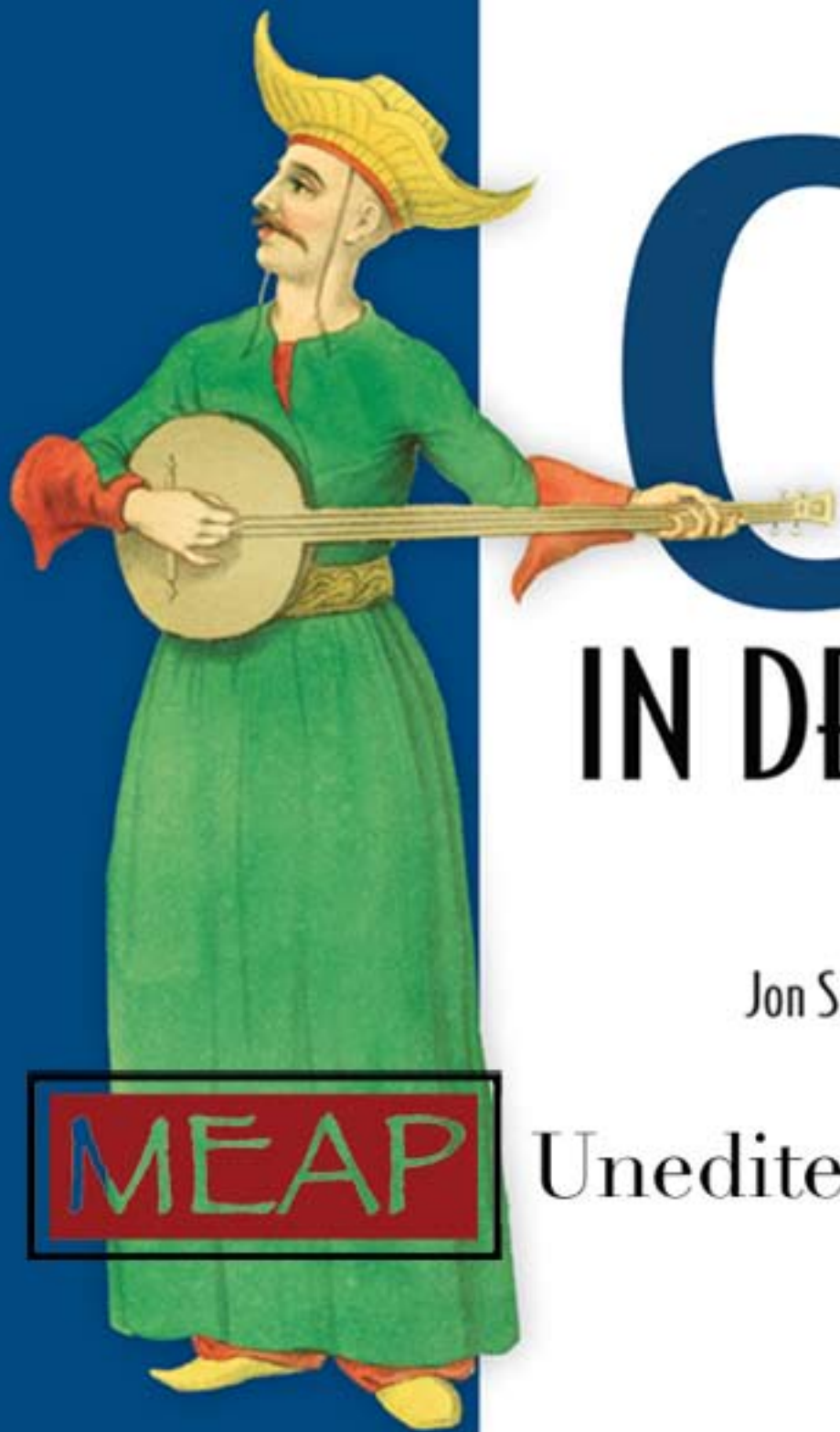


What You Need To Master (♯ 2 and 3)



# C♯

## IN DEPTH

Jon Skeet

MEAP

Unedited Draft

 MANNING



**MEAP Edition**  
**Manning Early Access Program**

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

Please post comments or corrections to the Author Online forum at  
<http://www.manning-sandbox.com/forum.jspa?forumID=352>

## Table of Contents

### PART 1: Preparing for the journey

Chapter 1: The changing face C# development

Chapter 2: Core foundations: building on C#1

### PART 2: C# 2 – Solving the issues of C# 1

Chapter 3: Parameterized typing with generics

Chapter 4: Saying nothing with nullable types

Chapter 5: Fast-tracked delegates

Chapter 6: Implementing iterators the easy way

Chapter 7: Streamlining development

### PART 3: C# 3 – Revolutionizing how we code

Chapter 8: Cutting fluff with a smart compiler

Chapter 9: Lambda expressions and expression trees

Chapter 10: Adding behavior without inheritance: extension methods

Chapter 11: Completing the jigsaw: LINQ

Chapter 12: LINQing outside the box: non-memory LINQ

Chapter 13: Elegant code in the new era

Appendix A: C# terminology glossary

Appendix B: .NET 2.0 enhancement summary

Appendix C: .NET 3.5 enhancement summary

Appendix D: Online references

# Chapter 1

## *The changing face of C# development*

The world is changing at a pace which is sometimes terrifying, and technology is one of the fastest-moving areas of that change. Computing in particular seems to push itself constantly, both in hardware and in software. While many older computer languages are like bedrocks, rarely changing beyond being consolidated in terms of standardization, newer ones are still evolving. C# is such a language, and the implications of this are double-edged. On the one hand, there's always more to learn – the feeling of having mastered the language is unlikely to last for long, with a “V next” always looming. However, the upside is that if you embrace the new features, and you're willing to change your coding style to adopt the new idioms, you'll discover a more expressive, powerful way of developing software.

If you're really anxious to get coding straight away, and if you're confident in your understanding of C# 1, feel free to skip to part 2 and dive in. However, there's always more to coding than just the technicalities, and in this part I will be providing background which I hope you'll find valuable in terms of understanding the bigger picture, the reasons *why* both the C# language and the .NET framework have changed in the ways that they have.

In this chapter, we'll have a sneak peek at a few of the features the rest of the book will cover. We'll see that while C# 2 fixed a lot of the issues people encountered when using C# 1, the ideas in C# 3 could significantly change the way we write code and even think about the problem in the first place. I'll put the changes into a historical context, guide you through the maze of version numbers, and talk about how the book is presented, to help you get as much out of it as possible. Let's start off by looking at how some code might evolve over time, taking advantage of new features as they become available

### *1.1 Evolution in action: examples of code change*

I've always dreamed of doing magic tricks, and for this one section I get to live that dream. This is the only time that I won't explain how things work, or try to go one step at a time. Quite the opposite, in fact – the plan is to impress rather than educate. If you read the whole of this section without getting at least a little bit excited about what C# 2 and 3 can do, maybe this book isn't for you. With any luck though, you'll be eager to get to the details of how the tricks work – to slow down the sleight of hand until it's obvious what's going on – and that's what the rest of the book is for.

I should warn you that the example is very contrived – clearly designed to pack as many new features into as short a piece of code as possible. From C# 2, we'll see generics, properties with different access modifiers for getters and setters, nullable types, and anonymous methods. From C# 3, we'll see automatically implemented properties, enhanced collection initializers, enhanced object initializers, lambda expressions, extension methods, implicit typing, and LINQ query expressions. There are of course many other new features, but it would be impossible to demonstrate them all together in a meaningful way. Even though you usually wouldn't use even this select set of features in such a compact space, I'm sure you'll recognize the general tasks as ones which *do* crop up very frequently in real life code.

As well as being contrived, the example is also clichéd – but at least that makes it familiar. Yes, it’s a product/name/price example, the e-commerce virtual child of “hello, world”.

In order to keep things simple, I’ve split the code up into sections. What we want to do is:

- Define a `Product` type with a name and a price in dollars, and a way of retrieving a hard-coded list of products
- Print out the products in alphabetical order
- Print out all the products costing more than \$10
- Consider the implication of representing products with unknown prices

We’ll look at each of these areas separately, and see how as we move forward in versions of C#, we can accomplish the same tasks more simply and elegantly than was previously possible. In each case, the changes to the code will be in a **bold font**. Let’s start with the `Product` type itself.

### *1.1.1 Defining the Product type*

We’re not looking for anything particularly impressive from the `Product` type – just encapsulation of a couple of properties. To make life simpler for demonstration purposes, this is also where we create a list of pre-defined products. We override `ToString` so that when we print out the products elsewhere, they show useful values. Listing 1.1 shows the type as it might be written in C# 1. We’ll then move on to see how the same effect could be achieved in C# 2, then C# 3. This is the pattern we will follow for each of the other pieces of code.

#### **Listing 1.1 The Product type (C# 1)**

```
using System.Collections;

public class Product
{
    string name;
    public string Name
    {
        get { return name; }
    }

    decimal price;
    public decimal Price
    {
        get { return price; }
    }

    public Product(string name, decimal price)
    {
        this.name = name;
        this.price = price;
    }

    public static ArrayList GetSampleProducts()
    {
        ArrayList list = new ArrayList();
```

```

        list.Add(new Product("Company", 9.99m));
        list.Add(new Product("Assassins", 14.99m));
        list.Add(new Product("Frogs", 13.99m));
        list.Add(new Product("Sweeney Todd", 10.99m));
        return list;
    }

    public override string ToString()
    {
        return string.Format("{0}: {1}", name, price);
    }
}

```

Nothing in listing 1.1 should be hard to understand – it’s just C# 1 code, after all. There are four limitations which it demonstrates, however:

- Neither `IList` nor `ArrayList` provide compile-time information about what’s in them. We could have accidentally added a string to it in `GetSampleProducts` and the compiler wouldn’t have batted an eyelid.
- We’ve provided public “getter” properties which means that if we wanted matching “setters”, they would have to be public too. In this case it’s not too much of a problem to use the fields directly, but it would be if we had validation that ought to be applied every time a value was set. A property setter would be natural, but we may not want to expose it to the outside world. We’d have to have a private `SetPrice` method or something similar, and that asymmetry is ugly.
- The variables themselves are available to the rest of the class. They’re private, but it would be nice to encapsulate them within the properties, to make sure they’re not tampered with other than *through* those properties.
- There’s quite a lot of fluff involved in creating the properties and variables – code which complicates the simple task of encapsulating a string and a decimal.

Let’s see what C# 2 can do to improve matters (changes are in bold):

**Listing 1.2 Strongly typed collections and private setters (C# 2)**  
**using System.Collections.Generic;**

```

public class Product
{
    string name;
    public string Name
    {
        get { return name; }
        private set { name = value; }
    }

    decimal price;
    public decimal Price
    {
        get { return price; }
        private set { price = value; }
    }
}

```

```

public Product(string name, decimal price)
{
    Name = name;
    Price = price;
}

public static List<Product> GetSampleProducts()
{
    List<Product> list = new List<Product>();
    list.Add(new Product("Company", 9.99m));
    list.Add(new Product("Assassins", 14.99m));
    list.Add(new Product("Frogs", 13.99m));
    list.Add(new Product("Sweeney Todd", 10.99m));
    return list;
}

public override string ToString()
{
    return string.Format("{0}: {1}", name, price);
}
}

```

The code hasn't changed much, but we've addressed two of the problems. We now have properties with private setters (which we use in the constructor), and it doesn't take a genius to guess that `List<Product>` is telling the compiler that the list contains products. Attempting to add a different type to the list would result in a compiler error. The change to C# 2 leaves only two of the original four difficulties unanswered. Listing 1.3 will show how C# 3 tackles these.

### Listing 1.3 Automatically implemented properties and simpler initialization (C# 3)

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

```

class Product
{
    public string Name { get; private set; }
    public decimal Price { get; private set; }

    public Product(string name, decimal price)
    {
        Name = name;
        Price = price;
    }

    Product()
    {
    }

    public static List<Product> GetSampleProducts()
    {
        return new List<Product>
        {

```

```

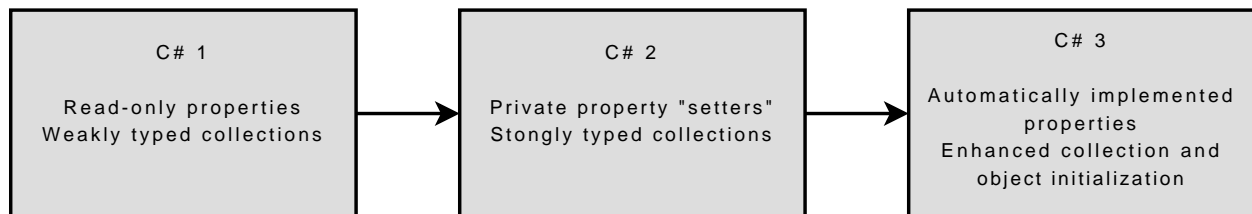
        new Product { Name="Company", Price = 9.99m },
        new Product { Name="Assassins", Price=14.99m },
        new Product { Name="Frogs", Price=13.99m },
        new Product { Name="Sweeney Todd", Price=10.99m }
    };
}

public override string ToString()
{
    return string.Format("{0}: {1}", Name, Price);
}
}

```

The properties now don't have any code (or visible variables!) associated with them and we're building the hard-coded list in a very different way. With no "name" and "price" variables to access, we're forced to use the properties everywhere in the class, improving consistency. We now have a private parameterless constructor for the sake of the new property-based initialization. In this example, we could actually have removed the public constructor completely, but it would make the class rather less useful in the real world.

Figure 1.1 shows a summary of how our Product type has evolved so far. I'll include a similar diagram after each task, so you can see the pattern of how C# 2 and 3 improve the code.



**Figure 1.1: Evolution of the Product type, showing greater encapsulation, stronger typing, and ease of initialization over time.**

So far the changes are relatively minimal. In fact the addition of generics (the `List<Product>` syntax) is probably the most important part of C# 2, but we've only seen part of its usefulness so far. There's nothing to get the heart racing yet, but we've only just started. Our next task is to print out the list of products in alphabetical order. That shouldn't be too hard...

### 1.1.2 *Sorting products by name*

The easiest way of displaying a list in a particular order is to sort the list and then run through it displaying items. In .NET 1.1, this involved using `ArrayList.Sort`, and in our case providing an `IComparer` implementation. We could have made the `Product` type implement `IComparable`, but we could only define one sort order that way, and it's not a huge stretch to imagine that we might want to sort by price at some stage as well as name. So, let's implement `IComparer`, sort the list, and then display it:

#### **Listing 1.4: Sorting an ArrayList using IComparer (C# 1)**

```

class ProductNameComparer : IComparer
{
    public int Compare(object x, object y)

```

```

    {
        Product first = (Product)x;
        Product second = (Product)y;
        return first.Name.CompareTo(second.Name);
    }
}
...

ArrayList products = Product.GetSampleProducts();
products.Sort(new ProductNameComparer());
foreach (Product product in products)
{
    Console.WriteLine (product);
}

```

The first thing to spot is that we've had to introduce an extra type to help us with the sorting. That's not a disaster, but it's a lot of code if we only want to sort by name in one place. Next, we see the casts in the Compare method. Casts are a way of telling the compiler that we know more information than it does – and that usually means there's a chance we're wrong. If the ArrayList we returned from GetSampleProducts *had* contained a string, that's where the code would go bang – when the comparison tried to cast the string to a Product.

We've also got a cast in the code which displays the sorted list. It's not obvious, because the compiler puts it in automatically, but the foreach loop implicitly casts each element of the list to Product. Again, that's a cast we'd ideally like to get rid of. Again, generics come to the rescue in C# 2. Here's the earlier code with the *only* change being the use of generics:

#### **Listing 1.5: Sorting a List<Product> using IComparer<Product> (C# 2)**

```

class ProductNameComparer : IComparer<Product>
{
    public int Compare(Product first, Product second)
    {
        return first.Name.CompareTo(second.Name);
    }
}
...

List<Product> products = Product.GetSampleProducts();
products.Sort(new ProductNameComparer());
foreach (Product product in products)
{
    Console.WriteLine (product);
}

```

The code for the comparer is simpler because we're given products to start with. No casting necessary. Similarly, the invisible cast in the foreach loop has gone. It's hard to tell the difference, given that it's invisible, but it really is gone. Honest. I wouldn't lie to you. At least, not in chapter 1...

That's an improvement, but it would be nice to be able to sort the products by simply specifying the comparison to make, without needing to implement an interface to do so. Listing 1.6 shows how to do precisely this, telling the Sort method how to compare two products using a delegate.

#### **Listing 1.6: Sorting a List<Product> using Comparison<Product> (C# 2)**

```
List<Product> products = Product.GetSampleProducts();
products.Sort(delegate(Product first, Product second)
    { return first.Name.CompareTo(second.Name); }
);
foreach (Product product in products)
{
    Console.WriteLine(product);
}
```

Behold the lack of the `ProductNameComparer` type. The statement in bold actually creates a delegate instance, which we provide to the Sort method in order to perform the comparisons. More on that feature (*anonymous methods*) in chapter 5. We've now fixed all the things we didn't like about the C# 1 version. That doesn't mean that C# 3 can't do better though. First we'll just replace the anonymous method with an even more compact way of creating a delegate instance, as shown in listing 1.7.

#### **Listing 1.7: Sorting using Comparison<Product> from a lambda expression (C# 3)**

```
List<Product> products = Product.GetSampleProducts();
products.Sort(
    (first, second) => first.Name.CompareTo(second.Name)
);
foreach (Product product in products)
{
    Console.WriteLine(product);
}
```

We've gained even more strange syntax (a *lambda expression*) which still creates a `Comparison<Product>` delegate just the same as listing 1.6 did, but this time in a shorter form. We haven't had to use the delegate keyword to introduce it, or even specify the types of the parameters. There's more though: with C# 3 we can easily print the names out in order without modifying the original list of products. Listing 1.8 shows this using the OrderBy method.

#### **Listing 1.8: Ordering a List<Product> using an extension method (C# 3)**

```
List<Product> products = Product.GetSampleProducts();

foreach (Product product in products.OrderBy(p => p.Name))
{
    Console.WriteLine (product);
}
```

We appear to be calling an OrderBy method, but if you look in MSDN you'll see that it doesn't even exist in `List<Product>`. We're able to call it due to the presence of an *extension method* which we'll see in more detail in chapter 10. We're not actually sorting the list "in place" any more, just retrieving the contents of the list in a particular order. Sometimes you'll need to change the actual list, sometimes an ordering without any other side-effects is better. The important point is that it's much more compact and readable

(once you understand the syntax, of course). We wanted to order the list by name, and that's exactly what the code says. It doesn't say to sort by comparing the name of one product with the name of another, like the C# 2 code did, or to sort by using an instance of another type which knows how to compare one product with another. It just says to order by name. The importance of this simplicity of expression is easy to underestimate, but it lies at the heart of what C# 3 has to offer.

We've seen a bit more of the power of C# 2 and 3 in this section, with quite a lot of (as yet) unexplained syntax, but even without understanding the details we can see the progress towards clearer, simpler code. Figure 1.2 shows that evolution.

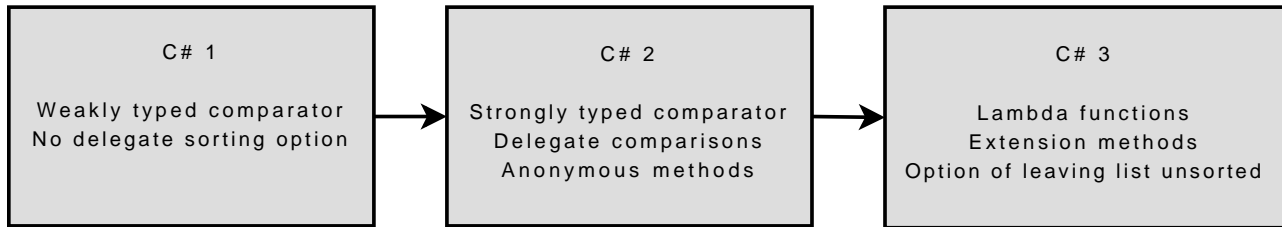


Figure 1.2: Features involved in making sorting easier in C# 2 and 3.

That's it for sorting. Let's do a different form of data manipulation now – querying.

### 1.1.3 Querying collections

Our next task is to find all the elements of the list which match a certain criterion – in particular, those with a price greater than \$10. In C# 1, we need to loop round, testing each element and printing it out where appropriate:

#### Listing 1.9 Looping, testing, printing out (C# 1)

```
ArrayList products = Product.GetSampleProducts();
foreach (Product product in products)
{
    if (product.Price > 10m)
    {
        Console.WriteLine(product);
    }
}
```

Okay, this is *not* difficult code to understand. However, it's worth bearing in mind how intertwined the three tasks are – looping with `foreach`, testing the criterion with `if`, then displaying the product with `Console.WriteLine`. The dependency is obvious because of the nesting. C# 2 lets us flatten things out a bit.

#### Listing 1.10 Separating testing from printing (C# 2)

```
List<Product> products = Product.GetSampleProducts();
Predicate<Product> test = delegate(Product p)
    { return p.Price > 10m; };
List<Product> matches = products.FindAll(test);

Action<Product> print = delegate(Product p)
    { Console.WriteLine (p); };
```

```
matches.ForEach (print);
```

I'm not going to claim this code is simpler than the C# 1 code – but it *is* a lot more powerful<sup>1</sup>. In particular, it makes it *very* easy to change the condition we're testing for and the action we take on each of the matches independently. The delegate variables involved (test and print) could be passed into a method – that same method could end up testing radically different conditions and taking radically different actions. Of course, we could have put all the testing and printing into one statement, as shown in listing 1.11.

#### Listing 1.11 Separating testing from printing redux (C# 2)

```
List<Product> products = Product.GetSampleProducts();  
products.FindAll (delegate(Product p) { return p.Price > 10;})  
    .ForEach (delegate(Product p) { Console.WriteLine(p); });
```

That's a bit better, but the `delegate(Product p)` is getting in the way, as are the braces. They're adding noise to the code, which hurts readability. I still prefer the C# 1 version, in the case where we only ever want to use the same test and perform the same action. (It may sound obvious, but it's worth remembering that there's nothing stopping us from using the C# 1 version when using C# 2 or 3. You wouldn't use a bulldozer to plant tulip bulbs, which is the kind of overkill we're using here.) C# 3 improves matters somewhat.

#### Listing 1.12 Testing with a lambda expression (C# 3)

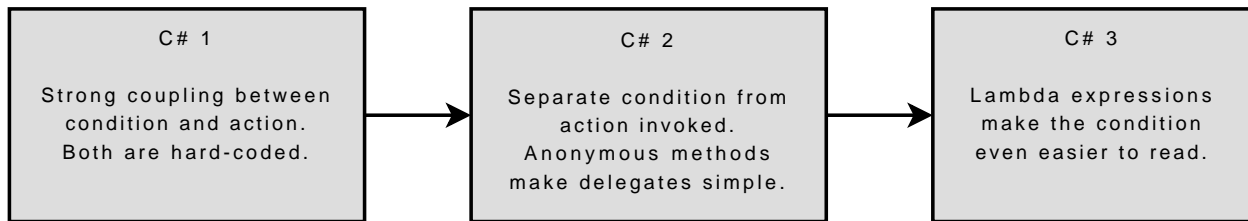
```
List<Product> products = Product.GetSampleProducts();  
foreach (Product product in products.Where(p => p.Price > 10))  
{  
    Console.WriteLine(product);  
}
```

The combination of the lambda expression putting the test in just the right place and a well-named method means we can *almost* read the code out loud and understand it without even thinking. We still have the flexibility of C# 2 – the argument to `Where` could come from a variable, and we could use an `Action<Product>` instead of the hard-coded `Console.WriteLine` call if we wanted to.

This task has really emphasized what we already knew from sorting –anonymous methods make writing a delegate simple, and lambda expressions are even more concise. In both cases, that brevity means that we can include the query or sort operation inside the first part of the `foreach` loop without losing clarity. Figure 1.3 summarizes the changes we've just seen.

---

<sup>1</sup> In some ways, this is cheating. We could have defined appropriate delegates in C# 1 and called them within the loop. The `FindAll` and `ForEach` methods in .NET 2.0 just help to encourage you to consider separation of concerns.



**Figure 1.3 Anonymous methods and lambda expressions aid separation of concerns and readability for C# 2 and 3.**

So, that's displayed the filtered list. Now let's consider a change to our initial assumptions about the data. What happens if we don't always know the price for a product? How can we cope with that within our Product class?

### 1.1.4 Representing an unknown price

I'm not going to present much code this time, but I'm sure it'll be a familiar problem to you, especially if you've done a lot of work with databases. Let's imagine our list of products contained not just products on sale right now, but ones which aren't available yet. In some cases, we may not know the price. If `decimal` were a reference type, we could just use `null` to represent the unknown price – but as it's a value type, we can't. How would you represent this in C# 1? There are three common alternatives:

- Create a reference type wrapper around `decimal`
- Maintain a separate boolean flag indicating whether or not the price is known
- Use a “magic value” (`decimal.MinValue` for example) to represent the unknown price

I hope you'll agree that none of these holds much appeal. Time for a little magic: we can solve the problem with the addition of a single extra character in the variable and property declarations. C# 2 makes matters a lot simpler by introducing the `Nullable<T>` structure and some syntactic sugar for it which lets us change the property declaration to:

```

decimal? price;
public decimal? Price
{
    get { return price; }
    private set { price =value; }
}
  
```

The constructor parameter changes to `decimal?` as well, and then we can pass in `null` as the argument, or say `Price = null;` within the class. That's a lot more expressive than any of the other solutions. The rest of the code just works “as is” – a product with an unknown price will be considered to be less expensive than \$10, which is probably what we'd want. To check whether a price is known or not we can compare it with `null`, or use the `HasValue` property – so to show all the products with unknown prices in C# 3, we'd write:

#### Listing 1.13 Displaying products with an unknown price (C# 2 and 3)

```

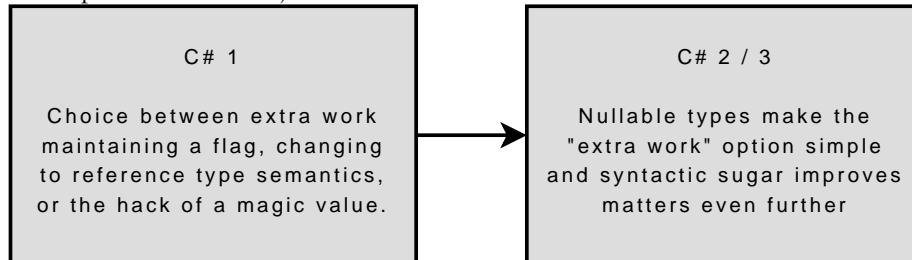
List<Product> products = Product.GetSampleProducts();
foreach (Product product in products.Where(p => p.Price==null))
  
```

```

{
    Console.WriteLine (product.Name) ;
}

```

The C# 2 code would be similar to listing 1.11, but using `return p.Price == null;` as the body for the anonymous method. There's no difference between C# 2 and 3 in terms of nullable types, so figure 1.4 represents the improvements with just two boxes.



**Figure 1.4** The options available for working round the lack of nullable types in C# 1, and the benefits of C# 2 and 3.

So, is that it? Everything we've seen so far is useful and important (particularly generics) but I'm not sure it really counts as *exciting*. There are some very cool things you can do with these features occasionally, but for the most part they're "just" making code a bit simpler, more reliable and more expressive. I value these things immensely, but they rarely impress me enough to call colleagues over to show how much can be done so simply. If you've seen any C# 3 code already, you were probably expecting to see something rather different – namely LINQ. This is where the fireworks start.

### 1.1.5 LINQ

LINQ, or Language Integrated Query, is what C# 3 is all about, at its heart. Whereas the features in C# 2 are arguably more about fixing annoyances in C# 1 than setting the world on fire, C# 3 is rather special. The reason none of the examples so far have used it is that they've all actually been simpler *without* it. That's not to say we couldn't use it anyway, of course – listing 1.12, for example, is equivalent to listing 1.14:

#### **Listing 1.14** First steps in LINQ: filtering a single collection and displaying the results

```

List<Product> products = Product.GetSampleProducts();
var filtered = from Product p in products
               where p.Price > 10
               select p;
foreach (Product product in filtered)
{
    Console.WriteLine (product);
}

```

Personally, I find the earlier listing easier to read, the only benefit to the LINQ version being that the "where" clause is simpler.

So if LINQ is no good, why is everyone making such a fuss about it? The first answer is that while it's not particularly suitable for simple tasks, it's very, *very* good for more complicated situations which would be very hard to read if written out in the equivalent method calls (and fiendish in C# 1 or 2). Let's make things just a little harder by introducing another type – `Supplier`. I haven't included the whole code here, but complete ready-to-compile code is provided on the book's web site. We'll concentrate on the fun stuff.

Each supplier has a Name (string) and a SupplierID (int). I've also added SupplierID as a property in Product and adapted the sample data appropriately. Admittedly that's not a very object oriented way of giving each product a supplier – it's much closer to how the data would be represented in a database. It makes this particular feature easier to demonstrate for now, but we'll see in chapter 12 that LINQ allows us to use a more natural model too.

Now let's look at the code to join the sample products with the sample suppliers (obviously based on the supplier ID), apply the same price filter as before to the products, sort by supplier name and then product name, and print out the name of both supplier and product for each match. That was a mouthful (fingerful?) to type, and in earlier versions of C# it would have been a nightmare to implement. In LINQ, it's almost trivial:

**Listing 1.15 A more complicated LINQ example: joining two collections, filtering them, ordering them on two criteria and then displaying a projection of the results**

```
List<Product> products = Product.GetSampleProducts();
List<Supplier> suppliers = Supplier.GetSampleSuppliers();
var filtered = from p in products
               join s in suppliers
               on p.SupplierID equals s.SupplierID
               where p.Price > 10
               orderby s.Name, p.Name
               select new {SupplierName=s.Name,
                          ProductName=p.Name};
foreach (var v in filtered)
{
    Console.WriteLine("Supplier={0}; Product={1}",
                      v.SupplierName, v.ProductName);
}
```

The more astute amongst you will have noticed that it looks remarkably like SQL<sup>2</sup>. Indeed, the reaction of many people on first hearing about LINQ (but before examining it closely) is to reject it as merely trying to put SQL into the language for the sake of talking to databases. Fortunately, LINQ has borrowed the syntax and some ideas from SQL, but as we've seen, you needn't be anywhere near a database in order to use it – none of the code we've run so far has touched a database at all. Indeed, we could be getting data from any number of sources. XML, for example. Suppose that instead of hard-coding our suppliers and products, we'd used the following XML file:

```
<?xml version="1.0"?>
<Data>
  <Products>
    <Product Name="Company" Price="9.99" SupplierID="1" />
    <Product Name="Assassins" Price="14.99" SupplierID="2" />
    <Product Name="Frogs" Price="13.99" SupplierID="1" />
    <Product Name="Sweeney Todd" Price="10.99" SupplierID="3" />
  </Products>

  <Suppliers>
```

---

<sup>2</sup> If you've ever worked with SQL in any form whatsoever but *didn't* notice the resemblance, I'm shocked.

```

    <Supplier Name="Solely Sondheim" SupplierID="1" />
    <Supplier Name="CD-by-CD-by-Sondheim" SupplierID="2" />
    <Supplier Name="Barbershop CDs" SupplierID="3" />
  </Suppliers>
</Data>

```

Well, the file is simple enough, but what's the best way of extracting the data from it? How do we query it? Join on it? Surely it's going to be somewhat harder than listing 1.14, right? Let's see how much work we have to do in LINQ to XML:

**Listing 1.16 LINQ to XML equivalent of listing 1.15: loading data from an XML file, querying it and displaying the results**

```

XDocument doc = XDocument.Load("data.xml");
var filtered = from p in doc.Descendants("Product")
               join s in doc.Descendants("Supplier")
               on (int)p.Attribute("SupplierID")
                  equals (int)s.Attribute("SupplierID")
               where (decimal)p.Attribute("Price") > 10
               orderby (string)s.Attribute("Name"),
                       (string)p.Attribute("Name")
               select new
               {
                 SupplierName = (string)s.Attribute("Name"),
                 ProductName = (string)p.Attribute("Name")
               };
foreach (var v in filtered)
{
    Console.WriteLine("Supplier={0}; Product={1}",
                      v.SupplierName, v.ProductName);
}

```

Well, it's not quite as straightforward, because we need to tell the system how it should understand the data (in terms of what attributes should be used as what types, etc) – but it's not far off. In particular, there's an obvious relationship between each part of the two listings. If it weren't for the line length limitations of books, you'd see an exact line-by-line correspondence between the two queries.

Impressed yet? Not quite convinced? Let's put the data where it's much more likely to be – in a database. There's some work (much of which can be automated) to let LINQ to SQL know about what to expect in what table, but it's all fairly straightforward. Here's the querying code:

**Listing 1.17 LINQ to SQL equivalent of listing 1.15: loading data from a database, querying it and displaying the results**

```

LinqDemoDb db = new LinqDemoDb(connectionString);

var filtered = from p in db.Products
               join s in db.Suppliers
               on p.SupplierID equals s.SupplierID
               where p.Price > 10
               orderby s.Name, p.Name
               select new

```

```

        {
            SupplierName = s.Name,
            ProductName = p.Name
        };
foreach (var v in filtered)
{
    Console.WriteLine("Supplier={0}; Product={1}",
        v.SupplierName, v.ProductName);
}

```

By now, this should be looking incredibly familiar. Everything below the “join” line is cut and paste directly from listing 1.14 with no changes. That’s impressive enough, but if you’re performance conscious you may be wondering why we would want to pull down all the data from the database and then apply these .NET queries and orderings. Why not get the database to do it? That’s what it’s good at, isn’t it? Well indeed – and that’s exactly what LINQ to SQL does. The code in listing 1.16 issues a database request which is basically the query translated into SQL. Even though we’ve *expressed* the query in C# code, it’s been *executed* as SQL.

We’ll see later that the way this query joins isn’t how we’d normally use LINQ to SQL – there’s a more relation-oriented way of approaching it when the schema and the entities know about the relationship between suppliers and products. The result is the same, however, and it shows just how similar LINQ to Objects (the in-memory LINQ operating on collections) and LINQ to SQL can be.

It’s important to understand that LINQ is flexible, too: you can write your own query translators. It’s not easy, but it can be well worth it. For instance, here’s an example using Amazon’s web service to query its available books:

```

var query =
    from book in new LinqToAmazon.AmazonBookSearch()
    where
        book.Title.Contains("ajax") &&
        (book.Publisher == "Manning") &&
        (book.Price <= 25) &&
        (book.Condition == BookCondition.New)
    select book;

```

This example was taken from the introduction<sup>3</sup> to “LINQ to Amazon” which is a LINQ provider written as an example for the “LINQ in Action” book. The query is easy to understand, and written in what appears to be “normal” C# 3 – but the provider is translating it into a web service call. How cool is that?

Hopefully by now your jaw is suitably close to the floor – mine certainly was the first time I tried an exercise like the database one we’ve just seen, when it worked pretty much first time. As I said, I won’t give any details of exactly what’s going on yet – but at least you know what to look forward to. Now that we’ve seen a little bit of the evolution of the C# language, it’s worth taking a little history lesson to see how other products and technologies have progressed in the same timeframe.

---

<sup>3</sup> <http://linqinaction.net/blogs/main/archive/2006/06/26/Introducing-Linq-to-Amazon.aspx>

## 1.2 *A brief history of C# (and related technologies)*

When I was learning French and German at school, the teachers always told me that I would never be proficient in those languages until I started *thinking* in them. Unfortunately I never achieved that goal, but I *do* think in C# (and a few other languages)<sup>4</sup>. There are people who are quite capable of programming reasonably reliably in a computer language without ever getting *comfortable* (or even intimate) with it. They will always write their code with an accent, usually one reminiscent of whatever language they *are* comfortable in.

While you can learn the mechanics of C# without knowing anything about the context in which it was designed, you'll have a closer relationship with it if you understand why it looks the way it does – its ancestry, effectively. The technological landscape and its evolution have a significant impact on how both languages and libraries evolve, so let's take a brief walk through C#'s history, seeing how it fits in with the stories of other technologies, both those from Microsoft and those developed elsewhere. This is by no means a comprehensive history of computing at the end of the 20th century and the start of the 21st – any attempt at such a history would take a whole (large) book in itself. However, I've included the products and technologies which I believe have most strongly influenced .NET and C# in particular.

### 1.2.1 *The world before C#*

We're actually going to start with Java. While it would be a stretch to claim that C# and .NET definitely wouldn't have come into being without Java, it would also be hard to argue that it had no effect. Java 1.0 was released in January 1996, and the world went applet mad. Briefly. Java was very slow (at the time it was 100% interpreted) and most of the applets on the web were fairly useless. The speed gradually improved as just-in-time compilers (JITs) were introduced, and developers started looking at using Java on the server side instead of on the client. Java 1.2 (or Java 2, depending on whether you talk developer version numbers or marketing version numbers) overhauled the core libraries significantly, the servlet API and JavaServer Pages took off, and Sun's Hotspot engine boosted the performance significantly. Java is reasonably portable, despite the "write once, debug everywhere" skit on Sun's catchphrase of "write once, run anywhere". The idea of letting coders develop enterprise Java applications on Windows with friendly IDEs and then deploy (without even recompiling) to powerful Unix servers was a compelling proposition – and clearly something of a threat to Microsoft.

Microsoft created their own JVM, which had reasonable performance and a *very* fast start-up time, and even released an IDE for it, named J++. However, they introduced incompatible extensions into their platform, and Sun sued Microsoft for violating licensing terms, starting a *very* long (and frankly tedious) legal battle. The main impact of this legal battle was felt long before the case was concluded – while the rest of the world moved onto Java 1.2 and beyond, Microsoft's version of Java stayed at 1.1, which made it effectively obsolete pretty rapidly. It was clear that whatever Microsoft's vision of the future was, Java itself was unlikely to be a major part of it.

In the same period, Microsoft's Active Server Pages (ASP) gained popularity too. After an initial launch in December 1996, two further versions were released in 1997 and 2000. ASP made dynamic web development much simpler for developers on Microsoft servers, and eventually third parties ported it to non-Windows platforms. Despite being a great step forward in the Windows world, ASP didn't tend to promote

---

<sup>4</sup> Not all the time, I hasten to add. Only when I'm coding.

the separation of presentation logic, business logic and data persistence which most of the vast array of Java web frameworks encouraged.

### *1.2.2 C# and .NET are born*

C# and .NET were properly unveiled at PDC (Professional Developers Conference) in July 2000, although some elements had been preannounced before then, and there had been talk about the same technologies under different names (including COOL, COM3, Lightning) for a long time. Not that Microsoft hadn't been busy with other things, of course – that year also saw both Windows Me and Windows 2000 being released, with the latter being wildly successful compared with the former.

Microsoft didn't "go it alone" with C# and .NET, and indeed when the specifications for C# and the Common Language Infrastructure (CLI) were submitted to ECMA (an international standards body), they were co-sponsored by Hewlett-Packard and Intel along with Microsoft. ECMA ratified the specification (with some modifications) and later versions of C# and the CLI have gone through the same process. C# and Java are "open" in different ways, with Microsoft favoring the standardization path and Sun gradually open-sourcing Java and allowing or even encouraging other Java runtime environments. There are alternative CLI and C# implementations, the most visible being the Mono project<sup>5</sup>, but they don't generally implement the whole of what we think of as the .NET framework. Commercial reliance on and support of non-Microsoft implementations is small, outside of Novell, which sponsors the Mono project.

Although C# and .NET weren't released until 2002 (along with Visual Studio .NET 2002), betas were available long before then, and by the time everything was official, C# was already a popular language. ASP.NET was launched as part of .NET 1.0, and it was clear that Microsoft had no plans to do anything more with either "ASP Classic" or "VB Classic" – much to the annoyance of many VB6 developers. While VB.NET *looks* similar to VB6, there are enough differences to make the transition a non-trivial one – not least of which is learning the .NET framework. Many developers have decided to go straight from VB6 to C#, for various reasons.

### *1.2.3 Minor updates with .NET 1.1 and the first major step: .NET 2.0*

As is often the case, the 1.0 release was fairly quickly followed by .NET 1.1, which launched with Visual Studio .NET 2003 and included C# 1.2. There were few significant changes to either the language or the framework libraries – in some senses it was more of a service pack than a truly new release. Despite the small number of changes, it's rare to see anyone using .NET 1.0 at the time of writing, although 1.1 is still very much alive and kicking, partly due to the OS requirements of 2.0.

While Microsoft was busy bringing its new platform to the world, Sun (and its other significant partners, including IBM) hadn't left Java stagnating. Not quite, anyway. Java 1.5 (Java 5 for the marketing folk amongst you) was launched in September 2004, with easily the largest set of language enhancements in any Java release, including generics, enums (supported in a very cool way – far more object-oriented than the "named numbers" that C# provides), an enhanced for loop (`foreach` to you and me), annotations (read: attributes), "varargs" (broadly equivalent to parameter arrays of C# – the `params` modifier), and automatic boxing/unboxing. It would be foolish to suggest that all of these enhancements were due to C# having taken off (after all, putting generics into the language had been talked about since 1997) but it's also worth acknowledging the competition for the mindshare of developers. For Sun, Microsoft and other players, it's

---

<sup>5</sup> <http://www.mono-project.com>

not just about coming up with a great language: it's about persuading developers to write software for their platform.

C# and Java took the same approach to features from C++ such as templates and macros – features which can be very powerful, but also often make code harder to understand. Both Java and C# shipped without the functionality to start with, and then worked out ways of implementing it to a limited extent, providing as much value as possible with as few risks and drawbacks as possible. The results for generics look quite similar on the most superficial level, but differ significantly under the surface, as we'll see in chapter 3.

### **The pioneering role of Microsoft Research**

Microsoft Research are responsible for some of the new directions for .NET and C#. They published a paper on .NET generics as early as May 2001 (yes, even before .NET 1.0 had been released!) and worked on an extension called C $\omega$  (pronounced C omega), which included – amongst other things – some of the ideas which later formed LINQ. Another C# extension, Spec#, adds contracts to C#, allowing the compiler to do more verification automatically. We will have to wait and see whether any or all of the ideas of Spec# eventually become part of C# itself.

C# 2 was released in November 2005, as part of .NET 2.0 and alongside Visual Studio 2005 and VB8. Visual Studio became more productive to work with as an IDE – particularly now that refactoring was finally included – and the significant improvements to both the language and the platform were warmly welcomed by most developers.

As a sign of just how quickly the world is moving on – and of how long it takes to actually bring a product to market – it's worth noting that the first announcements about C# 3 were made at the PDC in September 2005, which was two months *before* C# 2 was released. The sad part is that while it seems to take two years to bring a product from announcement to market, it seems that the industry takes another year or two – at least – to start widely embracing it. As mentioned earlier, many companies are only now transitioning from .NET 1.1 to 2.0. We can only hope that it will be a shorter path to widespread adoption of .NET 3.0 and 3.5. (C# 3 goes along with .NET 3.5. I'll talk about the version numbers in the next section.)

One of the reasons .NET 2.0 took so long to come out is that it was being embedded within SQL Server 2005, with the obvious robustness and reliability concerns that go hand in hand with such a system. This allows .NET code to execute right inside the database, with potential for much richer logic to sit so close to the data. Database folk tend to be rather cautious, and only time will tell how widely this ability is used – but it's a powerful tool to have available if you find you need it.

### *1.2.4 “Next generation” products*

In November 2006 (a year after .NET 2.0 was released), Microsoft launched Vista, Office 2007 and Exchange Server 2007. This included launching .NET 3.0, which comes pre-installed on Vista. Over time, this is likely to aid adoption of .NET client applications for two reasons. Firstly, the old “.NET isn't installed on all computers” objection will become less relevant – you can safely assume that if the user is running Vista, they'll be able to run a .NET application. Secondly, Windows Presentation Foundation (WPF) is now the rich client platform of choice for developers in Microsoft's view – and it's only available from .NET.

Again, while Microsoft were busy with Vista and other products, the rest of the world was innovating too. Lightweight frameworks have been gaining momentum, and Object Relational Mapping (ORM) now has a significant developer “mindshare”, partly due to high quality free frameworks such as Hibernate. The SQL aspect of LINQ is much more than just the querying side we've seen so far, and marks a more definite step from Microsoft than its previous lukewarm ventures into this area, such as ObjectSpaces. Only time will tell

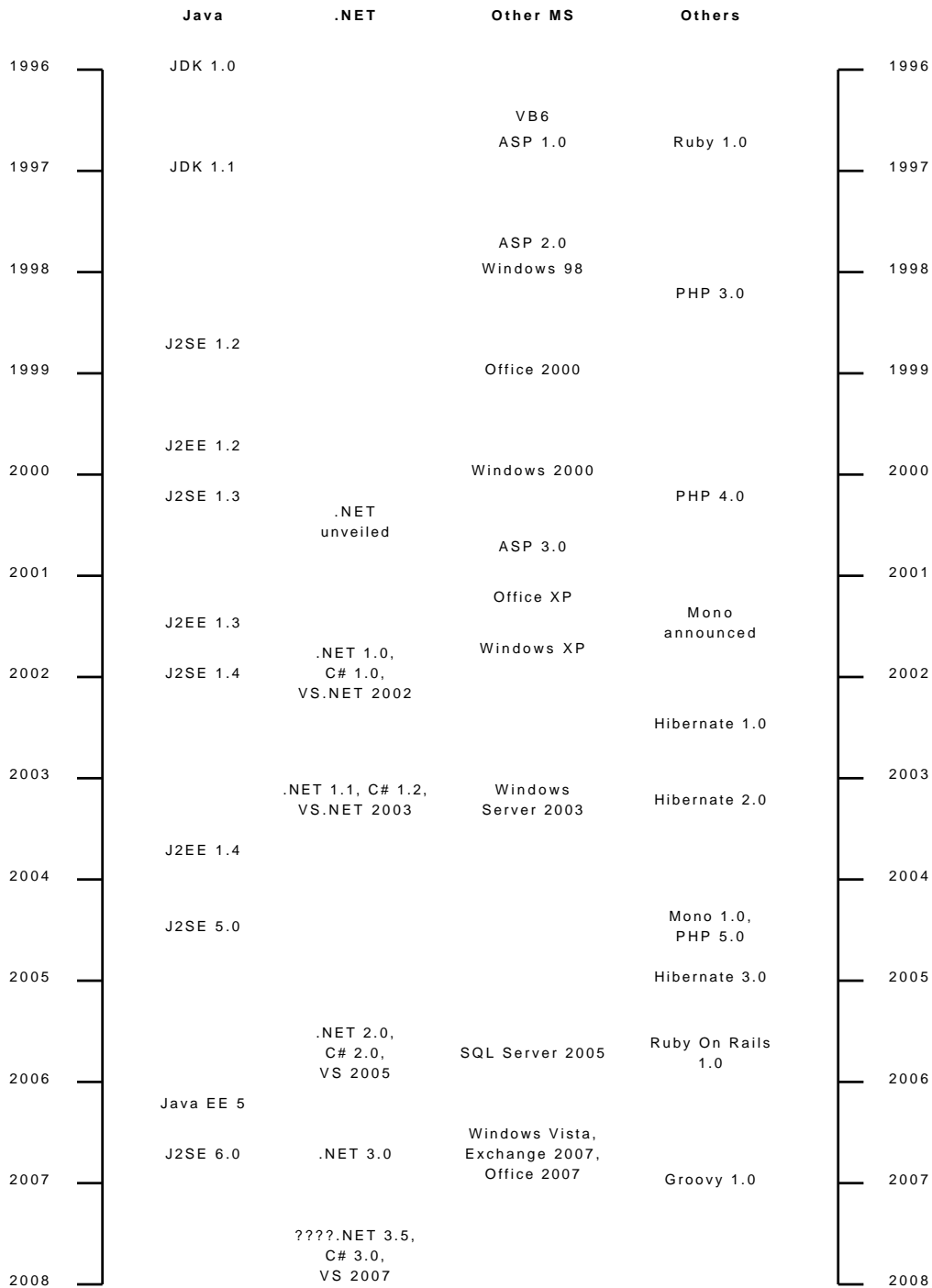
whether LINQ hits the elusive sweet spot of making database access truly simple – it’s certainly very promising.

Dynamic languages have also been increasingly important, with many options vying for developers’ attention. Ruby – and particularly the Ruby on Rails framework – has had a large impact (with ports for Java and .NET), and other projects such as Groovy on the Java platform and IronPython on .NET gaining support. Microsoft have released the Dynamic Runtime Library (DLR) which is a layer on top of the CLR to make it more amenable to dynamic languages. This is part of the Silverlight product, another battleground but this time for Rich Internet Applications (RIA), where Microsoft is competing with Adobe Flex and Sun’s JavaFX.

The last collection of items in this somewhat potted history is .NET 3.5, C# 3, VB9 and Visual Studio 2008. (Still to be written as they’re released...)

### *1.2.5 Historical perspective and the fight for developer support*

It’s hard to describe all of these strands interweaving through history and yet keep a bird’s-eye view of the period. Figure 1.5 shows a collection of timelines with some of the major milestones described earlier, within different technological areas. The list is not comprehensive, of course, but it gives some indication of which product versions were competing at different times.



**Figure 1.5 Timeline of releases for C#, .NET and related technologies.**

There are many ways to look at technological histories, and many untold stories influencing events behind the scenes. It's possible that this retrospective over-emphasizes the influence of Java on the development of .NET and C#, and that may well partly be due to my mixed allegiances to both technologies. However, to me it seems that the large wars for developer support are between the following camps:

- Native code (primarily C and C++) developers, who will have to be convinced about the reliability and performance of managed code before changing their habits. C++/CLI is the obvious way of dipping a

toe in the water here, but its popularity may not be all that Microsoft had hoped for.

- VB6 developers who may have antipathy towards Microsoft for abandoning their preferred platform, but will need to decide which way to jump sooner or later – and .NET is the most obvious choice for most people at this stage. Some may cross straight to C#, others making the smaller move to VB.NET.
- Scripting and dynamic language developers who value the immediacy of changes. Familiar languages running on managed platforms can act as Trojan horses here, encouraging developers to learn the associated frameworks for use in their dynamic code, which then lowers the barrier to entry for learning the traditional object oriented languages for the relevant platform. The IronPython programmer of today may well become the C# programmer of tomorrow.
- “Traditional” managed developers, primarily writing C#, VB.NET or Java. Here the war is not about whether running under some sort of managed environment is a good thing or not, but *which* managed environment to use. The battlegrounds are primarily in tools, portability, performance, and libraries, all of which have come on in leaps and bounds. Competition between different .NET languages is partly internal to Microsoft, with each team wanting its own language to have the best support – and features developed primarily for one language can often be used by another in the fullness of time.
- Web developers have already had to move from static HTML to dynamically generated content, through a nicer user experience involving developer pain with AJAX, and now the age of rich internet applications is upon us, with three very significant contenders in Microsoft, Adobe and Sun. At the time of writing, it’s too early to tell whether there will be a clear winner here or whether the three can all garner enough support to make them viable for a long time to come. Although it’s possible to use a .NET-based RIA solution with a Java-based server to some extent, the development process is significantly easier when technologies are aligned, so capturing the market here is important for all parties.

One thing is clear from all of this – it’s a very good time to be a developer. Lots of companies are investing a lot of time and money in making software development a fun and profitable industry to be in. Given the changes we’ve seen over the last decade or so, it’s very hard to predict what programming will look like in another decade, but it’ll be a fantastic journey getting there.

I mentioned earlier that C# 3 is effectively part of .NET 3.5. The subject of the version numbers chosen by Microsoft and what’s in which version is a slightly convoluted one, but it’s important that we all agree on what we mean when we talk about a particular version. Fortunately, having taken a broad look at lots of technologies in this section, in the next one we’re just going to focus on .NET again – there are enough version numbers around to make that quite confusing enough, thank you very much.

### *1.2.6 Untangling version number chaos*

A newcomer to the industry might think that coming up with version numbers would be easy. You start with 1, then move onto 2, then 3 in a logical progression, right? If only that were the case... Software products and projects of all natures like to keep minor version changes distinct from major ones, and then there are patch levels, service packs, build numbers and so forth. In addition, there are the codenames which are widely used and then abandoned, much to the frustration of “bleeding edge” book authors and publishers. Fortunately from the point of view of C# *as a language* we can make life reasonably straightforward.

#### **Keeping it simple: C# 1, C# 2 and C# 3**

Throughout this book, I’ll refer to C# versions as just 1, 2 and 3. There’s little point in distinguishing between the two 1.x versions, and no point in adding a cumbersome extra “.0” every time I refer to the different versions – which of course I’ll be doing quite a lot.

We don't just need to keep track of the language, unfortunately. There are five things we're interested in, when it comes to versioning.

1. "The .NET framework" (framework libraries and CLR as an installable package)
2. Framework libraries (the types available within the framework)
3. The CLR
4. C# (the language)
5. Visual Studio – version number and codename

Just for kicks, we'll throw in the Visual Basic numbering and naming too. (Visual Studio is abbreviated to VS and Visual Basic is abbreviated to VB for reasons of space.) Table 1.1 shows the different version numbers

**Table 1.1 Cross-reference table for versions of different products and technologies**

.NET	Framework libraries (max)	CLR	C#	Visual Studio	Visual Basic
1.0	1.0	1.0	1.0	VS.NET 2002 (no codename)	VB.NET 7.0
1.1	1.1	1.1	1.2 <sup>6</sup>	VS.NET 2003 (Everett)	VB.NET 7.1
2.0	2.0	2.0	2.0	VS 2005 (Whidbey)	VB 8.0
3.0	3.0	2.0	2.0	VS 2005 (extension previews), VS 2008 (full support)	VB 8.0
3.5	3.5	2.0	3.0	VS 2008 (Orcas)	VB 9.0

Note how both Visual Studio and Visual Basic lost the ".NET" moniker between 2003 and 2005, indicating Microsoft's emphasis on this being *the* tool for Windows development, as far as it's concerned.

As you can see, so far the version of the overall framework has followed the libraries exactly. However, it would be possible for a new version of the CLR with more capabilities to still be released with the existing libraries, so we could (for instance) have .NET 4.0 with libraries from 3.5, a CLR 3.0 and a C# 3 compiler. Let's hope it doesn't come to that. As it is, Microsoft have already confounded developers somewhat with the last two lines of the table.

.NET 3.0 is really just the addition of four libraries: Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WF<sup>7</sup>) and Windows CardSpace. None of the existing library classes were changed, and neither was the CLR, nor any of the languages targeting the CLR, so creating a whole new major version number for this feels a bit over the top.

Next comes .NET 3.5. This time as well as completely new classes (notably LINQ) there are many enhancements to the *base class libraries* (BCL – types within the namespaces such as System, System.IO; the core of the framework libraries). There's a new version of C#, without which this book would be considerably shorter, and a new version of Visual Studio to support that and VB 9.0. Apparently all of that isn't worth a major version number change though. There are service packs for both .NET 2.0 and 3.0, and both service packs ship with Visual Studio 2008 – so while you can target .NET 2.0 and 3.0 with the latest

---

<sup>6</sup> I've no idea why this isn't 1.1. I only discovered that it was 1.2 while researching this book. That's the numbering according to Microsoft's version of the specification, at least. I decided not to confuse matters further by also including the ECMA-334 edition number here, although that's another story in its own right.

<sup>7</sup> Not WWF due to wrestling and wildlife conflicts.

and greatest IDE (as well as 3.5, of course) you should be aware that what you'll *really* be compiling and running against is 2.0SP1, 3.0SP1 or 3.5.

Okay, rant over. It's only version numbers, after all – but it *is* important to understand what each version means, if for no other reason than communication. If someone says they're using “3.0” you need to check whether they mean C# 3 or .NET 3.0.

If all this talk of history and versioning is making you want to get back onto the familiar ground of actual programming, don't worry – we're nearly there. Indeed, if you fancy writing some code right now, the next section invites you to do just that, as I introduce the style I'll be using for most of the examples in this book.

## 1.3 Fully functional code in snippet form

One of the challenges when writing a book about a computer language (other than scripting languages) is that complete programs – ones that the reader can compile and run with no source code other than what's presented – get pretty long pretty quickly. I wanted to get round this, to provide you with code which you could easily type in and experiment with: I believe that actually *trying* something is a much better way of learning about it than just reading.

The solution I've come up with isn't applicable to all situations, but it will serve us well for most of the example code. It would be awful to use for “real” development, but is specifically tailored to the context we're working in: presenting and playing with code which can be compiled and run with the minimal amount of fuss. That's not to say you should only use it for experimentation when reading this book – I've found it useful as a general way of testing the behavior of small pieces of code.

### 1.3.1 Snippets and their expansions

With the right assembly references and the right `using` statements, you can accomplish quite a lot in a fairly short amount of C# code – but it's the fluff involved in writing those `using` statements, then declaring a class, then declaring a `Main` method before you've even written the first line of *useful* code which is the killer. Occasionally another method or type is required, but that can be worked around. Essentially, the snippets I provide ignore the fluff that gets in the way of simple programs, concentrating on the important part. So, for example, suppose I presented the following snippet:

#### **Listing 1.18 The first snippet, which simply displays two words on separate lines.**

```
foreach (string x in new string[] { "Hello", "There" })
{
    Console.WriteLine (x);
}
```

That clearly won't compile on its own – there's no class declaration, for a start. The code from listing 1.18 corresponds to the full program shown in listing 1.19:

#### **Listing 1.19 The expanded version of listing 1.18, where the snippet has been placed in context to create a full program.**

```
using System;
public class Snippet
{
    static void Main(string[] args)
```

```

    {
        foreach (string x in new string[] { "Hello", "There" })
        {
            Console.WriteLine (x);
        }
    }
}

```

Occasionally extra methods or even types are required, with a bit of code in the Main method to access them. I indicate this by listing the non-Main code, then an ellipsis (...) and then the Main code. So this:

**Listing 1.20 A code snippet with an extra method, called within the Main method.**

```

static string[] GetGreetingWords()
{
    return new string[] { "Hello", "There" }
}

...

foreach (string x in GetGreetingWords())
{
    Console.WriteLine (x);
}

```

would turn into this:

**Listing 1.21 The first expanded version of listing 1.20. The extra method has been placed directly within the class, and the rest of the code is within the Main method.**

```

using System;
public class Snippet
{
    static string[] GetGreetingWords()
    {
        return new string[] { "Hello", "There" }
    }

    static void Main(string[] args)
    {
        foreach (string x in GetGreetingWords())
        {
            Console.WriteLine (x);
        }
    }
}

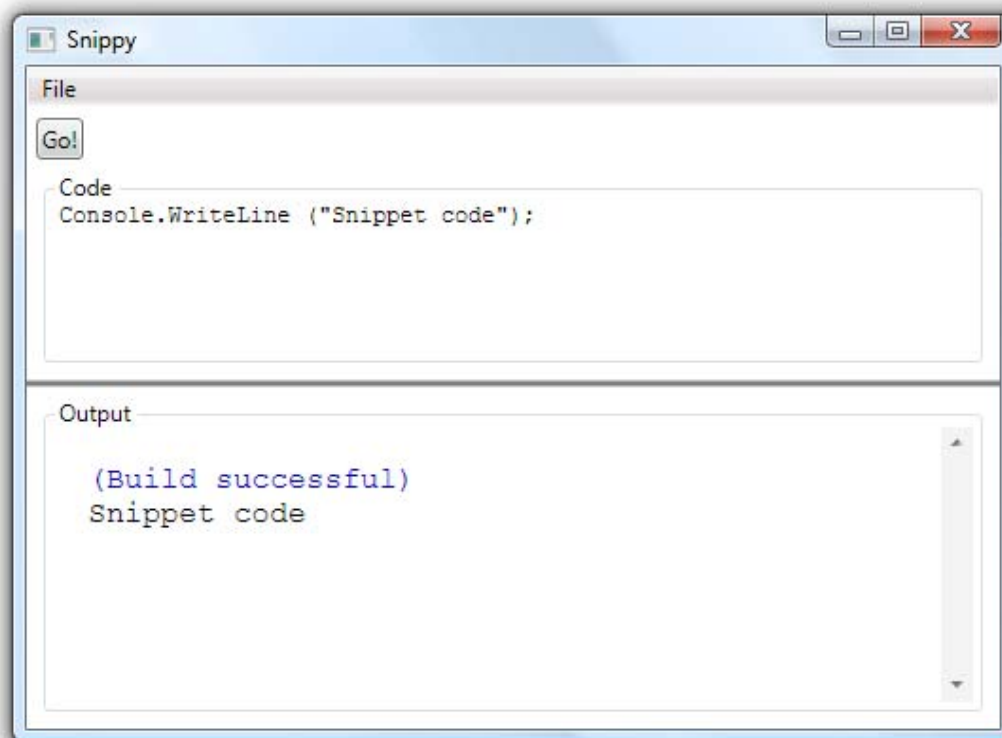
```

Types declared in snippets will be nested within the Snippet class, but that's very rarely a problem.

Now that we understand what snippets are and what they look like when they're expanded, let's make them a bit more user friendly.

### 1.3.2 Introducing Snippy

Just knowing what the code would look like isn't terribly helpful, so I've written a small tool which you can download from the book's web site. It's written in WPF, so you'll need to have .NET 3.0 or higher installed. Figure 1.6 shows a screenshot of it in action:



**Figure 1.6 Snippy in action. The code in the top area is converted into a full program, then run. Its output is shown in the bottom area.**

It's not a visual masterpiece, but it does the job. You can edit the code, compile it and run it. It will automatically detect which version of the framework you're using, and include suitable `using` statements and references. You can also optionally select a lower version of the framework, in which case the set of namespaces and assemblies used will be limited appropriately. It doesn't try to work out which `using` statements are actually *required* by the code, so the full code is rather longer than the examples I've shown in listings 1.19 and 1.21, but having extra `using` statements is harmless.

Everything in the C# 2 section of the book compiles and runs with only .NET 2.0 installed, and all the snippets compile and run with .NET 3.5 installed. There's a single button to compile and run, as you're unlikely to want to do anything after a successful compilation other than running the code. The option to export the code allows you to then easily open it up in Visual Studio if you want to explore the code further with all the comforts that Visual Studio brings, such as Intellisense. Speaking of Visual Studio, all the samples

work in the Express editions of Visual C# 2005 and 2008, although of course the examples which are specific to C# 3 don't run in Visual C# 2005<sup>8</sup>.

As I mentioned earlier, not all examples work this way – the examples in this chapter, for instance, all require the `Product` type which isn't included in every snippet. From this point onwards, however, I will give fair warning whenever a listing *isn't* a snippet – so unless you hear otherwise, you should be able to type it in and play around with it. Of course, if you don't like manually typing in code from books, you can download all of the code from the book's web site.

## 1.4 Summary

In this chapter, I've shown (but not explained) some of the features which are tackled in depth in the rest of the book. There are plenty more which haven't been shown here, and all the features we've seen so far have further “sub-features” associated with them. Hopefully what you've seen here has whetted your appetite for the rest of the book.

After looking through some actual code, we took a step back to consider the history of C# and the .NET framework. No technology is developed in a vacuum, and when it's commercial (whether or not it directly comes with a price tag) you can guarantee that the funding body sees a business opportunity in that development. I've not been through Microsoft's internal company memos, nor interviewed Bill Gates, but I've given my view on the reasons Microsoft has invested so much in .NET, and what the rest of the world has been doing in the same period. By talking *around* the language, I hope I'll have made you more comfortable *in* the language, and what it's trying to achieve.

We then performed a little detour by way of version numbers. This was mainly to make sure that you'll understand what I mean when I refer to particular .NET and C# version numbers (and how different those two can be!) but it might also help when talking with other people who may not have quite as clear a grasp on the matter as you now do. It's important to be able to get to the bottom of what people actually mean when they talk about a particular version, and with the information in this chapter you should be able to ask appropriate questions to get an accurate picture. This could be particularly useful if you ever talk to other developers in a support role – establishing the operating environment is always critical.

Finally, I described how code will be presented in this book, and introduced Snippy, the application you can use to run the code quickly if you don't want to download the full set of complete samples from the book's web site. This system of code snippets is designed to pack the book with the really *interesting* parts of code samples – the bits which demonstrate the language features I'll be explaining – without removing the possibility of actually *running* the code yourself.

There's one more area we need to cover before we dive into the features of C# 2, and that's C# 1. Obviously as an author I have no idea how knowledgeable you are about C# 1, but I *do* have some understanding of the areas of C# 1 which are typically only understood fairly vaguely. Some of these areas are critical to getting the most out of C# 2 and 3, so in the next chapter I'll go over them in some detail.

---

<sup>8</sup> Some of them may run in Visual Studio 2005 with the C# 3 extension Community Technology Preview (CTP) installed, but I make no guarantees. The language has changed in a few ways since the final CTP was released, and I haven't tested any of the code in this environment. Visual C# 2008 Express is free though, so why not give it a try?