[13기 표은서] C# 기초 세션 2차시

1. 객체지향 개념

1.1 객체란?

현실 세계에서 실재하는 모든 대상을 변수(상태/속성)와 함수(행동)으로 추상화시킨 개념

- 여기서 추상화는 **대상의 본질적인 특징을 추출하여 표현**한 것

1.2 OOP란?

Object Oriented Programming의 약자로, ***객체 지향 프로그래밍***

- 즉, **하나의 소프트웨어가 동작하는 원리**를 그것을 구성하는 여러 객체 간의 상호작용으로 정의하고, 이에 따라 객체를 중심으로 소프트웨어를 설계하고 개발해야한다는 프로그래밍 패러다임

1.3 OOP의 4가지 특징

- **추상화, 상속, 다형성, 캡슐화**
- 위 4가지 특징에 기반하여 객체를 중심으로 설계된 프로그램은 개발 생산성이 뛰어날 것으로 기대

1.3.1 추상화

어떤 대상/집단의 공통적이고 본질적인 특징을 추출하여 정의한 것

- 어떤 대상을 구현할 때, 그 대상의 본질적인 특징을 정의하고, 이것에 기반하여 대상을 객체로 구현하는 것
- 이때 대상의 본질적인 특징을 정의하는데 프로그래밍적으로 활용되는 개념:
  - abstract class(추상 클래스)
  - interface(인터페이스)

1.3.2 상속

객체를 추상화/구현할 때, 기존에 구현한 클래스를 재활용하여 구현할 수 있는 것

- 이때 재활용한 클래스는 **상위 클래스**, 기존 클래스를 재활용하여 구현한 클래스는 **하위 클래스**가 됨
- 대상을 추상화하여 객체로 구현할 때, 대상의 특징을 일부분 구현한 객체가 있을 경우, 이를 확장하여 대상을 추상화/구현할 수 있다는 뜻
- 상속을 통해 **코드의 재사용성을 높임**

1.3.3 다형성

어떤 객체의 속성이나 기능이 상황에 따라 여러 형태로 변할 수 있다는 것

- OOP에서 다형성을 구현하는 예시로는 상속/구현 상황에서 **메소드 오버라이딩/오버로딩**
- 다형성을 통해 개발 유연성, 코드 재사용성을 제고시킴
- 다형성이 구현된 구조에서 상위 객체의 타입으로 하위 객체를 참조 가능
  - 메소드 오버라이딩: 변형
  - 메소드 오버로딩:

1.3.4 캡슐화

클래스 내의 연관된 속성(property)이나 함수(method)를 하나의 캡슐로 묶어 외부로부터 클래스로의 접근을 최소화하는 것

1. 외부로부터 클래스의 변수, 함수를 보호
2. 외부에는 필요한 요소만 노출하고, 내부의 상세한 동작은 은닉
   - public, static, private, protected와 같은 접근 제한자를 통해 캡슐화를 구현함

접근 제한자	설명
public	***하위 클래스와 인스턴스에서 접근 가능*** 모든 외부(파생클래스 포함)에서 이 타입(클래스, 구조체, 인터페이스 등)을 액세스할 수 있음 (개별 타입 멤버의 액세스 권한은 해당 멤버의 접근 제한자에 따라 별도 제한 가능)
static	***클래스와 외부에서 접근 가능, 인스턴스에서는 접근 불가***
private	***클래스 본인만 접근 가능, 하위 클래스와 인스턴스에서 접근 불가*** 동일 클래스와 구조체 내의 멤버만 접근 가능

접근 제한자	설명
protected	***클래스 본인과 하위 클래스에서 접근 가능, 인스턴스 접근 불가*** 파생클래스에서 이 클래스 멤버를 액세스 가능

- **객체 간의 결합도를 감소시키고, 응집도를 강화하는데 기여**

  ⇒ 즉, 유지보수의 용이성이 좋아지는 효과

  - 객체 간 결합도: 객체 간에 의존하는 정도
  - 객체의 응집도: 한 객체의 자율성, 특정 역할에 대한 독립적인 책임

1.4 요약

- **추상화**: 대상의 본질적인 특성을 추출하여 객체로 표현하는 것
- **상속**: 기존에 있는 클래스를 활용하여 객체를 추상화 및 구현, 코드 재사용성 증가
- **다형성**: 메소드 오버라이딩 및 오버로딩으로 상황에 따라 여러 기능 구현 / 하위 클래스는 상위 클래스 타입으로 참조 가능 / 코드 재사용성 증가
- **캡슐화**: 외부로부터 데이터 보호, 은닉 / 유지보수의 용이성 증가

---

2. as와 is 연산자

2.1 as 연산자

객체를 캐스팅할 때 사용되는 연산자

- 캐스팅에 성공하면 캐스팅 결과를 리턴하고, 캐스팅에 실패하면 null 값을 리턴
- 클래스간 타입 변환을 명시적으로 할 때, 컴파일에서 에러가 발생하지 않지만 프로그램을 실행하면 'InvalidCastException'이 발생하는 경우

  ⇒ as 연산자를 사용하여 캐스팅
- 단, as 연산자는 reference type 간의 캐스팅에만 가능 (value type은 불가)

2.2 is 연산자

as 연산자처럼 캐스팅 성공유무를 확인할 때 사용하는 연산자

- as 연산자는 캐스팅 결과를 리턴하지만, is 연산자는 캐스팅이 가능하면 true, 불가능하면 false를 리턴
- 따라서 단지 캐스팅 성공유무만 판단 가능

2.3 예제

```
int i;

// error: Cannot implicitlt convert type 'string' to 'int'
i = "Hello";
```

- i를 int로 선언한 후 문자열 "Hello"를 할당할 수 없음
- 형식 변환 필요 → 명시적 변환(캐스팅)
  1. as 연산자
     - 형식 변환 연산자(캐스팅)과 동일한 역할로, 캐스팅에 실패할 경우 **null을 리턴**
  2. is 연산자
     - 객체가 해당 형식에 해당하는지를 검사하여 그 결과를 **bool 값**으로 반환

```
// 일반적인 캐스트 방식: 예외 처리 이용
try
{
    Student objStudent = (Student)myobject;
}
catch (Exception es)
{
    MessageBox.Show(ex.Message);
}

// as
Stdent objStudent = myobject as Student;
if (objStudent != null)
{
    // use object
}

// is
if (myobject is Student)
{
```

```
    Student objStudent = (Student)myobject;
}
```

---

3. 추상 클래스와 인터페이스

3.1 추상 클래스(abstract class)란?

미완성된 클래스로, 추상 메소드(미완성된 메소드)를 포함한 클래스

- 목적: 자식 클래스에서 공유할 수 있도록 추상 클래스(부모 클래스)의 공통적인 정의를 제공하는 것
- 추상 클래스 안에서 선언되는 모든 멤버 변수, 메소드, 이벤트들은 접근 제한자를 적지 않으면 모두 private임

3.1.1 추상 클래스의 특징

1. new 연산자를 이용하여 인스턴스를 생성할 수 없음
2. 오직 자식 클래스에서 상속을 통해서만 구현 가능
3. 추상 메소드와 추상 프로퍼티를 가질 수 있음

3.1.2 추상 메소드의 특징

1. 추상 메소드는 암시적으로 가상 메소드임
2. 추상 메소드의 선언은 오직 추상 클래스에서만 허용
3. 추상 메소드를 선언할 때 static 또는 virtual 키워드를 사용할 수 없음
4. 추상 메소드는 실제 구현을 제공하지 않기 때문에 메소드 본문이 없음

3.1.3 추상 클래스 및 추상 메소드의 선언

- 추상 클래스 선언: class 앞에 abstract 키워드를 붙임
- 추상 메소드 선언: 추상 클래스 안에서 abstract 키워드를 붙이고 {}를 지운 다음 ;를 붙임

```
abstract class Animal
{
    public string name;
    public abstract void Move(); // 추상 메소드
    public void Move2()
    {
        Console.WriteLine("일반 메소드");
    }
}
```

3.1.4 추상 메소드의 구현

: 자식 클래스에서 오버라이딩을 통해 재정의

```
class Dog : Animal
{
    public override void Move()
    {
        Console.WriteLine("네 발로 이동한다.");
    }
}
```

3.2 인터페이스(Interface)란?

추상 클래스처럼 완성되지 않은 불완전한 것으로, 추상 클래스보다 추상화 정도가 높음

- 다른 클래스를 작성하는데 도움을 줄 목적으로 작성
- 인터페이스에 속한 메소드는 모두 가상 메소드에 속함

3.2.1 인터페이스의 특징

1. 인터페이스는 메소드, 프로퍼티, 인덱서, 이벤트만을 가질 수 있음
2. 인터페이스는 구현부가 없음
3. 인터페이스는 접근 제한자를 사용할 수 없고 모든 것이 public으로 선언됨
4. 인터페이스는 new 연산자를 이용하여 인스턴스를 생성할 수는 없지만 참조변수로는 만들 수 있음

3.2.2 인터페이스의 장점

1. 개발시간의 단축
2. 표준화 가능
3. 서로 관계 없는 클래스들에게 관계를 맺어줄 수 있음
4. 독립적인 프로그래밍 가능

3.2.3 추상 클래스와 인터페이스의 차이

- 추상 클래스는 말 그래도 클래스로 정의된 타입이라 **다중 상속이 불가능**
- 인터페이스의 경우 클래스가 아니기 때문에 **다중 상속이 허용**

3.2.4 인터페이스의 선언 및 구현

```
interface ILogger // 인터페이스
{
    void WriteLog(string log);
}

class ConsoleLogger : ILogger // 인터페이스를 구현한 클래스
{
    public void WriteLog(string log)
    {
        Console.WriteLine("{0} {1}", DateTime.Now.ToLocalTime(), log);
    }
}
```

- 인터페이스도 이를 구현한 클래스의 부모라고 할 수 있기 때문에, 해당 인터페이스 타입의 참조변수로 이를 구현한 클래스의 인스턴스를 참조 가능

```
class Program
{
    static void Main(string[] args)
    {
        ILogger logger = new ConsoleLogger();
    }
}
```

---

4. 델리게이트

4.1 델리게이트(Delegate)란?

번역하면 대리자라는 뜻으로, 무엇인가를 대신할 수 있는 역할

- 선언 위치: namespace에 자유롭게 위치 가능, 클래스 내부에서도 사용 가능

  ⇒ 클래스(class)와 동급
- 메소드를 참조하는 대리자
- 이때 메소드는 반환형, 매개변수의 종류와 개수가 모두 일치해야 함

4.1.1 델리게이트 사용법

- 선언

  [지정자] delegate [반환형] [델리게이트이름] (매개변수 타입 매개변수 이름);

  → `delegate void dele (int a);`
- 메소드를 직접 호출하는 것이 아니라 델리게이트를 통해 간접적으로 호출하기 때문에 델리게이트의 생성자를 꼭 쓸 필요는 없음
- 마치 여러 메소드들의 집합을 가리키는 포인터라고 볼 수 있음

  ⇒ 이를 활용해 이름 없는 메소드를 대리자를 활용해 만들어 호출 가능

4.1.2 델리게이트 활용

- 델리게이트 자체는 클래스지만 델리게이트 객체는 변경 가능한 변수임

  ⇒ 실행 중에도 언제든지 변경될 수 있는 변수라는 점을 활용
- 함수들을 델리게이트 배열에 등록하면 델리게이트 변수의 첨자만으로 쉽게 필요한 연산을 수행하게 할 수 있고, 내용이 추가되어도 델리게이트 배열에만 추가해도 됨
  - 델리게이트 배열에 메소드 참조를 등록 → 배열 첨자로 해당 메소드를 불러와 결과로 나타냄
- 또한 변수이므로 메소드를 인수로도 전달 가능
- 메소드에 메소드를 전달하여 실행 중에 호출 가능: 콜백(callback)
  - 델리게이트에 전달하여 사용할 메소드 생성 → 인자에 메소드 전달

---

5. 액션과 이벤트, 제네릭

5.1 액션(action)이란?

반환값이 없는 메소드를 대신 호출하는 제네릭 대리자로, 주로 출력 등과 엮어서 사용 가능

```
class DelegatePractice
{
    static void Main()
    {
        //Action 대리자 생성 : 반환값이 없으므로 인자는 모두 매개변수
        Action<string> printf = Console.Write;
        Action<int, int> printAdd = (int a, int b) => Console.WriteLine(a + b);

        //대리자를 통한 호출
        printf("Hello Action!\n");
        printAdd(3, 6);
    }
}
//출력 : Hello Action! 9
```

5.2 이벤트(event)란?

키 입력이나 클릭과 같은 동작(트리거)

- 이벤트 핸들러(event handler): 특정 이벤트와 연결하고자 만든 메소드
  - 특정한 이벤트가 오면, 대리자 형식으로 이벤트를 수행하도록 함
- 기본적으로 이벤트는 대리자의 방식
- 사용자 입력과 같은 이벤트 발생 시 이벤트 핸들러가 가리키는 이벤트 관련 메소드들을 실행시키는 구조
  - 언제 발생할지 모르는 이벤트를 예측할 필요 X
  - 프로그램이 자동으로 인식해 이벤트를 수행 가능

5.3 제네릭(generic)이란?

코드의 재사용성과 유연성을 향상해주는 도구로, 데이터 형식을 일반화하여 재사용 가능한 코드를 작성할 수 있게 해줌

- 제네릭을 사용하면 다양한 형식의 데이터를 처리하는 메소드와 클래스를 작성할 수 있으며, 컴파일 시점에서 안정성을 보장함
- when?
  1. 여러 데이터 형식에 대해 동일한 로직을 적용해야 할 때
  2. 컬렉션 타입에서 다양한 데이터 형식을 저장하고 관리해야 할 때
  3. 데이터 형식에 따라 다른 연산을 수행해야 할 때