1) 인터페이스와 열거형

매 강의 강의자료 시작에 PDF파일을 올려두었어요!

▼ PDF 파일

[수업 목표]

- C#에서 인터페이스와 열거형의 개념을 이해합니다.
- 인터페이스의 사용법과 구현 방법을 학습합니다.
- 열거형의 역할과 사용 방법을 학습합니다.

[목차]

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

---

01. 다중 상속을 사용하지 않는 이유

✔️ 다중 상속의 문제를 알아봅시다

▼ **1) 사용하지 않는 이유!**

1. **다이아몬드 문제(Diamond Problem):**
   다중 상속을 허용하면 **한 클래스가 두 개 이상의 부모 클래스로부터 동일한 멤버를 상속**받을 수 있습니다. 이 경우, 같은 이름의 멤버를 가지고 있을 때 어떤 부모 클래스의 멤버를 사용해야 하는지 모호해집니다. 이런 모호성을 해결하기 위한 규칙이 필요하게 되는데, 이로 인해 코드가 복잡해지고 가독성이 저하될 수 있습니다.
2. **설계의 복잡성 증가:**
   다중 상속을 허용하면 클래스 간의 **관계가 복잡**해집니다. 클래스가 다중 상속을 받을 경우, 어떤 클래스로부터 어떤 멤버를 상속받을지 결정해야 합니다. 이로 인해 클래스 간의 상속 관계를 파악하기 어려워지고 코드의 유지 보수성이 저하될 수 있습니다.
3. **이름 충돌과 충돌 해결의 어려움:**
   다중 상속을 허용하면 **여러 부모 클래스로부터 상속받은 멤버들이 이름이 충돌**할 수 있습니다. 이러한 충돌을 해결하기 위해 충돌하는 멤버를 재정의해야 하거나 명시적으로 부모 클래스를 지정해야 할 수 있습니다. 이는 코드의 복잡성을 증가시키고 오류 발생 가능성을 높입니다.
4. **설계의 일관성과 단순성 유지:**
   C#은 단일 상속을 통해 설계의 일관성과 단순성을 유지하고자 합니다. 단일 상속을 통해 클래스 간의 관계를 명확하게 만들고 코드의 가독성과 이해도를 높일 수 있습니다. 또한 인터페이스를 사용하여 다중 상속이 필요한 경우에도 유사한 기능을 구현할 수 있습니다.

02. 인터페이스를 사용하는 이유

✔️ 인터페이스를 왜 사용하는지 알고 사용해야 겠죠?

▼ **1) 사용하는 이유!**

1. **코드의 재사용성:**
   인터페이스를 사용하면 **다른 클래스에서 해당 인터페이스를 구현하여 동일한 기능**

을 공유할 수 있습니다. 인터페이스를 통해 다양한 클래스가 동일한 동작을 수행할 수 있으므로 코드의 재사용성이 향상됩니다.

2. **다중 상속 제공:**
   C#에서는 클래스는 단일 상속만을 지원하지만, 인터페이스는 **다중 상속을 지원**합니다. 클래스가 여러 인터페이스를 구현함으로써 여러 개의 기능을 조합할 수 있습니다. 다중 상속을 통해 클래스는 더 다양한 동작을 수행할 수 있습니다.
3. **유연한 설계:**
   인터페이스를 사용하면 **클래스와 인터페이스 간에 느슨한 결합**을 형성할 수 있습니다. 클래스는 인터페이스를 구현하기만 하면 되므로, 클래스의 내부 구현에 대한 변경 없이 인터페이스의 동작을 변경하거나 새로운 인터페이스를 추가할 수 있습니다. 이는 유연하고 확장 가능한 소프트웨어 설계를 가능하게 합니다.

03. 인터페이스 (Interface)

✔️ 다중 상속의 시작

▼ 1) 인터페이스 특징

- 인터페이스란 클래스가 구현해야 하는 멤버들을 정의하는 것입니다.
- 인터페이스는 클래스의 일종이 아니며, 클래스에 대한 제약 조건을 명시하는 것입니다.
- 클래스가 인터페이스를 구현할 경우, 모든 인터페이스 멤버를 구현해야 합니다.
- 인터페이스는 다중 상속을 지원합니다.

▼ 2) 인터페이스 구현

- **인터페이스 및 멤버 정의하기**

```
interface IMyInterface
{
    void Method1();
    int Method2(string str);
}
```

- **인터페이스 구현하기**

```
class MyClass : IMyInterface
{
    public void Method1()
    {
        // 구현
    }

    public int Method2(string str)
    {
        // 구현
        return 0;
    }
}
```

▼ 3) 사용 예제

▼ 인터페이스 구현해보기

1. 인터페이스 정의하기:

```
public interface IMovable
{
    void Move(int x, int y); // 이동 메서드 선언
}
```

2. 인터페이스를 구현하는 클래스 생성하기:

```
public class Player : IMovable
{
    public void Move(int x, int y)
    {
        // 플레이어의 이동 구현
    }
}

public class Enemy : IMovable
{
    public void Move(int x, int y)
    {
        // 적의 이동 구현
    }
}
```

3. 인터페이스를 사용하여 객체 이동하기:

```
IMovable movableObject1 = new Player();
IMovable movableObject2 = new Enemy();

movableObject1.Move(5, 0); // 플레이어 이동
movableObject2.Move(1, 9); // 적 이동
```

▼ 아이템 사용 구현 예제

```
// 아이템을 사용할 수 있는 인터페이스
public interface IUsable
{
    void Use();
}

// 아이템 클래스
public class Item : IUsable
{
    public string Name { get; set; }

    public void Use()
    {
        Console.WriteLine("아이템 {0}을 사용했습니다.", Name);
    }
}

// 플레이어 클래스
public class Player
{
    public void UseItem(IUsable item)
    {
        item.Use();
    }
}

// 게임 실행
static void Main()
{
    Player player = new Player();
    Item item = new Item { Name = "Health Potion" };
    player.UseItem(item);
}
```

▼ 다중 상속 구현 예제

```
// 인터페이스 1
public interface IItemPickable
{
```

```
    void PickUp();
}

// 인터페이스 2
public interface IDroppable
{
    void Drop();
}

// 아이템 클래스
public class Item : IItemPickable, IDroppable
{
    public string Name { get; set; }

    public void PickUp()
    {
        Console.WriteLine("아이템 {0}을 주웠습니다.", Name);
    }

    public void Drop()
    {
        Console.WriteLine("아이템 {0}을 버렸습니다.", Name);
    }
}

// 플레이어 클래스
public class Player
{
    public void InteractWithItem(IItemPickable item)
    {
        item.PickUp();
    }

    public void DropItem(IDroppable item)
    {
        item.Drop();
    }
}

// 게임 실행
static void Main()
{
    Player player = new Player();
    Item item = new Item { Name = "Sword" };

    // 아이템 주울 수 있음
    player.InteractWithItem(item);

    // 아이템 버릴 수 있음
    player.DropItem(item);
}
```

▼ 4) 인터페이스 vs 추상클래스

인터페이스의 특징과 장단점:

- 인터페이스는 **추상적인 동작만 정의**하고, 구현을 갖지 않습니다.
- **다중 상속이 가능**하며, 여러 클래스가 동일한 인터페이스를 구현할 수 있습니다.
- 클래스들 간의 결합도를 낮추고, 유연한 상호작용을 가능하게 합니다.
- 코드의 재사용성과 확장성을 향상시킵니다.
- 단점으로는 인터페이스를 구현하는 클래스가 모든 동작을 구현해야 한다는 의무를 가지기 때문에 작업량이 증가할 수 있습니다.

추상 클래스의 특징과 장단점:

- 추상 클래스는 **일부 동작의 구현을 가지며, 추상 메서드를 포함**할 수 있습니다.
- **단일 상속만 가능**하며, 다른 클래스와 함께 상속 계층 구조를 형성할 수 있습니다.
- 공통된 동작을 추상화하여 코드의 중복을 방지하고, 확장성을 제공합니다.
- 구현된 동작을 가지고 있기 때문에, 하위 클래스에서 재정의하지 않아도 될 경우 유용합니다.
- 단점으로는 다중 상속이 불가능하고, 상속을 통해 밀접하게 결합된 클래스들을 형성하므로 유연성이 제한될 수 있습니다.

04. 열거형 (Enums)

✔️ 코드의 가독성을 높여줘

▼ 1) 사용하는 이유

1. **가독성:**
   열거형을 사용하면 일련의 **연관된 상수들을 명명**할 수 있습니다. 이를 통해 코드의 가독성이 향상되고, 상수를 사용할 때 실수로 잘못된 값을 할당하는 것을 방지할 수 있습니다.
2. **자기 문서화(Self-documenting):**
   열거형은 **의미 있는 이름을 사용**하여 상수를 명명할 수 있습니다. 이를 통해 코드의 가독성이 향상되며, 상수의 의미를 명확하게 설명할 수 있습니다.

3. **스위치 문과의 호환성:**
   열거형은 스위치 문과 함께 사용될 때 유용합니다. 열거형을 사용하면 스위치 문에서 다양한 상수 값에 대한 분기를 쉽게 작성할 수 있습니다.

▼ 2) 열거형 특징

- 열거형은 서로 관련된 상수들의 집합을 정의할 때 사용됩니다.
- 열거형의 각 상수는 정수 값으로 지정됩니다.

▼ 3) 열거형 구현

- 열거형 정의

```
enum MyEnum
{
    Value1,
    Value2,
    Value3
}
```

- 열거형 사용

```
MyEnum myEnum = MyEnum.Value1;
```

- 열거형 상수 값 지정

```
enum MyEnum
{
    Value1 = 10,
    Value2,
    Value3 = 20
}
```

- 열거형 형변환

```
int intValue = (int)MyEnum.Value1;  // 열거형 값을 정수로 변환
MyEnum enumValue = (MyEnum)intValue;  // 정수를 열거형으로 변환
```

- 스위치문과의 사용

```
switch(enumValue)
{
    case MyEnum.Value1:
        // Value1에 대한 처리
        break;
    case MyEnum.Value2:
        // Value2에 대한 처리
        break;
    case MyEnum.Value3:
        // Value3에 대한 처리
        break;
    default:
        // 기본 처리
        break;
}
```

▼ 4) 사용 예제

- **요일 출력**

```
enum DaysOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

class Program
{
    static void Main(string[] args)
    {
        DaysOfWeek day = DaysOfWeek.Monday;
        Console.WriteLine("Today is " + day);
    }
}
```

- **입력 월 출력하기**

```
// 월 열거형
public enum Month
{
    January = 1,
    February,
    March,
    April,
```

```
    May,
    June,
    July,
    August,
    September,
    October,
    November,
    December
}

// 처리하는 함수
public void ProcessMonth(int month)
{
    if (month >= (int)Month.January && month <= (int)Month.December)
    {
        Month selectedMonth = (Month)month;
        Console.WriteLine("선택한 월은 {0}입니다.", selectedMonth);
        // 월에 따른 처리 로직 추가
    }
    else
    {
        Console.WriteLine("올바른 월을 입력해주세요.");
    }
}

// 실행 예제
static void Main()
{
    int userInput = 7; // 사용자 입력 예시
    ProcessMonth(userInput);
}
```

- **게임 사용 사례**

```
// 게임 상태
enum GameState
{
    MainMenu,
    Playing,
    Paused,
    GameOver
}

// 방향
enum Direction
{
    Up,
    Down,
    Left,
    Right
}

// 아이템 등급
enum ItemRarity
{
```

```
    Common,
    Uncommon,
    Rare,
    Epic
}
```

다음 강의

12강 예외 처리 및 값형과 참조형