멤버의 이용 가능 여부를 public:(이용할 수 있다)과 private:(이용할 수 없다) 두 개로 나누는 것만으로도 충분하지만 클래스의 사용 방법에 따라서 의미가 바뀌는 protected:(객체를 만든다면 이용할 수 없다/상속하는 것이라면 이용할 수 있다)가 있는 것이 편리할 것으로 C++의 설계자는 생각했던 것입니다.

이러한 이해 방식은 C++에 한정되지 않고 객체 지향 프로그램에서 일반적인 것이 되고 있습니다. 예를 들면 UML 클래스 다이어그램에 멤버를 첨가하였을 경우에는 다음 그림처럼 사각형을 3개의 영역으로 나누어 상단에 클래스명, 중간에 멤버 변수, 하단에 멤버 함수를 기입합니다. 각 멤버 앞에는 public:을 나타내는 +기호, private:를 나타내는 -기호, 혹은 protected:를 나타내는 #기호 중 아무것이나 부가할 수 있습니다.

POINT

UML의 클래스 다이어그램에서는 public:을 +, private:를 -, protected:를 #으로 나타낸다.

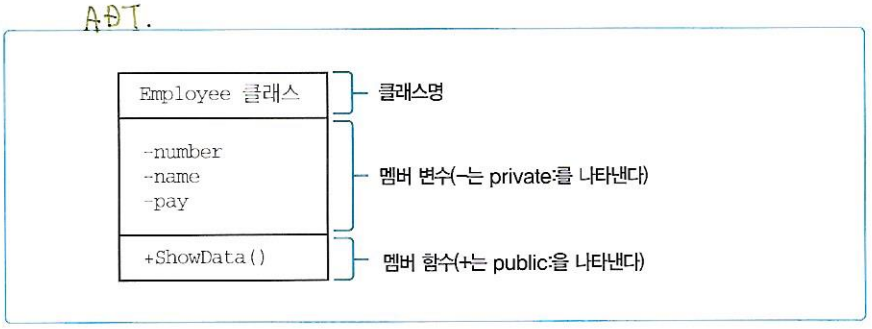

그림 4-4
UML 클래스 다이어그램에서의 접근 지정어

캡슐화의 목적

접근 지정어가 없는 프로그래밍 언어에서는 프로그램 속에 존재하는 변수나 함수를 프로그램 이외의 부분에서 항상 이용할 수 있게 됩니다. 접근 지정어를 사용하는 장점은 특정한 변수나 멤버 함수 이용을 불가능하게 하는 것에 있습니다. 즉, 클래스의 사용자에게 이용할 가치가 없는 멤버를 감추는 것입니다. 멤버를 감추는 것을 캡슐화라고 합니다. 캡슐화는 객체 지향 프로그래밍의 3개의 핵심 중 하나입니다. 캡슐화가 무엇에 도움이 되는가, 어떤 장면에서 캡슐화를 실행하면 좋은가에 대한 문제는 객체 지향 프로그래밍의 인식 방법에 따라 다양합니다.

POINT

캡슐화란 클래스의 사용자에게 가치가 없는 멤버를 감추는 것이다.

먼저 '객체 지향 프로그래밍이란 부품을 조합하여 프로그램을 작성하는 것'이라는 인식 방법에 비교해 보겠습니다. 클래스가 부품에 걸맞다는 뜻입니다. 부품은 사용자에게 있어서 사용하기 쉬운 것이어야 합니다. 100개의 멤버 변수와 200개의 멤버 함수를 가진 클래스가 있다고 가정합니다. 도합 300개의 멤버가 모두 public:로 공개되어 있다면 사용자는 어떤 멤버를 어떻게 사용하면 좋을지, 곤혹스러워 집니다.