**Creating and Using Generic Classes**

Generic classes work by including a type parameter, T, in the class or interface declaration. You do not need to specify the type of T until you instantiate the class. To create a generic class, you need to:

* Add the type parameter T in angle brackets after the class name.
* Use the type parameter T in place of type names in your class members.

The following example shows how to create a generic class:

// Creating a Generic Class  
public class CustomList<T>  
{  
   public T this[int index] { get; set; }  
   public void Add(T item)  
   {  
      // Method logic goes here.  
   }  
   public void Remove(T item)  
   {  
      // Method logic goes here.  
   }   
}

When you create an instance of your generic class, you specify the type you want to supply as a type parameter. For example, if you want to use your custom list to store objects of type Coffee, you would supply Coffee as the type parameter.

The following example shows how to instantiate a generic class:

//Instantiating a Generic Class  
CustomList<Coffee> clc = new CustomList<Coffee>;  
Coffee coffee1 = new Coffee();  
Coffee coffee2 = new Coffee();  
clc.Add(coffee1);  
clc.Add(coffee2);  
Coffee firstCoffee = clc[0];

When you instantiate a class, every instance of T within the class is effectively replaced with the type parameter you supply. For example, if you instantiate the CustomList class with a type parameter of Coffee:

* The Add method will only accept an argument of type Coffee.
* The Remove method will only accept an argument of type Coffee.
* The indexer will always provide a return value of type Coffee.

**Advantages of Generics**

The use of generic classes, particularly for collections, offers three main advantages over non-generic approaches: type safety, no casting, and no boxing and unboxing.

**Type Safety**

Consider an example where you use an ArrayList to store a collection of Coffee objects. You can add objects of any type to an ArrayList. Suppose a developer adds an object of type Tea to the collection. The code will build without complaint. However, a runtime exception will occur if the Sort method is called, because the collection is unable to compare objects of different types. Furthermore, when you retrieve an object from the collection, you must cast the object to the correct type. If you attempt to cast the object to the wrong type, an invalid cast runtime exception will occur.

The following example shows the type safety limitations of the ArrayList approach:

// Type Safety Limitations for Non-Generic Collections  
var coffee1 = new Coffee();  
var coffee2 = new Coffee();  
var tea1 = new Tea();  
var arrayList1 = new ArrayList();  
arrayList1.Add(coffee1);  
arrayList1.Add(coffee2);  
arrayList1.Add(tea1);  
// The Sort method throws a runtime exception because the collection is not homogenous.  
arrayList1.Sort();  
// The cast throws a runtime exception because you cannot cast a Tea instance to a Coffee instance.  
Coffee coffee3 = (Coffee)arrayList1[2];

As an alternative to the ArrayList, suppose you use a generic List<T> to store a collection of Coffee objects. When you instantiate the list, you provide a type argument of Coffee. In this case, your list is guaranteed to be homogenous, because your code will not build if you attempt to add an object of any other type. The Sort method will work because your collection is homogenous. Finally, the indexer returns objects of type Coffee, rather than System.Object, so there is no risk of invalid cast exceptions.

The following example shows an alternative to the ArrayList approach using the generic List<T> class:

// Type Safety in Generic Collections  
var coffee1 = new Coffee();  
var coffee2 = new Coffee();  
var tea1 = new Tea();  
var genericList1 = new List<Coffee>();  
genericList1.Add(coffee1);  
genericList1.Add(coffee2);  
// This line causes a build error, as the argument is not of type Coffee.  
genericList1.Add(tea1);  
// The Sort method will work because the collection is guaranteed to be homogenous.  
arrayList1.Sort();  
// The indexer returns objects of type Coffee, so there is no need to cast the return value.  
Coffee coffee3 = genericList[1];

**No Casting**

Casting is a computationally expensive process. When you add items to an ArrayList, your items are implicitly cast to the System.Object type. When you retrieve items from an ArrayList, you must explicitly cast them back to their original type. Using generics to add and retrieve items without casting improves the performance of your application.

**No Boxing and Unboxing**

If you want to store value types in an ArrayList, the items must be boxed when they are added to the collection and unboxed when they are retrieved. Boxing and unboxing incurs a large computational cost and can significantly slow your applications, especially when you iterate over large collections. By contrast, you can add value types to generic lists without boxing and unboxing the value.  
The following example shows the difference between generic and non-generic collections with regard to boxing and unboxing:

// Boxing and Unboxing: Generic vs. Non-Generic Collections  
int number1 = 1;  
var arrayList1 = new ArrayList();  
// This statement boxes the Int32 value as a System.Object.  
arrayList1.Add(number1);  
// This statement unboxes the Int32 value.  
int number2 = (int)arrayList1[0];  
var genericList1 = new List<Int32>();  
//This statement adds an Int32 value without boxing.  
genericList1.Add(number1);  
//This statement retrieves the Int32 value without unboxing.  
int number3 = genericList1[0];

**Constraining Generics**

In some cases, you may need to restrict the types that developers can supply as arguments when they instantiate your generic class. The nature of these constraints will depend on the logic you implement in your generic class. For example, if a collection class uses a property named AverageRating to sort the items in a collection, you would need to constrain the type parameter to classes that include the AverageRating property. Suppose the AverageRating property is defined by the IBeverage interface. To implement this restriction, you would constrain the type parameter to classes that implement the IBeverage interface by using the where keyword.

The following example shows how to constrain a type parameter to classes that implement a particular interface:

// Constraining Type Parameters by Interface  
public class CustomList<T> where T : IBeverage  
{  
}

You can apply the following six types of constraint to type parameters:

|  |  |
| --- | --- |
| **Constraint** | **Description** |
| where T : <name of interface> | The type argument must be, or implement, the specified interface. |
| where T : <name of base class> | The type argument must be, or derive from, the specified class |
| where T : U | The type argument must be, or derive from, the supplied type argument U |
| where T : new() | The type argument must have a public default constructor |
| where T : struct | The type argument must be a value type |
| where T : class | The type argument must be a reference type |

You can apply the following six types of constraint to type parameters:

// Apply Multiple Type Constraints  
public class CustomList<T> where T : IBeverage, IComparable<T>, new()  
{  
}

**Using Generic List Collections**

One of the most common and important uses of generics is in collection classes. Generic collections fall into two broad categories: generic list collections and generic dictionary collections. A generic list stores a collection of objects of type T.

**The List<T> Class**

The List<T> class provides a strongly-typed alternative to the ArrayList class. Like the ArrayList class, the List<T> class includes methods to:

* Add an item.
* Remove an item.
* Insert an item at a specified index.
* Sort the items in the collection by using the default comparer or a specified comparer.
* Reorder all or part of the collection.

The following example shows how to use the List<T> class.

// Using the List<T> Class  
string s1 = "Latte";  
string s2 = "Espresso";  
string s3 = "Americano";  
string s4 = "Cappuccino";  
string s5 = "Mocha";  
// Add the items to a strongly-typed collection.  
var coffeeBeverages = new List<String>();  
coffeeBeverages.Add(s1);  
coffeeBeverages.Add(s2);  
coffeeBeverages.Add(s3);  
coffeeBeverages.Add(s4);  
coffeeBeverages.Add(s5);  
// Sort the items using the default comparer.   
// For objects of type String, the default comparer sorts the items alphabetically.  
coffeeBeverages.Sort();  
// Write the collection to a console window.  
foreach(String coffeeBeverage in coffeeBeverages)  
{  
   Console.WriteLine(coffeeBeverage);  
}

**Other Generic List Classes**

The System.Collections.Generic namespace also includes various generic collections that provide more specialized functionality:

* The LinkedList<T> class provides a generic collection in which each item is linked to the previous item in the collection and the next item in the collection. Each item in the collection is represented by a LinkedListNode<T> object, which contains a value of type T, a reference to the parent LinkedList<T> instance, a reference to the previous item in the collection, and a reference to the next item in the collection.
* The Queue<T> class represents a strongly typed first in, last out collection of objects.
* The Stack<T> class represents a strongly typed last in, last out collection of objects.

**Using Generic Dictionary Collections**  
  
Dictionary classes store collections of key value pairs. The value is the object you want to store, and the key is the object you use to index and retrieve the value. For example, you might use a dictionary class to store coffee recipes, where the key is the name of the coffee and the value is the recipe for that coffee. In the case of generic dictionaries, both the key and the value are strongly typed.   
The Dictionary<TKey, TValue> Class  
The Dictionary<TKey, TValue> class provides a general purpose, strongly typed dictionary class. You can add duplicate values to the collection, but the keys must be unique. The class will throw an ArgumentException if you attempt to add a key that already exists in the dictionary.

The following example shows how to use the Dictionary<TKey, TValue> class:

// Using the Dictionary<TKey, TValue> Class  
// Create a new dictionary of strings with string keys.  
var coffeeCodes = new Dictionary<String, String>();  
// Add some entries to the dictionary.  
coffeeCodes.Add("CAL", "Café Au Lait");  
coffeeCodes.Add("CSM", "Cinammon Spice Mocha");  
coffeeCodes.Add("ER", "Espresso Romano");  
coffeeCodes.Add("RM", "Raspberry Mocha");  
coffeeCodes.Add("IC", "Iced Coffee");  
// This statement would result in an ArgumentException because the key already exists.  
// coffeeCodes.Add("IC", "Instant Coffee");  
// To retrieve the value associated with a key, you can use the indexer.  
// This will throw a KeyNotFoundException if the key does not exist.  
Console.WriteLine("The value associated with the key \"CAL\" is {0}", coffeeCodes["CAL"]);  
// Alternatively, you can use the TryGetValue method.  
// This returns true if the key exists and false if the key does not exist.  
string csmValue = "";  
if(coffeeCodes.TryGetValue("CSM", out csmValue))  
{  
   Console.WriteLine("The value associated with the key \"CSM\" is {0}", csmValue);  
}  
else  
{  
   Console.WriteLine("The key \"CSM\" was not found");  
}  
// You can also use the indexer to change the value associated with a key.  
coffeeCodes["IC"] = "Instant Coffee";

**Other Generic Dictionary Classes**

The SortedList<TKey, TValue> and SortedDictionary<TKey, TValue> classes both provide generic dictionaries in which the entries are sorted by key. The difference between these classes is in the underlying implementation:

* The SortedList generic class uses less memory than the SortedDictionary generic class.
* The SortedDictionary class is faster and more efficient at inserting and removing unsorted data.
* **Using Collection Interfaces**
* The System.Collections.Generic namespace provides a range of generic collections to suit various scenarios. However, there will be circumstances when you will want to create your own generic collection classes in order to provide more specialized functionality. For example, you might want to store data in a tree structure or create a circular linked list.
* Where should you start when you want to create a custom collection class? All collections have certain things in common. For example, you will typically want to be able to enumerate the items in the collection by using a foreach loop, and you will need methods to add items, remove items, and clear the list. The picture on the slide shows that the .NET Framework provides a hierarchical set of interfaces that define the characteristics and behaviors of collections. These interfaces build on one another to define progressively more specific functionality.
* **The IEnumerable and IEnumerable<T> Interfaces**
* If you want to be able to use a foreach loop to enumerate over the items in your custom generic collection, you must implement the IEnumerable<T> interface. The IEnumerable<T> method defines a single method named GetEnumerator(). This method must return an object of type IEnumerator<T>. The foreach statement relies on this enumerator object to iterate through the collection.
* The IEnumerable<T> interface inherits from the IEnumerable interface, which also defines a single method named GetEnumerator(). When an interface inherits from another interface, it exposes all the members of the parent interface. In other words, if you implement IEnumerable<T>, you also need to implement IEnumerable.
* **The ICollection<T> Interface**
* The ICollection<T> interface defines the basic functionality that is common to all generic collections. The interface inherits from IEnumerable<T>, which means that if you want to implement ICollection<T>, you must also implement the members defined by IEnumerable<T> and IEnumerable.
* The ICollection<T> interface defines the following methods:

|  |  |
| --- | --- |
| **Name** | **Description** |
| Add | Adds an item of type T to the collection. |
| Clear | Removes all items from the collection. |
| Contains | Indicates whether the collection contains a specific value. |
| CopyTo | Copies the items in the collection to an array. |
| Remove | Removes a specific object from the collection. |

* The ICollection<T> interface defines the following properties:

|  |  |
| --- | --- |
| **Name** | **Description** |
| Count | Gets the number of items in the collection. |
| IsReadOnly | Indicates whether the collection is read-only. |

* **The IList<T> Interface**
* The IList<T> interface defines the core functionality for generic list classes. You should implement this interface if you are defining a linear collection of single values. In addition to the members it inherits from ICollection<T>, the IList<T> interface defines methods and properties that enable you to use indexers to work with the items in the collection. For example, if you create a list named myList, you can use myList[0] to access the first item in the collection.
* The IList<T> interface defines the following methods:

|  |  |
| --- | --- |
| **Name** | **Description** |
| Insert | Inserts an item into the collection at the specified index. |
| RemoveAt | Removes the item at the specified index from the collection. |

* The IList<T> interface defines the following properties:

|  |  |
| --- | --- |
| **Name** | **Description** |
| IndexOf | Determines the position of a specified item in the collection. |

* The IDictionary<TKey, TValue> Interface  
  The IDictionary<TKey, TValue> interface defines the core functionality for generic dictionary classes. You should implement this interface if you are defining a collection of key-value pairs. In addition to the members it inherits from ICollection<T>, the IDictionary<T> interface defines methods and properties that are specific to working with key-value pairs.  
  The IDictionary<TKey, TValue> interface defines the following methods:

|  |  |
| --- | --- |
| **Name** | **Description** |
| Add | Adds an item with the specified key and value to the collection. |
| ContainsKey | Indicates whether the collection includes a key-value pair with the specified key. |
| GetEnumerator | Returns an enumerator of KeyValuePair<TKey, TValue> objects. |
| Remove | Removes the item with the specified key from the collection. |
| TryGetValue | Attempts to set the value of an output parameter to the value associated with a specified key. If the key exists, the method returns true. If the key does not exist, the method returns false and the output parameter is unchanged. |

* The IDictionary<TKey, TValue> interface defines the following properties:

|  |  |
| --- | --- |
| **Name** | **Description** |
| Item | Gets or sets the value of an item in the collection, based on a specified key. This property enables you to use indexer notation, for example myDictionary[myKey] = myValue. |
| Keys | Returns the keys in the collection as an ICollection<T> instance. |
| Values | Returns the values in the collection as an ICollection<T> instance. |

**Creating Enumerable Collections**

To enumerate over a collection, you typically use a foreach loop. The foreach loop exposes each item in the collection in turn, in an order that is appropriate to the collection. The foreach statement masks some of the complexities of enumeration. For the foreach statement to work, a generic collection class must implement the IEnumerable<T> interface. This interface exposes a method, GetEnumerator, which must return an IEnumerator<T> type.

The IEnumerator<T> Interface  
The IEnumerator<T> interface defines the functionality that all enumerators must implement.

The IEnumerator<T> interface defines the following methods:

|  |  |
| --- | --- |
| **Name** | **Description** |
| MoveNext | Advanced the enumerator to the next item in the collection. |
| Reset | Sets the enumerator to its starting position, which is before the first item in the collection. |

The IEnumerator<T> interface defines the following properties:

|  |  |
| --- | --- |
| **Name** | **Description** |
| Current | Gets the item that the enumerator is pointing to. |

An enumerator is essentially a pointer to the items in the collection. The starting point for the pointer is before the first item. When you call the MoveNext method, the pointer advances to the next element in the collection. The MoveNext method returns true if the enumerator was able to advance one position, or false if it has reached the end of the collection. At any point during the enumeration, the Current property returns the item to which the enumerator is currently pointing.

When you create an enumerator, you must define:

* Which item the enumerator should treat as the first item in the collection.
* In what order the enumerator should move through the items in the collection.

**The IEnumerable<T> Interface**

The IEnumerable<T> interface defines a single method named GetEnumerator. This returns an IEnumerator<T> instance.  
The GetEnumerator method returns the default enumerator for your collection class. This is the enumerator that a foreach loop will use, unless you specify an alternative. However, you can create additional methods to expose alternative enumerators.  
The following example shows a custom collection class that implements a default enumerator, together with an alternative enumerator that enumerates the collection in reverse order:

// Default and Alternative Enumerators  
class CustomCollection<T> : IEnumerable<T>  
{  
   public IEnumerator<T> Backwards()  
   {  
      // This method returns an alternative enumerator.  
      // The implementation details are not shown.  
   }  
   #region IEnumerable<T> Members  
   public IEnumerator<T> GetEnumerator()  
   {  
      // This method returns the default enumerator.  
      // The implementation details are not shown.  
   }  
   #endregion  
   #region IEnumerable Members  
   IEnumerator IEnumerable.GetEnumerator()  
   {  
      // This method is required because IEnumerable<T> inherits from IEnumerable  
      throw new NotImplementedException();  
   }  
   #endregion  
}

The following example shows how you can use a default enumerator or an alternative enumerator to iterate through a collection:

// Enumerating a Collection  
CustomCollection<Int32> numbers = new CustomCollection<Int32>();  
// Add some items to the collection.  
// Use the default enumerator to iterate through the collection:  
foreach (int number in numbers)  
{  
   // …  
}  
// Use the alternative enumerator to iterate through the collection:  
foreach(int number in numbers.Backwards())  
{  
   // …  
}

**Implementing the Enumerator**

You can provide an enumerator by creating a custom class that implements the IEnumerator<T> interface. However, if your custom collection class uses an underlying enumerable type to store data, you can use an iterator to implement the IEnumerable<T> interface without actually providing an IEnumerator<T> implementation. The best way to understand iterators is to start with a simple example.

The following example shows how you can use an iterator to implement an enumerator:

// Implementing an Enumerator by Using an Iterator  
using System;  
using System.Collections;  
using System.Collections.Generic;  
class BasicCollection<T> : IEnumerable<T>  
{  
    private List<T> data = new List<T>();  
    public void FillList(params T [] items)  
    {  
        foreach (var datum in items)  
          data.Add(datum);  
    }  
    IEnumerator<T> IEnumerable<T>.GetEnumerator()  
    {  
        foreach (var datum in data)  
        {  
            yield return datum;  
        }  
    }  
    IEnumerator IEnumerable.GetEnumerator()  
    {  
        throw new NotImplementedException();  
    }  
}

The example shows a custom generic collection class that uses a List<T> instance to store data. The List<T> instance is populated by the FillList method. When the GetEnumerator method is called, a foreach loop enumerates the underlying collection. Within the foreach loop, a yield return statement is used to return each item in the collection. It is this yield return statement that defines the iterator—essentially, the yield return statement pauses execution to return the current item to the caller before the next element in the sequence is retrieved. In this way, although the GetEnumerator method does not appear to return an IEnumerator type, the compiler is able to build an enumerator from the iteration logic that you provided.