● 제어자(modifier): 클래스, 멤버변수, 또는 메서드의 선언부에 함께 사용되어 부가적인 의미를 부여한다. 제어자의 종듀는 크게 접근 제어자와 그 외의 제어자로 나눌 수 있다.

접근 제어자	public, protected, (default), private
그 외 제어자	static, final, abstract, native, transient, snychronized, volatile, strictfp

하나의 대상에 대해서 여러 제어자를 조합하여 사용하는 것이 가능하지만, 접근 제어자는 한 가지만 선택해 사용할 수 있다. 제어자간의 순서는 주로 접근 제어자를 제일 왼쪽에 놓는 경향이 있다.

▶ static 클래스의, 공통적인: static이 붙은 멤버변수와 메서드, 그리고 초기화 블럭은 인스턴스가 아닌 클래스에 관계된 것이기 때문에 인스턴스를 생성하지 않고도 사용할 수 있다.

static 멤버변수	static(클래스)멤버변수는 모든 인스턴스가 공유하기 때문에 인스턴스에 관계 없이 같은 값을 갖는다. - 모든 인스턴스에 공통적으로 사용되는 클래스변수가 된다. - 클래스변수는 인스턴스를 생성하지 않고도 사용 가능하다. - 클래스가 메모리에 로드될 때 생성된다.
static 메서드	인스턴스메서드와 static메서드의 근본적인 차이는 메서드 내에서 인스턴스 멤버를 사용하는가의 여부에 있다. - static메서드 내에서는 인스턴스멤버들을 직접 사용할 수 없다. 인스턴스 멤버를 사용하지 않는 메서드는 static을 붙여서 static메서드로 선언하는 것을 고려해야 한다. 인스턴스를 생성하지 않고도 호출이 가능해서 더 편리하고 속도도 더 빠르기 때문이다. - 인스턴스를 생성하지 않고도 호출이 가능한 static method가 된다.

static 초기화 블럭	static초기화 블럭은 클래스가 메모리에 로드될 때 단 한 번만 수행되며, 주로 클래스변수(static변수)를 초기화하는 데 주로 사용된다.
	
	class StaticTest {
	static int width;
	static int height;
	
	static { // 클래스 초기화 블럭
	// static 변수의 복잡한 초기화 수행
	}
	}

▶ final 마지막의, 변경될 수 없는

final 클래스	확장될 수 없는 클래스로 다른 클래스의 조상이 될 수 없다.
final 메서드	변경될 수 없는 메서드로 오버라이딩을 통해 재정의 될 수 없다.

final 멤버변수	값을 변경할 수 없는 상수가 된다.
final 지역변수	

```
// Ex)

final class FinalTest {                 // 조상이 될 수 없는 클래스
    final int MAX_SIZE = 10;            // 값을 변경할 수 없는 상수(멤버변수)

    final void getMaxSize() {           // 오버라이딩 할 수 없는 메서드(변경불가)
        final int LV = MAX_SIZE;        // 값을 변경할 수 없는 상수(지역변수)
    }
}
```

☞ final 멤버변수는 생성자를 이용해 초기화할 수 있다: 일반적으로 final이 붙은 변수는 선언과 동시에 초기화를 하지만, 인스턴스변수의 경우 생성자에서 초기화 되도록 할 수 있다. 클래스 내에 매개변수를 갖는 생성자를 선언하여, 인스턴스를 생성할 때 final이 붙은 멤버변수를 초기화하는데 필요한 값을 생성자의 매개변수로부터 제공받는 것이다. 이 기능을 활용하면 각 인스턴스마다 final이 붙은 멤버변수가 다른 값을 갖도록 하는 것이 가능하다. 단, 상수지만 선언과 함께 초기화하지 않고 생성자에서 단 한번만 초기화할 수 있다.

```
class Card {
    final int NUMBER;  // 상수지만 선언과 함께 초기화하지 않고 생성자에서 단 한번만 초기화한다
    final String KIND;
    static final int WIDTH = 100;
    static final int HEIGHT = 250;

    Card(String kind, int num) {  // 매개변수로 넘겨받은 값으로 초기화하는 생성자
        KIND = kind;
        NUMBER = num;
    }

    public String toString() {
        return KIND + " " + NUMBER;
    }
}

class Ex {
    public static void main(String[] args) {
        Card c = new Card("CLOVER", 10);
        System.out.println(c.KIND);     // CLOVER
        System.out.println(c.NUMBER);   // 10
        System.out.println(c);          // CLOVER 10
    }

}
```

▶ abstract 추상의, 미완성의: 메서드의 선언부만 작성하고 실제 수행내용은 구현하지 않은(구현부가 없는) 추상 메서드를 선언하는데 사용되며, 클래스 내에 추상메서드가 존재한다는 것을 알리기 위해 클래스에도 사용된다. 추상메서드는 상속을 통해서 완성시켜야 한다. 상속받아서 완전한 클래스를 만든 후에 객체 생성이 가능하다.

abstract 클래스	클래스 내에 추상메서드가 선언되어 있음을 의미한다. 미완성 설계도이므로 인스턴스를

	생성할 수 없다.
abstract 메서드	선언부만 작성하고 구현부는 작성하지 않은 추상 메서드임을 알린다.

드물지만 추상 메서드가 없는 클래스도 abstract을 붙여서 추상클래스로 만드는 경우도 있다. 예를 들면 java.awt.event.WindowAdapter은 아래와 같이 아무런 내용이 없는 메서드들만 정의되어 있다. 이런 클래스는 인스턴스를 생성해봐야 할 수 있는 것이 아무것도 없기 때문에, 아예 인스턴스를 생성하지 못하도록 클래스 앞에 제어자 abstract을 붙여놓은 것이다. 이 클래스 자체로는 쓸모가 없지만, 다든 클래스가 이 클래스를 상속받아서 일부 원하는 메서드만 오버라이딩해도 된다는 장점이 있다. 만일 이 클래스가 없다면 아무런 내용도 없는 메서드를 잔뜩 오버라이딩해야 한다.

● 접근 제어자(access modifier): 클래스, 멤버변수, 메서드, 생성자에 사용되어 외부에서 접근하지 못하도록 제어하는 역할을 한다. 접근 제어자가 지정되어 있지 않으면 접근 제어자가 default임을 의미한다. 범위가 넓은 쪽에서 좁은 쪽의 순으로 나열하면 다음과 같다: public > protected > (default) > private

private	같은 클래스 내에서만 접근이 가능하다.
(default)	같은 패키지 내의 클레스에서만 접근이 가능하다.
protected	같은 패키지 내의 클레스에서만, 그리고 다든 패키지의 자손클래스에서 접근이 가능하다.
public	접근 제한이 없다.

제어자	같은 클래스	같은 패키지	자손 클래스	전체
public				
protected				
(default)				
private				

대상	사용 가능한 접근 제어자
클래스	public, (default)
메서드	public, protected, (default), private
멤버변수	
지역변수	없음

▶ 접근 제어자를 이용한 캡슐화(encapsulation): 접근 제어자를 사용하는 이유는 두 가지다. 첫번째 이유는 외부로부터 데이터를 보호하기 위한 데이터 감추기(data hiding)이다. 클래스 내부에 선언된 데이터가 유효한 값을 유지하도록, 또는 외부에서 함부로 변경하지 못하도록 외부로부터의 접근을 제한하기 위해서이다. 또 다든 이유는 외부에는 불필요한, 클래스 내부적으로만 사용되는 부분을 감추기 위해서이다. 예를 들면 내부 작업을 위해 임시로 사용되는 멤버변수나 부분작업을 처리하기 위한 메서드 등의 멤버들을 클래스 내부에 감추기 위해서이다. 외부에서 접근할 필요가 없는 멤버들을 private으로 지정하여 외부에 노출시키지 않으므로써 복잡성을 줄일 수 있다. 이 두 이유는 모두 객체지향개념의 캡슐화(encapsulation)에 해당하는 내용이다.

만약 메서드 하나를 변경해야 한다고 가정했을 대, 메서드의 접근 제어자가 public이라면 메서드를 변경한 후에 오류가 없는지 테스트해야 하는 범위가 넓다. 하지만 접근 제어자가 default라면 패키지 내부만 확인해 보면 되고, private이면 클래스 하나만 살펴보면 된다. 이처럼 접근 제어자 하나가 대로는 상당한 차이를 만들어낼 수 있기 때문에, 접근 제어자를 적절하게 선택해서 접근 범위를 최소화하도록 노력해야 한다.

☞ 겟터(getter): 멤버변수의 값을 읽는 메서드다. private이나 protected(상속을 통해 확장될 것이 예상되는 클래스일 경우)으로 설정되어 외부에서 직접 접근할 수 없는 멤버변수의 값을 반환하는 일을 한다. 겟터 메서드는 외부에서 접근 가능하도록 해야 하기 때문에 보통 public으로 설정해 준다.

☞ 세터(setter): 멤버변수의 값을 변경하는 메서드다. 매개변수에 지정된 값을 검사하여 조건에 맞는 값일 때만 멤버변수의 값을 변경하도록 작성되어 있다. private이나 protected(상속을 통해 확장될 것이 예상되는 클래스일 경우) 설정된 멤버변수는 외부에서 직접 접근할 수 없기 때문에, 세터를 통해서 유효성 검사를 통과할 때만 값을 지정해 줄 수 있다. 세터 메서드를 사용하는 클래스의 경우 클래스의 생성자의 구현 부분에 세터 메서드를 사용한다. 세터 메서드는 외부에서 접근 가능하도록 해야 하기 때문에 보통 public으로 설정해 주고, 리턴 타입은 void다.

```
class Time {
    private int hour, minute, second;

    public int getHour() {
        return hour;
    }

    public void setHour(int hour) {
        if (hour < 0 || hour > 23) {
            return;
        }
        this.hour = hour;
    }

    public int getMinute() {
        return minute;
    }

    public void setMinute(int minute) {
        if (minute < 0 || minute > 59) {
            return;
        }
        this.minute = minute;
    }

    public int getSecond() {
        return second;
    }

    public void setSecond(int second) {
        if (second < 0 || second > 59) {
            return;
        }
        this.second = second;
    }

    public String toString() {
        return hour + ":" + minute + ":" + second;
    }

    Time (int hour, int minute, int second) {
        setHour(hour);
        setMinute(minute);
        setSecond(second);
    }

}

public class TimeTest {
    public static void main(String[] args) {
        Time t = new Time(12, 35, 30);
        t.setHour(t.getHour() + 1);
        System.out.println(t);      // 13:35:30
    }
}
```

☞ 생성자의 접근 제어자: 인스턴스 생성을 제어할 수 있다. 보통 생성자의 접근 제한자는 클래스의 그것과 같지만, 다르게 지정할 수도 있다. 생성자의 접근 제어자를 private으로 지정하면, 외부에서 생성자에 접근할 수 없으므로 인

스턴스를 생성할 수 없게 된다. 그래도 클래스 내부에서는 인스턴스를 생성할 수 있다. 대신, 인스턴스를 생성해서 반환해주는 public메서드를 제공함으로써 외부에서 이 클래스의 인스턴스를 사용할 수 있도록 할 수 있다. 단, 이 메서드를 사용하기 전에 인스턴스가 미리 생성되어 있어야 하고 (이때 인스턴스를 저장하는 참조변수는 private static이어야 한다), 이 메서드는 public인 동시에 static이어야 한다 (인스턴스를 생성하지 않고도 호출할 수 있어야 하기 때문이다). 이처럼 생성자를 통해 직접 인스턴스를 생성하지 못하게 하고 public메서드를 통해 인스턴스에 접근하게 함으로써 사용할 수 있는 인스턴스의 갯수를 제한할 수 있다. 또한 생성자가 private인 클래스는 다른 클래스의 조상이 될 수 없다. 왜냐하면, 자손클래스의 인스턴스를 생성할 때 조상클래스의 생성자를 호출해야만 하는데, 생성자의 접근 제어자가 private이므로 자손클래스에서 호출하는 것이 불가능하기 때문이다. 그래서 클래스 앞에 final을 더 추가하여 상속할 수 없는 클래스라는 것을 알리는 것이 좋다.

```
final class Calculation {
    int x = 2;
    int y = 3;

    private static Calculation c = new Calculation();
    // Calculation()에서 사용될 수 있도록 인스턴스가 미리 생성되어야 하므로
    // static이어야 한다.

    private Calculation() {
    }

    public static Calculation getInstance() {
        return c;
    }
}

class Ex {
    public static void main(String[] args) {
        Calculation c = Calculation.getInstance();
        System.out.println(c.x); // 2
    }
}
```

▶ 제어자(modifier)의 조합

클래스	public, (default), final, abstract
메서드	모든 접근 제어자, final, abstract, static
멤버변수	모든 접근 제어자, final, static
지역변수	final

1. 메서드에 static과 abstract은 함께 사용할 수 없다. static메서드는 몸통이 있는 메서드에만 사용할 수 있기 때문이다.
2. 클래스에 abstract과 final은 동시에 사용할 수 없다. 클래스에 사용되는 final은 클래스를 확장할 수 없다는 의미인데 abstract은 상속을 통해서 완성되어야 한다는 의미이므로 서로 모순되기 때문이다.
3. abtract메서드의 접근 제어자가 prviate일 수 없다. abstract메서드는 자손클래스에서 구현해주어야 하는데 접근 제어자가 private이면, 자손클래스에서 접근할 수 없기 때문이다.

4. 메서드에 private과 final은 같이 사용할 수 없다.
접근 제어자가 private인 메서드는 오버라이딩 될 수 없기 때문이다. 이 둘 중 하나만 사용해도 의미가 충분하다.

Resource: 자바의 정석