# Interfaces and Abstract Class

# 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;
  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

## 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
  - 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
## 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

## 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 1
}  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

- 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);
  }
  ...
}
```