

## CODE LISTINGS AND FIGURES

# CHAPTER 1

```
1 public class Bookshelf
2 {
3     public IEnumerable Books { get { ... } }
4 }
```

---

```
1 public class Bookshelf
2 {
3     public IEnumerable<Book> Books { get { ... } }
4 }
```

---

```
1 string Method(string x, string? y)
```

---

```
1 var book = new { Title = "Lost in the Snow", Author = "Holly Webb" };
2 string title = book.Title;
3 string author = book.Author;
```

---

1

```
1 Dictionary<string, string> map1 = new Dictionary<string, string>();
2
3 var map2 = new Dictionary<string, string>();
```

---

1  
2

```
1 var book = (title: "Lost in the Snow", author: "Holly Webb");
2 Console.WriteLine(book.title);
```

---

```
1 button.Click += new EventHandler(HandleButtonClick);
```

1

```
1 button.Click += HandleButtonClick;
```

1

```
1 button.Click += delegate { MessageBox.Show("Clicked!"); };
```

1

```
1 button.Click += (sender, args) => MessageBox.Show("Clicked!");
```

1

```
1 var customer = new Customer();
2 customer.Name = "Jon";
3 customer.Address = "UK";
4 var item1 = new OrderItem();
5 item1.ItemId = "abcd123";
6 item1.Quantity = 1;
7 var item2 = new OrderItem();
8 item2.ItemId = "fghi456";
9 item2.Quantity = 2;
10 var order = new Order();
11 order.OrderId = "xyz";
12 order.Customer = customer;
13 order.Items.Add(item1);
14 order.Items.Add(item2);
```

```
1 var order = new Order
2 {
```

```
3      OrderId = "xyz",
4      Customer = new Customer { Name = "Jon", Address = "UK" },
5      Items =
6      {
7          new OrderItem { ItemId = "abcd123", Quantity = 1 },
8          new OrderItem { ItemId = "fghi456", Quantity = 2 }
9      }
10 };
```

---

```
1 private string name;
2 public string Name
3 {
4     get { return name; }
5     set { name = value; }
6 }
```

---

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

---

```
1 public int Count { get { return list.Count; } }
2
3 public IEnumerable<string> GetEnumerator()
4 {
5     return list.GetEnumerator();
6 }
```

---

```
1 public int Count => list.Count;
2
3 public IEnumerable<string> GetEnumerator() => list.GetEnumerator();
```

---

```
1 throw new KeyNotFoundException(
2     "No calendar system for ID " + id + " exists");
```

---

```
1 throw new KeyNotFoundException(  
2     string.Format("No calendar system for ID {0} exists", id));
```

---

```
1 throw new KeyNotFoundException($"No calendar system for ID {id} exists");
```

---

```
1 var offers =  
2     from product in db.Products  
3     where product.SalePrice <= product.Price / 2  
4     orderby product.SalePrice  
5     select new {  
6         product.Id, product.Description,  
7         product.SalePrice, product.Price  
8     };
```

---

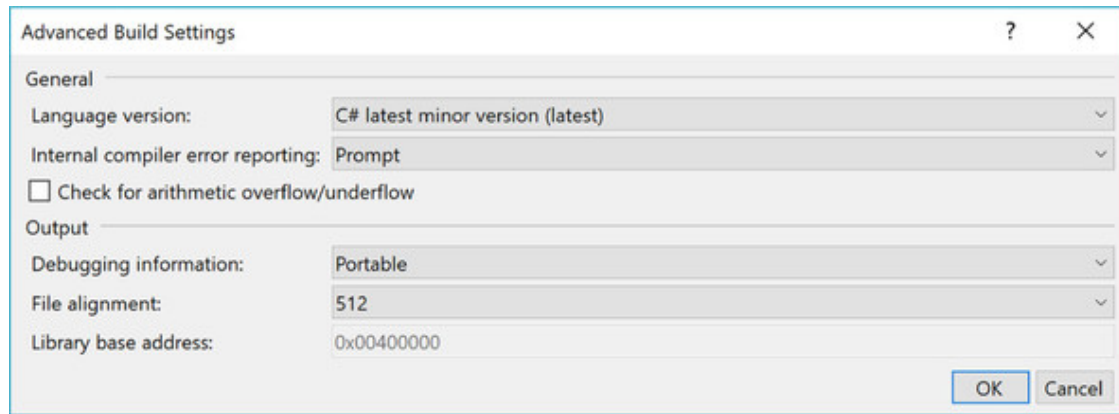
```
1 private async Task UpdateStatus()  
2 {  
3     Task<Weather> weatherTask = GetWeatherAsync();  
4     Task<EmailStatus> emailTask = GetEmailStatusAsync(); 1  
5     Weather weather = await weatherTask;  
6     EmailStatus email = await emailTask; 2  
7  
8     weatherLabel.Text = weather.Description;  
9     inboxLabel.Text = email.InboxCount.ToString();  
10 } 3
```

---

```
1 <PropertyGroup>  
2     ... 1  
3     <LangVersion>latest</LangVersion> 2  
4 </PropertyGroup>
```

---

**Figure 1.1. Language version settings in Visual Studio**



# CHAPTER 2

## Listing 2.1. Generating and printing names by using arrays

```
1 static string[] GenerateNames()
2 {
3     string[] names = new string[4];
4     names[0] = "Gamma";
5     names[1] = "Vlissides";
6     names[2] = "Johnson";
7     names[3] = "Helm";
8     return names;
9 }
10
11 static void PrintNames(string[] names)
12 {
13     foreach (string name in names)
14     {
15         Console.WriteLine(name);
16     }
17 }
```

1

## Listing 2.2. Generating and printing names by using `ArrayList`

```
1 static ArrayList GenerateNames()
2 {
3     ArrayList names = new ArrayList();
4     names.Add("Gamma");
5     names.Add("Vlissides");
6     names.Add("Johnson");
7     names.Add("Helm");
8     return names;
9 }
10 static void PrintNames(ArrayList names)
11 {
12     foreach (string name in names)
13     {
14         Console.WriteLine(name);
15     }
16 }
```

1

## Listing 2.3. Generating and printing names by using `StringCollection`

```
1 static StringCollection GenerateNames()
2 {
3     StringCollection names = new StringCollection();
```

```

4     names.Add("Gamma");
5     names.Add("Vlissides");
6     names.Add("Johnson");
7     names.Add("Helm");
8     return names;
9 }
10
11 static void PrintNames(StringCollection names)
12 {
13     foreach (string name in names)
14     {
15         Console.WriteLine(name);
16     }
17 }

```

---

## Listing 2.4. Generating and printing names with `List<T>`

```

1 static List<string> GenerateNames()
2 {
3     List<string> names = new List<string>();
4     names.Add("Gamma");
5     names.Add("Vlissides");
6     names.Add("Johnson");
7     names.Add("Helm");
8     return names;
9 }
10
11 static void PrintNames(List<string> names)
12 {
13     foreach (string name in names)
14     {
15         Console.WriteLine(name);
16     }
17 }

```

---

Figure 2.1. Relationship between method parameters and arguments

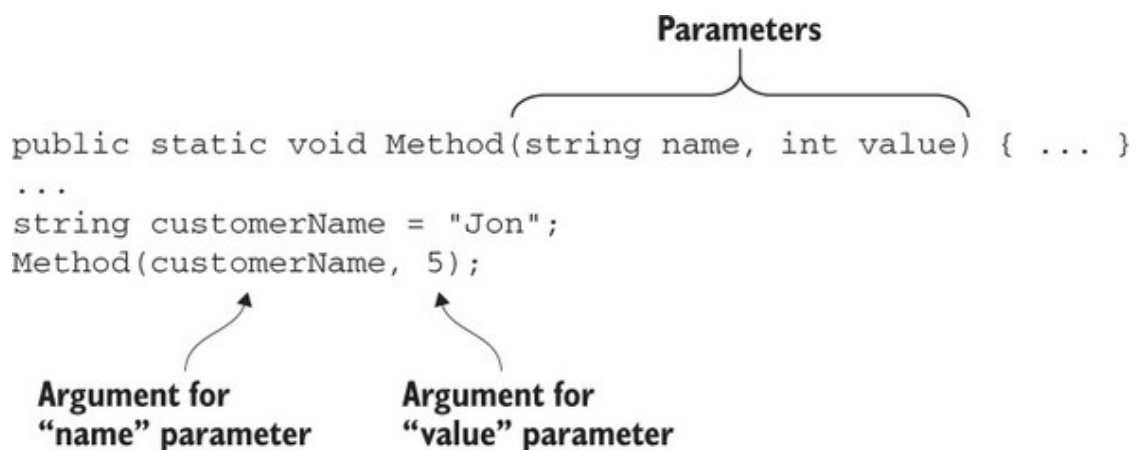
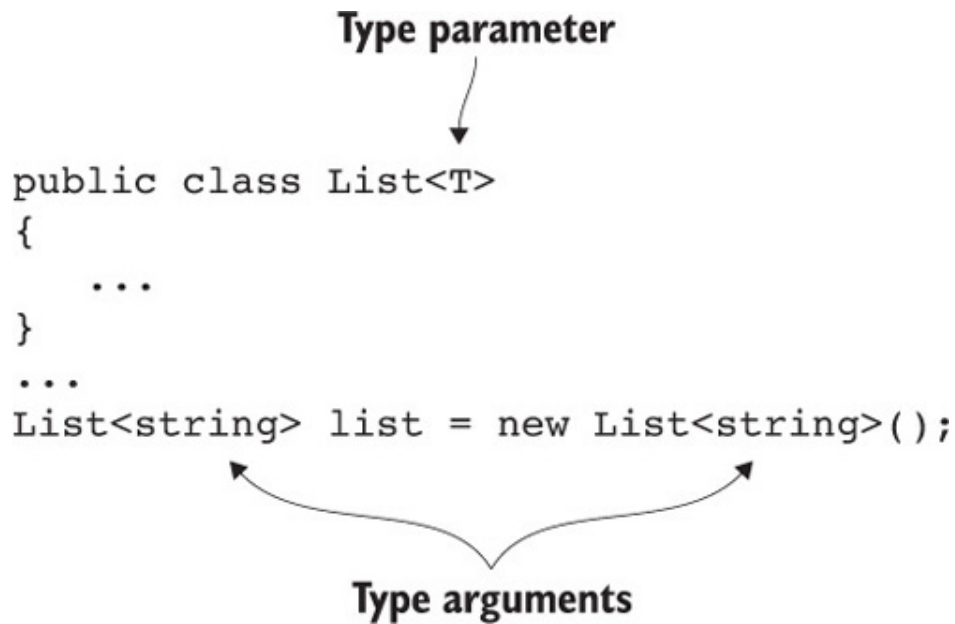




Figure 2.2. Relationship between type parameters and type arguments



```
1 public void Add(T item)
```

### Listing 2.5. Copying elements from one collection to another

```
1 public static List<T> CopyAtMost(
2     List input, int maxElements)
3 {
4     int actualCount = Math.Min(input.Count, maxElements);
5     List<T> ret = new List<T>(actualCount);
6     for (int i = 0; i < actualCount; i++)
7     {
8         ret.Add(input[i]);
9     }
10    return ret;
11 }
12
13 static void Main()
14 {
15     List<int> numbers = new List<int>();
16     numbers.Add(5);
17     numbers.Add(10);
18     numbers.Add(20);
19
20     List<int> firstTwo = CopyAtMost<int>(numbers, 2);
21     Console.WriteLine(firstTwo.Count);
22 }
```

1

2

3

```
1 public class List<T> : IEnumerable<T>
```

---

```
1 public class Dictionary<TKey, TValue>
```

---

```
1 public void Method() {}  
2 public void Method<T>() {}  
3 public void Method<T1, T2>() {}
```

---

1  
2  
3

```
1 public enum IAmConfusing {}  
2 public class IAmConfusing<T> {}  
3 public struct IAmConfusing<T1, T2> {}  
4 public delegate void IAmConfusing<T1, T2, T3> {}  
5 public interface IAmConfusing<T1, T2, T3, T4> {}
```

---



```
1 public void Method<TFirst>() {}  
2 public void Method<TSecond>() {}
```

---

1

```
1 public void Method<T, T>() {}
```

---

1

```
1 public class ValidatingList<TItem>  
2 {  
3     private readonly List<TItem> items = new List<TItem>();  
4 }
```

---

1

```
1 public static List<T> CopyAtMost<T>(List<T> input, int maxElements)
```

---

```
1 List<int> numbers = new List<int>();  
2 ...  
3 List<int> firstTwo = CopyAtMost<int>(numbers, 2);
```

---

```
1 List<int> numbers = new List<int>();  
2 ...  
3 List<int> firstTwo = CopyAtMost(numbers, 2);
```

---

```
1 public static Tuple<T1> Create<T1>(T1 item1)  
2 {  
3     return new Tuple<T1>(item1);  
4 }  
5  
6 public static Tuple<T1, T2> Create<T1, T2>(T1 item1, T2 item2)  
7 {  
8     return new Tuple<T1, T2>(item1, item2);  
9 }
```

---

```
1 new Tuple<int, string, int>(10, "x", 20)
```

---

```
1 Tuple.Create(10, "x", 20)
```

---

```
1 static void PrintItems(List<IFormattable> items)
```

---

```
1 static void PrintItems<T>(List<T> items) where T : IFormattable
```

---

## Listing 2.6. Printing items in the invariant culture by using type constraints

```
1 static void PrintItems<T>(List<T> items) where T : IFormattable
2 {
3     CultureInfo culture = CultureInfo.InvariantCulture;
4     foreach (T item in items)
5     {
6         Console.WriteLine(item.ToString(null, culture));
7     }
8 }
```

---

```
1 public void Sort(List<T> items) where T : IComparable<T>
```

---

```
1 T first = ...;
2 T second = ...;
3 int comparison = first.CompareTo(second);
```

---

```
1 TResult Method<TArg, TResult>(TArg input)
2     where TArg : IComparable<TArg>
3     where TResult : class, new()
```

---

1  
2  
3

```

1 public T LastOrDefault<T>(IEnumerable<T> source)
2 {
3     T ret = default(T);
4     foreach (T item in source)
5     {
6         ret = item;
7     }
8     return ret;
9 }

```

- If you ask for the base type of the generic type definition of a class declared as

```
class StringDictionary<T> : Dictionary<string, T>
```

you'd end up with a type with one "concrete" type argument (, for the type parameter of ) and one type argument that's still a type parameter (, for the type parameter).

### Listing 2.7. Printing the result of the `typeof` operator

```

1 static void PrintType<T>()
2 {
3     Console.WriteLine("typeof(T) = {0}", typeof(T));
4     Console.WriteLine("typeof(List<T>) = {0}", typeof(List<T>));
5 }
6
7 static void Main()
8 {
9     PrintType<string>();
10    PrintType<int>();
11 }

```

```

1 typeof(T) = System.String
2 typeof(List<T>) = System.Collections.Generic.List`1[System.String]
3 typeof(T) = System.Int32
4 typeof(List<T>) = System.Collections.Generic.List`1[System.Int32]

```

## Listing 2.8. Exploring static fields in generic types

```
1 class GenericCounter<T>
2 {
3     private static int value;
4
5     static GenericCounter()
6     {
7         Console.WriteLine("Initializing counter for {0}", typeof(T));
8     }
9
10    public static void Increment()
11    {
12        value++;
13    }
14
15    public static void Display()
16    {
17        Console.WriteLine("Counter for {0}: {1}", typeof(T), value);
18    }
19 }
20
21 class GenericCounterDemo
22 {
23     static void Main()
24     {
25         GenericCounter<string>.Increment();
26         GenericCounter<string>.Increment();
27         GenericCounter<string>.Display();
28         GenericCounter<int>.Display();
29         GenericCounter<int>.Increment();
30         GenericCounter<int>.Display();
31     }
32 }
```

1

2

3

```
1 Initializing counter for System.String
2 Counter for System.String: 2
3 Initializing counter for System.Int32
4 Counter for System.Int32: 0
5 Counter for System.Int32: 1
```

```
1 class Outer<TOuter>
2 {
3     class Inner<TInner>
4     {
5         static int value;
6     }
7 }
```

```
1 public struct Nullable<T> where T : struct 1
2 {
3     private readonly T value;
4     private readonly bool hasValue;
5
6     public Nullable(T value) 2
7     {
8         this.value = value;
9         this.hasValue = true;
10    }
11
12    public bool HasValue { get { return hasValue; } }
13
14    public T Value 3
15    {
16        get
17        {
18            if (!hasValue)
19            {
20                throw new InvalidOperationException();
21            }
22            return value; 4
23        }
24    }
25 }
```

---

```
1 Nullable<int> nullable = new Nullable<int>();
2 Console.WriteLine(nullable.HasValue); 1
```

---

```
1 public void DisplayMaxPrice(Nullable<decimal> maxPriceFilter)
2 {
3     if (maxPriceFilter.HasValue)
4     {
5         Console.WriteLine("Maximum price: {0}", maxPriceFilter.Value);
6     }
7     else
8     {
9         Console.WriteLine("No maximum price set.");
10    }
11 }
```

---

```
1 int x = 5;
2 object o = x;
```

---

Listing 2.9. The effects of boxing nullable value type values

```
1 Nullable<int> noValue = new Nullable<int>();
2 object noValueBoxed = noValue;
3 Console.WriteLine(noValueBoxed == null);
4
5 Nullable<int> someValue = new Nullable<int>(5);
6 object someValueBoxed = someValue;
7 Console.WriteLine(someValueBoxed.GetType());
```

Listing 2.10. Calling GetType on nullable values leads to surprising results

```
1 Nullable<int> noValue = new Nullable<int>();
2 // Console.WriteLine(noValue.GetType());
3
4 Nullable<int> someValue = new Nullable<int>(5);
5 Console.WriteLine(someValue.GetType());
```

```
1 int? x = new int?();
2
3 int? x = null;
```

```
1 if (x != null)
2
3 if (x.HasValue)
```

Table 2.1. Examples of lifted operators applied to nullable integers (view table figure)

Expression	Lifted operator	Result
-nullInt	int? -(int? x)	null
-five	int? -(int? x)	-5
five + nullInt	int? +(int? x, int? y)	null



Expression	Lifted operator	Result
five + five	int? +(int? x, int? y)	10
four & nullInt	int? &(int? x, int? y)	null
four & five	int? &(int? x, int? y)	4
nullInt == nullInt	bool ==(int? x, int? y)	true
five == five	bool ==(int? x, int? y)	true
five == nullInt	bool ==(int? x, int? y)	false
five == four	bool ==(int? x, int? y)	false
four < five	bool <(int? x, int? y)	true
nullInt < five	bool <(int? x, int? y)	false
five < nullInt	bool <(int? x, int? y)	false
nullInt < nullInt	bool <(int? x, int? y)	false
nullInt <= nullInt	bool <=(int? x, int? y)	false

**Table 2.2. Truth table for `Nullable<bool>` operators (view table figure)**

x	y	x & y	x   y	x ^ y	!x
true true	true false	true false	true true	false true	false false
true false	null true	null false	<b>true</b> true	null true	false true
false false	false null	false <b>false</b>	false null	false null	true true
null null null	true false	null <b>false</b>	<b>true</b> null	null null null	null null null
	null	null	null		

```

1 static void PrintValueAsInt32(object o)
2 {
3     int? nullable = o as int?;
4     Console.WriteLine(nullable.HasValue ?
5         nullable.Value.ToString() : "null");
6 }
7 ...
8 PrintValueAsInt32(5);
9 PrintValueAsInt32("some string");

```

1  
2

```

1 int? a = 5;
2 int b = 10;
3 int c = a ?? b;

```

```
1 Console.WriteLine("hello");
```

---

```
1 private void HandleButtonClick(object sender, EventArgs e)
```

---

```
1 EventHandler handler = new EventHandler(HandleButtonClick);
```

---

```
1 EventHandler handler = HandleButtonClick;
```

---

```
1 button.Click += HandleButtonClick;
```

---

```
1 EventHandler handler = delegate
2 {
3     Console.WriteLine("Event raised");
4 };
```

---

```
1 EventHandler handler = delegate(object sender, EventArgs args)
2 {
3     Console.WriteLine("Event raised. sender={0}; args={1}",
4         sender.GetType(), args.GetType());
5 };
```

---

```
1 void AddClickLogger(Control control, string message)
2 {
3     control.Click += delegate
4     {
5         Console.WriteLine("Control clicked: {0}", message);
6     }
7 }
```

---

```
1 public delegate void Printer(string message);
2
3 public void PrintAnything(object obj)
4 {
5     Console.WriteLine(obj);
6 }
```

---

```
1 Printer p1 = new Printer(PrintAnything);
2 Printer p2 = PrintAnything;
```

---

```
1 public delegate void GeneralPrinter(object obj);
```

---

```
1 GeneralPrinter generalPrinter = ...;
2 Printer printer = new Printer(generalPrinter);
```

---

1  
2

```
1 public delegate object ObjectProvider();
2 public delegate string StringProvider();
3
4 StringProvider stringProvider = ...;
5 ObjectProvider objectProvider =
6     new ObjectProvider(stringProvider);
```

---

1  
2  
3

```
1 public delegate void Int32Printer(int x);  
2 public delegate void Int64Printer(long x);  
3  
4 Int64Printer int64Printer = ...;  
5 Int32Printer int32Printer =  
6     new Int32Printer(int64Printer);
```

1

2

3

### Listing 2.11. A simple iterator yielding integers

```
1 static IEnumerable<int> CreateSimpleIterator()  
2 {  
3     yield return 10;  
4     for (int i = 0; i < 3; i++)  
5     {  
6         yield return i;  
7     }  
8     yield return 20;  
9 }
```

```
1 foreach (int value in CreateSimpleIterator())  
2 {  
3     Console.WriteLine(value);  
4 }
```

```
1 10  
2 0  
3 1  
4 2  
5 20
```

### Listing 2.12. The expansion of a `foreach` loop

```
1 IEnumerable<int> enumerable = CreateSimpleIterator();  
2 using (IEnumerator<int> enumerator =  
3     enumerable.GetEnumerator())  
4 {
```

1

2

```

5     while (enumerator.MoveNext())
6     {
7         int value = enumerator.Current;
8         Console.WriteLine(value);
9     }
10 }

```

---

```

1 static IEnumerable<int> CreateSimpleIterator()
2 {
3     yield return 10;
4     for (int i = 0; i < 3; i++)
5     {
6         yield return i;
7     }
8     yield return 20;
9 }

```

---

### Listing 2.13. Iterating over the Fibonacci sequence

```

1 static IEnumerable<int> Fibonacci()
2 {
3     int current = 0;
4     int next = 1;
5     while (true)
6     {
7         yield return current;
8         int oldCurrent = current;
9         current = next;
10        next = next + oldCurrent;
11    }
12 }
13
14 static void Main()
15 {
16     foreach (var value in Fibonacci())
17     {
18         Console.WriteLine(value);
19         if (value > 1000)
20         {
21             break;
22         }
23     }
24 }

```

---

### Listing 2.14. An iterator that logs its progress

```
1 static IEnumerable<string> Iterator()
2 {
3     try
4     {
5         Console.WriteLine("Before first yield");
6         yield return "first";
7         Console.WriteLine("Between yields");
8         yield return "second";
9         Console.WriteLine("After second yield");
10    }
11    finally
12    {
13        Console.WriteLine("In finally block");
14    }
15 }
```

---

### Listing 2.15. A simple `foreach` loop to iterate and log

```
1 static void Main()
2 {
3     foreach (string value in Iterator())
4     {
5         Console.WriteLine("Received value: {0}", value);
6     }
7 }
```

---

```
1 Before first yield
2 Received value: first
3 Between yields
4 Received value: second
5 After second yield
6 In finally block
```

---

### Listing 2.16. Breaking out of a `foreach` loop by using an iterator

```
1 static void Main()
2 {
3     foreach (string value in Iterator())
4     {
5         Console.WriteLine("Received value: {0}", value);
6         if (value != null)
7         {
8             break;
9         }
10    }
11 }
```

- 1 Before first **yield**
  - 2 Received **value**: first
  - 3 In **finally** block
- 

### Listing 2.17. Expansion of [listing 2.16](#) to not use a `foreach` loop

```
1 static void Main()
2 {
3     IEnumerable<string> enumerable = Iterator();
4     using (IEnumerator<string> enumerator = enumerable.GetEnumerator())
5     {
6         while (enumerator.MoveNext())
7         {
8             string value = enumerator.Current;
9             Console.WriteLine("Received value: {0}", value);
10            if (value != null)
11            {
12                break;
13            }
14        }
15    }
16 }
```

---

### Listing 2.18. Reading lines from a file

```
1 static IEnumerable<string> ReadLines(string path)
2 {
3     using (TextReader reader = File.OpenText(path))
4     {
5         string line;
6         while ((line = reader.ReadLine()) != null)
7         {
8             yield return line;
9         }
10    }
11 }
```

---

### Listing 2.19. Sample iterator method to decompile

```

1 public static IEnumerable<int> GenerateIntegers(int count)
2 {
3     try
4     {
5         for (int i = 0; i < count; i++)
6         {
7             Console.WriteLine("Yielding {0}", i);
8             yield return i;
9             int doubled = i * 2;
10            Console.WriteLine("Yielding {0}", doubled);
11            yield return doubled;
12        }
13    }
14    finally
15    {
16        Console.WriteLine("In finally block");
17    }
18 }

```

---

## Listing 2.20. Infrastructure of the generated code for an iterator

```

1 public static IEnumerable<int> GenerateIntegers(
2     int count)
3 {
4     GeneratedClass ret = new GeneratedClass(-2);
5     ret.count = count;
6     return ret;
7 }
8
9 private class GeneratedClass
10     : IEnumerable<int>, IEnumerator<int>
11 {
12     public int count;
13     private int state;
14     private int current;
15     private int initialThreadId;
16     private int i;
17
18     public GeneratedClass(int state)
19     {
20         this.state = state;
21         initialThreadId = Environment.CurrentManagedThreadId;
22     }
23
24     public bool MoveNext() { ... }
25
26     public IEnumerator<int> GetEnumerator() { ... }
27
28     public void Reset()
29     {
30         throw new NotSupportedException();
31     }
32     public void Dispose() { ... }
33
34     public int Current { get { return current; } }
35
36     private void Finally1() { ... }
37
38     IEnumerator IEnumerable.GetEnumerator()
39     {
40         return GetEnumerator();
41     }

```



```
42
43     object IEnumerator.Current { get { return current; } }
44 }
```

---

11

10

11

```
1 for (int i = 0; i < count; i++)
2 {
3     Console.WriteLine("Yielding {0}", i);
4     yield return i;
5     int doubled = i * 2;
6     Console.WriteLine("Yielding {0}", doubled);
7     yield return doubled;
8 }
```

---

### Listing 2.21. Simplified `MoveNext()` method

```
1 public bool MoveNext()
2 {
3     try
4     {
5         switch (state)
6         {
7
8         }
9
10    }
11    fault
12    {
13        Dispose();
14    }
15 }
```

---

1

2

3

4

### Listing 2.22. A simple partial class

```
1 partial class PartialDemo
2 {
3     public static void MethodInPart1()
4     {
5         MethodInPart2();
6     }
7 }
8
9 partial class PartialDemo
10 {
11     private static void MethodInPart2()
12     {
13         Console.WriteLine("In MethodInPart2");
14 }
```

1

2

```
15 }
}
```

## Listing 2.23. Two partial methods—one implemented, one not

```
1 partial class PartialMethodsDemo
2 {
3     public PartialMethodsDemo()
4     {
5         OnConstruction();
6     }
7
8     public override string ToString()
9     {
10         string ret = "Original return value";
11         CustomizeToString(ref ret);
12         return ret;
13     }
14
15     partial void OnConstruction();
16     partial void CustomizeToString(ref string text);
17 }
18
19 partial class PartialMethodsDemo
20 {
21     partial void CustomizeToString(ref string text)
22     {
23         text += " - customized!";
24     }
25 }
```

## Listing 2.24. Demonstration of static classes

```
1 static class StaticClassDemo
2 {
3     public static void StaticMethod() { }
4
5     public void InstanceMethod() { }
6
7     public class RegularNestedClass
8     {
9         public void InstanceMethod() { }
10    }
11 }
12 ...
13 StaticClassDemo.StaticMethod();
14
15 StaticClassDemo localVariable = null;
16 List<StaticClassDemo> list =
17     new List<StaticClassDemo>();
```

```
1 private string text;
2
3 public string Text
4 {
5     get { return text; }
6     private set { text = value; }
7 }
```

---

## Listing 2.25. Namespace aliases in C# 1

```
1 using System;
2 using WinForms = System.Windows.Forms;
3 using WebForms = System.Web.UI.WebControls;
4
5 class Test
6 {
7     static void Main()
8     {
9         Console.WriteLine(typeof(WinForms.Button));
10        Console.WriteLine(typeof(WebForms.Button));
11    }
12 }
```

---

1

2

```
1 static void Main()
2 {
3     Console.WriteLine(typeof(WinForms::Button));
4     Console.WriteLine(typeof(WebForms::Button));
5 }
```

---

```
1 extern alias FirstAlias;
2 extern alias SecondAlias;
```

---

```
1 extern alias JsonNet;
2 extern alias JsonNetAlternative;
3
4 using JsonNet::Newtonsoft.Json.Linq;
5 using AltJsonObject = JsonNetAlternative::Newtonsoft.Json.Linq.JsonObject;
```

```
6 ...  
7 JObject obj = new JObject();  
8 AltJObject alt = new AltJObject();
```

---

1  
2

```
1 #pragma warning disable CS0219  
2 int variable = CallSomeMethod();  
3 #pragma warning restore CS0219
```

---

## Listing 2.26. Using fixed-size buffers for a versioned chunk of binary data

```
1 unsafe struct VersionedData  
2 {  
3     public int Major;  
4     public int Minor;  
5     public fixed byte Data[16];  
6 }  
7  
8 unsafe static void Main()  
9 {  
10     VersionedData versioned = new VersionedData();  
11     versioned.Major = 2;  
12     versioned.Minor = 1;  
13     versioned.Data[10] = 20;  
14 }
```

---

```
1 [assembly: InternalsVisibleTo("MyProduct.Test")]
```

---

```
1 [assembly: InternalsVisibleTo("NodaTime.Test, PublicKey=0024...4669")]
```

---

# CHAPTER 3

```
1 private string name;  
2 public string Name  
3 {  
4     get { return name; }  
5     set { name = value; }  
6 }
```

---

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

---

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

---

```
1 var language = "C#";
```

---

```
1 string language = "C#";
```

---

```
1 var x;  
2 x = 10;  
3  
4 var y = null;
```

---

1

2

```
1 Dictionary<string, List<decimal>> mapping =  
2     new Dictionary<string, List<decimal>>();
```

---

```
1 var mapping = new Dictionary<string, List<decimal>>();
```

---

```
1 int[] array = new int[10];
```

---

```
1 int[] array1 = { 1, 2, 3, 4, 5};  
2 int[] array2 = new int[] { 1, 2, 3, 4, 5};
```

---

```
1 int[] array;  
2 array = { 1, 2, 3, 4, 5 };
```

---

1

```
1 array = new int[] { 1, 2, 3, 4, 5 };
```

---

```
1 array = new[] { 1, 2, 3, 4, 5 };
```

---

```
1 var array = new[,] { { 1, 2, 3 }, { 4, 5, 6 } };
```

**Table 3.1. Examples of type inference for implicitly typed arrays (view table figure)**

Expression	Result	Notes
<code>new[] { 10, 20 }</code>	<code>int[]</code>	All elements are of type <code>int</code> .
<code>new[] { null, null }</code>	Error	No elements have types.
<code>new[] { "xyz", null }</code>	<code>string[]</code>	Only candidate type is <code>string</code> , and the <code>null</code> literal can be converted to <code>string</code> .
<code>new[] { "abc", new object() }</code>	<code>object[]</code>	Candidate types of <code>string</code> and <code>object</code> ; implicit conversion from <code>string</code> to <code>object</code> but not vice versa.
<code>new[] { 10, new DateTime() }</code>	Error	Candidate types of <code>int</code> and <code>DateTime</code> but no conversion from either to the other.
<code>new[] { 10, null }</code>	Error	Only candidate type is <code>int</code> , but there's no conversion from <code>null</code> to <code>int</code> .

**Listing 3.1. Modeling an order in an e-commerce system**

```
1 public class Order
2 {
3     private readonly List<OrderItem> items = new List<OrderItem>();
4
5     public string OrderId { get; set; }
6     public Customer Customer { get; set; }
7     public List<OrderItem> Items { get { return items; } }
8 }
9
10 public class Customer
11 {
12     public string Name { get; set; }
13     public string Address { get; set; }
14 }
15
16 public class OrderItem
17 {
18     public string ItemId { get; set; }
19     public int Quantity { get; set; }
20 }
```

### Listing 3.2. Creating and populating an order without object and collection initializers

```
1 var customer = new Customer();  
2 customer.Name = "Jon";  
3 customer.Address = "UK";  
4  
5 var item1 = new OrderItem();  
6 item1.ItemId = "abcd123";  
7 item1.Quantity = 1;  
8  
9 var item2 = new OrderItem();  
10 item2.ItemId = "fghi456";  
11 item2.Quantity = 2;  
12  
13 var order = new Order();  
14 order.OrderId = "xyz";  
15 order.Customer = customer;  
16 order.Items.Add(item1);  
17 order.Items.Add(item2);
```

1

2

3

4

### Listing 3.3. Creating and populating an order with object and collection initializers

```
1 var order = new Order  
2 {  
3     OrderId = "xyz",  
4     Customer = new Customer { Name = "Jon", Address = "UK" },  
5     Items =  
6     {  
7         new OrderItem { ItemId = "abcd123", Quantity = 1 },  
8         new OrderItem { ItemId = "fghi456", Quantity = 2 }  
9     }  
10 };
```

```
1 Order order = new Order() { OrderId = "xyz" };  
2 Order order = new Order { OrderId = "xyz" };
```

```
1 Order order = new Order;
```

1



### Listing 3.4. Modifying default headers on a new `HttpClient` with a nested object initializer

```
1 HttpClient client = new HttpClient
2 {
3     DefaultRequestHeaders =
4     {
5         From = "user@example.com",
6         Date = DateTimeOffset.UtcNow
7     }
8 };
```

---

```
1 HttpClient client = new HttpClient();
2 var headers = client.DefaultRequestHeaders;
3 headers.From = "user@example.com";
4 headers.Date = DateTimeOffset.UtcNow;
```

---

```
1 var order = new Order
2 {
3     OrderId = "xyz",
4     Customer = new Customer { Name = "Jon", Address = "UK" },
5     Items =
6     {
7         new OrderItem { ItemId = "abcd123", Quantity = 1 },
8         new OrderItem { ItemId = "fghi456", Quantity = 2 }
9     }
10 };
```

---

```
1 var beatles = new List<string> { "John", "Paul", "Ringo", "George" };
```

---

```
1 var beatles = new List<string>();
2 beatles.Add("John");
3 beatles.Add("Paul");
4 beatles.Add("Ringo");
5 beatles.Add("George");
```

---

```
1 var releaseYears = new Dictionary<string, int>
2 {
3     { "Please please me", 1963 },
4     { "Revolver", 1966 },
5     { "Sgt. Pepper's Lonely Hearts Club Band", 1967 },
6     { "Abbey Road", 1970 }
7 };
```

---

```
1 var releaseYears = new Dictionary<string, int>();
2 releaseYears.Add("Please please me", 1963);
3 releaseYears.Add("Revolver", 1966);
4 releaseYears.Add("Sgt. Pepper's Lonely Hearts Club Band", 1967);
5 releaseYears.Add("Abbey Road", 1970);
```

---

```
1 DateTime invalid = new DateTime(2020, 1, 1) { TimeSpan.FromDays(10) };
```

---

1

### Listing 3.5. Anonymous type with `Name` and `Score` properties

```
1 var player = new
2 {
3     Name = "Rajesh",
4     Score = 3500
5 };
6
7 Console.WriteLine("Player name: {0}", player.Name);
8 Console.WriteLine("Player score: {0}", player.Score);
```

---

1

2

```
1 var flattenedItem = new
2 {
3     order.OrderId,
4     CustomerName = customer.Name,
5     customer.Address,
6     item.ItemId,
7     item.Quantity
8 };
```

```
1 var flattenedItem = new
2 {
3     OrderId = order.OrderId,
4     CustomerName = customer.Name,
5     Address = customer.Address,
6     ItemId = item.ItemId,
7     Quantity = item.Quantity
8 };
```

---

```
1 SomeProperty = variable.SomeProperty
```

---

```
1 variable.SomeProperty
```

---

```
1 var player = new { Name = "Pam", Score = 4000 };
2 player = new { Name = "James", Score = 5000 };
```

---

```
1 var players = new[]
2 {
3     new { Name = "Priti", Score = 6000 },
4     new { Name = "Chris", Score = 7000 },
5     new { Name = "Amanda", Score = 8000 },
6 };
```

---

```
1 var players = new[]
2 {
```

```
3     new { Name = "Priti", Score = 6000 },
4     new { Score = 7000, Name = "Chris" },
5     new { Name = "Amanda", Score = 8000 },
6 };
```

---

```
1 Action<string> action = delegate(string message)
2 {
3     Console.WriteLine("In delegate: {0}", message);
4 };
5 action("Message");
```

---

1

2

```
1 parameter-list => body
```

---

```
1 Action<string> action = (string message) =>
2 {
3     Console.WriteLine("In delegate: {0}", message);
4 };
5 action("Message");
```

---

```
1 Action<string> action =
2     (string message) => Console.WriteLine("In delegate: {0}", message);
```

---

```
1 Action<string> action =
2     (message) => Console.WriteLine("In delegate: {0}", message);
```

---

```
1 Action<string> action =  
2     message => Console.WriteLine("In delegate: {0}", message);
```

---

```
1 Func<int, int, int> multiply =  
2     (int x, int y) => { return x * y; }; 1  
3  
4 Func<int, int, int> multiply = (int x, int y) => x * y; 2  
5  
6 Func<int, int, int> multiply = (x, y) => x * y;  
7 (Two parameters, so you can't remove parentheses) 3
```

---

```
1 Func<string, int> squareLength = (string text) => 1  
2 {  
3     int length = text.Length;  
4     return length * length;  
5 };  
6  
7 Func<string, int> squareLength = (text) => 2  
8 {  
9     int length = text.Length;  
10    return length * length;  
11 };  
12  
13 Func<string, int> squareLength = text => 3  
14 {  
15     int length = text.Length;  
16     return length * length;  
17 };  
18 (Can't do anything else immediately; body has two statements)
```

---

```
1 Func<string, int> squareLength = text => text.Length * text.Length;
```

---

### Listing 3.6. Capturing variables in a lambda expression

```
1 class CapturedVariablesDemo  
2 {  
3     private string instanceField = "instance field";  
4  
5     public Action<string> createAction(string methodParameter)
```

```

6      {
7          string methodLocal = "method local";
8          string uncaptured = "uncaptured local";
9
10         Action<string> action = lambdaParameter =>
11         {
12             string lambdaLocal = "lambda local";
13             Console.WriteLine("Instance field: {0}", instanceField);
14             Console.WriteLine("Method parameter: {0}", methodParameter);
15             Console.WriteLine("Method local: {0}", methodLocal);
16             Console.WriteLine("Lambda parameter: {0}", lambdaParameter);
17             Console.WriteLine("Lambda local: {0}", lambdaLocal);
18         };
19         methodLocal = "modified method local";
20         return action;
21     }
22 }
23
24 In other code
25 var demo = new CapturedVariablesDemo();
26 Action<string> action = demo.CreateAction("method argument");
27 action("lambda argument");

```

---

```

1 public Action<string> CreateAction(string methodParameter)
2 {
3     string methodLocal = "method local";
4     string uncaptured = "uncaptured local";
5
6     Action<string> action = lambdaParameter =>
7     {
8         string lambdaLocal = "lambda local";
9         Console.WriteLine("Instance field: {0}", instanceField);
10        Console.WriteLine("Method parameter: {0}", methodParameter);
11        Console.WriteLine("Method local: {0}", methodLocal);
12        Console.WriteLine("Lambda parameter: {0}", lambdaParameter);
13        Console.WriteLine("Lambda local: {0}", lambdaLocal);
14    };
15    methodLocal = "modified method local";
16    return action;
17 }

```

---

### Listing 3.7. Translation of a lambda expression with captured variables

```

1 private class LambdaContext 1
2 {
3     public CapturedVariablesDemoImpl originalThis;
4     public string methodParameter; 2
5     public string methodLocal;
6
7     public void Method(string lambdaParameter) 3
8     {
9         string lambdaLocal = "lambda local";
10        Console.WriteLine("Instance field: {0}",
11            originalThis.instanceField);
12        Console.WriteLine("Method parameter: {0}", methodParameter);

```

```

13     Console.WriteLine("Method local: {0}", methodLocal);
14     Console.WriteLine("Lambda parameter: {0}", lambdaParameter);
15     Console.WriteLine("Lambda local: {0}", lambdaLocal);
16 }
17 }
18
19 public Action<string> CreateAction(string methodParameter)
20 {
21     LambdaContext context = new LambdaContext();
22     context.originalThis = this;
23     context.methodParameter = methodParameter;
24     context.methodLocal = "method local";
25     string uncaptured = "uncaptured local";
26
27
28     Action<string> action = context.Method;
29     context.methodLocal = "modified method local";
30     return action;
31 }

```

4

### Listing 3.8. Instantiating a local variable multiple times

```

1 static List<Action> CreateActions()
2 {
3     List<Action> actions = new List<Action>();
4     for (int i = 0; i < 5; i++)
5     {
6         string text = string.Format("message {0}", i);
7         actions.Add(() => Console.WriteLine(text));
8     }
9     return actions;
10 }
11
12 In other code
13 List actions = CreateActions();
14 foreach (Action action in actions)
15 {
16     action();
17 }

```

1

2

### Listing 3.9. Creating multiple context instances, one for each instantiation

```

1 private class LambdaContext
2 {
3     public string text;
4
5     public void Method()
6     {
7         Console.WriteLine(text);
8     }
9 }
10
11 static List<Action> CreateActions()
12 {
13     List<Action> actions = new List<Action>();

```

```

14     for (int i = 0; i < 5; i++)
15     {
16         LambdaContext context = new LambdaContext();
17         context.text = string.Format("message {0}", i);
18         actions.Add(context.Method);
19     }
20     return actions;
21 }

```

### Listing 3.10. Capturing variables from multiple scopes

```

1  static List<Action> CreateCountingActions()
2  {
3      List<Action> actions = new List<Action>();
4      int outerCounter = 0;
5      for (int i = 0; i < 2; i++)
6      {
7          int innerCounter = 0;
8          Action action = () =>
9          {
10             Console.WriteLine(
11                 "Outer: {0}; Inner: {1}",
12                 outerCounter, innerCounter);
13             outerCounter++;
14             innerCounter++;
15         };
16         actions.Add(action);
17     }
18     return actions;
19 }
20
21 In other code
22 List<Action> actions = CreateCountingActions();
23 actions[0]();
24 actions[0]();
25 actions[1]();
26 actions[1]();

```

```

1 Outer: 0; Inner: 0
2 Outer: 1; Inner: 1
3 Outer: 2; Inner: 0
4 Outer: 3; Inner: 1

```

### Listing 3.11. Capturing variables from multiple scopes leads to multiple classes

```

1 private class OuterContext
2 {

```



```

3     public int outerCounter;
4 }
5
6 private class InnerContext
7 {
8     public OuterContext outerContext;
9     public int innerCounter;
10
11     public void Method()
12     {
13         Console.WriteLine(
14             "Outer: {0}; Inner: {1}",
15             outerContext.outerCounter, innerCounter);
16         outerContext.outerCounter++;
17         innerCounter++;
18     }
19 }
20
21 static List<Action> CreateCountingActions()
22 {
23     List<Action> actions = new List<Action>();
24     OuterContext outerContext = new OuterContext();
25     outerContext.outerCounter = 0;
26     for (int i = 0; i < 2; i++)
27     {
28         InnerContext innerContext = new InnerContext();
29         innerContext.outerContext = outerContext;
30         innerContext.innerCounter = 0;
31         Action action = innerContext.Method();
32         actions.Add(action);
33     }
34     return actions;
35 }

```

### Listing 3.12. A simple expression tree to add two integers

```

1 Expression<Func<int, int, int>> adder = (x, y) => x + y;
2 Console.WriteLine(adder);

```

```

1 (x, y) => x + y

```

### Listing 3.13. Handwritten code to create an expression tree to add two integers

```

1 ParameterExpression xParameter = Expression.Parameter(typeof(int), "x");
2 ParameterExpression yParameter = Expression.Parameter(typeof(int), "y");
3 Expression body = Expression.Add(xParameter, yParameter);
4 ParameterExpression[] parameters = new[] { xParameter, yParameter };

```

```
5
6 Expression<Func<int, int, int>> adder =
7     Expression.Lambda<Func<int, int, int>>(body, parameters);
8 Console.WriteLine(adder);
```

---

```
1 Expression<Func<int, int, int>> adder = (x, y) => { return x + y; };
```

---

### Listing 3.14. Compiling an expression tree to a delegate and invoking the result

```
1 Expression<Func<int, int, int>> adder = (x, y) => x + y;
2 Func<int, int, int> executableAdder = adder.Compile();
3 Console.WriteLine(executableAdder(2, 3));
```

1

2

```
1 ExampleClass.Method(x, y);
```

---

```
1 x.Method(y);
```

---

### Listing 3.15. `ToInstant` extension method targeting `DateTimeOffset` from Noda Time

```
1 using System;
2
3 namespace NodaTime.Extensions
4 {
5     public static class DateTimeOffsetExtensions
6     {
7         public static Instant ToInstant(this DateTimeOffset dateTimeOffset)
8         {
9             return Instant.FromDateTimeOffset(dateTimeOffset);
10        }
11    }
12 }
```

### Listing 3.16. Invoking the `ToInstant()` extension method outside Noda Time

```
1 using NodaTime.Extensions;
2 using System;
3
4 namespace CSharpInDepth.Chapter03
5 {
6     class ExtensionMethodInvocation
7     {
8         static void Main()
9         {
10             var currentInstant =
11                 DateTimeOffset.UtcNow.ToInstant();
12             Console.WriteLine(currentInstant);
13         }
14     }
15 }
```

1

2

## EXTENSION METHODS CAN BE CALLED ON NULL VALUES

Extension methods differ from instance methods in terms of their null handling. Let's look back at our initial example:

```
1 x.Method(y);
```

If `Method` were an instance method and `x` were a null reference, that would throw a `NullReferenceException`. Instead, if `Method` is an extension method, it'll be called with `x` as the first argument even if `x` is null. Sometimes the method will specify that the first argument must not be null, in which case it should validate it and throw an `ArgumentNullException`. In other cases, the extension method may have been explicitly designed to handle a null first argument gracefully.

### Listing 3.17. A simple query on strings

```
1 string[] words = { "keys", "coat", "laptop", "bottle" }; 1
2 IEnumerable<string> query = words
3     .Where(word => word.Length > 4)
4     .OrderBy(word => word) 2
5     .Select(word => word.ToUpper());
6
7 foreach (string word in query)
8 {
9     Console.WriteLine(word);
10 }
```

---

### Listing 3.18. A simple query without using extension methods

```
1 string[] words = { "keys", "coat", "laptop", "bottle" };
2 IEnumerable<string> query =
3     Enumerable.Select(
4         Enumerable.OrderBy(
5             Enumerable.Where(words, word => word.Length > 4),
6             word => word),
7         word => word.ToUpper());
```

---

### Listing 3.19. A simple query in multiple statements

```
1 string[] words = { "keys", "coat", "laptop", "bottle" };
2 var tmp1 = Enumerable.Where(words, word => word.Length > 4);
3 var tmp2 = Enumerable.OrderBy(tmp1, word => word);
4 var query = Enumerable.Select(tmp2, word => word.ToUpper());
```

---

```
1 IEnumerable<string> query = words
2     .Where(word => word.Length > 4)
3     .OrderBy(word => word)
4     .Select(word => word.ToUpper());
```

---

### Listing 3.20. Introductory query expression with filtering, ordering, and projection

```
1 IEnumerable<string> query = from word in words
2                             where word.Length > 4
3                             orderby word
4                             select word.ToUpper();
```

```
1 from word in words
2 where word.Length > 4
3 orderby word
4 select word.ToUpper()
```

---

1

2

### Listing 3.21. A `let` clause introducing a new range variable

```
1 from word in words
2 let length = word.Length
3 where length > 4
4 orderby length
5 select string.Format("{0}: {1}", length, word.ToUpper());
```

---

### Listing 3.22. Query translation using a transparent identifier

```
1 words.Select(word => new { word, length = word.Length })
2     .Where(tmp => tmp.length > 4)
3     .OrderBy(tmp => tmp.length)
4     .Select(tmp =>
5         string.Format("{0}: {1}", tmp.length, tmp.word.ToUpper()));
```

---

```
1 from word in words
2 where word.Length > 4
3 select word
```

---

```
1 words.Where(word => word.Length > 4)
```

---

```
1 var products = from product in dbContext.Products
2                 where product.StockCount > 0
3                 orderby product.Price descending
4                 select new { product.Name, product.Price };

```

---

# CHAPTER 4

## Listing 4.1. Taking a substring by using dynamic typing

```
1 dynamic text = "hello world"; 1
2 string world = text.Substring(6); 2
3 Console.WriteLine(world);
4
5 string broken = text.SUBSTR(6); 3
6 Console.WriteLine(broken);
```

---

```
1 dynamic text = "hello world";
```

---

```
1 dynamic text = "hello world";
2 string world = text.Substring(6);
```

---

## Listing 4.2. Addition of dynamic values

```
1 static void Add(dynamic d) 1
2 {
3     Console.WriteLine(d + d);
4 }
5
6 Add("text");
7 Add(10);
8 Add(TimeSpan.FromMinutes(45)); 2
```

---

```
1 texttext
2 20
3 01:30:00
```

### Listing 4.3. Dynamic method overload resolution

```
1 static void SampleMethod(int value)
2 {
3     Console.WriteLine("Method with int parameter");
4 }
5
6 static void SampleMethod(decimal value)
7 {
8     Console.WriteLine("Method with decimal parameter");
9 }
10
11 static void SampleMethod(object value)
12 {
13     Console.WriteLine("Method with object parameter");
14 }
15 static void CallMethod(dynamic d)
16 {
17     SampleMethod(d);
18 }
19
20 CallMethod(10);
21 CallMethod(10.5m);
22 CallMethod(10L);
23 CallMethod("text");
```

1

2

```
1 Method with int parameter
2 Method with decimal parameter
3 Method with decimal parameter
4 Method with object parameter
```

### Listing 4.4. Examples of compile-time failures involving dynamic values

```
1 dynamic d = new object();
2 int invalid1 = "text".Substring(0, 1, 2, d);
3 bool invalid2 = string.Equals<int>("foo", d);
4 string invalid3 = new string(d, "broken");
5 char invalid4 = "text"[d, d];
```

1

2

3

4



```
1 dynamic database = new Database(connectionString);
2 var books = database.Books.SearchByAuthor("Holly Webb");
3 foreach (var book in books)
4 {
5     Console.WriteLine(book.Title);
6 }
```

#### Listing 4.5. Storing and retrieving items in an `ExpandoObject`

```
1 dynamic expando = new ExpandoObject();
2 expando.SomeData = "Some data";
3 Action<string> action =
4     input => Console.WriteLine("The input was '{0}'", input);
5 expando.FakeMethod = action;
6
7 Console.WriteLine(expando.SomeData);
8 expando.FakeMethod("hello");
9
10 IDictionary<string, object> dictionary = expando;
11 Console.WriteLine("Keys: {0}",
12     string.Join(", ", dictionary.Keys));
13
14 dictionary["OtherData"] = "other";
15 Console.WriteLine(expando.OtherData);
```

1

2

3

4

5

#### Listing 4.6. Using JSON data dynamically

```
1 string json = @"
2     {
3         'name': 'Jon Skeet',
4         'address': {
5             'town': 'Reading',
6             'country': 'UK'
7         }
8     }".Replace('\', '"');
9
10 JObject obj1 = JObject.Parse(json);
11
12 Console.WriteLine(obj1["address"]["town"]);
13
14 dynamic obj2 = obj1;
15 Console.WriteLine(obj2.address.town);
```

1

2

3

4

```
1 public interface IDynamicMetaObjectProvider
2 {
3
```

```
4     DynamicMetaObject GetMetaObject(Expression parameter);  
    }
```

```
1 public virtual DynamicMetaObject BindInvokeMember  
2     (InvokeMemberBinder binder, DynamicMetaObject[] args);
```

### Listing 4.7. Example of intended use of dynamic behavior

```
1 dynamic example = new SimpleDynamicExample();  
2 example.CallSomeMethod("x", 10);  
3 Console.WriteLine(example.SomeProperty);
```

```
1 Invoked: CallSomeMethod(x, 10)  
2 Fetched: SomeProperty
```

### Listing 4.8. Implementing SimpleDynamicExample

```
1 class SimpleDynamicExample : DynamicObject  
2 {  
3     public override bool TryInvokeMember(  
4         InvokeMemberBinder binder,  
5         object[] args,  
6         out object result)  
7     {  
8         Console.WriteLine("Invoked: {0}({1})",  
9             binder.Name, string.Join(", ", args));  
10        result = null;  
11        return true;  
12    }  
13  
14    public override bool TryGetMember(  
15        GetMemberBinder binder,  
16        out object result)  
17    {  
18        result = "Fetch
```

1

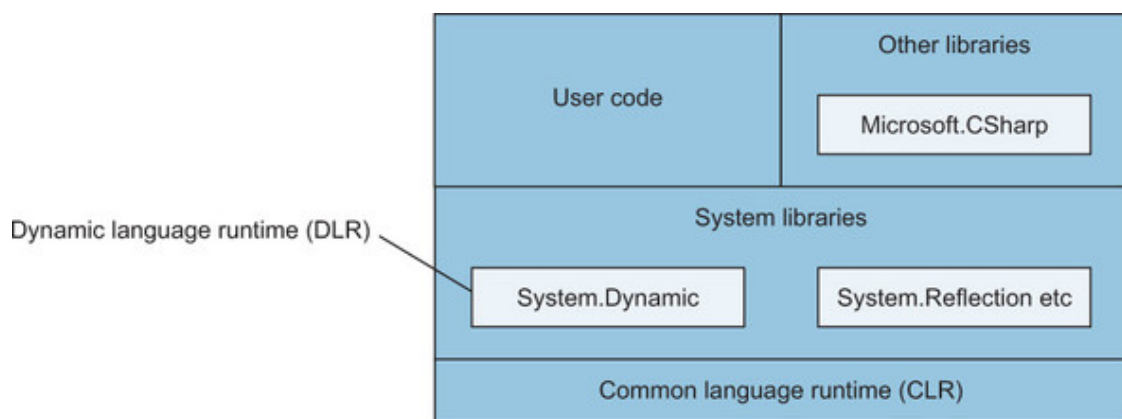
2

```

1 dynamic database = new Database(connectionString);
2 var books = database.Books.SearchByAuthor("Holly Webb");
3 foreach (var book in books)
4 {
5     Console.WriteLine(book.Title);
6 }

```

**Figure 4.1. Graphical representation of components involved in dynamic typing**



```

1 dynamic text = "hello world";
2 string world = text.Substring(6);

```

**Listing 4.9. The result of decompiling two simple dynamic operations**

```

1 using Microsoft.CSharp.RuntimeBinder;
2 using System;
3 using System.Runtime.CompilerServices;
4
5 class DynamicTypingDecompiled
6 {
7     private static class CallSites
8     {
9         public static CallSite<Func<CallSite, object, int, object>>
10             method;
11         public static CallSite<Func<CallSite, object, string>>
12             conversion;
13     }

```

```

14
15 static void Main()
16 {
17     object text = "hello world";
18     if (CallSites.method == null)
19     {
20         CSharpArgumentInfo[] argumentInfo = new[]
21         {
22             CSharpArgumentInfo.Create(
23                 CSharpArgumentInfoFlags.None, null),
24             CSharpArgumentInfo.Create(
25                 CSharpArgumentInfoFlags.Constant |
26                 CSharpArgumentInfoFlags.UseCompileTimeType,
27                 null)
28         };
29         CallSiteBinder binder =
30             Binder.InvokeMember(CSharpBinderFlags.None, "Substring",
31                 null, typeof(DynamicTypingDecompiled), argumentInfo);
32         CallSites.method =
33             CallSite<Func<CallSite, object, int, object>>.Create(binder);
34     }
35
36     if (CallSites.conversion == null)
37     {
38         CallSiteBinder binder =
39             Binder.Convert(CSharpBinderFlags.None, typeof(string),
40                 typeof(DynamicTypingDecompiled));
41         CallSites.conversion =
42             CallSite<Func<CallSite, object, string>>.Create(binder);
43     }
44     object result = CallSites.method.Target(
45         CallSites.method, text, 6);
46
47     string str =
48         CallSites.conversion.Target(CallSites.conversion, result);
49 }
50 }

```

2

3

4

5

```

1 class DynamicSequence : IEnumerable<dynamic>
2 class DynamicListSequence : IEnumerable<List<dynamic>>
3 class DynamicConstraint1<T> : IEnumerable<T> where T : dynamic
4 class DynamicConstraint2<T> : IEnumerable<T> where T : List<dynamic>

```

```

1 class DynamicList : List<dynamic>
2 class ListOfDynamicSequences : List<IEnumerable<dynamic>>
3 IEnumerable<dynamic> x = new List<dynamic> { 1, 0.5 }.Select(x => x * 2);

```

### Listing 4.10. A LINQ query over a list of dynamic values

```
1 List<dynamic> source = new List<dynamic>
2 {
3     5,
4     2.75,
5     TimeSpan.FromSeconds(45)
6 };
7 IEnumerable<dynamic> query = source.Select(x => x * 2);
8 foreach (dynamic value in query)
9 {
10     Console.WriteLine(value);
11 }
```

---

### Listing 4.11. Attempting to call an extension method on a dynamic target

```
1 dynamic source = new List<dynamic>
2 {
3     5,
4     2.75,
5     TimeSpan.FromSeconds(45)
6 };
7 bool result = source.Any();
```

---

```
1 bool result = Enumerable.Any(source);
```

---

```
1 dynamic function = x => x * 2;
2 Console.WriteLine(function(0.75));
```

---

```
1 dynamic function = (Func<dynamic, dynamic>) (x => x * 2);
2 Console.WriteLine(function(0.75));
```

---

```

1 dynamic source = new List<dynamic>
2 {
3     5,
4     2.75,
5     TimeSpan.FromSeconds(45)
6 };
7 dynamic result = source.Select(x => x * 2);

```

#### Listing 4.12. Attempting to use a dynamic element type in an IQueryable<T>

```

1 List<dynamic> source = new List<dynamic>
2 {
3     5,
4     2.75,
5     TimeSpan.FromSeconds(45)
6 };
7 IEnumerable<dynamic> query = source
8     .AsQueryable()
9     .Select(x => x * 2);

```

1

#### Listing 4.13. Dynamic access to a property of an anonymous type

```

1 static void PrintName(dynamic obj)
2 {
3     Console.WriteLine(obj.Name);
4 }
5
6 static void Main()
7 {
8     var x = new { Name = "Abc" };
9     var y = new { Name = "Def", Score = 10 };
10    PrintName(x);
11    PrintName(y);
12 }

```

#### Listing 4.14. Example of explicit interface implementation

```

1 List<int> list1 = new List<int>();
2 Console.WriteLine(list1.IsFixedSize);
3
4 IList list2 = list1;
5 Console.WriteLine(list2.IsFixedSize);
6
7 dynamic list3 = list1;
8 Console.WriteLine(list3.IsFixedSize);

```

1

2

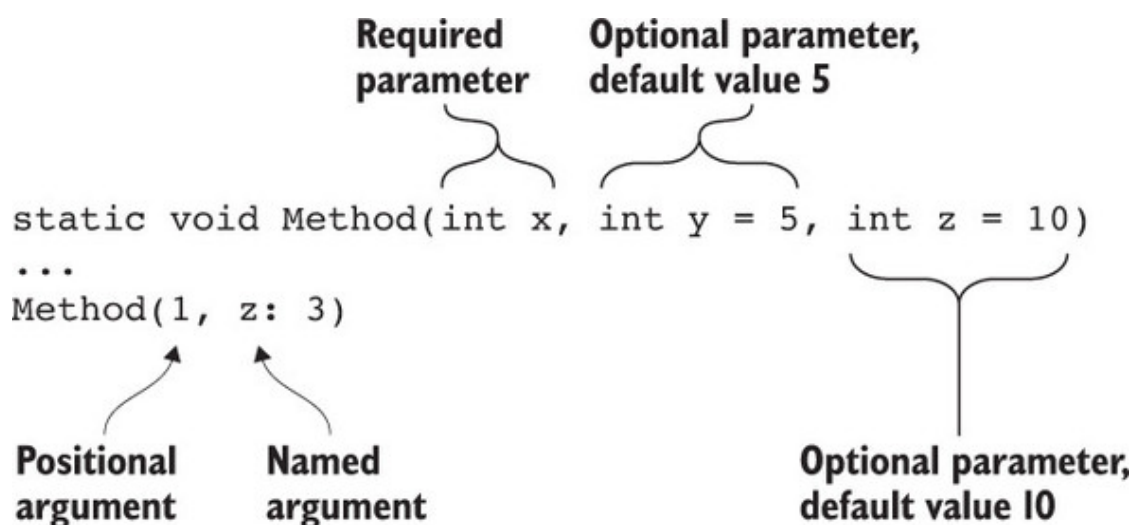
3

```
1 dynamic value = ...;
2 value.SomeProperty.SomeMethod();
```

#### Listing 4.15. Calling a method with optional parameters

```
1 static void Method(int x, int y = 5, int z = 10) 1
2 {
3     Console.WriteLine("x={0}; y={1}; z={2}", x, y, z); 2
4 }
5
6 ...
7
8 Method(1, 2, 3);
9 Method(x: 1, y: 2, z: 3);
10 Method(z: 3, y: 2, x: 1); 3
11 Method(1, 2);
12 Method(1, y: 2);
13 Method(1, z: 3); 4
14 Method(1); 5
15 Method(x: 1); 6
```

Figure 4.2. Syntax of optional/required parameters and named/positional arguments



```
1 static void Method(int x, int y = 5, int z = 10)
```

**Table 4.1. Examples of valid method calls for named arguments and optional parameters** [\(view table figure\)](#)

Call	Resulting arguments	Notes
Method(1, 2, 3)	x=1; y=2; z=3	All positional arguments. Regular call from before C# 4.
Method(1)	x=1; y=5; z=10	Compiler supplies values for y and z, as there are no corresponding arguments.
Method()	n/a	Invalid: no argument corresponds to x.
Method(y: 2)	n/a	Invalid: no argument corresponds to x. Compiler supplies value for y as there's no corresponding argument. It was skipped by using a named argument for z.
Method(1, z: 3)	x=1; y=5; z=3	Invalid: two arguments correspond to x.
Method(1, x: 2, z: 3)	n/a	Invalid: two arguments correspond to y.
Method(1, y: 2, y: 2)	n/a	Named arguments can be in any order,
Method(z: 3, y: 2, x: 1)	x=1; y=2; z=3	

```
1 int tmp1 = 0;
2 Method(x: tmp1++, y: tmp1++, z: tmp1++);
3
4 int tmp2 = 0;
5 Method(z: tmp2++, y: tmp2++, x: tmp2++);
```

1

2

```
1 int tmp3 = 0;
2 int argX = tmp3++;
3 int argY = tmp3++;
4 int argZ = tmp3++;
5 Method(x: argX, y: argY, z: argZ);
```



```
1 public static Method(int x, int y = 5, int z = 10)
```

---

```
1 public static Method(int a, int b = 5, int c = 10)
```

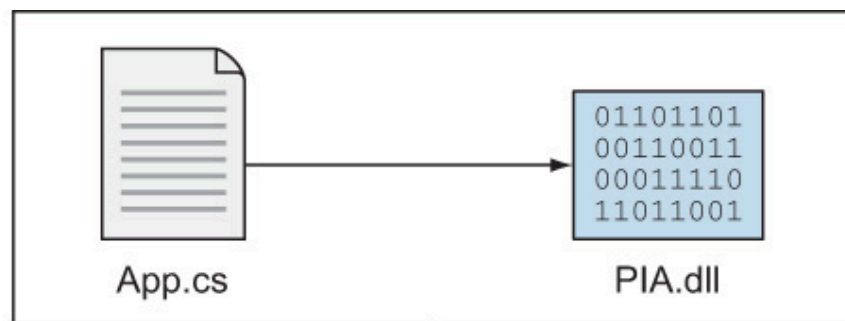
---

```
1 MessageBox.Show(text: "This is text", caption: "This is the title");
```

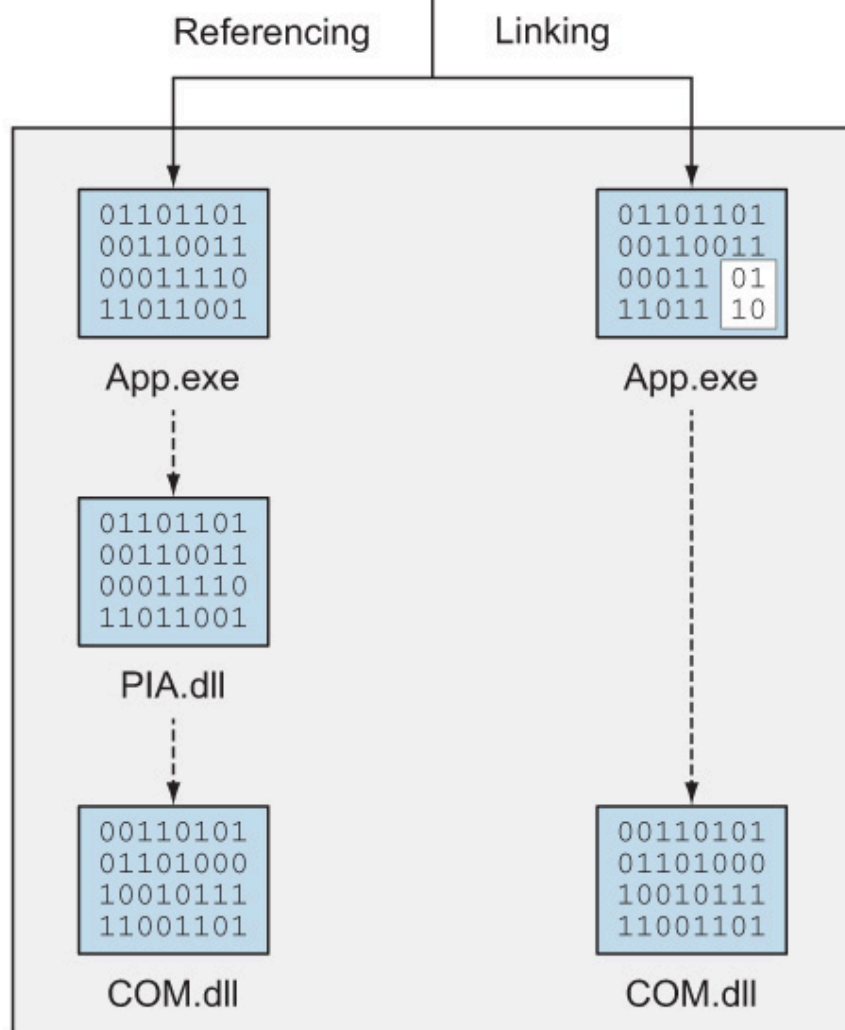
---

**Figure 4.3. Comparing referencing and linking**

Compile time



Execution time



#### Listing 4.16. Setting a range of values in Excel with implicit dynamic conversion

```
1 var app = new Application { Visible = true };
2 app.Workbooks.Add();
3 Worksheet sheet = app.ActiveSheet;
4 Range start = sheet.Cells[1, 1];
5 Range end = sheet.Cells[1, 20];
6 sheet.Range[start, end].Value = Enumerable.Range(1, 20).ToArray();
```

#### Listing 4.17. Creating a Word document and saving it before C# 4

```

1  object missing = Type.Missing;
2
3  Application app = new Application { Visible = true };
4  Document doc = app.Documents.Add
5      (ref missing, ref missing,
6       ref missing, ref missing);
7  Paragraph para = doc.Paragraphs.Add(ref missing);
8  para.Range.Text = "Awkward old code";
9
10 object fileName = "demo1.docx";
11 doc.SaveAs2(ref fileName, ref missing,
12             ref missing, ref missing, ref missing,
13             ref missing, ref missing, ref missing,
14             ref missing, ref missing, ref missing,
15             ref missing, ref missing, ref missing,
16             ref missing, ref missing);
17
18 doc.Close(ref missing, ref missing, ref missing);
19 app.Application.Quit(
20     ref missing, ref missing, ref missing);

```

#### Listing 4.18. Creating a Word document and saving it using C# 4

```

1 Application app = new Application { Visible = true };
2 Document doc = app.Documents.Add();
3 Paragraph para = doc.Paragraphs.Add();
4 para.Range.Text = "Simple new code";
5
6 doc.SaveAs2(FileName: "demo2.docx");
7
8 doc.Close();
9 app.Application.Quit();

```

```

1 SynonymInfo SynonymInfo(string Word, ref object LanguageId = Type.Missing)

```

#### Listing 4.19. Accessing a named indexer

```

1 Application app = new Application { Visible = false };
2
3 object missing = Type.Missing;
4 SynonymInfo info = app.get_SynonymInfo("method", ref missing);
5 Console.WriteLine("'method' has {0} meanings", info.MeaningCount);
6
7 info = app.SynonymInfo["index"];
8 Console.WriteLine("'index' has {0} meanings", info.MeaningCount);

```

```
1 IEnumerable<string> strings = new List<string> { "a", "b", "c" };
2 IEnumerable<object> objects = strings;
```

---

```
1 IList<string> strings = new List<string> { "a", "b", "c" };
2 IList<object> objects = strings;
```

---

1

```
1 IList<string> strings = new List<string> { "a", "b", "c" };
2 IList<object> objects = strings;
3 objects.Add(new object());
4 string element = strings[3];
```

---

1

2

```
1 Action<object> objectAction = obj => Console.WriteLine(obj);
2 Action<string> stringAction = objectAction;
3 stringAction("Print me");
```

---

```
1 public interface IEnumerable<out T>
2 public delegate void Action<in T>
3 public interface IList<T>
```

---

```
1 public delegate void InvalidCovariant<out T>(T input)
```

---

```
1 public interface IInvalidContravariant<in T>
2 {
3     T GetValue();
4 }
```

---

```
1 public TResult Func<in T, out TResult>(T arg)
```

---

```
1 public class SimpleEnumerable<T> : IEnumerable<T>
2 {
3
4 }
```

---

1

2

#### Listing 4.20. Creating a `List<object>` from a string query without variance

```
1 IEnumerable<string> strings = new[] { "a", "b", "cdefg", "hij" };
2 List<object> list = strings
3     .Where(x => x.Length > 1)
4     .Cast<object>()
5     .ToList();
```

---

#### Listing 4.21. Creating a `List<object>` from a string query by using variance

```
1 IEnumerable<string> strings = new[] { "a", "b", "cdefg", "hij" };
2 List<object> list = strings
3     .Where(x => x.Length > 1)
4     .ToList<object>();
```

---

#### Listing 4.22. Sorting a `List<Circle>` with an `IComparer<Shape>`

```
1 List<Circle> circles = new List<Circle>
2 {
```

```
3     new Circle(5.3),
4     new Circle(2),
5     new Circle(10.5)
6 };
7 circles.Sort(new AreaComparer());
8 foreach (Circle circle in circles)
9 {
10     Console.WriteLine(circle.Radius);
11 }
```

---

# CHAPTER 5

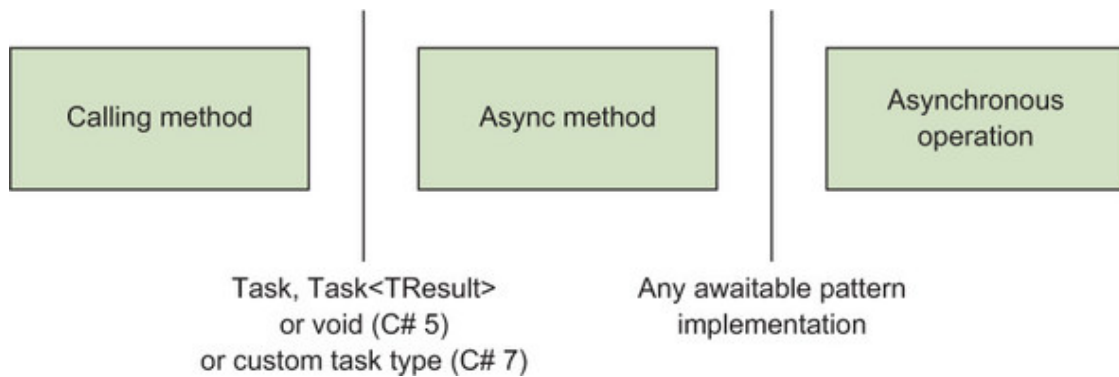
## Listing 5.1. Displaying a page length asynchronously

```
1 public class AsyncIntro : Form
2 {
3     private static readonly HttpClient client = new HttpClient();
4     private readonly Label label;
5     private readonly Button button;
6     public AsyncIntro()
7     {
8         label = new Label
9         {
10             Location = new Point(10, 20),
11             Text = "Length"
12         };
13         button = new Button
14         {
15             Location = new Point(10, 50),
16             Text = "Click"
17         };
18         button.Click += DisplayWebSiteLength;
19         AutoSize = true;
20         Controls.Add(label);
21         Controls.Add(button);
22     }
23
24     async void DisplayWebSiteLength(object sender, EventArgs e)
25     {
26         label.Text = "Fetching...";
27         string text = await client.GetStringAsync(
28             "http://csharpindepth.com");
29         label.Text = text.Length.ToString();
30     }
31
32     static void Main()
33     {
34         Application.Run(new AsyncIntro());
35     }
36 }
```

```
1 async void DisplayWebSiteLength(object sender, EventArgs e)
2 {
3     label.Text = "Fetching...";
4     Task<string> task = client.GetStringAsync("http://csharpindepth.com");
5     string text = await task;
6     label.Text = text.Length.ToString();
7 }
```

```
1 Console.WriteLine("First");
2 Console.WriteLine("Second");
```

Figure 5.1. Modeling asynchronous boundaries

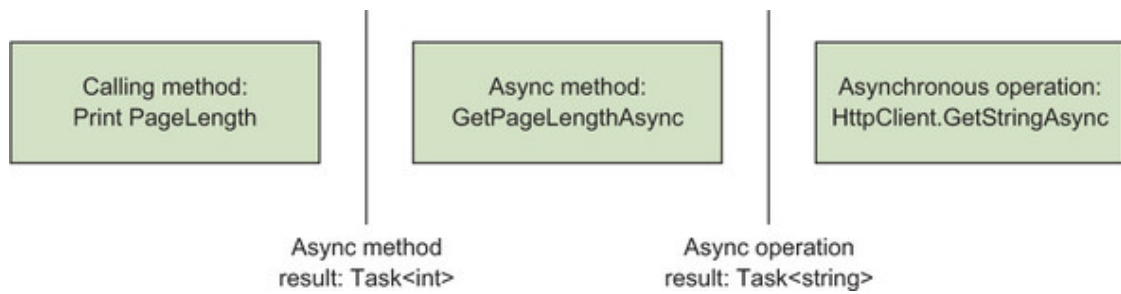


Listing 5.2. Retrieving a page length in an asynchronous method

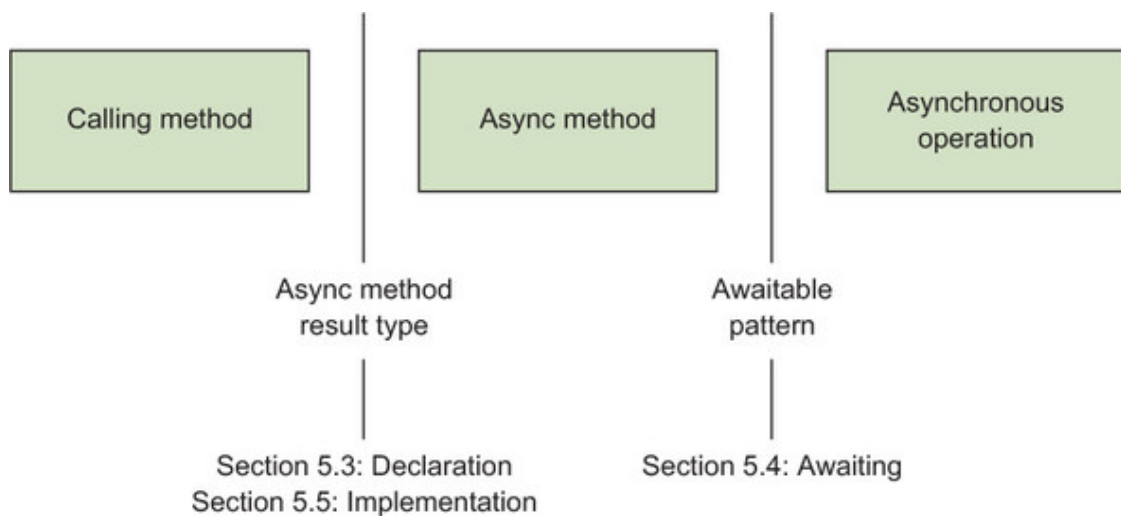
```
1 static readonly HttpClient client = new HttpClient();
2
3 static async Task<int> GetPageLengthAsync(string url)
4 {
5     Task<string> fetchTextTask = client.GetStringAsync(url);
6     int length = (await fetchTextTask).Length;
7     return length;
8 }
9
10 static void PrintPageLength()
11 {
12     Task<int> lengthTask =
13         GetPageLengthAsync("http://csharpindepth.com");
14     Console.WriteLine(lengthTask.Result);
15 }
```

Figure 5.2. Applying the details of [listing 5.2](#) to the general pattern shown in [figure 5.1](#)





**Figure 5.3. Demonstrating how [sections 5.3](#), [5.4](#), and [5.5](#) fit into the conceptual model of asynchrony**



```
1 public static async Task<int> FooAsync() { ... }
2 public async static Task<int> FooAsync() { ... }
3 async public Task<int> FooAsync() { ... }
4 public async virtual Task<int> FooAsync() { ... }
```

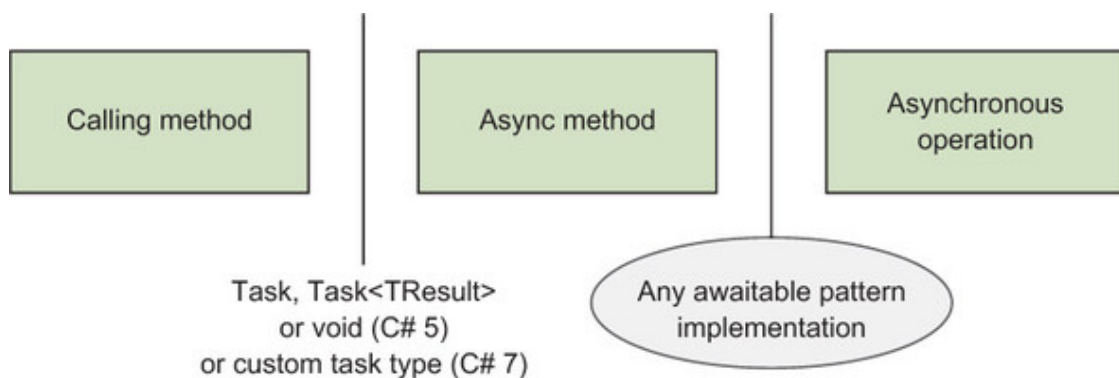
```
1 private async void LoadStockPrice(object sender, EventArgs e)
2 {
3     string ticker = tickerInput.Text;
4     decimal price = await stockPriceService.FetchPriceAsync(ticker);
5     priceDisplay.Text = price.ToString("c");
6 }
```

```
1 loadStockPriceButton.Click += LoadStockPrice;
```

```
1 int result = await foo.Bar().Baz();
```

```
1 int result = await (foo.Bar().Baz());
```

**Figure 5.4.** The awaitable pattern enables async methods to asynchronously wait for operations to complete



```
1 public class Task
2 {
3     public static YieldAwaitable Yield();
4 }
5
6 public struct YieldAwaitable
7 {
8     public YieldAwaiter GetAwaiter();
9
10    public struct YieldAwaiter : INotifyCompletion
11    {
12        public bool IsCompleted { get; }
13        public void OnCompleted(Action continuation);
14        public void GetResult();
15    }
16 }
```

```
1 public async Task ValidPrintYieldPrint()
2 {
3     Console.WriteLine("Before yielding");
4     await Task.Yield();
5     Console.WriteLine("After yielding");
6 }
```

1

```
1 public async Task InvalidPrintYieldPrint()
2 {
3     Console.WriteLine("Before yielding");
4     var result = await Task.Yield();
5     Console.WriteLine("After yielding");
6 }
```

1

```
1 var result = Console.WriteLine("WriteLine is a void method");
```

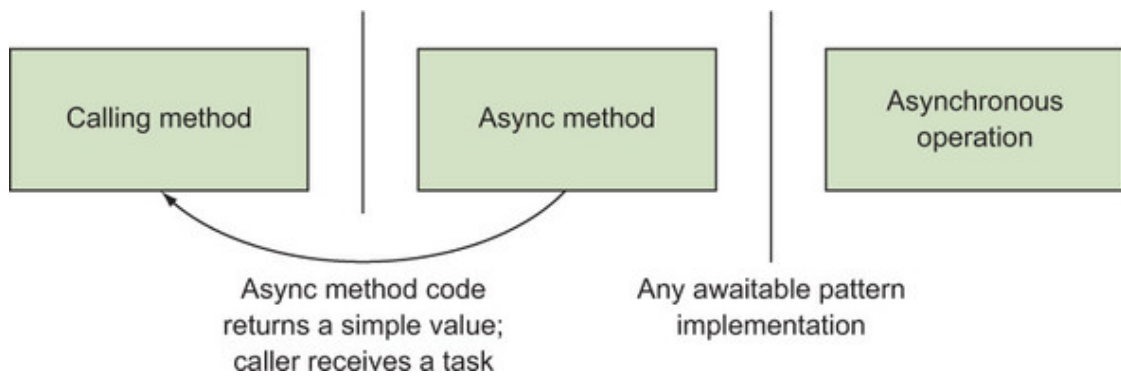
### Listing 5.3. Using unsafe code in an async method

```
1 static async Task DelayWithResultOfUnsafeCode(string text)
2 {
3     int total = 0;
4     unsafe
5     {
6         fixed (char* textPointer = text)
7         {
8             char* p = textPointer;
9             while (*p != 0)
10            {
11                total += *p;
12                p++;
13            }
14        }
15    }
16    Console.WriteLine("Delaying for " + total + "ms");
17    await Task.Delay(total);
18    Console.WriteLine("Delay complete");
19 }
```

1

2

Figure 5.5. Returning a result from an async method to its caller



```
1 static async Task<int> GetPageLengthAsync(string url)
2 {
3     Task<string> fetchTextTask = client.GetStringAsync(url);
4     int length = (await fetchTextTask).Length;
5     return length;
6 }
```

```
1 string pageText = await new HttpClient().GetStringAsync(url);
```

```
1 Task<string> task = new HttpClient().GetStringAsync(url);
2 string pageText = await task;
```

```
1 AddPayment(await employee.GetHourlyRateAsync() *
2             await timeSheet.GetHoursWorkedAsync(employee.Id));
```

```
1 Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
2 decimal hourlyRate = await hourlyRateTask;
3 Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
4 int hoursWorked = await hoursWorkedTask;
5 AddPayment(hourlyRate * hoursWorked);
```

```
1 Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
2 Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
3 AddPayment(await hourlyRateTask * await hoursWorkedTask);
```

---

### Listing 5.4. Awaiting completed and noncompleted tasks

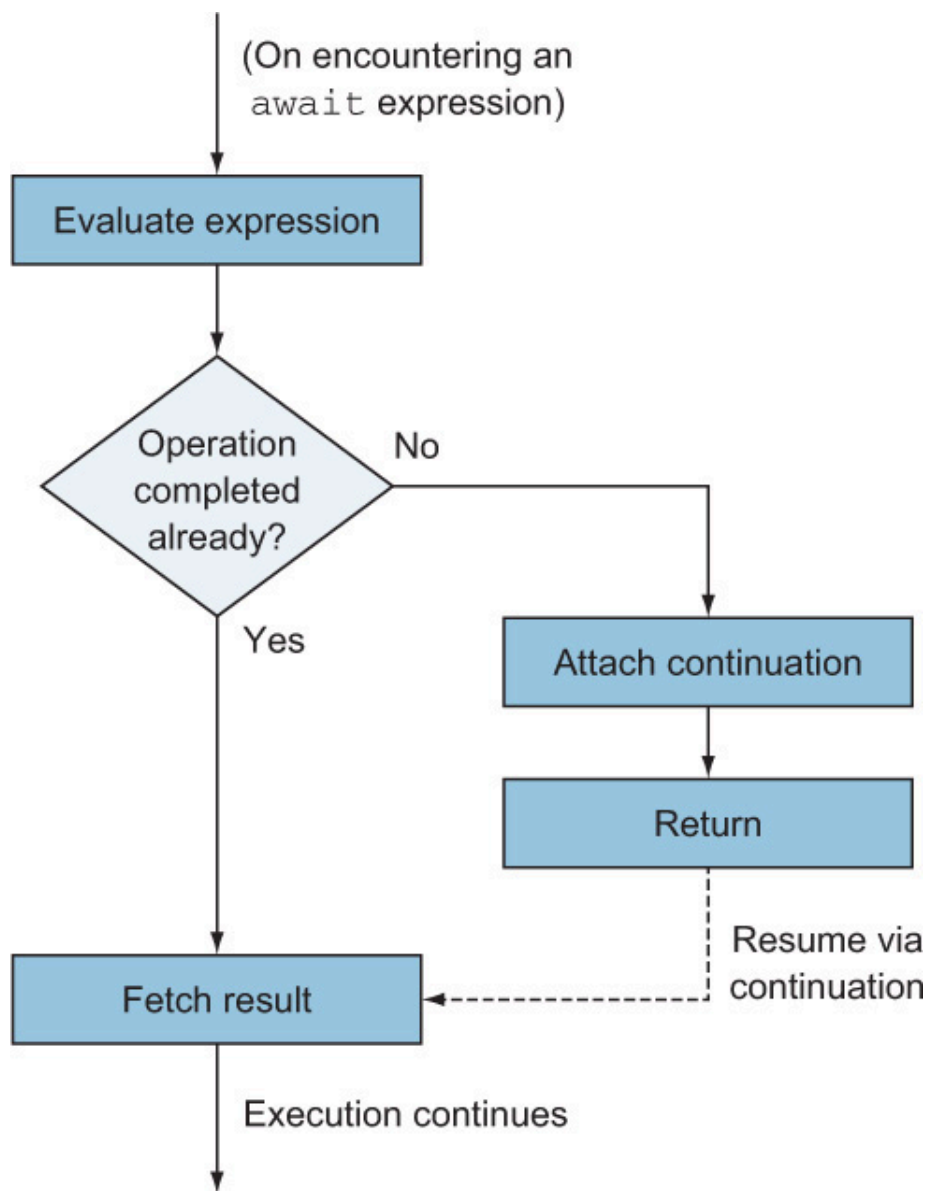
```
1 static void Main()
2 {
3     Task task = DemoCompletedAsync();
4     Console.WriteLine("Method returned");
5     task.Wait();
6     Console.WriteLine("Task completed");
7 }
8
9 static async Task DemoCompletedAsync()
10 {
11     Console.WriteLine("Before first await");
12     await Task.FromResult(10);
13     Console.WriteLine("Between awaits");
14     await Task.Delay(1000);
15     Console.WriteLine("After second await");
16 }
```

---

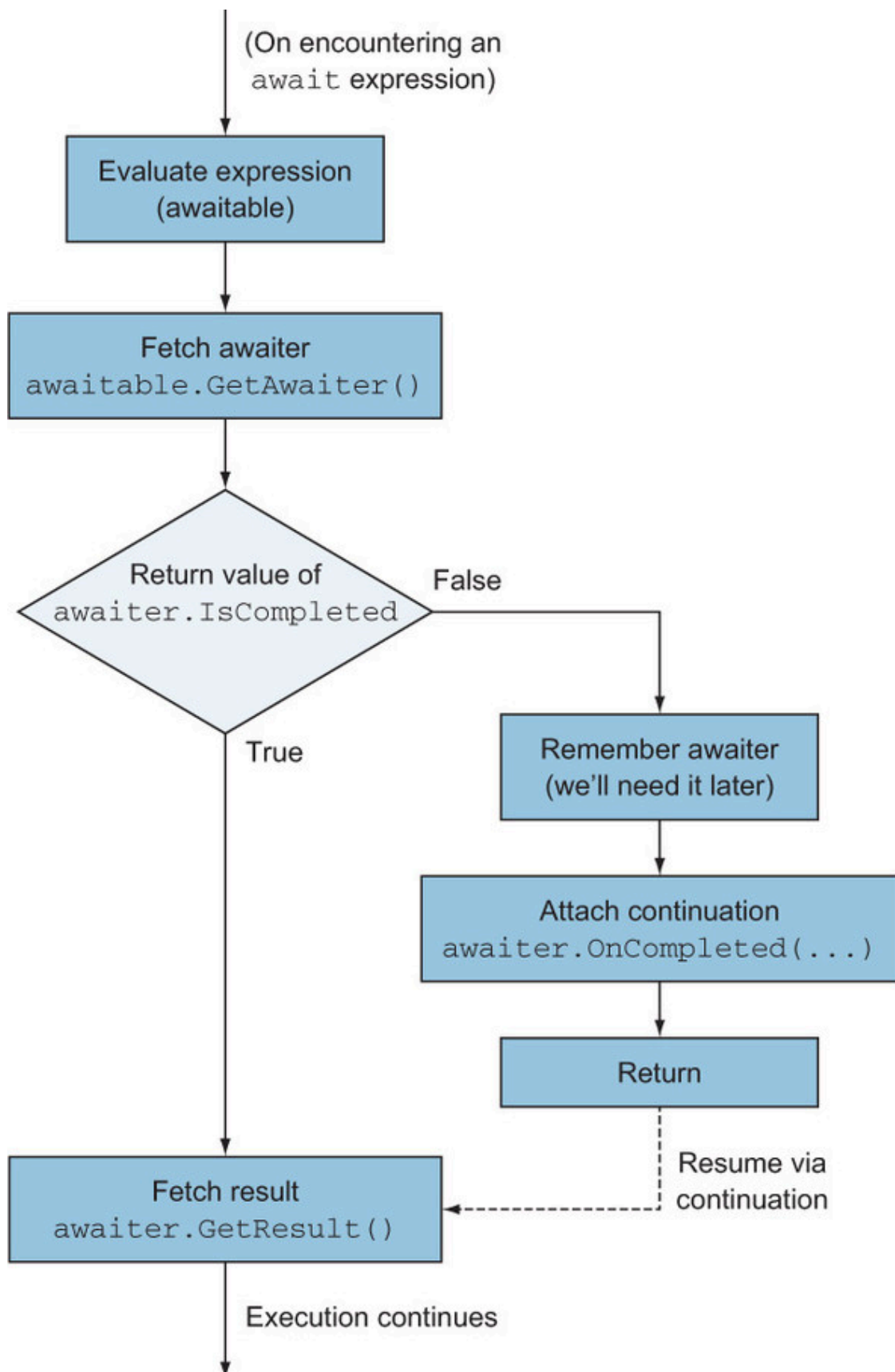
```
1 Before first await
2 Between awaits
3 Method returned
4 After second await
5 Task completed
```

---

Figure 5.6. User-visible model of await handling



**Figure 5.7.** Await handling via the awaitable pattern



### Listing 5.5. Catching exceptions when fetching web pages

```
1 async Task<string> FetchFirstSuccessfulAsync(IEnumerable<string> urls)
2 {
3     var client = new HttpClient();
4     foreach (string url in urls)
5     {
6         try
7         {
```

```

8         return await client.GetStringAsync(url);
9     }
10    catch (HttpRequestException exception)
11    {
12        Console.WriteLine("Failed to fetch {0}: {1}",
13            url, exception.Message);
14    }
15 }
16 throw new HttpRequestException("No URLs succeeded");
17 }

```

## Listing 5.6. Broken argument validation in an async method

```

1  static async Task MainAsync()
2  {
3      Task<int> task = ComputeLengthAsync(null);
4      Console.WriteLine("Fetched the task");
5      int length = await task;
6      Console.WriteLine("Length: {0}", length);
7  }
8
9  static async Task<int> ComputeLengthAsync(string text)
10 {
11     if (text == null)
12     {
13         throw new ArgumentNullException("text");
14     }
15     await Task.Delay(500);
16     return text.Length;
17 }

```

## Listing 5.7. Eager argument validation with a separate method

```

1  static Task<int> ComputeLengthAsync(string text)
2  {
3      if (text == null)
4      {
5          throw new ArgumentNullException("text");
6      }
7      return ComputeLengthAsyncImpl(text);
8  }
9
10 static async Task<int> ComputeLengthAsyncImpl(string text)
11 {
12     await Task.Delay(500);
13     return text.Length;
14 }

```



## Listing 5.8. Creating a canceled task by throwing `OperationCanceledException`

```
1 static async Task ThrowCancellationException()
2 {
3     throw new OperationCanceledException();
4 }
5 ...
6 Task task = ThrowCancellationException();
7 Console.WriteLine(task.Status);
```

---

```
1 Func<Task> lambda = async () => await Task.Delay(1000);
2 Func<Task<int>> anonMethod = async delegate()
3 {
4     Console.WriteLine("Started");
5     await Task.Delay(1000);
6     Console.WriteLine("Finished");
7     return 10;
8 };
```

---

## Listing 5.9. Creating and calling an asynchronous function using a lambda expression

```
1 Func<int, Task<int>> function = async x =>
2 {
3     Console.WriteLine("Starting... x={0}", x);
4     await Task.Delay(x * 1000);
5     Console.WriteLine("Finished... x={0}", x);
6     return x * 2;
7 };
8 Task<int> first = function(5);
9 Task<int> second = function(3);
10 Console.WriteLine("First result: {0}", first.Result);
11 Console.WriteLine("Second result: {0}", second.Result);
```

---

```
1 Starting... x=5
2 Starting... x=3
3 Finished... x=3
4 Finished... x=5
5 First result: 10
6 Second result: 6
```

---

## Listing 5.10. Wrapping a stream for efficient asynchronous byte-wise access

```
1 public sealed class ByteStream : IDisposable
2 {
3     private readonly Stream stream;
4     private readonly byte[] buffer;
5     private int position;
6     private int bufferedBytes;
7
8     public ByteStream(Stream stream)
9     {
10         this.stream = stream;
11         buffer = new byte[1024 * 8];
12     }
13
14     public async ValueTask<byte?> ReadByteAsync()
15     {
16         if (position == bufferedBytes)
17         {
18             position = 0;
19             bufferedBytes = await
20                 stream.ReadAsync(buffer, 0, buffer.Length)
21                 .ConfigureAwait(false);
22             if (bufferedBytes == 0)
23             {
24                 return null;
25             }
26         }
27         return buffer[position++];
28     }
29
30     public void Dispose()
31     {
32         stream.Dispose();
33     }
34 }
35
36 Sample usage
37 using (var stream = new ByteStream(File.OpenRead("file.dat")))
38 {
39     while ((nextByte = await stream.ReadByteAsync()).HasValue)
40     {
41         ConsumeByte(nextByte.Value);
42     }
43 }
```

1  
2

3

4

5  
6

7

8

9

## Listing 5.11. Skeleton of the members required for a generic task type

```
1 [AsyncMethodBuilder(typeof(CustomTaskBuilder<>))]
2 public class CustomTask<T>
3 {
4     public CustomTaskAwaiter<T> GetAwaiter();
5 }
6
7 public class CustomTaskAwaiter<T> : INotifyCompletion
8 {
9     public bool IsCompleted { get; }
10    public T GetResult();
11    public void OnCompleted(Action continuation);
12 }
13
```

```

14 public class CustomTaskBuilder<T>
15 {
16     public static CustomTaskBuilder<T> Create();
17
18     public void Start<TStateMachine>(ref TStateMachine stateMachine)
19         where TStateMachine : IAsyncStateMachine;
20
21     public void SetStateMachine(IAsyncStateMachine stateMachine);
22     public void SetException(Exception exception);
23     public void SetResult(T result);
24
25     public void AwaitOnCompleted<TAwaiter, TStateMachine>
26         (ref TAwaiter awaiter, ref TStateMachine stateMachine)
27         where TAwaiter : INotifyCompletion
28         where TStateMachine : IAsyncStateMachine;
29
30     public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>
31         (ref TAwaiter awaiter, ref TStateMachine stateMachine)
32         where TAwaiter : INotifyCompletion
33         where TStateMachine : IAsyncStateMachine;
34
35     public CustomTask<T> Task { get; }
36 }

```

---

### Listing 5.12. A simple async entry point

```

1 static async Task Main()
2 {
3     Console.WriteLine("Before delay");
4     await Task.Delay(1000);
5     Console.WriteLine("After delay");
6 }

```

---

```

1 static void <Main>()
2 {
3     Main().GetAwaiter().GetResult();
4 }

```

---

1

```

1 static async Task<int> GetPageLengthAsync(string url)
2 {
3     var fetchTextTask = client.GetStringAsync(url);
4     int length = (await fetchTextTask).Length;
5
6     return length;
7 }

```

---

1

```
1 static async Task<int> GetPageLengthAsync(string url)
2 {
3     var fetchTextTask = client.GetStringAsync(url).ConfigureAwait(false);
4     int length = (await fetchTextTask).Length;
5
6     return length;
7 }
```

---

1

```
1 string text = await client.GetStringAsync(url).ConfigureAwait(false);
```

---

```
1 Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
2 decimal hourlyRate = await hourlyRateTask;
3 Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
4 int hoursWorked = await hoursWorkedTask;
5 AddPayment(hourlyRate * hoursWorked);
```

---

```
1 Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
2 Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
3 AddPayment(await hourlyRateTask * await hoursWorkedTask);
```

---

```
1 Task<decimal> hourlyRateTask = employee.GetHourlyRateAsync();
2 Task<int> hoursWorkedTask = timeSheet.GetHoursWorkedAsync(employee.Id);
3 decimal hourlyRate = await hourlyRateTask;
4 int hoursWorked = await hoursWorkedTask;
5 AddPayment(hourlyRate * hoursWorked);
```

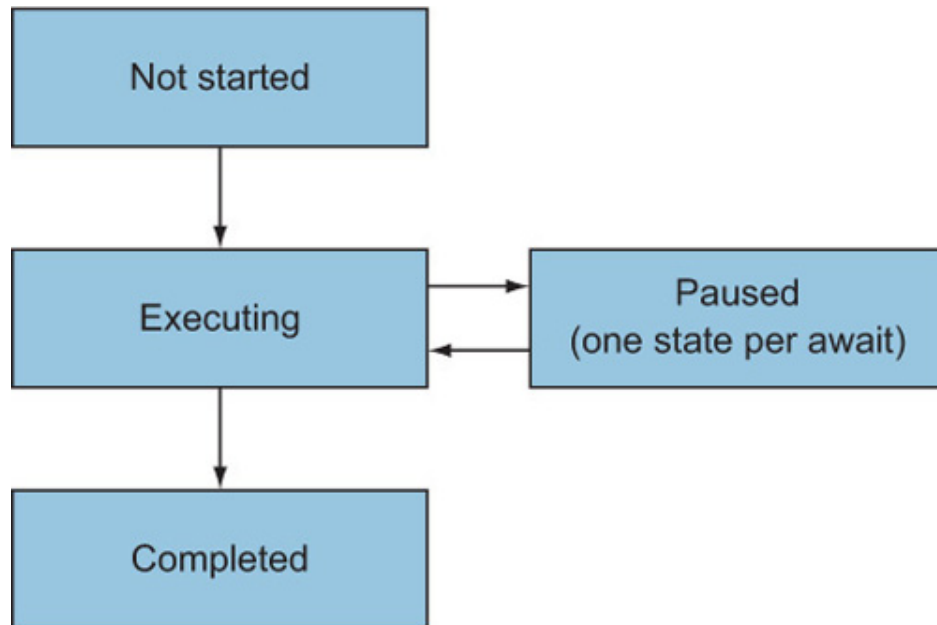
---

```
1 [Test]
2 public async Task FooAsync()
```

3 {  
4  
5 }

# CHAPTER 6

Figure 6.1. State transition diagram



Listing 6.1. Simple introductory async method

```
1 static async Task PrintAndWait(TimeSpan delay)
2 {
3     Console.WriteLine("Before first delay");
4     await Task.Delay(delay);
5     Console.WriteLine("Between delays");
6     await Task.Delay(delay);
7     Console.WriteLine("After second delay");
8 }
```

Listing 6.2. Generated code for [listing 6.1](#) (except for `MoveNext` )

```
1 Stub method
2 [AsyncStateMachine(typeof(PrintAndWaitStateMachine))]
3 [DebuggerStepThrough]
4 private static unsafe Task PrintAndWait(TimeSpan delay)
5 {
6     var machine = new PrintAndWaitStateMachine
7     {
8         delay = delay,
9         builder = AsyncTaskMethodBuilder.Create(),
10        state = -1
11    }
```

```

11     };
12     machine.builder.Start(ref machine);
13     return machine.builder.Task;
14 }
15
16 Private struct for the state machine
17 [CompilerGenerated]
18 private struct PrintAndWaitStateMachine : IAsyncStateMachine
19 {
20     public int state;
21     public AsyncTaskMethodBuilder builder;
22     private TaskAwaiter awaiter;
23     public TimeSpan delay;
24
25     void IAsyncStateMachine.MoveNext()
26     {
27     }
28
29     [DebuggerHidden]
30     void IAsyncStateMachine.SetStateMachine(
31         IAsyncStateMachine stateMachine)
32     {
33         this.builder.SetStateMachine(stateMachine);
34     }
35 }

```

9

```

1 [AsyncStateMachine(typeof(PrintAndWaitStateMachine))]
2 [DebuggerStepThrough]
3 private static unsafe Task PrintAndWait(TimeSpan delay)
4 {
5     var machine = new PrintAndWaitStateMachine
6     {
7         delay = delay,
8         builder = AsyncTaskMethodBuilder.Create(),
9         state = -1
10    };
11    machine.builder.Start(ref machine);
12    return machine.builder.Task;
13 }

```

```

1 var builder = machine.builder;
2 builder.Start(ref machine);
3 return builder.Task;

```

1

```

1 [CompilerGenerated]
2 private struct PrintAndWaitStateMachine : IAsyncStateMachine
3 {
4     public int state;

```

```
5     public AsyncTaskMethodBuilder builder;
6     private TaskAwaiter awaiter;
7     public TimeSpan delay;
8
9     void IAsyncStateMachine.MoveNext()
10    {
11
12    }
13
14    [DebuggerHidden]
15    void IAsyncStateMachine.SetStateMachine(
16        IAsyncStateMachine stateMachine)
17    {
18        this.builder.SetStateMachine(stateMachine);
19    }
20 }
```

---

1

```
1 public async Task LocalVariableDemoAsync()
2 {
3     int x = DateTime.UtcNow.Second;
4     int y = DateTime.UtcNow.Second;
5     Console.WriteLine(y);
6     await Task.Delay();
7     Console.WriteLine(x);
8 }
```

---

1

2

3

```
1 public async Task TemporaryStackDemoAsync()
2 {
3     Task<int> task = Task.FromResult(10);
4     DateTime now = DateTime.UtcNow;
5     int result = now.Second + now.Hours * await task;
6 }
```

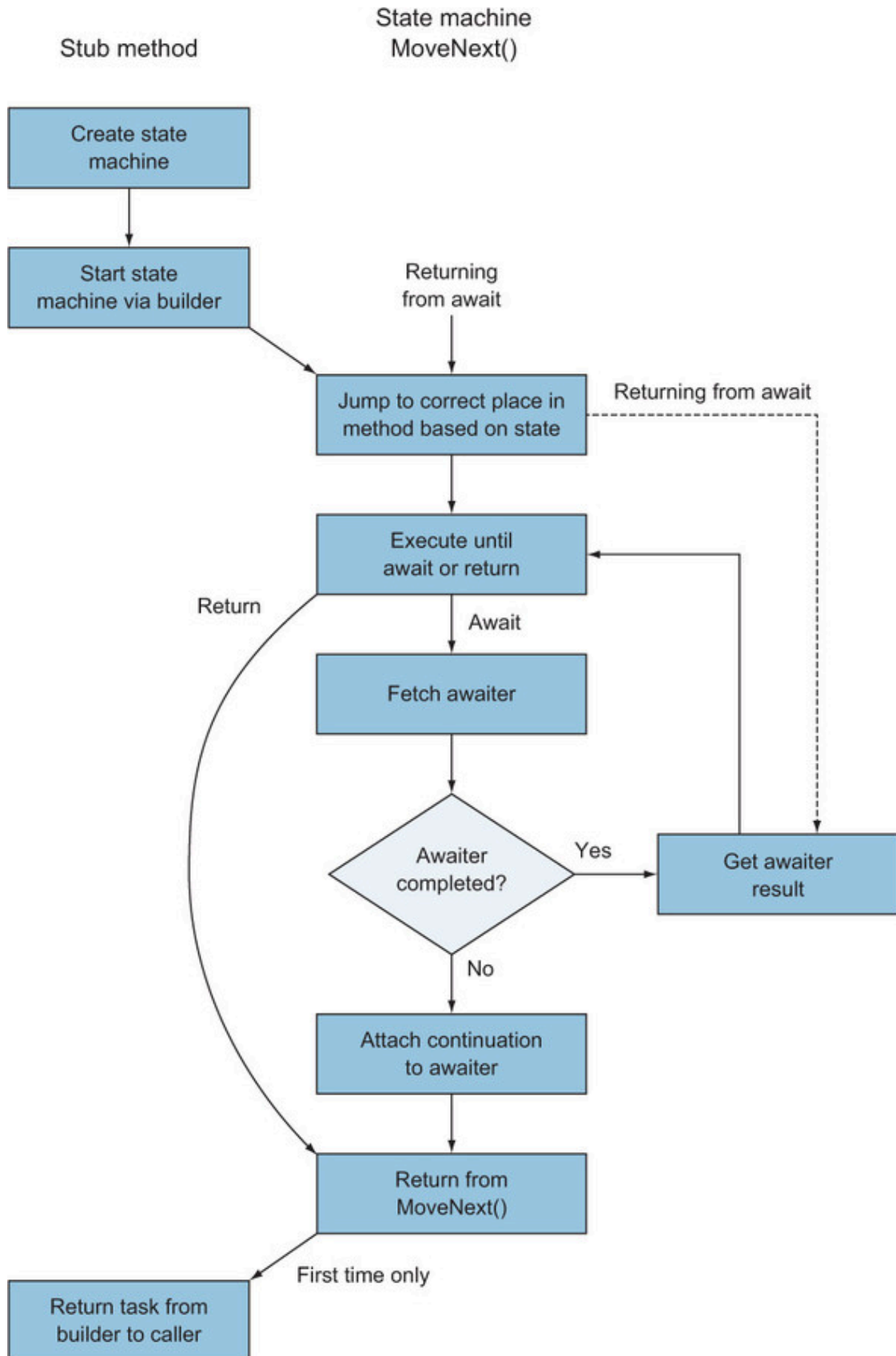
---

```
1 public async Task TemporaryStackDemoAsync()
2 {
3     Task<int> task = Task.FromResult(10);
4     DateTime now = DateTime.UtcNow;
5     int tmp1 = now.Second;
6     int tmp2 = now.Hours;
7     int result = tmp1 + tmp2 * await task;
8 }
```

---



Figure 6.2. Flowchart of an async method



```

1 void IAsyncStateMachine.SetStateMachine(
2     IAsyncStateMachine stateMachine)
3 {
4     this.builder.SetStateMachine(stateMachine);
5 }

```

```

1 void BoxAndRemember<TStateMachine>(ref TStateMachine stateMachine)
2     where TStateMachine : IStateMachine
3 {
4     IStateMachine boxed = stateMachine;
5     boxed.SetStateMachine(boxed);
6 }

```

---

```

1 static async Task PrintAndWait(TimeSpan delay)
2 {
3     Console.WriteLine("Before first delay");
4     await Task.Delay(delay);
5     Console.WriteLine("Between delays");
6     await Task.Delay(delay);
7     Console.WriteLine("After second delay");
8 }

```

---

### Listing 6.3. The decompiled `MoveNext()` method from [listing 6.1](#)

```

1 void IAsyncStateMachine.MoveNext()
2 {
3     int num = this.state;
4     try
5     {
6         TaskAwaiter awaiter1;
7         switch (num)
8         {
9             default:
10                goto MethodStart;
11             case 0:
12                goto FirstAwaitContinuation;
13             case 1:
14                goto SecondAwaitContinuation;
15         }
16     MethodStart:
17         Console.WriteLine("Before first delay");
18         awaiter1 = Task.Delay(this.delay).GetAwaiter();
19         if (awaiter1.IsCompleted)
20         {
21             goto GetFirstAwaitResult;
22         }
23         this.state = num = 0;
24         this.awaiter = awaiter1;
25         this.builder.AwaitUnsafeOnCompleted(ref awaiter1, ref this);
26         return;
27     FirstAwaitContinuation:
28         awaiter1 = this.awaiter;
29         this.awaiter = default(TaskAwaiter);

```

```

30     this.state = num = -1;
31     GetFirstAwaitResult:
32         awaiter1.GetResult();
33         Console.WriteLine("Between delays");
34         TaskAwaiter awaiter2 = Task.Delay(this.delay).GetAwaiter();
35         if (awaiter2.IsCompleted)
36         {
37             goto GetSecondAwaitResult;
38         }
39         this.state = num = 1;
40         this.awaiter = awaiter2;
41         this.builder.AwaitUnsafeOnCompleted(ref awaiter2, ref this);
42         return;
43     SecondAwaitContinuation:
44         awaiter2 = this.awaiter;
45         this.awaiter = default(TaskAwaiter);
46         this.state = num = -1;
47     GetSecondAwaitResult:
48         awaiter2.GetResult();
49         Console.WriteLine("After second delay");
50     }
51     catch (Exception exception)
52     {
53         this.state = -2;
54         this.builder.SetException(exception);
55         return;
56     }
57     this.state = -2;
58     this.builder.SetResult();
59 }

```

---

#### Listing 6.4. Pseudocode of a `MoveNext()` method

```

1  void IAsyncStateMachine.MoveNext()
2  {
3      try
4      {
5          switch (this.state)
6          {
7              default: goto MethodStart;
8              case 0: goto Label0A;
9              case 1: goto Label1A;
10             case 2: goto Label2A;
11         }
12     }
13     MethodStart:
14
15
16     Label0A:
17
18     Label0B:
19
20
21 }
22 catch (Exception e)
23 {
24     this.state = -2;
25     builder.SetException(e);
26     return;
27 }
28 this.state = -2;
29

```

1

2

3

4

5

6

```
30     builder.SetResult();  
    }
```

8

7

### Listing 6.5. A section of [listing 6.3](#) corresponding to a single await

```
1  awaiter1 = Task.Delay(this.delay).GetAwaiter();  
2      if (awaiter1.IsCompleted)  
3      {  
4          goto GetFirstAwaitResult;  
5      }  
6      this.state = num = 0;  
7      this.awaiter = awaiter1;  
8      this.builder.AwaitUnsafeOnCompleted(ref awaiter1, ref this);  
9      return;  
10 FirstAwaitContinuation:  
11     awaiter1 = this.awaiter;  
12     this.awaiter = default(TaskAwaiter);  
13     this.state = num = -1;  
14 GetFirstAwaitResult:  
15     awaiter1.GetResult();
```

```
1 await Task.Delay(delay);
```

### Listing 6.6. Introducing a loop between await expressions

```
1  static async Task PrintAndWaitWithSimpleLoop(TimeSpan delay)  
2  {  
3      Console.WriteLine("Before first delay");  
4      await Task.Delay(delay);  
5      for (int i = 0; i < 3; i++)  
6      {  
7          Console.WriteLine("Between delays");  
8      }  
9      await Task.Delay(delay);  
10     Console.WriteLine("After second delay");  
11 }
```

```
1 GetFirstAwaitResult:  
2     awaiter1.GetResult();  
3
```

```
4 Console.WriteLine("Between delays");
TaskAwaiter awaiter2 = Task.Delay(this.delay).GetAwaiter();
```

---

```
1 GetFirstAwaitResult:
2     awaiter1.GetResult();
3     for (int i = 0; i < 3; i++)
4     {
5         Console.WriteLine("Between delays");
6     }
7     TaskAwaiter awaiter2 = Task.Delay(this.delay).GetAwaiter();
```

---

### Listing 6.7. Awaiting in a loop

```
1 static async Task AwaitInLoop(TimeSpan delay)
2 {
3     Console.WriteLine("Before loop");
4     for (int i = 0; i < 3; i++)
5     {
6         Console.WriteLine("Before await in loop");
7         await Task.Delay(delay);
8         Console.WriteLine("After await in loop");
9     }
10    Console.WriteLine("After loop delay");
11 }
```

---

### Listing 6.8. Decompiled loop without using any loop constructs

```
1 switch (num)
2 {
3     default:
4         goto MethodStart;
5     case 0:
6         goto AwaitContinuation;
7 }
8 MethodStart:
9     Console.WriteLine("Before loop");
10    this.i = 0;
11    goto ForLoopCondition;
12 ForLoopBody:
13    Console.WriteLine("Before await in loop");
14    TaskAwaiter awaiter = Task.Delay(this.delay).GetAwaiter();
15    if (awaiter.IsCompleted)
16    {
17        goto GetAwaitResult;
18    }
19    this.state = num = 0;
20    this.awaiter = awaiter;
```

1  
2  
3

```

21     this.builder.AwaitUnsafeOnCompleted(ref awaiter, ref this);
22     return;
23 AwaitContinuation:
24     awaiter = this.awaiter;
25     this.awaiter = default(TaskAwaiter);
26     this.state = num = -1;
27 GetAwaitResult:
28     awaiter.GetResult();
29     Console.WriteLine("After await in loop");
30     this.i++;
31 ForLoopCondition:
32     if (this.i < 3)
33     {
34         goto ForLoopBody;
35     }
36     Console.WriteLine("After loop delay");

```

4

5

6

---

### Listing 6.9. Awaiting within a `try` block

```

1 static async Task AwaitInTryFinally(TimeSpan delay)
2 {
3     Console.WriteLine("Before try block");
4     await Task.Delay(delay);
5     try
6     {
7         Console.WriteLine("Before await");
8         await Task.Delay(delay);
9         Console.WriteLine("After await");
10    }
11    finally
12    {
13        Console.WriteLine("In finally block");
14    }
15    Console.WriteLine("After finally block");
16 }

```

---

```

1 switch (num)
2 {
3     default:
4         goto MethodStart;
5     case 0:
6         goto AwaitContinuation;
7 }
8 MethodStart:
9     ...
10    try
11    {
12        ...
13    AwaitContinuation:
14        ...
15    GetAwaitResult:
16        ...
17    }
18    finally
19    {

```

```
20     ...
21 }
22 ...
```

## Listing 6.10. Decompiled await within `try/finally`

```
1  switch (num)
2  {
3      default:
4          goto MethodStart;
5      case 0:
6          goto AwaitContinuationTrampoline;
7  }
8  MethodStart:
9      Console.WriteLine("Before try");
10 AwaitContinuationTrampoline:
11     try
12     {
13         switch (num)
14         {
15             default:
16                 goto TryBlockStart;
17             case 0:
18                 goto AwaitContinuation;
19         }
20     TryBlockStart:
21         Console.WriteLine("Before await");
22         TaskAwaiter awaiter = Task.Delay(this.delay).GetAwaiter();
23         if (awaiter.IsCompleted)
24         {
25             goto GetAwaitResult;
26         }
27         this.state = num = 0;
28         this.awaiter = awaiter;
29         this.builder.AwaitUnsafeOnCompleted(ref awaiter, ref this);
30         return;
31     AwaitContinuation:
32         awaiter = this.awaiter;
33         this.awaiter = default(TaskAwaiter);
34         this.state = num = -1;
35     GetAwaitResult:
36         awaiter.GetResult();
37         Console.WriteLine("After await");
38     }
39     finally
40     {
41         if (num < 0)
42         {
43             Console.WriteLine("In finally block");
44         }
45     }
46     Console.WriteLine("After finally block");
```

1

2

3

4

## Listing 6.11. A sample custom task builder

```
1 public class CustomTaskBuilder<T>
2 {
3     public static CustomTaskBuilder<T> Create();
4     public void Start<TStateMachine>(ref TStateMachine stateMachine)
5         where TStateMachine : IAsyncStateMachine;
6     public CustomTask<T> Task { get; }
7
8     public void AwaitOnCompleted<TAwaiter, TStateMachine>
9         (ref TAwaiter awaiter, ref TStateMachine stateMachine)
10        where TAwaiter : INotifyCompletion
11        where TStateMachine : IAsyncStateMachine;
12    public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>
13        (ref TAwaiter awaiter, ref TStateMachine stateMachine)
14        where TAwaiter : INotifyCompletion
15        where TStateMachine : IAsyncStateMachine;
16    public void SetStateMachine(IAsyncStateMachine stateMachine);
17
18    public void SetException(Exception exception);
19    public void SetResult(T result);
20 }
```

---



# CHAPTER 7

```
1 foreach (string name in names)
2 {
3     Console.WriteLine(name);
4 }
```

---

```
1 string name;
2 using (var iterator = names.GetEnumerator())
3 {
4     while (iterator.MoveNext())
5     {
6         name = iterator.Current;
7         Console.WriteLine(name);
8     }
9 }
```

---

## Listing 7.1. Capturing the iteration variable in a `foreach` loop

```
1 List<string> names = new List<string> { "x", "y", "z" };
2 var actions = new List<Action>();
3 foreach (string name in names)
4 {
5     actions.Add(() => Console.WriteLine(name));
6 }
7 foreach (Action action in actions)
8 {
9     action();
10 }
```

---

## Listing 7.2. Capturing the iteration variable in a `for` loop

```
1 List<string> names = new List<string> { "x", "y", "z" };
2 var actions = new List<Action>();
3 for (int i = 0; i < names.Count; i++)
4 {
5     actions.Add(() => Console.WriteLine(names[i]));
6 }
7 foreach (Action action in actions)
```

---

```
8 {  
9     action();  
10 }
```

---

3

### Listing 7.3. Basic demonstration of caller member attributes

```
1 static void ShowInfo(  
2     [CallerFilePath] string file = null,  
3     [CallerLineNumber] int line = 0,  
4     [CallerMemberName] string member = null)  
5 {  
6     Console.WriteLine("{0}:{1} - {2}", file, line, member);  
7 }  
8  
9 static void Main()  
10 {  
11     ShowInfo();  
12     ShowInfo("LiesAndDamnedLies.java", -10);  
13 }
```

---

1

2

```
1 C:\Users\jon\Projects\CSharpInDepth\Chapter07\CallerInfoDemo.cs:20 - Main  
2 LiesAndDamnedLies.java:-10 - Main
```

---

```
1 public delegate void PropertyChangedEventHandler(  
2     Object sender, PropertyChangedEventArgs e)
```

---

```
1 public PropertyChangedEventArgs(string propertyName)
```

---

### Listing 7.4. Implementing `INotifyPropertyChanged` the old way

```
1 class OldPropertyNotifier : INotifyPropertyChanged  
2 {  
3     public event PropertyChangedEventHandler PropertyChanged;
```

```

4     private int firstValue;
5     public int FirstValue
6     {
7         get { return firstValue; }
8         set
9         {
10            if (value != firstValue)
11            {
12                firstValue = value;
13                NotifyPropertyChanged("FirstValue");
14            }
15        }
16    }
17
18    // (Other properties with the same pattern)
19
20    private void NotifyPropertyChanged(string propertyName)
21    {
22        PropertyChangedEventHandler handler = PropertyChanged;
23        if (handler != null)
24        {
25            handler(this, new PropertyChangedEventArgs(propertyName));
26        }
27    }
28 }

```

---

### Listing 7.5. Using caller information to implement `INotifyPropertyChanged`

```

1  if (value != firstValue)
2  {
3      firstValue = value;
4      NotifyPropertyChanged();
5  }
6
7  void NotifyPropertyChanged([CallerMemberName] string propertyName = null)
8  {
9
10 }

```

---

1

2

### Listing 7.6. Caller information attributes and dynamic typing

```

1  static void ShowLine(string message,
2      [CallerLineNumber] int line = 0)
3  {
4      Console.WriteLine("{0}: {1}", line, message);
5  }
6
7  static int GetLineNumber(
8      [CallerLineNumber] int line = 0)
9  {
10     return line;
11 }
12
13 static void Main()

```

---

1

2

```

14 {
15     dynamic message = "Some message";
16     ShowLine(message);
17     ShowLine((string) message);
18     ShowLine(message, GetLineNumber());
19 }

```

## Listing 7.7. Caller information in a constructor

```

1 public abstract class BaseClass
2 {
3     protected BaseClass(
4         [CallerFilePath] string file = "Unspecified file",
5         [CallerLineNumber] int line = -1,
6         [CallerMemberName] string member = "Unspecified member")
7     {
8         Console.WriteLine("{0}:{1} - {2}", file, line, member);
9     }
10 }
11
12 public class Derived1 : BaseClass { }
13
14 public class Derived2 : BaseClass
15 {
16     public Derived2() { }
17 }
18
19 public class Derived3 : BaseClass
20 {
21     public Derived3() : base() {}
22 }

```

## Listing 7.8. Caller information in query expressions

```

1 string[] source =
2 {
3     "the", "quick", "brown", "fox",
4     "jumped", "over", "the", "lazy", "dog"
5 };
6 var query = from word in source
7             where word.Length > 3
8             select word.ToUpperInvariant();
9 Console.WriteLine("Data:");
10 Console.WriteLine(string.Join(", ", query));
11 Console.WriteLine("CallerInfo:");
12 Console.WriteLine(string.Join(
13     Environment.NewLine, query.CallerInfo));

```

```
1 Data:
2 QUICK, BROWN, JUMPED, OVER, LAZY
3 CallerInfo:
4 CallerInfoLinq.cs:91 - Main
5 CallerInfoLinq.cs:92 - Main
```

---

### Listing 7.9. Attribute class that captures caller information

```
1 [AttributeUsage(AttributeTargets.All)]
2 public class MemberDescriptionAttribute : Attribute
3 {
4     public MemberDescriptionAttribute(
5         [CallerFilePath] string file = "Unspecified file",
6         [CallerLineNumber] int line = 0,
7         [CallerMemberName] string member = "Unspecified member")
8     {
9         File = file;
10        Line = line;
11        Member = member;
12    }
13
14    public string File { get; }
15    public int Line { get; }
16    public string Member { get; }
17
18    public override string ToString() =>
19        $"{Path.GetFileName(File)}:{Line} - {Member}";
20 }
```

---

### Listing 7.10. Applying the attribute to a class and a method

```
1 using MDA = MemberDescriptionAttribute; 1
2
3 [MemberDescription]
4 class CallerNameInAttribute 2
5 {
6     [MemberDescription]
7     public void Method<[MemberDescription] T>(
8         [MemberDescription] int parameter) { } 3
9
10    static void Main()
11    {
12        var typeInfo = typeof(CallerNameInAttribute).GetTypeInfo();
13        var methodInfo = typeInfo.GetDeclaredMethod("Method");
14        var paramInfo = methodInfo.GetParameters()[0];
15        var typeParamInfo =
16            methodInfo.GetGenericArguments()[0].GetTypeInfo();
17        Console.WriteLine(typeInfo.GetCustomAttribute<MDA>());
18        Console.WriteLine(methodInfo.GetCustomAttribute<MDA>());
19        Console.WriteLine(paramInfo.GetCustomAttribute<MDA>());
20        Console.WriteLine(typeParamInfo.GetCustomAttribute<MDA>());
21    }
22 }
```

```
1 CallerNameInAttribute.cs:36 - Unspecified member
2 CallerNameInAttribute.cs:39 - Method
3 CallerNameInAttribute.cs:40 - Method
4 CallerNameInAttribute.cs:40 - Method
```

---

# CHAPTER 8

## Listing 8.1. `Point` class with public fields

```
1 public sealed class Point
2 {
3     public double X;
4     public double Y;
5 }
```

---

## Listing 8.2. `Point` class with properties in C# 1

```
1 public sealed class Point
2 {
3     private double x, y;
4     public double X { get { return x; } set { x = value; } }
5     public double Y { get { return y; } set { y = value; } }
6 }
```

---

## Listing 8.3. `Point` class with properties in C# 3

```
1 public sealed class Point
2 {
3     public double X { get; set; }
4     public double Y { get; set; }
5 }
```

---

## Listing 8.4. `Point` class with read-only properties via manual implementation in C# 3

```
1 public sealed class Point
2 {
3     private readonly double x, y;
4     public double X { get { return x; } }
5     public double Y { get { return y; } }
6
7     public Point(double x, double y)
8     {
9         this.x = x;
```

1

2

```
10     this.y = y;
11 }
12 }
```

3

### Listing 8.5. `Point` class with publicly read-only properties via automatic implementation with private setters in C# 3

```
1 public sealed class Point
2 {
3     public double X { get; private set; }
4     public double Y { get; private set; }
5
6     public Point(double x, double y)
7     {
8         X = x;
9         Y = y;
10    }
11 }
```

### Listing 8.6. `Point` class using read-only automatically implemented properties

```
1 public sealed class Point
2 {
3     public double X { get; }
4     public double Y { get; }
5
6     public Point(double x, double y)
7     {
8         X = x;
9         Y = y;
10    }
11 }
```

1

2

### Listing 8.7. `Person` class with manual property in C# 2

```
1 public class Person
2 {
3     private List<Person> friends = new List<Person>();
4     public List<Person> Friends
5     {
6         get { return friends; }
7         set { friends = value; }
8     }
9 }
```

1

2



### Listing 8.8. `Person` class with automatically implemented property in C# 3

```
1 public class Person
2 {
3     public List<Person> Friends { get; set; }
4
5     public Person()
6     {
7         Friends = new List<Person>();
8     }
9 }
```

1

2

### Listing 8.9. `Person` class with automatically implemented read/write property in C# 6

```
1 public class Person
2 {
3     public List<Person> Friends { get; set; } =
4         new List<Person>();
5 }
```

1

### Listing 8.10. `Person` class with automatically implemented read-only property in C# 6

```
1 public class Person
2 {
3     public List<Person> Friends { get; } =
4         new List<Person>();
5 }
```

1

### Listing 8.11. `Point` struct in C# 5 using automatically implemented properties

```
1 public struct Point
2 {
3     public double X { get; private set; }
4     public double Y { get; private set; }
5
6     public Point(double x, double y) : this()
7     {
8         X = x;
```

1

2

```
9         Y = y;  
10     }  
11 }
```

3

### Listing 8.12. `Point` struct in C# 6 using automatically implemented properties

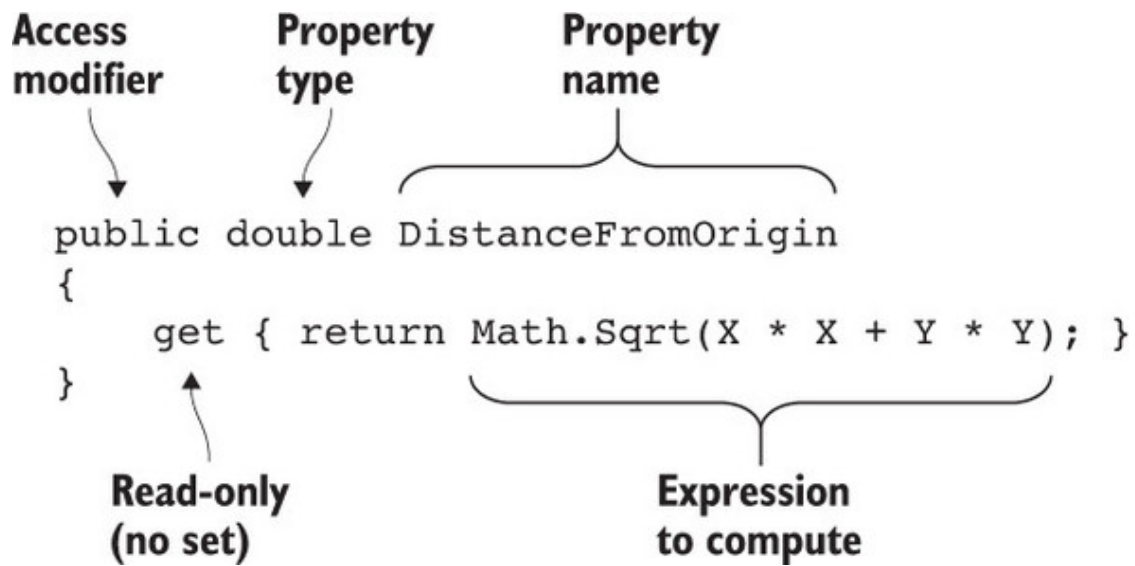
```
1 public struct Point  
2 {  
3     public double X { get; }  
4     public double Y { get; }  
5  
6     public Point(double x, double y)  
7     {  
8         X = x;  
9         Y = y;  
10    }  
11 }
```

### Listing 8.13. Adding a `DistanceFromOrigin` property to `Point`

```
1 public sealed class Point  
2 {  
3     public double X { get; }  
4     public double Y { get; }  
5  
6     public Point(double x, double y)  
7     {  
8         X = x;  
9         Y = y;  
10    }  
11  
12     public double DistanceFromOrigin  
13     {  
14         get { return Math.Sqrt(X * X + Y * Y); }  
15     }  
16 }
```

1

Figure 8.1. Annotated property declaration showing important aspects



```
1 public double DistanceFromOrigin => Math.Sqrt(X * X + Y * Y);
```

## NO, THIS ISN'T A LAMBDA EXPRESSION

Yes, you've seen this element of syntax before. Lambda expressions were introduced in C# 3 as a brief way of declaring delegates and expression trees. For example:

```
1 Func<string, int> stringLength = text => text.Length;
```

Expression-bodied members use the `=>` syntax but aren't lambda expressions. The preceding declaration of `DistanceFromOrigin` doesn't involve any delegates or expression trees; it only instructs the compiler to create a read-only property that computes the given expression and returns the result.

When talking about the syntax out loud, I usually describe `=>` as a *fat arrow*.

## Listing 8.14. Delegating properties in Noda Time

```
1 public struct LocalDateTime
2 {
3     public LocalDate Date { get; }
4     public int Year => Date.Year;
5     public int Month => Date.Month;
6     public int Day => Date.Day;
7
8     public LocalTime TimeOfDay { get; }
9     public int Hour => TimeOfDay.Hour;
10    public int Minute => TimeOfDay.Minute;
11    public int Second => TimeOfDay.Second;
12
13 }
14 }
```

1  
2  
3  
4  
5

```
1 public int NanosecondOfSecond =>
2     (int) (NanosecondOfDay % NodaConstants.NanosecondsPerSecond);
```

### IMPORTANT CAVEAT

Expression-bodied properties have one downside: there's only a single-character difference between a read-only property and a public read/write field. In most cases, if you make a mistake, a compile-time error will occur, due to using other properties or fields within a field initializer, but for static properties or properties returning a constant value, it's an easy mistake to make. Consider the difference between the following declarations:

```
1 // Declares a read-only property
2 public int Foo => 0;
3 // Declares a read/write public field
4 public int Foo = 0;
```

This has been a problem for me a couple of times, but after you're aware of it, checking for it is easy enough. Make sure your code reviewers are aware of it, too, and you're unlikely to get caught.

### Listing 8.15. Simple methods and operators in C# 5

```
1 public static Point Add(Point left, Vector right)
2 {
3     return left + right;
4 }
5
6 public static Point operator+(Point left, Vector right)
7 {
8     return new Point(left.X + right.X,
9         left.Y + right.Y);
10 }
```

1

2

### Listing 8.16. Expression-bodied methods and operators in C# 6

```
1 public static Point Add(Point left, Vector right) => left + right;
2
3 public static Point operator+(Point left, Vector right) =>
4     new Point(left.X + right.X, left.Y + right.Y);
```

```
1 public static void Log(string text)
2 {
3     Console.WriteLine("{0:o}: {1}", DateTime.UtcNow, text)
4 }
```

```
1 public static void Log(string text) =>
2     Console.WriteLine("{0:o}: {1}", DateTime.UtcNow, text);
```

### Listing 8.17. `IReadOnlyList<T>` implementation using expression-bodied members

```
1 public sealed class ReadOnlyListView<T> : IReadOnlyList<T>
2 {
3     private readonly IList<T> list;
4
5     public ReadOnlyListView(IList<T> list)
6     {
```

```

7      this.list = list;
8  }
9  public T this[int index] => list[index];
10 public int Count => list.Count;
11 public IEnumerator<T> GetEnumerator() =>
12     list.GetEnumerator();
13 IEnumerator IEnumerable.GetEnumerator() =>
14     GetEnumerator();
15 }

```

## Listing 8.18. Extra expression-bodied members in C# 7

```

1  public class Demo
2  {
3      static Demo() =>
4          Console.WriteLine("Static constructor called");
5      ~Demo() => Console.WriteLine("Finalizer called");
6
7      private string name;
8      private readonly int[] values = new int[10];
9
10     public Demo(string name) => this.name = name;
11
12     private PropertyChangedEventHandler handler;
13     public event PropertyChangedEventHandler PropertyChanged
14     {
15         add => handler += value;
16         remove => handler -= value;
17     }
18
19     public int this[int index]
20     {
21         get => values[index];
22         set => values[index] = value;
23     }
24
25     public string Name
26     {
27         get => name;
28         set => name = value;
29     }
30 }

```

```

1  public int this[int index]
2  {
3      get => values[index];
4      set
5      {
6          if (value < 0)
7          {
8              throw new ArgumentOutOfRangeException();
9          }
10         Values[index] = value;
11     }
12 }

```

```
1 public ZonedDateTime InZone(  
2     DateTimeZone zone,  
3     ZoneLocalMappingResolver resolver)  
4 {  
5     Preconditions.checkNotNull(zone);  
6     Preconditions.checkNotNull(resolver);  
7     return zone.ResolveLocal(this, resolver);  
8 }
```

---

```
1 public ZonedDateTime InZone(  
2     DateTimeZone zone,  
3     ZoneLocalMappingResolver resolver) =>  
4     Preconditions.checkNotNull(zone)  
5         .ResolveLocal(  
6             this,  
7             Preconditions.checkNotNull(resolver);
```

---



```
1 public int Minute  
2 {  
3     get  
4     {  
5         int minuteOfDay = (int) NanosecondOfDay / NanosecondsPerMinute;  
6         return minuteOfDay % MinutesPerHour;  
7     }  
8 }
```

---

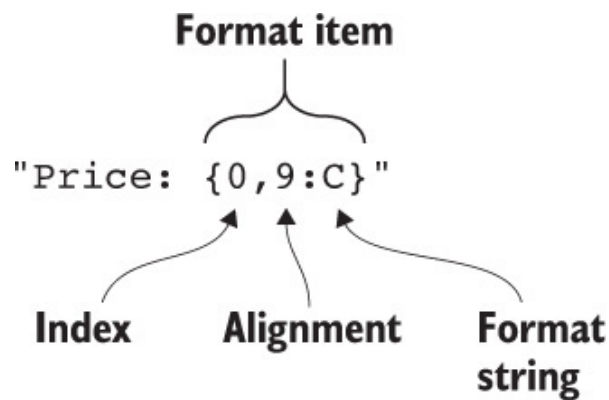
```
1 public int Minute =>  
2     ((int) NanosecondOfDay / NodaConstants.NanosecondsPerMinute) %  
3     NodaConstants.MinutesPerHour;
```

---

# CHAPTER 9

```
1 Console.Write("What's your name? ");
2 string name = Console.ReadLine();
3 Console.WriteLine("Hello, {0}!", name);
```

Figure 9.1. A composite format string with a format item to display a price



```
1 Price:    $95.25
2 Tip:      $19.05
3 Total:    $114.30
```

Listing 9.1. Displaying a price, tip, and total with values aligned

```
1 decimal price = 95.25m;
2 decimal tip = price * 0.2m;
3 Console.WriteLine("Price: {0,9:C}", price);
4 Console.WriteLine("Tip:    {0,9:C}", tip);
5 Console.WriteLine("Total:  {0,9:C}", price + tip);
```



```
1 static string Format(IFormatProvider provider,  
2     string format, params object[] args)  
3 static string Format(string format, params object[] args)
```

---

```
1 var usEnglish = CultureInfo.GetCultureInfo("en-US");  
2 var birthDate = new DateTime(1976, 6, 19);  
3 string formatted = string.Format(usEnglish, "Jon was born on {0:d}", birthDate);
```

---

## Listing 9.2. Formatting a single date in every culture

```
1 var cultures = CultureInfo.GetCultures(CultureTypes.AllCultures);  
2 var birthDate = new DateTime(1976, 6, 19);  
3 foreach (var culture in cultures)  
4 {  
5     string text = string.Format(  
6         culture, "{0,-15} {1,12:d}", culture.Name, birthDate);  
7     Console.WriteLine(text);  
8 }
```

---

```
1 ...  
2 tg-Cyrl          19.06.1976  
3 tg-Cyrl-TJ      19.06.1976  
4 th              19/6/2519  
5 th-TH           19/6/2519  
6 ti              19/06/1976  
7 ti-ER           19/06/1976  
8 ...  
9 ur-PK           19/06/1976  
10 uz              19/06/1976  
11 uz-Arab         29/03 1355  
12 uz-Arab-AF      29/03 1355  
13 uz-Cyrl         19/06/1976  
14 uz-Cyrl-UZ      19/06/1976  
15 ...
```

---

```
1 string url = string.Format(  
2     CultureInfo.InvariantCulture,  
3     "{0}?date={1:yyyy-MM-dd}",  
4
```

```
5    webServiceBaseUrl,
    searchDate);
```

## C# 5—old-style style formatting

```
Console.Write("What's your name? ");
string name = Console.ReadLine();
Console.WriteLine("Hello, {0}!",
    name);
```

## C# 6—interpolated string literal

```
Console.Write("What's your name? ");
string name = Console.ReadLine();
Console.WriteLine($"Hello, {name}!");
```

### Listing 9.3. Aligned values using interpolated string literals

```
1 decimal price = 95.25m;
2 decimal tip = price * 0.2m;
3 Console.WriteLine($"Price: {price,9:C}");
4 Console.WriteLine($"Tip: {tip,9:C}");
5 Console.WriteLine($"Total: {price + tip,9:C}");
```

```
1 string sql = @"
2     SELECT City, ZipCode
3     FROM Address
4     WHERE Country = 'US'";
5 Regex lettersDotDigits = new Regex(@"[a-z]+\.\d+");
6 string file = @"c:\users\skeet\Test\Test.cs"
```

### Listing 9.4. Aligned values using a single interpolated verbatim string literal

```
1 decimal price = 95.25m;
2 decimal tip = price * 0.2m;
3 Console.WriteLine($"@Price: {price,9:C}
4 Tip: {tip,9:C}
5 Total: {price + tip,9:C}");
```

```
1 int x = 10;
2 int y = 20;
3
```

```
4 string text = $"x={x}, y={y}";  
  Console.WriteLine(text);
```

---

```
1 int x = 10;  
2 int y = 20;  
3 string text = string.Format("x={0}, y={1}", x, y);  
4 Console.WriteLine(text);
```

---

```
1 var compositeFormatString = "Jon was born on {0:d}";  
2 var value = new DateTime(1976, 6, 19);  
3 var culture = CultureInfo.GetCultureInfo("en-US");  
4 var result = string.Format(culture, compositeFormatString, value);
```

---

```
1 var dateOfBirth = new DateTime(1976, 6, 19);  
2 FormattableString formattableString =  
3     $"Jon was born on {dateOfBirth:d}";  
4 var culture = CultureInfo.GetCultureInfo("en-US");  
5 var result = formattableString.ToString(culture);
```

---

1

2

```
1 int x = 10;  
2 int y = 20;  
3 FormattableString formattable = $"x={x}, y={y}";
```

---

```
1 int x = 10;  
2 int y = 20;  
3 FormattableString formattable = FormattableStringFactory.Create(  
4     "x={0}, y={1}", x, y);
```

---

## Listing 9.5. Members declared by `FormattableString`

```
1 public abstract class FormattableString : IFormattable
2 {
3     protected FormattableString();
4     public abstract object GetArgument(int index);
5     public abstract object[] GetArguments();
6     public static string Invariant(FormattableString formattable);
7     string IFormattable.ToString
8         (string ignored, IFormatProvider formatProvider);
9     public override string ToString();
10    public abstract string ToString(IFormatProvider formatProvider);
11    public abstract int ArgumentCount { get; }
12    public abstract string Format { get; }
13 }
```

## Listing 9.6. Formatting a date in the invariant culture

```
1 DateTime date = DateTime.UtcNow;
2
3 string parameter1 = string.Format(
4     CultureInfo.InvariantCulture,
5     "x={0:yyyy-MM-dd}",
6     date);
7
8 string parameter2 =
9     ((FormattableString)$"x={date:yyyy-MM-dd}")
10    .ToString(CultureInfo.InvariantCulture);
11
12 string parameter3 = FormattableString.Invariant(
13     $"x={date:yyyy-MM-dd}");
14
15 string parameter4 = Invariant($"x={date:yyyy-MM-dd}");
```

1

2

3

4

```
1 FormattableString fs = $"The current date and time is: {DateTime.Now:g}";
2 string formatted = fs.ToString(CultureInfo.GetCultureInfo("en-US"));
```

## Listing 9.7. Awooga! Awooga! Do not use this code!

```
1 var tag = Console.ReadLine();
2 using (var conn = new SqlConnection(connectionString))
3 {
4     conn.Open();
5     string sql =
6         @"SELECT Description FROM Entries
7         WHERE Tag='{tag}' AND UserId={userId}";
```

1

```

8      using (var command = new SqlCommand(sql, conn))
9      {
10          using (var reader = command.ExecuteReader())
11          {
12              ...
13          }
14      }
15 }

```

## Listing 9.8. Safe SQL parameterization using `FormattableString`

```

1  var tag = Console.ReadLine();
2  using (var conn = new SqlConnection(connectionString))
3  {
4      conn.Open();
5      using (var command = conn.NewSqlCommand(
6          @"SELECT Description FROM Entries
7          WHERE Tag={tag:NVarChar}
8          AND UserId={userId:Int}")
9      {
10         using (var reader = command.ExecuteReader())
11         {
12             // Use the data
13         }
14     }
15 }

```

```

1 SELECT Description FROM Entries
2 WHERE Tag={0:NVarChar} AND UserId={1:Int}

```

```

1 SELECT Description FROM Entries
2 WHERE Tag=@p0 AND UserId=@p1

```

## Listing 9.9. Implementing safe SQL formatting

```

1  public static class SqlFormattableString
2  {
3      public static SqlCommand NewSqlCommand(
4          this SqlConnection conn, FormattableString formattableString)
5      {

```

```

6         SqlParameter[] sqlParameters = formattableString.GetArguments()
7         .Select((value, position) =>
8             new SqlParameter(Invariant($"@p{position}"), value))
9         .ToArray();
10        object[] formatArguments = sqlParameters
11        .Select(p => new FormatCapturingParameter(p))
12        .ToArray();
13        string sql = string.Format(formattableString.Format,
14            formatArguments);
15        var command = new SqlCommand(sql, conn);
16        command.Parameters.AddRange(sqlParameters);
17        return command;
18    }
19
20    private class FormatCapturingParameter : IFormattable
21    {
22        private readonly SqlParameter parameter;
23
24        internal FormatCapturingParameter(SqlParameter parameter)
25        {
26            this.parameter = parameter;
27        }
28        public string ToString(string format, IFormatProvider formatProvider)
29        {
30            if (!string.IsNullOrEmpty(format))
31            {
32                parameter.SqlDbType = (SqlDbType) Enum.Parse(
33                    typeof(SqlDbType), format, true);
34            }
35            return parameter.ParameterName;
36        }
37    }
38 }

```

---

## Listing 9.10. Implementing `FormattableString` from scratch

```

1 using System.Globalization;
2
3 namespace System.Runtime.CompilerServices
4 {
5     internal static class FormattableStringFactory
6     {
7         internal static FormattableString Create(
8             string format, params object[] arguments) =>
9             new FormattableString(format, arguments);
10    }
11 }
12
13 namespace System
14 {
15     internal class FormattableString : IFormattable
16     {
17         public string Format { get; }
18         private readonly object[] arguments;
19
20         internal FormattableString(string format, object[] arguments)
21         {
22             Format = format;
23             this.arguments = arguments;
24         }
25
26         public object GetArgument(int index) => arguments[index];

```

```

27     public object[] GetArguments() => arguments;
28     public int ArgumentCount => arguments.Length;
29     public static string Invariant(FormattableString formattable) =>
30         formattable?.ToString(CultureInfo.InvariantCulture);
31     public string ToString(IFormatProvider formatProvider) =>
32         string.Format(formatProvider, Format, arguments);
33     public string ToString(
34         string ignored, IFormatProvider formatProvider) =>
35         ToString(formatProvider);
36     }
37 }

```

---

```

1 throw new ArgumentException(Invariant(
2     $"Start date {start:yyyy-MM-dd} should not be earlier than year 2000."))

```

---

```

1 Console.WriteLine($"Price: {price,9:C}");

```

---

```

1 int alignment = GetAlignmentFromValues(allTheValues);
2 Console.WriteLine($"Price: {{0,{alignment}:C}}", price);

```

---

### Listing 9.11. Even `FormattableString` evaluates expressions eagerly

```

1 string value = "Before";
2 FormattableString formattable = $"Current value: {value}";
3 Console.WriteLine(formattable);
4
5 value = "After";
6 Console.WriteLine(formattable);

```

---

1

2

```

1 Console.WriteLine($"Adult? {age >= 18 ? "Yes" : "No"}");

```

---

```
1 Console.WriteLine($"Adult? {(age >= 18 ? "Yes" : "No")}");
```

---

```
1 Preconditions.CheckArgument(  
2     start.Year < 2000,  
3     Invariant($"Start date {start:yyyy-MM-dd} should not be earlier than year  
4     2000."));
```

---

```
1 Logger.Debug("Received request with {0} bytes", request.Length);
```

---

```
1 Logger.Debug($"Received request with {request.Length} bytes");
```

---

```
1 private static string FormatMemberDebugName(MemberInfo m) =>  
2     string.Format("{0}.{1}({2})",  
3         m.DeclaringType.Name,  
4         m.Name,  
5         string.Join(", ", GetParameters(m).Select(p => p.ParameterType)));
```

---

```
1 Console.Write("What's your name? ");  
2 string name = Console.ReadLine();  
3 Console.WriteLine("Hello, {0}!", name);
```

---



```
1 Console.WriteLine($"Hello {((Func<string>))() =>
2 {
3     Console.Write("What's your name? ");
4     return Console.ReadLine();
5 })))()!"");
```

---

### Listing 9.12. Printing out the names of a class, method, field, and parameter

```
1 using System;
2
3 class SimpleNameof
4 {
5     private string field;
6
7     static void Main(string[] args)
8     {
9         Console.WriteLine(nameof(SimpleNameof));
10        Console.WriteLine(nameof(Main));
11        Console.WriteLine(nameof(args));
12        Console.WriteLine(nameof(field));
13    }
14 }
```

---

```
1 SimpleNameof
2 Main
3 args
4 field
```

---

### Listing 9.13. A simple method using its parameter twice in the body

```
1 static void RenameDemo(string oldName)
2 {
3     Console.WriteLine($"{nameof(oldName)} = {oldName}");
4 }
```

---

Figure 9.2. Renaming an identifier in Visual Studio

**Rename: oldName** X

New name: newName

☐ Include comments

☐ Include strings

☐ Preview changes

Rename will update 3 references in 1 file.

Apply

```
static void RenameDemo(string newName)
{
    Console.WriteLine($"{nameof(newName)} = {newName}");
}
```

```
1 public ZonedDateTime InZone(
2     DateTimeZone zone,
3     ZoneLocalMappingResolver resolver)
4 {
5     Preconditions.CheckNotNull(zone, nameof(zone));
6     Preconditions.CheckNotNull(resolver, nameof(resolver));
7     return zone.ResolveLocal(this, resolver);
8 }
```

### Listing 9.14. Using `nameof` to raise a property change notification

```
1 public class Rectangle : INotifyPropertyChanged
2 {
3     public event PropertyChangedEventHandler PropertyChanged;
4
5     private double width;
6     private double height;
7
8     public double Width
9     {
10         get { return width; }
11         set
12         {
13             if (width == value)
14             {
15                 return;
16             }
17             width = value;
18             RaisePropertyChanged();
19             RaisePropertyChanged(nameof(Area));
20         }
21     }
22
23     public double Height { ... }
```

1

2

3

```

24
25     public double Area => Width * Height; 4
26
27     private void RaisePropertyChanged(
28         [CallerMemberName] string propertyName = null) { ... } 5
29
30 } 6

```

---

### Listing 9.15. Specifying a test case source with `nameof`

```

1 static readonly IEnumerable<DateTimeZone> AllZones = 1
2     DateTimeZoneProviders.Tzdb.GetAllZones();
3
4 [Test]
5 [TestCaseSource(nameof(AllZones))]
6 public void AllZonesStartAndEnd(DateTimeZone zone) 2
7 { 3
8     ...
9 } 4

```

---

```

1 [DerivedProperty(nameof(Area))]
2 public double Width { ... }

```

---

```

1 public class Employee
2 {
3     [ForeignKey(nameof(Employer))]
4     public Guid EmployerId { get; set; }
5     public Company Employer { get; set; }
6 }

```

---

```

1 [TestCaseSource(typeof(Cultures), nameof(Cultures.AllCultures))]

```

---

```

1 [TestSource(typeof(Cultures), "AllCultures")]

```

---

### Listing 9.16. All the valid ways of accessing names of members in other types

```
1 class OtherClass
2 {
3     public static int StaticMember => 3;
4     public int InstanceMember => 3;
5 }
6
7 class QualifiedNameof
8 {
9     static void Main()
10    {
11        OtherClass instance = null;
12        Console.WriteLine(nameof(instance.InstanceMember));
13        Console.WriteLine(nameof(OtherClass.StaticMember));
14        Console.WriteLine(nameof(OtherClass.InstanceMember));
15    }
16 }
```

```
1 static string Method<T>() => nameof(T);
```

1

### Listing 9.17. Using an alias in the `nameof` operator

```
1 using System;
2
3 using GuidAlias = System.Guid;
4
5 class Test
6 {
7     static void Main()
8     {
9         Console.WriteLine(nameof(GuidAlias));
10    }
11 }
```

```
1 nameof(float)
2 nameof(Guid?)
3 nameof(String[])
```

1

2

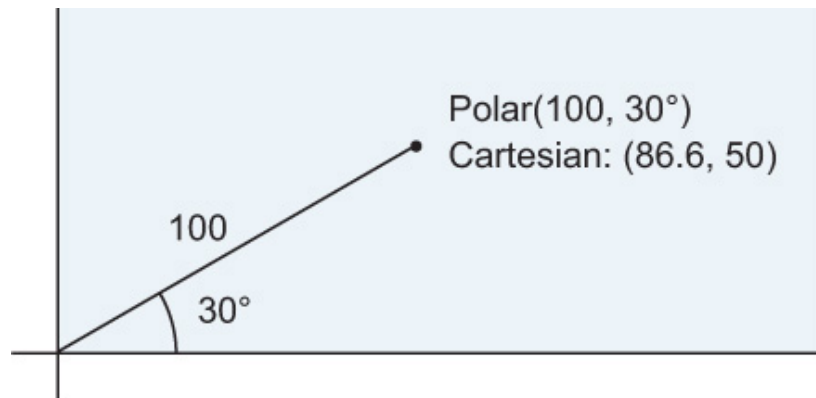
3

```
1 nameof(Single)
2 nameof(Nullable<Guid>)
```

---

# CHAPTER 10

Figure 10.1. An example of polar and Cartesian coordinates



Listing 10.1. Polar-to-Cartesian conversion in C# 5

```
1 using System;
2 ...
3 static Point PolarToCartesian(double degrees, double magnitude)
4 {
5     double radians = degrees * Math.PI / 180;
6     return new Point(
7         Math.Cos(radians) * magnitude,
8         Math.Sin(radians) * magnitude);
9 }
```

1  
2

Listing 10.2. Polar-to-Cartesian conversion in C# 6

```
1 using static System.Math;
2 ...
3 static Point PolarToCartesian(double degrees, double magnitude)
4 {
5     double radians = degrees * PI / 180;
6     return new Point(
7         Cos(radians) * magnitude,
8         Sin(radians) * magnitude);
9 }
```

1  
2

```
1 using static type-name-or-alias;
```

C# 5 code

```
using System.Reflection;
...
var fields = type.GetFields(
BindingFlags.Instance |
BindingFlags.Static |
BindingFlags.Public |
BindingFlags.NonPublic)
```

With using static in C# 6

```
using static System.Reflection.BindingFlags;
...
var fields = type.GetFields(
Instance | Static | Public | NonPublic);
```

C# 5 code

```
using System.Net;
...
switch (response.StatusCode)
{
case HttpStatusCode.OK:
...
case HttpStatusCode.TemporaryRedirect:
case HttpStatusCode.Redirect:
case HttpStatusCode.RedirectMethod:
...
case HttpStatusCode.NotFound:
...
default:
...
}
```

With using static in C# 6

```
}
using static
System.Net.HttpStatusCode;
...
switch (response.StatusCode)
{
case OK:
...
case TemporaryRedirect:
case Redirect:
case RedirectMethod:
...
case NotFound:
...
default:
...
}
```

```
1 message Outer {
2   message Inner {
3     string text = 1;
4   }
5
6   Inner inner = 1;
7 }
```

```
1 public class Outer
2 {
3     public static class Types
4     {
5         public class Inner
6         {
7             public string Text { get; set; }
8         }
9     }
10
11     public Types.Inner Inner { get; set; }
12 }
```

---

```
1 using static Outer.Types;
2 ...
3 Outer outer = new Outer { Inner = new Inner { Text = "Some text here" } };
```

---

## THE IMPORTED TYPE DOESN'T HAVE TO BE STATIC

The `static` part of `using static` doesn't mean that the type you import must be static. The examples shown so far have been, but you can import regular types, too. That lets you access the static members of those types without qualification:

```
1 using static System.String;
2 ...
3 string[] elements = { "a", "b" };
4 Console.WriteLine(Join(" ", elements));    1
```

---

- **1 Access String.Join by its simple name**

I haven't found this to be as useful as the earlier examples, but it's available if you want it. Any nested types are made available by their simple names, too. There's one exception to the set of static members



that's imported with a `using static` directive that isn't quite so straightforward, and that's extension methods.

### Listing 10.3. Selective importing of extension methods

```
1 using static System.Linq.Queryable;
2 ...
3 var query = new[] { "a", "bc", "d" }.AsQueryable();
4
5 Expression<Func<string, bool>> expr =
6     x => x.Length > 1;
7 Func<string, bool> del = x => x.Length > 1;
8
9 var valid = query.Where(expr);
10 var invalid = query.Where(del);
```

1  
2  
3  
4

### Listing 10.4. Attempting to call `Enumerable.Count` in two ways

```
1 using System.Collections.Generic;
2 using static System.Linq.Enumerable;
3 ...
4 IEnumerable<string> strings = new[] { "a", "b", "c" };
5
6 int valid = strings.Count();
7 int invalid = Count(strings);
```

1  
2

```
1 Button button = new Button { Text = "Go", BackColor = Color.Red };
2 List<int> numbers = new List<int> { 5, 10, 20 };
```

```
1 string text = "This text needs truncating";
2 StringBuilder builder = new StringBuilder(text)
3 {
4     Length = 10
5 };
6 builder[9] = '\u2026';
7 Console.OutputEncoding = Encoding.UTF8;
8 Console.WriteLine(builder);
```

1  
2  
3  
4

## Listing 10.5. Using an indexer in a `StringBuilder` object initializer

```
1 string text = "This text needs truncating";
2 StringBuilder builder = new StringBuilder(text)
3 {
4     Length = 10,
5     [9] = '\u2026'
6 };
7 Console.OutputEncoding = Encoding.UTF8;
8 Console.WriteLine(builder);
```

1  
2  
3  
4

## Listing 10.6. Two ways of initializing a dictionary

```
1 var collectionInitializer = new Dictionary<string, int>
2 {
3     { "A", 20 },
4     { "B", 30 },
5     { "B", 40 }
6 };
7
8 var objectInitializer = new Dictionary<string, int>
9 {
10     ["A"] = 20,
11     ["B"] = 30,
12     ["B"] = 40
13 };
```

1  
2

## Listing 10.7. A schemaless entity type with key properties

```
1 public sealed class SchemalessEntity
2     : IEnumerable<KeyValuePair<string, object>>
3 {
4     private readonly IDictionary<string, object> properties =
5         new Dictionary<string, object>();
6
7     public string Key { get; set; }
8     public string ParentKey { get; set; }
9
10    public object this[string propertyKey]
11    {
12        get { return properties[propertyKey]; }
13        set { properties[propertyKey] = value; }
14    }
15
16    public void Add(string propertyKey, object value)
17    {
18        properties.Add(propertyKey, value);
19    }
20
21    public IEnumerator<KeyValuePair<string, object>> GetEnumerator() =>
22        properties.GetEnumerator();
23
24 }
```

```
25     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();  
    }
```

---

### Listing 10.8. Two ways of initializing a SchemalessEntity

```
1 SchemalessEntity parent = new SchemalessEntity { Key = "parent-key" };  
2 SchemalessEntity child1 = new SchemalessEntity  
3 {  
4     { "name", "Jon Skeet" },  
5     { "location", "Reading, UK" }  
6 };  
7 child1.Key = "child-key";  
8 child1.ParentKey = parent.Key;  
9  
10 SchemalessEntity child2 = new SchemalessEntity  
11 {  
12     Key = "child-key",  
13     ParentKey = parent.Key,  
14     ["name"] = "Jon Skeet",  
15     ["location"] = "Reading, UK"  
16 };
```

1  
2  
3  
4

---

```
1 List<string> strings = new List<string>  
2 {  
3     10,  
4     "hello",  
5     { 20, 3 }  
6 };
```

---

```
1 List<string> strings = new List<string>();  
2 strings.Add(10);  
3 strings.Add("hello");  
4 strings.Add(20, 3);
```

---

```
1 public static class StringListExtensions  
2 {  
3     public static void Add(  
4         this List<string> list, int value, int count = 1)  
5     {  
6         list.AddRange(Enumerable.Repeat(value.ToString(), count));  
7     }  
8 }
```

```
7     }  
8 }
```

---

```
1 Person jon = new Person  
2 {  
3     Name = "Jon",  
4     Contacts = { allContacts.Where(c => c.Town == "Reading") }  
5 };
```

---

### Listing 10.9. Exposing explicit interface implementations via extension methods

```
1 static class ListExtensions  
2 {  
3     public static void Add<T>(this List<T> list, IEnumerable<T> collection)  
4     {  
5         list.AddRange(collection);  
6     }  
7 }
```

---

### Listing 10.10. Adding a type-argument-specific `Add` method for dictionaries

```
1 static class PersonDictionaryExtensions  
2 {  
3     public static void Add(  
4         this Dictionary<string, Person> dictionary, Person person)  
5     {  
6         dictionary.Add(person.Name, person);  
7     }  
8 }
```

---

```
1 var dictionary = new Dictionary<string, Person>  
2 {  
3     { new Person { Name = "Jon" } },  
4     { new Person { Name = "Holly" } }  
5 };
```

---

## Listing 10.11. Exposing explicit interface implementations via extension methods

```
1 public static class DictionaryExtensions
2 {
3     public static void Add<TKey, TValue>(
4         this IDictionary<TKey, TValue> dictionary,
5         TKey key, TValue value)
6     {
7         dictionary.Add(key, value);
8     }
9 }
```

---

```
1 var dictionary = new ConcurrentDictionary<string, int>
2 {
3     { "x", 10 },
4     { "y", 20 }
5 };
```

---

```
1 var readingCustomers = allCustomers
2     .Where(c => c.Profile.DefaultShippingAddress.Town == "Reading");
```

---

```
1 var readingCustomers = allCustomers
2     .Where(c => c.Profile != null &&
3         c.Profile.DefaultShippingAddress != null &&
4         c.Profile.DefaultShippingAddress.Town == "Reading");
```

---

```
1 var readingCustomers = allCustomers
2     .Where(c => c.Profile?.DefaultShippingAddress?.Town == "Reading");
```

---

```
1 c => c.Profile?.DefaultShippingAddress?.Town == "Reading"
```

---

```

1 string result;
2 var tmp1 = c.Profile;
3 if (tmp1 == null)
4 {
5     result = null;
6 }
7 else
8 {
9     var tmp2 = tmp1.DefaultShippingAddress;
10    if (tmp2 == null)
11    {
12        result = null;
13    }
14    else
15    {
16        result = tmp2.Town;
17    }
18 }
19 return result == "Reading";

```

```

1 c => c.Profile?.DefaultShippingAddress?.Town?.Equals("Reading")

```

**Table 10.1. Options for performing Boolean comparisons using the null conditional operator (view table figure)**

You don't want to enter the body if name is null	You do want to enter the body if name is null
if (name?.Equals("X") ?? false)	if (name?.Equals("X") == true)
if (name?.Equals("X") ?? true)	if (name?.Equals("X") != false)

```

1 int[] array = null;
2 int? firstElement = array?[0];

```

```
1 EventHandler handler = Click;
2 if (handler != null)
3 {
4     handler(this, EventArgs.Empty);
5 }
```

---

```
1 Click?.Invoke(this, EventArgs.Empty);
```

---

### Listing 10.12. Sketch of a null-conditional-friendly logging API

```
1 public interface ILogger 1
2 {
3     IActiveLogger Debug { get; }
4     IActiveLogger Info { get; }
5     IActiveLogger Warning { get; } 2
6     IActiveLogger Error { get; }
7 }
8
9 public interface IActiveLogger 3
10 {
11     void Log(string message);
12 }
```

---

```
1 logger.Debug?.Log($"Received request for URL {request.Url}");
```

---

```
1 string authorName = book.Element("author")?.Attribute("name")?.Value;
2 string authorName = (string) book.Element("author")?.Attribute("name");
```

---

```
1 person?.Name = "";
2 stats?.RequestCount++;
3 array?[index] = 10;
```

---

```

1 try
2 {
3     ...
4 }
5 catch (WebException e)
6 {
7     if (e.Status != WebExceptionStatus.ConnectFailure)
8     {
9         throw;
10    }
11    ...
12 }

```

1

2

3

```

1 try
2 {
3     ...
4 }
5 catch (WebException e)
6     when (e.Status == WebExceptionStatus.ConnectFailure)
7 {
8     ...
9 }

```

1

2

3

### Listing 10.13. Throwing three exceptions and catching two of them

```

1 string[] messages =
2 {
3     "You can catch this",
4     "You can catch this too",
5     "This won't be caught"
6 };
7 foreach (string message in messages)
8 {
9     try
10    {
11        throw new Exception(message);
12    }
13    catch (Exception e)
14        when (e.Message.Contains("catch"))
15    {
16        Console.WriteLine($"Caught '{e.Message}');
17    }
18 }

```

1

2

3

4



```
1 Caught 'You can catch this'
2 Caught 'You can catch this too'
```

---

### Listing 10.14. A three-level demonstration of exception filtering

```
1 static bool LogAndReturn(string message, bool result)
2 {
3     Console.WriteLine(message);
4     return result;
5 }
6
7 static void Top()
8 {
9     try
10    {
11        throw new Exception();
12    }
13    finally
14    {
15        Console.WriteLine("Top finally");
16    }
17 }
18
19 static void Middle()
20 {
21     try
22     {
23         Top();
24     }
25     catch (Exception e)
26         when (LogAndReturn("Middle filter", false))
27     {
28         Console.WriteLine("Caught in middle");
29     }
30     finally
31     {
32         Console.WriteLine("Middle finally");
33     }
34 }
35
36 static void Bottom()
37 {
38     try
39     {
40         Middle();
41     }
42     catch (IOException e)
43         when (LogAndReturn("Never called", true))
44     {
45     }
46     catch (Exception e)
47         when (LogAndReturn("Bottom filter", true))
48     {
49         Console.WriteLine("Caught in Bottom");
50     }
51 }
52
53 static void Main()
```

1

2

3

4

5

6

```
54 {  
55     Bottom();  
56 }
```



---

8

```
1 Middle filter  
2 Bottom filter  
3 Top finally  
4 Middle finally  
5 Caught in Bottom
```

---

**Figure 10.2.** Execution flow of [listing 10.14](#)

Stack	Explanation of progress	Output
<div>Top</div> <div>Middle</div> <div>Bottom</div>	<p>Bang!</p> <p>Top throws exception. First pass starts.</p>	
<div>Top</div> <div>Middle</div> <div>Bottom</div>	 <p>Walking down the stack: exception filter in middle evaluated. It returns false, so keep going.</p>	Middle filter
<div>Top</div> <div>Middle</div> <div>Bottom</div>	 <p>Walking down the stack: exception filter in bottom evaluated. It returns true, so first pass is complete! Second pass starts.</p>	Bottom filter
<div>Top</div> <div>Middle</div> <div>Bottom</div>	<p>Finally block in top executes. Stack unwinds.</p>	Top finally
<div>Middle</div> <div>Bottom</div>	<p>Finally block in middle executes. Stack unwinds.</p>	Middle finally
<div>Bottom</div>	<p>Catch block in bottom executes. Finished!</p>	Caught in bottom

```

1 try
2 {
3     ...
4 }
5 catch (WebException e)
6     when (e.Status == WebExceptionStatus.ConnectFailure)
7 {
8     ...
9 }
10 catch (WebException e)
11     when (e.Status == WebExceptionStatus.NameResolutionFailure)

```

1

2

```
12 {  
13     ...  
14 }
```

3

### Listing 10.15. A simple retry loop

```
1 static T Retry<T>(Func<T> operation, int attempts)  
2 {  
3     while (true)  
4     {  
5         try  
6         {  
7             attempts--;  
8             return operation();  
9         }  
10        catch (Exception e) when (attempts > 0)  
11        {  
12            Console.WriteLine($"Failed: {e}");  
13            Console.WriteLine($"Attempts left: {attempts}");  
14            Thread.Sleep(5000);  
15        }  
16    }  
17 }
```

```
1 Func<DateTime> temporamentalCall = () =>  
2 {  
3     DateTime utcNow = DateTime.UtcNow;  
4     if (utcNow.Second < 20)  
5     {  
6         throw new Exception("I don't like the start of a minute");  
7     }  
8     return utcNow;  
9 };  
10  
11 var result = Retry(temporamentalCall, 3);  
12 Console.WriteLine(result);
```

### Listing 10.16. Logging in a filter

```
1 static void Main()  
2 {  
3     try  
4     {  
5         UnreliableMethod();  
6     }  
7     catch (Exception e) when (Log(e))  
8     {
```

```
9     }
10 }
11
12 static void UnreliableMethod()
13 {
14     throw new Exception("Bang!");
15 }
16
17 static bool Log(Exception e)
18 {
19     Console.WriteLine($"{DateTime.UtcNow}: {e.GetType()} {e.Message}");
20     return false;
21 }
```

---

```
1 catch (IOException e)
2 {
3     ...
4 }
```

---

```
1 catch (Exception tmp) when (tmp is IOException)
2 {
3     IOException e = (IOException) tmp;
4     ...
5 }
```

---

```
1 catch (Exception e) when (condition)
2 {
3     ...
4 }
```

---

```
1 catch (Exception e)
2 {
3     if (!condition)
4     {
5         throw;
6     }
7     ...
8 }
```

---



# CHAPTER 11

```
1 static (int min, int max) MinMax(IEnumerable<int> source)
```

```
1 int[] values = { 2, 7, 3, -5, 1, 0, 10 };  
2 var extremes = MinMax(values);  
3 Console.WriteLine(extremes.min);  
4 Console.WriteLine(extremes.max);
```

1  
2  
3

Figure 11.1. A tuple literal with element values `5` and `"text"`. The second element is named `title`.

Unnamed element

(5, title: "text")

Named element

Figure 11.2. A tuple type with element types `int` and `Guid`. The first element is named `x`.

Unnamed element

(int x, Guid)

Named element

### Listing 11.1. Representing the minimum and maximum values of a sequence as a tuple

```
1 static (int min, int max) MinMax(                                     1
2     IEnumerable<int> source)
3 {
4     using (var iterator = source.GetEnumerator())
5     {
6         if (!iterator.MoveNext())                                     2
7         {
8             throw new InvalidOperationException(
9                 "Cannot find min/max of an empty sequence");
10        }
11        int min = iterator.Current;                                   3
12        int max = iterator.Current;
13        while (iterator.MoveNext())
14        {
15            min = Math.Min(min, iterator.Current);                   4
16            max = Math.Max(max, iterator.Current);
17        }                                                           5
18        return (min, max);
19    }
20 }
```

```
1 return (min, max);
```

```
1 var result = (min: min, max: max);
```

```
1 from emp in employees
2 join dept in departments on emp.DepartmentId equals dept.Id
3 select new { emp.Name, emp.Title, DepartmentName = dept.Name };
```

```
1 from emp in employees
2 join dept in departments on emp.DepartmentId equals dept.Id
3 select (name: emp.Name, title: emp.Title, departmentName: dept.Name);
```



```

1 from emp in employees
2 join dept in departments on emp.DepartmentId equals dept.Id
3 select (emp.Name, emp.Title, DepartmentName: dept.Name);

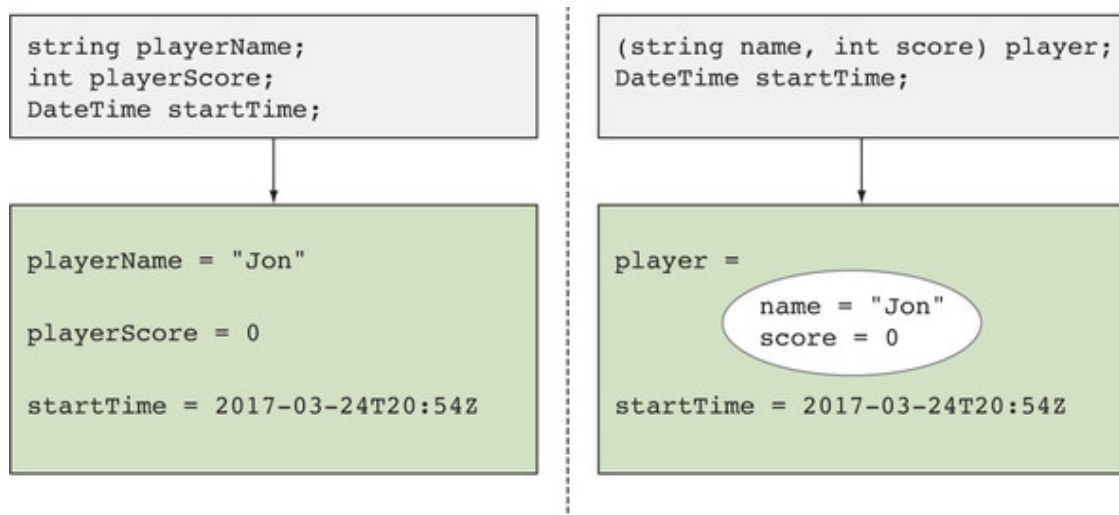
```

```

1 List<int> list = new List<int> { 5, 1, -6, 2 };
2 var tuple = (list.Count, Min: list.Min(), Max: list.Max());
3 Console.WriteLine(tuple.Count);
4 Console.WriteLine(tuple.Min);
5 Console.WriteLine(tuple.Max);

```

**Figure 11.3.** Three separate variables on the left; two variables, one of which is a tuple, on the right



**Listing 11.2.** Reading and writing tuple elements by name and position

```

1 var tuple = (x: 5, 10);
2 Console.WriteLine(tuple.x);
3 Console.WriteLine(tuple.Item1);
4 Console.WriteLine(tuple.Item2);
5 tuple.x = 100;
6 Console.WriteLine(tuple.Item1);

```

1  
2  
3  
4

### Listing 11.3. Using a tuple instead of two local variables in `MinMax`

```
1 static (int min, int max) MinMax(IEnumerable<int> source)
2 {
3     using (var iterator = source.GetEnumerator())
4     {
5         if (!iterator.MoveNext())
6         {
7             throw new InvalidOperationException(
8                 "Cannot find min/max of an empty sequence");
9         }
10        var result = (min: iterator.Current,
11                     max: iterator.Current);
12        while (iterator.MoveNext()) 1
13        {
14            result.min = Math.Min(result.min, iterator.Current);
15            result.max = Math.Max(result.max, iterator.Current);
16        } 2
17        return result;
18    }
19 } 3
```

```
1 result.min = Math.Min(result.min, iterator.Current);
2 result.max = Math.Max(result.max, iterator.Current);
```

### Listing 11.4. Reassigning the result tuple in one statement in `MinMax`

```
1 static (int min, int max) MinMax(IEnumerable<int> source)
2 {
3     using (var iterator = source.GetEnumerator())
4     {
5         if (!iterator.MoveNext())
6         {
7             throw new InvalidOperationException(
8                 "Cannot find min/max of an empty sequence");
9         }
10        var result = (min: iterator.Current, max: iterator.Current);
11        while (iterator.MoveNext())
12        {
13            result = (Math.Min(result.min, iterator.Current),
14                     Math.Max(result.max, iterator.Current));
15        } 1
16        return result;
17    }
18 }
```

### Listing 11.5. Implementing the Fibonacci sequence without tuples

```
1 static IEnumerable<int> Fibonacci()
2 {
3     int current = 0;
4     int next = 1;
5     while (true)
6     {
7         yield return current;
8         int nextNext = current + next;
9         current = next;
10        next = nextNext;
11    }
12 }
```

---

### Listing 11.6. Implementing the Fibonacci sequence with tuples

```
1 static IEnumerable<int> Fibonacci()
2 {
3     var pair = (current: 0, next: 1);
4     while (true)
5     {
6         yield return pair.current;
7         pair = (pair.next, pair.current + pair.next);
8     }
9 }
```

---

### Listing 11.7. Separating concerns of sequence generation for Fibonacci

```
1 static IEnumerable<TResult>
2     GenerateSequence<TState, TResult>(
3         TState seed,
4         Func<TState, TState> generator,
5         Func<TState, TResult> resultSelector)
6 {
7     var state = seed;
8     while (true)
9     {
10        yield return resultSelector(state);
11        state = generator(state);
12    }
13 }
14
15 Sample usage
16 var fibonacci = GenerateSequence(
17     (current: 0, next: 1),
18     pair => (pair.next, pair.current + pair.next),
19     pair => pair.current);
```

---

```
1 var valid = (10, 20);
```

```
1 var invalid = (10, null);
```

```
var tuple = (x: 10, 20);  
  
var array = new[] { ("a", 10) };  
  
string[] input = { "a", "b" };  
var query = input  
    .Select(x => (x, x.Length));
```

```
(int x, int) tuple = (x: 10, 20);  
  
(string, int)[] array = { ("a", 10) };  
  
string[] input = { "a", "b" };  
IEnumerable<(string, int)> query =  
    input.Select<string, (string, int)>  
        (x => (x, x.Length));
```

## LAMBDA EXPRESSION PARAMETERS CAN LOOK LIKE TUPLES

Lambda expressions with a single parameter aren't confusing, but if you use two parameters, they can look like tuples. As an example, let's look at a useful method that just uses the LINQ `Select` overload that provides the projection with the index of the element as well as the value. It's often useful to propagate the index through the other operations, so it makes sense to put the two pieces of data in a tuple. That means you end up with this method:

```
1 static IEnumerable<(T value, int index)> WithIndex<T>  
2     (this IEnumerable<T> source) =>  
3     source.Select((value, index) => (value, index));
```

Concentrate on the lambda expression:

```
1 (value, index) => (value, index)
```

Here the first occurrence of `(value, index)` isn't a tuple literal; it's the sequence of parameters for the lambda expression. The second occurrence *is* a tuple literal, the result of the lambda expression.

There's nothing wrong here. I just don't want it to take you by surprise when you see something similar.

```
1 (byte, object) tuple = (5, "text");
```

`(5, "text")`

↓ ✓   ↓ ✓   ✓

`(byte, object)`

```
1 (byte, string) tuple = (300, "text");
```

`(300, "text")`

↓ ✗   ↓ ✓   ✗

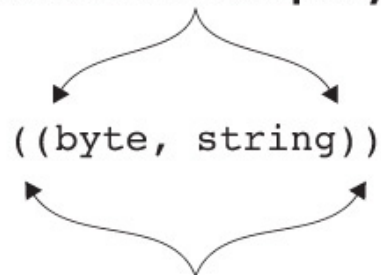
`(byte, string)`

```
1 error CS0029: Cannot implicitly convert type 'int' to 'byte'
```

```
1 int x = 300;  
2 var tuple = ((byte, string)) (x, "text");
```

Figure 11.4. Explaining the parentheses in an explicit tuple conversion

**Start and end of tuple type**



```
1 int x = 300;  
2 var tuple = ((byte) x, "text");
```

```
1 static (int min, int max) MinMax(IEnumerable<int> source)
```

```
1 return (min, max);
```

```
1 (int a, int b, int c, int, int) tuple =  
2   (a: 10, wrong: 20, 30, pointless: 40, 50);
```

```
1 warning CS8123: The tuple element name 'wrong' is ignored because a different
2     name is specified by the target type '(int a, int b, int c, int, int)'.
3 warning CS8123: The tuple element name 'pointless' is ignored because a
4     different name is specified by the target type '(int a, int b, int c,
5     int, int)'
```

---

```
1 return (min: min, max: max);
```

---

```
1 return (max: max, min: min);
```

---

1

```
1 var t1 = (300, "text");
2 (long, string) t2 = t1;
3 (byte, string) t3 = t1;
4 (byte, string) t4 = ((byte, string)) t1;
5 (object, object) t5 = t1;
6 (string, string) t6 = ((string, string)) t1;
```

---

1  
2  
3  
4  
5  
6

```
1 var source = (a: 10, wrong: 20, 30, pointless: 40, 50);
2 (int a, int b, int c, int, int) tuple = source;
```

---

```
1 public void Method((int, int) tuple) {}
2 public void Method((int x, int y) tuple) {}
```

---

```
1 error CS0111: Type 'Program' already defines a member called 'Method' with
2     the same parameter types
```

---

```
1 IEnumerable<(string, string)> stringPairs = new (string, string)[10];
2 IEnumerable<(object, object)> objectPairs = stringPairs;
```

---

```
1 interface ISample
2 {
3     void Method((int x, string) tuple);
4 }
5
6 public void Method((string x, object) tuple) {}
7 public void Method((int, string) tuple) {}
8 public void Method((int x, string extra) tuple) {}
9 public void Method((int wrong, string) tuple) {}
10 public void Method((int x, string, int) tuple) {}
11 public void Method((int x, string) tuple) {}
```

---

1  
2  
3  
4  
5  
6

### Listing 11.8. Equality and inequality operators

```
1 var t1 = (x: "x", y: "y", z: 1);
2 var t2 = ("x", "y", 1);
3
4 Console.WriteLine(t1 == t2);
5 Console.WriteLine(t1.Item1 == t2.Item1 &&
6     t1.Item2 == t2.Item2 &&
7     t1.Item3 == t2.Item3);
8
9 Console.WriteLine(t1 != t2);
10 Console.WriteLine(t1.Item1 != t2.Item1 ||
11     t1.Item2 != t2.Item2 ||
12     t1.Item3 != t2.Item3);
```

---

1  
2  
3  
4

Figure 11.5. Compiler translation of tuple type handling into use of ValueTuple



```
var tuple = (x: 10, y: 20);  
Console.WriteLine(tuple.x);  
Console.WriteLine(tuple.y);
```

Compiler translation

```
var tuple = new ValueTuple<int, int>(10, 20);  
Console.WriteLine(tuple.Item1);  
Console.WriteLine(tuple.Item2);
```

```
1 [return: TupleElementNames(new[] {"min", "max"})]  
2 public static ValueTuple<int, int> MinMax(IEnumerable<int> numbers)
```

```
1 (int, string) t1 = (300, "text");  
2 (long, string) t2 = t1;  
3 (byte, string) t3 = ((byte) t1);
```

```
1 var t1 = new ValueTuple<int, string>(300, "text");  
2 var t2 = new ValueTuple<long, string>(t1.Item1, t1.Item2);  
3 var t3 = new ValueTuple<byte, string>((byte) t1.Item1, t1.Item2);
```

```
1 var tuple = (x: (string) null, y: "text", z: 10);  
2 Console.WriteLine(tuple.ToString());
```

1  
2

```
1 (, text, 10)
```

### Listing 11.9. Finding and ordering distinct points

```
1 var points = new[]
2 {
3     (1, 2), (10, 3), (-1, 5), (2, 1),
4     (10, 3), (2, 1), (1, 1)
5 };
6 var distinctPoints = points.Distinct();
7 Console.WriteLine($"{distinctPoints.Count()} distinct points");
8 Console.WriteLine("Points in order:");
9 foreach (var point in distinctPoints.OrderBy(p => p))
10 {
11     Console.WriteLine(point);
12 }
```

---

```
1 5 distinct points
2 Points in order:
3 (-1, 5)
4 (1, 1)
5 (1, 2)
6 (2, 1)
7 (10, 3)
```

---

```
1 public interface IStructuralEquatable
2 {
3     bool Equals(Object, IEqualityComparer);
4     int GetHashCode(IEqualityComparer);
5 }
6
7 public interface IStructuralComparable
8 {
9     int CompareTo(Object, IComparer);
10 }
```

---

### Listing 11.10. Structural comparisons with a case-insensitive comparer

```
1 static void Main()
2 {
3     var Ab = ("A", "b");
```

```

4      var aB = ("a", "B");
5      var aa = ("a", "a");
6      var ba = ("b", "a");
7
8      Compare(Ab, aB);
9      Compare(aB, aa);
10     Compare(aB, ba);
11 }
12
13 static void Compare<T>(T x, T y)
14     where T : IStructuralEquatable, IStructuralComparable
15 {
16     var comparison = x.CompareTo(
17         y, StringComparer.OrdinalIgnoreCase);
18     var equal = x.Equals(
19         y, StringComparer.OrdinalIgnoreCase);
20
21     Console.WriteLine(
22         $"{x} and {y} - comparison: {comparison}; equal: {equal}");
23 }

```

```

1 (A, b) and (a, B) - comparison: 0; equal: True
2 (a, B) and (a, a) - comparison: 1; equal: False
3 (a, B) and (b, a) - comparison: -1; equal: False

```

```

1 ValueTuple<int, int, int, int, int, int, int, ValueTuple<int>>

```

```

1 ValueTuple<A, B, C, D, E, F, G, ValueTuple<H, I>>

```

```

1 var tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16);
2 Console.WriteLine(tuple.Item16);

```

```

1 var tuple = ValueTuple.Create(5, 10);

```

### Listing 11.11. Displaying the highest-scoring player for a date

```
1 public void DisplayHighScoreForDate(LocalDate date)
2 {
3     var filteredGames = allGames.Where(game => game.Date == date);
4     string highestPlayer = null;
5     int highestScore = -1;
6     foreach (var game in filteredGames)
7     {
8         if (game.Score > highestScore)
9         {
10             highestPlayer = game.PlayerName;
11             highestScore = game.Score;
12         }
13     }
14     Console.WriteLine(highestPlayer == null
15         ? "No games played"
16         : $"Highest score was {highestScore} by {highestPlayer}");
17 }
```

---

### Listing 11.12. A refactoring to use a tuple local variable

```
1 public void DisplayHighScoreForDate(LocalDate date)
2 {
3     var filteredGames = allGames.Where(game => game.Date == date);
4     (string player, int score) highest = (null, -1);
5     foreach (var game in filteredGames)
6     {
7         if (game.Score > highest.score)
8         {
9             highest = (game.PlayerName, game.Score);
10        }
11    }
12    Console.WriteLine(highest.player == null
13        ? "No games played"
14        : $"Highest score was {highest.score} by {highest.player}");
15 }
```

---

```
1 private readonly ZoneInterval[] periods;
2 private readonly IZoneIntervalMapWithMinMax tailZone;
3 private readonly Instant tailZoneStart;
4 private readonly ZoneInterval firstTailZoneInterval;
```

---

```
1 private readonly ZoneInterval[] periods;
2 private readonly
3     (IZoneIntervalMapWithMinMax intervalMap,
4      Instant start,
5      ZoneInterval firstInterval) tailZone;
```

---

```
1 dynamic tuple = (x: 10, y: 20);
2 Console.WriteLine(tuple.x);
```

---

```
1 Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException:
2     'System.ValueTuple<int,int>' does not contain a definition for 'x'
```

---

```
1 var tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9);
2 Console.WriteLine(tuple.Item9);
3 dynamic d = tuple;
4 Console.WriteLine(d.Item9);
```

---

1

2

# CHAPTER 12

## Listing 12.1. Overview of deconstruction using tuples

```
1 var tuple = (10, "text"); 1
2
3 var (a, b) = tuple; 2
4
5 (int c, string d) = tuple; 3
6
7 int e;
8 string f;
9 (e, f) = tuple; 4
10
11 Console.WriteLine($"a: {a}; b: {b}");
12 Console.WriteLine($"c: {c}; d: {d}");
13 Console.WriteLine($"e: {e}; f: {f}"); 5
```

```
1 a: 10; b: text
2 c: 10; d: text
3 e: 10; f: text
```

## TUPLE DECLARATION AND DECONSTRUCTION SYNTAX

The language specification regards deconstruction as closely related to other tuple features. Deconstruction syntax is described in terms of a *tuple expression* even when you're not deconstructing tuples (which you'll see in [section 12.2](#)). You probably don't need to worry too much about that, but you should be aware of potential causes for confusion. Consider these two statements:

```
1 (int c, string d) = tuple;
2 (int c, string d) x = tuple;
```

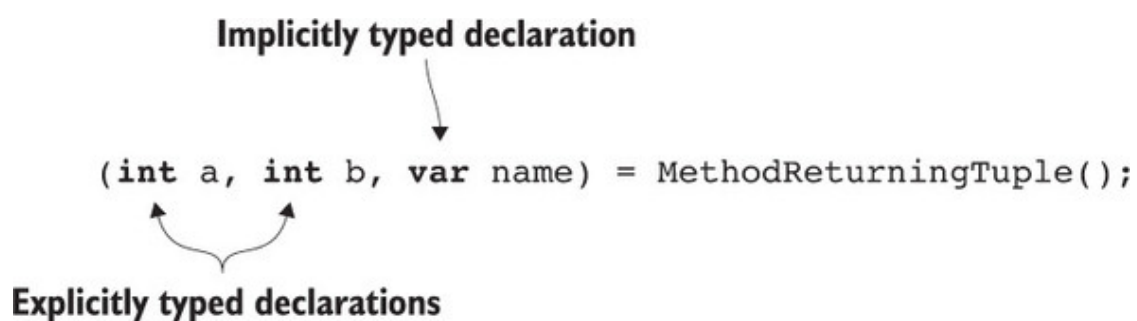
The first uses deconstruction to declare two variables (`c` and `d`); the second is a declaration of a single variable (`x`) of tuple type `(int c, string d)`. I don't think this similarity was a design mistake, but it can take a little getting used to just like expression-bodied members looking like lambda expressions.

### Listing 12.2. Calling a method and deconstructing the result into three variables

```
1 static (int x, int y, string text) MethodReturningTuple() => (1, 2, "t");
2
3 static void Main()
4 {
5     (int a, int b, string name) = MethodReturningTuple();
6     Console.WriteLine($"a: {a}; b: {b}; name: {name}");
7 }
```

```
1 static void Main()
2 {
3     var tmp = MethodReturningTuple();
4     int a = tmp.x;
5     int b = tmp.y;
6     string name = tmp.text;
7
8     Console.WriteLine($"a: {a}; b: {b}; name: {name}");
9 }
```

### Figure 12.1. Mixing implicit and explicit typing in deconstruction



### Figure 12.2. Deconstruction involving implicit conversions

### Implicitly typed declaration

`(long a, var b, XNamespace name) = MethodReturningTuple();`

Explicitly typed declarations  
using conversions

```
1 var (a, b, name) = MethodReturningTuple();
```

```
1 var (a, long b, name) = MethodReturningTuple();
```

1

```
1 var tuple = (1, 2, 3, 4);
2 var (x, y, _, _) = tuple;
3 Console.WriteLine(_);
```

1

2

3

```
1 var tuple = (10, "text");
2 int e;
3 string f;
4 (e, f) = tuple;
```

### Listing 12.3. Assignments to existing variables using deconstruction

```
1 static (int x, int y, string text) MethodReturningTuple() => (1, 2, "t");
2
3 static void Main()
4 {
5     int a = 20;
6     int b = 30;
7     string name = "before";
8     Console.WriteLine($"a: {a}; b: {b}; name: {name}");
```



```
9
10 (a, b, name) = MethodReturningTuple();
11
12 Console.WriteLine($"a: {a}; b: {b}; name: {name}");
13 }
```

2

3

## DECLARATIONS OR ASSIGNMENTS: NOT A MIXTURE

Deconstruction allows you to either declare and initialize variables or execute a sequence of assignments. You can't mix the two. For example, this is invalid:

```
1 int x;
2 (x, int y) = (1, 2);
```

It's fine for the assignments to use a variety of targets, however: some existing local variables, some fields, some properties, and so on.

### Listing 12.4. Simple constructor assignments using deconstruction and a tuple literal

```
1 public sealed class Point
2 {
3     public double X { get; }
4     public double Y { get; }
5
6     public Point(double x, double y) => (X, Y) = (x, y);
7 }
```

### Listing 12.5. Deconstruction in which evaluation order matters

```
1 StringBuilder builder = new StringBuilder("12345");
2 StringBuilder original = builder;
3
4 (builder, builder.Length) =
5     (new StringBuilder("67890"), 3);
6
7 Console.WriteLine(original);
8 Console.WriteLine(builder);
```

1

2

3

## Listing 12.6. Slow-motion deconstruction to show evaluation order

```
1  StringBuilder builder = new StringBuilder("12345");
2  StringBuilder original = builder;
3
4  StringBuilder targetForLength = builder;                                1
5
6  (StringBuilder, int) tuple =
7      (new StringBuilder("67890"), 3);                                    2
8
9  builder = tuple.Item1;
10 targetForLength.Length = tuple.Item2;                                    3
11
12 Console.WriteLine(original);
13 Console.WriteLine(builder);
```

---

```
1 123
2 67890
```

---

```
1 (string text, Func<int, int> func) =
2     (null, x => x * 2);                                                    1
3 (text, func) = ("text", x => x * 3);                                       2
```

---

```
1 (byte x, byte y) = (5, 10);
```

---

```
1 public void Deconstruct(out double x, out double y)
2 {
3     x = X;
4     y = Y;
5 }
```

---

## Listing 12.7. Deconstructing a `Point` to two variables

```
1 var point = new Point(1.5, 20);
2 var (x, y) = point;
3 Console.WriteLine($"x = {x}");
4 Console.WriteLine($"y = {y}");
```

```
1 public Point(double x, double y) => (X, Y) = (x, y);
2 public void Deconstruct(out double x, out double y) => (x, y) = (X, Y);
```

## Listing 12.8. Using an extension method to deconstruct `DateTime`

```
1 static void Deconstruct(
2     this DateTime dateTime,
3     out int year, out int month, out int day) =>
4     (year, month, day) =
5     (dateTime.Year, dateTime.Month, dateTime.Day);
6
7 static void Main()
8 {
9     DateTime now = DateTime.UtcNow;
10    var (year, month, day) = now;
11    Console.WriteLine(
12        $"{year:0000}-{month:00}-{day:00}");
13 }
```

## Listing 12.9. Using `Deconstruct` overloads

```
1 static void Deconstruct(
2     this DateTime dateTime,
3     out int year, out int month, out int day) =>
4     (year, month, day) =
5     (dateTime.Year, dateTime.Month, dateTime.Day);
6
7 static void Deconstruct(
8     this DateTime dateTime,
9     out int year, out int month, out int day,
10    out int hour, out int minute, out int second) =>
11    (year, month, day, hour, minute, second) =
12    (dateTime.Year, dateTime.Month, dateTime.Day,
13     dateTime.Hour, dateTime.Minute, dateTime.Second);
14
15 static void Main()
16 {
```

```
17     DateTime birthday = new DateTime(1976, 6, 19);
18     DateTime now = DateTime.UtcNow;
19
20     var (year, month, day, hour, minute, second) = now;
21     (year, month, day) = birthday;
22 }
```

3

4

```
1 (int x, string y) = target;
```

```
1 target.Deconstruct(out var tmpX, out var tmpY);
2 int x = tmpX;
3 string y = tmpY;
```

## Listing 12.10. Computing a perimeter without patterns

```
1 static double Perimeter(Shape shape)
2 {
3     if (shape == null)
4         throw new ArgumentNullException(nameof(shape));
5     Rectangle rect = shape as Rectangle;
6     if (rect != null)
7         return 2 * (rect.Height + rect.Width);
8     Circle circle = shape as Circle;
9     if (circle != null)
10        return 2 * PI * circle.Radius;
11    Triangle triangle = shape as Triangle;
12    if (triangle != null)
13        return triangle.SideA + triangle.SideB + triangle.SideC;
14    throw new ArgumentException(
15        $"Shape type {shape.GetType()} perimeter unknown", nameof(shape));
16 }
```

## Listing 12.11. Computing a perimeter with patterns

```
1 static double Perimeter(Shape shape)
2 {
3     switch (shape)
4     {
5         case null:
6             throw new ArgumentNullException(nameof(shape));
```

```

7      case Rectangle rect:
8          return 2 * (rect.Height + rect.Width);
9      case Circle circle:
10         return 2 * PI * circle.Radius;
11     case Triangle tri:
12         return tri.SideA + tri.SideB + tri.SideC;
13     default:
14         throw new ArgumentException(...);
15 }
16 }

```

## Listing 12.12. Simple constant matches

```

1 static void Match(object input)
2 {
3     if (input is "hello")
4         Console.WriteLine("Input is string hello");
5     else if (input is 5L)
6         Console.WriteLine("Input is long 5");
7     else if (input is 10)
8         Console.WriteLine("Input is int 10");
9     else
10        Console.WriteLine("Input didn't match hello, long 5 or int 10");
11 }
12 static void Main()
13 {
14     Match("hello");
15     Match(5L);
16     Match(7);
17     Match(10);
18     Match(10L);
19 }

```

```

1 Input is string hello
2 Input is long 5
3 Input didn't match hello, long 5 or int 10
4 Input is int 10
5 Input didn't match hello, long 5 or int 10

```

```

1 long x = 10L;
2 if (x is 10)
3 {
4     Console.WriteLine("x is 10");
5 }

```

### Listing 12.13. Using type patterns instead of `as/if`

```
1 static double Perimeter(Shape shape)
2 {
3     if (shape == null)
4         throw new ArgumentNullException(nameof(shape));
5     if (shape is Rectangle rect)
6         return 2 * (rect.Height + rect.Width);
7     if (shape is Circle circle)
8         return 2 * PI * circle.Radius;
9     if (shape is Triangle triangle)
10        return triangle.SideA + triangle.SideB + triangle.SideC;
11    throw new ArgumentException(
12        $"Shape type {shape.GetType()} perimeter unknown", nameof(shape));
13 }
```

---

### Listing 12.14. Behavior of nullable value types in type patterns

```
1 static void Main()
2 {
3     CheckType<int?>(null);
4     CheckType<int?>(5);
5     CheckType<int?>("text");
6     CheckType<string>(null);
7     CheckType<string>(5);
8     CheckType<string>("text");
9 }
10
11 static void CheckType<T>(object value)
12 {
13     if (value is T t)
14     {
15         Console.WriteLine($"Yes! {t} is a {typeof(T)}");
16     }
17     else
18     {
19         Console.WriteLine($"No! {value ?? "null"} is not a {typeof(T)}");
20     }
21 }
```

---

```
1 No! null is not a System.Nullable`1[System.Int32]
2 Yes! 5 is a System.Nullable`1[System.Int32]
3 No! text is not a System.Nullable`1[System.Int32]
4 No! null is not a System.String
5 No! 5 is not a System.String
6 Yes! text is a System.String
```

---

```
1 x is SomeType y
```

---

### Listing 12.15. Generic method using type patterns

```
1 static void DisplayShapes(List shapes) where T : Shape
2 {
3     foreach (T shape in shapes)                                1
4     {
5         switch (shape)                                         2
6         {
7             case Circle c:
8                 Console.WriteLine($"Circle radius {c.Radius}"); 3
9                 break;
10            case Rectangle r:
11                Console.WriteLine($"Rectangle {r.Width} x {r.Height}");
12                break;
13            case Triangle t:
14                Console.WriteLine(
15                    $"Triangle sides {t.SideA}, {t.SideB}, {t.SideC}");
16                break;
17        }
18    }
19 }
```

---

```
1 if (shape is Circle)
2 {
3     Circle c = (Circle) shape;
4 }
```

---

```
1 if (shape is Circle)
2 {
3     Circle c = (Circle) (object) shape;
4 }
```

---

```
1 someExpression is var x
```

## Listing 12.16. Using the `var` pattern to introduce a variable on error

```
1 static double Perimeter(Shape shape)
2 {
3     switch (shape ?? CreateRandomShape())
4     {
5         case Rectangle rect:
6             return 2 * (rect.Height + rect.Width);
7         case Circle circle:
8             return 2 * PI * circle.Radius;
9         case Triangle triangle:
10            return triangle.SideA + triangle.SideB + triangle.SideC;
11        case var actualShape:
12            throw new InvalidOperationException(
13                $"Shape type {actualShape.GetType()} perimeter unknown");
14    }
15 }
```

---

```
1 static int length = GetObject() is string text ? text.Length : -1;
```

---

```
1 string text = input as string;
2 if (text != null)
3 {
4     Console.WriteLine(text);
5 }
```

---

```
1 if (input is string text)
2 {
3     Console.WriteLine(text);
4 }
```

---



```

1 if (input is string text)
2 {
3     Console.WriteLine("Input was already a string; using that");
4 }
5 else if (input is StringBuilder builder)
6 {
7     Console.WriteLine("Input was a StringBuilder; using that");
8     text = builder.ToString();
9 }
10 else
11 {
12     Console.WriteLine(
13         $"Unable to use value of type ${input.GetType()}. Enter text:");
14     text = Console.ReadLine();
15 }
16 Console.WriteLine($"Final result: {text}");

```

---

```

1 if (input is int x && x > 100)
2 {
3     Console.WriteLine($"Input was a large integer: {x}");
4 }

```

---

```

1 if ((input is int x && x > 100) || (input is long y && y > 100))
2 {
3     Console.WriteLine($"Input was a large integer of some kind");
4 }

```

---

```

1 case pattern when expression:

```

---

## Listing 12.17. Implementing the Fibonacci sequence recursively with patterns

```

1 static int Fib(int n)
2 {
3     switch (n)
4     {
5         case 0: return 0;
6         case 1: return 1;
7         case var _ when n > 1: return Fib(n - 2) + Fib(n - 1);
8         default: throw new ArgumentOutOfRangeException(

```

```
9         nameof(n), "Input must be non-negative");
10     }
11 }
```

---

3

```
1 private string GetUid(TypeReference type, bool useTypeArgumentNames)
2 {
3     switch (type)
4     {
5         case ByReferenceType brt:
6             return $"{GetUid(brt.ElementType, useTypeArgumentNames)}@";
7         case GenericParameter gp when useTypeArgumentNames:
8             return gp.Name;
9         case GenericParameter gp when gp.DeclaringType != null:
10            return $"{gp.Position}";
11         case GenericParameter gp when gp.DeclaringMethod != null:
12            return $"{gp.Position}";
13         case GenericParameter gp:
14            throw new InvalidOperationException(
15                "Unhandled generic parameter");
16         case GenericInstanceType git:
17            return "(This part of the real code is long and irrelevant)";
18         default:
19            return type.FullName.Replace('/', '.');
20     }
21 }
```

---

### Listing 12.18. Using multiple `case` labels with patterns for a single `case` body

```
1 static void CheckBounds(object input)
2 {
3     switch (input)
4     {
5         case int x when x > 1000:
6         case long y when y > 10000L:
7             Console.WriteLine("Value is too large");
8             break;
9         case int x when x < -1000:
10        case long y when y < -10000L:
11            Console.WriteLine("Value is too low");
12            break;
13        default:
14            Console.WriteLine("Value is in range");
15            break;
16    }
17 }
```

---

```
1 case GenericParameter gp when useTypeArgumentNames:
2     return gp.Name;
3 case GenericParameter gp when gp.DeclaringType != null:
4     return $"{gp.Position}";
5 case GenericParameter gp when gp.DeclaringMethod != null:
6     return $"{gp.Position}";
7 case GenericParameter gp:
8     throw new InvalidOperationException(...);
```

---

```
1 int[] values = { 2, 7, 3, -5, 1, 0, 10 };
2 var (min, max) = MinMax(values);
3 Console.WriteLine(min);
4 Console.WriteLine(max);
```

---

# CHAPTER 13

Figure 13.1. Representing a variable as a piece of paper

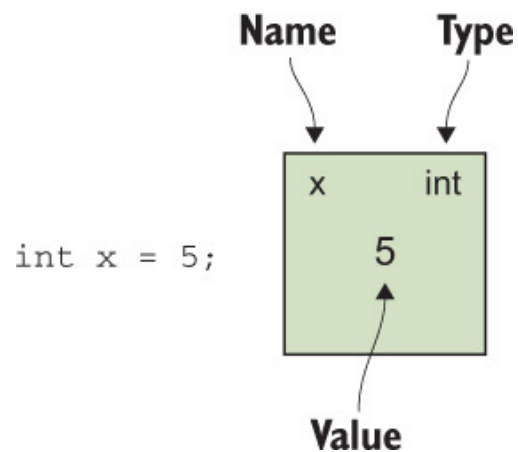


Figure 13.2. Assignment copying a value into a new variable

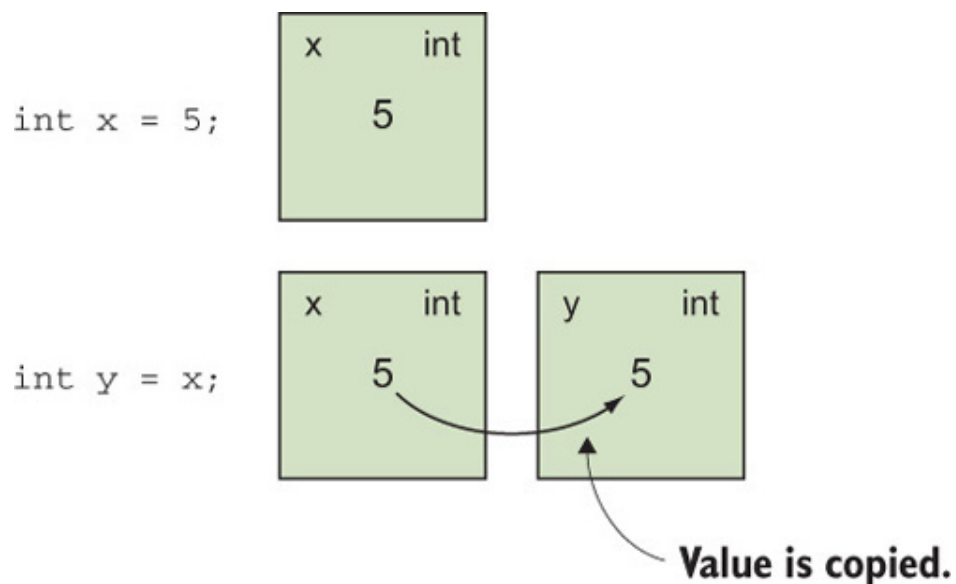


Figure 13.3. Calling a method with value parameters: the parameters are new variables that start with the values of the arguments.

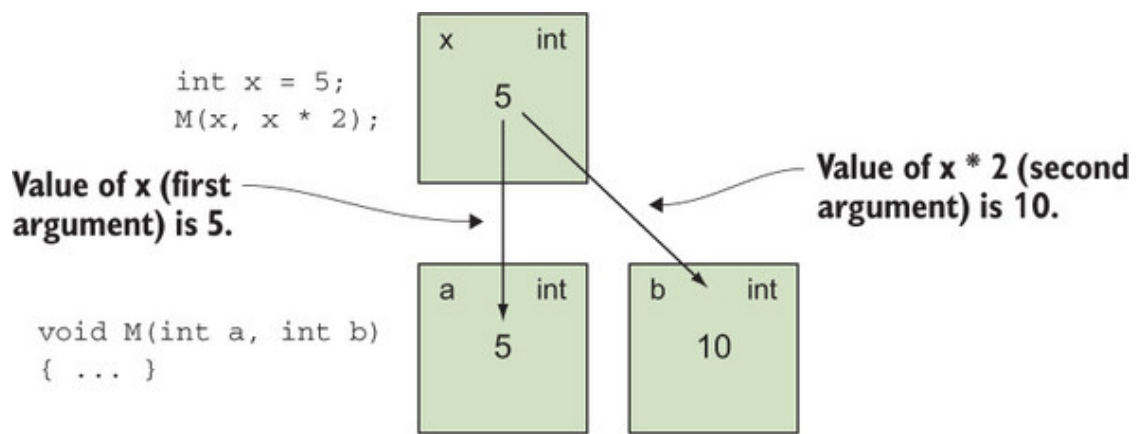
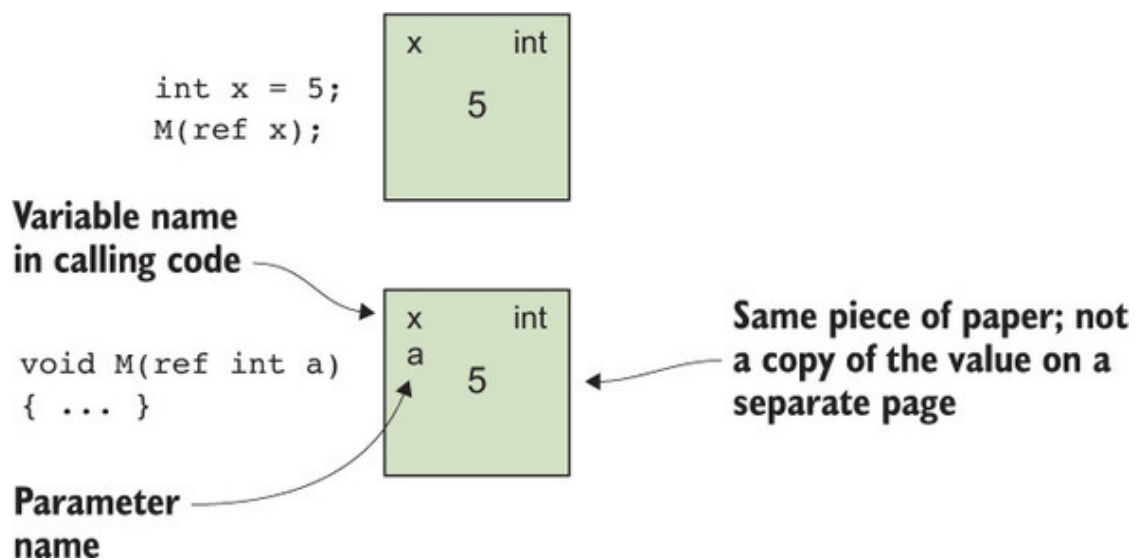


Figure 13.4. A ref parameter uses the same piece of paper rather than creating a new one with a copy of the value.



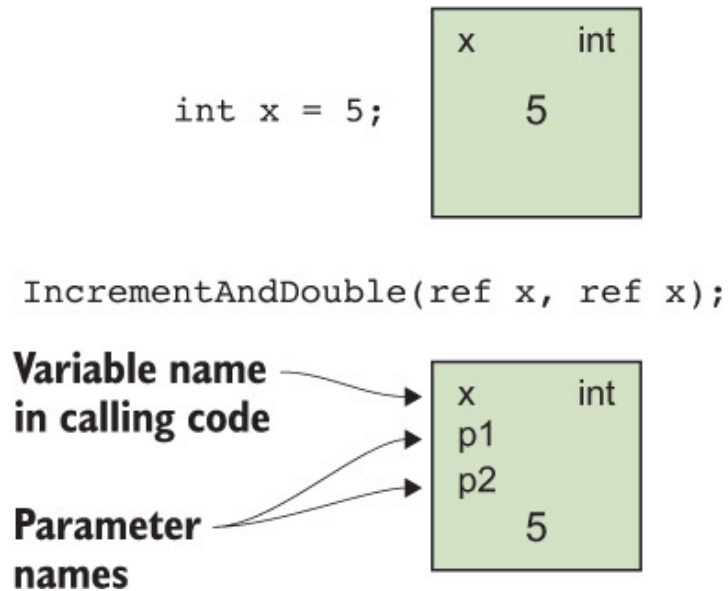
Listing 13.1. Using the same variable for multiple ref parameters

```

1 static void Main()
2 {
3     int x = 5;
4     IncrementAndDouble(ref x, ref x);
5     Console.WriteLine(x);
6 }
7
8 static void IncrementAndDouble(ref int p1, ref int p2)
9 {
10    p1++;
11    p2 *= 2;
12 }

```

Figure 13.5. Two ref parameters referring to the same piece of paper



Listing 13.2. Incrementing twice via two variables

```
1 int x = 10;
2 ref int y = ref x;
3 x++;
4 y++;
5 Console.WriteLine(x);
```

Listing 13.3. Modifying array elements using ref local

```
1 var array = new (int x, int y)[10];
2
3 for (int i = 0; i < array.Length; i++)
4 {
5     array[i] = (i, i);
6 }
7
8 for (int i = 0; i < array.Length; i++)
9 {
10    ref var element = ref array[i];
11    element.x++;
12    element.y *= 2;
13 }
```

1

2

```
1 for (int i = 0; i < array.Length; i++)
2 {
```

```
3     array[i].x++;
4     array[i].y *= 2;
5 }
```

---

```
1 for (int i = 0; i < array.Length; i++)
2 {
3     var tuple = array[i];
4     tuple.x++;
5     tuple.y *= 2;
6     array[i] = tuple;
7 }
```

---

#### Listing 13.4. Aliasing the field of a specific object by using ref local

```
1 class RefLocalField
2 {
3     private int value;
4
5     static void Main()
6     {
7         var obj = new RefLocalField();
8         ref int tmp = ref obj.value;
9         tmp = 10;
10        Console.WriteLine(obj.value);
11
12        obj = new RefLocalField();
13        Console.WriteLine(tmp);
14        Console.WriteLine(obj.value);
15    }
16 }
```

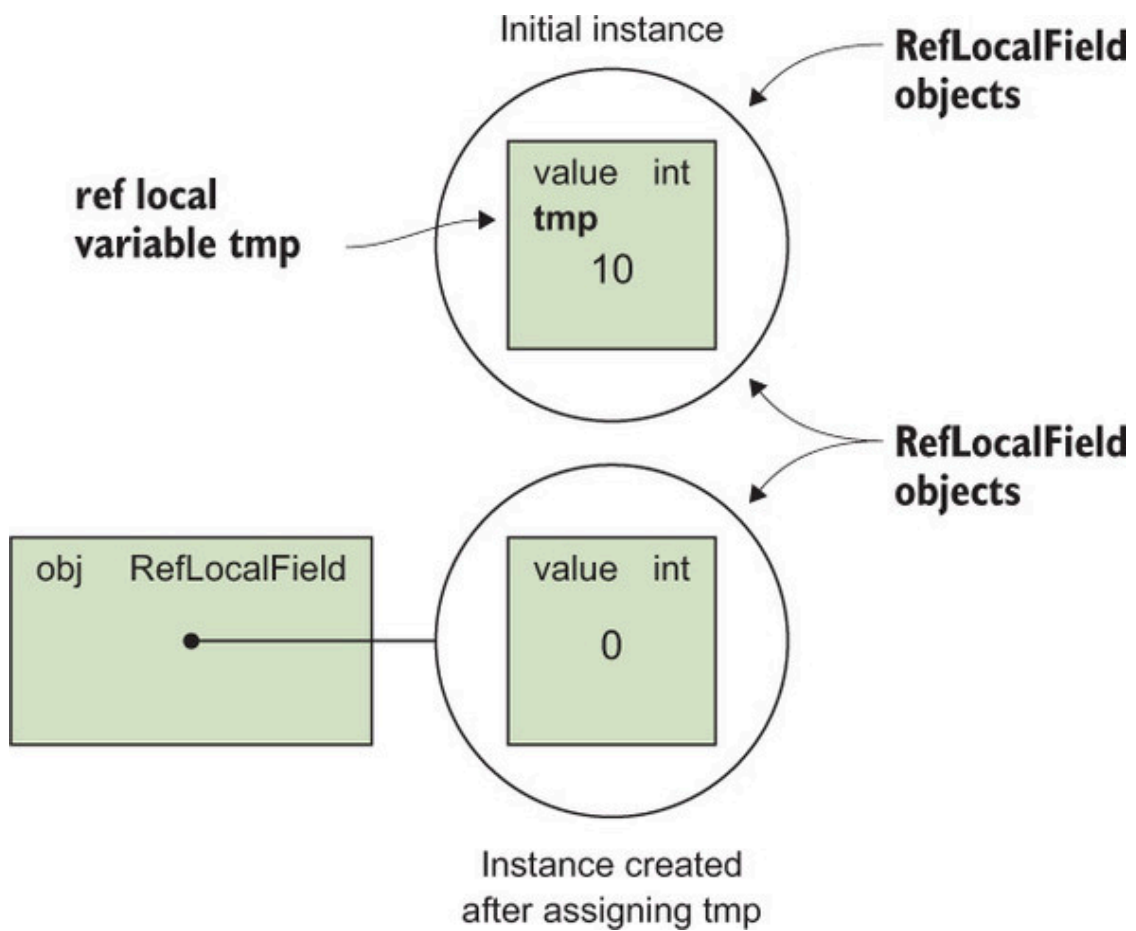
1  
2  
3  
4  
5  
6  
7

---

```
1 10
2 10
3 0
```

---

Figure 13.6. At the end of [listing 13.4](#), the tmp variable refers to a field in the first instance created, whereas the value of obj refers to a different instance.



```

1 int x = 10;
2 ref int invalid;
3 invalid = ref int x;

```

```

1 for (int i = 0; i < array.Length; i++)
2 {
3     ref var element = ref array[i];
4     ...
5 }

```

```

1 int x;
2 ref int y = ref x;
3 x = 10;
4 Console.WriteLine(y);

```



```
1 int x = 10;
2 int y = 20;
3 ref int r = ref x;
4 r++;
5 r = ref y;
6 r++;
7 Console.WriteLine($"x={x}; y={y}");
```

1

2

```
1 class MixedVariables
2 {
3     private int writableField;
4     private readonly int readonlyField;
5
6     public void TryIncrementBoth()
7     {
8         ref int x = ref writableField;
9         ref int y = ref readonlyField;
10
11         x++;
12         y++;
13     }
14 }
```

1

2

3

### Listing 13.5. Identity conversion in ref local declaration

```
1 (int x, int y) tuple1 = (10, 20);
2 ref (int a, int b) tuple2 = ref tuple1;
3 tuple2.a = 30;
4 Console.WriteLine(tuple1.x);
```

### Listing 13.6. Simplest possible ref return demonstration

```
1 static void Main()
2 {
3     int x = 10;
4     ref int y = ref RefReturn(ref x);
5     y++;
6     Console.WriteLine(x);
7 }
8
9 static ref int RefReturn(ref int p)
10 {
11     return ref p;
12 }
```

```
1 ref int y = ref x;
```

---

```
1 delegate ref int RefFuncInt32();
```

---

### Listing 13.7. Incrementing the result of a ref return directly

```
1 static void Main()  
2 {  
3     int x = 10;  
4     RefReturn(ref x)++;  
5     Console.WriteLine(x);  
6 }  
7  
8 static ref int RefReturn(ref int p)  
9 {  
10     return ref p;  
11 }
```

---

1

```
1 RefReturn(ref RefReturn(ref RefReturn(ref x)))++;
```

---

### Listing 13.8. A ref return indexer exposing array elements

```
1 class ArrayHolder  
2 {  
3     private readonly int[] array = new int[10];  
4     public ref int this[int index] => ref array[index];  
5 }  
6  
7 static void Main()  
8 {  
9     ArrayHolder holder = new ArrayHolder();  
10    ref int x = ref holder[0];  
11    ref int y = ref holder[0];  
12  
13    x = 20;
```

---

1

2

```
14     Console.WriteLine(y);
15 }
```

3  
4

```
1 condition ? expression1 : expression2
```

### Listing 13.9. Counting even and odd elements in a sequence

```
1 static (int even, int odd) CountEvenAndOdd(IEnumerable<int> values)
2 {
3     var result = (even: 0, odd: 0);
4     foreach (var value in values)
5     {
6         ref int counter = ref (value & 1) == 0 ?
7             ref result.even : ref result.odd;
8         counter++;
9     }
10    return result;
11 }
```

1  
2

### Listing 13.10. `ref readonly` return and local

```
1 static readonly int field = DateTime.UtcNow.Second;
2
3 static ref readonly int GetFieldAlias() => ref field;
4
5 static void Main()
6 {
7     ref readonly int local = ref GetFieldAlias();
8     Console.WriteLine(local);
9 }
```

1  
2  
3

### Listing 13.11. A read-only view over an array with copy-free reads

```
1 class ReadOnlyArrayView<T>
2 {
3     private readonly T[] values;
4
5     public ReadOnlyArrayView(T[] values) =>
6         this.values = values;
7 }
```

1

```

8     public ref readonly T this[int index] =>
9         ref values[index];
10 }
11 ...
12 static void Main()
13 {
14     var array = new int[] { 10, 20, 30 };
15     var view = new ReadOnlyArrayView<int>(array);
16
17     ref readonly int element = ref view[0];
18     Console.WriteLine(element);
19     array[0] = 100;
20     Console.WriteLine(element);
21 }

```

2

3

### Listing 13.12. Valid and invalid possibilities for passing arguments for `in` parameters

```

1 static void PrintDateTime(in DateTime value)
2 {
3     string text = value.ToString(
4         "yyyy-MM-dd'T'HH:mm:ss",
5         CultureInfo.InvariantCulture);
6     Console.WriteLine(text);
7 }
8
9 static void Main()
10 {
11     DateTime start = DateTime.UtcNow;
12     PrintDateTime(start);
13     PrintDateTime(in start);
14     PrintDateTime(start.AddMinutes(1));
15     PrintDateTime(in start.AddMinutes(1));
16 }

```

1

2

3

4

5

### Listing 13.13. `in` parameter and value parameter differences in the face of side effects

```

1 static void InParameter(in int p, Action action)
2 {
3     Console.WriteLine("Start of InParameter method");
4     Console.WriteLine($"p = {p}");
5     action();
6     Console.WriteLine($"p = {p}");
7 }
8
9 static void ValueParameter(int p, Action action)
10 {
11     Console.WriteLine("Start of ValueParameter method");
12     Console.WriteLine($"p = {p}");
13     action();
14     Console.WriteLine($"p = {p}");
15 }
16
17 static void Main()

```

```

18 {
19     int x = 10;
20     InParameter(x, () => x++);
21     int y = 10;
22     ValueParameter(y, () => y++);
23 }

```

---

```

1 Start of InParameter method
2 p = 10
3 p = 11
4 Start of ValueParameter method
5 p = 10
6 p = 10

```

---

```

1 void Method(int x) { ... }
2 void Method(in int x) { ... }

```

---

```

1 int x = 5;
2 Method(5);
3 Method(x);
4 Method(in x);

```

---

1  
2  
3

### Listing 13.14. Using `in` parameters safely

```

1  public static double PublicMethod(
2      LargeStruct first,
3      LargeStruct second)
4  {
5      double firstResult = PrivateMethod(in first);
6      double secondResult = PrivateMethod(in second);
7      return firstResult + secondResult;
8  }
9
10 private static double PrivateMethod(
11     in LargeStruct input)
12 {
13     double scale = GetScale(in input);
14     return (input.X + input.Y + input.Z) * scale;
15 }
16

```

1

2

```
17 private static double GetScale(in LargeStruct input) =>
18     input.Weight * input.Score;
```

---

3

### Listing 13.15. A trivial year/month/day struct

```
1 public struct YearMonthDay
2 {
3     public int Year { get; }
4     public int Month { get; }
5     public int Day { get; }
6
7     public YearMonthDay(int year, int month, int day) =>
8         (Year, Month, Day) = (year, month, day);
9 }
```

---

### Listing 13.16. Accessing properties via a read-only or read-write field

```
1 class ImplicitFieldCopy
2 {
3     private readonly YearMonthDay readOnlyField =
4         new YearMonthDay(2018, 3, 1);
5     private YearMonthDay readWriteField =
6         new YearMonthDay(2018, 3, 1);
7
8     public void CheckYear()
9     {
10         int readOnlyFieldYear = readOnlyField.Year;
11         int readWriteFieldYear = readWriteField.Year;
12     }
13 }
```

---

```
1 ldflld valuetype YearMonthDay ImplicitFieldCopy::readOnlyField
2 stloc.0
3 ldloca.s V_0
4 call instance int32 YearMonthDay::get_Year()
```

---

```
1 ldfllda valuetype YearMonthDay ImplicitFieldCopy::readWriteField
2 call instance int32 YearMonthDay::get_Year()
```

---

```
1 private void PrintYearMonthDay(YearMonthDay input) =>
2     Console.WriteLine($"{input.Year} {input.Month} {input.Day}");
```

---

```
1 ldarga.s input
2 call instance int32 Chapter13.YearMonthDay::get_Year()
```

---

```
1 private void PrintYearMonthDay(in YearMonthDay input) =>
2     Console.WriteLine($"{input.Year} {input.Month} {input.Day}");
```

---

```
1 ldarg.1
2 ldobj Chapter13.YearMonthDay
3 stloc.0
4 ldloca.s V_0
5 call instance int32 YearMonthDay::get_Year()
```

---

```
1 public readonly struct YearMonthDay
2 {
3     public int Year { get; }
4     public int Month { get; }
5     public int Day { get; }
6
7     public YearMonthDay(int year, int month, int day) =>
8         (Year, Month, Day) = (year, month, day);
9 }
```

---

```
1 ldarg.1
2 call instance int32 YearMonthDay::get_Year()
```

---

```
1 void IXmlSerializable.ReadXml(XmlReader reader)
2 {
3     var pattern = /* some suitable text parsing pattern for the type */;
4     var text = /* extract text from the XmlReader */;
5     this = pattern.Parse(text).Value;
6 }
```

---

### Listing 13.17. A trivial `Vector3D` struct

```
1 public readonly struct Vector3D
2 {
3     public double X { get; }
4     public double Y { get; }
5     public double Z { get; }
6
7     public Vector3D(double x, double y, double z)
8     {
9         X = x;
10        Y = y;
11        Z = z;
12    }
13 }
```

---

```
1 double magnitude = VectorUtilities.Magnitude(vector);
```

---

```
1 public static double Magnitude(this Vector3D vector)
```

---

### Listing 13.18. Extension methods using `ref` and `in`

```
1 public static double Magnitude(this in Vector3D vec) =>
2     Math.Sqrt(vec.X * vec.X + vec.Y * vec.Y + vec.Z * vec.Z);
3
```



```
4 public static void OffsetBy(this ref Vector3D orig, in Vector3D off) =>
5     orig = new Vector3D(orig.X + off.X, orig.Y + off.Y, orig.Z + off.Z);
```

---

### Listing 13.19. Calling `ref` and `in` extension methods

```
1 var vector = new Vector3D(1.5, 2.0, 3.0);
2 var offset = new Vector3D(5.0, 2.5, -1.0);
3
4 vector.OffsetBy(offset);
5
6 Console.WriteLine($"({vector.X}, {vector.Y}, {vector.Z})");
7 Console.WriteLine(vector.Magnitude());
```

---

```
1 (6.5, 4.5, 2)
2 8.15475321515004
```

---

```
1 ref readonly var alias = ref vector;
2 alias.OffsetBy(offset);
```

---

1

```
1 static void Method(this string target)
2 static void Method(this IDisposable target)
3 static void Method<T>(this T target)
4 static void Method<T>(this T target) where T : IComparable<T>
5 static void Method<T>(this T target) where T : struct
```

---

```
1 static void Method(this ref int target)
2 static void Method<T>(this ref T target) where T : struct
3 static void Method<T>(this ref T target) where T : struct, IComparable<T>
4 static void Method<T>(this ref int target, T other)
5 static void Method(this in int target)
6 static void Method(this in Guid target)
7 static void Method<T>(this in Guid target, T other)
```

---

```
1 static void Method(this ref string target) 1
2 static void Method<T>(this ref T target) 2
3     where T : IComparable<T> 3
4 static void Method<T>(this in string target) 3
5 static void Method<T>(this in T target) 4
6     where T : struct 4
```

---

```
1 public ref struct RefLikeStruct
2 {
3
4 }
```

---

1

```
1 int ReadData(byte[] buffer, int offset, int length)
```

---

### Listing 13.20. Generating a random string by using a `char[]`

```
1 static string Generate(string alphabet, Random random, int length)
2 {
3     char[] chars = new char[length];
4     for (int i = 0; i < length; i++)
5     {
6         chars[i] = alphabet[random.Next(alphabet.Length)];
7     }
8     return new string(chars);
9 }
```

---

```
1 string alphabet = "abcdefghijklmnopqrstuvwxyz";
2 Random random = new Random();
3 Console.WriteLine(Generate(alphabet, random, 10));
```

---

### Listing 13.21. Generating a random string by using `stackalloc` and a pointer

```
1 unsafe static string Generate(string alphabet, Random random, int length)
2 {
3     char* chars = stackalloc char[length];
4     for (int i = 0; i < length; i++)
5     {
6         chars[i] = alphabet[random.Next(alphabet.Length)];
7     }
8     return new string(chars);
9 }
```

---

### Listing 13.22. Generating a random string by using `stackalloc` and a `Span<char>`

```
1 static string Generate(string alphabet, Random random, int length)
2 {
3     Span<char> chars = stackalloc char[length];
4     for (int i = 0; i < length; i++)
5     {
6         chars[i] = alphabet[random.Next(alphabet.Length)];
7     }
8     return new string(chars);
9 }
```

---

```
1 public static string Create<TState>(
2     int length, TState state, SpanAction<char, TState> action)
```

---

```
1 delegate void SpanAction<T, in TArg>(Span<T> span, TArg arg);
```

---

### Listing 13.23. Generating a random string with `string.Create`

```
1 static string Generate(string alphabet, Random random, int length) =>
2     string.Create(length, (alphabet, random), (span, state) =>
3     {
4         var alphabet2 = state.alphabet;
```

```
5      var random2 = state.random;
6      for (int i = 0; i < span.Length; i++)
7
8      {
9          span[i] = alphabet2[random2.Next(alphabet2.Length)];
10     }
11     });
```

---

```
1 var alphabet2 = state.alphabet;
2 var random2 = state.random;
```

---

```
1 Span<int> span = stackalloc int[] { 1, 2, 3 };
2 int* pointer = stackalloc int[] { 4, 5, 6 };
```

---

```
1 fixed (int* ptr = value)
2 {
3
4 }
```

---

1

2

# CHAPTER 14

## Listing 14.1. A simple local method that accesses a local variable

```
1 static void Main()
2 {
3     int x = 10;
4     PrintAndIncrementX();
5     PrintAndIncrementX();
6     Console.WriteLine($"After calls, x = {x}");
7
8     void PrintAndIncrementX()
9     {
10         Console.WriteLine($"x = {x}");
11         x++;
12     }
13 }
```

---

```
1 static void Invalid()
2 {
3     for (int i = 0; i < 10; i++)
4     {
5         PrintI();
6     }
7
8     void PrintI() => Console.WriteLine(i);
9 }
```

---

```
1 static void Valid()
2 {
3     for (int i = 0; i < 10; i++)
4     {
5         PrintI();
6
7         void PrintI() => Console.WriteLine(i);
8     }
9 }
```

---

```
1 static void Invalid()
2 {
3     void PrintI() => Console.WriteLine(i);
4     int i = 10;
5     PrintI();
6 }
```

---

1

```
1 static void Invalid(ref int p)
2 {
3     PrintAndIncrementP();
4     void PrintAndIncrementP() =>
5         Console.WriteLine(p++);
6 }
```

---

1

```
1 static void Valid(ref int p)
2 {
3     PrintAndIncrement(ref p);
4     void PrintAndIncrement(ref int x) => Console.WriteLine(x++);
5
6 }
```

---

```
1 static void AttemptToReadNotDefinitelyAssignedVariable()
2 {
3     int i;
4     void PrintI() => Console.WriteLine(i);
5     PrintI();
6     i = 10;
7     PrintI();
8 }
```

---

1

2

```
1 static void DefinitelyAssignInMethod()
2 {
3     int i;
4     AssignI();
5     Console.WriteLine(i);
6     void AssignI() => i = 10;
7 }
```

---

1

2

3

```

1 class Demo
2 {
3     private readonly int value;
4
5     public Demo()
6     {
7         AssignValue();
8         void AssignValue()
9         {
10             value = 10;
11         }
12     }
13 }

```

1

## Listing 14.2. Local method modifying a local variable

```

1 static void Main()
2 {
3     int i = 0;
4     AddToI(5);
5     AddToI(10);
6     Console.WriteLine(i);
7     void AddToI(int amount) => i += amount;
8 }

```

## Listing 14.3. What Roslyn does with [listing 14.2](#)

```

1 private struct MainLocals
2 {
3     public int i;
4 }
5
6 static void Main()
7 {
8     MainLocals locals = new MainLocals();
9     locals.i = 0;
10    AddToI(5, ref locals);
11    AddToI(10, ref locals);
12    Console.WriteLine(locals.i);
13 }
14
15 static void AddToI(int amount, ref MainLocals locals)
16 {
17     locals.i += amount;
18 }

```

1

2

3

4

## Listing 14.4. Capturing variables from multiple scopes

```
1 static void Main()
2 {
3     DateTime now = DateTime.UtcNow;
4     int hour = now.Hour;
5     if (hour > 5)
6     {
7         int minute = now.Minute;
8         PrintValues();
9
10        void PrintValues() =>
11            Console.WriteLine($"hour = {hour}; minute = {minute}");
12    }
13 }
```

---

## Listing 14.5. What Roslyn does with [listing 14.4](#)

```
1 struct OuterScope
2 {
3     public int hour;
4 }
5 struct InnerScope
6 {
7     public int minute;
8 }
9
10 static void Main()
11 {
12     DateTime now = DateTime.UtcNow;
13     OuterScope outer = new OuterScope();
14     outer.hour = now.Hour;
15     if (outer.hour > 5)
16     {
17         InnerScope inner = new InnerScope();
18         inner.minute = now.Minute;
19         PrintValues(ref outer, ref inner);
20     }
21 }
22
23 static void PrintValues(
24     ref OuterScope outer, ref InnerScope inner)
25 {
26     Console.WriteLine($"hour = {outer.hour}; minute = {inner.minute}");
27 }
```

---

## Listing 14.6. Method group conversion of a local method

```
1 static void Main()
2 {
3     Action counter = CreateCounter();
```



```

4     counter();
5     counter();
6 }
7
8 static Action CreateCounter()
9 {
10     int count = 0;
11     return Count;
12     void Count() => Console.WriteLine(count++);
13 }

```

```

1 static Action CreateCounter()
2 {
3     int count = 0;
4     return () => Console.WriteLine(count++);
5 }

```

### Listing 14.7. What Roslyn does with [listing 14.6](#)

```

1 static void Main()
2 {
3     Action counter = CreateCounter();
4     counter();
5     counter();
6 }
7
8 static Action CreateCounter()
9 {
10     CountHolder holder = new CountHolder();
11     holder.count = 0;
12     return holder.Count;
13 }
14
15 private class CountHolder
16 {
17     public int count;
18
19     public void Count() => Console.WriteLine(count++);
20 }

```

### Listing 14.8. Implementing `Select` without local methods

```

1 public static IEnumerable<TResult> Select<TSource, TResult>(
2     this IEnumerable<TSource> source,
3     Func<TSource, TResult> selector)
4 {
5     Preconditions.CheckNotNull(source, nameof(source));

```

```

6     Preconditions.CheckNotNull(
7         selector, nameof(selector));
8     return SelectImpl(source, selector);
9 }
10
11 private static IEnumerable<TResult> SelectImpl<TSource, TResult>(
12     IEnumerable<TSource> source,
13     Func<TSource, TResult> selector)
14 {
15     foreach (TSource item in source)
16     {
17         yield return selector(item);
18     }
19 }

```

### Listing 14.9. Implementing `Select` with a local method

```

1 public static IEnumerable<TResult> Select<TSource, TResult>(
2     this IEnumerable source,
3     Func selector)
4 {
5     Preconditions.CheckNotNull(source, nameof(source));
6     Preconditions.CheckNotNull(selector, nameof(selector));
7     return SelectImpl(source, selector);
8
9     IEnumerable<TResult> SelectImpl(
10         IEnumerable validatedSource,
11         Func validatedSelector)
12     {
13         foreach (TSource item in validatedSource)
14         {
15             yield return validatedSelector(item);
16         }
17     }
18 }

```

```

1 static int? ParseInt32(string text)
2 {
3     int value;
4     return int.TryParse(text, out value) ? value : (int?) null;
5 }

```

```

1 static int? ParseInt32(string text) =>
2     int.TryParse(text, out int value) ? value : (int?) null;

```

```
1 static int? ParseAndSum(string text1, string text2) =>
2     int.TryParse(text1, out int value1) &&
3     int.TryParse(text2, out int value2)
4     ? value1 + value2 : (int?) null;
```

---

### Listing 14.10. Using an `out` variable in a constructor initializer

```
1 class ParsedText
2 {
3     public string Text { get; }
4     public bool Valid { get; }
5
6     protected ParsedText(string text, bool valid)
7     {
8         Text = text;
9         Valid = valid;
10    }
11 }
12
13 class ParsedInt32 : ParsedText
14 {
15     public int? Value { get; }
16
17     public ParsedInt32(string text)
18         : base(text, int.TryParse(text, out int parseResult))
19     {
20         Value = Valid ? parseResult : (int?) null;
21     }
22 }
```

---

```
1 byte b1 = 135;
2 byte b2 = 0x83;
3 byte b3 = 0b10000111;
```

---



```
1 byte b1 = 135;
2 byte b2 = 0x83;
3 byte b3 = 0b10000111;
4 byte b4 = 0b1000_0111;
```

---

```
1 int maxInt32 = 2_147_483_647;  
2 decimal largeSalary = 123_456_789.12m;  
3 ulong alternatingBytes = 0xff_00_ff_00_ff_00_ff_00;  
4 ulong alternatingWords = 0xffff_0000_ffff_0000;  
5 ulong alternatingDWords = 0xffffffff_00000000;
```

---

```
1 int wideFifteen = 1_____5;  
2 ulong notQuiteAlternatingWords = 0xffff_000_ffff_0000;
```

---

```
1 public void UnimplementedMethod() =>                                     1  
2     throw new NotImplementedException();  
3  
4 public void TestPredicateNeverCalledOnEmptySequence()  
5 {  
6     int count = new string[0]  
7         .Count(x => throw new Exception("Bang!"));  
8     Assert.AreEqual(0, count);                                           2  
9 }  
10  
11 public static T CheckNotNull<T>(T value, string paramName) where T : class  
12     => value ??  
13     throw new ArgumentNullException(paramName);  
14                                                                           3  
15 public static Name =>  
16     initialized  
17     ? data["name"]  
18     : throw new Exception("...");  
                                                                           4
```

---

```
1 int invalid = throw new Exception("This would make no sense");  
2 Console.WriteLine(throw new Exception("Nor would this"));
```

---

```
1 static T GetValueOrDefault<T>(IList<T> list, int index)  
2 {  
3     return index >= 0 && index < list.Count ? list[index] : default(T);  
4 }
```

---

```
1 public async Task<string> FetchValueAsync(  
2     string key,  
3     CancellationToken cancellationToken = default(CancellationToken))
```

---

```
1 public async Task<string> FetchValueAsync(  
2     string key, CancellationToken cancellationToken = default)
```

---

```
1 var intArray = new[] { default, 5 };  
2 var stringArray = new[] { default, "text" };
```

---

```
1 var invalid = default;  
2 var alsoInvalid = new[] { default };
```

---

### Listing 14.11. Specifying a default literal as a method argument

```
1 static void PrintValue(int value = 10)  
2 {  
3     Console.WriteLine(value);  
4 }  
5  
6 static void Main()  
7 {  
8     PrintValue(default);  
9 }
```

---

1

2

```
1 client.UploadCsv(table, null, csvData, options);
```

---

```
1 TableSchema schema = null;
2 client.UploadCsv(table, schema, csvData, options);
```

---

```
1 client.UploadCsv(table, schema: null, csvData, options);
```

---

```
1 void M(int x, int y, int z){}
2
3 M(5, z: 15, y: 10);
4 M(5, y: 10, 15);
5 M(y: 10, 5, 15);
```

---

1  
2  
3

## Listing 14.12. New constraints in C# 7.3

```
1 enum SampleEnum {}
2 static void EnumMethod<T>() where T : struct, Enum {}
3 static void DelegateMethod<T>() where T : Delegate {}
4 static void UnmanagedMethod<T>() where T : unmanaged {}
5 ...
6 EnumMethod();
7 EnumMethod();
8
9 DelegateMethod();
10 DelegateMethod();
11 DelegateMethod();
12
13 UnmanagedMethod<int>();
14 UnmanagedMethod<string>();
```

---

1  
2  
3  
4  
5

```
1 static void Method<T>(object x) where T : struct =>
2     Console.WriteLine($"{typeof(T)} is a struct");
3
4 static void Method<T>(string x) where T : class =>
5     Console.WriteLine($"{typeof(T)} is a reference type");
6 ...
7 Method<int>("text");
```

1  
2

```
1 [Demo]
2 private string name;
3 public string Name
4 {
5     get { return name; }
6     set { name = value; }
7 }
```

---

```
1 [field: Demo]
2 public string Name { get; set; }
```

---

# CHAPTER 15

## Listing 15.1. Initial model before C# 8

```
1 public class Customer
2 {
3     public string Name { get; set; }
4     public Address Address { get; set; }
5 }
6
7 public class Address
8 {
9     public string Country { get; set; }
10 }
```

```
1 Customer customer = ...;
2 Console.WriteLine(customer.Address.Country);
```

**Table 15.1. Support for nullability and non-nullability for reference and value types in C# 7 (view table figure)**

Nullable		Non-nullable
Reference types	Implicit	Not supported
Value types	Nullable<T> or ? suffix	Default

**Table 15.2. Support for nullability and non-nullability for reference and value types in C# 8 (view table figure)**

Nullable		Non-nullable
Reference types	No CLR type representation, but the ? suffix as an annotation	Default when nullable reference type support is enabled
Value types	Nullable<T> or ? suffix	Default



## Listing 15.2. Model with non-nullable properties everywhere

```
1 public class Customer
2 {
3     public string Name { get; set; }
4     public Address Address { get; set; }
5
6     public Customer(string name, Address address) =>
7         (Name, Address) = (name, address);
8 }
9
10 public class Address
11 {
12     public string Country { get; set; }
13
14     public Address(string country) =>
15         Country = country;
16 }
```

---

```
1 Customer customer = ...;
2 Console.WriteLine(customer.Address.Country);
```

---

```
1 string FirstOrSecond(string? first, string second) =>
2     first ?? second;
```

---

## Listing 15.3. Making the customer `Address` property nullable

```
1 public class Customer
2 {
3     public string Name { get; set; }
4     public Address? Address { get; set; }
5
6     public Customer(string name) =>
7         Name = name;
8 }
```

---

1

2

```
1 CS8602 Possible dereference of a null reference.
```

---

```
1 Console.WriteLine(customer.Address.Country);
```

---

### Listing 15.4. Safe dereferencing using the null conditional operator

```
1 Console.WriteLine(customer.Address?.Country ?? "(Address unknown)");
```

---

### Listing 15.5. Checking a reference with a local variable

```
1 Address? address = customer.Address; 1  
2 if (address != null)  
3 {  
4     Console.WriteLine(address.Country); 2  
5 }  
6 else  
7 {  
8     Console.WriteLine("(Address unknown)");  
9 }
```

---

### Listing 15.6. Checking a reference with repeated property access

```
1 if (customer.Address != null)  
2 {  
3     Console.WriteLine(customer.Address.Country);  
4 }  
5 else  
6 {  
7     Console.WriteLine("(Address unknown)");  
8 }
```

---

## Listing 15.7. Using the bang operator to satisfy the compiler

```
1 static void PrintLength(string? text) 1
2 {
3     if (!string.IsNullOrEmpty(text)) 2
4     {
5         Console.WriteLine($"{text}: {text!.Length}"); 3
6     }
7     else
8     {
9         Console.WriteLine("Empty or null");
10    }
11 }
```

---

## Listing 15.8. Using the bang operator in unit tests

```
1 public class Customer
2 {
3     public string Name { get; }
4     public Address? Address { get; }
5
6     public Customer(string name, Address? address)
7     {
8         Name = name ?? throw new ArgumentNullException(nameof(name));
9         Address = address;
10    }
11 }
12
13 public class Address
14 {
15     public string Country { get; }
16
17     public Address(string country)
18     {
19         Country = country ??
20             throw new ArgumentNullException(nameof(country));
21    }
22 }
23
24 [Test]
25 public void Customer_NameValidation()
26 {
27     Address address = new Address("UK");
28     Assert.Throws<ArgumentNullException>(
29         () => new Customer(null!, address));
30 }
```

---

1

```
1 () => new Customer(null, address)
```

---

```
1 [CanBeNull] DateTimeZone GetZoneOrNull([NotNull] string id);
```

---

```
1 string? a = ...;
2 if (!string.IsNullOrEmpty(a))
3 {
4     Console.WriteLine(a.Length);
5 }
6
7 object b = ...;
8 if (!ReferenceEquals(b, null))
9 {
10     Console.WriteLine(b.GetHashCode());
11 }
12
13 XElement c = ...;
14 string d = (string) c;
```

---

```
1 public class Wrapper<T>
2 {
3     public T Value { get; set; }
4 }
```

---

```
1 static void PrintLength(string text)
2 {
3     string validated =
4         text ?? throw new ArgumentNullException(nameof(text));
5     Console.WriteLine(validated.Length);
6 }
```

---

```
1 static void PrintLength(string text!)
2 {
3     Console.WriteLine(text.Length);
4 }
```

---

```

1 static double Perimeter(Shape shape)
2 {
3     switch (shape)
4     {
5         case null:
6             throw new ArgumentNullException(nameof(shape));
7         case Rectangle rect:
8             return 2 * (rect.Height + rect.Width);
9         case Circle circle:
10            return 2 * PI * circle.Radius;
11        case Triangle triangle:
12            return triangle.SideA + triangle.SideB + triangle.SideC;
13        default:
14            throw new ArgumentException(
15                $"Shape type {shape.GetType()} perimeter unknown",
16                nameof(shape));
17    }
18 }

```

---

### Listing 15.9. Converting a switch statement into a switch expression

```

1 static double Perimeter(Shape shape)
2 {
3     return shape switch
4     {
5         null => throw new ArgumentNullException(nameof(shape)),
6         Rectangle rect => 2 * (rect.Height + rect.Width),
7         Circle circle => 2 * PI * circle.Radius,
8         Triangle triangle =>
9             triangle.SideA + triangle.SideB + triangle.SideC,
10        _ => throw new ArgumentException(
11            $"Shape type {shape.GetType()} perimeter unknown",
12            nameof(shape))
13    };
14 }

```

---

```

1 double circumference = shape switch
2 {
3
4 };

```

---

1

### Listing 15.10. Using a switch expression to implement an expression-bodied method

```

1 static double Perimeter(Shape shape) =>
2     shape switch
3     {

```

```

4      null => throw new ArgumentNullException(nameof(shape)),
5      Rectangle rect => 2 * (rect.Height + rect.Width),
6      Circle circle => 2 * PI * circle.Radius,
7      Triangle triangle =>
8          triangle.SideA + triangle.SideB + triangle.SideC,
9      _ => throw new ArgumentException(
10         $"Shape type {shape.GetType()} perimeter unknown",
11         nameof(shape))
12 };

```

---

```

1 Rectangle rect => 2 * (rect.Height + rect.Width),
2 Circle circle => 2 * PI * circle.Radius,
3 Triangle triangle => triangle.SideA + triangle.SideB + triangle.SideC,

```

---

### Listing 15.11. Matching nested patterns

```

1 static double Perimeter(Shape shape) => shape switch
2 {
3     null => throw new ArgumentNullException(nameof(shape)),
4     Rectangle { Height: var h, Width: var w } => 2 * (h + w),
5     Circle { Radius: var r } => 2 * PI * r,
6     Triangle { SideA: var a, SideB: var b, SideC: var c } => a + b + c,
7     _ => throw new ArgumentException(
8         $"Shape type {shape.GetType()} perimeter unknown", nameof(shape))
9 };

```

---

```

1 Rectangle { Height: 0 } rect => $"Flat rectangle of width {rect.Width}"

```

---

```

1 public void Deconstruct
2     (out double sideA, out double sideB, out double sideC) =>
3     (sideA, sideB, sideC) = (SideA, SideB, SideC);

```

---

```

1 Triangle { SideA: var a, SideB: var b, SideC: var c } => a + b + c

```

---

```
1 Triangle (var a, var b, var c) => a + b + c
```

---

```
1 public class Customer
2 {
3     public string Name { get; set; }
4     public Address Address { get; set; }
5 }
6
7 public class Address
8 {
9     public string Country { get; set; }
10 }
```

---

### Listing 15.12. Matching customers against multiple patterns concisely

```
1 static void Greet(Customer customer)
2 {
3     string greeting = customer switch
4     {
5         { Address: { Country: "UK" } } =>
6             "Welcome, customer from the United Kingdom!",
7         { Address: { Country: "USA" } } =>
8             "Welcome, customer from the USA!",
9         { Address: { Country: string country } } =>
10            $"Welcome, customer from {country}!",
11         { Address: { } } =>
12            "Welcome, customer whose address has no country!",
13         { } =>
14            "Welcome, customer of an unknown address!",
15         _ =>
16            "Welcome, nullness my old friend!"
17     };
18     Console.WriteLine(greeting);
19 }
```

---

### Listing 15.13. Trimming the first and last character from a string with a range

```
1 string quotedText = "'This text was in quotes'";
2 Console.WriteLine(quotedText);
3 Console.WriteLine(quotedText.Substring(1..^1));
```

```
1 'This text was in quotes'
2 This text was in quotes
```

---

## Listing 15.14. Index and range literals

```
1 Index start = 2;
2 Index end = ^2;
3 Range all = ..;
4 Range startOnly = start..;
5 Range endOnly = ..end;
6 Range startAndEnd = start..end;
7 Range implicitIndexes = 1..5;
```

---

## Listing 15.15. Using indexer overloads for index and range in a string and a span

```
1 string text = "hello world";
2 Console.WriteLine(text[2]);
3 Console.WriteLine(text[^3]);
4 Console.WriteLine(text[2..7])
5
6 Span<int> span = stackalloc int[] { 5, 2, 7, 8, 2, 4, 3 };
7 Console.WriteLine(span[2]);
8 Console.WriteLine(span[^3]);
9 Span<int> slice = span[2..7];
10 Console.WriteLine(string.Join(", ", slice.ToArray()));
```

---

1

2

3

4

5

6

```
1 l
2 r
3 llo w
4 7
5 2
6 7, 8, 2, 4, 3
```

---



```
1 public interface IAsyncDisposable
2 {
3     Task DisposeAsync();
4 }
```

---

### Listing 15.16. Implementing IAsyncDisposable and calling it with using await

```
1 class AsyncResource : IAsyncDisposable
2 {
3     public async Task DisposeAsync()
4     {
5         Console.WriteLine("Disposing asynchronously...");
6         await Task.Delay(2000);
7         Console.WriteLine("... done");
8     }
9
10    public async Task PerformWorkAsync()
11    {
12        Console.WriteLine("Performing work asynchronously...");
13        await Task.Delay(2000);
14        Console.WriteLine("... done");
15    }
16 }
17 async static Task Main()
18 {
19     using await (var resource = new AsyncResource())
20     {
21         await resource.PerformWorkAsync();
22     }
23     Console.WriteLine("After the using await statement");
24 }
```

---

```
1 Performing work asynchronously...
2 ... done
3 Disposing asynchronously...
4 ... done
5 After the using await statement
```

---

```
1 foreach await (var item in asyncSequence)
2 {
3
4 }
```

---

```

1 public interface IAsyncEnumerable<out T>
2 {
3     IAsyncEnumerator<T> GetAsyncEnumerator();
4 }
5
6 public interface IAsyncEnumerator<out T>
7 {
8     Task<bool> WaitForNextAsync();
9     T TryGetNext(out bool success);
10 }

```

---

### Listing 15.17. Simplified RPC-based service for listing cities

```

1 public interface IGeoService
2 {
3     Task<ListCitiesResponse> ListCitiesAsync(ListCitiesRequest request);
4 }
5
6 public class ListCitiesRequest
7 {
8     public string PageToken { get; }
9     public ListCitiesRequest(string pageToken) =>
10         PageToken = pageToken;
11 }
12
13 public class ListCitiesResponse
14 {
15     public string NextPageToken { get; }
16     public List<string> Cities { get; }
17
18     public ListCitiesResponse(string nextPageToken, List<string> cities) =>
19         (NextPageToken, Cities) = (nextPageToken, cities);
20 }

```

---

### Listing 15.18. Wrapper around the RPC service to provide a simpler API

```

1 public class GeoClient
2 {
3     public GeoClient(IGeoService service) { ... }
4     public IAsyncEnumerable<string> ListCitiesAsync() { ... }
5 }

```

---

1  
2

### Listing 15.19. Using `foreach await` with a `GeoClient`

```

1 var client = new GeoClient(service);
2

```

```
3 foreach await (var city in client.ListCitiesAsync())
4 {
5     Console.WriteLine(city);
6 }
```

---

### Listing 15.20. Implementing `ListCitiesAsync` with an iterator

```
1 public async IEnumerable<string> ListCitiesAsync()
2 {
3     string pageToken = null;
4     do
5     {
6         var request = new ListCitiesRequest(pageToken);
7         var response = await service.ListCitiesAsync(request);
8         foreach (var city in response.Cities)
9         {
10             yield return city;
11         }
12         pageToken = response.NextPageToken;
13     } while (pageToken != null);
14 }
```

---

```
1 public interface IEnumerable<T>
2 {
3     IEnumerator<T> GetEnumerator();
4
5     int Count()
6     {
7         using (var iterator = GetEnumerator())
8         {
9             int count = 0;
10            while (iterator.MoveNext())
11            {
12                count++;
13            }
14        }
15    }
16    return count;
17 }
```

---

```
1 public class Point(int X, int Y, int Z);
```

---

```
1 public Point With(int X = this.X, int Y = this.Y, int Z = this.Z) =>  
2     new Point(X, Y, Z);
```

---

```
1 var newPoint = oldPoint.WithX(10).WithY(20);
```

---

```
1 var newPoint = oldPoint.With(X: 10, Y: 20);
```

---

```
1 var newPoint = oldPoint with { X = 10, Y = 20 };
```

---

```
1 Dictionary<string, List<DateTime>> entryTimesByName =  
2     new Dictionary<string, List<DateTime>>();
```

---

```
1 Dictionary<string, List<DateTime>> entryTimesByName = new();
```

---