Chapter: 12

**C# Only : LINQ, LEs, Pre-processors, RTTI**

LINQ, Query keywords & clauses, Lambda Expressions (LEs),Query Methods, Preprocessor, RTTI, Nullable Types, Pointers , unsafe, fixed, Conversion Operators , Collections, Other Keywords

**C#\_12.1 LINQ Intro**

A program might obtain information from a customer list, look up product information in a catalog, or access an employee’s record. In many cases, such data is stored in a database that is *separate from the application*. For example, a product catalog might be stored in a relational database. In the past, interacting with such a database would involve generating queries using SQL (Structured Query Language). Other sources of data, such as XML, required their own approaches. Therefore, prior to C# 3.0, support for such queries was *not built into C#*. LINQ changes this.

* LINQ: LINQ stands for language-integrated query. It encompasses a *set of features* that lets you retrieve information from a data source. LINQ adds to C# the ability to generate queries for any LINQ-compatible data source. LINQ gives you a ***uniform way to access data*** because:
* The syntax used for the query is the same, no matter what data source is used. This means that the syntax used to query data in a relational database is the same as that used to query data stored in an array, for example.
* It is *no longer necessary* to use SQL or any other non-C# mechanism. The query capability is fully integrated into the C# language.
* In addition to using LINQ with SQL, LINQ can be used with XML files and ADO.NET datasets. Perhaps equally important, it can also be used with C# arrays and collections *(described later in this chapter)*.
* LINQ is supported by a set of interrelated features, including the *query* *syntax* added to the C#, *lambda expressions*, *anonymous types*, and *extension methods*.

**C#\_12.2 QUERY**

At the core of LINQ is the query. A query specifies ***what data will be obtained from a data source***. For example, a query on an inventory database might request a list of ***out-of-stock items***. A query on a log of Internet usage could ask for a ***list of the websites with the highest hit counts***. Although these queries differ in their specifics, all can be expressed using the same LINQ syntactic elements.

* After a query has been created, it can be executed. One way this is done is by using the query in a foreach loop. Executing a query causes its results to be obtained. Thus, using a query involves two key steps.

|  |  |
| --- | --- |
| 1. First, the form of the query is created. | 1. Second, the query is executed. |

* Therefore, the query defines what to retrieve from a data source. ***Executing the query*** actually obtains the results.
* IEnumerable Interface: In order for a source of data to be used by LINQ, it must implement the IEnumerable interface. There are *generic* and *non-generic* forms.
* Generic form: In general, it is easier if the data source implements the generic version, ***IEnumerable<T>***, where ***T*** specifies the type of data being enumerated. This interface is declared in **System.Collections.Generic**.
* Arrays and IEnumerable<T>: A class that implements ***IEnumerable<T>*** supports enumeration, which means that its contents can be obtained one at a time in sequence. All C# arrays support ***IEnumerable<T>***. Thus, we can use *arrays to demonstrate* *the central concepts of LINQ*. However, ***LINQ*** *is not limited to* ***arrays***.
* C#\_Example 1 (A Simple Query): Before going into any more theory, let’s work through a simple LINQ example. The following program uses a query to obtain the positive values contained in an array of integers:

**using System; using System.Linq;**

**class** SimpQuery { **static void Main()** { **int[]** nums = { 1, -2, 3, 0, -4, 5 };

**var** posNums = **from** n **in** nums **where** n > 0 **select** n; *// Create a query that obtains only positive numbers.*

**Console.WriteLine**("The positive values in nums:");

**foreach**(**int** i **in** posNums) **Console.WriteLine**(i); */\* Execute the query and display the results. \*/* }}

* ***using System.Linq;*** To use the ***LINQ*** features, you must include the ***System.Linq*** namespace.
* An ***array*** of int called nums is declared. All ***arrays in C# are implicitly convertible*** to IEnumerable<T>. This makes ***any C# array usable*** as a *LINQ* *data source*.

|  |  |
| --- | --- |
| * Next, a query is declared that retrieves those elements in nums that are positive: | ***var posNums = from n in nums***  ***where n > 0***  ***select n;*** |

* The *variable* posNums is called the query variable. It *refers* *to* the *set of rules* defined by the *query*.
* Notice that it uses var to ***implicitly declare*** posNums. As you know, this makes posNums an ***implicitly typed variable***. In queries, it is often convenient to use *implicitly typed variables*, although you can also explicitly declare the type, which must be *some form* of IEnumerable<T>. The variable posNums is then ***assigned*** the query expression.
* **form** , **where** and **group** clauses:
* from: All queries begin with ***from***. This clause specifies two items:

1. The *first* is the range variable, which will ***receive elements*** obtained from the data source. In this case, the range variable is ***n***.
2. The *second* item is the data source, which, in this case, is the ***nums*** *array*. The type of the range variable is inferred from the data source. In this case, the type of ***n*** is ***int***.

* Generalized syntax of the ***from*** clause: ***from range-variable in data-source***
* where: The next clause in the query is ***where***. It specifies a condition that an element in the data source must meet in order to be obtained by the query. Its general form is: ***where boolean-expression***
* The ***boolean-expression*** must produce a ***bool*** result. (This *expression* is also called a *predicate*.) There can be more than one ***where*** clause in a query. A ***where*** clause acts as a filter on the data source, allowing only *certain items through*.
* In the program, this ***where*** clause is used: ***where n > 0*** It will be true only for an element whose value is greater than zero. This expression will be evaluated for every ***n*** in ***nums*** when the query executes. Only those values that satisfy *this condition* will be *obtained*.
* select and group : All queries end with either a select clause or a group clause. In this example, the select clause is used.
* ***select*** specifies precisely what is obtained by the query. For *simple queries*, such as in this example, the range value is selected. Therefore, it returns those *integers* from nums that satisfy the where *clause*. In more sophisticated situations, it is possible to finely tune what is selected. For example, when querying a ***mailing list***, you might ***return*** just the ***last name of each recipient***, rather than the ***entire address***.
* Notice that the ***select*** clause ends with a semicolon. Because *select* ends a query, it *ends the statement* and requires a *semicolon*. Notice, however, that the other clauses in the query ***do not end*** with a semicolon.
* Executing the query: At this point, a query variable called ***posNums*** has been created, but *no results have been obtained*. A *query simply defines a set of rules*. The *results are obtained* when the *query is executed*. Simply declaring the ***query*** posNums *does not mean that it contains the results of the query*.
* Use foreach to execute the query: To execute the query, the program uses the ***foreach*** loop shown here:

**foreach**(**int** i **in** posNums) **Console.WriteLine**(i);

* Notice that ***posNums*** is specified as the collection being iterated over.
* When the ***foreach*** executes, the rules defined by the query specified by ***posNums*** are executed. With *each pass through the loop*, the *next element returned* by the query is obtained. The process ends when there are *no more elements to retrieve*.
* In this case, the type of the iteration variable ***i*** is explicitly specified as ***int*** because this is the *type of the elements retrieved by the query*, here it is easy to know the ***type of the value selected by the query***. However, it will be easier (in some cases) to *implicitly specify* the type of the iteration variable by using ***var***.
* The same query can be executed two or more times, with the *possibility of differing results* if the underlying ***data source changes between executions***.

**C#\_12.3 Executing a Query Multiple times**

Because a query defines a set of rules that is used to retrieve data, *but does not, itself, produce results*, the *same query* can be *run multiple times*. *If the data source changes between runs, then the results of the query may differ*. Therefore, ***once you define a query***, ***executing it will*** *always produce the* *most current results*.

* Each execution of a ***query*** produces its own results, which are obtained by enumerating the ***current contents*** of the data source. Therefore, *if the data source changes, so, too, might the results of executing a query*.
* The benefits of this approach are quite significant. For example, if you are obtaining a *list of pending orders* for an *online store*, then you want each execution of your query to produce all orders, including those just entered.
* C#\_Example 2(Query Multiple times): In the *following* version of the *preceding program*, the contents of the ***nums*** array are changed between two executions of ***posNums***:

|  |
| --- |
| using System; using System.Linq; using System.Collections.Generic;  class SimpQuery { static void Main() { int[] nums = { 1, -2, 3, 0, -4, 5 };  var posNums = from n in nums where n > 0 select n; *// Create a query that obtains only positive numbers.*  */\* Execute the query and display the results.\*/* Console.WriteLine("The positive values in nums:"); foreach(int i in posNums) Console.WriteLine(i);  */\* Change nums. \*/* Console.WriteLine("\nSetting nums[1] to 99."); nums[1] = 99;  */\* Execute the query a after change \*/* Console.WriteLine("The positive values in nums:"); foreach(int i in posNums) Console.WriteLine(i); }} |

* After the value in ***nums[1]*** was changed from ***–2*** to ***99***, the result of *re-running* the *query* reflects the *change*.

**C#\_12.4 Relation between types in a QUERY**

A query *involves variables* whose *types* *relate to one another*. These are the *query variable*, the *range variable*, and the *data source*. The type of the range variable is dependent upon the type of the data source.

* In many cases, C# can infer the type of the range variable. As long as the data source implements ***IEnumerable<T>***, the type inference can be made, because ***T*** describes the ***type of the elements*** in the *data source*. (As mentioned, all ***arrays*** implement ***IEnumerable<T>***, as do many other data sources.)
* However, if the data source implements the non-generic version of ***IEnumerable***, then you will need to explicitly specify the type of the range variable. This is *done by specifying* its type in the ***from*** clause. For example, assuming the preceding examples, this shows how to explicitly declare ***n*** to be an ***int***:

***var*** *posNums* ***= from int*** *n* ***in*** *nums* ***// ...***

* Of course, the explicit type specification is *not needed* here, because all arrays are implicitly convertible to ***IEnumerable<T>***, which enables the type of the range variable to be inferred.
* Type of object returned by a query: The *type* of *object* returned by a *query* is an instance of ***IEnumerable<T>***, where ***T*** is the type of the elements. Thus, the *type* of the *query variable* must be an *instance* of ***IEnumerable<T>***.
* The value of T is determined by the type of the value specified by the select clause. In the case of the preceding example, ***T*** is ***int*** because ***n*** is an ***int***. (As explained, **n** is an *int* because *int* is the ***type of elements*** stored in *nums*.)
* C#\_Example 2's query using explicit type: The query could have been written like this, with the *type* *explicitly specified* as ***IEnumerable <int>***:

**IEnumerable<int>** posNums = **from** n **in** nums **where** n > 0 **select** n;

[ Which was " var posNums = from n in nums where n > 0 select n;" ]

* The key point is that the type of the item selected by select must agree with the type argument passed to ***IEnumerable<T>*** used to declare the query variable.
* Implicit variable recommended: Often, query variables use ***var*** rather than explicitly specifying the type because this *lets the compiler infer the proper type from* the *select* *clause*. This approach is ***particularly useful*** when select returns *something* other than an *element from* the data source.
* Type inside the foreach lop: When a query is executed by the foreach loop, the type of the iteration variable must be the same as the type of the range variable. In the preceding examples, this type was explicitly specified as int.
* Implicit variable recommended: You can let the compiler *infer* the type by specifying this variable as ***var***. As you will see, there are also some cases in which **var** *must be used* because the type name of the data is unknown.

**C#\_12.5 Query: Details (with clauses and keyword)**

All queries share a general form, which is based on a set of *contextual keywords*, shown here:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| contextual keywords for QUERIES | | | | | | | | | | | |
| ***ascending*** | ***descending*** | ***equals*** | ***from*** | ***group*** | ***in*** | ***Into*** | ***join*** | ***let*** | ***on*** | ***orderby*** | ***select*** |

* Of above keywords, the following begin query clauses:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Use following to begin a QUERY CLAUSE | | | | | | |
| ***from*** | ***group*** | ***join*** | ***let*** | ***orderby*** | ***select*** | ***Where*** |

|  |  |
| --- | --- |
| * A query must begin with from and end with either a select or group. * The select clause determines what type of value is enumerated by the query. | * The group clause returns the data by groups, with each group able to be enumerated individually. * where specifies *criteria* that an item must meet in order for it to be *returned*. |

**C#\_12.5.1 WHERE (Filter Values)**

*where* is used to *filter the data* returned by a *query*. The preceding examples have shown only its simplest form, in which a *single condition* is used. However, you can use where to *filter data* based on more than one condition. One way to do this is through the use of multiple where clauses. For example, consider the following program that displays only those values in the ***array*** that are both ***positive*** and ***less than 10***. int[] **nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };**

var **posNums =** from **n** in **nums** where **n > 0** where **n < 10** select **n;** *// a query that obtains positive values less than 10:*

Console.WriteLine**("The positive values less than 10:");** foreach**(**int **i** in **posNums)** Console.WriteLine**(i);** *//Execute the query*

|  |  |
| --- | --- |
| * Combining conditions in a single where: In the above example we can use a single where in which both tests are combined into a single expression. Here is the query rewritten to use this approach: | **var** posNums = **from** n **in** nums  **where** n > 0 **&&** n < 10  **select** n; |

* In general, a where condition can use any valid C# expression that evaluates to a ***Boolean result***.

**C#\_12.5.2 ORDERBY (Sort Results)**

Often you will want the results of a query to be sorted. LINQ gives you an easy way to produce sorted results: the ***orderby*** clause. The general form of ***orderby*** is:

***orderby sort-on how***

* The item on which to *sort* is specified by ***sort-on***. This can be *as inclusive as the entire element* stored in the data source or *as restricted as a portion of a single field* within the element.
* Value of ***how*** determines if the sort is ascending or descending, and it must be either ***ascending*** or ***descending***. The default direction is ascending.
* C#\_Example 3: **using System**; **using System.Linq**;

**class** OrderbyDemo { **static void Main()** { **int[]** nums = { 10, -19, 4, 7, 2, -5, 0 };

**var** sNums = **from** n **in** nums **orderby** n **select** n; */\* Create a query that obtains the values in sorted order.\*/* }}

* To change the order to *descending*, simply specify the ***descending*** option, as shown here:

**var** sNums = **from** n **in** nums **orderby** n **descending** **select** n;

**C#\_12.5.3 SELECT**

The select clause determines what types of elements are obtained by a query. Its general form is: ***select expression***

* Projecting using select: so far, we have been using ***select*** to return the ***range variable***. Thus, *expression* has simply named the *range variable*.
* However, *select* is not limited to this *simple action*. *It can return a specific portion* of the range variable, the result of applying some operation or transformation to the range variable, or even a new type of object that is ***constructed from pieces of the information obtained from the*** range variable. This is called projecting.
* Following displays the ***square roots*** of the ***positive*** values contained in an array of ***double*** values.

**double[]** nums = { -10.0, 16.4, 12.125, 100.85, -2.2, 25.25, -3.5 } ;

**var** sqrRoots = **from** n **in** nums **where** n > 0 **select Math.Sqrt**(n); *// Create a query that returns the square roots of the positive values in nums.*

* Pay special attention to the select clause: ***select Math.Sqrt(n);***

It returns the square root of the range variable. It does this by obtaining the result of *passing* the range variable to ***Math.Sqrt()***, which returns the square root of its argument.

* You can use ***select*** to generate any type of sequence you need, based on the values obtained from the data source.
* C#\_Example 4: Here is a program that shows another way to use select. It creates a class called EmailAddress that contains two properties. The first holds a *person’s name*. The second contains an *e-mail address*. The program then creates an array that contains several EmailAddress entries. The program uses a query to ***obtain a list*** of just the ***e-mail addresses*** by themselves.

|  |  |
| --- | --- |
| using System; using System.Linq;  class EmailAddress {  public string Name { get; set; }  public string Address { get; set; }  public EmailAddress(string n, string a) { Name = n;  Address = a; } } | class SelectDemo2 { static void Main() {  EmailAddress[] addrs = { new EmailAddress("gg", "gg@hh.com"),  new EmailAddress("tt", "tt@yoyo.com"),  new EmailAddress("tata", "tata@baba.com") };  var eAddrs = from entry in addrs select entry.Address; *// Create a query that selects e-mail addresses.*  Console.WriteLine("The e-mail addresses :"); foreach(string s in eAddrs) Console.WriteLine(" " + s); }} |

* Pay special attention to the select clause: ***select entry.Address;*** Instead of returning the entire range variable, it returns only the Address portion.
* This means that the query returns a sequence of strings, not a *sequence* of EmailAddress objects. This is why the foreach loop *specifies* s as a string. As explained, the type of sequence *returned* by a query is *determined* by the type of value *returned* by the select clause.
* Collect specified data and insert it into another object: One of the more powerful features of select is its ability to return a sequence that contains elements created during the execution of the query.
* C#\_Example 5: Consider the following program. It defines a *class* called ***ContactInfo***, which stores a *name*, *e-mail* address, and *telephone* number.
* It also defines the ***EmailAddress*** class used by the preceding example.
* Inside Main(), an array of ContactInfo is created. Then, a query is declared in which the *data source* is an array of ContactInfo, but the *sequence returned* contains EmailAddress objects. Thus, the *type* of the sequence returned by select isn't ContactInfo, but rather EmailAddress, and ***these*** *objects* ***are created during the*** *execution* ***of the*** *query*.

|  |  |
| --- | --- |
| using System; using System.Linq;  class ContactInfo { public string Name { get; set; }  public string Email { get; set; }  public string Phone { get; set; }  public ContactInfo(string n, string a, string p) {  Name = n; Email = a; Phone = p; }  }  class EmailAddress {  public string Name { get; set; }  public string Address { get; set; }  public EmailAddress(string n, string a) { Name = n; Address = a; } } | class SelectDemo3 { static void Main() {  ContactInfo[] contacts = { new ContactInfo("Herb", "lolo@uu.com", "555-1010"),  new ContactInfo("Tom", "Tom@hoho.com", "555-1101"),  new ContactInfo("Sara", "Sara@hoho.com", "555-0110") };  *// Create a query that creates a list of EmailAddress objects.*  var emailList = from entry in contacts  select new EmailAddress(entry.Name, entry.Email);  Console.WriteLine("The e-mail list is");  foreach(EmailAddress e in emailList) *// Execute the query and display the results.*  Console.WriteLine(" {0}: {1}", e.Name, e.Address );  }} |

* The key point of this example is that the ***type*** of sequence generated by a ***query*** can consist of objects created by the ***query***.

**C#\_12.5.4 GROUP (Group Results)**

group clause enables you to create results that are grouped by keys. Using the *sequence* obtained from a group, you can easily *access all of the data associated* with a key. This makes group an easy and effective way to retrieve data that is organized into ***sequences of related items***.

* The ***group*** clause is *one of only two clauses* that can end a query. (The other is select.) The ***group*** clause has the following general form:

***group range-variable by key***

* It returns ***data grouped into sequences***, with each sequence sharing the key specified by ***key***.
* The result of ***group*** is a sequence that *contains elements* of type ***IGrouping<TKey, TElement>***, which is declared in the ***System.Linq*** *namespace*. It *defines a collection* of *objects* that share a common key.
* The *type* of *query variable* in a ***query*** that returns a group is ***IEnumerable<IGrouping<TKey, TElement>>***. ***IGrouping*** defines a read-only property called ***Key***, which returns the key associated with each sequence.
* C#\_Example 6: Following declares an array that contains a list of websites. It creates a query that *groups the list* by *top-level domain name*, such as .org, .com, .tv.

|  |  |  |
| --- | --- | --- |
| using System; using System.Linq;  class GroupDemo { static void Main() {  string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",  "hsNameD.com", "hsNameE.org", "hsNameF.org",  "hsNameG.tv", "hsNameH.net", "hsNameI.tv" };  *// Create a query that groups websites by top-level domain name.*  var webAddrs = from addr in websites  where addr.*LastIndexOf*(".") != -1  group addr by addr.*Substring*(addr.*LastIndexOf*("."));  *// Execute the query and display the results.*  foreach(var sites in webAddrs) { Console.WriteLine("Websites grouped by " + sites.Key);  foreach(var site in sites) Console.WriteLine(" " + site);  Console.WriteLine(); } }} | | Output: **Websites grouped by .com**  **hsNameA.com**  **hsNameD.com**  **Websites grouped by .net**  **hsNameB.net**  **hsNameC.net**  **hsNameH.net**  **Websites grouped by .org**  **hsNameE.org**  **hsNameF.org**  **Websites grouped by .tv**  **hsNameG.tv**  **hsNameI.tv** |
| * The ***key*** is obtained by use of the ***LastIndexOf()*** and ***Substring()*** methods defined by string. *[These are described in* C#\_2.8 strings*. The version of* ***Substring()*** *used here returns the substring that starts at the specified index and runs to the end of the invoking string.]* | * Notice how this is achieved by the *group* *clause*:   **var** webAddrs = **from** addr **in** websites  **where** addr.**LastIndexOf**(".") != -1  **group** addr **by** addr.**Substring**(addr.**LastIndexOf**(".")); | |

* The index of the last period in a website name is found using ***LastIndexOf()***. Using this index, the ***Substring()*** method obtains the *remainder of the string*, which is the *part* of the *website name* that contains the *top-level domain name*.
* Notice the use of the ***where*** clause to ***filter out any strings that don’t contain a period*** "**.**". The ***LastIndex()*** method returns ***–1*** if the specified string is not contained in the invoking string.
* Because the *sequence* obtained when *webAddrs* is executed is a *list of groups*, you will need to use two ***foreach*** loops to access the *members* of *each group*. The outer loop obtains each group. The inner loop ***enumerates*** the *members* within the *group*.
* The iteration variable of the ***outer foreach loop*** must be an ***IGrouping*** instance compatible with the ***key*** and *element type*. In the example, both the ***keys*** and elements are ***string***. i.e, the type of sites *iteration variable* of the outer loop is ***IGrouping<string, string>***.

|  |  |
| --- | --- |
| * The type of the iteration variable of the inner loop is string. For brevity, the example implicitly declares these variables, but they could have been explicitly declared as shown here: | **foreach**(**IGrouping<string, string>** sites **in** webAddrs) {  **Console.WriteLine**("Websites grouped by " + sites.**Key**);  **foreach**(string site in sites) **Console.WriteLine**(" " + site);  **Console.WriteLine**(); } |

**C#\_12.5.5 INTO (Create a Continuation)**

* QUERY CONTINUATION using into with group/select: When using ***select*** or ***group***, you will sometimes want to *generate a temporary result* that will be used by a subsequent part of the query to produce the final result. This is called a query continuation (or just a continuation for short), and it is accomplished through the use of ***into*** with a ***select*** or ***group*** clause. It has this general form: ***into name query-body***
* where ***name*** is the name of the range variable that iterates over the temporary result and is used by the *continuing query*, specified by *query-body*. This is why ***into*** is called a *query continuation* when used with ***select*** or ***group***—*it continues the query*.
* A query continuation *embodies the concept* of building a new query that queries the results of the preceding query.
* NOTE: There is also a form of ***into*** that can be used with ***join***, which creates a ***group join***. This is described later.
* C#\_Example 7: Following uses into with group. Reworks the GroupDemo example shown earlier, which creates a *list of websites grouped by top-level domain name*.
* In this case, the initial results are queried by a range variable called ***ws***. This result is then filtered to remove all groups that have *fewer* than *three elements*.

|  |  |
| --- | --- |
| using System; using System.Linq;  class IntoDemo { static void Main() {  string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",  "hsNameD.com", "hsNameE.org", "hsNameF.org",  "hsNameG.tv", "hsNameH.net", "hsNameI.tv" };  */\* Create a query that groups websites by top-level domain name, but select only those groups that have more than two members. \*/*  var webAddrs = from addr in websites  where addr.LastIndexOf(".") != -1  group addr by addr.Substring(addr.LastIndexOf("."))  into ws *//Put* ***temporary*** *results into* ***ws****.*  where ws.Count() > 2  select ws;  */\* ws is the range variable over the set of groups returned when the first half of the query is executed.\*/* | *// Execute the query and display the results.*  Console.WriteLine("Top-level domains with more than 2 members.\n");  foreach(var sites in webAddrs) {  Console.WriteLine("Contents of " + sites.Key + " domain:");  foreach(var site in sites) Console.WriteLine(" " + site);  Console.WriteLine(); }  }} |
| Output:  **Top-level domains with more than 2 members.**  **Contents of .net domain:**  **hsNameB.net**  **hsNameC.net**  **hsNameH.net** |

* As the output shows, only the .net group is returned because it is the *only group* that has *more than two elements*.

|  |  |
| --- | --- |
| * First, the *results* of the ***group*** *clause* are *stored* (creating a *temporary* *result*) and a *new query begins*, which *operates* on the *stored results*. * The range variable of the ***new query*** is ***ws***. At this point, ***ws*** will range over each group returned by the first query. (It ***ranges over groups*** because the first query results in a sequence of groups.) | * In the program, pay special attention to this sequence of clauses in the query:   **group** addr **by** addr.**Substring**(addr.**LastIndexOf**(".", addr.**Length**))  **into** ws  **where** ws.**Count**() > 2  **select** ws; |

* Next, the ***where*** clause filters the query so that the *final result contains only those groups* that contain *more than two members*. This determination is made by calling ***Count()***, which is an extension method that is *implemented* for all ***IEnumerable<T>*** objects. It *returns* the *number of elements* in a sequence. (extension methods discussed in this chapter.) The *resulting sequence of groups* is returned by the ***select*** clause.

**C#\_12.5.6 LET (Create a Variable in a Query)**

In a query, you will sometimes want to *temporarily retain a value*. For example, you might want to create an *enumerable variable* that can, itself, be *queried*. Or, you might want to *store* a value that will be *used later* on in a ***where*** *clause*. Whatever the purpose, these types of actions can be accomplished through the use of ***let***. The ***let*** clause has this general form: ***let name = expression***

* Here, ***name*** is an *identifier that is assigned the value* of ***expression***. The *type* of ***name*** is *inferred* from the *type* of the ***expression***.
* C#\_Example 8(*Use a let clause and a nested from clause*): Here is an example that shows how ***let*** can be used to *create another enumerable data source*.
* The *query* takes as input an *array of strings*. It then converts those *strings* into *char arrays*. This is accomplished by use of another *string method* called ***ToCharArray()***, which returns an array *containing the characters* in the string [Recall C#\_2.8 strings ].
* The result is *assigned to a variable* called ***chrArray***, which is then used by another ***from*** clause to obtain the *individual characters* in the *array*. The *query* then *sorts* the *characters* and returns the *resulting* *sequence*.

|  |  |
| --- | --- |
| using System; using System.Linq;  class LetDemo { static void Main() { string[] strs = { "alpha", "beta", "gamma" };  */\* Create a query that obtains the characters in the strings, returned in sorted order. Notice the use of a nested from clause.\*/*  var chrs = from str in strs  let chrArray = str.ToCharArray() */\* chrArray refers to an array of* ***char*** *obtained from str. \*/*  from ch in chrArray orderby ch  select ch; | Console.WriteLine("Individual characters in sorted order:");  foreach(char c in chrs) Console.Write(c + " "); }}  output:  **The individual characters in sorted order:**  **a a a a a b e g h l m m p t** |

* Notice how ***let*** assigns to chrArraya reference to the array returned by ***str.ToCharArray()***: **let** chrArray = str.**ToCharArray()**
* After the ***let*** clause, other clauses can make use of ***chrArray***. Furthermore, because all arrays in C# implement ***IEnumerable<T>***, ***chrArray*** can be used as a data source for a second, nested ***from*** clause. This is what happens in the example. It uses the ***nested from*** to enumerate the individual characters in the array, sorting them into ascending sequence and ***returning the result***.

|  |  |
| --- | --- |
| * Hold a non-enumerable value using let: You can also use a ***let*** clause to hold a non-enumerable value. For example, a more efficient way to write the query used in the IntoDemo program shown in the preceding section C#\_12.5.6. * Here, the index of the last occurrence of a period is assigned to ***idx***. This value is then used by ***Substring()***. This *prevents* the *search* for the *period* from having to be *conducted twice*. | var webAddrs = from addr in websites  */\* Call LastIndexOf( ) only once, storing the result in idx.\*/*  let idx = addr.LastIndexOf(".")  where idx != -1  group addr by addr.Substring(idx) into ws  where ws.Count() > 2  select ws; |

**C#\_12.5.7 JOIN (Join Two Sequences)**

When working with databases, it is common to want to create a sequence that correlates data from two different data sources. Such an action is easy to accomplish in LINQ through the use of the ***join*** clause. The general form of ***join*** is shown here (in context with the ***from*** clause):

**from** *range-varA* **in** *data\_sourceA*

**join** *range-varB* **in** *data\_sourceB*

**on** *range-varA.property* **equals** *range-varB.property*

* The key to using ***join*** is to understand that each data source must *contain data in common* and ***that data*** can be compared for equality. Thus, in the general form, ***data\_sourceA*** and ***data\_sourceB*** *must have something in common* that can be compared.
* The items being compared are specified by the ***on*** *section*. Thus, when ***range-varA.property*** is equal to ***range-varB.property***, the correlation succeeds. In essence, ***join*** acts like a filter, allowing *only those elements* that ***share a common value*** to *pass through*.
* When using ***join***, often the *sequence returned* is a *composite of portions* of the two data sources. Thus, ***join*** lets you generate a new list that contains elements from *two different data sources*. This enables you to *organize data* in a new way.
* C#\_Example 9: The following program creates a class called ***Item***, which encapsulates an item’s name with its number. It creates another class called ***InStockStatus***, which links an *item number* with a ***Boolean*** property that indicates *whether or not* the item is in stock. It also creates a class called Temp, which has two fields, one ***string*** and one ***bool***. Objects of this class will ***hold the result*** of the query. The query uses ***join*** to produce a *list* in which an *item’s* *name* is associated with its *in-stock* *status*.

|  |  |
| --- | --- |
| using System; using System.Linq;  *// A class that links an item name with its number.*  class Item { public string Name { get; set; }  public int ItemNumber { get; set; }  public Item(string n, int inum) { Name = n; ItemNumber = inum; } }  *// A class that links an item number with its in-stock status.*  class InStockStatus { public int ItemNumber { get; set; }  public bool InStock { get; set; }  public InStockStatus(int n, bool b) { ItemNumber = n; InStock = b; } } | *// A class that encapsulates a name with its status.*  class Temp {public string Name { get; set; }  public bool InStock { get; set; }  public Temp(string n, bool b) { Name = n; InStock = b; } }  class JoinDemo { static void Main() {  Item[] items = { new Item("Pliers", 1424), new Item("Hammer", 7892),  new Item("Wrench", 8534), new Item("Saw", 6411) };  InStockStatus[] statusList = {  new InStockStatus(1424, true), new InStockStatus(7892, false),  new InStockStatus(8534, true), new InStockStatus(6411, true) }; |
| */\* Create a query that joins Item with InStockStatus to produce a list of item names and availability. Notice that a sequence of Temp objects is produced. \*/*  var inStockList = from item in items  join entry in statusList on item.ItemNumber equals entry.ItemNumber */\* Join two lists based on ItemNumber.\*/*  select new Temp(item.Name, entry.InStock); *//* *Return a Temp object that contains the result of the join*.    Console.WriteLine("Item\tAvailable\n");  foreach(Temp t in inStockList) Console.WriteLine("{0}\t{1}", t.Name, t.InStock); }} | |

* To understand how ***join*** works, let’s walk through each line in the query. The ***query*** begins in the normal fashion with this ***from*** clause:

**var** inStockList = **from** item **in** items

* This clause specifies that ***item*** is the range variable for the data source specified by ***items***. Inside ***Main()*** the ***items*** array contains objects of *type* Item, which ***encapsulate*** a ***name*** and a ***number*** for an ***inventory item***.
* Next comes the ***join*** clause: **join** entry **in** statusList **on** item.ItemNumber **equals** entry.ItemNumber
* It specifies that ***entry*** is the range variable for the ***statusList*** data source. The ***statusList*** array contains ***objects of type*** InStockStatus, which ***link an item*** number ***with its status***.
* Thus, ***items*** and ***statusList*** have a property in common: the item number. This is used by the ***on/equals*** portion of the ***join*** clause to describe the correlation. Thus, ***join*** matches items from the two data sources when their *item numbers* are *equal*.
* Finally, the ***select*** clause returns a ***Temp*** object that contains an ***item*** ’s name along with its *in-stock status*:

***select new*** *Temp(item.Name, entry.InStock);* Therefore, the sequence obtained by the *query* consists of ***Temp*** *objects*.

* Although the preceding example is fairly straightforward, ***join*** supports substantially more sophisticated operations. For example, you can use ***into*** with ***join*** to create a ***group join***, which creates a result that consists of an element from the first sequence and a group of all matching elements from the second sequence. (You’ll see an example of this a bit later in this chapter.) In general, the time and effort needed to fully master join are well worth the investment because it gives you the ability to reorganize data at runtime. This is a powerful capability. It is made even more powerful by the use of anonymous types, described in the next section.

**C#\_12.6 Anonymous Types and Object Initializers**

An anonymous type is a class that has no name. Its primary use is to create an object returned by the *select* clause.

* Often, the outcome of a query is a sequence of objects that either is a composite of two (or more) data sources (such as in the case of join) or includes a subset of the members of one data source. In either case, the ***type being returned is often needed only because of the query and is not used elsewhere in the program***. In this case, using an anonymous type eliminates the *need to declare a class that will be used* simply to hold the outcome of the query.
* ANONYMOUS type and OBJECT INITIALIZATION without explicitly invoking a constructor: An *anonymous type* is created through the use of this general form: ***new { nameA = valueA, nameB = valueB, ... }***
* Here, the names specify identifiers that translate into read-only properties, which are ***initialized by the values***. For example,

***new { Count = 10, Max = 100, Min = 0 }***

* This creates a *class type* that has three *public read-only properties*: ***Count***, ***Max***, and ***Min***. These are given the values ***10***, ***100***, and ***0***, respectively. These properties can be referred to by name by other code. This syntax is called OBJECT INITIALIZATION.
* OBJECT INITIALIZATION provides a way to initialize an object ***without explicitly invoking a*** constructor. This is necessary in the case of *anonymous types* because there is no way to *explicitly call a constructor*. (CONSTRUCTORS have the same name as their class. In the case of an anonymous class, there is no name.)
* Because an anonymous type has *no name*, you *must use* an *implicitly typed variable* to refer to it. This lets the ***compiler*** infer the ***proper type***. For example,

***var myOb = new { Count = 10, Max = 100, Min = 0 }***

* creates a variable called ***myOb*** that is assigned a *reference* to the *object* created by the *anonymous type expression*. This means that these statements are legal: ***Console.WriteLine("Count is " + myOb.Count); if(i <= myOb.Max && i >= myOb.Min) // ...***
* Remember, when an *anonymous type* is created, the *identifiers* that you specify become *read-only public properties*. i.e they can be used by other parts of code.
* Although the term anonymous type is used, it’s not quite completely true! The type is anonymous relative to you, the *programmer*. However, the compiler does give it an internal name. Thus, anonymous types do not violate *C#’s strong type-checking rules*.
* C#\_Example 10: To fully understand the value of ***anonymous types***, consider this rewrite of the previous program that demonstrated ***join***. Recall that in the previous version, a ***class*** called ***Temp*** was needed to encapsulate the result of the ***join***. Through the use of an *anonymous type*, this “*placeholder*” *class* is *no longer needed and no longer clutters* the source code to the program.

|  |  |
| --- | --- |
| using System; using System.Linq;  *// A class that links an item name with its number.*  class Item { */\* same as C#\_Example 9 \*/* }  *// A class that links an item number with its in-stock status.*  class InStockStatus { */\* same as C#\_Example 9 \*/* }  class AnonTypeDemo { static void Main() {  Item[] items = { */\* same as C#\_Example 9 \*/* };  InStockStatus[] statusList = { */\* same as C#\_Example 9 \*/* }; | */\* Create a query that joins Item with InStockStatus to produce a list of item names and availability. \*/*  */\* Now, an anonymous type is used. \*/*  var inStockList = from item in items  join entry in statusList on item.ItemNumber equals entry.ItemNumber  select new { Name = item.Name, InStock = entry.InStock }; */\* Return an anonymous type \*/*  Console.WriteLine("Item\tAvailable\n");  foreach(var t in inStockList) Console.WriteLine("{0}\t{1}", t.Name, t.InStock); }} |

* Pay special attention to the ***select*** clause: **select** **new** { Name = item.Name, InStock = entry.InStock };
* It returns an *object* of an anonymous type that has two read-only properties, ***Name*** and ***InStock***. These are given the values specified by the item’s name and availability. Because of the anonymous type, there is *no longer any need* for the ***Temp class***.
* Notice the ***foreach loop***. It now uses ***var*** to declare the iteration variable. This is necessary because the type of the object contained in ***inStockList*** has *no name*. This situation is ***one of the reasons that C# 3.0 added*** implicitly typed variables. They are *needed to support* anonymous types.
* Simplified syntax using projection initializer: In some cases, including the one just shown, *you can simplify the syntax of the anonymous type* through the use of a projection initializer. In this case, you *simply specify* the name of the initializer by itself. This name automatically becomes the name of the property. For example, *here is another way to code the* select clause *used by the preceding program*:

***select new { item.Name, entry.InStock };***

* Here, the property names are still ***Name*** and ***InStock***, just as before. The compiler *automatically* “projects” the identifiers ***Name*** and ***InStock***, making them the property names of the anonymous type. Also, as before, the properties are given the values specified by ***item.Name*** and ***entry.InStock***.
* OBJECT INITIALIZATION syntax used by an ANONYMOUS type can be used for NAMED types: The object initialization syntax can also be used with named types. For example, given this class: ***class MyClass { public int Alpha { get; set; }***

***public int Beta { get; set; } }***

This declaration is legal: **var** myOb = **new** MyClass { Alpha = 10, Beta = 20 };

After this statement executes, the line **Console.WriteLine("Alpha: {0}, Beta {1}", myOb.Alpha, myOb.Beta);** displays ***Alpha: 10, Beta 20***

* Although OBJECT INITIALIZERS can be used with NAMED classes, their primary use is with ANONYMOUS types. Therefore, normally, you should explicitly call a constructor when working with named classes.

**C#\_12.7 GROUP JOIN**

You can use into with join to create a group join, which creates a sequence in which each entry in the result consists of an entry from the first sequence and a group of all matching elements from the second sequence.

* C#\_Example 11: The following example uses a group join to create a list in which various transports, such as cars, boats, and planes, are organized by their general transportation category, which is land, sea, or air.
* The program first creates a class called Transport that ***links a transport type*** with its classification. Inside ***Main()***, it creates two input sequences. The first is an array of strings that contains the names of the general means by which one travels: land, sea, and air.
* The second is an array of Transport that encapsulates *various means of transportation*. It then uses a group join to produce a list of transports that are organized by their category.

|  |  |
| --- | --- |
| using System; using System.Linq;  */\* This class links the name of a transport, such as Train, with its general classification, such as land, sea, or air. \*/*  class Transport { public string Name { get; set; }  public string How { get; set; }  public Transport(string n, string h) { Name = n; How = h; } }  class GroupJoinDemo { static void Main() {  string[] travelTypes = { "Air", "Sea", "Land", }; *// An array of transport classifications.*  *// An array of transports.*  Transport[] transports = { new Transport("Bicycle", "Land"), new Transport("Balloon", "Air"),  new Transport("Boat", "Sea"), new Transport("Jet", "Air"),  new Transport("Canoe", "Sea"), new Transport("Biplane", "Air"),  new Transport("Car", "Lnd"), new Transport("Cargo Ship", "Sea"),  new Transport("Train", "Land") }; | */\* Create a query that uses a group join to produce a list of item names and IDs organized by category. \*/*  var byHow = from how in travelTypes  join trans in transports on how equals trans.How  into lst  select new { How = how, Tlist = lst };  *// Execute the query and display the results.*  foreach(var t in byHow) {  Console.WriteLine("{0} transportation includes:", t.How);  foreach(var m in t.Tlist) Console.WriteLine(" " + m.Name);  Console.WriteLine(); } }} |

* from uses how to ***range over*** the travelTypes array. Recall that travelTypes contains an array of the general travel classifications, air, land, and sea. The join clause joins each travel type with those transports *that use that type*. Eg: Land is joined with Bicycle, Car, and Train. However, because of the into clause, for each travel type, the join produces a ***list of the transports*** *that use that type*. This list is represented by lst. Finally, select returns an anonymous type that ***encapsulates*** each value of how (the travel type) with a *list of transports*. This is why two foreach loops are needed to display the results of the query. The outer loop obtains an object that contains the ***name of the travel type*** and a ***list of the transports for that type***. The inner loop displays the ***individual transports***.

**C#\_12.8 Query Methods and Lambda Expressions (LEs)**

The query syntax described in the preceding sections is the way you will probably write most queries in C#. It is convenient, powerful, and compact. It is, however, not the only way to write a query.

* The other way is to use the query methods. These methods can be called on any enumerable object, such as an array. Many of the query methods require the use of *LE*. Because the query methods and *LEs* are intertwined.
* Why Two ways of creating queries in C# - The query syntax and the query methods: Actually, aside from the syntax involved, it really only has one way (i.e. *both approaches leads to same place*). Why? Because the query syntax is *compiled* into calls to the query methods! Thus, when you write something like: **where** x < 10 the compiler translates it into **Where**(x **=>** x < 10) (using the LE and Query method).
* Thus, the two approaches to creating a query ultimately lead to the same place.
* Which approach should be used in a C# program: In general, you will want to use the query syntax. It is cleaner and is fully integrated into the C#.

**C#\_12.8.1 Basic Query Methods**

The query methods are defined by System.Linq.Enumerable and are implemented as extension methods that extend the functionality of ***IEnumerable<T>***.

* An extension method adds functionality to ***another class***, but without the use of inheritance. QUERY METHODS can be called only on an OBJECT that implements ***IEnumerable<T>***.
* QUERY METHODS are also defined by ***System.Linq.Queryable***, which extends the functionality of ***IQueryable<T>***, but this interface is not used here.
* Some query methods: The ***Enumerable*** class provides many QUERY METHODS, but at the *core are those that correspond* to the QUERY KEYWORDS described earlier. These methods are shown here, along with the (equivalent) keywords to which they relate. Understand that these methods have OVERLOADED forms and only their simplest form is shown. However, this is also the form that you will usually use.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Query Keyword | **select** | **where** | **order** | **join** | **group** |
| Equivalent Query Method | ***Select(arg)*** | ***Where(arg)*** | ***OrderBy(arg) or OrderByDescending(arg)*** | ***Join(seq2, key1, key2, result)*** | ***GroupBy(arg)*** |

* Except for ***Join()***, the other methods take one argument, ***arg***, which is an object of type ***Func<T, TResult>***, as a parameter. This is a delegate type defined by LINQ. It is declared like this: ***delegate TResult Func<T, TResult>(T arg)***
* Here, ***TResult*** specifies the result of the delegate and ***T*** specifies the parameter type.
* In the ***query methods***, ***arg*** determines what action the ***query method*** takes. Eg: in the case of ***Where()***, ***arg*** determines *how the* ***query*** filters the data.
* Each of these *query methods* returns an *ENUMERABLE object*. Thus, the result of one can be used to execute a call on another, allowing the ***methods to be chained*** ***together***.
* The ***Join()*** method takes four arguments. The first is a REFERENCE to the second sequence to be joined. The ***first sequence*** is the one on which ***Join()*** is called. The ***key selector for the first sequence*** is passed via ***key1***, and the ***key selector for the second sequence*** is passed via ***key2***. The result of the ***join*** is described by result.
* The type of ***key1*** is ***Func<TOuter, TKey>***, and the type of ***key2*** is ***Func<TInner, TKey>***. The ***result*** argument is of type ***Func<TOuter, TInner, TResult>***.
* Here, ***TOuter*** is the element type of the ***invoking sequence***, ***TInner*** is the element type of the passed sequence, and ***TResult*** is the type of the resulting elements. An ENUMERABLE object is returned that contains the result of the ***join***.

**C#\_12.8.2 Lambda Expressions Introduction (LE)**

Although an argument to a query method such as ***Where()*** must be of type ***Func<T, TResult>***, it *does not need to be* an explicitly declared method. Instead, you will usually use a LE. LE offers a streamlined, yet powerful way to define what is, essentially, an *anonymous method*. The *C# compiler* automatically converts a LE ***into a form*** that can be passed to a ***Func<T, TResult>*** parameter.

* All ***LE*** use the lambda operator "***=>***". This operator divides a LE into two parts. On the left is specified the input parameter (or parameters). On the right is one of two things: an expression or a statement block. If the right side is an expression, then an EXPRESSION LAMBDA is being created. If the right side is a block of statements, then it is a STATEMENT LAMBDA.
* In an expression lambda, the expression on the right side of the **=>** acts on the parameter (or parameters) specified by the left side. The *result* of the *expression* becomes the result of the LAMBDA OPERATOR. Here is the *general form* of a *lambda expression* that takes only *one parameter*:

**param => expr**

* When more than one parameter is required, then this form is used: **(param-list) => expr**
* i.e. when two or more parameters are needed, they ***must be enclosed*** by parentheses. If no parameters are needed, then empty parentheses must be used.
* Here is a simple LE : **n => n > 0** For any **n**, this expression determines if ***n*** is greater than zero and returns the result.
* Here is another example: **count => count + 2** In this case, the result is the value of count ***increased*** by two.

**C#\_12.8.3 Create Queries by Using the Query Methods**

Using the QUERY METHODS in conjunction with LE, it is possible to create queries that do not use the C# query syntax. Instead, the QUERY METHODS are called. Let’s begin with a simple example. It reworks the first program in this chapter so that it uses calls to ***Where()*** and ***Select()*** rather than the QUERY KEYWORDS.

* C#\_Example 12: Following demonstrate Example of QUERY METHODS ***Where()*** and ***Select()***

**using System; using System.Linq;**

**class** SimpQuery { **static void Main()** { **int[]** nums = { 1, -2, 3, 0, -4, 5 };

**var** posNums = nums.**Where**(n **=>** n > 0) . **Select**(r **=>** r); *// Use Where() and Select() to create a simple query.*

**Console.WriteLine**("*The positive values in nums*:"); **foreach**(**int** i **in** posNums) **Console.WriteLine**(i); }}

* In the program, pay special attention to this line: ***var posNums = nums.Where(n => n > 0).Select(r => r);***
* This creates a query called ***posNums*** that creates a ***sequence of the positive values*** in ***nums***. It does this by use of the ***Where()*** method to filter the values and ***Select()*** to select the values. The ***Where()*** method can be invoked on ***nums*** because all ARRAYS implement ***IEnumerable<T>***, which supports the QUERY EXTENSION METHODS.
* Technically, ***Select()*** in the *preceding example is not necessary*, because in this simple case, the sequence returned by ***Where()*** already contains the result. However, you can use more sophisticated selection criteria, just as you did with the query syntax. For example, this query returns the ***positive values*** in ***nums*** ***increased by an order of magnitude***:

**var** posNums = nums**.Where**(n **=>** n>0)**.Select**(r **=>** r\*10);

* As you might expect, you can chain together other operations. For example, this query selects the positive values, sorts them into descending order, and *returns the resulting sequence*:

**var** posNums = nums.**Where**(n **=>** n > 0).**OrderByDescending**(j **=>** j);

* Here, ***j => j*** specifies that the *ordering* is *dependent on the INPUT PARAMETER*, which is an element from the sequence obtained from ***Where()***.
* C#\_Example 13: Following demonstrates the ***GroupBy()*** method. It reworks the GROUP EXAMPLE shown earlier.

|  |  |
| --- | --- |
| using System; using System.Linq;  class GroupByDemo { static void Main() {  string[] websites = { "hsNameA.com", "hsNameB.net", "hsNameC.net",  "hsNameD.com", "hsNameE.org", "hsNameF.org",  "hsNameG.tv", "hsNameH.net", "hsNameI.tv" }; | *// Use query methods to group websites by top-level domain name.*  var webAddrs = websites.Where(w => w.LastIndexOf(".") != 1).  GroupBy(x => x.Substring(x.LastIndexOf(".", x.Length)));  foreach(var sites in webAddrs) { Console.WriteLine("Websites grouped by " + sites.Key);  foreach(var site in sites) Console.WriteLine(" " + site);  Console.WriteLine(); } }} |

* This version produces the same output as before. The only difference is how the query is created. In this version, the QUERY METHODS are used.
* C#\_Example 14: Here is *another example*. Recall the join query used in the ***JoinDemo*** example in C#\_Example 9 shown earlier:

|  |  |
| --- | --- |
| Query KEYWORDS Used | Query METHODS Used |
| **var** inStockList = **from** item **in** items  **join** entry **in** statusList  **on** item.ItemNumber **equals** entry.ItemNumber  **select** **new** Temp(item.Name, entry.InStock);   * This query produces a sequence that contains objects that encapsulate the name and the in-stock status of an inventory item. This information is synthesized from joining the two lists called items and statusList. | The following version reworks this query so that it uses the Join( ) method rather than the C# query syntax:  *// Use Join() to produce a list of item names and status.*  ***var*** inStockList =  items.***Join***(statusList,  k1 **=>** k1.ItemNumber, k2 **=>** k2.ItemNumber,  (k1, k2) **=>** ***new*** Temp(k1.Name, k2.InStock) ); |

* We can use an anonymous type *could have been used instead of* the named class called ***Temp*** to hold the resulting object. This approach is shown next:

**var** inStockList = items.**Join**(statusList, k1 **=>** k1.ItemNumber, k2 **=>** k2.ItemNumber, (k1, k2) **=>** **new** {k1.Name, k2.InStock} );

**C#\_12.8.4 EXPRESSION TREE in LEs**

An expression tree is a representation of a LE as data. Thus, an expression tree, itself, cannot be executed. It can be CONVERTED into an EXECUTABLE form. EXPRESSION TREES are encapsulated by the ***System.Linq.Expressions.Expression<T>*** class. EXPRESSION TREES are useful in situations in which a QUERY will be *executed* by something *outside the program*, such as a DATABASE that uses SQL.

* By representing the Query as Data, the query can be converted into a FORMAT understood by the DATABASE. This process is used by the LINQ-to-SQL feature provided by *Visual C#*, for example. Thus, *expression trees* help *C#* support a *variety of data sources*.
* You can obtain an EXECUTABLE form of an EXPRESSION TREE by calling the ***Compile()*** method defined by ***Expression***. It returns a reference that *can be assigned* to a *DELEGATE* and then *EXECUTED*.
* RESTRICTION: Only EXPRESSION LAMBDAS can be represented by EXPRESSION TREES. They *cannot be used* to represent *STATEMENT LAMBDAS*.

**C#\_12.8.5 More Query-Related EXTENSION Methods**

In addition to the methods that correspond to the QUERY KEYWORDS, there are several other QUERY-related methods defined for ***IEnumerable<T>*** by ***Enumerable***. Here is a sampling of several commonly used methods. Because many of the methods are *overloaded*, only their *general form* is shown

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Method | | Description | Method | | Description |
| ***All(condition)*** | | Returns true if all elements in a sequence *satisfy a specified condition*. | ***First()*** | | Returns the first *element* in a *sequence*. |
| ***Any(condition)*** | | Returns true if any element in a sequence *satisfies a specified condition*. | ***Last()*** | | Returns the last *element* in a *sequence*. |
| ***Average()*** | | Returns the average of the values in a *numeric sequence*. | ***Max()*** | | Returns the maximum value in a *sequence*. |
| ***Contains(obj)*** | | Returns true if the sequence *contains the specified object*. | ***Min()*** | | Returns the minimum value in a *sequence*. |
| ***Count()*** | Returns the length of a sequence. This is the *number of elements that it contains*. | | | ***Sum()*** | Returns the summation of the values in a *numeric sequence*. |

* You have already seen ***Count()*** in action earlier in this chapter. Here is a program that demonstrates the others:

|  |  |
| --- | --- |
| * Use a query method with the query syntax: You can also use these extension methods within a query based on the C# query syntax. For example, this program uses ***Average()*** to obtain a *sequence* that contains only those values that are *less than the average* of the values in an array.   using System; using System.Linq;  class ExtMethods2 { static void Main() { int[] nums = { 1, 2, 4, 8, 6, 9, 10, 3, 6, 7 };  var ltAvg = from n in nums  let x = nums.Average() *//query method with the query syntax*  where n < x select n;  Console.WriteLine("The average is " + nums.Average() );  Console.WriteLine("These values are less than the average:");  foreach(int i in ltAvg) Console.WriteLine(i); }} | using System; using System.Linq;  class ExtMethods { static void Main() { int[] nums = { 3, 1, 2, 5, 4 };  Console.WriteLine("The minimum value is " + nums.Min());  Console.WriteLine("The maximum value is " + nums.Max());  Console.WriteLine("The first value is " + nums.First());  Console.WriteLine("The last value is " + nums.Last());  Console.WriteLine("The sum is " + nums.Sum());  Console.WriteLine("The average is " + nums.Average());  if(nums.All(n => n > 0)) Console.WriteLine("All values are greater than zero.");  if(nums.Any(n => (n % 2) == 0)) Console.WriteLine("At least one value is even.");  if(nums.Contains(3)) Console.WriteLine("The array contains 3."); }} |

**C#\_12.8.6 Deferred vs. Immediate Query Execution**

In LINQ, queries have two different modes of execution: IMMEDIATE and DEFERRED.

* DEFERRED EXECUTION: In general, a query defines a set of rules that are not actually executed until a ***foreach*** ***executes***. It is called DEFERRED EXECUTION.
* IMMEDIATE EXECUTION: If you use one of the EXTENSION METHODS that produce a non-sequence result, then the ***query must be executed to obtain that result***. For example, consider the ***Count()*** method. In order for ***Count()*** to return the number of elements in the sequence, the query must be executed, and *this is done automatically* when ***Count()*** is called. In this case, IMMEDIATE EXECUTION takes place, with the QUERY being EXECUTED AUTOMATICALLY *in order to obtain the result*. Therefore, even though you don’t explicitly use the query in a ***foreach*** loop, the ***query is still executed***.
* C#\_Example 15: Here is a simple example. It obtains the number of positive elements in the sequence. We used IMMEDIATE EXECUTION here,

**using** **System**; **using System.Linq**;

**class** ImmediateExec { **static void Main()** {

**int[]** nums = { 1, -2, 3, 0, -4, 5 };

**int** len = (**from** n **in** nums **where** n > 0 **select** n).**Count**();

**Console.WriteLine**("The number of positive values in nums: " + len); }}

* In the program, notice that no EXPLICIT ***foreach*** loop is specified. Instead, the query automatically executes because of the call to ***Count()***.

|  |  |
| --- | --- |
| * The QUERY in the preceding program *(written in one single line )* could also have been written like: | **var** posNums = **from** n **in** nums **where** n > 0 **select** n;  **int** len = posNums.**Count**(); *// query executes here* |

* In this case, ***Count()*** is called on the query variable. At that point, the *query* is *executed* to obtain the *count*.

**C#\_12.9 EXTENSION Methods: Details**

EXTENSION METHODS provide a means by which *functionality* can be added to a class *without using the normal inheritance* mechanism. Although you *won’t often create your own* *extension methods* (because the inheritance mechanism offers a better solution in many cases), it is still important that you understand how they work because of their integral importance to LINQ.

* An extension method is a static method that must be contained within a static, non-generic class. The *type* of its *first parameter* determines the *type* *of* *objects* on which the extension method can be called. Furthermore, the first parameter must be *modified* by this. The object *on which the method is invoked* is *passed automatically* to the *first parameter*. It is ***not explicitly passed*** in the argument list. A key point is that *even though an extension method is declared static*, it can still be ***called*** on an ***object***, just as if it were an ***instance method***.
* Here is the general form of an extension method: ***static ret-type name(this invoked-on-type ob, param-list)***
* Of course, if there are no arguments other than the one passed implicitly to ***ob***, then ***param-list*** will be empty.
* The first parameter is ***automatically passed*** the object *on which the method is invoked*.
* In general, an *extension method* will be a *public* *member* of its *class*.
* C#\_Example 16: Here is an example that creates *three* simple EXTENSION methods:

|  |  |
| --- | --- |
| **using System**;  **static** **class** MyExtMeths {  *// reciprocal of a double.*  **public static double** Reciprocal(**this** **double** v) { **return** 1.0 / v; }  *// Reverse the case of letters within a string*  **public static string** RevCase(**this** **string** str) { string temp = "";  **foreach**(**char** ch **in** str) {  **if**(Char.**IsLower**(ch)) temp += Char.**ToUpper**(ch);  **else** temp += Char.**ToLower**(ch); }  **return** temp; }  *// Return the absolute value of n / d.*  **public static double** AbsDivideBy(**this** **double** n, **double** d) {  **return** Math.Abs(n / d); } } | **class** ExtDemo { **static void Main()** { **double** val = 8.0;  **string** str = "Alpha Beta Gamma";  *// Call the Recip() extension method.*  **Console.WriteLine**("Reciprocal of {0} is {1}", val, val.Reciprocal());  *// Call the RevCase() extension method.*  **Console.WriteLine**(str + " after reversing case is " + str.RevCase());  *// Use AbsDivideBy();*  **Console.WriteLine**("Result of val.AbsDivideBy(-2): " + val.AbsDivideBy(-2));  }} |

* Notice that each EXTENSION METHOD is contained in a static class called ***MyExtMeths***. As explained, an extension method *must be declared within* a static class. Furthermore, this class must be in scope in order for the extension methods *that it contains* to be used. (This is why you need to include the System.Linq namespace when using LINQ.)
* Next, notice the calls to the EXTENSION METHODS. They are ***invoked on an object in just the same way*** that an INSTANCE METHOD is called. The main difference is that the *invoking object* is passed to the *first parameter* of the *extension method*. Therefore, when the expression: **val.AbsDivideBy(-2))** executes, val is passed to the n parameter of ***AbsDivideBy()*** and –2 is passed to the d parameter.

|  |  |
| --- | --- |
| * Because the methods ***Reciprocal()*** and ***AbsDivideBy()*** are defined for double, it is legal to invoke them on a double literal, as shown here: | ***8.0.Reciprocal()***  ***8.0.AbsDivideBy(-1)*** |

* Furthermore, ***RevCase()*** can be invoked like this: ***"AbCDe".RevCase()*** Here, the reversed-case version of a string literal is returned.

**C#\_12.10 Lambda Expressions: Details**

Although a principal use of LE is with LINQ, they are a feature that can be used with other aspects of C#. The reason is that a LE creates another type of anonymous function. (The other type of anonymous function is the anonymous method, described earlier in this book.) Thus, a LE can be *assigned to (or passed to)* a delegate. Because a lambda LE is usually *more streamlined* than the *equivalent* anonymous method, LEs are now the recommended approach in most cases.

* Expression Lambdas: The body of an expression lambda consists solely of the expression on the right side of the ***=>***. Thus, whatever action an expression lambda performs, it must take place within a single expression. EXPRESSION LAMBDAS are typically used with QUERIES, but they can be used whenever a DELEGATE *requires a method* that can be expressed in a *single expression*.
* LAMBDA with a DELEGATE: To use a lambda with a delegate involves two steps. First, you must declare the delegate type itself. Second, when you declare an instance of the delegate, assign to it the LE. Once this has been done, the LE can be executed by calling the delegate instance.
* C#\_Example 17: The following example illustrates the use of an expression lambda with a delegate. It declares two delegate types. It then assigns LEs to instances of those delegates. Finally, it executes the LEs through the delegate instances.

**using System;**

**delegate** **double** Transform(**double** v); *// Transform delegate takes one double argument and returns a double value.*

**delegate** **bool** TestInts(**int** w, **int** v); *// TestInts delegate takes two int arguments and returns a bool result.*

**class** ExpressionLambdaDemo { **static void Main()** {

**Transform** reciprocal = n => 1.0 / n; *// Assign an expression lambda to a delegate: Create a LE that returns the reciprocal of a value.*

**Console.WriteLine**("The reciprocal of 4 is " + reciprocal(4.0)); **Console.WriteLine**("The reciprocal of 10 is " + reciprocal(10.0));

**Console.WriteLine**();

**TestInts** isFactor = (n, d) => n % d == 0; *// Assign an expression lambda to a delegate: Create a LE that determines if one int is a factor of another.*

**Console.WriteLine**("Is 3 a factor of 9? " + isFactor(9, 3)); **Console.WriteLine**("Is 3 a factor of 10? " + isFactor(10, 3)); }}

* First notice how the delegates are declared. The Transform delegate takes a double argument and returns a double result. The TestInts delegate takes two int arguments and returns a bool result.

|  |  |
| --- | --- |
| * The first assigns to *reciprocal* a LE that returns the *reciprocal of the value that it is passed*. This expression can be assigned to a Transform delegate because it is compatible with Transform’s declaration. The argument used in the call to reciprocal is passed to n. The value returned is the result of the *expression* 1.0 / n. | * Next, pay special attention to these declarations:   ***Transform reciprocal = n => 1.0 / n;***  ***TestInts isFactor = (n, d) => n % d == 0;*** |

* The second statement assigns to isFactor an expression that returns true if the second argument is a factor of the first. This lambda takes two arguments, and it returns true if the first can be evenly divided by the second. Thus, it is compatible with the TestInts declaration. The two arguments passed to ***isFactor()*** when it is called are automatically passed to n and d, in that order.
* The ***parentheses*** around the parameters ***n*** and ***d*** are necessary. The *parentheses are optional only when one parameter is used*.
* Statement Lambdas: A statement lambda expands the types of operations that can be handled directly within a LE. For example, using a statement lambda, you can use loops, if statements, declare variables, and so on. A statement lambda is easy to create. Simply enclose the body of the LE within braces.

|  |  |
| --- | --- |
| * C#\_Example 18: Here is an example that uses a statement lambda to compute and return the factorial of an int value: * Notice, the statement lambda declares a variable called r, uses a for loop, and has a return statement. * Notice statement lambda's ending brace ends with semicolon ";" | using System;  delegate int IntOp(int end); // IntOp takes one int argument and returns an int result.  class StatementLambdaDemo { static void Main() { IntOp fact = n => { int r = 1;  for(int i=1; i <= n; i++) r = i \* r;  return r; }; // A statement lambda that returns the factorial of the value it is passed.  Console.WriteLine("The factorial of 3 is " + fact(3));  Console.WriteLine("The factorial of 5 is " + fact(5)); }} |

* In essence, a STATEMENT LAMBDA closely parallels an ANONYMOUS METHOD. Therefore, many anonymous methods will be converted to statement lambdas when *updating legacy code*.
* When a return statement occurs within a LE, it simply causes a return from the lambda. It does not cause the enclosing method to return.
* Learning more about LINQ:
* Begin by exploring the contents of System.Linq. Pay special attention to the capabilities of the extension methods defined by Enumerable.
* Next, expand your knowledge and expertise in writing LE. They are expected to play an increasingly important role in C# programming.
* Study the collections in System.Collections and System.Collections.Generic. An introduction to collections is presented in this chapter, but there is much more to learn.

**C#\_12.11 The Preprocessor**

C# defines several preprocessor directives, which affect the way that your program’s source file is interpreted by the compiler. These directives affect the text of the source file in which they occur, prior to the translation of the program into object code. The term preprocessor directive comes from the fact that these instructions were traditionally handled by a separate compilation phase called the preprocessor. Today’s modern compiler technology no longer requires a separate preprocessing stage to handle the directives, but the *name has stuck*. C# defines the following preprocessor directives:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***#define*** | ***#elif*** | ***#else*** | ***#endif*** | ***#endregion*** | ***#error*** |
| ***#if*** | ***#line*** | ***#pragma*** | ***#region*** | ***#undef*** | ***#warning*** |

* All preprocessor directives begin with a ***#*** sign. In addition, each preprocessing directive must be on its own line.
* #define : The #define directive defines a character sequence called a symbol. The existence or nonexistence of a symbol can be determined by #if or #elif, and is used to control compilation. ***#define symbol***
* Notice that there is no semicolon in this statement. There may be any number of spaces between #define and the symbol, but once the symbol begins, it is terminated only by a newline character. For example, to define the ***symbol*** EXPERIMENTAL, use this directive: ***#define*** *EXPERIMENTAL*
* #if and #endif : The #if and #endif directives allow you to conditionally compile a sequence of code based upon whether an expression involving one or more symbols evaluates to true.
* A ***symbol*** is ***true*** if it has been defined (i.e. defined by a #define directive). It is false otherwise. The general form of #if is:

|  |  |  |  |
| --- | --- | --- | --- |
| * If the ***expression*** following ***#if*** is ***true***, the code that is between it and ***#endif*** is compiled. Otherwise, the ***intervening code*** is skipped. The ***#endif*** *marks the end of an* ***#if*** *block*. * A ***symbol-expression*** can be *as simple as* just the *name of a symbol*. You can also use these operators in a symbol expression: **!**, **= =**, **!=**, **&&**, and **||**. Parentheses are also allowed. | | | ***#if*** *symbol-expression*  *statement sequence*  ***#endif*** |
| * C#\_Example 19: Here is a simple example that demonstrates condition compilation: * The program defines the symbol EXPERIMENTAL at the top *of the program*. * When ***#if*** is encountered, the symbol expression evaluates to true, and the first ***WriteLine()*** statement is compiled. If you ***remove*** the definition of EXPERIMENTAL and recompile the program, the first ***WriteLine()*** will not be compiled because the #if will evaluate to false. * In all cases, the second ***WriteLine()*** is compiled because it is not part of the #if block. | | **#define EXPERIMENTAL** *//defining a symbol*  **using System;**  class Test { static void Main() {  #if EXPERIMENTAL  Console.WriteLine("Compiled for experimental.");  #endif  Console.WriteLine("This is in all versions."); }} | |
| * Using SYMBOL EXPRESSION with **#if** : you can use a symbol expression in an ***#if***. * In this example, two symbols are defined, ***EXPERIMENTAL*** and ***TRIAL***. The second ***WriteLine()*** statement is compiled only if *both are defined*. * You can use the ***!*** to compile code when a *symbol is not defined*. For example,   **#if** !QC\_PASSED  **Console.WriteLine**("Code has not passed quality control.");  **#endif**   * The call to ***WriteLine()*** will be compiled only if ***QC\_PASSED*** has not been defined. | | For example: #define EXPERIMENTAL  **#define TRIAL**  **using System;**  **class** Test { **static void Main()** {  #if EXPERIMENTAL  Console.WriteLine("Compiled for experimental.");  #endif  #if EXPERIMENTAL && TRIAL  Console.Error.WriteLine("Testing experimental trial.");  #endif  Console.WriteLine("This is in all versions."); }} | |
| * #else : The ***#else*** directive works much like the ***else*** that is part of the C# language: It establishes an alternative if ***#if*** fails. Here is an example: * Since TRIAL is not defined, the #else portion of the second conditional code sequence is used. * Notice that #else *marks* *both the end* of the #if ***block*** and the *beginning* of the #else ***block***. This is necessary, because there can only be one #endif associated with any #if. Furthermore, there can be only one #else associated with any #if. * #elif: The #elif directive means “else if ” and establishes an if-else-if chain for ***multiple compilation options***. *#elif* is followed by a symbol expression. If the ***expression*** is ***true***, that block of code is compiled and no other #elif *expressions* are tested. Otherwise, the ***next block in the series is checked***. If no #elif succeeds, then if there is an #else, the code sequence associated with the #else is compiled. Otherwise, *no code in the entire* #if is compiled. | #define EXPERIMENTAL  using System;  class Test { static void Main() {  #if EXPERIMENTAL  Console.WriteLine("Compiled for experimental version.");  #else  Console.WriteLine("Compiled for release.");  #endif  #if EXPERIMENTAL && TRIAL  Console.Error.WriteLine("Testing experimental trial version.");  #else  Console.Error.WriteLine("Not experimental trial version.");  #endif  Console.WriteLine("This is in all versions."); }} | | |
| * Putting together all the pieces, here is the general form of ***#if/#else/#elif/#endif*** directives:   **#if** symbol-expression  *statement sequence*  **#elif** symbol-expression  *statement sequence*  **#elif** symbol-expression  *statement sequence*  .  .  .  **#else** symbol-expression  *statement sequence*  **#endif** | Here is an example that demonstrates **#elif** :  **#define RELEASE**  **using System;**  **class** Test { **static void Main()** {  **#if** EXPERIMENTAL  Console.WriteLine("Compiled for experimental version.");  **#elif** RELEASE  Console.WriteLine("Compiled for release.");  **#else**  Console.WriteLine("Compiled for internal testing.");  **#endif**  **#if** TRIAL && !RELEASE  Console.WriteLine("Trial version.");  **#endif**  Console.WriteLine("This is in all versions."); }} | | |

|  |  |
| --- | --- |
| * #undef : The #undef ***directive*** removes a *previously defined symbol*. That is, it “undefines” a symbol. The GENERAL FORM for #undef is:   ***#undef symbol***  For example:  **#define** MOBILE\_DEVICE  **#if** MOBILE\_DEVICE  // ...  **#undef** MOBILE\_DEVICE  *// At this point MOBILE\_DEVICE is undefined.*   * After the *#undef directive*, *MOBILE\_DEVICE* is no longer defined. * *#undef* is used principally ***to allow symbols to be localized to only those sections of code that need them***. | * #error: The #error directive forces the compiler to stop compilation. It is used for debugging. The general form of the #error directive is   **#error** error-message   * When the ***#error*** directive is encountered, the error message is displayed. For example, when the compiler encounters this line:   **#error Debug code still being compiled!**   * compilation stops and the error message “Debug code still being compiled!” is displayed. |
| * #warning: The ***#warning*** directive is similar to ***#error***, except that a warning rather than an error is produced. Thus, compilation is not stopped. The general form of the ***#warning*** directive is   **#warning** warning-message | * #region and #endregion : The ***#region*** and ***#endregion*** directives let you define a region that will be expanded or collapsed by the Visual Studio IDE when using the outlining feature. The general form is:   ***#region***  *// code sequence*  ***#endregion*** |

* #line : The ***#line*** directive sets the line number and filename for the file that contains the ***#line*** directive. The *number and the name are used when errors or warnings are output during compilation*. The general form for ***#line*** is: ***#line number “filename”***
* ***number*** is any positive integer and becomes the newline number, and the optional ***filename*** is any valid file identifier, which becomes the new filename.
* ***#line*** allows two options. The first is default, which ***returns*** *the* ***line numbering*** *to its* ***original condition***. It is used like this: ***#line default***
* The second is hidden. When stepping through a program, the *hidden* option allows a *debugger to bypass* lines between a ***#line hidden***

directive and the next ***#line*** directive that *does not include the* ***hidden*** *option*.

* #pragma : The ***#pragma*** directive gives instructions to the compiler, such as specifying an option. It has this general form: ***#pragma option***
* Here, ***option*** is the instruction passed to the compiler.
* In C# 3.0, there are two options supported by ***#pragma***. The first is ***warning***, which is used to **enable** or **disable** specific **compiler warnings**. It has these two forms: ***#pragma warning disable*** *warnings*

***#pragma warning restore*** *warnings*

* Here, ***warnings*** is a comma-separated list of warning numbers. To disable a warning, use the ***disable*** option. To enable a warning, use the ***restore*** option.
* For example, this ***#pragma*** statement disables warning 168, which indicates when a variable is declared but not used:

***#pragma*** ***warning*** ***disable*** 168

* The second ***#pragma*** option is ***checksum***. It is used to generate checksums for ASP.NET projects. It has this general form:

**#pragma checksum** "filename" "{GUID}" "check-sum"

* Here, filename is the name of the file, GUID is the globally unique identifier associated with ***filename***, and ***check-sum*** is a hexadecimal number that contains the checksum. This ***string*** must contain an even number of digits.

Note:

1. Differ from C/C++: The C# preprocessor directives have many similarities with the preprocessor directives supported by C and C++. Furthermore, in C/C++, you can use #define to perform textual substitutions, such as defining a name for a value, and to create function-like macros. But C# doesn’t support these uses of #define, in C#, #define is used only to define a symbol.
2. **#if** EXPERIMENTAL **Console.WriteLine**("Compiled for experimental version.");

* Above line is not possible and can cause error. The reason is that a directive only ends with a newline character. And the directives select portions of source-code in the *compile time* so avoid using the *single line* as above.

**C#\_12.12Runtime Type Identification (RTTI)**

In C#, it is possible to determine the type of an object at runtime. In fact, C# includes three keywords that support RUNTIME TYPE IDENTIFICATION (RTTI):

***is, as, typeof***

* Testing a Type with **"is"** : You can determine if an object is of a certain type by using the ***is*** operator. Its general form is: ***obj is type***

Here, obj is an expression that describes an object whose type is being tested against type. If the type of obj is the same as, or compatible with, type, then the outcome of this operation is true. Otherwise, it is false. Thus, if the outcome is true, obj can be cast to type. Here is an example:

|  |  |
| --- | --- |
| * Most of the ***is*** expressions are self-explanatory, but two warrant a closer look. First, notice this statement:   **if**(**b** **is** **A**) **Console.WriteLine**(". . . ");   * The ***if*** succeeds because ***b*** is an ***object*** of type ***B***, which is derived from type ***A***. Thus, ***b*** is compatible with ***A***. * However, the reverse is not true. When this line is executed:   **if**(**a** **is** **B**) **Console.WriteLine**(". . . "); | **using System**;  **class** A {}  **class** B : A {}  **class** UseIs { **static void Main()** { A a = **new** A(); B b = **new** B();  if(a is A) Console.WriteLine("a is an A");  if(b is A) Console.WriteLine("b is an A because it is derived from A"); *// true : b is an A.*  if(a is B) Console.WriteLine("This won't display -- a not derived from B"); *// false: a is not a B.*  if(b is B) Console.WriteLine("b is a B");  if(a is object) Console.WriteLine("a is an Object"); }} |

* The ***if*** does not succeed because ***a*** is of type ***A***, which is not derived from ***B***. Thus, they are ***not compatible***.
* **"as"** : Sometimes you will want to try a conversion at runtime but not throw an exception if the conversion fails (which is the case when a cast is used). To do this, use the ***as*** operator, which has this general form: ***expr as type***
* Here, ***expr*** is the expression being converted to type. If the ***conversion succeeds***, then a reference to type is returned. Otherwise, a null reference is returned. The ***as*** operator *can only be used to* perform *reference*, *boxing*, *unboxing*, or *identity conversions*.
* **"typeof"** : You can obtain type information about a given type by using ***typeof***, which has this general form: ***typeof(type)***
* Here, ***type*** is the type being obtained. It returns an instance of ***System.Type***, which is a class that describes the information associated with a type. Using this instance, you can retrieve information about the type.
* For example, this program displays the complete name for the ***StreamReader*** class:

**using System**; **using System.IO**;

**class** UseTypeof { **static void Main()** { **Type** t = **typeof**(StreamReader);

**Console.WriteLine**(t.FullName); }}

* ***System.Type*** contains many methods, fields, and properties that describe a ***type***. You will want to *explore it on your own*.

**C#\_12.13 Nullable Types**

Unused fields managing problem: To understand the problem, consider a simple customer database that keeps a record of the customer’s name, address, customer ID, invoice number, and current balance. In such a situation, it is possible to create a customer entry in which one or more of those fields would be unassigned. For example, a customer may simply request a catalog. In this case, no invoice number would be needed and the field would be unused.

* Prior to nullable types, handling the possibility of unused fields required either the use of placeholder values or an extra field that simply indicated whether a field was *used or not*. Of course, placeholder values *could only work* if ***there was a value that would otherwise not be valid***, which won’t be the case in all situations. Adding an extra field to indicate if a field was in *use works in all cases*, but having to *manually create* and *manage* such a *field* is an annoyance. The ***nullable*** ***type*** solves both problems.
* NULLABLE TYPE: A ***nullable type*** is a special version of a value type that is represented by a structure. ***In addition to the values*** defined by the *underlying type*, a nullable type *can store* the value null. Thus, a ***nullable*** type has the same range and characteristics as its underlying type.
* ***Nullable*** types simply adds the *ability to represent a value* that indicates that a variable of that type is unassigned.
* ***Nullable*** types are objects of ***System.Nullable<T>***, where ***T*** must be a non-nullable value type.
* Specify nullable type (Explicitly): First, you can explicitly use the type ***Nullable<T>***. For example, this declares variables of ***int*** and ***bool*** ***nullable*** types: ***Nullable<int>*** count; ***Nullable<bool>*** done;
* Specify nullable type (Use ? ): The second way to specify a nullable type is much shorter and is more commonly used. Simply ***follow the underlying*** type name with a ? . For example, here the more common way to declare variables of the ***nullable*** ***int*** and ***bool*** types: ***int?*** *count;* ***bool?*** *done;*
* When using *nullable* *types*, you will often see a nullable object created like this: ***int?*** count = null;
* This ***explicitly initializes*** ***count*** to ***null***. This satisfies the ***constraint*** that ***a variable must be given a value before it is used***. In this case, the *value* simply means *undefined*.
* ASSIGN a VALUE to a nullable variable: You can *assign* a value to a *nullable variable* in the *normal way* because a conversion from the underlying type to the nullable type is ***predefined***. For example, this assigns ***count*** the value ***100***: *count = 100;*
* Determining the value: There are two ways to determine whether a variable of a nullable type is null or ***contains a*** value.
* First, you can test its value against ***null***. For example, using ***count*** declared by the ***preceding statement***, the following *determines if it has a value*:

**if**(count != **null**) *// has a value*

* If ***count*** is not null, then it ***contains*** a value.
* The second way to determine if a nullable type contains a value is to use the ***HasValue*** read-only property defined by ***Nullable<T>***. It is:

***bool*** ***HasValue***

* ***HasValue*** will ***return*** ***true*** if the *instance* *on which it is called* ***contains a value***. It will ***return*** ***false*** otherwise. Using the ***HasValue*** ***property***, here is the second way to determine if the ***nullable*** object ***count*** has a value: ***if(count.HasValue)*** *// has a value*
* Obtaining the value: Assuming that a nullable object contains a value, you can *obtain its value* by using the ***Value*** *read-only property* defined by ***Nullable<T>***, which is: ***T Value***
* It *returns* the *value* of the *nullable instance* on which it is called.
* If you *try to obtain* a *value* from a variable that is *null*, a *System.InvalidOperationException* will be thrown.
* It is also possible to obtain the *value* of a *nullable* *instance* by ***casting*** it into its underlying type.
* C#\_Example 20: The following program puts together the pieces and demonstrates the basic mechanism that *handles a nullable type*:

**using System**;

**class** NullableDemo { **static void Main()** {

**int?** count = **null**; *//Declare a nullable type for int.*

**if**(count.**HasValue**) **Console.WriteLine**("count has this value: " + count.**Value**); **else** **Console.WriteLine**("count has no value");

count = 100;

**if**(count.**HasValue**) **Console.WriteLine**("count has this value: " + count.**Value**); **else** **Console.WriteLine**("count has no value"); }}

* The ?? Operator : If you attempt to use a cast to convert a ***nullable*** object to its underlying type, a *System.InvalidOperationException* will be thrown if the *nullable object* contains a *null value*. This can occur, for example, when you use a cast to ***assign the*** ***value*** of a nullable object to a variable of its underlying type.
* NULL COALESCING operator ?? : You can avoid the possibility of this exception begin thrown by using the **??** operator, which is called the *null* *coalescing operator*. It lets you specify a default value that will be used when the ***nullable*** object contains ***null***. It also eliminates the need for the ***cast***.
* The ***??*** operator has this general form: ***nullable-object ?? default-value***
* If *nullable-object* contains a *value*, then the value of the ***??*** is that *value*. Otherwise, the *value* of the ***??*** operation is *default-value*.

|  |  |
| --- | --- |
| * For example, in the following code, ***balance*** is ***null***. This causes ***currentBalance*** to be assigned the value ***0.0*** and no exception will be thrown.   **double?** balance = **null**;  **double** currentBalance;  currentBalance = balance **??** 0.0; | * In the next sequence, ***balance*** is given the value ***123.75***.   **double?** balance = 123.75;  **double** currentBalance;  currentBalance = balance **??** 0.0;   * Now, currentBalance will contain the value of balance, which is 123.75. |

* The right-hand expression of the **??** is evaluated only if the left-hand expression ***does not contain a value***.
* NULLABLE Objects and the RELATIONAL and LOGICAL Operators: Nullable objects can be used in relational expressions in just the *same way as their corresponding* non-nullable types.
* There is one additional rule that applies: When two nullable objects are compared using the **<**, **>**, **<=**, or **>=** operator, the result is ***false*** if either of the objects is ***null***. For example, consider this sequence:

**byte?** lower = 16; **byte?** upper = null; *// Here, lower is defined, but upper isn't.*

**if**(lower < upper) *//* ***less than*** *is false*

* Here, the result of the test for less than is false. However, somewhat counterintuitively, so is the inverse comparison:

**if**(lower > upper) *//* ***greater than*** *is also false!*

* Thus, when *one* (or *both*) of the *nullable objects* in a comparison is *null*, the *result* of that comparison is *always false*. NULL does not participate in an ORDERING relationship.
* Testing for null using == or != : You can test whether a nullable object contains null, by using the **==** or **!=** operator. For example, this is a valid test that will result in a true outcome: ***if(upper == null)*** *// ...*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| * Truth table for logical expression involving two **bool?** objects: When a logical expression involves two **bool?** objects, the outcome of that expression will be one of three values: ***true***, ***false***, or ***null*** *(****undefined****)*. The entries that are added to the truth table for the **&** and **|** operators that apply to **bool?** * Remember: When the **!** operator is applied to a **bool?** value that is **null**, the outcome is **null**. | **P** | **Q** | **P | Q** | **P & Q** |
| true | null | true | null |
| false | null | null | false |
| null | true | true | null |
| null | false | null | false |

**C#\_12.14 Unsafe Code**

“UNSAFE” code is the code that does not execute under the full management of the Common Language Runtime (CLR). As explained in Chapter 1, C# is normally used to create managed code. Since the unmanaged code is not subject to the same controls and constraints as managed code, it is called “unsafe” because it is impossible to verify that it won’t perform some type of harmful action. UNSAFE means that it is possible for the code to perform actions that are not subject to the *supervision* of the *managed context*.

* Managed code prevents the use of pointers. If you are familiar with C or C++, then you know that pointers are variables that hold the addresses of other objects. Thus, conceptually, *pointers* are a bit like *references* in C#.
* The main difference is that a ***pointer*** can point anywhere in memory; a ***reference*** ***always refers to an object of its type***. Since a pointer can point anywhere in memory, it is possible to misuse a pointer.
* It is also easy to introduce a coding error when using pointers. This is why C# does not support ***pointers*** when creating managed code.
* ***Pointers*** are, however, both useful and necessary for some types of programming (such as when writing code that interacts with a device.
* All pointer operations must be marked a*s unsafe*, since they execute outside the managed environment.
* A Brief Look at Pointers: A pointer is a variable that holds the address of some other object. For example, if ***p*** contains the address of ***y***, then ***p*** is said to “point to” ***y***. Pointer variables must be declared as such. The general form of a pointer variable declaration is: ***type\* var-name;***
* Here, ***type*** is the type of object to which the ***pointer will point***, and it must be a non-reference type. ***var-name*** is the name of the pointer variable. For example, to declare ***ip*** to be a pointer to an ***int***, use this declaration: ***int\* ip;*** For a float pointer, use: ***float\* fp;***
* In general, in a declaration statement, following a type name with an " **\*** " creates a pointer type.
* The type of data that a pointer will point to is determined by its referent type, which is also commonly referred to as the pointer’s base type. Thus, in the preceding examples, ***ip*** can be *used to point* to an ***int***, and ***fp*** can be *used to point* to a ***float***. Understand, however, that there is nothing that actually prevents a *pointer* from *pointing to something else*. This is why *pointers* are potentially *unsafe*.
* Remember that a pointer type can be declared only for non-reference types. This means that the referent type of a pointer can be any of the simple types, such as ***int***, ***double***, and ***char***; an ***enumeration*** type; or a ***struct*** (as long as its fields are all non-reference types).
* Pointer operators: There are two key pointer operators: **\*** and **&**.
* The & is a unary operator that returns the memory address of its operand. (Recall that a unary operator requires only one operand.) For example:

|  |  |
| --- | --- |
| int**\*** ip;  int num = 10;  ip = &num; | * It puts into ***ip*** the memory address of the variable ***num***. This address is the location of the variable in the ***computer’s internal memory***. It has nothing to do with the value of ***num***. Thus, ***ip*** does not contain the value ***10*** (num’s initial value). It contains the address at which num is stored. The operation of & can be *remembered* as returning “the address of” the variable it precedes. Therefore, the above assignment statement could be verbalized as “ip ***receives the address of*** num.” |

* The second operator is **\***, and it is the complement of **&**. It is a unary operator that dereferences the ***pointer***. In other words, it evaluates to the variable ***located at the address*** specified by its operand. Continuing with the same example, if ***ip*** contains the memory address of the variable ***num***, then
* **int** val; val = \*ip; will place into **val** the value **10**, which is the value of **num** (which is pointed to by **ip**). The operation of **\*** can be remembered as “**at address**.” In this case, then, the statement could be read as “**val** ***receives the value*** **at address** **ip**.”
* Pointers with structures: ***Pointers*** can also be used with structures. When you access a member of a structure through a pointer, you must use the **–>** operator rather than the dot (**.**) operator. The **–>** is informally called the **arrow operator**. For example, given this structure:

**struct** MyStruct{ **public** **int** x; **public** **int** y; **public** **int** sum() { **return** x + y; } }

|  |  |
| --- | --- |
| * Pointers can have simple arithmetic operations performed on them. For example, you can increment or decrement a pointer. Doing so causes it to point to the next or previous object of its referent type. * You can also add or subtract *integer values* to or from a pointer. You can subtract one pointer from another (which yields the number of elements of the referent type separating the two), but you CAN’T ADD pointers. | * Here is how you would access its members through a pointer:   **MyStruct** o = **new** MyStruct();  **MyStruct\*** p; *// declare a pointer*  p = &o; p->x = 10; p->y = 20;  **Console.WriteLine**("Sum is " + p->sum()); |

Note:

1. The declaration and use of pointers in C# parallels that of C/C++; if you know how to use pointers in C/C++, then you can use them in C#. But remember, the point of C# is to create managed code. Its ability to support unmanaged code allows it to be applied to a special class of problems. It is *not for normal C# programming*.
2. To **compile unmanaged code**, you must use the **/unsafe** compiler option. In general, if you need to create large amounts of code that execute outside of the CLR, then you are probably better off using C++.
3. Differ from C/C++: Declaring a pointer in C++, the \* is not distributive over a list of variables in a declaration. Thus, in C++, this statement, ***int\* p, q;***

declares an integer pointercalled ***p*** and an integercalled ***q***. It is equivalent to this two declarations: **int\*** p; **int** q;

* In C#, the **\*** is distributive and the declaration ***int\* p, q;*** creates two pointer variables. It is the same as: ***int\**** p; ***int\**** q;

This is an important difference to be aware of when porting C++ code to C#.

* The UNSAFE Keyword: Any code that uses pointers must be marked as unsafe by using the ***unsafe*** keyword. You can mark ***types*** (such as classes and structures), members (such as methods and operators), or individual blocks of code as ***unsafe***. For example, here is a program that uses pointers inside ***Main()***, which is marked unsafe:

|  |  |
| --- | --- |
| *// You need to compile this program by use of the* ***/unsafe*** *option.*  **using System**;  **class** UnsafeCode { **unsafe static void Main()** { *// Mark Main as unsafe.*  **int** count = 99;  **int\*** p; *// create an int pointer*  p = &count; *// put address of count into p* | **Console.WriteLine**("Initial value of count is " + \*p);  \*p = 10; *// assign to count via p*  **Console.WriteLine**("New value of count is " + \*p); }}   * This program uses the pointer ***p*** to obtain the value contained in ***count***, which is the object that ***p*** points to. Because of the pointer operations, it must be marked ***unsafe*** in order for it to be compiled. |

* Using fixed: The ***fixed*** keyword has two uses. The first is to prevent a managed object from being moved by the garbage collector. This is needed when a pointer refers to a field within such an object, for example. Since the ***pointer has no knowledge of the actions*** of the garbage collector, if the object is moved, the pointer will point to the wrong location. Here is the general form of fixed: ***fixed (type\* p = &fixedVar) {*** */\* use fixed object \*/* ***}***
* Here, ***p*** is a ***pointer*** that is *being assigned* the *address of a variable*. The variable will *remain* in its *current memory location* until the ***block of code*** has executed. You can also use a single statement for the target of a fixed statement. The ***fixed*** keyword can be used only in an ***unsafe*** context.

|  |  |
| --- | --- |
| * The second use of ***fixed*** is to create fixed-sized, single-dimensional arrays. These are referred to as fixed-size buffers. A fixed-size buffer is always a member of a struct. The purpose of a fixed-size buffer is to allow the creation of a *struct* in which the *array elements* that make up the *buffer* are *contained within the struct*. * Normally, when you include an array member in a struct, only a reference to the array is actually held within the struct. By using a fixed-size buffer, you cause the entire array to be contained within the struct. To create a fixed-size buffer, use this form: **fixed type buf-name[size];** * Here, ***type*** is the data type of the array, ***buf-name*** is the name of the fixed-size buffer, and size is the number of elements in the buffer. Fixed-size buffers can be specified only inside a struct. This results in a structure that can be used in situations in which the size of a struct is important, such as in mixed-language programming, interfacing to data *not created by a C# program*, or whenever a non-managed struct containing an array is required. | * Here is an example of fixed:   using System;  class Test { public int num;  public Test(int i) { num = i; } }  class UseFixed { unsafe static void Main() { *// Mark Main as unsafe.*  Test o = new Test(19);  fixed (int\* p = &o.num) { *// use fixed to put address of o.num into p*  Console.WriteLine("Initial value of o.num is " + \*p);  \*p = 10; // assign to o.num via p  Console.WriteLine("New value of o.num is " + \*p); }  }}   * Here, fixed prevents o *from being moved*. This is required because p *points* to o.num. If o *moved*, then o.num would *also move*. This would cause p to *point* to an *invalid location*. The use of *fixed* prevents this. |

* Fixed-size buffers can be used only within an unsafe context.

**C#\_12.15 Attributes**

C# allows you to add declarative information to a program in the form of an attribute. An attribute defines additional information that is associated with a class, structure, method, and so on. For example, you might define an attribute that determines the type of button that a *class will display*.

* Attributes: Attributes are specified between square brackets, preceding the ***item*** they apply to. You can define your own attribute or use attributes defined by C#. It is quite easy to use two of C#’s built-in attributes: ***Conditional*** and ***Obsolete***.
* Conditional: The attribute ***Conditional*** is perhaps C#’s most interesting attribute. It allows you to *create CONDITIONAL METHODS*. A conditional method is *invoked only when* a *specific symbol* has been *defined* via ***#define***. Otherwise, the method is BYPASSED. Thus, a conditional method offers an alternative to conditional compilation using ***#if***. To use the ***Conditional*** attribute, you must include the ***System.Diagnostics*** *namespace*. For Exaxmple:

|  |  |  |  |
| --- | --- | --- | --- |
| * Notice that the program defines the symbol TRIAL. Next, notice how the methods Trial( ) and Release( ) are coded. They are both preceded with the Conditional attribute, which has this general form: ***[Conditional symbol]***   where symbol is the symbol that determines whether the method will be executed. This attribute can be used only on methods. If the symbol is ***defined***, then when the method is called, it will be executed. If the symbol is ***not defined***, then the method is not executed.   * Inside ***Main()***, both ***Trial()*** and ***Release()*** are called. However, only TRIAL is defined. Thus, ***Trial()*** is executed. The call to ***Release()*** is ignored. [If you define RELEASE, then ***Release()*** will also be called. If you remove the definition for TRIAL, then ***Trial()*** will not be called.} | | * C#\_Example 21:   #define TRIAL  using System;  using System.Diagnostics;  class Test {  [Conditional("TRIAL")] void Trial() { Console.WriteLine("Trial version, not for Use."); }  [Conditional("RELEASE")] void Release() { Console.WriteLine("Final release version."); }  **static void Main()** { **Test** t = **new** Test();  t.**Trial()**; *// call only if TRIAL is defined*  t.**Release()**; */\* called only if RELEASE is defined \*/* }}  Output:  **Trial version, not for distribution.** | |
| * Restrictions: | * conditional methods ***cannot be preceded*** with the ***override*** keyword. * conditional methods ***must be members*** of a class or structure, not an interface; | | * conditional methods must ***return*** ***void;*** |

* Obsolete: The ***System.Obsolete*** attribute lets you mark a program element as obsolete. It has two basic forms. The first is: ***[Obsolete "message"]***
* Here, **message is displayed** when that **program element** is **compiled**. Here is a short example:

|  |  |
| --- | --- |
| * When the call to ***MyMeth()*** is encountered in ***Main()*** during program compilation, a warning will be generated that tells the user to use ***MyMeth2()*** instead. | **using System**;  **public class** Test { [**Obsolete**("Use MyMeth2, instead.")] **public static int** MyMeth(**int** a, **int** b){ **return** a/b; }  **public static int** MyMeth2(**int** a, **int** b) { **return** b == 0 ? 0 : a /b; } *// Improved version*  **static void Main()** { **Console.WriteLine**("4 / 3 is " + Test.MyMeth(4, 3)); *// Warning displayed for this.*  **Console.WriteLine**("4 / 3 is " + Test.MyMeth2(4, 3)); */\* No warning here \*/* }} |

* A second form of Obsolete is shown here: ***[Obsolete("message", error)]***
* Here, ***error*** is a ***Boolean value***. If it is ***true***, then the use of the ***obsolete*** item generates a compilation error rather than a warning. The difference is, of course, that a program containing an error cannot be compiled into an executable program.

**C#\_12.16 Conversion Operators (More like to operator overloading Ch. 3)**

In some situations, you will want to use an object of a class in an expression involving other types of data. Sometimes, overloading one or more operators can provide the means of doing this. However, in other cases, what you want is a simple type conversion from the class type to the target type. To handle these cases, C# allows you to create a CONVERSION OPERATOR. A conversion operator converts an object of your class into another type.

* There are two forms of CONVERSION OPERATORS: ***implicit*** and ***explicit***. The general form for each is shown here:

***public static operator implicit*** *target-type****(****source-type* ***v) { return*** *value****; }***

***public static operator explicit*** *target-type****(****source-type* ***v) { return*** *value****; }***

* Here, ***target-type*** is the target type that you are converting to, ***source-type*** is the type you are converting from, and ***value*** is the value of the class ***after*** ***conversion***.
* The conversion operators ***return*** data of type ***target-type***, and no other ***return-type*** ***specifier*** is allowed.
* If the CONVERSION OPERATOR specifies ***implicit***, then the conversion is invoked automatically, such as when an object is used in an expression with the target type.
* When the CONVERSION OPERATOR specifies ***explicit***, the conversion is invoked when a cast is used.
* You cannot define both an ***implicit*** and an ***explicit*** CONVERSION OPERATOR for the same target and source types.
* C#\_Example 22: To illustrate a conversion operator, We use ThreeD class once we discussed. ThreeD stores *three-dimensional coordinates*. Suppose you want to convert an object of type ThreeD into a numeric value so it can be used in a numeric expression. Further more, the conversion will take place by computing the distance from the point to the origin, which will be represented as a double. To accomplish this, you can use an IMPLICIT CONVERSION OPERATOR that looks like this:

**public static implicit operator double**(**ThreeD** op1) { **return** **Math.Sqrt**(op1.x \* op1.x + op1.y \* op1.y + op1.z \* op1.z); }

* It takes a ***ThreeD*** object and returns its distance to the origin as a ***double*** value. Folowing illustrates this conversion operator:

|  |  |
| --- | --- |
| using System;  *// A three-dimensional coordinate class.*  class ThreeD { int x, y, z; *// 3-D coordinates*  public ThreeD() { x = y = z = 0; }  public ThreeD(int i, int j, int k) { x = i; y = j; z = k; }  *// Overload binary +.*  public static ThreeD operator +(ThreeD op1, ThreeD op2) {  ThreeD result = new ThreeD();  result.x = op1.x + op2.x;  result.y = op1.y + op2.y;  result.z = op1.z + op2.z;  return result; }  *// An implicit conversion operator from ThreeD to double.*  public static implicit operator double(ThreeD op1) {  return Math.Sqrt(op1.x \* op1.x + op1.y \* op1.y + op1.z \* op1.z); }  *// It returns the distance from the origin to the specified point.*  *// Show X, Y, Z coordinates.*  public void Show() { Console.WriteLine(x + ", " + y + ", " + z); }  } | class ConversionOpDemo { static void Main() { ThreeD alpha = new ThreeD(12, 3, 4);  ThreeD beta = new ThreeD(10, 10, 10);  ThreeD gamma = new ThreeD();  double dist;  Console.Write("Here is alpha: "); alpha.Show(); Console.WriteLine();  Console.Write("Here is beta: "); beta.Show(); Console.WriteLine();  */\* Add alpha and beta together. This does NOT invoke the conversion operator because no conversion to double is needed. \*/*  gamma = alpha + beta;  Console.Write("Result of alpha + beta: "); gamma.Show(); Console.WriteLine();  */\* The following statement invokes the conversion operator because the value of a is assigned todist, which is a double. \*/*  dist = alpha; *// convert to double : Conversion Operator invoked.*  Console.WriteLine("Result of dist = alpha: " + dist);  Console.WriteLine();  *// Follwing also invokes the conversion operator because the expression requires a double value.*  if(beta > dist) Console.WriteLine("beta > origin"); *// Conversion Operator invoked.*  }} |

* As the program illustrates, when a ***ThreeD*** object is used in a double expression, such as ***dist = alpha***, the conversion is applied to the object. In this specific case, the conversion *returns the value* 13, which is ***alpha’s distance*** from the origin. However, when an expression does not require a conversion to ***double***, the CONVERSION OPERATOR is not called. This is why ***gamma = alpha + beta*** does not invoke operator ***double()***.
* Remember that you can create different conversion operators to meet different needs. You could define one that converts to ***long***, for example. Each conversion is applied automatically and independently.
* An implicit conversion operator is used automatically when a conversion is *required* in an expression, when passing an object to a method, in an assignment, and also when an explicit cast to the target type is used.
* Alternatively, you can create an EXPLICIT CONVERSION OPERATOR that is invoked only when an explicit cast is used. An explicit conversion operator is ***not invoked automatically***. For example, here is the conversion operator in the *previous program reworked* as an explicit conversion:

**public static explicit operator double**(**ThreeD** op1) { **return** **Math.Sqrt**(op1.x \* op1.x + op1.y \* op1.y + op1.z \* op1.z); } *// This is now explicit.*

* Now, this statement from the previous example ***dist = alpha;*** must be recoded to use an explicit cast, as: ***dist = (double) alpha;***
* Furthermore, this statement: ***if(beta > dist) Console.WriteLine(****"beta is farther from the origin"****);*** must be reworked like following:

**if**((**double**) beta > dist) **Console.WriteLine**("beta is farther from the origin");

* Because the ***conversion operator*** is now marked as ***explicit***, conversion to ***double*** must be ***explicitly cast*** in all cases.
* Why EXPLICIT CONVERSION? Since IMPLICIT CONVERSIONS are invoked automatically, without CAST: Implicit conversions should be used only in situations in which the conversion is inherently error-free. To ensure this, implicit conversions should be created only when two conditions are met.
* The first is that no loss of information, such as truncation, overflow, or loss of sign, occurs.
* The second is that the conversion does not throw an exception.
* If the conversion cannot meet these two requirements, then you should use an EXPLICIT CONVERSION.

|  |  |  |
| --- | --- | --- |
| * Restrictions: | * You cannot create a conversion from a built-in type to another built-in type. For example, you cannot redefine the conversion from double to int. * You cannot define both an implicit and an explicit conversion for the same source and target types. | * You cannot define a conversion to or from object. * You cannot define a conversion from a base class to a derived class. * You cannot define a conversion from or to an interface. |

**C#\_12.17 Introduction to Collections (Some kind of LIBRARY: standard collections)**

One of the most important parts of the .NET Framework is collections. As it relates to C#, a COLLECTION is a group of objects. The .NET FRAMEWORK contains a large number of INTERFACES and CLASSES that define and implement various types of COLLECTIONS. Collections simplify many programming tasks because they supply off-the-shelf solutions to several common, but sometimes tedious-to-develop, data structures. For example, there are BUILT-IN COLLECTIONS that support dynamic ARRAYS, LINKED LISTS, STACKS, QUEUES, and HASH TABLES.

* Containing both generic and non-generic COLLECTION CLASSES, the Collections API is very large. However, because collections are an increasingly important part of C# programming, they are a feature that you need to be aware of. Towards this end, this section provides a brief introduction to this important subsystem. As you advance in your study of C#, collections are definitely a feature that you will want to study further.
* Collection Basics: The principal benefit of collections is that they STANDARDIZE the way groups of objects are *handled by a program*. All COLLECTIONS are designed around a set of cleanly defined INTERFACES. Several built-in implementations of these INTERFACES are provided, which you can use as-is. You can also implement your own COLLECTION, but you will seldom need to.
* As mentioned, the .NET Framework defines both generic and non-generic collections. The original 1.0 release contained only non-generic collections, but the generic collections were added by the 2.0 release. Although both are still used, new code should focus on the generic collections because they are type-safe. (The original, non-generic collections store object references, which makes them vulnerable to type mismatch errors.) The non-generic collection CLASSES and INTERFACES are declared in ***System.Collections***. Generic collections are declared in ***System.Collections.Generic***.
* The basic functionality of the COLLECTIONS is defined by the interfaces that they implement. For generic collections, the foundation is the ***ICollection<T>*** interface, which is implemented by all generic collections. It inherits ***IEnumerable<T>*** (which extends ***IEnumerable***) and defines the methods shown here:

|  |  |
| --- | --- |
| Method | Description |
| **void Add(T obj)** | Adds obj to the invoking collection. |
| **void CopyTo(T[] target, int startIdx)** | Copies the contents of the invoking collection to the array specified by target, beginning at index specified by startIdx. |
| **void Clear()** | Deletes all elements from the invoking collection. |
| **bool Contains(T obj)** | Returns true if the invoking collection contains the object passed in obj and false otherwise. |
| **bool Remove(T obj)** | Removes the first occurrence of obj from the ***invoking collection***. Returns true if obj was removed and false if it was not found in the ***invoking collection***. |
| **IEnumerator GetEnumerator()** | Returns the non-generic enumerator for the collection. (Specified by ***IEnumerable***.) |
| **IEnumerator<T> GetEnumerator()** | Returns the enumerator for the collection. (Specified by ***IEnumerable<T>***.) |

* Methods that modify a COLLECTION will throw ***NotsupportedException*** if the collection is read-only.

|  |  |  |
| --- | --- | --- |
| * properties defined by ICollection<T>: ***ICollection<T>*** also defines the properties: | ***int Count { get; }*** | ***bool IsReadOnly { get; }*** |

* ***Count*** contains the number of items currently held in the collection. ***IsReadOnly*** is true if the collection is read-only. It is false if the collection is read/write.
* Because ***ICollection<T>*** inherits the ***IEnumerable<T>*** interface, it ensures that all of the COLLECTIONS CLASSES can be ENUMERATED ***(cycled through one element at a time)***.
* Furthermore, inheriting ***IEnumerable<T>*** allows a collection to be used as a data source for queries or iterated by the foreach loop. (Recall that only instances of objects that implement ***IEnumerable*** or ***IEnumerable<T>*** can be used as a data source for a query.) Because collections implement ***IEnumerable<T>***, they also support the extension methods defined for ***IEnumerable<T>***.
* Collections API defines several other interfaces that add functionality. For example, ***IList<T>*** extends ***ICollection<T>***, adding support for collections whose elements can be accessed through an index. The ***IDictionary<TK, TV>*** extends ***ICollection<T>*** to support the storage of key/value pairs.
* The Collections API provides several implementations of the collections interfaces. For example, the generic ***List<T>*** collection implements a type-safe dynamic array, which is an array that grows as needed. There are CLASSES that implement stacks and queues, such as ***Stack<T>*** and ***Queue<T>***. Other classes, such as ***Dictionary<TK, TV>***, store key/value pairs.

**C#\_12.18 List<T>**

Perhaps the most widely used collection is **List<T>**, which implements a generic, dynamic array. It has the constructors:

|  |  |  |
| --- | --- | --- |
| **public List()** | **public List(IEnumerable<T> c)** | **public List(int capacity)** |
| It builds an empty list with a default initial capacity. | It builds a list that is initialized with the elements of the collection specified by c and with an initial capacity equal to the number of elements | It builds a list that has the specified initial capacity. The capacity grows automatically as elements are added to a **List<T>**. Each time the list must be enlarged, its capacity is increased. |

* ***List<T>*** implements several INTERFACES, including ***IList<T>***. The ***IList<T>*** *INTERFACE* extends ***ICollection<T>***, ***IEnumerable<T>***, and ***IEnumerable***. The ***IList<T>*** *INTERFACE* defines the behavior of a generic collection that allows elements to be accessed via a zero-based index. In addition to the *methods* specified by the *interfaces* that it *extends*, ***IList<T>*** adds the methods shown here. If the collection is READ-ONLY, then the ***Insert()*** and ***RemoveAt()*** methods will throw a ***NotSupportedException***.

|  |  |
| --- | --- |
| Method | Description |
| ***int IndexOf(T obj)*** | Returns the index of obj if obj is ***contained*** within the invoking collection. If obj is ***not found***, –1 is returned. |
| ***void Insert(int idx, T obj)*** | Inserts obj at the index specified by idx. |
| ***void RemoveAt(int idx)*** | Removes the object at the index specified by idx from the invoking Collection. |

* Indexer by IList<T>: **IList<T>** defines the indexer: **T this[int idx] { get; set; }**
* This indexer ***sets*** or ***gets*** the value of the element at the ***index*** specified by ***idx***.
* In addition to the functionality defined by the interfaces that it implements, List<T> provides much of its own. For example, it supplies methods that sort a list, perform a binary search, and convert a list into an array. **List<T>** also defines a ***property*** called ***Capacity*** that gets or sets the capacity of the invoking list. The capacity is the number of elements that can be held *before* the list must be enlarged. (It is not the ***number of elements*** currently in the ***list***.) Because a list grows automatically, it is ***not*** ***necessary*** to set the ***capacity*** ***manually***. However, for efficiency reasons, you might want to ***set*** the ***capacity*** when you know in *advance* *how many elements* the list will contain. This prevents the overhead associated with the allocation of more memory.
* C#\_Example 23 (Create a Dynamic Array): A case study is presented that will give you an idea of their power and illustrate the general way in which they are used. It uses the ***List<T>*** collection. Here is a program that demonstrates the basic usage of ***List<T>.*** It creates a dynamic array of type ***int***. notice that the list AUTOMATICALLY EXPANDS and CONTRACTS, based on the ***number of elements*** that it contains.

|  |  |  |
| --- | --- | --- |
| using System;  using System.Collections.Generic;  class ListDemo { static void Main() {  *// Create a list of integers.*  List<int> lst = new List<int>();  Console.WriteLine("Initial number of elements: " + lst.Count);  Console.WriteLine();  Console.WriteLine("Adding 5 elements");  *// Add elements to the array list.*  lst.Add(1); lst.Add(-2); lst.Add(14); lst.Add(9); lst.Add(88);  Console.WriteLine("Number of elements: " + lst.Count);  *// Display the array list using array indexing.*  Console.Write("Contents: ");  for(int i=0; i < lst.Count; i++) Console.Write(lst[i] + " ");  Console.WriteLine("\n");    Console.WriteLine("Removing 2 elements");  *// Remove elements from the array list.*  lst.Remove(-2); lst.Remove(88);  Console.WriteLine("Number of elements: " + lst.Count); | *// Use foreach loop to display the list.*  Console.Write("Contents: ");  foreach(int i in lst) Console.Write(i + " ");  Console.WriteLine("\n");  Console.WriteLine("Adding 10 elements");  *// Add enough elements to force lst to grow.*  for(int i=0; i < 10; i++) lst.Add(i);  Console.WriteLine("Number of elements after adding 10: " + lst.Count); | |
| Console.Write("Contents: ");  foreach(int i in lst) Console.Write(i + " ");  Console.WriteLine("\n");  *// Change contents using array indexing.*  Console.WriteLine("Change 3 elements");  lst[0] = -10; lst[1] = -lst[1]; lst[2] = 99;  Console.Write("Contents: ");  foreach(int i in lst) Console.Write(i + " ");  Console.WriteLine(); }} | output  **Initial number of elements: 0**  **Adding 5 elements**  **Number of elements: 5**  **Contents: 1 -2 14 9 88**  **Removing 2 elements**  **Number of elements: 3**  **Contents: 1 14 9**  **Adding 10 elements**  **Number of elements after adding 10: 13**  **Contents: 1 14 9 0 1 2 3 4 5 6 7 8 9**  **Change first three elements**  **Contents: -10 -14 99 0 1 2 3 4 5 6 7 8 9** |

* The program begins by creating an instance of ***List<int>*** called ***lst***. This ***collection*** is initially empty. Notice how its size grows as elements are added. As explained, ***List<T>*** creates a dynamic array, which grows as needed to accommodate the number of elements that it must hold. Also notice how ***lst*** can be indexed, using the same syntax as that used to index an ***array***.
* Because the ***List<T>*** collection creates an ***indexable***, dynamic array, it is often used in place of an array. The principal advantage is that you don’t need to know how many elements will be stored in the list at compile time. Of course, arrays offer a bit better runtime efficiency, so using ***List<T>*** trades ***speed*** for convenience.

**C#\_12.19 Queue<T>**

In the preceding chapters, several examples have developed and evolved a queue class as a means of illustrating several fundamental C# programming concepts, such as encapsulation, properties, exceptions, and so on. Although creating your own data structures, such as a queue, is a good way to learn about C#, it is ***not something that you will normally need to do***. Instead, you will usually use one of the STANDARD COLLECTIONS.

* In the case of a QUEUE, STANDARD COLLECTION is ***Queue<T>***. It provides a high-performance implementation that is fully integrated into the overall Collections framework. ***Queue<T>*** is a dynamic collection that grows as needed to accommodate the elements it must store. ***Queue<T>*** defines the following constructors:

|  |  |  |
| --- | --- | --- |
| **public Queue()** | **public List(IEnumerable<T> c)** | **public Queue(int capacity)** |
| Creates an empty Queue with a default initial capacity. | Creates a queue that contains the elements of the collection specified by c. | Creates an empty queue with the initial capacity specified by capacity |

* In addition to the functionality defined by the collection interfaces that it implements, ***Queue<T>*** defines the methods shown here. To put an object in the queue, call ***Enqueue()***. To remove and return the ***object*** at the front of the queue, call ***Dequeue()***. An ***InvalidOperationException*** is thrown if you call ***Dequeue()*** when the invoking queue is empty. You can use ***Peek()*** to return, but not remove, *the next object*.

|  |  |
| --- | --- |
| Method | Description |
| ***public T Dequeue()*** | Returns the object at the front of the invoking queue. The object is removed in the process. |
| ***public void Enqueue(T v)*** | Adds v to the end of the queue. |
| ***public T Peek()*** | Returns the object at the front of the invoking queue, but does not remove it. |
| ***public T[ ] ToArray()*** | Returns an array that contains copies of the elements of the invoking queue. |
| ***public void TrimExcess()*** | Removes the excess capacity of the invoking queue if its size is less than 90 percent of its capacity. |

* C#\_Example 24 (Create a Queue): Following creates a short program that simulates using a queue to grant users access to a network. This example uses a queue to simulate scheduling access to a network by a collection of users. It doesn’t actually do any real scheduling. Instead, it simply fills a queue with the names of the users and then grants the users access based on the order in which they are entered into the queue. Of course, since this is a simulation, the program simply displays the user’s name when a user is granted access.

|  |  |
| --- | --- |
| using System; using System.Collections.Generic;  class QueueDemo { static void Main() {  Queue<string> userQ = new Queue<string>();  Console.WriteLine("Adding users to the network user queue.\n");  userQ.Enqueue("Eric"); userQ.Enqueue("Tom");  userQ.Enqueue("Ralph"); userQ.Enqueue("Ken");  Console.WriteLine("Granting network access in queue order.\n");  while(userQ.Count > 0) {  Console.WriteLine("Granting network access to: " + userQ.Dequeue()); }  Console.WriteLine("\nUser queue is exhausted."); }} | output  **Adding users to the network user queue.**  **Granting network access in queue order.**  **Granting network access to: Eric**  **Granting network access to: Tom**  **Granting network access to: Ralph**  **Granting network access to: Ken**  **User queue is exhausted.** |

* Queue<string> userQ = new Queue<string>(); Notice that a Queue called userQ is created that can hold references to objects of type string.
* puts user names into the queue: ***userQ.Enqueue("Eric"); userQ.Enqueue("Tom"); userQ.Enqueue("Ralph"); userQ.Enqueue("Ken");***
* To removes one name at a time, which simulates granting access to the network.

**while(userQ.Count > 0) { Console.WriteLine("Granting network access to: " + userQ.Dequeue()); }**

* The STANDARD ***Queue<T>*** CLASS offers a solution that all C# programmers will instantly recognize, thus making your programs easier to maintain.

**C#\_12.20 Other Keywords**

* internal Access Modifier: In addition to the access modifiers ***public***, ***private***, and ***protected***, which we have been using throughout this book, C# also defines ***internal***. ***internal*** declares that a member is known throughout all files in an assembly, but unknown outside that assembly.
* An assembly is a file (or files) that contains all deployment and version information for a program. Thus, in simplified terms, a member marked as ***internal*** is known throughout a program, but not elsewhere.
* Sizeof: To know the size, in bytes, of one of C#’s value types. To obtain this information, use the ***sizeof*** operator. It has this general form: ***sizeof(type)***
* Here, type is the type whose size is being obtained. Thus, it is intended primarily for special-case situations, especially when working with a blend of ***managed*** and ***unmanaged*** code.
* lock: The lock keyword is used when working with multiple threads. In C#, a program can contain two or more threads of execution. When this is the case, *pieces* of the *program* are *multitasked*. Thus, pieces of the program execute independently and simultaneously. This raises the prospect of a special type of problem: What if two threads try to use a resource that can be used by only one thread at a time? To solve this problem, you can create a critical code section that will be executed by one and only one thread at a time. This is accomplished by ***lock***. Its general form is: **lock(obj)** { */\* critical section \*/* }
* Here, ***obj*** is the object on which the lock is synchronized. If one thread has already entered the critical section, then a second thread will wait until the first thread exits the critical section.
* When the first thread leaves the critical section, the lock is released and the second thread can be **granted** the **lock**, at which point the second thread can execute the critical section.
* readonly: You can create a read-only field in a class by declaring it as readonly. A readonly field can be given a value only by using an initializer when it is declared, or by assigning it a value within a constructor. Once the value has been set, it can’t be changed outside the constructor. Thus, readonly fields are a good way to create constants, such as array dimensions, that are used throughout a program. Both ***static*** and ***non-static*** readonly fields are allowed. For example

|  |  |
| --- | --- |
| **using System**;  **class** MyClass { **public static** **readonly** **int** SIZE = 10; }  **class** DemoReadOnly { **static void Main()** { **int[]** nums = **new** **int**[MyClass.SIZE]; | **for**(**int** i=0; i < MyClass.SIZE; i++) nums[i] = i;  **foreach**(**int** i **in** nums) **Console.Write**(i + " ");  // MyClass.SIZE = 100; *// Error!!! can't change*  }} |

* Here, ***MyClass.SIZE*** is initialized to ***10***. After that, it can be used, but not changed. To prove this, try removing the comment symbol from before the last line and then compiling the program. As you will see, an error will result.
* stackalloc: You can allocate memory from the ***stack*** by using ***stackalloc***. It can be used only when initializing local variables and has this general form:

***type\* p = stackalloc type[size]***

* Here, ***p*** is a ***pointer*** that receives the address of the memory that is large enough to hold size number of objects of type. ***stackalloc*** must be used in an UNSAFE CONTEXT.
* Normally, *memory* for *objects* is allocated from the ***heap***, which is a region of free memory. Allocating memory from the ***stack*** is the exception. Variables allocated on the ***stack*** are not garbage-collected. Rather, they exist only while the block in which they are declared is executing. The only advantage to using ***stackalloc*** is that you don’t need to worry about the variables being MOVED about by the GARBAGE COLLECTOR.
* using: In addition to the using directive discussed earlier, ***using*** has a second form that is called the ***using*** statement. It has following two general forms:

|  |  |
| --- | --- |
| **using (obj)** { */\* use obj \*/* } | **using** **(type obj = initializer)** { */\* use obj \*/* } |

* Here, ***obj*** is an expression that must evaluate to an object that implements the ***System.IDisposable*** interface. It specifies a variable that will be used inside the ***using block***.
* In the first form, the object is declared outside the ***using*** statement. In the second form, the object is declared within the ***using*** statement.
* When the block concludes, the ***Dispose()*** method (defined by the ***System.IDisposable*** interface) will be called on ***obj***. Thus, a ***using*** statement provides a means by which objects are AUTOMATICALLY DISPOSED when they are no ***longer needed***. Remember, the ***using*** statement applies only to objects that implement the ***System.IDisposable*** interface. *Following demonstrates of each form of the using statement:*

**using System; using System.IO;**

**class** UsingDemo { **static void Main()** { **StreamReader** sr = **new** StreamReader("test.txt");

**using**(sr) { /\* . . . \*/ } *// Use object inside using statement.*

**using**(**StreamReader** sr2 = **new** StreamReader("test.txt")) { /\* . . . \*/} */ \* Create StreamReader inside the using statement. \*/* }}

* The class ***StreamReader*** implements the ***IDisposable*** interface (through its base class ***TextReader***). Thus, it can be used in a ***using*** statement. When the ***using*** statement ends, ***Dispose()*** is AUTOMATICALLY CALLED on the stream variables, thus ***closing*** the ***stream***.
* Const: The ***const*** modifier is used to declare fields or local variables that ***cannot be changed***. These variables must be given initial values when they are declared. Thus, a ***const*** variable is essentially a constant. For example, **const** **int** i = 10; creates a const variable called i that has the value 10.
* volatile: The ***volatile*** modifier tells the compiler that a field’s value *may be changed* by two or more concurrently executing threads. In this situation, one thread *may not know when the field has been changed by another* thread. This is important, because the C# compiler will automatically perform certain optimizations that work only when a field is accessed by a single thread of execution.
* To prevent these optimizations from being applied to a shared field, declare it ***volatile***. This tells the compiler that it must obtain the value of this field each time it is accessed.
* ***partial*** modifier: The ***partial*** modifier has two uses. FIRST, it can be used to allow a class, structure, or interface definition to be BROKEN into two or more PIECES, with EACH PIECE RESIDING in a SEPARATE FILE. When your program is compiled, the pieces of the class, structure, or interface are UNITED, forming the complete type. SECOND, in a ***partial*** class or structure, ***partial*** can be used to allow the declaration of a method to be separate from its implementation.
* Partial Types: When used to create a ***partial type***, the ***partial*** modifier has this general form: ***partial class typename {***// ...
* Here, ***typename*** is the name of the class, structure, or interface that is being SPLIT INTO PIECES. Each part of a partial type must be modified by partial. Here is an example that divides a simple XY coordinate class into THREE SEPARATE FILES.

|  |  |
| --- | --- |
| * To use XY, all files must be included in the COMPILE. For example, assuming the ***XY*** files are called ***xy1.cs***, ***xy2.cs***, and ***xy3.cs***, and that the ***Test*** ***class*** is contained in a file called ***test.cs***, then to compile ***Test***, use following command line:   ***csc test.cs xy1.cs xy2.cs xy3.cs***   * It is legal to have PARTIAL GENERIC CLASSES. However, the type parameters of each PARTIAL DECLARATION must match the other PARTS. | * First file is: **partial class** XY { **public** XY(**int** a, **int** b) {X = a; Y = b;} } * Second file is: **partial class** XY { **public** **int** X { **get**; **set**; } } * Third file is: **partial** **class** XY { **public** **int** Y { **get**; **set**; } } * The following file demonstrates the use of ***XY***:   **using System;**  **class** Test { **static void Main**() { **XY** xy = **new** XY(1, 2);  **Console.WriteLine**(xy.X + "," + xy.Y); }} |

* Partial Methods: Within a ***partial*** **type** that is a **class** or a **structure**, you can use ***partial*** to create a partial method. A partial method has its DECLARATION in one part and its IMPLEMENTATION in another part. Partial methods were added by C# 3.0.
* The key aspect of a partial method is that the IMPLEMENTATION IS NOT REQUIRED! When the **partial method** is **not implemented** by another part of the **CLASS** or **STRUCTURE**, then **all calls to** the **partial** **method** are **silently ignored**. This makes it possible for a ***class to specify, but not require, optional functionality***. If that functionality is not implemented, then it is simply ignored.
* C#\_Example 25: Following *reworks the preceding program* that creates a partialmethodcalled ***Show()***. It is called by another method called ***ShowXY()***.

|  |  |
| --- | --- |
| * Notice that ***Show()*** is declared in one part of ***XY*** and implemented by another part. The implementation displays the values of ***X*** and ***Y***. This means that when ***Show()*** is called by ***ShowXY(),*** the call has effect and it will, indeed, display ***X*** and ***Y***. However, if you COMMENT OUT the IMPLEMENTATION of ***Show()***, then the call to ***Show()*** within ***ShowXY()*** does nothing. * PARTIAL METHODS have several RESTRICTIONS: They must be ***void***; they cannot have ***access modifiers***; they cannot be ***virtual***; and they cannot use ***out*** ***parameters***. | using System;  partial class XY { public XY(int a, int b) { X = a; Y = b; }  partial void Show(); */\** ***Declare*** *a partial method.\*/* }  partial class XY { public int X { get; set; }  */\* partial method.\*/* partial void Show() { Console.WriteLine("{0}, {1}", X, Y); } }  partial class XY { public int Y { get; set; }  public void ShowXY() { Show(); } */ \** ***Call*** *a partial method.\*/* }  class Test { static void Main() { XY xy = new XY(1, 2); xy.ShowXY(); }} |

* yield: The ***yield*** contextual keyword is used with an **iterator**, which is a **method**, **operator**, or **accessor** that ***returns the members of a set of*** ***objects, one element at a time, in sequence***. Iterators are most often used with COLLECTIONS.
* Extern: The ***extern*** keyword has two uses. First, it *indicates* that a *method* is provided by *external code*, which is usually UNMANAGED CODE. Second, it is used to create an ALIAS for an EXTERNAL ASSEMBLY. *[Luhusaher, November 15, 2020]*
* **What Next?**
* Congratulations! – Bitch!!! *If you have read and worked through all the* **26 chapters**, *then you can call yourself a*

fucking

C-C++ & Java-C# programmer.

* Of course, there are still many, many things to learn about C#, its Libraries, and Subsystems, but you now have a solid foundation upon which you can build your knowledge and expertise. Here are a few of the topics to learn more about C#:

|  |  |
| --- | --- |
| * Creating Multithreaded Applications * Using Windows Forms * Using the Collection Classes | * Networking with C# and .Net * On your own, continue to explore and Experiment with C#. * The best way of learning a programming-language is Coading!!! |

**[Terminated]**