**What happens if you have unhandled constructor exceptions?**

An unhandled exception from a constructor can occur if the body of a constructor causes an exception to be raised. This is sometimes unavoidable, although it is inconvenient. For example, if you were unable to fully initialize the object, for example because of inconsistent or corrupt state, you would have no other choice but to throw an exception. The alternative would be to pretend everything was OK by returning successfully from the constructor, which could lead to failures later on in the program’s execution. It’s better just to have the program fail as soon as you notice a problem.

The reason that this topic is problematic enough to call out is that, in the face of an exception thrown by a constructor, the caller of your constructor will have no reference to your object. Consider what the IL sequence looks like for the C# code Foo f = new Foo():

newobj instance void Foo::.ctor()

stloc.0

The newobj instruction makes a call to the constructor, leaving the result on the execution stack. stloc.0 stores it in a local slot in the execution stack. But if an exception is raised as a result of the newobj instruction (because of insufficient memory to allocate the object, or because of an unhandled exception thrown from the constructor), the result will never be placed on the stack, and the stloc.0 will never execute because the CLR exception handling behavior will take over. The result is that the variable f will be null.

This can cause some unpredictable behavior. For example, consider a type that allocates resources in its constructor but then throws an exception. Normally, types like this will support the IDisposable interface—explained fully in Chapter 5—so that these resources can be reclaimed deterministically. Such code might look like this:

using (Foo f = new Foo())

{

// Do something interesting...

}

The semantics of the using statement are such that the variable inside the parenthesis will have its Dispose method called at the end of the block, regardless of whether exit is normal or due to an exception from the block’s body. But if an exception occurs during the call to Foo’s constructor, f will never get assigned a value! The result is that Dispose will never get called, and hopefully f’s finalizer will clean up the resources sometime later on.

**Constructors do not have return type and so they cannot return error codes (of course possible with *out* or *ref* parameter, but it is not recommended to use out or ref parameter). How are errors or exceptions handled in constructors? What if the calls that you make in the constructor can actually throw exceptions? How do you let the caller know something bad happened in a constructor? Or, How to let the invoker know that the instance of the object has not been created successfully, so that the invoker can perform appropriate actions?**

There are a few ways to do robust error/exception handling in constructors

1. Do as little in the constructor has you can. Then provide an Init() function in the constructor, which does the normal initialization stuff. The user can then call this function after creating an object. The problem here is, its up to the user to actually call the Init() function. The user could potentially miss this step, making this method error prone. However, there are a lot of places where this methodology is used. You are trying to eliminate error handling in the constructor by using this method
2. Another way to do this is by putting the object in a Zombie state. This is one approach you can take especially if you do not have the option of using exceptions. When you go with this option, you will also to do provide a function that will check the state of the object after construction. The downsides to this option is that, its up to the user to do these checks and the users will need to do this every time one attempts to create an object. It’s usually always better and cleaner to throw an exception instead. Use the Zombie option as a last resort.
3. The downsides to the above methods can be reduced by making the constructor private or protected, expose a CreateInstance() public method, and do all the error handling here rather than leave it to the user. But sometimes, it’s not possible to handle all the error conditions in a generic manner and you will need to throw an exception.
4. If an exception is thrown in the constructor, the destructor will not get called. So you need to handle and clean up as much as you can before you leave the constructor. The best way to do this is using the “resource allocation is initialization” technique in C++. But the basic idea is to assign resource allocation and cleanup to other objects. Basically, you are trying to get allocation out of the way (indirect) so that you don’t have to do it explicitly. When you don’t allocate something directly, you don’t have to release it either because it will be done by the component or class who deals with it. E.g. If you need to allocate some memory or open up a file, You can use smart objects (smart pointer, auto\_ptr, smart file handlers etc..) instead of calling new or fopen directly. When you do this, and if an exception is thrown in your constructor, the smart objects will automatically release the resources it acquired, as the stack unwinds. If you do not use the “resource allocation is initialization” technique (in java and C#), the user will need to wrap the statements in try/catch block and re-throw after cleaning up the mess, something like what the finally block does in Java or C#. Although this works in theory, it’s up to the user to make this work and it also always a source of errors and bugs (esp. memory and handle leaks) and is messy.

You have to be careful when throwing exceptions **in** a **constructor**, that is, if you want clean running code. Constructors should also return valid objects but sometimes can't due to errors. The best way to deal with something of this scenario is to present clear error messages that let the user know what they failed to give or present, and second is to use the finally clause. The finally clause will give about the only way to "clean up" your code after failure, although you must be careful here also. Using the finally clause means that it will execute every time the code is run(you know this already), which means some kind of flag must be created **in** order for proper cleanup.

As you have seen, there is no “one size fits all” rule to do error/exception handling in constructors. I have listed the most commonly used methods and one of these should work most of the time.

**How can one report errors to invoker of constructor using error code?**

There is a more clean solution to the problem. Constructors, as any other methods, can return many values to the calling program via “ref” parameters! It would be natural for constructors to validate their arguments and report errors. So here is solution:  
class Levels  
{  
private double[] \_buffer;  
private int \_size;

public Levels(int xsize, ref int xerror) //here goes my return value  
{  
if (xsize < 2)  
{  
xerror = -1; //"Need at least 2 Levels"  
return;  
}  
\_size = xsize;  
\_buffer = new double[xsize];  
}  
…}  
Now, in the main program:  
public static void Main()  
{  
int error = -2; //should be set to 0 if all goes well  
Levels support = new Levels(1, ref error); //error will tell me if all is OK  
if (error != 0)  
{  
Console.WriteLine("ABEND {0}", error); //ABnormal END  
Console.ReadKey();  
Environment.ExitCode = error;  
return; //min 2 levels  
}

The downsides to this option is that, it’s up to the user to do these checks and the users will need to do this every time one attempts to create an object. It’s usually always better and cleaner to throw an exception instead. Of course this provides one solution without the need to re-throw exception. Throwing exception is always costly.

**Write code that helps in understanding program for handling Constructor exception.**

A Constructor means a method which has the same name as of the class. Constructor initializes a new object belonging to the class automatically. The program given below consists of a parameterized constructor which takes String and int as a parameters. The main reason of exception occurring in the program is   ConstructorException c = new ConstructorException("gh", -1);For removing this exception we have to passs a value which is greater than 0 instead of -1..

**ConstructorException.java**

|  |
| --- |
| **public class**ConstructorException {      **private static**String name;     **private static int**age;     **public**ConstructorException(String Name, **int**Age) {         name = Name;         age = Age;         **if**(age < 0) {             **throw new**IllegalArgumentException("age out of range: " + age );         }         **if**(name == **null**) {             **throw new**IllegalArgumentException(                     "name is null");         }     }      **public static void**main(String[] args) {         ConstructorException c = **new**ConstructorException("gh", -1);     } } |

output of the program

|  |
| --- |
| Exception in thread "main" [java](http://www.roseindia.net/java/java-exception/constructor-exception-java.shtml).lang.IllegalArgumentException: age out of range: -1 at ConstructorException.<init>(ConstructorException.java:10) at ConstructorException.main(ConstructorException.java:19) Java Result: 1 |

Vtable

**What is the concept of V-Table in C#?**

vtable, is a mechanism used in a programming language to support dynamic dispatch (or run-time method binding).

Suppose a program contains several classes in an inheritance hierarchy: a superclass, Cat, and two subclasses, HouseCat and Lion. Class Cat defines a virtual function named speak, so its subclasses may provide an appropriate implementation (i.e., either meow or roar).

When the program calls the speak method on a Cat pointer (which can point to a Cat class, or any subclass of Cat), the run-time environment must be able to determine which implementation to call, depending on the actual type of object that is pointed to.

**When is a vtable created: compile time or run time? In particular, if a declare a class having one or more virtual functions, and in first case I create an object of it and in second case I don't create an object. What difference it makes in terms of number of vtable’s created?**  
A vtable is created in an instance of a class when the base class has one or more virtual functions.  The object of the class doesn't need a vtable because this object will call it's member functions.  An object subclassed off the object with virtual functions will have a vtable so that at runtime, an object of this class will use it's member function and not the base classes.

**Does that mean that a VTABLE is created per class, no matter their object is created or not; considering only classes having virtual functions and those derived from them.**

A vtable is created per \*object\*, because only at runtime will the object know if it the subclassed class or base class.  This could be a lot of memory if you have 10000 objects of a class with virtual functions, but for normal data demands, you'll never have to worry about the extra memory.

**Give example how virtual functions are called?**

To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as “vtable”, “virtual function table”, “virtual method table”, or “dispatch table”.

Because knowing how the virtual table works is not necessary to use virtual functions, this section can be considered optional reading.

The virtual table is actually quite simple, though it’s a little complex to describe in words. First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it’s own virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

Second, the compiler also adds a hidden pointer to the base class, which we will call \*\_\_vptr. \*\_\_vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class. Unlike the \*this pointer, which is actually a function parameter used by the compiler to resolve self-references, \*\_\_vptr is a real pointer. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that \*\_\_vptr is inherited by derived classes, which is important.

By now, you’re probably confused as to how these things all fit together, so let’s take a look at a simple example:

class Base

{

public:

virtual void function1() {};

virtual void function2() {};

};

class D1: public Base

{

public:

virtual void function1() {};

};

class D2: public Base

{

public:

virtual void function2() {};

};

Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.

The compiler also adds a hidden pointer to the most base class that uses virtual functions. Although the compiler does this automatically, we’ll put it in the next example just to show where it’s added:

class Base

{

public:

FunctionPointer \*\_\_vptr;

virtual void function1() {};

virtual void function2() {};

};

class D1: public Base

{

public:

virtual void function1() {};

};

class D2: public Base

{

public:

virtual void function2() {};

};

When a class object is created, \*\_\_vptr is set to point to the virtual table for that class. For example, when a object of type Base is created, \*\_\_vptr is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, \*\_\_vptr is set to point to the virtual table for D1 or D2 respectively.

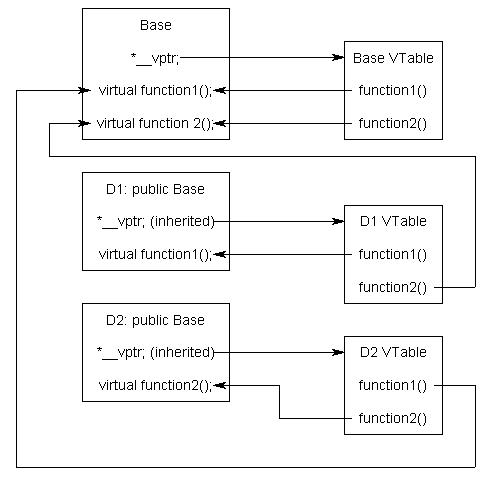
Now, let’s talk about how these virtual tables are filled out. Because there are only two virtual functions here, each virtual table will have two entries (one for function1(), and one for function2()). Remember that when these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.

Base’s virtual table is simple. An object of type Base can only access the members of Base. Base has no access to D1 or D2 functions. Consequently, the entry for function1 points to Base::function1(), and the entry for function2 points to Base::function2().

D1’s virtual table is slightly more complex. An object of type D1 can access members of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn’t overridden function2(), so the entry for function2 will point to Base::function2().

D2’s virtual table is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().

Here’s a picture of this graphically:



Although this diagram is kind of crazy looking, it’s really quite simple: the \*\_\_vptr in each class points to the virtual table for that class. The entries in the virtual table point to the most-derived version of the function objects of that class are allowed to call.

So consider what happens when we create an object of type D1:

int main()

{

D1 cClass;

}

Because cClass is a D1 object, cClass has it’s \*\_\_vptr set to the D1 virtual table.

Now, let’s set a base pointer to D1:

int main()

{

D1 cClass;

Base \*pClass = &cClass;

}

Note that because pClass is a base pointer, it only points to the Base portion of cClass. However, also note that \*\_\_vptr is in the Base portion of the class, so pClass has access to this pointer. Finally, note that pClass->\_\_vptr points to the D1 virtual table! Consequently, even though pClass is of type Base, it still has access to D1’s virtual table.

So what happens when we try to call pClass->function1()?

int main()

{

D1 cClass;

Base \*pClass = &cClass;

pClass->function1();

}

First, the program recognizes that function1() is a virtual function. Second, uses pClass->\_\_vptr to get to D1’s virtual table. Third, it looks up which version of function1() to call in D1’s virtual table. This has been set to D1::function1(). Therefore, pClass->function1() resolves to D1::function1()!

Now, you might be saying, “But what if Base really pointed to a Base object instead of a D1 object. Would it still call D1::function1()?”. The answer is no.

int main()

{

Base cClass;

Base \*pClass = &cClass;

pClass->function1();

}

In this case, when cClass is created, \_\_vptr points to Base’s virtual table, not D1’s virtual table. Consequently, pClass->\_\_vptr will also be pointing to Base’s virtual table. Base’s virtual table entry for function1() points to Base::function1(). Thus, pClass->function1() resolves to Base::function1(), which is the most-derived version of function1() that a Base object should be able to call.

By using these tables, the compiler and program are able to ensure function calls resolve to the appropriate virtual function, even if you’re only using a pointer or reference to a base class!

Calling a virtual function is slower than calling a non-virtual function for a couple of reasons: First, we have to use the \*\_\_vptr to get to the appropriate virtual table. Second, we have to index the virtual table to find the correct function to call. Only then can we call the function. As a result, we have to do 3 operations to find the function to call, as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call. However, with modern computers, this added time is usually fairly insignificant.

**Give example how virtual functions are called.**

A complete understanding of everything is very important in the programming world. I am taking this post to just give some explanations I know and the understanding I have with the Virtual Functions. This is an explanation given by one of the trainer who taught me C#. I'm not sure whether he has a blog or a personal website, if yes, I will definitely link him in my post.

Virtual functions are normal member functions of a class, which can be over-ridden in the derived classes. The whole functionality can be replaced in the over-riding function. In C#, the virtual functions will be declared with a keyword 'virtual' and the over-riding functions will be declared with 'override' key word.

Example in C#:

using System;

namespace ConsoleApplication1

{

class BaseClass

{

public BaseClass()

{

// TODO

}

virtual public void MyFunction1()

{

Console.WriteLine("MyFunction in Base class");

}

virtual public void MyFunction2()

{

Console.WriteLine("MyFunction in Base class");

}

}

class DerivedClass : BaseClass

{

public DerivedClass()

How the call to the derived class virtual function is being made? Let me present a C++ equivalent of the above code:

class BaseClass

{

public:

BaseClass()

{

}

virtual public void MyFunction1()

{

cout<<"MyFunction1 in Base class"<<endl;

}

virtual public void MyFunction2()

{

cout<<"MyFunction2 in Base class"<<endl;

}

};

class DerivedClass

{

public:

DerivedClass ()

{

}

public void MyFunction()

{

cout<<"MyFunction in Base class"<<endl;

}

};

void main()

{

DerivedClass \*obj;

obj->MyFunction();

}

I am going to explain the virtual functions with the C++ example and will give some more additional code which will explain the call semantics of the virtual functions. Whenever, there is a virtual function in the class, a v-table is constructed in the memory. The v-table has a list of addresses to the virtual functions of the class and pointers to the functions from each of the objects of the derived class. When a virtual function call is made, the v-table is used to get the addresses of the function and the function is called.

Let me also explain what are the steps that executed in a constructor. Before it executes the 1st line of code in the constructor body, the following steps are performed:

1. Base class constructer is called  
2. Call the contained object constructors  
3. Set the VPtr

The step we are interested is the 3rd step - setting the VPtr. Every object has a VPtr and is stored in the 1st memory location of the object memory. VPtr will have the address of the Virtual Table which has the function addresses of all the virtual functions available. Using the Vptr we can easily access the VirtualTable of the class and make function calls as normally done with the help of function pointers. So, as you all guessed by now, the base class constructor will be called and the VPtr of the base class will be set. But, when the call returns to the derived class constructor, the derived class VPtr will be set to its own VTable address.

How to get the VPtr of the class?

long \*vptr = (long \*) &obj;

// gets the vptr of the class which is stored in the 1st memory location of each object of the class

We got the VPtr, how we need to get the address of the VTable which is also a pointer.

long \*vtable = (long \*)\*vptr;

// get the vtable address

VTable can be considered as an array of function pointers. The addresses of virtual functions declared in the class will be stored in the order they are declared. The 0th location will have the 1st virtual function's address, 1st location have the 2nd virtual function address, so on and so forth. Now we have the address of the VTable in the pointer vtable. Lets declare a function pointer.

typedef void (\_\_stdcall \*FunctionPtr)();

FunctionPtr fp = (FunctionPtr)vtable[0];

// pointing to MyFunction1 which is in the 0th location of the v-table

We have also declared fp which is a function pointer holding a function that returns void, accepting no arguments. Let's now call the function:

fp();

fp = (FunctionPtr)vtable[1]; // pointing to 1st location which has MyFunction2

fp();

Now, lets talk about setting the VPtr (3rd step in a constructor - before any user written code is executed). It does set the Virtual Table address. It is just another step which is performed when ever a constructor is called. Now, lets review the below lines of code:

DerivedClass \*ptr = new DerivedClass();

ptr->MyFunction1(); // calls DerivedClass MyFunction1

ptr->BaseClass::BaseClass();

// BaseClass constructor is called - so VPtr altered to point to BaseClass's VTable

ptr->MyFunction1(); // calls BaseClass MyFunction1

From the comments you would have understood what is really happening. Fortunately, C# doesn't provide a way to call a base class constructor. However, you can use new operator to create a new copy of the BaseClass in this scenario to achieve the same result.

Are you a person who wish to combine many steps into one (hates to waste time in typing few more lines of code or love to minimize memory utilization by reducing the no. of variables declared or confuse someone who may try to understand your code)? If you are, following is another method of calling the first virtual function of that class:

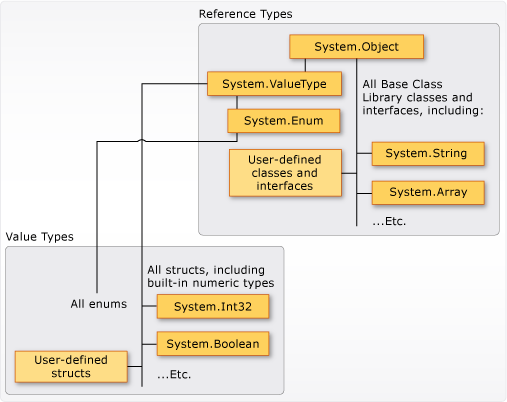
((void(\*)())\*(long \*)\*(long \*)&obj)();

**How value types and reference types related in the CTS?**

It is important to understand two fundamental points about the type system in the .NET Framework:

* It supports the principle of inheritance. Types can derive from other types, called base types. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of both base types in its inheritance hierarchy. All types, including built-in numeric types such as [System.Int32](http://msdn.microsoft.com/en-us/library/system.int32.aspx) (C# keyword: [int](http://msdn.microsoft.com/en-us/library/5kzh1b5w.aspx)), derive ultimately from a single base type, which is [System.Object](http://msdn.microsoft.com/en-us/library/system.object.aspx) (C# keyword: [object](http://msdn.microsoft.com/en-us/library/9kkx3h3c.aspx)). This unified type hierarchy is called the [Common Type System](http://msdn.microsoft.com/en-us/library/zcx1eb1e.aspx) (CTS). For more information about inheritance in C#, see [Inheritance (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/ms173149.aspx).
* Each type in the CTS is defined as either a value type or a reference type. This includes all custom types in the .NET Framework class library and also your own user-defined types. Types that you define by using the [struct](http://msdn.microsoft.com/en-us/library/ah19swz4.aspx) keyword are value types; all the built-in numeric types are structs. Types that you define by using the [class](http://msdn.microsoft.com/en-us/library/0b0thckt.aspx) keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.



**What is the advantage of all types derived from System.Object?**

Every object in the CLR derives from System.Object. Object is the base type of every type. In C#, the object keyword is an alias for System.Object. It can be convenient that every type in the CLR and in C# derives from Object. For example, you can treat a collection of instances of multiple types homogenously simply by casting them to Object references.

**Difference between abstract class and interface**

To know the difference between the two, a developer must think about [*abstraction*](http://en.wikipedia.org/wiki/Abstraction_(computer_science)#Abstraction_in_object_oriented_programming) and [*encapsulation*](http://en.wikipedia.org/wiki/Information_hiding); the two paradigm that [*Object Oriented Programming*](http://en.wikipedia.org/wiki/Object-oriented_programming) heavily relies on in modelling reality. Without inheritance and interfaces, we are stuck with complex trees of conditions, iterations and recursions that is probably duplicated again and again to describe a similar characteristic between two entities.

This post will discuss the difference between abstract and interface, along with an (awesome!!!) example – better than you’ve seen elsewhere.  
 **Abstract class:**

* cannot be instantiated.
* is a special type of class in which you can have members without implementation.
* as we know in C#/VB/Java, a class can only inherit from 1 class. This also applies for Abstract Class.
* normally used for framework-type library classes: providing default behavior for some of its class members, but forcing the developer to implement others.
* [is believed to be faster in Java](http://mindprod.com/jgloss/interfacevsabstract.html), **HOWEVER** I cannot find the same claim in .Net. The speed difference is \*probably\* negligible and only relevant to the most academic field.
* the aim: making sure something is \*eventually\* implemented.
* A class can inherit an abstract class without implementing all its abstract method.  
  However only a class that has all its method implemented can be instantiated to an object.
* **IS-A** relationship.
* e.g. Student **IS A** Person, Employee **IS A** Person.

// 'framework library' for a person

// a person can enrol and submit

// however, the class that consume this framework library

// need to provide 'where' the paperwork need to be sent

public abstract Person

{

public abstract SendPaperWork(string paperwork)

public void Enrol()

{

SendPaperWork("enrolment");

}

public void Submit()

{

SendPaperWork("report");

}

}

// by inheriting Person abstract class

// we are enabling student to enrol and submit

// however, SendPaperWork need to be implemented

// because we need to tell it explicitly 'where'

// to send the enrolment/ submission

public class Student : Person

{

public override SendPaperWork(string paperwork)

{

School.Send(paperwork);

}

}

// an employee send the paperwork to a different 'place' than student

public class Employee : Person

{

public override SendPaperWork(string paperwork)

{

Company.Send(paperwork);

}

}

**Interface:**

* cannot be instantiated.
* is a special type of abstract class in which all the members do not have any implementations.
* enables [polymorphism](http://en.wikipedia.org/wiki/Polymorphism_(computer_science)). A class can implement more than 1 Interfaces.
* normally used for application classes: providing contract for ensuring interactibility.
* the aim: making sure something is interchangeable.
* A class that implements an Interface need to contain all the implementation, otherwise the compiler will throw an error.
* **CAN-DO** relationship.
* e.g. Student **CAN** enrol, Student **CAN** submit assignment.

public interface ICanEnrol

{

void Enrol();

}

public interface ICanSubmit

{

void Submit();

}

public class Student : ICanEnrol, ICanSubmit

{

public void Enrol()

{

School.Send("enrolment");

}

public void Submit()

{

School.Send("report");

}

}

public class Employee : ICanEnrol, ICanSubmit

{

public void Enrol()

{

Company.Send("enrolment");

}

public void Submit()

{

Company.Send("report");

}

}

public class MailServer

{

public void SendAllSubmissions()

{

// AllSubmitters is a collection of students and employees

foreach (ICanSubmit submitter in AllSubmitters)

{

// The MailServer does not care if

// the submitter is a student

// or an employee, as long as it can submit

submitter.Submit()

}

}

}

**How the CLR Creates Runtime Objects?**

**Can we create a Java/C# class inside an interface?**

Yes, classes can be declared inside interfaces. This technique is sometimes used where the class is a constant type, return value or method argument in the interface. When a class is closely associated with the use of an interface it is convenient to declare it in the same compilation unit. This proximity also helps ensure that implementation changes to either are mutually compatible.

A class defined inside an interface is implicitly public static and operates as a top level class. The static modifier does not have the same effect on a nested class as it does with class variables and methods. The example below shows the definition of a StoreProcessor interface with nested StorageUnit class which is used in the two interface methods.

**Can an interface extend an abstract class?**

In Java/C# an interface cannot extend an abstract class. An interface may only extend a super-interface. And an abstract class may implement an interface. It may help to think of interfaces and classes as separate lines of inheritance that only come together when a class implements an interface, the relationship cannot be reversed.

**What is Marker Interface in .Net? How can I implement in Dot Net? What is the exact use of this? If possible please explain with an example.**

Marker interfaces as the name suggests, doesnt have anything in it, just used to mark a class.   
  
If you have worked with asp.net, you might have used IRequiredSessionState. This has no member, just acts as a Marker interface.   
  
In .NET we use Marker Interface using Attributes.

**Is it recommended to use Marker Interface?**

CONSIDER defining an interface if you need to support its functionality on types that already inherit from some other type.

AVOID using marker interfaces (interfaces with no members).

If you need to mark a class as having a specific characteristic (marker), in general, use a custom attribute rather than an interface.

*// Avoid*

public interface IImmutable {} *// empty interface*

public class Key: IImmutable {

...

}

*//Do*

[Immutable]

public class Key {

...

}

Methods can be implemented to reject parameters that are not marked with a specific attribute as follows:

public void Add(Key key, object value){

if(!key.GetType().IsDefined(typeof(ImmutableAttribute), false)){

throw new ArgumentException("The parameter must be immutable","key");

}

}

|  |
| --- |
| **RICO MARIANI** |
| Of course any kind of marking like this has a cost. Attribute testing is a lot more costly than type checking. You might find that it's necessary to use the marker interface approach for performance reasons—measure and see. My own experience is that true markers (with no members) don't come up very often. Most of the time, you need a no- kidding-around interface with actual functionality to do the job, in which case there is no choice to make. |

The problem with this approach is that the check for the custom attribute can occur only at runtime. Sometimes, it is very important that the check for the marker be done at compile-time. For example, a method that can serialize objects of any type might be more concerned with verifying the presence of the marker than with type verification at compile-time. Using marker interfaces might be acceptable in such situations. The following example illustrates this design approach:

public interface ITextSerializable {} *// empty interface*

public void Serialize(ITextSerializable item){

*// use reflection to serialize all public properties*

...

}

**DO** provide at least one type that is an implementation of an interface.

This helps to validate the design of the interface. For example, System.Collections.ArrayList is an implementation of the System.Collections.IList interface.

**DO** provide at least one API consuming each interface you define (a method taking the interface as a parameter or a property typed as the interface).

This helps to validate the interface design. For example, List<T>.Sort consumes IComparer<T> interface.

**DO NOT** add members to an interface that has previously shipped.

Doing so would break implementations of the interface. You should create a new interface to avoid versioning problems.

Except for the situations described in these guidelines, you should, in general, choose classes rather than interfaces in designing managed code reusable libraries.

**How we can use a class defined inside an interface?**   
**or Why we define a class inside an interface?**

A class is defined inside an interface to bind the interface to a TYPE.

A small but nonsense example:

interface employee{   
    class Role{   
          public String rollname;   
          public int Role id;   
          public Object person;   
     }   
    Role getRole();   
    // other methods   
}

In the above interface you are binding the Role type strongly to the employee interface (employee.Role).

Another usage than those linked by Eric P: defining a default/no-op implementation of the interface.

interface IEmployee

{

    void workHard ();

    void procrastinate ();

    class DefaultEmployee implements IEmployee

    {

        void workHard () { procrastinate(); };

        void procrastinate () {};

    }

}

Yet, another sample — implementation of Null Object Pattern:

interface IFoo

{

    void doFoo();

    IFoo NULL\_FOO = new NullFoo();

    final class NullFoo implements IFoo

    {

        public void doFoo () {};

        private NullFoo ()      {};

    }

}

...

IFoo foo = IFoo.NULL\_FOO;

...

bar.addFooListener (foo);

...

**Which of the following are true about the class defined inside an interface**1. it is not possible in the java Laungage.  
2. The class is always public.  
3. The class is always static.  
4. the class methods cannot call the methods declared in the interface.  
5. the class methods can call only the static methods declared in the interface.

You should know this first regarding nested classes in an interface:  
  
You can put the definition of a class inside the definition of an interface. The calss will be an inner class to the interface. An inner class to an interface will be static and public by default. The code structure would be like this :

**interface** Port {

//Methods & constants declared in the interface...

**class** Info {

//Definition of the class...

}

}

This declares the interface Port with an inner class Info. Objects of the inner class would be of type Port.Info. You might create one with a statement like this:  
  
Port.Info info = new Port.Info();  
  
A class that implements the inteface would have no direct connection with the inner class to the interface- it would just need to impleemnts the methods declared by the interface, but it is highly likely it would make use of objects of the inner class.  
  
On behalf of above description: Answer 2,3,4 are correct.  
5 is incorrect because interface methods can't be static.

**Write a method to copy items from one arraylist or array into other without using any .NET framework method like CopyTo() and counter.**

class Program

{

static void Main(string[] args)

{

List<string> toList;

List<string> fromList = new List<string>();

fromList.Add("One");

fromList.Add("Two");

fromList.Add("Three");

CopyTo(fromList, out toList);

foreach (var item in toList)

{

Console.WriteLine(item);

}

Console.ReadLine();

}

/// <summary>

/// Copies the contents

/// </summary>

/// <param name="fromList">List from which </param>

/// <param name="toList"></param>

private static void CopyTo(List<string> fromList, out List<string> toList)

{

toList = new List<string>();

//Like C# provides indexers to index elements within objects, C# also provides us with Enumerators to

//allow us enumerate through the elements within the objects. We need to use an Interface IEnumerator

//to enumerate through the elements of the object.

IEnumerator<string> enumeratorObj = fromList.GetEnumerator();

while (enumeratorObj.MoveNext())

{

toList.Add(enumeratorObj.Current);

}

}

}

**Why Array Index starts from Zero?**

1. The index of array, which is of the form a[i], is converted by the compiler in the form [a+i]. So, the index of first element is zero because [a+0] will give 'a' & the first array element can be accessed. Due to this, we can also access the array elements as 'i[a]' instead of 'a[i]', & this will not produce an error. But its compiler specific to start an array index with 1 or zero, as in pascal the array index starts with '1'.

2. This boils down to the concept of Binary digits. Take an array size of 64 for example. We start from 0 and end at 63. We require 6 bits. But, if we were to start from 1 and end at 64, we would require 7 bits to store the same number, thus increasing the storage size.

**Differentiate HashTable and Dictionary in C#.NET.**

|  |  |
| --- | --- |
| **HashTable** | **Dictionary** |
|  |  |
|  |  |
| Type unsafe | Type safe since it is a generic type. |
| Hashtable uses rehashing for collision resolution (when a collision occurs, tries another hash function to map the key to a bucket). | Dictionary relies on chaining (maintaining a list of items for each hash table bucket) to resolve collisions. |

**Why multiple inheritance is not supported by Java/C#?**

Due to the concept of overridding Java doesn't support multiple inheritance.

We have the concept of Deadly Diamond of Death .

i.e.

As we have an overridding concept in Java.

We know what overridding is where the method name including parameters return types should be same.

If mutliple inheritance is provided in java.

eg:

A<- B A-<C and now B C<-D

We have a method viz. display() in A we are overridding that in B and C. If D extends that two classes then if I call the display() then there will be ambiguity. To call which version of display().

As we dont have the concept of virtual functions what we have in c++.

This is the reason why we dont have multiple inheritance in java.

When the multiple inheritance is allowed, means when a language allows the class to extend multiple classes, that leads to the ambiguity as to which class method to consider, when two of its parents have the same method signature. This is called Diamond ring problem.

**What are the types of Inheritance?**

There are five types of inheritance.  
1. Single inheritance  
2. Multilevel inheritance  
3. Multiple inheritance  
4. Hierarchical inheritance  
5. Hybrid inheritance

We can take support for multiple inheritances also but with the help of Interface in Java/C#.