CHAPTER 17

Interfaces

An interface is used to specify members that deriving classes must implement. They are defined with the `interface` keyword followed by a name and a code block. Their naming convention is to start with a capital `I` and then to have each word initially capitalized.

```
interface IMyInterface {}
```

Interface Signatures

The interface code block can only contain signatures, and only those of methods, properties, indexers, and events. The interface members cannot have any implementations. Instead, their bodies are replaced by semicolons. They also cannot have any restrictive access modifiers since interface members are always public.

```
interface IMyInterface
{
  // Interface method
  int GetArea();

  // Interface property
  int Area { get; set; }
```


M. Olsson, *C# 8 Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-5577-3_17

```
  // Interface indexer
  int this[int index] { get; set; }

  // Interface event
  event System.EventHandler MyEvent;
}
```

Interface Example

In the following example, an interface called `IComparable` is defined with a single method named `Compare`.

```
interface IComparable
{
  int Compare(object o);
}
```

The class `Circle` defined next implements this interface by using the same notation as is used for inheritance. The `Circle` class must then define the `Compare` method, which for this class will return the difference between the circle radiuses. The implemented member must be public, in addition to having the same signature as the one defined in the interface.

```
class Circle : IComparable
{
  int r;
  public int Compare(object o)
  {
    return r - (o as Circle).r;
  }
}
```

Although a class can only inherit from one base class, it may implement any number of interfaces. It does so by specifying the interfaces in a comma-separated list after the base class.

Functionality Interface

`IComparable` demonstrates the first use of interfaces, which is to define a specific functionality that classes can share. It allows programmers to use the interface members without having to know the actual type of a class. To illustrate, the following method takes two `IComparable` objects and returns the largest one. This method will work for any two objects of the same class that implement the `IComparable` interface, because the method only uses the functionality exposed through that interface.

```
static object Largest(IComparable a, IComparable b)
{
  return (a.Compare(b) > 0) ? a : b;
}
```

Class Interface

A second way to use an interface is to provide an actual interface for a class, through which the class can be used. Such an interface defines the functionality that programmers using the class will need.

```
interface IMyClass
{
  void Exposed();
}
```

```
class MyClass : IMyClass
{
  public void Exposed() {}
  public void Hidden()  {}
}
```

The programmers can then view instances of the class through this interface by enclosing the objects in variables of the interface type.

```
IMyInterface m = new MyClass();
```

This abstraction provides two benefits. First, it makes it easier for other programmers to use the class since they now only have access to the members that are relevant to them. Second, it makes the class more flexible since its implementation can change without being noticeable by other programmers using the class, as long as the interface is followed.

Default Implementations

C# 8.0 added the ability to create default implementations for interface members. Consider the following example of a simple logging interface.

```
interface ILogger
{
  void Info(string message);
}

class ConsoleLogger : ILogger
{
  public void Info(string message)
  {
    Console.WriteLine(message);
  }
}
```

By providing a default implementation, this existing interface can be extended with a new member without breaking any classes using the interface.

```
interface ILogger
{
  void Info(string message);
  void Error(string message)
  {
    Console.WriteLine(message);
  }
}
```