

# Advanced C#

Mark Sapossnek

CS 594  
Computer Science Department  
Metropolitan College  
Boston University

## Prerequisites

- ◆ This module assumes that you understand the fundamentals of
  - Programming
    - Variables, statements, functions, loops, etc.
  - Object-oriented programming
    - Classes, inheritance, polymorphism, members, etc.
    - C++ or Java
  - Introduction to C#

## Learning Objectives

- ◆ Advanced features of the C# language
  - Creating custom types with interfaces, classes and structs
  - Delegates and events
  - Miscellaneous topics

## Agenda

- ◆ **Review Object-Oriented Concepts**
- ◆ Interfaces
- ◆ Classes and Structs
- ◆ Delegates
- ◆ Events
- ◆ Attributes
- ◆ Preprocessor Directives
- ◆ XML Comments
- ◆ Unsafe Code

## Review Key Object-Oriented Concepts

- ◆ Objects, instances and classes
- ◆ Identity
  - Every instance has a unique identity, regardless of its data
- ◆ Encapsulation
  - Data and function are packaged together
  - Information hiding
  - An object is an abstraction
    - User should NOT know implementation details

## Review Key Object-Oriented Concepts

- ◆ Interfaces
  - A well-defined contract
  - A set of function members
- ◆ Types
  - An object has a type, which specifies its interfaces and their implementations
  - A variable also can have a type
- ◆ Inheritance
  - Types are arranged in a hierarchy
    - Base/derived, superclass/subclass
  - Interface vs. implementation inheritance

## Review Key Object-Oriented Concepts

- ◆ Polymorphism
  - The ability to use an object without knowing its precise type
  - Three main kinds of polymorphism
    - Inheritance
    - Interfaces
    - Late binding
- ◆ Dependencies
  - For reuse and to facilitate development, systems should be loosely coupled
  - Dependencies should be minimized

## Agenda

- ◆ Review Object-Oriented Concepts
- ◆ **Interfaces**
- ◆ Classes and Structs
- ◆ Delegates
- ◆ Events
- ◆ Attributes
- ◆ Preprocessor Directives
- ◆ XML Comments
- ◆ Unsafe Code

## Interfaces

- ◆ An interface defines a contract
  - An interface is a type
  - Includes methods, properties, indexers, events
  - Any class or struct implementing an interface must support all parts of the contract
- ◆ Interfaces provide no implementation
  - When a class or struct implements an interface it must provide the implementation
- ◆ Interfaces provide polymorphism
  - Many classes and structs may implement a particular interface

## Interfaces Example

```
public interface IDelete {  
    void Delete();  
}  
public class TextBox : IDelete {  
    public void Delete() { ... }  
}  
public class Car : IDelete {  
    public void Delete() { ... }  
}
```

```
TextBox tb = new TextBox();  
IDelete iDel = tb;  
iDel.Delete();
```

```
Car c = new Car();  
iDel = c;  
iDel.Delete();
```

## Interfaces Multiple Inheritance

- ◆ Classes and structs can inherit from multiple interfaces
- ◆ Interfaces can inherit from multiple interfaces

```
interface IControl {  
    void Paint();  
}  
interface IListBox: IControl {  
    void SetItems(string[] items);  
}  
interface IComboBox: ITextBox, IListBox {  
}
```

## Interfaces Explicit Interface Members

- ◆ If two interfaces have the same method name, you can explicitly specify interface + method name to disambiguate their implementations

```
interface IControl {  
    void Delete();  
}  
interface IListBox: IControl {  
    void Delete();  
}  
interface IComboBox: ITextBox, IListBox {  
    void IControl.Delete();  
    void IListBox.Delete();  
}
```

## Agenda

- ◆ Review Object-Oriented Concepts
- ◆ Interfaces
- ◆ **Classes and Structs**
- ◆ Delegates
- ◆ Events
- ◆ Attributes
- ◆ Preprocessor Directives
- ◆ XML Comments
- ◆ Unsafe Code

## Classes and Structs Similarities

- ◆ Both are user-defined types
- ◆ Both can implement multiple interfaces
- ◆ Both can contain
  - Data
    - Fields, constants, events, arrays
  - Functions
    - Methods, properties, indexers, operators, constructors
  - Type definitions
    - Classes, structs, enums, interfaces, delegates

## Classes and Structs Differences

Class	Struct
Reference type	Value type
Can inherit from any non-sealed reference type	No inheritance (inherits only from <code>System.ValueType</code> )
Can have a destructor	No destructor
Can have user-defined parameterless constructor	No user-defined parameterless constructor

## Classes and Structs C# Structs vs. C++ Structs

- ◆ Very different from C++ struct

C++ Struct	C# Struct
Same as C++ class, but all members are <code>public</code>	User-defined value type
Can be allocated on the heap, on the stack or as a member (can be used as value or reference)	Always allocated on the stack or as a member
Members are always <code>public</code>	Members can be <code>public</code> , <code>internal</code> or <code>private</code>

## Classes and Structs Class

```
public class Car : Vehicle {
    public enum Make { GM, Honda, BMW }
    Make make;
    string vid;
    Point location;
    Car(Make m, string vid; Point loc) {
        this.make = m;
        this.vid = vid;
        this.location = loc;
    }
    public void Drive() {
        Console.WriteLine("vroom"); }
}
```

```
Car c =
    new Car(Car.Make.BMW,
            "JF3559QT98",
            new Point(3,7));
c.Drive();
```

## Classes and Structs Struct

```
public struct Point {
    int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int X { get { return x; }
                  set { x = value; } }
    public int Y { get { return y; }
                  set { y = value; } }
}
```

```
Point p = new Point(2,5);
p.X += 100;
int px = p.X; // px = 102
```

## Classes and Structs Static vs. Instance Members

- ◆ By default, members are per instance
  - Each instance gets its own fields
  - Methods apply to a specific instance
- ◆ Static members are per type
  - Static methods can't access instance data
  - No `this` variable in static methods
- ◆ Don't abuse static members
  - They are essentially object-oriented global data and global functions

## Classes and Structs Access Modifiers

- ◆ Access modifiers specify who can use a type or a member
- ◆ Access modifiers control encapsulation
- ◆ Top-level types (those directly in a namespace) can be `public` or `internal`
- ◆ Class members can be `public`, `private`, `protected`, `internal`, or `protected internal`
- ◆ Struct members can be `public`, `private` or `internal`

## Classes and Structs Access Modifiers

If the access modifier is	Then a member defined in type T and assembly A is accessible
<code>public</code>	to everyone
<code>private</code>	within T only (the default)
<code>protected</code>	to T or types derived from T
<code>internal</code>	to types within A
<code>protected internal</code>	to T or types derived from T or to types within A

## Classes and Structs Abstract Classes

- ◆ An abstract class is one that cannot be instantiated
- ◆ Intended to be used as a base class
- ◆ May contain abstract and non-abstract function members
- ◆ Similar to an interface
- ◆ Cannot be sealed

## Classes and Structs Sealed Classes

- ◆ A sealed class is one that cannot be used as a base class
- ◆ Sealed classes can't be abstract
- ◆ All structs are implicitly sealed
- ◆ Why seal a class?
  - To prevent unintended derivation
  - Code optimization
    - Virtual function calls can be resolved at compile-time

## Classes and Structs `this`

- ◆ The `this` keyword is a predefined variable available in non-static function members
  - Used to access data and function members unambiguously

```
class Person {
    string name;
    public Person(string name) {
        this.name = name;
    }
    public void Introduce(Person p) {
        if (p != this)
            Console.WriteLine("Hi, I'm " + name);
    }
}
```

## Classes and Structs

### base

- ◆ The `base` keyword is used to access class members that are hidden by similarly named members of the current class

```
class Shape {
    int x, y;
    public override string ToString() {
        return "x=" + x + ",y=" + y;
    }
}
class Circle : Shape {
    int r;
    public override string ToString() {
        return base.ToString() + ",r=" + r;
    }
}
```

## Classes and Structs

### Constants

- ◆ A constant is a data member that is evaluated at compile-time and is implicitly static (per type)
  - e.g. `Math.PI`

```
public class MyClass {
    public const string version = "1.0.0";
    public const string s1 = "abc" + "def";
    public const int i3 = 1 + 2;
    public const double PI_I3 = i3 * Math.PI;
    public const double s = Math.Sin(Math.PI); //ERROR
    ...
}
```

## Classes and Structs

### Fields

- ◆ A field is a member variable
- ◆ Holds data for a class or struct
- ◆ Can hold:
  - a class instance (a reference),
  - a struct instance (actual data), or
  - an array of class or struct instances (an array is actually a reference)

## Classes and Structs

### Readonly Fields

- ◆ Similar to a `const`, but is initialized at run-time in its declaration or in a constructor
  - Once initialized, it cannot be modified
- ◆ Differs from a constant
  - Initialized at run-time (vs. compile-time)
    - Don't have to re-compile clients
  - Can be static or per-instance

```
public class MyClass {
    public static readonly double d1 = Math.Sin(Math.PI);
    public readonly string s1;
    public MyClass(string s) { s1 = s; }
}
```

## Classes and Structs

### Properties

- ◆ A property is a virtual field
- ◆ Looks like a field, but is implemented with code

```
public class Button: Control {
    private string caption;
    public string Caption {
        get { return caption; }
        set { caption = value;
            Repaint(); }
    }
}
```

- ◆ Can be read-only, write-only, or read/write

```
Button b = new Button();
b.Caption = "OK";
String s = b.Caption;
```

## Classes and Structs

### Indexers

- ◆ An indexer lets an instance behave as a virtual array
- ◆ Can be overloaded (e.g. index by `int` and by `string`)

```
public class ListBox: Control {
    private string[] items;
    public string this[int index] {
        get { return items[index]; }
        set { items[index] = value;
            Repaint(); }
    }
}
```

- ◆ Can be read-only, write-only, or read/write

```
ListBox listBox = new ListBox();
listBox[0] = "hello";
Console.WriteLine(listBox[0]);
```

## Classes and Structs Methods

- ◆ All code executes in a method
  - Constructors, destructors and operators are special types of methods
  - Properties and indexers are implemented with get/set methods
- ◆ Methods have argument lists
- ◆ Methods contain statements
- ◆ Methods can return a value
  - Only if return type is not `void`

## Classes and Structs Method Argument Passing

- ◆ By default, data is passed by value
- ◆ A copy of the data is created and passed to the method
- ◆ For value types, variables cannot be modified by a method call
- ◆ For reference types, the instance can be modified by a method call, but the variable itself cannot be modified by a method call

## Classes and Structs Method Argument Passing

- ◆ The `ref` modifier causes arguments to be passed by reference
- ◆ Allows a method call to modify a variable
- ◆ Have to use `ref` modifier in method definition and the code that calls it
- ◆ Variable has to have a value before call

```
void RefFunction(ref int p) {  
    p++;  
}  
  
int x = 10;  
RefFunction(ref x);  
// x is now 11
```

## Classes and Structs Method Argument Passing

- ◆ The `out` modifier causes arguments to be passed out by reference
- ◆ Allows a method call to initialize a variable
- ◆ Have to use `out` modifier in method definition and the code that calls it
- ◆ Argument has to have a value before returning

```
void OutFunction(out int p) {  
    p = 22;  
}  
  
int x;  
OutFunction(out x);  
// x is now 22
```

## Classes and Structs Overloaded Methods

- ◆ A type may overload methods, i.e. provide multiple methods with the same name
- ◆ Each must have a unique signature
- ◆ Signature is based upon arguments only, the return value is ignored

```
void Print(int i);  
void Print(string s);  
void Print(char c);  
void Print(float f);  
int Print(float f); // Error: duplicate signature
```

## Classes and Structs Parameter Arrays

- ◆ Methods can have a variable number of arguments, called a parameter array
- ◆ `params` keyword declares parameter array
- ◆ Must be last argument

```
int Sum(params int[] intArr) {  
    int sum = 0;  
    foreach (int i in intArr)  
        sum += i;  
    return sum;  
}  
  
int sum = Sum(13,87,34);
```

## Classes and Structs Virtual Methods

- ♦ Methods may be virtual or non-virtual (default)
- ♦ Non-virtual methods are not polymorphic
  - They cannot be overridden
- ♦ Non-virtual methods cannot be abstract

```
class Foo {  
    public void DoSomething(int i) {  
        ...  
    }  
}
```

```
Foo f = new Foo();  
f.DoSomething();
```

## Classes and Structs Virtual Methods

- ♦ Defined in a base class
- ♦ Can be overridden in derived classes
  - Derived classes provide their own specialized implementation
- ♦ May contain a default implementation
  - Use abstract method if no default implementation
- ♦ A form of polymorphism
- ♦ Properties, indexers and events can also be virtual

## Classes and Structs Virtual Methods

```
class Shape {  
    public virtual void Draw() { ... }  
}  
class Box : Shape {  
    public override void Draw() { ... }  
}  
class Sphere : Shape {  
    public override void Draw() { ... }  
}
```

```
void HandleShape(Shape s) {  
    s.Draw();  
    ...  
}
```

```
HandleShape(new Box());  
HandleShape(new Sphere());  
HandleShape(new Shape());
```

## Classes and Structs Abstract Methods

- ♦ An abstract method is virtual and has no implementation
- ♦ Must belong to an abstract class
- ♦ Intended to be implemented in a derived class

## Classes and Structs Abstract Methods

```
abstract class Shape {  
    public abstract void Draw();  
}  
class Box : Shape {  
    public override void Draw() { ... }  
}  
class Sphere : Shape {  
    public override void Draw() { ... }  
}
```

```
void HandleShape(Shape s) {  
    s.Draw();  
    ...  
}
```

```
HandleShape(new Box());  
HandleShape(new Sphere());  
HandleShape(new Shape()); // Error!
```

## Classes and Structs Method Versioning

- ♦ Must explicitly use `override` or `new` keywords to specify versioning intent
- ♦ Avoids accidental overriding
- ♦ Methods are non-virtual by default
- ♦ C++ and Java product fragile base classes – cannot specify versioning intent

## Classes and Structs Method Versioning

```
class Base { // version 2
    public virtual void Foo() {
        Console.WriteLine("Base.Foo");
    }
}
```

```
class Derived: Base { // version 2b
    public override void Foo() {
        base.Foo();
        Console.WriteLine("Derived.Foo");
    }
}
```

## Classes and Structs Constructors

- ◆ Instance constructors are special methods that are called when a class or struct is instantiated
- ◆ Performs custom initialization
- ◆ Can be overloaded
- ◆ If a class doesn't define any constructors, an implicit parameterless constructor is created
- ◆ Cannot create a parameterless constructor for a struct
  - All fields initialized to zero/null

## Classes and Structs Constructor Initializers

- ◆ One constructor can call another with a constructor initializer
- ◆ Can call `this(...)` or `base(...)`
- ◆ Default constructor initializer is `base()`

```
class B {
    private int h;
    public B() { }
    public B(int h) { this.h = h; }
}
class D : B {
    private int i;
    public D() : this(24) { }
    public D(int i) { this.i = i; }
    public D(int h, int i) : base(h) { this.i = i; }
}
```

## Classes and Structs Static Constructors

- ◆ A static constructor lets you create initialization code that is called once for the class
- ◆ Guaranteed to be executed before the first instance of a class or struct is created and before any static member of the class or struct is accessed
- ◆ No other guarantees on execution order
- ◆ Only one static constructor per type
- ◆ Must be parameterless

## Classes and Structs Destructors

- ◆ A destructor is a method that is called before an instance is garbage collected
- ◆ Used to clean up any resources held by the instance, do bookkeeping, etc.
- ◆ Only classes, not structs can have destructors

```
class Foo {
    ~Foo() {
        Console.WriteLine("Destroyed {0}", this);
    }
}
```

## Classes and Structs Destructors

- ◆ Unlike C++, C# destructors are non-deterministic
- ◆ They are not guaranteed to be called at a specific time
- ◆ They are guaranteed to be called before shutdown
- ◆ Use the `using` statement and the `IDisposable` interface to achieve deterministic finalization



## Classes and Structs Operator Overloading

- ◆ User-defined operators
- ◆ Must be a static method

```
class Car {  
    string vid;  
    public static bool operator ==(Car x, Car y) {  
        return x.vid == y.vid;  
    }  
}
```

## Classes and Structs Operator Overloading

- ◆ Overloadable unary operators

+	-	!	~
true	false	++	--

- ◆ Overloadable binary operators

+	-	*	/	!	~
%	&		^	==	!=
<<	>>	<	>	<=	>=

## Classes and Structs Operator Overloading

- ◆ No overloading for member access, method invocation, assignment operators, nor these operators: sizeof, new, is, as, typeof, checked, unchecked, &&, ||, and ?:
- ◆ The && and || operators are automatically evaluated from & and |
- ◆ Overloading a binary operator (e.g. \*) implicitly overloads the corresponding assignment operator (e.g. \*=)

## Classes and Structs Operator Overloading

```
struct Vector {  
    int x, y;  
    public Vector(x, y) { this.x = x; this.y = y; }  
    public static Vector operator +(Vector a, Vector b) {  
        return Vector(a.x + b.x, a.y + b.y);  
    }  
    ...  
}
```

## Classes and Structs Conversion Operators

- ◆ User-defined explicit and implicit conversions

```
class Note {  
    int value;  
    // Convert to hertz - no loss of precision  
    public static implicit operator double(Note x) {  
        return ...;  
    }  
    // Convert to nearest note  
    public static explicit operator Note(double x) {  
        return ...;  
    }  
}
```

```
Note n = (Note)442.578;  
double d = n;
```

## Classes and Structs Implementing Interfaces

- ◆ Classes and structs can implement multiple interfaces
- ◆ A class or struct that inherits from an interface must implement all function members defined in that interface

## Classes and Structs Implementing Interfaces

```
public interface IDelete {
    void Delete();
}
public class TextBox : IDelete {
    public void Delete() { ... }
}
public class Car : IDelete {
    public void Delete() { ... }
}
```

```
TextBox tb = new TextBox();
IDelete iDel = tb;
iDel.Delete();

Car c = new Car();
iDel = c;
iDel.Delete();
```

## Classes and Structs Implementing Interfaces

- ◆ Explicit interface implementation
- ◆ Handles name collisions

```
public interface IDelete {
    void Delete();
}
public interface IFoo {
    void Delete();
}
public class TextBox : IDelete, IFoo {
    public void IDelete.Delete() { ... }
    public void IFoo.Delete() { ... }
}
```

## Classes and Structs Nested Types

- ◆ Declared within the scope of another type
- ◆ Nesting a type provides three benefits:
  - Nested type can access all the members of its enclosing type, regardless of access modifier
  - Nested type can be hidden from other types
  - Accessing a nested type from outside the enclosing type requires specifying the type name
- ◆ Nested types can be declared `new` to hide inherited types
- ◆ Unlike Java inner classes, nested types imply no relationship between instances

## Classes and Structs `is` Operator

- ◆ The `is` operator is used to dynamically test if the run-time type of an object is compatible with a given type

```
static void DoSomething(object o) {
    if (o is Car)
        ((Car)o).Drive();
}
```

- ◆ Don't abuse the `is` operator: it is preferable to design an appropriate type hierarchy with polymorphic methods

## Classes and Structs `as` Operator

- ◆ The `as` operator tries to convert a variable to a specified type; if no such conversion is possible the result is `null`

```
static void DoSomething(object o) {
    Car c = o as Car;
    if (c != null) c.Drive();
}
```

- ◆ More efficient than using `is` operator: test and convert in one operation
- ◆ Same design warning as with the `is` operator

## Classes and Structs `typeof` Operator

- ◆ The `typeof` operator returns the `System.Type` object for a specified type
- ◆ Can then use reflection to dynamically obtain information about the type

```
Console.WriteLine(typeof(int).FullName);
Console.WriteLine(typeof(System.Int).Name);
Console.WriteLine(typeof(float).Module);
Console.WriteLine(typeof(double).IsPublic);
Console.WriteLine(typeof(Car).MemberType);
```

## Agenda

- ◆ Review Object-Oriented Concepts
- ◆ Interfaces
- ◆ Classes and Structs
- ◆ **Delegates**
- ◆ Events
- ◆ Attributes
- ◆ Preprocessor Directives
- ◆ XML Comments
- ◆ Unsafe Code

## Delegates Overview

- ◆ A delegate is a reference type that defines a method signature
- ◆ A delegate instance holds one or more methods
  - Essentially an “object-oriented function pointer”
  - Methods can be static or non-static
  - Methods can return a value
- ◆ Provides polymorphism for individual functions
- ◆ Foundation for event handling

## Delegates Overview

```
delegate double Del(double x); // Declare

static void DemoDelegates() {
    Del delInst = new Del(Math.Sin); // Instantiate
    double x = delInst(1.0); // Invoke
}
```

## Delegates Multicast Delegates

- ◆ A delegate can hold and invoke multiple methods
  - Multicast delegates must contain only methods that return `void`, else there is a run-time exception
- ◆ Each delegate has an invocation list
  - Methods are invoked sequentially, in the order added
- ◆ The `+=` and `-=` operators are used to add and remove delegates, respectively
- ◆ `+=` and `-=` operators are thread-safe

## Delegates Multicast Delegates

```
delegate void SomeEvent(int x, int y);
static void Foo1(int x, int y) {
    Console.WriteLine("Foo1");
}
static void Foo2(int x, int y) {
    Console.WriteLine("Foo2");
}
public static void Main() {
    SomeEvent func = new SomeEvent(Foo1);
    func += new SomeEvent(Foo2);
    func(1,2); // Foo1 and Foo2 are called
    func -= new SomeEvent(Foo1);
    func(2,3); // Only Foo2 is called
}
```

## Delegates and Interfaces

- ◆ Could always use interfaces instead of delegates
- ◆ Interfaces are more powerful
  - Multiple methods
  - Inheritance
- ◆ Delegates are more elegant for event handlers
  - Less code
  - Can easily implement multiple event handlers on one class/struct

## Agenda

- ◆ Review Object-Oriented Concepts
- ◆ Interfaces
- ◆ Classes and Structs
- ◆ Delegates
- ◆ **Events**
- ◆ Attributes
- ◆ Preprocessor Directives
- ◆ XML Comments
- ◆ Unsafe Code

## Events Overview

- ◆ Event handling is a style of programming where one object notifies another that something of interest has occurred
  - A publish-subscribe programming model
- ◆ Events allow you to tie your own code into the functioning of an independently created component
- ◆ Events are a type of “callback” mechanism

## Events Overview

- ◆ Events are well suited for user-interfaces
  - The user does something (clicks a button, moves a mouse, changes a value, etc.) and the program reacts in response
- ◆ Many other uses, e.g.
  - Time-based events
  - Asynchronous operation completed
  - Email message has arrived
  - A web session has begun

## Events Overview

- ◆ C# has native support for events
- ◆ Based upon delegates
- ◆ An event is essentially a field holding a delegate
- ◆ However, public users of the class can only register delegates
  - They can only call += and -=
  - They can't invoke the event's delegate
- ◆ Multicast delegates allow multiple objects to register with the same event

## Events Example: Component-Side

- ◆ Define the event signature as a delegate

```
public delegate void EventHandler(object sender, EventArgs e);
```

- ◆ Define the event and firing logic

```
public class Button {
    public event EventHandler Click;

    protected void OnClick(EventArgs e) {
        // This is called when button is clicked
        if (Click != null) Click(this, e);
    }
}
```

## Events Example: User-Side

- ◆ Define and register an event handler

```
public class MyForm: Form {
    Button okButton;

    static void OkClicked(object sender, EventArgs e) {
        ShowMessage("You pressed the OK button");
    }

    public MyForm() {
        okButton = new Button(...);
        okButton.Caption = "OK";
        okButton.Click += new EventHandler(OkClicked);
    }
}
```

## Agenda

- ◆ Review Object-Oriented Concepts
- ◆ Interfaces
- ◆ Classes and Structs
- ◆ Delegates
- ◆ Events
- ◆ **Attributes**
- ◆ Preprocessor Directives
- ◆ XML Comments
- ◆ Unsafe Code

## Attributes Overview

- ◆ It's often necessary to associate information (metadata) with types and members, e.g.
  - Documentation URL for a class
  - Transaction context for a method
  - XML persistence mapping
  - COM ProgID for a class
- ◆ Attributes allow you to decorate a code element (assembly, module, type, member, return value and parameter) with additional information

## Attributes Overview

```
[HelpUrl("http://SomeUrl/APIDocs/SomeClass")]
class SomeClass {
    [Obsolete("Use SomeNewMethod instead")]
    public void SomeOldMethod() {
        ...
    }

    public string Test([SomeAttr()] string param1) {
        ...
    }
}
```

## Attributes Overview

- ◆ Attributes are superior to the alternatives
  - Modifying the source language
  - Using external files, e.g., .IDL, .DEF
- ◆ Attributes are extensible
  - Attributes allow you to add information not supported by C# itself
  - Not limited to predefined information
- ◆ Built into the .NET Framework, so they work across all .NET languages
  - Stored in assembly metadata

## Attributes Overview

- ◆ Some predefined .NET Framework attributes

Attribute Name	Description
Browsable	Should a property or event be displayed in the property window
Serializable	Allows a class or struct to be serialized
Obsolete	Compiler will complain if target is used
ProgId	COM Prog ID
Transaction	Transactional characteristics of a class

## Attributes Overview

- ◆ Attributes can be
  - Attached to types and members
  - Examined at run-time using reflection
- ◆ Completely extensible
  - Simply a class that inherits from `System.Attribute`
- ◆ Type-safe
  - Arguments checked at compile-time
- ◆ Extensive use in .NET Framework
  - XML, Web Services, security, serialization, component model, COM and P/Invoke interop, code configuration...

## Attributes Querying Attributes

```
[HelpUrl("http://SomeUrl/MyClass")]
class Class1 {}
[HelpUrl("http://SomeUrl/MyClass"),
 HelpUrl("http://SomeUrl/MyClass", Tag="ctor")]
class Class2 {}
```

```
Type type = typeof(MyClass);
foreach (object attr in type.GetCustomAttributes() ) {
    if ( attr is HelpUrlAttribute ) {
        HelpUrlAttribute ha = (HelpUrlAttribute) attr;
        myBrowser.Navigate( ha.Url );
    }
}
```

## Agenda

- ◆ Review Object-Oriented Concepts
- ◆ Interfaces
- ◆ Classes and Structs
- ◆ Delegates
- ◆ Events
- ◆ Attributes
- ◆ **Preprocessor Directives**
- ◆ XML Comments
- ◆ Unsafe Code

## Preprocessor Directives Overview

- ◆ C# provides preprocessor directives that serve a number of functions
- ◆ Unlike C++, there is not a separate preprocessor
  - The "preprocessor" name is preserved only for consistency with C++
- ◆ C++ preprocessor features removed include:
  - #include: Not really needed with one-stop programming; removal results in faster compilation
  - Macro version of #define: removed for clarity

## Preprocessor Directives Overview

Directive	Description
#define, #undef	Define and undefine conditional symbols
#if, #elif, #else, #endif	Conditionally skip sections of code
#error, #warning	Issue errors and warnings
#region, #end	Delimit outline regions
#line	Specify line number

## Preprocessor Directives Conditional Compilation

```
#define Debug
public class Debug {
    [Conditional("Debug")]
    public static void Assert(bool cond, String s) {
        if (!cond) {
            throw new AssertionException(s);
        }
    }
    void DoSomething() {
        ...
        // If Debug is not defined, the next line is
        // not even called
        Assert((x == y), "X should equal Y");
        ...
    }
}
```

## Preprocessor Directives Assertions

- ◆ By the way, assertions are an incredible way to improve the quality of your code
- ◆ An assertion is essentially a unit test built right into your code
- ◆ You should have assertions to test preconditions, postconditions and invariants
- ◆ Assertions are only enabled in debug builds
- ◆ Your code is QA'd every time it runs
- ◆ Must read: "*Writing Solid Code*", by Steve Maguire, Microsoft Press, ISBN 1-55615-551-4

## Agenda

- ◆ Review Object-Oriented Concepts
- ◆ Interfaces
- ◆ Classes and Structs
- ◆ Delegates
- ◆ Events
- ◆ Attributes
- ◆ Preprocessor Directives
- ◆ **XML Comments**
- ◆ Unsafe Code

## XML Comments Overview

- ◆ Programmers don't like to document code, so we need a way to make it easy for them to produce quality, up-to-date documentation
- ◆ C# lets you embed XML comments that document types, members, parameters, etc.
  - Denoted with triple slash: `///`
- ◆ XML document is generated when code is compiled with `/doc` argument
- ◆ Comes with predefined XML schema, but you can add your own tags too
  - Some are verified, e.g. parameters, exceptions, types

## XML Comments Overview

XML Tag	Description
<code>&lt;summary&gt;</code> , <code>&lt;remarks&gt;</code>	Type or member
<code>&lt;param&gt;</code>	Method parameter
<code>&lt;returns&gt;</code>	Method return value
<code>&lt;exception&gt;</code>	Exceptions thrown from method
<code>&lt;example&gt;</code> , <code>&lt;c&gt;</code> , <code>&lt;code&gt;</code>	Sample code
<code>&lt;see&gt;</code> , <code>&lt;seealso&gt;</code>	Cross references
<code>&lt;value&gt;</code>	Property
<code>&lt;paramref&gt;</code>	Use of a parameter
<code>&lt;list&gt;</code> , <code>&lt;item&gt;</code> , ...	Formatting hints
<code>&lt;permission&gt;</code>	Permission requirements

## XML Comments Overview

```
class XmlElement {
    /// <summary>
    /// Returns the attribute with the given name and
    /// namespace</summary>
    /// <param name="name">
    /// The name of the attribute</param>
    /// <param name="ns">
    /// The namespace of the attribute, or null if
    /// the attribute has no namespace</param>
    /// <return>
    /// The attribute value, or null if the attribute
    /// does not exist</return>
    /// <seealso cref="GetAttr(string)"/>
    public string GetAttr(string name, string ns) {
        ...
    }
}
```

## Agenda

- ◆ Review Object-Oriented Concepts
- ◆ Interfaces
- ◆ Classes and Structs
- ◆ Delegates
- ◆ Events
- ◆ Attributes
- ◆ Preprocessor Directives
- ◆ XML Comments
- ◆ **Unsafe Code**

## Unsafe Code Overview

- ◆ Developers sometime need total control
  - Performance extremes
  - Dealing with existing binary structures
  - Existing code
  - Advanced COM support, DLL import
- ◆ C# allows you to mark code as unsafe, allowing
  - Pointer types, pointer arithmetic
  - `->`, `*` operators
  - Unsafe casts
  - No garbage collection

## Unsafe Code Overview

- ◆ Lets you embed native C/C++ code
- ◆ Basically "inline C"
- ◆ Must ensure the GC doesn't move your data
  - Use `fixed` statement to pin data
  - Use `stackalloc` operator so memory is allocated on stack, and need not be pinned

```
unsafe void Foo() {
    char* buf = stackalloc char[256];
    for (char* p = buf; p < buf + 256; p++) *p = 0;
    ...
}
```

## Unsafe Code Overview

```
class FileStream: Stream {
    int handle;

    public unsafe int Read(byte[] buffer, int index,
        int count) {
        int n = 0;
        fixed (byte* p = buffer) {
            ReadFile(handle, p + index, count, &n, null);
        }
        return n;
    }

    [DllImport("kernel32", SetLastError=true)]
    static extern unsafe bool ReadFile(int hFile,
        void* lpBuffer, int nBytesToRead,
        int* nBytesRead, Overlapped* lpOverlapped);
}
```

## Unsafe Code C# and Pointers

- ◆ Power comes at a price!
  - Unsafe means unverifiable code
  - Stricter security requirements
    - Before the code can run
    - Downloading code

## More Resources

- <http://msdn.microsoft.com>
- [http://windows.oreilly.com/news/hejlsberg\\_0800.html](http://windows.oreilly.com/news/hejlsberg_0800.html)
- <http://www.csharp4help.com/>
- <http://www.csharp-station.com/>
- <http://www.csharpindex.com/>
- <http://msdn.microsoft.com/msdnmag/issues/0900/csharp/csharp.asp>
- <http://www.hitmill.com/programming/dotNET/csharp.html>
- <http://www.c-sharpcorner.com/>
- [http://msdn.microsoft.com/library/default.asp?URL=/library/dotnet/csspec/vclrfcsharp-spec\\_Start.htm](http://msdn.microsoft.com/library/default.asp?URL=/library/dotnet/csspec/vclrfcsharp-spec_Start.htm)