Interfaces

An interface is similar to a class, but it provides a specification rather than an implementation for its members. An interface is special in the following ways:

- Interface members are *all implicitly abstract*. In contrast, a class can provide both abstract members and concrete members with implementations.
- A class (or struct) can implement *multiple* interfaces. In contrast, a class can inherit from only a *single*class, and a struct cannot inherit at all (aside from deriving from `System.ValueType`).

Extending an Interface

Interfaces may derive from other interfaces. For instance:

```
public interface IUndoable             { void
Undo(); }

public interface IRedoable : IUndoable { void
Redo(); }
```

`IRedoable` "inherits" all the members of `IUndoable`. In other words, types that implement `IRedoable`must also implement the members of `IUndoable`.

Explicit Interface Implementation

Implementing multiple interfaces can sometimes result in a collision between member signatures. You can resolve such

collisions by *explicitly implementing* an interface member. Consider the following example:

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2
{
  public void Foo()
  {
    Console.WriteLine ("Widget's implementation
of I1.Foo");
  }

  int I2.Foo()
  {
    Console.WriteLine ("Widget's implementation
of I2.Foo");
    return 42;
  }
}
```

Because both `I1` and `I2` have conflicting `Foo` signatures, `Widget` explicitly implements `I2`'s `Foo` method. This lets the two methods coexist in one class. The only way to call an explicitly implemented member is to cast to its interface:

```
Widget w = new Widget();

w.Foo();                          // Widget's
implementation of I1.Foo

((I1)w).Foo();                    // Widget's
implementation of I1.Foo
```

```
((I2)w).Foo();                  // Widget's
implementation of I2.Foo
```

Enums

An enum is a special value type that lets you specify a group of named numeric constants. For example:

```
public enum BorderSide { Left, Right, Top,
Bottom }
```

We can use this enum type as follows:

```
BorderSide topSide = BorderSide.Top;

bool isTop = (topSide == BorderSide.Top);   //
true
```

Each enum member has an underlying integral value. By default:

- Underlying values are of type `int`.
- The constants `0`, `1`, `2`... are automatically assigned, in the declaration order of the enum members.

You may specify an alternative integral type, as follows:

```
public enum BorderSide : byte { Left, Right,
Top, Bottom }
```

You may also specify an explicit underlying value for each enum member:

```
public enum BorderSide : byte { Left=1, Right=2,
Top=10, Bottom=11 }
```

Nested Types

A *nested type* is declared within the scope of another type. For example:

```
public class TopLevel
{
  public class Nested { }        // Nested class
  public enum Color { Red, Blue, Tan }//Nested
//enum
}
```

A nested type has the following features:

- It can access the enclosing type's private members and everything else the enclosing type can access.
- It can be declared with the full range of access modifiers, rather than just `public` and `internal`.
- The default accessibility for a nested type is `private` rather than `internal`.
- Accessing a nested type from outside the enclosing type requires qualification with the enclosing type's name (like when accessing static members).

For example, to access `Color.Red` from outside our `TopLevel` class, we'd have to do this:

```
TopLevel.Color color = TopLevel.Color.Red;
```

All types (classes, structs, interfaces, delegates, and enums) can be nested inside either a class or a struct.
Here is an example of accessing a private member of a type from a nested type:

```
public class TopLevel
{
  static int x;
  class Nested
  {
    static void Foo() { Console.WriteLine
(TopLevel.x); }
  }
}
```

Here is an example of applying the `protected` access modifier to a nested type:

```
public class TopLevel
{
  protected class Nested { }
}

public class SubTopLevel : TopLevel
{
  static void Foo() { new TopLevel.Nested(); }
}
```

Here is an example of referring to a nested type from outside the enclosing type:

```
public class TopLevel
{
  public class Nested { }
}

class Test
{
  TopLevel.Nested n;
}
```