Advanced C#

This material is based on the original slides of Dr. Mark Sapossnek, Computer Science Department, Boston University, Mosh Teitelbaum, evoch, LLC, and Joe Hummel, Lake Forest College

Outline

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

Key Object-Oriented Concepts

- Objects, instances and classes
- Encapsulation
  - Data and function are packaged together
  - Information hiding
  - An object is an abstraction
    - User should NOT know implementation details
- Interfaces
  - A well-defined contract
  - A set of function members

Key Object-Oriented Concepts

- Inheritance
  - Types are arranged in a hierarchy
    - Base/derived, superclass/subclass
- Polymorphism
  - The ability to use an object without knowing its precise type
- Dependencies
  - For reuse and to facilitate development, systems should be loosely coupled
  - Dependencies should be minimized

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();
}
```

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 `System.ValueType`)
Can have a destructor	No destructor
Can have user-defined parameterless constructor	No user-defined parameterless constructor

Classes and Structs
Class

```
public class Car : Vehicle {
  public enum Make { GM, Honda, BMW }
  Make make;
  string vid;
  Point location;
  public 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

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

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
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
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

```
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
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 modifer
  - 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

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);
```

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
- Foundation for elegant event handling

Delegates
Overview

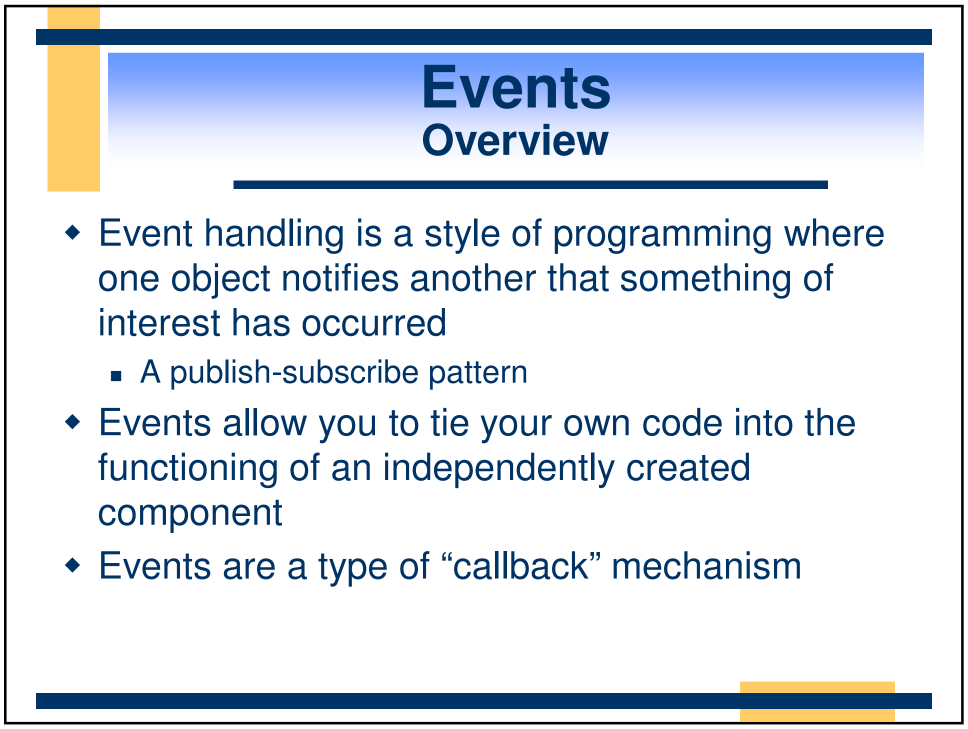
```
delegate double Del(double x);      // Declare

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

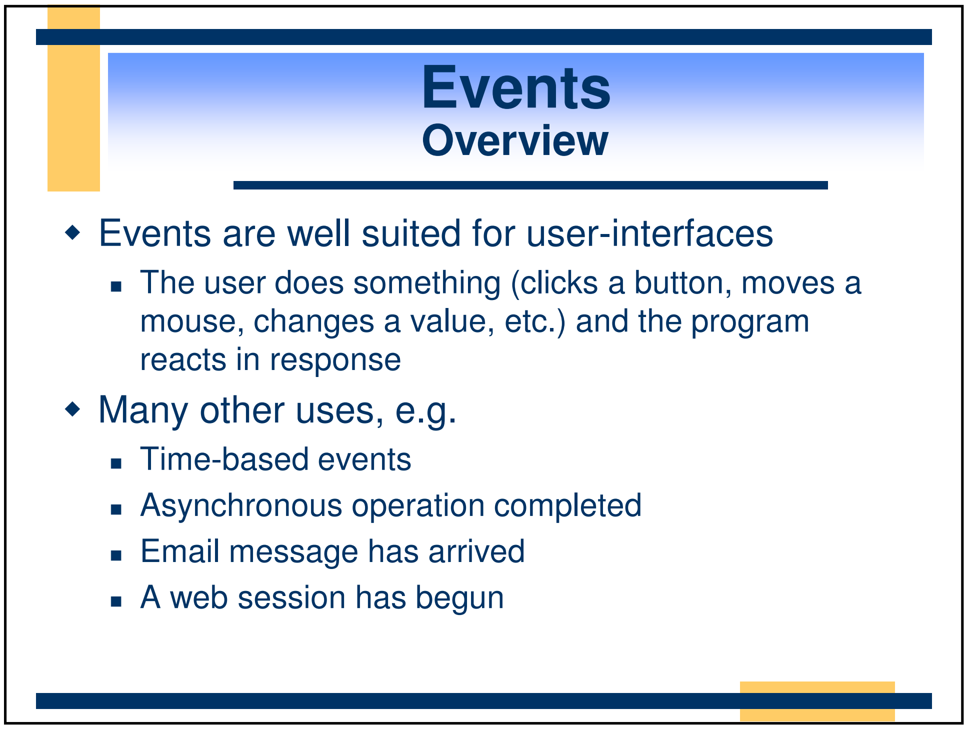
Delegates
Multicast Delegates

```
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
}
```



Events
Overview

- Event handling is a style of programming where one object notifies another that something of interest has occurred
  - A publish-subscribe pattern
- 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);
  }
}
```

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 to you 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 );
  }
}
```

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
`<summary>, <remarks>`	Type or member
`<param>`	Method parameter
`<returns>`	Method return value
`<exception>`	**Exceptions thrown from method**
`<example>, <c>, <code>`	**Sample code**
`<see>, <seealso>`	**Cross references**
`<value>`	**Property**
`<paramref>`	**Use of a parameter**
`<list>, <item>, ...`	**Formatting hints**
`<permission>`	**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) {
      ...
   }
}
```

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);
}
```

Calling Unmanaged Code from Managed Code

- C# and Visual Studio make it easy to write applications in managed code

Unfortunately

- Many cool features aren't available via the Framework
  - Managed code must call the native OS API's

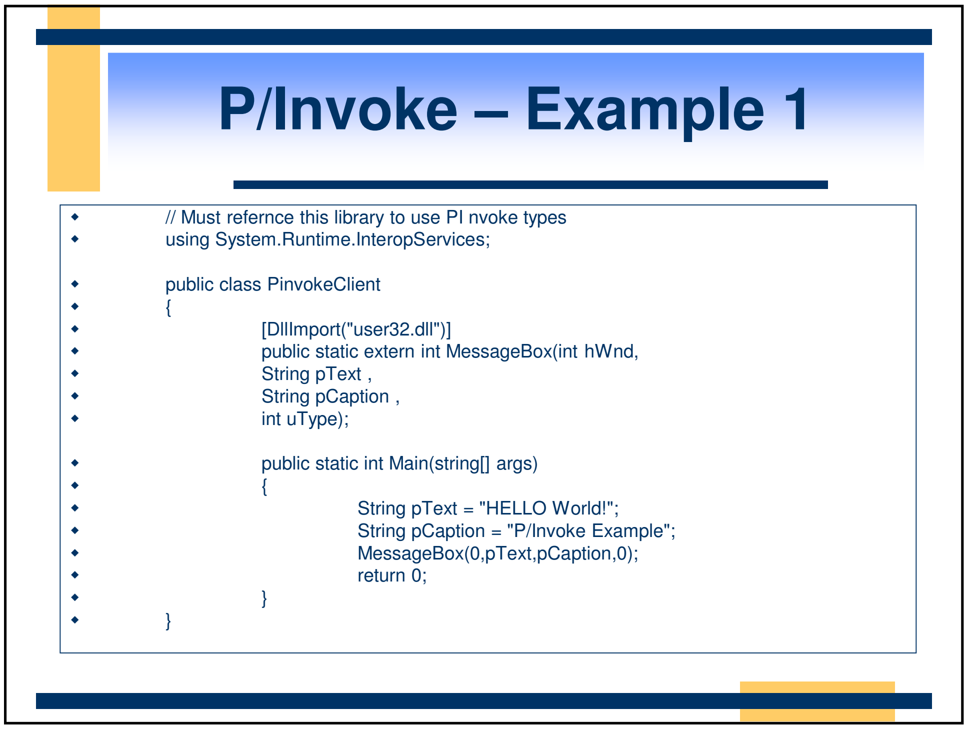
Platform Invoke (P/Invoke)

- The mechanism which allows managed code to invoke unmanaged functions residing in native DLLs
  - `System.Runtime.InteropServices` namespace
- Available in full .NET Framework and .NET Compact Framework

Declaration

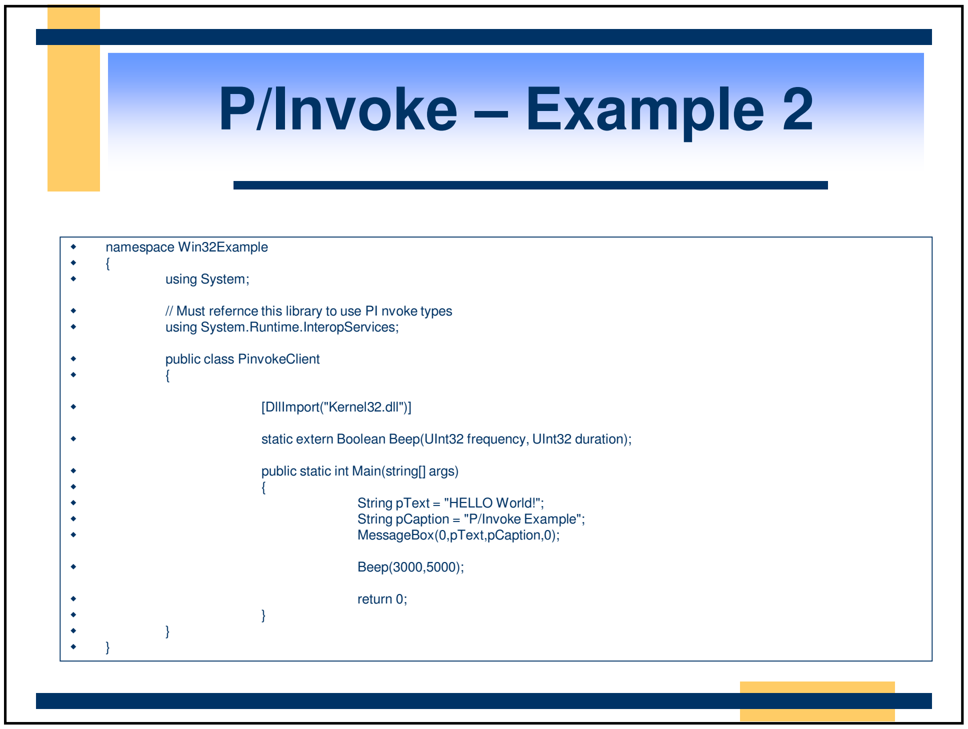
Unmanaged function(s) to be called must be declared

- Specify unmanaged function signature as a `static extern` function
  - Must declare parameters and return values as *managed* types
- Tag the function signature with the `DllImportAttribute` tag
  - State name of DLL containing the function



P/Invoke – Example 1

```
// Must refernce this library to use PI nvoke types
using System.Runtime.InteropServices;

public class PinvokeClient
{
        [DllImport("user32.dll")]
        public static extern int MessageBox(int hWnd,
        String pText ,
        String pCaption ,
        int uType);

        public static int Main(string[] args)
        {
                String pText = "HELLO World!";
                String pCaption = "P/Invoke Example";
                MessageBox(0,pText,pCaption,0);
                return 0;
        }
}
```



P/Invoke – Example 2

```
namespace Win32Example
{
        using System;

        // Must refernce this library to use PI nvoke types
        using System.Runtime.InteropServices;

        public class PinvokeClient
        {

                [DllImport("Kernel32.dll")]

                static extern Boolean Beep(UInt32 frequency, UInt32 duration);

                public static int Main(string[] args)
                {
                        String pText = "HELLO World!";
                        String pCaption = "P/Invoke Example";
                        MessageBox(0,pText,pCaption,0);

                        Beep(3000,5000);

                        return 0;
                }
        }
}
```

Error Handling

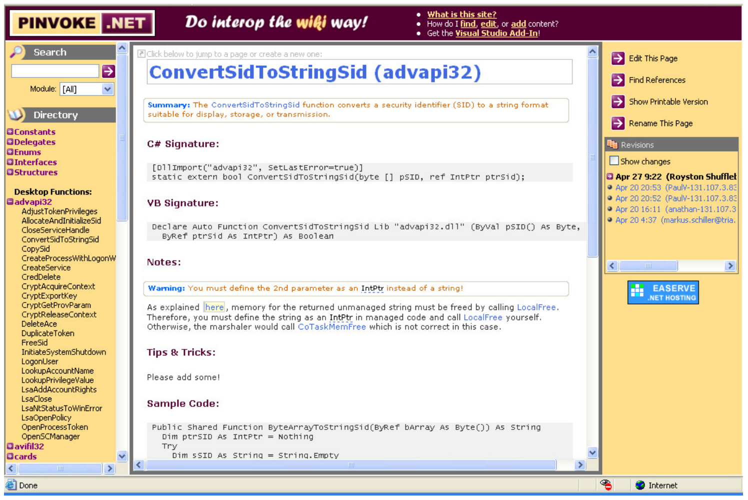
- Two things can go wrong when calling native method
  - **CLR cannot find or invoke native function**
    - `NotSupportedException` thrown if
      - arguments contain invalid data
      - function declared with incorrect parameters
    - `MissingMethodException` thrown if CLR cannot find the native function in the stated DLL
  - **Native function produces an error**
    - Boolean return value indicates failure
    - Call Marshal.GetLastWin32Error() to retrieve the error code

P/Invoke Data Marshalling

- Marshalling is the process of moving data between managed and unmanaged code
- Marshalling is automatic for
  - Value types
  - Reference types composed of simple value types
  - One dimensional arrays of simple types

C# Type	Native C++ type, pass by value	Native C++ type, pass by ref
`byte`	`BYTE, char`	`BYTE*, char*`
`short` `ushort`	`SHORT` `WORD`	`SHORT*` `WORD*`
`int` `uint`	`int` `DWORD`	`int*` `DWORD`
`long`	unsupported	`INT64*`
`float`	unsupported	`float*`
`double`	unsupported	`double*`
`IntPtr`	`PVOID`	`PVOID*`
`bool`	`BYTE`	`BYTE*`
`string`	`LPCWSTR`	unsupported

www.pinvoke.net



ConvertSidToStringSid (advapi32)

Summary: The ConvertSidToStringSid function converts a security identifier (SID) to a string format suitable for display, storage, or transmission.

C# Signature:

```
[DllImport("advapi32", SetLastError=true)]
static extern bool ConvertSidToStringSid(byte [] pSID, ref IntPtr ptrSid);
```

VB Signature:

```
Declare Auto Function ConvertSidToStringSid Lib "advapi32.dll" (ByVal pSID() As Byte,
  ByRef ptrSid As IntPtr) As Boolean
```

Notes:

Warning: You must define the 2nd parameter as an IntPtr instead of a string!

As explained here, memory for the returned unmanaged string must be freed by calling LocalFree. Therefore, you must define the string as an IntPtr in managed code and call LocalFree yourself. Otherwise, the marshaler would call CoTaskMemFree which is not correct in this case.

Tips & Tricks:

Please add some!

Sample Code:

```
Public Shared Function ByteArrayToStringSid(ByRef bArray As Byte()) As String
  Dim ptrSID As IntPtr = Nothing
  Try
    Dim sSID As String = String.Empty
```

A place to find & share PInvoke signatures & types

Two Hugely Different Worlds

<u>COM</u>

- Type libraries (incomplete!)
- Reference counted
- GUIDS
- Immutable types
- DLL issues/Hell
- Interface based
- HRESULTS

<u>Managed Code</u>

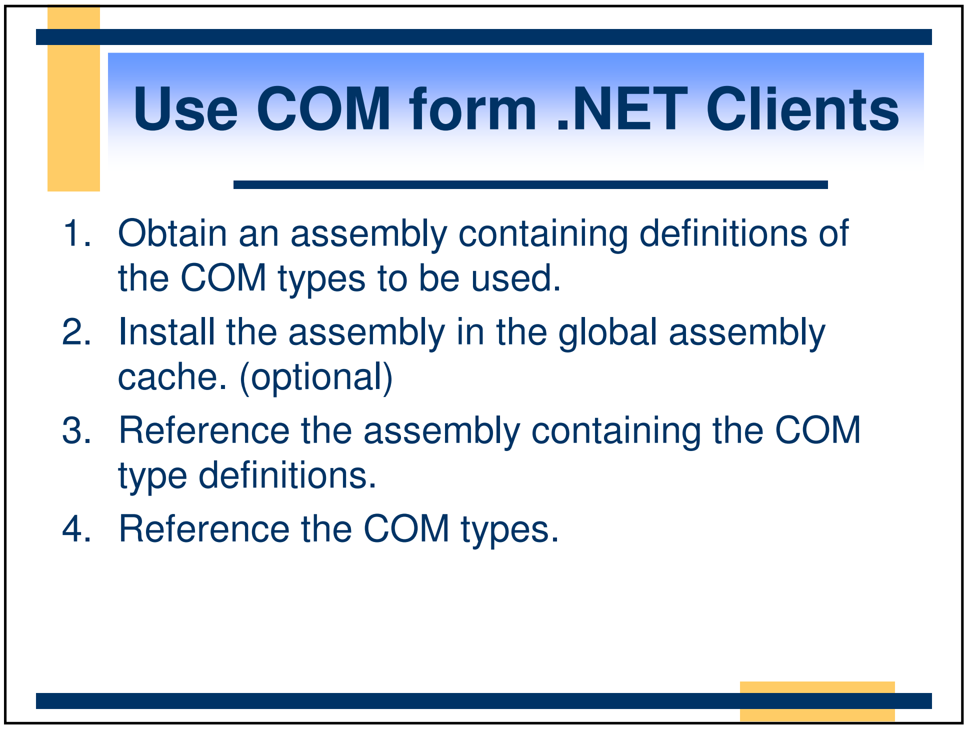
- Meta data
- Garbage collected
- Strong Names
- Flexible runtime bind
- Assemblies
- Object based
- Exceptions

Has Anyone Seen the Bridge?

- We need a bridge to go between 2 worlds
- What do we need to create the bridge?
  - .NET has no clue what COM type info means!
  - .NET requires complete type information & metadata
  - We need an import that transforms **COM type info into .NET metadata**
- This *technically* gives us *Interop Assembly (IA)*
- At *runtime*, we need the bridge (RCW) that uses IA & gets us there – Runtime Callable Wrapper
- Note: Many people use IA and RCW terms interchangeably

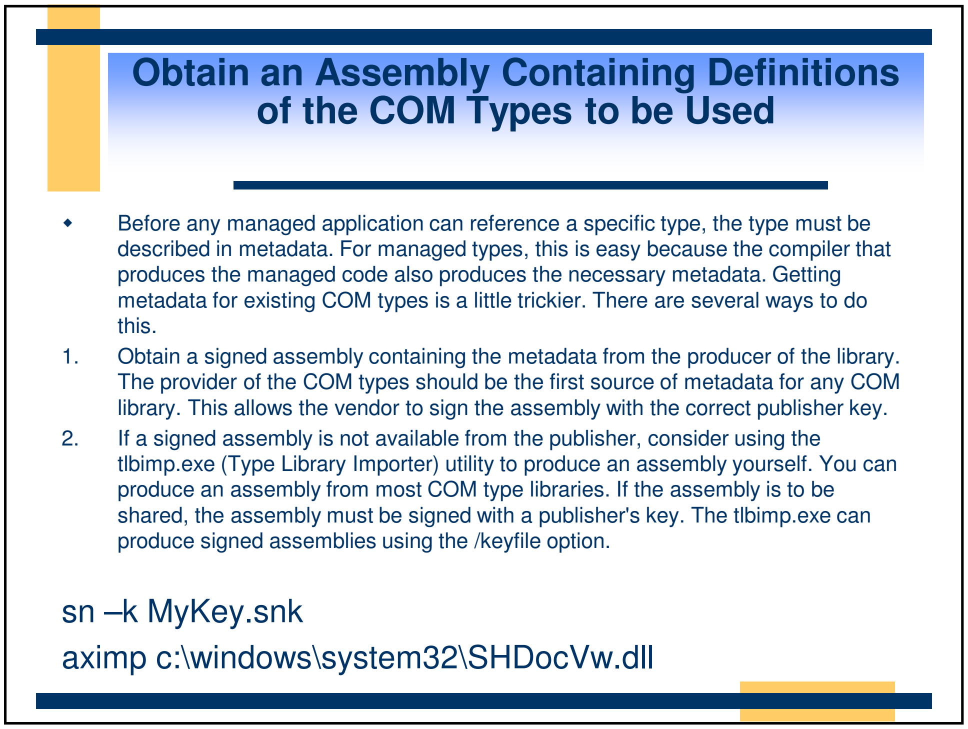
What is an RCW?

- .NET clients never talk to COM objects directly
  - Want to talk .NET to .NET (hide details)
- .NET proxy *dynamically* created by CLR based on metadata in *Interop Assembly*
- RCW Roles
  - Provide .NET programming model (new, etc.)
  - Marshal calls back and forth (proxy)
  - Proxy COM interfaces
  - Preserve object identity
  - Maintain COM object lifetime



Use COM form .NET Clients

1. Obtain an assembly containing definitions of the COM types to be used.
2. Install the assembly in the global assembly cache. (optional)
3. Reference the assembly containing the COM type definitions.
4. Reference the COM types.



Obtain an Assembly Containing Definitions of the COM Types to be Used

- Before any managed application can reference a specific type, the type must be described in metadata. For managed types, this is easy because the compiler that produces the managed code also produces the necessary metadata. Getting metadata for existing COM types is a little trickier. There are several ways to do this.

1. Obtain a signed assembly containing the metadata from the producer of the library. The provider of the COM types should be the first source of metadata for any COM library. This allows the vendor to sign the assembly with the correct publisher key.
2. If a signed assembly is not available from the publisher, consider using the tlbimp.exe (Type Library Importer) utility to produce an assembly yourself. You can produce an assembly from most COM type libraries. If the assembly is to be shared, the assembly must be signed with a publisher's key. The tlbimp.exe can produce signed assemblies using the /keyfile option.

```
sn –k MyKey.snk
aximp c:\windows\system32\SHDocVw.dll
```

Install the Assembly in the Global Assembly Cache

- If you want your assembly containing definitions of the COM types to be shared among several applications, it must be installed in the global assembly cache (GAC). Use gacutil.exe to install an assembly in the GAC.

```
gacutil /i ShDocVw.dll
```

Reference the Assembly Containing the Type Definitions

- With C#, you can reference the assembly using the compiler /r switch or you can add reference to the project directly from Visual Studio development tool.

```
csc TestClient.cs /r:SHDocVw.dll
```

Reference the COM Types

-
  ```
  namespace TestClient {
  public class Test {
      public static void Main(){
      SHDocVw.InternetExplorer explorer;
      SHDocVw.IWebBrowserApp webBrowser;
      explorer = new  SHDocVw.InternetExplorer();
      webBrowser = (SHDocVw.IWebBrowserApp) explorer;
      webBrowser.Visible = true;
      webBrowser.GoHome();
      }
    }
  }
  ```

Examples: Examine Attached Code

- Classes / Constructors / Destructors
- Properties
- Events / Delegates
- Publish-subscribe pattern
- XML Documentation
- P/Invoke
- COM Interop