1) 클래스와 객체

매 강의 강의자료 시작에 PDF파일을 올려두었어요!

▼ PDF 파일

[수업 목표]

- C#에서 클래스와 객체를 이해하고 사용하는 방법을 익힙니다.
- 생성자와 소멸자, 접근 제한자, Properties 등 클래스의 주요 요소를 학습합니다.
- 클래스를 활용하여 프로그램을 작성하는 연습을 합니다.

[목차]

모든 토글을 열고 닫는 단축키
Windows : `Ctrl` + `alt` + `t`
Mac : `⌘` + `⌥` + `t`

---

01. 객체지향 프로그래밍(Object-Oriented Programming, OOP)의 특징

✔️ 객체지향 프로그래밍의 특징을 확인하고 이해해 보자

▼ 1) 특징

1. 캡슐화 (Encapsulation):
    - 관련된 데이터와 기능을 하나의 단위로 묶는 것을 의미합니다.
    - 클래스를 사용하여 데이터와 해당 데이터를 조작하는 메서드를 함께 캡슐화하여 정보를 은닉하고, 외부에서 직접적인 접근을 제한함으로써 안정성과 유지보수성을 높입니다.
2. 상속 (Inheritance):
    - 상속은 기존의 클래스를 확장하여 새로운 클래스를 만드는 메커니즘입니다.
    - 부모 클래스(상위 클래스, 슈퍼 클래스)의 특성과 동작을 자식 클래스(하위 클래스, 서브 클래스)가 상속받아 재사용할 수 있습니다.
    - 코드의 중복을 줄이고, 클래스 간 계층 구조를 구성하여 코드의 구조화와 유지보수를 용이하게 합니다.
3. 다형성 (Polymorphism):
    - 다형성은 하나의 인터페이스나 기능을 다양한 방식으로 구현하거나 사용할 수 있는 능력을 의미합니다.
    - 하나의 메서드 이름이 다양한 객체에서 다르게 동작할 수 있도록 하는 것으로, 오버로딩과 오버라이딩을 통해 구현됩니다.
    - 유연하고 확장 가능한 코드 작성을 가능하게 하며, 코드의 가독성과 재사용성을 높입니다.
4. 추상화 (Abstraction):
    - 추상화는 복잡한 시스템이나 개념을 단순화하여 필요한 기능에 집중하는 것을 의미합니다.
    - 클래스나 인터페이스를 사용하여 실제 세계의 개념을 모델링하고, 필요한 부분에 대한 명세를 정의합니다.
    - 세부 구현 내용을 감추고 핵심 개념에 집중함으로써 코드의 이해와 유지보수를 용이하게 합니다.
5. 객체 (Object):

- 객체는 클래스로부터 생성된 실체로, 데이터와 해당 데이터를 조작하는 메서드를 가지고 있습니다.
- 객체는 상태(데이터)와 행동(메서드)을 가지며, 실제 세계의 개체나 개념을 모델링합니다.
- 객체들 간의 상호작용을 통해 프로그램이 동작하고, 모듈화와 재사용성을 높입니다.

▼ 2) 클래스의 구성 요소

1. 필드 (Fields): 클래스에서 사용되는 변수를 말한다. 객체의 상태를 나타내는 데이터를 저장하기 위해 사용된다.
2. 메서드 (Methods): 클래스에서 수행되는 동작이 정의된다. 객체의 동작을 구현하기 위해 사용된다.
3. 생성자 (Constructors): 객체를 초기화하는 역할을 한다. 객체가 생성될 때 자동으로 호출되며, 필드를 초기화하는 등의 작업을 수행한다.
4. 소멸자 (Destructors): 객체가 소멸될 때 호출되는 메서드로, 메모리나 리소스의 해제 등의 작업을 수행한다.

▼ 3) 클래스

- 클래스는 객체를 생성하기 위한 템플릿 또는 설계도 역할을 한다.
- 클래스는 속성과 동작을 가진다. 속성은 필드로, 동작은 메서드로 표현된다.
- 객체를 생성하기 위해서는 클래스를 사용하여 인스턴스를 만들어야 한다.

- 붕어빵틀로 비유하면, 클래스는 붕어빵을 만들기 위한 틀이라고 할 수 있다.

▼ 4) 객체

- 객체는 클래스의 인스턴스이다. 클래스의 실체화된 형태라고 할 수 있다.
- 객체는 클래스로부터 생성되며, 각 객체는 독립적인 상태를 가지고 있다. 즉, 객체마다 고유한 데이터를 가질 수 있다.

- 붕어빵 틀로 비유하면, 객체는 붕어빵이라고 할 수 있다. 붕어빵 틀로 만들어진 실제 붕어빵이 객체라고 볼 수 있다.

▼ 5) 클래스 선언과 인스턴스

- 클래스(Class)는 데이터와 메서드를 하나로 묶은 사용자 정의 타입입니다.

▼ [코드스니펫] 클래스 구조 기초

```
class Person
{
    public string Name;
    public int Age;

    public void PrintInfo()
    {
        Console.WriteLine("Name: " + Name);
        Console.WriteLine("Age: " + Age);
    }
}

Person p = new Person();
p.Name = "John";
p.Age = 30;
p.PrintInfo(); // 출력: Name: John, Age: 30
```

▼ 6) 구조체 vs 클래스

- 구조체와 클래스는 모두 사용자 정의 형식을 만드는 데 사용될 수 있습니다.
- 구조체는 값 형식이며, 스택에 할당되고 복사될 때 값이 복사됩니다.
- 클래스는 참조 형식이며, 힙에 할당되고 참조로 전달되므로 성능 측면에서 다소 차이가 있습니다.
- 구조체는 상속을 받을 수 없지만 클래스는 단일 상속 및 다중 상속이 가능하다.
- 구조체는 작은 크기의 데이터 저장이나 단순한 데이터 구조에 적합하며, 클래스는 더 복잡한 객체를 표현하고 다양한 기능을 제공하기 위해 사용됩니다.

02. 접근 제한자

✔️ 클래스의 접근 권한을 지정하여 데이터를 보호할수 있다

▼ 1) 접근 제한자란?

접근 제한자(Access Modifier)는 클래스, 필드, 메서드 등의 접근 가능한 범위를 지정하는 키워드입니다. 접근 제한자는 클래스의 캡슐화를 제어하는 역할을 합니다.

다음은 대표적인 접근 제한자 예시이다.

- public: 외부에서 자유롭게 접근이 가능합니다.
- private: 같은 클래스 내부에서만 접근 가능합니다.
- protected: 같은 클래스 내부와 상속받은 클래스에서만 접근 가능합니다.

```
class Person
{
    public string Name;          // 외부에서 자유롭게 접근 가능
    private int Age;            // 같은 클래스 내부에서만 접근 가능
    protected string Address;  // 같은 클래스 내부와 상속받은 클래스에서만 접근 가능
}
```

03. 필드와 메서드

✔️ 클래스의 구성 요소들을 확인해보자

- 클래스는 필드와 메서드로 구성됩니다.
- 필드는 클래스 내부에 선언되어 있는 변수로, 클래스의 상태를 나타내는 데이터를 저장합니다.
- 메서드는 클래스 내부에 선언되어 있는 함수로, 클래스의 동작을 정의하고 실행합니다.

▼ 1) 필드 (Fields)

- 필드는 클래스나 구조체 내에서 객체의 상태(데이터)를 저장하는 변수입니다.
- 객체의 특징이나 속성을 표현하며, 클래스의 멤버 변수로 선언됩니다.
- 보통 필드는 private 접근 제한자를 사용하여 외부에서 직접적인 접근을 제한하고, 필요한 경우에는 프로퍼티를 통해 간접적으로 접근할 수 있도록 합니다.

```
class Player
{
    // 필드 선언
    private string name;
    private int level;
```

```
}
```

▼ 2) 메서드 (Methods)

- 메서드는 클래스나 구조체에서 객체의 동작(기능)을 정의하는 함수입니다.
- 객체의 행동이나 동작을 구현하며, 클래스의 멤버 함수로 선언됩니다.
- 메서드는 입력값을 받아서 처리하고, 결과값을 반환할 수도 있습니다.
- 객체의 상태를 변경하거나 다른 메서드를 호출하여 작업을 수행합니다.
- 보통 메서드는 public 접근 제한자를 사용하여 외부에서 호출할 수 있도록 합니다.

```
class Player
{
    // 필드
    private string name;
    private int level;

    // 메서드
    public void Attack()
    {
        // 공격 동작 구현
    }
}
```

- 메서드를 호출하기 위해서는 해당 메서드가 속해 있는 클래스의 인스턴스를 생성해야 합니다. 이후 생성된 인스턴스를 통해 메서드를 호출할 수 있습니다.

```
Player player = new Player();  // Player 클래스의 인스턴스 생성
player.Attack();  // Attack 메서드 호출
```

04. 생성자와 소멸자

클래스는 생성자와 소멸자라는 특별한 종류의 메서드를 가질 수 있습니다. 생성자는 클래스의 인스턴스를 생성할 때 호출되는 메서드이며, 소멸자는 클래스의 인스턴스가 메모리에서 해제될 때 호출되는 메서드입니다.

▼ 1) 생성자 (Constructor)

1. 생성자란?

- 생성자는 객체가 생성될 때 호출되는 특별한 메서드입니다.
- 클래스의 인스턴스(객체)를 초기화하고, 필요한 초기값을 설정하는 역할을 수행합니다.
- 생성자는 클래스와 동일한 이름을 가지며, 반환 타입이 없습니다.
- 객체를 생성할 때 new 키워드와 함께 호출됩니다.

2. 생성자의 특징
    - 객체를 초기화하는 과정에서 필요한 작업을 수행할 수 있습니다.
    - 생성자는 여러 개 정의할 수 있으며, 매개변수의 개수와 타입에 따라 다른 생성자를 호출할 수 있습니다. 이를 생성자 오버로딩이라고 합니다.
    - 기본적으로 매개변수가 없는 디폴트 생성자가 클래스에 자동으로 생성되지만, 사용자가 직접 정의한 생성자가 있는 경우 디폴트 생성자가 자동으로 생성되지 않습니다.

▼ [코드스니펫] 생성자 기초

```
class Person
{
    private string name;
    private int age;

    public void PrintInfo()
    {
        Console.WriteLine($"Name: {name}, Age: {age}");
    }
}



Person person1 = new Person();
```

```
class Person
{
    private string name;
    private int age;

    // 매개변수가 없는 디폴트 생성자
    public Person()
    {
        name = "Unknown";
```

```
        age = 0;
    }

    // 매개변수를 받는 생성자
    public Person(string newName, int newAge)
    {
        name = newName;
        age = newAge;
    }

    public void PrintInfo()
    {
        Console.WriteLine($"Name: {name}, Age: {age}");
    }
}
```

```
Person person1 = new Person();                    // 디폴트 생성자 호출
Person person2 = new Person("John", 25);          // 매개변수를 받는 생성자 호출
```

▼ 2) 소멸자 (Destructor)

1. 소멸자란?
    - 소멸자는 객체가 소멸되는 시점에서 자동으로 호출되는 특별한 메서드입니다.
    - 객체의 사용이 종료되고 메모리에서 해제될 때 자동으로 호출되어 필요한 정리 작업을 수행합니다.
    - 클래스와 동일한 이름을 가지며, 이름 앞에 ~ 기호를 붙여 표현합니다.
    - 소멸자는 반환 타입이 없고 매개변수를 가질 수 없습니다.
    - C#에서는 가비지 컬렉터(Garbage Collector)에 의해 관리되는 메모리 해제를 담당하므로, 명시적으로 소멸자를 호출하는 것은 일반적으로 권장되지 않습니다.
2. 소멸자의 역할
    - 자원 해제: 파일 핸들, 네트워크 연결, 데이터베이스 연결 등의 외부 리소스를 사용한 경우, 소멸자를 통해 해당 리소스를 해제할 수 있습니다.
    - 메모리 해제: 객체가 사용한 메모리를 해제하고 관련된 자원을 정리할 수 있습니다.

- 로깅 및 디버깅: 객체가 소멸되는 시점에 로깅 작업을 수행하거나 디버깅 정보를 기록할 수 있습니다.

```
class Person
{
    private string name;

    public Person(string newName)
    {
        name = newName;
        Console.WriteLine("Person 객체 생성");
    }

    ~Person()
    {
        Console.WriteLine("Person 객체 소멸");
    }
}
```

05. 프로퍼티 (Property)

▼ 1) 프로퍼티란?

- 프로퍼티는 클래스 멤버로서, 객체의 필드 값을 읽거나 설정하는데 사용되는 접근자(Accessor) 메서드의 조합입니다.
- 객체의 필드에 직접 접근하지 않고, 간접적으로 값을 설정하거나 읽을 수 있도록 합니다.
- 필드에 대한 접근 제어와 데이터 유효성 검사 등을 수행할 수 있습니다.
- 프로퍼티는 필드와 마찬가지로 객체의 상태를 나타내는 데이터 역할을 하지만, 외부에서 접근할 때 추가적인 로직을 수행할 수 있습니다.

▼ 2) 프로퍼티 구문

- 프로퍼티는 get과 set 접근자를 사용하여 값을 읽고 설정하는 동작을 정의합니다.
- get 접근자는 프로퍼티의 값을 반환하고, set 접근자는 프로퍼티의 값을 설정합니다.
- 필요에 따라 get 또는 set 접근자 중 하나를 생략하여 읽기 전용 또는 쓰기 전용 프로퍼티를 정의할 수 있습니다.

```
[접근 제한자] [데이터 타입] 프로퍼티명
{
    get
    {
        // 필드를 반환하거나 다른 로직 수행
    }
    set
    {
        // 필드에 값을 설정하거나 다른 로직 수행
    }
}
```

▼ 3) 프로퍼티 사용 예시

다음은 Player 클래스의 이름과 레벨을 나타내는 name과 level 필드를 캡슐화한 프로퍼티의 예시입니다.

```
class Person
{
    private string name;
    private int age;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int Age
    {
        get { return age; }
        set { age = value; }
    }
}
```

```
Person person = new Person();
person.Name = "John";   // Name 프로퍼티에 값 설정
person.Age = 25;        // Age 프로퍼티에 값 설정

Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");  // Name과 Age 프로
```

▼ 4) 프로퍼티 접근 제한자 적용 & 유효성 검사 예제

```
class Person
{
    private string name;
    private int age;

    public string Name
    {
        get { return name; }
        private set { name = value; }
    }

    public int Age
    {
        get { return age; }
        set
        {
            if (value >= 0)
                age = value;
        }
    }
}
```

```
Person person = new Person();
person.Name = "John";     // 컴파일 오류: Name 프로퍼티의 set 접근자는 private입니다.
person.Age = -10;         // 유효성 검사에 의해 나이 값이 설정되지 않습니다.

Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");  // Name과 Age 프로퍼티에 접근하여 값을 출력합니다.
```

▼ 5) 자동 프로퍼티 (Auto Property)

- 자동 프로퍼티는 프로퍼티를 간단하게 정의하고 사용할 수 있는 편리한 기능입니다.
- 필드의 선언과 접근자 메서드의 구현을 컴파일러가 자동으로 처리하여 개발자가 간단한 구문으로 프로퍼티를 정의할 수 있습니다.

자동 프로퍼티는 다음과 같은 구문을 가집니다.

```
[접근 제한자] [데이터 타입] 프로퍼티명 { get; set; }
```

아래는 자동 프로퍼티를 사용한 예시입니다.

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

```
Person person = new Person();
person.Name = "John";     // 값을 설정
person.Age = 25;          // 값을 설정

Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");  // 값을 읽어 출력
```

Tip

- 클래스와 객체를 사용하여 프로그램을 작성할 때, 객체 지향 프로그래밍(OOP)의 원칙을 지키도록 노력해야 합니다. 캡슐화, 상속관계, 다형성 등 OOP 개념을 이해하고 활용하여 유지보수성이 높고 재사용 가능한 코드를 작성할 수 있습니다.
- Properties를 사용하여 필드 접근을 제어하면, 코드의 안정성과 가독성을 높일 수 있습니다.
- 클래스의 접근 제한자를 적절히 사용하여 필요한 부분만 외부에서 접근 가능하도록 설정하는 것이 좋습니다.

이전 강의

7강 메서드와 구조체

다음 강의

9강 상속과 다형성