Interface Basics in C#

Overview

An **interface** in C# is a reference type that defines a contract that classes must follow. It specifies what members a class must implement, but not how they should be implemented.

Interface Characteristics

1. An **interface** is a reference type.
2. An **interface** is a contract that enforces the implementation of specified members in a class.

Allowed Members in an Interface

- **Property signatures**
- **Method signatures**
- **Default implemented methods** (Introduced in C# 8.0, available in .NET Core 3.1 and later)
- **Static members** (Introduced in C# 8.0, available in .NET Core 3.1 and later)

Access Modifiers in Interfaces

- `public` is the **default** access modifier for interface members.
- `private` is **not allowed** for signature members (properties or methods).

Implementing Interfaces

There are **two ways** to implement an interface:

1. **Explicit Interface Implementation**
    - Used when a class implements multiple interfaces that have members with the same signature.
    - Helps avoid naming conflicts.
2. **Implicit Interface Implementation**
    - The default way to implement an interface.
    - Methods and properties are implemented normally in the class.

Interface Member Properties

- Interface members are **public** by default.
- Interface members are **abstract** by default.
- Interface members are **virtual** by default.

Interface Instantiation

- **You cannot create an instance of an interface directly.**

```
IExample obj = new IExample(); // INVALID
```

- **You can create a reference to an interface.**

```
IExample reference; // VALID
```

- **Default implemented methods can only be accessed through an interface reference.**

Property Implementation Rules

- If an interface defines a property with **only a `get` accessor**, you **can add a `set` accessor** in the implementing class.
- If an abstract class defines a property with **only a `get` accessor**, you **cannot add a `set` accessor** in a concrete class.

Additional Notes

- **Underscore (_) as a digit separator:**

```
int value = 10_000; // The underscore is ignored and does not affect the value.
```

- **Descending comparison example:**

```
return -this.Salary.CompareTo(other?.Salary); // Negating reverses sorting order
```

Conclusion

Understanding interfaces and their capabilities is crucial in C# programming. Interfaces promote **abstraction**, **polymorphism**, and **separation of concerns**, making code more modular and maintainable.