**Introducing Collections**

When you create multiple items of the same type, regardless of whether they are integers, strings, or a custom type such as **Coffee**, you need a way of managing the items as a set. You need to be able to count the number of items in the set, add items to or remove items from the set, and iterate through the set one item at a time. To do this, you can use a *collection*.

Collections are an essential tool for managing multiple items. They are also central to developing graphical applications. Controls such as drop-down list boxes and menus are typically data-bound to collections.

**Choosing Collections**

All collection classes share various common characteristics. To manage a collection of items, you must be able to:

* Add items to the collection.
* Remove items from the collection.
* Retrieve specific items from the collection.
* Count the number of items in the collection.
* Iterate through the items in the collection, one item at a time.

Every collection class in Visual C# provides methods and properties that support these core operations. Beyond these operations, however, you will want to manage collections in different ways depending on the specific requirements of your application. Collection classes in Visual C# fall into the following broad categories:

* List classes store linear collections of items. You can think of a list class as a one-dimensional array that dynamically expands as you add items. For example, you might use a list class to maintain a list of available beverages at your coffee shop.
* Dictionary classes store a collection of key/value pairs. Each item in the collection consists of two objects—the key and the value. The value is the object you want to store and retrieve, and the key is the object that you use to index and look up the value. In most dictionary classes, the key must be unique, whereas duplicate values are perfectly acceptable. For example, you might use a dictionary class to maintain a list of coffee recipes. The key would contain the unique name of the coffee, and the value would contain the ingredients and the instructions for making the coffee.
* Queue classes represent a first in, first out collection of objects. Items are retrieved from the collection in the same order they were added. For example, you might use a queue class to process orders in a coffee shop to ensure that customers receive their drinks in turn.
* Stack classes represent a last in, first out collection of objects. The item that you added to the collection last is the first item you retrieve. For example, you might use a stack class to determine the 10 most recent visitors to your coffee shop.

When you choose a built-in collection class for a specific scenario, ask yourself the following questions:

* Do you need a list, a dictionary, a stack, or a queue?
* Will you need to sort the collection?
* How large do you expect the collection to get?
* If you are using a dictionary class, will you need to retrieve items by index as well as by key?
* Does your collection consist solely of strings?

If you can answer all of these questions, you will be able to select the Visual C# collection class that best meets your needs.

**Standard Collection Classes**

The System.Collections namespace provides a range of general-purpose collections that includes lists, dictionaries, queues, and stacks. The following table shows the most important collection classes in the System.Collections namespace:

|  |  |
| --- | --- |
| **Class** | **Description** |
| ArrayList | The ArrayList is a general-purpose list that stores a linear collection of objects. The ArrayList includes methods and properties that enable you to add items, remove items, count the number of items in the collection, and sort the collection. |
| BitArray | The BitArray is a list class that represents a collection of bits as Boolean values. The BitArray is most commonly used for bitwise operations and Boolean arithmetic, and includes methods to perform common Boolean operations such as AND, NOT, and XOR. |
| Hashtable | The Hashtable class is a general-purpose dictionary class that stores a collection of key/value pairs. The Hashtable includes methods and properties that enable you to retrieve items by key, add items, remove items, and check for particular keys and values within the collection. |
| Queue | The Queue class is a first in, first out collection of objects. The Queue includes methods to add objects to the back of the queue (Enqueue) and retrieve objects from the front of the queue (Dequeue). |
| SortedList | The SortedList class stores a collection of key/value pairs that are sorted by key. In addition to the functionality provided by the Hashtable class, the SortedList enables you to retrieve items either by key or by index. |
| Stack | The Stack class is a first in, last out collection of objects. The Stack includes methods to view the top item in the collection without removing it (Peek), add an item to the top of the stack (Push), and remove and return the item at the top of the stack (Pop). |

**Specialized Collection Classes**

The System.Collections.Specialized namespace provides collection classes that are suitable for more specialized requirements, such as specialized dictionary collections and strongly typed string collections. The following table shows the most important collection classes in the System.Collections.Specialized namespace:

|  |  |
| --- | --- |
| **Class** | **Description** |
| ListDictionary | The ListDictionary is a dictionary class that is optimized for small collections. As a general rule, if your collection includes 10 items or fewer, use a ListDictionary. If your collection is larger, use a Hashtable. |
| HybridDictionary | The HybridDictionary is a dictionary class that you can use when you cannot estimate the size of the collection. The HybridDictionary uses a ListDictionary implementation when the collection size is small, and switches to a Hashtable implementation as the collection size grows larger. |
| OrderedDictionary | The OrderedDictionary is an indexed dictionary class that enables you to retrieve items by key or by index. Note that unlike the SortedList class, items in an OrderedDictionary are not sorted by key. |
| NameValueCollection | The NameValueCollection is an indexed dictionary class in which both the key and the value are strings. The NameValueCollection will throw an error if you attempt to set a key or a value to anything other than a string. You can retrieve items by key or by index. |
| StringCollection | The StringCollection is a list class in which every item in the collection is a string. Use this class when you want to store a simple, linear collection of strings.. |
| StringDictionary | The StringDictionary is a dictionary class in which both the key and the value are strings. Unlike the NameValueCollection class, you cannot retrieve items from a StringDictionary by index. |
| BitVector32 | The BitVector32 is a struct that can represent a 32-bit value as both a bit array and an integer value. Unlike the BitArray class, which can expand indefinitely, the BitVector32 struct is a fixed 32-bit size. As a result, the BitVector32 is more efficient than the BitArray for small values. You can divide a BitVector32 instance into sections to efficiently store multiple values. |

**Using Collections**

The most commonly used list collection is the ArrayList class. The ArrayList stores items as a linear collection of objects. You can add objects of any type to an ArrayList collection, but the ArrayList represents each item in the collection as a System.Object instance. When you add an item to an ArrayList collection, the ArrayList implicitly casts, or converts, your item to the Object type. When you retrieve items from the collection, you must explicitly cast the object back to its original type.

The following example shows how to add and retrieve items from an ArrayList collection:

*Adding and Retrieving Items from an ArrayList*  
  
// Create a new ArrayList collection.  
ArrayList beverages = new ArrayList();

// Create some items to add to the collection.  
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");  
Coffee coffee2 = new Coffee(3, "Arabica", "Vietnam");  
Coffee coffee3 = new Coffee(4, "Robusta", "Indonesia");

// Add the items to the collection.  
// Items are implicitly cast to the Object type when you add them.  
beverages.Add(coffee1);  
beverages.Add(coffee2);  
beverages.Add(coffee3);

// Retrieve items from the collection.  
// Items must be explicitly cast back to their original type.  
Coffee firstCoffee = (Coffee)beverages[0];  
Coffee secondCoffee = (Coffee)beverages[1];

When you work with collections, one of your most common programming tasks will be to iterate over the collection. Essentially, this means that you retrieve each item from the collection in turn, usually to render a list of items, to evaluate each item against some criteria, or to extract specific member values from each item. To iterate over a collection, you use a **foreach** loop. The **foreach** loop exposes each item from the collection in turn, using the variable name you specify in the loop declaration.

The following example shows how to iterate over an ArrayList collection:

// Iterating Over a List Collection  
foreach(Coffee coffee in beverages)  
{  
   Console.WriteLine("Bean type: {0}", coffee.Bean);  
   Console.WriteLine("Country of origin: {0}", coffee.CountryOfOrigin);  
   Console.WriteLine("Strength (1-5): {0}", coffee.Strength);  
}  
  
**Using Dictionary Collections**  
  
Dictionary classes store collections of key/value pairs. The most commonly used dictionary class is the Hashtable. When you add an item to a Hashtable collection, you must specify a key and a value. Both the key and the value can be instances of any type, but the Hashtable implicitly casts both the key and the value to the Object type. When you retrieve values from the collection, you must explicitly cast the object back to its original type.

The following example shows how to add and retrieve items from a Hashtable collection. In this case both the key and the value are strings:

*Adding and Retrieving Items from a Hashtable*  
  
// Create a new Hashtable collection.  
Hashtable ingredients = new Hashtable();

// Add some key/value pairs to the collection.  
ingredients.Add("Café au Lait", "Coffee, Milk");  
ingredients.Add("Café Mocha", "Coffee, Milk, Chocolate");  
ingredients.Add("Cappuccino", "Coffee, Milk, Foam");  
ingredients.Add("Irish Coffee", "Coffee, Whiskey, Cream, Sugar");  
ingredients.Add("Macchiato", "Coffee, Milk, Foam");

// Check whether a key exists.  
if(ingredients.ContainsKey("Café Mocha"))  
{  
   // Retrieve the value associated with a key.  
   Console.WriteLine("The ingredients of a Café Mocha are: {0}", ingredients["Café Mocha"]);  
}

Dictionary classes, such as the Hashtable, actually contain two enumerable collections—the keys and the values. You can iterate over either of these collections. In most scenarios, however, you are likely to iterate through the key collection, for example to retrieve the value associated with each key in turn.

The following example shows how to iterate over the keys in a Hashtable collection and retrieve the value associated with each key:

// Iterating Over a Dictionary Collection  
foreach(string key in ingredients.Keys)  
{  
   // For each key in turn, retrieve the value associated with the key.    
   Console.WriteLine("The ingredients of a {0} are {1}", key, ingredients[key]);  
}

**Query Collections with Predicates, and Lambda Expressions**

Some collections in the .NET Framework do not support the use of array notation for accessing items in the collection.  These collections provide the Find method for locating items in the collection.  The Find method requires a predicate to be used as the criteria for its search.  In this case, the predicate becomes a method that will examine each item in the collection, returning a Boolean value based on the match results.  The search ends once an item is found.

Predicates are typically expressed in the form of a lambda expression.   Lambda expressions are a C# expression that returns a method.  Similar to the methods you are familiar with already, a lambda expression contains a list of parameters and a method body, but it does not contain a method name nor does it contain a return type.  The return type is inferred from the context in which the lambda expression is used.

The following is an example of a lambda expression for querying a collection of Employee objects, the collection is also shown for reference.

List<Employee> employees= new List<Employee>()  
{  
new Employee() { empID = 001, Name = "Tom", Department= "Sales"},  
new Employee() { empID = 024, Name = "Joan", Department= "HR"},  
new Employee() { empID = 023, Name = "Fred", Department= "Accounting" },  
new Employee() { empID = 040, Name = "Mike", Department= "Sales" },  
};

// Find the member of the list that has an employee id of 023  
Employee match = employees.Find((Employee p) => { return p.empID == 023; });  
Console.WriteLine("empID: {0}\nName: {1}\nDepartment: {2}", match.empID, match.Name, match.Department);

Here is the output generated by this code:  
empID: 023  
Name: Fred  
Department: Accounting

The lambda expression in the above code is (Employee p) => { return p.empID == 023; }.