CAB201 - Week 7 Notes

Oliver Vu

September 6, 2024

1 Inheritance

Inheritance is a mechanism in object-oriented programming where a new class (child or subclass) inherits fields, properties and methods from an existing class (parent or superclass). It allows code reuse, avoids redundancy represents an *is-a* relationship between the parent and child classes.

- `Parent Class` (Superclass/Base class): A class that is inherited from.
- `Child Class` (Subclass/Derived class): A class that inherits from the parent class.

1.1 Parent-Child Class Example

Listing 1: C# Inheritance Example with Comments

```
public class Animal
{
    // Public properties accessible from child classes
    public string Name { get; set; }
    public int Age { get; set; }

    // Constructor to initialize Name and Age
    public Animal(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Method to simulate eating
    public void Eat()
    {
        Console.WriteLine($"{Name} is eating.");
    }

    // New method to display the animal's information
    public void DisplayInfo()
    {
        Console.WriteLine($"Animal Name: {Name}, Age: {Age}");
    }
}

public class Dog : Animal
{
    // Additional property specific to the Dog class
    public string Breed { get; set; }
```

```
    // Constructor to initialize Name, Age, and Breed using base class constructor
    public Dog(string name, int age, string breed) : base(name, age)
    {
        Breed = breed;
    }

    // Method to simulate barking
    public void Bark()
    {
        Console.WriteLine($"{Name} says: Woof!");
    }

    // Overriding the base class method to display additional information
    public void DisplayInfo()
    {
        base.DisplayInfo(); // Call base class method
        Console.WriteLine($"Breed: {Breed}");
    }
}

class Test
{
    static void Main(string[] args)
    {
        // Create a Dog instance using the constructor
        Dog myDog = new Dog("Buddy", 3, "Golden Retriever");

        // Calling methods from both the parent and child classes
        myDog.Eat();           // Output: Buddy is eating.
        myDog.Bark();          // Output: Buddy says: Woof!
        myDog.DisplayInfo();   // Output: Animal Name: Buddy, Age: 3 | Breed: Golden Retriever
    }
}
```

In this example:

- `Animal` is the parent class, and `Dog` is the child class.
- The `Dog` class inherits the `Name` and `Age` properties and the `Eat` method from the `Animal` class.
- The `Dog` class has its own method `Bark`, which is not part of the `Animal` class.

2 Access Modifiers

In C#, access modifiers control the visibility of class members (fields, methods, properties, etc.). The main access modifiers are:

- **private**: Accessible only within the same class.
- **protected**: Accessible within the same class and by derived (child) classes.
- **public**: Accessible from any class, anywhere.

Example

Listing 2: Access Modifiers Example in C#

```
using System;
```

```
namespace Week7_samples
{
    class Parent
    {
        // Fields
        private int privateNumber = 10;          // Only accessible within Parent
        protected int protectedNumber = 20;      // Accessible in Parent and Child classes
        public int publicNumber = 30;            // Accessible anywhere
    }

    class Child : Parent
    {
        public void AccessTest()
        {
            // Cannot access privateNumber in Parent class because it is private
            // Console.WriteLine(privateNumber);  // Error: private

            // Can access protectedNumber in Parent class because it is protected
            Console.WriteLine("Accessing protectedNumber in Child:" + protectedNumber);

            // Can access publicNumber in Parent class because it is public
            Console.WriteLine("Accessing publicNumber in Child:" + publicNumber);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Create an object of Child class
            Child childObject = new Child();
            childObject.AccessTest();

            // Create an object of Parent class
            Parent parentObject = new Parent();

            // Cannot access privateNumber or protectedNumber from outside
            // Console.WriteLine(parentObject.privateNumber);  // Error: private
            // Console.WriteLine(parentObject.protectedNumber);  // Error: protected

            // Can access publicNumber as it is public
            Console.WriteLine("Accessing publicNumber in Main:" + parentObject.publicNumber)
        }
    }
}
```

Explanation

- **Private**: The `privateNumber` field is private and can only be accessed within the `Parent` class. Even the `Child` class cannot access it.
- **Protected**: The `protectedNumber` field is protected, meaning it is accessible within the `Parent` class and any classes that inherit from `Parent`, such as `Child`. However, it cannot be accessed directly from outside, like in the `Main` method.
- **Public**: The `publicNumber` field is public and can be accessed from any class, including the `Child` class and the `Main` method.

The output from running this code would be:

- Accessing protectedNumber in Child: 20
- Accessing publicNumber in Child: 30
- Accessing publicNumber in Main: 30

This demonstrates how the different access levels affect visibility across classes and within inheritance.

3 Virtual Methods

A **virtual method** in C# is a method defined in a base (parent) class that can be overridden by a derived (child) class. This allows the child class to provide its own implementation of the method, while the parent class provides a default implementation. The keyword `virtual` is used in the parent class, and the keyword `override` is used in the child class to provide the new implementation.

Example

Listing 3: Virtual Method Example in C#

```
using System;

namespace Week7_samples
{
    // Parent Class
    class Animal
    {
        protected string name;
        private int age;

        public Animal(string name, int age)
        {
            this.name = name;
            this.age = age;
        }

        // Virtual method that can be overridden in child classes
        public virtual void MakeSound()
        {
            Console.WriteLine($"{name} is making a sound");
        }
    }

    // Child class Cat overrides the MakeSound method
    class Cat : Animal
    {
        private string breed;

        public Cat(string name, int age, string breed) : base(name, age)
        {
            this.breed = breed;
        }

        // Override the virtual method from Animal
        public override void MakeSound()
        {
```

```
            Console.WriteLine($"{name} says: Meow!");
        }
    }

    // Child class Dog does not override the MakeSound method
    class Dog : Animal
    {
        public Dog(string name, int age) : base(name, age)
        {
            // We don't need to do anything here
        }

        // If we don't override the method, the parent class method will be used
    }

    class Program
    {
        static void Main(string[] args)
        {
            Cat my_cat = new Cat("Kitty", 2, "British Shorthair");

            // The MakeSound method in the Cat class will be called
            my_cat.MakeSound();  // Output: Kitty says: Meow!

            Animal my_animal = new Animal("Mic", 3);

            // The MakeSound method in the Animal class will be called
            my_animal.MakeSound();  // Output: Mic is making a sound
        }
    }
}
```

Explanation

- **Parent Class (Animal)**: The `Animal` class has a virtual method called `MakeSound()`, which provides a default implementation.
- **Child Class (Cat)**: The `Cat` class overrides the `MakeSound()` method. The `override` keyword is used to provide a specific implementation that outputs `Meow`.
- **Child Class (Dog)**: The `Dog` class does not override the `MakeSound()` method, so it uses the method from the parent class.

How It Works

- When calling `MakeSound()` on the `my_cat` object (which is of type `Cat`), the overridden method in the `Cat` class is called, outputting: `Kitty says: Meow!`.
- When calling `MakeSound()` on the `my_animal` object (which is of type `Animal`), the parent class implementation is used, outputting: `Mic is making a sound`.
- If a child class does not override a virtual method, the parent class's method is used by default.

The concept of virtual methods provides flexibility in object-oriented design by allowing child classes to define their own behaviours for methods while also allowing default behaviour to be inherited from the parent class.

3.1 Commonly Overridden Methods in Every Class

In C#, all classes implicitly inherit from the `System.Object` class, which provides several methods that can be overridden to customize behavior. Some of the most commonly overridden methods include:

1. `ToString()`

The `ToString()` method returns a string representation of an object. By default, it returns the class name, but it is often overridden to return more meaningful information.

Listing 4: Overriding ToString()

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

    // Override ToString() to provide a meaningful string representation
    public override string ToString()
    {
        return $"Name: {Name}, Age: {Age}";
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person person = new Person { Name = "Alice", Age = 30 };
        Console.WriteLine(person.ToString());  // Output: Name: Alice, Age: 30
    }
}
```

Explanation:

- The default `ToString()` method in the `Person` class would return the class name, but after overriding it, it returns a string with the person's name and age.
- Overriding `ToString()` improves the readability and debugging experience when printing object instances.

2. `Equals()`

The `Equals()` method determines whether two object instances are considered equal. The default implementation checks for reference equality, but this can be overridden to check for value equality instead.

Listing 5: Overriding Equals()

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

    // Override Equals() to compare the values of objects
    public override bool Equals(object obj)
    {
        if (obj == null || GetType() != obj.GetType())
            return false;

        Person other = (Person)obj;
        return (Name == other.Name) && (Age == other.Age);
    }
```

```
    // Always override GetHashCode when Equals is overridden
    public override int GetHashCode()
    {
        return (Name, Age).GetHashCode();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person { Name = "Alice", Age = 30 };
        Person p2 = new Person { Name = "Alice", Age = 30 };
        Console.WriteLine(p1.Equals(p2));  // Output: True
    }
}
```

Explanation:

- The default `Equals()` method checks whether two object references point to the same memory location (reference equality).
- Overriding `Equals()` allows comparing the values of the objects, such as the `Name` and `Age` properties in the `Person` class.
- When overriding `Equals()`, it is recommended to also override `GetHashCode()` to ensure consistency, especially when objects are used in collections like dictionaries or hash sets.

3. `GetHashCode()`

The `GetHashCode()` method returns a hash code (an integer) for the object, which is used in hash-based collections such as `Dictionary<>` or `HashSet<>`. If you override `Equals()`, you must also override `GetHashCode()` to ensure that objects that are considered equal have the same hash code.

Listing 6: Overriding GetHashCode()

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

    // Override GetHashCode() to generate a hash code for the object
    public override int GetHashCode()
    {
        return (Name, Age).GetHashCode();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person { Name = "Alice", Age = 30 };
        Console.WriteLine(p1.GetHashCode());  // Output: Some hash code based on Name and Age
    }
}
```

Explanation:

- `GetHashCode()` should generate the same hash code for two objects that are considered equal by the `Equals()` method.
- In the example, the hash code is generated using a combination of the `Name` and `Age` fields.
- `GetHashCode()` is crucial when storing objects in collections that use hashing algorithms, such as dictionaries.

Summary of Common Methods to Override

- `ToString()`: Provides a string representation of the object.
- `Equals()`: Determines whether two objects are equal based on their values.
- `GetHashCode()`: Generates a hash code for the object, which is used in hash-based collections.

Overriding these methods helps customize the behavior of objects in C# to suit specific needs, particularly when dealing with collections, comparisons, and debugging.

4 Enums

An **enum** (short for enumeration) is a value type in C# that allows you to define a set of named constants. Enums are useful when a variable can only take one out of a small set of possible values. Internally, each constant is assigned an integer value starting from 0, but you can also manually assign values if needed.

Enums improve code readability and maintainability by replacing hard-coded numbers with descriptive names.

Example

Listing 7: Enum Example in C#

```
using System;

namespace Week7_samples
{
    // Enum to represent different directions
    enum Direction
    {
        North,      // 0 by default
        South,      // 1
        East,       // 2
        West        // 3
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Declare a variable of type Direction
            Direction myDirection = Direction.North;

            // Check the value of the enum variable
            if (myDirection == Direction.North)
            {
                Console.WriteLine("You are heading North.");
            }

            // Output the integer value of an enum constant
            Console.WriteLine($"East has the value: {(int)Direction.East}");
        }
```

```
    }
}
```

Explanation

- **Enum Definition**: The `Direction` enum defines four possible values: `North`, `South`, `East`, and `West`. By default, `North` is assigned the value 0, `South` the value 1, and so on.
- **Enum Usage**: In the `Main` method, the `Direction` enum is