**C#\_5.1 Interface**

A derived class must provide its own implementation of each abstract method defined by its base class. Thus, an abstract method specifies the interface to the method, but not the implementation. In C#, you can fully separate a class’ interface from its implementation by using the keyword interface. [Similar as Java pat 5.7].

|  |  |
| --- | --- |
| * Here is a simplified form of an interface declaration:   **interface name { ret-type method-name1(param-list);**  **ret-type method-name2(param-list);**  **// ...**  **ret-type method-nameN(param-list); }** | * The name of the interface is specified by name. * Methods are declared using only their return type and signature. They are, essentially, abstract methods, no method can have an implementation. Thus, each class that includes an interface must implement all of the methods declared by that interface. * In an interface, methods are implicitly public, and no explicit access specifier is allowed. |

* Implementing Interfaces: Once an interface has been defined, one or more classes can implement that interface. To implement an interface, the name of the interface is specified after the class name in just the same way that a base class is specified. The general form of a class that implements an interface is:

***class class-name : interface-name { /\* class-body \*/ }***

* The name of the interface being implemented is specified in interface-name.
* When a class implements an interface, the class must implement the entire interface. It cannot pick and choose which parts to implement.
* Classes can implement more than one interface. To implement more than one interface, the interfaces are separated with a comma.
* A class can inherit a base class and implement one or more interfaces. In this case, the name of the base class must come first in the comma-separated list.
* The methods that implement an interface must be declared public. The reason is that methods are implicitly public within an interface.
* Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.
* C#\_Example 1: Here is an example that implements the ISeries interface shown earlier. It creates a class called ByTwos, which generates a series of numbers, each two greater than the previous one.

|  |  |  |  |
| --- | --- | --- | --- |
| **public interface** ISeries{  **int** GetNext();  **void** Reset();  **void** SetStart(**int** x); } | **class** ByTwos : ISeries{ **int** start, val; **public** ByTwos(){ start = 0; val = 0; } */\* constructor \*/*  **public** **int** GetNext(){ val += 2; **return** val; }  **public** **void** Reset(){ start = 0; val = 0; }  **public** **void** SetStart(**int** x) { start = x; val = x; } } | | |
| **using System**;  **class** ISeriesDemo { **static void Main()** { ByTwos ob = **new** ByTwos();  **for**(**int** i=0; i < 5; i++) **Console.WriteLine**("Next value is " + ob.GetNext());  **Console.WriteLine**("\nResetting"); ob.Reset(); **for**(**int** i=0; i < 5; i++)  **Console.WriteLine**("Next value is " + ob.GetNext());  **Console.WriteLine**("\nStarting at 100"); ob.SetStart(100); **for**(**int** i=0; i < 5; i++)  **Console.WriteLine**("Next value is " + ob.GetNext()); }} | | | * To compile ISeriesDemo, you must include the classes ISeries, ByTwos, and ISeriesDemo in the compilation. If you called these files ISeries.cs, ByTwos.cs, and ISeriesDemo.cs, then the command line will compile the program:   **>csc ISeries.cs ByTwos.cs ISeriesDemo.cs**   * If you are using the Visual Studio IDE, then simply add all three files to your C# project. * It is valid to put all three of these classes in the same file. |
| * Add a method not defined by interface: It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of ByTwos adds the method ***GetPrevious()***, which returns the previous value: * Notice that the addition of ***GetPrevious()*** required a change to implementations of the methods defined by ***ISeries***. However, since the interface to those methods stays the same, the change is seamless and does not break preexisting code. | | **class** ByTwos : ISeries { **int** start, val, prev;  **public** ByTwos() { start = 0; val = 0; prev = -2; }  **public** **int** GetNext() { prev = val; val += 2; **return** val; }  **public** **void** Reset() { start = 0; val = 0; prev = -2; }  **public** **void** SetStart(**int** x) {start = x; val = x; prev = x - 2; }  *// A method not specified by ISeries.*  **int** GetPrevious() { **return** prev; } } | |

**C#\_5.2 Using Interface References (Similar to Java part 5.9)**

**C#\_5.3 Interface Properties and Interface Indexers**

* Interface Properties: Like methods, properties are specified in an interface without any body. Here is the general form of a property specification:

**type name { get; set; }**

* Of course, only get or set will be present for read-only or write-only properties, respectively.
* Although the declaration of a property in an interface looks similar to how an auto-implemented property is declared in a class, the two are not the same. The interface declaration does not cause the property to be auto-implemented. It only specifies the name and type of the property. Implementation is left to each implementing class.
* Access modifiers not allowed on the accessors when a property is declared in an interface. Eg: the set accessor, cannot be specified as private in an interface.
* C#\_Example 2: Use a property in an interface - ***ISeries*** interface and the ***ByTwos*** class that uses a property to obtain and set the next element in the series:

|  |  |
| --- | --- |
| **using** **System**;  **public interface** ISeries {  *// An interface property.*  **int** Next { **get**; */\* return the next number in series \*/*  **set**; */\* set next number \*/* } }  *// Implement ISeries.*  **class** ByTwos : ISeries { **int** val;  **public** ByTwos() { val = 0; }  *// Get or set value.*  **public int** Next { **get** {val += 2; **return** val; } **set** { val = value; } } } | *// Demonstrate an interface property.*  **class** ISeriesDemo3 { **static void Main()** {  **ByTwos** ob = **new** ByTwos();  *// Access series through a property.*  **for**(**int** i=0; i < 5; i++) **Console.WriteLine**("Next value is " + ob.Next);  **Console.WriteLine**("\nStarting at 21");  ob.Next = 21;  **for**(**int** i=0; i < 5; i++) **Console.WriteLine**("Next value is " + ob.Next);  }} |

* Interface Indexers: An indexer declared in an interface has this general form: **element-type this[int index] { get; set; }**
* As before, only get or set will be present for read-only or write-only indexers, respectively. No access modifiers are allowed on the accessors when an indexer is declared in an interface. *Example:* Here is another version of ***ISeries*** that adds a *read-only* indexer that returns the ***ith*** element in the series:

|  |  |  |
| --- | --- | --- |
| using System;  public interface ISeries {  */\* An interface property.\*/*  int Next { get; set; }  */\* An interface indexer Declare a read-only indexer in ISeries \*/*  int this[int index] { get; }  */\*****get*** *returns specified number in series \*/*  } | *// Implement ISeries.*  class ByTwos : ISeries { int val;  public ByTwos() { val = 0; }  *// Get or set a value using a property.*  public int Next { get { val += 2; return val; }  set { val = value; } }  *// Get a value using an indexer: Implement the indexer.*  public int this[int index] {  get { val = 0;  for(int i=0; i<index; i++) val += 2;  return val; } } } | *// Demonstrate an interface indexer.*  class ISeriesDemo4 { static void Main() {  ByTwos ob = new ByTwos();  *// Access series through a property.*  for(int i=0; i < 5; i++) Console.WriteLine("Next value is " + ob.Next);  Console.WriteLine("\nStarting at 21"); ob.Next = 21;  for(int i=0; i < 5; i++) Console.WriteLine("Next value is " + ob.Next);  Console.WriteLine("\nResetting to 0"); ob.Next = 0;  *// Access series through an indexer.*  for(int i=0; i < 5; i++) Console.WriteLine("Next value is " + ob[i]); }} |

**C#\_5.4 Interfaces Can Be Inherited:** *( Must provide implementations for all the members defined within the interface inheritance chain)*

The syntax is the same as for inheriting classes. Similar Java part 5.11. **public interface IA{ } public interface IB : IA{ }**

* Name hiding during interface inheritance: When one interface inherits another, it is possible to declare a member in the derived interface that hides a member defined by the base interface. When a member in a derived *interface* has the same signature as one in the base *interface*, the base interface name is hidden. As is the case with class inheritance, this hiding will cause a warning message, unless you specify the derived interface member with new.

**C#\_5.5 Explicit Implementations**

When implementing a member of an interface, it is possible to fully qualify its name with its interface name. Doing this creates an explicit interface member implementation, or explicit implementation. For example, given: **interface** IMyIF { **int** MyMeth(**int** x); } it is legal to implement IMyIF as:

**class** MyClass : IMyIF{ **int** IMyIF.MyMeth(**int** x){ **return** x/3; } }

* As you can see, when the ***MyMeth()*** member of ***IMyIF*** is implemented, its complete name, including its interface name, is specified.
* There are two reasons for explicit implementation.
* First, it is possible for a class to implement two interfaces, which both declare methods by the same name and type signature. Qualifying the names with their interfaces removes the ambiguity from this situation.
* Second, when you ***implement a method*** using its fully qualified name, you are providing an implementation that cannot be accessed through an object of the class. Thus, an explicit implementation gives you a way to implement an interface method so that it is not a public member of the implementing class.

|  |  |
| --- | --- |
| Explicit implementation removes the ambiguity. | Explicitly implement an interface member. |
| interface IMyIF\_A { int Meth(int x); }  interface IMyIF\_B { int Meth(int x); }  *// MyClass implements both interfaces.*  class MyClass : IMyIF\_A, IMyIF\_B { IMyIF\_A a\_ob;  IMyIF\_B b\_ob;  *// Explicitly implement the two Meth()s.*  int IMyIF\_A.Meth(int x) { return x + x; }  int IMyIF\_B.Meth(int x) { return x \* x; }  *// Call Meth() through an interface reference.*  public int MethA(int x){  a\_ob = this; return a\_ob.Meth(x); */\* calls IMyIF\_A \*/* }  public int MethB(int x){  b\_ob = this; return b\_ob.Meth(x); */\* calls IMyIF\_B \*/* }  }  class FQIFNames { static void Main() { MyClass ob = new MyClass();  Console.WriteLine(ob.MethA(3));  Console.WriteLine(ob.MethB(3)); }} | using System;  interface IEven { bool IsOdd(int x);  bool IsEven(int x); }  class MyClass : IEven {  bool IEven.IsOdd(int x) { if((x%2) != 0) return true; else return false; }  *// Normal implementation.*  public bool IsEven(int x) { IEven o = this; */\* Interface reference to invoking object \*/*  return !o.IsOdd(x); }  }  class Demo { static void Main() { MyClass ob = new MyClass();  bool result;  result = ob.IsEven(4);  if(result) Console.WriteLine("4 is even.");  else Console.WriteLine("3 is odd.");  *//* result = ob.IsOdd(); *// Error, not exposed.*  }} |
| * Notice that ***Meth()*** has the same signature in both ***IMyIF\_A*** and ***IMyIF\_B***. Explicit implementation removes the ambiguity. | * Since ***IsOdd()*** is implemented explicitly, it is not exposed as a ***public*** member of ***MyClass***. Instead, ***IsOdd()*** can be accessed only through an ***interface reference***. This is why it is invoked through o in the implementation for ***IsEven()***. |

**C#\_5.6 Structures**

Classes are reference types. Class objects are accessed through a reference. But value types are accessed directly. Accessing class objects through a reference adds overhead onto every access. It also consumes space.

* Structure: C# offers the structure to access an object directly, in the way that value types are. A structure is similar to a class, but is a value type, rather than a reference type. Structures are declared using the keyword struct and are syntactically similar to classes. Here is the general form of a struct:

***struct name : interfaces {*** */\* member declarations \*/* ***}***

* The ***name of the structure*** is specified by name. Example is given after some rules.

|  |  |  |  |
| --- | --- | --- | --- |
| * Rules: | * Structures cannot inherit other structures or classes, or be used as a base for other structures or classes. * However, a structure can implement one or more interfaces. These are specified after the structure name using a comma-separated list. * Structures can also define constructors, but not destructors. * You cannot define a ***default (parameterless) constructor*** for a structure. The reason for this is that a default constructor is automatically defined for all structures and this default constructor can’t be changed. * Like classes, structure members include methods, fields, indexers, properties, operator methods, and events. * A structure object can be created using new in the same way as a class object, but it is not required. * When new is used, the specified constructor is called. When new is not used, the object is still created, but it is not initialized. | | |
| **using System; /\*Example of a Structure \*/**  **struct** Account {  **public** **string** name;  **public** **double** balance;  **public** Account(**string** n, **double** b) { name = n; balance = b; } } | | | **class** StructDemo { **static void Main()** {  **Account** acc1 = **new** Account("Tom", 1232.22); ***// explicit constructor***  **Account** acc2 = **new** Account(); ***// default constructor***  **Account** acc3; ***// no constructor***  /\* . . . \*/ } } |
| * struct/Structures in C# and C++ not the same: | | * In C#, a struct defines a value type, and a class defines a reference type. * In C++, struct defines a class type. Thus, in C++, struct and class are ***nearly equivalent***. (Only difference between struct and class in C++ is: default access of their members, which is private for class and public for struct.) | |

**C#\_5.6 Enumerations (C/C++/Java similar)**

An enumeration is a set of named integer constants. For example, an enumeration of the coins used in the US is: penny, nickel, dime, quarter, half-dollar, dollar.

* The keyword enum declares an enumerated type. The general form for an enumeration is: **enum name { enumeration list };**
* Here, the ***type name*** of the ***enumeration*** is specified by name. The enumeration list is a comma-separated list of identifiers. Eg: an enumeration called ***Coin***:

**enum** Coin { Penny, Nickel, Dime, Quarter, HalfDollar, Dollar};

* Each of the symbols stands for an integer value. However, no implicit conversions are defined between an enum type and the built-in integer types, so an explicit cast must be used. Also, a cast is required when converting between two enumeration types.
* Since enumerations represent integer values, you can use an enumeration to control a switch statement or as the control variable in a for loop.
* Each enumeration symbol is given a value one greater than the symbol that precedes it.
* By default, the value of the first enumeration symbol is 0. Therefore, in the Coin enumeration, Penny is 0, Nickel is 1, Dime is 2, and so on.
* The members of an enumeration are accessed through their type name via the dot operator. For example: ***Console.WriteLine(Coin.Penny + " " + Coin.Nickel);***
* C#\_Example 3: Here is a program that illustrates the Coin enumeration:

|  |  |
| --- | --- |
| using System;  class EnumDemo { enum Coin { Penny, Nickel, Dime, Quarter, HalfDollar, Dollar };  static void Main() { Coin c; *// declare an enum variable*  *// Use c to cycle through the enum by use of a for loop.*  for(c = Coin.Penny; c <= Coin.Dollar; c++) {  Console.WriteLine(c + " has value of " + (int) c); | switch(c) { *// Use an enumeration value to control a switch.*  case Coin.Nickel : Console.WriteLine("A nickel is 5 pennies."); break;  case Coin.Dime : Console.WriteLine("A dime is 2 nickels."); break;  case Coin.Quarter : Console.WriteLine("A quarter is 5 nickels."); break;  case Coin.HalfDollar : Console.WriteLine("A half-dollar is 5 dimes."); break;  case Coin.Dollar : Console.WriteLine("A dollar is 10 dimes."); break;  } Console.WriteLine(); } */\* for* *loop ends \*/* }} |

* Notice how both the ***for*** loop and the ***switch*** statement are controlled by ***c***, which is a variable of type Coin. To obtain *coins* value, a cast to int is required.
* Initialize an Enumeration: You can specify the value of one or more of the enumeration symbols by using an initializer. Do this by following the symbol with an equal sign and an integer value. Symbols that appear after initializers are assigned values greater than the previous initialization value. For example,
* To assigns the value of ***100*** to ***Quarter****:* **enum Coin { Penny, Nickel, Dime, Quarter=100, HalfDollar, Dollar};**
* Now the values of the enumeration members are: *Penny* 0, *Nickel* 1, *Dime* 2, *Quarter* 100, *HalfDollar* 101, *Dollar* 102
* Specifying the Underlying Type of an Enumeration: By default, enumerations are based on type int, but you can create an enumeration of any integral type, except for type char. To specify a type other than int, put the underlying type after the enumeration name, separated by a colon. For example, this statement makes Coin an enumeration based on byte: **enum Coin : byte { Penny, Nickel, Dime, Quarter, HalfDollar, Dollar};**
* Now, ***Coin.Penny***, is a byte quantity.