Classes and Interfaces

Windows Programming Course

Agenda

1. Classes
2. Interface component: concept and interface implementation in C#

1.
Classes

Class members (Contents of Classes)

```
public class C {
   ... fields, constants ...          // for object-oriented programming
   ... methods ...
   ... constructors, destructors ...

   ... properties ...                 // for component-based programming
   ... events ...

   ... indexers ...                   // for amenity
   ... overloaded operators ...

   ... nested types (classes, interfaces, structs, enums, delegates)
   ...
}
```

Characteristics of Class in C#

- Objects are allocated on the heap (classes are reference types)
- Objects must be created with new
- Classes can inherit from one other class (single code inheritance)
- Classes can implement multiple interfaces (multiple type inheritance)

Access modifiers

Access modifiers can indicate the visibility of a class/method/property, such as `public` or `private`.

MODIFIER	APPLY TO	DESCRIPTION
`public`	Any types or members	The item is visible to any other code.
`protected`	Any member of a type, and any nested type	The item is visible only to any derived type.
`private`	Any member of a type, and any nested type	The item is visible only inside the type to which it belongs.

Access modifiers (cont.)

MODIFIER	APPLY TO	DESCRIPTION
`internal`	Any types or members	The item is visible only within its containing assembly.
`protected internal`	Any member of a type, and any nested type	The item is visible to any code within its containing assembly and to any code inside a derived type. Practically this means protected or internal, either `protected` (from any assembly) or `internal` (from within the assembly).
`private protected`	Any member of a type, and any nested type	Contrary to the access modifier `protected internal` which means either `protected` or `internal`, `private protected` combines `protected internal` with an and. Access is allowed only for derived types that are within the same assembly, but not from other assemblies. This access modifier is new with C# 7.2.

Fields

Fields are any variables associated with the class.

```
class PhoneCustomer {
  public const string DayOfSendingBill = "Monday";
  public int CustomerID;
  public string FirstName;
  public string LastName;
}
```

Readonly Fields

To guarantee that fields of an object cannot be changed, you can declare fields with the `readonly` modifier.

```
public class Document {
  private readonly DateTime _creationTime;
  public Document()
  {
    _creationTime = DateTime.Now;
  }
}
```

Properties

Declare a property:

```
private int _age;
public int Age
{
  get
  {
    return _age;
  }
  set
  {
    _age = value;
  }
}
```

Expression-Bodied Property Accessors (from C# 7):

```
private int _age;
public int Age
{
  get => _age;
  set => _age = value;
}
```

Auto-Implement Properties

```
public int Age { get; set; }

public int Age { get; set; } = 50;
```

Properties – Access Modifiers

C# allows the set and get accessors to have differing access modifiers. This would allow a property to have a public get and a private or protected set.

```
public string Name {
  get => _name;
  private set => _name = value;
}
public int Age { get; private set; }
```

Properties – Read-Only Properties

It is possible to create a read-only property by simply omitting the set accessor from the property definition.

```
private string _name;
public string Name
{
  get => _name;
}
```

Properties – Expression-Bodied Properties

It is possible to create a read-only property by simply omitting the set accessor from the property definition.

```
public class Person {
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public string FullName => $"{FirstName} {LastName}";
}
```

Methods

```
[modifiers] return_type MethodName([parameters])
{
  // Method body
}
```

e.g.:

```
public bool IsSquare(Rectangle rect)
{
  return (rect.Height == rect.Width);
}
// Expression-Bodied Methods
public bool IsSquare(Rectangle rect) => rect.Height == rect.Width;
```

Methods – Overloading

Methods of a class may have the same name:

- if they have different numbers of parameters, or
- if they have different parameter types, or
- if they have different parameter kinds (value, ref/out).

```
class ResultDisplayer {
  public void DisplayResult(string result)
  {
    Console.Writeln("Output: " + result);
  }
  public void DisplayResult(int result)
  {
    DisplayResult(result.ToString("D"));
  }
  public void DisplayResult(int result1, int result2)
  {
     DisplayResult((result1 + result2).ToString("D"));
  }
}
```

Methods – Optional Arguments

Parameters can also be optional by supplying a default value for optional parameters, which must be the last ones defined.

e.g.:

```
public void TestMethod(int notOptionalNumber, int optionalNumber = 42) {
  Console.WriteLine(optionalNumber + notOptionalNumber);
}

TestMethod(11); // 53
TestMethod(11, 22); // 33
```

Methods – Variable number of Arguments

Using optional arguments to define a variable number of arguments.

```
public void AnyNumberOfArguments(params int[] data) {
  foreach (var x in data) {
    Console.WriteLine(x);
  }
}
```

Constructors

The syntax for declaring basic constructors is a method that has the same name as the containing class and that does not have any return type:

- Constructors can be overloaded.
- A construcor may call another constructor with `this`.

```
public MyClass() : this(0) // zeroparameter constructor
{
  // construction code
}
public MyClass(int number) // another overload
{
  // construction code
}
```

Constructors – Default constructor

If no constructor was declared in a class, the compiler generates a parameterless default constructor:

```
class C { int x; }
C c = new C();      // ok
```

If a constructor was declared, **no** default constructor is generated:

```
class C {
  int x;
  public C(int y) { x = y; }
}
C c1 = new C();    // compilation error
C c2 = new C(3);   // ok
```

Constructors – Static constructors

Use static constructor if the class has some static fields or properties that need to be initialized from an external resource before the class is first used:

- Must be parameterless (also for structs) and have no `public` or `private` modifier.
- There must be just one static constructor per class/struct.
- Is invoked once before this type is used for the first time.

```
class MyClass {
  static MyClass()
  {
    // initialization code
  }
  // rest of class definition
}
```

2.
Interfaces

Interface

- Interface = purely abstract class; only signatures, no implementation.
- May contain methods, properties, indexers and events
  (no fields, constants, constructors, destructors, operators, nested types).
- Interface members are implicitly public abstract (virtual).
- Interface members must not be static.
- Classes and structs may implement multiple interfaces.
- Interfaces can extend other interfaces.

```
public interface IShape {
   void Draw(int x, int y); // methods
   string Name { get; } // property
}
```

Implementing

```
public interface IShape {
   void Draw(int x, int y); // methods
   string Name { get; } // property
}

public class Rectangle : IShape {
   public void Draw(int x, int y) {
      Console.Writeln(string.Format("Drawing a rectangle ({0}-{1})", x, y));
   }
   public string Name {
      get {
         return "Rectangle";
   }
}
public class Circle : IShape {
   public void Draw(int x, int y) {
      Console.Writeln(string.Format("Drawing a circle ({0}-{1})", x, y));
   }
   public string Name => "Circle";
}
```

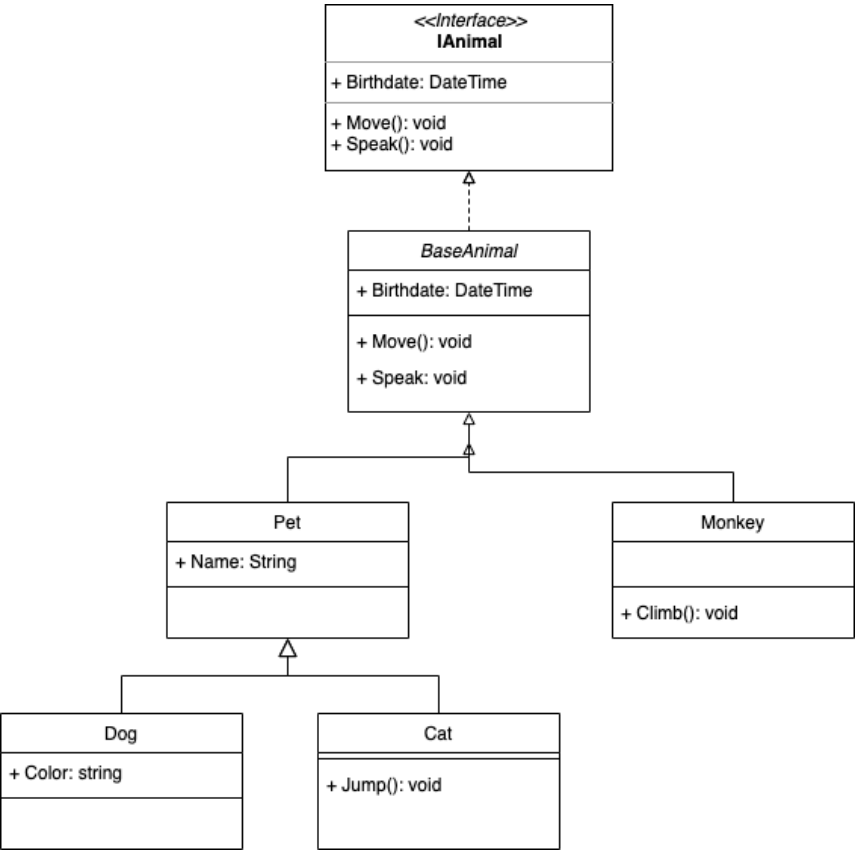
Using

```
var shapes = new ArrayList();
// Assignment
shapes.Add(new Retangle());
Shapes.Add(new Circle());

// Method calls
for (int i = 0; i < shapes.length; i++) {
 shapes[i].Draw((i+1)*10, (i+1)*10);
}

// Type check
int circleTotal = 0;
for (int i = 0; i < shapes.length; i++) {
  if (shapes[i] is Circle) circleTotal++;
}
Console.Writeln(string.Format("There is {0} circles in the list", circleTotal));
```

Hands-on Exercise



Create a console application:

- Create classes/interface follow the diagram
- Implement methods Speak/Move (Write to console like the slide 24)
- In the method Program.Main, create some animals and call methods Move/Speak (ref slide 25)
- Try to use `List<IAnimal>` instead of array `IShape[]`

Thanks!

Any questions?

You can find me at:

tranminhphuoc@gmail.com