객체지향프로그래밍_정리

객체들이 서로 각자의 일을 알고 있다.
그러한 객체들의 상호작용으로, 일을 할 수 있게 만들면 된다.

#매서드_체이닝

- 하나의 객체에 여러 개의 매서드를 점(.)으로 이어서 호출하는 기법
- return this;와 같은 코드가 있어야 한다.

```
class Calculator { private int result = 0;

public Calculator add(int num) { this.result += num;  return this;

public void displayResult() { System.out.println("현재 결과: " + this.result); } }
```

#Staticfield

- 메모리에 미리 올라가있다
- 객체마다 고유한 값을 가질 필요가 없는 공통 데이터나 상수를 효율적으로 관리

```
public class MyClass {
    // static 필드: 클래스에 속함
    public static int staticField = 10;

    // non-static 메서드: 객체에 속함
    public void nonStaticMethod() {
        System.out.println("이것은 인스턴스 메서드입니다.");
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        // MyClass 객체(인스턴스) 생성
        MyClass myObject = new MyClass();

        // 1. 객체를 통해 static 필드 접근
        System.out.println(myObject.staticField); // 10 (경고는 뜨지만 가능함)

        // 2. 객체를 통해 non-static 메서드 접근
        myObject.nonStaticMethod(); // 정상 작동
    }
}
```

인스턴스화 되면
스태틱한 필드 / 그렇지 않은 매서드든 다 접근해서 사용가능
클래스명.매서드 이렇게 쓰는것이 좋다.

#매서드_오버로딩
똑같은 이름의 매서드를 여러번 정의 가능
중복된 이름으로 출력하기 편하다.

- 가독성이 좋다.
  숫자로 출력 문자로 출력 시 print라는 매서드만 있으면, 직관적
  여러형태로 오버로딩을 하고 있다.
- 사용자 편의성 증대
  출력이 문자 숫자 정수 실수 등 가리지 않는다. 다 다르면 힘들다.
  다형성. 껍데기는 하난데 구현은 여러개 println
  두 개의 정수를 더하는 경우: `sum(int a, int b)`
  세 개의 정수를 더하는 경우: `sum(int a, int b, int c)`
  두 개의 실수를 더하는 경우: `sum(double a, double b)`
  `sumTwoInts`, `sumThreeInts`, `sumTwoDoubles`
  sum으로 통일하지 않으면, 전부 다른 이름을 붙여야함.
- 코드의 일관성 유지
  기능적으로 동일한 역할을 하는 메서드들을 같은 이름으로 묶어 관리

```java
public class Calculator {
    // 1. 두 개의 정수를 더하는 메서드
    public int add(int a, int b) {
        return a + b;
    }

    // 2. 세 개의 정수를 더하는 메서드 (매개변수 개수가 다름)
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // 3. 두 개의 실수를 더하는 메서드 (매개변수 타입이 다름)
    public double add(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator myCalculator = new Calculator();

        // 정수 2개 호출
        System.out.println(myCalculator.add(1, 2));         // 출력: 3

        // 정수 3개 호출
        System.out.println(myCalculator.add(1, 2, 3));      // 출력: 6

        // 실수 2개 호출
        System.out.println(myCalculator.add(1.5, 2.5));     // 출력: 4.0
    }
}
```

add에 3개의 메서드가 있어 add메서드 하나만 호출하면 세가지 기능 가능

#변수_스코프
변수가 선언되고 사용되는 유효범위

클래스 변수 (Static Variable)

```
public class ScopeExample {
    // 클래스 변수: 프로그램 전체에서 공유됨
    public static int classVariable = 10;
```

- 선언 위치 : 클래스 내부, 메서드 외부에서 static 키워드와 함께 선언
- 유효 범위 : 클래스 전체에서 접근 가능. 객체를 생성하지 않고도 클래스명.변수명 형태로 사용 가능. 모든 객체가 이 변수를 공유.
  매서드가 할당될때 매서드가 끝나면 전부 끝
- 생명 주기 : 프로그램이 시작될 때 메모리에 할당되어 프로그램이 끝날 때까지 유지

인스턴스 변수

- 클래스 내부에 static이 없이 선언되는 변수.
- 유효범위 : 객체가 생성될때 메모리에 할당, 객체가 메모리에서 소멸될때 해지.

```
public String studentName;
public int studentId;
```

지역변수

- 메서드, 생성자, 코드블록{} 내에서 생성되는 변수
- 해당 블록 내에서만 유효, 블록을 벗어나면 변수는 소멸. 반드시 사용되기 전에 초기화.

```
public void printInfo() {
// 3. 지역 변수 (Local Variable)
// 이 메서드 내에서만 사용되는 임시 변수.
String tempMessage = "--- 학생 정보 ---"; System.out.println(tempMessage); // 지역 변수 접근
```

```
System.out.println("학교 이름: " + Student.schoolName); // 클래스 변수 접근 (클래스명.변수명)
System.out.println("이름: " + this.studentName);
// 인스턴스 변수 접근
(this.변수명) System.out.println("학번: " + this.studentId);
// 인스턴스 변수 접근
(this.변수명) }
```

전역스코프

- **전역 스코프**는 프로그램 전체에서 접근할 수 있는 가장 넓은 범위입니다.

전역 스코프에 선언된 변수는 **전역 변수**라고 부르며, 프로그램의 어떤 위치에서든 자유롭게 사용하고 수정할 수 있습니다.

- **장점**: 프로그램의 여러 부분에서 공통된 데이터를 쉽게 공유할 수 있습니다.
- **단점**:

  ```
  - 변수 이름이 충돌할 위험이 큽니다.

  - 어떤 코드에서 변수 값이 변경되었는지 추적하기 어려워, 예상치 못한 오류(버그)를 발생시키기 쉽습니다.

  - 프로그램 종료 시까지 메모리에 남아있어 메모리 관리에 비효율적일 수 있습니다
  ```

  블록스코프
- **블록 스코프**는 `{}`(중괄호)로 묶인 특정 코드 블록 내에서만 유효한 범위입니다.

블록 스코프에 선언된 변수는 해당 블록 안에서만 접근할 수 있으며, 블록을 벗어나면 더 이상 유효하지 않게 됩니다. 주로 `if`문, `for`문, `while`문, `try-catch`문 등에서 사용됩니다.

- **장점**:
  - 변수 이름 충돌을 방지하여 코드를 안전하게 만듭니다.
  - 변수가 필요한 특정 블록 내에서만 존재하므로 메모리 관리에 효율적입니다.
  - 코드의 가독성과 유지보수성을 높여줍니다.
- **단점**: 특정 블록 밖에서는 변수를 사용할 수 없습니다.

#생성자

클래스로부터 객체를 생성할 때 호출되는 특별한 메서드.

- **주요 역할**
  - **객체 초기화**: 객체의 필드(속성)들을 원하는 초기값으로 설정합니다.
  - **객체 생성 과정 관리**: 객체 생성에 필요한 준비 작업을 수행합니다.
- **생성자의 특징**
  - **클래스와 동일한 이름**: 생성자의 이름은 반드시 클래스의 이름과 같아야 합니다.
  - **반환 타입 없음**: 생성자는 값을 반환하지 않으며, `void`와 같은 반환 타입을 명시하지 않습니다.
  - **자동 호출**: `new` 키워드를 사용해서 객체를 생성할 때 자동으로 한 번만 호출됩니다. 개발자가 명시적으로 호출할 수 없습니다.
- **`this()`와 `this.`의 구분**
  - `this()`: **생성자 내부에서 다른 생성자를 호출**하는 데 사용됩니다.
  - `this.`: **내 자신(인스턴스)**의 속성이나 메서드를 가리키는 데 사용됩니다.

#객체지향_프로그래밍의_핵심요소

객체는 **속성(데이터)**과 **기능(동작)**으로 구성됩니다.

- **필드(Field)**: 객체의 **속성** 또는 **상태**를 나타내는 변수입니다. (예: 자동차의 `color`, 사람의 `age`).
- **메서드(Method)**: 객체의 **기능** 또는 **행동**을 나타내는 함수입니다. (예: 자동차의 `accelerate()`, 사람의 `sleep()`).
- **생성자(Constructor)**: 객체를 만들 때 필드를 초기화하는 특별한 메서드입니다.

#접근_제한자

클래스, 필드, 메서드에 대한 접근 범위를 제어합니다.

- **`public`**: **모든 클래스**에서 접근 가능합니다. 외부로 공개되어야 하는 멤버에 사용됩니다.
- **`private`**: **오직 해당 클래스 내부**에서만 접근 가능합니다. **정보 은닉(Information Hiding)**이라는 중요한 개념과 연결됩니다.
- **예시**: 로또 프로그램에서 로또 번호를 담는 필드는 `private`으로 선언하여 외부에서 직접 변경하지 못하게 하고, `public`으로 선언된 `getNumber()`와 같은 메서드를 통해서만 접근하게 만듭니다.

#추상화_Abstraction

객체의 복잡한 내부를 숨기고, 필수적인 부분만 외부에 노출하는 과정입니다. 사용자는 '어떻게'가 아닌 '무엇을'에 집중하게 됩니다.

- **예시**: 로또머신이 `getNumber()`를 제공하면 사용자는 내부에서 번호를 어떻게 뽑는지 몰라도 버튼만 누르면 번호를 받을 수 있습니다.

#상속_Inheritance

기존 클래스(부모)의 속성과 기능을 물려받아 새로운 클래스(자식)를 만드는 것입니다.

- **개념**: **일반화**와 **확장**의 개념을 합친 것입니다. 부모가 가진 공통된 속성이나 기능을 물려받아 `내 것`처럼 사용하며, 고유한 기능을 **확장**할 수 있습니다.
- **관계**: `is-a` 또는 `kind-of` 관계가 성립할 때만 사용해야 합니다. (예: `고양이`는 `동물`이다.) 논리적으로 어색한 관계(예: `비행기`가 `자동차`를 상속)에서는 사용하지 않습니다.
- **키워드**: `extends` 뒤에 상속받을 클래스명을 작성합니다.
- **장점**: 코드의 **재사용성**과 **확장성**을 높여 유지보수가 용이해집니다.

#상속에서_기억해야_할_3가지_중요한_규칙

1. **부모는 자식을 가리킬 수 있다 (다형성)**
   - `Parent p = new Child();` 와 같이 상위 타입의 변수가 하위 타입의 객체를 가리킬 수 있습니다.
   - `Child c = new Parent();` 는 불가능합니다.
2. **필드는 참조 변수의 타입을 따른다.**
   - 부모와 자식 클래스에 동일한 이름의 필드가 있을 때, **참조 변수의 타입**에 따라 접근하는 필드가 달라집니다.
3. **메서드는 오버라이드 되면 무조건 자식의 것이 사용된다.**
   - 부모의 메서드를 자식에서 재정의(오버라이딩)하면, 부모 타입 변수가 자식 객체를 가리키더라도 **실제로 호출되는 것은 자식의 메서드**입니다.

#서블릿
자바에서 웹의 요청을 받는 기술

#상속_추가정리
기존 클래스(부모)의 속성과 기능을 물려받아 새로운 클래스(자식)를 만드는 것입니다.

- **개념**: **일반화**와 **확장**의 개념을 합친 것. 부모가 가진 공통된 속성이나 기능을 물려받아 `내 것`처럼 사용하며, 고유한 기능을 **확장**할 수 있습니다.
- **관계**: `is-a` 또는 `kind-of` 관계가 성립할 때만 사용해야 합니다.

    - **올바른 예**: `고양이`는 `동물`이다.
    - **어색한 예**: `조류`와 `나비` (공통 속성이 있어도 관계가 어색함). `비행기`가 `자동차`를 상속받는 것은 불가능.
    - 코드가 중복되면 상속을 고려해 볼 필요가 있습니다.
- **목적**:
    - **유지보수와 확장성**: 코드를 재사용하여 중복을 막고, 한 곳(부모)에 물려놓고 다른 쪽(자식)에서 연결만 하는 것이 효율적입니다.
    - **메모리 절감**: 각 클래스마다 동일한 코드를 담는 것보다 상속을 통해 효율적으로 관리합니다.
- **단일 상속**: 자바는 **단일 상속**만을 지원합니다. 즉, 부모 클래스가 오직 하나여야 합니다. (다중 상속을 허용하는 언어도 있습니다.)
- **예시**: 자바의 `Thread` 클래스는 이미 상속 구조로 만들어져 있어, 개발자는 복잡한 멀티태스킹 체계를 따로 만들지 않고 상속받아 사용하면 됩니다.

상속의 3가지 핵심 규칙

1. **부모는 자식을 가리킬 수 있다 (다형성)**
    - `Parent1 p = new Child1();` (가능)
    - `Child1 c = new Parent1();` (오류)
    - **설명**: '새'는 `비둘기`를 포함하는 개념이지만, `비둘기`는 '새'에 포함되는 작은 개념이므로, 자식 타입 변수로 부모 객체를 가리킬 수 없습니다.
2. **필드는 참조 변수의 타입을 따른다.**
    - **개념**: 부모와 자식에 동일한 이름의 필드가 있을 때, 어떤 필드에 접근할지는 **참조 변수의 타입**에 따라 결정됩니다. (예: `Parent1 pp = new Parent1();`일 때 `pp.i`는 부모의 `10`을 출력)
    - **주의**: 필드 재정의는 메서드 오버라이딩과 다르게 동작합니다.
3. **메서드는 오버라이드 되면 무조건 자식의 것이 사용된다.**
    - **개념**: 부모 타입 변수가 자식 객체를 가리키고 있더라도, **실제 객체의 타입**인 자식의 메서드가 호출됩니다.

객체와*관련된*추가_개념

- **Object 클래스**: 자바의 모든 객체의 가장 큰 부모 클래스는 `Object`입니다.
- **`instanceof`**: 객체가 특정 타입인지 검사하는 키워드. 타입 캐스팅 시 오류를 방지하기 위해 사용됩니다.
- **형변환 (Type Casting)**:
    - **조건**: 기본형 데이터 타입(숫자, boolean)이나 **상속 관계**에 있는 객체들 사이에서만 형변환이 가능합니다.

    - **중요성**: 형변환 시 인스턴스(객체)의 실제 타입이 중요합니다. `p`의 인스턴스가 `Child`여야만 `Child`로 형변환할 수 있습니다. (`pp`처럼 인스턴스가 `Parent`인 경우에는 불가).
- **정보 은닉과 접근 제한자**:
    - **필드가 `private`인 경우**: 외부에서 직접 접근할 수 없으므로, **`getter`와 `setter` 메서드**를 통해 간접적으로 접근하게 만듭니다.

멀티태스킹과_스레드

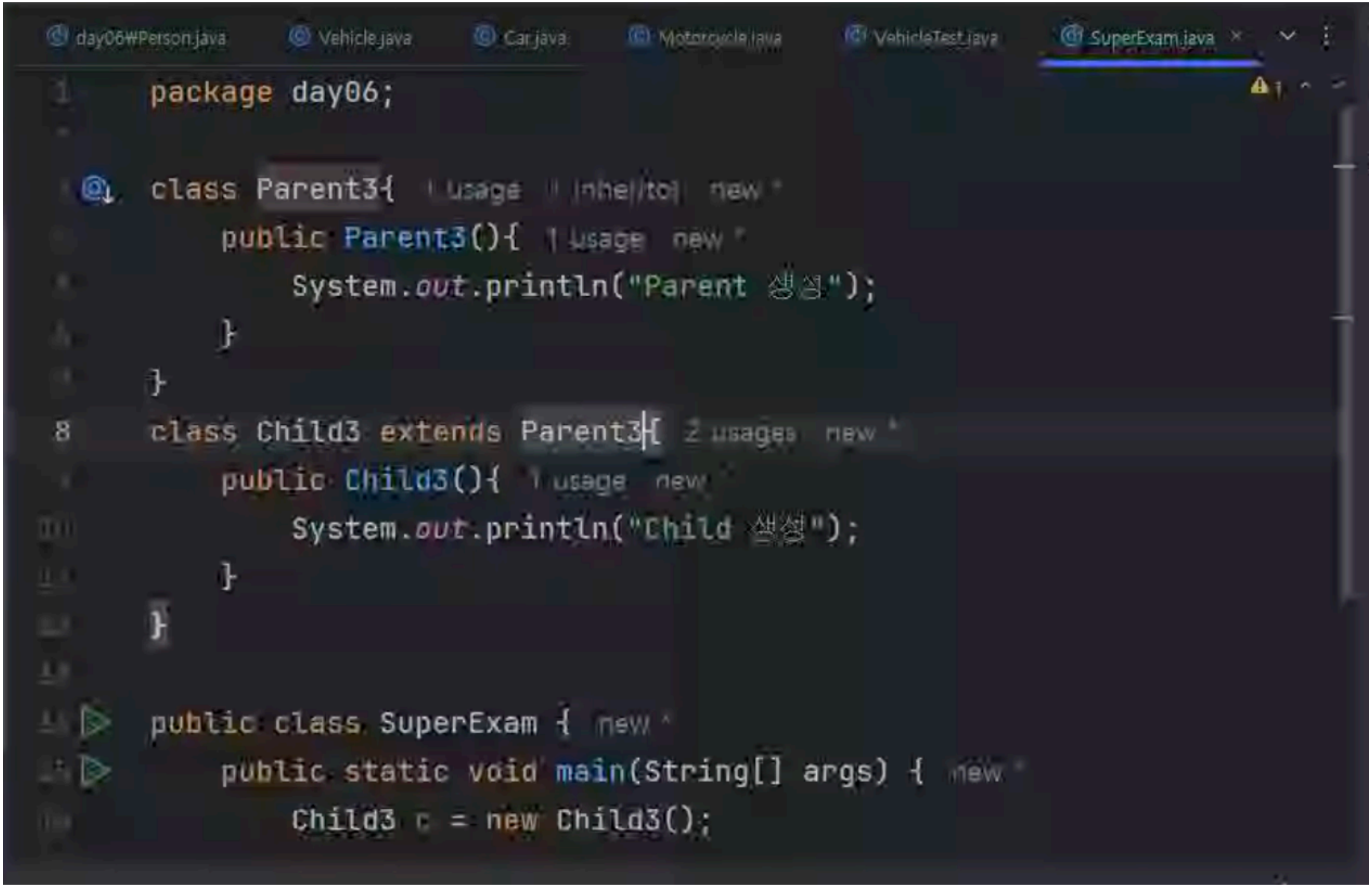
- **멀티태스킹**: 동시에 여러 일을 하는 것처럼 보이지만, 실제로는 빠르게 작업을 전환(컨텍스트 스위칭)하며 직렬로 수행하는 것입니다.
- **자바 스레드**: 자바의 스레드(`Thread`) 클래스는 멀티태스킹을 지원하는 핵심적인 기능으로, 상속을 통해 개발자가 쉽게 스레드를 구현할 수 있도록 도와줍니다.

`Mac의 IntelliJ에서 getter와 setter를 생성하는 단축키.`

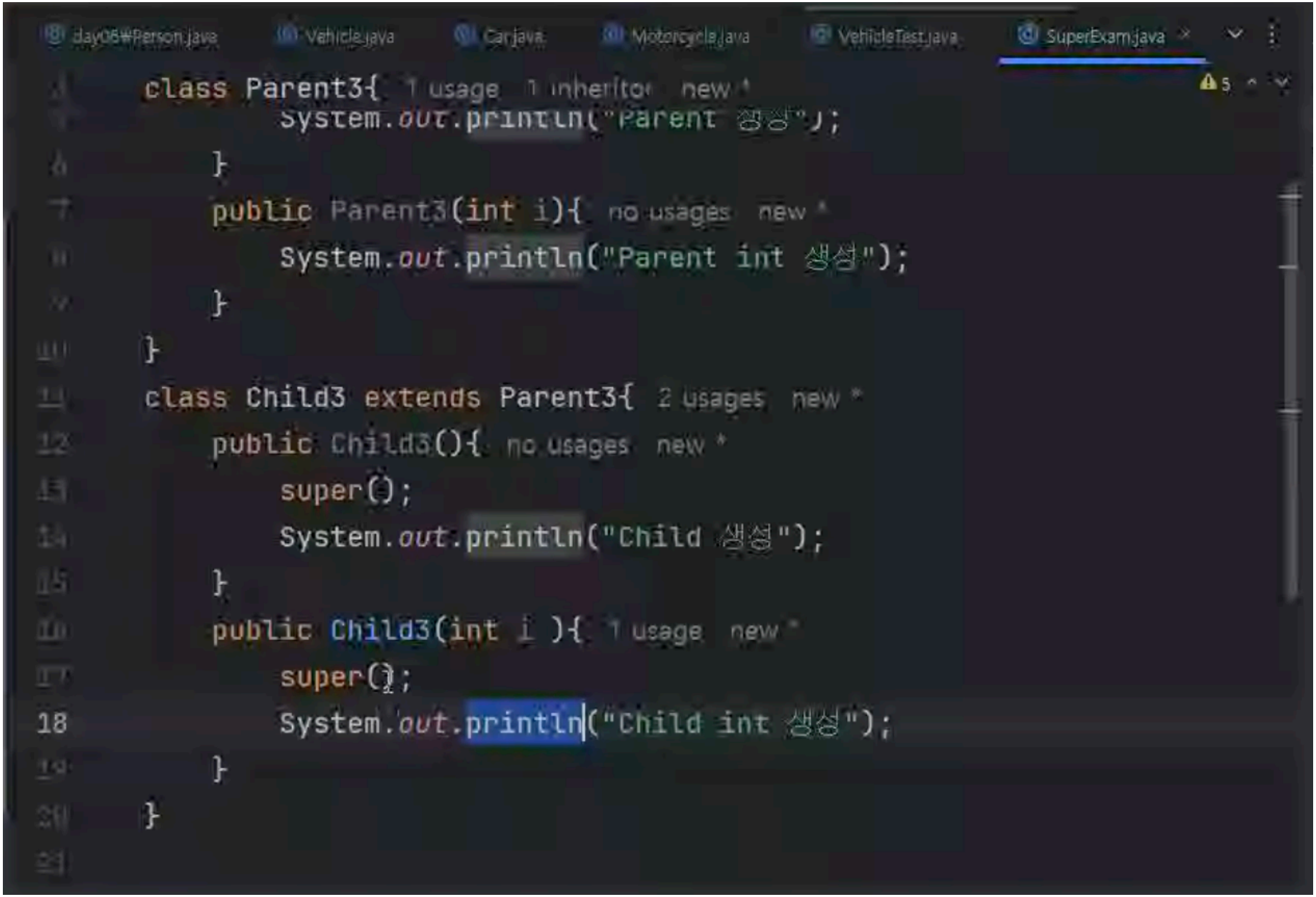
`⌘` + `N` (또는 **`^` + `Enter`**)

이 단축키를 누르면 "Generate" 메뉴가 나타나고, 여기서 **Getter**, **Setter**, 또는 **Getter and Setter**를 선택.

this 는 나 자신, super는 내 상속자


day06₩Person.java
Vehicle.java
Car.java
Motorcycle.java
VehicleTest.java
SuperExam.java
package day06;
class Parent3{ 1 usage no inheritors new *
public Parent3(){ 1 usage new *
System.out.println("Parent 생성");
}
}
class Child3 extends Parent3{ 2 usages new *
public Child3(){ 1 usage new *
System.out.println("Child 생성");
}
}
public class SuperExam { new *
public static void main(String[] args) { new *
Child3 c = new Child3();

ㅇㅇㅇ공백ㅇㅇㅇ



super는 생략되어있다. 명시 따로 안하면.
부모가 가진 필드에 접근을 할 수 있다. 자식것에서

super()하면 부모의 생성자에도 접근 가능

부모의 생성자가 실행되어야 나도 실행될 수 있다.

기본은 부모의 디폴트 생성자만 실행된다.
또다른 생성자를 통해서 객체를 초기화 하고싶을땐
명시적으로 써준다. 부모의 다른 생성자를 호출.
부모의 디폴트 생성자가 없을 경우 오류 발생
부모의 디폴트 생성자가 없는 경우 자식에서 부모의 생성자를 명시적으로 호출.
super(i:10);

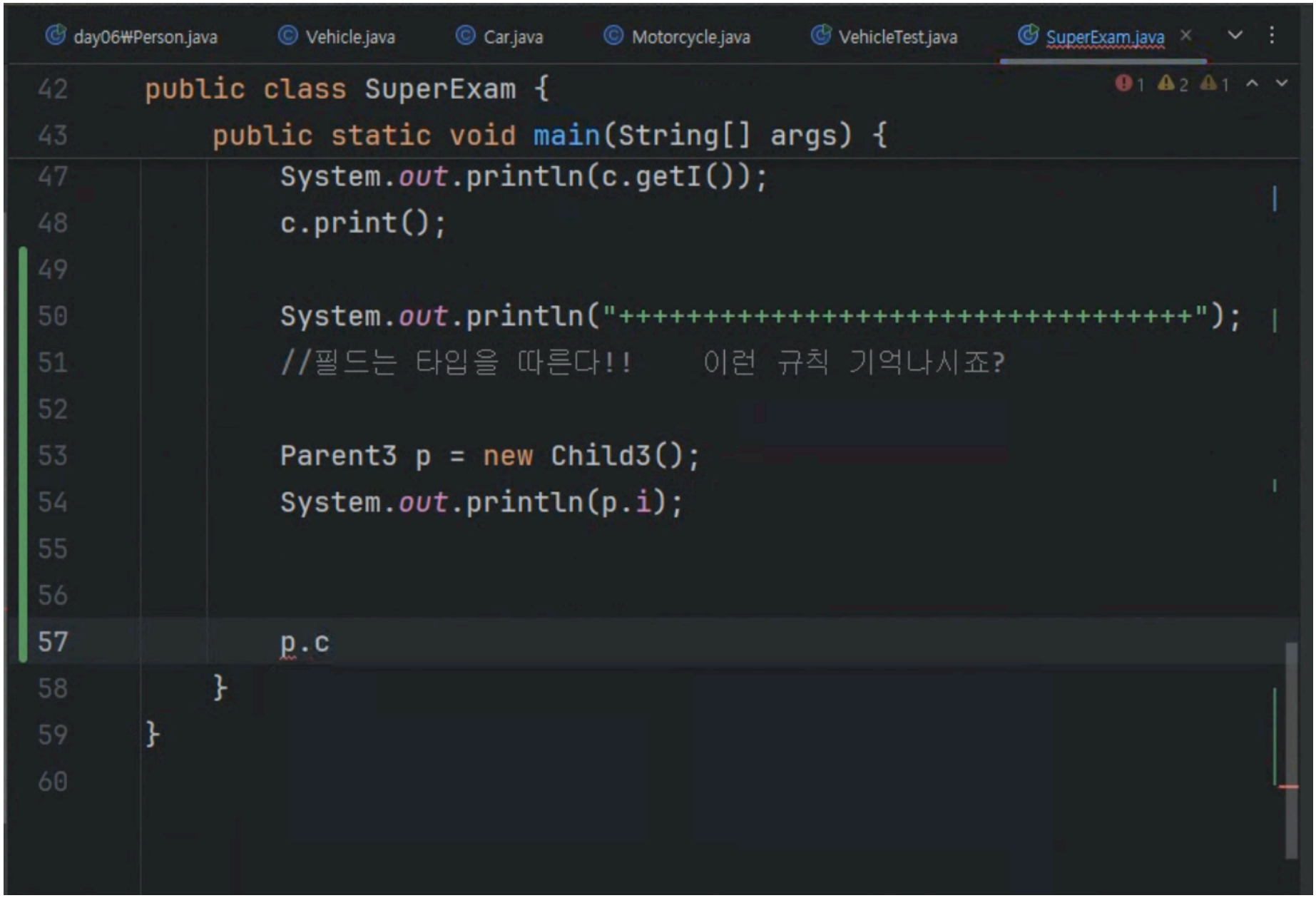
```
class Parent3{
    int i = 10;
    public void print(){
        System.out.println("parent : "+i);
    }
//    public Parent3(){
//        System.out.println("Parent 생성");
//    }
    public Parent3(int i){
        System.out.println("Parent int 생성");
    }
}
```

```
class Child3 extends Parent3{
    int i = 20;

    public void print(){
        super.print();
        System.out.println("Child : "+i);
    }
    public int getI(){
        return super.i;
//        return i;
    }
```

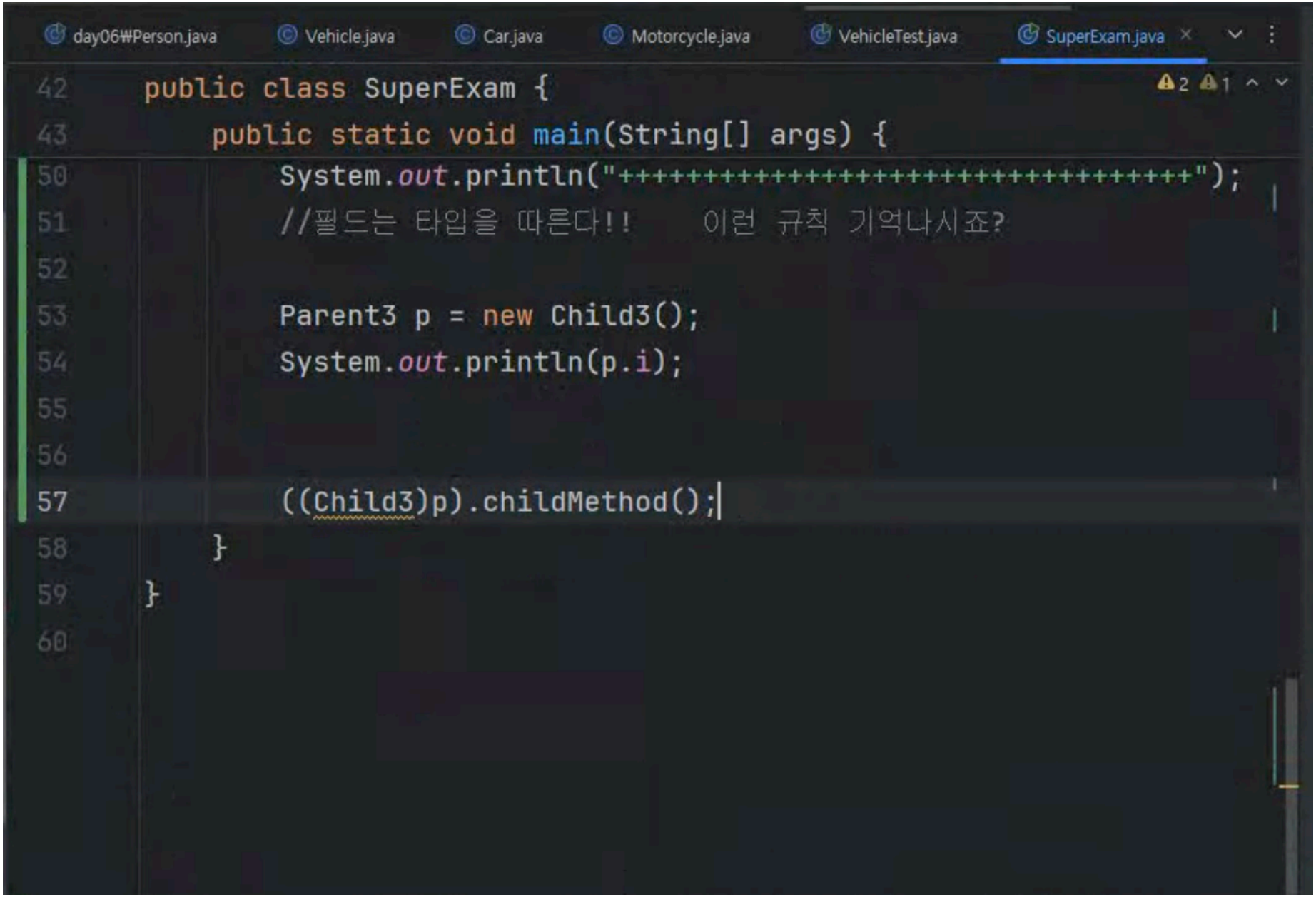
```
//    super - 부모의 인스턴스를 가리킴.
    public Child3(){
//        super();   생략가능
        super(10);   // 만약 부모가 디폴트생성자를 가지지 않는다면
        // 반드시 명시적으로 부모의 다른 생성자를 호출해줘야만함.
        System.out.println("Child 생성");
    }
    public Child3(int i ){
        super(i);  //명시적으로 호출 해야함
        System.out.println("Child int 생성");
    }
}
```

부모는 없는데 child가 메서드를 갖는다면?



부모는 물려준것까지만 안다.
인스턴스 있는데, 부모타입으로는 자식 개체에 접근이 안됨.(오버라이드 하면)
형변환 해줘야 한다.

부모는 본인이 물려준거 아니면 못쓴다.
자식이 새롭게 추가해놓은거에 대해서.
부모가 child로 형변환 해서 사용가능
인스턴스가 있음에도 불구하고



#상속과_참조_변수의_관계

핵심 개념: 부모 타입의 참조 변수는 자식 객체를 가리킬 수 있지만, 부모가 물려준 멤버(속성, 기능)만 접근할 수 있다.

1. 부모는 물려준 것까지만 안다.

- **설명**: 부모 타입의 변수로는 부모 클래스에 정의된 멤버(메서드, 필드)에만 접근할 수 있다.
- **결과**: 자식 클래스가 새롭게 추가한 멤버는 부모 타입의 변수로 접근할 수 없다. 컴파일러는 이를 알지 못하기 때문에 오류가 발생한다.

예시: `Parent p = new Child();`

- `p.parentMethod();` → **가능**
- `p.childMethod();` → **불가능 (컴파일 오류)**

2. 형변환을 통한 접근

- **설명**: 부모 타입 변수가 자식 객체를 가리키고 있을 때, 자식만의 고유한 멤버에 접근하려면 **명시적 형변환(다운캐스팅)**을 해주어야 한다.
- **조건**: 형변환하려는 객체가 실제로 그 타입의 인스턴스여야만 한다. `instanceof`로 확인하면 안전하다.

예시:

```
Parent p = new Child();

if (p instanceof Child) {
    Child c = (Child) p; // 형변환
    c.childMethod();     // 이제 자식의 메서드 접근 가능
}
```

3. 오버라이드된 메서드의 예외

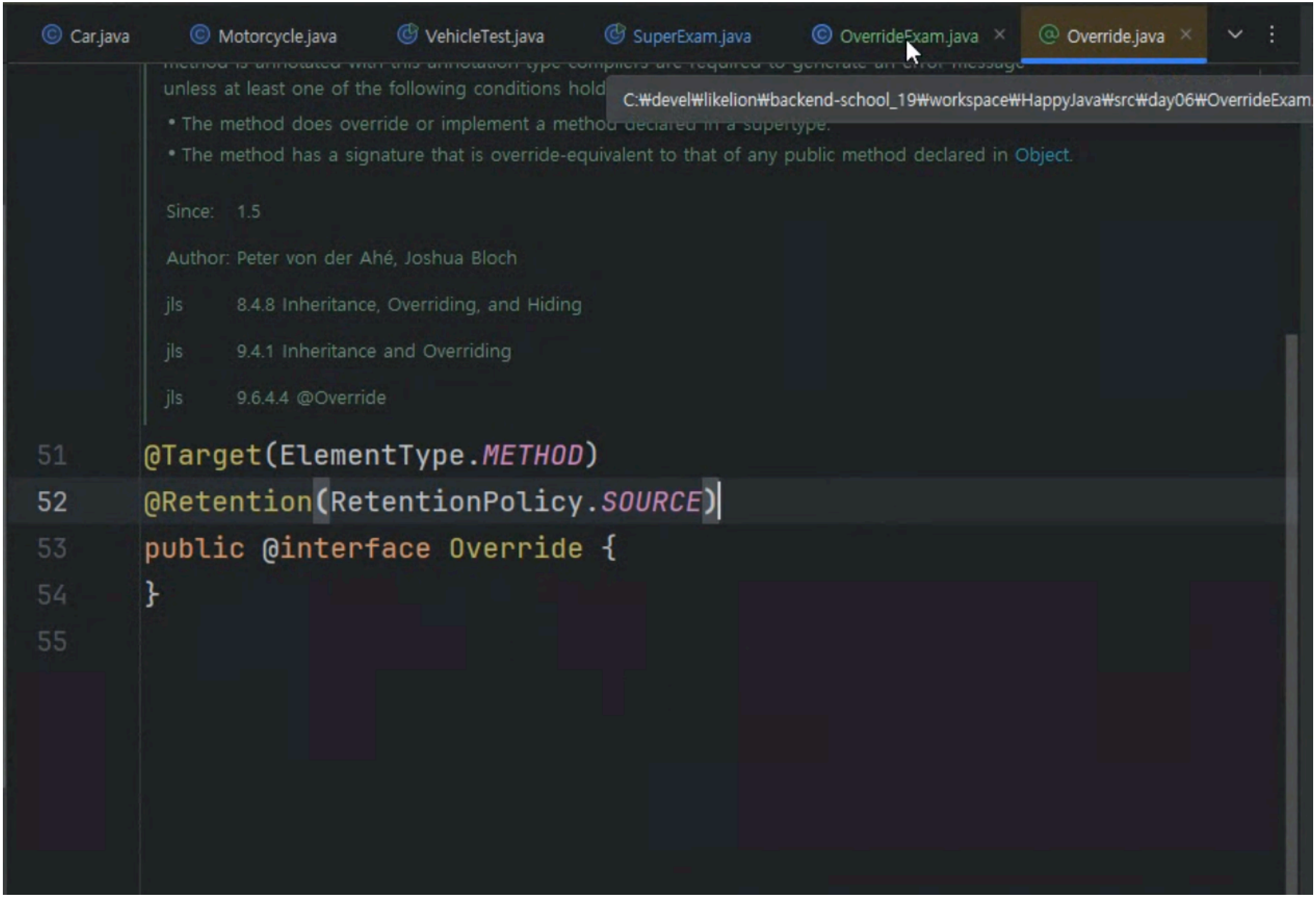
- **설명**: 필드와 달리, **오버라이드된 메서드**는 참조 변수의 타입과 상관없이 **실제 객체의 메서드가 호출**된다.
- **결과**: `Parent p = new Child();` 에서 `p.show()`를 호출하면, 자식의 `show()` 메서드가 실행된다.

요약:

- **필드(변수)**: 참조 변수의 타입을 따른다.
- **메서드**: 오버라이드되었다면 실제 객체의 타입을 따른다.
- **자식 고유 멤버**: 부모 타입 변수로는 접근 불가, 형변환이 필요하다.

#오버라이딩

- 부모가 가진것을 자식 클래스에서 똑같이 가졌지만, 재정의.



#_애노테이션 @

해당 태그를 해놓으면, override시 에러가 될만한 사항들을 IDE가 검출해준다.

#다형성

하나의 객체가 여러가지 형태를 가질 수 있다.
부모클래스 타입의 변수에 자식클래스 타입의 객체를 할당 가능

```
동물[] 동물들 = new 동물[2];
동물들[0] = new 개();
동물들[1] = new 고양이();

for (동물 a : 동물들) {
    a.울음소리(); // 각 객체에 맞는 울음소리가 출력됨
}
```

이런 다형성을 통해, 공통된 메서드 호출을 통해 각 객체의 고유한 행동을 실행시킬 수 있으며, 새로운 자식 클래스가 추가되더라도 기존 코드를 수정할 필요가 없습니다.

다음 진도
#추상클래스

도은님 회고시간 팁 정리 -- 차후 배우겠지만, 미리 찍먹해보기

`new`가 `heap`에 할당해주는것임.

객체를 생성했는데 거기에 할당을 안해주면, 안됨. 공간이 필요한데
객체가 필요한 공간을 `new`로 요청하는 것임.

`new`가 있으면 생성자
`new`는 키워드.

#업캐스팅과 #다운캐스팅 은
상속에서 나온다.

#getter #setter 어떤 값을 가져온다.
겟은 가져오고 셋은 값을 집어넣는다. 값을 할당해준다.
프론트에서 데이터를 그냥 쏴주지 않는다.
"암호화" 해서 쏴주는데 패키징을 겟이나 셋으로 서로 공유할때 필요한것.
게터부분 세터부분을 표시하고, 그 표시를 해주는게 #어노테이션 @

으로 감춘후에 겟이랑 세터로 접근

`private`은 보안이라기보다 접근자만 제어. `캡슐화`가 보안.
`private`은 접근성.
이쪽 디렉토리에서만 쓰겠다.
`public`으로 하면 접근권한 자체가 많아진다.

프론트단에서 게터세터 정보공유
xxx게터 , 세터로 정보공유시 필요

나중에 스프링을 쓰면, 자바보다는 스프링을 쓰는 느낌.
자바문법이 있으나, 어노테이션이랑 함수 위주.
자바로 문제푸는것과 느낌이 다르다.

반복문 조건문보다 함수 호출하는게 더 중요

라이브러리 외우고, 네이밍 방식을 외우는게 중요.
로직을 구성하고 이해하는게 가장 중요.
-> 로직 중요

이 클래스에서 저 클래스 어떻게 생겼는데... -> 결국, 읽어내는 눈이 필요.
