Chapter 06. 클래스

6.1 객체 지향 프로그래밍

객체(object)란?

- 물리적으로 존재하거나 개념적인 것 중에서 다른 것과 식별 가능한 것
- 속성(**필드, field**)와 동작(**메소드, method**)으로 구성

객체 간의 관계

- 집합 관계
  - 완성품과 부품의 관계
  - 예) 자동차와 부품(엔진, 타이어, 핸들 등)의 관계
- 사용 관계
  - 다른 객체의 필드를 읽고 변경하거나 메소드를 호출하는 관계
  - 예) 사람과 자동차의 관계 (자동차에게 달린다, 멈춘다 등의 메소드 호출)
- 상속 관계
  - 부모와 자식 관계
  - 기계와 자동차의 관계

객체 지향 프로그래밍의 특징

- 캡슐화 (Encapsulation)
  - 객체의 데이터(필드), 동작(메소드)을 하나로 묶고 실제 구현 내용을 외부에 감추는 것
  - 외부 객체는 객체 내부의 구조 알지 못하며 객체가 노출해서 제공하는 필드와 메소드만 이용 가능
  - 외부의 잘못된 사용으로 인해 객체가 손상되지 않도록 하기 위해 사용
- 상속 (Inheritance)
  - 부모 객체가 자기가 가지고 있는 필드와 메소드를 자식 객체에게 물려주어 사용할 수 있도록 하는 것
  - 코드의 재사용성 높여줌
  - 유지 보수 시간을 최소화시켜 줌
- 다형성 (Polymorphism)
  - 사용 방법은 동일하지만 실행 결과가 다양하게 나오는 성질
  - 자동 타입 변환과 재정의 기술이 필요

6.2 객체와 클래스

- 객체 지향 프로그래밍에서 객체를 생성하려면 설계도에 해당하는 클래스(class)가 필요
- 클래스로부터 생성된 객체 : 인스턴스(Instance)
- 클래스로부터 객체를 만드는 과정 : 인스턴스화

6.3 클래스 선언

- 객체 생성을 위한 설계도를 작성하는 작업

```
package ch06.sec03;

public class SportsCar {
}

class Tire {
}
```

6.4 객체 생성과 클래스 변수

Student.java

```
package ch06.sec04;

public class Student {
}
```

StudentExample.java

```
package ch06.sec04;

public class StudentExample {
    public static void main(String[] args) {
        Student s1 = new Student();
        System.out.println("s1 변수가 Student 객체를 참조합니다.");

        Student s2 = new Student();
        System.out.println("s2 변수가 또 다른 Student 객체를 참조합니다.");
    }
}
```

6.5 클래스의 구성 멤버

- 필드(Field)
  - 객체의 데이터를 저장하는 역할
- 생성자(Constructor)
  - 객체의 초기화 역할
  - 리턴 타입이 없고 이름은 클래스 이름과 동일
- 메소드(Method)
  - 객체가 수행할 동작
  - 객체와 객체간의 상호 작용을 위해 호출됨

6.6 필드 선언과 사용

필드 선언

Car.java

```
package ch06.sec06.exam01;

public class Car {
    //필드 선언
    String model;
    boolean start;
    int speed;
}
```

CarExample.java

```
package ch06.sec06.exam01;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

        //Car 객체의 필드값 읽기
        System.out.println("모델명: " + myCar.model);
        System.out.println("시동여부: " + myCar.start);
        System.out.println("현재속도: " + myCar.speed);
    }
}
```

필드 사용

- 필드값을 읽고 변경

Car.java

```
package ch06.sec06.exam02;

public class Car {
    //필드 선언
    String company = "현대자동차";
    String model = "그랜저";
    String color = "검정";
    int maxSpeed = 350;
    int speed;
}
```

CarExample.java

```
package ch06.sec06.exam02;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

        //Car 객체의 필드값 읽기
        System.out.println("제작회사: " + myCar.company);
        System.out.println("모델명: " + myCar.model);
        System.out.println("색깔: " + myCar.color);
        System.out.println("최고속도: " + myCar.maxSpeed);
        System.out.println("현재속도: " + myCar.speed);

        //Car 객체의 필드값 변경
        myCar.speed = 60;
        System.out.println("수정된 속도: " + myCar.speed);
    }
}
```

6.7 생성자 선언과 호출

기본 생성자

- 클래스에 생성자 선언이 없으면 컴파일러는 기본 생성자(Default Constructor)를 바이트코드 파일에 자동 추가

생성자 선언

- 객체를 다양하게 초기화하기 위해 개발자가 생성자 직접 선언 가능
- 리턴 타입이 없고 클래스 이름과 동일

Car.java

```
package ch06.sec07.exam01;

public class Car {
    //생성자 선언
    Car(String model, String color, int maxSpeed) {
    }
}
```

CarExample.java

```
package ch06.sec07.exam01;
```

```
public class CarExample {
    public static void main(String[] args) {
        Car myCar = new Car("그랜저", "검정", 250);
        //Car myCar = new Car();  //기본 생성자 호출 못함
    }
}
```

필드 초기화

Korean.java

```
package ch06.sec07.exam03;

public class Korean {
    // 필드 선언
    String nation = "대한민국";
    String name;
    String ssn;

    // 생성자 선언
    public Korean(String name, String ssn) {
        this.name = name;
        this.ssn = ssn;
    }
}
```

KoreanExample.java

```
package ch06.sec07.exam02;

public class KoreanExample {
    public static void main(String[] args) {
        //Korean 객체 생성
        Korean k1 = new Korean("박자바", "011225-1234567");
        //Korean 객체 데이터 읽기
        System.out.println("k1.nation : " + k1.nation);
        System.out.println("k1.name : " + k1.name);
        System.out.println("k1.ssn : " + k1.ssn);
        System.out.println();

        //또 다른 Korean 객체 생성
        Korean k2 = new Korean("김자바", "930525-0654321");
        //또 다른 Korean 객체 데이터 읽기
        System.out.println("k2.nation : " + k2.nation);
        System.out.println("k2.name : " + k2.name);
        System.out.println("k2.ssn : " + k2.ssn);
    }
}
```

생성자 오버로딩

Car.java

```
package ch06.sec07.exam04;

public class Car {
    //필드 선언
    String company = "현대자동차";
    String model;
    String color;
    int maxSpeed;

    //생성자 선언
    Car() {}

    Car(String model) {
        this.model = model;
    }

    Car(String model, String color) {
        this.model = model;
        this.color = color;
    }

    Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }
}
```

CarExample.java

```
package ch06.sec07.exam04;

public class CarExample {
    public static void main(String[] args) {
        Car car1 = new Car();
        System.out.println("car1.company : " + car1.company);
        System.out.println();

        Car car2 = new Car("자가용");
        System.out.println("car2.company : " + car2.company);
        System.out.println("car2.model : " + car2.model);
        System.out.println();

        Car car3 = new Car("자가용", "빨강");
        System.out.println("car3.company : " + car3.company);
        System.out.println("car3.model : " + car3.model);
```

```
        System.out.println("car3.color : " + car3.color);
        System.out.println();

        Car car4 = new Car("택시", "검정", 200);
        System.out.println("car4.company : " + car4.company);
        System.out.println("car4.model : " + car4.model);
        System.out.println("car4.color : " + car4.color);
        System.out.println("car4.maxSpeed : " + car4.maxSpeed);
    }
}
```

다른 생성자 호출

생성자 오버로딩이 많아질 경우 생성자 간의 중복된 코드가 발생할 수 있는데, 이 경우 공통 코드를 한 생성자에만 집중적으로 작성하고, 나머지 생성자는 `this(...)`를 사용하여 공통 코드를 가지고 있는 생성자를 호출하는 방법으로 개선 가능하다. 이는 생성자의 첫 줄에 작성되며 다른 생성자를 호출하는 역할을 하고 호출되는 생성자의 실행이 끝나면 원래 생성자로 돌아와서 다음 실행문을 실행한다.

Car.java

```
package ch06.sec07.exam05;

public class Car {
    // 필드 선언
    String company = "현대자동차";
    String model;
    String color;
    int maxSpeed;

    Car(String model) {
        //세번째 생성자 호출
        this(model, "은색", 250);
    }

    Car(String model, String color) {
        //세번째 생성자 호출
        this(model, color, 250);
    }

    Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }
}
```

6.8 메소드 선언과 호출

메소드 선언

```
package ch06.sec08.exam01;

public class Calculator {
    //리턴값이 없는 메소드 선언
    void powerOn() {
        System.out.println("전원을 켭니다.");
    }

    //리턴값이 없는 메소드 선언
    void powerOff() {
        System.out.println("전원을 끕니다.");
    }

    //호출 시 두 정수 값을 전달받고,
    //호출한 곳으로 결과값 int를 리턴하는 메소드 선언
    int plus(int x, int y) {
        int result = x + y;
        return result; //리턴값 지정;
    }

    //호출 시 두 정수 값을 전달받고,
    //호출한 곳으로 결과값 double을 리턴하는 메소드 선언
    double divide(int x, int y) {
        double result = (double)x / (double)y;
        return result; //리턴값 지정;
    }
}
```

메소드 호출

```
package ch06.sec08.exam01;

public class CalculatorExample {
    public static void main(String[] args) {
        //Calculator 객체 생성
        Calculator myCalc = new Calculator();

        //리턴값이 없는 powerOn() 메소드 호출
        myCalc.powerOn();

        //plus 메소드 호출 시 5와 6을 매개값으로 제공하고,
        //덧셈 결과를 리턴 받아 result1 변수에 대입
        int result1 = myCalc.plus(5, 6);
        System.out.println("result1: " + result1);

        int x = 10;
        int y = 4;
        //divide() 메소드 호출 시 변수 x와 y의 값을 매개값으로 제공하고,
```

```
        //나눗셈 결과를 리턴 받아 result2 변수에 대입
        double result2 = myCalc.divide(x, y);
        System.out.println("result2: " + result2);

        //리턴값이 없는 powerOff() 메소드 호출
        myCalc.powerOff();
    }
}
```

가변길이 매개변수

- 매개변수의 개수와 상관없이 매개값을 줄 수 있음
- 메소드 호출 시 매개값을 쉼표로 구분해서 개수와 상관없이 제공할 수 있음
- 매개값들은 자동으로 배열 항목으로 변환되어 메소드에서 사용

Computer.java

```
package ch06.sec08.exam02;

public class Computer {
    //가변길이 매개변수를 갖는 메소드 선언
    int sum(int ... values) {
        //sum 변수 선언
        int sum = 0;

        //values는 배열 타입의 변수처럼 사용
        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }

        //합산 결과를 리턴
        return sum;
    }
}
```

ComputerExample.java

```
package ch06.sec08.exam02;

public class ComputerExample {
    public static void main(String[] args) {
        //Computer 객체 생성
        Computer myCom = new Computer();

        //sum() 메소드 호출 시 매개값 1, 2, 3을 제공하고
        //합산 결과를 리턴 받아 result1 변수에 대입
        int result1 = myCom.sum(1, 2, 3);
        System.out.println("result1: " + result1);
```

```
        //sum() 메소드 호출 시 매개값 1, 2, 3, 4, 5를 제공하고
        //합산 결과를 리턴 받아 result2 변수에 대입
        int result2 = myCom.sum(1, 2, 3, 4, 5);
        System.out.println("result2: " + result2);

        //sum() 메소드 호출 시 배열을 제공하고
        //합산 결과를 리턴 받아 result3 변수에 대입
        int[] values = { 1, 2, 3, 4, 5 };
        int result3 = myCom.sum(values);
        System.out.println("result3: " + result3);

        //sum() 메소드 호출 시 배열을 제공하고
        //합산 결과를 리턴 받아 result4 변수에 대입
        int result4 = myCom.sum(new int[] { 1, 2, 3, 4, 5 });
        System.out.println("result4: " + result4);
    }
}
```

return 문

- 메소드의 실행을 강제 종료하고 호출한 곳으로 돌아간다는 의미

Car.java

```
package ch06.sec08.exam03;

public class Car {
    //필드 선언
    int gas;

    //리턴값이 없는 메소드로 매개값을 받아서 gas 필드값을 변경
    void setGas(int gas) {
        this.gas = gas;
    }

    //리턴값이 boolean인 메소드로 gas 필드값이 0이면 false를, 0이 아니면 true를 리턴
    boolean isLeftGas() {
        if (gas == 0) {
            System.out.println("gas가 없습니다.");
            return false; // false를 리턴하고 메소드 종료
        }
        System.out.println("gas가 있습니다.");
        return true; // true를 리턴하고 메소드 종료
    }

    //리턴값이 없는 메소드로 gas 필드값이 0이면 return 문으로 메소드를 종료
    void run() {
        while (true) {
            if (gas > 0) {
                System.out.println("달립니다.(gas잔량:" + gas + ")");
                gas -= 1;
```

```
            } else {
                System.out.println("멈춥니다.(gas잔량:" + gas + ")");
                return; // 메소드 종료
            }
        }
    }
}
```

CarExample.java

```
package ch06.sec08.exam03;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

        //리턴값이 없는 setGas() 메소드 호출
        myCar.setGas(5);

        //isLeftGas() 메소드를 호출해서 받은 리턴값이 true일 경우 if 블록 실행
        if(myCar.isLeftGas()) {
            System.out.println("출발합니다.");

            //리턴값이 없는 run() 메소드 호출
            myCar.run();
        }

        System.out.println("gas를 주입하세요.");
    }
}
```

결과

```
gas가 있습니다.
출발합니다.
달립니다.(gas잔량:5)
달립니다.(gas잔량:4)
달립니다.(gas잔량:3)
달립니다.(gas잔량:2)
달립니다.(gas잔량:1)
멈춥니다.(gas잔량:0)
gas를 주입하세요.
```

메소드 오버로딩

- 메소드 이름은 같되 매개변수의 타입, 개수, 순서가 다른 메소드를 여러 개 선언하는 것
- 다양한 매개값을 처리하기 위해 사용

- 대표적인 예 : `System.out.println()` 메소드

Calculator.java

```
package ch06.sec08.exam04;

public class Calculator {
    //정사각형의 넓이
    double areaRectangle(double width) {
        return width * width;
    }

    //직사각형의 넓이
    double areaRectangle(double width, double height) {
        return width * height;
    }
}
```

CalculatorExample.java

```
package ch06.sec08.exam04;

public class CalculatorExample {
    public static void main(String[] args) {
        //객체 생성
        Calculator myCalcu = new Calculator();

        //정사각형의 넓이 구하기
        double result1 = myCalcu.areaRectangle(10);

        //직사각형의 넓이 구하기
        double result2 = myCalcu.areaRectangle(10, 20);

        System.out.println("정사각형 넓이=" + result1);
        System.out.println("직사각형 넓이=" + result2);
    }
}
```

구분	설명
인스턴스(instance) 멤버	객체에 소속된 멤버(객체를 생성해야만 사용할 수 있는 멤버)
정적(static) 멤버	클래스에 고정된 멤버(객체 없이도 사용할 수 있는 멤버)

6.9 인스턴스 멤버

Car.java

```java
package ch06.sec09;

public class Car {
    //필드 선언
    String model;
    int speed;

    //생성자 선언
    Car(String model) {
        this.model = model; //매개변수를 필드에 대입(this 생략 불가)
    }

    //메소드 선언
    void setSpeed(int speed) {
        this.speed = speed; //매개변수를 필드에 대입(this 생략 불가)
    }

    void run() {
        this.setSpeed(100);
        System.out.println(this.model + "가 달립니다.(시속:" + this.speed +
"km/h)");
    }   // this 전부 생략 가능
}
```

CarExample.java

```java
package ch06.sec09;

public class CarExample {
    public static void main(String[] args) {
        Car myCar = new Car("포르쉐");
        Car yourCar = new Car("벤츠");

        myCar.run();
        yourCar.run();
    }
}
```

- Car 클래스의 필드와 메소드는 인스턴스 멤버이므로 외부 클래스에서 사용하기 위해서는 객체를 먼저 생성하고 참조 변수로 접근해서 사용해야 함
- 필드는 객체마다 따로 존재하고 메소드는 각 객체마다 존재하지 않고 메소드 영역에 저장되고 공유됨
- 객체 내부에서 인스턴스 멤버에 접근하기 위해 `this` 사용

6.10 정적 멤버

자바는 클래스 로더(loader)를 이용해서 클래스를 메소드 영역에 저장하고 사용. 정적(static) 멤버란 메소드 영역의 클래스에 고정적으로 위치하는 멤버를 말한다. 그렇기에 정적 멤버는 객체를 생성할 필요 없이 클래스를

통해서 바로 사용이 가능하다.
인스턴스 필드를 이용하지 않는 메소드는 정적 메소드로 선언하는 것이 좋다.

Calculator.java

```
package ch06.sec10.exam01;

public class Calculator {
    static double pi = 3.14159;

    static int plus(int x, int y) {
        return x + y;
    }

    static int minus(int x, int y) {
        return x - y;
    }
}
```

CalculatorExample.java

```
package ch06.sec10.exam01;

public class CalculatorExample {
    public static void main(String[] args) {
        double result1 = 10 * 10 * Calculator.pi;
        int result2 = Calculator.plus(10, 5);
        int result3 = Calculator.minus(10, 5);

        System.out.println("result1 : " + result1);
        System.out.println("result2 : " + result2);
        System.out.println("result3 : " + result3);
    }
}
```

정적 블록

정적 필드는 필드 선언과 동시에 초기값을 주는 것이 일반적이지만 복잡한 초기화 작업이 필요하다면 정적 블록(static block)을 이용해야 한다. 정적 블록은 클래스가 메모리로 로딩될 때 자동으로 실행한다.

Television.java

```
package ch06.sec10.exam02;

public class Television {
    static String company = "MyCompany";
    static String model = "LCD";
    static String info;
```

```
    static {
        info = company + "-" + model;
        }
}
```

TelevisionExample.java

```
package ch06.sec10.exam02;

public class TelevisionExample {
    public static void main(String[] args) {
        System.out.println(Television.info);
        }
}
```

인스턴스 멤버 사용 불가

정적 메소드와 정적 블록은 객체가 없어도 실행된다는 특징 때문에 내부에 인스턴스 필드나 인스턴스 메소드를 사용할 수 없다. 또한 객체 자신의 참조인 `this`도 사용할 수 없다. 정적 메소드와 정적 블록에서 인스턴스 멤버를 사용하고 싶다면 객체를 먼저 생성하고 참조 변수로 접근해야 한다.

`main()` 메소드 또한 정적 메소드이므로 객체 생성 없이 인스턴스 필드와 인스턴스 메소드를 `main()` 메소드에서 바로 사용할 수 없다.

Car.java

```
package ch06.sec10.exam03;

public class Car {
    //인스턴스 필드 선언
    int speed;

    //인스턴스 메소드 선언
    void run() {
        System.out.println(speed + "으로 달립니다.");
    }

    static void simulate() {
        //객체 생성
        Car myCar = new Car();
        //인스턴스 멤버 사용
        myCar.speed = 200;
        myCar.run();
    }

    public static void main(String[] args) {
        //정적 메소드 호출
        simulate();
```

```
        //객체 생성
        Car myCar = new Car();
        //인스턴스 멤버 사용
        myCar.speed = 60;
        myCar.run();
    }
}
```

6.11 final 필드와 상수

- 경우에 따라 값을 변경하는 것을 막고 읽기만 허용해야 할 때 사용

final 필드 선언

1. 필드 선언 시에 초기값 대입
2. 생성자에서 초기값 대입

Korean.java

```
package ch06.sec11.exam01;

public class Korean {
    //인스턴스 final 필드 선언
    final String nation = "대한민국";
    final String ssn;

    //인스턴스 필드 선언
    String name;

    //생성자 선언
    public Korean(String ssn, String name) {
        this.ssn = ssn;
        this.name = name;
    }
}
```

KoreanExample.java

```
package ch06.sec11.exam01;

public class KoreanExample {
    public static void main(String[] args) {
        //객체 생성 시 주민등록번호와 이름 전달
        Korean k1 = new Korean("123456-1234567", "감자바");

        //필드값 읽기
        System.out.println(k1.nation);
```

```
        System.out.println(k1.ssn);
        System.out.println(k1.name);

        //Final 필드는 값을 변경할 수 없음
        //k1.nation = "USA";
        //k1.ssn = "123-12-1234";

        //비 final 필드는 값 변경 가능
        k1.name = "김자바";
    }
}
```

- nation은 고정값이므로 선언 시에 초기값 대입
- ssn은 Korean 객체가 생성될 때 부여되므로 생성자 매개값으로 받아 초기값 대입

상수(constant) 선언

- 원주율 파이나 지구의 무게 및 둘레 등 불변의 값을 저장하는 필드
- `static`이면서 `final`인 특성
- 초기값은 선언 시 혹은 정적 블록에서
- 모두 대문자로 작성, 언더바로 단어들 연결

Earth.java

```
package ch06.sec11.exam02;

public class Earth {
    //상수 선언 및 초기화
    static final double EARTH_RADIUS = 6400;

    //상수 선언
    static final double EARTH_SURFACE_AREA;

    //정적 블록에서 상수 초기화
    static {
        EARTH_SURFACE_AREA = 4 * Math.PI * EARTH_RADIUS * EARTH_RADIUS;
    }
}
```

EarthExample.java

```
package ch06.sec11.exam02;

public class EarthExample {
    public static void main(String[] args) {
        //상수 읽기
        System.out.println("지구의 반지름: " + Earth.EARTH_RADIUS + "km");
        System.out.println("지구의 표면적: " + Earth.EARTH_SURFACE_AREA + "km^2");
```

```
    }
}
```

6.12 패키지

- 클래스의 일부분이며, 클래스를 식별하는 용도로 사용됨
- 주로 개발 회사의 도메인 이름의 역순으로 만듬
- 상위 패키지와 하위 패키지를 도트(.)로 구분
- 패키지에 속한 바이트코드 파일(~.class)은 따로 떼어내어 다른 디렉토리로 이동할 수 없음

패키지 선언

- 패키지 디렉토리는 클래스를 컴파일하는 과정에서 자동으로 생성
- 항상 소스 파일 최상단 위치
- 패키지 이름은 모두 소문자로 작성
- 중복되지 않도록 회사 도메인 이름의 역순으로 작성하고 마지막에는 프로젝트 이름을 붙여주는 것이 일반적

import 문

- 다른 패키지에 있는 클래스를 사용하려면 import 문을 이용해서 어떤 패키지의 클래스를 사용하는지 명시
- 하위 패키지를 포함하지 않음

hankook-SnowTire.java

```
package ch06.sec12.hankook;

public class SnowTire {
}
```

hankook-Tire.java

```
package ch06.sec12.hankook;

public class Tire {
}
```

kumho-AllSeasonTire.java

```
package ch06.sec12.kumho;

public class AllSeasonTire {
}
```

kumho-Tire.java

```
package ch06.sec12.kumho;

public class Tire {
}
```

hyundai-Car.java

```
package ch06.sec12.hyundai;

//import 문으로 다른 패키지 클래스 사용을 명시
import ch06.sec12.hankook.SnowTire;
import ch06.sec12.kumho.AllSeasonTire;

public class Car {
    //부품 필드 선언
    ch06.sec12.hankook.Tire tire1 = new ch06.sec12.hankook.Tire();
    ch06.sec12.kumho.Tire tire2 = new ch06.sec12.kumho.Tire();
    SnowTire tire3 = new SnowTire();
    AllSeasonTire tire4 = new AllSeasonTire();
}
```

6.13 접근 제한자

중요한 필드와 메소드가 외부로 노출되지 않도록 해 객체의 무결성 (결점이 없는 성질)을 유지하기 위해 경우에 따라 객체의 필드를 외부에서 변경하거나 메소드를 호출할 수 없도록 막아야 할 필요가 있다. 이를 위해 **접근 제한자 (Access Modifier)** 를 사용한다.

접근 제한자	제한 대상	제한 범위
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	같은 패키지이거나, 자식 객체만 사용 가능
(default)	클래스, 필드, 생성자, 메소드	같은 패키지
private	필드, 생성자, 메소드	객체 내부

package1 - A.java

```
package ch06.sec13.exam03.package1;

public class A {
    //public 접근 제한을 갖는 필드 선언
    public int field1;
```

```
    //default 접근 제한을 갖는 필드 선언
    int field2;
    //private 접근 제한을 갖는 필드 선언
    private int field3;

    //생성자 선언
    public A() {
        field1 = 1;         //o
        field2 = 1;         //o
        field3 = 1;         //o

        method1();          //o
        method2();          //o
        method3();          //o
    }

    //public 접근 제한을 갖는 메소드 선언
    public void method1() {
    }

    //default 접근 제한을 갖는 메소드 선언
    void method2() {
    }

    //private 접근 제한을 갖는 메소드 선언
    private void method3() {
    }
}
```

package1 - B.java

```
package ch06.sec13.exam03.package1;

public class B {
    public void method() {
        //객체 생성
        A a = new A();

        //필드값 변경
        a.field1 = 1;       // o
        a.field2 = 1;       // o
        //a.field3 = 1;     // x

        //메소드 호출
        a.method1();        // o
        a.method2();        // o
        //a.method3();      // x
    }
}
```

package2 - C.java

```
package ch06.sec13.exam03.package2;

import ch06.sec13.exam03.package1.*;

public class C {
    public C() {
        //객체 생성
        A a = new A();

        //필드값 변경
        a.field1 = 1;       // (o)
        //a.field2 = 1;     // (x)
        //a.field3 = 1;     // (x)

        //메소드 호출
        a.method1();        // (o)
        //a.method2();      // (x)
        //a.method3();      // (x)
    }
}
```

6.14 Getter와 Setter

객체의 필드(데이터)를 외부에서 마음대로 읽고 변경할 경우 객체의 무결성(결점이 없는 성질)이 깨질 수 있다. 객체 지향 프로그래밍에서는 직접적인 외부에서의 필드 접근을 막고 대신 메소드를 통해 필드에 접근하는 것을 선호하는데, 메소드는 데이터를 검증해서 유효한 값만 필드에 저장할 수 있기 때문이다. 이러한 역할을 하는 메소드가 **Setter**이다.

마찬가지로 외부에서 객체의 필드를 읽을 때에도 메소드가 필요한 경우가 있다. 필드값이 객체 외부에서 사용하기에 부적절한 경우, 메소드로 적절한 값으로 변환해서 리턴할 수 있기 때문이다. 이러한 역할을 하는 메소드가 **Getter**이다.

Car.java

```
package ch06.sec14;

public class Car {
    //필드 선언
    private int speed;
    private boolean stop;

    //speed 필드의 Getter/Setter 선언
    public int getSpeed() {
        return speed;
    }
    public void setSpeed(int speed) {
        if(speed < 0) {
```

```
            this.speed = 0;
            return;
        } else {
            this.speed = speed;
        }
    }
    //stop 필드의 Getter/Setter 선언
    public boolean isStop() {
        return stop;
    }
    public void setStop(boolean stop) {
        this.stop = stop;
        if(stop == true) this.speed = 0;
    }
}
```

CarExample.java

```
package ch06.sec14;

public class CarExample {
    public static void main(String[] args) {
        //객체 생성
        Car myCar = new Car();

        //잘못된 속도 변경
        myCar.setSpeed(-50);
        System.out.println("현재 속도: " + myCar.getSpeed());

        //올바른 속도 변경
        myCar.setSpeed(60);
        System.out.println("현재 속도: " + myCar.getSpeed());

        //멈춤
        if(!myCar.isStop()) {
            myCar.setStop(true);
        }
        System.out.println("현재 속도: " + myCar.getSpeed());
    }
}
```

6.15 싱글톤 패턴

- 애플리케이션 전체에서 단 한 개의 객체만 생성해서 사용하고 싶을 때 적용
- 생성자를 private 접근 제한해 외부에서 new 연산자로 생성자를 호출할 수 없도록 막음
- 외부에서 객체를 얻는 유일한 방법은 getInstance() 메소드를 호출하는 것

Singleton.java

```
package ch06.sec15;

public class Singleton {
    //private 접근 권한을 갖는 정적 필드 선언과 초기화
    private static Singleton singleton = new Singleton();

    //private 접근 권한을 갖는 생성자 선언
    private Singleton() {
    }

    //public 접근 권한을 갖는 정적 메소드 선언
    public static Singleton getInstance() {
        return singleton;
    }
}
```

SingletonExample.java

```
javapackage ch06.sec15;

public class SingletonExample {
    public static void main(String[] args) {
        /*
        Singleton obj1 = new Singleton(); //컴파일 에러
        Singleton obj2 = new Singleton(); //컴파일 에러
         */

        //정적 메소드를 호출해서 싱글톤 객체 얻음
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();

        //동일한 객체를 참조하는지 확인
        if(obj1 == obj2) {
            System.out.println("같은 Singleton 객체입니다.");
        } else {
            System.out.println("다른 Singleton 객체입니다.");
        }
    }
}
```