9. 상속

#1.인강/0.자바/2.자바-기본

상속 - 시작

상속 관계가 왜 필요한지 이해하기 위해 다음 예제 코드를 만들어서 실행해보자.

예제 코드

패키지 위치에 주의하자

```
package extends1.ex1;

public class ElectricCar {

    public void move() {
        System.out.println("차를 이동합니다.");
    }

    public void charge() {
        System.out.println("충전합니다.");
    }
}
```

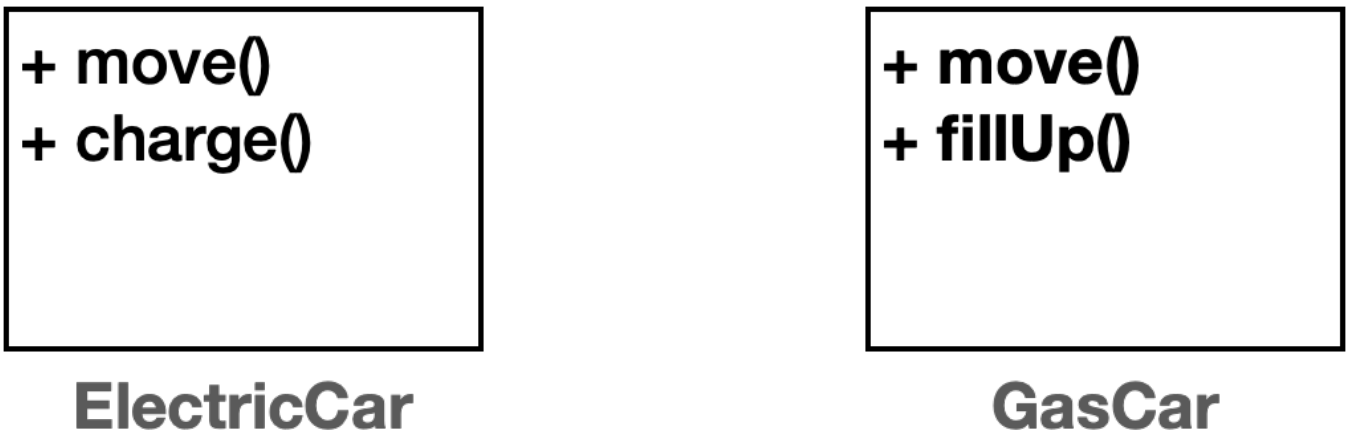
```
package extends1.ex1;

public class GasCar {
```

```
    public void move() {
        System.out.println("차를 이동합니다.");
    }

    public void fillUp() {
        System.out.println("기름을 주유합니다.");
    }
}
```

```
package extends1.ex1;

public class CarMain {

    public static void main(String[] args) {
        ElectricCar electricCar = new ElectricCar();
        electricCar.move();
        electricCar.charge();

        GasCar gasCar = new GasCar();
        gasCar.move();
        gasCar.fillUp();
    }
}
```

실행 결과

```
차를 이동합니다.
충전합니다.
차를 이동합니다.
기름을 주유합니다.
```



전기차(`ElectricCar`)와 가솔린차(`GasCar`)를 만들었다. 전기차는 이동(`move()`), 충전(`charge()`) 기능이 있고, 가솔린차는 이동(`move()`), 주유(`fillUp()`) 기능이 있다.

전기차와 가솔린차는 자동차(`Car`)의 좀 더 구체적인 개념이다. 반대로 자동차(`Car`)는 전기차와 가솔린차를 포함하는 추상적인 개념이다. 그래서인지 잘 보면 둘의 공통 기능이 보인다. 바로 이동(`move()`)이다.
전기차든 가솔린차든 주유하는 방식이 다른 것이지 이동하는 것은 똑같다. 이런 경우 상속 관계를 사용하는 것이 효과적이다.

상속 관계

상속은 객체 지향 프로그래밍의 핵심 요소 중 하나로, 기존 클래스의 필드와 메서드를 새로운 클래스에서 재사용하게 해준다. 이름 그대로 기존 클래스의 속성과 기능을 그대로 물려받는 것이다. 상속을 사용하려면 `extends` 키워드를 사용하면 된다. 그리고 `extends` **대상은 하나만 선택**할 수 있다.

용어 정리

- **부모 클래스 (슈퍼 클래스)**: 상속을 통해 자신의 필드와 메서드를 다른 클래스에 제공하는 클래스
- **자식 클래스 (서브 클래스)**: 부모 클래스로부터 필드와 메서드를 상속받는 클래스

주의!
지금부터 코드를 작성할 때 기존 코드를 유지하기 위해, 새로운 패키지에 기존 코드를 옮겨가면서 코드를 작성할 것이다.
클래스의 이름이 같기 때문에 패키지 명과 import 사용에 주의해야 한다.

상속 관계를 사용하도록 코드를 작성해보자.

기존 코드를 유지하기 위해 ex2 패키지를 새로 만들자

```
package extends1.ex2;

public class Car {
    public void move() {
        System.out.println("차를 이동합니다.");
    }
}
```

`Car`는 부모 클래스가 된다. 여기에는 자동차의 공통 기능인 `move()`가 포함되어 있다.

```
package extends1.ex2;

public class ElectricCar extends Car {

    public void charge() {
        System.out.println("충전합니다.");
    }
}
```

전기차는 `extends Car`를 사용해서 부모 클래스인 `Car`를 상속 받는다. 상속 덕분에 `ElectricCar`에서도 `move()`를 사용할 수 있다.

```
package extends1.ex2;

public class GasCar extends Car {

    public void fillUp() {
        System.out.println("기름을 주유합니다.");
    }
}
```

가솔린차도 전기차와 마찬가지로 `extends Car`를 사용해서 부모 클래스인 `Car`를 상속 받는다. 상속 덕분에 여기서도 `move()`를 사용할 수 있다.

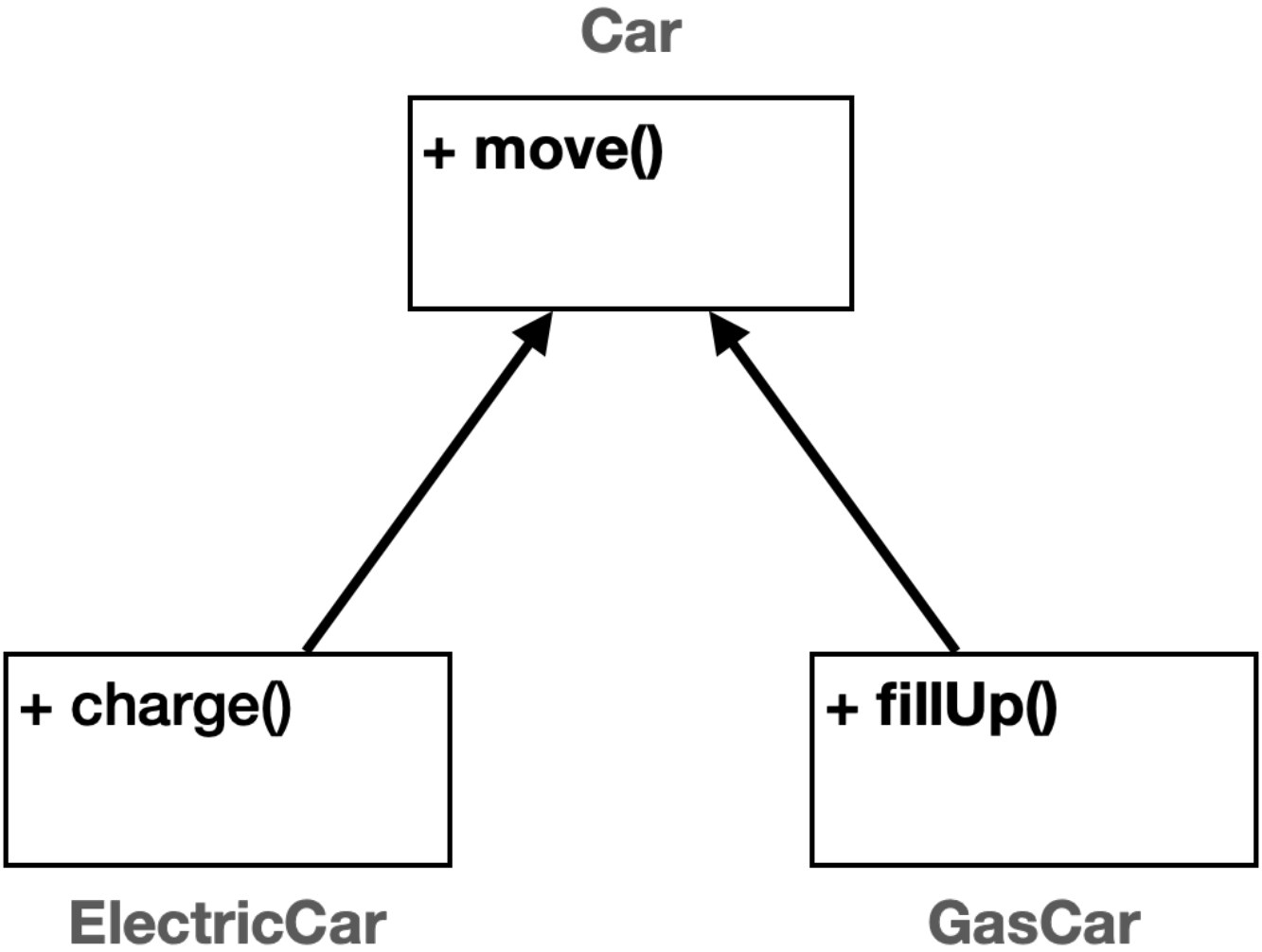
```
package extends1.ex2;

public class CarMain {

    public static void main(String[] args) {
        ElectricCar electricCar = new ElectricCar();
        electricCar.move();
        electricCar.charge();

        GasCar gasCar = new GasCar();
        gasCar.move();
        gasCar.fillUp();
    }
}
```

실행 결과

```
차를 이동합니다.
충전합니다.
차를 이동합니다.
기름을 주유합니다.
```

실행 결과는 기존 예제와 완전히 동일하다.

상속 구조도



전기차와 가솔린차가 `Car`를 상속 받은 덕분에 `electricCar.move()`, `gasCar.move()`를 사용할 수 있다.

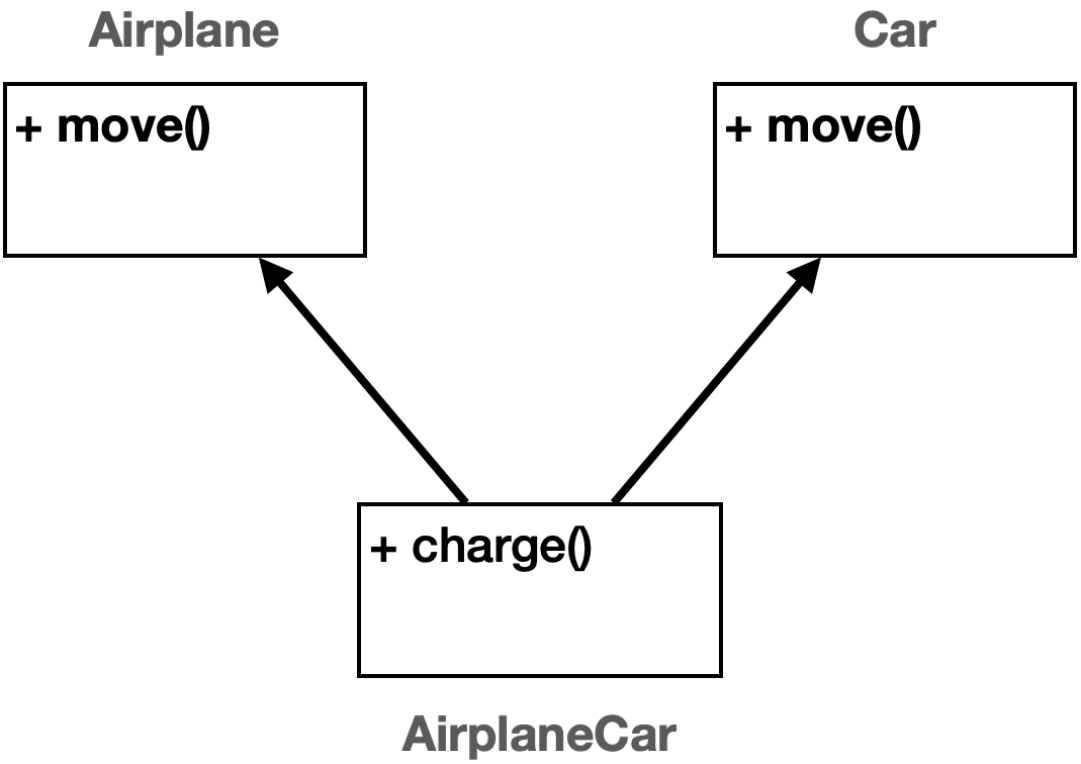
참고로 당연한 이야기지만 상속은 부모의 기능을 자식이 물려 받는 것이다. 따라서 자식이 부모의 기능을 물려 받아서 사용할 수 있다. 반대로 부모 클래스는 자식 클래스에 접근할 수 없다. 자식 클래스는 부모 클래스의 기능을 물려 받기 때문에 접근할 수 있지만, 그 반대는 아니다. 부모 코드를 보자! 자식에 대한 정보가 하나도 없다. 반면에 자식 코드는 `extends Car`를 통해서 부모를 알고 있다.

단일 상속

참고로 자바는 다중 상속을 지원하지 않는다. 그래서 `extend` 대상은 하나만 선택할 수 있다. 부모를 하나만 선택할 수

있다는 뜻이다. 물론 부모가 또 다른 부모를 하나 가지는 것은 괜찮다.

다중 상속 그림



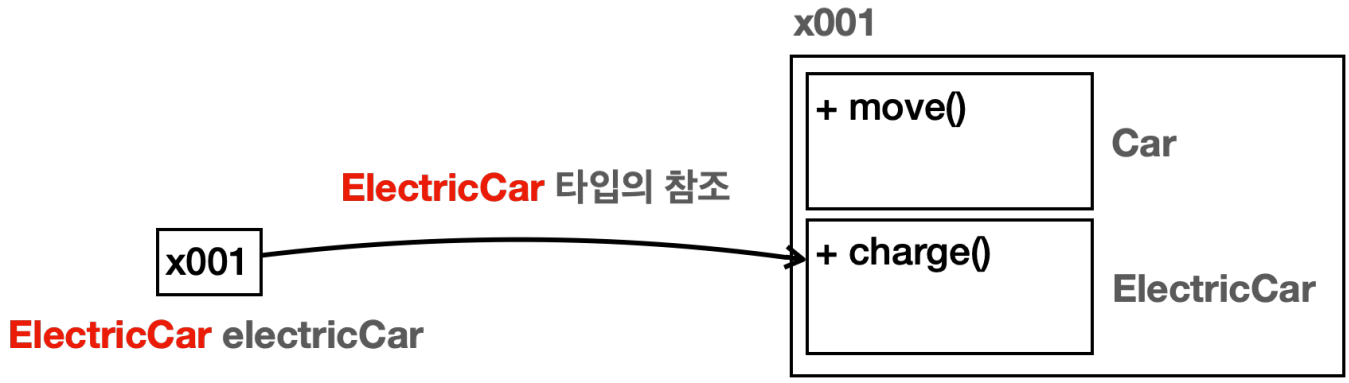
만약 비행기와 자동차를 상속 받아서 하늘을 나는 자동차를 만든다고 가정해보자. 만약 그림과 같이 다중 상속을 사용하게 되면 `AirplaneCar` 입장에서 `move()`를 호출할 때 어떤 부모의 `move()`를 사용해야 할지 애매한 문제가 발생한다. 이것을 다이아몬드 문제라 한다. 그리고 다중 상속을 사용하면 클래스 계층 구조가 매우 복잡해지 수 있다. 이런 문제점 때문에 자바는 클래스의 다중 상속을 허용하지 않는다. 대신에 이후에 설명한 인터페이스의 다중 구현을 허용해서 이러한 문제를 피한다.

상속과 메모리 구조

이 부분을 제대로 이해하는 것이 앞으로 정말 중요하다!

상속 관계를 객체로 생성할 때 메모리 구조를 확인해보자.

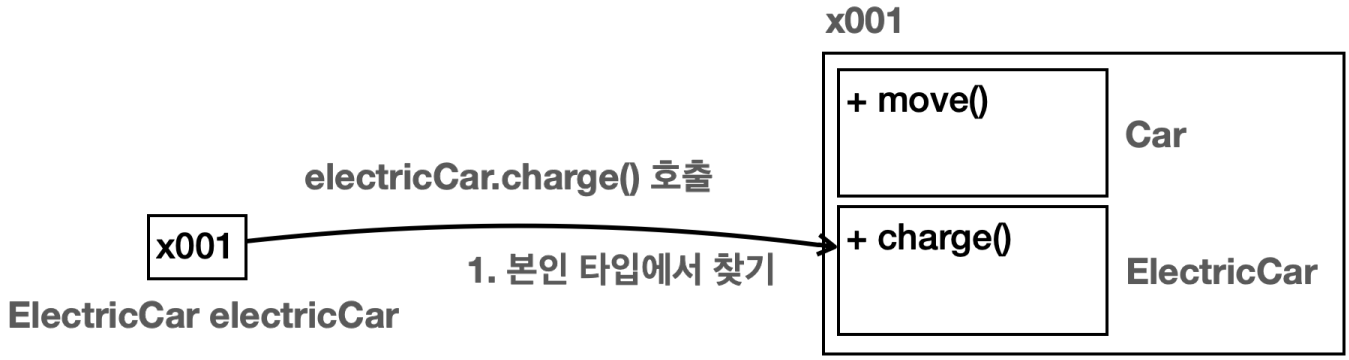
```
ElectricCar electricCar = new ElectricCar();
```



`new ElectricCar()`를 호출하면 `ElectricCar` 뿐만 아니라 상속 관계에 있는 `Car`까지 함께 포함해서 인스턴스를 생성한다. 참조값은 `x001`로 하나이지만 실제로 그 안에서는 `Car`, `ElectricCar`라는 두가지 클래스 정보가 공존하는 것이다.

상속이라고 해서 단순하게 부모의 필드와 메서드만 물려 받는게 아니다. 상속 관계를 사용하면 부모 클래스도 함께 포함해서 생성된다. 외부에서 볼때는 하나의 인스턴스를 생성하는 것 같지만 내부에서는 부모와 자식이 모두 생성되고 공간도 구분된다.

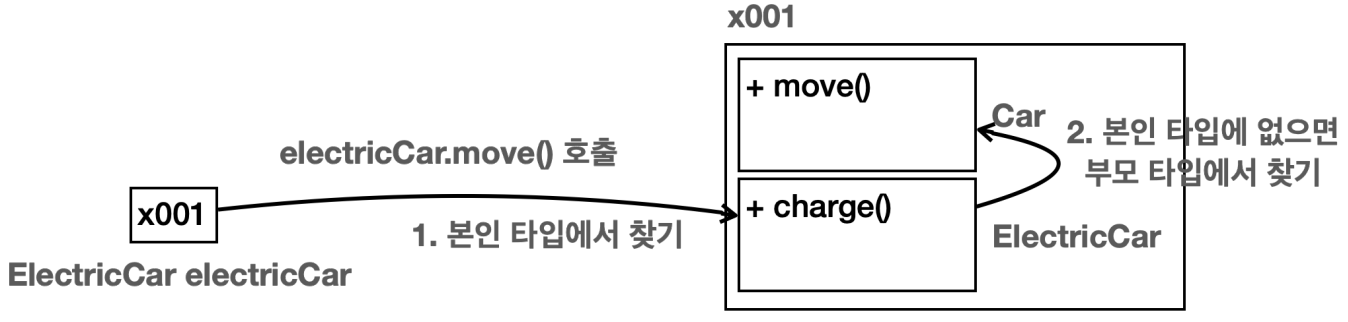
`electricCar.charge()` **호출**



`electricCar.charge()`를 호출하면 참조값을 확인해서 `x001.charge()`를 호출한다. 따라서 `x001`을 찾아서 `charge()`를 호출하면 되는 것이다. 그런데 상속 관계의 경우에는 내부에 부모와 자식이 모두 존재한다. 이때 부모인 `Car`를 통해서 `charge()`를 찾을지 아니면 `ElectricCar`를 통해서 `charge()`를 찾을지 선택해야 한다.

이때는 **호출하는 변수의 타입(클래스)을 기준으로 선택**한다. `electricCar` 변수의 타입이 `ElectricCar`이므로 인스턴스 내부에 같은 타입인 `ElectricCar`를 통해서 `charge()`를 호출한다.

`electricCar.move()` **호출**



`electricCar.move()`를 호출하면 먼저 `x001` 참조로 이동한다. 내부에는 `Car`, `ElectricCar` 두가지 타입이 있다. 이때 호출하는 변수인 `electricCar`의 타입이 `ElectricCar`이므로 이 타입을 선택한다.
그런데 `ElectricCar`에는 `move()` 메서드가 없다. 상속 관계에서는 자식 타입에 해당 기능이 없으면 부모 타입으로 올라가서 찾는다. 이 경우 `ElectricCar`의 부모인 `Car`로 올라가서 `move()`를 찾는다. 부모인 `Car`에 `move()`가 있으므로 부모에 있는 `move()` 메서드를 호출한다.

만약 부모에서도 해당 기능을 찾지 못하면 더 상위 부모에서 필요한 기능을 찾아본다. 부모에 부모로 계속 올라가면서 필드나 메서드를 찾는 것이다. 물론 계속 찾아도 없으면 컴파일 오류가 발생한다.

지금까지 설명한 상속과 메모리 구조는 반드시 이해해야 한다!

- 상속 관계의 객체를 생성하면 그 내부에는 부모와 자식이 모두 생성된다.
- 상속 관계의 객체를 호출할 때, 대상 타입을 정해야 한다. 이때 호출자의 타입을 통해 대상 타입을 찾는다.
- 현재 타입에서 기능을 찾지 못하면 상위 부모 타입으로 기능을 찾아서 실행한다. 기능을 찾지 못하면 컴파일 오류가 발생한다.

상속과 기능 추가

이번에는 상속 관계의 장점을 알아보기 위해, 상속 관계에 다음 기능을 추가해보자.

- 모든 차량에 문열기(`openDoor()`) 기능을 추가해야 한다.
- 새로운 수소차(`HydrogenCar`)를 추가해야 한다.
  - 수소차는 `fillHydrogen()` 기능을 통해 수소를 충전할 수 있다.

기존 코드를 유지하기 위해 ex3 패키지를 새로 만들자

```
package extends1.ex3;

public class Car {

    public void move() {
        System.out.println("차를 이동합니다.");
    }

    //추가
    public void openDoor() {
        System.out.println("문을 엽니다.");
    }
}
```

모든 차량에 문열기 기능을 추가할 때는 상위 부모인 `Car`에 `openDoor()` 기능을 추가하면 된다. 이렇게 하면 `Car`의 자식들은 해당 기능을 모두 물려받게 된다. 만약 상속 관계가 아니었다면 각각의 차량에 해당 기능을 모두 추가해야 한다.

```
package extends1.ex3;

public class ElectricCar extends Car {

    public void charge() {
        System.out.println("충전합니다.");
    }
}
```

기존 코드와 같다.

```
package extends1.ex3;

public class GasCar extends Car {

    public void fillUp() {
        System.out.println("기름을 주유합니다.");
    }
}
```

기존 코드와 같다.

```
package extends1.ex3;

//추가
public class HydrogenCar extends Car {

    public void fillHydrogen() {
        System.out.println("수소를 충전합니다.");
    }
}
```

수소차를 추가했다. `Car`를 상속받은 덕분에 `move()`, `openDoor()`와 같은 기능을 바로 사용할 수 있다.
수소차는 전용 기능인 수소 충전(`fillHydrogen()`) 기능을 제공한다.

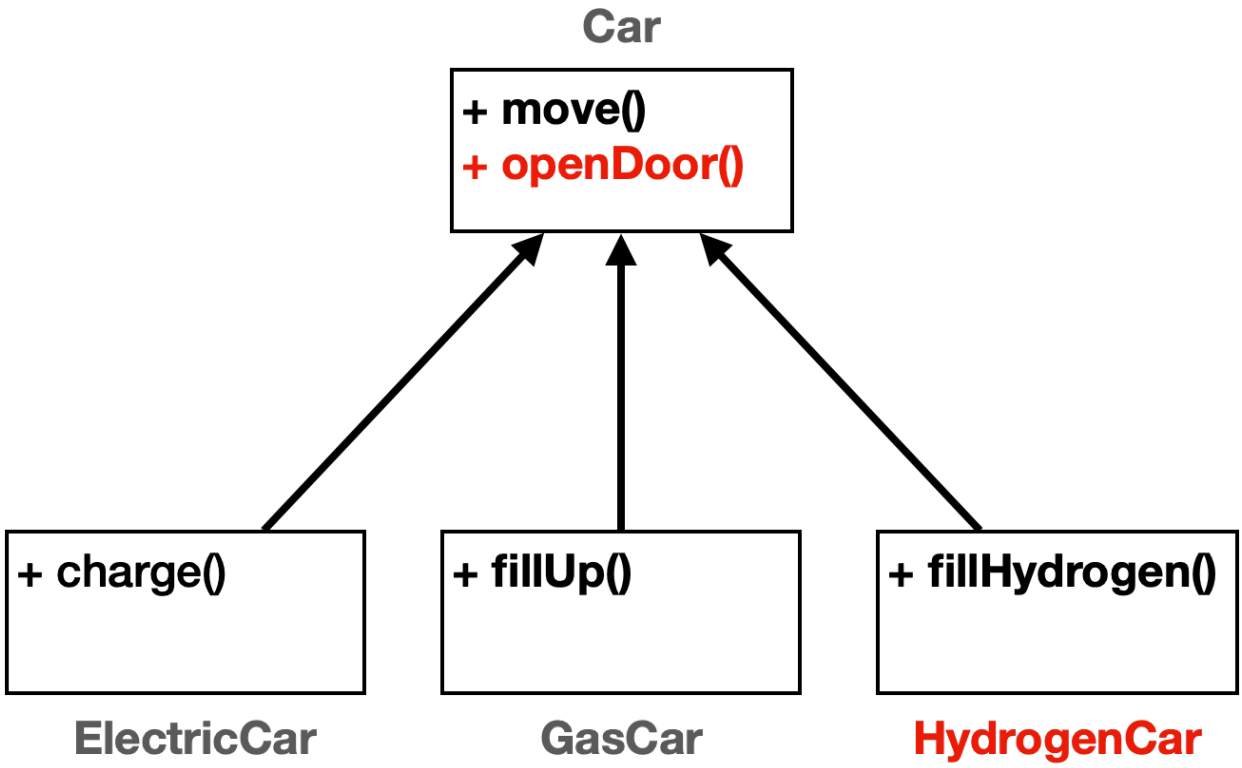
```
package extends1.ex3;

public class CarMain {
```

```
    public static void main(String[] args) {
        ElectricCar electricCar = new ElectricCar();
        electricCar.move();
        electricCar.charge();
        electricCar.openDoor();

        GasCar gasCar = new GasCar();
        gasCar.move();
        gasCar.fillUp();
        gasCar.openDoor();

        HydrogenCar hydrogenCar = new HydrogenCar();
        hydrogenCar.move();
        hydrogenCar.fillHydrogen();
        hydrogenCar.openDoor();
    }
}
```

실행 결과

```
차를 이동합니다.
충전합니다.
문을 엽니다.
차를 이동합니다.
기름을 주유합니다.
문을 엽니다.
차를 이동합니다.
수소를 충전합니다.
문을 엽니다.
```

기능 추가와 클래스 확장



상속 관계 덕분에 중복은 줄어들고, 새로운 수소차를 편리하게 확장(extend)한 것을 알 수 있다.

상속과 메서드 오버라이딩

부모 타입의 기능을 자식에서는 다르게 재정의 하고 싶을 수 있다.
예를 들어서 자동차의 경우 `Car.move()` 라는 기능이 있다. 이 기능을 사용하면 단순히 "차를 이동합니다."라고 출력한다. 전기차의 경우 보통 더 빠르기 때문에 전치가가 `move()` 를 호출한 경우에는 "전기차를 빠르게 이동합니다."라고 출력을 변경하고 싶다.

이렇게 부모에게서 상속 받은 기능을 자식이 **재정의 하는 것을 메서드 오버라이딩(Overriding)**이라 한다.

기존 코드를 유지하기 위해 overriding 패키지를 새로 만들자

```
package extends1.overriding;

public class Car {

    public void move() {
        System.out.println("차를 이동합니다.");
    }
```

```
    public void openDoor() {
        System.out.println("문을 엽니다.");
    }
}
```

기존 코드와 같다.

```
package extends1.overriding;

public class GasCar extends Car {

    public void fillUp() {
        System.out.println("기름을 주유합니다.");
    }
}
```

기존 코드와 같다.

```
package extends1.overriding;

public class ElectricCar extends Car {

    @Override
    public void move() {
        System.out.println("전기차를 빠르게 이동합니다.");
    }

    public void charge() {
        System.out.println("충전합니다.");
    }
}
```

`ElectricCar`는 부모인 `Car`의 `move()` 기능을 그대로 사용하고 싶지 않다. 메서드 이름은 같지만 새로운 기능을 사용하고 싶다. 그래서 `ElectricCar`의 `move()` 메서드를 새로 만들었다.
이렇게 부모의 기능을 자식이 새로 재정의하는 것을 메서드 오버라이딩이라 한다.
이제 `ElectricCar`의 `move()`를 호출하면 `Car`의 `move()`가 아니라 `ElectricCar`의 `move()`가 호출된다.

@Override
`@`이 붙은 부분을 애노테이션이라 한다. 애노테이션은 주석과 비슷한데, 프로그램이 읽을 수 있는 특별한 주석이라 생각하면 된다. 애노테이션에 대한 자세한 내용은 따로 설명한다.

이 애노테이션은 상위 클래스의 메서드를 오버라이드하는 것임을 나타낸다.

이름 그대로 오버라이딩한 메서드 위에 이 애노테이션을 붙여야 한다.
컴파일러는 이 애노테이션을 보고 메서드가 정확히 오버라이드 되었는지 확인한다. 오버라이딩 조건을 만족시키지 않으면 컴파일 에러를 발생시킨다. 따라서 실수로 오버라이딩을 못하는 경우를 방지해준다. 예를 들어서 이 경우에 만약 부모에 `move()` 메서드가 없다면 컴파일 오류가 발생한다. 참고로 이 기능은 필수는 아니지만 코드의 명확성을 위해 붙여주는 것이 좋다.

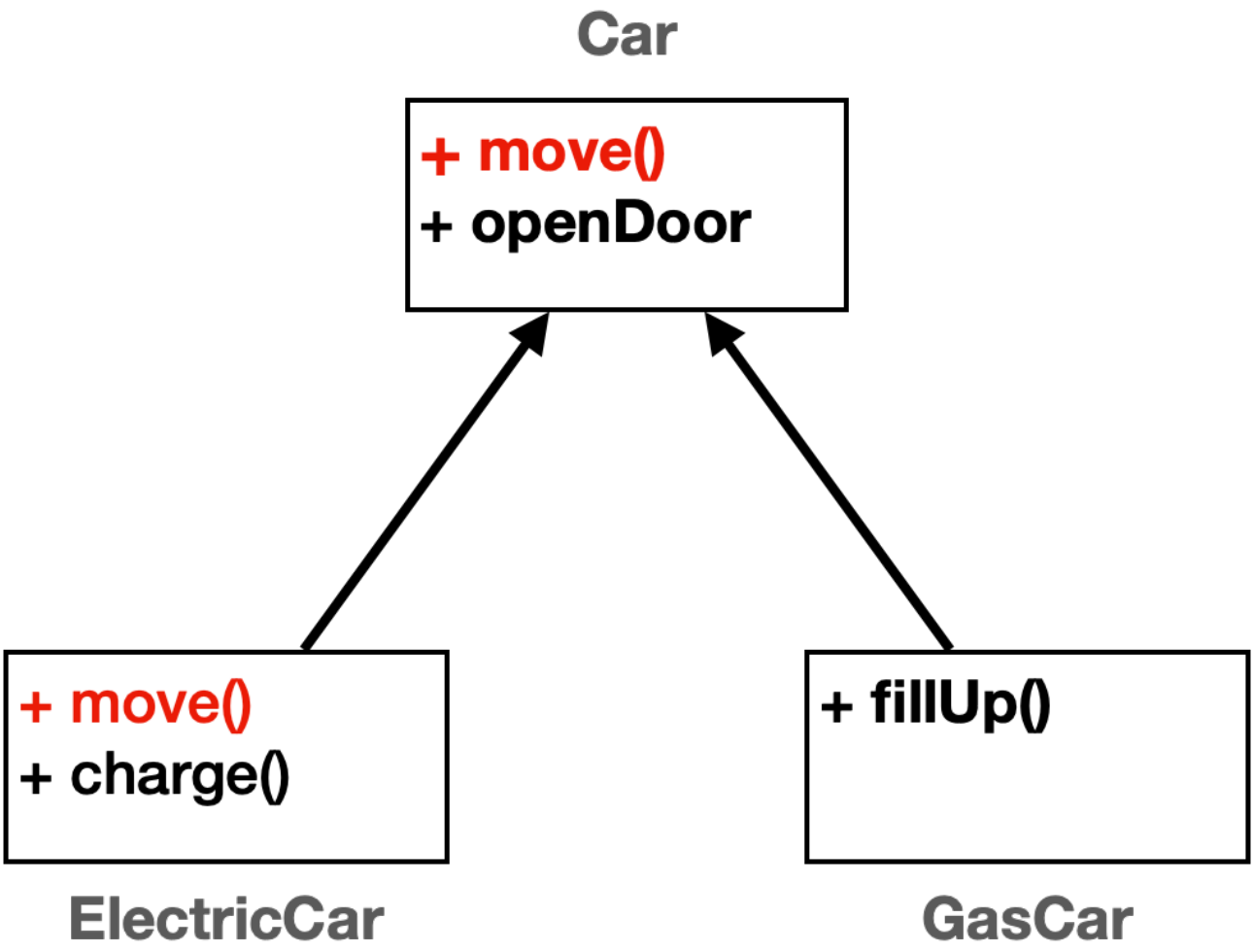
```
package extends1.overriding;

public class CarMain {

    public static void main(String[] args) {
        ElectricCar electricCar = new ElectricCar();
        electricCar.move();

        GasCar gasCar = new GasCar();
        gasCar.move();
    }
}
```

실행 결과

```
전기차를 빠르게 이동합니다.
차를 이동합니다.
```

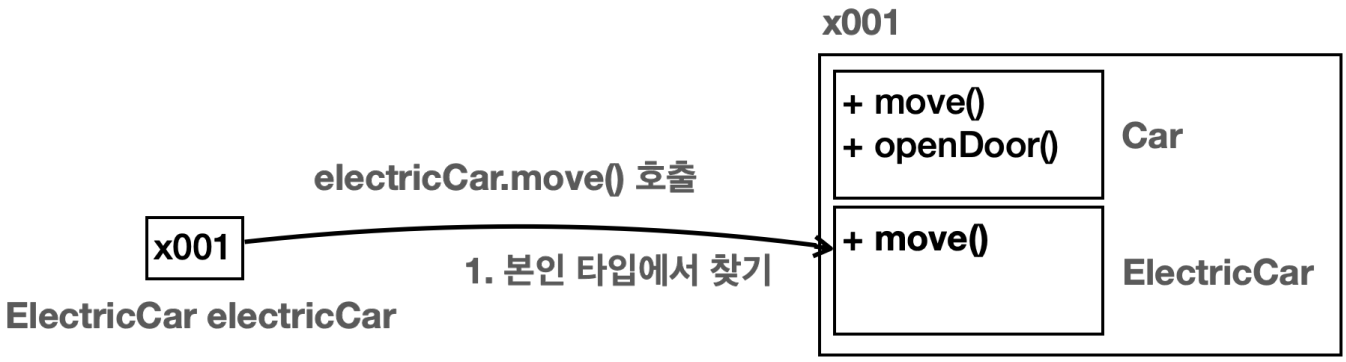
실행 결과를 보면 `electricCar.move()`를 호출했을 때 오버라이딩한 `ElectricCar.move()` 메서드가 실행된 것을 확인할 수 있다.

오버라이딩과 클래스



`Car`의 `move()` 메서드를 `ElectricCar`에서 오버라이딩 했다.

오버라이딩과 메모리 구조



1. `electricCar.move()`를 호출한다.
2. 호출한 `electricCar`의 타입은 `ElectricCar`이다. 따라서 인스턴스 내부의 `ElectricCar`타입에서 시작한다.
3. `ElectricCar` 타입에 `move()` 메서드가 있다. 해당 메서드를 실행한다. 이때 실행할 메서드를 이미 찾았으므로 부모 타입을 찾지 않는다.

오버로딩(Overloading)과 오버라이딩(Overriding)

- **메서드 오버로딩**: 메서드 이름이 같고 매개변수(파라미터)가 다른 메서드를 여러개 정의하는 것을 메서드 오버로

딩(Overloading)이라 한다. 오버로딩은 번역하면 과적인데, 과하게 물건을 담았다는 뜻이다. 따라서 같은 이름의 메서드를 여러개 정의했다고 이해하면 된다.

- **메서드 오버라이딩**: 메서드 오버라이딩은 하위 클래스에서 상위 클래스의 메서드를 재정의하는 과정을 의미한다. 따라서 상속 관계에서 사용한다. 부모의 기능을 자식이 다시 정의하는 것이다. 오버라이딩을 단순히 해석하면 무언가를 넘어서 타는 것을 말한다. 자식의 새로운 기능이 부모의 기존 기능을 넘어 타서 기존 기능을 새로운 기능으로 덮어버린다고 이해하면 된다. 오버라이딩을 우리말로 번역하면 무언가를 다시 정의한다고 해서 **재정의**라 한다. 상속 관계에서는 기존 기능을 다시 정의한다고 이해하면 된다. 실무에서는 메서드 오버라이딩, 메서드 재정의 둘 다 사용한다.

메서드 오버라이딩 조건

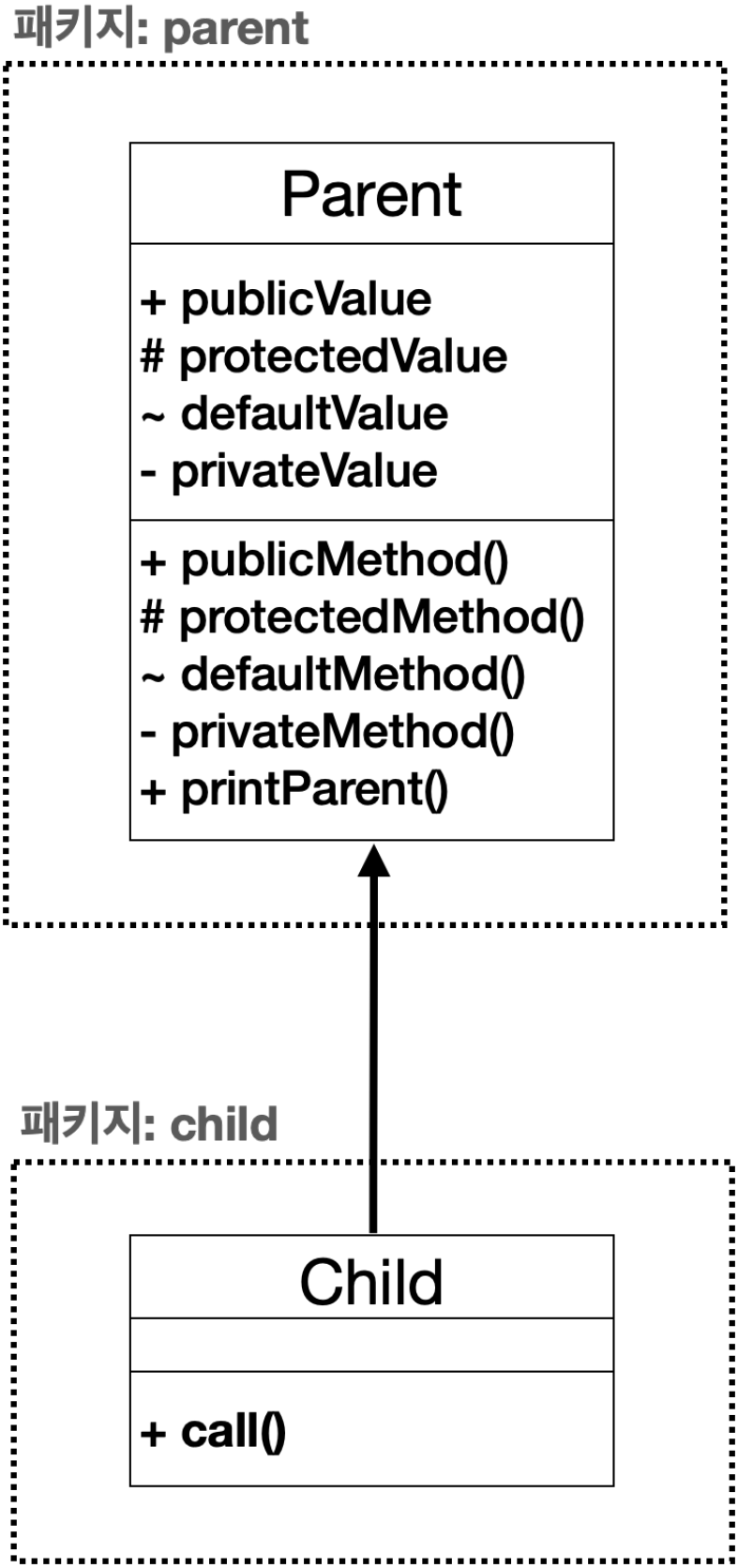
메서드 오버라이딩은 다음과 같은 까다로운 조건을 가지고 있다.
다음 내용은 아직 학습하지 않은 내용들도 있으므로 모두 이해하려고 하기 보다는 참고만 하자.
지금은 단순히 **부모 메서드와 같은 메서드를 오버라이딩 할 수 있다 정도로 이해하면 충분한다.**

메서드 오버라이딩 조건

- **메서드 이름**: 메서드 이름이 같아야 한다.
- **메서드 매개변수(파라미터)**: 매개변수(파라미터) 타입, 순서, 개수가 같아야 한다.
- **반환 타입**: **반환 타입이 같아야 한다**. 단 반환 타입이 하위 클래스 타입일 수 있다.
- **접근 제어자**: 오버라이딩 메서드의 접근 제어자는 상위 클래스의 메서드보다 더 제한적이어서는 안된다. 예를 들어, 상위 클래스의 메서드가 `protected`로 선언되어 있으면 하위 클래스에서 이를 `public` 또는 `protected`로 오버라이드할 수 있지만, `private` 또는 `default`로 오버라이드 할 수 없다.
- **예외**: 오버라이딩 메서드는 상위 클래스의 메서드보다 더 많은 체크 예외를 `throws`로 선언할 수 없다. 하지만 더 적거나 같은 수의 예외, 또는 하위 타입의 예외는 선언할 수 있다. 예외를 학습해야 이해할 수 있다. 예외는 뒤에서 다룬다.
- `static`, `final`, `private`: 키워드가 붙은 메서드는 오버라이딩 될 수 없다.
  - `static`은 클래스 레벨에서 작동하므로 인스턴스 레벨에서 사용하는 오버라이딩이 의미가 없다. 쉽게 이야기해서 그냥 클래스 이름을 통해 필요한 곳에 직접 접근하면 된다.
  - `final` 메서드는 재정의를 금지한다.
  - `private` 메서드는 해당 클래스에서만 접근 가능하기 때문에 하위 클래스에서 보이지 않는다. 따라서 오버라이딩 할 수 없다.
- **생성자 오버라이딩**: 생성자는 오버라이딩 할 수 없다.

상속과 접근 제어

상속 관계와 접근 제어에 대해 알아보자. 참고로 접근 제어를 자세히 설명하기 위해 부모와 자식의 패키지를 따로 분리하였다. 이 부분에 유의해서 예제를 만들어보자.



접근 제어자를 표현하기 위해 UML 표기법을 일부 사용했다.

- `+` : public
- `#` : protected
- `~` : default
- `-` : private

접근 제어자를 잠시 복습해보자.

접근 제어자의 종류

- `private` : 모든 외부 호출을 막는다.
- `default` (package-private): 같은 패키지안에서 호출은 허용한다.
- `protected` : 같은 패키지안에서 호출은 허용한다. 패키지가 달라도 상속 관계의 호출은 허용한다.
- `public` : 모든 외부 호출을 허용한다.

순서대로 `private` 이 가장 많이 차단하고, `public` 이 가장 많이 허용한다.
`private -> default -> protected -> public`

그림과 같이 다양한 접근 제어자를 사용하도록 코드를 작성해보자.

```
package extends1.access.parent;

public class Parent {

    public int publicValue;
    protected int protectedValue;
    int defaultValue;
    private int privateValue;

    public void publicMethod() {
        System.out.println("Parent.publicMethod");
    }
    protected void protectedMethod() {
        System.out.println("Parent.protectedMethod");
    }
    void defaultMethod() {
        System.out.println("Parent.defaultMethod");
    }
    private void privateMethod() {
        System.out.println("Parent.privateMethod");
    }
```

```
    public void printParent() {
        System.out.println("==Parent 메서드 안==");
        System.out.println("publicValue = " + publicValue);
        System.out.println("protectedValue = " + protectedValue);
        System.out.println("defaultValue = " + defaultValue); //부모 메서드 안에서 접
근 가능
        System.out.println("privateValue = " + privateValue); //부모 메서드 안에서 접
근 가능

        //부모 메서드 안에서 모두 접근 가능
        defaultMethod();
        privateMethod();
    }
}
```

부모 클래스인 `Parent`에는 `public`, `protected`, `default`, `private`과 같은 모든 접근 제어자가 필드와 메서드에 모두 존재한다.

```
package extends1.access.child;

import extends1.access.parent.Parent;

public class Child extends Parent {

    public void call() {
        publicValue = 1;
        protectedValue = 1; //상속 관계 or 같은 패키지
        //defaultValue = 1; //다른 패키지 접근 불가, 컴파일 오류
        //privateValue = 1; //접근 불가, 컴파일 오류

        publicMethod();
        protectedMethod(); //상속 관계 or 같은 패키지
        //defaultMethod(); //다른 패키지 접근 불가, 컴파일 오류
        //privateMethod(); //접근 불가, 컴파일 오류

        printParent();
    }
}
```

둘의 패키지가 다르다는 부분의 유의하자

자식 클래스인 `Child`에서 부모 클래스인 `Parent`에 얼마나 접근할 수 있는지 확인해보자.

- `publicValue = 1` : 부모의 `public` 필드에 접근한다. `public` 이므로 접근할 수 있다.
- `protectedValue = 1` : 부모의 `protected` 필드에 접근한다. 자식과 부모는 다른 패키지이지만, **상속 관계이므로 접근할 수 있다.**
- `defaultValue = 1` : 부모의 `default` 필드에 접근한다. 자식과 부모가 다른 패키지이므로 접근할 수 없다.
- `privateValue = 1` : 부모의 `private` 필드에 접근한다. `private` 은 모든 외부 접근을 막으므로 자식이라도 호출할 수 없다.

메서드의 경우도 앞서 설명한 필드와 동일하다.

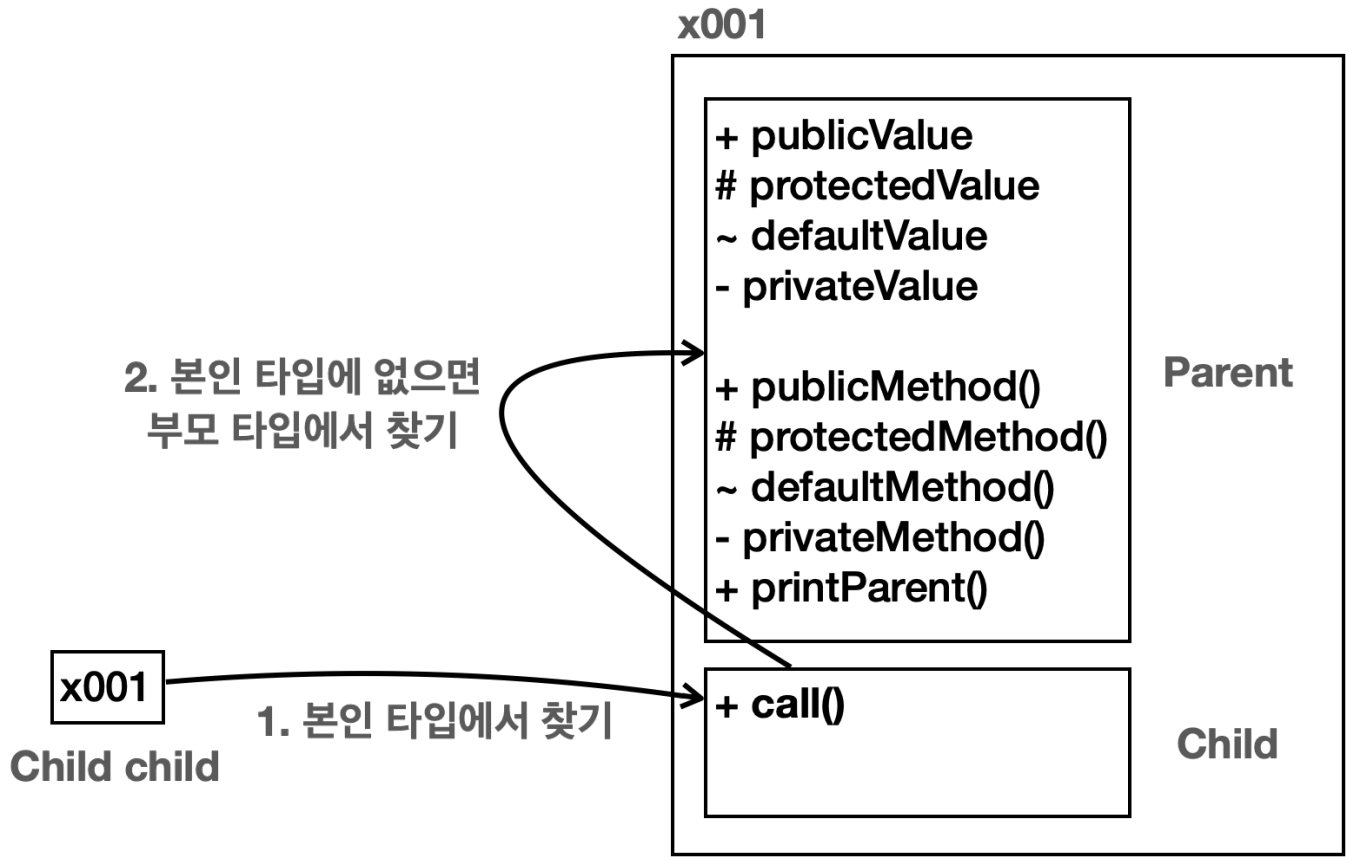
```
package extends1.access;

import extends1.access.child.Child;

public class ExtendsAccessMain {

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

실행 결과

```
Parent.publicMethod
Parent.protectedMethod
==Parent 메서드 안==
publicValue = 1
protectedValue = 1
defaultValue = 0
privateValue = 0
Parent.defaultMethod
Parent.privateMethod
```

코드를 실행해보면 `Child.call()` ➜ `Parent.printParent()` 순서로 호출한다.
`Child` 는 부모의 `public` , `protected` 필드나 메서드만 접근할 수 있다. 반면에 `Parent.printParent()` 의 경우 `Parent` 안에 있는 메서드이기 때문에 `Parent` 자신의 모든 필드와 메서드에 얼마든지 접근할 수 있다.

접근 제어와 메모리 구조

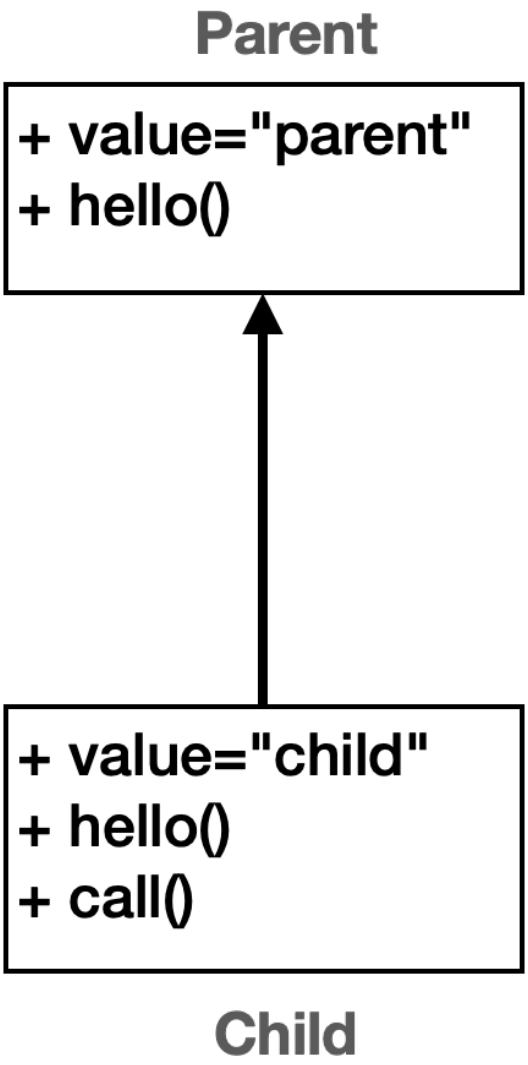


본인 타입에 없으면 부모 타입에서 기능을 찾는데, 이때 접근 제어자가 영향을 준다. 왜냐하면 객체 내부에서는 자식과 부모가 구분되어 있기 때문이다. 결국 자식 타입에서 부모 타입의 기능을 호출할 때, 부모 입장에서 보면 외부에서 호출한 것과 같다.

super - 부모 참조

부모와 자식의 필드명이 같거나 메서드가 오버라이딩 되어 있으면, 자식에서 부모의 필드나 메서드를 호출할 수 없다. 이때 `super` 키워드를 사용하면 부모를 참조할 수 있다. `super`는 이름 그대로 부모 클래스에 대한 참조를 나타낸다.

다음 예를 보자. 부모의 필드명과 자식의 필드명이 둘다 `value`로 똑같다. 메서드도 `hello()`로 자식에서 오버라이딩 되어 있다. 이때 자식 클래스에서 부모 클래스의 `value`와 `hello()`를 호출하고 싶다면 `super` 키워드를 사용하면 된다.




```java
package extends1.super1;

public class Parent {
    public String value = "parent";

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

```java
package extends1.super1;

public class Child extends Parent {
    public String value = "child";

    @Override
    public void hello() {
```

```
        System.out.println("Child.hello");
    }

    public void call() {
        System.out.println("this value = " + this.value); //this 생략 가능
        System.out.println("super value = " + super.value);

        this.hello(); //this 생략 가능
        super.hello();
    }
}
```

`call()` 메서드를 보자.

- `this`는 자기 자신의 참조를 뜻한다. `this`는 생략할 수 있다.
- `super`는 부모 클래스에 대한 참조를 뜻한다.
- 필드 이름과 메서드 이름이 같지만 `super`를 사용해서 부모 클래스에 있는 기능을 사용할 수 있다.

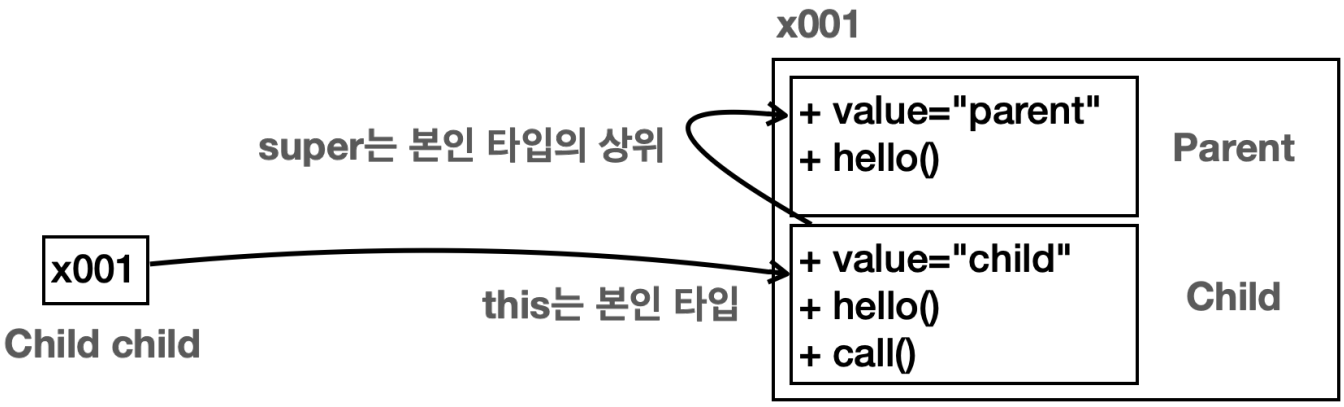
```
package extends1.super1;

public class Super1Main {
    public static void main(String[] args) {
        Child child = new Child();
        child.call();
    }
}
```

실행 결과

```
this value = child
super value = parent
Child.hello
Parent.hello
```

실행 결과를 보면 `super`를 사용한 경우 부모 클래스의 기능을 사용한 것을 확인할 수 있다.

super 메모리 그림



super - 생성자

상속 관계의 인스턴스를 생성하면 결국 메모리 내부에는 자식과 부모 클래스가 각각 다 만들어진다. `Child`를 만들면 부모인 `Parent`까지 함께 만들어지는 것이다. 따라서 각각의 생성자도 모두 호출되어야 한다.

상속 관계를 사용하면 자식 클래스의 생성자에서 부모 클래스의 생성자를 반드시 호출해야 한다.(규칙)

상속 관계에서 부모의 생성자를 호출할 때는 `super(...)`를 사용하면 된다.

예제를 통해 상속 관계에서 생성자를 어떻게 사용하는지 알아보자.

```
package extends1.super2;

public class ClassA {

    public ClassA() {
        System.out.println("ClassA 생성자");
    }
}
```

- `ClassA`는 최상위 부모 클래스이다.

```
package extends1.super2;

public class ClassB extends ClassA {

    public ClassB(int a) {
        super(); //기본 생성자 생략 가능
        System.out.println("ClassB 생성자 a="+a);
```

```
    }

    public ClassB(int a, int b) {
        super(); //기본 생성자 생략 가능
        System.out.println("ClassB 생성자 a="+a + " b=" + b);
    }

}
```

- `ClassB`는 `ClassA`를 상속 받았다. 상속을 받으면 **생성자의 첫줄**에 `super(...)`를 사용해서 부모 클래스의 생성자를 호출해야 한다.
  - 예외로 생성자 첫줄에 `this(...)`를 사용할 수는 있다. 하지만 `super(...)`는 자식의 생성자 안에서 언젠가는 반드시 호출해야 한다.
- 부모 클래스의 생성자가 기본 생성자(파라미터가 없는 생성자)인 경우에는 `super()`를 생략할 수 있다.
  - 상속 관계에서 첫줄에 `super(...)`를 생략하면 자바는 부모의 기본 생성자를 호출하는 `super()`를 자동으로 만들어준다.
  - 참고로 기본 생성자를 많이 사용하기 때문에 편의상 이런 기능을 제공한다.

```
package extends1.super2;

public class ClassC extends ClassB {

    public ClassC() {
        super(10, 20);
        System.out.println("ClassC 생성자");
    }
}
```

- `ClassC`는 `ClassB`를 상속 받았다. `ClassB` 다음 두 생성자가 있다.
  - `ClassB(int a)`
  - `ClassB(int a, int b)`
- 생성자는 하나만 호출할 수 있다. 두 생성자 중에 하나를 선택하면 된다.
  - `super(10, 20)`를 통해 부모 클래스의 `ClassB(int a, int b)` 생성자를 선택했다.
- 참고로 `ClassC`의 부모인 `ClassB`에는 기본 생성자가 없다. 따라서 부모의 기본 생성자를 호출하는 `super()`를 사용하거나 생략할 수 없다.

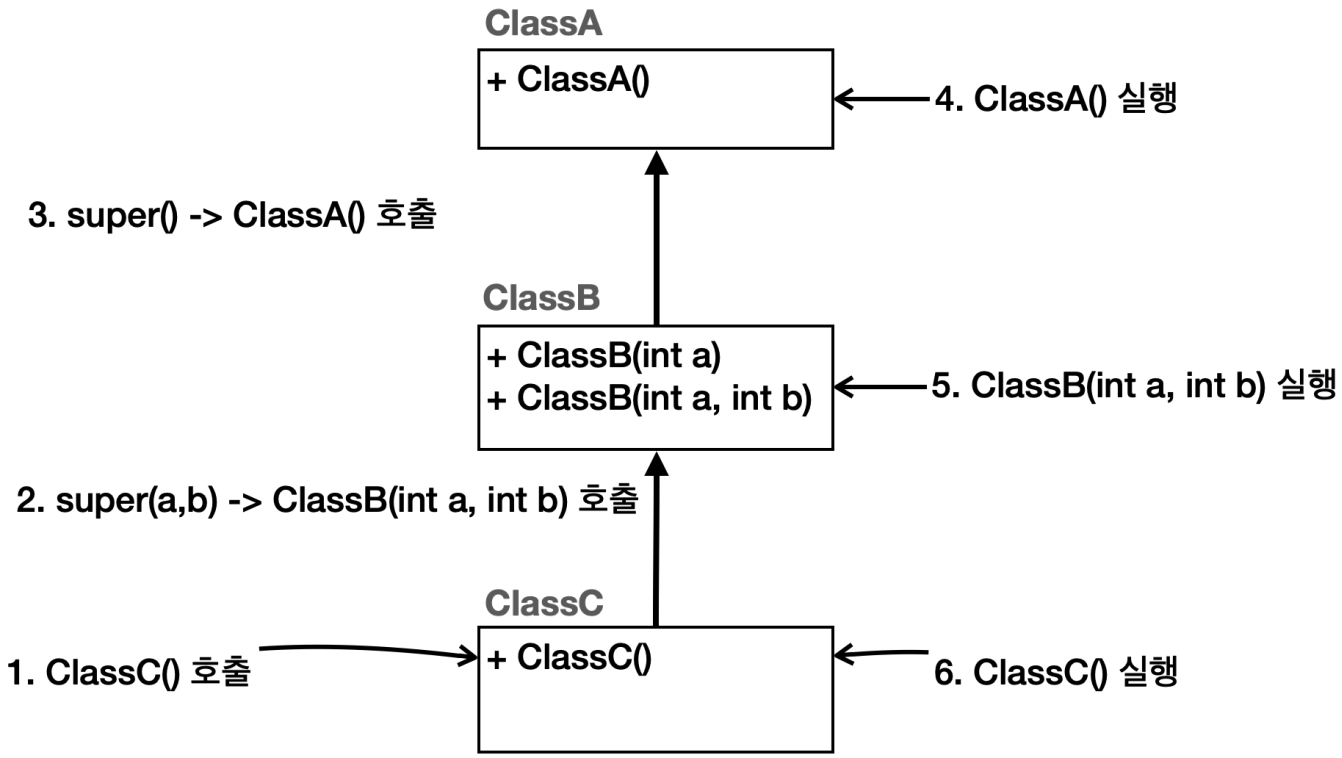
```
package extends1.super2;

public class Super2Main {

    public static void main(String[] args) {
        ClassC classC = new ClassC();
```

```
    }
}
```

실행 결과

```
ClassA 생성자
ClassB 생성자 a=10 b=20
ClassC 생성자
```

실행해보면 `ClassA` ➜ `ClassB` ➜ `ClassC` 순서로 실행된다. 생성자의 실행 순서가 결과적으로 최상위 부모부터 실행되어서 하나씩 아래로 내려오는 것이다. 따라서 초기화는 최상위 부모부터 이루어진다. 왜냐하면 자식 생성자의 첫 줄에서 부모의 생성자를 호출해야 하기 때문이다.



1~3까지의 과정

`new ClassC()`를 통해 `ClassC` 인스턴스를 생성한다. 이때 `ClassC()`의 생성자가 먼저 호출되는 것이 맞다. 하지만 `ClassC()`의 성생자는 가장 먼저 `super(..)`를 통해 `ClassB(...)`의 생성자를 호출한다. `ClassB()`의 생성자도 부모인 `ClassA()`의 생성자를 가장 먼저 호출한다.

4~6까지의 과정

- `ClassA()`의 생성자는 최상위 부모이다. 생성자 코드를 실행하면서 `"ClassA 생성자"`를 출력한다. `ClassA()` 생성자 호출이 끝나면 `ClassA()`를 호출한 `ClassB(...)` 생성자로 제어권이 돌아간다.
- `ClassB(...)` 생성자가 코드를 실행하면서 `"ClassB 생성자 a=10 b=20"`를 출력한다. 생성자 호출이 끝나면 `ClassB(...)`를 호출한 `ClassC()`의 생성자로 제어권이 돌아간다.
- `ClassC()`가 마지막으로 생성자 코드를 실행하면서 `"ClassC 생성자"`를 출력한다.

정리

- 상속 관계의 생성자 호출은 결과적으로 부모에서 자식 순서로 실행된다. 따라서 부모의 데이터를 먼저 초기화하고 그 다음에 자식의 데이터를 초기화한다.
- 상속 관계에서 자식 클래스의 생성자 첫줄에 반드시 `super(...)`를 호출해야 한다. 단 기본 생성자(`super()`)인 경우 생략할 수 있다.

this(...)와 함께 사용

코드의 첫줄에 `this(...)`를 사용하더라도 반드시 한번은 `super(...)`를 호출해야 한다.

코드 변경

```
package extends1.super2;

public class ClassB extends ClassA {

    public ClassB(int a) {
        this(a, 0); //기본 생성자 생략 가능
        System.out.println("ClassB 생성자 a=" + a);
    }

    public ClassB(int a, int b) {
        super(); //기본 생성자 생략 가능
        System.out.println("ClassB 생성자 a=" + a + " b=" + b);
    }
}
```

```
package extends1.super2;

public class Super2Main {

    public static void main(String[] args) {
        //ClassC classC = new ClassC();
        ClassB classB = new ClassB(100);
    }
}
```

실행 결과

```
ClassA 생성자
ClassB 생성자 a=100 b=0
ClassB 생성자 a=100
```

문제와 풀이

문제: 상속 관계 상품

쇼핑몰의 판매 상품을 만들어보자.

- `Book`, `Album`, `Movie` 이렇게 3가지 상품을 클래스로 만들어야 한다.
- 코드 중복이 없게 상속 관계를 사용하자. 부모 클래스는 `Item`이라는 이름을 사용하면 된다.
- 공통 속성: `name`, `price`
  - `Book`: 저자(`author`), isbn(`isbn`)
  - `Album`: 아티스트(`artist`)
  - `Movie`: 감독(`director`), 배우(`actor`)

다음 코드와 실행결과를 참고해서 `Item`, `Book`, `Album`, `Movie` 클래스를 만들어보자.

```
package extends1.ex;

public class ShopMain {

    public static void main(String[] args) {
        Book book = new Book("JAVA", 10000, "han", "12345");
        Album album = new Album("앨범1", 15000,"seo");
        Movie movie = new Movie("영화1", 18000,"감독1", "배우1");

        book.print();
        album.print();
        movie.print();

        int sum = book.getPrice() + album.getPrice() + movie.getPrice();
        System.out.println("상품 가격의 합: " + sum);
    }
}
```

실행 결과

```
이름:JAVA, 가격:10000
- 저자:han, isbn:12345
이름:앨범1, 가격:15000
- 아티스트:seo
```

```
이름:영화1, 가격:18000
- 감독:감독1, 배우:배우1
상품 가격의 합: 43000
```

정답

```java
package extends1.ex;

public class Item {
    private String name;
    private int price;

    public Item(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public int getPrice() {
        return price;
    }

    public void print() {
        System.out.println("이름:" + name + ", 가격:" + price);
    }
}
```

```java
package extends1.ex;

public class Book extends Item {
    private String author;
    private String isbn;

    public Book(String name, int price, String author, String isbn) {
        super(name, price);
        this.author = author;
        this.isbn = isbn;
    }

    @Override
    public void print() {
        super.print();
        System.out.println("- 저자:" + author + ", isbn:" + isbn);
```

```
    }

}
```

```
package extends1.ex;

public class Album extends Item {
    private String artist;

    public Album(String name, int price, String artist) {
        super(name, price);
        this.artist = artist;
    }

    @Override
    public void print() {
        super.print();
        System.out.println("- 아티스트:" + artist);
    }
}
```

```
package extends1.ex;

public class Movie extends Item {
    private String director;
    private String actor;

    public Movie(String name, int price, String director, String actor) {
        super(name, price);
        this.director = director;
        this.actor = actor;
    }

    @Override
    public void print() {
        super.print();
        System.out.println("- 감독:" + director + ", 배우:" + actor);
    }
}
```

정리

클래스와 메서드에 사용되는 final

클래스에 `final`

- 상속 끝!
- `final`로 선언된 클래스는 확장될 수 없다. 다른 클래스가 `final`로 선언된 클래스를 상속받을 수 없다.
- 예: `public final class MyFinalClass {...}`

메서드에 `final`

- 오버라이딩 끝!
- `final`로 선언된 메서드는 오버라이드 될 수 없다. 상속받은 서브 클래스에서 이 메서드를 변경할 수 없다.
- 예: `public final void myFinalMethod() {...}`