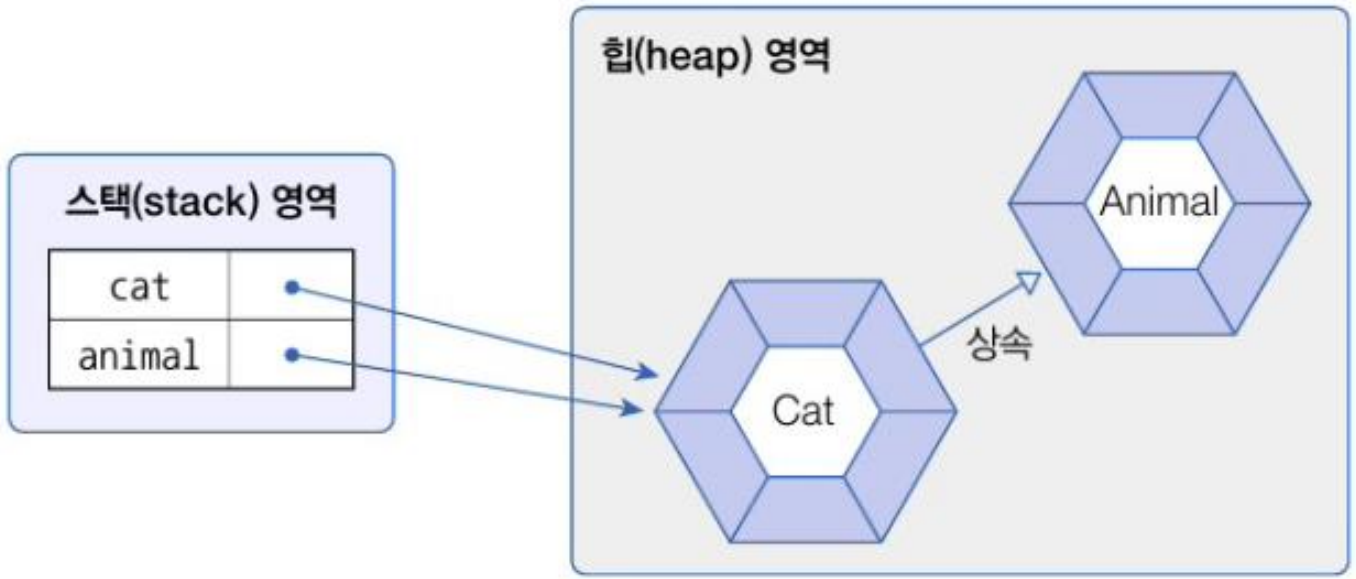
- 자식은 부모의 특징과 기능을 상속받기 때문에 부모와 동일하게 취급



자동 타입 변환

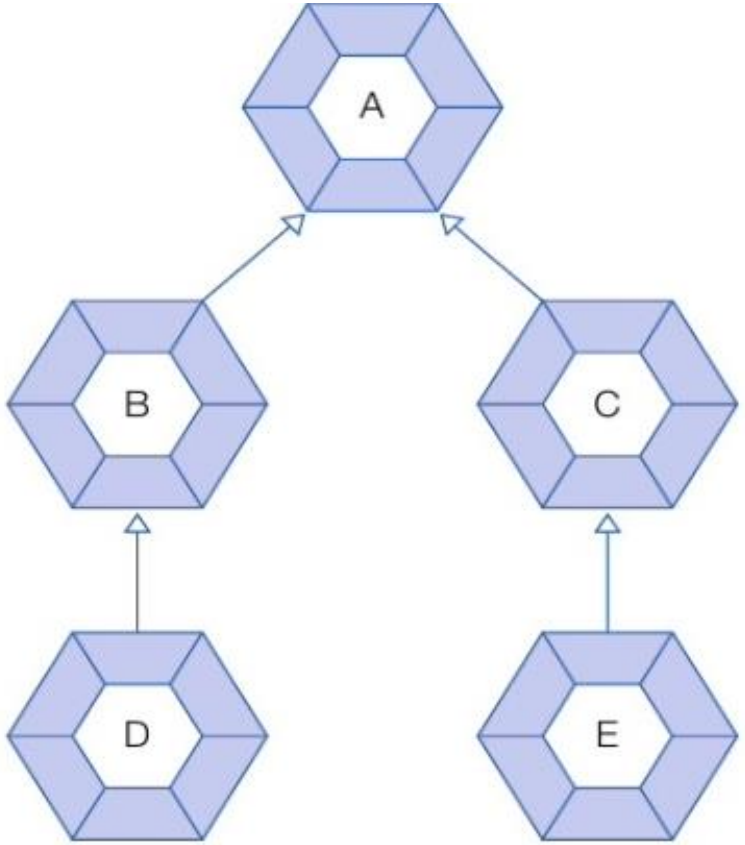
```
Cat cat = new Cat();
Animal animal = cat;
```

Animal animal = new Cat();도 가능




```
cat == animal    //true
```

자동 타입 변환

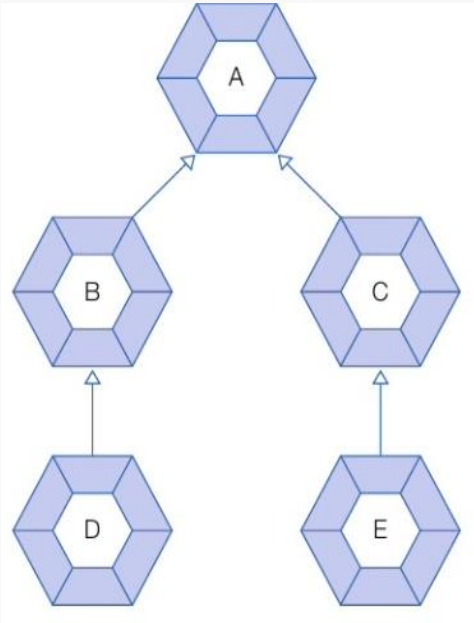



```
B b = new B();
C c = new C();
D d = new D();
E e = new E();
```

```
A a1 = b;   (가능)
A a2 = c;   (가능)
A a3 = d;   (가능)
A a4 = e;   (가능)

B b1 = d;   (가능)
C c1 = e;   (가능)

B b3 = e;   (불가능)
C c2 = d;   (불가능)
```

PromotionExample.java

```
package ch07.sec07.exam01;

class A {
}

class B extends A {
}

class C extends A {
}

class D extends B {
}

class E extends C {
}
```

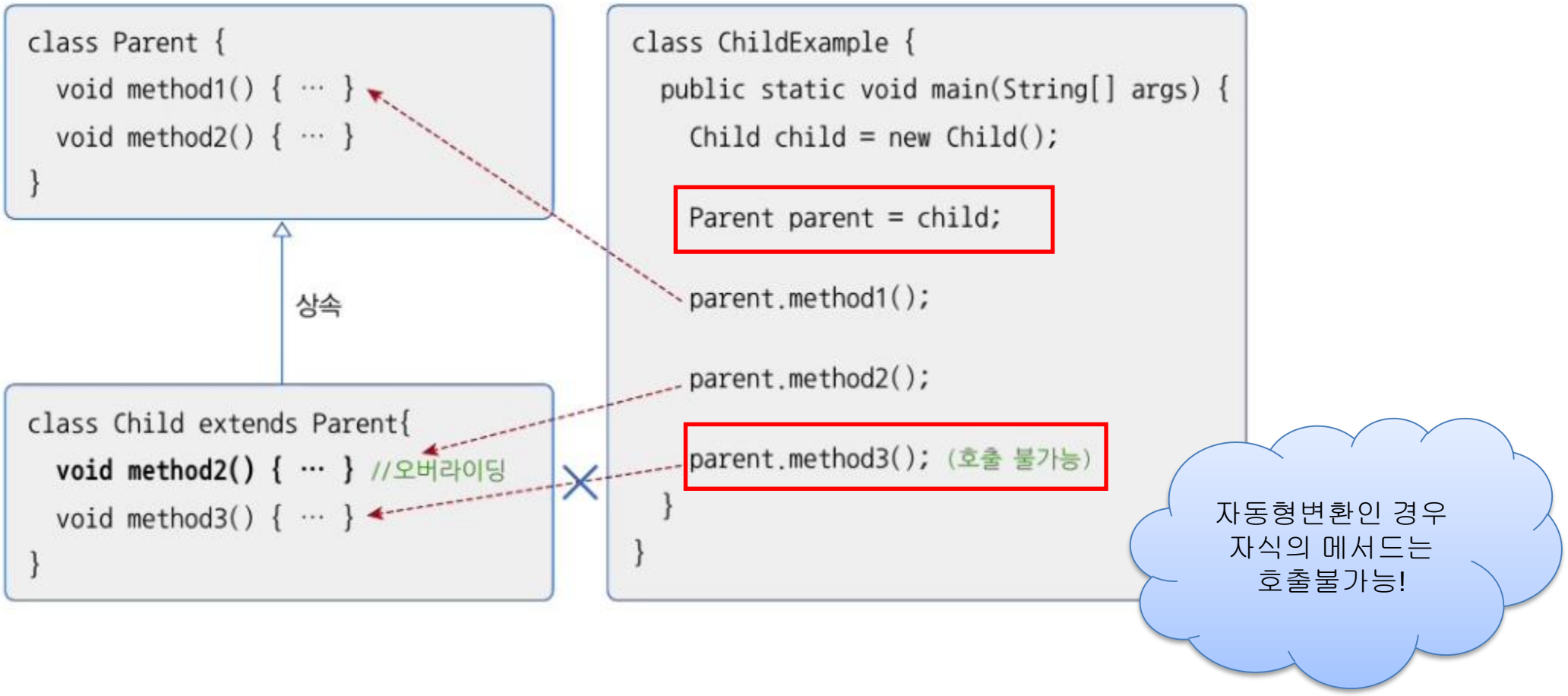



```
public class PromotionExample {
        public static void main(String[] args) {
                B b = new B();
                C c = new C();
                D d = new D();
                E e = new E();

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

                B b1 = d;
                C c1 = e;

                // B b3 = e;
                // C c2 = d;
        }
}
```

자동 타입 변환(상속 관계에 있음)

컴파일 에러(상속 관계에 있지 않음)

자동 타입 변환(자동형변환)



Parent.java

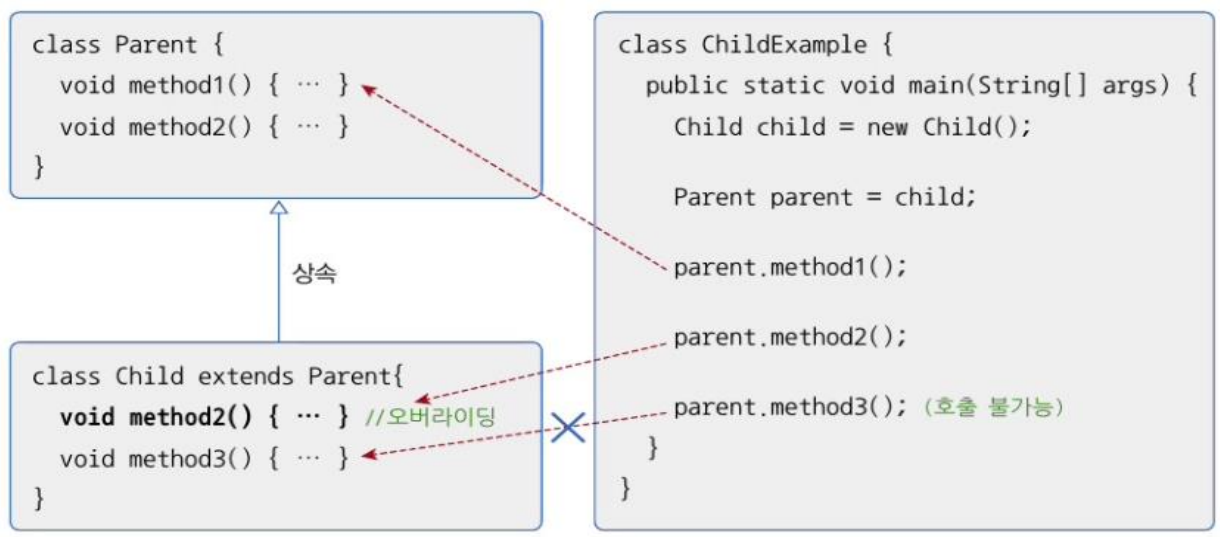
```
package ch07.sec07.exam02;

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

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

Child.java

```
package ch07.sec07.exam02;

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

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

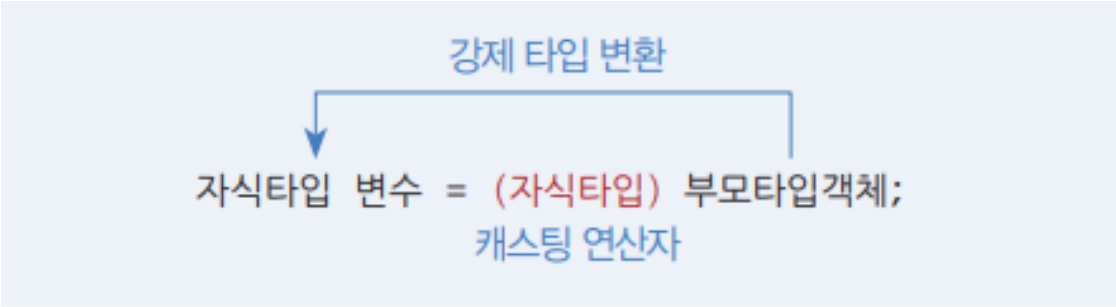
Parent.java

```
package ch07.sec07.exam02;

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

                //자동 타입 변환
                Parent parent = child;

                //메소드 호출
                parent.method1();
                parent.method2();
                //parent.method3(); (호출 불가능)
        }
}
```



강제 타입 변환

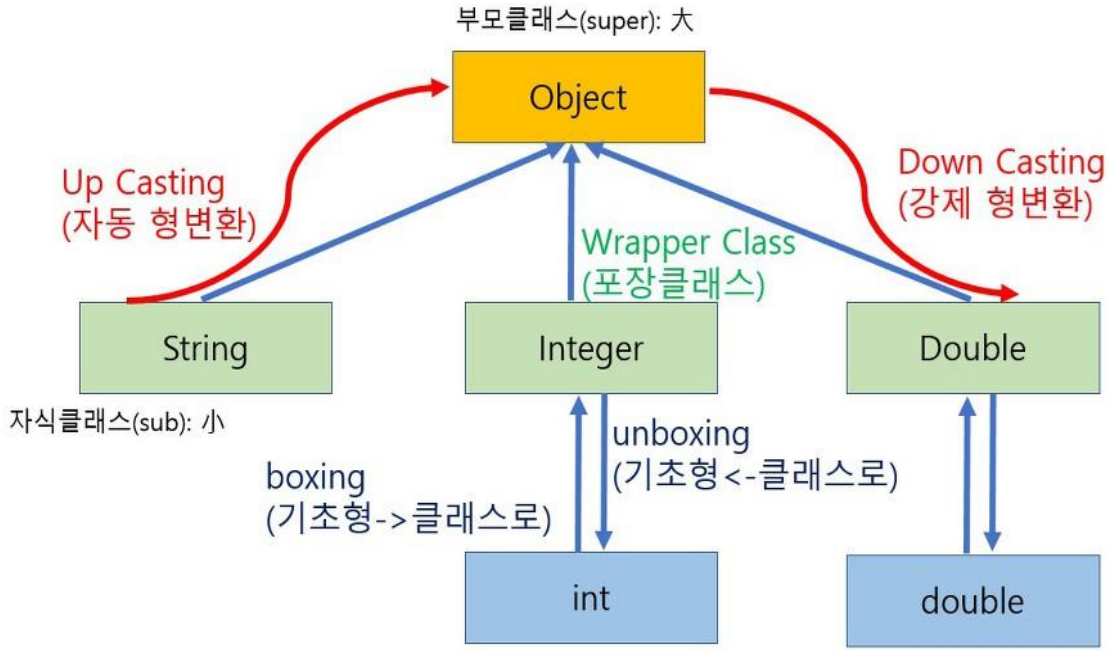
- 부모 타입은 자식 타입으로 자동 변환되지 않음. 대신 캐스팅 연산자로 강제 타입 변환 가능




```
Parent parent = new Child();     //자동 타입 변환
Child child = (Child) parent;    //강제 타입 변환
```

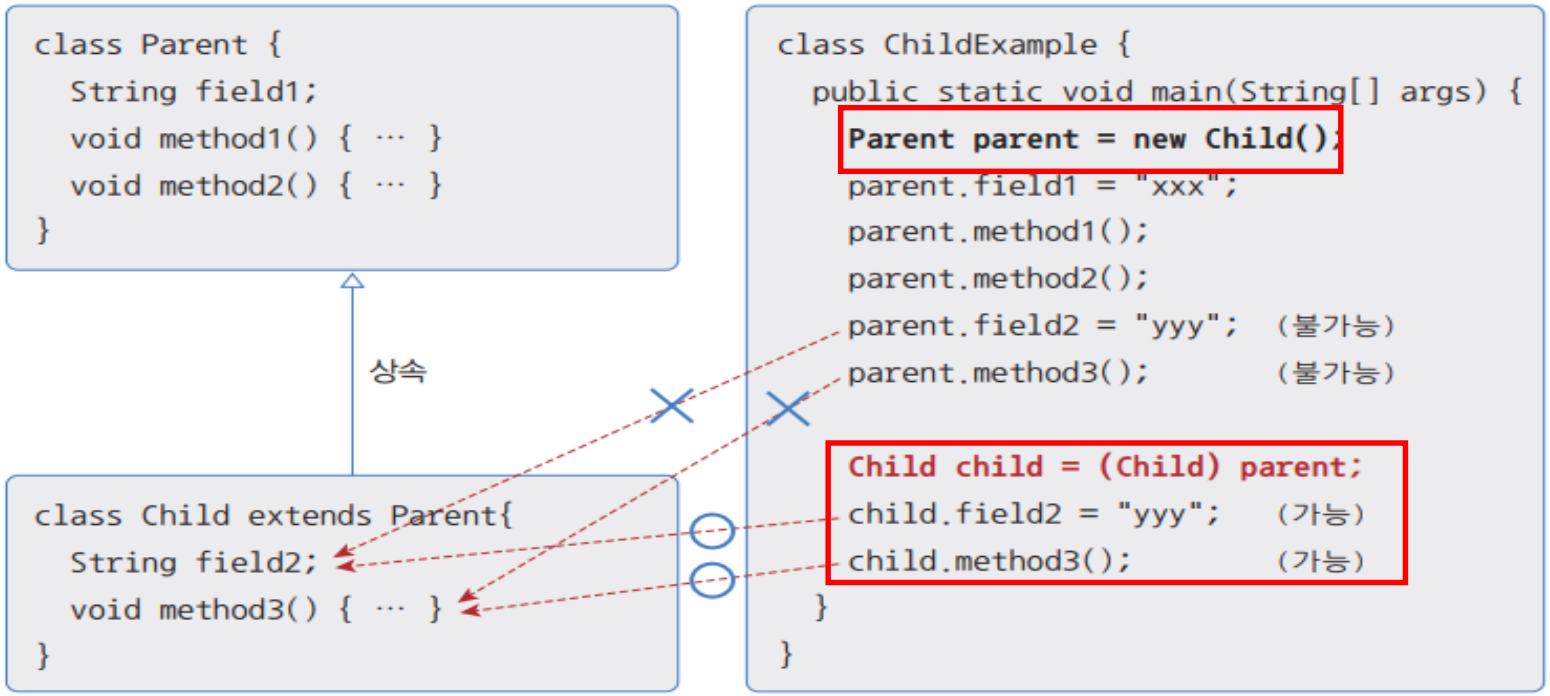
CastingTest.java

```
package ch07.sec07.exam02;

public class CastringTest {
        public static void main(String[] s0){
                List list = new ArrayList();
                list.add(new String("test")); //자동형변환
                //Object <-- String
                list.add(new Random());
                list.add(new Date());

                Object o = list.get(0);
                //System.out.println(o.charAt(0));  불가능
                //Object에 charAt()이 없음.
                String s = (String)list.get(0); //강제형변환
                System.out.println(s.charAt(0)); //가능
        }
}
```



강제 타입 변환

- 자식 객체가 부모 타입으로 자동 변환하면 부모 타입에 선언된 필드와 메소드만 사용 가능



Parent.java

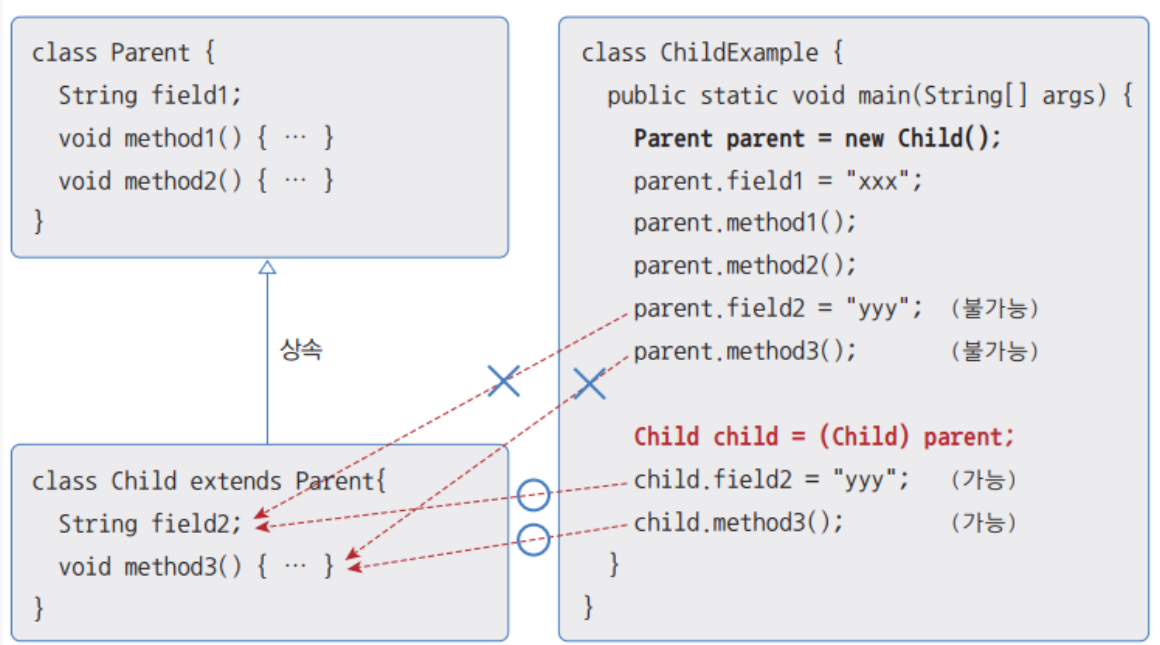
```
package ch07.sec07.exam03;

public class Parent {
	//필드 선언
	public String field1;

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

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

Child.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()");
	}
}
```

Parent.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)
        }
}
```

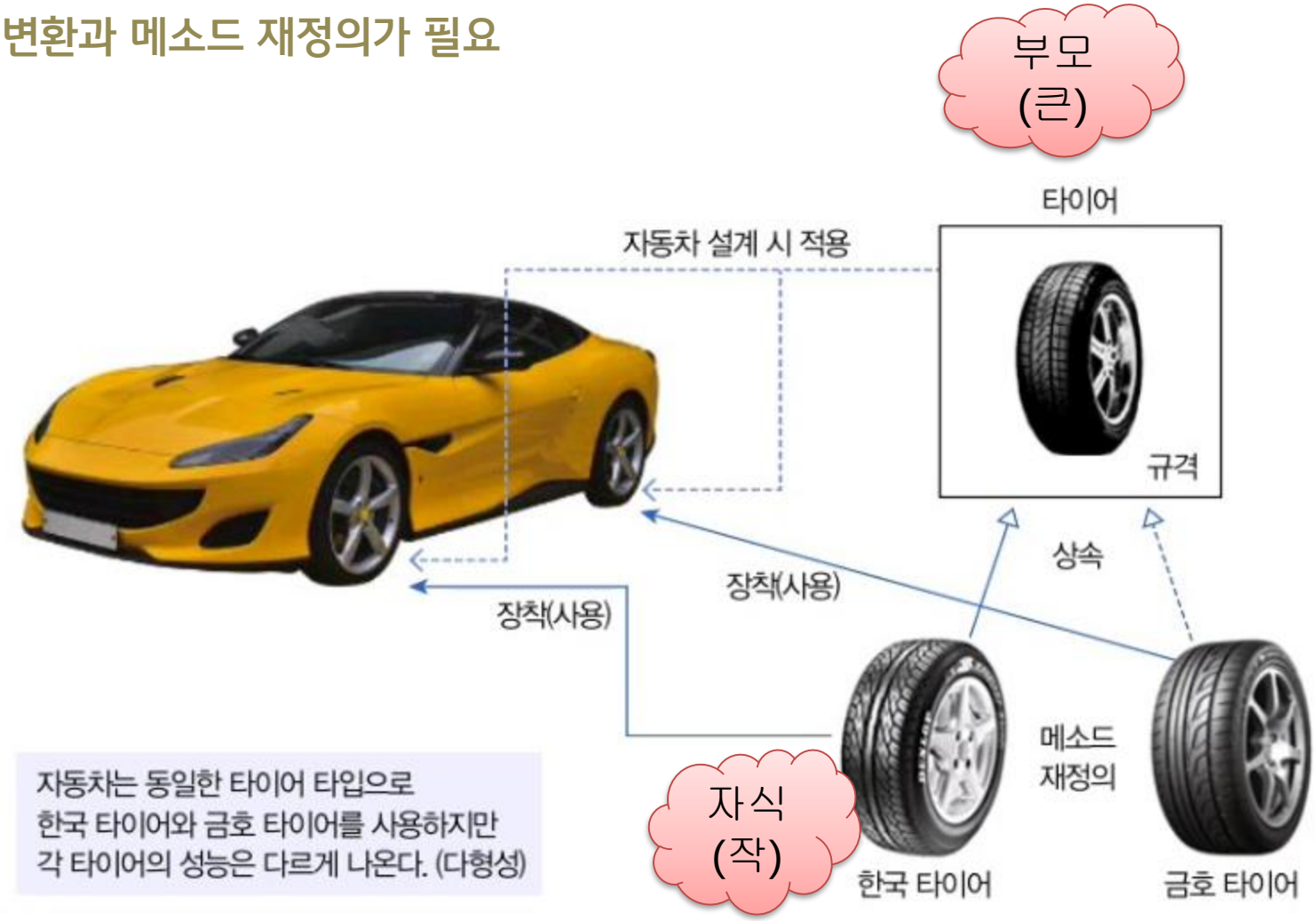


다형성

- 사용 방법은 동일하지만 실행 결과가 다양하게 나오는 성질
- 다형성을 구현하기 위해서는 자동 타입 변환과 메소드 재정의가 필요
- 필드 다형성, 매개변수 다형성

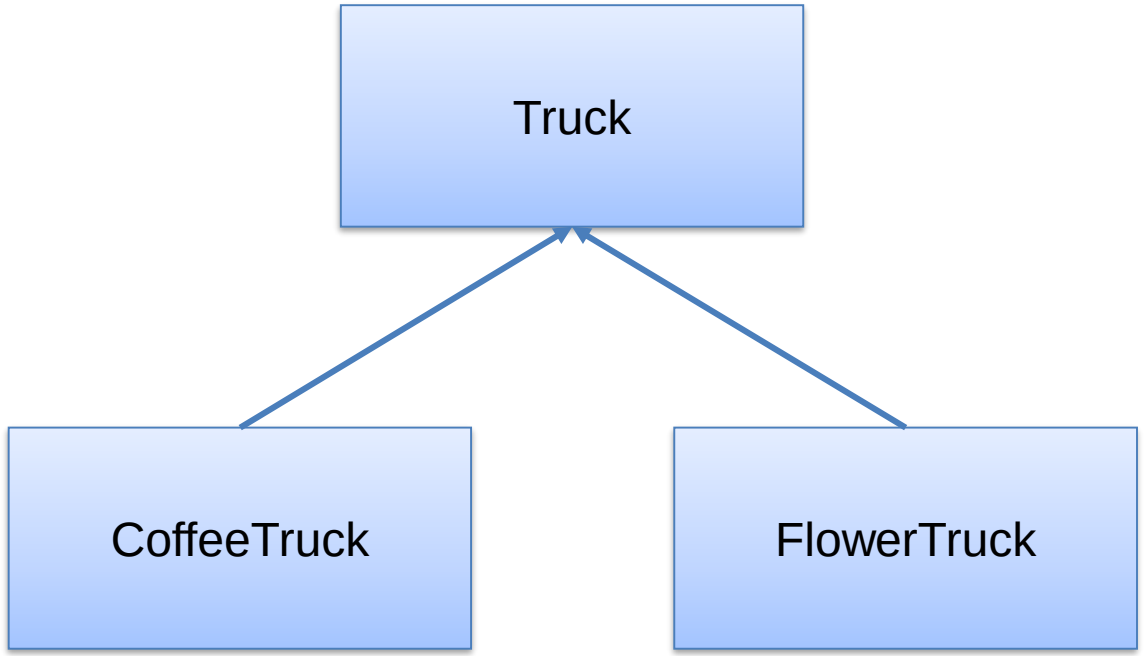
하나의 이름으로
다양한 형태를
사용할 수 있는 특징

→ 클래스 만들 때 “타이어 ” 로 지정해두면 “한국타이어”,”금호타이어”를 모두 조립해서 넣을 수 있음.



필드 다형성, 매개변수 다형성

```
public class Car {
        Truck truck; //필드 다형성

        Car (Truck truck){ //매개변수 다형성
                this.truck = truck;
        }
}
----------------------------------------
Car car1 = new Car(new CoffeeTruck() );
car1.sell(); //꽃? 커피?

Car car2 = new Car(new FlowerTruck() );
car2.sell(); //꽃? 커피?
```



필드 다형성

- 필드 타입은 동일하지만, 대입되는 객체가 달라져서 실행 결과가 다양하게 나올 수 있는 것

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

  //메소드 선언
  public void run() {
    tire.roll();
  }
}
```

tire 필드에 대입된 객체의 roll() 메소드 호출

```
//Car 객체 생성
Car myCar = new Car();
//HankookTire 장착
myCar.tire = new HankookTire();
//KumhoTire 장착
myCar.tire = new KumhoTire();
```

- 앞으로 더 개선된 타이어도 Car클래스를 수정하지 않고도 모두 넣을 수 있는 좋은 코드가 만들어짐.
- 조립시 필요한 부품 조립해서 사용 가능

```
myCar.run();
```

대입된(장착된) 타이어의 roll() 메소드 호출

Tire.java

```
package ch07.sec08.exam01;

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

HankookTire.java

```
package ch07.sec08.exam01;

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

KumhoTire.java

```
package ch07.sec08.exam01;

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

Car.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();
	}
}
```

CarExample.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();
	}
}
```

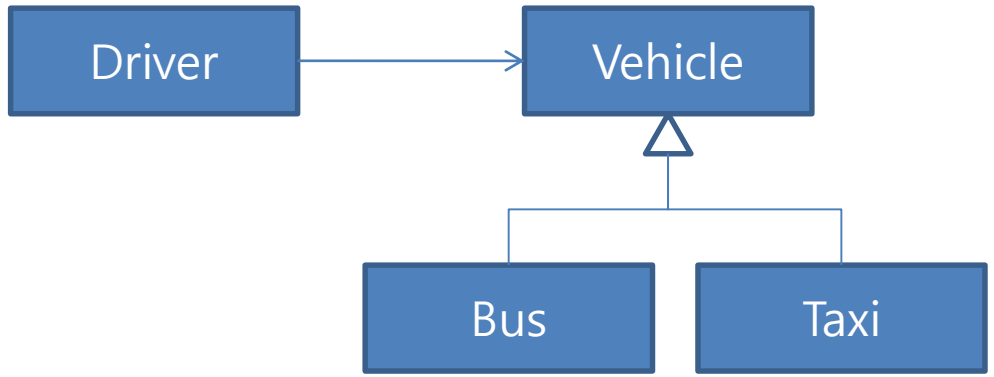
```
회전합니다.
한국 타이어가 회전합니다.
금호 타이어가 회전합니다.
```

매개변수 다형성

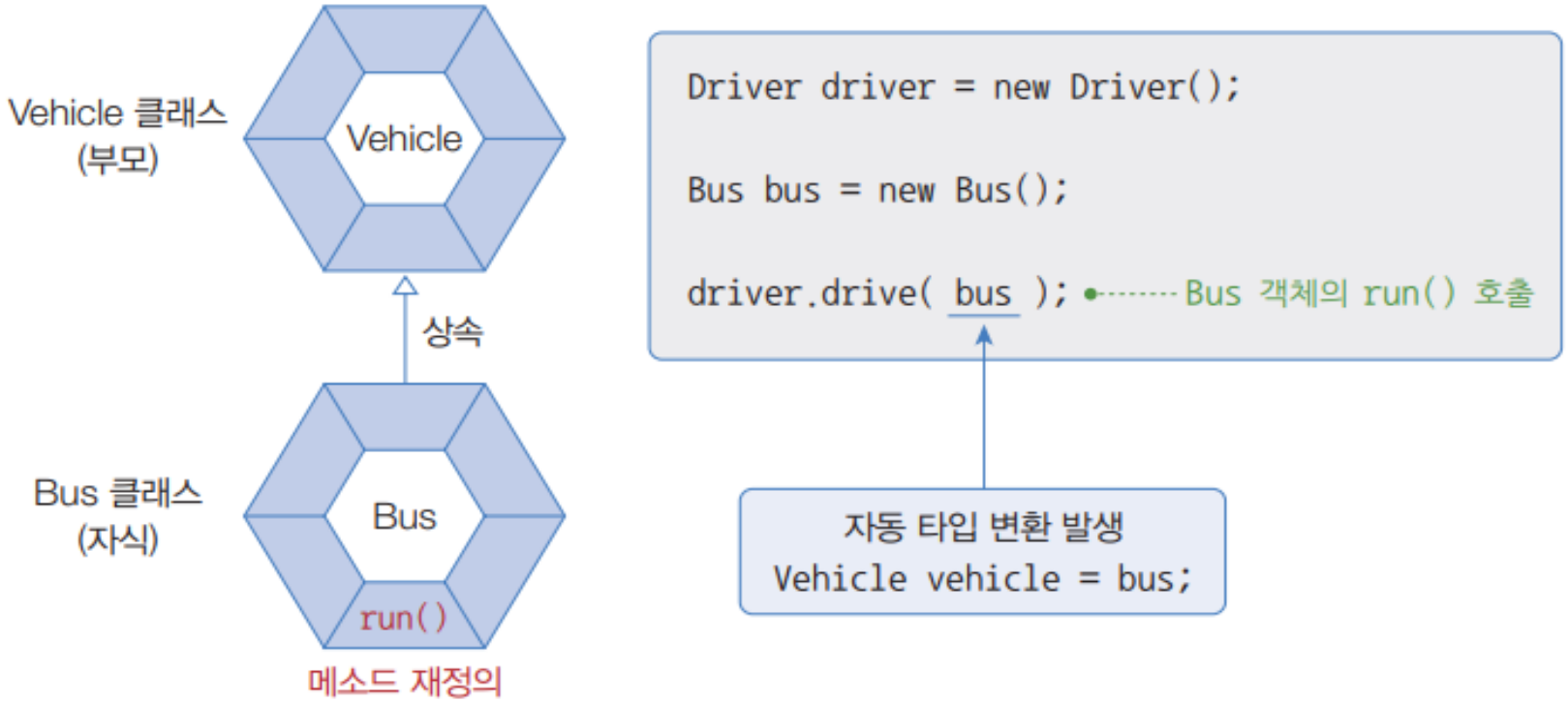
- 메소드가 클래스 타입의 매개변수를 가지고 있을 경우,
  - 호출할 때 동일한 타입의 자식 객체를 제공할 수 있음
- 어떤 자식 객체가 제공되느냐에 따라서 메소드의 실행 결과가 달라짐(전략 strategy 패턴)

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

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



매개변수 다형성




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

자식 객체

자식 객체가 재정의한 run() 메소드 호출

Vehicle.java

```
package ch07.sec08.exam02;

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

Bus.java

```
package ch07.sec08.exam02;

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

Taxi.java

```
package ch07.sec08.exam02;

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

Driver.java

```
package ch07.sec08.exam02;

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

DriverExample.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);                    // driver.drive(new Bus()); 와 동일

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

```
버스가 달립니다.
택시가 달립니다.
```

상속, extends

- 클래스를 만들 때 기존의 것을 재사용해서 만드는 것
- 재사용
- 하나의 클래스만 재사용할 수 있다. (단일 상속)

상속시 생성자

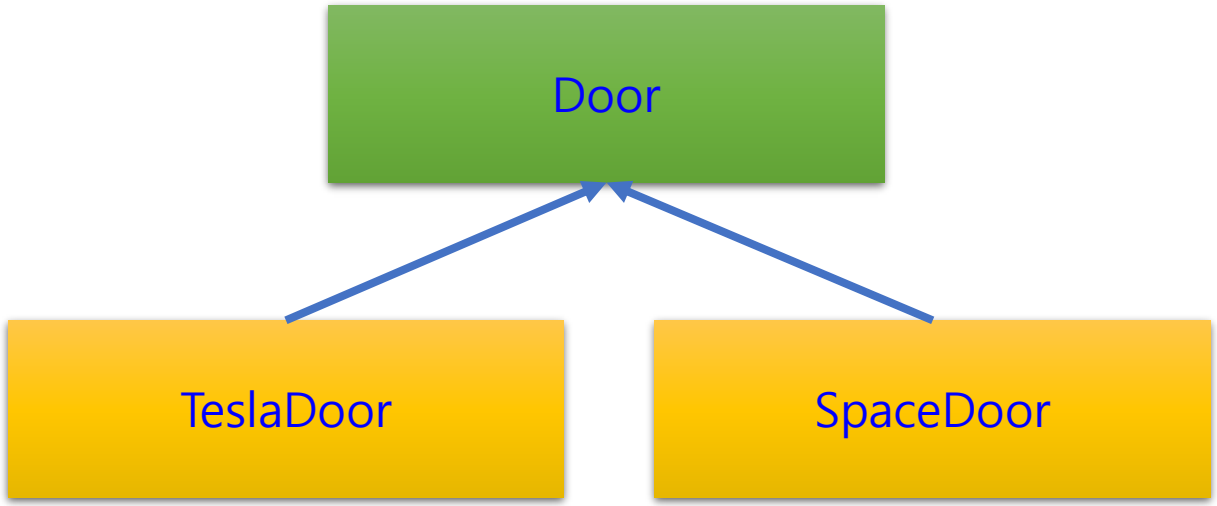
- 부모(기존 클래스, 수퍼), 자식(재사용한 클래스, 서브)

오버라이드(재정의)

- 자식 클래스에서 메서드를 재정의

클래스 형변환(다형성)

- 형변환 : 큰 변수 ←→ 작은 변수 넣는 것. 각 변수의 타입으로 변환되어 들어감.
- 기본형 형변환 : 메서드를 쓰지 않고 형변환하는 경우는 기본형끼리만 가능
- 큰 ← 작 (자동형변환) : byte x = 100; int y = x;
- 작 ← 큰(강제형변환) : int x = 100; byte y = (byte) x;
- 참조형 형변환 : 상속관계에 있는 클래스간 만 가능(개념 기준으로 크기 나눔)
- 큰 ← 작(자동형변환) : Car car = new Car();, Truck truck = new Truck();, car = truck;
- 작 ← 큰(강제형변환) : truck = (Truck) Car;


Door
TeslaDoor
SpaceDoor