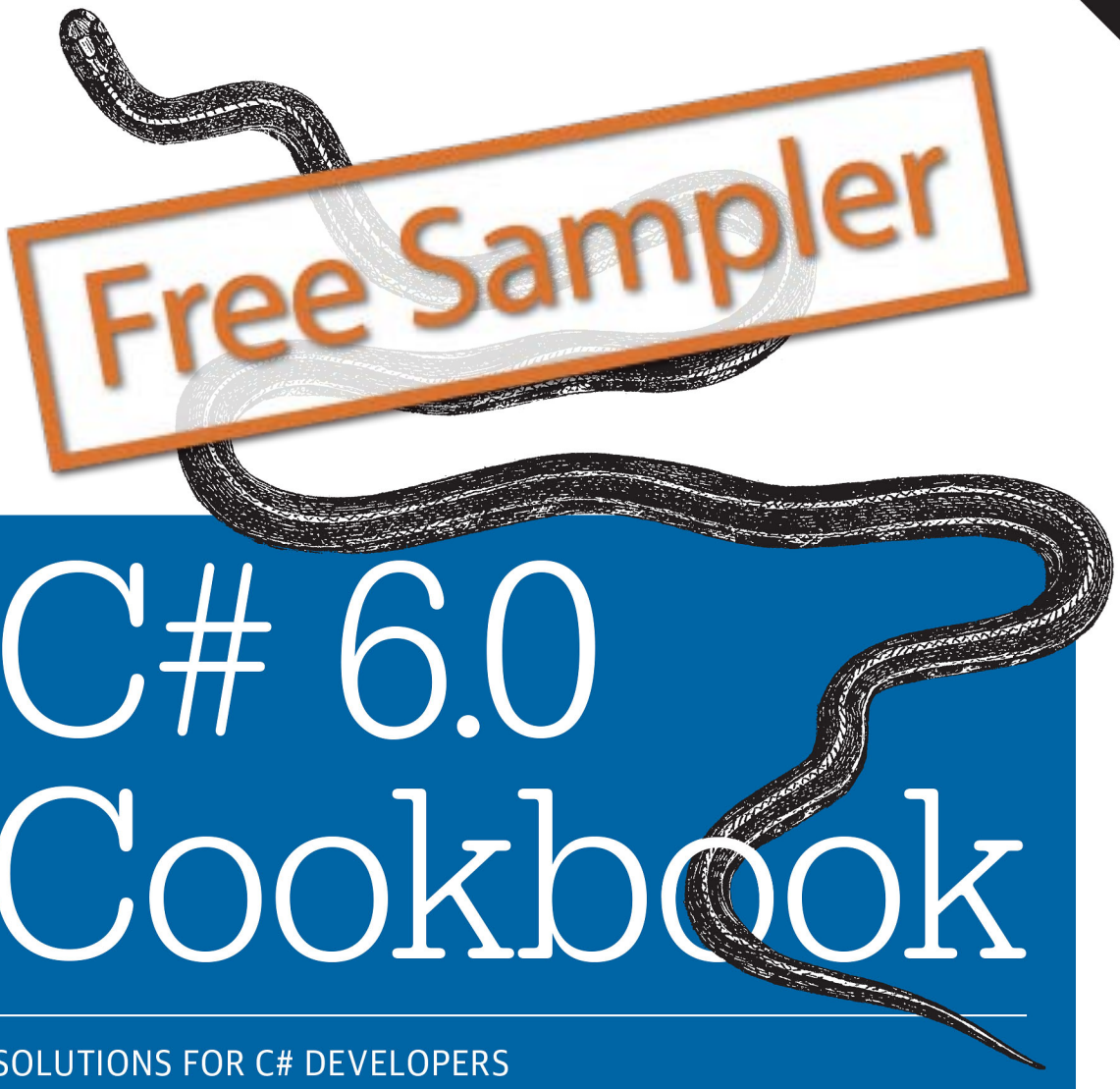
O'REILLY®

4th Edition



C# 6.0 Cookbook

SOLUTIONS FOR C# DEVELOPERS

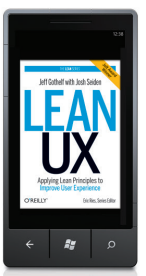
Jay Hilyard & Stephen Teilhet

O'Reilly ebooks.

Your bookshelf on your devices.

PDF

Mobi



ePub

DAISY

When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in four DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O'Reilly ebooks through the iBookstore, the Android Marketplace, and Amazon.com.

O'REILLY®

C# 6.0 Cookbook
by Jay Hilyard and Stephen Teilhet

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian MacDonald
Production Editor: Nicholas Adams
Copyeditor: Rachel Monaghan
Proofreader: Kim Cofer

Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

January 2004: First Edition
January 2006: Second Edition
December 2007: Third Edition
October 2015: Fourth Edition

Revision History for the Fourth Edition
2015-09-28: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491921463* for release details.

978-1-4919-2146-3

[LSI]

Table of Contents

CHAPTER 1

Classes and Generics

1.0 Introduction

The recipes in this chapter cover the foundation of the C# language. Topics include classes and structures, how they are used, how they are different, and when you would use one over the other. Building on this, we will construct classes that have inherent functionality such as being sortable, searchable, disposable, and cloneable. In addition, we will dive into topics such as union types, field initialization, lambdas, partial methods, single and multicast delegates, closures, functors, and more. This chapter also contains a recipe on parsing command-line parameters, which is always a favorite.

Before diving into the recipes, let's review some key information about the object-oriented capabilities of classes, structures, and generics. Classes are much more flexible than structures. Like classes, structures can implement interfaces, but unlike classes, they cannot inherit from a class or a structure. This limitation precludes creating structure hierarchies, as you can do with classes. Polymorphism, as implemented through an abstract base class, is also prohibited when you are using a structure, since a structure cannot inherit from another class with the exception of boxing to `Object`, `ValueType`, or `Enum`.

Structures, like any other value type, implicitly inherit from `System.ValueType`. At first glance, a structure is similar to a class, but it is actually very different. Knowing when to use a structure over a class will help you tremendously when you're designing an application. Using a structure incorrectly can result in inefficient and hard-to-modify code.

Structures have two performance advantages over reference types. First, if a structure is allocated on the stack (i.e., it is not contained within a reference type), access to the structure and its data is somewhat faster than access to a reference type on the heap.

Reference-type objects must follow their reference onto the heap in order to get at their data. However, this performance advantage pales in comparison to the second performance advantage of structures—namely, that cleaning up the memory allocated to a structure on the stack requires a simple change of the address to which the stack pointer points, which is done at the return of a method call. This call is extremely fast compared to allowing the garbage collector to automatically clean up reference types for you in the managed heap; however, the cost of the garbage collector is deferred so that it's not immediately noticeable.

The performance of structures falls short in comparison to that of classes when they are passed by value to other methods. Because they reside on the stack, a structure and its data have to be copied to a new local variable (the method's parameter that is used to receive the structure) when it is passed by value to a method. This copying takes more time than passing a method a single reference to an object, unless the structure is the same size as or smaller than the machine's pointer size; thus, a structure with a size of 32 bits is just as cheap to pass as a reference (which happens to be the size of a pointer) on a 32-bit machine. Keep this in mind when choosing between a class and a structure. While creating, accessing, and destroying a class's object may take longer, it also might not balance the performance hit when a structure is passed by value a large number of times to one or more methods. Keeping the size of the structure small minimizes the performance hit of passing it around by value.

Use a class if:

- Its identity is important. Structures get copied implicitly when being passed by value into a method.
- It will have a large memory footprint.
- Its fields need initializers.
- You need to inherit from a base class.
- You need polymorphic behavior; that is, you need to implement an abstract base class from which you will create several similar classes that inherit from this abstract base class. (Note that polymorphism can be implemented via interfaces as well, but it is usually not a good idea to place an interface on a value type, since, if the structure is converted to the interface type, you will incur a performance penalty from the boxing operation.)

Use a structure if:

- It will act like a primitive type (`int`, `long`, `byte`, etc.).
- It must have a small memory footprint.
- You are calling a `P/Invoke` method that requires a structure to be passed in by value. *Platform Invoke*, or P/Invoke for short, allows managed code to call out to

an unmanaged method exposed from within a DLL. Many times, an unmanaged DLL method requires a structure to be passed in to it; using a structure is an efficient method of doing this and is the only way if the structure is being passed by value.

- You need to reduce the impact of garbage collection on application performance.
- Its fields need to be initialized only to their default values. This value would be `zero` for numeric types, `false` for Boolean types, and `null` for reference types. Note that in C# 6.0 structs can have a default constructor that can be used to initialize the struct's fields to nondefault values.
- You do not need to inherit from a base class (other than `ValueType`, from which all structs inherit).
- You do not need polymorphic behavior.

Structures can also cause degradation in performance when they are passed to methods that require an object, such as any of the nongeneric collection types in the Framework Class Library (FCL). Passing a structure (or any simple type, for that matter) into a method requiring an object causes the structure to be boxed. *Boxing* is wrapping a value type in an object. This operation is time-consuming and may degrade performance.

Finally, adding generics to this mix allows you to write type-safe and efficient collection- and pattern-based code. Generics add quite a bit of programming power, but with that power comes the responsibility to use it correctly. If you are considering converting your `ArrayList`, `Queue`, `Stack`, and `Hashtable` objects to use their generic counterparts, consider reading Recipes 1.9 and 1.10. As you will read, the conversion is not always simple and easy, and there are reasons why you might not want to do this conversion at all.

1.1 Creating Union-Type Structures

Problem

You need to create a data type that behaves like a union type in C++. A union type is useful mainly in interop scenarios in which the unmanaged code accepts and/or returns a union type; we suggest that you do not use it in other situations.

Solution

Use a structure and mark it with the `StructLayout` attribute (specifying the `LayoutKind.Explicit` layout kind in the constructor). In addition, mark each field in the structure with the `FieldOffset` attribute. The following structure defines a union in which a single signed numeric value can be stored:

```
using System.Runtime.InteropServices;
[StructLayoutAttribute(LayoutKind.Explicit)]
struct SignedNumber
{
   [FieldOffsetAttribute(0)]
   public sbyte Num1;
   [FieldOffsetAttribute(0)]
   public short Num2;
   [FieldOffsetAttribute(0)]
   public int Num3;
   [FieldOffsetAttribute(0)]
   public long Num4;
   [FieldOffsetAttribute(0)]
   public float Num5;
   [FieldOffsetAttribute(0)]
   public double Num6;
}
```

The next structure is similar to the `SignedNumber` structure, except that it can contain a `String` type in addition to the signed numeric value:

```
[StructLayoutAttribute(LayoutKind.Explicit)]
struct SignedNumberWithText
{
   [FieldOffsetAttribute(0)]
   public sbyte Num1;
   [FieldOffsetAttribute(0)]
   public short Num2;
   [FieldOffsetAttribute(0)]
   public int Num3;
   [FieldOffsetAttribute(0)]
   public long Num4;
   [FieldOffsetAttribute(0)]
   public float Num5;
   [FieldOffsetAttribute(0)]
   public double Num6;
   [FieldOffsetAttribute(16)]
   public string Text1;
}
```

Discussion

Unions are structures usually found in C++ code; however, there is a way to duplicate that type of structure using a C# structure data type. A *union* is a structure that accepts more than one type at a specific location in memory for that structure. For example, the `SignedNumber` structure is a union-type structure built using a C# structure. This structure accepts any type of signed numeric type (`sbyte`, `int`, `long`, etc.), but it accepts this numeric type at only one location, or offset, within the structure.

Since `StructLayoutAttribute` can be applied to both structures and classes, you can also use a class when creating a union data type.

Notice the `FieldOffsetAttribute` has the value `0` passed to its constructor. This denotes that this field will be offset by zero bytes from the beginning of the structure. This attribute is used in tandem with the `StructLayoutAttribute` to manually enforce where the fields in this structure will start (that is, the offset from the beginning of this structure in memory where each field will start). The `FieldOffsetAttribute` can be used only with a `StructLayoutAttribute` set to `LayoutKind.Explicit`. In addition, it cannot be used on static members within this structure.

Unions can become problematic, since several types are essentially laid on top of one another. The biggest problem is extracting the correct data type from a union structure. Consider what happens if you choose to store the `long` numeric value `long.MaxValue` in the `SignedNumber` structure. Later, you might accidentally attempt to extract a `byte` data type value from this same structure. In doing so, you will get back only the first byte of the long value.

Another problem is starting fields at the correct offset. The `SignedNumberWithText` union overlays numerous signed numeric data types at the zeroth offset. The last field in this structure is laid out at the 16th byte offset from the beginning of this structure in memory. If you accidentally overlay the string field `Text1` on top of any of the other signed numeric data types, you will get an exception at runtime. The basic rule is that you are allowed to overlay a value type on another value type, but you cannot overlay a reference type over a value type. If the `Text1` field is marked with the following attribute:

```
[FieldOffsetAttribute(14)]
```

this exception is thrown at runtime (note that the compiler does not catch this problem):

```
System.TypeLoadException: Could not load type 'SignedNumberWithText' from
assembly 'CSharpRecipes, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=fe85c3941fbcc4c5' because it contains an object field at
offset 14 that is incorrectly aligned or overlapped by a non-object field.
```

It is imperative to get the offsets correct when you're using complex unions in C#.

See Also

The "StructLayoutAttribute Class" topic in the MSDN documentation.

1.2 Making a Type Sortable

Problem

You have a data type that will be stored as an element in a `List<T>` or a `SortedList<K,V>`. You would like to use the `List<T>.Sort` method or the internal sorting mechanism of `SortedList<K,V>` to allow custom sorting of your data types in the array. In addition, you may need to use this type in a `SortedList` collection.

Solution

Example 1-1 demonstrates how to implement the `IComparable<T>` interface. The `Square` class shown in Example 1-1 implements this interface in such a way that the `List<T>` and `SortedList<K,V>` collections can sort and search for these `Square` objects.

Example 1-1. Making a type sortable by implementing IComparable<T>

```
public class Square : IComparable<Square>
{
    public Square(){}

    public Square(int height, int width)
    {
        this.Height = height;
        this.Width = width;
    }

    public int Height { get; set; }

    public int Width { get; set; }

    public int CompareTo(object obj)
    {
        Square square = obj as Square;
        if (square != null)
            return CompareTo(square);
        throw
          new ArgumentException(
              "Both objects being compared must be of type Square.");
    }

    public override string ToString()=>
                          ($"Height: {this.Height}     Width: {this.Width}");

    public override bool Equals(object obj)
    {
        if (obj == null)
```

```
            return false;

        Square square = obj as Square;
        if(square != null)
           return this.Height == square.Height;
        return false;
    }

    public override int GetHashCode()
    {
        return this.Height.GetHashCode() | this.Width.GetHashCode();
    }

    public static bool operator ==(Square x, Square y) => x.Equals(y);
    public static bool operator !=(Square x, Square y) => !(x == y);
    public static bool operator <(Square x, Square y) => (x.CompareTo(y) < 0);
    public static bool operator >(Square x, Square y) => (x.CompareTo(y) > 0);

    public int CompareTo(Square other)
    {
        long area1 = this.Height * this.Width;
        long area2 = other.Height * other.Width;

        if (area1 == area2)
            return 0;
        else if (area1 > area2)
            return 1;
        else if (area1 < area2)
            return -1;
        else
            return -1;
    }
}
```

Discussion

By implementing the `IComparable<T>` interface on your class (or structure), you can take advantage of the sorting routines of the `List<T>` and `SortedList<K,V>` classes. The algorithms for sorting are built into these classes; all you have to do is tell them how to sort your classes via the code you implement in the `IComparable<T>.CompareTo` method.

When you sort a list of `Square` objects by calling the `List<Square>.Sort` method, the list is sorted via the `IComparable<Square>` interface of the `Square` objects. The `Add` method of the `SortedList<K,V>` class uses this interface to sort the objects as they are being added to the `SortedList<K,V>`.

`IComparer<T>` is designed to solve the problem of allowing objects to be sorted based on different criteria in different contexts. This interface also allows you to sort types that you did not write. If you also wanted to sort the `Square` objects by height, you

could create a new class called `CompareHeight`, shown in Example 1-2, which would also implement the `IComparer<Square>` interface.

Example 1-2. Making a type sortable by implementing IComparer

```
public class CompareHeight : IComparer<Square>
{
    public int Compare(object firstSquare, object secondSquare)
    {
        Square square1 = firstSquare as Square;
        Square square2 = secondSquare as Square;
        if (square1 == null || square2 == null)
            throw (new ArgumentException("Both parameters must be of type Square."));
        else
           return Compare(firstSquare,secondSquare);
    }

    #region IComparer<Square> Members

    public int Compare(Square x, Square y)
    {
        if (x.Height == y.Height)
            return 0;
        else if (x.Height > y.Height)
            return 1;
        else if (x.Height < y.Height)
            return -1;
        else
            return -1;
    }

    #endregion
}
```

This class is then passed in to the `IComparer` parameter of the `Sort` routine. Now you can specify different ways to sort your `Square` objects. The comparison method implemented in the comparer must be consistent and apply a *total ordering* so that when the comparison function declares equality for two items, it is absolutely true and not a result of one item not being greater than another or one item not being less than another.

For best performance, keep the `CompareTo` method short and efficient, because it will be called multiple times by the `Sort` methods. For example, in sorting an array with four items, the `Compare` method is called 10 times.

The `TestSort` method shown in Example 1-3 demonstrates how to use the `Square` and `CompareHeight` classes with the `List<Square>` and `SortedList<int,Square>` instances.

Example 1-3. TestSort method

```
public static void TestSort()
{
    List<Square> listOfSquares = new List<Square>(){
                                 new Square(1,3),
                                 new Square(4,3),
                                 new Square(2,1),
                                 new Square(6,1)};
    // Test a List<String>
    Console.WriteLine("List<String>");
    Console.WriteLine("Original list");
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }

    Console.WriteLine();
    IComparer<Square> heightCompare = new CompareHeight();
    listOfSquares.Sort(heightCompare);
    Console.WriteLine("Sorted list using IComparer<Square>=heightCompare");
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }

    Console.WriteLine();
    Console.WriteLine("Sorted list using IComparable<Square>");
    listOfSquares.Sort();
    foreach (Square square in listOfSquares)
    {
        Console.WriteLine(square.ToString());
    }


    // Test a SORTEDLIST
    var sortedListOfSquares = new SortedList<int,Square>(){
                            { 0, new Square(1,3)},
                            { 2, new Square(3,3)},
                            { 1, new Square(2,1)},
                            { 3, new Square(6,1)}};

    Console.WriteLine();
    Console.WriteLine();
    Console.WriteLine("SortedList<Square>");
    foreach (KeyValuePair<int,Square> kvp in sortedListOfSquares)
    {
```

```
        Console.WriteLine ($"{kvp.Key} : {kvp.Value}");
    }
}
```

This code displays the following output:

```
List<String>
Original list
Height:1 Width:3
Height:4 Width:3
Height:2 Width:1
Height:6 Width:1

Sorted list using IComparer<Square>=heightCompare
Height:1 Width:3
Height:2 Width:1
Height:4 Width:3
Height:6 Width:1

Sorted list using IComparable<Square>
Height:2 Width:1
Height:1 Width:3
Height:6 Width:1
Height:4 Width:3

SortedList<Square>
0 : Height:1 Width:3
1 : Height:2 Width:1
2 : Height:3 Width:3
3 : Height:6 Width:1
```

See Also

Recipe 1.3, and the "IComparable<T> Interface" topic in the MSDN documentation.

1.3 Making a Type Searchable

Problem

You have a data type that will be stored as elements in a `List<T>`. You would like to use the `BinarySearch` method to allow for custom searching of your data types in the list.

Solution

Use the `IComparable<T>` and `IComparer<T>` interfaces. The `Square` class, from Recipe 1.2, implements the `IComparable<T>` interface in such a way that the `List<T>` and `SortedList<K,V>` collections can sort and search an array or collection of `Square` objects.

Discussion

By implementing the `IComparable<T>` interface on your class (or structure), you can take advantage of the search routines of the `List<T>` and `SortedList<K,V>` classes. The algorithms for searching are built into these classes; all you have to do is tell them how to search your classes via the code you implement in the `IComparable<T>`. `CompareTo` method.

To implement the `CompareTo` method, see Recipe 1.2.

The `List<T>` class provides a `BinarySearch` method to perform a search on the elements in that list. The elements are compared against an object passed to the `BinarySearch` method in the object parameter. The `SortedList` class does not have a `BinarySearch` method; instead, it has the `ContainsKey` method, which performs a binary search on the key contained in the list. The `ContainsValue` method of the `SortedList` class performs a linear search when searching for values. This linear search uses the `Equals` method of the elements in the `SortedList` collection to do its work. The `Compare` and `CompareTo` methods do not have any effect on the operation of the linear search performed in the `SortedList` class, but they do have an effect on binary searches.

To perform an accurate search using the `BinarySearch` methods of the `List<T>` class, you must first sort the `List<T>` using its `Sort` method. In addition, if you pass an `IComparer<T>` interface to the `BinarySearch` method, you must also pass the same interface to the `Sort` method. Otherwise, the `BinarySearch` method might not be able to find the object you are looking for.

The `TestSort` method shown in Example 1-4 demonstrates how to use the `Square` and `CompareHeight` classes with the `List<Square>` and `SortedList<int,Square>` collection instances.

Example 1-4. Making a type searchable

```
public static void TestSearch()
{
    List<Square> listOfSquares = new List<Square> {new Square(1,3),
                                                   new Square(4,3),
                                                   new Square(2,1),
                                                   new Square(6,1)};
    IComparer<Square> heightCompare = new CompareHeight();

    // Test a List<Square>
    Console.WriteLine("List<Square>");
    Console.WriteLine("Original list");
```

```
foreach (Square square in listOfSquares)
{
    Console.WriteLine(square.ToString());
}

Console.WriteLine();
Console.WriteLine("Sorted list using IComparer<Square>=heightCompare");
listOfSquares.Sort(heightCompare);
foreach (Square square in listOfSquares)
{
    Console.WriteLine(square.ToString());
}

Console.WriteLine();
Console.WriteLine("Search using IComparer<Square>=heightCompare");
int found = listOfSquares.BinarySearch(new Square(1,3), heightCompare);
Console.WriteLine($"Found (1,3): {found}");

Console.WriteLine();
Console.WriteLine("Sorted list using IComparable<Square>");
listOfSquares.Sort();
foreach (Square square in listOfSquares)
{
    Console.WriteLine(square.ToString());
}

Console.WriteLine();
Console.WriteLine("Search using IComparable<Square>");
found = listOfSquares.BinarySearch(new Square(6,1)); // Use IComparable
Console.WriteLine($"Found (6,1): {found}");

// Test a SortedList<Square>
var sortedListOfSquares = new SortedList<int,Square>(){
                              {0, new Square(1,3)},
                              {2, new Square(4,3)},
                              {1, new Square(2,1)},
                              {4, new Square(6,1)}};
Console.WriteLine();
Console.WriteLine("SortedList<Square>");
foreach (KeyValuePair<int,Square> kvp in sortedListOfSquares)
{
    Console.WriteLine ($"{kvp.Key} : {kvp.Value}");
}

Console.WriteLine();
bool foundItem = sortedListOfSquares.ContainsKey(2);
Console.WriteLine($"sortedListOfSquares.ContainsKey(2): {foundItem}");

// Does not use IComparer or IComparable
// -- uses a linear search along with the Equals method
// which has not been overloaded
Console.WriteLine();
```

```
    Square value = new Square(6,1);
    foundItem = sortedListOfSquares.ContainsValue(value);
    Console.WriteLine("sortedListOfSquares.ContainsValue " +
                        $"(new Square(6,1)): {foundItem}");
}
```

This code displays the following:

```
    List"Square>
    Original list
    Height:1 Width:3
    Height:4 Width:3
    Height:2 Width:1
    Height:6 Width:1

    Sorted list using IComparer"Square>=heightCompare
    Height:1 Width:3
    Height:2 Width:1
    Height:4 Width:3
    Height:6 Width:1

    Search using IComparer"Square>=heightCompare
    Found (1,3): 0

    Sorted list using IComparable"Square>
    Height:2 Width:1
    Height:1 Width:3
    Height:6 Width:1
    Height:4 Width:3

    Search using IComparable"Square>
    Found (6,1): 2


    SortedList"Square>
    0 : Height:1 Width:3
    1 : Height:2 Width:1
    2 : Height:4 Width:3
    4 : Height:6 Width:1

    sortedListOfSquares.ContainsKey(2): True
    sortedListOfSquares.ContainsValue(new Square(6,1)): True
```

See Also

Recipe 1.2, and the "IComparable<T> Interface" and "IComparer<T> Interface" topics in the MSDN documentation.

1.4 Returning Multiple Items from a Method

Problem

In many cases, a single return value for a method is not enough. You need a way to return more than one item from a method.

Solution

Use the `out` keyword on parameters that will act as return parameters. The following method accepts an `inputShape` parameter and calculates `height`, `width`, and `depth` from that value:

```
public void ReturnDimensions(int inputShape,
                             out int height,
                             out int width,
                             out int depth)
{
    height = 0;
    width = 0;
    depth = 0;

    // Calculate height, width, and depth from the inputShape value.
}
```

This method would be called in the following manner:

```
// Declare output parameters.
int height;
int width;
int depth;

// Call method and return the height, width, and depth.
Obj.ReturnDimensions(1, out height, out width, out depth);
```

Another method is to return a class or structure containing all the return values. The previous method has been modified here to return a structure instead of using `out` arguments:

```
public Dimensions ReturnDimensions(int inputShape)
{
    // The default ctor automatically defaults this structure's members to 0.
    Dimensions objDim = new Dimensions();

    // Calculate objDim.Height, objDim.Width, objDim.Depth
    //    from the inputShape value...

    return objDim;
}
```

where `Dimensions` is defined as follows:

```
public struct Dimensions
{
    public int Height;
    public int Width;
    public int Depth;
}
```

This method would now be called in this manner:

```
// Call method and return the height, width, and depth.
Dimensions objDim = obj.ReturnDimensions(1);
```

Rather than returning a user-defined class or structure from this method, you can use a `Tuple` object containing all the return values. The previous method has been modified here to return a `Tuple`:

```
public Tuple<int, int, int> ReturnDimensionsAsTuple(int inputShape)
{
    // Calculate objDim.Height, objDim.Width, objDim.Depth from the inputShape
    // value e.g. {5, 10, 15}

    // Create a Tuple with calculated values
    var objDim = Tuple.Create<int, int, int>(5, 10, 15);

    return (objDim);
}
```

This method would now be called in this manner:

```
// Call method and return the height, width, and depth.
Tuple<int, int, int> objDim = obj.ReturnDimensions(1);
```

Discussion

Marking a parameter in a method signature with the `out` keyword indicates that this parameter will be initialized and returned by this method. This trick is useful when a method is required to return more than one value. A method can, at most, have only one return value, but through the use of the `out` keyword, you can mark several parameters as a kind of return value.

To set up an `out` parameter, mark the parameter in the method signature with the `out` keyword as shown here:

```
public void ReturnDimensions(int inputShape,
                             out int height,
                             out int width,
                             out int depth)
{
    ...
}
```

To call this method, you must also mark the calling method's arguments with the `out` keyword, shown here:

```
obj.ReturnDimensions(1, out height, out width, out depth);
```

The `out` arguments in this method call do not have to be initialized; they can simply be declared and passed in to the `ReturnDimensions` method. Regardless of whether they are initialized before the method call, they must be initialized before they are used within the `ReturnDimensions` method. Even if they are not used through every path in the `ReturnDimensions` method, they still must be initialized. That is why this method starts out with the following three lines of code:

```
height = 0;
width = 0;
depth = 0;
```

You may be wondering why you couldn't use a `ref` parameter instead of the `out` parameter, as both allow a method to change the value of an argument marked as such. The answer is that an `out` parameter makes the code somewhat self-documenting. You know that when an `out` parameter is encountered, it is acting as a return value. In addition, an `out` parameter does not require the extra work to be initialized before it is passed in to the method, while a `ref` parameter does.

An `out` parameter does not have to be marshaled when the method is called; rather, it is marshaled once when the method returns the data to the caller. Any other type of call (by-value or by-reference using the `ref` keyword) requires that the value be marshaled in both directions. Using the `out` keyword in marshaling scenarios improves remoting performance.

An `out` parameter is great when there are only a few values that need to be returned, but when you start encountering 4, 5, 6, or more values that need to be returned, it can get unwieldy. Another option for returning multiple values is to create and return a user-defined class/structure or to use a `Tuple` to package up all the values that need to be returned by a method.

The first option, using a class/structure to return the values, is straightforward. Just create the type (in this example it is a structure) like so:

```
public struct Dimensions
{
    public int Height;
    public int Width;
    public int Depth;
}
```

Fill in each field of this structure with the required data and then return it from the method as shown in the Solution section.

The second option, using a `Tuple`, is an even more elegant solution than using a user-defined object. A `Tuple` can be created to hold any number of values of varying types. In addition, the data you store in the `Tuple` is immutable; once you add the data to the `Tuple` through the constructor or the static `Create` method, that data cannot be changed.

`Tuples` can accept up to and including eight separate values. If you need to return more than eight values, you will need to use the special `Tuple` class:

```
Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> Class
```

When creating a `Tuple` with more than eight values, you cannot use the static `Create` method—you must instead use the constructor of the class. This is how you would create a `Tuple` of 10 integer values:

```
var values = new Tuple<int, int, int, int, int, int, int, Tuple<int, int, int>> (
                    1, 2, 3, 4, 5, 6, 7, new Tuple<int, int, int> (8, 9, 10));
```

Of course, you can continue to add more `Tuples` to the end of each embedded `Tuple`, creating any size `Tuple` that you need.

See Also

The "Tuple Class" and "Tuple<T1, T2, T3, T4, T5, T6, T7, TRest> Class" topics in the MSDN documentation.

1.5 Parsing Command-Line Parameters

Problem

You require your applications to accept one or more command-line parameters in a standard format (described in the Discussion section). You need to access and parse the entire command line passed to your application.

Solution

In Example 1-5, use the following classes together to help with parsing command-line parameters: `Argument`, `ArgumentDefinition`, and `ArgumentSemanticAnalyzer`.

Example 1-5. Argument class

```
using System;
using System.Diagnostics;
using System.Linq;
```

```
using System.Collections.ObjectModel;

public sealed class Argument
{
    public string Original { get; }
    public string Switch { get; private set; }
    public ReadOnlyCollection<string> SubArguments { get; }
    private List<string> subArguments;
    public Argument(string original)
    {
        Original = original;
        Switch = string.Empty;
        subArguments = new List<string>();
        SubArguments = new ReadOnlyCollection<string>(subArguments);
        Parse();
    }

    private void Parse()
    {
        if (string.IsNullOrEmpty(Original))
        {
            return;
        }
        char[] switchChars = { '/', '-' };
        if (!switchChars.Contains(Original[0]))

        {
            return;
        }
        string switchString = Original.Substring(1);
        string subArgsString = string.Empty;
        int colon = switchString.IndexOf(':');
        if (colon >= 0)
        {
            subArgsString = switchString.Substring(colon + 1);
            switchString = switchString.Substring(0, colon);
        }
        Switch = switchString;
        if (!string.IsNullOrEmpty(subArgsString))
            subArguments.AddRange(subArgsString.Split(';'));
    }

    // A set of predicates that provide useful information about itself
    //   Implemented using lambdas
    public bool IsSimple => SubArguments.Count == 0;
    public bool IsSimpleSwitch =>
                !string.IsNullOrEmpty(Switch) && SubArguments.Count == 0;
    public bool IsCompoundSwitch =>
                !string.IsNullOrEmpty(Switch) && SubArguments.Count == 1;
    public bool IsComplexSwitch =>
                !string.IsNullOrEmpty(Switch) && SubArguments.Count > 0;
}
```

```
public sealed class ArgumentDefinition
{
     public string ArgumentSwitch { get; }
     public string Syntax { get; }
     public string Description { get; }
     public Func<Argument, bool> Verifier { get; }

     public ArgumentDefinition(string argumentSwitch,
                               string syntax,
                               string description,
                               Func<Argument, bool> verifier)
    {
        ArgumentSwitch = argumentSwitch.ToUpper();
        Syntax = syntax;
        Description = description;
        Verifier = verifier;
    }

    public bool Verify(Argument arg) => Verifier(arg);
}

public sealed class ArgumentSemanticAnalyzer
{

    private List<ArgumentDefinition> argumentDefinitions =
        new List<ArgumentDefinition>();
    private Dictionary<string, Action<Argument>> argumentActions =
        new Dictionary<string, Action<Argument>>();

    public ReadOnlyCollection<Argument> UnrecognizedArguments { get; private set; }
    public ReadOnlyCollection<Argument> MalformedArguments { get; private set; }
    public ReadOnlyCollection<Argument> RepeatedArguments { get; private set; }

    public ReadOnlyCollection<ArgumentDefinition> ArgumentDefinitions =>
           new ReadOnlyCollection<ArgumentDefinition>(argumentDefinitions);

    public IEnumerable<string> DefinedSwitches =>
                           from argumentDefinition in argumentDefinitions
                           select argumentDefinition.ArgumentSwitch;

    public void AddArgumentVerifier(ArgumentDefinition verifier) =>
        argumentDefinitions.Add(verifier);

    public void RemoveArgumentVerifier(ArgumentDefinition verifier)
    {
        var verifiersToRemove = from v in argumentDefinitions
                                where v.ArgumentSwitch == verifier.ArgumentSwitch
                                select v;
        foreach (var v in verifiersToRemove)
            argumentDefinitions.Remove(v);
    }
```

```
public void AddArgumentAction(string argumentSwitch, Action<Argument> action) =>
    argumentActions.Add(argumentSwitch, action);

public void RemoveArgumentAction(string argumentSwitch)
{
    if (argumentActions.Keys.Contains(argumentSwitch))
        argumentActions.Remove(argumentSwitch);
}

public bool VerifyArguments(IEnumerable<Argument> arguments)
{
   // no parameter to verify with, fail.
   if (!argumentDefinitions.Any())

        return false;

    // Identify if any of the arguments are not defined
    this.UnrecognizedArguments =
             ( from argument in arguments
               where !DefinedSwitches.Contains(argument.Switch.ToUpper())
               select argument).ToList().AsReadOnly();

    if (this.UnrecognizedArguments.Any())
        return false;

    //Check for all the arguments where the switch matches a known switch,
    //but our well-formedness predicate is false.
    this.MalformedArguments = ( from argument in arguments
                                join argumentDefinition in argumentDefinitions
                                on argument.Switch.ToUpper() equals
                                    argumentDefinition.ArgumentSwitch
                                where !argumentDefinition.Verify(argument)
                                select argument).ToList().AsReadOnly();

    if (this.MalformedArguments.Any())
        return false;

    //Sort the arguments into "groups" by their switch, count every group,
    //and select any groups that contain more than one element,
    //We then get a read-only list of the items.
    this.RepeatedArguments =
            (from argumentGroup in
                from argument in arguments
                where !argument.IsSimple
                group argument by argument.Switch.ToUpper()
            where argumentGroup.Count() > 1
            select argumentGroup).SelectMany(ag => ag).ToList().AsReadOnly();

    if (this.RepeatedArguments.Any())
    return false;
```

```
        return true;
    }

    public void EvaluateArguments(IEnumerable<Argument> arguments)
    {
        //Now we just apply each action:
        foreach (Argument argument in arguments)
            argumentActions[argument.Switch.ToUpper()](argument);
    }

    public string InvalidArgumentsDisplay()
    {
        StringBuilder builder = new StringBuilder();
        builder.AppendFormat($"Invalid arguments: {Environment.NewLine}");
        // Add the unrecognized arguments

        FormatInvalidArguments(builder, this.UnrecognizedArguments,
            "Unrecognized argument: {0}{1}");

        // Add the malformed arguments
        FormatInvalidArguments(builder, this.MalformedArguments,
            "Malformed argument: {0}{1}");

       // For the repeated arguments, we want to group them for the display,
       // so group by switch and then add it to the string being built.
       var argumentGroups = from argument in this.RepeatedArguments
                            group argument by argument.Switch.ToUpper() into ag
                            select new { Switch = ag.Key, Instances = ag};

       foreach (var argumentGroup in argumentGroups)
       {
           builder.AppendFormat($"Repeated argument:
                                {argumentGroup.Switch}{Environment.NewLine}");
           FormatInvalidArguments(builder, argumentGroup.Instances.ToList(),
               "\t{0}{1}");
       }
       return builder.ToString();
   }

   private void FormatInvalidArguments(StringBuilder builder,
      IEnumerable<Argument> invalidArguments, string errorFormat)
   {
      if (invalidArguments != null)
      {
          foreach (Argument argument in invalidArguments)
          {
              builder.AppendFormat(errorFormat,
                  argument.Original, Environment.NewLine);
          }
      }
   }
}
```

Here is one example of how to use these classes to process the command line for an application:

```
public static void Main(string[] argumentStrings)
{
    var arguments = (from argument in argumentStrings
        select new Argument(argument)).ToArray();

    Console.Write("Command line: ");
    foreach (Argument a in arguments)
    {
        Console.Write($"{a.Original} ");
    }
    Console.WriteLine("");

    ArgumentSemanticAnalyzer analyzer = new ArgumentSemanticAnalyzer();
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("output",
            "/output:[path to output]",
            "Specifies the location of the output file.",
            x => x.IsCompoundSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("trialMode",
            "/trialmode",
            "If this is specified it places the product into trial mode",
            x => x.IsSimpleSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("DEBUGOUTPUT",
            "/debugoutput:[value1];[value2];[value3]",
            "A listing of the files the debug output " +
            "information will be written to",
            x => x.IsComplexSwitch));
    analyzer.AddArgumentVerifier(
        new ArgumentDefinition("",
            "[literal value]",
            "A literal value",
            x => x.IsSimple));

    if (!analyzer.VerifyArguments(arguments))
    {
        string invalidArguments = analyzer.InvalidArgumentsDisplay();
        Console.WriteLine(invalidArguments);
        ShowUsage(analyzer);
        return;
    }

    // Set up holders for the command line parsing results
    string output = string.Empty;
    bool trialmode = false;
    IEnumerable<string> debugOutput = null;
    List<string> literals = new List<string>();
```

```
    //For each parsed argument, we want to apply an action,
    // so add them to the analyzer.
    analyzer.AddArgumentAction("OUTPUT", x => { output = x.SubArguments[0]; });
    analyzer.AddArgumentAction("TRIALMODE", x => { trialmode = true; });
    analyzer.AddArgumentAction("DEBUGOUTPUT", x =>
                                            { debugOutput = x.SubArguments;
    });

    analyzer.AddArgumentAction("", x=>{literals.Add(x.Original);});

    // check the arguments and run the actions
    analyzer.EvaluateArguments(arguments);

    // display the results
    Console.WriteLine("");
    Console.WriteLine($"OUTPUT: {output}");
    Console.WriteLine($"TRIALMODE: {trialmode}");
    if (debugOutput != null)
    {
        foreach (string item in debugOutput)

        {
             Console.WriteLine($"DEBUGOUTPUT: {item}");
        }
    }
    foreach (string literal in literals)
    {
        Console.WriteLine($"LITERAL: {literal}");
    }
}

public static void ShowUsage(ArgumentSemanticAnalyzer analyzer)
{
    Console.WriteLine("Program.exe allows the following arguments:");
    foreach (ArgumentDefinition definition in analyzer.ArgumentDefinitions)
    {
        Console.WriteLine($"\t{definition.ArgumentSwitch}:
                          ({definition.Description}){Environment.NewLine}
                          \tSyntax: {definition.Syntax}");
    }
}
```

Discussion

Before you can parse command-line parameters, you must decide upon a common format. The format for this recipe follows the command-line format for the Visual C# .NET language compiler. The format used is defined as follows:

- All command-line arguments are separated by one or more whitespace characters.

- Each argument may start with either a `-` or `/` character, but not both. If it does not, that argument is considered a literal, such as a filename.
- Each argument that starts with either the `-` or `/` character may be divided up into a switch followed by a colon followed by one or more arguments separated with the `;` character. The command-line parameter `-sw:arg1;arg2;arg3` is divided up into a switch (`sw`) and three arguments (`arg1`, `arg2`, and `arg3`). Note that there should not be any spaces in the full argument; otherwise, the runtime command-line parser will split up the argument into two or more arguments.
- Strings delineated with double quotes, such as `"c:\test\file.log"`, will have their double quotes stripped off. This is a function of the operating system interpreting the arguments passed in to your application.
- Single quotes are not stripped off.
- To preserve double quotes, precede the double quote character with the `\` escape sequence character.
- The `\` character is handled as an escape sequence character only when followed by a double quote—in which case, only the double quote is displayed.
- The `^` character is handled by the *runtime* command-line parser as a special character.

Fortunately, the runtime command-line parser handles most of this before your application receives the individual parsed arguments.

The runtime command-line parser passes a `string[]` containing each parsed argument to the entry point of your application. The entry point can take one of the following forms:

```
public static void Main()
public static int Main()
public static void Main(string[] args)
public static int Main(string[] args)
```

The first two accept no arguments, but the last two accept the array of parsed command-line arguments. Note that the static `Environment.CommandLine` property will also return a string containing the entire command line, and the static `Environment.GetCommandLineArgs` method will return an array of strings containing the parsed command-line arguments.

The three classes presented in the Solution address the phases of dealing with the command-line arguments:

`Argument`
: Encapsulates a single command-line argument and is responsible for parsing the argument.

`ArgumentDefinition`
: Defines an argument that will be valid for the current command line.

`ArgumentSemanticAnalyzer`
: Performs the verification and retrieval of the arguments based on the `ArgumentDefinitions` that are set up.

Passing in the following command-line arguments to this application:

```
MyApp c:\input\infile.txt -output:d:\outfile.txt -trialmode
```

results in the following parsed switches and arguments:

```
Command line: c:\input\infile.txt -output:d:\outfile.txt -trialmode
OUTPUT: d:\outfile.txt
TRIALMODE: True
LITERAL: c:\input\infile.txt
```

If you input command-line parameters incorrectly, such as forgetting to add arguments to the `-output` switch, you get the following output:

```
Command line: c:\input\infile.txt -output: -trialmode
Invalid arguments:
Malformed argument: -output

Program.exe allows the following arguments:
       OUTPUT: (Specifies the location of the output file.)
       Syntax: /output:[path to output]
       TRIALMODE: (If this is specified, it places the product into trial mode)
       Syntax: /trialmode
       DEBUGOUTPUT: (A listing of the files the debug output information will be
                     written to)
       Syntax: /debugoutput:[value1];[value2];[value3]
       : (A literal value)
       Syntax: [literal value]
```

There are a few items in the code that are worth pointing out.

Each `Argument` instance needs to be able to determine certain things about itself; accordingly, a set of predicates that tell us useful information about this `Argument` are exposed as properties on the `Argument`. The `ArgumentSemanticAnalyzer` will use these properties to determine the characteristics of the argument:

```
public bool IsSimple => SubArguments.Count == 0;
public bool IsSimpleSwitch =>
            !string.IsNullOrEmpty(Switch) && SubArguments.Count == 0;
public bool IsCompoundSwitch =>
            !string.IsNullOrEmpty(Switch) && SubArguments.Count == 1;
public bool IsComplexSwitch =>
            !string.IsNullOrEmpty(Switch) && SubArguments.Count > 0;
```

For more information on lambda expressions, see the introduction to Chapter 4. Also see Recipe 1.16 for a discussion of using lambda expressions to implement closures.

In a number of places in the code, the `ToArray` or `ToList` methods are called on the result of a LINQ query:

```
var arguments = (from argument in argumentStrings
                 select new Argument(argument)).ToArray();
```

This is because query results use deferred execution, which means that not only are the results calculated in a lazy manner, but they are recalculated every time they are accessed. Using the `ToArray` or `ToList` methods forces the eager evaluation of the results and generates a copy that will not reevaluate during each usage. The query logic does not know if the collection being worked on is changing or not, so it has to reevaluate each time unless you make a "point in time" copy using these methods.

To verify that these arguments are correct, we must create an `ArgumentDefinition` and associate it for each acceptable argument type with the `ArgumentSemanticAnalyzer`:

```
ArgumentSemanticAnalyzer analyzer = new ArgumentSemanticAnalyzer();
analyzer.AddArgumentVerifier(
        new ArgumentDefinition("output",
            "/output:[path to output]",
            "Specifies the location of the output file.",
            x => x.IsCompoundSwitch));
analyzer.AddArgumentVerifier(
        new ArgumentDefinition("trialMode",
            "/trialmode",
            "If this is specified it places the product into trial mode",
            x => x.IsSimpleSwitch));
analyzer.AddArgumentVerifier(
        new ArgumentDefinition("DEBUGOUTPUT",
            "/debugoutput:[value1];[value2];[value3]",
            "A listing of the files the debug output " +
            "information will be written to",
            x => x.IsComplexSwitch));
analyzer.AddArgumentVerifier(
        new ArgumentDefinition("",
            "[literal value]",
            "A literal value",
            x => x.IsSimple));
```

There are four parts to each `ArgumentDefinition`: the argument switch, a string showing the syntax of the argument, a description of the argument, and the verification predicate to verify the argument. This information can be used to verify the argument, as shown here:

```
//Check for all the arguments where the switch matches a known switch,
//but our well-formedness predicate is false.
this.MalformedArguments = ( from argument in arguments
                            join argumentDefinition in argumentDefinitions
                              on argument.Switch.ToUpper() equals
                                  argumentDefinition.ArgumentSwitch
                              where !argumentDefinition.Verify(argument)
                              select argument).ToList().AsReadOnly();
```

The `ArgumentDefinitions` also allow you to compose a usage method for the program:

```
public static void ShowUsage(ArgumentSemanticAnalyzer analyzer)
{
    Console.WriteLine("Program.exe allows the following arguments:");
    foreach (ArgumentDefinition definition in analyzer.ArgumentDefinitions)
    {
        Console.WriteLine("\t{0}: ({1}){2}\tSyntax: {3}",
            definition.ArgumentSwitch, definition.Description,
            Environment.NewLine,definition.Syntax);
    }
}
```

To get the values of the arguments so they can be used, we need to extract the information out of the parsed arguments. For the Solution example, we would need the following information:

```
// Set up holders for the command line parsing results
string output = string.Empty;
bool trialmode = false;
IEnumerable<string> debugOutput = null;
List<string> literals = new List<string>();
```

How are these values filled in? Well, we need to associate an action with each `Argument` to determine how the value should be retrieved from an `Argument` instance. The action is a predicate, which makes this a very powerful approach, as any predicate can be used here. Here is where those `Argument` actions are defined and associated with the `ArgumentSemanticAnalyzer`:

```
//For each parsed argument, we want to apply an action,
//    so add them to the analyzer.
analyzer.AddArgumentAction("OUTPUT", x => { output = x.SubArguments[0]; });
analyzer.AddArgumentAction("TRIALMODE", x => { trialmode = true; });
analyzer.AddArgumentAction("DEBUGOUTPUT", x =>
                                          { debugOutput = x.SubArguments;});
analyzer.AddArgumentAction("", x=>{literals.Add(x.Original);});
```

Now that all of the actions are set up, we can retrieve the values by using the `EvaluateArguments` method on the `ArgumentSemanticAnalyzer`:

```
// check the arguments and run the actions
analyzer.EvaluateArguments(arguments);
```

Now the arguments have been filled in by the execution of the actions, and the program can run with those values:

```
// Run the program passing in the argument values:
Program program = new Program(output, trialmode, debugOutput, literals);
program.Run();
```

The verification of the arguments uses LINQ to query for unrecognized, malformed, or repeated arguments, any of which will cause the parameters to be invalid:

```
public bool VerifyArguments(IEnumerable<Argument> arguments)
{
    // no parameter to verify with, fail.
    if (!argumentDefinitions.Any())
        return false;

    // Identify if any of the arguments are not defined
    this.UnrecognizedArguments =
                (  from argument in arguments
                   where !DefinedSwitches.Contains(argument.Switch.ToUpper())
                   select argument).ToList().AsReadOnly();

    if (this.UnrecognizedArguments.Any())
        return false;

    //Check for all the arguments where the switch matches a known switch,
    //but our well-formedness predicate is false.
    this.MalformedArguments = ( from argument in arguments
                                join argumentDefinition in argumentDefinitions
                                on argument.Switch.ToUpper() equals
                                    argumentDefinition.ArgumentSwitch
                                where !argumentDefinition.Verify(argument)
                                select argument).ToList().AsReadOnly();
    if (this.MalformedArguments.Any())
        return false;

    //Sort the arguments into "groups" by their switch, count every group,
    //and select any groups that contain more than one element.
    //We then get a read-only list of the items.
    this.RepeatedArguments =
                (from argumentGroup in
                    from argument in arguments
                    where !argument.IsSimple
                    group argument by argument.Switch.ToUpper()
                where argumentGroup.Count() > 1
                select argumentGroup).SelectMany(ag => ag).ToList().AsReadOnly();

    if (this.RepeatedArguments.Any())
        return false;

    return true;
}
```

Look at how much easier it is to understand each phase of the verification, compared with how it would be done before LINQ—with multiple nested loops, switches, `IndexOf`s, and other mechanisms. Each query concisely states in the language of the problem domain what task it is attempting to perform.

LINQ is designed to help with problems where data must be sorted, searched, grouped, filtered, and projected. Use it!

See Also

The "Main" and "Command-Line Arguments" topics in the MSDN documentation.

1.6 Initializing a Constant Field at Runtime

Problem

A field marked as `const` can be initialized only at compile time. You need to initialize a field to a valid value at runtime, not at compile time. This field must then act as if it were a constant field for the rest of the application's life.

Solution

You have two choices when declaring a constant value in your code. You can use a `readonly` field or a `const` field. Each has its own strengths and weaknesses. However, if you need to initialize a `constant` field at runtime, you must use a `readonly` field:

```
public class Foo
{
    public readonly int bar;

    public Foo() {}

    public Foo(int constInitValue)
    {
        bar = constInitValue;
    }

    // Rest of class...
}
```

This is not possible using a `const` field. A `const` field can be initialized only at compile time:

```
public class Foo
{
```

```
    public const int bar;     // This line causes a compile-time error.

    public Foo() {}

    public Foo(int constInitValue)
    {
        bar = constInitValue; // This line also causes a compile-time error.
    }
    // Rest of class...
}
```

Discussion

A `readonly` field allows initialization to take place only in the constructor at runtime, whereas a `const` field must be initialized at compile time. Therefore, implementing a `readonly` field is the only way to allow a field that must be constant to be initialized at runtime.

There are only two ways to initialize a `readonly` field. The first is by adding an initializer to the field itself:

```
public readonly int bar = 100;
```

The second way is to initialize the `readonly` field through a constructor. This approach is demonstrated by the code in the Solution to this recipe. If you look at the following class:

```
public class Foo
{
    public readonly int x;
    public const int y = 1;

    public Foo() {}
    public Foo(int roInitValue)
    {
        x = roInitValue;
    }

    // Rest of class...
}
```

you'll see it is compiled into the following IL (intermediate language):

```
.class auto ansi nested public beforefieldinit Foo
       extends [mscorlib]System.Object        {
.field public static literal int32 y = int32(0x00000001) //<<-- const field
.field public initonly int32 x                          //<<-- readonly field
.method public hidebysig specialname rtspecialname
        instance void  .ctor(int32 roInitValue) cil managed
{
  // Code size       16 (0x10)
  .maxstack  8
```

```
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  nop
  IL_0007:  nop
  IL_0008:  ldarg.0
  IL_0009:  ldarg.1
  IL_000a:  stfld      int32 CSharpRecipes.ClassesAndGenerics/Foo::x
  IL_000f:  ret
} // end of method Foo::.ctor
.method public hidebysig specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       9 (0x9)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  nop
  IL_0007:  nop
  IL_0008:  ret
} // end of method Foo::.ctor
} // End of class Foo
```

Notice that a `const` field is compiled into a `static` field, and a `readonly` field is compiled into an `instance` field. Therefore, you need only a class name to access a `const` field.

A common argument against using `const` fields is that they do not version as well as `readonly` fields. If you rebuild a component that defines a `const` field and the value of that `const` changes in a later version, any other components that were built against the old version won't pick up the new value. If there is any chance that a field is going to change, don't make it a `const` field.

The following code shows how to use an instance `readonly` field:

```
Foo obj1 = new Foo(100);
Console.WriteLine(obj1.bar);
```

See Also

The "const" and "readonly" keywords in the MSDN documentation.

1.7 Building Cloneable Classes

Problem

You need a method of performing a shallow cloning operation, a deep cloning operation, or both on a data type that may also reference other types, but the `ICloneable` interface should not be used, as it violates the .NET Framework Design Guidelines.

Solution

To resolve the issue with using `ICloneable`, create two other interfaces to establish a copying pattern, `IShallowCopy<T>` and `IDeepCopy<T>`:

```
public interface IShallowCopy<T>
{
    T ShallowCopy();
}
public interface IDeepCopy<T>
{
    T DeepCopy();
}
```

Shallow copying means that the copied object's fields will reference the same objects as the original object. To allow shallow copying, implement the `IShallowCopy<T>` interface in the class:

```
using System;
using System.Collections;
using System.Collections.Generic;

public class ShallowClone : IShallowCopy<ShallowClone>
{
    public int Data = 1;
    public List<string> ListData = new List<string>();
    public object ObjData = new object();

    public ShallowClone ShallowCopy() => (ShallowClone)this.MemberwiseClone();
}
```

Deep copying, or cloning, means that the copied object's fields will reference new copies of the original object's fields. To allow deep copying, implement the `IDeepCopy<T>` interface in the class:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

[Serializable]
```

```
public class DeepClone : IDeepCopy<DeepClone>
{
    public int data = 1;
    public List<string> ListData = new List<string>();
    public object objData = new object();

    public DeepClone DeepCopy()
    {
        BinaryFormatter BF = new BinaryFormatter();
        MemoryStream memStream = new MemoryStream();

        BF.Serialize(memStream, this);
        memStream.Flush();
        memStream.Position = 0;

        return (DeepClone)BF.Deserialize(memStream);
    }
}
```

To support both shallow and deep methods of copying, implement both interfaces. The code might appear as follows:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

[Serializable]
public class MultiClone : IShallowCopy<MultiClone>,
                          IDeepCopy<MultiClone>
{
    public int data = 1;
    public List<string> ListData = new List<string>();
    public object objData = new object();

    public MultiClone ShallowCopy() => (MultiClone)this.MemberwiseClone();

    public MultiClone DeepCopy()
    {
        BinaryFormatter BF = new BinaryFormatter();
        MemoryStream memStream = new MemoryStream();

        BF.Serialize(memStream, this);
        memStream.Flush();
        memStream.Position = 0;

        return (MultiClone)BF.Deserialize(memStream);
    }
}
```

Discussion

The .NET Framework has an interface named `ICloneable`, which was originally designed as the means through which cloning is implemented in .NET. The design recommendation is now that this interface not be used in any public API, because it lends itself to different interpretations. The interface looks like this:

```
public interface ICloneable
{
    object Clone();
}
```

Notice that there is a single method, `Clone`, that returns an object. Is the clone a shallow copy of the object or a deep copy? You can't know from the interface, as the implementation could go either way. This is why it should not be used, and the `IShallowCopy<T>` and `IDeepCopy<T>` interfaces are introduced here.

Cloning is the ability to make an exact copy (a clone) of an instance of a type. Cloning may take one of two forms: a shallow copy or a deep copy. Shallow copying is relatively easy: it involves copying the object on which the `ShallowCopy` method was called.

The reference type fields in the original object are copied over, as are the value type fields. This means that if the original object contains a field of type `StreamWriter`, for instance, the cloned object will point to this same instance of the original object's `StreamWriter`; a new object is not created.

There is no need to deal with `static` fields when performing a cloning operation. There is only one memory location reserved for each static field per class, per application domain. The cloned object will have access to the same static fields as the original.

Support for shallow copying is implemented by the `MemberwiseClone` method of the `Object` class, which serves as the base class for all .NET classes. So the following code allows a shallow copy to be created and returned by the `Clone` method:

```
public ShallowClone ShallowCopy() => (ShallowClone)this.MemberwiseClone();
```

Making a deep copy is the second way of cloning an object. A deep copy will make a copy of the original object just as the shallow copy does; however, a deep copy will also make separate copies of each reference type field in the original object. Therefore, if the original object contains a `StreamWriter` type field, the cloned object will also contain a `StreamWriter` type field, but the cloned object's `StreamWriter` field will point to a new `StreamWriter` object, not that of the original object.

Support for deep copying is not automatically provided by the .NET Framework, but the following code illustrates an easy way of implementing a deep copy:

```
BinaryFormatter BF = new BinaryFormatter();
MemoryStream memStream = new MemoryStream();

BF.Serialize(memStream, this);
memStream.Flush();
memStream.Position = 0;

return (BF.Deserialize(memStream));
```

Basically, the original object is serialized out to a memory stream via binary serialization, and then it is deserialized into a new object, which is returned to the caller. It is important to reposition the memory stream pointer back to the start of the stream before calling the `Deserialize` method; otherwise, an exception will be thrown indicating that the serialized object contains no data.

Performing a deep copy using object serialization allows you to change the underlying object without having to modify the code that performs the deep copy. If you performed the deep copy by hand, you'd have to make a new instance of all the instance fields of the original object and copy them over to the cloned object. This is a tedious chore in and of itself. If you make a change to the fields of the object being cloned, you must also change the deep copy code to reflect that modification. Using serialization, you rely on the serializer to dynamically find and serialize all fields contained in the object. If the object is modified, the serializer will still make a deep copy without any code modifications.

One reason you might want to do a deep copy by hand is that the serialization technique presented in this recipe works properly only when everything in your object is serializable. Of course, manual cloning doesn't always help there either—some objects are just inherently uncloneable. Suppose you have a network management application in which an object represents a particular printer on your network. What's it supposed to do when you clone it? Fax a purchase order for a new printer?

One problem inherent with deep copying is performing a deep copy on a nested data structure with circular references. This recipe makes it possible to deal with circular references, although it's a tricky problem. So, in fact, you don't need to avoid circular references if you are using this recipe.

See Also

Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries by Krzysztof Cwalina and Brad Abrams (Addison-Wesley Professional), and the "Object.MemberwiseClone Method" topic in the MSDN documentation.

1.8 Ensuring an Object's Disposal

Problem

You require a way to always have something happen when an object's work is done or it goes out of scope.

Solution

Use the `using` statement:

```
using System;
using System.IO;

// ...

using(FileStream FS = new FileStream("Test.txt", FileMode.Create))
{
    FS.WriteByte((byte)1);
    FS.WriteByte((byte)2);
    FS.WriteByte((byte)3);

    using(StreamWriter SW = new StreamWriter(FS))
    {
        SW.WriteLine("some text.");
    }
}
```

Discussion

The `using` statement is very easy to use and saves you the hassle of writing extra code. If this Solution had not used the `using` statement, it would look like this:

```
FileStream FS = new FileStream("Test.txt", FileMode.Create);
try
{
    FS.WriteByte((byte)1);
    FS.WriteByte((byte)2);
    FS.WriteByte((byte)3);

    StreamWriter SW = new StreamWriter(FS);

    try
    {
       SW.WriteLine("some text.");
    }
    finally
    {
        if (SW != null)
        {
```

```
            ((IDisposable)SW).Dispose();
        }
    }
}
finally
{
   if (FS != null)
   {
       ((IDisposable)FS).Dispose();
   }
}
```

Several points to note about the `using` statement:

- There is a `using` directive, such as:

  ```
  using System.IO;
  ```

  which should be differentiated from the `using` statement. This is potentially confusing to developers first getting into this language.
- The variable(s) defined in the `using` statement clause must all be of the same type, and they must have an initializer. However, you are allowed multiple `using` statements in front of a single code block, so this isn't a significant restriction.
- Any variables defined in the `using` clause are considered read-only in the body of the `using` statement. This prevents a developer from inadvertently switching the variable to refer to a different object and causing problems when attempting to dispose of the object that the variable initially referenced.
- The variable should not be declared outside of the `using` block and then initialized inside of the `using` clause.

This last point is described by the following code:

```
FileStream FS;
using(FS = new FileStream("Test.txt", FileMode.Create))
{
    FS.WriteByte((byte)1);
    FS.WriteByte((byte)2);
    FS.WriteByte((byte)3);

    using(StreamWriter SW = new StreamWriter(FS))
    {
        SW.WriteLine("some text.");
    }
}
```

For this example code, you will not have a problem. But consider that the variable `FS` is usable outside of the `using` block. Essentially, you could revisit this code and modify it as follows:

```
FileStream FS;
using(FS = new FileStream("Test.txt", FileMode.Create))
{
    FS.WriteByte((byte)1);
    FS.WriteByte((byte)2);
    FS.WriteByte((byte)3);

    using(StreamWriter SW = new StreamWriter(FS))
    {
        SW.WriteLine("some text.");
    }
}
FS.WriteByte((byte)4);
```

This code compiles but throws an `ObjectDisposedException` on the last line of this code snippet because the `Dispose` method has already been called on the `FS` object. The object has not yet been collected at this point and still remains in memory in the disposed state.

See Also

The "Cleaning Up Unmanaged Resources," "IDisposable Interface," "Using foreach with Collections," and "Implementing Finalize and Dispose to Clean Up Unmanaged Resources" topics in the MSDN documentation.

1.9 Deciding When and Where to Use Generics

Problem

You want to use generic types in a new project or convert nongeneric types in an existing project to their generic equivalents. However, you do not really know why you would want to do this, and you do not know which nongeneric types should be converted to generic.

Solution

In deciding when and where to use generic types, you need to consider several things:

- Will your type contain or be operating on various unspecified data types (e.g., a collection type)? If so, creating a generic type will offer several benefits over creating a nongeneric type. If your type will operate on only a single specific type, then you may not need to create a generic type.
- If your type will be operating on value types, so that boxing and unboxing operations will occur, you should consider using generics to prevent the performance penalty incurred from boxing and unboxing operations.

- The stronger type checking associated with generics will aid in finding errors sooner (i.e., during compile time as opposed to runtime), thus shortening your bug-fixing cycle.
- Is your code suffering from "code bloat," with you writing multiple classes to handle different data types on which they operate (e.g., a specialized `ArrayList` that stores only `StreamReader`s and another that stores only `StreamWriter`s)? It is easier to write the code once and have it just work for each of the data types it operates on.
- Generics allow for greater clarity of code. By eliminating code bloat and forcing stronger type checking on your types, they make your code easier to read and understand.

Discussion

In most cases, your code will benefit from using a generic type. Generics allow for more efficient code reuse, faster performance, stronger type checking, and easier-to-read code.

See Also

The "Generics Overview" and "Benefits of Generics" topics in the MSDN documentation.

1.10 Understanding Generic Types

Problem

You need to understand how the .NET types work for generics and how generic .NET types differ from regular .NET types.

Solution

A couple of quick experiments can show the differences between regular .NET types and generic .NET types. Before we get deep into the code, if you are unfamiliar with generics, jump to the Discussion section in this recipe for a detailed explanation about generics and then come back to this section.

Now, when a regular .NET type is defined, it looks like the `FixedSizeCollection` type defined in Example 1-6.

Example 1-6. FixedSizeCollection: a regular .NET type

```
public class FixedSizeCollection
{
    /// <summary>
    /// Constructor that increments static counter
    /// and sets the maximum number of items
    /// </summary>
    /// <param name="maxItems"></param>
    public FixedSizeCollection(int maxItems)
    {
         FixedSizeCollection.InstanceCount++;
         this.Items = new object[maxItems];
    }
    /// <summary>
    /// Add an item to the class whose type
    /// is unknown as only object can hold any type
    /// </summary>
    /// <param name="item">item to add</param>
    /// <returns>the index of the item added</returns>
    public int AddItem(object item)
    {
        if (this.ItemCount < this.Items.Length)
        {
            this.Items[this.ItemCount] = item;
            return this.ItemCount++;
        }
        else
            throw new Exception("Item queue is full");
    }

    /// <summary>
    /// Get an item from the class
    /// </summary>
    /// <param name="index">the index of the item to get</param>
    /// <returns>an item of type object</returns>
    public object GetItem(int index)
    {
        if (index >= this.Items.Length &&
            index >= 0)
            throw new ArgumentOutOfRangeException(nameof(index));
        return this.Items[index];
     }

     #region Properties
     /// <summary>
     /// Static instance counter hangs off of the Type for
     /// StandardClass
     /// </summary>
     public static int InstanceCount { get; set; }

     /// <summary>
```

```
    /// The count of the items the class holds
    /// </summary>
    public int ItemCount { get; private set; }

    /// <summary>
    /// The items in the class
    /// </summary>
    private object[] Items { get; set; }
    #endregion // Properties

    /// <summary>
    /// ToString override to provide class detail
    /// </summary>
    /// <returns>formatted string with class details</returns>
    public override string ToString() =>
                $"There are {FixedSizeCollection.InstanceCount.ToString()}
                instances of {this.GetType().ToString()} and this instance
                contains {this.ItemCount} items...";
}
```

`FixedSizeCollection` has a static integer property variable, `InstanceCount`, which is incremented in the instance constructor, and a `ToString` override that prints out how many instances of `FixedSizeCollection` exist in this `AppDomain.FixedSizeCollection`. Additionally, this collection class contains an array of `objects(Items)`, the size of which is determined by the item count passed in to the constructor. `FixedSizeCollection` also implements methods that add and retrieve items (`AddItem`, `GetItem`) and a read-only property to get the number of items currently in the array (`ItemCount`).

`FixedSizeCollection<T>` is a generic .NET type with the same static property `InstanceCount` field, the instance constructor that counts the number of instantiations, and the overridden `ToString` method to tell you how many instances there are of this type. `FixedSizeCollection<T>` also has an `Items` array property and methods corresponding to those in `FixedSizeCollection`, as you can see in Example 1-7.

Example 1-7. FixedSizeCollection<T>: a generic .NET type

```
/// <summary>
/// A generic class to show instance counting
/// </summary>
/// <typeparam name="T">the type parameter used for the array storage</typeparam>
public class FixedSizeCollection<T>
{
    /// <summary>
    /// Constructor that increments static counter and sets up internal storage
    /// </summary>
    /// <param name="items"></param>
    public FixedSizeCollection(int items)
```

```
{
    FixedSizeCollection<T>.InstanceCount++;
    this.Items = new T[items];
}

/// <summary>
/// Add an item to the class whose type
/// is determined by the instantiating type
/// </summary>
/// <param name="item">item to add</param>
/// <returns>the zero-based index of the item added</returns>
public int AddItem(T item)
{
    if (this.ItemCount < this.Items.Length)
    {
        this.Items[this.ItemCount] = item;
        return this.ItemCount++;
    }
    else
        throw new Exception("Item queue is full");
}

/// <summary>
/// Get an item from the class
/// </summary>
/// <param name="index">the zero-based index of the item to get</param>
/// <returns>an item of the instantiating type</returns>
public T GetItem(int index)
{
    if (index >= this.Items.Length &&
        index >= 0)
        throw new ArgumentOutOfRangeException(nameof(index));

    return this.Items[index];
}

#region Properties
/// <summary>
/// Static instance counter hangs off of the
/// instantiated Type for
/// GenericClass
/// </summary>
public static int InstanceCount { get; set; }

/// <summary>
/// The count of the items the class holds
/// </summary>
public int ItemCount { get; private set; }

/// <summary>
/// The items in the class
/// </summary>
```

```
    private T[] Items { get; set; }
    #endregion // Properties

    /// <summary>
    /// ToString override to provide class detail
    /// </summary>
    /// <returns>formatted string with class details</returns>
    public override string ToString() =>
                $"There are {FixedSizeCollection<T>.InstanceCount.ToString()}
                instances of {this.GetType().ToString()} and this instance
                contains {this.ItemCount} items...";
}
```

Things start to differ a little with `FixedSizeCollection<T>` when you look at the `Items` array property implementation. The `Items` array is declared as:

```
    private T[] Items { get; set; }
```

instead of:

```
    private object[] Items { get; set; }
```

The `Items` array property uses the type parameter of the generic class (`<T>`) to determine what types of items are allowed. `FixedSizeCollection` uses `object` for the `Items` array property type, which allows any type to be stored in the array of items (since all types are convertible to `object`), while `FixedSizeCollection<T>` provides type safety by allowing the type parameter to dictate what types of objects are permitted. Notice also that the properties have no associated private backing field declared for storing the array. This is an example of using the new *automatically implemented properties* feature that was originally introduced in C# 3.0. Under the covers, the C# compiler is creating a storage element of the property's type, but you don't have to write the code for the property storage anymore if you don't have specific code that has to execute when accessing the properties. To make the property read-only, simply mark the `set;` declaration `private`.

The next difference is visible in the method declarations of `AddItem` and `GetItem`. `AddItem` now takes a parameter of type `T`, whereas in `FixedSizeCollection`, it took a parameter of type `object`. `GetItem` now returns a value of type `T`, whereas in `FixedSizeCollection`, it returned a value of type `object`. These changes allow the methods in `FixedSizeCollection<T>` to use the instantiated type to store and retrieve the items in the array, instead of having to allow any `object` to be stored as in `FixedSizeCollection`:

```
    /// <summary>
    /// Add an item to the class whose type
    /// is determined by the instantiating type
    /// </summary>
    /// <param name="item">item to add</param>
    /// <returns>the zero-based index of the item added</returns>
```

```
public int AddItem(T item)
{
    if (this.ItemCount < this.Items.Length)
    {
        this.Items[this.ItemCount] = item;
        return this.ItemCount++;
    }
    else
        throw new Exception("Item queue is full");
}

/// <summary>
/// Get an item from the class
/// </summary>
/// <param name="index">the zero-based index of the item to get</param>
/// <returns>an item of the instantiating type</returns>
public T GetItem(int index)
{

    if (index >= this.Items.Length &&
        index >= 0)
        throw new ArgumentOutOfRangeException("index");

    return this.Items[index];
}
```

This provides a few advantages, first and foremost of which is the type safety provided by `FixedSizeCollection<T>` for items in the array. It was possible to write code like this in `FixedSizeCollection`:

```
// Regular class
FixedSizeCollection C = new FixedSizeCollection(5);
Console.WriteLine(C);

string s1 = "s1";
string s2 = "s2";
string s3 = "s3";
int i1 = 1;

// Add to the fixed size collection (as object).
C.AddItem(s1);
C.AddItem(s2);
C.AddItem(s3);
// Add an int to the string array, perfectly OK.
C.AddItem(i1);
```

But `FixedSizeCollection<T>` will give a compiler error if you try the same thing:

```
// Generic class
FixedSizeCollection<string> gC = new FixedSizeCollection<string>(5);
Console.WriteLine(gC);

string s1 = "s1";
```

```
string s2 = "s2";
string s3 = "s3";
int i1 = 1;
// Add to the generic class (as string).
gC.AddItem(s1);
gC.AddItem(s2);
gC.AddItem(s3);
// Try to add an int to the string instance, denied by compiler.
// error CS1503: Argument '1': cannot convert from 'int' to 'string'
//gC.AddItem(i1);
```

Having the compiler prevent this before it can become the source of runtime bugs is a very good idea.

It may not be immediately noticeable, but the integer is actually boxed when it is added to the object array in `FixedSizeCollection`, as you can see in the IL for the call to `GetItem` on `FixedSizeCollection`:

```
IL_0177: ldloc.2
IL_0178: ldloc.s i1
IL_017a: box [mscorlib]System.Int32
IL_017f: callvirt instance int32
       CSharpRecipes.ClassesAndGenerics/FixedSizeCollection::AddItem(object)
```

This boxing turns the `int`, which is a value type, into a reference type (`object`) for storage in the array. This requires you to do extra work to store value types in the `object` array.

You'll encounter another problem when you go to retrieve an item from the class in the `FixedSizeCollection` implementation. Take a look at how `FixedSizeCollection.GetItem` retrieves an item:

```
// Hold the retrieved string.
string sHolder;

// Have to cast or get error CS0266:
// Cannot implicitly convert type 'object' to 'string'
sHolder = (string)C.GetItem(1);
```

Since the item returned by `FixedSizeCollection.GetItem` is of type `object`, you need to cast it to a `string` in order to get what you hope is a `string` for index 1. It may not be a `string`—all you know for sure is that it's an `object`—but you have to cast it to a more specific type coming out so you can assign it properly.

These issues are both fixed by the `FixedSizeCollection<T>` implementation. Unlike with `FixedSizeCollection`, no unboxing is required in `FixedSizeCollection<T>`, since the return type of `GetItem` is the instantiated type, and the compiler enforces this by looking at the value being returned:

```
// Hold the retrieved string.
string sHolder;
```

```
int iHolder;

// No cast necessary
sHolder = gC.GetItem(1);

// Try to get a string into an int.
// error CS0029: Cannot implicitly convert type 'string' to 'int'
//iHolder = gC.GetItem(1);
```

To see one other difference between the two types, instantiate a few instances of each like so:

```
// Regular class
FixedSizeCollection A = new FixedSizeCollection(5);
Console.WriteLine(A);
FixedSizeCollection B = new FixedSizeCollection(5);
Console.WriteLine(B);
FixedSizeCollection C = new FixedSizeCollection(5);
Console.WriteLine(C);

// generic class
FixedSizeCollection<bool> gA = new FixedSizeCollection<bool>(5);
Console.WriteLine(gA);
FixedSizeCollection<int> gB = new FixedSizeCollection<int>(5);
Console.WriteLine(gB);
FixedSizeCollection<string> gC = new FixedSizeCollection<string>(5);
Console.WriteLine(gC);
FixedSizeCollection<string> gD = new FixedSizeCollection<string>(5);
Console.WriteLine(gD);
```

The output from the preceding code shows this:

```
There are 1 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection
    and this instance contains 0 items...
There are 2 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection
    and this instance contains 0 items...
There are 3 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection
    and this instance contains 0 items...
There are 1 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection'1
    [System.Boolean] and this instance contains 0 items...
There are 1 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection'1
    [System.Int32] and this instance contains 0 items...
There are 1 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection'1
    [System.String] and this instance contains 0 items...
There are 2 instances of CSharpRecipes.ClassesAndGenerics+FixedSizeCollection'1
    [System.String] and this instance contains 0 items...
```

Discussion

The type parameters in generics allow you to create type-safe code without knowing the final type you will be working with. In many instances, you want the types to have

certain characteristics, in which case you place constraints on the type (see Recipe 1.12). Methods can have generic type parameters whether or not the class itself does.

Notice that while `FixedSizeCollection` has three instances, `FixedSizeCollection<T>` has one instance in which it was declared with `bool` as the type, one instance in which `int` was the type, and two instances in which `string` was the type. This means that, while there is one .NET `Type` object created for each nongeneric class, there is one .NET `Type` object for every constructed type of a generic class.

`FixedSizeCollection` has three instances in the example code because `FixedSizeCollection` has only one type that is maintained by the CLR. With generics, one type is maintained for each combination of the class template and the type arguments passed when a type instance is constructed. In other words, you get one .NET type for `FixedSizeCollection<bool>`, one .NET type for `FixedSizeCollection<int>`, and a third .NET type for `FixedSizeCollection<string>`.

The static `InstanceCount` property helps to illustrate this point, as static properties of a class are actually connected to the type that the CLR hangs on to. The CLR creates any given type only once and then maintains it until the `AppDomain` unloads. This is why the output from the calls to `ToString` on these objects shows that the count is `3` for `FixedSizeCollection` (as there is truly only one of these) and `1` or `2` for the `FixedSizeCollection<T>` types.

See Also

The "Generic Type Parameters" and "Generic Classes" topics in the MSDN documentation.

1.11 Reversing the Contents of a Sorted List

Problem

You want to be able to reverse the contents of a sorted list of items while maintaining the ability to access them in both array and list styles like `SortedList` and the generic `SortedList<T>` classes provide. Neither `SortedList` nor `SortedList<T>` provides a direct way to accomplish this without reloading the list.

Solution

Use *LINQ to Objects* to query the `SortedList<T>` and apply a descending order to the information in the list. After you instantiate a `SortedList<TKey, TValue>`, the key of which is an `int` and the value of which is a `string`, a series of unordered numbers and their text representations are inserted into the list. Those items are then displayed:

```
SortedList<int, string> data = new SortedList<int, string>()
    { [2]="two", [5]="five", [3]="three", [1]="one" };

foreach (KeyValuePair<int, string> kvp in data)
{
    Console.WriteLine($"\t {kvp.Key}\t{kvp.Value}");
}
```

The output for the list is shown sorted in ascending order (the default):

```
1    one
2    two
3    three
5    five
```

Now you reverse the sort order by creating a query using LINQ to Objects and setting the `orderby` clause to `descending`. The results are then displayed from the query result set:

```
// query ordering by descending
var query = from d in data
            orderby d.Key descending
            select d;

foreach (KeyValuePair<int, string> kvp in query)
{
    Console.WriteLine($"\t {kvp.Key}\t{kvp.Value}");
}
```

This time the output is in descending order:

```
5    five
3    three
2    two
1    one
```

When you add a new item to the list, it is added in the ascending sort order, but by querying again after adding all of the items, you keep the ordering of the list intact:

```
data.Add(4, "four");

// requery ordering by descending
query = from d in data
        orderby d.Key descending
        select d;

foreach (KeyValuePair<int, string> kvp in query)
{
    Console.WriteLine($"\t {kvp.Key}\t{kvp.Value}");
}
Console.WriteLine("");

// Just go against the original list for ascending
foreach (KeyValuePair<int, string> kvp in data)
```

```
{
    Console.WriteLine($"\t {kvp.Key}\t{kvp.Value}");
}
```

You can see the output in both descending and ascending order with the new item:

```
5    five
4    four
3    three
2    two
1    one

1    one
2    two
3    three
4    four
5    five
```

Discussion

A `SortedList` blends array and list syntax to allow you to access the data in either format, which can be a handy thing to do. The data is accessible as key/value pairs or directly by index and will not allow you to add duplicate keys. In addition, values that are reference or nullable types can be `null`, but keys cannot. You can iterate over the items using a `foreach` loop, with `KeyValuePair` being the type returned. While accessing elements of the `SortedList<T>`, you may only read from them. The usual iterator syntax prohibits you from updating or deleting elements of the list while reading, as it will invalidate the iterator.

The `orderby` clause in the query orders the result set of the query either in `ascending` (the default) or `descending` order. This sorting is accomplished through use of the default comparer for the element type, so you can alter it by overriding the `Equals` method for elements that are custom classes. You can specify multiple keys for the `orderby` clause, which has the effect of nesting the sort order, such as sorting by "last name" and then "first name."

See Also

The "SortedList," "Generic KeyValuePair Structure," and "Generic SortedList" topics in the MSDN documentation.

1.12 Constraining Type Arguments

Problem

Your generic type needs to be created with a type argument that must support the members of a particular interface, such as `IDisposable`.

Solution

Use constraints to force the type arguments of a generic type to be of a type that implements one or more particular interfaces:

```
public class DisposableList<T> : IList<T>
    where T : class, IDisposable
{
    private List<T> _items = new List<T>();

    // Private method that will dispose of items in the list
    private void Delete(T item) => item.Dispose();

    // IList<T> Members
    public int IndexOf(T item) => _items.IndexOf(item);

    public void Insert(int index, T item) => _items.Insert(index, item);

    public T this[int index]
    {
        get    {return (_items[index]);}
        set    {_items[index] = value;}
    }

    public void RemoveAt(int index)
    {
        Delete(this[index]);
        _items.RemoveAt(index);
    }

    // ICollection<T> Members
    public void Add(T item) => _items.Add(item);

    public bool Contains(T item) => _items.Contains(item);

    public void CopyTo(T[] array, int arrayIndex) =>
                    _items.CopyTo(array, arrayIndex);

    public int Count  => _items.Count;

    public bool IsReadOnly  => false;

    // IEnumerable<T> Members
    public IEnumerator<T> GetEnumerator()=> _items.GetEnumerator();

    // IEnumerable Members
    IEnumerator IEnumerable.GetEnumerator()=> _items.GetEnumerator();

    // Other members
    public void Clear()
    {
        for (int index = 0; index < _items.Count; index++)
```

```
            {
                Delete(_items[index]);
            }

            _items.Clear();
        }

        public bool Remove(T item)
        {
            int index = _items.IndexOf(item);

            if (index >= 0)
            {
                Delete(_items[index]);
                _items.RemoveAt(index);

                return (true);
            }
            else
            {
                return (false);
            }
        }
    }
```

This `DisposableList` class allows only an object that implements `IDisposable` to be passed in as a type argument to this class. The reason for this is that whenever an object is removed from a `DisposableList` object, the `Dispose` method is always called on that object. This allows you to transparently handle the management of any object stored within this `DisposableList` object.

The following code exercises a `DisposableList` object:

```
    public static void TestDisposableListCls()
    {
        DisposableList<StreamReader> dl = new DisposableList<StreamReader>();

         // Create a few test objects.
         StreamReader tr1 = new StreamReader("C:\\Windows\\system.ini");
         StreamReader tr2 = new StreamReader("c:\\Windows\\vmgcoinstall.log");
         StreamReader tr3 = new StreamReader("c:\\Windows\\Starter.xml");

         // Add the test object to the DisposableList.
         dl.Add(tr1);
         dl.Insert(0, tr2);
         dl.Add(tr3);

         foreach(StreamReader sr in dl)
         {
             Console.WriteLine($"sr.ReadLine() == {sr.ReadLine()}");
         }
```

```
        // Call Dispose before any of the disposable objects are
        // removed from the DisposableList.
        dl.RemoveAt(0);
        dl.Remove(tr1);
        dl.Clear();
    }
```

Discussion

The `where` keyword is used to constrain a type parameter to accept only arguments that satisfy the given constraint. For example, the `DisposableList` has the constraint that any type argument `T` must implement the `IDisposable` interface:

```
public class DisposableList<T> : IList<T>
        where T : IDisposable
```

This means that the following code will compile successfully:

```
DisposableList<StreamReader> dl = new DisposableList<StreamReader>();
```

but the following code will not:

```
DisposableList<string> dl = new DisposableList<string>();
```

This is because the `string` type does not implement the `IDisposable` interface, and the `StreamReader` type does.

Other constraints on the type argument are allowed, in addition to requiring one or more specific interfaces to be implemented. You can force a type argument to be inherited from a specific base class, such as the `TextReader` class:

```
public class DisposableList<T> : IList<T>
       where T : System.IO.TextReader, IDisposable
```

You can also determine if the type argument is narrowed down to only value types or only reference types. The following class declaration is constrained to using only value types:

```
public class DisposableList<T> : IList<T>
        where T : struct
```

This class declaration is constrained to only reference types:

```
public class DisposableList<T> : IList<T>
        where T : class
```

In addition, you can also require any type argument to implement a public default constructor:

```
public class DisposableList<T> : IList<T>
       where T : IDisposable, new()
```

Using constraints allows you to write generic types that accept a narrower set of available type arguments. If the `IDisposable` constraint is omitted in the Solution for this

recipe, a compile-time error will occur. This is because not all of the types that can be used as the type argument for the `DisposableList` class will implement the `IDisposable` interface. If you skip this compile-time check, a `DisposableList` object may contain objects that do not have a public no-argument `Dispose` method. In this case, a runtime exception will occur. Generics and constraints in particular force strict type checking of the class-type arguments and allow you to catch these problems at compile time rather than at runtime.

See Also

The "where Keyword" topic in the MSDN documentation.

1.13 Initializing Generic Variables to Their Default Values

Problem

You have a generic class that contains a variable of the same type as the type parameter defined by the class itself. Upon construction of your generic object, you want that variable to be initialized to its default value.

Solution

Simply use the `default` keyword to initialize that variable to its default value:

```
public class DefaultValueExample<T>
{
    T data = default(T);

    public bool IsDefaultData()
    {
        T temp = default(T);

        if (temp.Equals(data))
        {
            return (true);
        }
        else
        {
            return (false);
        }
    }

    public void SetData(T val) => data = value;
}
```

The code to exercise this class is shown here:

```
public static void ShowSettingFieldsToDefaults()
{
```

```
        DefaultValueExample<int> dv = new DefaultValueExample<int>();

        // Check if the data is set to its default value; true is returned.
        bool isDefault = dv.IsDefaultData();
        Console.WriteLine($"Initial data: {isDefault}");
        // Set data.
        dv.SetData(100);
        // Check again, this time a false is returned.
        isDefault = dv.IsDefaultData();
        Console.WriteLine($"Set data: {isDefault}");
    }
```

The first call to `IsDefaultData` returns `true`, while the second returns `false`. The output is shown here:

```
Initial data: True
Set data: False
```

Discussion

When initializing a variable of the same type parameter as the generic class, you cannot just set that variable to `null`. What if the type parameter is a value type such as an `int` or `char`? This will not work because value types cannot be `null`. You may be thinking that a `nullable` type such as `long?` or `Nullable<long>` can be set to `null` (see "Using Nullable Types (C# Programming Guide)" in the MSDN documentation for more on nullable types). However, the compiler has no way of knowing what type argument the user will use to construct the type.

The `default` keyword allows you to tell the compiler that at compile time the default value of this variable should be used. If the type argument supplied is a numeric value (e.g., `int`, `long`, `decimal`), then the default value is `0`. If the type argument supplied is a reference type, then the default value is `null`. If the type argument supplied is a struct, then you determine the default value by initializing each member field to its default value.

See Also

The "Using Nullable Types (C# Programming Guide)" and "default Keyword in Generic Code" topics in the MSDN documentation.

1.14 Adding Hooks to Generated Entities

Problem

You have a process to generate your partial class business entity definitions, and you want to add a lightweight notification mechanism.

Solution

Use partial methods to add hooks in the generated code for the business entities.

The process to generate the entities may be from UML (Unified Modeling Language), a data set, or some other object-modeling facility, but when the code is generated as partial classes, add partial method hooks into the templates for the properties that call a `ChangingProperty` partial method, as shown in the `GeneratedEntity` class:

```
public partial class GeneratedEntity
{
    public GeneratedEntity(string entityName)
    {
        this.EntityName = entityName;
    }

    partial void ChangingProperty(string name, string originalValue,
                                  stringnewValue);

    public string EntityName { get; }
    private string _FirstName;
    public string FirstName
    {
        get { return _FirstName; }
        set
        {
            ChangingProperty("FirstName",_FirstName,value);
            _FirstName = value;
        }
    }
    private string _State;
    public string State
    {
        get { return _State; }
        set
        {
            ChangingProperty("State",_State,value);
            _State = value;
        }
    }
}
```

The `GeneratedEntity` has two properties, `FirstName` and `State`. Notice each of these properties has the same boilerplate code that calls the `ChangingProperty` method with the name of the property, the original, and the new values. If the generated class is used at this point, the `ChangingProperty` declaration and method will be removed by the compiler, as there is no implementation for `ChangingProperty`. If an implementation is supplied to report on property changes as shown here, then all of the partial method code for `ChangingProperty` will be retained and executed:

```
public partial class GeneratedEntity
{
    partial void ChangingProperty(string name, string originalValue,
                                  string newValue)
    {
        Console.WriteLine($"Changed property ({name}) for entity " +
                          $"{this.EntityName} from " +
                          $"{originalValue} to {newValue}");
    }
}
```

Discussion

When using partial methods, be aware of the following:

- You indicate a partial method with the `partial` modifier.
- Partial methods can be declared only in partial classes.
- Partial methods might have only a declaration and no body.
- From a signature standpoint, a partial method can have arguments, require a void return value, and must not have any access modifier, and `partial` implies that this method is private and can be static, generic, or unsafe.
- For generic partial methods, constraints must be repeated on the declaring and implementing versions.
- A partial method may not implement an interface member since interface members must be public.
- None of the virtual, abstract, override, new, sealed, or extern modifiers may be used.
- Arguments to a partial method cannot use `out`, but they can use `ref`.

Partial methods are similar to conditional methods, except that the method definition is always present in conditional methods, even when the condition is not met. Partial methods do not retain the method definition if there is no matching implementation. The code in the Solution could be used like this:

```
public static void TestPartialMethods()
{
    Console.WriteLine("Start entity work");
    GeneratedEntity entity = new GeneratedEntity("FirstEntity");
    entity.FirstName = "Bob";
    entity.State = "NH";
    GeneratedEntity secondEntity = new GeneratedEntity("SecondEntity");
    entity.FirstName = "Jay";
    secondEntity.FirstName = "Steve";
    secondEntity.State = "MA";
    entity.FirstName = "Barry";
```

```
        secondEntity.State = "WA";
        secondEntity.FirstName = "Matt";
        Console.WriteLine("End entity work");
    }
```

to produce the following output when the `ChangingProperty` implementation is provided:

```
Start entity work
Changed property (FirstName) for entity FirstEntity from to Bob
Changed property (State) for entity FirstEntity from to NH
Changed property (FirstName) for entity FirstEntity from Bob to Jay
Changed property (FirstName) for entity SecondEntity from to Steve
Changed property (State) for entity SecondEntity from to MA
Changed property (FirstName) for entity FirstEntity from Jay to Barry
Changed property (State) for entity SecondEntity from MA to WA
Changed property (FirstName) for entity SecondEntity from Steve to Matt
End entity work
```

or to produce the following output when the `ChangingProperty` implementation is *not* provided:

```
Start entity work
End entity work
```

See Also

The "Partial Methods" and "partial (Method)" topics in the MSDN documentation.

1.15 Controlling How a Delegate Fires Within a Multicast Delegate

Problem

You have combined multiple delegates to create a multicast delegate. When this multicast delegate is invoked, each delegate within it is invoked in turn. You need to exert more control over the order in which each delegate is invoked, firing only a subset of delegates, or firing each delegate based on the success or failure of previous delegates. Additionally, you need to be able to handle the return value of each delegate separately.

Solution

Use the `GetInvocationList` method to obtain an array of `Delegate` objects. Next, iterate over this array using a `for` (if enumerating in a nonstandard order) or `foreach` (for enumerating in a standard order) loop. You can then invoke each `Delegate` object in the array individually and, optionally, retrieve each delegate's unique return value.

In C#, all delegate types support multicast—that is, any delegate instance can invoke multiple methods each time the instance is invoked if it has been set up to do so. In this recipe, we use the term *multicast* to describe a delegate that has been set up to invoke multiple methods.

The following method creates a multicast delegate called `allInstances` and then uses `GetInvocationList` to allow each delegate to be invoked individually, in reverse order. The `Func<int>` generic delegate is used to create delegate instances that return an `int`:

```
public static void InvokeInReverse()
{
    Func<int> myDelegateInstance1 = TestInvokeIntReturn.Method1;
    Func<int> myDelegateInstance2 = TestInvokeIntReturn.Method2;
    Func<int> myDelegateInstance3 = TestInvokeIntReturn.Method3;

    Func<int> allInstances =
            myDelegateInstance1 +
            myDelegateInstance2 +
            myDelegateInstance3;

    Console.WriteLine("Fire delegates in reverse");
    Delegate[] delegateList = allInstances.GetInvocationList();
    foreach (Func<int> instance in delegateList.Reverse())
    {
        instance();
    }
}
```

Note that to roll over the delegate list retrieved using `GetInvocationList`, we use the `IEnumerable<T>` extension method `Reverse` so that we get the items in the opposite order of how the enumeration would normally produce them.

As the following methods demonstrate by firing every other delegate, you don't have to invoke all of the delegates in the list. `InvokeEveryOtherOperation` uses an extension method created here for `IEnumerable<T>` called `EveryOther` that will return only every other item from the enumeration.

If a unicast delegate was used and you called `GetInvocationList` on it, you will receive a list of one delegate instance.

```
public static void InvokeEveryOtherOperation()
{
    Func<int> myDelegateInstance1 = TestInvokeIntReturn.Method1;
    Func<int> myDelegateInstance2 = TestInvokeIntReturn.Method2;
    Func<int> myDelegateInstance3 = TestInvokeIntReturn.Method3;
```

```
        Func<int> allInstances = myDelegateInstance1 +
                                 myDelegateInstance2 +
                                 myDelegateInstance3;

        Delegate[] delegateList = allInstances.GetInvocationList();
        Console.WriteLine("Invoke every other delegate");
        foreach (Func<int> instance in delegateList.EveryOther())
        {
            // invoke the delegate
            int retVal = instance();
            Console.WriteLine($"Delegate returned {retVal}");
        }
    }

    static IEnumerable<T> EveryOther<T>(this IEnumerable<T> enumerable)
    {
        bool retNext = true;
        foreach (T t in enumerable)
        {
            if (retNext) yield return t;
            retNext = !retNext;
        }
    }
```

The following class contains each of the methods that will be called by the multicast delegate `allInstances`:

```
public class TestInvokeIntReturn
{
    public static int Method1()
    {
        Console.WriteLine("Invoked Method1");
        return 1;
    }

    public static int Method2()
    {
        Console.WriteLine("Invoked Method2");
        return 2;
    }

    public static int Method3()
    {
        Console.WriteLine("Invoked Method3");
        return 3;
    }
}
```

You can also specify whether to continue firing delegates in the list based on the return value of the currently firing delegate. The following method fires each delegate, stopping only when a delegate returns a `false` value:

```
public static void InvokeWithTest()
{
    Func<bool> myDelegateInstanceBool1 = TestInvokeBoolReturn.Method1;
    Func<bool> myDelegateInstanceBool2 = TestInvokeBoolReturn.Method2;
    Func<bool> myDelegateInstanceBool3 = TestInvokeBoolReturn.Method3;

    Func<bool> allInstancesBool =
            myDelegateInstanceBool1 +
            myDelegateInstanceBool2 +
            myDelegateInstanceBool3;

    Console.WriteLine(
        "Invoke individually (Call based on previous return value):");
    foreach (Func<bool> instance in allInstancesBool.GetInvocationList())
    {
        if (!instance())
            break;
    }
}
```

The following class contains each of the methods that will be called by the multicast delegate `allInstancesBool`:

```
public class TestInvokeBoolReturn
{
    public static bool Method1()
    {
        Console.WriteLine("Invoked Method1");
        return true;
    }

    public static bool Method2()
    {
        Console.WriteLine("Invoked Method2");
        return false;
    }

    public static bool Method3()
    {
        Console.WriteLine("Invoked Method3");
        return true;
    }
}
```

Discussion

A delegate, when called, will invoke all delegates stored within its *invocation list*. These delegates are usually invoked sequentially from the first to the last one added. Using the `GetInvocationList` method of the `MulticastDelegate` class, you can obtain each delegate in the invocation list of a multicast delegate. This method accepts no parameters and returns an array of `Delegate` objects that corresponds to

the invocation list of the delegate on which this method was called. The returned `Delegate` array contains the delegates of the invocation list in the order in which they would normally be called; that is, the first element in the `Delegate` array contains the `Delegate` object that is normally called first.

This application of the `GetInvocationList` method enables you to control exactly when and how the delegates in a multicast delegate are invoked, and to prevent the continued invocation of delegates when one delegate fails. This ability is important if each delegate is manipulating data, and one of the delegates fails in its duties but does not throw an exception. If one delegate fails in its duties and the remaining delegates rely on all previous delegates to succeed, you must quit invoking delegates at the point of failure.

This recipe handles a delegate failure more efficiently and also provides more flexibility in dealing with these errors. For example, you can write logic to specify which delegates are to be invoked, based on the return values of previously invoked delegates. The following method creates a multicast delegate called `All` and then uses `GetInvocationList` to fire each delegate individually. After firing each delegate, it captures the return value:

```
public static void TestIndividualInvokesReturnValue()
{
    Func<int> myDelegateInstance1 = TestInvokeIntReturn.Method1;
    Func<int> myDelegateInstance2 = TestInvokeIntReturn.Method2;
    Func<int> myDelegateInstance3 = TestInvokeIntReturn.Method3;

    Func<int> allInstances =
            myDelegateInstance1 +
            myDelegateInstance2 +
            myDelegateInstance3;

    Console.WriteLine("Invoke individually (Obtain each return value):");
    foreach (Func<int> instance in allInstances.GetInvocationList())
    {
        int retVal = instance();
        Console.WriteLine($"\tOutput: {retVal}");
    }
}
```

One quirk of a multicast delegate is that if any or all delegates within its invocation list return a value, only the value of the last invoked delegate is returned; all others are lost. This loss can become annoying—or worse, if your code requires these return values. Consider a case in which the `allInstances` delegate was invoked normally, as in the following code:

```
retVal = allInstances();
Console.WriteLine(retVal);
```

The value 3 would be displayed because `Method3` was the last method invoked by the `allInstances` delegate. None of the other return values would be captured.

By using the `GetInvocationList` method of the `MulticastDelegate` class, you can get around this limitation. This method returns an array of `Delegate` objects that can each be invoked separately. Note that this method does not invoke each delegate; it simply returns an array of them to the caller. By invoking each delegate separately, you can retrieve each return value from each invoked delegate.

Note that any `out` or `ref` parameters will also be lost when a multicast delegate is invoked. This recipe allows you to obtain the `out` and/or `ref` parameters of each invoked delegate within the multicast delegate.

However, you still need to be aware that any unhandled exceptions emanating from one of these invoked delegates will be bubbled up to the method `TestIndividualInvokesReturnValue` presented in this recipe. If an exception does occur in a delegate that is invoked from within a multicast delegate and that exception is unhandled, any remaining delegates are not invoked. This is the expected behavior of a multicast delegate. However, in some circumstances, you'd like to be able to handle exceptions thrown from individual delegates and then determine at that point whether to continue invoking the remaining delegates.

An unhandled exception will force the invocation of delegates to cease. Exceptions should be used only for exceptional circumstances, not for control flow.

In the following `TestIndividualInvokesExceptions` method, if an exception is caught it is logged to the event log and displayed, and then the code continues to invoke delegates:

```
public static void TestIndividualInvokesExceptions()
{

    Func<int> myDelegateInstance1 = TestInvokeIntReturn.Method1;
    Func<int> myDelegateInstance2 = TestInvokeIntReturn.Method2;
    Func<int> myDelegateInstance3 = TestInvokeIntReturn.Method3;

    Func<int> allInstances =
            myDelegateInstance1 +
            myDelegateInstance2 +
            myDelegateInstance3;

    Console.WriteLine("Invoke individually (handle exceptions):");

    // Create an instance of a wrapper exception to hold any exceptions
    // encountered during the invocations of the delegate instances
```

```
        List<Exception> invocationExceptions = new List<Exception>();

        foreach (Func<int> instance in allInstances.GetInvocationList())
        {
            try
            {
                int retVal = instance();
                Console.WriteLine($"\tOutput: {retVal}");
            }
            catch (Exception ex)
            {
                // Display and log the exception and continue
                Console.WriteLine(ex.ToString());
                EventLog myLog = new EventLog();
                myLog.Source = "MyApplicationSource";
                myLog.WriteEntry(
                    $"Failure invoking {instance.Method.Name} with error " +
                    $"{ex.ToString()}",
                    EventLogEntryType.Error);
                // add this exception to the list
                invocationExceptions.Add(ex);
            }
        }
        // if we caught any exceptions along the way, throw our
        // wrapper exception with all of them in it.
        if (invocationExceptions.Count > 0)
        {
            throw new MulticastInvocationException(invocationExceptions);
        }
    }
```

The `MulticastInvocationException` class, used in the previous code, can have multiple exceptions added to it. It exposes a `ReadOnlyCollection<Exception>` through the `InvocationExceptions` property, as shown here:

```
    [Serializable]
    public class MulticastInvocationException : Exception
    {
        private List<Exception> _invocationExceptions;

        public MulticastInvocationException()
            : base()
        {
        }

        public MulticastInvocationException(
        IEnumerable<Exception> invocationExceptions)
        {
            _invocationExceptions = new List<Exception>(invocationExceptions);
        }

        public MulticastInvocationException(string message)
```

```
        : base(message)
    {
    }

    public MulticastInvocationException(string message, Exception innerException)
        :base(message,innerException)
    {
    }

    protected MulticastInvocationException(SerializationInfo info,
        StreamingContext
               context) :
        base(info, context)
    {
        _invocationExceptions =
            (List<Exception>)info.GetValue("InvocationExceptions",
                typeof(List<Exception>));
    }

    [SecurityPermissionAttribute(SecurityAction.Demand,
        SerializationFormatter = true)]
    public override void GetObjectData(
       SerializationInfo info, StreamingContext context)
    {
        info.AddValue("InvocationExceptions", this.InvocationExceptions);
        base.GetObjectData(info, context);
    }

    public ReadOnlyCollection<Exception> InvocationExceptions =>
                new ReadOnlyCollection<Exception>(_invocationExceptions);
}
```

This strategy allows for as fine-grained handling of exceptions as you need. One option is to store all of the exceptions that occur during delegate processing, and then wrap all of the exceptions encountered during processing in a custom exception. After processing completes, throw the custom exception.

By adding a `finally` block to this `try-catch` block, you can be assured that code within this `finally` block is executed after every delegate returns. This technique is useful if you want to interleave code between calls to delegates, such as code to clean up objects that are not needed or code to verify that each delegate left the data it touched in a stable state.

See Also

The "Delegate Class" and "Delegate.GetInvocationList Method" topics in the MSDN documentation.

1.16 Using Closures in C#

Problem

You want to associate a small amount of state with some behavior without going to the trouble of building a new class.

Solution

Use lambda expressions to implement *closures*, functions that capture the state of the environment that is in scope where they are declared. Put more simply, closures are current state plus some behavior that can read and modify that state. Lambda expressions have the capacity to capture external variables and extend their lifetime, which makes closures possible in C#.

For more information on lambda expressions, see the introduction to Chapter 4.

As an example, you will build a quick reporting system that tracks sales personnel and their revenue production versus commissions. The closure behavior is that you can build one bit of code that does the commission calculations per quarter and works on every salesperson.

First, you have to define your sales personnel:

```
class SalesPerson
{
    // CTOR's
    public SalesPerson()
    {
    }

    public SalesPerson(string name,
                       decimal annualQuota,
                       decimal commissionRate)
    {
        this.Name = name;
        this.AnnualQuota = annualQuota;
        this.CommissionRate = commissionRate;
    }

    // Private Members
    decimal _commission;

    // Properties
```

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

    public decimal AnnualQuota { get; set; }

    public decimal CommissionRate { get; set; }

    public decimal Commission
    {
        get { return _commission; }
        set
        {
            _commission = value;
            this.TotalCommission += _commission;
        }
   }
   public decimal TotalCommission {get; private set; }
}
```

Sales personnel have a name, an annual quota, a commission rate for sales, and some storage for holding a quarterly commission and a total commission. Now that you have something to work with, let's write a bit of code to do the work of calculating the commissions:

```
delegate void CalculateEarnings(SalesPerson sp);

static CalculateEarnings GetEarningsCalculator(decimal quarterlySales,
                                  decimal bonusRate)
{
     return salesPerson =>
     {
         // Figure out the salesperson's quota for the quarter.
         decimal quarterlyQuota = (salesPerson.AnnualQuota / 4);
         // Did he make quota for the quarter?
         if (quarterlySales < quarterlyQuota)
         {
             // Didn't make quota, no commission
             salesPerson.Commission = 0;
         }
         // Check for bonus-level performance (200% of quota).
         else if (quarterlySales > (quarterlyQuota * 2.0m))
         {
             decimal baseCommission = quarterlyQuota *
             salesPerson.CommissionRate;
             salesPerson.Commission = (baseCommission +
                     ((quarterlySales - quarterlyQuota) *
                     (salesPerson.CommissionRate * (1 + bonusRate))));
         }
         else // Just regular commission
         {
             salesPerson.Commission =
                salesPerson.CommissionRate * quarterlySales;
         }
```

```
        };
    }
```

You've declared the delegate type as `CalculateEarnings`, and it takes a `SalesPerson` type. You have a factory method to construct an instance of this delegate for you, called `GetEarningsCalculator`, which creates a lambda expression to calculate the `SalesPerson`'s commission and returns a `CalculateEarnings` instantiation.

To get started, create your array of salespeople:

```
// set up the salespeople...
SalesPerson[] salesPeople = {
    new SalesPerson { Name="Chas", AnnualQuota=100000m, CommissionRate=0.10m },
    new SalesPerson { Name="Ray", AnnualQuota=200000m, CommissionRate=0.025m },
    new SalesPerson { Name="Biff", AnnualQuota=50000m, CommissionRate=0.001m }};
```

Then set up the earnings calculators based on quarterly earnings:

```
public class QuarterlyEarning
{
    public string Name { get; set; }
    public decimal Earnings { get; set; }
    public decimal Rate { get; set; }
}
QuarterlyEarning[] quarterlyEarnings =
           { new QuarterlyEarning(){ Name="Q1", Earnings = 65000m, Rate = 0.1m },
             new QuarterlyEarning(){ Name="Q2", Earnings = 20000m, Rate = 0.1m },
             new QuarterlyEarning(){ Name="Q3", Earnings = 37000m, Rate = 0.1m },
             new QuarterlyEarning(){ Name="Q4", Earnings = 110000m, Rate = 0.15m}
           };

var calculators = from e in quarterlyEarnings
                  select new
                  {
                      Calculator =
                          GetEarningsCalculator(e.Earnings, e.Rate),
                      QuarterlyEarning = e
                  };
```

Finally, run the numbers for each quarter for all the salespeople, and then you can generate the annual report from this data by calling `WriteCommissionReport`. This will tell the executives which sales personnel are worth keeping:

```
decimal annualEarnings = 0;
foreach (var c in calculators)
{
    WriteQuarterlyReport(c.QuarterlyEarning.Name,
        c.QuarterlyEarning.Earnings, c.Calculator, salesPeople);
    annualEarnings += c.QuarterlyEarning.Earnings;
}

// Let's see who is worth keeping...
WriteCommissionReport(annualEarnings, salesPeople);
```

`WriteQuarterlyReport` invokes the `CalculateEarnings` lambda expression implementation (`eCalc`) for every `SalesPerson` and modifies the state to assign quarterly commission values based on the commission rates for each one:

```
static void WriteQuarterlyReport(string quarter,
                          decimal quarterlySales,
                          CalculateEarnings eCalc,
                          SalesPerson[] salesPeople)
{
    Console.WriteLine($"{quarter} Sales Earnings on Quarterly Sales of
                      {quarterlySales.ToString("C")}:");
    foreach (SalesPerson salesPerson in salesPeople)
    {
        // Calc commission
        eCalc(salesPerson);
        // Report
        Console.WriteLine($"\tSales person {salesPerson.Name} " +
                          "made a commission of : " +
                         $"{salesPerson.Commission.ToString("C")}");
    }
}
```

`WriteCommissionReport` checks the revenue earned by the individual salesperson against his commission, and if his commission is more than 20 percent of the revenue he generated, you recommend action be taken:

```
static void WriteCommissionReport(decimal annualEarnings,
                                  SalesPerson[] salesPeople)
{
    decimal revenueProduced = ((annualEarnings) / salesPeople.Length);
    Console.WriteLine("");

    Console.WriteLine($"Annual Earnings were {annualEarnings.ToString("C")}");
    Console.WriteLine("");
    var whoToCan = from salesPerson in salesPeople
                   select new
                   {
                       // if his commission is more than 20%
                       // of what he produced, can him
                       CanThem = (revenueProduced * 0.2m) <
                                   salesPerson.TotalCommission,
                       salesPerson.Name,
                       salesPerson.TotalCommission
                   };

    foreach (var salesPersonInfo in whoToCan)
    {
        Console.WriteLine($"\t\tPaid {salesPersonInfo.Name} " +
                 $"{salesPersonInfo.TotalCommission.ToString("C")} to produce" +
                 $"{revenueProduced.ToString("C")}");
        if (salesPersonInfo.CanThem)
        {
```

```
            Console.WriteLine($"\t\t\tFIRE {salesPersonInfo.Name}!");
        }
    }
}
```

The output for your revenue- and commission-tracking program is listed here for your enjoyment:

```
Q1 Sales Earnings on Quarterly Sales of $65,000.00:
         SalesPerson Chas made a commission of : $6,900.00
         SalesPerson Ray made a commission of : $1,625.00
         SalesPerson Biff made a commission of : $70.25
Q2 Sales Earnings on Quarterly Sales of $20,000.00:
         SalesPerson Chas made a commission of : $0.00
         SalesPerson Ray made a commission of : $0.00
         SalesPerson Biff made a commission of : $20.00
Q3 Sales Earnings on Quarterly Sales of $37,000.00:
         SalesPerson Chas made a commission of : $3,700.00
         SalesPerson Ray made a commission of : $0.00
         SalesPerson Biff made a commission of : $39.45
Q4 Sales Earnings on Quarterly Sales of $110,000.00:
         SalesPerson Chas made a commission of : $12,275.00
         SalesPerson Ray made a commission of : $2,975.00
         SalesPerson Biff made a commission of : $124.63

Annual Earnings were $232,000.00

    Paid Chas $22,875.00 to produce $77,333.33
        FIRE Chas!
    Paid Ray $4,600.00 to produce $77,333.33
    Paid Biff $254.33 to produce $77,333.33
```

Discussion

One of the best descriptions of closures in C# is to think of an object as a set of methods associated with data and to think of a closure as a set of data associated with a function. If you need to have several different operations on the same data, an object approach may make more sense. These are two different angles on the same problem, and the type of problem you are solving will help you decide which is the right approach. It just depends on your inclination as to which way to go. There are times when 100% pure object-oriented programming can get tedious and is unnecessary, and closures are a nice way to solve some of those problems. The `SalesPerson` commission example presented here is a demonstration of what you can do with closures. It could have been done without them, but at the expense of writing more class and method code.

Closures were defined earlier, but there is a stricter definition that essentially implies that the behavior associated with the state should not be able to modify the state in order to be a true closure. We tend to agree more with the first definition, as it

expresses what a closure should be, not how it should be implemented, which seems too restrictive. Whether you choose to think of this approach as a neat side feature of lambda expressions or you feel it is worthy of being called a closure, it is another programming trick for your toolbox and should not be dismissed.

See Also

Recipe 1.17 and the "Lambda Expressions" topic in the MSDN documentation.

1.17 Performing Multiple Operations on a List Using Functors

Problem

You want to be able to perform multiple operations on an entire collection of objects at once, while keeping the operations functionally segmented.

Solution

Use a *functor* (or *function object*) as the vehicle for transforming the collection. A functor is any object that can be called as a function. Examples are a delegate, a function, a function pointer, or even an object that defines `operator` for us C/C++ converts.

Needing to perform multiple operations on a collection is a reasonably common scenario in software. Let's say that you have a stock portfolio with a bunch of stocks in it. Your `StockPortfolio` class would have a `List` of `Stock` objects and would allow you to add stocks:

```
public class StockPortfolio : IEnumerable<Stock>
{
    List<Stock> _stocks;

    public StockPortfolio()
    {
        _stocks = new List<Stock>();
    }

    public void Add(string ticker, double gainLoss)
    {
        _stocks.Add(new Stock() {Ticker=ticker, GainLoss=gainLoss});
    }

    public IEnumerable<Stock> GetWorstPerformers(int topNumber) =>
          _stocks.OrderBy((Stock stock) => stock.GainLoss).Take(topNumber);

    public void SellStocks(IEnumerable<Stock> stocks)
```

```
    {
        foreach(Stock s in stocks)
            _stocks.Remove(s);
    }

    public void PrintPortfolio(string title)
    {
        Console.WriteLine(title);
        _stocks.DisplayStocks();
    }

    #region IEnumerable<Stock> Members
    public IEnumerator<Stock> GetEnumerator() => _stocks.GetEnumerator();
    #endregion

    #region IEnumerable Members
    IEnumerator IEnumerable.GetEnumerator() => this.GetEnumerator();
    #endregion
}
```

The `Stock` class is rather simple. You just need a ticker symbol for the stock and its percentage of gain or loss:

```
public class Stock
{
    public double GainLoss { get; set; }
    public string Ticker { get; set; }
}
```

To use this `StockPortfolio`, you add a few stocks to it with gain/loss percentages and print out your starting portfolio. Once you have the portfolio, you want to get a list of the three worst-performing stocks, so you can improve your portfolio by selling them and print out your portfolio again:

```
StockPortfolio tech = new StockPortfolio() {
    {"OU81", -10.5},
    {"C#6VR", 2.0},
    {"PCKD", 12.3},
    {"BTML", 0.5},
    {"NOVB", -35.2},
    {"MGDCD", 15.7},
    {"GNRCS", 4.0},
    {"FNCTR", 9.16},
    {"LMBDA", 9.12},
    {"PCLS", 6.11}};

tech.PrintPortfolio("Starting Portfolio");
// sell the worst 3 performers
var worstPerformers = tech.GetWorstPerformers(3);
Console.WriteLine("Selling the worst performers:");
worstPerformers.DisplayStocks();
```

```
tech.SellStocks(worstPerformers);
tech.PrintPortfolio("After Selling Worst 3 Performers");
```

So far, nothing terribly interesting is happening. Let's take a look at how you figured out what the three worst performers were by looking at the internals of the `GetWorstPerformers` method:

```
public IEnumerable<Stock> GetWorstPerformers(int topNumber) => _stocks.OrderBy(
                          (Stock stock) => stock.GainLoss).Take(topNumber);
```

First you make sure the list is sorted with the worst-performing stocks at the front by calling the `OrderBy` extension method on `IEnumerable<T>`. The `OrderBy` method takes a lambda expression that provides the gain/loss percentage for comparison for the number of stocks indicated by `topNumber` in the `Take` extension method.

`GetWorstPerformers` returns an `IEnumerable<Stock>` full of the three worst performers. Since they aren't making any money, you should cash in and sell them. For your purposes, selling is simply removing them from the list of stocks in `StockPortfolio`. To accomplish this, you use yet another functor to iterate over the list of stocks handed to the `SellStocks` function (the list of worst-performing ones, in your case), and then remove that stock from the internal list that the `StockPortfolio` class maintains:

```
public void SellStocks(IEnumerable<Stock> stocks)
{
    foreach(Stock s in stocks)
        _stocks.Remove(s);
}
```

Discussion

Functors come in a few different flavors: a *generator* (a function with no parameters), a *unary function* (a function with one parameter), and a *binary function* (a function with two parameters). If the functor happens to return a Boolean value, then it gets an even more special naming convention: a unary function that returns a Boolean is called a *predicate*, and a binary function with a Boolean return is called a *binary predicate*. There are both `Predicate<T>` and `BinaryPredicate<T>` delegates defined in the Framework to facilitate these uses of functors.

The `List<T>` and `System.Array` classes take predicates (`Predicate<T>`, `BinaryPredicate<T>`), actions (`Action<T>`), comparisons (`Comparison<T>`), and converters (`Converter<T,U>`). This allows these collections to be operated on in a much more general way than was previously possible.

Thinking in terms of functors can be challenging at first, but once you put a bit of time into it, you start to see powerful possibilities open up before you. Any code you

can write once, debug once, and use many times is valuable, and functors can help you achieve that.

The output for the example is listed here:

```
Starting Portfolio
  (OU81) lost 10.5%
  (C#6VR) gained 2%
  (PCKD) gained 12.3%
  (BTML) gained 0.5%
  (NOVB) lost 35.2%
  (MGDCD) gained 15.7%
  (GNRCS) gained 4%
  (FNCTR) gained 9.16%
  (LMBDA) gained 9.12%
  (PCLS) gained 6.11%
Selling the worst performers:
  (NOVB) lost 35.2%
  (OU81) lost 10.5%
  (BTML) gained 0.5%
After Selling Worst 3 Performers
  (C#6VR) gained 2%
  (PCKD) gained 12.3%
  (MGDCD) gained 15.7%
  (GNRCS) gained 4%
  (FNCTR) gained 9.16%
  (LMBDA) gained 9.12%
  (PCLS) gained 6.11%
```

See Also

The "System.Collections.Generic.List<T>," "System.Linq.Enumerable Class," and "System.Array" topics in the MSDN documentation.

1.18 Controlling Struct Field Initialization

Problem

You need to be able to control the initialization of a struct depending on whether you want the struct to initialize all of its internal fields to their standard default values based on their type (e.g., `int` is initialized to `0` and `string` is initialized to an empty string), to a nonstandard set of default values, or to a set of predefined values.

Solution

We can use the various constructors for a struct to accomplish our goals. To initialize all the internal fields in a struct to their standard default values based on their type, we simply use the default initialization of structs, which will be demonstrated later. To

initialize the struct's fields to a set of predefined values, we use an overloaded constructor. Finally, to initialize our struct to a set of nonstandard default values, we need to use optional arguments in the struct's constructor. With optional arguments, structs are able to set their internal fields based on the default values placed on the optional arguments in the constructor's parameter list.

The data structure in Example 1-8 uses an overloaded constructor to initialize all the fields of the structure.

Example 1-8. Struct with an overloaded constructor

```
public struct Data
{
    public Data(int intData, float floatData, string strData,
                char charData, bool boolData)
    {
        IntData = intData;
        FloatData = floatData;
        StrData = strData;
        CharData = charData;
        BoolData = boolData;
    }

    public int IntData { get; }
    public float FloatData { get; }
    public string StrData { get; }
    public char CharData { get; }
    public bool BoolData { get; }

    public override string ToString()=> IntData + " :: " + FloatData + " :: " +
                        StrData + " :: " + CharData + " :: " + BoolData;
}
```

This is the typical way to initialize the values of the struct's fields. Note also that an implicit default constructor exists that allows this struct to initialize its fields to their default values. However, you may want to have each field initialized with nondefault values. The data structure in Example 1-9 uses an overloaded constructor with optional arguments to initialize all the fields of the structure with nondefault values.

Example 1-9. Struct with optional arguments in the constructor

```
public struct Data
{
    public Data(int intData, float floatData = 1.1f, string strData = "a",
                char charData = 'a', bool boolData = true) : this()
    {
        IntData = intData;
        FloatData = floatData;
        StrData = strData;
```

```
        CharData = charData;
        BoolData = boolData;
    }

    public int IntData { get; }
    public float FloatData { get; }
    public string StrData { get; }
    public char CharData { get; }
    public bool BoolData { get; }

    public override string ToString()=> IntData + " :: " + FloatData + " :: " +
                        StrData + " :: " + CharData + " :: " + BoolData;
}
```

Of course, a new initialization method could be introduced that makes this even easier. But you need to explicitly call it, as shown in Example 1-10.

Example 1-10. Struct with an explicit initialization method

```
public struct Data
{
    public void Init()
    {
        IntData = 2;
        FloatData = 1.1f;
        StrData = "AA";
        CharData = 'A';
        BoolData = true;
    }

    public int IntData { get; private set; }
    public float FloatData { get; private set; }
    public string StrData { get; private set; }
    public char CharData { get; private set; }
    public bool BoolData { get; private set; }

    public override string ToString()=> IntData + " :: " + FloatData + " :: " +
                        StrData + " :: " + CharData + " :: " + BoolData;
}
```

Note that when using an explicit initialization method such as `Init`, you'll need to add a private property setter for each property in order for each field to be initialized.

Discussion

We can now create instances of the struct in Example 1-8 using different techniques. Each technique uses a different method of initializing this struct object. The first technique uses the `default` keyword to create this struct:

```
    Data dat = default(Data);
```

The `default` keyword simply creates an instance of this struct with all of its fields initialized to their default values. Essentially this causes all numeric types to default to `0`, `bool` defaults to `false`, `char` defaults to `'\0'`, and `string` and other reference types default to `null`.

Now, this is great if you don't mind reference types and `char` to be set to `null` values, but say that you need to set these types to something other than `null` when the struct is created. The second technique for creating an instance of this struct does just this; it uses a default parameterless constructor:

```
Data dat = new Data();
```

This code causes the default parameterless constructor to be invoked. The caveat with using a default parameterless constructor on a struct is that the `new` keyword must be used to create an instance of this struct. If the `new` keyword is not used, then this default constructor will not be invoked. Therefore, the following code will not call the default parameterless constructor:

```
Data[] dat = new Data[4];
```

Rather, the system-defined default values for each of the struct's fields will be used.

There are two ways to get around this. You could use the overly lengthy way of creating an array of `Data` structs:

```
Data[] dat = new Data[4];

dat[0] = new Data();
dat[1] = new Data();
dat[2] = new Data();
dat[3] = new Data();
```

or

```
ArrayList dat = new ArrayList();
dat.Add(new Data());
dat.Add(new Data());
dat.Add(new Data());
dat.Add(new Data());
```

Or you could use the more terse option, which uses LINQ:

```
Data[] dataList = new Data[4];
dataList = (from d in dataList
            select new Data()).ToArray();
```

The LINQ expression iterates over the `Data` array, explicitly invoking the default parameterless constructor for each `Data` type struct element.

If neither of the first two options will work for your particular case, you could always create an overloaded constructor that takes arguments for each field that you want to

initialize. This third technique requires that the overloaded constructor is used to create a new instance of this struct:

```
public Data(int intData, float floatData, string strData,
            char charData, bool boolData)
{
    IntData = intData;
    FloatData = floatData;
    StrData = strData;
    CharData = charData;
    BoolData = boolData;
}
```

This constructor explicitly initialized each field to a user-supplied value:

```
Data dat = new Data(2, 2.2f, "blank", 'a', false);
```

With C# 6.0 you not only have the option of initializing a struct's fields with the system default values or using an overloaded constructor to initialize its fields to user-defined values, but now you have the additional option of using an overloaded constructor with optional arguments to initialize the struct's fields to nonsystem-default values. This is shown in Example 1-9. The constructor with optional arguments looks like this:

```
public Data(int intData, float floatData = 1.1f, string strData = "a",
            char charData = 'a', bool boolData = true) : this()
{
    ...
}
```

The one issue with using this type of constructor is that you must supply at least one of the parameter values to this constructor. If the `intData` argument also had an associated optional argument:

```
public Data(int intData = 2, float floatData = 1.1f, string strData = "a",
            char charData = 'a', bool boolData = true) : this()
{
    ...
}
```

then this code:

```
Data dat = new Data();
```

would call the default parameterless constructor for the struct, not the overloaded constructor. This is why at least one of the parameters must be passed into this constructor:

```
Data dat = new Data(3);
```

Now we call the overloaded constructor, setting the first parameter, `intData`, to 3 and the rest of the parameters to their optional values.

As a final option, you can add an explicit initialization method to the struct to initialize the fields to nondefault values. This technique is shown in Example 1-10.

You add the `Init` method to the struct, and must call it after the struct is initialized either by using the `new` or `default` keyword. The `Init` method then initializes each field to a nondefault value. The only other code modification that you need to make to the struct's properties is adding a private setter method. This allows the `Init` method to set the internal fields without having to expose them to the outside world.

See Also

The "Struct" topics in the MSDN documentation.

1.19 Checking for null in a More Concise Way

Problem

You are constantly writing unwieldy `if-then` statements to determine whether an object is `null`. You need a more concise and simpler way to write this type of code.

Solution

Use the new `null`-conditional operator introduced in C# 6.0. In the past you would typically have to check to make sure an object is not `null` before using it in the following manner:

```
if (val != null)
{
    val.Trim().ToUpper();
    ...
}
```

Now you can simply use the `null`-conditional operator:

```
val?.Trim().ToUpper();
```

This simplified syntax determines if `val` is `null`; if so, the `Trim` and `ToUpper` methods will not be invoked and you will not throw that annoying `NullReferenceException`. If `val` is not `null`, the `Trim` and `ToUpper` methods will be invoked.

This operator can also be employed to test each object for `null` when the dot operator is used to chain a series of object member accesses:

```
Person?.Address?.State?.Trim();
```

In this case, if any of the first three objects (`Person`, `Address`, or `State`) is `null`, the dot operator is not invoked for that `null` object and execution of this expression ceases.

The null-conditional operator works not only on regular objects, but also on arrays and indexes as well as the indexed element that is returned. For example, if val is of type string[], this code will check to see if the val variable is null:

```
val?[0].ToUpper();
```

whereas this code checks to see if the actual string element stored in the zeroth indexed position in the val array is null:

```
val[0]?.ToUpper();
```

This code is also valid; it determines if both val and the zeroth indexed element are not null:

```
val?[0]?.ToUpper();
```

Another area where the null-conditional operator shines is with invoking delegates and events. For instance, if you have a simple delegate:

```
public delegate bool Approval();
```

and instantiate it using a lambda expression, which for simplicity's sake just returns true all the time:

```
Approval approvalDelegate = () => { return true; };
```

then later in the code when you want to invoke this delegate you don't have to write any bulky conditional code to determine whether it is null; you simply use the null-conditional operator:

```
approvalDelegate?.Invoke()
```

Discussion

Essentially the null-conditional operator works similarly to the ternary operator (?:). The code:

```
val?.Trim();
```

is shorthand for:

```
(val != null ) ? (string)val.Trim() : null
```

assuming val is of type string.

Let's take a look at what happens when a value type is returned such as in the following code:

```
val?.Length;
```

The expression is modified to return a nullable value type such as int?:

```
(val != null ) ? (int?)val.Length : null
```

This means you can't simply use the `null`-conditional operator and then assign the returned value to just any type—it has to be a nullable type. Therefore, this code will not compile:

```
int len = val?.Length;
```

but this code will:

```
int? len = val?.Length;
```

Notice that we have to make the return type a nullable type only when it is a value type.

Additionally, you cannot attempt to use the `null`-conditional operator where a non-nullable type is expected. For example, the array size expects an `int` value, so you cannot compile this code:

```
byte[] data = new byte[val?.Length];
```

However, you could use the `GetValueOrDefault` method to convert the nullable type's value into a non-nullable-friendly value:

```
byte[] data = new byte[(val?.Length).GetValueOrDefault()];
```

This way if `val` is really `null`, the `byte` array will be initialized to the default value for integer types, which is `0`. Just be aware that this method will return the default value for that value type, which is `0` for numeric types and `false` for `bool` types. Your code must take this into account so that your application's behavior is consistent. In this example, the `byte` array is of size 0 if the `val` object is of length 0 or is `null`, so your application logic must account for that.

You also need to take care when using this operator in conditional statements:

```
if (val?.Length > 0)
    Console.WriteLine("val.length > 0");
else
    Console.WriteLine("val.length = 0 or null");
```

In this conditional statement, if the `val` variable is non-`null` and its length is greater than 0, the true block of the `if` statement is executed and the text `"val.length > 0"` is displayed. If `val` is `null`, the false block is displayed and the text `"val.length = 0 or null"` is displayed. However, you don't know which `val` really is—`null` or `0`?

If you need to check for `val` having a length of 0, you could add an extra check to the `if-else` statement to take into account all conditions:

```
if (val?.Length > 0)
    Console.WriteLine("val.Length > 0");
else if (val?.Length == 0)
    Console.WriteLine("val.Length = 0");
else
    Console.WriteLine("val.Length = null");
```

The `switch` statement operates in a similar manner:

```
switch (val?.Length)
{
    case 0:
        Console.WriteLine("val.Length = 0");
        break;
    case 1:
        Console.WriteLine("val.Length = 1");
        break;
    default:
        Console.WriteLine("val.Length > 1 or val.Length = null");
        break;
}
```

If `val` is `null`, execution will fall through to the `default` block. You won't know if the length of `val` is greater than 1 or `null` unless you perform more checks.

Take care when using this operator in conditional statements. This can lead to logic errors in your code if you are not careful.

See Also

The "Null-Conditional Operator" topics in the MSDN documentation.