Essential C# 12.0 (Paperback) Errata

Page xxvi

- ***Chapter 9**, Introducing Structs and Records:* C# 9.0 introduced the concepts of records for structs and expanded it to reference types in C# 10. Although not as prevalent as defining reference types, it is sometimes necessary to define value types that behave in a fashion similar to the primitive types built into C#. In defining how to create custom structures, this chapter also describes the idiosyncrasies they may introduce.

- *Chapter 10, Well-Formed Types:* This chapter discusses more advanced type definition. It explains how to implement operators, such as + and casts, and

Page 2

Hello, World

The best way to learn a new programming language is to write code. The first example is the classic `HelloWorld` program. In this program, you will display some text to the screen.

Listing 1.1 shows the complete HelloWorld program; in the following sections, you will compile and run the code.

Begin 9.0

Listing 1.1: HelloWorld in C#[2]

```
Console.WriteLine("Hello. My name is Inigo Montoya.");
```

(A listing this simple requires a feature—**top-level statements**—enabled in C# 9.0 to help with learning C#. An alternative listing—arguably a more typical listing—is shown later in Listing 1.6. See Chapter 4 for more information about top-level statements.)

Chapter 5

End 9.0

Pages 51-52:
the "Begin 11.0" and "End 11.0" in the sidebar should be changed to "Begin 10.0" and "End 10.0." Additionally, this "Advanced Topic" does not cover the "using static directive" as indicated by the title.

Page 62: Based on C# 12.0 and .NET 8.0, in the "Advanced Topic: Round-Trip Formatting," both the example and the output are incorrect. The entire "Advanced Topic" needs to be rewritten.

Page 70: Listing 2.16: New Lines within String Interpolation does not match the code and the title.

Page 102:

```
Console.WriteLine(uppercase);
```

2.24

This listing differs from Listing 2.18 in two ways. First, rather than using the explicit data type `string` for the declaration, Listing 3.3 uses `var`. The resultant CIL code is identical to using `string` explicitly. However, `var` indicates to the compiler that it should infer the data type from the value assigned within the declaration—in this case, the value returned by `Console.ReadLine()`.

Second, the variables `text` and `uppercase` are initialized by their declarations. Not doing so would result in an error at compile time. As mentioned earlier, the compiler determines the data type of the initializing expression and declares the

4. Introduced in C# 3.0.

Tuples 103

Page 103:

Language Contrast: C++/Visual Basic/JavaScript—`void*`, `Variant`, and `var`

An implicitly typed variable is not the equivalent of `void*` in C++, `Variant` in Visual Basic, or `var` in JavaScript. In each of these cases, the variable declaration is restrictive because the variable may be assigned a value of any type, just as can be done in C# with a variable declaration of type `object`. In contrast, `var` is definitively typed by the compiler; once established at declaration, the type may not change, and type checks and member calls are verified at compile time.

less restrictive

Tuples

On occasion, you will find it useful to combine data elements together. Consider, for example, information about a country such as the poorest country in the world in 2022: Burundi, whose capital is Bujumbura, with a GDP per capita of $263.67. Given the constructs we have established so far, we could store each data element in individual variables, but the result would be no association of the data elements together. That is, $263.67 would have no association with Burundi, except perhaps by a common suffix or prefix in the variable names. Another option would be to

---

Page 104-106, Table 3.1
change all 2017 to 2022

Page 110:
For completeness, the System.Value exists but will rarely be used, 改为：For completeness, the System.ValueType exists but will rarely be used,

Page 114, Table 3.2, Forward Accessing an Array
change “// Retrieve second item from the end (Python)” to “// Retrieve third item from the end (Python)”

Page 119

The initialization follows the pattern in which there is an array of three elements of type `int[]`, and each element has the same size; in this example, the size is 3. Note that the sizes of each `int[]` element must all be identical. The declaration shown in Listing 3.16, therefore, is not valid.

```
int[,] cells = {
    {1, 0, 2, 0},
    {1, 2, 0},
    {1, 2},
    {1}
};
```

119

Listing 3.16: A Multidimensional Array with Inconsistent Size, Causing an Error

```
// ERROR: Each dimension must be consistently sized
// ...
```

Representing tic-tac-toe does not require an integer in each position. One alternative is to construct a separate virtual board for each player, with each board containing a `bool` that indicates which positions the players selected. Listing 3.17 corresponds to a three-dimensional board.

Page 126, Listing 3.26, the last 2 statements:

```
// 6. C#, COBOL, Java, C++, TypeScript, Swift, Python, Lisp, JavaScript
Console.WriteLine($@"  ..: {string.Join(", ", languages[..]) }");

// 7. C#, COBOL, Java, C++, TypeScript, Swift, Python, Lisp, JavaScript
Console.WriteLine($@" 0..^0: {string.Join(", ", languages[0..^0])}");
```

Page 130, Listing 3.28:

output not match source code

Page 139, Listing 4.2:

正文说的是U.S debt，代码清单用的是World Debt

Page 147:

decimal decimalNumber = 4.2M;

double doubleNumber1 = 0.1F * 42F;

double doubleNumber2 = 0.1D * 42D;

float floatNumber = 0.1F * 42F;

// 1. Display: 4.2 != 4.2000002861023 - True

Console.WriteLine($"{decimalNumber} != {(decimal)doubleNumber1} - {decimalNumber != (decimal)doubleNumber1}");

// 2. Display: 4.2 != 4.200000286102295 - True
Console.WriteLine($"{(double)decimalNumber} != {doubleNumber1} - {(double)decimalNumber != doubleNumber1}");

// 3. Display: (float)4.2M != 4.2000003F - True
Console.WriteLine($"(float){(float)decimalNumber}M != {floatNumber}F - {(float)decimalNumber != floatNumber}");

// 4. Display: 4.200000286102295 != 4.2 - True
Console.WriteLine($"{doubleNumber1} != {doubleNumber2} - {doubleNumber1 != doubleNumber2}");

// 5. Display: 4.2000003F != 4.2D - True
Console.WriteLine($"{floatNumber}F != {doubleNumber2}D - {floatNumber != doubleNumber2}");

// 6. Display: 4.199999809265137 != 4.2 - True
Console.WriteLine($"{(double)4.2F} != {4.2D} - {(double)4.2F != 4.2D}");

// 7. Display: 4.2F != 4.2D - True
Console.WriteLine($"{4.2F}F != {4.2D}D - { 4.2F != 4.2D}");

Output:

- `4.2 != 4.2000002861023` - True
- `4.2 != 4.200000286102295` - True
- `(float)4.2M != 4.2000003F` - True
- `4.200000286102295 != 4.2` - True
- `4.2000003F != 4.2D` - True
- `4.199999809265137 != 4.2` - True
- `4.2F != 4.2D` - True

Page 159-160:


```
        // Input is less than or equal to 0
        Console.WriteLine("Exiting...");
    else
        // Condition 2.
        if (input < 9) // line 20
            // Input is less than 9
            Console.WriteLine(
                $"Tic-tac-toe has more than {input}" +
                " maximum turns.");
        else
            // Condition 3.
            if (input > 9) // line 26
                // Input is greater than 9
                Console.WriteLine(
                    $"Tic-tac-toe has fewer than {input}" +
                    " maximum turns.");
            // Condition 4.
            else
                // Input equals 9
                Console.WriteLine(  // line 33
                    "Correct, tic-tac-toe " +
                    "has a maximum of 9 turns.");
```

Output 4.13

```
What is the maximum number of turns in tic-tac-toe? (Enter 0 to exit.):
 9
Correct, tic-tac-toe has a maximum of 9 turns.
```

Assume the user enters 9 when prompted at line 14. Here is the execution path:

1. *Condition 1:* Check if input is less than 0. Since it is not, jump to Condition 2.
2. *Condition 2:* Check if input is less than 9. Since it is not, jump to Condition 3.
3. *Condition 3:* Check if input is greater than 9. Since it is not, jump to Condition 4.

Page 187:

Output 4.20

```
and = 4
or = 15
xor = 11
```

Combining a bitmap with a mask using something like `fields &= mask` clears the bits in `fields` that are not set in the `mask`. The opposite, `fields &= ~mask`, clears the bits in `fields` that are set in `mask`.

Bitwise Complement Operator (~)

nint, nuint,

The **bitwise complement operator** takes the complement of each bit in the operand, where the operand can be an `int`, `uint`, `long`, or `ulong`. The expression `~1`, therefore, returns the value with binary notation `1111 1111 1111 1111 1111 1111 1111 1110`, and `~(1<<31)` returns the number with binary notation `0111 1111 1111 1111 1111 1111 1111 1111`.

Control Flow Statements, Continued

Now that we've described Boolean expressions in more detail, we can more clearly describe the control flow statements supported by C#. Many of these statements will be familiar to experienced programmers, so you can skim this section looking for details specific to C#. Note in particular the `foreach` loop, as it may be new to many programmers.

Page 212-213:

Specifying Line Numbers (#line)

The #line directive controls on which line number the C# compiler reports an error or warning. It is used predominantly by utilities and designers that emit C# code. In Listing 4.62, the actual line numbers within the file appear on the left.

Listing 4.62: The #line Preprocessor Directive

```
#line 113 "TicTacToe.cs"
#warning "Same move allowed multiple times."
#line default
```

Including the #line directive causes the compiler to report the warning found on line 125 as though it were on line 113, as shown in the compiler error message in Output 4.30.

```
        // See https://www.regular-expressions.info/ for
        // more information.
        const string pattern = $"""
            (?<{firstName}>\w+)\s+((?<{
            initial}>\w)\.\s+)?(?<{
            lastName}>\w+)\s*
            """;

        Console.WriteLine(
            "Enter your full name (e.g. Inigo T. Montoya): ");
        string name = Console.ReadLine()!;

        // No need to qualify RegEx type with
        // System.Text.RegularExpressions because
        // of the using directive above
        Match match = Regex.Match(name, pattern);

        if (match.Success)
        {
            Console.WriteLine(
                $"{firstName}: {match.Groups[firstName]}");
            Console.WriteLine(
                $"{initial}: {match.Groups[initial]}");
            Console.WriteLine(
                $"{lastName}: {match.Groups[lastName]}");
        }
    }
}
```

Output 5.2 The output does not match the code listing.

```
Hello, my name is Inigo Montoya
```

Global Using Directives

If you search the subdirectory of a C# 10.0 (or higher) project, you will notice there is a file with the extension `.GlobalUsing.g.cs`, generally found in an `obj` folder subdirectory, and it contains multiple global using directives such as those found in Listing 5.9.

LISTING 5.9: Implicit Usings Generated Global Using Declaratives[3]

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
global using global::System.Linq;
global using global::System.Net.Http;
global using global::System.Threading;
global using global::System.Threading.Tasks;
```

3. Although removed from this listing for elucidation purposes, the generated global using statements prefix the namespace with "global::" as in `global using global::System`. Discussion of namespace alias qualifiers appears later in the section.

Page 239:

ADVANCED TOPIC

Nested Using Directives

Not only can you have using directives at the top of a file, but you can also include them at the top of a namespace declaration. For example, if a new namespace, EssentialCSharp, were declared, it would be possible to add a using declarative at the top of the namespace declaration (see Listing 5.12).

Listing 5.12: Specifying the `using` Directive inside a Namespace Declaration

```
// The using directive imports all types from the
// specified namespace into the entire file
using System.Text.RegularExpressions;

public class Program
{
    public static void Main()
    {
        // ...
        // No need to qualify RegEx type with
        // System.Text.RegularExpressions because
        // of the using directive above
        Match match = Regex.Match(name, pattern);
        // ...
    }
}
```

wrong listing

The difference between placing the using directive at the top of a file and placing

Page 209:

Defining Preprocessor Symbols (#define, #undef)

You can define a preprocessor symbol in two ways. The first is with the `#define` directive, as shown in Listing 4.58.

Listing 4.58: A `#define` Example

```
#define CSHARP2PLUS
```

The second method uses the `define` command line. Output 4.27 demonstrates this with Dotnet command-line interface.

Output 4.27

To add multiple definitions, separate them with a semicolon. The advantage of the `define` compiler option is that no source code changes are required, so you may refer https://mikehadlow.blogspot.com/2020/08/c-preprocessor-directive-symbols-from.html for a workaround.

With a **parameter array** declaration, it is possible to access each corresponding argument as a member of the `params` array. In the `Combine()` method implementation, you iterate over the elements of the `paths` array and call `System.IO.Path.Combine()`. This method automatically combines the parts of the path, appropriately using the platform-specific directory separator character. Note that `PathEx.Combine()` is for demonstration only as it provides a rough implementation of what `System.IO.Path.Combine()` does already.

There are a few notable characteristics of the parameter array:

There is no PathEx.Combine() in anywhere, not even PathEx class.

- The parameter array is not necessarily the only parameter on a method.

The C# specification includes additional rules governing implicit conversion between `byte`, `ushort`, `uint`, `ulong`, and the other numeric types. In general, though, it is better to use a cast to make the intended target method more recognizable.

Basic Error Handling with Exceptions

This section examines how to handle error reporting via a mechanism known as **exception handling**. With exception handling, a method can pass information about an error to a calling method without using a return value or explicitly providing any parameters to do so. Listing 5.26 with Output 5.10 contains a slight modification to Listing 1.16—the `HeyYou` program from Chapter 1. Instead of requesting the last name of the user, it prompts for the user's age.

1.18

Listing 5.26: Converting a `string` to an `int`

```
public static void Main()
{
    string? firstName;
    string ageText;
    int age;

    Console.WriteLine("Hey you!");

    Console.Write("Enter your first name: ");
    firstName = Console.ReadLine();

    Console.Write("Enter your age: ");
    // Assume not null for clarity
    ageText = Console.ReadLine()!;
    age = int.Parse(ageText);

    Console.WriteLine(
        $"Hi { firstName }!  You are { age * 12 } months old.");
}
```

Output 5.10

```
Hey you!
Enter your first name: Inigo
```

Page 277, Figure 5.1 needs to be redone to align with Listing 5.27. an example(Chinese version):

开始

```
Console.Write("请输入你的名字: ");
firstName = Console.ReadLine();
Console.Write("请输入你的年龄: ");

ageText = Console.ReadLine ();
```

Try块:

```
age = int.Parse(ageText);
Console.WriteLine (
    $"你好, {firstName}! 你有{age * 12}个
    月大了。");
```

抛出FormatException异常了吗? 是 → FormatException Catch块; 否 → 抛出Exception异常了吗

FormatException Catch块:

```
Console.WriteLine (
    $"你输入的年龄'{ ageText }'不是一个有效
     的整数。");
result = 1;
```

抛出Exception异常了吗 是 → Exception Catch块; 否 → Finally块

Exception Catch块:

```
Console.WriteLine (
    $"出错啦: { exception.Message }");
result = 1;
```

Finally块:

```
Console.WriteLine (
    $"再见, { firstName }。");
```

```
return result;
```

结束

```
if (is null) throw new ArgumentNullException(...)
```

You can accomplish this in a single statement using the null coalescing assignment operator with a `throw ArgumentNullException` expression if the parameter value is null (see Listing 5.32).

it's a null coalescing operator

Listing 5.32: Parameter Validation by Throwing `ArgumentNullException`

```
httpsUrl = httpsUrl ??
    throw new ArgumentNullException(nameof(httpsUrl));
fileName = fileName??
    throw new ArgumentNullException(nameof(fileName));


// ...
```

With .NET 7.0, you can use the `ArgumentNullException.ThrowIfNull()` method (see Listing 5.33).

Listing 5.33: Parameter Validation with `ArgumentException.ThrowIfNull()`

```
ArgumentNullException.ThrowIfNull(httpsUrl);
ArgumentNullException.ThrowIfNull(fileName);

// ...
```

Additional Parameter Validation

There are obviously a myriad of other type constraints that a method may have on its parameters. Perhaps a string argument should not be an empty string, should not be comprised only of whitespace, or must have “HTTPS” as a prefix. Listing 5.34 displays the full `DownloadSSSL()` method, demonstrating this validation.

Listing 5.34: Custom Parameter Validation

```
public class Program
{
    public static int Main(string[] args)
    {
        int result = 0;
        if(args.Length != 2 )
        {
```

```
#endif

    if (!httpsUrl.ToUpper().StartsWith("HTTPS"))
    {
        throw new ArgumentException("URL must start with 'HTTPS'.");
    }

    HttpClient client = new();
    byte[] response =
        client.GetByteArrayAsync(httpsUrl).Result;
    client.Dispose();
    File.WriteAllBytes(fileName!, response);
    Console.WriteLine($"Downloaded '{fileName}' from '{httpsUrl}'.");
}
}
```

When using .NET 7.0 or higher, you can rely on the `ArgumentException.ThrowIfNullOrEmpty()` method to check for both null and an empty string. And, if you invoke the `string.Trim()` method when invoking `ThrowIfNullOrEmpty()`, you can throw an exception if the argument content is only whitespace. (Admittedly, the exception message will not indicate whitespace only is invalid.) The equivalent code for .NET 6.0 or earlier is shown in the else directive.

#if

If the null, empty, and whitespace validation pass, Listing 5.34 has an if statement that checks for the "HTTPS" prefix. If the validation fails, the resulting code throws an `ArgumentException`, with a custom message describing the problem.

Introducing the `nameof` Operator

xxx() is not technically a type, return is.

When the parameter fails validation, it is necessary to throw an exception—generally of type `ArgumentException()` or `ArgumentNullException()`. Both exceptions take an argument of type `string` called `paramName` that identifies the name of the parameter that is invalid. In Listing 5.33, we use the `nameof` operator[11] for this argument. The `nameof` operator takes an identifier, like the `httpsUrl` variable, and returns a string representation of that name—in this case, `"httpsUrl"`.

11. Introduced in C# 6.0.

Page 312


```
    // ...
}
```

Once the write operations are completed, both the `FileStream` and the `StreamWriter` need to be closed so that they are not left open indefinitely while waiting for the garbage collector to run. Listing 6.13 does not include any error handling, so if an exception is thrown, neither `Close()` method will be called.

The load process is similar (see Listing 6.14 with Output 6.4).

no, there is no Close() method anymore. maybe a legacy from like "Essential C# 5.0"? (see below)

Listing 6.14: Data Retrieval from a File

```
using System;
```

```
        employee.Salary = reader.ReadLine();

        // Dispose the StreamReader and its Stream
        reader.Dispose();  // Automatically closes the stream

        return employee;
    }
}

public class Program
{
    public static void Main()
    {
        Employee employee1;

        Employee employee2 = new();
        employee2.SetName("Inigo", "Montoya");
        employee2.Save();

        // Modify employee2 after saving
        IncreaseSalary(employee2);

        // Load employee1 from the saved version of employee2
        employee1 = DataStorage.Load("Inigo", "Montoya");

        Console.WriteLine(
            $"{ employee1.GetName() }: { employee1.Salary }");
    }
    // ...
}
```

Output 6.4

```
Name changed to 'Inigo Montoya'
Inigo Montoya:
```

The reverse of the save process appears in Listing 6.14, which uses a `StreamReader` rather than a `StreamWriter`. Again, `Close()` needs to be called on both `FileStream` and `StreamReader` once the data has been read.

Output 6.4 does not show any salary after `Inigo Montoya:` because `Salary` was not set to `Enough to survive on` by a call to `IncreaseSalary()` until after the call to `Save()`.

.ThrowIfNullOrEmpty() method to check for both null and an empty string. And, if you invoke the string.Trim() method when invoking ThrowIfNullOrEmpty(), you can throw an exception if the argument content is only whitespace. (Admittedly, the exception message will not indicate whitespace only is invalid.) The equivalent code for .NET 6.0 or earlier is shown in the else directive.

If the null, empty, and whitespace validation pass, Listing 5.34 has an if statement that checks for the "HTTPS" prefix. If the validation fails, the resulting code throws an ArgumentException, with a custom message describing the problem.

Introducing the nameof Operator

When the parameter fails validation, it is necessary to throw an exception—generally of type ArgumentException() or ArgumentNullException(). Both exceptions take an argument of type string called paramName that identifies the name of the parameter that is invalid. In Listing 5.33, we use the nameof operator[11] for this argument. The nameof operator takes an identifier, like the httpsUrl variable, and returns a string representation of that name—in this case, "httpsUrl".

5.32 ?

11. Introduced in C# 6.0.

Notice in Listing 6.24 that the getters and setters that are part of the property include the `specialname` metadata. This modifier is what IDEs, such as Visual Studio, use as a flag to hide the members from IntelliSense.

An automatically implemented property is almost identical to one for which you define the backing field explicitly. In place of the manually defined backing field, the C# compiler generates a field with the name `<PropertyName>k_BackingField` in CIL. This generated field includes an attribute (see Chapter 18) called `System.Runtime.CompilerServices.CompilerGeneratedAttribute`. Both the getters and the setters are decorated with the same attribute because they, too, are generated—with the same implementation as in Listings 5.23 and 5.24.

? ?

Constructors

Begi

Now that you have added fields to a class and can store data, you need to consider the validity of that data. As you saw in Listing 6.6, it is possible to instantiate an object using the `new` operator. The result, however, is the ability to create an employee with invalid data. Immediately following the assignment of `employee`,

```
        LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string? Salary { get; set; } = "Not Enough";

    // ...
}
```

Developers should take care when using both assignment at declaration time and assignment within constructors. Assignments within the constructor will occur after any assignments are made when a property or field is declared (such as `string Salary { get; set; } = "Not enough"` in Listing 6.26). Therefore, assignment within a constructor will override any value assigned at declaration time. This subtlety can lead to a misinterpretation of the code by a casual reader who assumes the value after instantiation is the one assigned in the property or field declaration. Therefore, it is worth considering a coding style that does not mix both declaration assignment and constructor assignment for the same field or property

Listing 6.28

Page 52

Notice that since we don't include an explicit constructor for `Book`, we rely on the automatically generated default constructor to instantiate the book. This is ideal since the required members define how to construct the object in place of any constructor. Providing a constructor with parameters for the required members will result in having to specify the values both as constructor arguments and, redundantly, in the object initializer. A constructor with a `Title` parameter, for example, will result in an instantiation like this:

```
Book book = new("A People's History of the United States")
{
Title= "A People's History of the United States",
Isbn="978-0062397348"
};
```

To avoid this redundancy, you can decorate a constructor with the `SetRequireParameters` attribute, instructing the compiler to disable all the object initializer requirements when invoking the associated constructor. (Effectively, the `SetRequireParameters` attribute instructs the compiler that the developer will take care of setting all the required members so the compiler can ignore checking for initialization assignment. Unfortunately, however, there is little to no verification that such initialization did occur—the compiler doesn't check. See Listing 6.40 for an example of how to use the `SetsRequiredMembers` attribute.

Listing 6.40: Disabling `required` Object Initialization

```
[SetsRequiredMembers]
public Book(int id)
{
    Id = id;
```

Listing 6.38(source code) & Page 352

EssentialCSharp / src / Chapter06 / Listing06.38.RequiredProperties.cs

BenjaminMichaelis feat: Get NetCore Preprocessor Directive, Reference Assembly, and Pac...

Code | Blame | 66 lines (60 loc) · 1.3 KB

```
1   namespace AddisonWesley.Michaelis.EssentialCSharp.Chapter06.Listing06_38;
2
3   #if NET7_0_OR_GREATER
4   #region INCLUDE
5   public class Book
6   {
7       public Book()
8       {
9           // Look up employee name...
10          // ...
11      }
12
13      string? _Title;
```

352 Chapter 6: Classes

Notice that since we don't include an explicit constructor for Book, we rely on the automatically generated default constructor to instantiate the book. This is ideal since

Page 365-366

LISTING 6.49: Declaring a Static Constructor

```
public class Employee
{
    static Employee()
    {
        Random randomGenerator = new();
        NextId = randomGenerator.Next(101, 999);
    }

    // ...
```



```
    public static int NextId = 42;
    // ...
}
```

Listing 6.49 assigns the initial value of `NextId` to be a random integer between 100 and 1,000. Because the initial value involves a method call, the `NextId` initialization code appears within a static constructor and not as part of the declaration.

Page 367

might provide a performance improvement if initialization of static members is expensive and is not needed before accessing a static field. For this reason, you should consider either initializing static fields inline rather than using a static constructor or initializing them at declaration time.

Guidelines

CONSIDER initializing static fields inline rather than explicitly using static constructors or declaration assigned values.

confused..

Static Properties

You also can declare properties as static. For example, Listing 6.50 wraps the data for the next ID into a property.

Page 388

When declaring a derived class, follow the class identifier with a colon and then the base class, as Listing 7.1 demonstrates.

Listing 7.1: Deriving One Class from Another

```
public class PdaItem
{
    [DisallowNull]
    public string? Name { get; set; }
    public DateTime LastUpdated { get; set; }
}
// Define the Contact class as inheriting the PdaItem class
public class Contact : PdaItem
{
    public string? Address { get; set; }
    public string? Phone { get; set; }
```

```
    // ...
}
```

inherited

Listing 7.2 shows how to access the properties defined in `Contact`.

Listing 7.2: Using Inherited Methods property

```
public class Program
{
    public static void Main()
    {
        Contact contact = new();
        contact.Name = "Inigo Montoya";

        // ...
    }
}
```

Even though `Contact` does not directly have a property called `Name`, all

Page 402-403

The third row is missing in Table 7.1.:

<table>
<tr><th>Activity</th><th>Code</th></tr>
<tr><td>Later, Programmer A adds the Name property, but instead of implementing the getter as FirstName + " " + LastName, she implements it as LastName + ", " + FirstName. Furthermore, she doesn't define the property as virtual, and she uses the property in a DisplayName() method.</td><td><pre>
// ...
public class Person
{
  public string Name
  {
      get
      {
          return LastName + ", " + FirstName;
      }

      set
      {
          string[] names = value.Split(", ");
          // Error handling not shown
          LastName = names[0];
          FirstName = names[1];
      }
  }
  public static void Display(Person person)
  {
      // Display <LastName>, <FirstName>
      Console.WriteLine(person.Name);
  }
}
</pre></td></tr>
</table>

Page 417


```
Address: 221B Baker Street, London, England
_________
Subject: Soccer tournament
Start: 7/18/2008 12:00:00 AM
End: 7/19/2008 12:00:00 AM
Location: Estádio da Machava
_________
FirstName: Hercule
LastName: Poirot
Address: Apt 56B, Whitehaven Mansions, Sandhurst Sq, London
```

In this way, you can call the method on the base class, but the implementation is specific to the derived class. Output 7.5 shows that the `List()` method from Listing 7.18 is able to successfully display both `Contacts` and `Addresses`, and display them in a way tailored to each. The invocation of the abstract `GetSummary()` method actually invokes the overriding method specific to the instance.

Appointments

All Classes Derive from `System.Object`

Given any class, whether a custom class or one built into the system, the methods shown in Table 7.2 will be defined.

Page 419, Listing 7.20

```
32
33    public class Program
34    {
35        private static object? GetObjectById(string id)
36        {
37            #region EXCLUDE
38            if(id is null) throw new ArgumentNullException("id");
39
40            if (id.StartsWith(nameof(Employee)))
41            {
42                return new Employee("Inigo", "Montoya", id.Remove(0,nameof(Employee).Length));
43            }
44            else if (id.StartsWith(nameof(Person)))
45            {
46                return new Person("Inigo", "Montoya");
47            }
48            return null;
49            #endregion EXCLUDE
50        }
```

?

return id;

Page 427


```
            _ => null,// Set the button to indicate an invalid value
    }) is not null;
}
```

The order of precedence is `not`, `and`, `or`; thus the first two examples don't need parentheses. Parentheses, however, are allowed to change the default order of precedence as demonstrated (arbitrarily) by match expressions in the `switch` expression of Listing 7.24. 7.25

Be aware that when using the `or` and `not` operators, you cannot also declare a variable. For instance:

```
if (input is "data" or string text) { }
```

will result in an error: "CS8780: A variable may not be declared within a 'not' or 'or' pattern." Doing so would result in ambiguity about whether initialization of `text` occurred.

```
    }
}
```

In both Listing 7.27 and Listing 7.28, we pattern match against a tuple that is populated with the length and the elements of `args`. In the first match expression, we check for one argument and the action `"cat"`. In the second match expression, we evaluate whether the first item in the array is equal to `"encrypt"`. In addition, Listing 7.27 assigns the third element in the tuple to the variable `fileName` if the initial match expression evaluates to true. The `switch` statement in Listing 7.28 doesn't make the variable assignment since the input operand of the `switch` statement is the same for all match expressions. ??

Each element match can be a constant or a variable. Since the tuple is instantiated before the `is` operator executes, we can't use the `"encrypt"` scenario first because `args[FileName]` would not be a valid index if the `"cat"` action was requested.

Positional Patterns (C# 8.0)

Building on the deconstructor construct introduced in C# 7.0 (see Chapter 6), C# 8.0 enables positional pattern matching with a syntax that closely matches tuple pattern matching (see Listing 7.29).

Listing 7.29: Positional Pattern Matching with the `is` Operator

```
using System.Drawing;
```

```
    };
}
```

The `System.Drawing.Point` type doesn't have a deconstructor. However, we are able to add one as an extension method, which satisfies the criteria for converting the `Point` to a tuple of `X` and `Y`. The deconstructor is then used to match the order of each comma-separated match expression in the pattern. In the example of `IsVisibleOnVGAScreen()`, `X` is matched with `>=0 and <=1920` and `Y` with `>=0 and <=1080`. Similar range expressions are used with the `switch` expression in `GetQuadrant()`.

Begin 9.0

Property Patterns (C# 8.0 & 10.0)

With property patterns, the match expression is based on property names and values of the data type identified in the `switch` expression, as shown in Listing 7.30.

Listing 7.30: Property Pattern Matching with the `is` Operator ??

```
public record class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Role { get; set; }

    public Employee(int id, string name, string role) =>
        (Id, Name, Role) = (id, name, role);
}


public class ExpenseItem
{
    public int Id { get; set; }
    public string ItemName { get; set; }
    public decimal CostAmount { get; set; }
    public DateTime ExpenseDate { get; set; }
    public Employee Employee { get; set; }

    public ExpenseItem(
        int id, string name, decimal amount, DateTime date,
            Employee employee) =>
                (Id, Employee, ItemName, CostAmount, ExpenseDate) =
                (id, employee, name, amount, date);
```

expression uses an expanded property pattern match syntax introduced in C# 10 and identified by the dot notation that accesses subproperties (`Length` and `Role`).

The second match expression for Listing 7.30, `{ ItemName: { Length: > 0 }, Employee: {Role: "Manager" }, ExpenseDate: DateTime date }`, is also a property match expression (rather than an expanded property match expression) but with a slightly different syntax. The property match expression was introduced in C# 7.0 and is still supported, albeit with a warning, because the expanded syntax is simpler and preferred in all cases.

This second match expression includes a property match for `ExpenseDate` for which it declares a variable, `date`. While `date` is still used as part of the `switch` filter, it appears in a **when clause** rather than the match expression.

When Clause

For all forms of constant and relational match expressions, it is necessary to use a constant. This, however, can be restrictive given that frequently the comparative operand is not known at compiler time. To avoid the restriction, C# 7.0 includes a `when` clause into which you can add any conditional expression (an expression that returns a Boolean).

The code in Listing 7.30 declares a DateTime date variable to check that the `ExpenseItem.ExpenseDate` is no older than 30 days. However, because `age` is dependent on the current date and time, the value isn't constant, and we can't use a match expression. Instead, the code uses a `when` clause following the match expression where the `date` value is compared to `DateTime.Now.AddDays(-30)`. The `when` clause provides a catchall location for (optionally)

Pattern Matching with Unrelated Types

An interesting capability of pattern matching is that it becomes possible to extract data from unrelated types and morph the data into a common format. Listing 7.31 provides an example.

LISTING 7.31: Pattern Matching within a `switch` Expression

```
public static string? CompositeFormatDate(
        object input, string compositeFormatString) =>
    input switch
    {
        DateTime
            { Year: int year, Month: int month, Day: int day }
            => (year, month, day),
        DateTimeOffset
            { Year: int year, Month: int month, Day: int day }
                => (year, month, day),
        DateOnly
            { Year: int year, Month: int month, Day: int day }
                => (year, month, day),
        string dateText => DateTime.TryParse(
            dateText, out DateTime dateTime) ?
                (dateTime.Year, dateTime.Month, dateTime.Day) :
                // default ((int Year, int Month, int Day)?)
                // preferable but not covered until Chapter 12.
                ((int Year, int Month, int Day)?) null,
        _ => null
    } is (int, int, int) date ? string.Format(
        compositeFormatString, date.Year, date.Month, date.Day) : null;
```

The first match expression of the `switch` expression in Listing 7.32 uses type pattern matching (C# 7.0) to check whether the input is of type `DateTime`. If the result is `true`, it passes the result to the property pattern matching to declare and assign the values `year`, `month`, and `day`; it then uses those variables in a tuple expression that returns the tuple `(year, month, day)`. The `DateTimeOffset` and `DateOnly` match expressions work the same way.

7.31

Given a match on the `string` match expression, if `TryParse()` is unsuccessful, we return a `default((int Year, int Month, int Day)?)`,[2]

2. See Chapter 12 for more information.

```
        null => 1,
        Contact contact when ReferenceEquals(this, obj) => 0,
        Contact { LastName: string lastName }
            when LastName.CompareTo(lastName) != 0 =>
                LastName.CompareTo(lastName),
        Contact { FirstName: string firstName }
            when FirstName.CompareTo(firstName) != 0 =>
                FirstName.CompareTo(firstName),
        Contact _ => 0,
        _ => throw new ArgumentException(
            $"The parameter is not a value of type { nameof(Contact) }",
            nameof(obj))
    };
    #endregion

    #region IListable Members
    string?[] IListable.CellValues
    {
        get
        {
            return new string?[]
            {
                FirstName,
                LastName,
                Phone,
                Address
            };
        }
    }
    #endregion

    // ...
}
```


Once a class declares that it implements an interface, all (abstract[2]) members of the interface must be implemented. An abstract class is permitted to supply an abstract implementation of an interface member. A non-abstract implementation may throw a `NotImplementedException` type exception in the method body, but an implementation of the member must always be supplied.

One important characteristic of interfaces is that they can never be instantiated; you cannot use `new` to create an interface, so interfaces do not have instance constructors or finalizers. Interface instances are available only by instantiating a type that implements the interface. Furthermore, interfaces cannot include static members.[3] One key interface purpose is polymorphism, and polymorphism without an instance of the implementing type has little value.

Each (non-implemented[4]) interface member is abstract, forcing the derived class to implement it. Therefore, it is not possible to use the abstract modifier on interface members explicitly.[5]

When implementing an interface member in a type, there are two ways to do so: **explicitly** or **implicitly**. So far, we've seen only implicit implementations, where the type member that implements the interface member is a public member of the implementing type.

Explicit Member Implementation

Explicitly implemented methods are available only by calling them through the interface itself; this is typically achieved by casting an object to the interface. For example, to call `IListable.CellValues` in Listing 8.4, you must first cast the contact to `IListable` because of `CellValues`' explicit implementation.

Listing 8.4: Calling Explicit Interface Member Implementations

```
string?[] values;
Contact contact = new("Inigo Montoya");
```

Page 454


```
// ...

// ERROR:  Unable to call .CellValues directly
//         on a contact
values = contact.CellValues;

// First cast to IListable
values = ((IListable)contact).CellValues;

// ...
```

should be commented out because the #if COMPILEERROR directive is not shown in the book

The cast and the call to `CellValues` occur within the same statement in this case. Alternatively, you could assign `contact` to an `IListable` variable before calling `CellValues`.

To declare an explicit interface member implementation, prefix the member name with the interface name (see Listing 8.5).

associated with the interface, there is no need to modify them with `virtual`, `override`, or `public`. In fact, these modifiers are not allowed. The method is not treated as a public member of the class, so marking it as `public` would be misleading.

Note that even though the `override` keyword is not allowed on an interface, we will still use the term "override" when referring to members that implement the interface-defined signature.

Implicit Member Implementation 8.3

Notice that `CompareTo()` in Listing 8.5 does not include the `IComparable` prefix; it is implemented implicitly. With implicit member implementation, it is necessary only for the member to be public and for the member's signature to match the interface member's signature. Interface member implementation does not require use of the `override` keyword or any indication that this member is tied to the interface. Furthermore, since the member is declared just like any other class member, code that calls implicitly implemented members can do so directly, just as it would any other class member:

Table 8.1: Default Interface Refactoring Features

C# 8.0–Introduced Interface Construct	Sample Code

Static Members
The ability to define static members on the interface including fields, constructors, and methods. (This includes support for defining a static `Main` method —an entry point into your program.) The default accessibility for static members on interfaces is public.
	`public interface ISampleInterface`
	`{`
	`private static string? _Field;`
	`public static string? Field`
	`{`
	`get => _Field;`
	`private set => _Field = value; }`
	`static IsampleInterface() =>`
	`Field = "Nelson Mandela";`
	`public static string? GetField() => Field;`
	`}`

Implemented Instance Properties and Methods
You can define implemented properties and members on interfaces. Since instance fields are not supported, properties cannot work against backing fields. Also, without instance fields support, there is no automatically implemented property support.
Note that to access a default implemented property, it is necessary to cast to the interface containing the member. The class (`Person`) does not have the default interface member available unless it is implemented.
	`public interface IPerson`
	`{`
	`// Standard abstract property definitions`
	`string FirstName { get; set; }`
	`string LastName { get; set; }`
	`string MiddleName { get; set; }` remove this line
	`// Implemented instance properties and methods`
	`public string Name => GetName();`
	`public string GetName() =>`
	`$"{FirstName} {LastName}";`
	`}`
	`public class Person :IPerson`
	`{`
	`// ...`
	`}`
	`public class Program`
	`{`
	`public static void Main()`
	`{`
	`Person inigo = new Person("Inigo", "Montoya");`
	`Console.Write(`
	`((IPerson)inigo).Name);`
	`}`
	`}`

`public` Access Modifier
The default for all instance interface members. Use this keyword to help clarify the accessibility of the code. Note, however, that the compiler-generated CIL code is identical with or without the `public` access modifier.
	`public interface IPerson`
	`{`
	`// All members are public by default`
	`string FirstName { get; set; }`
	`public string LastName { get; set; }`
	`string Initials =>`
	`$"{FirstName[0]}{LastName[0]}";`
	`public string Name => GetName();`
	`public string GetName() =>`
	`$"{FirstName} {LastName}";`
	`}`

private members must include an implementation.	

internal Access Modifier
internal members are only visible from within the same assembly in which they are declared.
	`public interface IPerson`
	`{`
	`string FirstName { get; set; }`
	`string LastName { get; set; }`
	`string Name => GetName();`
	`internal string GetName() =>`
	`$"{FirstName} {LastName}";`
	`}`

private protected Access Modifier
A super set of private and protected; private protected members are visible from within the same assembly and from within other interfaces that derive from the containing interface. Like protected members, classes external to the assembly cannot see protected internal members.
	`public interface IPerson`
	`{`
	`string FirstName { get; set; }`
	`string LastName { get; set; }`
	`string Name => GetName();`
	`protected internal string GetName() =>`
	`$"{FirstName} {LastName}";`
	`}`

Table 8.1: Default Interface Refactoring Features (continued)

C# 8.0–Introduced Interface Construct	Sample Code

private protected Access Modifier
Accessing a private protected member is only
	`class Program`
	`{`
	`static void Main()`
	`{`
	`IPerson? person = null;`

Page 472

Table 8.1: Default Interface Refactoring Features (continued)

<table>
<tr><th>C# 8.0–Introduced Interface Construct</th><th>Sample Code</th></tr>
<tr><td><code>private protected</code> Access Modifier
Accessing a <code>private protected</code> member is only possible from the containing interface or interfaces that derive from the implementing interface. Even classes implanting the interface cannot access a <code>private protected</code> member, as demonstrated by the <code>PersonTitle</code> property in <code>Person</code>.</td><td><pre>
class Program
{
static void Main()
{
IPerson? person = null;
        // Non-deriving classes cannot call
        // private protected member.
        // _ = person?.GetName();
Console.WriteLine(person);
}
}
public interface IPerson
{
string FirstName { get; }
string LastName { get; }
string Name => GetName();
private protected string GetName() =>
$"{FirstName} {LastName}";
}
public interface IEmployee: IPerson
{
int EmpoyeeId => GetName().GetHashCode();
}
public class Person : IPerson
{
public Person(
string firstName, string lastName)
{
FirstName = firstName ??
throw new ArgumentNullException(nameof(firstName));
LastName = lastName ??
throw new ArgumentNullException(nameof(lastName));
}
public string FirstName { get; }
public string LastName { get; }
// private protected interface members
// are not accessible in derived classes.
// public int PersonTitle =>
// GetName().ToUpper();
}
</pre></td></tr>
</table>

'implementing classes'

why int?

Page 473

![628f53b1fdd71e3fa3e280bb4109fc41.png](:/a574014127ff4e12a600d8db37c29f58

Output 8.3

```
Invoking ((IExecuteProcessActivity)activity).Run()...
IWorkflowActivity.Start()...
ExecuteProcessActivity.RedirectStandardInOut()...
ExecuteProcessActivity.IExecuteProcessActivity.ExecutProcess()...
IExecuteProcessActivity.RestoreStandardInOut()...
IWorkflowActivity.Stop()..

Invoking activity.Run()...
Executing non-polymorphic Run() with process 'dotnet'.
```

Notice that `IWorkflowActivity.Run()` is sealed and, therefore, not virtual. This prevents any derived types from changing its implementation. Any invocation of `Run()`, given a `IWorkflowActivity` type, will always execute the `IWorkflowActivity` implementation.

`IWorkflowActivity`'s `Start()` and `Stop()` methods are private, so they are invisible to all other types. Even though `IExecutProcessActivity` seemingly has start/stop-type activities, `IWorkflowActivity` doesn't allow for replacing its implementations.

`IWorkflowActivity` defines a protected `InternalRun()` method that allows `IExecuteProcessActivity` (and `ExecuteProcessActivity`, if desirable) to overload it. However, notice that no member of `ExecuteProcessActivity` can invoke `InternalRun()`. Perhaps that method should never be run out of sequence from `Start()` and `Stop()`, so only an interface (`IWorkflowActivity` or `IExecuteProcessActivity`) in the hierarchy is allowed to invoke the protected member.

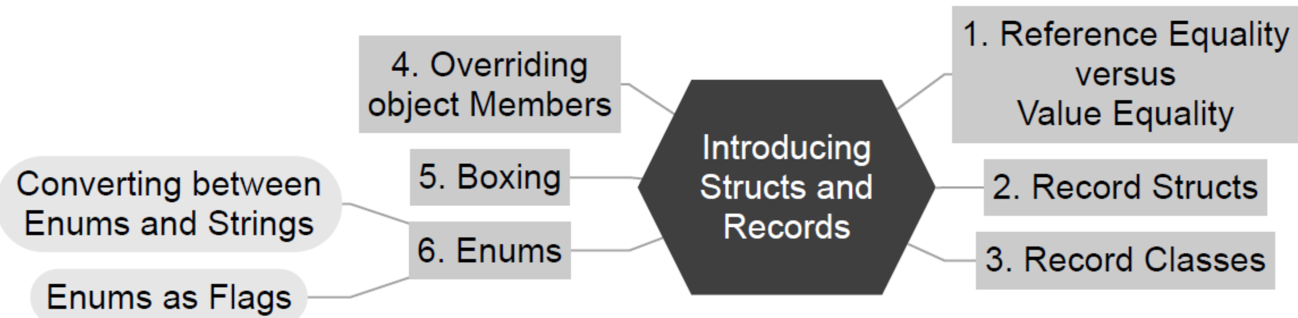
All interface members that are protected can override any default interface member if they do so explicitly. For example, both the `RedirectStandardInOut()` and `RestoreStandardInOut()` implementations on `ExecuteProcessActivity` are prefixed with `IExecuteProcessActivity`. And, like with the protected `InternalRun()` method, the type implementing the interface cannot invoke the protected members; for example, `ExecuteProcessActivity` can't invoke `RedirectStandardInOut()` and `RestoreStandardInOut()`, even though they are implemented on the same type.

Even though only one of them is explicitly declared as virtual, both `RedirectStandardInOut()` and `RestoreStandardInOut()` are virtual

9 Introducing Structs and Records

You have used value types throughout this book; for example, `int` is a value type. This chapter discusses not only using value types but also defining custom value types. One of the key concepts for a value type is the ability to compare instances for the same value, a concept also possible on reference types. However, rather than coding this feature from scratch, C# 9.0 and 10.0 provide shortcuts via the record construct using a record struct and a record class, respectively. This chapter explores structs, records, and a specific value type called an enum.

using a record class and a record struct, respectively.



While there are noticeable complications to correctly implementing custom-built structs, they are relatively rare. They obviously play an important role within C# development, but the number of custom-built structs declared by typical developers is usually tiny compared to the number of custom-built classes. Heavy use of custom-built structs is most common in code intended to interoperate with unmanaged code. Furthermore, they should not be defined unless a single value

BEGINNER TOPIC

Categories of Types

All types discussed so far have fallen into one of two categories: reference types and value types. The differences between the types in each category stem from differences in copying strategies, which in turn result in each type being stored differently in memory. As a review, this Beginner Topic reintroduces the value type/ reference type discussion for those readers who are unfamiliar with these issues.

Value Types

Variables of **value types** directly contain their values, as shown in Figure 9.1. The variable name is associated directly with the storage location in memory where the value is stored. Because of this, when a second variable is assigned the value of an original variable, a copy of the original variable's value is made to the storage location associated with the second variable. Two variables never refer to the same storage location (unless one or both are `out` or `ref` parameters, which are, by definition, aliases for another variable). Changing the value of the original variable will not affect the value in the second variable, because each variable is associated with a different storage location. Consequently, changing the value of one value type variable cannot affect the value of any other value type variable.

could be succinctly rephrased as :
"Modifying the value of one value type variable won't affect another, as each has a separate storage location."

Page 493

> **NOTE**
>
> Calling `ReferenceEquals()` on value types will always return `false`.

While value types can never be reference equal, a reference type can support value equals. `object` includes a static method called `ReferenceEquals()` that checks whether the two arguments are identical—reference equal. In addition, `object` defines a virtual `Equals()` method that, by default, relies on `ReferenceEquals()` for reference types. However, this `Equals()` method can be customized (on both reference types and value types) so that they implement value equality rather than reference.

In fact, you should always override `Equals()` on value types because otherwise you would be left with the default implementation that compares only the first field for value equality. This, however, is insufficient. If a value type had three properties, for example, then checking for value equality by comparing only the first one would likely be inadequate. For example, given an `Angle` object, a comparison of the first property (`Degrees`) is not adequate. Two `Angles` are equal only if `Degrees`,

sure? from MSDN document:

Remarks

The ValueType.Equals(Object) method overrides Object.Equals(Object) and provides the default implementation of value equality for all value types in the .NET Framework.

The default implementation calls Object.Equals(Object) on each field of the current instance and `obj` and returns `true` if all fields are equal.

> Tip
>
> Particularly if your value type contains fields that are reference types, you should override the **Equals(Object)** method. This can improve performance and enable you to more closely represent the meaning of equality for the type.

9.2 9.4

From the one line of code in Listing 9.1, Listing 9.3 shows a significant amount of code is generated. To start, the `Angle` is not a class but a struct, and the `record` contextual keyword dropped. In C#, to define a custom value type it must be a struct. And, while C# allows you to write the entire struct from scratch as demonstrated by Listing 9.3, C# 9.0's `record` keyword jumpstarts much of the boilerplate code such that there is no longer any reason not to use the `record` keyword to start, especially since you can define custom versions of all the generated code, thus overriding the default generated implementations as needed. 9.4

Begin 10.0

Note that in C# 9.0, the record keyword without the struct keyword was all that the compiler allowed. However, in C# 10.0 (with the introduction of records for structs), the explicit declaration of `record class` was added. For clarity, we recommend using `record class` always, rather than the abbreviated `record` -only syntax.

Guidelines

DO **use `record struct` when declaring a struct (C# 10.0).**

DO **use `record class` (C# 10.0) for clarity, rather than the abbreviated `record`-only syntax.**

All value types are implicitly sealed (you can't derive from them). In addition, all non-enum value types (enums are discussed later in the chapter) derive from `System.ValueType`. Consequently, the inheritance chain for structs is always from `object` to `System.ValueType` to the custom struct.

`System.ValueType` brings with it the behavior of value types by overriding all the virtual methods of `object`: `Equals()`, `ToString()`, and `GetHashCode()`. However, this implantation is generally not sufficient, leaving the developer with the responsibility of further specializing each of these methods—of course, that is until C# 10.0 and the introduction of the `record` modifier, where

supporting the concept of read-only access to members, developers declare the behavioral intent of whether a member can modify the object instance. Note that properties that are not automatically implemented can use the `readonly` modifier on either the getter or the setter (although the latter would be strange). To decorate both, the `readonly` modifier would be placed on the property itself, rather than on the getter and setter individually.

While allowable, using the `readonly` modifier on struct members is redundant when the struct is read-only. However, favor read-only (`{get;}`) or `init`-only setter (`{get;init;}`) automatically implemented properties over fields within structs.

End 8.0

Guidelines

DO use the `readonly` modifier on a struct definition, making value types immutable.

DO use read-only or init-only setter automatically implemented properties rather than fields within structs.

Remarkably, the tuple (`System.ValueTuple`) is one example that breaks the immutable guideline. To understand why it is an exception, see https://intellitect.com/WhyTupleBreaksTheImmutableRuleshttps://IntelliTect.com/WhyTupleBreaksTheImmutableRules.

Cloning a record struct is a memory copy, the same as when creating a copy to invoke a method with pass by value. For this reason, you can't change the implementation for cloning a record struct.

For record classes, the process is slightly different. The C# compiler generates a hidden `Clone()` method (shown in Listing 9.5) that in turn invokes the copy constructor with a parameter for the source instance. You can also view that same code in Listing 9.11:

LISTING 9.11: Cloning Record Classes via the `Clone` Method

```
// Actual name in IL is "<Clone>$". However,
// you can't add a Clone method to a record.
public Coordinate Clone() => new(this);

protected Coordinate(Coordinate original)
{
    Longitude = original.Longitude;
    Latitude = original.Latitude;
}
```

The clone method in turn calls the copy constructor.[4] Note that in the case where the object initializer syntax is used, the assigned property cannot be read-only (either a setter or an init-only setter is required). Also, the `Clone()` method uses a special non-C#–compliant name, making it accessible only via the with operator. If you want

4. A constructor that takes single parameter or the containing type. See Chapter 6.

to customize the clone behavior, however, you can provide your own implementation of the copy constructor.

All increments by 0.1

Record Constructors

Note that the record declaration of Listing 9.1 looks virtually identical to the constructor signature in Listing 9.3. Similarly, with the record class associated code in Listing 9.4 and its equivalent C#-generated equivalent in Listing 9.5, a record declaration and its positional parameters provide the structure for the C# compiler to generate a constructor with an equivalent-looking signature.

As with the properties themselves, one minor peculiarity with the record is that the positional parameters are, by convention, PascalCase, and, therefore, so are the parameters for the constructor. (Consequently, when initializing the properties, the generated constructor code uses the `this` qualifier to distinguish the parameters from the properties.)

You can add additional constructors to the record's definition. For example, you could provide a constructor that has strings rather than integers as parameters, as shown in Listing 9.12:

Listing 9.12: Adding Additional Record Constructors

```
public Angle(
    string degrees, string minutes, string seconds)
    : this(int.Parse(degrees),
           int.Parse(minutes),
           int.Parse(seconds))
{ }
```

The implementation of an additional constructor uses the same syntax as any other constructor. The only additional constraint is that it must invoke the `this` constructor initializer—it must call the record-generated constructor or another constructor that calls the record-generated constructor. This ensures that the initialization of the positional parameter–generated properties are all initialized.

Record Struct Initialization

Prior to C# 10.0, no default constructor could be defined. Regardless, if not explicitly instantiating a struct via the `new` operator's call to the constructor, all data contained

constructors or member initialization at declaration time.

Begin 11.0

Prior to C# 11.0, every constructor in a struct must initialize all fields (and read-only, automatically implemented properties[6]) within the struct. Failure to initialize all data within the struct causes a compile-time error in C# 10.0 and earlier:

```
CS0171: Field must be fully assigned before control is returned to
the caller. Consider updating to language version '11.0'
to auto-default the field.
```

Because of the struct's field initialization requirement, the succinctness of read-only field declaration, automatically implemented property support, and the guideline to avoid accessing fields from outside of their wrapping property, you

5. Enabled starting in C# 10.0
6. Initialization via a read-only, automatically implemented property is sufficient starting in C# 6.0, because the backing field is unknown and its initialization would not be possible otherwise.

should favor read-only, automatically implemented properties over fields within structs.

```
}
```

This enables pattern matching like that used in Listing 9.2 (displayed again in Listing 9.15):

Listing 9.15: Pattern Matching

```
if (angle is (int, int, int, string) angleData)
{
    // ...
}
```

Overriding `object` Members

Chapter 6 discussed how all classes and structs ultimately derive from `object`. In addition, it reviewed each method available on `object` and discussed how some of them are virtual. This section discusses the details concerning overriding these virtual methods. Doing so is automatic for records, but the generated code also provides an example of what is required if you choose to customize the generated code or implement an override on a non-record type implementation.

Overriding `ToString()`

By default, calling `ToString()` on any object will return the fully qualified name of the object type. Calling `ToString()` on a `System.IO.FileStream` object will return the string `System.IO.FileStream`, for example. For some classes, however, `ToString()` can be more meaningful. On `string`, for example, `ToString()` returns the string value itself. Similarly, when invoking `ToString()` on an `Angle` (Listing 9.1), the result returns:

```
Angle { Degrees = 90, Minutes = 0, Seconds = 0, Name = }
```

++.1

(This also happens to be the output of Listing 9.2.)

Write methods such as `System.Console.WriteLine()` and `System.Diagnostics.Trace.Write()` call an object's `ToString()` method,[7] so overriding the method often outputs more meaningful information than the default implementation.

Overriding `ToString()` requires nothing more than declaring the `ToString()` method as `override` and returning a `string`. Take Listing 9.16 for example:

Listing 9.16: Overriding the `ToString` Method

```
public override string ToString()
{
    string prefix =
        string.IsNullOrWhiteSpace(Name) ? string.Empty : Name + ": ";
```

> **AVOID** including mutable data when overriding the equality-related members on mutable reference types or if the implementation would be significantly slower with such overriding.
>
> **DO** implement all the equality-related methods when implementing `IEquatable`.

Begin 10.0

ADVANCED TOPIC

Overriding `GetHashCode()`

If you rely on a record construct, `GetHashCode()` is automatically implemented for you as part of the value equality implementation (see Listing 9.3 and Listing 9.5). Without the record implementation, you have to implement `GetHashCode()` on your own, however, if you are providing an equality implementation. Even with a record, if you customize the `Equals()` implementation, you will likely want to override `GetHashCode()` to use a similar set of values as the new `Equals()` implementation. And, if you override only `Equals()` and not `GetHashCode()`, you will have a warning that:

```
CS0659: '<Class Name>' overrides Object.Equals(object o) but
does not override Object.GetHashCode(),
```

In other words, when not leveraging the record construct, overriding equals requires that you also override `GetHashCode()`.

The purpose of the hash code is to *efficiently balance a hash table* by generating a number that corresponds to the value of an object. And, while there are numerous guidelines (see https://bit.ly/39yP8lm for a discussion), the easiest approach is as follows:

1. Rely on the record generated implementation (see Listing 9.2). If you need to override `GetHashCode()`, you are probably overriding `Equals()`and using a record is the best approach by default.

2. Call `System.HashCode`'s `Combine()` method, specifying each of the identifying fields (see Listing 9.22):

LISTING 9.22: Overriding `GetHashCode` with `Combine` Method

```
public override int GetHashCode() =>
    HashCode.Combine(Degrees, Minutes, Seconds);
```

3. Invoke `ValueTuple`'s `GetHashCode()` method using the fields that produce your object's uniqueness as the tuple elements, as demonstrated in Listing 9.23. (If the identifying fields are numbers, be wary of mistakenly using the fields themselves rather than their hash code values.)

LISTING 9.23: Overriding `GetHashCode` with `Combine` Method

```
public override int GetHashCode() =>
    (Degrees, Minutes, Seconds).GetHashCode();
```

4. `ValueTuple` invokes `HashCode.Combine()`; thus, it may be easier to remember that you can adequately create a `ValueTuple` with the same identifying fields and invoke the resulting tuple's `GetHashCode()` member

In summary, use a record if overriding `GetHashCode()` and `Equals()` unless there is a strong reason not to (such as an older version of C# in which records are not available).

Fortunately, once you have determined that you need `GetHashCode()`, you can follow some well-established `GetHashCode()` implementation principles:

- *Required:* Equal objects must have equal hash codes (if `a.Equals(b)`, then `a.GetHashCode() == b.GetHashCode()`).
- *Required:* `GetHashCode()`'s returns should be constant (the same value), even if the object's data changes. In many cases, you should cache the method return to enforce this constraint. However, when caching the value, be sure not to use the hash code when checking equality; if you do, two identical objects —one with a cached hash code of changed identity properties—will not return the correct result.

 9.25

```
317811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465,
14930352, 24157817, 39088169, 63245986, 102334155, 165580141,
```

The code shown in Listing 9.9, when compiled, produces five box instructions and three unbox instructions in the resultant CIL.

```
        Angle angle = new(25, 58, 23);
        // Example 1: Simple box operation
        object objectAngle = angle;  // Box
        Console.Write(((Angle)objectAngle).Degrees);

        // Example 2: Unbox, modify unboxed value,
        //            and discard value
        ((Angle)objectAngle).MoveTo
            (26, 58, 23);
        Console.Write(", " + ((Angle)objectAngle).Degrees);

        // Example 3: Box, modify boxed value,
        //            and discard reference to box
        ((IAngle)angle).MoveTo(26, 58, 23);
        Console.Write(", " + ((Angle)angle).Degrees);

        // Example 4: Modify boxed value directly
        ((IAngle)objectAngle).MoveTo(26, 58, 23);
        Console.WriteLine(", " + ((Angle)objectAngle).Degrees);

        // ...
    }
}
```

9.27

Output 9.2

```
25, 25, 25, 26
```

Listing 9.11 uses the `Angle` struct and `IAngle` interface. Note also that the `IAngle.MoveTo()` interface changes `Angle` to be mutable. This change brings out some of the idiosyncrasies of mutable value types and, in so doing, demonstrates the importance of the guideline that advocates making structs immutable.

In Example 1 of Listing 9.11, after you initialize `angle`, you then box it into a variable called `objectAngle`. Next, Example 2 calls `MoveTo()` to change `_Degrees` to 26. However, as the output demonstrates, no change actually occurs the first time. The problem is that to call `MoveTo()`, the compiler unboxes `objectAngle` and (by definition) makes a copy of the value. Value types are copied by value—that is why they are called value types. Although the resultant value is successfully modified at execution time, this copy of the value is discarded and no change occurs on the heap location referenced by `objectAngle`.

is assigned 11, the value assigned to `Connected`. (In this case, you do not need to

Listing 9.31: Defining an Enum Type

```
enum ConnectionState : short
{
    Disconnected,
    Connecting = 10,
    Connected,
    Joined = Connected,
    Disconnecting
}
```

prefix `Connected` with the enum name, since it appears within its scope.) `Disconnecting` is 12.

An enum always has an underlying type, which may be any integral type other than `char`. In fact, the enum type's performance is identical to that of the underlying type. By default, the underlying value type is `int`, but you can specify a different type using inheritance type syntax. Instead of `int`, for example, Listing 9.15 uses a `short`. For consistency, the syntax for enums emulates the syntax of inheritance, but it doesn't actually create an inheritance relationship. The base class for all enums is `System.Enum`, which in turn is derived from `System.ValueType`. Furthermore, these enums are sealed; you can't derive from an existing enum type to add more members.

Guidelines

and performance advantages of value types. Programmers should not be overly concerned about using value types. Value types permeate virtually every chapter of this book, yet the idiosyncrasies associated with them come into play infrequently. We have staged the code surrounding each issue to demonstrate the concern, but in reality, these types of patterns rarely occur. The key to avoiding most of them is to follow the guideline of not creating mutable value types—and following this constraint explains why you don't encounter them within the built-in value types.

Perhaps the only issue to occur with some frequency is repetitive boxing operations within loops. However, generics greatly reduce boxing, and even without them, performance is rarely affected enough to warrant their avoidance until a particular algorithm with boxing is identified as a bottleneck.

This chapter also introduced enums. Enumerated types are a standard construct available in many programming languages. They help improve both API usability and code readability.

Chapter 10 presents more guidelines for creating well-formed types—both value types and reference types. It begins by defining operator-overloading methods. These two topics apply to both structs and classes, but they are somewhat more important when completing a struct definition and making it well formed.

Just a reminder, the section "Overriding object Members" has been moved to Chapter 9.

Output 10.1

```
51° 52' 0 E  -1° -20' 0 N
48° 52' 0 E  -2° -20' 0 N
51° 52' 0 E  -1° -20' 0 N
```

For `Coordinate`, you implement the − and + operators to return coordinate locations after adding/subtracting `Arc`. This allows you to string multiple operators and operands together, as in `result = ((coordinate1 + arc1) + arc2) + arc3`. Moreover, by supporting the same operators (+/-) on `Arc` (see Listing 10.4 later in this chapter), you could eliminate the parentheses. This approach works because the result of the first operand `(arc1 + arc2)` is another `Arc`, which you can then add to the next operand of type `Arc` or `Coordinate`. ?

In contrast, consider what would happen if you provided a − operator that had two `Coordinates` as parameters and returned a `double` corresponding to the distance between the two `coordinates`. Adding a `double` to a `Coordinate` is undefined, so you could not string together operators and operands. Caution is in order when defining operators that return a different type, because doing so is counterintuitive.

Combining Assignment with Binary Operators (+=, -=, *=, /=, %=, &=, ...)

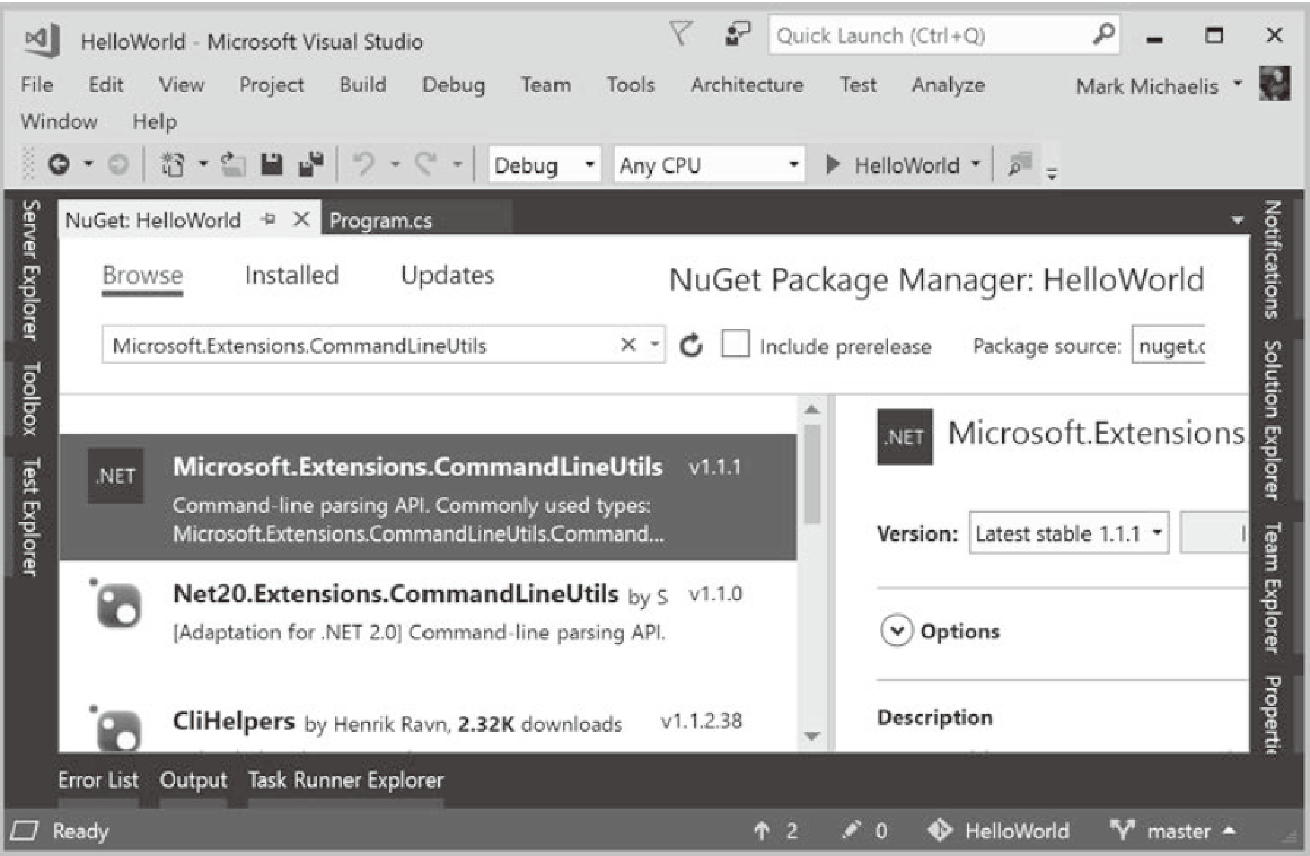


Figure 10.2: The Browse filter

```
        logger.LogInformation($@"Hospital Emergency Codes: = '{
            string.Join("', '", args)}'");
        // ...

        logger.LogWarning("This is a test of the emergency...");
        // ...
    }
}
```

Update Figure 10.2 to match the new Microsoft.Extensions.Logging.Console library.

Page 568-569

LISTING 10.9: Defining a File-Scoped Namespace

```
// Define the namespace AddisonWesley.Michaelis.EssentialCSharp
namespace AddisonWesley.Michaelis.EssentialCSharp;
class Program
{
```


? ?

```
    // ...
}
```

A **file-scoped namespace declaration** (added in C# 10.0) has a statement like syntax with the ending semicolon. The file-scoped namespace declaration must precede all other member definitions in the file and there can be only one such declaration. And, given the declaration, all members within the file will be assigned to that namespace. In Listing 10.9, for example, `Program` is placed into the namespace `AddisonWesley.Michaelis.EssentialCSharp`, making its full name `AddisonWesley.Michaelis.EssentialCSharp.Program`. If you are programming in C# 10.0 or later, I recommend using this form. It cuts down on unnecessary indentation and handles all standard cases of namespace declaration. Additionally, except for HelloWorld scenarios, you should specify a namespace for all your types.

Generating an XML Documentation File

The compiler checks that the XML comments are well formed and issues a warning if they are not. To generate the XML file, add a `DocumentationFile` element to the `ProjectProperties` element. PropertyGroup

```
<DocumentationFile>$(OutputPath)\$(TargetFramework)\$(AssemblyName).x
</DocumentationFile>
```

DataStorage

This element causes an XML file to be generated during the build into the output directory using the `<assemblyname>.xml` as the filename. Using the `CommentSamples` class listed earlier and the compiler options listed here, the resultant `CommentSamples.XML` file appears as shown in Listing 10.13.

LISTING 10.13: `Comments.xml`

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>DataStorage</name>
    </assembly>
    <members>
        <member name="T:DataStorage">
            <summary>
                DataStorage is used to persist and retrieve
```



```
                employee data from the files.
```

statement expressly for the purpose (see Listing 10.17).

Listing 10.17: Invoking the `using` Statement

```
public static class Program
{
    public static void Search()
    {
        // C# 8.0
```



```
        using TemporaryFileStream fileStream1 = new();

        // Prior to C# 8.0
        using (TemporaryFileStream fileStream2 =
            new(),
            fileStream3 = new())
        {
            // Use temporary file stream;
        }
    }
}
```

In the first highlighted code snippet, the resultant CIL code is identical to the code that would be created if the programmer specified an explicit try/finally block, where `fileStream.Dispose()` is called in the finally block. The `using` statement, however, provides a syntax shortcut for the try/finally block.

Within this `using` statement, you can instantiate more than one variable by separating each variable from the others with a comma. The key considerations are that all variables must be of the same type, the type must implement `IDisposable`, and initialization occurs at the time of declaration. To enforce the use of the same type, the data type is specified only once rather than before each variable declaration.

C# 8.0 introduces a potential simplification with regard to resource cleanup. As shown in the second highlighted snippet of Listing 10.17, you can prefix the declaration of a disposable resource (one that implements `IDisposable`) with the

Summary

This chapter provided a whirlwind tour of many topics related to building solid class libraries. All the topics pertain to internal development as well, but they are much more critical to building robust classes. Ultimately, the focus here was on forming more robust and programmable APIs. In the category of robustness, we can include

Chapter 10, Summary section:
This topic has been moved to Chapter 9.

namespaces and garbage collection. Both of these topics fit in the programmability category as well, along with overriding `object`'s virtual members, operator overloading, and XML comments for documentation.

Exception handling heavily depends on inheritance, by defining an exception hierarchy and enforcing custom exceptions to fit within this hierarchy. Furthermore, the C# compiler uses inheritance to verify catch block order. In Chapter 11, you will see why inheritance is such a core part of exception handling.

Specifying a Default Value with the `default` operator

Listing 12.11 included a constructor that takes the initial values for both `first` and `second` and assigns them to `First` and `Second`. Since `Pair<T>` is a struct, any constructor you provide must initialize all fields and automatically implemented properties. This presents a problem, however.

Consider a constructor for `Pair<T>` that initializes only half of the pair at instantiation time. Defining such a constructor, as shown in Listing 12.12, causes a compile-time error because the field `Second` is still uninitialized at the end of the constructor. Providing initialization for `Second` presents a problem because you don't know the data type of `T`. If it is a nullable type, `null` would work, but this approach would not work if `T` were a non-nullable type.

Listing 12.12: Not Initializing All Fields, Causing a Compile-Time Error

```
public struct Pair<T> : IPair<T>
{
    // ERROR:  Field 'Pair<T>.Second' must be fully assigned
    //         before control leaves the constructor
    // public Pair(T first)
    // {
    //     First = first;
    // }

    // ...
}
```

Only applicable for C# versions before 11.0. The introduction of the Auto-default struct feature in C# 11.0 resolves this issue.

To deal with this scenario, C# provides the `default` operator. The default value of `int`, for example,[2] could be specified with `default`. In the case of `T`, which `Second` requires, you can use `default`, as shown in Listing 12.13.

Listing 12.16: Using Arity to Overload a Type Definition

```
public class Tuple
{
    // ...
}
public class Tuple<T1> // : IStructuralEquatable,
                       // IStructuralComparable, IComparable
{
    // ...
}
public class Tuple<T1, T2> // : IStructuralEquatable,
                           // IStructuralComparable, IComparable
{
    // ...
}
public class Tuple<T1, T2, T3> // : IStructuralEquatable,
                               // IStructuralComparable, IComparable
{
    // ...
}
public class Tuple<T1, T2, T3, T4> // : IStructuralEquatable,
                                   // IStructuralComparable, IComparable
{
    // ...
}
public class Tuple<T1, T2, T3, T4, T5> // : IStructuralEquatable,
                                       // IStructuralComparable,
                                       // IComparable
{
    // ...
}
public class Tuple<T1, T2, T3, T4, T5, T6>
    // : IStructuralEquatable, IStructuralComparable, IComparable
{
    // ...
}
public class Tuple<T1, T2, T3, T4, T5, T6, T7>
    // : IStructuralEquatable, IStructuralComparable, IComparable
{
    // ...
}
public class Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>
    // : IStructuralEquatable, IStructuralComparable, IComparable
{
```

or ValueTuple?


```
    // ...
}
```

The `ValueTuple<...>` set of classes was designed for the same purpose as the

`Pair<T>` and `Pair<TFirst, TSecond>` classes, except that together they can handle eight type arguments. In fact, using the last `ValueTuple` shown in Listing 12.16, `TRest` can be used to store another `ValueTuple`, making the number of elements of the tuple practically unlimited.

Another interesting member of the tuple family of classes is the nongeneric `ValueTuple` class. This class has eight static factory methods for instantiating the various generic tuple types. Although each generic type could be instantiated directly using its constructor, the `ValueTuple` type's factory methods allow for inference of the type arguments via the `Create()` method. Listing 12.17 demonstrates using the `Create()` method in combination with type inference.[5]

Listing 12.17: Comparing `System.ValueTuple` Instantiation Approaches

```
#if !PRECSHARP7
(string, Contact) keyValuePair;
keyValuePair =
    ("555-55-5555", new Contact("Inigo Montoya"));
#else // Use System.ValueTupe<string,Contact> prior to C# 7.0
ValueTuple<string, Contact> keyValuePair;
keyValuePair =
    ValueTuple.Create(
        "555-55-5555", new Contact("Inigo Montoya"));
keyValuePair =
    new ValueTuple<string, Contact>(
        "555-55-5555", new Contact("Inigo Montoya"));
#endif // !PRECSHARP7
```

Obviously, when the `ValueTuple` gets large, the number of type parameters to specify could be cumbersome without the `Create()` factory methods.[6]

---

5. Simpler for C# 6.0.
6. Similar tuple class added in C# 4.0: `System.Tuple`.

To specify an interface for the constraint, you declare an **interface type constraint**. This constraint even circumvents the need to cast to call an explicit interface member implementation.

Type Parameter Constraints

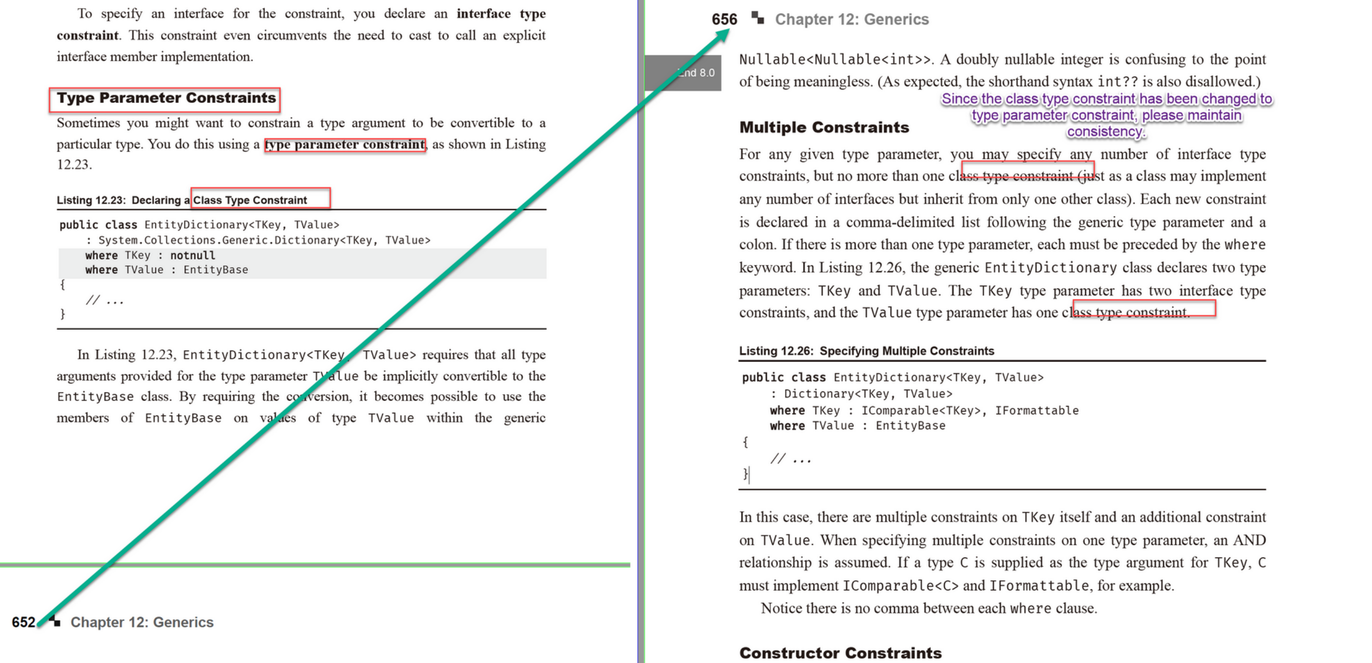
Sometimes you might want to constrain a type argument to be convertible to a particular type. You do this using a **type parameter constraint**, as shown in Listing 12.23.

Listing 12.23: Declaring a Class Type Constraint

```
public class EntityDictionary<TKey, TValue>
    : System.Collections.Generic.Dictionary<TKey, TValue>
    where TKey : notnull
    where TValue : EntityBase
{
    // ...
}
```

In Listing 12.23, `EntityDictionary<TKey, TValue>` requires that all type arguments provided for the type parameter `TValue` be implicitly convertible to the `EntityBase` class. By requiring the conversion, it becomes possible to use the members of `EntityBase` on values of type `TValue` within the generic

Page 656



To specify an interface for the constraint, you declare an **interface type constraint**. This constraint even circumvents the need to cast to call an explicit interface member implementation.

Type Parameter Constraints

Sometimes you might want to constrain a type argument to be convertible to a particular type. You do this using a **type parameter constraint**, as shown in Listing 12.23.

Listing 12.23: Declaring a Class Type Constraint

```
public class EntityDictionary<TKey, TValue>
    : System.Collections.Generic.Dictionary<TKey, TValue>
    where TKey : notnull
    where TValue : EntityBase
{
    // ...
}
```

In Listing 12.23, EntityDictionary<TKey, TValue> requires that all type arguments provided for the type parameter TValue be implicitly convertible to the EntityBase class. By requiring the conversion, it becomes possible to use the members of EntityBase on values of type TValue within the generic

652 Chapter 12: Generics

656 Chapter 12: Generics

End 8.0

Nullable<Nullable<int>>. A doubly nullable integer is confusing to the point of being meaningless. (As expected, the shorthand syntax int?? is also disallowed.)

Multiple Constraints

For any given type parameter, you may specify any number of interface type constraints, but no more than one class type constraint (just as a class may implement any number of interfaces but inherit from only one other class). Each new constraint is declared in a comma-delimited list following the generic type parameter and a colon. If there is more than one type parameter, each must be preceded by the where keyword. In Listing 12.26, the generic EntityDictionary class declares two type parameters: TKey and TValue. The TKey type parameter has two interface type constraints, and the TValue type parameter has one class type constraint.

Listing 12.26: Specifying Multiple Constraints

```
public class EntityDictionary<TKey, TValue>
    : Dictionary<TKey, TValue>
    where TKey : IComparable<TKey>, IFormattable
    where TValue : EntityBase
{
    // ...
}
```

In this case, there are multiple constraints on TKey itself and an additional constraint on TValue. When specifying multiple constraints on one type parameter, an AND relationship is assumed. If a type C is supplied as the type argument for TKey, C must implement IComparable<C> and IFormattable, for example.

Notice there is no comma between each where clause.

Constructor Constraints

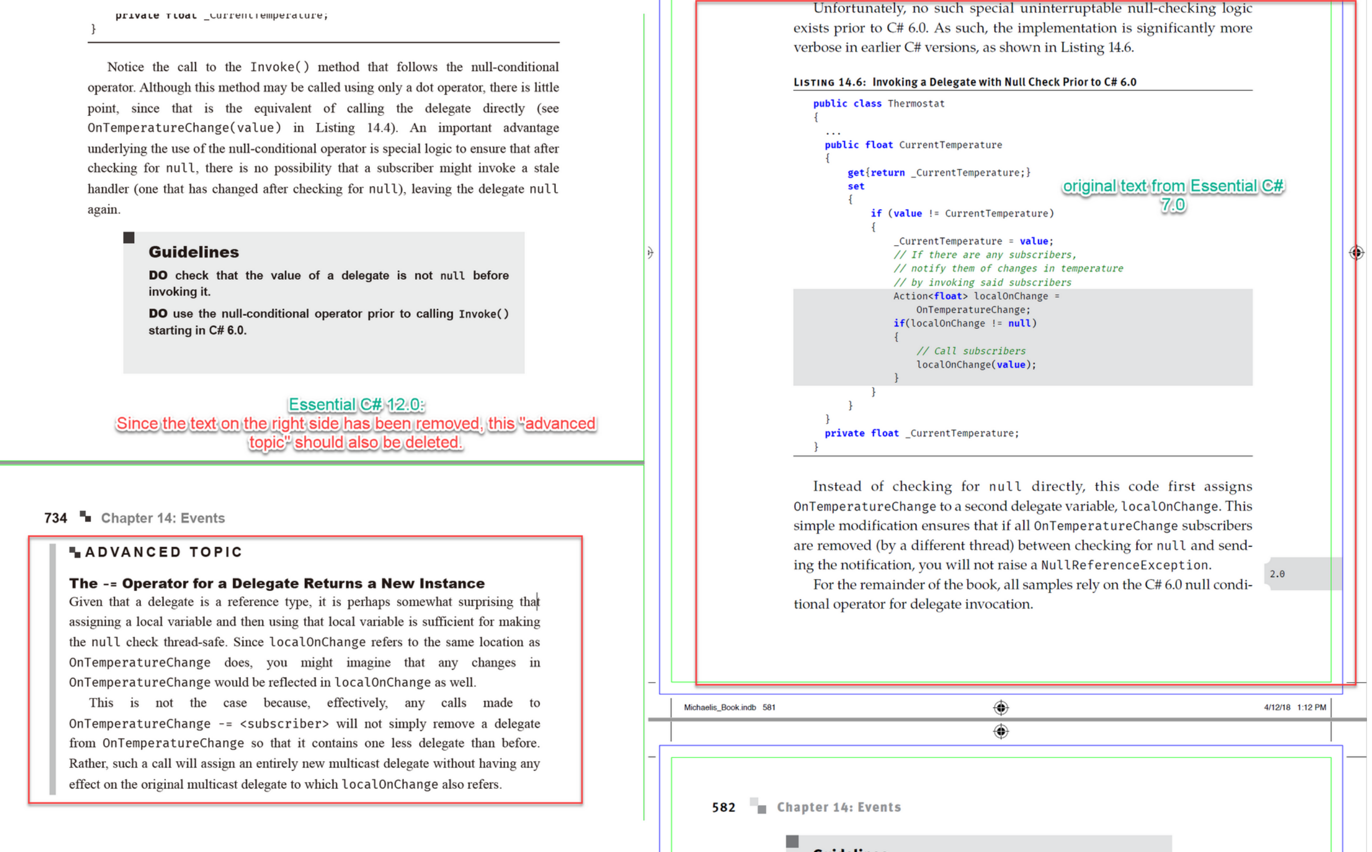
Page 682

682 Chapter 12: Generics

tackle these topics, we will investigate expressions, which provide a significant improvement[16] for working with collections.

16. Starting in C# 3.0.

Page 734



```
    private float _currentTemperature;
}
```

Notice the call to the Invoke() method that follows the null-conditional operator. Although this method may be called using only a dot operator, there is little point, since that is the equivalent of calling the delegate directly (see OnTemperatureChange(value) in Listing 14.4). An important advantage underlying the use of the null-conditional operator is special logic to ensure that after checking for null, there is no possibility that a subscriber might invoke a stale handler (one that has changed after checking for null), leaving the delegate null again.

Guidelines

DO check that the value of a delegate is not null before invoking it.

DO use the null-conditional operator prior to calling Invoke() starting in C# 6.0.

734 Chapter 14: Events

ADVANCED TOPIC

The -= Operator for a Delegate Returns a New Instance

Given that a delegate is a reference type, it is perhaps somewhat surprising that assigning a local variable and then using that local variable is sufficient for making the null check thread-safe. Since localOnChange refers to the same location as OnTemperatureChange does, you might imagine that any changes in OnTemperatureChange would be reflected in localOnChange as well.

This is not the case because, effectively, any calls made to OnTemperatureChange -= <subscriber> will not simply remove a delegate from OnTemperatureChange so that it contains one less delegate than before. Rather, such a call will assign an entirely new multicast delegate without having any effect on the original multicast delegate to which localOnChange also refers.

Unfortunately, no such special uninterruptable null-checking logic exists prior to C# 6.0. As such, the implementation is significantly more verbose in earlier C# versions, as shown in Listing 14.6.

Listing 14.6: Invoking a Delegate with Null Check Prior to C# 6.0

```
public class Thermostat
{
  ...
  public float CurrentTemperature
  {
      get{return _CurrentTemperature;}
      set
      {
          if (value != CurrentTemperature)
          {
              _CurrentTemperature = value;
              // If there are any subscribers,
              // notify them of changes in temperature
              // by invoking said subscribers
              Action<float> localOnChange =
                  OnTemperatureChange;
              if(localOnChange != null)
              {
                  // Call subscribers
                  localOnChange(value);
              }
          }
      }
  }
  private float _CurrentTemperature;
}
```


Instead of checking for null directly, this code first assigns OnTemperatureChange to a second delegate variable, localOnChange. This simple modification ensures that if all OnTemperatureChange subscribers are removed (by a different thread) between checking for null and sending the notification, you will not raise a NullReferenceException.

2.0

For the remainder of the book, all samples rely on the C# 6.0 null conditional operator for delegate invocation.

Michaelis_Book.indb 581 4/12/18 1:12 PM

582 Chapter 14: Events

Guidelines

Listing 14_09 (Source Code)

```
85              catch(AggregateException exception)
86              {
87                  Console.WriteLine(exception.Message);
88                  if (exception.InnerExceptions.Count < 1)        > 1
89                  {
90                      // Enumerate the exceptions only if there is more than
91                      // one because with one the message gets merged into the
92                      // Aggregate exception message.
93                      foreach (Exception item in exception.InnerExceptions)
94                      {
95                          Console.WriteLine("\t{0}: {1}",
96                              item.GetType(), item.Message);
97                      }
98                  }
99              }
100         }
101     }
```

Page 775

At the time of declaration, lambda expressions are not executed. In fact, it isn't until the lambda expressions are invoked that the code within them begins to execute. Figure 15.2 shows the sequence of operations. 15.16

As Figure 15.2 shows, three calls in Listing 15.14 trigger the lambda expression, and each time it is fairly implicit. If the lambda expression were expensive (such as a call to a database), it would therefore be important to minimize the lambda expression's execution.

First, the execution is triggered within the `foreach` loop. As described earlier in the chapter, the `foreach` loop breaks down into a `MoveNext()` call, and each call results in the lambda expression's execution for each item in the original collection. While iterating, the runtime invokes the lambda expression for each item to determine whether the item satisfies the predicate.

Program Enumerable Console IEnumerable<Patent> IEnumerable<Patent> IEnumerator

Page 787-Page 789

Listings 15.22 and 15.23 yield identical results. However, the book notes that Listing 15.22 omits the "Marketing" department, which is devoid of any associated employees.

Listing 16_02(Source Code)

EssentialCSharp / src / Chapter16 / Listing16.02.ProjectingUsingQueryExpressions.cs

BenjaminMichaelis Convert to File Scoped Namespaces (#472)

Code Blame 37 lines (34 loc) · 937 Bytes

```
1  namespace AddisonWesley.Michaelis.EssentialCSharp.Chapter16.Listing16_02;
2
3  #region INCLUDE
4  using System;
5  using System.Collections.Generic;
6  using System.IO;
7  using System.Linq;
8  #region EXCLUDE
9  public class Program
10 {
11     public static void Main()
12     {
13         List1(Directory.GetCurrentDirectory(), "*");
14     }
15     #endregion EXCLUDE
16     public static void List1
17         (string rootDirectory, string searchPattern)
18     {
19         #region HIGHLIGHT
20         IEnumerable<string> fileNames = Directory.GetFiles(
21             rootDirectory, searchPattern);
22
23         IEnumerable<FileInfo> fileInfos =
24             from fileName in fileNames
25             select new FileInfo(fileName);
26         #endregion HIGHLIGHT
27
28         foreach (FileInfo fileInfo in fileInfos)
29         {
30             Console.WriteLine(
31                 $@".{ fileInfo.Name } ({
32                     fileInfo.LastWriteTime })");
33         }
34     }
35     //...
36     #endregion INCLUDE
37 }
```

Actually, the dot seems unnecessary.
It can be removed.

```
        Console.WriteLine(
            $"5. delegateInvocations={ delegateInvocations }");

        // Cache the value so future counts will not trigger
        // another invocation of the query
        List<string> selectionCache = selection.ToList();

        Console.WriteLine(
            $"6. delegateInvocations={ delegateInvocations }");

        // Retrieve the count from the cached collection
        Console.WriteLine(
            $"7. selectionCache count={ selectionCache.Count }");

        Console.WriteLine(
            $"8. delegateInvocations={ delegateInvocations }");
    }
    //...
```

Output 16.4

```
1. delegateInvocations=0
2. Contextual keyword count=28
3. delegateInvocations=28
4. Contextual keyword count=28
5. delegateInvocations=56
6. delegateInvocations=84
7. selectionCache count=28
8. delegateInvocations=84
```

Please double-check the output of this code listing as it appears to differ from the actual results.

Listing 16.7: Sorting Using a Query Expression with an `orderby` Clause

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;
// ...
    public static void ListByFileSize1(
        string rootDirectory, string searchPattern)
    {
        IEnumerable<string> fileNames =
            from fileName in Directory.EnumerateFiles(
                rootDirectory, searchPattern)
            orderby (new FileInfo(fileName)).Length descending,
                fileName
            select fileName;

        foreach(string fileName in fileNames)
        {
            Console.WriteLine(fileName);
        }
    }
    //...
```

Listing 16.7 uses the `orderby` clause to sort the files returned by `Directory.GetFiles()` first by file size in descending order, and then by filename in ascending order. Multiple sort criteria are separated by commas, such that first the items are ordered by size, and then, if the size is the same, they are ordered by filename. `ascending` and `descending` are contextual keywords indicating the sort order direction. Specifying the order as ascending or descending is optional; if the direction is omitted (as it is here on `filename`), the default is `ascending`.

The `let` Clause

Listing 16.11: Selecting a Tuple Following the group Clause

```
using System;
using System.Collections.Generic;
using System.Linq;
// ...
    private static void GroupKeywords1()
    {
        IEnumerable<IGrouping<bool, string>> keywordGroups =
            from word in CSharp.Keywords
            group word by word.Contains('*');

        IEnumerable<(bool IsContextualKeyword,
            IGrouping<bool, string> Items)> selection =
            from groups in keywordGroups
            select
            (
                IsContextualKeyword: groups.Key,
                Items: groups
            );

        foreach (
            (bool isContextualKeyword, IGrouping<bool, string> items)
                in selection)
        {

            Console.WriteLine(Environment.NewLine + "{0}:",
                isContextualKeyword ?
                    "Contextual Keywords" : "Keywords");
            foreach (string keyword in items)
            {
                Console.Write(" " + keyword.Replace("*", null));
            }
        }
    }
    //...
```

It's not necessary to name the subcollection property Items.

Output 16.7

```
Keywords:
 abstract as base bool break byte case catch char checked class
const continue decimal default delegate do double else enum
event explicit extern false finally fixed float for foreach goto if
implicit in int interface internal is lock long namespace new null
operator out override object params private protected public
readonly ref return sbyte sealed short sizeof stackalloc static
string struct switch this throw true try typeof uint ulong unsafe
```

```
ushort using virtual unchecked void volatile while
Contextual Keywords:
 add alias ascending async await by descending dynamic equals from
get global group into join let nameof nonnull on orderby partial remove
select set value var where yield
```

The `group` clause results in a query that produces a collection of `IGrouping<TKey, TElement>` objects—just as the `GroupBy()` standard query operator did (see Chapter 15). The `select` clause in the subsequent query uses a tuple to effectively rename `IGrouping<TKey, TElement>.Key` to `IsContextualKeyword` and to name the subcollection property `Items`. With this change, the nested `foreach` loop uses `wordGroup.Items` rather than `wordGroup` directly, as shown in Listing 16.10. Another potential item to add to the tuple would be a count of the items within the subcollection. This functionality is already available through LINQ's `wordGroup.Items.Count()` method, however, so the benefit of adding it to the anonymous type directly is questionable.

Query Continuation with `into`

As you saw in Listing 16.11, you can use an existing query as the input to a second query. However, it is not necessary to write an entirely new query expression when you want to use the results of one query as the input to another. You can extend any query with a **query continuation clause** using the contextual keyword `into`. A query continuation is nothing more than syntactic sugar for creating two queries and using the first as the input to the second. The range variable introduced by the `into` clause (`groups` in Listing 16.11) becomes the range variable for the remainder of the query; any previous range variables are logically a part of the earlier query and cannot be used in the query continuation. Listing 16.12 rewrites the code of Listing 16.11 to use a query continuation instead of two queries.

Listing 16.12: Selecting without the Query Continuation

```
using System;
using System.Collections.Generic;
using System.Linq;
// ...
    private static void GroupKeywords1()
    {
        IEnumerable<(bool IsContextualKeyword,
```

Page 839


```
        list.Sort();

        Console.WriteLine(
            $"In alphabetical order { list[0] } is the "
            + $"first dwarf while { list[^1] } is the last.");

        list.Remove("Grumpy");
    }
}
```

Output 17.1

```
In alphabetical order Bashful is the first dwarf while Sneezy is the
last.
```

Best to match with the "index from end" operator (^) in the code.

C# is zero-index based; therefore, index 0 in Listing 17.1 corresponds to the first element and index 6 indicates the seventh element. Retrieving elements by index does not involve a search. Rather, it entails a quick and simple “jump” operation to a location in memory.

A `List<T>` is an ordered collection; the `Add()` method appends the given item

Page 851


```
    }
}
```

To create a dictionary that uses this equality comparer, you can use the constructor `new Dictionary<Contact, string>(new ContactEquality)`.

BEGINNER TOPIC

9

Requirements of Equality Comparisons

As discussed in Chapter 10, several important rules apply to the equality and hash code algorithms. Conformance to these rules is critical in the context of collections.

Finally, `GetHashCode()` and `Equals()` must not throw exceptions. Notice how the code in Listing 17.8 is careful to never dereference a `null` reference, for example.

To summarize, here are the key principles:

- Equal objects must have equal hash codes.
- The hash code of an object should not change for the life of the instance (at least while it is in the hash table).
- The hashing algorithm should quickly produce a well-distributed hash.

The hashing algorithm should avoid throwing exceptions in all possible object states.

Sorted Collections: `SortedDictionary<TKey, TValue>` and `SortedList<T>`

Listing 17.9(Source Code)

MarkMichaelis Added Span<T> example (#556) ✓

Code Blame 51 lines (45 loc) · 1.74 KB

```
namespace AddisonWesley.Michaelis.EssentialCSharp.Chapter17.Listing17_09;

using System.Diagnostics;
using System.Runtime.CompilerServices;

public class Program
{
    public static void Main()
    {
        #region INCLUDE
        string[] languages = new [] {
            "C#", "COBOL", "Java",
            "C++", "TypeScript", "Python",};

        // Create a Span<string> from the arrays first 3 elements.
        Span<string> languageSpan = languages.AsSpan(0, 2);
        languages[0] = "R";
        Assert(languages[0] == languageSpan[0]);
        Assert("R" == languageSpan[0]);
        languageSpan[0] = "Lisp";
        Assert(languages[0] == languageSpan[0]);
        Assert("Lisp" == languages[0]);

        int[] numbers = languages.Select(item => item.Length).ToArray();
        // Create a Span<string> from the arrays first 3 elements.
        Span<int> numbersSpan = numbers.AsSpan(0, 2);
        Assert(numbers[1] == numbersSpan[1]);
        numbersSpan[1] = 42;
        Assert(numbers[1] == numbersSpan[1]);
        Assert(42 == numbers[1]);

        const string bigWord = "supercalifragilisticexpialidocious";
        // Create a Span<char> from a suffix portion of the word.
        #if NET8_0_OR_GREATER
        ReadOnlySpan<char> expialidocious = bigWord.AsSpan(20..);
        #else // NET8_0_OR_GREATER
        ReadOnlySpan<char> expialidocious = bigWord.AsSpan(20, 14);
        #endif // NET8_0_OR_GREATER
        Assert(expialidocious.ToString() == "expialidocious");
#endregion INCLUDE
    }
```

Span<int>

Listing 17.9: `Span<T>` **Examples**

```
string[] languages = new [] {
    "C#", "COBOL", "Java",
    "C++", "TypeScript", "Python",};

// Create a Span<string> from the arrays first 3 elements.
Span<string> languageSpan = languages.AsSpan(0, 2);
languages[0] = "R";
Assert(languages[0] == languageSpan[0]);
Assert("R" == languageSpan[0]);
languageSpan[0] = "Lisp";
Assert(languages[0] == languageSpan[0]);
Assert("Lisp" == languages[0]);

int[] numbers = languages.Select(item => item.Length).ToArray();
// Create a Span<string> from the arrays first 3 elements.
Span<int> numbersSpan = numbers.AsSpan(0, 2);
Assert(numbers[1] == numbersSpan[1]);
numbersSpan[1] = 42;
Assert(numbers[1] == numbersSpan[1]);
Assert(42 == numbers[1]);

const string bigWord = "supercalifragilisticexpialidocious";
// Create a Span<char> from a suffix portion of the word.
#if NET8_0_OR_GREATER
ReadOnlySpan<char> expialidocious = bigWord.AsSpan(20..);
#else // NET8_0_OR_GREATER
ReadOnlySpan<char> expialidocious = bigWord.AsSpan(20, 14);
#endif // NET8_0_OR_GREATER
Assert(expialidocious.ToString() == "expialidocious");
```

To demonstrate the behavior of shared memory, review how we can assign a new value to `languages[0]` and the corresponding element in `languagesSpan` will also update, and vice versa. Furthermore, this behavior applies whether using a collection of reference types or value types.

There is also a `ReadOnlySpan<T>`, which allows using the same construct on an immutable array such as a `string`, rather than allocating an entirely new string. With `ReadOnlySpan<T>` we can have a new string that points to a slice of the old string, a significantly more performance-based approach when a slice of the original array is all that is needed.

Listing 17.13(Source Code)

EssentialCSharp / src / Chapter17 / Listing17.13.IteratorInterfacesPattern.cs

MarkMichaelis Added Span<T> example (#556)

Code Blame 46 lines (41 loc) · 1008 Bytes

```
namespace AddisonWesley.Michaelis.EssentialCSharp.Chapter17.Listing17_13;

#region INCLUDE
using System.Collections;
using System.Collections.Generic;

public class BinaryTree<T> :
#region HIGHLIGHT
IEnumerable<T>
#endregion HIGHLIGHT
{
    public BinaryTree(T value)
    {
        Value = value;
    }

    #region IEnumerable<T>
    #region HIGHLIGHT
    public IEnumerator<T> GetEnumerator()
    #endregion HIGHLIGHT
    {
        #region EXCLUDE
        return new List<T>.Enumerator(); // This will be implemented in 16.16
    }
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator(); // This will be implemented in 16.16
        #endregion EXCLUDE
    }
    #endregion IEnumerable<T>

    public T Value { get; }
    public Pair<BinaryTree<T>> SubItems { get; set; }
}
```

It is always safe to call GetEnumerator() again; "fresh" enumerator objects will be created when necessary.

Figure 17.8 shows a high-level sequence diagram of what takes place. Remember that the MoveNext() method appears on the IEnumerator<T> interface.

17.14 In Listing 17.13, the foreach statement at the call site initiates a call to GetEnumerator() on the CSharpBuiltInTypes instance called keywords. Given the iterator instance (referenced by iterator), foreach begins each iteration with a call to MoveNext(). Within the iterator, you yield a value back to the foreach statement at the call site. After the yield return statement, the GetEnumerator() method seemingly pauses until the next MoveNext() request. Back at the loop body, the foreach statement displays the yielded value on the screen. It then loops back around and calls MoveNext() on the iterator again. Notice that the second time, control picks up at the second yield return statement. Once again, the foreach displays on the screen what CSharpBuiltInTypes yielded and starts the loop again. This process continues until there are no more yield return statements within the iterator. At that point, the foreach loop at the call site terminates because MoveNext() returns false.

Page 886

```
private static void DisplayHelp()
```

2. The .NET Standard 1.6 added the `CommandLineUtils NuGet` package, which also provides a command-line parsing mechanism. For more information, see my MSDN article on the topic at http://itl.tc/sept2016.

update URL, in my case (Chinese Edition):
http://t.cn/EZsdDn9


```
{
    // Display the command-line help.
```

The code that checks for an attribute is relatively simple. Given a `PropertyInfo` object (obtained via reflection), you call `GetCustomAttributes()` and specify the attribute sought, then indicate whether to check any overloaded methods. (Alternatively, you can call the `GetCustomAttributes()` method without the attribute type to return all of the attributes.)

Although it is possible to place code for finding the `CommandLineSwitchRequiredAttribute` attribute within the `CommandLineHandler`'s code directly, it makes for better object encapsulation to place the code within the `CommandLineSwitchRequiredAttribute` class itself. This is frequently the pattern for custom attributes. What better location to place code for finding an attribute than in a static method on the attribute class?

Initializing an Attribute through a Constructor

The call to `GetCustomAttributes()` returns an array of objects that can be cast to an `Attribute` array. Because the attribute in our example didn't have any instance members, the only metadata information that it provided in the returned attribute was whether it appeared. Attributes can also encapsulate data, however. Listing 18.15 defines a `CommandLineAliasAttribute` attribute—a custom attribute that provides alias command-line options. For example, you can provide command-line support for `/Help` or `/?` as an abbreviation. Similarly, `/S` could provide an alias to `/Subfolders` that indicates the command should traverse all the subdirectories.

Listing 18.15: Providing an Attribute Constructor

```
public class CommandLineSwitchAliasAttribute : Attribute
{
    public CommandLineSwitchAliasAttribute(string alias)
    {
        Alias = alias;
    }
    public string Alias { get; }
}
public class CommandLineInfo
{
    [CommandLineSwitchAlias("?")]
    public bool Help { get; set; }
```


CommandLineSwitchAliasAttribute

```
    // ...
}
```

To support this functionality, you need to provide a constructor for the attribute. Specifically, for the alias, you need a constructor that takes a string argument. (Similarly, if you want to allow multiple aliases, you need to define an attribute that has a `params string` array for a parameter.)

When applying an attribute to a construct, only constant values and `typeof()` expressions are allowed as arguments. This constraint is required to enable their serialization into the resultant CIL. It implies that an attribute constructor should require parameters of the appropriate types; creating a constructor that takes arguments of type `System.DateTime` would be of little value, as there are no `System.DateTime` constants in C#.

The objects returned from `PropertyInfo.GetCustomAttributes()` will be initialized with the specified constructor arguments, as demonstrated in Listing 18.16.

Listing 18.16: Retrieving a Specific Attribute and Checking Its Initialization

```
PropertyInfo property =
    typeof(CommandLineInfo).GetProperty("Help")!;
CommandLineSwitchAliasAttribute? attribute =
    (CommandLineSwitchAliasAttribute?)
        property.GetCustomAttribute(
        typeof(CommandLineSwitchAliasAttribute), false);
if(attribute?.Alias == "?")
{
    Console.WriteLine("Help(?)");
};
```

Furthermore, as Listing 18.17 and Listing 18.18 demonstrate, you can use similar code in a `GetSwitches()` method on `CommandLineAliasAttribute` that returns a dictionary collection of all the switches, including those from the property names, and associate each name with the corresponding attribute on the command-line object.

```
{
}
```

Output 18.6

```
...Program+CommandLineInfo.cs(24,17): error CS0592: Attribute
'CommandLineSwitchAlias' is not valid on this declaration type. It is
valid on 'property, indexer' declarations only.
```

`AttributeUsageAttribute`'s constructor takes an `AttributeTargets` flag. This enum provides a list of all possible targets that the runtime allows an attribute to decorate. For example, if you also allowed `CommandLineSwitchAliasAttribute` on a field, you would update the `AttributeUsageAttribute` class, as shown in Listing 18.21.

Listing 18.21: Limiting an Attribute's Usage with `AttributeUsageAttribute`

```
// Restrict the attribute to properties and methods
[AttributeUsage(
  AttributeTargets.Field | AttributeTargets.Property)]
public class CommandLineSwitchAliasAttribute : Attribute
{
    // ...
}
```

Guidelines

DO apply the `AttributeUsageAttribute` class to custom attributes.

writes out the strings for each enumeration flag that is set. In Listing 18.23, `file.Attributes.ToString()` returns `"ReadOnly, Hidden"` rather than the 3 it would have returned without the `FlagsAttribute` flag. If two enumeration values are the same, the `ToString()` call would return the first one. As mentioned earlier, however, you should use caution when relying on this outcome because it is not localizable.

Parsing a value from a string to the enumeration also works, provided that each enumeration value identifier is separated by a comma.

Note that `FlagsAttribute` does not automatically assign the unique flag values or check that flags have unique values. The values of each enumeration item still must be assigned explicitly.

Predefined Attributes

CommandLineSwitchRequiredAttribute

The `AttributeUsageAttribute` attribute has a special characteristic that you haven't seen yet in the custom attributes you have created in this book. This attribute affects the behavior of the compiler, causing it to sometimes report an error. Unlike the reflection code that you wrote earlier for retrieving `CommandLineRequiredAttribute` and `CommandLineSwitchAliasAttribute`, `AttributeUsageAttribute` has no runtime code; instead, it has built-in compiler support.

`AttributeUsageAttribute` is a predefined attribute. Not only do such attributes provide additional metadata about the constructs they decorate, but the runtime and compiler also behave differently to facilitate these attributes' functionality. Attributes such as `AttributeUsageAttribute`, `FlagsAttribute`, `ObsoleteAttribute`, and `ConditionalAttribute` are examples of predefined attributes. They implement special behavior that only the CLI provider or compiler can offer because there are no extension points for additional noncustom attributes. In contrast, custom attributes are entirely passive. Listing 18.23 includes a couple of predefined attributes; Chapter 19 includes a few more.

The C# compiler recognizes the attribute on a called method during compilation; if the preprocessor identifier is not defined, it then eliminates any calls to the method.

This example defined `CONDITION_A`, so `MethodA()` executed normally. `CONDITION_B`, however, was not defined either through `#define` or by using the `csc.exe /Define` option. As a result, all calls to `Program.MethodB()` from within this assembly will do nothing.

Functionally, `ConditionalAttribute` is similar to placing an `#if/#endif` around the method invocation. The syntax is cleaner, however, because developers create the effect by adding the `ConditionalAttribute` attribute to the target method without making any changes to the caller itself.

The C# compiler notices the attribute on a called method during compilation; assuming the preprocessor identifier exists, it then eliminates any calls to the method. `ConditionalAttribute`, however, does not affect the compiled CIL code on the target method itself (besides the addition of the attribute metadata). Instead, it affects

Listing 18.28: Invoking a `Caller*` Attributes Method

```
ExpectedException<DivideByZeroException>.AssertExceptionThrown(
    () => throw new DivideByZeroException());
```

Now, even though no argument is specified in the source code for the `testActionMemberName` parameter (for example), the string value "Method" still is injected and available in the `AssertExceptionThrown()` method. Obviously, there is a limited set of such `Caller*` attributes as shown in Table 18.1.

Table 18.1: Caller* Attributes[7]

Attribute	Description	Type
`CallerFilePathAttribute`	Full path of the source file that contains the caller. The full path is the path at compile time.	`string`
`CallerLineNumberAttribute`	Line number in the source file from which the method is called.	`System.Int32`
`CallerMemberNameAttribute`	Method name or property name of the caller.	`string`
`CallerArgumentExpressionAttribute`	String representation of the argument expression.	`string`

All the `Caller*` attributes are limited to use on parameter constructs and mostly self-explanatory from Listing 18.27. The one exception is the `CallerArgumentExpressionAttrribute` since it requires its own parameter. The `CallerArgumentExpressionAttrribute` identifies (as text) the expression used for another parameter within the method. For example, in Listing 18.27 the expression `() => throw new Exception()` is for the value of `testAction`. And, since the `CallerArgumentExpressionAttrribute` decorating the `testExpression` parameter has `nameof(testAction)` as a parameter for the

7 https://learn.microsoft.com /dotnet/csharp/language-reference/attributes/caller-information

Page 926-927

To understand this apparent paradox, let's reexamine Listing 18.31. Notice the call to retrieve the "FirstName" element:

18.30

10. Functionality added in C# 4.0.

First()

```
Element.Descendants("LastName").FirstOrDefault().Value
```

The listing uses a `string` ("LastName") to identify the element name, but no compile-time verification occurs to ensure that the string is correct. If the casing was inconsistent with the element name or if there was a space, the compile would still succeed, even though a `NullReferenceException` would occur with the call to the `Value` property. Furthermore, the compiler does not attempt to verify that the "FirstName" element even exists; if it doesn't, we would also get the `NullReferenceException` message. In other words, in spite of all the type-safety advantages, type safety doesn't offer many benefits when you're accessing the dynamic data stored within the XML element.

a synchronous delegate into an asynchronous task. The worker thread writes plus signs to the console, while the main thread writes hyphens.

Starting the task obtains a thread from the thread pool, creating a second point of control, and executes the delegate on that thread. As shown in Listing 19.1, the point of control on the main thread continues normally after the call to start the task (`Task.Run()`).

Listing 19.1: Invoking an Asynchronous Task

```
using System;
using System.Threading.Tasks;

public class Program
{
    public static void Main()
    {
        const int repetitions = 10000;
        // Use Task.Factory.StartNew<string>() for
        // TPL prior to .NET 4.5
        Task task = Task.Run(() =>
            {
                for(int count = 0; count < repetitions; count++)
                {
                    Console.Write('-');
                }
            });
        for(int count = 0; count < repetitions; count++)
        {
            Console.Write('+');
        }
```

```
        try
        {
            Clock.Start();
            // Register a callback to receive notifications
            // of any unhandled exception

            AppDomain.CurrentDomain.UnhandledException +=
              (s, e) =>
              {
                  Message("Event handler starting");
                  Delay(4000);
              };
            Thread thread = new(() =>
            {
                Message("Throwing exception.");
                throw new Exception();
            });
            thread.Start();

            Delay(2000);
        }
        finally
        {
            Message("Finally block running.");
        }
    }

    static void Delay(int i)
    {
        Message($"Sleeping for {i} ms");
        Thread.Sleep(i);
        Message("Awake");
    }

    static void Message(string text)
    {
        Console.WriteLine("{0}:{1:0000}:{2}",
            Thread.CurrentThread.ManagedThreadId,
            Clock.ElapsedMilliseconds, text);
```



```
    }
}
```

OUTPUT 19.4

```
3:0047:Throwing exception.
3:0052:Unhandled exception handler starting.
3:0055:Sleeping for 4000 ms
1:0058:Sleeping for 2000 ms
1:2059:Awake
1:2060:Finally block running.
3:4059:Awake
Unhandled Exception: System.Exception: Exception of type 'System.
Exception' was thrown.
```

Page 996-997

Thus, this listing is both producing and consuming an async stream.

The signature for `GetAsyncEnumerator()` includes a `CancellationToken` parameter. Because the `await foreach` loop generates the code that calls `GetAsyncEnumerator()`, the way to inject a cancellation token and provide cancellation is via the `WithCancellation()` extension method. As Figure 20.2 shows, there's no `WithCancellation()` method on `IAsyncEnumerable<T>` directly. To support cancellation in an async stream method, add an optional `CancellationToken` with an `EnumeratorCancellationAttribute` as demonstrated by the `EncryptFilesAsunc` method declaration:

```
static public async IAsyncEnumerable<string>
EncryptFilesAsync(
string directoryPath = null,
string searchPattern = "*",
[EnumeratorCancellation] CancellationToken
```

typo


```
cancellationToken = default)
{ ... }
```

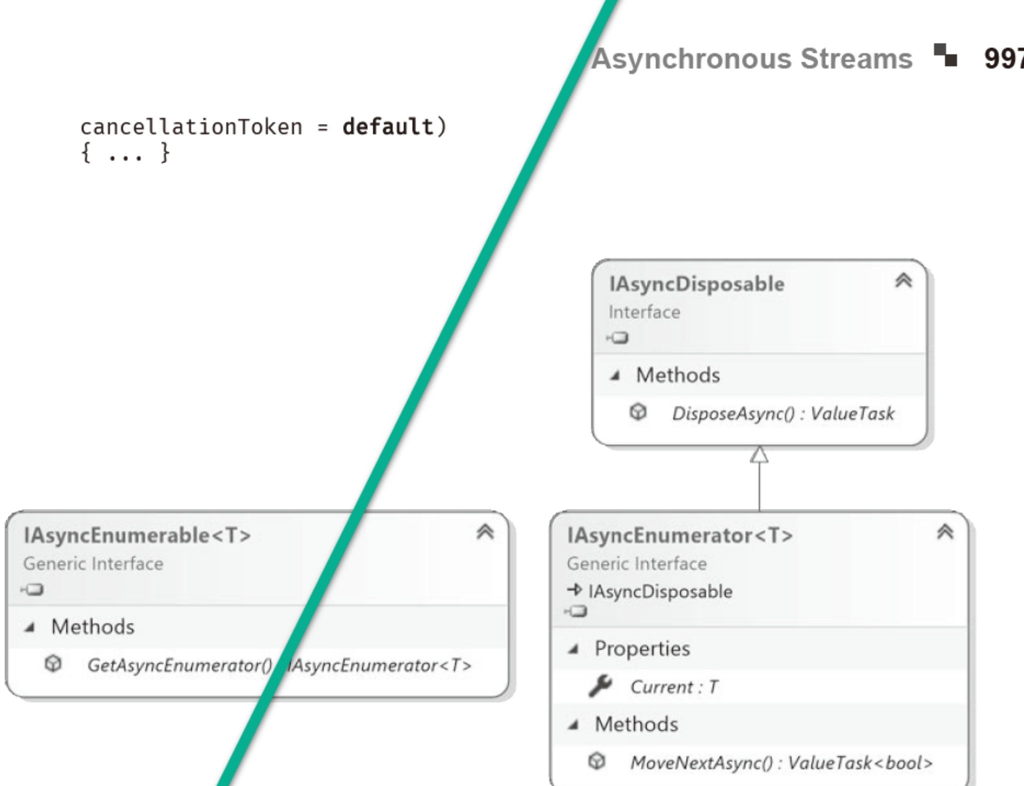


Figure 20.2: `IAsycnEnumerable<T>` and related interfaces

In Listing 20.6, you provide an async stream method that returns the `IAsyncEnumerable<T>` interface. As with the non-`async` iterators, however, you

Regardless of whether the `await` statements occur within an iteration or as separate entries, they will execute serially, one after the other and in the same order they were invoked from the calling thread. The underlying implementation is to string them together in the semantic equivalent of `Task.ContinueWith()` except that all of the code between the `await` operators will execute in the caller's synchronization context.

The need to support TAP from the UI is one of the key scenarios that led to TAP's creation. A second scenario takes place on the server, when a request comes in from a client to query an entire table's worth of data from the database. As querying the data could be time-consuming, a new thread should be created rather than consuming one from the limited number allocated to the thread pool. The problem with this approach is that the work to query from the database is executing entirely on another machine. There is no reason to block an entire thread, given that the thread is generally not active anyway.

To summarize, TAP was created to address these key problems:

- There is a need to allow long-running activities to occur without blocking the UI thread.

This paragraph is somewhat confusing, and I suggest rephrased it as follows, do you think it's appropriate?

Supporting TAP from the UI is a fundamental scenario that led to the development of TAP. Another common scenario occurs on the server side when a client's request involves retrieving a large dataset from a database. Given that database queries can be time-consuming, it's crucial to handle them efficiently. Instead of creating new threads or using threads from the limited thread pool, it's advisable to utilize asynchronous programming patterns. These patterns avoid blocking threads while waiting for the database response, especially since the actual query operations are executed on a separate machine (the database server). This approach ensures that server resources are used optimally, allowing threads to manage other tasks rather than being idle during I/O operations.

- Creating a new thread (or `Task`) for non–CPU-intensive work is relatively expensive when you consider that all the thread is doing is waiting for the activity to complete.

parallel. These types of computations are the easiest ones to speed up by adding parallelism.

Listing 21.1: A `for` Loop Synchronously Calculating Pi in Sections

```
using System;
using AddisonWesley.Michaelis.EssentialCSharp.Shared;

public class Program
{
    const int TotalDigits = 100;
    const int BatchSize = 10;

    public static void Main()
    {
        string pi = "";
        const int iterations = TotalDigits / BatchSize;
        for(int i = 0; i < iterations; i++)
        {
            pi += PiCalculator.Calculate(
                BatchSize, i * BatchSize);
        }

        Console.WriteLine(pi);
    }
}
// ...
using System;

public static class PiCalculator
{
    public static string Calculate(
        int digits, int startingAt)
    {
        //...
    }

    //...
}
// ...
```

The code and its output are inconsistent.

The `for` loop executes each iteration synchronously and sequentially. However, because the pi calculation algorithm splits the pi calculation into independent pieces, it is not necessary to compute the pieces sequentially providing the results are

Output 21.1

```
>3.141592653589793238462643383279502884197169399375105820974944592307816
```

```
4062862089986280348253421170679821480865132823066470938446095505822317253
5940812848111745028410270193852110555964462294895493038196442881097566593
3446128475648233786783165271201909145648566923460348610454326648213393607
2602491412737245870066063155881748815209209628292540917153643678925903600
11330530548820466521384146951941511609433057270365759591953092186117
38193261179310511854807446237996274956735188575272489122793818301194912
```

Canceling a Parallel Loop

Unlike a task, which requires an explicit call if it is to block until it completes, a parallel loop executes iterations in parallel but does not itself return until the entire parallel loop completes. Canceling a parallel loop, therefore, generally involves the invocation of the cancellation request from a thread other than the one executing the parallel loop. In Listing 21.5, we invoke `Parallel.ForEach<T>()` using `Task.Run()`. In this manner, not only does the query execute in parallel, but it also executes asynchronously, allowing the code to prompt the user to "Press any key to exit."

Listing 21.5: Canceling a Parallel Loop

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    // ...

    static void EncryptFiles(
        string directoryPath, string searchPattern)
    {
```



```
        string stars =
            "*".PadRight(Console.WindowWidth - 1, '*');

        IEnumerable<string> files = Directory.GetFiles(
           directoryPath, searchPattern,
           SearchOption.AllDirectories);

        CancellationTokenSource cts = new();
        ParallelOptions parallelOptions = new()
            { CancellationToken = cts.Token };
        cts.Token.Register(
            () => Console.WriteLine("Canceling..."));

        Console.WriteLine("Press ENTER to exit.");

        Task task = Task.Run(() =>
            {
                try
                {
```

Page 1056
![edbe803adf420a38d084dea23544db15.png](:/caea6069b02443fc9a46c96e1fee4cd1

Page 1058


```
        this, new TemperatureEventArgs(value));
}
```

This code is valid as long as no race condition arises between this method and the event subscribers. However, the code is not atomic, so multiple threads could introduce a race condition. It is possible that between the time when `OnTemperatureChange` is checked for `null` and when the event is actually fired, `OnTemperatureChange` could be set to `null`, thereby throwing a `NullReferenceException`. In other words, if multiple threads could potentially access a delegate simultaneously, it is necessary to synchronize the assignment and firing of the delegate.

All that is necessary is to use the null-conditional operator:

```
OnTemperatureChanged
 OnTemperature?.Invoke(
 this, new TemperatureEventArgs( value ) );
```

WaitHandle

The base class for `Mutex` is `System.Threading.WaitHandle`. It is a fundamental synchronization class used by the `Mutex`, `EventWaitHandle`, and `Semaphore` synchronization classes. The key methods on a `WaitHandle` are the `WaitOne()` methods, which block execution until the `WaitHandle` instance is signaled or set. The `WaitOne()` methods include several overloads allowing for an indefinite wait: `void WaitOne()`, a millisecond-timed wait; `bool WaitOne(int milliseconds)`; and `bool WaitOne(TimeSpan timeout)`, a `TimeSpan` wait. The versions that return a Boolean will return a value of `true` whenever the `WaitHandle` is signaled before the timeout.

In addition to the `WaitHandle` instance methods, there are two key static members: `WaitAll()` and `WaitAny()`. Like their instance cousins, these static members support timeouts. In addition, they take a collection of `WaitHandles`, in the form of an array, so that they can respond to signals coming from within the collection.

Note that `WaitHandle` contains a handle (of type `SafeWaitHandle`) that implements `IDisposable`. As such, care is needed to ensure that `WaitHandles` are disposed when they are no longer needed.

The WaitOne() methods include several overloads: bool WaitOne() for an indefinite wait, bool WaitOne(int milliseconds) for a wait timed in milliseconds, and bool WaitOne(TimeSpan timeout) for a wait determined by a TimeSpan duration.

Page 1066

Revise the prompts to correspond with Listing 22.10.

Table 22.3: Execution Path with ManualResetEvent Synchronization

Main()	DoWork()
...	
Console.WriteLine("Application started....");	
Task task = new Task(DoWork);	
Console.WriteLine("Starting thread....");	
task.Start();	
_DoWorkSignaledResetEvent.Wait();	Console.WriteLine("DoWork() started....");
	_DoWorkSignaledResetEvent.Set();
Console.WriteLine("Thread executing...");	_MainSignaledResetEvent.Wait();
_MainSignaledResetEvent.Set();	
task.Wait();	Console.WriteLine("DoWork() ending....");
Console.WriteLine("Thread completed");	
Console.WriteLine("Application exiting....");	

Page 1069

Table 22.4: Concurrent Collection Classes (continued)

Collection Class	Description
ConcurrentDictionary<TKey, TValue>	A thread-safe dictionary; a collection of keys and values.
ConcurrentQueue<T>*	A thread-safe queue supporting first in, first out (FIFO) semantics on objects of type T.
ConcurrentStack<T>*	A thread-safe stack supporting first in, last out (FILO) semantics on objects of type T.

missing this one

* Collection classes that implement IProducerConsumerCollection<T>.

A common pattern enabled by concurrent collections is support for thread-safe access by producers and consumers. Classes that implement IProducerConsumerCollection<T> (identified by an asterisk in Table 22.4) are specifically designed to provide such support. This enables one or more classes to pump data into the collection while a different set of classes reads it out, removing

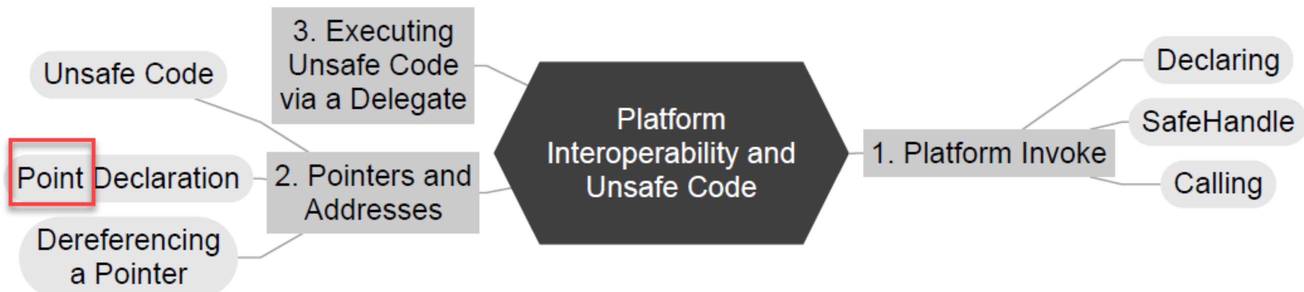
Chapter 20

`async`/`await` pattern[5] and the `Task.Delay()` method added in .NET 4.5. As we pointed out in Chapter 19, one key feature of TAP is that the code executing after an `async` call will continue in a supported thread context, thereby avoiding any UI cross-threading issues. Listing 22.13 provides an example of how to use the `Task.Delay()` method.

23

Platform Interoperability and Unsafe Code

C# has great capabilities, especially when you consider that the underlying framework is entirely managed. Sometimes, however, you need to escape out of all the safety that C# provides and step back into the world of memory addresses and pointers. C# supports this action in two significant ways. The first option is to go through Platform Invoke (P/Invoke) and calls into APIs exposed by unmanaged dynamic link libraries (DLLs). The second way is through **unsafe code**, which enables access to memory pointers and addresses.



The majority of the chapter discusses interoperability with unmanaged code and the use of unsafe code. This discussion culminates with a small program that determines the processor ID of a computer. The code requires that you do the following:

Parameter Data Types

Assuming the developer has identified the targeted DLL and exported function, the most difficult step is identifying or creating the managed data types that correspond to the unmanaged types in the external function.[1] Listing 23.2 shows a more difficult API.

LISTING 23.2: The `VirtualAllocEx()` API

```
LPVOID VirtualAllocEx(
    HANDLE hProcess,        // The handle to a process. The
                            // function allocates memory within
                            // the virtual address space of this
                            // process.
    LPVOID lpAddress,       // The pointer that specifies a
                            // desired starting address for the
                            // region of pages that you want to
                            // allocate. If lpAddress is NULL,
                            // the function determines where to
                            // allocate the region.
    SIZE_T dwSize,          // The size of the region of memory to
                            // allocate, in bytes. If lpAddress
                            // is NULL, the function rounds dwSize
                            // up to the next page boundary.
    DWORD flAllocationType, // The type of memory allocation
    DWORD flProtect);       // The type of memory allocation
```

Page 1093

DO simplify the wrapper methods by choosing default values for unnecessary parameters.

DO use the `SetLastErrorAttribute` on `Windows` to turn APIs that use `SetLastError` error codes into methods that throw `Win32Exception`.

DO extend `SafeHandle` or implement `IDisposable` and create a finalizer to ensure that unmanaged resources can be cleaned up effectively.

DO use delegate types that match the signature of the desired method when an unmanaged API requires a function pointer.

DO use `ref` parameters rather than pointer types when possible.

DO use the `SetLastError` field of `DllImport` attribute on Windows to turn APIs that use `SetLastError` error codes into methods that throw Win32Exception.

Page 1105


```
↪void>)(IntPtr)codeBytesPtr;
                    method(&buffer[0]);
                }
                Console.Write("Processor Id: ");
                char[] chars = new char[Buffer.Length];
                Encoding.ASCII.GetChars(buffer, chars);
                Console.WriteLine(chars);
            } // unsafe
        }
        else
        {
            Console.WriteLine("This sample is only valid for Windows");
        }
        return 0;
    }
}

[System.Runtime.CompilerServices.InlineArrayAttribute(Length)]
public struct Buffer
{
    public const int Length = 10;

    private byte _element0;
}
```

12(at least)

Output 23.6

```
Processor Id: GenuineIntel
```