Generics Eliminate Duplicate Code

Prior to .NET 2.0 when you needed a single method to work with different data types the only way to accomplish this was to pass an ‘object’ data type to that method. Working with the object data type introduces performance problems and bugs that can occur at runtime. The alternative is to create a new method for each data type that you wished to work with. This introduces a maintenance nightmare and leads to a larger API for programmers to learn. An example of using individual methods is shown in the code snippet that follows. Notice the calls to two different “ConvertTo” methods; ConvertToInt and ConvertToDateTime. The only difference between these two methods is the data types being passed in as parameters.

```
private void HardCodedNonGenericsSample()
{
  object value = "1";

  int i = ConvertToInt(value, default(int));

  value = "1/1/2014";
  DateTime dt = ConvertToDateTime(value, default(DateTime));
}
```

The ConvertToInt method shown in the following code snippet accepts two ‘object’ parameters and returns an ‘int’ data type.

```
public int ConvertToInt(object value, object defaultValue)
{
  if (value == null || value.Equals(DBNull.Value))
    return Convert.ToInt32(defaultValue);
  else
    return Convert.ToInt32(value);
}
```

Now look at the ConvertToDateTime method shown below. It is almost the exact same code except the return value is different and the use of the Convert.ToDateTime method instead of the Convert.ToInt32 method.

```
public DateTime ConvertToDateTime(object value,
 object defaultValue)
{
  if (value == null || value.Equals(DBNull.Value))
    return Convert.ToDateTime(defaultValue);
  else
    return Convert.ToDateTime(value);
}
```

Create One Generic Method

The two methods shown above can be rewritten in one method by using generics. To convert the above two methods into one you simply look at the data types in the two methods that are different. You substitute these differences with a “T” which stands for type parameter. The result is shown in the following code snippet:

```
public T ConvertTo<T>(object value, object defaultValue)
{
  if (value == null || value.Equals(DBNull.Value))
    return (T)Convert.ChangeType(defaultValue, typeof(T));
  else
    return (T)Convert.ChangeType(value, typeof(T));
}
```

The code “public T” means you have a public method that passes back the type specified in the <T> that comes after the method name. For the return type you cast either the ‘defaultValue’ or the ‘value’ to the type that was passed in.

To use this new ConvertTo method you pass in the data type you are converting into with a less than sign and a greater than sign as shown in the following code snippet:

```
private void HardCodedGenericsSample()
{
  object value = "1";

  int i = ConvertTo<int>(value, default(int));

  value = "1/1/2014";
  DateTime dt = ConvertTo<DateTime>(value, default(DateTime));
}
```

Generic Lists

Prior to .NET 2.0 you were required to create your own collection classes to provide type-safety. Type safety means you create a class that only allows you to pass in one type of object. For example, you may have a collection of string, int, or Product objects. You cannot pass an int to a string collection or a Product object to an int collection. To create a type-safe collection you inherit from the CollectionBase class and override many properties and methods. Listing 2 shows some of the code you are required to write for each unique collection class you wish to create. As you can see, this is quite a bit of code.

```
public class Int16Collection : CollectionBase
{
  public Int16 this[int index]
  {
    get
    {
      return ((Int16)List[index]);
    }
    set
    {
      List[index] = value;
    }
  }

  public int Add(Int16 value)
  {
    return (List.Add(value));
  }

  public int IndexOf(Int16 value)
  {
    return (List.IndexOf(value));
  }

  public void Insert(int index, Int16 value)
  {
    List.Insert(index, value);
  }

  public void Remove(Int16 value)
  {
    List.Remove(value);
  }

  public bool Contains(Int16 value)
  {
    return (List.Contains(value));
  }

  // Additional methods...
}
```

Listing 1: An example of using the old CollectionBase class.

Instead of writing all the code shown in Listing 2, you use one of the Generic collection classes instead. For example, you can replace the code in Listing 2 with just the following three lines of code!

```
public class IntCollection : List<int>
{
}
```

The class **IntCollection** class created in the previous code snippet is type-safe and will only accept an int data type. You cannot add a string or a decimal type to this collection. You get all of the same features you get with CollectionBase such as the ability to add, remove, insert and check to see if a value is contained within the collection. But you do not have to write all of the code for it.

The generic List<T> class is just one example of the many list classes available to you in the **System.Collections.Generic** namespace. Other examples are Dictionary<TKey, TValue> which allows you to store key/value pairs generically. You also have stacks, queues and linked lists implemented using generics. All of these save a ton of code and a ton of time.

Summary

Generics have been a great addition to .NET for many years now. Take advantage of these great constructs to cut down the amount of code you have to write.