This page is specific to

Microsoft Visual Studio 2010/.NET Framework 4

.NET Framework Class Library

**Transaction Class**

Represents a transaction.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifInheritance Hierarchy

[System.Object](http://msdn.microsoft.com/en-us/library/system.object.aspx)  
**System.Transactions.Transaction**  
[System.Transactions.CommittableTransaction](http://msdn.microsoft.com/en-us/library/system.transactions.committabletransaction.aspx)  
[System.Transactions.DependentTransaction](http://msdn.microsoft.com/en-us/library/system.transactions.dependenttransaction.aspx)  
[System.Transactions.SubordinateTransaction](http://msdn.microsoft.com/en-us/library/system.transactions.subordinatetransaction.aspx)

**Namespace:** [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx)  
**Assembly:** System.Transactions (in System.Transactions.dll)

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifSyntax

Visual Basic

<SerializableAttribute> \_

Public Class Transaction \_

Implements IDisposable, ISerializable

C#

[SerializableAttribute]

public class Transaction : IDisposable,

ISerializable

Visual C++

[SerializableAttribute]

public ref class Transaction : IDisposable,

ISerializable

F#

[<SerializableAttribute>]

type Transaction =

class

interface IDisposable

interface ISerializable

end

The **Transaction** type exposes the following members.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifProperties

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Description** |
| Description: Public propertyDescription: Static member | [Current](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.current.aspx) | Gets or sets the ambient transaction. |
| Description: Public property | [IsolationLevel](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.isolationlevel.aspx) | Gets the isolation level of the transaction. |
| Description: Public property | [TransactionInformation](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.transactioninformation.aspx) | Retrieves additional information about a transaction. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifMethods

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Description** |
| Description: Public method | [Clone](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.clone.aspx) | Creates a clone of the transaction. |
| Description: Public method | [DependentClone](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.dependentclone.aspx) | Creates a dependent clone of the transaction. |
| Description: Public method | [Dispose](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.dispose.aspx) | Releases the resources that are held by the object. |
| Description: Public method | [EnlistDurable(Guid, IEnlistmentNotification, EnlistmentOptions)](http://msdn.microsoft.com/en-us/library/ms149773.aspx) | Enlists a durable resource manager that supports two phase commit to participate in a transaction. |
| Description: Public method | [EnlistDurable(Guid, ISinglePhaseNotification, EnlistmentOptions)](http://msdn.microsoft.com/en-us/library/ms149774.aspx) | Enlists a durable resource manager that supports single phase commit optimization to participate in a transaction. |
| Description: Public method | [EnlistPromotableSinglePhase](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.enlistpromotablesinglephase.aspx) | Enlists a resource manager that has an internal transaction using a promotable single phase enlistment (PSPE). |
| Description: Public method | [EnlistVolatile(IEnlistmentNotification, EnlistmentOptions)](http://msdn.microsoft.com/en-us/library/ms149779.aspx) | Enlists a volatile resource manager that supports two phase commit to participate in a transaction. |
| Description: Public method | [EnlistVolatile(ISinglePhaseNotification, EnlistmentOptions)](http://msdn.microsoft.com/en-us/library/ms149780.aspx) | Enlists a volatile resource manager that supports single phase commit optimization to participate in a transaction. |
| Description: Public method | [Equals](http://msdn.microsoft.com/en-us/library/ms149782.aspx) | Determines whether this transaction and the specified object are equal. (Overrides [Object..::.Equals(Object)](http://msdn.microsoft.com/en-us/library/bsc2ak47.aspx).) |
| Description: Protected method | [Finalize](http://msdn.microsoft.com/en-us/library/system.object.finalize.aspx) | Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from [Object](http://msdn.microsoft.com/en-us/library/system.object.aspx).) |
| Description: Public method | [GetHashCode](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.gethashcode.aspx) | Returns the hash code for this instance. (Overrides [Object..::.GetHashCode()()()](http://msdn.microsoft.com/en-us/library/system.object.gethashcode.aspx).) |
| Description: Public method | [GetType](http://msdn.microsoft.com/en-us/library/system.object.gettype.aspx) | Gets the [Type](http://msdn.microsoft.com/en-us/library/system.type.aspx) of the current instance. (Inherited from [Object](http://msdn.microsoft.com/en-us/library/system.object.aspx).) |
| Description: Protected method | [MemberwiseClone](http://msdn.microsoft.com/en-us/library/system.object.memberwiseclone.aspx) | Creates a shallow copy of the current [Object](http://msdn.microsoft.com/en-us/library/system.object.aspx). (Inherited from [Object](http://msdn.microsoft.com/en-us/library/system.object.aspx).) |
| Description: Public method | [Rollback()()()](http://msdn.microsoft.com/en-us/library/ms149787.aspx) | Rolls back (aborts) the transaction. |
| Description: Public method | [Rollback(Exception)](http://msdn.microsoft.com/en-us/library/ms149788.aspx) | Rolls back (aborts) the transaction. |
| Description: Public method | [ToString](http://msdn.microsoft.com/en-us/library/system.object.tostring.aspx) | Returns a string that represents the current object. (Inherited from [Object](http://msdn.microsoft.com/en-us/library/system.object.aspx).) |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifEvents

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Description** |
| Description: Public event | [TransactionCompleted](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.transactioncompleted.aspx) | Indicates that the transaction is completed. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifOperators

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Description** |
| Description: Public operatorDescription: Static member | [Equality](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.op_equality.aspx) | Tests whether two specified **Transaction** instances are equivalent. |
| Description: Public operatorDescription: Static member | [Inequality](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.op_inequality.aspx) | Returns a value that indicates whether two **Transaction** instances are not equal. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gif

Explicit Interface Implementations

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Description** |
| Description: Explicit interface implemetationDescription: Private method | [ISerializable..::.GetObjectData](http://msdn.microsoft.com/en-us/library/bb344312.aspx) | Gets a [SerializationInfo](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.serializationinfo.aspx) with the data required to serialize this transaction. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifRemarks

The [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions.aspx) namespace provides both an explicit programming model based on the **Transaction** class, as well as an implicit programming model using the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.aspx) class, in which transactions are automatically managed by the infrastructure. It is highly recommended that you use the easier implicit model for development. To get started, please see the [Implementing An Implicit Transaction Using Transaction Scope](http://msdn.microsoft.com/en-us/library/ms172152.aspx) topic. For more information on writing a transactional application, see [Writing A Transactional Application](http://msdn.microsoft.com/en-us/library/ms229973.aspx).

The **Transaction** class contains methods used by developers implementing resource managers for enlistment. It also provides functionalities for cloning a transaction and controlling the current transaction context. You can obtain the current transaction, if one is set, using the static [Current](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.current.aspx) property.

This page is specific to

Microsoft Visual Studio 2008/.NET Framework 3.5

**Implementing an Implicit Transaction using Transaction Scope**

The [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope(VS.90).aspx) class provides a simple way to mark a block of code as participating in a transaction, without requiring you to interact with the transaction itself. A transaction scope can select and manage the ambient transaction automatically. Due to its ease of use and efficiency, it is recommended that you use the **TransactionScope** class when developing a transaction application.

In addition, you do not need to enlist resources explicitly with the transaction. Any [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions(VS.90).aspx) resource manager (such as SQL Server 2005) can detect the existence of an ambient transaction created by the scope and automatically enlist.

**Creating a transaction scope**

The following sample shows a simple usage of the **TransactionScope** class.

The transaction scope is started once you create a new **TransactionScope** object. As illustrated in the code sample, it is recommended that you create scopes with a **using** statement. The **using** statement is available both in C# and in Visual Basic, and works like a **try...finally** block to ensure that the scope is disposed of properly.

When you instantiate **TransactionScope**, the transaction manager determines which transaction to participate in. Once determined, the scope always participates in that transaction. The decision is based on two factors: whether an ambient transaction is present and the value of the **TransactionScopeOption** parameter in the constructor. The ambient transaction is the transaction within which your code executes. You can obtain a reference to the ambient transaction by calling the static [Current](http://msdn.microsoft.com/en-us/library/system.transactions.transaction.current(VS.90).aspx) property of the [Transaction](http://msdn.microsoft.com/en-us/library/system.transactions.transaction(VS.90).aspx) class. For more information on how this parameter is used, see the Managing transaction flow using TransactionScopeOption section of this topic.

**Completing a transaction scope**

When your application completes all the work it wants to perform in a transaction, you should call the [Complete](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.complete(VS.90).aspx) method only once to inform the transaction manager that it is acceptable to commit the transaction. It is very good practice to put the call to **Complete** as the last statement in the **using** block.

Failing to call this method aborts the transaction, because the transaction manager interprets this as a system failure, or equivalent to an exception thrown within the scope of transaction. However, calling this method does not guarantee that the transaction wil be committed. It is merely a way of informing the transaction manager of your status. After calling the **Complete** method, you can no longer access the ambient transaction by using the **Current** property, and attempting to do so will result in an exception being thrown.

If the **TransactionScope** object created the transaction initially, the actual work of committing the transaction by the transaction manager occurs after the last line of code in the **using** block. If it did not create the transaction, the commit occurs whenever [Commit](http://msdn.microsoft.com/en-us/library/system.transactions.committabletransaction.commit(VS.90).aspx) is called by the owner of the [CommittableTransaction](http://msdn.microsoft.com/en-us/library/system.transactions.committabletransaction(VS.90).aspx) object. At that point the Transaction Manager calls the resource managers and informs them to either commit or rollback, based on whether the **Complete** method was called on the **TransactionScope** object.

The **using** statement ensures that the [Dispose](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope.dispose(VS.90).aspx) method of the **TransactionScope** object is called even if an exception occurs. The **Dispose** method marks the end of the transaction scope. Exceptions that occur after calling this method may not affect the transaction. This method also restores the ambient transaction to it previous state.

A [TransactionAbortedException](http://msdn.microsoft.com/en-us/library/system.transactions.transactionabortedexception(VS.90).aspx) is thrown if the scope creates the transaction, and the transaction is aborted. A [TransactionIndoubtException](http://msdn.microsoft.com/en-us/library/system.transactions.transactionindoubtexception(VS.90).aspx) is thrown if the transaction manager cannot reach a Commit decision. No exception is thrown if the transaction is committed.

**Rolling back a transaction**

If you want to rollback a transaction, you should not call the **Complete** method within the transaction scope. For example, you can throw an exception within the scope. The transaction in which it participates in will be rolled back.

**Managing transaction flow using TransactionScopeOption**

Transaction scope can be nested by calling a method that uses a **TransactionScope** from within a method that uses its own scope, as is the case with the RootMethod method in the following example,

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl11_code');" \o "Copy Code)

void RootMethod()

{

using(TransactionScope scope = new TransactionScope())

{

/\* Perform transactional work here \*/

SomeMethod();

scope.Complete();

}

}

void SomeMethod()

{

using(TransactionScope scope = new TransactionScope())

{

/\* Perform transactional work here \*/

scope.Complete();

}

}

The top-most transaction scope is referred to as the root scope.

The **TransactionScope** class provides several overloaded constructors that accept an enumeration of the type [TransactionScopeOption](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscopeoption(VS.90).aspx), which defines the transactional behavior of the scope.

A **TransactionScope** object has three options:

* Join the ambient transaction, or create a new one if one does not exist.
* Be a new root scope, that is, start a new transaction and have that transaction be the new ambient transaction inside its own scope.
* Not take part in a transaction at all. There is no ambient transaction as a result.

If the scope is instantiated with [Required](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscopeoption(VS.90).aspx), and an ambient transaction is present, the scope joins that transaction. If, on the other hand, there is no ambient transaction, then the scope creates a new transaction, and become the root scope. This is the default value. When **Required** is used, the code inside the scope does not need to behave differently whether it is the root or just joining the ambient transaction. It should operate identically in both cases.

If the scope is instantiated with [RequiresNew](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscopeoption(VS.90).aspx), it is always the root scope. It starts a new transaction, and its transaction becomes the new ambient transaction inside the scope.

If the scope is instantiated with [Suppress](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscopeoption(VS.90).aspx), it never takes part in a transaction, regardless of whether an ambient transaction is present. A scope instantiated with this value always have **null** as its ambient transaction.

The above options are summarized in the following table.

|  |  |  |
| --- | --- | --- |
| **TransactionScopeOption** | **Ambient Transaction** | **The scope takes part in** |
| Required | No | New Transaction (will be the root) |
| Requires New | No | New Transaction (will be the root) |
| Suppress | No | No Transaction |
| Required | Yes | Ambient Transaction |
| Requires New | Yes | New Transaction (will be the root) |
| Suppress | Yes | No Transaction |

When a **TransactionScope** object joins an existing ambient transaction, disposing of the scope object may not end the transaction, unless the scope aborts the transaction. If the ambient transaction was created by a root scope, only when the root scope is disposed of, does **Commit** get called on the transaction. If the transaction was created manually, the transaction ends when it is either aborted, or committed by its creator.

The following example shows a **TransactionScope** object that creates three nested scope objects, each instantiated with a different **TransactionScopeOption** value.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl16_code');" \o "Copy Code)

using(TransactionScope scope1 = new TransactionScope())

//Default is Required

{

using(TransactionScope scope2 = new

TransactionScope(TransactionScopeOption.Required))

{

...

}

using(TransactionScope scope3 = new TransactionScope(TransactionScopeOption.RequiresNew))

{

...

}

using(TransactionScope scope4 = new

TransactionScope(TransactionScopeOption.Suppress))

{

...

}

}

The example shows a code block without any ambient transaction creating a new scope (scope1) with **Required**. The scope scope1 is a root scope as it creates a new transaction (Transaction A) and makes Transaction A the ambient transaction. Scope1 then creates three more objects, each with a different **TransactionScopeOption** value. For example, scope2 is created with **Required**, and since there is an ambient transaction, it joins the first transaction created by scope1. Note that scope3 is the root scope of a new transaction, and that scope4 has no ambient transaction.

Although the default and most commonly used value of **TransactionScopeOption** is **Required**, each of the other values has its unique purpose.

**Suppress** is useful when you want to preserve the operations performed by the code section, and do not want to abort the ambient transaction if the operations fail. For example, when you want to perform logging or audit operations, or when you want to publish events to subscribers regardless of whether your ambient transaction commits or aborts. This value allows you to have a non-transactional code section inside a transaction scope, as shown in the following example.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl17_code');" \o "Copy Code)

using(TransactionScope scope1 = new TransactionScope())

{

try

{

//Start of non-transactional section

using(TransactionScope scope2 = new

TransactionScope(TransactionScopeOption.Suppress))

{

//Do non-transactional work here

}

//Restores ambient transaction here

}

catch

{}

//Rest of scope1

}

**Voting inside a nested scope**

Although a nested scope can join the ambient transaction of the root scope, calling **Complete** in the nested scope has no affect on the root scope. Only if all the scopes from the root scope down to the last nested scope vote to commit the transaction, will the transaction be committed.

**Setting the TransactionScope timeout**

Some of the overloaded constructors of **TransactionScope** accept a value of type [TimeSpan](http://msdn.microsoft.com/en-us/library/system.timespan(VS.90).aspx), which is used to control the timeout of the transaction. A timeout set to zero means an infinite timeout. Infinite timeout is useful mostly for debugging, when you want to isolate a problem in your business logic by stepping through your code, and you do not want the transaction you debug to time out while you attempt to locate the problem. Be extremely careful using the infinite timeout value in all other cases, because it overrides the safeguards against transaction deadlocks.

You typically set the **TransactionScope** timeout to values other than default in two cases. The first is during development, when you want to test the way your application handles aborted transactions. By setting the timeout to a small value (such as one millisecond), you cause your transaction to fail and can thus observe your error handling code. The second case in which you set the value to be less than the default timeout is when you believe that the scope is involved in resource contention, resulting in deadlocks. In that case, you want to abort the transaction as soon as possible and not wait for the default timeout to expire.

When a scope joins an ambient transaction but specifies a smaller timeout than the one the ambient transaction is set to, the new, shorter timeout is enforced on the **TransactionScope** object, and the scope must end within the nested time specified, or the transaction is automatically aborted. If the nested scope's timeout is more than that of the ambient transaction, it has no effect.

**Setting the TransactionScope isolation level**

Some of the overloaded constructors of **TransactionScope** accept a structure of type [TransactionOptions](http://msdn.microsoft.com/en-us/library/system.transactions.transactionoptions(VS.90).aspx) to specify an isolation level, in addition to a timeout value. By default, the transaction executes with isolation level set to [Serializable](http://msdn.microsoft.com/en-us/library/system.transactions.isolationlevel(VS.90).aspx). Selecting an isolation level other than **Serializable** is commonly used for read-intensive systems. This requires a solid understanding of transaction processing theory and the semantics of the transaction itself, the concurrency issues involved, and the consequences for system consistency.

In addition, not all resource managers support all levels of isolation, and they may elect to take part in the transaction at a higher level than the one configured.

Every isolation level besides **Serializable** is susceptible to inconsistency resulting from other transactions accessing the same information. The difference between the different isolation levels is in the way read and write locks are used. A lock can be held only when the transaction accesses the data in the resource manager, or it can be held until the transaction is committed or aborted. The former is better for throughput, the latter for consistency. The two kinds of locks and the two kinds of operations (read/write) give four basic isolation levels. See [IsolationLevel](http://msdn.microsoft.com/en-us/library/system.transactions.isolationlevel(VS.90).aspx) for more information.

When using nested **TransactionScope** objects, all nested scopes must be configured to use exactly the same isolation level if they want to join the ambient transaction. If a nested **TransactionScope** object tries to join the ambient transaction yet it specifies a different isolation level, an [ArgumentException](http://msdn.microsoft.com/en-us/library/system.argumentexception(VS.90).aspx) is thrown.

**Interop with COM+**

When you create a new **TransactionScope** instance, you can use the [EnterpriseServicesInteropOption](http://msdn.microsoft.com/en-us/library/system.transactions.enterpriseservicesinteropoption(VS.90).aspx) enumeration in one of the constructors to specify how to interact with COM+. For more information on this, see [Interoperability with Enterprise Services and COM+ Transactions](http://msdn.microsoft.com/en-us/library/ms229974(VS.90).aspx).

This page is specific to

Microsoft Visual Studio 2008/.NET Framework 3.5

**Writing a Transactional Application**

As a transactional application programmer, you can take advantage of the two programming models provided by the [System.Transactions](http://msdn.microsoft.com/en-us/library/system.transactions(VS.90).aspx) namespace to create a transaction. You can utilize the explicit programming model by using the [Transaction](http://msdn.microsoft.com/en-us/library/system.transactions.transaction(VS.90).aspx) class, or the implicit programming model in which transactions are automatically managed by the infrastructure, by using the [TransactionScope](http://msdn.microsoft.com/en-us/library/system.transactions.transactionscope(VS.90).aspx) class.. It is recommended that you use the implicit transaction model for development. You can find more information on how to use a transaction scope in the [Implementing an Implicit Transaction using Transaction Scope](http://msdn.microsoft.com/en-us/library/ms172152(VS.90).aspx) topic.

Both models support committing a transaction when the program reaches a consistent state. If the commit succeeds, the transaction is durably committed. If the commit fails, the transaction aborts. If the application program cannot successfully complete the transaction, it attempts to abort and undo the transaction's effects.

## In This Section

### Creating a Transaction

The **System.Transactions** namespace provides two models for creating a transaction. These models are covered in the following topics.

[Implementing an Implicit Transaction using Transaction Scope](http://msdn.microsoft.com/en-us/library/ms172152(VS.90).aspx)

Describes how the **System.Transactions** namespace supports creating implicit transactions using the **TransactionScope** class.

[Implementing an Explicit Transaction using CommittableTransaction](http://msdn.microsoft.com/en-us/library/ms172146(VS.90).aspx)

Describes how the **System.Transactions** namespace supports creating explicit transactions using the [CommittableTransaction](http://msdn.microsoft.com/en-us/library/system.transactions.committabletransaction(VS.90).aspx) class.

### Escalating Transaction Management

When a transaction needs to access a resource in another application domain, or if you want to enlist in another durable resource manager, the transaction is automatically escalated to be managed by the MSDTC. Transaction escalation is covered in the [Transaction Management Escalation](http://msdn.microsoft.com/en-us/library/ms229978(VS.90).aspx) topic.

### Concurrency

The topic [Managing Concurrency with DependentTransaction](http://msdn.microsoft.com/en-us/library/ms229976(VS.90).aspx) demonstrates how concurrency can be achieved between asynchronous tasks by using the [DependentTransaction](http://msdn.microsoft.com/en-us/library/system.transactions.dependenttransaction(VS.90).aspx) class.

### COM+ Interop

The topic [Interoperability with Enterprise Services and COM+ Transactions](http://msdn.microsoft.com/en-us/library/ms229974(VS.90).aspx) illustrates how you can make your distributed transactions interact with COM+ transactions.

### Diagnostics

[Diagnostic Traces](http://msdn.microsoft.com/en-us/library/ms229979(VS.90).aspx) describes how you can use the trace codes that are generated by the **System.Transactions** infrastructure to troubleshoot errors in your applications.

### Working within ASP.NET

The [Using System.Transactions in ASP.NET](http://msdn.microsoft.com/en-us/library/ms229977(VS.90).aspx) topic describes how you can successfully use **System.Transactions** inside an ASP.NET application.