Object Oriented Programming - 2

A5 – Principles Applied Program to Design

Ikromkhujaev Ilyoskhuja

Access Modifiers

By now, you are quite familiar with the public keyword that appears in many of our examples:

```
public string color;
```

The `public` keyword is an access modifier, which is used to set the access level/visibility for classes, fields, methods and properties.

Access Modifiers

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the same class
`protected`	The code is accessible within the same class, or in a class that is inherited from that class. You will learn more about inheritance in a later chapter
`internal`	The code is only accessible within its own assembly, but not from another assembly. You will learn more about this in a later chapter

- There's also two combinations: protected internal and private protected.
- For now, lets focus on `public` and `private` modifiers.

Private Modifier

If you declare a field with a `private` access modifier, it can only be accessed within the same class:

Example

```
class Car
{
  private string model = "Mustang";

  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
Mustang
```

Private Modifier

If you try to access it outside the class, an error will occur:

Example

```
class Car
{
  private string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
'Car.model' is inaccessible due to its protection level
The field 'Car.model' is assigned but its value is never used
```

Public Modifier

If you declare a field with a public access modifier, it is accessible for all classes:

Example

```
class Car
{
  public string model = "Mustang";
}

class Program
{
  static void Main(string[] args)
  {
    Car myObj = new Car();
    Console.WriteLine(myObj.model);
  }
}
```

The output will be:

```
Mustang
```

Why Access Modifiers?

- To control the visibility of class members (the security level of each individual class and class member).
- To achieve "**Encapsulation**" - which is the process of making sure that "sensitive" data is hidden from users. This is done by declaring fields as `private`.
- By default, all members of a class are `private` if you don't specify an access modifier:

Example

```
class Car
{
  string model;  // private
  string year;   // private
}
```

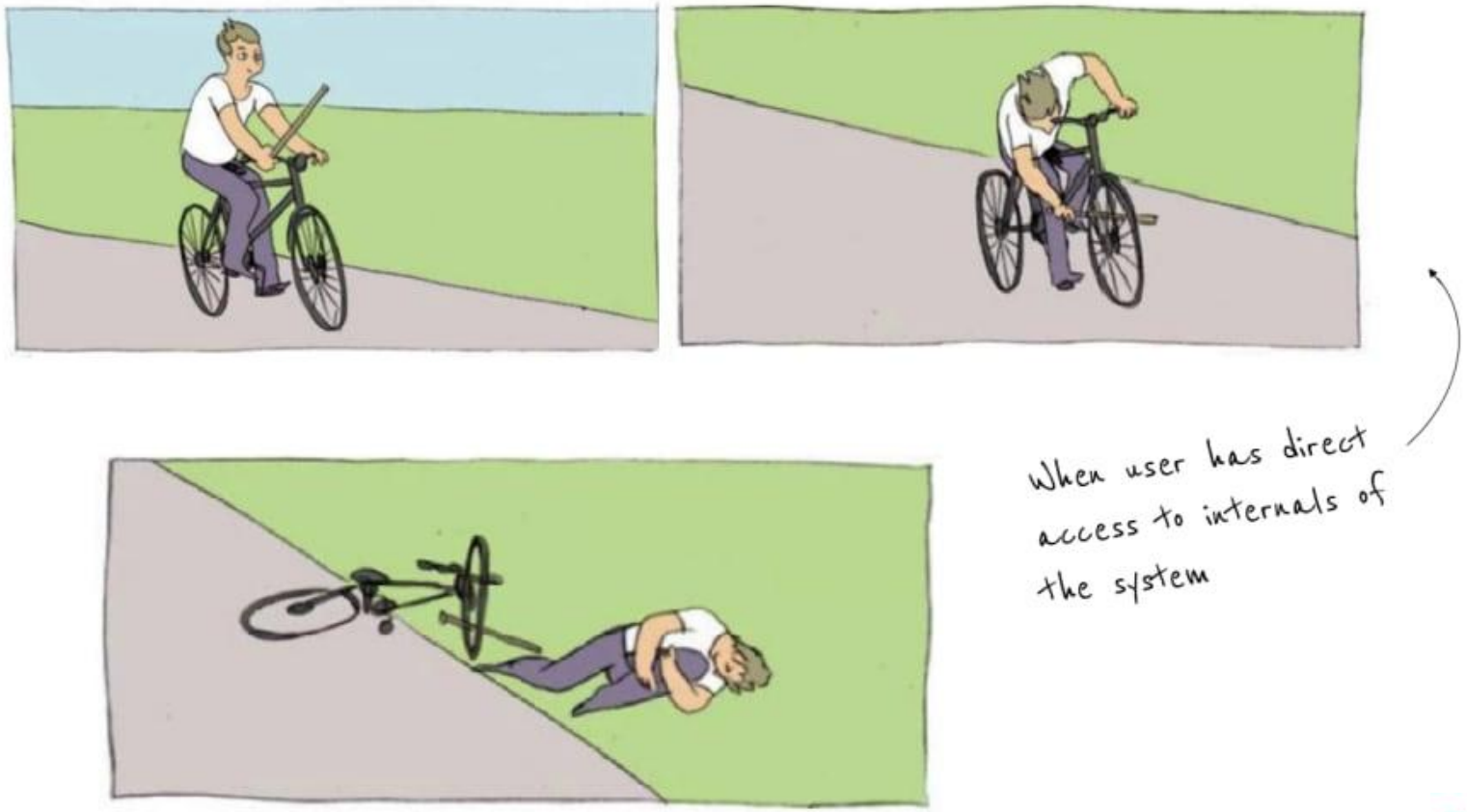
Properties and Encapsulation

Before we start to explain properties, you should have a basic understanding of "**Encapsulation**".

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- Declare fields/variables as private
- Provide public get and set methods, through **properties**, to access and update the value of a private field

Encapsulation



Properties

`Private` variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties.

A property is like a combination of a variable and a method, and it has two methods: a `get` and a `set` method:

Example

```
class Person
{
  private string name; // field

  public string Name   // property
  {
    get { return name; }   // get method
    set { name = value; }  // set method
  }
}
```

Properties

- The `Name` property is associated with the `name` field. It is a good practice to use the same name for both the property and the private field, but with an uppercase first letter.
- The get method returns the value of the variable `name`.
- The set method assigns a `value` to the `name` variable. The value keyword represents the `value` we assign to the property.

Example

```
class Person
{
  private string name; // field

  public string Name   // property
  {
    get { return name; }   // get method
    set { name = value; }  // set method
  }
}
```

Properties

Now we can use the `Name` property to access and update the `private` field of the `Person` class:

Example

```
class Person
{
  private string name; // field
  public string Name   // property
  {
    get { return name; }
    set { name = value; }
  }
}

class Program
{
  static void Main(string[] args)
  {
    Person myObj = new Person();
    myObj.Name = "Liam";
    Console.WriteLine(myObj.Name);
  }
}
```

The output will be:

```
Liam
```

Automatic Properties (Short Hand)

C# also provides a way to use short-hand / automatic properties, where you do not have to define the field for the property, and you only have to write `get`; and `set`; inside the property.

Example

Using automatic properties:

```
class Person
{
  public string Name  // property
  { get; set; }
}

class Program
{
  static void Main(string[] args)
  {
    Person myObj = new Person();
    myObj.Name = "Liam";
    Console.WriteLine(myObj.Name);
  }
}
```

The output will be:

```
Liam
```

Inheritance (Derived and Base Class)

In C#, it is possible to inherit fields and methods from one class to another. We group the "inheritance concept" into two categories:

- Derived Class (child) - the class that inherits from another class
- Base Class (parent) - the class being inherited from

To inherit from a class, use the `:` symbol.

Inheritance (Derived and Base Class)

In the example below, the `Car` class (child) inherits the fields and methods from the `Vehicle` class (parent):

Example

```
class Vehicle  // base class (parent)
{
  public string brand = "Ford";  // Vehicle field
  public void honk()             // Vehicle method
  {
    Console.WriteLine("Tuut, tuut!");
  }
}

class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}

class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand field (from the Vehicle class) and the value of the modelName from
    Console.WriteLine(myCar.brand + " " + myCar.modelName);
  }
}
```

Why and When to Use “Inheritance”

It is useful for code reusability: reuse fields and methods of an existing class when you create a new class.

Example

```
class Vehicle  // base class (parent)
{
  public string brand = "Ford";  // Vehicle field
  public void honk()             // Vehicle method
  {
    Console.WriteLine("Tuut, tuut!");
  }
}

class Car : Vehicle  // derived class (child)
{
  public string modelName = "Mustang";  // Car field
}

class Program
{
  static void Main(string[] args)
  {
    // Create a myCar object
    Car myCar = new Car();

    // Call the honk() method (From the Vehicle class) on the myCar object
    myCar.honk();

    // Display the value of the brand field (from the Vehicle class) and the value of the modelName from
    Console.WriteLine(myCar.brand + " " + myCar.modelName);
  }
}
```

The sealed Keyword

If you don't want other classes to inherit from a class, use the `sealed` keyword:

If you try to access a `sealed` class, C# will generate an error:

```
sealed class Vehicle
{
  ...
}

class Car : Vehicle
{
  ...
}
```

The error message will be something like this:

```
'Car': cannot derive from sealed type 'Vehicle'
```

Polymorphism and Overriding Methods

Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.

- Like we specified before; **Inheritance** lets us inherit fields and methods from another class.
- Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

Polymorphism and Overriding Methods

For example, think of a base class called `Animal` that has a method called `animalSound()`. Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

Example

```
class Animal  // Base class (parent)
{
  public void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Pig : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}
```

Polymorphism and Overriding Methods

Now we can create `Pig` and `Dog` objects and call the animalSound() method on both of them:

Example

```
class Animal  // Base class (parent)
{
  public void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Pig : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

The output will be:

```
The animal makes a sound
The animal makes a sound
The animal makes a sound
```

Not The Output I Was Looking For

The output from the example above was probably not what you expected. That is because the base class method overrides the derived class method, when they share the same name.

Polymorphism and Overriding Methods

Now we can create `Pig` and `Dog` objects and call the animalSound() method on both of them:

Example

```
class Animal  // Base class (parent)
{
  public virtual void animalSound()
  {
    Console.WriteLine("The animal makes a sound");
  }
}

class Pig : Animal  // Derived class (child)
{
  public override void animalSound()
  {
    Console.WriteLine("The pig says: wee wee");
  }
}

class Dog : Animal  // Derived class (child)
{
  public override void animalSound()
  {
    Console.WriteLine("The dog says: bow wow");
  }
}

class Program
{
  static void Main(string[] args)
  {
    Animal myAnimal = new Animal();  // Create a Animal object
    Animal myPig = new Pig();  // Create a Pig object
    Animal myDog = new Dog();  // Create a Dog object

    myAnimal.animalSound();
    myPig.animalSound();
    myDog.animalSound();
  }
}
```

The output will be:

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```

Not The Output I Was Looking For

The output from the example above was probably not what you expected. That is because the base class method overrides the derived class method, when they share the same name.

However, C# provides an option to override the base class method, by adding the `virtual` keyword to the method inside the base class, and by using the `override` keyword for each derived class methods:

Next lecture

- In the next lecture we continue focusing on «Quality of Software Applications»
- “Microsoft Visual C# Step by Step” Microsoft Press