Contents

Programming Concepts (C#)

2/23/2019 • 2 minutes to read • Edit Online

This section explains programming concepts in the C# language.

In This Section

TITLE	DESCRIPTION
Assemblies in .NET	Describes how to create and use assemblies.
Asynchronous Programming with async and await (C#)	Describes how to write asynchronous solutions by using the async and await keywords in C#. Includes a walkthrough.
Attributes (C#)	Discusses how to provide additional information about programming elements such as types, fields, methods, and properties by using attributes.
Caller Information (C#)	Describes how to obtain information about the caller of a method. This information includes the file path and the line number of the source code and the member name of the caller.
Collections (C#)	Describes some of the types of collections provided by the .NET Framework. Demonstrates how to use simple collections and collections of key/value pairs.
Covariance and Contravariance (C#)	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
Expression Trees (C#)	Explains how you can use expression trees to enable dynamic modification of executable code.
Iterators (C#)	Describes iterators, which are used to step through collections and return elements one at a time.
Language-Integrated Query (LINQ) (C#)	Discusses the powerful query capabilities in the language syntax of C#, and the model for querying relational databases, XML documents, datasets, and in-memory collections.
Object-Oriented Programming (C#)	Describes common object-oriented concepts, including encapsulation, inheritance, and polymorphism.
Reflection (C#)	Explains how to use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.
Serialization (C#)	Describes key concepts in binary, XML, and SOAP serialization.

Related Sections

Performance Tips	Discusses several basic rules that may help you increase the performance of your application.

2 minutes to read

The Task asynchronous programming model in C#

5/6/2019 • 10 minutes to read • Edit Online

The Task asynchronous programming model (TAP) provides an abstraction over asynchronous code. You write code as a sequence of statements, just like always. You can read that code as though each statement completes before the next begins. The compiler performs a number of transformations because some of those statements may start work and return a Task that represents the ongoing work.

That's the goal of this syntax: enable code that reads like a sequence of statements, but executes in a much more complicated order based on external resource allocation and when tasks complete. It's analogous to how people give instructions for processes that include asynchronous tasks. Throughout this article, you'll use an example of instructions for making a breakfast to see how the `async` and `await` keywords make it easier to reason about code that includes a series of asynchronous instructions. You'd write the instructions something like the following list to explain how to make a breakfast:

1. Pour a cup of coffee.
2. Heat up a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you'd execute those instructions **asynchronously**. you'd start warming the pan for eggs, then start the bacon. You'd put the bread in the toaster, then start the eggs. At each step of the process, you'd start a task, then turn your attention to tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all these tasks. Continuing the breakfast analogy, one person can make breakfast asynchronously by starting the next task before the first completes. The cooking progresses whether or not someone is watching it. As soon as you start warming the pan for the eggs, you can begin frying the bacon. Once the bacon starts, you can put the bread into the toaster.

For a parallel algorithm, you'd need multiple cooks (or threads). One would make the eggs, one the bacon, and so on. Each one would be focused on just that one task. Each cook (or thread) would be blocked synchronously waiting for bacon to be ready to flip, or the toast to pop.

Now, consider those same instructions written as C# statements:

```
static void Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    Egg eggs = FryEggs(2);
    Console.WriteLine("eggs are ready");
    Bacon bacon = FryBacon(3);
    Console.WriteLine("bacon is ready");
    Toast toast = ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");
}
```

Computers don't interpret those instructions the same way people do. The computer will block on each statement until the work is complete before moving on to the next statement. That creates an unsatisfying breakfast. The later tasks wouldn't be started until the earlier tasks had completed. It would take much longer to create the breakfast, and some items would have gotten cold before being served.

If you want the computer to execute the above instructions asynchronously, you must write asynchronous code.

These concerns are important for the programs you write today. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't make a phone appear frozen while it's downloading data from the web. When you write server programs, you don't want threads blocked. Those threads could be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for those blocked threads.

Successful modern applications require asynchronous code. Without language support, writing asynchronous code required callbacks, completion events, or other means that obscured the original intent of the code. The advantage of the synchronous code is that it's easy to understand. The step-by-step actions make it easy to scan and understand. Traditional asynchronous models forced you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

Don't block, await instead

The preceding code demonstrates a bad practice: constructing synchronous code to perform asynchronous operations. As written, this code blocks the thread executing it from doing any other work. It won't be interrupted while any of the tasks are in progress. It would be as though you stared at the toaster after putting the bread in. You'd ignore anyone talking to you until the toast popped.

Let's start by updating this code so that the thread doesn't block while tasks are running. The `await` keyword provides a non-blocking way to start a task, then continue execution when that task completes. A simple asynchronous version of the make a breakfast code would look like the following snippet:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    Egg eggs = await FryEggs(2);
    Console.WriteLine("eggs are ready");
    Bacon bacon = await FryBacon(3);
    Console.WriteLine("bacon is ready");
    Toast toast = await ToastBread(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");
}
```

This code doesn't block while the eggs or the bacon are cooking. This code won't start any other tasks though. You'd still put the toast in the toaster and stare at it until it pops. But at least, you'd respond to anyone that wanted your attention. In a restaurant where multiple orders are placed, the cook could start another breakfast while the first is cooking.

Now, the thread working on the breakfast isn't blocked while awaiting any started task that hasn't yet finished. For some applications, this change is all that's needed. A GUI application still responds to the user with just this change. However, for this scenario, you want more. You don't want each of the component tasks to be executed sequentially. It's better to start each of the component tasks before awaiting the previous task's completion.

Start tasks concurrently

In many scenarios, you want to start several independent tasks immediately. Then, as each task finishes, you can continue other work that's ready. In the breakfast analogy, that's how you get breakfast done more quickly. You also get everything done close to the same time. You'll get a hot breakfast.

The System.Threading.Tasks.Task and related types are classes you can use to reason about tasks that are in progress. That enables you to write code that more closely resembles the way you'd actually create breakfast. You'd start cooking the eggs, bacon, and toast at the same time. As each requires action, you'd turn your attention to that task, take care of the next action, then await for something else that requires your attention.

You start a task and hold on to the Task object that represents the work. You'll `await` each task before working with its result.

Let's make these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than awaiting them:

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Task<Egg> eggTask = FryEggs(2);
Egg eggs = await eggTask;
Console.WriteLine("eggs are ready");
Task<Bacon> baconTask = FryBacon(3);
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");
Task<Toast> toastTask = ToastBread(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Console.WriteLine("Breakfast is ready!");
```

Next, you can move the `await` statements for the bacon and eggs to the end of the method, before serving breakfast:

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Task<Egg> eggTask = FryEggs(2);
Task<Bacon> baconTask = FryBacon(3);
Task<Toast> toastTask = ToastBread(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("oj is ready");

Egg eggs = await eggTask;
Console.WriteLine("eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");

Console.WriteLine("Breakfast is ready!");
```

The preceding code works better. You start all the asynchronous tasks at once. You await each task only when you need the results. The preceding code may be similar to code in a web application that makes requests of different microservices, then combines the results into a single page. You'll make all the requests immediately, then `await` all those tasks and compose the web page.

Composition with tasks

You have everything ready for breakfast at the same time except the toast. Making the toast is the composition of an asynchronous operation (toasting the bread), and synchronous operations (adding the butter and the jam). Updating this code illustrates an important concept:

> **IMPORTANT**
> The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

The preceding code showed you that you can use Task or Task<TResult> objects to hold running tasks. You `await` each task before using its result. The next step is to create methods that represent the combination of other work. Before serving breakfast, you want to await the task that represents toasting the bread before adding butter and

jam. You can represent that work with the following code:

```
async Task<Toast> makeToastWithButterAndJamAsync(int number)
{
    var plainToast = await ToastBreadAsync(number);
    ApplyButter(plainToast);
    ApplyJam(plainToast);
    return plainToast;
}
```

The preceding method has the `async` modifier in its signature. That signals to the compiler that this method contains an `await` statement; it contains asynchronous operations. This method represents the task that toasts the bread, then adds butter and jam. This method returns a Task<TResult> that represents the composition of those three operations. The main block of code now becomes:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = makeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");
    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");
    var toast = await toastTask;
    Console.WriteLine("toast is ready");
    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");

    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> makeToastWithButterAndJamAsync(int number)
    {
        var plainToast = await ToastBreadAsync(number);
        ApplyButter(plainToast);
        ApplyJam(plainToast);
        return plainToast;
    }
}
```

The previous change illustrated an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to await that task. You can start other tasks concurrently.

Await tasks efficiently

The series of `await` statements at the end of the preceding code can be improved by using methods of the `Task` class. One of those APIs is WhenAll, which returns a Task that completes when all the tasks in its argument list have completed, as shown in the following code:

```
await Task.WhenAll(eggTask, baconTask, toastTask);
Console.WriteLine("eggs are ready");
Console.WriteLine("bacon is ready");
Console.WriteLine("toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Another option is to use WhenAny, which returns a `Task<Task>` that completes when any of its arguments completes. You can await the returned task, knowing that it has already finished. The following code shows how you could use WhenAny to await the first task to finish and then process its result. After processing the result from the completed task, you remove that completed task from the list of tasks passed to `WhenAny`.

```
var allTasks = new List<Task>{eggsTask, baconTask, toastTask};
while (allTasks.Any())
{
    Task finished = await Task.WhenAny(allTasks);
    if (finished == eggsTask)
    {
        Console.WriteLine("eggs are ready");
        allTasks.Remove(eggsTask);
        var eggs = await eggsTask;
    } else if (finished == baconTask)
    {
        Console.WriteLine("bacon is ready");
        allTasks.Remove(baconTask);
        var bacon = await baconTask;
    } else if (finished == toastTask)
    {
        Console.WriteLine("toast is ready");
        allTasks.Remove(toastTask);
        var toast = await toastTask;
    } else
            allTasks.Remove(finished);
}
Console.WriteLine("Breakfast is ready!");
```

After all those changes, the final version of `Main` looks like the following code:

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = makeToastWithButterAndJamAsync(2);

    var allTasks = new List<Task>{eggsTask, baconTask, toastTask};
    while (allTasks.Any())
    {
        Task finished = await Task.WhenAny(allTasks);
        if (finished == eggsTask)
        {
            Console.WriteLine("eggs are ready");
            allTasks.Remove(eggsTask);
            var eggs = await eggsTask;
        } else if (finished == baconTask)
        {
            Console.WriteLine("bacon is ready");
            allTasks.Remove(baconTask);
            var bacon = await baconTask;
        } else if (finished == toastTask)
        {
            Console.WriteLine("toast is ready");
            allTasks.Remove(toastTask);
            var toast = await toastTask;
        } else
                allTasks.Remove(finished);
    }
    Console.WriteLine("Breakfast is ready!");

    async Task<Toast> makeToastWithButterAndJamAsync(int number)
    {
        var plainToast = await ToastBreadAsync(number);
        ApplyButter(plainToast);
        ApplyJam(plainToast);
        return plainToast;
    }
}
```

This final code is asynchronous. It more accurately reflects how a person would cook a breakfast. Compare the preceding code with the first code sample in this article. The core actions are still clear from reading the code. You can read this code the same way you'd read those instructions for making a breakfast at the beginning of this article. The language features for `async` and `await` provide the translation every person makes to follow those written instructions: start tasks as you can and don't block waiting for tasks to complete.

Attributes (C#)

2/20/2019 • 5 minutes to read • Edit Online

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called *reflection*. For more information, see Reflection (C#).

Attributes have the following properties:

- Attributes add metadata to your program. *Metadata* is information about the types defined in a program. All .NET assemblies contain a specified set of metadata that describes the types and type members defined in the assembly. You can add custom attributes to specify any additional information that is required. For more information, see, Creating Custom Attributes (C#).
- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Your program can examine its own metadata or the metadata in other programs by using reflection. For more information, see Accessing Attributes by Using Reflection (C#).

Using attributes

Attributes can be placed on most any declaration, though a specific attribute might restrict the types of declarations on which it is valid. In C#, you specify an attribute by placing the name of the attribute enclosed in square brackets ([]) above the declaration of the entity to which it applies.

In this example, the SerializableAttribute attribute is used to apply a specific characteristic to a class:

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

A method with the attribute DllImportAttribute is declared like the following example:

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

More than one attribute can be placed on a declaration as the following example shows:

```
using System.Runtime.InteropServices;
```

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is ConditionalAttribute:

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

> **NOTE**
>
> By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET libraries. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but `DllImportAttribute` is the attribute's actual name in the .NET Framework Class Library.

Attribute parameters

Many attributes have parameters, which can be positional, unnamed, or named. Any positional parameters must be specified in a certain order and cannot be omitted. Named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted. Positional parameters correspond to the parameters of the attribute constructor. Named or optional parameters correspond to either properties or fields of the attribute. Refer to the individual attribute's documentation for information on default parameter values.

Attribute targets

The *target* of an attribute is the entity which the attribute applies to. For example, an attribute may apply to a class, a particular method, or an entire assembly. By default, an attribute applies to the element that it precedes. But you can also explicitly identify, for example, whether an attribute is applied to a method, or to its parameter, or to its return value.

To explicitly identify an attribute target, use the following syntax:

```
[target : attribute-list]
```

The list of possible `target` values is shown in the following table.

TARGET VALUE	APPLIES TO
`assembly`	Entire assembly
`module`	Current assembly module
`field`	Field in a class or a struct
`event`	Event
`method`	Method or `get` and `set` property accessors

TARGET VALUE	APPLIES TO
`param`	Method parameters or `set` property accessor parameters
`property`	Property
`return`	Return value of a method, property indexer, or `get` property accessor
`type`	Struct, class, interface, enum, or delegate

You would specify the `field` target value to apply an attribute to the backing field created for an auto-implemented property.

The following example shows how to apply attributes to assemblies and modules. For more information, see Common Attributes (C#).

```
using System;
using System.Reflection;
[assembly: AssemblyTitleAttribute("Production assembly 4")]
[module: CLSCompliant(true)]
```

The following example shows how to apply attributes to methods, method parameters, and method return values in C#.

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to return value
[return: ValidatedContract]
int Method3() { return 0; }
```

NOTE

Regardless of the targets on which `ValidatedContract` is defined to be valid, the `return` target has to be specified, even if `ValidatedContract` were defined to apply only to return values. In other words, the compiler will not use `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see AttributeUsage (C#).

Common uses for attributes

The following list includes a few of the common uses of attributes in code:

- Marking methods using the `WebMethod` attribute in Web services to indicate that the method should be callable over the SOAP protocol. For more information, see WebMethodAttribute.
- Describing how to marshal method parameters when interoperating with native code. For more information, see MarshalAsAttribute.
- Describing the COM properties for classes, methods, and interfaces.
- Calling unmanaged code using the DllImportAttribute class.
- Describing your assembly in terms of title, version, description, or trademark.

- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtaining information about the caller to a method.

Related sections

For more information, see:

- Creating Custom Attributes (C#)
- Accessing Attributes by Using Reflection (C#)
- How to: Create a C/C++ Union by Using Attributes (C#)
- Common Attributes (C#)
- Caller Information (C#)

See also

- C# Programming Guide
- Reflection (C#)
- Attributes
- Using Attributes in C#

Caller Information (C#)

10/3/2018 • 2 minutes to read • Edit Online

By using Caller Info attributes, you can obtain information about the caller to a method. You can obtain file path of the source code, the line number in the source code, and the member name of the caller. This information is helpful for tracing, debugging, and creating diagnostic tools.

To obtain this information, you use attributes that are applied to optional parameters, each of which has a default value. The following table lists the Caller Info attributes that are defined in the System.Runtime.CompilerServices namespace:

ATTRIBUTE	DESCRIPTION	TYPE
CallerFilePathAttribute	Full path of the source file that contains the caller. This is the file path at compile time.	`String`
CallerLineNumberAttribute	Line number in the source file at which the method is called.	`Integer`
CallerMemberNameAttribute	Method or property name of the caller. See Member Names later in this topic.	`String`

Example

The following example shows how to use Caller Info attributes. On each call to the `TraceMessage` method, the caller information is substituted as arguments to the optional parameters.

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",
        [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",
        [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)
{
    System.Diagnostics.Trace.WriteLine("message: " + message);
    System.Diagnostics.Trace.WriteLine("member name: " + memberName);
    System.Diagnostics.Trace.WriteLine("source file path: " + sourceFilePath);
    System.Diagnostics.Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
//  message: Something happened.
//  member name: DoProcessing
//  source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
//  source line number: 31
```

Remarks

You must specify an explicit default value for each optional parameter. You can't apply Caller Info attributes to parameters that aren't specified as optional.

The Caller Info attributes don't make a parameter optional. Instead, they affect the default value that's passed in when the argument is omitted.

Caller Info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the StackTrace property for exceptions, the results aren't affected by obfuscation.

You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that **Rename Refactoring** doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the INotifyPropertyChanged interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed, so that the control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the `CallerMemberName` attribute.

CALLS OCCURS WITHIN	MEMBER NAME RESULT
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"
Static constructor	The string ".cctor"
Destructor	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the method or property to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

See also

- Attributes (C#)
- Common Attributes (C#)
- Named and Optional Arguments
- Programming Concepts (C#)

Collections (C#)

4/11/2019 • 13 minutes to read • Edit Online

For many applications, you want to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections of objects.

Arrays are most useful for creating and working with a fixed number of strongly-typed objects. For information about arrays, see Arrays.

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change. For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

A collection is a class, so you must declare an instance of the class before you can add elements to that collection.

If your collection contains elements of only one data type, you can use one of the classes in the System.Collections.Generic namespace. A generic collection enforces type safety so that no other data type can be added to it. When you retrieve an element from a generic collection, you do not have to determine its data type or convert it.

> **NOTE**
> For the examples in this topic, include using directives for the `System.Collections.Generic` and `System.Linq` namespaces.

In this topic

Using a Simple Collection

The examples in this section use the generic List<T> class, which enables you to work with a strongly typed list of objects.

The following example creates a list of strings and then iterates through the strings by using a foreach statement.

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

If the contents of a collection are known in advance, you can use a *collection initializer* to initialize the collection. For more information, see Object and Collection Initializers.

The following example is the same as the previous example, except a collection initializer is used to add elements to the collection.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

You can use a for statement instead of a `foreach` statement to iterate through a collection. You accomplish this by accessing the collection elements by the index position. The index of the elements starts at 0 and ends at the element count minus 1.

The following example iterates through the elements of a collection by using `for` instead of `foreach`.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

The following example removes an element from the collection by specifying the object to remove.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

The following example removes elements from a generic list. Instead of a `foreach` statement, a for statement that iterates in descending order is used. This is because the RemoveAt method causes elements after a removed element to have a lower index value.

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

For the type of elements in the List<T>, you can also define your own class. In the following example, the `Galaxy` class that is used by the List<T> is defined in the code.

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
        {
            new Galaxy() { Name="Tadpole", MegaLightYears=400},
            new Galaxy() { Name="Pinwheel", MegaLightYears=25},
            new Galaxy() { Name="Milky Way", MegaLightYears=0},
            new Galaxy() { Name="Andromeda", MegaLightYears=3}
        };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "   " + theGalaxy.MegaLightYears);
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

Kinds of Collections

Many common collections are provided by the .NET Framework. Each type of collection is designed for a specific purpose.

Some of the common collection classes are described in this section:

- System.Collections.Generic classes
- System.Collections.Concurrent classes
- System.Collections classes

System.Collections.Generic Classes

You can create a generic collection by using one of the classes in the System.Collections.Generic namespace. A generic collection is useful when every item in the collection has the same data type. A generic collection enforces strong typing by allowing only the desired data type to be added.

The following table lists some of the frequently used classes of the System.Collections.Generic namespace:

CLASS	DESCRIPTION
Dictionary<TKey,TValue>	Represents a collection of key/value pairs that are organized based on the key.
List<T>	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
Queue<T>	Represents a first in, first out (FIFO) collection of objects.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.

CLASS	DESCRIPTION
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

For additional information, see Commonly Used Collection Types, Selecting a Collection Class, and System.Collections.Generic.

System.Collections.Concurrent Classes

In the .NET Framework 4 or newer, the collections in the System.Collections.Concurrent namespace provide efficient thread-safe operations for accessing collection items from multiple threads.

The classes in the System.Collections.Concurrent namespace should be used instead of the corresponding types in the System.Collections.Generic and System.Collections namespaces whenever multiple threads are accessing the collection concurrently. For more information, see Thread-Safe Collections and System.Collections.Concurrent.

Some classes included in the System.Collections.Concurrent namespace are BlockingCollection<T>, ConcurrentDictionary<TKey,TValue>, ConcurrentQueue<T>, and ConcurrentStack<T>.

System.Collections Classes

The classes in the System.Collections namespace do not store elements as specifically typed objects, but as objects of type `Object`.

Whenever possible, you should use the generic collections in the System.Collections.Generic namespace or the System.Collections.Concurrent namespace instead of the legacy types in the `System.Collections` namespace.

The following table lists some of the frequently used classes in the `System.Collections` namespace:

CLASS	DESCRIPTION
ArrayList	Represents an array of objects whose size is dynamically increased as required.
Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.
Queue	Represents a first in, first out (FIFO) collection of objects.
Stack	Represents a last in, first out (LIFO) collection of objects.

The System.Collections.Specialized namespace provides specialized and strongly typed collection classes, such as string-only collections and linked-list and hybrid dictionaries.

Implementing a Collection of Key/Value Pairs

The Dictionary<TKey,TValue> generic collection enables you to access to elements in a collection by using the key of each element. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is fast because the `Dictionary` class is implemented as a hash table.

The following example creates a `Dictionary` collection and iterates through the dictionary by using a `foreach` statement.

```
private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

To instead use a collection initializer to build the `Dictionary` collection, you can replace the `BuildDictionary` and `AddToDictionary` methods with the following method.

```
private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}
```

The following example uses the ContainsKey method and the Item[TKey] property of `Dictionary` to quickly find an

item by key. The `Item` property enables you to access an item in the `elements` collection by using the `elements[symbol]` in C#.

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("found: " + theElement.Name);
    }
}
```

The following example instead uses the TryGetValue method quickly find an item by key.

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

Using LINQ to Access a Collection

LINQ (Language-Integrated Query) can be used to access collections. LINQ queries provide filtering, ordering, and grouping capabilities. For more information, see Getting Started with LINQ in C#.

The following example runs a LINQ query against a generic `List`. The LINQ query returns a different collection that contains the results.

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    //  Calcium 20
    //  Potassium 19
    //  Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

Sorting a Collection

The following example illustrates a procedure for sorting a collection. The example sorts instances of the `Car` class that are stored in a List<T>. The `Car` class implements the IComparable<T> interface, which requires that the CompareTo method be implemented.

Each call to the CompareTo method makes a single comparison that is used for sorting. User-written code in the `CompareTo` method returns a value for each comparison of the current object with another object. The value returned is less than zero if the current object is less than the other object, greater than zero if the current object is greater than the other object, and zero if they are equal. This enables you to define in code the criteria for greater than, less than, and equal.

In the `ListCars` method, the `cars.Sort()` statement sorts the list. This call to the Sort method of the List<T> causes the `CompareTo` method to be called automatically for the `Car` objects in the `List`.

```
private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
```

```
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.Write(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    //  blue  50 car4
    //  blue  30 car5
    //  blue  20 car1
    //  green 50 car7
    //  green 10 car3
    //  red   60 car6
    //  red   50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}
```

Defining a Custom Collection

You can define a collection by implementing the IEnumerable<T> or IEnumerable interface.

Although you can define a custom collection, it is usually better to instead use the collections that are included in the .NET Framework, which are described in Kinds of Collections earlier in this topic.

The following example defines a custom collection class named `AllColors`. This class implements the IEnumerable interface, which requires that the GetEnumerator method be implemented.

The `GetEnumerator` method returns an instance of the `ColorEnumerator` class. `ColorEnumerator` implements the IEnumerator interface, which requires that the Current property, MoveNext method, and Reset method be implemented.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
```

```
    }
}

// Element class.
public class Color
{
    public string Name { get; set; }
}
```

Iterators

An *iterator* is used to perform a custom iteration over a collection. An iterator can be a method or a `get` accessor. An iterator uses a yield return statement to return each element of the collection one at a time.

You call an iterator by using a foreach statement. Each iteration of the `foreach` loop calls the iterator. When a `yield return` statement is reached in the iterator, an expression is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

For more information, see Iterators (C#).

The following example uses an iterator method. The iterator method has a `yield return` statement that is inside a for loop. In the `ListEvenNumbers` method, each iteration of the `foreach` statement body creates a call to the iterator method, which proceeds to the next `yield return` statement.

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

See also

- Object and Collection Initializers
- Programming Concepts (C#)
- Option Strict Statement
- LINQ to Objects (C#)
- Parallel LINQ (PLINQ)
- Collections and Data Structures
- Selecting a Collection Class
- Comparisons and Sorts Within Collections

- When to Use Generic Collections

Covariance and Contravariance (C#)

5/4/2018 • 3 minutes to read • Edit Online

In C#, covariance and contravariance enable implicit reference conversion for array types, delegate types, and generic type arguments. Covariance preserves assignment compatibility and contravariance reverses it.

The following code demonstrates the difference between assignment compatibility, covariance, and contravariance.

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

Covariance for arrays enables implicit conversion of an array of a more derived type to an array of a less derived type. But this operation is not type safe, as shown in the following code example.

```
object[] array = new String[10];
// The following statement produces a run-time exception.
// array[0] = 10;
```

Covariance and contravariance support for method groups allows for matching method signatures with delegate types. This enables you to assign to delegates not only methods that have matching signatures, but also methods that return more derived types (covariance) or that accept parameters that have less derived types (contravariance) than that specified by the delegate type. For more information, see Variance in Delegates (C#) and Using Variance in Delegates (C#).

The following code example shows covariance and contravariance support for method groups.

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

In .NET Framework 4 or newer C# supports covariance and contravariance in generic interfaces and delegates and allows for implicit conversion of generic type parameters. For more information, see Variance in Generic Interfaces (C#) and Variance in Delegates (C#).

The following code example shows implicit reference conversion for generic interfaces.

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

A generic interface or delegate is called *variant* if its generic parameters are declared covariant or contravariant. C# enables you to create your own variant interfaces and delegates. For more information, see Creating Variant Generic Interfaces (C#) and Variance in Delegates (C#).

Related Topics

TITLE	DESCRIPTION
Variance in Generic Interfaces (C#)	Discusses covariance and contravariance in generic interfaces and provides a list of variant generic interfaces in the .NET Framework.
Creating Variant Generic Interfaces (C#)	Shows how to create custom variant interfaces.
Using Variance in Interfaces for Generic Collections (C#)	Shows how covariance and contravariance support in the IEnumerable<T> and IComparable<T> interfaces can help you reuse code.
Variance in Delegates (C#)	Discusses covariance and contravariance in generic and non-generic delegates and provides a list of variant generic delegates in the .NET Framework.
Using Variance in Delegates (C#)	Shows how to use covariance and contravariance support in non-generic delegates to match method signatures with delegate types.
Using Variance for Func and Action Generic Delegates (C#)	Shows how covariance and contravariance support in the `Func` and `Action` delegates can help you reuse code.

Expression Trees (C#)

1/23/2019 • 4 minutes to read • Edit Online

Expression trees represent code in a tree-like data structure, where each node is an expression, for example, a method call or a binary operation such as `x < y`.

You can compile and run code represented by expression trees. This enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. For more information about expression trees in LINQ, see How to: Use Expression Trees to Build Dynamic Queries (C#).

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and the .NET Framework and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL). For more information about the DLR, see Dynamic Language Runtime Overview.

You can have the C# or Visual Basic compiler create an expression tree for you based on an anonymous lambda expression, or you can create expression trees manually by using the System.Linq.Expressions namespace.

Creating Expression Trees from Lambda Expressions

When a lambda expression is assigned to a variable of type Expression<TDelegate>, the compiler emits code to build an expression tree that represents the lambda expression.

The C# compiler can generate expression trees only from expression lambdas (or single-line lambdas). It cannot parse statement lambdas (or multi-line lambdas). For more information about lambda expressions in C#, see Lambda Expressions.

The following code examples demonstrate how to have the C# compiler create an expression tree that represents the lambda expression `num => num < 5`.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Creating Expression Trees by Using the API

To create expression trees by using the API, use the Expression class. This class contains static factory methods that create expression tree nodes of specific types, for example, ParameterExpression, which represents a variable or parameter, or MethodCallExpression, which represents a method call. ParameterExpression, MethodCallExpression, and the other expression-specific types are also defined in the System.Linq.Expressions namespace. These types derive from the abstract type Expression.

The following code example demonstrates how to create an expression tree that represents the lambda expression `num => num < 5` by using the API.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

In .NET Framework 4 or later, the expression trees API also supports assignments and control flow expressions such as loops, conditional blocks, and `try-catch` blocks. By using the API, you can create expression trees that are more complex than those that can be created from lambda expressions by the C# compiler. The following example demonstrates how to create an expression tree that calculates the factorial of a number.

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
    // Adding a loop.
        Expression.Loop(
    // Adding a conditional block into the loop.
           Expression.IfThenElse(
    // Condition: value > 1
               Expression.GreaterThan(value, Expression.Constant(1)),
    // If true: result *= value --
               Expression.MultiplyAssign(result,
                   Expression.PostDecrementAssign(value)),
    // If false, exit the loop and go to the label.
               Expression.Break(label, result)
           ),
    // Label to jump to.
       label
    )
);

// Compile and execute an expression tree.
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()(5);

Console.WriteLine(factorial);
// Prints 120.
```

For more information, see Generating Dynamic Methods with Expression Trees in Visual Studio 2010, which also applies to later versions of Visual Studio.

Parsing Expression Trees

The following code example demonstrates how the expression tree that represents the lambda expression

`num => num < 5` can be decomposed into its parts.

```
// Add the following using directive to your code file:
// using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
                  param.Name, left.Name, operation.NodeType, right.Value);

// This code produces the following output:

// Decomposed expression: num => num LessThan 5
```

Immutability of Expression Trees

Expression trees should be immutable. This means that if you want to modify an expression tree, you must construct a new expression tree by copying the existing one and replacing nodes in it. You can use an expression tree visitor to traverse the existing expression tree. For more information, see How to: Modify Expression Trees (C#).

Compiling Expression Trees

The Expression<TDelegate> type provides the Compile method that compiles the code represented by an expression tree into an executable delegate.

The following code example demonstrates how to compile an expression tree and run the resulting code.

```
// Creating an expression tree.
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));

// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

For more information, see How to: Execute Expression Trees (C#).

See also

- System.Linq.Expressions
- How to: Execute Expression Trees (C#)

- How to: Modify Expression Trees (C#)
- Lambda Expressions
- Dynamic Language Runtime Overview
- Programming Concepts (C#)

Iterators (C#)

5/15/2019 • 7 minutes to read • Edit Online

An *iterator* can be used to step through collections such as lists and arrays.

An iterator method or `get` accessor performs a custom iteration over a collection. An iterator method uses the yield return statement to return each element one at a time. When a `yield return` statement is reached, the current location in code is remembered. Execution is restarted from that location the next time the iterator function is called.

You consume an iterator from client code by using a foreach statement or by using a LINQ query.

In the following example, the first iteration of the `foreach` loop causes execution to proceed in the `SomeNumbers` iterator method until the first `yield return` statement is reached. This iteration returns a value of 3, and the current location in the iterator method is retained. On the next iteration of the loop, execution in the iterator method continues from where it left off, again stopping when it reaches a `yield return` statement. This iteration returns a value of 5, and the current location in the iterator method is again retained. The loop completes when the end of the iterator method is reached.

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

The return type of an iterator method or `get` accessor can be IEnumerable, IEnumerable<T>, IEnumerator, or IEnumerator<T>.

You can use a `yield break` statement to end the iteration.

> **NOTE**
> For all examples in this topic except the Simple Iterator example, include using directives for the `System.Collections` and `System.Collections.Generic` namespaces.

Simple Iterator

The following example has a single `yield return` statement that is inside a for loop. In `Main`, each iteration of the `foreach` statement body creates a call to the iterator function, which proceeds to the next `yield return` statement.

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

Creating a Collection Class

In the following example, the `DaysOfTheWeek` class implements the IEnumerable interface, which requires a GetEnumerator method. The compiler implicitly calls the `GetEnumerator` method, which returns an IEnumerator.

The `GetEnumerator` method returns each string one at a time by using the `yield return` statement.

```
static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}
```

The following example creates a `Zoo` class that contains a collection of animals.

The `foreach` statement that refers to the class instance (`theZoo`) implicitly calls the `GetEnumerator` method. The `foreach` statements that refer to the `Birds` and `Mammals` properties use the `AnimalsForType` named iterator method.

```
static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler

    foreach (string name in theZoo.Birds)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros

    Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods.
```

```
    // Private methods.
    private IEnumerable AnimalsForType(Animal.TypeEnum type)
    {
        foreach (Animal theAnimal in animals)
        {
            if (theAnimal.Type == type)
            {
                yield return theAnimal.Name;
            }
        }
    }

    // Private class.
    private class Animal
    {
        public enum TypeEnum { Bird, Mammal }

        public string Name { get; set; }
        public TypeEnum Type { get; set; }
    }
}
```

Using Iterators with a Generic List

In the following example, the Stack<T> generic class implements the IEnumerable<T> generic interface. The Push method assigns values to an array of type `T`. The GetEnumerator method returns the array values by using the `yield return` statement.

In addition to the generic GetEnumerator method, the non-generic GetEnumerator method must also be implemented. This is because IEnumerable<T> inherits from IEnumerable. The non-generic implementation defers to the generic implementation.

The example uses named iterators to support various ways of iterating through the same collection of data. These named iterators are the `TopToBottom` and `BottomToTop` properties, and the `TopN` method.

The `BottomToTop` property uses an iterator in a `get` accessor.

```
static void Main()
{
    Stack<int> theStack = new Stack<int>();

    //  Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }

    // Retrieve items from the stack.
    // foreach is allowed because theStack implements IEnumerable<int>.
    foreach (int number in theStack)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    // foreach is allowed, because theStack.TopToBottom returns IEnumerable(Of Integer).
    foreach (int number in theStack.TopToBottom)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3 2 1 0

    foreach (int number in theStack.BottomToTop)
```

```
    foreach (int number in theStack.BottomToTop)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 9 8 7 6 5 4 3

    Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()
    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public IEnumerable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumerable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumerable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;

        for (int index = top - 1; index >= startIndex; index--)
```

```
        for (int index = top - 1; index >= startIndex; index--)
        {
            yield return values[index];
        }
    }

}
```

Syntax Information

An iterator can occur as a method or `get` accessor. An iterator cannot occur in an event, instance constructor, static constructor, or static finalizer.

An implicit conversion must exist from the expression type in the `yield return` statement to the type argument for the IEnumerable<T> returned by the iterator.

In C#, an iterator method cannot have any `in`, `ref`, or `out` parameters.

In C#, "yield" is not a reserved word and has special meaning only when it is used before a `return` or `break` keyword.

Technical Implementation

Although you write an iterator as a method, the compiler translates it into a nested class that is, in effect, a state machine. This class keeps track of the position of the iterator as long the `foreach` loop in the client code continues.

To see what the compiler does, you can use the Ildasm.exe tool to view the Microsoft intermediate language code that's generated for an iterator method.

When you create an iterator for a class or struct, you don't have to implement the whole IEnumerator interface. When the compiler detects the iterator, it automatically generates the `Current`, `MoveNext`, and `Dispose` methods of the IEnumerator or IEnumerator<T> interface.

On each successive iteration of the `foreach` loop (or the direct call to `IEnumerator.MoveNext`), the next iterator code body resumes after the previous `yield return` statement. It then continues to the next `yield return` statement until the end of the iterator body is reached, or until a `yield break` statement is encountered.

Iterators don't support the IEnumerator.Reset method. To reiterate from the start, you must obtain a new iterator. Calling Reset on the iterator returned by an iterator method throws a NotSupportedException.

For additional information, see the C# Language Specification.

Use of Iterators

Iterators enable you to maintain the simplicity of a `foreach` loop when you need to use complex code to populate a list sequence. This can be useful when you want to do the following:

- Modify the list sequence after the first `foreach` loop iteration.
- Avoid fully loading a large list before the first iteration of a `foreach` loop. An example is a paged fetch to load a batch of table rows. Another example is the EnumerateFiles method, which implements iterators within the .NET Framework.
- Encapsulate building the list in the iterator. In the iterator method, you can build the list and then yield each result in a loop.

See also

- System.Collections.Generic
- IEnumerable<T>
- foreach, in
- yield
- Using foreach with Arrays
- Generics

Language Integrated Query (LINQ)

4/28/2019 • 3 minutes to read • Edit Online

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language. Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. With LINQ, a query is a first-class language construct, just like classes, methods, events.

For a developer who writes queries, the most visible "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative *query syntax*. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO .NET Datasets, XML documents and streams, and .NET collections.

The following example shows the complete query operation. The complete operation includes creating a data source, defining the query expression, and executing the query in a `foreach` statement.

```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

Query expression overview

- Query expressions can be used to query and to transform data from any LINQ-enabled data source. For example, a single query can retrieve data from a SQL database, and produce an XML stream as output.
- Query expressions are easy to master because they use many familiar C# language constructs.
- The variables in a query expression are all strongly typed, although in many cases you do not have to provide the type explicitly because the compiler can infer it. For more information, see Type relationships in LINQ query operations.
- A query is not executed until you iterate over the query variable, for example, in a `foreach` statement. For more information, see Introduction to LINQ queries.

- At compile time, query expressions are converted to Standard Query Operator method calls according to the rules set forth in the C# specification. Any query that can be expressed by using query syntax can also be expressed by using method syntax. However, in most cases query syntax is more readable and concise. For more information, see C# language specification and Standard query operators overview.
- As a rule when you write LINQ queries, we recommend that you use query syntax whenever possible and method syntax whenever necessary. There is no semantic or performance difference between the two different forms. Query expressions are often more readable than equivalent expressions written in method syntax.
- Some query operations, such as Count or Max, have no equivalent query expression clause and must therefore be expressed as a method call. Method syntax can be combined with query syntax in various ways. For more information, see Query syntax and method syntax in LINQ.
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. IEnumerable<T> queries are compiled to delegates. IQueryable and IQueryable<T> queries are compiled to expression trees. For more information, see Expression trees.

Next steps

To learn more details about LINQ, start by becoming familiar with some basic concepts in Query expression basics, and then read the documentation for the LINQ technology in which you are interested:

- XML documents: LINQ to XML
- ADO.NET Entity Framework: LINQ to entities
- .NET collections, files, strings and so on: LINQ to objects

To gain a deeper understanding of LINQ in general, see LINQ in C#.

To start working with LINQ in C#, see the tutorial Working with LINQ.

Object-Oriented Programming (C#)

5/18/2019 • 9 minutes to read • Edit Online

C# provides full support for object-oriented programming including encapsulation, inheritance, and polymorphism.

Encapsulation means that a group of related properties, methods, and other members are treated as a single unit or object.

Inheritance describes the ability to create new classes based on an existing class.

Polymorphism means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.

This section describes the following concepts:

Classes and Objects

The terms *class* and *object* are sometimes used interchangeably, but in fact, classes describe the *type* of objects, while objects are usable *instances* of classes. So, the act of creating an object is called *instantiation*. Using the blueprint analogy, a class is a blueprint, and an object is a building made from that blueprint.

To define a class:

```
class SampleClass
{
}
```

C# also provides a light version of classes called *structures* that are useful when you need to create large array of objects and do not want to consume too much memory for that.

To define a structure:

```
struct SampleStruct
{
}
```

For more information, see:

- class
- struct

Class Members

Each class can have different *class members* that include properties that describe class data, methods that define class behavior, and events that provide communication between different classes and objects.

Properties and Fields

Fields and properties represent information that an object contains. Fields are like variables because they can be read or set directly.

To define a field:

```
class SampleClass
{
    public string sampleField;
}
```

Properties have get and set procedures, which provide more control on how values are set or returned.

C# allows you either to create a private field for storing the property value or use so-called auto-implemented properties that create this field automatically behind the scenes and provide the basic logic for the property procedures.

To define an auto-implemented property:

```
class SampleClass
{
    public int SampleProperty { get; set; }
}
```

If you need to perform some additional operations for reading and writing the property value, define a field for storing the property value and provide the basic logic for storing and retrieving it:

```
class SampleClass
{
    private int _sample;
    public int Sample
    {
        // Return the value stored in a field.
        get { return _sample; }
        // Store the value in the field.
        set { _sample = value; }
    }
}
```

Most properties have methods or procedures to both set and get the property value. However, you can create read-only or write-only properties to restrict them from being modified or read. In C#, you can omit the `get` or `set` property method. However, auto-implemented properties cannot be read-only or write-only.

For more information, see:

- get
- set

Methods

A *method* is an action that an object can perform.

To define a method of a class:

```
class SampleClass
{
    public int sampleMethod(string sampleParam)
    {
        // Insert code here
    }
}
```

A class can have several implementations, or *overloads*, of the same method that differ in the number of parameters or parameter types.

To overload a method:

```
public int sampleMethod(string sampleParam) {};
public int sampleMethod(int sampleParam) {}
```

In most cases you declare a method within a class definition. However, C# also supports *extension methods* that allow you to add methods to an existing class outside the actual definition of the class.

For more information, see:

- Methods
- Extension Methods

Constructors

Constructors are class methods that are executed automatically when an object of a given type is created. Constructors usually initialize the data members of the new object. A constructor can run only once when a class is created. Furthermore, the code in the constructor always runs before any other code in a class. However, you can create multiple constructor overloads in the same way as for any other method.

To define a constructor for a class:

```
public class SampleClass
{
    public SampleClass()
    {
        // Add code here
    }
}
```

For more information, see:

Constructors.

Finalizers

Finalizers are used to destruct instances of classes. In the .NET Framework, the garbage collector automatically manages the allocation and release of memory for the managed objects in your application. However, you may still need finalizers to clean up any unmanaged resources that your application creates. There can be only one finalizers for a class.

For more information about finalizers and garbage collection in the .NET Framework, see Garbage Collection.

Events

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or raises) the event is called the *publisher* and the classes that receive (or handle) the event are called *subscribers*. For more information about events, how they are raised and handled, see Events.

- To declare an event in a class, use the event keyword.
- To raise an event, invoke the event delegate.
- To subscribe to an event, use the `+=` operator; to unsubscribe from an event, use the `-=` operator.

Nested Classes

A class defined within another class is called *nested*. By default, the nested class is private.

```
class Container
{
    class Nested
    {
        // Add code here.
    }
}
```

To create an instance of the nested class, use the name of the container class followed by the dot and then followed by the name of the nested class:

```
Container.Nested nestedInstance = new Container.Nested()
```

Access Modifiers and Access Levels

All classes and class members can specify what access level they provide to other classes by using *access modifiers*.

The following access modifiers are available:

C# MODIFIER	DEFINITION
public	The type or member can be accessed by any other code in the same assembly or another assembly that references it.

C# MODIFIER	DEFINITION
private	The type or member can only be accessed by code in the same class.
protected	The type or member can only be accessed by code in the same class or in a derived class.
internal	The type or member can be accessed by any code in the same assembly, but not from another assembly.
protected internal	The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.
private protected	The type or member can be accessed by code in the same class or in a derived class within the base class assembly.

For more information, see Access Modifiers.

Instantiating Classes

To create an object, you need to instantiate a class, or create a class instance.

```
SampleClass sampleObject = new SampleClass();
```

After instantiating a class, you can assign values to the instance's properties and fields and invoke class methods.

```
// Set a property value.
sampleObject.sampleProperty = "Sample String";
// Call a method.
sampleObject.sampleMethod();
```

To assign values to properties during the class instantiation process, use object initializers:

```
// Set a property value.
SampleClass sampleObject = new SampleClass
    { FirstProperty = "A", SecondProperty = "B" };
```

For more information, see:

- new Operator
- Object and Collection Initializers

Static Classes and Members

A static member of the class is a property, procedure, or field that is shared by all instances of a class.

To define a static member:

```
static class SampleClass
{
    public static string SampleString = "Sample String";
}
```

To access the static member, use the name of the class without creating an object of this class:

```
Console.WriteLine(SampleClass.SampleString);
```

Static classes in C# have static members only and cannot be instantiated. Static members also cannot access non-static properties, fields or methods

For more information, see: static.

Anonymous Types

Anonymous types enable you to create objects without writing a class definition for the data type. Instead, the compiler generates a class for you. The class has no usable name and contains the properties you specify in declaring the object.

To create an instance of an anonymous type:

```
// sampleObject is an instance of a simple anonymous type.
var sampleObject =
    new { FirstProperty = "A", SecondProperty = "B" };
```

For more information, see: Anonymous Types.

Inheritance

Inheritance enables you to create a new class that reuses, extends, and modifies the behavior that is defined in another class. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. However, all classes in C# implicitly inherit from the Object class that supports .NET class hierarchy and provides low-level services to all classes.

> **NOTE**
> C# doesn't support multiple inheritance. That is, you can specify only one base class for a derived class.

To inherit from a base class:

```
class DerivedClass:BaseClass {}
```

By default all classes can be inherited. However, you can specify whether a class must not be used as a base class, or create a class that can be used as a base class only.

To specify that a class cannot be used as a base class:

```
public sealed class A { }
```

To specify that a class can be used as a base class only and cannot be instantiated:

```
public abstract class B { }
```

For more information, see:

- sealed
- abstract

Overriding Members

By default, a derived class inherits all members from its base class. If you want to change the behavior of the inherited member, you need to override it. That is, you can define a new implementation of the method, property or event in the derived class.

The following modifiers are used to control how properties and methods are overridden:

C# MODIFIER	DEFINITION
virtual	Allows a class member to be overridden in a derived class.
override	Overrides a virtual (overridable) member defined in the base class.
abstract	Requires that a class member to be overridden in the derived class.
new Modifier	Hides a member inherited from a base class

Interfaces

Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes. An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.

To define an interface:

```
interface ISampleInterface
{
    void doSomething();
}
```

To implement an interface in a class:

```
class SampleClass : ISampleInterface
{
    void ISampleInterface.doSomething()
    {
        // Method implementation.
    }
}
```

For more information, see:

Interfaces

interface

Generics

Classes, structures, interfaces and methods in the .NET Framework can include *type parameters* that define types of objects that they can store or use. The most common example of generics is a collection, where you can specify the type of objects to be stored in a collection.

To define a generic class:

```
public class SampleGeneric<T>
{
    public T Field;
}
```

To create an instance of a generic class:

```
SampleGeneric<string> sampleObject = new SampleGeneric<string>();
sampleObject.Field = "Sample string";
```

For more information, see:

- Generics
- Generics

Delegates

A *delegate* is a type that defines a method signature, and can provide a reference to any method with a compatible signature. You can invoke (or call) the method through the delegate. Delegates are used to pass methods as arguments to other methods.

> **NOTE**
> Event handlers are nothing more than methods that are invoked through delegates. For more information about using delegates in event handling, see Events.

To create a delegate:

```
public delegate void SampleDelegate(string str);
```

To create a reference to a method that matches the signature specified by the delegate:

```
class SampleClass
{
    // Method that matches the SampleDelegate signature.
    public static void sampleMethod(string message)
    {
        // Add code here.
    }
    // Method that instantiates the delegate.
    void SampleDelegate()
    {
        SampleDelegate sd = sampleMethod;
        sd("Sample string");
    }
}
```

For more information, see:

- Delegates
- delegate

See also

- C# Programming Guide

Reflection (C#)

4/28/2019 • 2 minutes to read • Edit Online

Reflection provides objects (of type Type) that describe assemblies, modules and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them. For more information, see Attributes.

Here's a simple example of reflection using the static method `GetType` - inherited by all types from the `Object` base class - to obtain the type of a variable:

```
// Using GetType to obtain type information:
int i = 42;
System.Type type = i.GetType();
System.Console.WriteLine(type);
```

The output is:

```
System.Int32
```

The following example uses reflection to obtain the full name of the loaded assembly.

```
// Using Reflection to get information of an Assembly:
System.Reflection.Assembly info = typeof(System.Int32).Assembly;
System.Console.WriteLine(info);
```

The output is:

```
mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

NOTE

The C# keywords `protected` and `internal` have no meaning in IL and are not used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. To identify an `internal` method using reflection, use the IsAssembly property. To identify a `protected internal` method, use the IsFamilyOrAssembly.

Reflection Overview

Reflection is useful in the following situations:

- When you have to access attributes in your program's metadata. For more information, see Retrieving Information Stored in Attributes.
- For examining and instantiating types in an assembly.
- For building new types at runtime. Use classes in System.Reflection.Emit.
- For performing late binding, accessing methods on types created at run time. See the topic Dynamically Loading and Using Types.

Related Sections

For more information:

- Reflection
- Viewing Type Information
- Reflection and Generic Types
- System.Reflection.Emit
- Retrieving Information Stored in Attributes

See also

- C# Programming Guide
- Assemblies in the Common Language Runtime

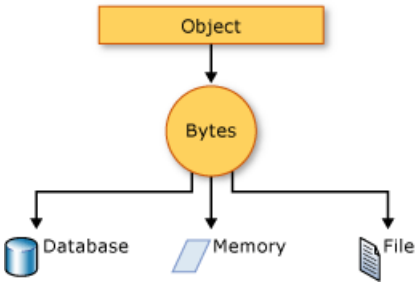
Serialization (C#)

3/25/2019 • 3 minutes to read • Edit Online

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

How serialization works

This illustration shows the overall process of serialization:



The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory.

Uses for serialization

Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

Making an object serializable

To serialize an object, you need the object to be serialized, a stream to contain the serialized object, and a Formatter. System.Runtime.Serialization contains the classes necessary for serializing and deserializing objects.

Apply the SerializableAttribute attribute to a type to indicate that instances of this type can be serialized. An exception is thrown if you attempt to serialize but the type doesn't have the SerializableAttribute attribute.

If you don't want a field within your class to be serializable, apply the NonSerializedAttribute attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked SerializableAttribute, those objects will also be serialized.

Binary and XML serialization

You can use binary or XML serialization. In binary serialization, all members, even members that are read-only, are serialized, and performance is enhanced. XML serialization provides more readable code, and greater flexibility of object sharing and usage for interoperability purposes.

Binary serialization

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-

based network streams.

XML serialization

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML. System.Xml.Serialization contains the classes necessary for serializing and deserializing XML.

You apply attributes to classes and class members to control the way the XmlSerializer serializes or deserializes an instance of the class.

Basic and custom serialization

Serialization can be performed in two ways, basic and custom. Basic serialization uses the .NET Framework to automatically serialize the object.

Basic serialization

The only requirement in basic serialization is that the object has the SerializableAttribute attribute applied. The NonSerializedAttribute can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems. You would use custom serialization when versioning issues are important. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

Custom serialization

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked SerializableAttribute and implement the ISerializable interface.

If you want your object to be deserialized in a custom manner as well, you must use a custom constructor.

Designer serialization

Designer serialization is a special form of serialization that involves the kind of object persistence associated with development tools. Designer serialization is the process of converting an object graph into a source file that can later be used to recover the object graph. A source file can contain code, markup, or even SQL table information.

Related Topics and Examples

Walkthrough: Persisting an Object in Visual Studio (C#)
Demonstrates how serialization can be used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object is instantiated.

How to: Read Object Data from an XML File (C#)
Shows how to read object data that was previously written to an XML file using the XmlSerializer class.

How to: Write Object Data to an XML File (C#)
Shows how to write the object from a class to an XML file using the XmlSerializer class.