C# : Methods, Properties, Events

Members Only

C# Type Members

- Methods
- Fields & Properties
- Events
- Operators
- Indexers
- Constructors and Destructors

Methods

- **Methods define behavior**
- **Every method has a return type**
  - void if not value returned
- **Every method has zero or more parameters**
  - Use params keyword to accept a variable number of parameters
- **Every method has a signature**
  - Name of method + parameters (type and modifiers are significant)

```
public void WriteAsBytes(int value)
{
      byte[] bytes = BitConverter.GetBytes(value);

      foreach(byte b in bytes)
      {
          Console.Write("0x{0:X2} ", b);
      }
}
```

Methods - Review

- **Instance methods versus static methods**
  - Instance methods invoked via object, static methods via type
- **Abstract methods**
  - Provide no implementation, implicitly virtual
- **Virtual methods**
  - Can override in a derived class
- **Partial methods**
  - Part of a partial class
- **Extension methods**
  - Described in the LINQ module

Method Overloading

- **Define multiple methods with the same name in a single class**
  - Methods require a unique signature
- **Compiler finds and invokes the best match**

```
public void WriteAsBytes(int value)
{
    // ...
}

public void WriteAsBytes(double value)
{
    // ...
}
```

Fields

- **Fields are variables of a class**
  - Static fields and instance fields
- **Read-only fields**
  - Can only assign values in the declaration or in a constructor

```
public class Animal
{
    private readonly string _name;

    public Animal(string name)
    {
        _name = name;
    }
}
```

Properties

- **Like fields, but do not denote a storage location**
  - Every property defines a get and/or a set accessor
  - Often used to expose and control fields
  - Access level for get and set are independent
- **Automatically implemented properties use a hidden field**
  - Only accessible via property

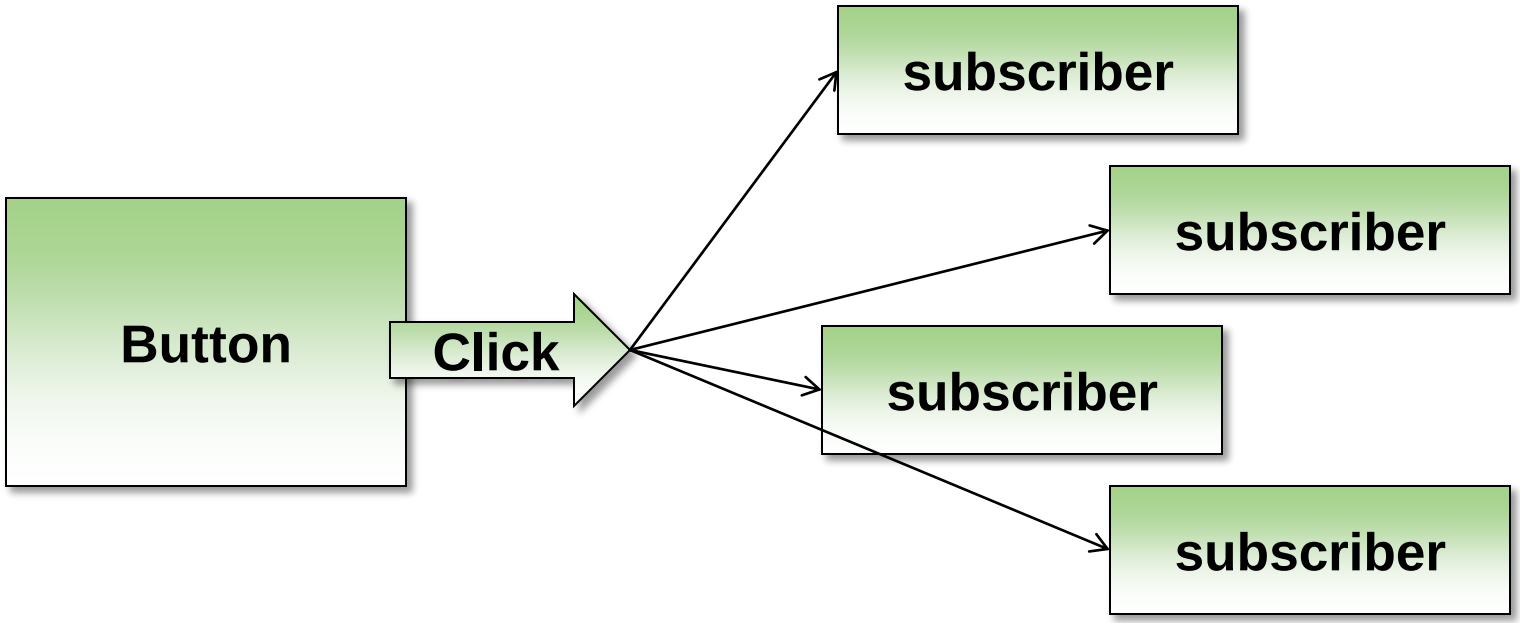
```
public string Name
{
    get;
    set;
}
```

```
private string _name;

public string Name
{
    get { return _name; }
    set
    {
        if(!String.IsNullOrEmpty(value))
        {
            _name = value;
        }
    }
}
```

Events

- **Allows a class to send notifications to other classes or objects**
  - Publisher raises the event
  - One or more subscribers process the event



Delegates

- **A delegate is a type that references methods**
  - Similar to a function pointer, but type safe
  - Can invoke methods via a delegate
  - Ideal for callback methods and events

```
public delegate void WriteMessage(string message);
```

```
Logger logger = new Logger();
WriteMessage write = new WriteMessage(logger.WriteMessage);
write("Success!!");
```

```
public class Logger
{
    public void WriteMessage(String message)
    {
        Console.WriteLine(message);
    }
}
```

Subscribing To Events

- **Use the += and -= to attach and detach event handlers**
  - Can attached named or anonymous methods

```
public static void Initialize()
{
    _submitButton.Click += new RoutedEventHandler(_submitButton_Click);
}

static void _submitButton_Click(object sender, RoutedEventArgs e)
{
    // ... respond to event
}
```

Publishing Events

- **Create custom event arguments (or use a built-in type)**
  - Always derive from the base EventArgs class
- **Define a delegate (or use a built-in delegate)**
- **Define an event in your class**
- **Raise the event at the appropriate time**

```
public event NameChangingEventHandler NameChanging;

private bool OnNameChanging(string oldName, string newName)
{
    if(NameChanging != null)
    {
        NameChangingEventArgs args = new NameChangingEventArgs();
        args.Cancel = false;
        args.NewName = newName;
        args.OldName = oldName;
        NameChanging(this, args);
    }
}
```

Indexers

- **Enables indexing of an object (like an array)**

```
public class Zoo
{
    public Animal this[int index]
    {
        get { return _animals[index]; }
        set { _animals[index] = value; }
    }

    private Animal[] _animals;
}
```

```
Animal FindFirstAnimal(Zoo zoo)
{
    Animal first = zoo[0];
    return first;
}
```

Operator Overloading

- You can overload most unary and binary operators
  - + - * / ++ -- == < >
- Overload using static methods
- Caution - use the principle of least surprise

```
public struct Complex
{
    public int real;
    public int imaginary;

    public static Complex operator +(Complex c1, Complex c2)
    {
        return new Complex(c1.real + c2.real,
                           c1.imaginary + c2.imaginary);
    }
     …
}
```

```
Complex c1 = new Complex(2, -1);
Complex c2 = new Complex(-1, 2);

Complex c3 = c1 + c2;
```

Conversion Operators

- **Convert an object from one type to the other**
- **Implicit or explicit conversion operators**
  - Explicit operators require a type cast
  - Compiler will automatically find implicit operators

```
public struct Price
{
    public static implicit operator Price(int value)
    {
        Price price = new Price();
        price._value = value;
        return price;
    }

    private int _value;
}
```

```
Price p1 = 3;
Price p2 = (Price)3;
```

Constructors

- **Instance constructors and static constructor**
- **Can overload instance constructors**
  - No return type
  - Not inherited
  - Use this keyword or base keyword to pass control to another ctor
- **Variable initializers**
  - Easy syntax to create variable and initialize properties

```
NameChangingEventArgs args = new NameChangingEventArgs
{
    Cancel = false,
    NewName = newName,
    OldName = oldName
};
```

Destructor

- **Used to cleanup an object instance**
  - Cannot overload a class destructor
  - Cannot explicitly invoke a destructor
- **Use destructors to release unmanaged resources**
  - You should always implement IDisposable
  - See http://msdn.microsoft.com/en-us/library/fs2xkftw(VS.80).aspx

```
~Animal()
{
    // clean up unmanaged resources
}
```

Summary

- **Members are used to craft an abstraction**
  - Fields and properties for state
  - Methods for behavior
  - Events for notification
- **Overload operators with caution!**