Unofficial Fluent C# Errata (and notes)

Table of Contents

Introduction

This document lists the errors that I (Jon Skeet) have found in Fluent C# by Rebecca Riordan. As well as definite mistakes, I've included formatting problems, some suggestions around best practices and other areas of disagreement. I've tried to make it clear which kind of item is which. Some of these (particularly formatting suggestions) are definitely just nitpicks - but they're things I noticed, and things that I know *I* would appreciate being pointed out to me if someone found them in something I'd written. Small things like indentation can easily be fixed for future printings, improving the overall quality of the book.

The more important items are the ones described as "technical error" or "invalid code" - and if, like me, you care about technical terms being used correctly, the "terminology" items will be important to you.

At the moment, anyone can read and comment on this document - I may need to restrict it if I receive a lot of spam comments, but if that happens I'll work out another approach to get feedback.

Currently this list hasn't been reviewed by the author or publisher. Given that I'm just as fallible as anyone else, please don't simply believe me over anyone else - use your own judgement. As one aid, however, I frequently refer to the C# Language Specification, or "the spec" for short. Where I give section numbers, these refer to the C# 4 specification.

Finally, please don't regard this list as definitive by any means. These are just the items I noticed while reading the book - I haven't acted as a technical reviewer for the book, or anything similar. You'll notice that the density of notes is much higher for the chapters on the C# language, which is probably partly due to me knowing more about C# than (say) WPF - and even the notes I *have* got on the WPF sections are largely things like typos and layout issues. I can't comment on the accuracy of the other sections. If you have the book yourself and have found any errors, please feel free to mail them to me (skeet@pobox.com) and I'll include them here after checking them.

Addition as of January 2nd 2012: I've started receiving additional errata from other readers. I've marked each on as "reader contribution" to make this clear; in each case I've at least checked that the comment makes sense and is a sensible suggestion to improve the next edition of the book.

Chapter 1

Page 35, ambiguous language

Compiled languages... “The output is always specific to a single platform”

It could be argued that the output of the C# compiler (well, the MS C# compiler, at least) *is* always specific to the CLI, but basically the distinction between “compiled” and “interpreted” is more fuzzy than the page makes clear. Page 36 improves this somewhat when it introduces a JIT, but it’s still not made clear that C# *is* a compiled language. Also note that C# itself doesn’t *require* a JIT model - Mono is capable of both ahead-of-time compilation to native code (e.g. for iOS) and running the IL in “interpreted” mode. The language specification makes certain requirements, but it *doesn’t* require IL as the output format.

Page 36, terminology question (“CLR-compliant”)

“C# is a CLR-compliant language”

The term “CLR-compliant” is not one I’m familiar with. It only gets 174 hits on MSDN, compared with 44,000+ for “CLS-compliant”. It’s not clear whether it’s meant to be an official term or not; I would personally avoid using it due to this ambiguity. (CLS-compliance is a well-defined term, on the other hand.)

Page 38, terminology (CLI vs CLR)

“Microsoft’s version of the CLR runs on various versions of Windows operating systems [...]. The CLR has also been ported to run [...]”

This seems to be conflating two things. The CLR is the execution engine part of Microsoft’s implementation of the CLI specification. It’s not that the CLR has been ported to other platforms; it’s that the CLI has been implemented on other platforms.

Chapter 2

Page 51, sample code correctness question

The code sample on page 51 which is meant to implement control resizing looks exceedingly suspect to me. The author specifies that it “probably wouldn’t work for any other form” - but it’s not clear that it would work even for *this* form. It scales the textboxes in proportion to the overall scaling of the form, not taking into account the size of the labels which also have to be accommodated (with fixed width according to the pictures). I’m not a VB programmer so it’s *possible* that this would actually work somehow, but it’s not at all clear to me how.

Page 54, diagram issue

The diagram as the bottom left of page 54 has “WPF Browser Application” ringed, with a caption of “This is what we’ll create” - whereas the text nearby says, “Just choose WPF Application from the Windows category of Visual C# projects.”

Page 56, awkward wording (reader contribution)

"If it isn't visible, choose Properties Window from the View menu."
This should be in the order of execution: "... in the View menu, choose the Properties Window."
Or better yet, on a separate line indicating that it is something the student does:
"View menu -> Properties Window".
(There are similar occurrences throughout the remainder of the book.)

Page 62, ambiguous wording

Intellisense is described as “kind of like auto-complete, but a lot smarter.” A lot smarter than *which* auto-complete? Auto-complete is a generic term, and I’d say that Intellisense counts as one implementation of auto-complete. It’s like saying, “A Dyson is kind of like a vacuum cleaner, but a lot more powerful.”

Chapter 3

Page 88, code presentation nit-pick

(Yes, this one's really minor.) The code sample shows a method call without any whitespace:

```
    MessageBox.Show("message","caption",MessageBoxButton.
```

Given that this is aimed at beginners, I would really encourage "clean" code presentation where at all possible - and dense code like this becomes very hard to read. There's plenty of room here for a space after each comma.

Chapter 4

Page 98, terminology (syntax errors)

"Syntax errors are the typos of the programming world" - with an example of

```
    Messagebox("Hello");
```

(The "b" should be "B" - although usually I'd expect this to refer to the MessageBox class, not a *method* called MessageBox.) I don't view this as a syntax error at all. It's perfectly valid *syntax* for an invocation (of either a delegate instance or a method) but member lookup fails. I would reserve the term "syntax error" for a sequence of valid lexical tokens which *can't* form a valid C# program. For example:

```
    MessageBox("Hello";
```

Here the lack of a closing parenthesis for the member invocation makes this code *syntactically* invalid.

Page 98, terminology (compilation errors)

The previous point is followed by:
"Compilation errors are half-way between syntax errors and logical errors." This suggests that syntax errors aren't in themselves compilation errors, which they are. Furthermore, if we've already included member lookup failures within syntax errors, it's not clear what the author would count as just a "compilation error". No example is given, unfortunately.

Page 100, invalid code (reader contribution)

This variable declaration:

```
     Int32 result
```

should have a semi-colon to complete the statement (as the previous two do).

Page 104, disagreement: debugging vs unit testing

"As a programmer, you'll spend a lot of your time in break mode (if you're like most of us, it will feel like you spend more time debugging than writing code)"
I would argue that if you spend a large proportion of your time in the debugger, you're doing it wrong. This would have been a great point to introduce the idea of unit testing - maybe pointing out that while experimentation was good in some situations, automated testing is important to gain confidence in the correctness of your code in the face of changes elsewhere.

Page 109, disagreement: installation and web apps...

"Your application may be a better mousetrap, but people aren't going to beat a path to your door unless they can install it."
This makes an assumption that the user has to install an application in order to use it. That's simply not always the case, and this is one example of the client-application bias in the book. While web applications still need to be deployed, they don't need to be installed by the user - and things like Windows Phone 7 applications don't require separate setup projects.

Page 112, incomplete instructions (reader contribution)

"In the New Project dialog, select Setup Wizard from the Other Project Types, Setup and Deployment category."
Under "Setup and Deploymenet" there are two options available: InstallShield LE and Visual Studio Installer. The text should be more specific - something like this:
"In the New Project dialog, select Other Project Types, Setup and Deployment category, Visual Studio Installer, Setup Wizard."

Chapter 5

Page 123, terminology (nouns and verbs, sentences and paragraphs)

"Just like spoken English, C# has nouns and verbs, sentences and paragraphs."
No it doesn't. (I'd argue that *spoken* English doesn't have paragraphs either, but that's another matter.) Those terms are meaningless as far as the C# language is concerned, and trying to bend the *real* terms to be a bit like natural language leads to more confusion than clarity. The analogy is pretty thin right from the start, and I believe the book would be better off without it.

Page 124, minor technical error (comments being ignored by the compiler)

"Comments are ignored by the compiler, but are really important for the programmer."
The C# compiler doesn't ignore XML comments if you ask it to generate an XML file from your code with the "/doc:filename" flag.

Page 124, ambiguous language (directives)

(Label for directives): "These are instructions to the compiler and the .NET framework"
It's hard to see how these can be instructions to the framework itself, and surely all code is an instruction to the compiler in some respect. I can't see how a reader is going to gain any insight from this statement.

Page 124 and 125, terminology (declarations and statements)

Both the diagram on page 124 and the "declared elements" heading on page 125 claim that declarations are statements. Some are (local variable declaration statements, for example) but many are not (namespace declarations, type declarations, and type member declarations for example).

Page 124, terminology (commands)

Page 124 is where we start seeing the term "command" used. This is not part of the C# language - there's no such thing as a "command" in C#. There are for statements, while statements etc - but these are simply kinds of statements, not "commands". If the author wishes to make up terminology, it would be worth pointing out to the reader that she's doing so, so that the reader doesn't expect other developers who already know C# to use the same terminology.

Page 126, technical errors around punctuation

"C# doesn't have exclamation points [...], and question marks mean a variable can be empty. We'll talk about that in chapter 10."
I have three problems with this:

- Exclamation points are used in logical negation and inequality expressions
- Question marks are used for the conditional operator, the null-coalescing operator, and for nullable value types - the last of which aren't the same as meaning "a variable can be empty"
- I can't find anything in chapter 10 which talks about nullable types anyway (there's no index entry for null or Nullable)

Page 127, poor naming

The name `myClass` violates naming conventions *and* is generally poor in terms of description. It should be `MyClass` in order to follow Microsoft naming conventions, and class names should describe their purpose - `myClass` doesn't give any indication as to what it's for. In a book it's reasonably natural to need to give examples which don't serve any really useful purpose - they're just pedagogical - but it would be worth explicitly stating this, and possibly using more obviously-example-driven names. My fear is that having read this book, developers will be happy to stick with the names suggested by Visual Studio when creating Windows Forms applications for example - whole applications with Form1, Form2, Form3 etc instead of meaningful names.

For a beginners' book in particular, a small section emphasizing the importance of naming would be useful.

Page 128, technical error: statements

Many lines on page 128 are labeled as statements when they're not:

- `using` directives
- A namespace declaration
- A class declaration
- A field declaration

Page 129, technical error: declarations

"There are two types of declared elements in C#: variables [...] and constants"
There are *lots* of different declarations within C#. Those two are the two kinds of declaration *statements*, but the other kinds (namespaces, classes, methods etc) are still declarations. See the C# 4 spec sections 9.2, 10.1, 10.4-10.13, 11.1, 13.1, 14.1, and 15.1.

Page 130, technical errors: const

The book contains this line of code:

```
private const Elephant dumbo = new Elephant();
```

This is supposedly a statement. Problems:

- It's not a statement; it's a (non-local) constant declaration - see section 10.4
- `new Elephant()` is not a constant expression, so this won't even compile. It would if it were `readonly` instead of `const`, but that's very different. (I don't remember seeing the readonly modifier covered in the book.)

Page 131/132, omissions

The book asks what's wrong with this declaration:

```
Const char myChar = "B";
```

It correctly states that "Const" should be "const", but omits the problem that there's no conversion from the string literal "B" to char. This is covered above, but it should have been included here as "something which was wrong with the declaration." Additionally, although it's *not* an error, the reader hasn't seen that the C# alias "char" for "Char" so they may well have guessed at that being an error.

Page 133, clarification (identifiers)

"An identifier can contain letters, numerals or punctuation after the first character."
Well, it can contain *some* punctuation. Not just *any* punctuation though. "a.b" isn't a valid identifier.

Page 134, terminology (parameter and argument)

"Microsoft recommends camel casing for private identifiers [...] and the values you pass to methods. (These are called parameters [...].)"
No, the values you pass to methods are the *arguments*. The parameters are the variables which *receive* the argument values as their initial values. This is mentioned on P173, but it would have been nice to be accurate in the description here.

Page 134, naming conventions

"Pascal case is the recommended default convention. Use it for everything except private identifiers and parameters."
Arguably this one's a matter of taste, but I've seen *very* little code where PascalCase is the "default." I guess it depends on what you mean by a "private identifier" - if that includes local variables, that's fine... but in that case, the book violates the convention all over the place, using PascalCase for local variables. This looks completely non-idiomatic to my eyes - I'm used to pretty much *all* experienced C# developers using camelCase for local variables. This is a big distraction throughout the whole book.

(For reference, the Design Guidelines for Developing Class Libraries book contains the most commonly reference naming conventions. Despite the name of the book, these conventions are really appropriate for *all* developers - consistent naming isn't just a benefit for class library developers.)

Page 136, confusion (at least) around memory allocation

This code is presented:

```
Elephant Dumbo;
Dumbo = new Elephant();
```

The labels claim that "No memory has been allocated" after the first statement. This is confusing. The variable represents a storage location - it has memory allocated for it *somewhere*. There's then this statement for the second line:
"The new Elephant is created here. Two things happen at this point: memory is allocated, and the object is given an initial value."
It's not clear whether "the object" here refers to the *real* object which has just been created (assuming Elephant is a class, which hasn't been made clear) or the variable. I *suspect* the author is just using the word "object" incorrectly to mean the variable, but if I can't understand what she's saying despite knowing C# well, I can't see how a reader is going to be any the wiser.

Page 137, does new Char() create an object?

This is potentially a terminology gap. I typically use the word *value* for a reference, pointer or value type value, and *object* to mean an instance of a class (or boxed value type value). The code on page 137 I'm talking about looks like this:

```
Char Letter;
```

```
    Letter = new Char();
    Letter = 'A';
```
This is described as:

1. Declare the identifier
2. Create the object
3. Assign a value

My issues with this are around the terminology:

- You don't declare an identifier; you declare a variable by *specifying* an identifier as its name.
- The second line assigns a value as well - it'll be effectively U+0000 in this case.

Page 137, constructors use identifiers too...

(Referring to `new Char()`.)
"Notice that it has parentheses after it. That tells the compiler that you're calling a bit of code, not referring to an identifier."
Well you *are* referring to an identifier - but that identifier is *identifying* the name of the type whose constructor you want to call.

Page 139, clarification (casting)

"You can get around strong typing constraints simply by telling the compiler that it's what you really mean to do."
Well, to some extent. The compiler will prevent you from using casts which can clearly *never* work (e.g. a cast from a string expression to int) and the CLR will maintain real type safety at execution time - however much you cast, you're not *really* going to be able to store a value of 4.5 in an integer variable.

Page 142, disagreement (commenting)

"How do you know what to comment? This is what experience has taught the programming community: First, comment everything you think you might not remember in six months. Then go back and comment everything you think you will."
Absolutely not. Write good comments describing what methods are meant to *do* (particularly for APIs between components), but then only write comments in the *implementation* when it's non-obvious - for example when you can't take a simpler approach due to a bug in an external library. Overly-commented code is *less* clear than code which has merely been *carefully* commented.

Page 143, typo

The code in the middle of page 143 contains "Messagebox.Show" which should be "MessageBox.Show".

Page 145, clarification (reader contribution)

In some versions of Visual Studio, you need to make sure all settings are shown before you'll see the "Task List" settings. To do this, open Tools Menu -> Options and tick the "Show all settings" checkbox in the bottom left.

Page 145, inconsistency (reader contribution)

Step 2 of the instructions on page 145 talks about adding a "ToDo" comment (and step 3

confirms that it should be "Todo" (explicitly stating that capitalization is unimportant) but the *example* on step 2 is:
`//Hack: Make a note of something dodgy`
It should be "TODO" (in some capitalization) rather than "Hack".

Page 148, terminology (attributes)

"They're like directives in many ways"
Not really. There are a *few* uses that the compiler really knows about, but not many. They certainly don't deserve to live as a branch under a "directives" box in the diagram.

Page 148, confusing diagram

There's a "directives" box under a "directives" box - how confusing! What's this really supposed to mean?

Page 149, most commonly used directives

"The directive that's used most often is `#if...#else...#endif`."
I suspect that the *vast* majority of C# developers have more "using" directives (`using System.Console;` etc) than preprocessing directives - and quite possibly more `#region` directives than condition compilation directives, too.

Page 149, unnecessary and confusing brackets

The example of conditional compilation looks like this:

```
    #if (DEBUG)
```

This makes it look like it's really an expression in an "if" statement, and that the brackets are necessary. They're not, and I've rarely seen them used.

Page 151, terminology (defining vs declaration)

The name of the topic is "defining directives" - the name in the specification is "declaration directives"; see section 2.5.3

Page 151, terminology (constant vs conditional compilation symbol)

"[...] easy way to control the definition of the two most common constants used with directives"
The use of the word "constant" here suggests that the symbol has a *value* - whereas it's really just defined or not. The C# spec calls these *conditional compilation symbols*. See section 2.5.1 of the spec.

Page 151, technical error (preprocessor symbol case sensitivity)

"Directive constants aren't case-sensitive"
Yes they are, like everything else in C#. Hold-over from the VB version of the book, perhaps?

Chapter 6

Page 156, terminology (expressions as statements)

"Expressions are a kind of statement"
No they're not. For example, "x + 1" is a valid expression, but it's not a valid statement.

Page 157, terminology (lambdas)

"A lambda is an anonymous expression, roughly equivalent to using var to declare an anonymous variable."
Problems:

- A lambda expression really isn't much like using var
- A lambda expression is a kind of anonymous *function*, along with anonymous methods. There's no such term as "anonymous expression"
- var isn't used to declare an anonymous variable; it's used to give a variable a type implicitly. The variable still has a name, so it's not anonymous.

Page 158, terminology ("equations")

"What in spoken English is generally called an "equation", in C# is called a literal expression."
No it's not. "x + 1" sounds somewhat like an equation, but it's not a literal. The "1" part *is* a literal, but wouldn't be called an "equation" by most people.

Page 158, terminology (objects and types)

"technically called types, of which objects are one kind"
An object isn't a kind of type - a class is. I *suspect* that's what's meant here, but it's not clear at all.

Page 159, terminology (expressions)

"Like transitive verbs in spoken English, a C# expression performs some action on some thing."
In the literal expression "5", what action is being taken, and on what? What about a type expression, such as Foo in "`if (x is Foo)`"? These are both expressions, but they don't represent any kind of action being taken.

Page 159, terminology (statements aren't expressions)

This code is deemed to be an expression:

```
    TextBox.Text = "Hello, world";
```

Due to the semi-colon, it's a *statement*, not an expression. Without the semi-colon it would be an assignment expression.

Page 160, code formatting (spaces)

The conditional operator is shown with this code:

```
    x? y : z
```

I've never seen the conditional operator used without a space before the ? but with spaces around the other punctuation - it looks too much like it's trying to be a nullable type to me. I'd insert the space for consistency.

Page 160, terminology (true)

"true looks like a value, and the truth is it usually behaves like a value, too. But because it's actually an operator, you can re-define the meaning of true and false"
You can't really re-define the meaning of true and false. They're only operators in the context of operator overloading, and then I *believe* they're only used in the context of user-defined conditional logic operators (section 7.12.2 of the spec). I don't think this really counts as "redefining the meaning of true and false". Both true and false *are* valid literal expressions though (section 2.4.4.1).

Page 161, terminology (remainder vs modulus)

This describes the % operator as the "modulus" operator, when it's more properly called the "remainder" operator. (There's an index entry for "modulo" in the spec, but the word doesn't appear in the page it refers to - section 7.8.3.) The difference is only important in that I'd expect a genuine modulus operator to *always* return a non-negative value, whereas a remainder can be negative. Different languages use the terms to mean slightly different things, unfortunately, so this is a somewhat personal opinion - but I'd definitely use "remainder" to be on the safe side and to be consistent with the spec.

Eric Lippert has an interesting blog post on this matter too.

Page 162, typo (reader contribution)

The final answer given on page 162 is "x++;" which is fine, but then notes are given:
"Another trick. If you said x - x + 1 or x +=1 you were right [...]"
The first option here should be "x = x + 1" - using "=" instead of "-".

Page 163, inconsistency of subtraction and addition

This code is used to talk about precedence:

```
    2 + 3 * 4
```

But then the end of the paragraph reads: "because C# evaluates expressions according to a specified precedence, and multiplication precedes subtraction." The "subtraction" here should be "addition".

Page 163, disagreement (when to use brackets)

"but best practice is to use parentheses to disambiguate an equation"
Well, only when it's not clear to start with. It's easy to go overboard with this (just like commenting) and end up with code which is harder to read. I wouldn't use brackets for *just* a simple addition and multiplication, for example. (As noted in comments, this is definitely a matter of opinion - but simply claiming it's "best practice" is overboard IMO.)

Page 163/164, confusion (precedence)

The description of precedence (particularly "the part of the equation inside the parentheses is always executed first") is confusing. It's important to distinguish between precedence (which is about grouping) and execution order. Eric Lippert has two great blog posts about this:

- Precedence vs Associativity vs Order

- Precedencs vs order, redux

Page 163/164, technical error (precedence)

Page 163 contains this code as a test:

```
z = x++ * --y;
```

The explanation on page 164 states:

"The ++ operator is placed after the variable x, which gives it the lowest precedence of this set, so it's evaluated last, giving x the value of 2."

This is incorrect. The expression "x++" has to be evaluated *before* the multiplication, as it's an operand. However, the evaluated value of the expression is the value of x *before* the increment. In particular, if the expression were "x++ * x++" (with x=1 to start with) then the result would be 2, with x being 3 after evaluation - because the left operand would be evaluated first (giving a result of 1, x=2) then the right operand would be evaluated (giving a result of 2, x=3), then the multiplication would be performed, giving a result of 2.

Page 165, typo - unopened parenthesis (reader contribution)

The third sentence of page 165 has a closing parenthesis, but no opening parenthesis:

"Be careful of the capitalization)."

It wouldn't make sense to put the opening parenthesis earlier than the start of this sentence, so I'd suggest a correction to:

"(Be careful of the capitalization.)"

Page 166, terminology (conditional operators vs conditional logical operators)

To remove ambiguity, I would refer to || and && as *conditional logical operators*, whereas ?: is simply the *conditional operator*. This occurs on page 169 as well.

Page 166, technical error (compile-time vs execution-time evaluation)

When discussing the || operator, there's this statement:

"Since x is indeed smaller than y, (x < y) is true, so the compiler never looks at the increment operator."

No, the *compiler* looks at it as normal (and will complain if it's invalid etc). It's just never *executed*. That's a big difference.

Page 168, technical error or typo (XOR operator)

To determine if exactly one of x or y is true, this code is proposed:

```
x > y
```

I believe the intention was to have a caret instead, for the XOR operator:

```
x ^ y
```

Page 168, code formatting

The code "x >y" has no space after the ">", for no particularly obvious reason.

Page 168, alternative answer and wording

The answer is given as:

"Are either x and y true?"

```
x | y
```

The question should be phrased as "Are either x *or* y true?" - and using the conditional logical || operator is a valid (and usually preferred) alternative.

Page 171, terminology (more "object" confusion)

"You've seen that you create an object using a declaration statement"
No, a declaration doesn't always create an object, and you can create an object without using a declaration (e.g. calling a constructor in a method argument). It just declares something. Again, this seems like simply sloppy use of the word "object".
Page 171, terminology (yet more "object" confusion)
"As we'll see, you can define several different kinds of objects (called TYPES) in C#, but the one you'll probably work with most often is the OBJECT."
I can't see how this can possibly be read in a meaningful fashion. I suspect it should really be more like this:
"As we'll see, you can declare several different kinds of types in C#, but the one you'll probably work with most often is the CLASS."

Page 172, terminology (yet more "object" vagueness)

"MessageBox is an object (in this case a class)"
No, MessageBox is a *type* (in this case a class). Given that the word "object" appears to mean any of type, class, variable, or object, I'm going to stop reporting this ambiguity now - I suspect it keeps cropping up throughout the book though.

Page 173, terminology (parameter vs argument)

"A parameter specifies a value (either a literal value or a variable) that you pass to the method to work with."
No, that's an argument. The parameter is the local variable which *receives* the argument as its initial value (assuming it's a by-value parameter). This is actually discussed at the bottom in "Words for the wise" but the author hasn't used the terms consistently.

Additionally, arguments *don't* have to be literal values or variables. They can be the results of property accesses, method invocations, lambda expressions etc.

Page 175, technical error (operators)

"Symbolic operators like "+" and "/" are actually syntactic sugar. The compiler translates them into method calls like Int32.Add(Int32 x, Int32 y)"
Not for Int32 it doesn't - it (the Microsoft implementation, at least) translates them into IL "add" instructions. Using DateTime as a better example, the method called is the DateTime.op_Addition(DateTime, TimeSpan) member, which isn't the same as the Add method.

Page 176, terminology (instance of the method)

"while Window.Show() is called on an instance of the method"
I suspect this should be "an instance of the Window type".

Page 177, C# type aliases

It would be useful to indicate that this table is incomplete - it leaves off uint, byte, sbyte, decimal, float, double, long, ulong, short, ushort. It would also be worth noting that the full .NET type name is "System.Boolean" etc; the C# compiler doesn't resolve it as if you'd just typed Boolean; it knows the precise type involved. I don't mind whether the table is completed or just noted to be incomplete; for the sake of avoiding too much confusion, it may well make sense to keep it to just the types already present, so long as there's an appropriate note.

If the complete table were required, it might look like the one below - I've added an extra comment just for more information. (The extra information probably wouldn't be appropriate in the book, but readers may find it useful.)

C# alias	**.NET framework name**	**Comments**
sbyte	System.SByte	8-bit signed integer.
byte	System.Byte	8-bit unsigned integer
short	System.Int16	16-bit signed integer
ushort	System.UInt16	16-bit unsigned integer
int	System.Int32	32-bit signed integer
uint	System.UInt32	32-bit unsigned integer
long	System.Int64	64-bit signed integer
ulong	System.UInt64	64-bit unsigned integer
char	System.Char	Works as a 16-bit unsigned integer, but should be treated as a "UTF-16 code unit" - which in *most cases* can be regarded as "Unicode character", although reality is slightly more complicated.
string	System.String	Sequence of char values - essentially a piece of text
float	System.Float	32-bit binary floating point number
double	System.Double	64-bit binary floating point number
decimal	System.Decimal	128-bit decimal floating point number
object	System.Object	Every non-pointer type in C# is convertible to object. See Eric Lippert's blog post for more information.

Note that there's also "void" which is related to the System.Void framework type, but this is not quite the same as the aliases listed above; you can't use System.Void in C# code whereas you can use Int32 in most places where you can use int, for example.

Chapter 7

Page 181, terminology (commands)

"In this chapter, we'll finish up our examination of the C# language by examining the intransitive verbs of the language: commands."
Again, the analogy is somewhat tortured - and the term "command" is not a standard one.

Page 188, code formatting

On the same page which states:
"The braces aren't absolutely necessary here because there's only a single statement in each clause, but it's never a mistake to use them, and many programmers (including me) always use them."
I thoroughly applaud that sentiment - but find it surprising that on the very same page, the bottom example *doesn't* use braces for the nested if block. This is continued throughout the book - the bad practice (IMO) of using if statements without braces occurs all over the place.

Page 188, nit-pick around the necessity of bracing

"These braces are necessary, because the else clause contains multiple statements in a block"
There are two things wrong with this:

- The braces around the first statement block wouldn't be necessary even if that were the case
- The braces *aren't* necessary here as the else clause only contains a single statement (an if-else statement)

Even without the braces, the "else" clauses would behave appropriately. From the spec, section 8.7.1, "An else part is associated with the lexically nearest preceding "if" that is allowed by the syntax." So this whole example *could* be written without any braces at all.

Page 189, technical error (necessity for case clauses)

"You must have at least one case clause, but you're not required to have more than one."
Incorrect. A switch statement with *just* a default label compiles with no warnings or errors. A switch statement without *any* labels compiles with a warning (CS1522) but no error.

Page 189, disagreement (whether or not to have a default)

"The default case is optional, but a very, very good idea."
That entirely depends on the context. If you're deliberately switching on something which may not match any of the given cases, and you don't want to take any particular action in that situation, why is it a "very, very good idea" to have an empty default label?

Page 189, technical error (invalid code)

The code in the bottom right of the page is invalid (error CS0163) because control reaches the end of the default label. Just like case labels, the compiler ensures that the end of a default label is *unreachable*.

The code should be something like:

```
switch (x)
{
 case 1:
 case 3:
   MessageBox.Show("1");
   break;
 case 2:
   MessageBox.Show("2");
   break;
 default:
   MessageBox.Show("Something else");
   break; // This line is missing in the book
}
```
Note that any other way of avoiding the end of the default case being reachable would work too - throwing an exception, or returning from the method. Using a break statement is the most common approach, but it's useful to know that the rule is about reachability, *not* about "there must be a break statement". See section 8.1 of the C# 4 specification for more information about reachability.

Page 190, code error (reader contribution)

The left-hand example code on page 190 ends with "break" just before the closing brace. There are two problems with this:

- It doesn't have a closing semi-colon, so it's not a valid break statement
- It's not in a switch statement or a loop (as far as we can see) so it's not valid to have a break statement here anyway.

Basically, the break statement should be removed.

Page 190, code error (reader contribution)

The examples at the top of page 190 are meant to achieve the same result - but the side-effects on the left-hand side all modify x, whereas the side-effects on the right-hand side all modify y. (It's not a matter of the right-hand side using y universally - it's still *reacting* to the value of x.)

Page 190, formatting error (wrong font)

The code at the bottom of page 190 is in a proportional font, rather than the more conventional (for programming) monospaced font which is used in the examples at the top of the page.

Page 191/192, confusing requirement

The example at the bottom of page 191/192 is unfortunate:
"If x is equal to 0, increment x. If x is equal to 5, decrement x. If x is equal to y, set x to 10."
In this specific case, it doesn't matter if we view the first two conditions as being evaluated "at the same time" because 0+1 is not 5. However, suppose it were:
"If x is equal to 0, increment x by y. If x is equal to 5, decrement x. If x is equal to y, set x to 10."
At that point, what should we do if x is 0 and y is 5? Should we check for x being equal to 5

after incrementing it by y, leaving x as 4 overall? It's this logical question of whether we're evaluating *one value* against multiple cases, or whether we're checking a condition and then potentially changing state before the next condition is evaluated which is the most important distinction between using "if" vs using "switch" here - the fact that the final condition can't even be expressed with a switch statement is almost incidental.

The previous example (in the same exercise) was similarly worded, but used a switch statement to achieve it - leaving this as a basically ambiguous exercise, which *happens* not to cause problems because of the values involved. Exercises should not be expressed in such ambiguous terms in my opinion.

Page 192, alternative approach (reader suggestion)

Following on from the confusing requirements, a reader has suggested noting an alternative to the three independent (and always evaluated "if" statements). We can ensure that only *one* statement body is executed using if / else:

```
if (x == 0)
{
  x++;
}
else if (x == 5)
{
  x--;
}
else if (x == y)
{
  x = 10;
}
```

As this point, when reading the code you don't need to worry whether a side-effect within one statement body (e.g. `x++;`) will affect the evaluation of one of the later conditions.

Page 192, typo in code

The top question on Page 191 (repeated on page 192) talks about setting the value of a - but the *answer* only ever sets the value of x, not a. To answer the question as answered, the code should be:

```
switch (a)
{
    case 3:
        a = 5;
        break;
    case 10:
        a = 15;
        break;
    case 0:
        break;
    default:
```

```
                MessageBox.Show("something";
                break;
        }
```

The changes are to the two assignments.

Alternatively, the question could be changed to "If a is equal to 3, set x to 5. If a is equal to 10, set x to 15. [...]" However, if that *sort* of change is made, I'd suggest using "b" instead of "x", in order to keep the first question entirely separate from the second.

Page 194, terminology (for loops)

"There are three categories of iteration statements: those that execute a statement block a specific number of times [...]"
That's really not a category of iteration statement. It so happens that if you use a "for" loop in a particular way, it will execute a number of times (barring break, continue and return) but that's not really how it should be described.

Page 195, incorrect quotes (occurs elsewhere too)

The sample "for" loop on page 195 uses two "open double quote" characters. C# doesn't use these "curly quotes" - it only deals uses the quotes within the ASCII character set, which are typically shown as "straight quotes".

This occurs quite frequently in the code, but is particularly noticeable on page 195 as there are two opening quotes rather than the more reasonable-looking (but invalid) open/close quote pair.

Page 196, confusing example (reader suggestion)

The third example of a for loop on page 196 is as follows:

```
    for (Int32 x = 0; x < 5; x++, MessageBox.Show(x.ToString()))
      ;
```

This is explained and discouraged at the bottom of the page, but it could be confusing for readers given that it's not clear from the syntax diagram on page 195 that this is even allowed.

To be honest, I can't remember ever seeing code which tried to do this - given that there are any number of weird things you *can* do if you really try, I'm not sure it was worth giving this example at all - or the example itself should have been put in a separate box.

Page 197, disagreement (for vs while)

"But in a classic for loop, the condition section only evaluates the control variable. If your condition is more complicated, it's usually clearer to use one of the two conditional iterations [sic] statements that are specific designed for that purpose."
I wouldn't use that as the way of deciding whether or not to use a for loop - I'd use the question of whether I want something to be executed at the end of each iteration (regardless of whether the iteration was terminated early with a continue statement) and whether or not I wanted an extra initialization step which was capable of declaring local variables for the given scope. If the condition itself becomes too complicated for a "for" loop, I wouldn't use it

verbatim in a while or do/while loop either - I'd extract it to a method.

Page 198, code error (extra semi-colon - valid but unintended; reader contribution)

The code on page 198 to explain the difference between do/while and while contains a stray semi-colon:

```
Int32 x = 5;
do
{
 MessageBox.Show("Inside do");
} while (x < 5);

while (x < 5);
{
 MessageBox.Show("Inside while");
}
```

The second while loop isn't what it appears to be: the semi-colon before the opening brace makes it behave like this:

```
while (x < 5) { /* do nothing */ }

// And then this is executed...
MessageBox.Show("Inside while");
```

So contrary to the book's claim, "Inside while" (which *isn't* inside a while loop) will be displayed once. The correct fix is just to remove the semi-colon:

```
while (x < 5)
{
 MessageBox.Show("Inside while");
}
```

Now it won't be shown at all, as x is never less than 5.

Page 201, poor example (goto)

"The code snippet below would be a lot easier to understand if it were written as a switch statement, but there's no way to duplicate the if (x == 2 | x == 3) condition as a constant." Well assuming we're happy to only evaluate x once (the use of | instead of || here is already dubious) you can just use a switch with case labels for 2 and 3:

```
switch (x)
{
  case 1:
    x++;
    break;
  case 2:
  case 3:
    x--;
    break;
  default:
```

```
            x = y;
            break;
        }
```
No need for a goto - and much nicer, IMO. The possibility of having more than one case is even explicitly called out on page 189. If the author *really* wants to give an example for using goto, she should come up with a better one.

Page 204, meaningless example

Page 204 gives some sample code, and then says:
"To understand what's wrong with this code, ask yourself some questions"
We don't know if there's *anything* actually wrong with the code - we're never told what it's meant to achieve.

Page 205, invalid code sample

The right hand side of page 205 contains this code:
```
        if (a == z) | (a == q)
```
This is invalid due to the positioning of the parentheses. It's also not clear why "|" is used instead of "||" here; I personally use | *very* rarely (I almost always want short-circuiting) and that's been the case everywhere I've worked, too. Whether the two conditions are parenthesized or not is a matter of taste, as == has higher precedence ("binds tighter") than either | or ||. So any of these are valid:
```
        if ((a == z) | (a == q))
        if (a == z | a == q)
        if ((a == z) || (a == q))
        if (a == z || a == q)
```

When using the "Conditional OR" operator (the double pipe) I probably wouldn't bother with extra bracketing; when using the "Logical OR" (the single pipe) I probably *would*, to make it clear I wasn't trying to evaluate "z | a". So I would probably use the first or last of these options - most likely the last.

Page 207, terminology (variables vs values for arguments)

The exercise on page 207 asks what should happen if:
"A variable passed to a method call contains the wrong type of data"
I think it's important to make the point that (out/ref parameters aside), the argument is a *value*, not a *variable*. (It's also important to distinguish between the compile-time type of the argument expression and the execution-time type of its value.)

Page 208, disagreement (behaviour for inappropriate argument types)

The stated answer to the exercise above is that an exception should be thrown. There's a label saying that *sometimes* the wrong value can slip in anyway. Well, the "wrong type" here can't violate the declared parameter type, as otherwise that would give a compile-time error. An exception is only appropriate if the value is invalid for the call - and the wrong type is one of the *least* likely problems here, due to compile-time type safety.

Page 209, code style (explicit comparison with true)

The code within page 209 includes:

```
if (ThereIsAProblem == true)
```
This sort of thing occurs frequently through the book, and to my mind isn't as clear as simply using the fact that we've already got a Boolean expression:
```
if (ThereIsAProblem)
```
(I've only highlighted this occurrence.)

Page 210, technical error (try / catch / finally)

The syntax shown on page 210 implies that a try block *must* always have a catch block; it's actually optional - you have to have *either* at least one catch block *or* a finally block, or both.

Page 210, technical error (catch block)

The syntax shown on page 210 implies that both the type and variable name in a catch block are required; they're both optional (although you have to specify the type if you want to specify a variable, and you don't include the parentheses if you're not specifying either of them). So all of these are valid:
```
catch(IOException e) { … }
catch(IOException) { … }
catch { … }
```

The last form is almost never useful as of .NET 2.0; before .NET 2.0 (or in 2.0+ with a suitable flag) it's possible for unmanaged code to throw an exception which a "catch (Exception)" block wouldn't catch. These days that exception is coerced into a "normal" one though. You should almost never be catching so broadly though...

Page 210, code style (throw ex)

The sample code includes:
```
catch(Exception ex)
{
    throw ex;
}
```
Not only is it a bad idea to catch Exception in most cases, but throwing like this will destroy the stack trace, making it harder to diagnose the original problem. If you're not going to do anything other than rethrow then the catch block shouldn't exist at all, but if you *do* need to rethrow, just use the throw statement like this:
```
catch(Exception ex)
{
    throw;
}
```

Page 210, missing description of important feature

(This is the most appropriate place for this description.)
The code sample shows a finally block being used to close something. This is fine - but not idiomatic, as typically you'd use a "using" statement for this instead. I can't find anything describing the "using" statement in the book, which is a significant oversight in my view. (If your reaction is "the book can't include everything" then ask yourself which is more likely to be important to a fledgling programmer: handling resources cleanly, or adding drop shadows

and blur effects, which are both covered in chapter 19.) Page 318 *claims* that it will appear later in the book, but I haven't seen any sign of it, and none of "using statement", IDisposable, or Dispose appears in the index.

Note that a using statement doesn't allow you to specify a catch block - but try/finally blocks, typically for resource cleanup, should be *much* more common in most code than try/catch or try/catch/finally blocks - and the using statement is the idiomatic way of representing a try/finally block where the finally block just disposes of a resource. Personally I'd rather use a using statement with a nested try/catch block than explicitly use a try/catch/finally block, most of the time.

So the code on page 210 would usually look more like this:

```
using (Stream input = File.OpenRead("test.dat"))
{
  // Use the stream here
}
```

It's not clear what "x" was meant to be in the original example, but in very many cases a variable can be declared at the point of resource acquisition - and when you do that in a using statement, it means you *can't* accidentally use the variable again after the resource has been released, because it's out of scope.

Page 210, inconsistency (catch variable naming; reader contribution)

On page 210, there's a label for a catch block saying:

'The name specified here is the identifier you'll use inside the catch block to refer to the exception. The convention is to call it "e".'

However, in the code sample on the same page, we have:

```
catch (Exception ex)
{
  throw ex;
}
```

Personally I've seen both "ex" and "e" used a lot, and I don't have a particular preference - but it would be nice to see consistency here.

Page 211, disagreement (exceptions and users)

"The Message property is set when the Exception is created. Because it may be displayed to the user, it should be as informative as possible and include some suggestions about what needs to be done to fix things."

I disagree with this approach. If an exception is going to be shown to a user, it should be for the sake of passing it to technical support. If the exception is aware of something the *user* can do to fix the situation, that should be handled by the application appropriately - potentially by offering to take the corrective action on their behalf. *That* message needs to be appropriate for users, but exception messages should be aimed at *developers*. What on earth would a non-technical user be able to deduce from (say) a NullReferenceException and a mysterious stack trace? It's effectively incomprehensible information which can only be understood by support staff.

Note on some ambiguiities around this: Microsoft's guidelines *do* talk about localizing exception messages, which has been a bad idea in my experience, and they *do* talk about the possibility of users seeing errors. However, when Microsoft talks about including information which would help you correct the error, I believe they're talking about the *developer* correcting the error, not the user. In my view an exception should *not* be shown to users by default. They should be sent to the developers, with the user able to have a look at what the report contains if they really want. It therefore makes sense to be *careful* in your exception messages - not to be vulgar, for example - but I wouldn't expect the user to actually be able to correct a problem based on the exception message.

On the matter of localization: I once worked on a project which *did* decide to localize all exception messages, despite the fact that all the developers were in the UK. The results were:

- It was *much* more effort to include useful information in an exception message. It involved resource files, translations etc. As a result, you'd often reuse an existing message rather than creating a specific one for the situation, so you'd end up conveying less information.
- Any code which threw an exception was harder to understand, because the message wasn't just there in the code in plain text.
- When stack traces came back from other countries, they were harder to understand - we'd have to try to find the localized text, then get back to the original resource identifier, then translate it back into English. What a waste of time.

It *does* (potentially) make sense to localize exception messages if you're building a class library which will definitely be used by developers in multiple cultures, particularly if your native tongue isn't English. Many developers will be used to reading technical messages in English even if it's not their native tongue, and sometimes a well-written English message will be more useful than a poor translation to another language... but it's all a judgement call at that point. You'd need to balance the perceived benefits with the sort of costs listed above.

Page 214, clarification (order of catch blocks)

"The order of the catch blocks is important here - you should always list them from most specific to most general, because once an exception is matched, none of the other blocks is evaluated."

This fails to mention that unless some of the exception types are generic type parameters, the compiler will catch this problem anyway - so it's not something you have to consciously be careful about.

Page 214, invalid code

Not only does the code on page 214 have the same problems noted earlier (catching Exception, having a catch block for no reason *other* than to rethrow, and rethrowing in a way which destroys the stack trace) but it's also invalid:

```
catch (Exception ex)
{
    throw e;
```

```
        }
```

Note the different identifiers (ex and e).

Page 219, insufficient information (reader contribution)

"The easiest way to search for SystemException in the Object Brower, and then expand the Derived Types node."

It may be worth noting that if you can't see a "Derived Types" entry when you expand a type, it may be disabled. To re-enable it:

- Click the Object Browser Settings icon in the ribbon.
- Check Derived Types.

You should then be able to expand System.SystemException, and expand its list of derived types.

Chapter 8

Page 223, terminology (object again!)

"The statement below contains both a class and an object. Which do you think is which?"

```
    public String Message;
```

There's no object in that statement at all. There's an access modifier, the name of a class (String), and the name of a public mutable field (Message). Assuming an object of the containing class is created, the presence of this declaration won't create any new objects in itself. (It will start with a default value of a null reference, and may then be assigned a different value which may be to a "new" string, or it may be to an existing one.)

Page 225, incompleteness (delegates)

The diagram on page 225 shows the kinds of types which can be declared in C# - but omits delegates.

Page 229, disagreement (the class designer)

It's not at all clear to me why so much time is given over to the class designer. I find it's significantly less effort to type (say) a method declaration into the source file than to go into the designer and click all over the place to get what I want. Why not just show the code?

Page 232, incorrect screenshot

On page 231, we're asked to create an Ingredient class - but the screenshot on page 232 shows just a Recipe class. The screenshot should be changed to show just an Ingredient class.

Page 232, missing class (reader contribution)

On page 232, the text talks about adding the Component class - but not the Recipe class. Given the highly step-by-step nature of the rest of this section, it's odd to see the Recipe class on pages 233 and 237. The simplest fix is to change page 232 to talk about adding both the Component and Recipe classes. The "review" section on page 240 talks about adding the Recipe class, but by then we've already seen it in several places.

Page 234, technical error (private access)

Private members are stated to be accessible by "the type that declares them" - they're also accessible to any *nested* types within the same containing type. (This can be very handy sometimes.)

Page 234, disagreement (utility of internal)

While I agree that protected internal is used relatively rarely, I would suggest that in well-encapsulated code, many types *should* be internal. If only your assembly (and test assemblies which can be given access via InternalsVisibleToAttribute) is going to need to access a particular UI window, for example, why make it public?

Page 238, disagreement (naming of fields)

(This is definitely *not* an error - it's purely a matter of opinion.)

Personally I *don't* use an "m_" prefix for field names. I've worked at companies which use prefixes; I've worked at companies which don't. Personally I don't like them much - they interfere with the way I subvocalize code. However, so long as you only use them for *private* members - and so long as you're consistent with yourself and anyone else working on the code - it's fine either way.

I find that the capitalization makes enough difference to alert me to whether something is a variable or a property. Of course, it means you have to follow the conventions rigidly, but that's not really hard.

Page 240, clarification (reader contribution)

"When you told the Class Designer that Component inherited from Ingredient, it changed the diagram, but it also made a change to the properties shown in the Properties window for the Component class."
It may be worth noting that this is the "Properties" *window* (with the class name, access, etc), not the Properties part of the "Class Details" window (which lists the methods, properties, events and so on).

Page 241, code formatting (indentation)

The declaration for the QuantityOnHand property hasn't been properly indented - the "p" of public is indented one space more than the opening brace.

Page 242, technical error (incorrect default accessibility)

"Any of the access modifiers can be used with the class statement [...]. If you don't specify the modifier, the class will be private, which is hardly ever what you want."
This is incorrect. Only *nested* classes can be private, protected, or protected internal - and the default for top-level classes is internal. (And again, the class declaration isn't a statement.)

Page 242, disagreement (reason for using explicit access modifiers)

While I now agree that access modifiers should be stated explicitly, my reasons are different. I view this as a matter of making an explicit choice: each time I declare a member, I should be putting some thought into how accessible it should be. The defaults are actually very good in C# (and *very* easy to remember: the default is always the most private it can be for that declaration) but making it clear to someone reading your code that this is the result of a deliberate choice is a good thing.

Page 242, incompleteness of modifiers

I find it odd that "partial" has been included in the options for a class declaration, but not "sealed", "abstract" etc. Is this because the book is heavily client-UI focused, and so many classes are partial because of the designer side?

The C# spec actually *does* list "partial" separately to the other modifiers, but that includes *all* the modifiers in the "class-modifiers" category, whereas the label on the diagram in the book suggests that the author is only considering access modifiers (public, internal etc).

Page 246, poor sample code

The sample code for the QuantityOnHand setter throws just Exception - when ArgumentOutOfRangeException would be much more appropriate. This is a book targeted at novices - it shouldn't give them bad habits such as throwing Exception (which is almost *never* appropriate).

Page 249, terminology (immutable)

"For the sake of efficiency, some properties of a type may be immutable"
A property is read-only rather than immutable per se. I'd also argue that there are *far* more important reasons for creating immutable types than efficiency - clarity and thread safety being the obvious two.

Page 249, technical error (read/write-only automatically implemented properties)

In the context of read-only or write-only properties:
"In either case, the syntax is simple. You simply omit the get or set accessor in a full property declaration or omit an accessor with auto-implemented properties."
This is incorrect. You can't create a fully read-only or write-only automatically-implemented property. It won't compile. The closest you can come is an automatically-implemented property with (say) a public getter and a private setter.

Page 252, terminology (signature)

"The return type, name and parameters of a method define its [...] signature"
(This is actually a multiple choice question, but I'm assuming the author thinks that "signature" is the right option - "membership" would be incorrect.)
The return type is *not* part of the signature of the method. See section 10.6 of the specification for confirmation: "The return type is not part of a method's signature, nor are the names of the type parameters or the formal parameters."

Page 253, terminology (methods)

The diagram on page 253 shows constructors and destructors as methods. They're not methods, although they are members.

Page 255, naming consistency (reader contribution + Jon's opinion)

The text on page 255 talks about an Allocate method with two parameters: quantity and schedule. The screenshot shows a single parameter called Quantity (starting with an upper-case Q). Presumably the "schedule" parameter is missing as the screen-shot is meant to have been taken while in the process of adding the parameters, but the existing name should be consistent with the text - and .NET naming conventions.

According to Microsoft's .NET naming conventions this should be "quantity" (as described in the text). The capitalization rules for identifiers are easy to follow in terms of whether to use PascalCase (first letter is upper-case) or camelCase (first letter is lower-case):

- Do use Pascal casing for all public member, type, and namespace names consisting of multiple words.
- Do use camel casing for parameter names.

Note that this *only* talks about public members: what you do with private member names is up to you. Personally I still use PascalCase for all methods and type names, and camelCase for all non-constant fields. (I view a static readonly field of an immutable type as being philosophically equivalent to a const member, even though there are some subtle differences.) So for this code I'd use:

```
public class Component : Ingredient
{
    private int quantityAllocated;
    public int QuantityAllocated
    {
        get { return quantityAllocated; }
    }

    public void Allocate(int quantity, bool schedule)
    {
      // Code here
    }
}
```

(I'd probably put the QuantityAllocated property on a single line in this case, but there isn't room on the page.)

Page 256, terminology (instantiation)

"You can't create types or members within a method, but you can instantiate types using a variable declaration."

Variable declarations are mostly orthogonal to instantiation. You can instantiate a type without using a variable at all, and you can declare a variable without instantiating a type. For example:

```
// Instantiation, no variable
Console.WriteLine(new StringBuilder());

// Variable declaration and instantiation
StringBuilder foo = new StringBuilder();
// No new objects created in the final line!
// Variable declaration and assignment but no instantiation.
StringBuilder bar = foo;
```

Page 256, technical error (accessibility)

"There are a few constraints on the access modifier of an instance method. First, it cannot be less restrictive than the type that defines it."

Yes it can. For example:

```
internal class Foo
{
    public void Bar()
    {
    }
```

```
        }
```

Aside from anything else, if the given rule were correct it would be impossible to override ToString, GetHashCode or Equals within an internal class... The book goes on:

"And second, for pretty much the same reason, it cannot be less restrictive than the type of any of its parameters."

This is *nearly* true, but not quite. For example:

```
        internal class Foo
        {
            public void Bar(Baz baz)
            {
            }
        }
        internal class Baz {}
```

Here it's okay for the public Bar method to have a parameter of internal type Baz because the declaring type Foo is internal too. This is all to do with *accessibility domains* - see section 3.5.2 of the spec.

Page 257, code errors

The method shown on page 257 has a number of errors. Here it is verbatim (slightly smaller than normal just to make the lines fit):

```
        public void Allocate(Int32 quantity, Boolean schedule)
        {
         if (quantity < 0)
          throw new ArgumentException("quantity must be greater than 0");
         return;
         this.QuantityOnHand -= m_quantityOnHand;
         m_quantityAllocated += m_quantityOnHand;
         if (schedule == true)
          throw new NotImplementedException("Not Yet Built");
        }
```

Problems (in addition to the comparison with true and the less-than-ideal exception choice):

- The unconditional return midway through the method means the rest of the code is unreachable.
- The check should probably compare the quantity parameter with QuantityOnHand as well as 0 - if we've only got 5 units, it doesn't make sense to allocate 10, just like it doesn't make sense to allocate -5 units. This may be best as two separate checks, in order to give sensible exception messages in each case.
- Subtracting m_quantityOnHand from QuantityOnHand will leave 0 on hand. This should probably be the "quantity" parameter (which isn't otherwise used after the first check).
- Ditto the increment for m_quantityAllocated (which due to the line before, will currently be a no-op, as we'll have set m_quantityOnHand to 0...)
- Throwing an exception indicating that the method isn't fully implemented *after* modifying the object is really bad practice. This final check should be the *very first* line of the method.

Page 258, disagreement (single return statement)

"As a general rule, every method should have a single return statement."
I disagree with even this loose statement. The author *does* acknowledge that simplicity should trump dogma, but I wouldn't even try to make a single return statement a general aim. Just write the simplest code which achieves the desired result - that's the only aim you need here. Sometimes that will be a single return statement - pretty often it won't be.

In particular, if you find yourself with a local variable which is only ever written to once per execution path, and then used as the return value without executing any more useful code in any execution path, you should *strongly* consider removing the variable and replacing each assignment with a return. If you know you've done everything you need to in a method, why not give that information to someone reading the method?

Page 260, disagreement (how to implement an overload)

A better implementation to the one suggested (IMO) would be:

```
public void Allocate(Int32 quantity)
{
    Allocate(quantity, false);
}
```

Why reproduce the same logic, when you can just delegate from one overload to the other?

Page 260, terminology (default constructors)

This is a matter of opinion, but I believe it's clearer to use the term "default constructor" *only* for one provided by the compiler. A parameterless constructor you provide yourself is just a parameterless constructor - why give it a different name?

Page 260, disagreement (always providing a parameterless constructor)

The book describes how adding a parameter to a constructor signature will break all calling code, then:
"The solution? Always define the default constructor yourself, even if it doesn't do anything."
Absolutely not. There should only be a parameterless constructor if it makes sense to be able to instantiate the class without providing any other information. If the class is changing in a way that *requires* more information to be provided than before, then yes, that's a breaking change. Either you need to rethink the change, or take the hit of providing that information from all the current callers. The alternative is to never have any confidence that your types have a consistent set of information, because you can't enforce it even at creation time.

Page 262, technical error (incorrect code)

The syntax for constructor chaining is given incorrectly on page 262. Here's a snippet of the code in the book:

```
public myClass()
{
 this.myClass(0);
}
```

That's not the right syntax for constructor chaining. It should be:

```
public myClass() : this(0)
```

```
{
}
```

The same problem can be seen at the top of page 265.
(This may be another case of code from the VB version of the book invading the C# version...)

Page 262, poor code example (naming)

The sample code on page 262 uses a class name of "myClass". This is both meaningless and violates .NET naming conventions. Additionally, this example uses the C# alias "int" instead of "Int32" for parameters - that's fine on its own (and is my preferred style) but it's inconsistent with the code in the rest of the book.

Page 264, technical error (memory allocation)

"When you create a variable with a declaration statement, .NET allocates enough memory to store the specified type's properties and make the identifier you've declared available for use."
No it doesn't. Leaving aside when allocation actually takes place, only enough space is allocated for the variable to have a value. For reference types, that value will just be a reference - so only a reference-sized space is required.

Page 264, technical error (public variables)

"The public keyword can only be applied to types and members, not variables."
Well not *local* variables - but fields are variables too, and they're type members, so can be declared public.

Page 264, code style (parameter naming)

The MySentence method has this declaration:

```
public string MySentence(string Name)
```

This violates .NET naming conventions due to the parameter name - as well as being inconsistent in its use of string vs String compared with the rest of the book.

Page 265, technical error (invalid code)

Aside from the syntax error in the code at the top of the page for constructor chaining, the code at the *bottom* of the page contains an invalid for statement:

```
for (Int32 y; y <= 5; y++)
{
 x++;
}
```

This will fail with error CS0165: Use of unassigned local variable 'y'

Page 267, confusing explanation (garbage collection)

The explanation of memory allocation and garbage collection on page 267 is sufficiently wrong-headed that it basically needs a complete rewrite. It ignores stack allocation, assumes that only a single variable ever refers to an object, claims that .NET applications have a default memory limit of 2GB, and generally I can't see it making garbage collection *clearer* for anyone.

Chapter 9

Page 271, terminology (structures)

"Structures, which are like baby classes"
No, they're not. I know this is only a preamble, but that's a lazy description which is more likely to harm than help. (This is effectively repeated on page 276, where structures are described as "lightweight classes".)

Page 273, incomplete diagram (delegates)

The diagram on page 273 mentions classes, structures, interfaces, and enumerations - but not delegates.

Page 273, incomplete diagram (nested types)

The diagram on page 273 shows the possible members of classes and structures, but omits the possibility of having one type as the member of another.

Page 273, technical error (static and interfaces)

The diagram on page 273 implies that interface members can be static. They can't. From section 13.2 of the spec: "... nor can an interface contain static members of any kind."

Pages 274/275, technical error (values types and the stack)

The picture on page 274 and the labels on page 275 propagate the myth of "reference types live on the heap, value types live on the stack." This is a common myth. It's an over-simplistic description which ends up causing confusion in the long term. See Eric Lippert's blog post for more details (there are various others by Eric which are, of course, well worth reading).

Page 274, technical error (value type magic)

"The .NET Framework type System.ValueType, itself a reference type, contains the magic that causes its children to be stored directly on the stack."
Leaving the previous point about the stack to one side, the "magic" isn't in the code for System.ValueType - it's in the CLR. ValueType *does* contain code for things like Equals, but not the code to determine how instances are stored in memory.

Page 277, terminology ("objects" being passed)

"On the other hand, objects tend to get passed around"
No, *values* are passed around - either value type values, or references. Actual *objects* are never passed round.

Page 280, invalid code

This code is invalid:

```
public struct FirstStructure
{
 public FirstStructure(Int32 x, Int32 y)
```

```
        {
         this.X = x;
         this.Y = y;
        }

        public Int32 X { get; set; }
        public Int32 Y { get; set; }
       }
```
The problem is with using automatically implemented properties in structures - the fields must all be definitely assigned before any properties are called, so the constructor has to be changed to:
```
        public FirstStructure(Int32 x, Int32 y) : this()
        {
         this.X = x;
         this.Y = y;
        }
```
However, that's not the code that I'd *recommend* anyway. Mutable value types like this are evil (and they certainly *shouldn't* be presented to beginners like this, with no warning) - this structure should declare two read-only private fields (x and y) and declare two read-only properties which return their values.

Page 281, disagreement (value type members)

"Since the properties of a structure should themselves be value types"
Not always. Usually, but not always - heck, it's pretty rare to create your own value types anyway, but when you do you *might* have a reason to include a reference type field. I do so in Noda Time so that (say) a LocalDateTime has a reference to the appropriate calendar. This does create problems with the default value of the value type (which will have null as the value for the reference type field) but that can be handled in various ways.

Page 284, bad naming (enum)

The exercise on page 284 suggests creating a "DaysOfTheWeek" enum. It should be singular (DayOfWeek or DayOfTheWeek) by .NET naming conventions. It's also pretty pointless, given that System.DayOfWeek already exists.

Page 285, invalid code (enum underlying types)

The sample code for declaring an enum is:
```
      enum Ordinal : Int16 {First, Second, Fifth = 5};
```
This will not compile; the C# aliases *must* be used for the underlying type in an enum declaration. The correct declaration is:
```
      enum Ordinal : short {First, Second, Fifth = 5}
```
Note that I've removed the redundant semi-colon at the end of the declaration too.

Page 286, invalid code (typeof vs GetType)

This code sample is invalid:
```
      String ConstantName = Enum.GetName(GetType(DayOfTheWeek), 0);
```
The author talks about GetType as an operator a few times, before correcting herself to

typeof later on. I suspect this may be a case of mistaken cut and paste from the VB version of the book.

The corrected version is:

```
String ConstantName = Enum.GetName(typeof(DayOfTheWeek), 0);
```

Note that although Enum.GetName *can* be used with the numeric value associated with an enum value, it's more commonly used with an actual enum value:

```
DayOfTheWeek day = DayOfTheWeek.Tuesday;
String name = Enum.GetName(typeof(DayOfTheWeek), day);
```

Page 288, invalid code (exercise answer)

The following code, given as an answer to an exercise, is invalid:

```
public void ShowDay(DayOfTheWeek day)
{
 if (day > 6)
  throw new ArgumentException("Invalid day argument.");
  return;

 String DayName = Enum.GetName(GetType(DayOfTheWeek), day);
 MessageBox.Show(DayName);
```

Problems:

- There's no closing brace to complete the method
- The rogue return statement (indented as shown, as if it were part of the "if" body) will make the rest of the method unreachable
- There's no validation that the value isn't negative (which it could easily be)
- 6 is a magic number here - it would be better to use Enum.IsDefined(typeof(DayOfTheWeek), day) - or use my Unconstrained Melody library which can do similar things without boxing
- Calling GetName is considerably more complicated than calling ToString

Page 291, invalid code (interface declaration)

The example of how to declare an interface is broken as it tries to declare a method as explicitly public. This is prohibited by the C# spec in section 13.2: "It is a compile-time error for interface member declarations to include any modifiers."

Page 291, invalid code (interface implementation)

The example of how to implement an interface is broken as the property MyProperty is (implicitly) private.

Page 293: disagreement (explicit interface implementation)

"it's never wrong to choose explicit [interface] implementation"

Well if you want to be able to override an implementation in a derived class but still call the original implementation, it's wrong... or if you want to be able to call the method using "dynamic". Basically explicit interface implementation comes with a bunch of restrictions and oddities - it's a more complex choice than the one presented here.

Page 294, disagreement (frequency of casting)

"Another time it can happen is when you need to pass an object of one type to a method that declares a different type as a parameter. That's called casting, and you'll do a lot of it."
Leaving aside the matter of whether *objects* are ever passed, I'd say that if you're doing a lot of casting, that's a design smell - you should usually *see* whether you can change your design to make the code type-safe without casting.

Page 295, terminology (implicit casting)

The term used in the specification is "implicit conversion" rather than "implicit casting".

Page 296, incorrect example (missing type)

The example of casting on page 296 uses a type (MyComponent) which doesn't exist in the class diagram which is meant to show the types involved.

Page 297, disagreement (is then cast)

"It's common to use the is keyword in a pattern like the one shown - test then cast."
It shouldn't be common - because using the as operator (as shown below) is generally preferred. I'd generally only use "is then cast" when unboxing (where you *can't* use as easily, unless you unbox to the corresponding nullable value type).

Page 299, technical error (boxing)

The following code is shown (in two snippets):

```
Int32 x = 5;
x.ToString();
```

The label then claims:
"ToString() is a method declared by System.Object, so x needs to be boxed."
Except it doesn't, actually. The rules around casting are quite complex, but in this case the compiler emit a non-virtual call to Int32.ToString. In other cases it can emit a virtual call but using "constrained" in the IL. Whatever the details, there's no boxing in the case given here.

Page 302, code formatting

The indentation in the code on page 302 has various problems.

Page 303, terminology ("by value" and "by reference")

"When you pass a variable as an argument to a method, the method receives a copy of the value on the stack. In the case of reference types, it receives a copy of the reference to the values in the heap (in other words, reference by reference and value by value)."
The first sentence of this isn't unreasonable (the stack is an implementation detail, but hey) but the second sentence talking about "reference by reference" is incorrect. "Pass by reference" has a very specific meaning which *isn't* appropriate here, and it begs the question of what happens when you use "ref" with a reference type parameter.

Page 303, diagram confusion

The diagram which is meant to explain the previous paragraphs is incomprehensible to me. It's really not clear what it's meant to be showing - I can't even tell whether or not it's accurate, because I can't understand it.

Page 304, typo

The bottom bullet in the list should read "passed by reference" rather than "passed by value" - otherwise it directly contradicts the third bullet.

Page 305, terminology (passing by reference)

"The C# keyword for passing a value by reference is ref."
No, in this case the author really *does* mean variable. As per section 10.6.1.2 of the spec: "When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword ref followed by a variable-reference." So you *can't* just use any value - you *must* use a variable. The variable's *value* is not passed at all.

Chapter 10

Page 313, incomplete coverage of numeric types

The eleven numeric types mentioned in the chapter ignores System.Numerics.{BigInteger, Complex}. Even if they weren't going to be covered, they should have been mentioned.

Page 316, terminology (namespaces vs assemblies)

"We'll need to add a reference to the System.Speech namespace. You add references from the Add Reference dialog"
No, that adds a reference to an assembly. Assemblies and namespaces are different things. For example, to use most of the types in the System.Linq namespace, you need to add a reference to the System.Core assembly.

Page 318, terminology (namespaces vs assemblies)

"You must add a namespace reference to the project before you can reference it in the using directive."
You have to add an *assembly* reference before using any type within that assembly (regardless of using directives), and adding a using directive for a type or namespace which is unknown via any of the assembly references will cause a compile-time error.

Page 318, technical error (using directives and pre-processor directives)

"Although it doesn't begin with a hash sign, like #if or #define, this version of using really is a pre-processor directive"
No, it's not - at least not in the spec terminology. It's a directive, but it's not a preprocessor directive. Those are all covered by 2.5 of the spec, whereas using directives are in section 9.4.

Page 319, technical error (namespace names)

"The name of the namespace must be a valid identifier."
No, it has to be a *qualified-identifier* in spec terms. In particular, identifiers can't contain periods (so you can't declare a variable called foo.bar) whereas namespace declarations can:

```
namespace Foo.Bar { … }
```

(Oh, and namespace declarations aren't statements.)

Page 326, disagreement (CPU architectures)

"Most modern computers have 32-bit processors"
Increasingly, desktops and laptops typically have 64-bit processors. That doesn't mean it would be a good idea to start using "long" everywhere though.

Page 326, bad example code (invalid optimization)

The code on page 326 attempts to demonstrate an optimization.
The code before optimization:

```
for (Int32 x = 1; x <= 10000; x++)
{
```

```
         a = b / 123;
        }
```
The code after "optimization":
```
        c = 1 / 123;
        for (Int32 x = 1; x <= 10000; x++)
        {
         a = b * c;
        }
```
Even though this example is hopelessly incomplete (it doesn't show the types of a, b or c) it's still clearly incorrect - because in the "optimized" version, for any non-infinite values of b, a will simply be 0 afterwards, as c is 0. The code performs "1 / 123" as *integer* division, giving a result of 0.

Moral of the story: don't try to be clever. Make your code correct before you try to optimize it (and make sure you have tests so that you can validate it's still correct afterwards). Then make sure you're optimizing the right *bit* of code. Then use appropriate measuring tools to check that your optimizations are really helping. Then check that the performance benefit is worth any reduction in readability or elegance.

Page 327: partial disagreement (floating point)

"Well, here's the tricky thing about floating points: they're only approximations of a value."
It really depends on what's meant here. Each floating point value *is* an exact value. However, there may have been an approximation involved in converting it into a binary floating point value to start with. For example, the literal "0.1d" is only approximately 0.1, but it *is* an exact value in itself.

Page 327: disagreement (how much you need to understand)

"For general use, there's really only one technique you need to remember"
I think that's an over-simplification. I'm not saying you need to understand *everything* about floating point, but you're likely to come unstuck if you believe that *all* you need to know is not to compare them for equality. There's the situations in which it's appropriate to use each type, for example.

(A couple of articles I've written for those who want to know more: binary floating point and decimal floating point in .NET. There are, of course, hundreds of similar posts around the web.)

Page 327: technical error (incorrect code)

This code is suggested instead of an equality test:
```
        if (MyFloat1 - MyFloat2 < 0.0000001)
```
Problems with this:

- It should be using `Math.Abs(MyFloat1 - MyFloat2)` - otherwise if MyFloat1=0 and MyFloat2=100000, that's still going to evaluate to "true"...
- It fails to take scale into account. Two tiny, tiny values may need to be very close indeed (with a difference far smaller than 0.0000001) to be usefully considered equal - whereas two very *large* values (e.g. 10^{20}) could sensibly be considered "equal" even

if they differ by a thousand or more.

Page 328, terminology (decimal)

"Like a floating point number, the .NET Framework Decimal type is also used to store real numbers."
Not just *like* a floating point number - decimal *is* a floating point type. It's just a floating *decimal* point type instead of a floating *binary* point type.

Page 328, confusion and incompleteness (literals)

"Like any value type, decimals can be initialized by assigning them a value directly"
It's not really clear what's meant by this - possibly that there are literals for decimal? If so, it would be useful to show an example (e.g. 0.1m) - although there certainly aren't literals for *all* value types. The suggestion to call the five-parameter constructor is somewhat bizarre - I think it's *highly* unlikely that an entry-level programmer is going to need that any time soon.

Page 330, formatting error

There's an odd line break about a third of the way down page 330, with a line containing just "and" on its own.

Page 331, ambiguous exercise

"Determine whether Char1 or Char2 comes first in the alphabet"
This is ambiguous - does 'X' come before 'a'? Does 'x' come before 'A'? What about non-alphabetic or non-ASCII characters?

Page 332, technical error (Unicode characters)

"... you should be aware that a single logical character might map to more than one Unicode character. Accented characters, for example, require two characters."
No they don't. Look at the Unicode code chart for C1 controls and Latin-1 supplement - lots of accented characters there, in single characters. Now you *can* normalize these to separate the accent from the character it's applied to, but you certainly don't have to.

It's possible the author was really thinking of characters outside the Basic Multilingual Plane, which require two UTF-16 code units to represent a single Unicode code point - but that's getting somewhat more technical, and it's unlikely that the target readership is really interested in going that deep.

Page 333, technical error (String is a class, not a struct)

"In the .NET Framework, they are represented by the System.String structure. Like any value type, you can instantiate a String directly or by calling one of its constructors."
It's still not clear what is meant by "instantiate a String directly" but String is definitely *not* a value type. From section 4.2.4 of the spec: "The string type is a sealed class type that inherits directly from object."

Page 333, technical error (supposed conversion from char to string)

(With an example of using a string literal to assign a value to a String variable.)

"Make sure to use double quotes here. Single quotes delimit a Char, not a String. If you use them, the code will work, but it will call an extra (unnecessary) conversion operation."
No, it will fail to compile. Trying to use multiple characters in a char literal will give CS1012 ("Too many characters in character literal") and trying to assign a valid char literal to a string variable will give CS0029 ("Cannot implicitly convert type 'char' to 'string'").

Page 333, typo in code example

The code example given contains:

```
String MyString = "John said, \"My Document\Files.\"";
```

There should be an "s" on the end of Document. The same mistake is repeated in the "fixed" version using a verbatim string literal.

Page 333, terminology (escape sequence)

"\r" is referred to as "return" in the book; it's more commonly "carriage return". (Additionally, I would personally use "line feed" for "\n" but here the spec agrees with the book, calling just "new line".)

Page 334, confusing example (string literals)

Here's a code sample at the top of page 334:

```
String String1 = "Hello";
String String2 = "Hello";
Boolean TheSame = (String1 == String2);
```

This doesn't achieve what it looks like it does. The same code will produce the same result in Java, even though Java doesn't have operator overloading, so would be comparing references. This code is comparing two *identical* references - String1 and String2 refer to the exact same object, not two *equal* objects. (That's guaranteed by the spec section 2.4.4.5: "Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator (§7.10.7) appear in the same program, these string literals refer to the same string instance.")

Now in C#, the two strings *are* actually compared by semantic value when the == operator is used, due to operator overloading (but only when the operands' compile-time types are both string; another subtlety). However, this example doesn't require (or therefore prove) this. It's a weak test, which doesn't really explain what's going on.

Page 334, inappropriate analogy (Windows Explorer)

"You may have noticed that Windows Explorer sorts "File 20" before "File 5"."
No it doesn't - I've just tried. It did at one point, but Windows XP introduced a smarter version of Explorer which has a natural sort order - it applies a certain amount of semantic intelligence to file names.

Page 334, unexplained subtlety (string concatenation)

Here's some code which is given as an example of string concatenation:

```
String MyString = "Hello, " + "World!";
```

The text implies that the + operator here is just a shorthand for calling String.Concat - but in this case the compiler performs the concatenation at *compile-time*, so String.Concat isn't executed at all.

Page 337, terminology (immutable)

"Most of the value types in the .NET Framework are immutable, which means that when you assign a new value to them, the CLR will actually create a new instance of the type and throw the old one away."
That's not what "immutable" means at all. Read Eric Lippert's blog for a detailed discussion of various kinds of immutability.

Page 337, technical error (strings)

(In the context of value types.)
"The exception, of course, is Strings, which can sometimes be very, very big indeed."
Except that String isn't a value type.

Page 337, technical error (mutability)

"The StringBuilder class is a reference type, and like all reference types, it is mutable."
No, reference types are *not* all mutable. System.String is pretty much the canonical example of an immutable reference type.

Page 339, disagreement (strength of .NET's date/time APIs)

"But I warn you, even with the excellent support the .NET Framework provides, handling dates and times is always tricky."
I agree that working with dates and times can be tricky - but I strongly disagree with the claim that .NET's support is "excellent". If it were excellent, I wouldn't have started the Noda Time project. Please read my blog post entitled "What's wrong with DateTime anyway?" for more details of how the .NET date/time support is far from excellent.

Page 339, technical error (meaning of DateTimeOffset)

"The DateTimeOffset represents a particular date or time in a particular place."
No, DateTimeOffset knows nothing about *place*. It represents an instant in time with an associated local offset from UTC, but that offset may be valid for that instant in multiple places with different time zones.

Page 343, terminology (Julian dates)

"When you hear someone talking about "Julian dates", they almost certainly mean a date in the month, day, year format common in the United States"
I've never heard it used to mean that. I've usually heard it with a meaning of something like "the number of days since the Julian epoch of January 1, 4713 BC" - and potentially some other meanings, but nothing about just a month/day/year format.

Page 345, disagreement (power of TimeSpan)

When talking about periods of time:
"All that tedium goes away with the .NET Framework TimeSpan structure."
No it doesn't - TimeSpan is fine if you only care about units which never change in length - hours, minutes[1] etc. But try finding out how many months are between two DateTime values,

[1] Depending on which time system you use, minutes and hours don't have to be constant time either, due to leap seconds. In practice, I don't believe that .NET has any handling built-in for leap seconds.

and it becomes considerably trickier.

(Noda Time doesn't either.) This is a source of some relief to be honest, as I find leap seconds incomprehensible. Most business applications really don't need to care about them. A *day* can sensibly be said to not be 24 hours though, due to daylight saving time changes. That's only appicable when you start considering times in a particular time zone though. Date and time is *really* complicated...

Chapter 11

Page 357, terminology (awful use of "set")

The word "set" has a very clear meaning within computer science, which is very much *not* the same as the author's use. A "set" can only contain any particular element once, and is usually unordered. In this chapter, the author is talking about collections of all kinds - not just sets. The excuse of "the framework has a type called Collection" doesn't really wash when the framework *also* has types called ISet, HashSet and SortedSet - which follow the *traditional* meaning of the word "set". This misappropriation of the word "set" is just asking for trouble.

Page 359, technical error (array mutability)

"Arrays are immutable, like Strings."

No they're not. They're really not. The *size* can't be changed after you've created one, but you can mutate the contents to your heart's content:

```
int[] x = new int[2];
x[0] = 5;
x[1] = 10;
x[0] = 15;
```

Page 359, technical error (resizing arrays)

"It is possible to change the size of single-dimension arrays using the static Resize() method."

No it's not. That doesn't *actually* change the size of an existing array. It creates a *new* array and copies the contents of the old one into the new one, which isn't the same thing at all.

Page 359, technical error (array equivalent of StringBuilder)

"There's no Array equivalent of StringBuilder"

Well that's pretty much what List<T> and ArrayList are, really. Like StringBuilder (or at least old implementations of StringBuilder) they maintain a buffer, allowing you to "resize" the list by adding or removing elements, only copying where necessary. It's really a pretty striking similarity.

Page 359, technical error (broken code)

The code at the bottom of page 360 is broken:

```
String[][] Menu = {
 new String[] = { "a", "b" },
 new String[] = { "one", "two", "three" },
 new String[] = { "x", "y", "z", "omega" }
}
```

Problems:

- The inappropriate use of =
- No trailing semi-colon.

Fixed code:

```
String[][] Menu = {
 new String[] { "a", "b" },
 new String[] { "one", "two", "three" },
 new String[] { "x", "y", "z", "omega" }
};
```

Page 361, typo (broken code)

More broken sample code:

```
String[] Weekend = new String[2];
Weekend[0] = "Saturday";
Weekend[1]" = "Sunday";
```

The double quote after [1] should be removed.

Page 363, typo (broken syntax description)

The syntax description of foreach at the top of page 363 is missing a closing) on the first line.

Page 364, nasty code (if condition return true else return false)

Here's a pattern which crops up repeatedly, and first shows up (that I noticed) on page 364:

```
Boolean IsWeekend(String theDay)
{
 if (theDay == "Saturday" || theDay == "Sunday")
  return true;
 else
  return false;
}
```

Normally I'd suggest changing something of the form "if (condition) return x; else return y;" into "return condition ? x : y;" but in this case it's even simpler:

```
Boolean IsWeekend(String theDay)
{
 return theDay == "Saturday" || theDay == "Sunday";
}
```

No need for an if statement or anything like it. This crops up again and again, and it makes me wince every time.

Page 367, technical error (arrays are reference types)

"But the Array is a value type, and as such is immutable."

As mentioned before, arrays aren't immutable - and array types (including Array itself) are definitely reference types, not value types.

Page 367, odd explanation (author doesn't know about Array.IndexOf?)

(About ArrayList)

"... and because the elements are Objects, you can search the set without using a predicate"

Just like you can with arrays, using Array.IndexOf. Maybe there's a different point being made here, but if so it's not clear.

Page 370, technical errors (use of ArrayList indexer vs Add; reader contribution)

The answers to the exercise on page 369 *implies* that this code will create an ArrayList and then set the third item to "Hello":

```
    ArrayList MyAL = new ArrayList(10);
    MyAL[2] = "Hello";
```

That will actually fail with an ArgumentOutOfRangeException on the second line. The author isn't distinguishing between the list's *capacity* (which is 10) and its actual *size* (which is 0 to start with). You can't use The ArrayList indexer to *add* new items to a list, only to change the value of elements within the existing range. We'd have to call Add at least three times (or populate the list in another way) before using the indexer like this.

A similar (but very slighlty different) mistake occurs with the next part of the exercise:
'Insert "World" as a new item at the end of the MyAL'
This time the correct answer is given:

```
    MyAL.Add("World");
```

but there's a comment by it:
'You could also have used something like `MyAL[MyAL.Count - 1] = "World"` but that would be far more complicated'

That would *not* have added a new item to the end of the list - it would have *replaced* the final element of the list (or failed if the list was empty).

Page 371/372/374, technical error (Hashtable vs Dictionary)

The author appears to think that keys in a Hashtable don't have to be unique. The text in the book is somewhat unclear about the difference between keys and hash codes. Page 371 has this in a label:

"The Hashtable and Dictionary store items in key/value paiors. The keys of a Dictionary must be unique. Hash codes don't have to be, but it's best if they are."

An exercise then asks the reader to consider the right collection to model real world items, with these descriptions:
"A rolodex has one card for every phone number."
"A dictionary has a single entry for a word, but a word can have multiple meanings."

The answer given is:
"A Rolodex is like a Dictionary, while a dictionary is like a Hashtable."

A later exercise gives this as a question to be fixed in:
"The main difference between a Hashtable and a Dictionary is that the key of a Dictionary must be:"

The first label *could* be put down to just a confusing description (possibly hinting at hash code collisions), but the rest suggest that the author thinks there's a real difference here in terms of key requirements. There isn't. Both Dictionary<TKey, TValue> and Hashtable can store a single value per key, where keys are compared for equality using Equals and GetHashCode (optionally via something like IEqualityComparer).

Page 377, confusion (generics)

The use of the words "generic" and "generics" are very confusing here:
"Instantiating a generic" - a generic *what*?
"Generics are also a kind of parameter" - this sounds like the author is talking about type parameters, rather than (say) generic types or generic methods.
"Notice the <T>. That's the syntax for a generic parameter." This should definitely be "generic type parameter."

Page 378, technical error (numbers of type parameters)

The exercise refers to Dictionary<T>, SortedDictionary<T> and SortedList<T>. All of these types *really* have two type parameters: Dictionary<TKey, TValue>, SortedDictionary<TKey, TValue> and SortedList<TKey, TValue>.

Page 380, technical error (misunderstanding of HashSet<T>)

"This type is like a Dictionary, but the Add() method takes just the value and calls GetHashCode() to generate the key."
This is a fundamental misunderstanding of what HashSet<T> is about. Yes, hash codes are used - but not as unique keys. Conceptually a HashSet<T> just has elements, and the hash codes of the elements are used to make checking to see whether the set contains any candidate element really quick.

Page 380, technical error (misunderstanding of SortedList<T>)

"This type is like a SortedDictionary, but you have to provide a custom routine to compare the keys."
No you don't. You *can*, but you don't have to. The difference is mostly in terms of the internal data structures and the corresponding performance characteristics.

Chapter 12

Page 397, typo

The description of encapsulation includes "encapsulatin" in the second line.

Page 400, technical error (code sample)

The FinancialDetails class at the bottom right of page 400 contains a read-only automatically-implemented property:

```
public Decimal AccountBalance {get;}
```

This won't compile.

Page 400, bad practice (inappropriate mutability)

The FinancialDetails class (which is a somewhat vague name to start with) allows the name of the bank and the account number to be mutated. I can't imagine a situation where this would be a good idea.

Page 401, confusion over casting

Page 401 starts with a claim which sounds very odd:
"Polymorphism isn't the same as casting. When you cast a type to another, it behaves like the new type. The code that's executed is that of the type to which it is cast, not the original type."
That's not *generally* true - in particular it's not true when it comes to virtual members, assuming the cast is actually performing a reference conversion. However, the "truth" of this is demonstrated using one member hiding another - which is rarely a good idea, and rarely comes up in well-designed code in my experience. Given that this page doesn't really explain anything about overriding etc, one would *expect* that this code would print System.String:

```
Object x = (Object) "this is a string";
Console.WriteLine(x.ToString());
```

(The cast is obviously unnecessary here, but it's just to comply absolutely with what's written in the book.) Here a reader could reasonably expect the ToString method call to execute the code in the Object implementation, which returns the name of the type of the object. Of course, it doesn't - because ToString is overridden in String.

Page 402, missing backreference

"We've also seen that the Object.GetType() method (inherited by every type in the .NET Framework) returns an instance of the Type class"
I don't remember seeing that anywhere... was it removed in the editing process?

Page 403, invalid code (try/catch without braces)

The code at the bottom of page 403 is missing braces:

```
try
 ISomething newThing = (ISomething) thing;
catch
 throw new ArgumentException("Can't use thing");
```

Braces aren't optional for try/catch statements.

Page 403, invalid code (missing closing brace)

The code at the bottom of page 403 is missing a closing brace for the "else" statement.

Page 404, invalid code (missing closing brace)

The code is still missing a brace in the "answer" version on page 404, although it's not clear *which* brace is missing, as the final closing brace given isn't aligned with any opening brace.

Page 406 / 407, invalid code (missing "class" keyword)

All seven class declarations on pages 406 and 407 are missing the "class" keyword, e.g.

```
public sealed Recipe
{
}
```

Page 406, invalid code (method missing braces entirely)

The TabsToTsps method (which would be better named TablespoonsToTeaspoons in my view) doesn't have any braces at all, and so is invalid.

Page 407, invalid code (class should be abstract)

The Recipe class at the top of page 407 contains an abstract method, so must be declared abstract.

Page 407, invalid code (methods missing return types)

The Recipe and RecipeChild classes at the bottom of page 407 contain DoSomething methods which are declared without a return type.

Page 408, technical error (static classes and methods)

"In theory, a static class can be inherited and a static method can be overridden, but best practice dictates that they should also be sealed."
No, you *can't* derive from a static class (spec section 10.1.1.3), you can't override a static method (spec section 10.6.4) and you can't seal either of them (sections 10.1.1.3 and 10.6). Page 409 repeats this error.

Page 408, terminology (inheriting vs overriding)

"Sealed members cannot be inherited"
Yes, sealed members *are* inherited - they just can't be overridden, which is a very different matter.

Page 408, terminology and a typo ("replacing" with new, and overrides vs override)

"A class with virtual members provides a default implementation, but the members can be replaced in classes that inherit from it using the new or overrides keywords"
Using "new" to *hide* a member doesn't *replace* it. The implementation is still present and will be used by anything using a compile-time type of the base class. That's completely different to "override" which effectively *does* override the implementation (although still available via non-virtual base.Foo() calls, of course.)

Additionally, in C# the keyword is override, not overrides. I suspect this is a cut and paste error from the VB version of the book.

Page 409, technical error (singleton pattern)

In the context of when the static keyword is useful:
"When there should only ever be a single instance of the object. This is callerd the singleton pattern. The .NET Framework Application class, for example, provides methods and properties that let you manage your application as a whole. Since there's only ever one instance of the application itself within the application scope, Application is a static class."
A static class is *not* an example of the singleton pattern. The singleton pattern enforces that only a *single* instance of the type is ever created. That instance is then used with normal instance members. A static class enforces that there are *no* instances of the type. See my article on the singleton pattern for how it's *actually* implemented in .NET.

Page 409, bad practice (global variables)

"sometimes you need certain values to be available to every object in the application. The name of the current user, for example, or the user's interface preferences are good examples of this kind of information. A static class is a good option for storing this kind of information."
If it's really a dependency of every type, then inject an instance *as* a dependency into every type - that way it's really clear that it's a dependency, and you increase testability. I can't see why the *business logic* of an application should care about the user's interface preferences, for example - or indeed why most of an application is likely to care about the name of the current user.

Global state is almost always a headache, and should be avoided wherever possible.

Page 409, incompleteness (extension methods)

One of the uses I *do* like for static classes is to host extension methods - which aren't mentioned on page 409, but should be in my view.

Page 411, disagreement (abstract class members)

"An abstract class is very like an interface. It defines the members that must be implemented by an inheriting class, but doesn't usually provide any implementation."
On the contrary, in my experience an abstract class usually provides implementations of many members, only leaving a *few* abstract - possibly to be used by the concrete implementations, as per the template method pattern. An abstract class which doesn't define *any* implementation would be quite unusual.

Page 412, disagreement (usefulness of the class designer)

"Where the Class Designer really comes into its own when working with abstract classes is in implementing the necessary members in their descendants. Simply right-click an inheriting class and choose Implement Abstract Class from the Intellisense menu."
Or just declare the base class in the normal code editor, put the cursor in the abstract class name and hit Ctrl-Period... there's really no need to get a designer involved here.

Page 412, invalid and bad code (missing class keyword, bad naming conventions)

The code on page 412 attempts to declare a public abstract class "myClass" with public properties "myProperty" and "myAbstractProperty". All of these names violate .NET naming conventions, and the class declaration doesn't include the "class" keyword, making it invalid to start with.

Page 415, terminology ("semi-abstract class")

There's no such term as "semi-abstract class" in C#. The implication in the text that an abstract class forces *all* members to be overridden is incorrect. An abstract class doesn't even have to have *any* abstract members - if it's declared to be abstract, it's abstract.

Page 417, poor example

The code example which demonstrates the first use of an overridden method is a poor one. (MyChild derives from MyParent; MyParent declares the SayHello method and MyChild overrides it.)

```
MyChild x = new MyChild();
x.SayHello();
```

This would invoke the SayHello() method in MyChild whether it was overriding the base class implementation *or hiding it*. It would be a much better example if it actually showed polymorphism in action:

```
MyParent x = new MyChild();
x.SayHello();
```

Page 419, bad naming

Just like page 412, the classes on page 419 all violate .NET naming conventions. (They do include the "class" keyword though.)

Page 419, terminology (overridable vs inheritable)

"It might seem that only being able to mark a method as sealed when you use the override keyword is a limitation, but in fact the definition of DontTouchMe() in myNew above is effectively sealed. Do you know why? What keyword would you need to add to make it inheritable?"

Again, the method will be inherited by any derived class anyway. It just can't be overridden. See section 10.3.3 of the specification for what is inherited in a class.

Page 421 and 422, technical error (extension methods don't have to be public)

When describing how to write extension methods, the book talks about declaring a public static class and public static methods. Both of these could be internal instead - extension methods don't have to be public. This error is repeated on page 422, with:

"The class and the method must both be public and static."

(They *do* have to be static, but not public.)

Page 421, unclear description

"The first parameter of the method must be the name of the class the method extends, preceded by the keyword this."

That's an odd way to describe it, in my view. I would use something like:

"An extension method must have at least one parameter. Simply add the "this" keyword before the type of the first parameter to make it 'extend' that type. Note that the first parameter can't be 'ref' or 'out' parameter."
(Note that the type doesn't have to be a class.)

Page 421, terminology (using directive, not using statement)

"After you've written the extension, write the using statement and an example of calling the extension method"
This should be using *directive*.

Page 422, technical error (invalid code)

The example of an extension method declaration looks like this:

```
      public static void PrintRecipeCard(this Recipe)
```

The parameter doesn't have a name, so is invalid.

Page 422, terminology (namespaces aren't referenced)

"To call the method, you'll need to reference the namespace"
Here "reference" sounds more like adding an *assembly* reference. A using directive *imports* the namespace (as per section 7.6.5.2 of the C# spec: "If namespaces imported by using namespace directives in the given namespace or compilation unit [...]"

Page 422, terminology (extension methods can't "override")

"You cannot re-define a method using extension methods. The compiler will always give precedence to the instance method, so your override will simply never be called."
The word "override" is misleading here - "extension method" would be *much* better, as an extension method can never even *try* to override a method. (Static methods can't use the override modifier at all.)

Chapter 13

Page 429, disagreement (code duplication impact)

"If there's any bit of code that appears in more than one place, you've got a smelly problem"
That's an overstatement to my mind. In particular, it can lead to artificially trying to make things look the same (often via an abuse of inheritance) when they're really not. One motto of my current team leader is that "things are different until they've been proved the same" - as humans we're always *looking* for patterns, and we're all too fond of finding them when they're not *really* present.

The risk of having to change some logic in more than one place is very real - but so is the risk of having one piece of logic which actually represents two operations which merely *start off* doing the same thing. When you later want to make those two operations do *different* things, you can end up in a real mess if you've only got one piece of code to represent both of them.

I'm not suggesting cut and paste willy-nilly, but just that having the same logic in two places *isn't* necessarily a smell, as it allows independent evolution of two distinct operations.

Page 429, disagreement (commenting)

Another instance of an earlier disagreement:
"You should write code that makes comments seem unnecessary. (But comment everything anyway.)"
I would say you should try whenever possible to write code that makes comments *actually* unnecessary. Code which has comments for every operation, however obvious, is very tedious to read and maintain - and the comments can get out of sync with reality remarkably quickly unless you're very careful.

Page 437 / 438, invalid code (Integer instead of Int32)

The method on page 437 (and 438) uses a parameter type of Integer - it should be Int32.

Page 437 / 438, poor naming and possibly invalid code

The code on page 437 (and 438) uses a variable called "servingsModifier" which is described with "servingsModifier contains the amount for a single serving" - but it's *initialized* as:

```
servingsModifier = this.Servings / servings;
```

Problems:

- That looks like a *scaling factor* to me - not an "amount" of anything.
- The variable isn't declared in the method; it's not clear whether it's meant to be an instance variable, or whether this is just invalid code.
- Both Servings and servings are integers, so if the existing value of Servings is (say) 24 and you ask for 47 servings, you'll end up with a servingsModifier of 1, which really isn't desirable.

Page 437 / 438, poor modelling

After the previous line comes this loop:
```
foreach (RecipeItem ingr in Items)
{
 ingr.Amount *= servingsModifier;
 ingr.GetIngredientNutrition();
}
```
Problems:

- This modifies the existing recipe - it would be much better in my view to create a new instance of Recipe with the scaled values. It *may* even be worth having two separate classes - one for the recipe "template" and one for the recipe "instance" that we want to print, execute, whatever. They feel conceptually different to me - for example, it might make sense to make the "instance" version have a concept of how far we've got through the list of steps.
- It's using integer arithmetic, with a bunch of problems (Amount is an int). This is a separate instance of the problem shown in the previous item.
- It's not clear what the GetIngredientNutrition method does, but using a method prefixed with Get but ignoring the return value seems like a very bad idea to me. Either this should be RecalculateIngredientNutrition, or it should be a non-void method and the return value should be used.
- The new number of servings is never stored - so if you have a recipe with an initial value of Servings of 1, and you repeatedly call RecalculateRecipe(2) you'll end up doubling the ingredient quantities every time.

While it's possible that these problems are mentioned later (I don't *think* they are) they shouldn't be left to stand without a warning here, in my view.

Page 437 / 438, code formatting

The brace which closes the "if" statement on both page 437 and 438 is indented too far.

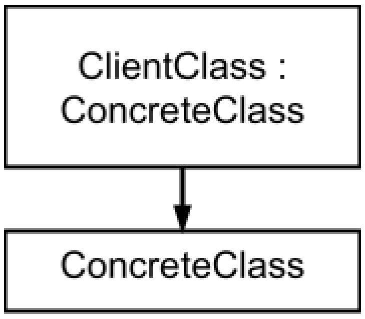
Page 443, terminology

"you have to create a String or Int16 (which are concrete classes, after all)"
Int16 isn't a class, it's a struct. Concrete *type* would be correct though.

Page 443-445, confusion (dependency inversion and programming to interfaces)

It's entirely possible that I don't understand dependency inversion, but these pages *look* like they're mistaken to me. I'm more familiar with "dependency injection" and "inversion of control". I'm not going to claim to be any sort of authority on the relationship between these three, but the description given seems incorrect to me.

We have two classes ClientClass and ConcreteClass which are initially shown like this:

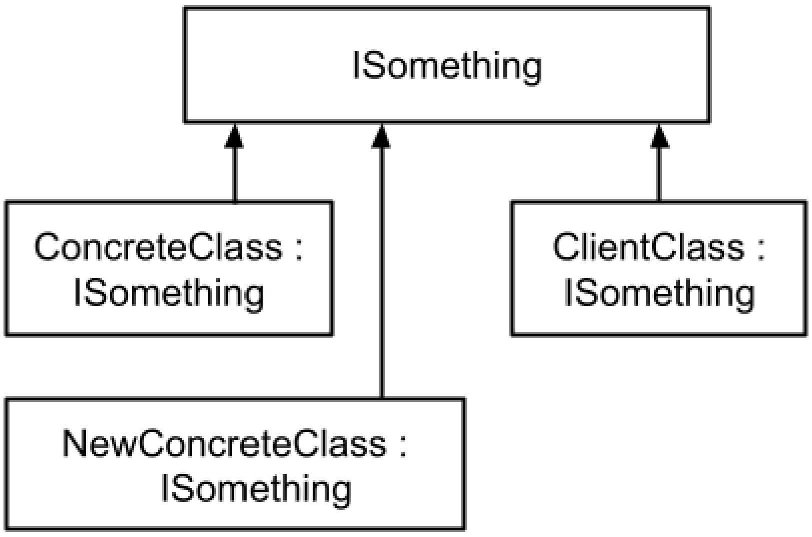


To start with, it's not clear what's going on here. Does ClientClass just *create* an instance of ClientClass, or does the ":" really represent inheritance (as in a normal class declaration)?

There's a label of "When a client class instanties a concrete class, it is dependent upon it. If ConcreteClass changes, ClientClass needs to be (at the very least) re-compiled."

That doesn't really clarify the diagram for me - and it's also incorrect in that if the API of ConcreteClass doesn't change (just the implementation) then ClientClass shouldn't need to be recompiled.

The improved version (i.e. after applying dependency inversion) looks like this:



The label for ClientClass says: "The dependency is inverted when both classes depend on an abstraction" - but it's not clear how those dependencies are expressed. I'd expect ClientClass to *use* on the interface (i.e. access the members of the interface, given an implementation) and ConcreteClass to *implement* the interface.

This makes it *look* like ClientClass is meant to *implement* the interface - but surely it should just be *using* the interface (e.g. as the type of an instance variable within the class). It's entirely possible that this is what the author meant, but the "ClientClass : ISomeInterface" part *suggests* that it implements the interface.

Being a code-oriented kinda guy, I'd have found this clearer (with some description, of course):

Bad code, before - tightly coupled, inflexible, and hard to test:
```
public class SomeDependency { ... }

public class Client
{
    private readonly SomeDependency dependency;

    public Client()
    {
        dependency = new Dependency();
    }

    // Code using dependency
}
```

Better code: type of dependency is abstracted, actual instance is injected, allowing for alternative implementations in production, and test doubles for testing.

```
public interface ISomeDependency { ... }

public class SomeDependency : ISomeDependency { ... }

// Maybe at some point...
public class AnotherImplementation : ISomeDependency { ... }

public class Client
{
    private readonly ISomeDependency dependency;

    public Client(ISomeDependency dependency)
    {
        this.dependency = dependency;
    }
    // Code using dependency
}
```

Now it's clear where the interface comes in, how the client uses the interface, how the concrete class implements the interface etc. Some extra text could describe IoC containers, potentially - not that they're required to use this pattern, of course.

Page 448, code formatting and parameter type

The method at the bottom of page 448 is shown as:
```
ChangeServings(Integer
desiredServings)
{
 //do nothing
```

```
        }
```
This has the same problem we saw earlier of using Integer instead of Int32, and for clarity the parameter name should be moved to the same line as the rest of the declaration - there's plenty of room.

Page 451, naming conventions

I know it's only meant to be demonstrating the Law of Demeter, but this should still use more conventional names, in my view (ideally with a realistic example, such as customer addresses):

```
        aFriend = new MyFriend();
        aFriend.aFriendlyProperty = 3;
        aFriend.aFriendsFriend = new aFriendsFriend();
```
(As an aside, I feel the importance of the Law of Demeter is often overstated. It's worth bearing in mind, but it's very context-sensitive.)

Page 467, typo

The interface shown in the class diagram is named IYieldBehaviors (plural) but in the text it's singular (as it should be). The same mistake occurs on page 469.

Page 477, technical error or lack of clarity

"Because a delegate defines only a method signature, any class that exposes a method with the correct signature can respond to an event."
The word "exposes" here could *imply* that the method has to be public or internal. It only needs to be visible to whichever code creates the delegate instance - it's quite common to use private methods for delegates.

There's also the matter of anonymous functions (lambda expressions and anonymous methods), but that's more a matter of completeness.

Page 478, lack of clarity (convention vs requirements for events)

"The second argument passed to an event handler is an instance of System.EventArgs."
That's certainly the convention, but it's not actually required for events.

Page 478, technical error (delegate requirements)

"All delegates have the same basic pattern: they are declared as public, return void, and have two arguments."
Note that this is talking about *delegates*, not events. Even bearing conventions in mind, that statement is certainly not true for delegates. They don't need to be public, they can have different return types, and any number of parameters.

Page 478, incompleteness (event declaration syntax)

"Of course you need to declare the event so the CLR knows about it. The syntax is simple:

```
        public event <DelegateType> <EventName>;
```
That's *one* way to do it (declaring a *field-like event*) - although it doesn't have to be public, of course. Even if the author doesn't want to go into the longer declaration syntax, it would be better to at least acknowledge its existence.

Page 478, bad code (event raising method)

This code is provided for raising an event:

```
protected void On<EventName>(<EventName>Args e)
{
 <EventName>(this, e);
}
```

That will fail with a NullReferenceException if there are no subscribers to the event. More idiomatic code would be something like:

```
protected void OnSomeEvent(EventArgs e)
{
    EventHandler handler = SomeEvent;
    if (handler != null)
    {
        handler(this, e);
    }
}
```

(Adjust according to delegate type and event name, of course.)

Page 479, bad code (public event handler)

The event handler method shown in bullet 1 of page 479 is public. This is very rarely a good idea - an event handler method should almost never be called or otherwise referred to, apart from the event subscription code itself, which is usually within the containing type. It's therefore better to make it private.

Page 479, technical error (delegate constructor)

"The constructor for a a delegate take the method name as its only parameter."
Not really. The *delegate-creation-expression* in question uses a method group, but that name isn't passed to a constructor as such. The compiler deals with the translation, basically. See section 7.6.10.5 of the C# spec for more details.

Page 479, incompleteness (only the C# 1.1 delegate creation expression is given)

This code is given to create an instance of EventHandler and assign it to a variable:

```
EventHandler myDelegate = new EventHandler(MyEventHandler);
```

From C# 2 onwards, you don't need to be this verbose:

```
EventHandler myDelegate = MyEventHandler;
```

This can be used for event subscription as well. This isn't shown anywhere in the book as far as I can tell. For example, the code on page 480 can be simplified to:

```
myEvent += MyEventHandler;
```

The other problems highlighted above (a public event handler and an unsafe "raising" method) are also present on the complete example in page 480.

Page 491, disagreement (role of Silverlight)

Silverlight is described like this:
"Silverlight is closely related to WPF. It runs over the Web (as a replacement for Web Forms), on the desktop (with some limitations) and on mobile devices like Windows Phone. If you ned [sic] to deploy your application on multiple platforms, Silverlight is probably your

best option."
I don't think it's a good idea to talk about Silverlight acting as a replacement for Web Forms. ASP.NET MVC is the technology which best fits that description, in my view - and it's also probably the best bet for a truly portable solution. That's a replacement in terms of achieving the same aim (a genuinely portable web application) with a better technology stack. Changing from a web app to a rich-client application (whether it's Silverlight, Flash or something similar) is a more fundamental shift than one implementation stack replacing another.

P491, typo

As shown above, "ned" should be "need".

Chapter 15

Page 514, confusing comment

"... have you noticed that XAML is really smart about interpreting value types? In C# you have to be really careful about using 1.0 to specify a Double rather than an Integer [...]"
Leaving aside the use of Integer instead of Int32 for the moment, it's not clear what benefit is being claimed here. I'm *glad* that the C# compiler will flag up the use of a floating-point literal being assigned to a variable with an integer type - it keeps my code clearer. If I want the integer 1, I'll write 1 rather than 1.0. If this *is* the point the author was trying to make, I disagree with it - if it's not, then I'm just confused about what she was trying to say.

Page 514, VB-ism

The above quote continues like this (from the start of that second sentence)
"In C# you have to be really careful about using 1.0 to specify a Double rather than an Integer, or #01/01/01# for a date."
The "#01/01/01#" part really gives away that this was either cut and paste from the VB version of the book, or the author was at least thinking in VB when she wrote this. C# has no literal syntax for DateTime values.

Page 521, typo (I think)

(About the Children and Parent properties)
"You can use these properties to negotiate up and down the logical tree."
It feels like this should probably be *navigate* rather than *negotiate*. It's possible the author really did mean negotiate though.

Chapter 16

Page 535, illegible text due to bitmap rendering

The diagram which makes up the majority of page 535 is rendered as a bitmap for some reason - and it's been rendered at a relatively low resolution, which leaves some of the labels very hard to read. (Most diagrams don't have this problem, although the one on page 554 isn't great either.)

Page 545, arrows are misaligned

On page 545 there's a sort of "logical screenshot" containing five rectangles. There are five arrows which are meant to show which three are TextBox elements and which two are Rectangle elements - but some of the arrows point into empty space, and two point to the same element.

Page 561, unexplained reference to nullable value type

Page 561 talks about the result of ShowDialog():
"To open a window as a dialog, you use the ShowDialog() method, and you have to capture the result in a nullable Boolean. (It has to be nullable because not all windows with return a result.)"
I don't believe nullable value types are covered anywhere in the book. Page 126 implies they'll be covered in chapter 10, but other than the word "Nullable" appearing faintly in one diagram, I can't see any explanation of them.

Page 579 / 580, bad practice (suggesting the use of GetType()

From the exercise on page 579:
"Our recipe hierarchy has two levels. You'll only want to add a new tab if the user double-clicks a second level item [...]. How do you think you can check that? (Hint: remember the GetType() method?)"
The suggested solution on page 580 is:
"You can test for Parent.GetType().Name == "TreeView"."
That's brittle in terms of:

- Inheritance (i.e. a subclass of TreeView should probably still count)
- Susceptible to typos (hard-coding a name as a string)
- Namespace-ambiguous (so a parent which was a TreeView from a different namespace would still count - is that really desirable?)
- Null-sensitive (it will throw an exception if Parent is null) - I don't know whether Parent ever *will* be null, admittedly.

Using "is" instead would fix all of these problems - but fundamentally this feels like something we shouldn't be asking the *UI* anyway - it should be about the *data*, shouldn't it? I'm not a front-end programmer by any means, but this just feels like a bit of a hack. Maybe it's the way that things work in WPF (in which case at least use the "is" operator) but I'd rather have a cleaner solution, such that each item had event handlers attached which were appropriate for the kind of data it was displaying.
(This code is then present on page 586, as part of a fuller example.)

Page 586-587, ugly code (use an object initializer)

Much of the code on pages 586 and 587 is just setting properties and adding elements to collections. These are both best handled using the object/collection initializer syntax introduced in C# 3 - and again, I'd argue that if room was too tight to just *add* information about object/collection initializers, there are plenty of trade-offs I'd have been happy to make in order to find room for them.

Chapter 17

Page 592, typo (three rather than two)

"There are three differences in the field associated with a dependency property"
There are then two bullet points, leaving the reader wondering what the third difference is.

Page 592, confusing description

(This is in the same section, talking about *fields*. It's the second difference.)
"It is declared public, static and readonly, instead of private, which is considered best practice for a normal property."
The final word should probably be "field" rather than property, and it's not entirely unusual to have a public static readonly field - String.Empty is such a field.

What *isn't* mentioned is that we're really comparing apples with oranges here - a "normal field" stores the data value itself (or a reference to the object containing the value). In the case of a dependency property, the value of the field is an extra level of indirection - it's *metadata* about the property, really.

Page 593, unnecessarily verbose code

The field + property implementation snippet at the top of page 593 would be more idiomatically implemented using a single-line automatically-implemented property.

Page 597 and 598, typo (bullet numbering)

The numbered lists on both page 597 and 598 contain "3" twice per list, and no "2".

Page 615, incomplete description (delegates)

"Remember that a delegate is simply a data type that holds a method, as opposed to, say, a String or an instance of the RecipeItem class."
While this probably wouldn't be the right place to bring it up anyway, delegates can hold references to *multiple* actions, not just one. (The separation between System.Delegate and System.MulticastDelegate is a red herring as in practice all delegate types derive from MulticastDelegate.)

Page 615, invalid code (typo of period for semi-colon)

Not only does the example at the bottom of page 615 use the nasty "if (condition) return true; else return false;" pattern, but it fails to do so correctly due to a typo - there's a period after "return false" instead of a semi-colon.

Page 617, typo

In the first paragraph:
"Like the coerce value method, the method reference by the PropertyChangedCallback delegate"
I suspect "reference" should be "referenced" here.

Page 618-620, opportunity for method group conversions

The calls to the PropertyMetadata constructors are another great example of where using method group conversions would be simpler. For example, this code on page 620:

```
Register("TestDp",
  typeof(String),
  typeof(MainWindow),
  new PropertyMetadata("nothing",
    new PropertyChangedCallback(OnTestDpChanged),
    new CoerceValueCallback(CoerceTestDp)),
  new ValidateValueCallback(ValidateTestDp));
```

can be simplified to:

```
Register("TestDp",
  typeof(String),
  typeof(MainWindow),
  new PropertyMetadata("nothing", OnTestDpChanged, CoerceTestDp),
  ValidateTestDp);
```

Page 619, layout

For some reason there's a rectangle around the second paragraph of text on page 619. I don't think it's meant to be there.

Chapter 18

Page 637, unnecessary ToString() calls (personal preference)

This is a matter of taste, but personally when I'm already in the context of string concatenation, I don't bother with explicit ToString calls. (I'm aware of the boxing concerns for value types, but that's rarely significant in my experience.)
So this code (indented and wrapped as per the book):

```
 StringBuilder eventDetailsSB = new StringBuilder();
 eventDetailsSB.AppendLine("Event:    " + e.RoutedEvent.Name);
 eventDetailsSB.AppendLine("Sender:   " + sender.GetType().ToString());
 eventDetailsSB.AppendLine("Source:   " + e.Source.GetType().ToString());
 eventDetailsSB.AppendLine("Original: " + e.OriginalSource.GetType().
ToString());
```

would become:

```
StringBuilder eventDetailsSB = new StringBuilder();
eventDetailsSB.AppendLine("Event:    " + e.RoutedEvent.Name);
eventDetailsSB.AppendLine("Sender:   " + sender.GetType());
eventDetailsSB.AppendLine("Source:   " + e.Source.GetType());
eventDetailsSB.AppendLine("Original: " + e.OriginalSource.GetType());
```

Alternatively, I might use a more fluent approach:

```
StringBuilder eventDetailsSB = new StringBuilder()
    .AppendFormat("Event:    {0}", e.RoutedEvent.Name).AppendLine()
    .AppendFormat("Sender:   {0}", sender.GetType()).AppendLine()
    .AppendFormat("Source:   {0}", e.Source.GetType()).AppendLine()
    .AppendFormat("Original: {0}", e.OriginalSource.GetType()).AppendLine();
```

Page 641, typo (parameter name)

"... you can use the AddHandler() method and pass true as the HandledEventsToo parameter"
The name of the parameter is "handledEventsToo" with a lower-case "h" (as per .NET naming conventions).

Page 642, disagreement (diagnostics)

The author recommends using message boxes for diagnostics - Console.WriteLine is shown, but the author's preference is for message boxes:
"personally I find it [Console.WriteLine] no easier to call and a little cluttered to read."
That suggests that these diagnostics are always removed before shipping - which means they can't be used to trace problems encountered in the field. I certainly wouldn't want message boxes popping up all over the place as a user!

There's a much better solution than either of these: use a proper logging framework (e.g. log4net) which can be used both in development *and* in a production system, with the level of diagnostic output (and the destination) controlled at execution time via a configuration file rather than hard-coded.

Page 644, technical error (I believe)

"If you're using one of the .NET Framework argument classes, you can use the RoutedEventHandler as your event delegate, but if you declare a new argument type, you'll also need to declare a delegate."
What would go wrong if you just use the generic EventHandler<TEventArgs>? That's what it's there for... between that and the Func/Action delegate families, I rarely find myself declaring my own delegate types these days.

Page 645, typo (not invalid, just odd)

The method on page 645 is declared as

```
      protected OnMyCustomEvent ()
```

The space between OnMyCustomEvent and () should be removed.

Page 652, terminology

In a label referring to ApplicationCommands, EditingCommands, NavigationCommands, MediaCommands and ComponentCommands.
"The routed commands defined by the Framework are static classes."
I'd say they're *exposed* via static properties in static classes. The commands themselves aren't static classes - they're instances of RoutedUICommand and the like.

Page 661, incompleteness (exposing routed commands)

"While it's possible to implement a new command type, usually you'll create an instance of RoutedCommand or RoutedUICommand as a public static property."
This should be a public static *read-only* property. Making the property read/write would be a really bad idea.

Chapter 19

Page 676, confusing comment

After setting the A/R/G/B properties on a Color separately:
"Did you work out the cast to byte? You could have used FromARGB(), but the code would have been really long because of all those casts."
The casts are needed either way, so it's not like separating them really reduces the length of the code. If the properties *are* to be set separately, I'd personally use an object initializer instead.

Page 683, code formatting

The closing brace on the ShowGradient_Click method is mysteriously indented about 8 spaces.

Page 683, code clarity

The code on page 683 would be a lot clearer in my view with judicial use of the conditional operator.

Chapter 20

Page 715, typo

The third paragraph on page 715 starts "You could create a Class that declares [...]" - I suspect that "Class" should be "class" here. (This may be a VB-ism that crept in.)

Page 736, unfair C# vs XAML comparison

In an attempt to show how much clearer XAML can be than C# (which I'm sure is true in many cases), the author shows this C# code:

```
if (this.IsEnabled == true)
{
 if (this.IsSelected == true)
  this.FontWeight = FontWeights.Bold;
 else
  this.FontWeight = FontWeights.Normal;
}
else
 this.FontWeight = FontWeights.Light;
```

The comment that accompanies this includes:
"Nested if statements are very, very complex and easy to mess up."
So don't do that. A conditional works well here, either by inverting the "enabled" condition:

```
this.FontWeight = !IsEnabled ? FontWeights.Light
                : IsSelected ? FontWeights.Bold
                : FontWeights.Normal;
```

… or by explicitly stating both parts of the condition in each case:

```
this.FontWeight =
      (IsEnabled && IsSelected) ? FontWeights.Light
    : (IsEnabled && !IsSelected) ? FontWeights.Bold
    : FontWeights.Normal;
```

This involves redundant property checking, but is very explicit about the situations under which each font weight is used.

Another alternative is to create a method to determine the right font weight, and call that - it allows you to perform the determination separately from the assignment to the FontWeight property.

Once you're used to just reading down this sort of pattern, it becomes very easy to follow. Personally I would find *any* of these easier to get my head round than the XAML shown on page 736.

Page 747, layout issues (stretched diagram)

The diagram at the bottom of page 747 appears to be stretched horizontally, which looks odd.

Page 750, typo (Width should be Height)

The explanation of the exercise answer on page 750 includes: "If you don't explicitly set the Width property, it doesn't have a value [...]" - I believe this should be the Height property, given the rest of the explanation and the code.

Page 754, layout issues (swapped screenshots)

There are two screenshots on page 754, each of which is next to a bit of XAML - but I *think* the screenshots are the wrong way round.

Chapter 21

Page 779, layout issue (arrow tail overlays text)

The arrow in the top diagram of page 779 overlaps the text, so the "100%" of the label looks odd.

Chapter 22

Page 802, typo (unmatched parenthesis)

The middle paragraph on page 802 includes an open parenthesis which is never closed.

Page 803, typo (case of XmlDataProvider)

The text refers to an XMLDataProvider element, which I belive should be XmlDataProvider.

Page 806, typo (unmatched brace)

The second Binding shown on page 806 appears to be missing a closing brace:

```
{Binding Source={RelativeSource Mode={x:static RelativeSource.Self},
    Path=FontSize}
```

Page 813, confusion or bad practice

When finding out about building a bindable collection, we're first instructed to create a class which derives from ObservableCollection<T>. However, it's not clear *why* we supposedly need to create a new class rather than just using ObservableCollection<T> directly. No members of ObservableCollection<T> are being overridden in the code samples, and I generally prefer *not* to create classes just for the sake of it. I can't tell whether this is just pointless or whether there's some real reason for it.

(The sample showing the constructor of the new class also explicitly calls base() with a label of "You need to call the base class's constructor" - *implying* that you need to do so explicitly.)

Page 813, bad and/or invalid code

The example continues with this sample code:

```
public MyBindable items = new MyBindable();
MyBindable.Add(new MyType());
```

It's not clear:

- Where this code should appear
- Why we'd want a *public*, mutable field (ick)
- Why Add is being called on the class (MyBindable) rather than the field.

Page 813, invalid code

The example continues in the next snippet with:

```
this.DataContext MyBindable;
```

I suspect this should be an assignment, and probably using the field rather than the name of the collection type.

Page 831, longwinded code

The converter code on page 831 would be simpler using the conditional operator and implicit calls to ToString.

Page 831, code formatting

The XAML near the bottom contains a Window.Resources element whose closing tag is indented more than it should be.

Pages 834 and 835, bad code (inappropriate use of "as")

Both pages 834 and 835 contain code which uses the "as" operator and then unconditionally dereferences the result. Here's the code on page 234:

```
Ingredient ing = obj as Ingredient;
return (ing.OnHand > 50);
```

If you're so certain that the value *really, really* should be of the right type, a cast is more appropriate than using "as" - that way if you're wrong, you get an InvalidCastException instead of a less-informative NullReferenceException.