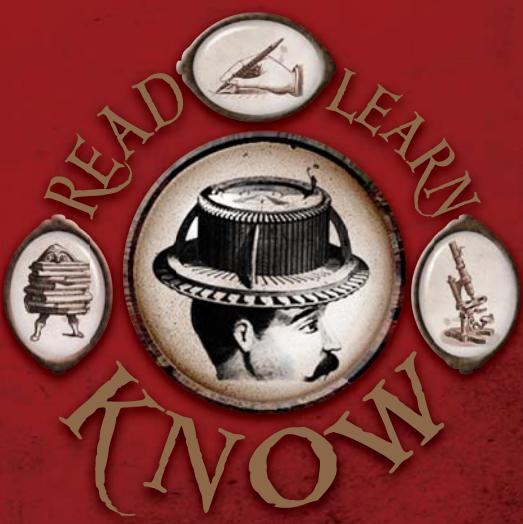


SAMS

FLUENT VISUAL BASIC®

REBECCA M. RIORDAN

www.it-ebooks.info



ASSOCIATE PUBLISHER
Greg Wiegand

SIGNING EDITOR
Neil Rowe

MANAGING EDITOR
Kristy Hart

PROJECT EDITOR
Andy Beaster

INDEXER
Cheryl Lenser

PROOFREADER
Karen Gill

TECHNICAL EDITOR
John Hardesty

PUBLISHING COORDINATOR
Cindy Teeters

COVER DESIGNER
Gary Adair

COMPOSITION
Rebecca Riordan

FLUENT VISUAL BASIC[©]
Copyright © 2011 by Rebecca Riordan

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 9780672335808

ISBN-10: 0672335808

Library of Congress Cataloging-in-Publication Data is on file.

Printed in the United States of America

First Printing November 2011

TRADEMARKS

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

The Windlass Lowercase and Brandywine fonts are copyrights of the Scriptorium foundry, www.fontcraft.com.

WARNING AND DISCLAIMER

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

BULK SALES

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearson.com

ACKNOWLEDGEMENTS

Yes, I know it says "Rebecca M. Riordan" on the cover, but that's not really true. Without the assistance of some amazing people, this book would never have made it out of my head, much less into your hands. So, in order of appearance, I would like to thank:

Neil Rowe, my editor, who took a chance on a very different way of writing computer tutorials. Without Neil's leap of faith, Fluent Learning would never have happened. My technical reviewers, David Sceppa, Eric Weinburger and John Hardesty, who collectively caught an embarrassing number of code typos and I-knew-what-I-meant obscurities. Finally, my copy editor, Karen Gill, who not only made sure the language in the book resembles English, but also expressed an unexpected and greatly appreciated enjoyment in the project. (Any remaining errors and infelicities are, of course, my responsibility.)

Jake von Slatt of [The Steampunk Workshop](http://steampunkworkshop.com) (steampunkworkshop.com), Samantha Wright (samantha-wright.deviantart.com) and Mindbloom (mindbloom.com) were all gracious enough to allow me to use their images. These are all seriously cool people, folks. I can't urge you strongly enough to go explore their sites.

GETTING STARTED

INTRODUCTION.....I

Fluent Learning Because
This book isn't for everyone
What you'll learn
What you'll need
How it works

APPLICATION DEVELOPMENT .. 9

The development process
System design
Creating executables

THE .NET PLATFORM43

.NET Components
Say hello
Say what?

THE VISUAL STUDIO UI..... 67

Solutions, projects and stuff
Take control
Get some help

TESTING & DEPLOYMENT93

Errors & exceptions
Deployment

Find out what this whole "being a programmer" thing is all about and how to use the tools you'll need to build applications.

Learn how to speak Visual Basic. It's a language, much like English, Spanish or Latin, only simpler.

THE LANGUAGE

PART I: NOUNSI2I

Statements
Declared elements
Comments
Directives & Attributes

PART 2: TRANSITIVE VERBS ...I55

Literal expressions
Object expressions

PART 3: INTRANSITIVE VERBS..I8I

Control of flow commands
Exception handling commands

Discover the secret to efficient programming: The best code is the code you don't have to write yourself.

THE .NET FRAMEWORK LIBRARY

CLASSES IN THE .NET FRAMEWORK .22I THE CLASS LIBRARY, PART I ...305

The Class Designer
Class definitions
Fields & properties
Methods

Namespaces
The Object Browser
Numeric data
Character data
Times & dates

OTHER FRAMEWORK TYPES.....269

Structures
Enumerations
Interfaces
Working with types

THE CLASS LIBRARY, PART 2 ...349
Arrays
Specialized Sets
Generics

CONTENTS

Put all you've learned to good use by learning how to use Microsoft's latest and greatest interface platform.

Stand on the shoulders of the experts by learning the best programming practices and how to implement them.

BEST PRACTICE

OOA & D	381
Type relationships	
OOP principles	
Type modifiers	

PROGRAMMING PRINCIPLES.....	425
The Single Responsibility Principle	
The Open/Closed Principle	
The Liskov Substitution Principle	
The Law of Demeter	

PATTERNS	457
The Strategy Pattern	
The Observer Pattern	
Architectural Patterns	

WPF	501
-----------	-----

Fundamentals	
WPF types	
XAML & Visual Basic	

WPF CONTROLS	531
--------------------	-----

WPF panels	
Control classes	
Content controls	
Items controls	

DEPENDENCY PROPERTIES ...	591
---------------------------	-----

The basics	
Creating dependency properties	

WPF INTERACTIONS.....	627
-----------------------	-----

Routed events	
WPF Commands	

WPF GRAPHICS	669
--------------------	-----

Color	
Brushes	
Pens	
Typography	
Effects	

RESOURCES	719
-----------------	-----

Resource dictionaries	
Styles	
Property triggers	
Event triggers	

TEMPLATES	765
-----------------	-----

Building controls	
Building control templates	
The VisualStateManager	

WPF BINDING	797
-------------------	-----

Creating bindings	
Binding to collections	
Working with collections	

TELL US WHAT YOU THINK!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As a Executive Editor for Sams, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of email I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and email address, phone, or fax number. I will carefully review your comments and share them with the author and editors who worked on this book.

Email: feedback@sampsprinting.com

Fax: 317-428-3310

Mail: Neil Rowe, Executive Editor
Sams Publishing
800 East 96th Street

Indianapolis, IN 46240 USA



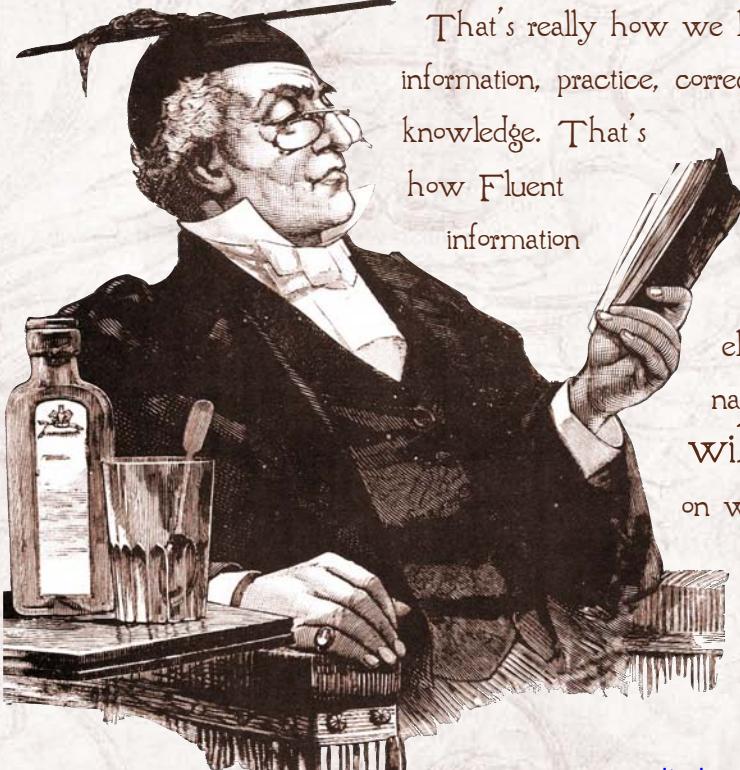
Welcome!

I'm so glad you stopped by.

This book doesn't look much like other technical tutorials, does it? Well, for once, looks aren't deceiving, because Fluent Learning books aren't much like other technical tutorials. **We don't want to teach you things.**

We want to help you learn things. We've done a lot of research into how people learn, and it turns out that talk, talk, talking at you (like most books do) isn't wrong, exactly, but it makes learning harder than it needs to be.

Did you learn to speak your native language by reading a book? Of course not; that's not how people learn. You heard people speaking, tried it for yourself, and then corrected yourself when other people pointed out your mistakes. Sure, you studied grammar and learned new words in school, but the basics ("More milk, Mommy") you learned by yourself. Now, barring accident and illness (or one-too-many mojitos), you're not likely to forget it, are you? And you don't have to think about the mechanics of speech, just what you want to say.

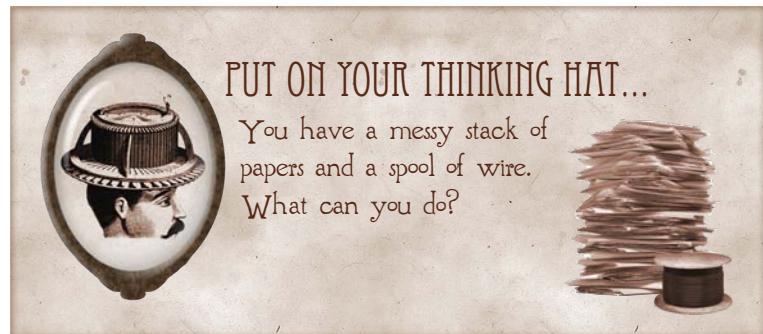


That's really how we learn everything. We gather some initial information, practice, correct our mistakes, and then add to our basic knowledge. That's not what happens in most tutorials, but that's how Fluent Learning works. I'll give you enough information to get started, give you some exercises to figure out how to do something, and then elaborate on what you've learned. Simple, natural, and if you do the work, you will learn. Soon you'll be able to concentrate on what you want to do, not how to do it, just like when you learned to speak. (But it won't take as long as learning to speak well.)

FLUENT LEARNING BECAUSE...

WE WANT TO LEARN, NOT LISTEN

Our minds like to learn anything the way we learned our native language: by trial & error. Instead of reading a lot of words, you'll do a lot of exercises. Real exercises, that make you think, not walkthroughs that tell you what to type. (But we'll have a few of those, too.)



PUT ON YOUR THINKING HAT...

You have a messy stack of papers and a spool of wire. What can you do?



WE WANT TO WORK, NOT PASS TESTS

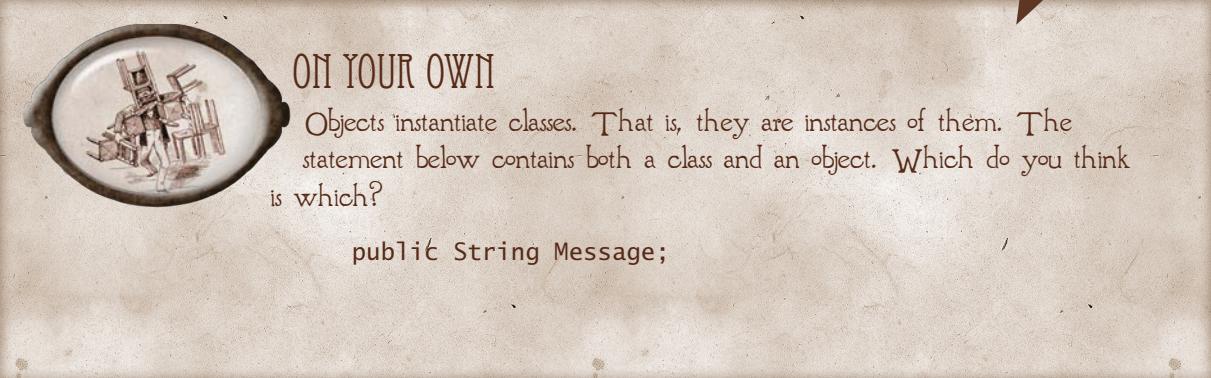
You want to be able to apply what you learn in the real world, not just pass a test on the subject. To help you do that, the On Your Own exercises invite you to make those connections as part of the learning process.



ON YOUR OWN

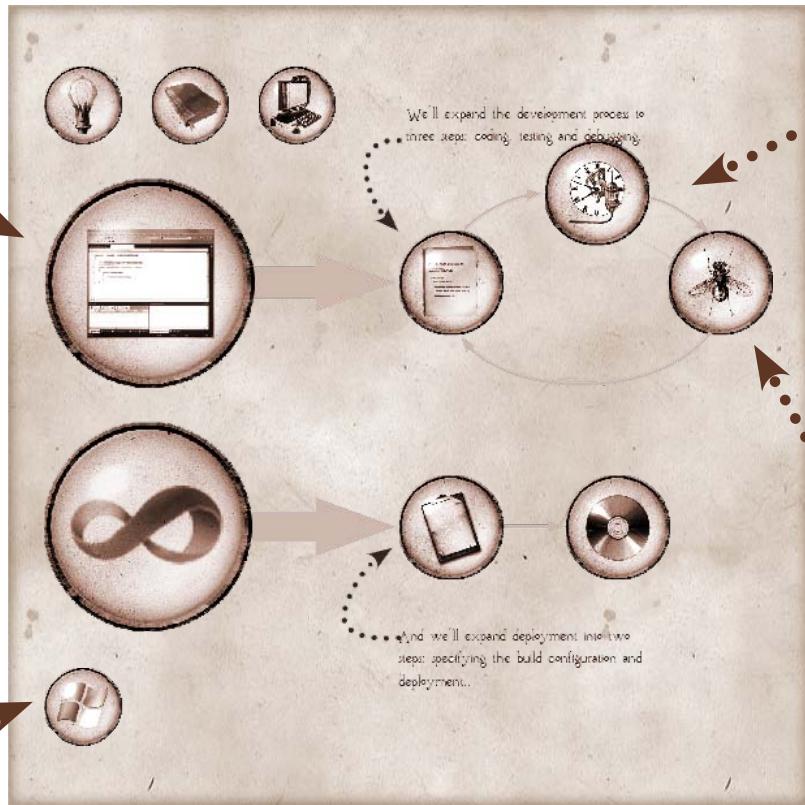
Objects instantiate classes. That is, they are instances of them. The statement below contains both a class and an object. Which do you think is which?

```
public String Message;
```



WE'RE ARTISTS, NOT FREIGHT TRAINS

Our minds don't chug along from point A to point B on a single track. Like any artist, we start with a sketch and then fill in the details. Rather than presenting all the information about a topic at one time, we'll start simply and add the details as you have more context. That way they'll stick.



WE DON'T LIVE IN A WORLD OF WORDS

Our minds absorb information through all our senses, not just speech. We'll use graphics, context and rhythmic language to appeal to sight, touch and sound.

WE WANT A MAP, NOT A MYSTERY

Our minds are constantly evaluating progress, but that's not possible unless we know where we're going. We'll use lots of signposts so you always know where you are and where you're headed.

THIS BOOK ISN'T FOR EVERYONE

I really hate it when technical books announce that they're for "everyone", don't you? It can't possibly be true, and saying that it is doesn't sell any more books in the long run. It just irritates people who get confused (because the book's too advanced for them) or bored (because the book's too basic). I don't want to irritate or bore you, so I'll say it plainly: This book isn't for everyone.

THIS BOOK IS FOR YOU IF:

- You don't know anything (or at least not much) about programming but want to learn.
- You've been using a computer for awhile but don't really know how applications get written.
- You've done a little playing around with Visual Studio and Visual Basic but don't really feel comfortable about your skills. It's scary not knowing what you don't know, and this book can help with that.

THIS BOOK PROBABLY ISN'T FOR YOU IF:

- You don't know how to use a computer at all. I assume that you know how to do things like opening applications, clicking buttons and choosing items from menus, so if you don't know that yet, you might want to start with a basic book on using Windows and come back to this one later.
- You know a version of VB, but haven't used it in awhile and want to brush up your skills. This book isn't well suited to browsing and skimming.

THIS BOOK MIGHT BE FOR YOU IF:

- You're a Visual Basic 6 programmer and want to learn Visual Basic .NET starting with the basics.
- You're a Visual Basic 6 programmer and want to learn to build WPF or Silverlight applications in Visual Basic .NET. There are lots of WPF/Silverlight tutorials around, and some of them are very good, but most of them assume you know C#. If you don't, this might be a good introductory book for you.
- You're a beginning Visual Basic programmer and want to learn patterns & best practices. There are books around that specialize on that subject, but they tend to be pretty advanced, so you might find this book more comfortable.

WHAT YOU'LL LEARN

You and I both know that you won't be an expert Visual Basic programmer after reading one book, no matter how good it might be. You won't be an expert after reading two books, or three, or a dozen. Of course you should read what the experts have to say. But the only way to build real expertise is to write a bunch of applications, make a bunch of mistakes, and fix them. It takes time and experience, and I can't give you that. But I can get you started.

AFTER YOU FINISH THIS BOOK YOU WILL:

- Understand the structure of the Visual Basic language, its basic syntax, expressions and commands.
- Understand the role that the .NET Framework plays in application development.
- Know the basics of the .NET Framework core types and how to use them.
- Have a working knowledge of designing object-oriented applications using design principles and patterns that will let you not just program, but program well.
- Have a working knowledge of the common design patterns used to develop modern applications.
- Have a working knowledge of the XAML declarative language.
- Understand how to write attractive, useful applications in Windows Presentation Foundation.
- Be ready to move on to more advanced books on specific programming topics.
- Be ready to start writing applications and figuring the rest out for yourself.

BUT YOU WON'T:

- Know everything there is to know about how the Visual Basic compiler or the CLR works. (I'm not sure anyone does.)
- Know every nook and cranny of the Framework Class Library. (But you will know how to explore it.)

WHAT YOU'LL NEED

You don't need much to get started. This book, of course. (You did buy this copy, didn't you?) And a copy of Visual Studio. You'll also need something to write with, because not all of the exercises are done at the computer, and you might want a notebook of some kind if you don't like writing in books.

GETTING VISUAL STUDIO

There are several different levels of Visual Studio. If you've already bought and installed a copy, you're good to go. If you haven't, you can use the free Visual Basic .NET Express edition. You can download it from the Microsoft Web site. Here's what you need to do on the day I'm writing this, but Microsoft has been known to move things around, so be forewarned: You may have to hunt around a bit.

1. From the main Microsoft Web site, choosing Visual Studio from the Products menu will take you to the Visual Studio home page.
2. Choosing Express from the list of products at the top of the page will take you to the Express versions page.
3. Choosing Visual Basic 2010 Express on the home page will take you to the product page. Choosing Download on the left side will take you to the Download page, where you can download an ISO image or install directly from the Web.

GETTING THE SOURCE CODE

You won't need to download the source code in order to do the exercises in this book, but it's available on the Web at informit.com/title/9780672335808.

HOW IT WORKS

If you've read this far, you already know that you're not going to be able to just sit back and listen to me talk at you. The core of this book is the exercises, and if you're going to learn, you'll need to work through them. Really do them. You can't just think about the answers. You need to sit down at the computer or pick up your pencil and do the work. Here's a taste of some of the things you'll be doing:



THE THINKING HAT

Most of the exercises in the book tell you to Put on Your Thinking Hat. I'll give you answers to these exercises, but it's really important to understand that you won't always get the answers completely right, and **that's okay**. It doesn't mean you're not "getting it". It means that I don't always play fair, because you learn as much (or more) from your mistakes.



ON YOUR OWN

Some exercises ask you to do things on your own, and I won't give you the answers. Sometimes there really aren't answers; they're just things you need to think about. It might be tempting to put these aside for "later", but it's best if you don't. They're part of the learning process.



TAKE A BREAK

From time to time I'll suggest that you take a break before you move on to the next section. Of course you can take breaks whenever you like, but these suggestions aren't because I think you might be getting tired. I make these suggestions because learning research tells us that if you stop for 15 minutes or so before you review, you're much more likely to transfer information to long-term memory. (In other words, you'll actually learn it.)



APPLICATION DEVELOPMENT

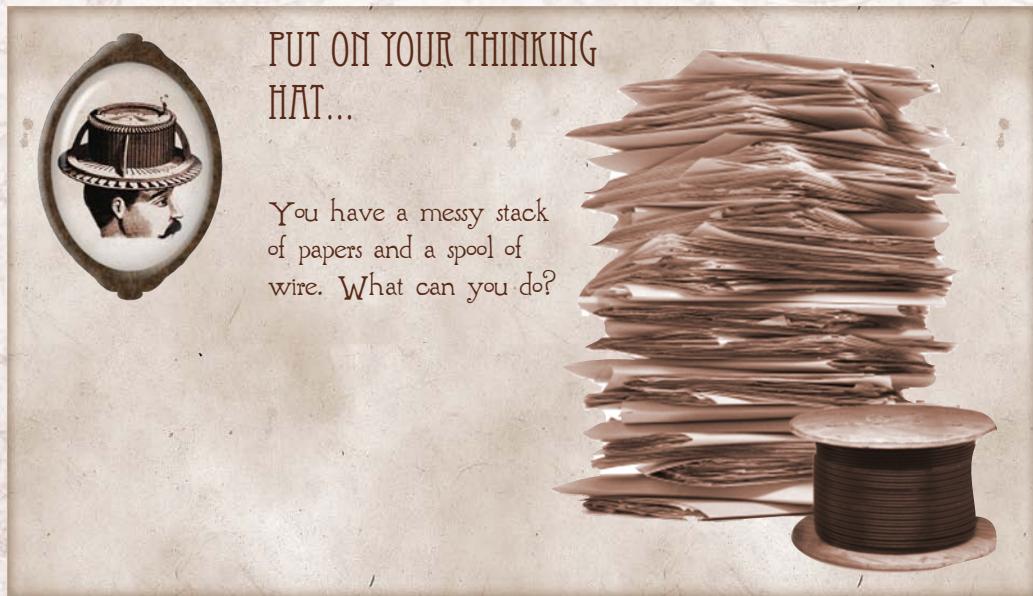
I

So, you want to learn to program. That's great. You're reading the right book. But what exactly is programming? Is it the same as software development?

And what about that other stuff you might have heard about...development platforms, application architectures, development methodologies, design patterns, best practices. It all starts to sound a little scary, doesn't it?

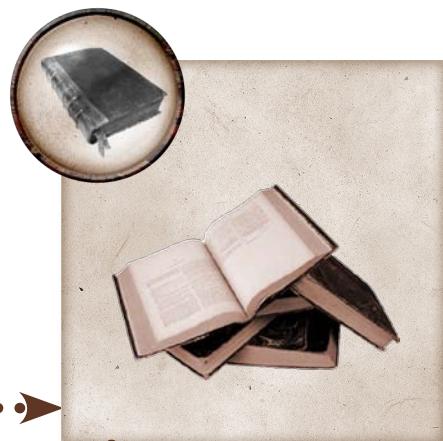
In this chapter, we'll figure out what all these things are, and where they all fit in the process of getting from here (an idea for some software) to there (a working application).

Relax. It's not as complicated as you think...



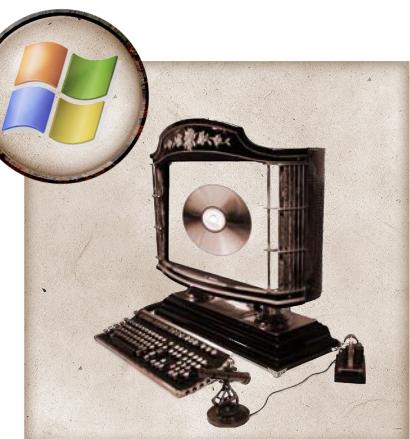


A project begins when someone has an IDEA.



The idea gets translated into a SPECIFICATION.

1 The SPECIFICATION tells the programmer what the system needs to do.

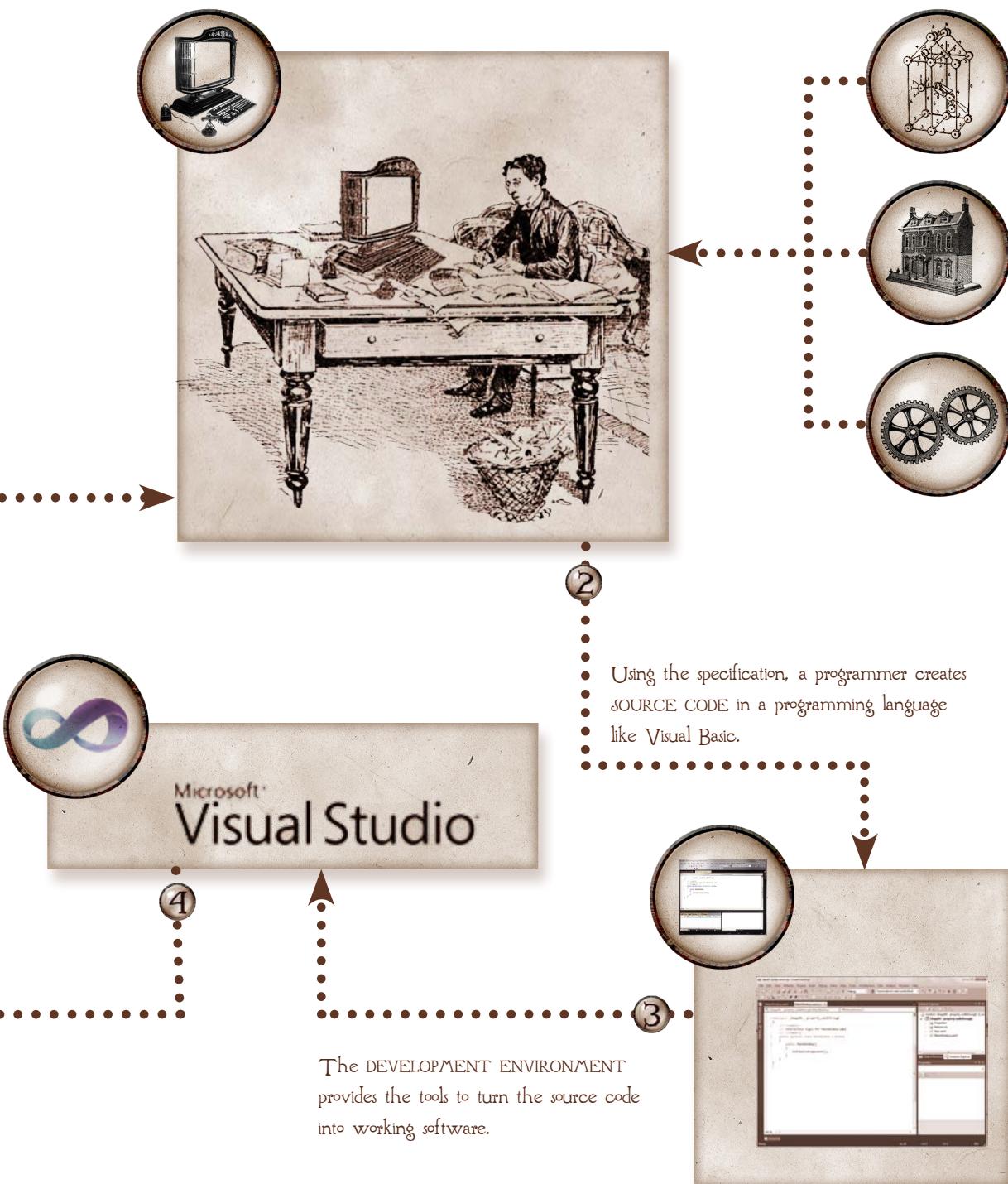


When people use your software, they'll often come up with new ideas...and the project starts all over again!

5

The software executes in a specific RUNTIME ENVIRONMENT, like Windows or a browser.







HOW'D YOU DO?

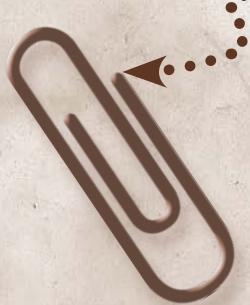
I gave you a messy stack of paper and spool of wire. Thinking Cap exercises don't always have a single right answer, but here are two things you might have done...



You might have bound the papers into a notebook. That's a good permanent solution, but a lot of work.



Or you might have simply made a paperclip. Easy and fast, but probably temporary.

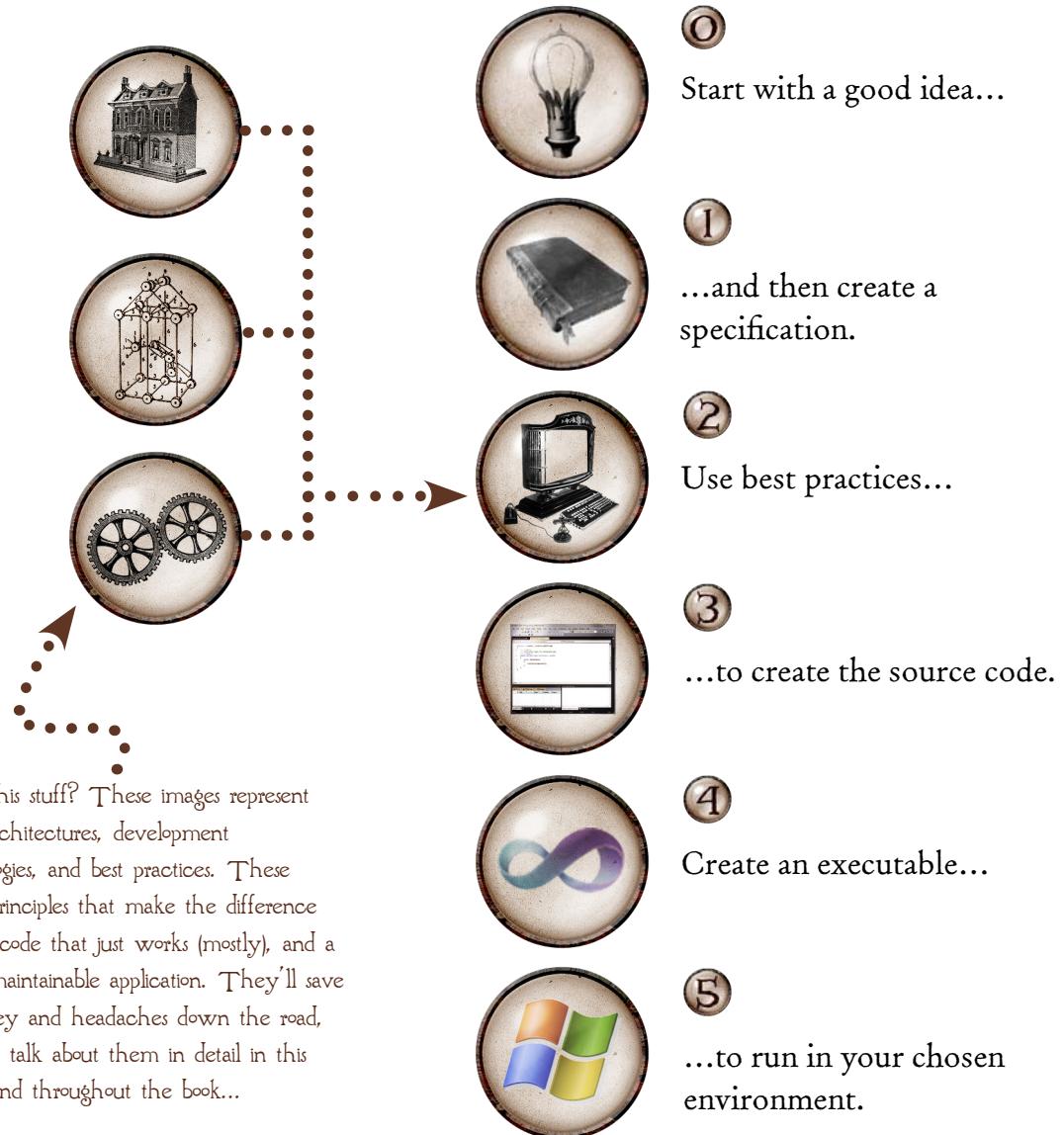


So what's the point here? That in application development the right answer is almost always "**it depends**". The best solution will depend on how the application is going to be used. It will depend on how much time and effort are available (or you're willing to invest). It will depend on what else is going on in the application.

The best solution depends on...well, it depends.

LET'S REVIEW THAT PROCESS...

Start with a good idea (that's the hard part!), and you're just five steps away from a solution. In this book we'll mostly be looking at how you write source code, but don't worry, we'll look at the rest of the steps too at least enough to get you started on the right path.



MEET NEIL & GORDON...

Before we get started, I'd like you to meet the clients we'll be working with. They run the hottest new cafe in town. But they're drowning in their own success. There's so much to manage! Schedules, menus, meetings, and now that their business is growing, they need to get all those yummy recipes out of their heads and into a useable form. The boys are geniuses in the kitchen, but they don't know where to start with this management stuff.

We'll be helping them out throughout the book.





TASK LIST

In the rest of this book we'll go into detail about step 3 of the development process: writing code, and writing it well. But there are three things we need to examine first:



THE DEVELOPMENT PROCESS

Everything we do fits into the context of the process of a whole, so we need to understand that first.



SYSTEM DESIGN

We know that the development process begins with an idea that gets translated into a specification that describes what the system is to do. We'll start, then, by looking at a few tools for capturing the system design.



CREATING EXECUTABLES

The integrated development environment (IDE) provides the tools that translate source code into an executable, a piece of software that a user or another piece of software can execute. We'll be using Visual Studio to create executables, and before we can make decisions about how to structure an application, we need a little more detail about how that works in the context of the .NET Framework.



ON YOUR OWN

Do you have some ideas about applications you'd like to write? Make a note of three or four of them. We'll come back to them again and again.

THE DEVELOPMENT PROCESS



The application development process begins when somebody needs a computer to do something. That “something” might be a change to an existing piece of software or a completely new system, but most applications are (or should be) developed in response to a need.

The “something” that the application is to do is captured in a **SPECIFICATION**. A specification can be as simple as a diagram scribbled on a napkin or many volumes of detailed description. It might be “complete” before any code gets written, or it might be developed over the course of the project.



You’ll use the specification to develop the application. In doing so, you’ll choose an effective **DEVELOPMENT METHODOLOGY** (the set of steps you’ll perform during development) and be guided by best **PRACTICES** (rules of thumb that experience has shown result in more efficient and maintainable code). You may also implement design **PATTERNS**, which are standard solutions to common programming problems.



...ONE MORE TIME

“Programming” is largely (but not only) a matter of writing SOURCE CODE. That means, for us, writing Visual Basic STATEMENTS and using the functionality provided by the .NET Framework. As you’ll see, the .NET Framework does a lot of the work for us.



Most programmers use an INTEGRATED DEVELOPMENT ENVIRONMENT (IDE) that combines a text editor (the bit that lets you type stuff), a compiler (the bit that translates your source code into object code, instructions that can be understood by the computer), a linker (the bit that combines bits of object code into an executable), and other useful tools like testers and visual designers. Like most Visual Basic programmers, we’ll be using Microsoft Visual Studio, and we’ll look at all these functions throughout this book.

Once the application is written and compiled into an executable, it operates in a specific RUNTIME ENVIRONMENT. In this book we’ll be developing applications that run under Microsoft Windows, but you should know that Visual Basic and Visual Studio can be used to write for several different platforms, including the Web and mobile devices.



TAKE A BREAK

That’s the first task done! Now would be a good time to take a short break and relax a bit before you come back to the rest of the chapter.



REVIEW

Before we move on, let's do one last exercise to make sure you've got the process down pat. The icons below are out of order. Number them correctly, and then describe each step.



⑥ _____

① _____

② _____

③ _____

④ _____

⑤ _____

⑥ _____



SYSTEM DESIGN

What's the difference between the sound my cat makes walking on the piano and the sounds made by a concert pianist? A plan.

When you're programming, the plan you work with is a specification. (You knew that, right?)

Here are some things you need to know about specifications:

- There's no single right way to prepare one.
- The detail required is determined by the complexity of the application and your experience. You'll need to go into more detail when the subject or the functionality is new to you.
- Best practice dictates that the detailed specification be developed during the course of the project, but that isn't always possible. Sometimes your client (or your boss) wants a complete specification before you start coding. Be prepared; either way requirements will change during the course of the project.
- Most nontrivial specifications will include a lot more than the use cases we'll discuss here: database schemas, screen layouts, architectural and processing diagrams...the details will depend on the application. (Remember: it depends.) But this isn't an application design book, so we're just going to talk about a few design tools that will get you started. (We'll look at a few others in later chapters.)

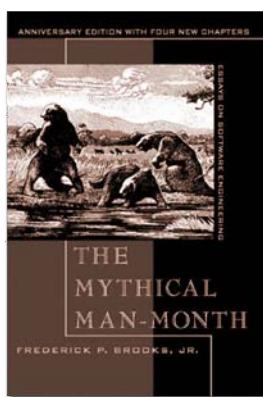




AGILE WATERFALLS

In the early days of software engineering, we tried to develop complete, detailed specifications before we started programming. This methodology was called the **WATERFALL MODEL** because each clearly defined stage of development (requirements, design, development, testing, deployment, maintenance) “flowed” into the next.

Finishing each stage before moving on to the next seems like a good idea, right? You’d think it would be efficient to know exactly what you need to do before you start to do it. The problem is, we never know exactly what a system needs to do, or how it needs to do it, until we’re actually using it.



The waterfall model frustrates users because it’s slow and inflexible, and it frustrates developers because the users “never know what they want”.

Frederick P. Brooks, in his 1975 classic the *Mythical Man-Month*, said, “Plan to throw one away. You will anyway.”

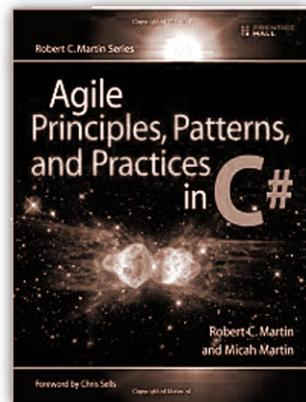
Pretty depressing idea, isn’t it?

The waterfall model does have its place, because in principle it’s still an efficient development model. Sometimes the system you’re working on is simple enough that you can know exactly what it needs to do in advance. Changes to existing systems are often simple enough to use

the waterfall model, even though the system itself may be extremely complex. (But be careful here...complex systems can be difficult to predict.) And, unfortunately, sometimes you’re working for a large organization that put its development methodology in place back in the Cretaceous period (about 1980), and they’ll dictate your process.

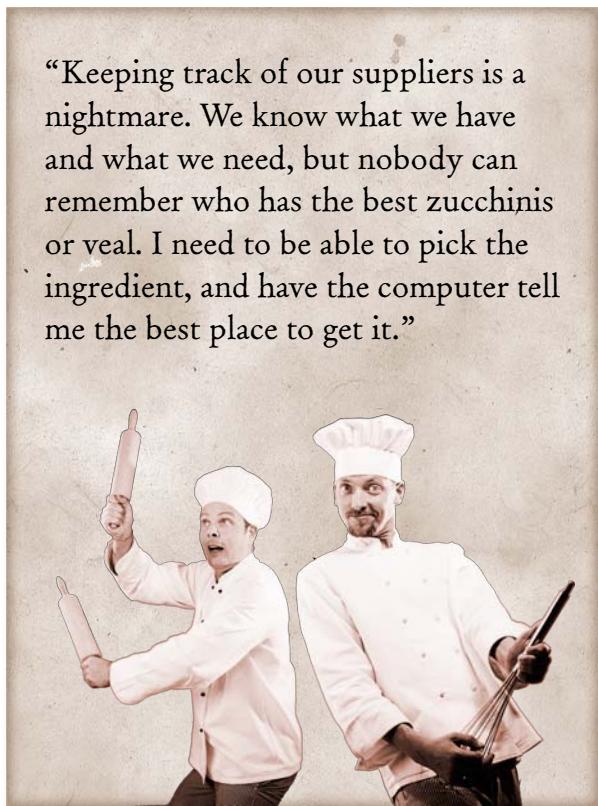
But we’ve learned a lot since then, and smart software engineers have developed new, more effective techniques. One of the best of these techniques is part of a methodology called **AGILE DEVELOPMENT** which, among other things, recommends that you only specify the details of the little piece of the software you’re going to work on right this minute. There’s a lot more to Agile Development, some of which we’ll talk about in later chapters. But if you’re keen (and I hope you are!) you can read the original Agile Manifesto, published in 2001, at www.agilemanifesto.org, or check out Robert C. Martin’s excellent introduction.

Whether you’re developing the specification all at once or a little at a time, though, you need a place to start. A **USE CASE**, which is simply a description of how an actor (either a user or another software component) interacts with the system, is one of the best.



USE CASES

USE CASES are just a step-by-step description of how a piece of software interacts with people or other software.



“Keeping track of our suppliers is a nightmare. We know what we have and what we need, but nobody can remember who has the best zucchinis or veal. I need to be able to pick the ingredient, and have the computer tell me the best place to get it.”

Let’s walk through putting together a simple use case. Based on Neil’s description of the problem, we can extrapolate an interaction that looks like this:

- ① System displays list of products, alphabetically by type (vegetable, meat, herb, etc.).
- ② User selects item from list; system displays top three suppliers, ordered by preference. For each supplier, display name, phone number, and delivery time.

And that’s it. It may not seem like it, but that really is enough to get started. (You want to make sure you only do enough to get started. It’s easy to over-complicate these things, and that’s a mistake.)

Notice, though, that this use case implies a few others—if the system is displaying a list of ingredients, that list has to be stored somewhere, and there has to be some way to enter it. Same with suppliers. But those are separate use cases. Make a note that they exist, but don’t let them distract you!

WHAT IF...?

Our use case describes an ideal interaction between the user and the system. That's great, but we all know that the world hardly ever runs according to plan, and a resilient system can't ignore that. So in addition to the ideal case, you'll need to capture what could go wrong (as best you can) and what the system should do when it does. Think of them as Murphy's Law cases...

In our simple example, the ingredient the user is looking for might not be on the list. Or there might be a system problem. For example, the database containing the ingredients and suppliers might not be available at runtime. What should the system do when these things happen?

Alternate paths through the system aren't always the result of problems, of course. Sometimes the user just needs to do something a little differently. Whatever their source, these alternate paths should be captured as part of the use case. Our revised version looks like this:

IDEAL CASE

- ① System displays list of products, alphabetically by type (vegetable, meat, herb, etc.).
- ② User selects item from list; system displays top three suppliers, ordered by preference. For each supplier, display name, phone number, and delivery time.

INGREDIENT NOT FOUND

System transfers user to ingredient maintenance screen.

SYSTEM ERROR

System displays descriptive message.



PUT ON YOUR THINKING HAT

Time to try writing some use cases on your own. Based on the descriptions below, try writing out the basic steps and one or two alternate paths for each case.

“Our recipes are based on a number of servings that’s right for the restaurant, but for catering, we often need to scale the recipes up or down. That math is really ugly, and it’s easy to make mistakes.”

“The catering side of the restaurant is starting to really take off. I keep the client’s contact details in my agenda, so if Gordon needs to contact them directly, he has to ask me for them. That’s a waste of time.”

Extra credit: Each of these use cases implies the existence of others. Can you identify them?



HOW'D YOU DO?

These cases are based on imaginary conversations. Your imaginary conversation might be different, so don't worry if you have slightly different results. The important thing is that you get some practice thinking through the steps.

"Our recipes are based on a number of servings that's right for the restaurant, but for catering, we often need to scale the recipes up or down. That math is really ugly, and it's easy to make mistakes."

Ideal Case:

1. System displays list of recipes, alphabetically by type.
2. User selects recipe from list. System displays formula.
3. User enters new batch size and chooses "Recalculate". System recalculates ingredient amounts and displays new values.

Recipe not found:

No action.



Even though the system takes no action, it's a good idea to think about what might go wrong.

System error:

System displays descriptive message.

"The catering side of the restaurant is starting to really take off. I keep the clients' contact details in my agenda, so if Gordon needs to contact them directly, he has to ask me for them. That's a waste of time."

Ideal Case:

1. System displays list of clients, alphabetically by type.
2. User selects client from list. System displays contact details.

Client not in list:

No action.

System error:

System displays descriptive message.

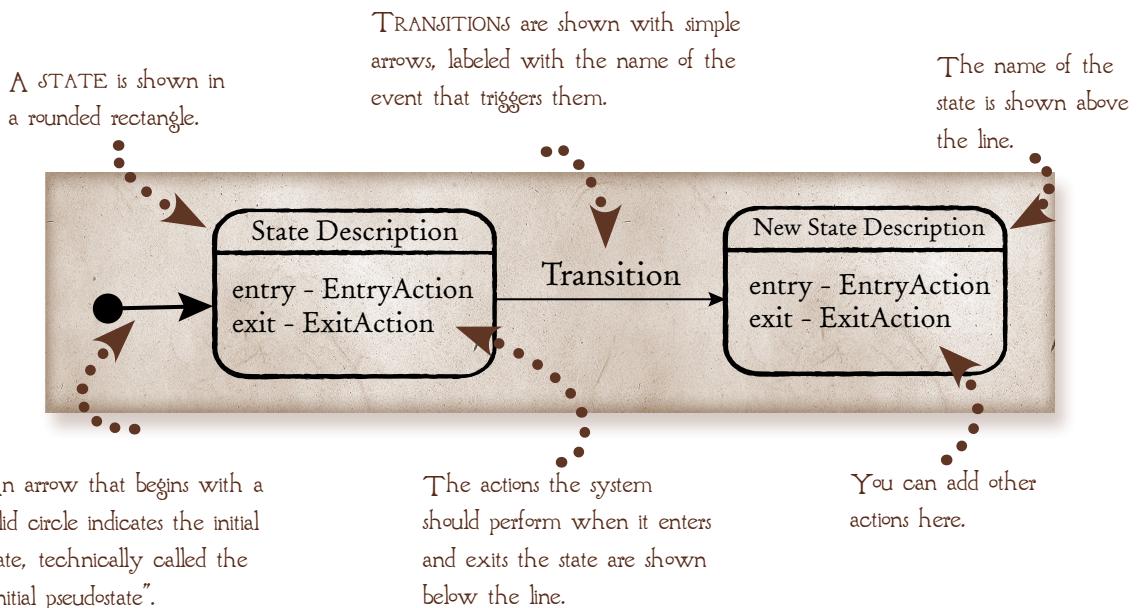
Extra Credit:

The client details system implies a case to enter and edit client details, while the portioning system implies a case to enter ingredients and one to enter recipes. Both systems probably require a printing case, as well.

Did you think of any others?

UML STATE DIAGRAMS

UML stands for the "Unified Modeling Language". It's a de facto industry standard for MODELING (drawing pictures of) a system. There are several different types of UML diagrams, and we'll look at a few more of them in later chapters. One type, called the UML STATE DIAGRAM, is useful for supplementing and clarifying complex use cases. (UML also defines a Use Case diagram, but frankly, I don't think it's much help.) Not surprisingly, a UML state diagram describes the behavior as a system as a series of STATES, the TRANSITIONS between them, and the EVENTS that trigger transitions. Here's an example:



WHY BOTHER?

Is it really worthwhile learning a strict modeling language? Well, I'm not going to make that decision for you, but UML does have some advantages:

You don't have to make up (and remember) your own symbols.

Lots of people can read and understand UML diagrams. That makes them easy to

use in teams and will help you understand books (like this one) that use UML diagrams.

There are tools that can help you manage UML diagrams. As we'll see, Visual Studio uses a kinda-sorta UML for generating source code in some of its designers. There are other programs and languages out there that do the same thing.

AN EXAMPLE

The abstract diagram on the previous page probably doesn't look all that helpful, but let's look at a real example. Here's the supplier use case in both formats...

IDEAL CASE

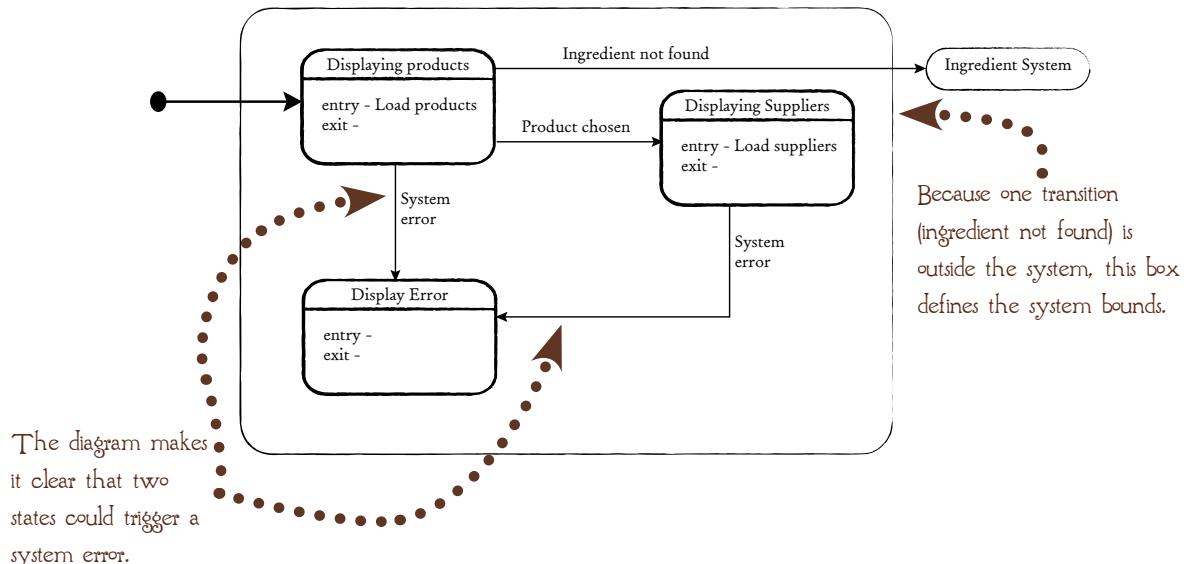
- ① System displays list of products, alphabetically by type (vegetable, meat, herb, etc.).
- ② User selects item from list; system displays top three suppliers, ordered by preference. For each supplier, display name, phone number, and delivery days.

INGREDIENT NOT IN LIST

System transfers user to ingredient maintenance screen.

SYSTEM ERROR

System displays descriptive message.



What do you think? Is the diagram clearer? Which would you use for a simple case like this? Either, both?



PUT ON YOUR THINKING HAT

The use cases we've been working with were very simple. Here's one that looks a bit more complicated. Prepare a use case and a state diagram based on Gordon's description.

"Our desserts use basic components—cake layers, fillings like chocolate ganache and pastry cream, and sometimes frostings—that are made in advance and then combined on the day they're served in the cafe. At the end of each day I prepare the dessert menu for the next day, and to do that I need to know what components and raw ingredients are in the cooler. Because some things may be needed for a catering job, the system needs to show me how many are available in total and how many are already allocated. Then, if we're getting low on something, I need to add it to the schedule to be prepared sometime in the next few days."



HOW'D YOU DO?

It's fine if your answers don't match mine exactly, as long as you thought through the issues.

Ideal Case:

1. System displays components and raw ingredients in cooler.
2. User selects ingredient from list. System displays details of quantity on hand and quantity available.
3. User allocates amount required. System updates details on exit.

Schedule:

System transfers to Scheduling component.

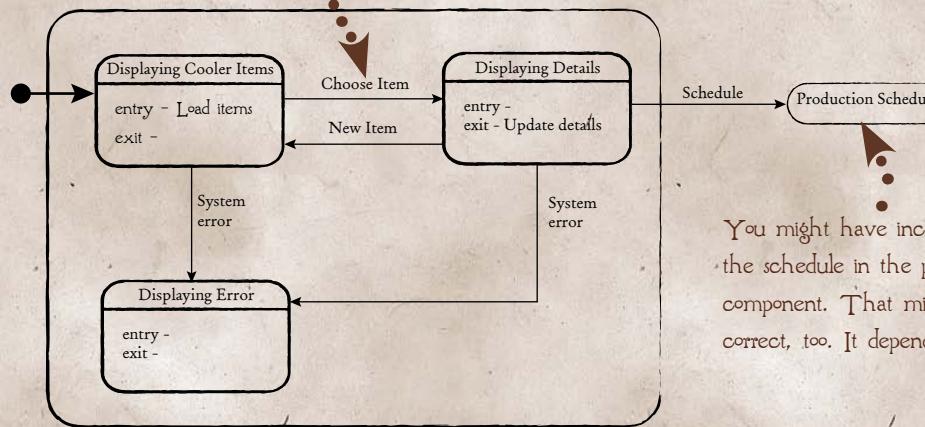
System error:

System displays descriptive message.



The use case shows some details, like the exact information displayed, that aren't shown on the diagram.

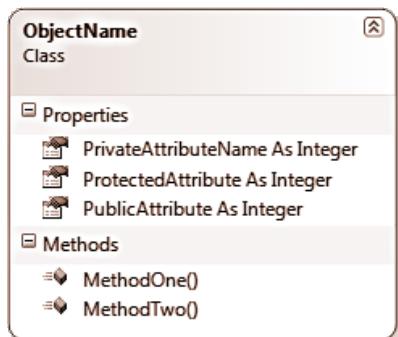
The diagram makes it clear
how the user can move
around the system.



You might have included
the schedule in the planning
component. That might be
correct, too. It depends.

SPECIFICATION COMPONENTS

You've seen two of the things that specifications commonly contain: use cases and UML state diagrams. Are you wondering what else might be in there? Here are some possibilities...

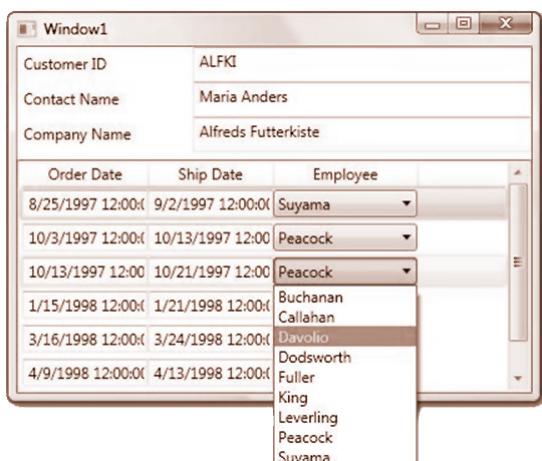
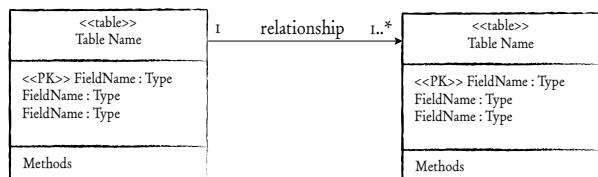


CLASS DIAGRAMS

A UML class diagram models the classes in the system. A **CLASS** is a programming construct that represents the characteristics of something. It has attributes (the things it knows) and methods (the things it can do). We'll talk about classes a lot beginning in Chapter 9.

DATABASE SCHEMAS

If the application is going to store data in a relational database, the specification will often include a schema diagram that describes the database structure. We won't be discussing them in this book.



SCREEN LAYOUTS

Some designers include screen layouts as part of the specification. (I'm one of them.) The layouts might be just rough sketches, or they could be complete graphics, depending on how complex the interface is and how important the "look and feel" are to the client.

USER REQUIREMENTS



The specification process is part of the overall development methodology. We took a quick look at two different methodologies. Can you describe how the specification process differs between them?

WATERFALL:

AGILE:

Use cases translate a user requirement into the steps that represent its functionality and the way actors interact with the system. What three things should you record when creating a use case? (Hint: One of them you only make a note of...)

- ① _____
- ② _____
- ③ _____

Use cases and UML state diagrams are good for showing different things. Name at least one thing that's easier to show on each:

...ONE MORE TIME

UML state diagrams show interactions in terms of system states, the transitions between them, and the events that trigger the transitions. Draw a basic UML state diagram below, showing the initial pseudostate, at least two states and a transition.



TAKE A BREAK

Task two complete! Now why don't you take a short break and relax a bit before you come back to complete the Review and tackle the final task: understanding the compilation process?



REVIEW

Before we move on, let's do one last analysis. Pick one of the applications you want to develop from the list you put together on page 13 and build the use case and a UML state diagram.



CREATING EXECUTABLES

When an orchestra plays, the musicians rely on machines to capture the sound and create something that can be re-created on a CD or a media player. Programmers do the same thing.

Computers don't speak English. They don't even understand Visual Basic, which is, as we'll see, very like English.

Computers understand moving bits around (bits as in zero or one, not bits as in pieces). They add bits together and perform logical operations on them. Most people don't want to translate "my recipe" into operations like that, so we use a language like Visual Basic to describe the way we want our applications to behave in a way that closely resembles the way we think about them.



MAKE A NOTE

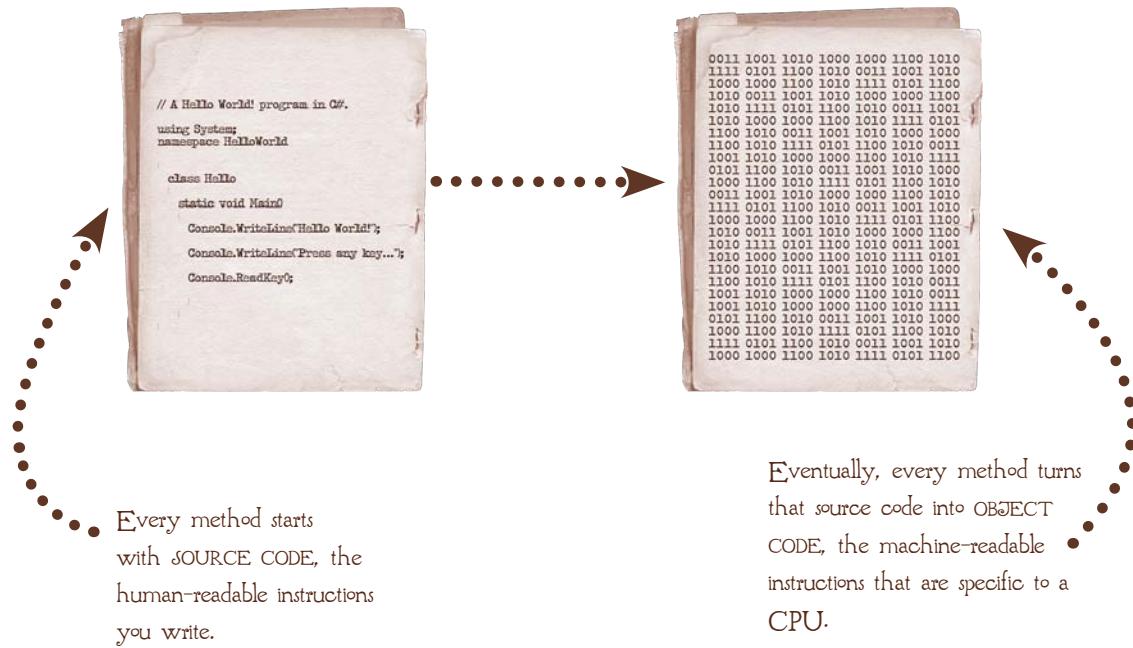
Here are some things you should know about creating executables:

- There are three primary ways to translate from source code to object code that can be understood by the computer. We'll be looking at each one of them in this section.
- Every method has benefits, and every method has problems.
- The translation method is usually determined by the language tools you use.

COMPILED OR INTERPRETED?

Traditionally, languages are either compiled or interpreted. Both methods achieve the same result—the source code you write in a language like Visual Basic is translated into object code that can be understood by a specific CPU. The difference is when.

Compiled languages perform the translation at design-time, while interpreted languages translate at runtime.



WORDS FOR THE WISE

DESIGN-TIME is when you're building the application.
RUNTIME is when the user is executing the application.



PROS AND CONS

There isn't a clear-cut answer to "which compilation method is better?"

COMPILED LANGUAGES...

Tend to run faster than interpreted languages.

Keep the executable file distinct from the source code, which makes it somewhat more secure.

BUT...

Testing the application during development requires constant re-compilation.

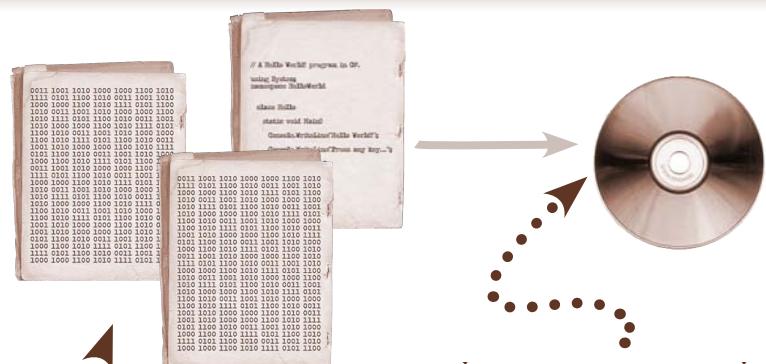
The output is always specific to a single platform.

INTERPRETED LANGUAGES...

- Make testing and changing the application while you're developing it fast and simple.
- Allow the same source code to be run on any platform that has an interpreter for the language.

BUT...

- They tend to run more slowly than compiled languages.
- The source code is available to everybody.

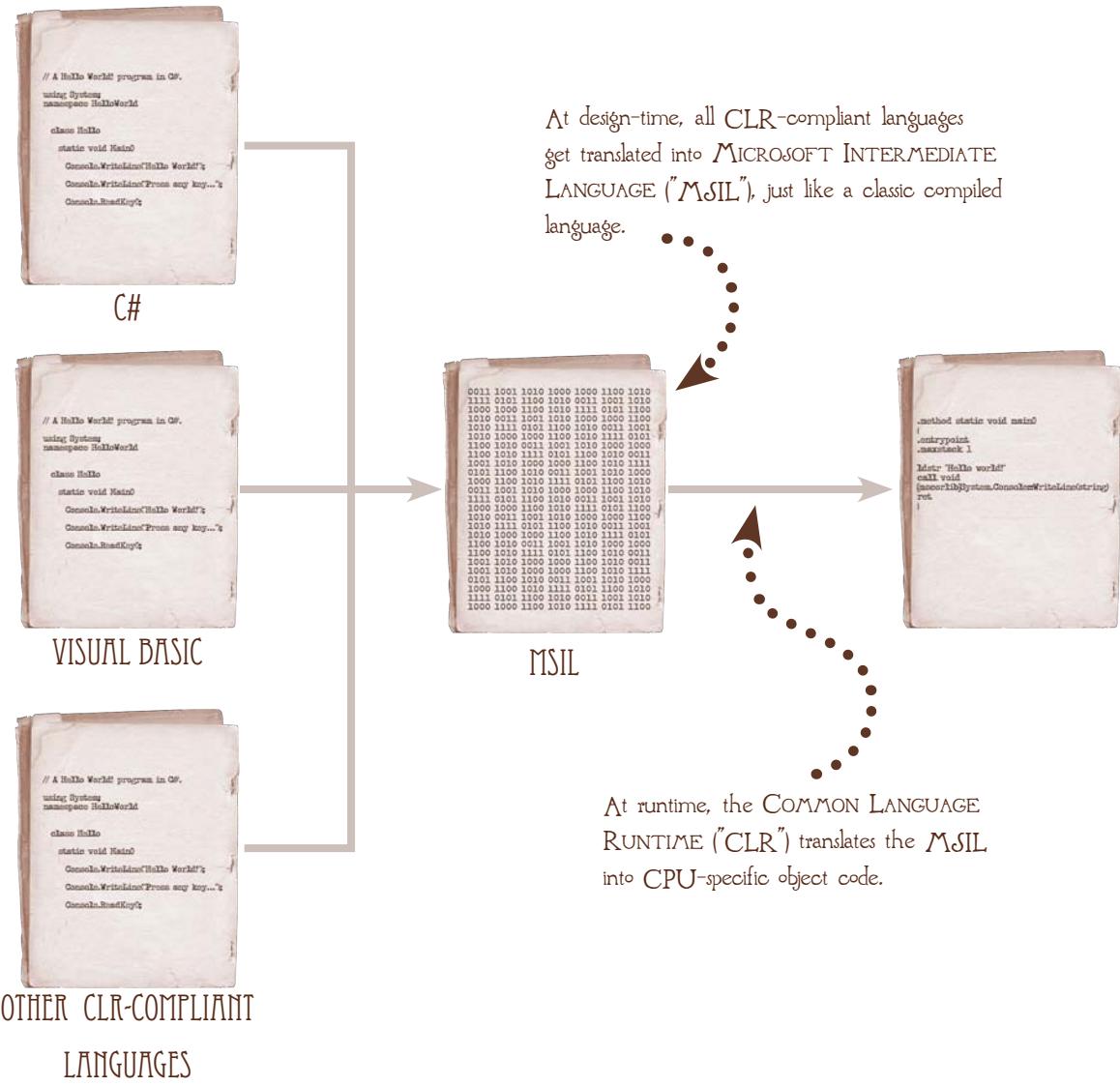


Multiple source files can be combined into a single executable. This is called LINKING.

In Windows, an EXECUTABLE can be an exe or a dll. Basically, an exe (pronounced "x-ee") is an application that a user can run, while a dll is a library ("dll" stands for dynamic link library) that is run by other pieces of software.

JUST-IN-TIME...

Visual Basic is a CLR-compliant language. We'll find out what that really means in the next chapter, but for now you need to know that applications that you write in Visual Basic use a different method of translating source code to object code called JUST-IN-TIME COMPIRATION ("JIT", pronounced like "bit", and sometimes used as a verb, as in, "Code in the CLR is jitted".)





PUT ON YOUR THINKING HAT

You've seen the pros and cons of interpreted and compiled languages. JIT compilation combines the two techniques. What do you think some of the pros and cons of this approach are?

PROS:

CONS:



PROS & CONS

It's important to understand some of the trade-offs in using JIT compilation, so let's look at some of those pros and cons in a little more detail...

THE CLR IS PLATFORM-AGNOSTIC

Microsoft's version of the CLR runs on various versions of Windows operating systems, including embedded and micro systems. The CLR has also been ported to run on Linux, Mac OS X, BSD and Sun Solaris, among other platforms. (For details, check out www.mono-project.com.) Parts of the CLR have been accepted as ECMA standards.

APPLICATION COMPONENTS CAN BE WRITTEN IN MULTIPLE LANGUAGES

Because all CLR-compliant languages are compiled to the same intermediate language, an application can use components written in any combination of them. For the first time, you can use those fast statistical routines written in C++ and the widget your buddy wrote in C# in your Visual Basic code. The CLR goes a long way toward achieving Microsoft's goal of making language "a lifestyle choice".

RUNTIME COMPILATION CAN BE SLOWER

The industry rule-of-thumb is that interpreted languages run 30 to 70 times slower than compiled languages. Although Microsoft provides a tool, NGen.exe (for "native generator"), that will perform the final compilation at design-time, tests have indicated that applications compiled to native object code don't actually run much faster, and, surprisingly, in some instances are actually slower. It appears that the CLR is really smart about adapting code to the runtime environment.

CLR-COMPLIANCE CONSTRAINS LANGUAGES

Some people have complained that since all CLR-compliant languages are compiled to MSIL, none of them can do anything that MSIL can't do, a kind of "lowest common denominator" programming. In reality, there isn't much the computer can do that MSIL can't do, and you always have the option of writing non-compliant code with C#, C++, or some other language, and calling those "unsafe" procedure from within a CLR-hosted application.



HOW'D YOU DO?

Here are some of the pros and cons of the JIT compilation method. You may have thought of others, and that's fine.

PROS:

- MSIL will run on any platform that supports the CLR.
- An application can use components written in any combination of CLR-compliant languages.
- Error-checking can be done at design-time.

CONS:

- Runtime compilation could be slower.
- The CLR places some constraints on languages.
- Compiling makes testing a little slower.



TAKE A BREAK

That's the final task done! Now take a short break and relax a bit before you come back to complete the final exercise and finish off the chapter.



REVIEW

Time for one last review before we move on to the .NET Platform in the next chapter.

What are the three methods of translating source code to object code?

What factors determine how much detail is required in a specification?

Why does the waterfall method seem like such a good idea? Why isn't it?

At what point in the process is source code interpreted?

At what point in the process is source code compiled?

What does JIT stand for?

What do each of these images stand for?



Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



THE .NET PLATFORM

2

In the last chapter we looked at how the process of application development works and examined the specification process and compilation methods in a little more detail. In this chapter, we'll start narrowing our focus by looking at the .NET Platform, the environment in which most Visual Basic applications run.

The .NET Platform (just "dot-Net" to its friends) includes the .NET FRAMEWORK, the programming and runtime infrastructure we'll be studying, as well as servers like SQL Server that manages data, and tools like Visual Studio and Expression Studio.

We'll also get our feet wet in this chapter by writing our first real, live application, the traditional "Hello, World!" that's a rite of passage for all beginning programmers.

ON YOUR OWN

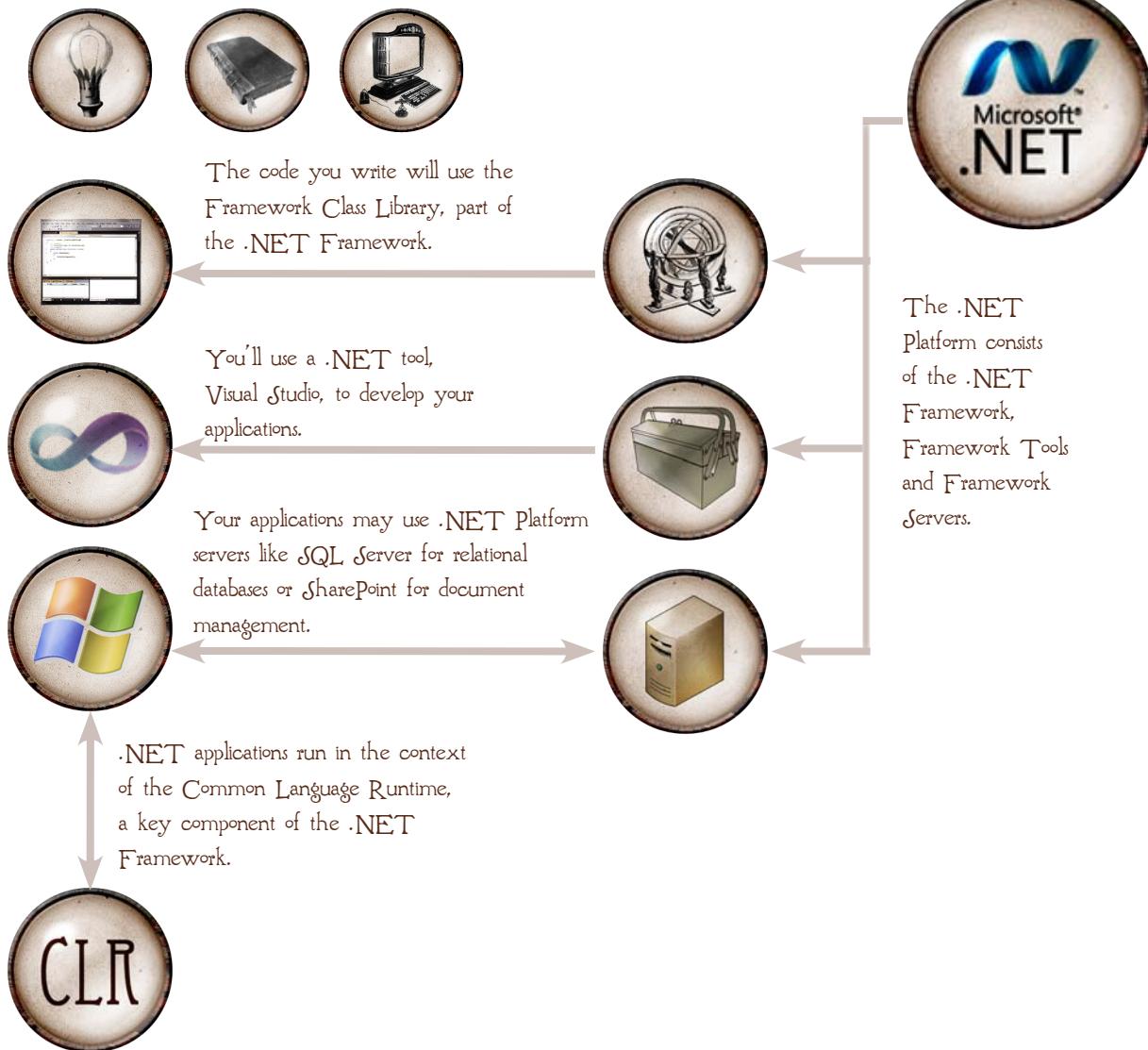


Remember that list of applications you want to develop? Most of them are going to use the .NET Framework, but as you go through the chapter, make some notes of other .NET Platform components that you think might be useful to you.



FITTING IT IN

You'll use .NET Platform components at both design-time and runtime.





TASK LIST

In the last chapter we looked at development at an abstract level. Now it's time to start looking at the specifics of the .NET Framework and Visual Basic .NET. Here's how we'll start:



.NET PLATFORM COMPONENTS

As talented as they are, Neil and Gordon can't do *everything* in the kitchen. They don't have to. And you don't have to do *everything* that has to be done in an application, either. But just like the boys, if you're going to control everything, you need to have a basic understanding of what every component does. So we'll start the chapter by examining the major components of the .NET Platform.



SAY HELLO

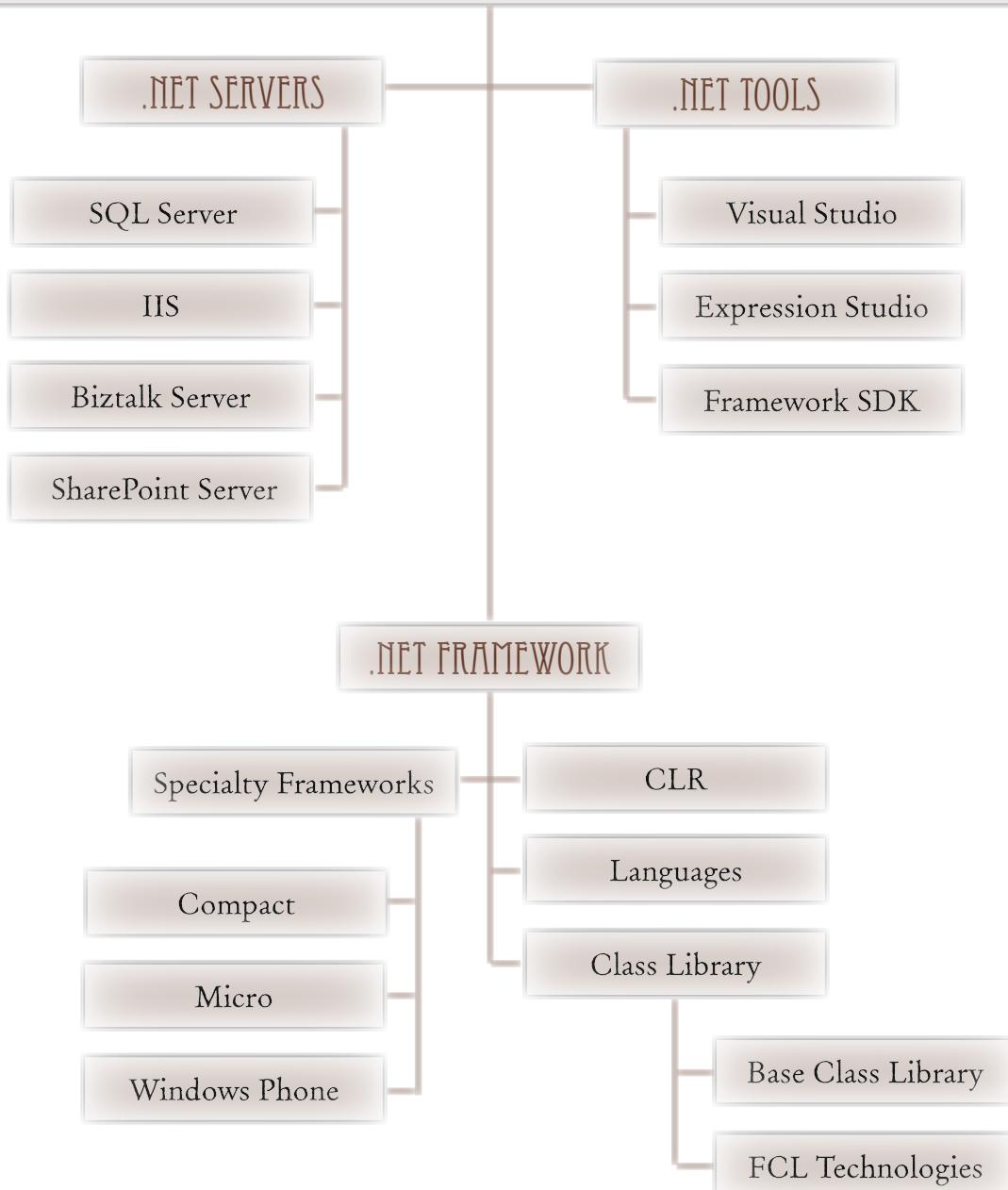
Once we've seen what the bits and pieces of the .NET Platform are, it's time to get our feet wet. It's a tradition that the first thing a programmer does when learning a new language is to get it to say hello to the world, and that's what we'll do by creating an application to introduce us to the world.



SAY WHAT?

Of course, you're not reading this book just to learn how to follow directions, so after saying hello we'll back up a little and examine what we've just done. It's important to understand the process because you'll do more or less the same thing for every application you build.

.NET PLATFORM COMPONENTS





PUT ON YOUR THINKING HAT

Can you answer these questions using the diagram?

These are the three primary categories of .NET Platform Components:

Visual Basic is one of these:

This is an alternative to Visual Studio for creating .NET applications:

You'd use one of these if you were writing for a mobile phone, for example:

This stands for Framework Class Library:

This is the .NET runtime environment. The letters stand for:

This is the primary development environment for .NET:



HOW'D YOU DO?

These are the three primary categories of .NET Platform Components:

.NET Servers

.NET Tools

.NET Framework

Visual Basic is one of these:

Framework language

This is an alternative to Visual Studio for creating .NET applications:

Framework SDK

You'd use one of these if you were writing for a mobile phone, for example:

Specialty Framework

This stands for Framework Class Library:

FCL

This is the .NET runtime environment. The letters stand for:

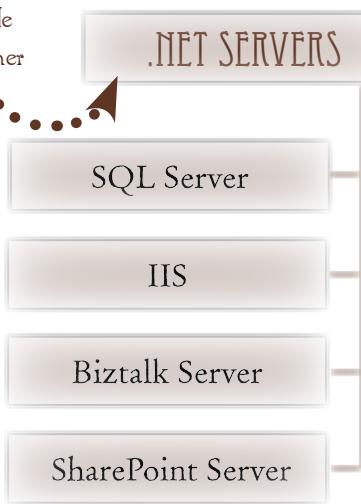
CLR -- Common Language Runtime

This is the primary development environment for .NET:

Visual Studio

.NET PLATFORM COMPONENTS

These provide services to other applications.



.NET TOOLS

You use these to create .NET applications.

Visual Studio

Expression Studio

Framework SDK

.NET FRAMEWORK

Specialty Frameworks

Compact

Micro

Windows Phone

CLR

Languages

Class Library

Base Class Library

FCL Technologies

Framework Technologies include things like ASP.NET and Silverlight that are used to develop Web applications, Entity Framework for working with data, and WPF/XAML, which we'll use to build our Windows interfaces.

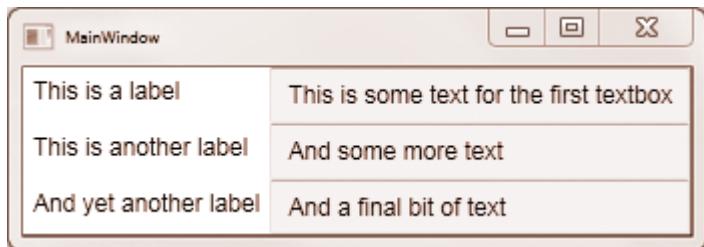
This is the primary focus of this book.

This provides useful code that can be used in any application. We'll start examining it in detail beginning in Chapter 10.

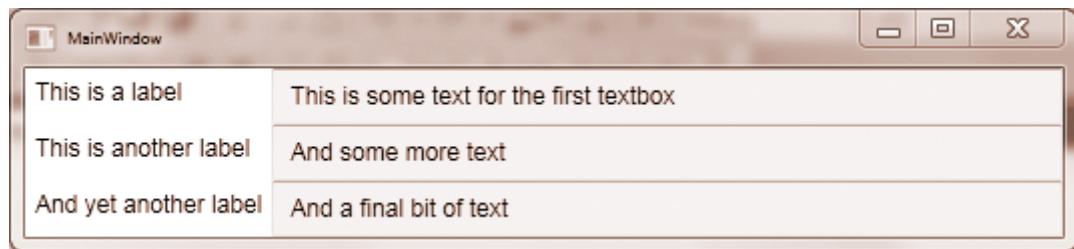
THE VALUE OF A FRAMEWORK

To get a sense of what the .NET Framework offers us, let's look at a common programming requirement.

Here's a very simple form. Just about any general-purpose programming language lets you create something like this easily.



When the user makes the form wider, you want the labels to stay the same size, but the textboxes to get bigger, like this:



Pretty simple stuff, right? You've seen it a million times in a million different applications. You'd probably find any application that *didn't* do this pretty clunky, wouldn't you? Let's look at what the programmer has to do to make it happen.

Before the release of .NET, Visual Basic 6 was the most popular language for developing Windows applications. There are bazillions of VB6 applications in existence, and almost as many VB6 programmers. Here's the VB6 code to resize our form (it probably wouldn't work for any other form):

Create a variable
to store the
original width of
the form.

Set the value
of the variable
when the form
is first loaded.

Loop through all
the controls on the
form and resize
the text boxes.

Reset the
variable.

```
Dim originalWidth As Integer

Sub Form_Load()
    originalWidth = Me.ScaleWidth
End Sub

Sub Form_Resize()
    If Me.WindowState <> vbMinimized Then
        Dim ctrl As Control
        For Each ctrl In Controls
            If TypeOf ctrl Is TextBox Then
                ctrl.Width = ctrl.Width * (Me.Width / originalWidth)
            End If
        Next
        originalWidth = Me.ScaleWidth
    End If

```

In reality, the resize routine would be a *lot* more complicated. If nothing else, it would have to account for changes in height as well as width. And, of course, the exact controls to be resized, and how they'd be resized would be different for every form and every type of control.

And the .NET version? There isn't one. Windows Presentation Foundation, which is one of the .NET Framework technologies (the one we'll be using), does all this for you...



TAKE A BREAK

We'll be looking at the .NET Framework in a lot more detail throughout the rest of the book, but you should have a sense of where the Framework fits in the overall .NET Platform now, so take a break before we tackle our next task.



REVIEW

What are the three categories of .NET Platform Components, and what does each of them do?

Which component will you always use when you program in Visual Basic.NET?



SAY HELLO

Ready to tackle some actual programming? Of course you are. In the next few pages we'll create the traditional first program for every beginner, "Hello, World". Here's what we'll need to do:

1

Create a Visual Studio project. A project is a container for your source code and related resources like connections to a database or resources that your application will use.

2

Use a Visual Studio Designer to create some widgets (in this case some controls in a WPF window), and set their properties. Visual Studio provides several designers for creating your applications visually instead of in code.

3

Use the Visual Studio Code Editor to write some actual Visual Basic .NET code. Unlike a simple text editor, the Code Editor provides a lot of features that make writing and managing your code simpler.

4

Run your application in debug mode, which helps you find any mistakes, from within Visual Studio.



1

CREATE THE PROJECT

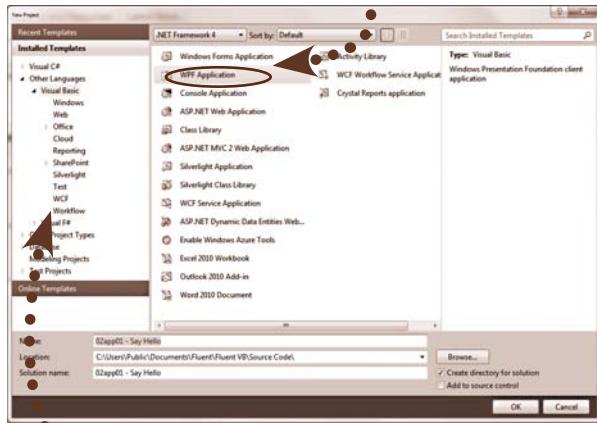
When you first load Visual Studio, the application will display a Start page that looks something like this, depending on which version of Visual Studio you have installed. This is Visual Studio 2010 Ultimate, but any version will be fine.

Click on New Project in the upper-left corner.

Click here. •••••



This is what
we'll create. •••••



••••• Your version of Visual Studio will probably have different options in this pane. Just make sure you're choosing the Visual Basic version.

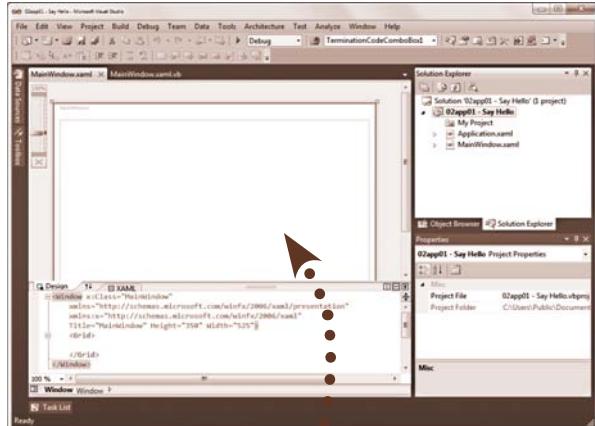
Next, Visual Studio will display the New Project dialog. Again, the details of this dialog will change a little depending on which version of Visual Studio you have installed. Don't let that throw you; just choose WPF Application from the Windows category of Visual Basic .NET projects.

You can leave the application and solution names as "WPFAApplication1", or change them to anything you like. Same with the location. (You might want to create a directory just for the projects you'll be creating for this book.)

When everything's the way you want it, click OK.



CREATE THE WINDOW

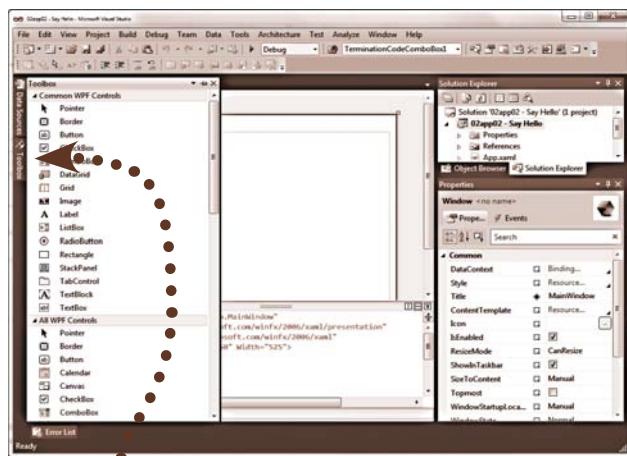


This is the DESIGN SURFACE.

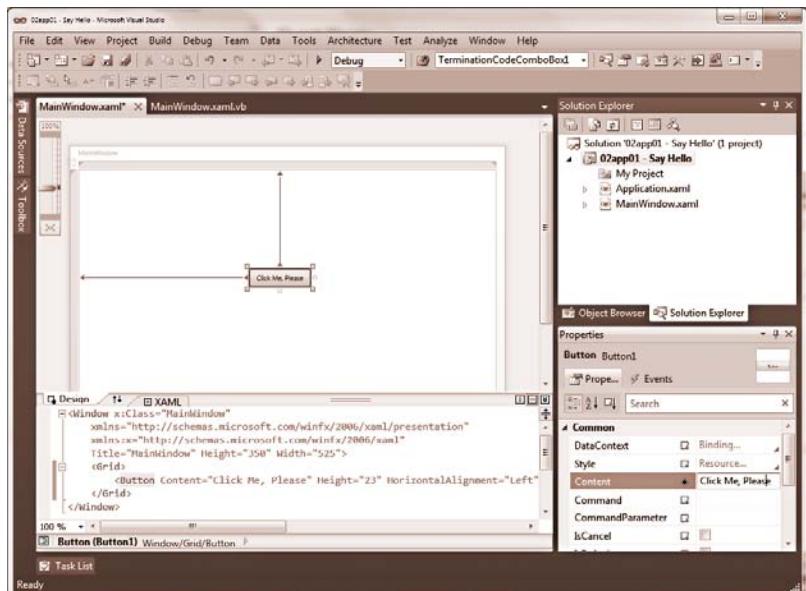
Find the word Toolbox on the left side of the screen and click it. The Toolbox will display on top of the Designer. (You can keep the Toolbox open beside the window by clicking the little push pin at the top. You don't have to do that now, but you can if you want to.)

The Toolbox contains all the widgets that you can drag onto a design surface when you're creating a window. Its contents will change depending on which designer is open (Visual Studio has several), and you can even add custom widgets to a Toolbox.

After you click OK on the New Project dialog, Visual Studio will work away for awhile and then display this screen. We'll look at what all the windows and tabs are a little later. For now, you need to know that the biggest part of the screen contains the WPF Designer, where you can create your application windows by dragging widgets onto the design surface. At the bottom of the screen, you can see the XAML that represents a WPF form. XAML is a special version of XML for defining forms. We'll learn how to use it starting in Chapter 15.



The Toolbox opens when you click its name here.



Use the Property window to control the widgets in the designer.

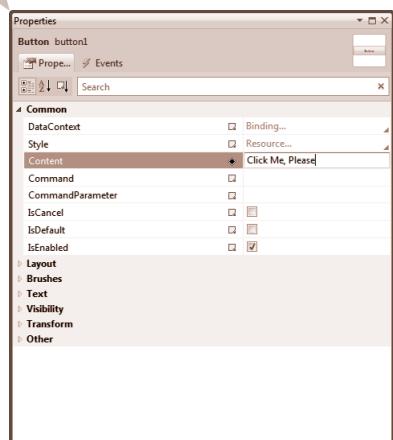
Find the button in the Toolbox and drag it onto the design surface. The Designer will display some guidelines to help you center the button in the window, but it's not really important that you do that for this example.

Notice that the Designer also adds some XAML code in the editor at the bottom of the screen. When you use a Visual Studio designer, it writes that code for you. How cool is that?

On the right side of the screen you'll see the Properties window. (If it isn't visible, choose Properties Window from the View menu.)

Like the Toolbox, the items in the Properties window are organized into groups. Make sure the button is selected in the Designer (click it if it isn't), and find the Content item in the Common section of the Properties window. (It will probably already be selected.) Change the value in the right cell from "Button" to "Click Me, Please" (or whatever appeals to you), and press the ENTER key.

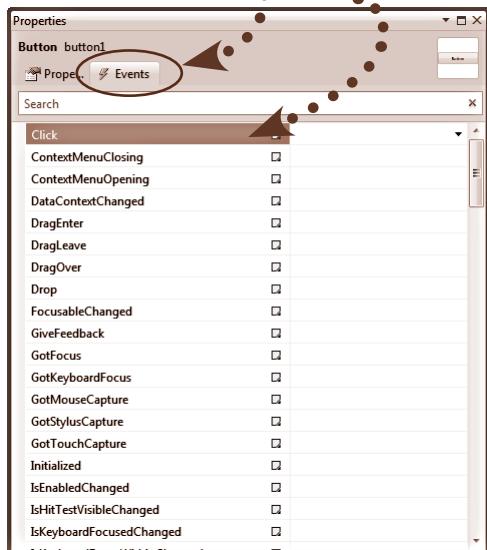
If you like, you can play around with setting some other properties. You can set the typeface in the Text section, or change the color of the button in the Brushes section. Just click in the pane and either type a new value or select one from a drop-down list.





WRITE SOME CODE!

Click here to display
the list of events... ...and then double-click
here.



When you double-click “Click” in the Events tab of the Properties window, Visual Studio will create an event handler for you and display the Editor window with the cursor positioned so you’re ready to add your code. An **EVENT HANDLER** is a piece of code that is executed when an event occurs—it handles events.

Add this line:

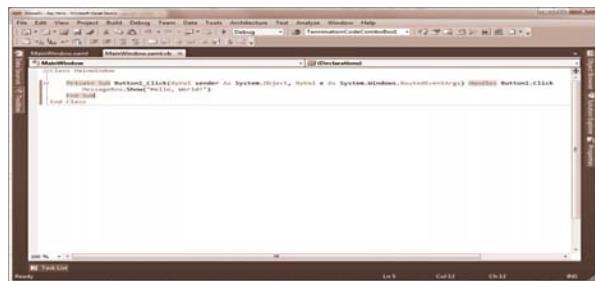
```
MessageBox.Show("Hello, World!")
```

When you start typing, the Editor will open an Intellisense list. You can select the item you want and press Tab to get Visual Studio to add the text to your code, or you can ignore it and keep typing. We’ll talk about Intellisense in detail in the next chapter.

The Properties window contains two tabs. We set the button content on the Properties tab. Now click the word Events at the top of the screen to display the Events tab.

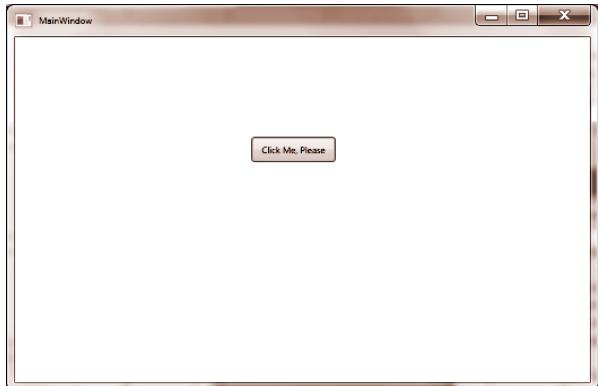
This tab displays all the things the selected widget can respond to. “Things the widget can respond to” are a kind of event. We’ll discuss events in detail later in the book.

Find the Click item in the list, and double-click it. Make sure you double-click the actual word “Click”, and not the cell on the right.



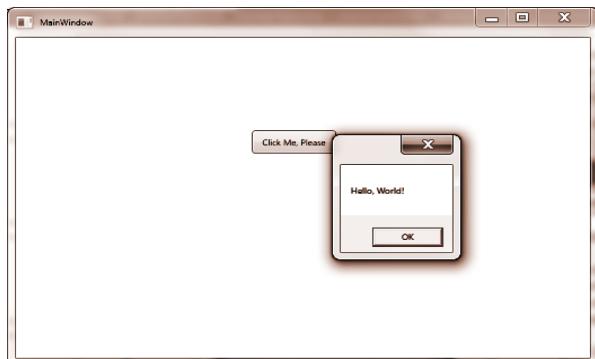


RUN IT



Click the button, and a message box will display the text you specified. The box might not appear in the middle of the window like the picture here; in fact, it probably won't, but it'll be on your screen somewhere, probably in the middle.

Press F5 to run the application. The window should open up on your screen, but if there's anything wrong, Visual Studio will give you an error message. Check the line of code you typed; that's the most likely place to have made a mistake.



TAKE A BREAK

Congratulations! You've officially written a program. Isn't that what programmers do?

Now why don't you take a short break before we come back to look at what we just did in detail?



NOW, WHAT WAS THAT?

"Hello, World!" may be traditional, but it's probably not the most exciting Windows application you've ever seen. In fact, it might just be the most boring application you've ever seen. So why bother? Because those basic steps are the same ones you'll use for every application you write.



CREATE A SOLUTION AND A PROJECT

Solutions are like filing cabinets that organize related projects. Projects are sets of source code files and other resources that create a single executable.



USE DESIGNERS & EDITORS TO CREATE THE CODE

We used the WPF Designer and Code Editor in our walk-through, but Visual Studio provides other versions of both.



TEST AND DEBUG THE CODE

Testing might identify a bug—something that doesn't work the way it should—or it might just reveal things you'd like to work differently.

A LITTLE BIT OF HISTORY

There's a story that goes around that errors in programs are called "bugs" because an actual bug (a moth) was found in the ENIAC, the first general-purpose electronic computer. Great story, isn't it? Unfortunately...the moth was found in the Harvard Mark II and it wasn't the first use of the word. In fact, "bugs" had been common in engineering at least as far back as Thomas Edison.

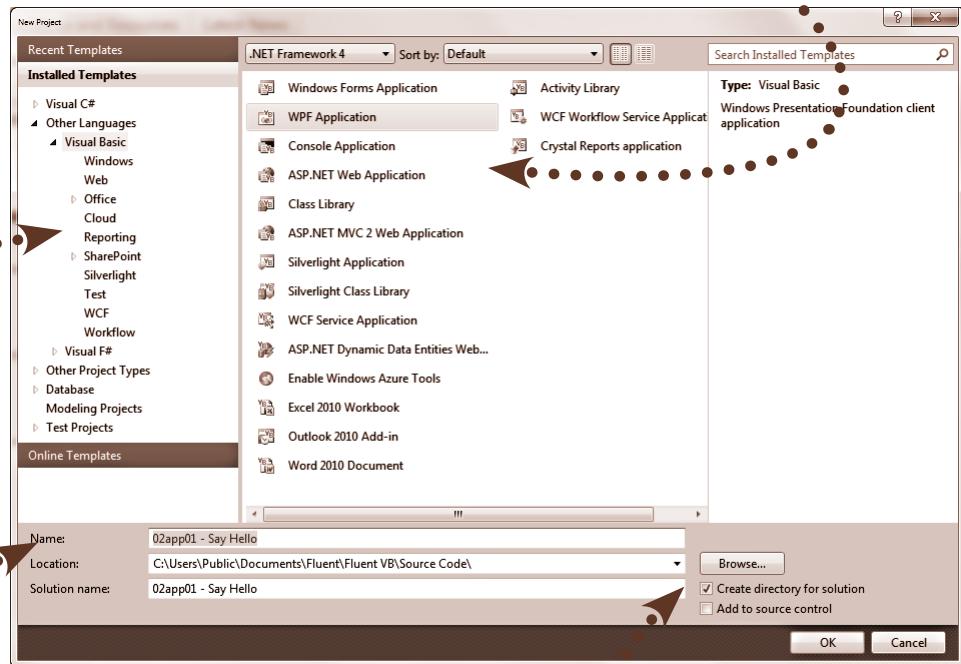


CREATING A PROJECT

The first thing we did was to create a new project. We'll look at projects and solutions in detail in the next chapter, but first let's have another look at that dialog.

You pick the kind of project you want to create from this list of templates. A template provides the standard files for a particular type. There's nothing magical about them, and you can even create your own, but they do save you a lot of work.

These categories control the templates shown to the right. They just make it easier to find things.



Visual Studio provides a default name for both the Project & Solution that are based on the template name. It usually makes sense to change them to something more meaningful.

This checkbox controls the directory structure that Visual Studio creates. You'll almost always want to leave it checked so it puts your project in a separate folder.



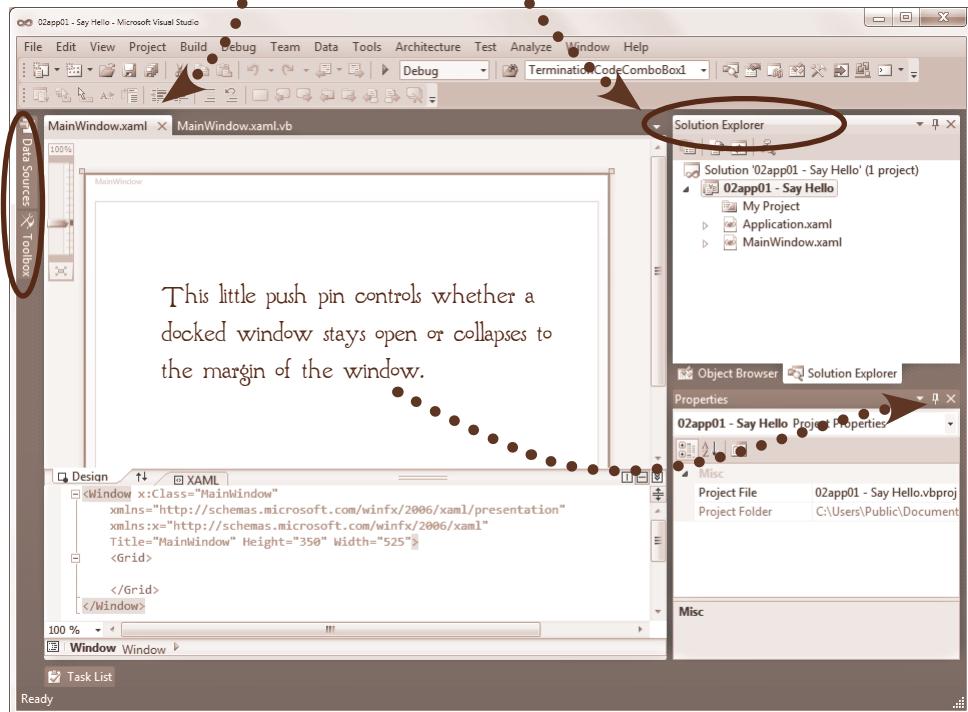
THE VISUAL STUDIO UI

As a Visual Basic programmer, you'll probably spend most of your time in Visual Studio.

We'll look at the UI in more detail in the next chapter, but for now let's have a look at the basic layout.

The Visual Studio window opens "documents" like source code or diagrams in tabs.

Other windows can be free-floating or docked, and when docked, they can be expanded or collapsed.



This little push pin controls whether a docked window stays open or collapses to the margin of the window.

This screenshot shows the WPF Designer, which by default shows two panes: a drag-and-drop designer at the top, and a XAML editor at the bottom. Like most things in Visual Studio, you can change the way this designer looks. Visual Studio has dozens of designers for different types of files. We'll look at a few more later in the book.

The Solution Explorer works like the Windows Explorer, showing you the projects in your solution and files in your projects.

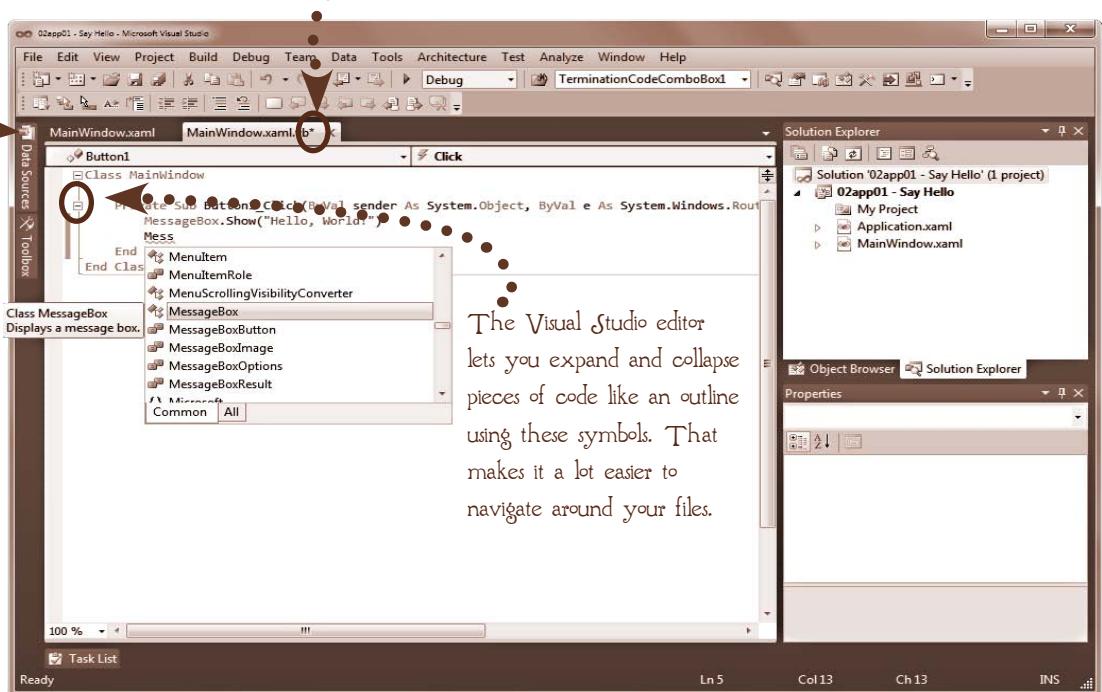


THE CODE EDITOR

Even though the .NET Framework reduces the amount of code you have to write and the Visual Studio designers write a lot of code for you, you'll still spend a lot of time using the Code Editor. We'll look at it in detail in the next chapter, but before we move on let's have a look at some of its special capabilities.

Source code is a "document" (actually a text file with the extension .vb), so it displays in a tab in Visual Studio.

The asterisk after the name means the file has unsaved changes.



The Visual Studio editor lets you expand and collapse pieces of code like an outline using these symbols. That makes it a lot easier to navigate around your files.

This is an example of Intellisense. It's kind of like auto-complete, but a lot smarter. Visual Studio only lists things that are meaningful in context, and as you can see, it displays a little tooltip about whatever is selected in the drop-down. Intellisense can do a lot more than this, too, like automatically adding code. We'll talk about it in the next chapter.



ON YOUR OWN

Before we finish up the chapter and go on, spend some time exploring the Visual Studio UI and playing with our little sample application:

Look at the project types available in the New Project dialog. Why do you think Visual Studio includes so many templates?

Going back to the list of applications you'd like to develop from Chapter 1, pick a couple and, based on the descriptions in the New Project dialog, think about what project types you might use to develop them.

Going back to the walk-through, our screen isn't very pretty. Using the WPF Designer, can you make it look a little nicer? (Hint: Select items on the design surface, and then look at the Brushes and Text sections of the Properties window.)

In the Code Editor, change the text that is displayed when you click the button.

Here's a real challenge: Using Intellisense to help you, can you add a caption to the MessageBox? (Hint: Your caption needs to be in quotation marks, just like the MessageBox text.)



TAKE A BREAK

That's the final task done! Why don't you take a short break and relax a bit before you come back to complete the final exercise and finish off the chapter?



REVIEW

Time for one last review before we dive into using Visual Studio in the next chapter.

What are the three steps that you use to create almost every application?

How do you control the appearance of widgets in the WPF Designer?

What are the three basic parts of the .NET Platform?

How do you expand and collapse code in the Code Editor?

What does the pushpin widget do in Visual Studio?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



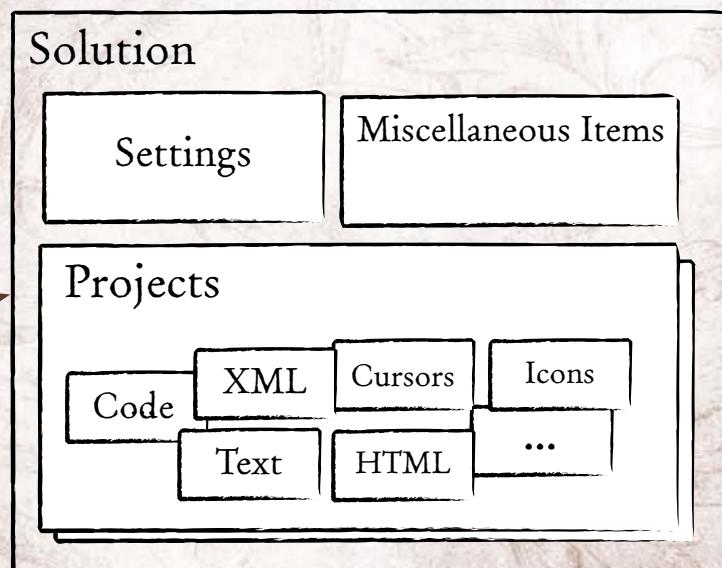
THE VISUAL STUDIO UI

3

In the last chapter you wrote your first program and saw the basics of the Visual Studio user interface (UI). Now it's time to look at these steps in more detail. We'll start by looking at how Visual Studio helps you manage a development project with Solutions and Projects, and then take a closer look at the UI and how to configure it to suit the way you work.

As a programmer, you'll spend a lot of time in the Code Editor, and so will we. We'll look at the basic text editing functions it provides and also at Intellisense and the Visual Studio help system.

Solutions contain Projects, Solution Settings that control how the application will be compiled and run, and Solution Items that aren't part of a specific project.



Solutions can also contain other files that aren't included in the application but are available from the Solution Explorer when the Solution is open.

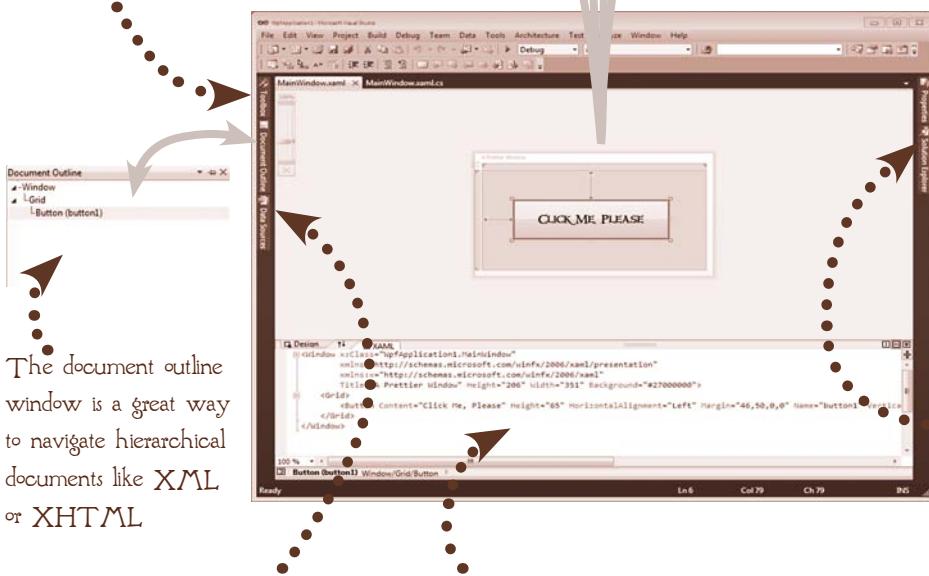


IN A NUTSHELL



The contents of the Toolbox change to reflect the kind of document you're working on.

By default, "documents" and certain kinds of dialogs are displayed as tabs in the central portion of the Visual Studio window.



The document outline window is a great way to navigate hierarchical documents like XML or XHTML.

The Data Source window is like an explorer for external data like a relational database.

Some Visual Studio Designers can display multiple panes. The WPF Designer shown here, for example, shows XAML and a design surface you can use for drag-and-drop.



The Properties window is a quick and easy way to change the attributes of something that's selected in a designer. You can also use it as a shortcut to create and manage event handlers.

- Like the Windows Explorer, the Solution Explorer helps you navigate & manage the files and folders associated with your project.



TASK LIST

A craftsman is master of his tools. As a programmer, your primary tool is Visual Studio, and in this chapter we'll begin the process of mastery by examining its user interface in detail.



SOLUTIONS, PROJECTS & STUFF

It's convenient to think of application development like writing an essay or book: You do some research, prepare an outline, and then produce a final document. Unfortunately, the development process isn't that neat. (Neither is writing, of course, at least not the way I do it.) Most development projects don't even have a single output that's equivalent to that essay. So we'll start this chapter by looking at the way Visual Studio uses Solutions, Projects and Solution Items to manage all the bits and pieces that you'll actually be working with.



TAKE CONTROL

I bet you've changed your Windows desktop. If you're like most people, you've added widgets to the sidebar, created some shortcuts, and rearranged the Start menu. All those little changes just make life a little easier by putting the tools you use all the time close to hand. Visual Studio does a pretty good job of arranging the user interface to accommodate general programming, but you'll benefit from making the same sorts of customizations to its workspace as you made to the Windows desktop, so the next thing we'll do is learn how to do just that.



GET SOME HELP

In the last chapter we saw an example of Intellisense when we were able to pick the `MessageBox.Show` command from a drop-down list. In this chapter, we'll look at Intellisense in more detail, along with some of the special error-checking capabilities that the Visual Studio Editor provides.



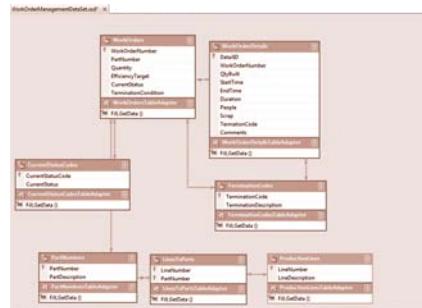
SOLUTIONS, PROJECTS...

Solutions are like filing cabinets that hold and manage Projects and other files. Easy enough in principle, but what exactly does that mean? Why would you have more than one Project? What are these "other files", and what exactly does "manage" mean? Let's look at some examples of the kinds of files you might include in a Solution.



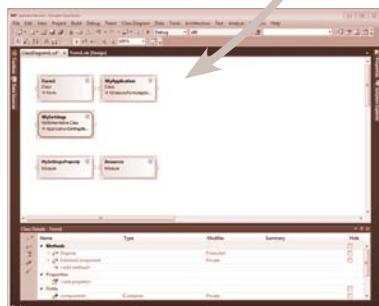
PRIMARY UI PROJECTS

You'll usually have a project (sometimes more than one) that contains the forms that comprise your application user interface.



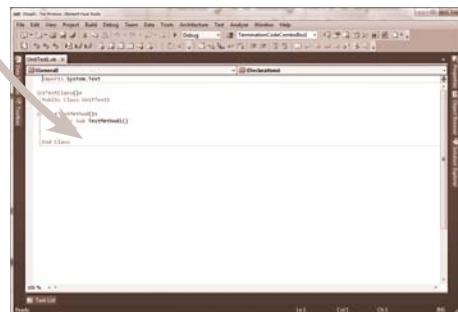
DATASET PROJECTS

If your application references a data source, you'll usually have a separate project that handles the data interface.



UTILITY CLASSES

In cooking, you often need to translate between teaspoons, tablespoons and cups. The classes that do that conversion might be useful in other applications, so it makes sense to put those in a separate project that we can reference when we need them.

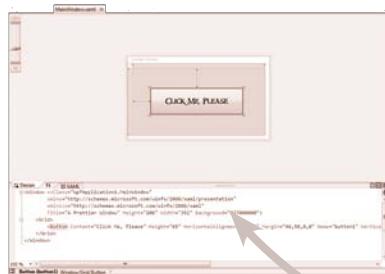


TEST PROJECTS

Visual Studio 2010 provides great support for a technique called Test-Driven Development. If you adopt this approach, you'll need projects that contain your tests.

...AND STUFF

Projects have multiple files, as well. You'll have code and designer files, of course, you've already seen that, but Visual Studio will also allow you to associate other files with the project, just to keep them handy.



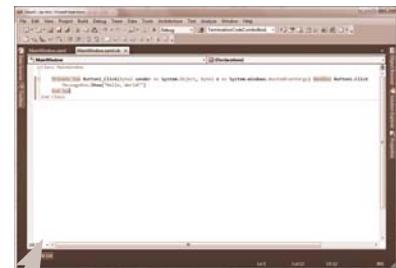
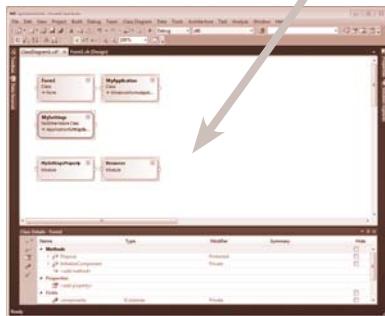
DESIGNER FILES

These are the files that are created by the Visual Studio Designers. They're text files, but if you mess with them outside the Designer, your changes might get overwritten the next time you use the Designer, so you'll typically leave them alone.

DESIGN DOCUMENTS

You can also include documents like this class diagram or even specifications.

These files aren't part of the project code.



SOURCE FILES

These are the files that contain your code. They have the same name as the files the Designer creates, with the extension ".vb".



RESOURCES

If your project includes things like custom cursors or icons, these are separate files in the Project.



PUT ON YOUR THINKING HAT

Find at least two ways to add a Project to a Solution, and two ways to add a file to a Project.



HOW'D YOU DO?

The problem was to figure out how to add Solutions and files....

TO ADD A PROJECT TO A SOLUTION



You can also add existing Projects, which is handy if you're reusing utilities or custom widgets, or exclude a project that you've added by accident.

The New Project item can be found on the File menu and on the context menu displayed when you right-click the Solution name in the Solution Explorer.



TO ADD A FILE TO A PROJECT



You can add new and existing items from the Solution Explorer by right-clicking the Project name.



The Project menu provides specific options for the most common types of Projects, or you can choose Add New Item... to display the New Item dialog.

ON YOUR OWN



You know how to add an item to a Project, and you know how to configure a simple WPF window, so let's put those two things together. Change your Hello, World application to display a window instead of a MessageBox:

- ① Add a new window to the Project. Accept the default name of `Window1`.
- ② Drag a Label from the Toolbox to the window design surface, and configure the window and label properties however you like. Have some fun! You can't hurt anything.
- ③ Display the code for the original form (NOT the one you just created). If the form is open, you can double-click the button or press F7. If the form isn't open, you can right-click the form name in the Solution Explorer and choose Show Code.
- ④ Delete the line that displays the MessageBox, and replace it with the following:

Declare a variable called "newWin". VARIABLES are just a name for a piece of memory that can store information or objects of a certain type. This one stores a Window.

```
Dim newWin As New Window1()  
newWin.ShowDialog()
```

The variable is
INITIALIZED (given an
initial value) with an
instance of `Window1`, the
Window you just created.

You'll use variables a lot when you program.
We'll look at them in Chapter 5.

This line calls the `ShowDialog()` method of the
window. METHODS are something an object can do.
We'll look at them in detail in Chapter 8.

- ⑤ Run the application by pressing F5, and then click the button on the first form.

TAKE A BREAK



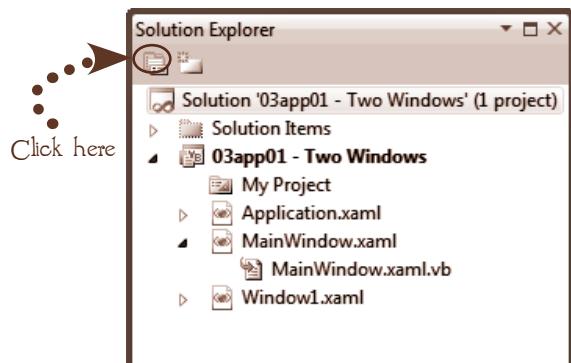
Why don't you take a quick break before we move on to controlling the way Solutions and Projects behave by setting their properties.

SOLUTION AND...

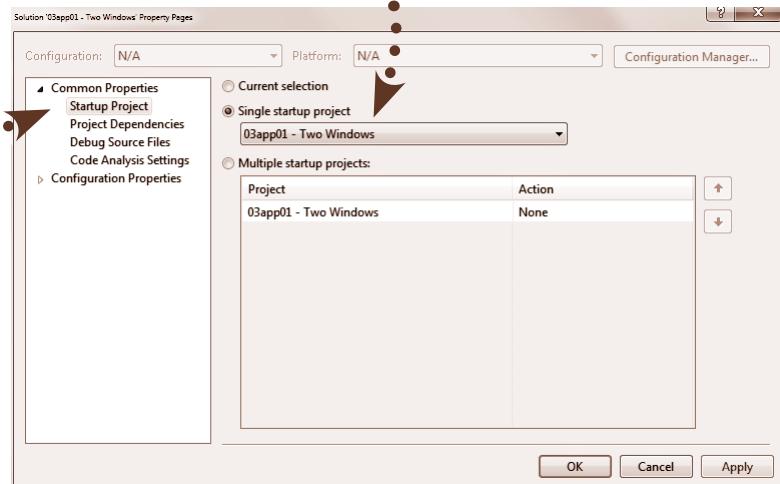
When you created and modified your windows on the design surface, you saw that you could control the appearance of the widgets by setting their properties. Well, Solution and Projects have properties, as well. We'll need some of these as we move through the book, so let's get started by looking at how to display the property dialogs.

The easiest way to display the Solution Properties dialog is to select the Solution in the Solution Explorer and click the Properties button on the Solution Explorer toolbar, but you can also choose Properties Page from the View menu.

Most of the Solution properties are managed by Visual Studio, and you only need to change them in unusual circumstances, but you'll often need to specify the Startup Project whenever you have a Solution that contains multiple Projects. The file specified as the Startup is the one that Visual Studio will run when you press F5 or run the final application.



By default, Visual Studio sets the first project you add to the Solution as the Startup Project. You can change that by choosing a different Project from the combobox.



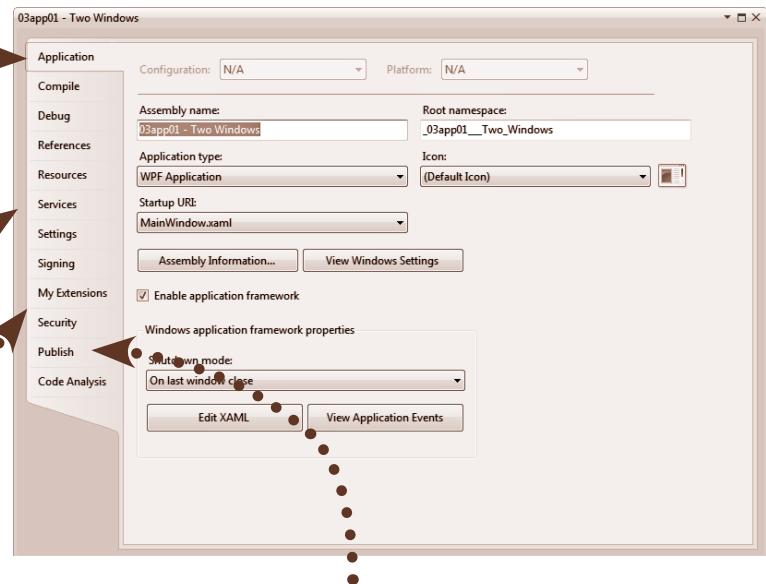
...PROJECT PROPERTIES

Solution properties display in a dialog box, but Visual Studio has a designer (called the Project Designer) for project properties that displays in a tab. You can display Project Designer by clicking the Solution Explorer toolbar button when a Project is selected or by choosing <ProjectName> Properties... from the Project menu when the Project is selected, or by right-clicking the Project name in the Solution Explorer and choosing Properties, or by selecting the Solution name in the Solution Explorer and pressing Alt+Enter. And probably by doing some other things that I haven't discovered yet...

The Application tab will change depending on the type of Project you choose, but it always controls the type of application and how it is compiled.

Resources and Settings are things like icons and strings that are included in the executable. We'll use these tabs in just a minute to create an icon for our application.

Signing and Security help you secure your application and its users from bad people and bad software. Security is an important issue, but it's also a huge one, so we won't be talking about it in any detail.



Not all of these options are available in every version of Visual Studio. The Code Analysis tab, for example, only appears in Visual Studio Premium and Ultimate. So don't panic if your screen looks a little different from this one. (You are opening these screens, right?)

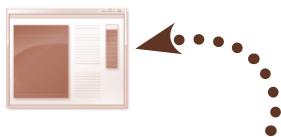
The Publish tab is used to deploy your application using ClickOnce. We'll talk about that in the next chapter.



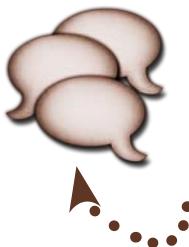
ADD AN ICON

All Windows applications need an icon. If you don't provide one, Visual Studio will use the default icon. The default image isn't very exciting, and it doesn't distinguish your application from all the others out there. So let's use the Project Properties dialog to add a custom icon to Hello, World. To make that happen in a WPF application (other application types can be a little different), we need just three steps:

- ① Specify the icon file in the Application tab of Project Properties.
- ② Build the application to make the icon available.
- ③ Set the icon property of the window in the WPF Designer.



This is the default icon. Pretty boring, huh?



We'll replace it with this one that looks a bit cooler and represents what our application actually does. This icon file is called `conversation.ico`, and it's included with the sample code. You can use this one, or any other icon file you like (try searching for `*.ico` in the Windows Explorer). Just copy it to the application folder for our sample app.

MAKE A NOTE

Visual Studio includes simple editors for most resource types, including icons, from the Resources tab of Project Properties, or you can use a third-party tool. You can even open most third-party tools right inside Visual Studio by right-clicking the resource and choosing Open With... .

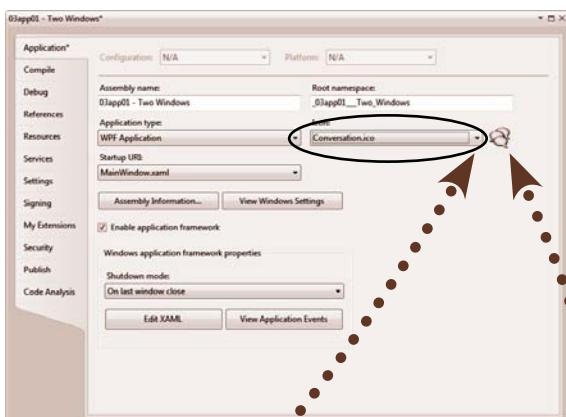
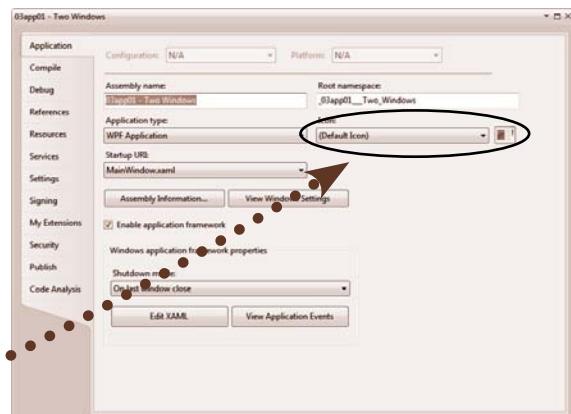
And if you're not feeling particularly artistic, there are lots of icon sets available for free or fee on the Web that you can use. Just be sure to respect the artists' terms of use, or the karma gods will get you, even if copyright law doesn't.



SET THE APPLICATION ICON PROPERTY

A If Hello, World isn't still open from the last exercise, open it from the Visual Studio Start Screen (it will be listed on the left side under Recent Projects) or from the File menu. Display the Project Properties using any of the techniques you've learned, and then select the Application tab.

We'll set the icon here.



Click here to open the dialog.

After you select your icon, Visual Studio will display a thumbnail of it here.

B Open the combobox and select <Browse...>, and Visual Studio will display a standard File Open dialog. The first time you open the dialog, Visual Studio might take you to the Microsoft Visual Studio\Common7\IDE folder, which can be a little scary, but just navigate to the folder for the application and choose conversation.ico (or whichever icon file you chose to use).

After you select the icon and click the Open button, Visual Studio will show the icon on the tab.



BUILD THE APPLICATION

There are a lot of files involved in creating a Visual Studio application. In addition to the source files that you create, the resource files like icons that you create or reference, and the final executable created by the compiler, there are intermediate files that Visual Studio creates for you. When you set properties or resources, you need to tell Visual Studio to recreate some of these "behind-the-scenes" files so that they're available to other components like the designers. You do that by BUILDING the application.

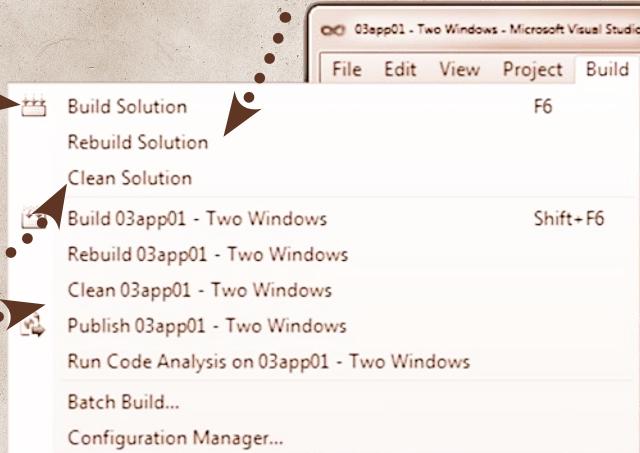
THE BUILD MENU

Most of the time you can use Build Solution, which only rebuilds the files that have changed.

Clean Solution removes any intermediate files created by Visual Studio but doesn't build them. If Rebuild Solution seems to be acting strangely, try cleaning the solution first.

These options build and clean just the Project, not the whole Solution. Because our application only has a single project, there isn't any real difference, but when you have a lot of Projects in a Solution, these alternatives can save a lot of time.

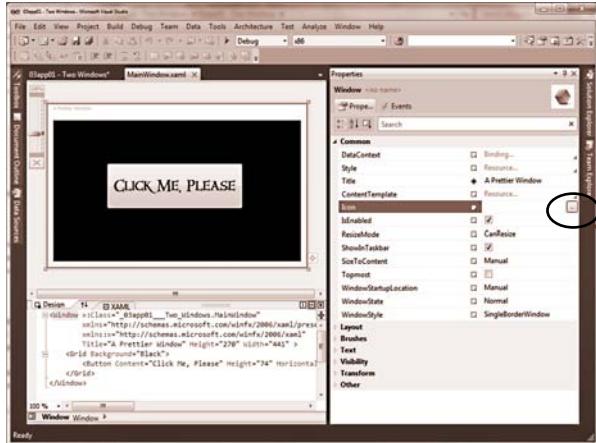
Rebuild Solution is a safer, but slower, choice. It rebuilds all the files, whether they've changed or not.



Press F5 or choose Build Solution from the Build menu so that the icon will be available to the WPF Designer.



SET THE WINDOW PROPERTY



A

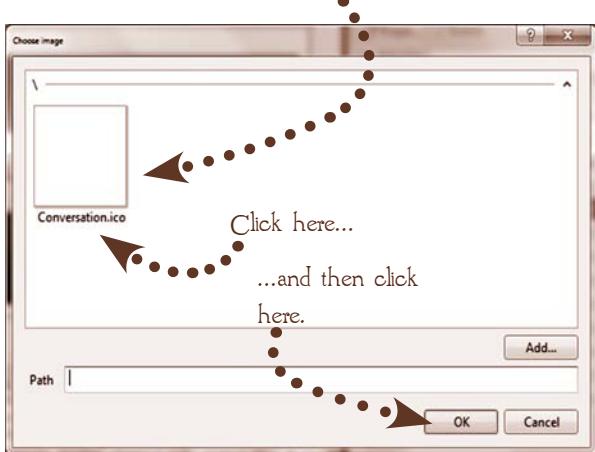
If necessary, double-click `MainWindow.xaml` in the Solution Explorer to open the WPF Designer. In the Properties window, find the Icon property, select it, and then click the ellipsis button.

Design. In the Properties window, find the Icon property, select it, and then click the ellipsis button.

Click here.

B After you click the ellipsis in the Properties window, Visual Studio will display the Choose Image dialog. Your new icon will be displayed, but there might not be a thumbnail. It's okay; Visual Studio just hasn't caught up with us.

Click on your icon, and then click OK to set the property.



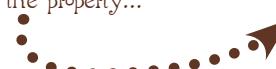
It's not a problem that there's no thumbnail.

After you click the OK button, Visual Studio will set the property to a long value that begins with `pack://application.../`. That's just WPF-Speak for "look in the application file", and we'll figure out how it all works later when we examine WPF Resources.

DID IT WORK?

Don't take my word for it. Run the application and find out...

The icon shows up in the window where we set the property...



But not in the window where we didn't. • • •



The application executable also uses the icon, as you can see in the bin/debug folder in Windows Explorer.



ONE MORE TIME...

Let's run through the steps to add an application icon one more time...



Add the icon file to the application in Project Properties.



Build the application to make the application available in the Designers.



Set the Icon property of the windows where you want the icon displayed.



TAKE A BREAK

You've finished the first task of this chapter, so take a short break to let it all settle before moving on. But before you go, stop for a minute to think about what you've achieved...

- You created an application that displayed a Window and a MessageBox.
- You changed the appearance of the Window.
- You added a second window to the Project and wrote the code to display it.
- You added an icon to the application and the window.

That's a lot when you see it listed like that, isn't it? Go, you!



REVIEW

Just a few exercises before we move on...

Solutions and Projects, Projects and Solutions. One's like a file folder, one's like a filing cabinet. Which is which?

/ Solution

Project

List three ways to add a Project:

①

②

③

List two ways to show the Solution Properties dialog:

①

②

List three ways to show Project Properties:

①

②

③

Change the application icon to something else. What happens to the window?

In the walkthrough, we only changed the icon of the main window. Add it to the other window in the application, as well.



TAKE CONTROL

Visual Studio is a Windows application, and for the most part it behaves like any Windows application, with menus and toolbars and document windows where you do your work. But the work you do in Visual Studio is quite specialized, and the IDE adds some special capabilities to make it possible to work just the way you want to work.

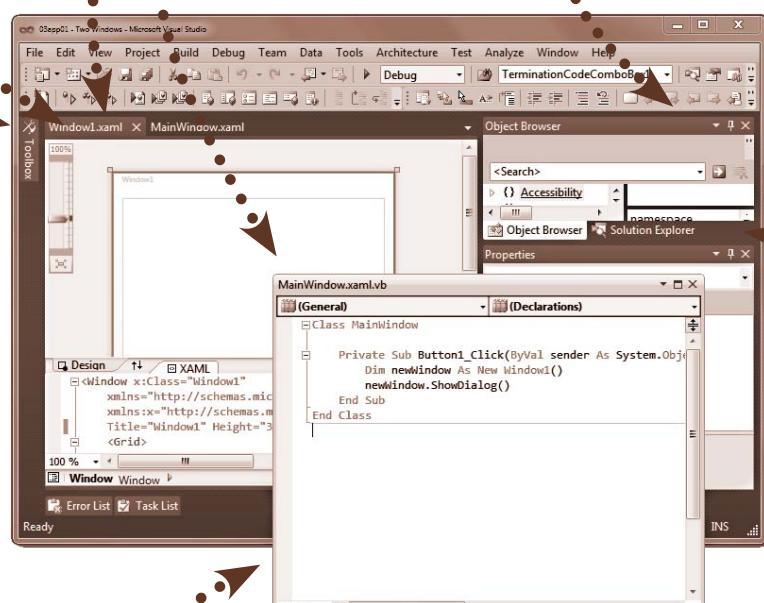
This is a document window displayed as a tab, the default view.

You can display documents in tabs or as separate windows. You can even drag them outside the main IDE window.

Tool windows can be docked to the edge of the IDE window or each other. When docked, they can be opened or closed.

This is a tool window that is docked and closed.

This is a document displayed in a floating window. It's not constrained by the Visual Studio IDE window.

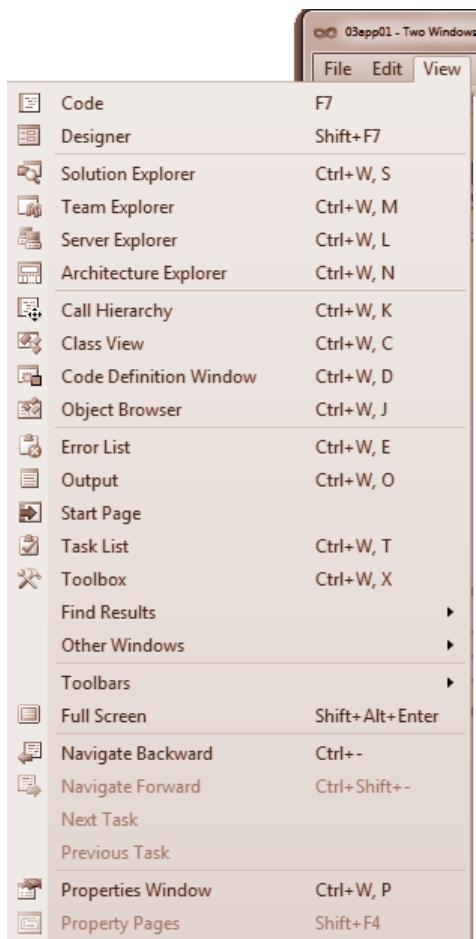


Tool windows that are docked to each other are displayed as tabs.

This is a tool window that is docked and open.

ARRANGING WINDOWS

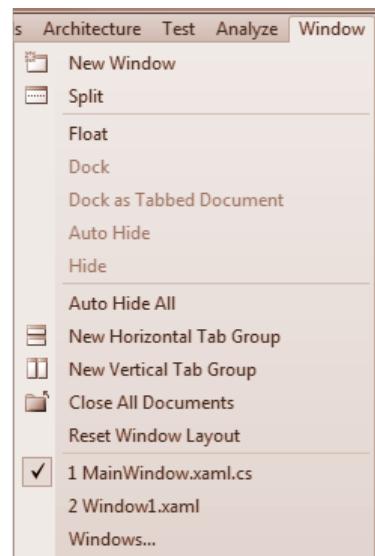
Like any Windows application, you can control the individual windows in Visual Studio through the View and Window menus or by dragging the title bar of a window.



The View menu controls the display of tool windows. (Don't confuse "tool window" with "Toolbox". The Toolbox is a tool window, but so are the Properties window and the Solution Explorer.)

Do you remember how to open a document window? Double-click on its name in the Solution Explorer.

The Window menu controls the display of open windows in the IDE. The most important item on this menu might be Reset Window Layout, which puts everything back in place when you get things messed up. (And you will, trust me.)



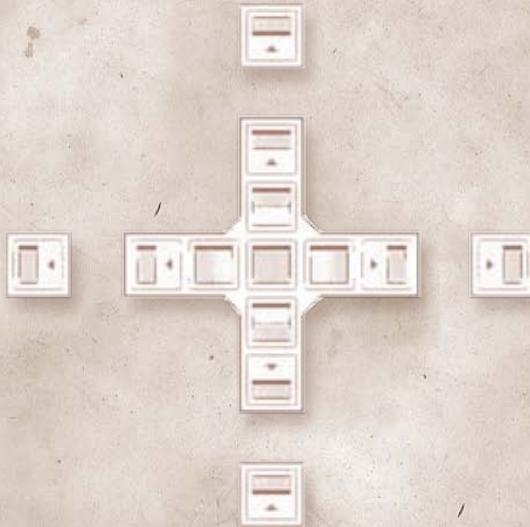


ON YOUR OWN

The best way to learn how to control the windows in Visual Studio is to play with them, so take a few minutes to move the Object Browser around. The Object Browser lets you look through a class hierarchy. You'll find out what that means in Chapter 7.

Remember, you can always start over by choosing Reset Window Layout from the Window menu...

- ① Show the Object Browser by selecting its name on the View menu. It will probably display as a tab. If it doesn't, drag its title bar until it does.
- ② Make it float by dragging its title bar, or selecting Float from the Window menu.
- ③ Dock it to the left side of the screen along with the Toolbox. It will be open when you first dock it, so click the to collapse it.
- ④ Drag it over so it displays as a tab in the same pane as the Solution Explorer.
- ⑤ Use Reset Window Layout to put everything back the way it was originally.
- ⑥ When you're dragging a window around, Visual Studio will display this odd docking widget. Can you work out what each bit does?



MORE THAN EDITING...

Basic text editing in Visual Studio complies with Microsoft Windows standards. You can double-click to select a word, ctrl-click to extend the selection, and cut, copy or paste selections just the way you're used to. But the Visual Studio Code Editor also includes a seemingly magical tool called Intellisense that turns it from a stenographer into a personal assistant.

Intellisense will even create standard code and objects for you.

Send this guy a "Don't call us, we'll call you" letter.

What colors does the new gizmo come in?

Intellisense shows you what an object can do as you type, so you don't need to memorize a lot of detailed syntax.

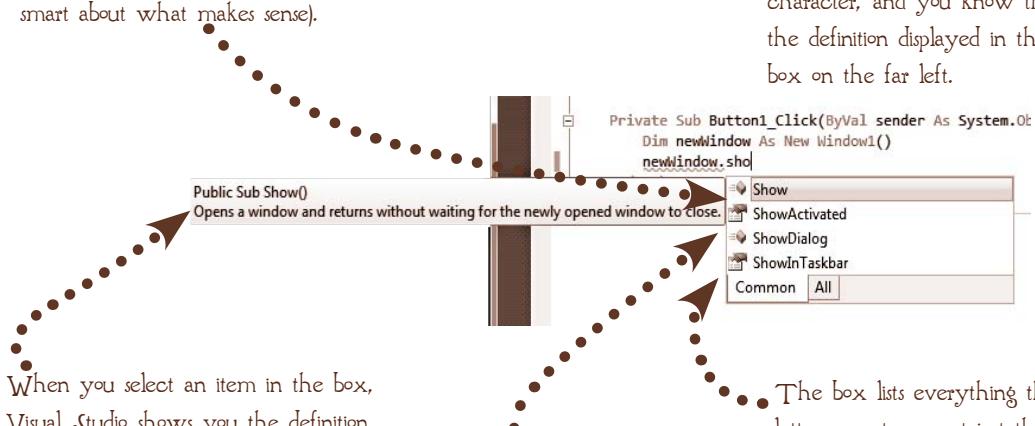
What are the specifications for this new gizmo?



LIST MEMBERS

The Intellisense capability that you'll probably use most often is the LIST MEMBERS function that displays a list of valid "things" that can be inserted where you're typing. You saw the list members function when you built your first application in Chapter 2.

As soon as you type a single character, the List Members box will open. Visual Studio is pretty smart about knowing what you can do, and it won't list things that don't apply (but it's not very smart about what makes sense).



When you select an item in the box, Visual Studio shows you the definition of the item, a short description, and, if the item is a method (something an object knows how to do), any exceptions the method can throw in a little help box.

An EXCEPTION is the way an object lets the rest of the program know something is wrong. We'll look at exceptions and what to do about them in Chapter 7.

To insert the selected item, press the Tab key or type the character after the item. In this case, that would be the ")" character, and you know that because of the definition displayed in the little help box on the far left.

To highlight an item in the List Members box, you can keep typing characters to limit the list or use the up and down arrow keys.

The box lists everything that contains the letters you type, not just the ones that begin with what you typed. Typing "n", for example, would display both "NewItem" and "EditNew", assuming they were available.

An illustration of a book cover with a textured, aged paper appearance. In the center is a circular seal with a decorative border. Inside the seal, there is a small illustration of a person sitting at a desk, possibly reading or writing. To the right of the seal, the words 'ON YOUR OWN' are written in a bold, serif font. Below this, there is a block of text: 'Try using Intellisense to change the ShowDialog() method call in the sample application to Show(). Run it. How is the behavior different?' The entire illustration is set against a light beige background.

PARAMETER INFORMATION

METHODS are things that objects can do. (We'll look at exactly what "object" and "method" mean in detail in Chapter 8.) Some methods take PARAMETERS, which are bits of information that you pass to the method to control exactly how it does whatever it is it does. In the bad old days before Intellisense, programmers spent a lot of time trying to remember exactly what parameters a method took, and in what order. The Intellisense Parameter Info box eliminates all that by showing you exactly what your options are.

As soon as you type the opening paren of a method call, Intellisense displays the Parameter Info box that shows you the method definition, a description of the method, and of the first parameter.

MessageBox.Show("A Message",
▲ 1 of 5 ▼ Show(messageBoxText As String, caption As String) As System.Windows.MessageBoxResult
Displays a message box that has a message and title bar caption; and that returns a result.
caption: A System.String that specifies the title bar caption to display.

If a method has different versions, you can use the up and down arrow keys to scroll through them.

As you type each parameter, Intellisense updates the display to show the description of the next one.

MessageBox.Show("A Message", "caption",

▲ 1 of 4 ▼ Show(messageBoxText As String, caption As String, button As System.Windows.MessageBoxButton) As System.Windows.MessageBoxResult
Displays a message box that has a message, title bar caption, and button; and that returns a result.
button: A System.Windows.MessageBoxButton value that specifies which button or buttons to display.

System.Windows.MessageBoxButton.OK = 0
The message box displays an OK button.

- MessageBoxButton.OK
- MessageBoxButton.OKCancel
- MessageBoxButton.YesNo
- MessageBoxButton.YesNoCancel

Common All

List Members works right alongside Parameter info, so sometimes the screen can get a bit crowded! Mostly it's helpful, but if all the windows get in your way, you can always make them go away by pressing Esc.

ROAD MAPS

Several editing functions help you keep track of where you are and what you've done. Using them is pretty intuitive, but here's a quick rundown:

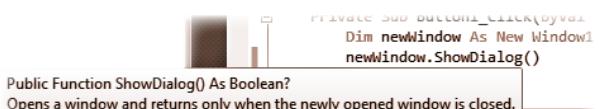
CODE OUTLINING

Click on the + or - characters in the left margin to collapse and expand units of code.

```
Class MainWindow
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        Dim newWindow As New Window1
        newWindow.ShowDialog()
    End Sub
End Class
```

QUICK INFO

Hover the mouse over any identifier, and Intellisense will display its definition and a description. (We'll see how to create descriptions for the code you write a bit later.)



REFERENCE HIGHLIGHTING

Select any symbol in the editor by clicking in it, and Visual Studio will highlight all the references to that symbol in the code.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Dim newWindow As New Window1()
    newWindow.ShowDialog()
End Sub
```



TAKE A BREAK

You've almost finished this chapter, so take a short break before you come back for the final Review.



REVIEW

Name at least one way to add a Project to a Solution:

Name at least one way to add a File to a Project:

Where would you assign an application icon?

What does this widget do?



On a new, blank line inside the `Button1_Click` event handler, type an “a” to trigger Intellisense. What’s the description of the `Array` object?

How many versions of the `Array.BinarySearch()` method are there?

Hover the mouse over one of the instances of the `Window` identifier. (There are two in your source file.) What’s the description of the window?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a C# programmer?

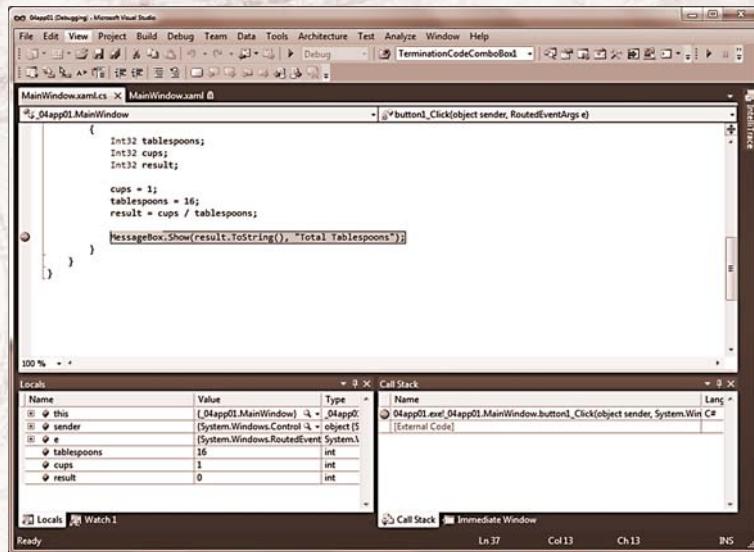
Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



TEST AND DEPLOY

4

The little applications we've been building so far have been simple throwaways, intended only to demonstrate the use of the Visual Studio IDE and you've been given pretty detailed instructions for creating them. But it won't be long before you're creating applications on your own, and if you get all your code right on the first try, you'll be the first to do so. So we'll start this chapter off by looking at the support



A screenshot of the Microsoft Visual Studio IDE. The main window shows the code for `MainWindow.xaml.cs`. The code contains a method `button1_Click` that calculates the total number of tablespoons from cups and displays a message box with the result. The Locals window shows variable values: `tablespoons` is 16, `cups` is 1, and `result` is 0. The Call Stack window shows the current call stack entry: `button1_Click(object sender, RoutedEventArgs e)`.

```
MainWindow.xaml.cs
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.RoutedEventArgs;

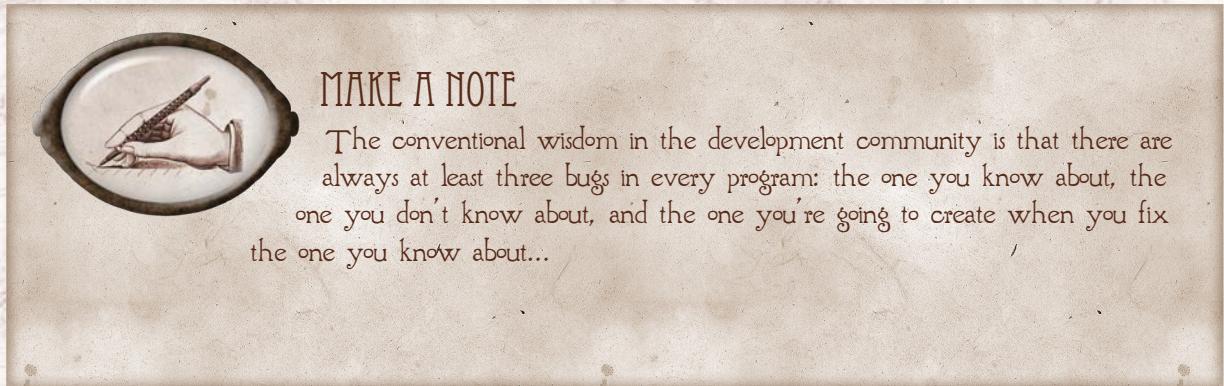
namespace _04app01
{
    public partial class MainWindow : Window
    {
        private void button1_Click(object sender, RoutedEventArgs e)
        {
            Int32 tablespoons;
            Int32 cups;
            Int32 result;

            cups = 1;
            tablespoons = 16;
            result = cups / tablespoons;

            MessageBox.Show(result.ToString(), "Total Tablespoons");
        }
    }
}
```

Name	Value	Type
this	[_04app01.MainWindow] 4	[_04app01.MainWindow]
sender	[System.Windows.Controls] 4	object [System.Windows.RoutedEventArgs]
e	[System.Windows.RoutedEventArgs] 5	System.Windows.RoutedEventArgs [System]
tablespoons	16	int
cups	1	int
result	0	int

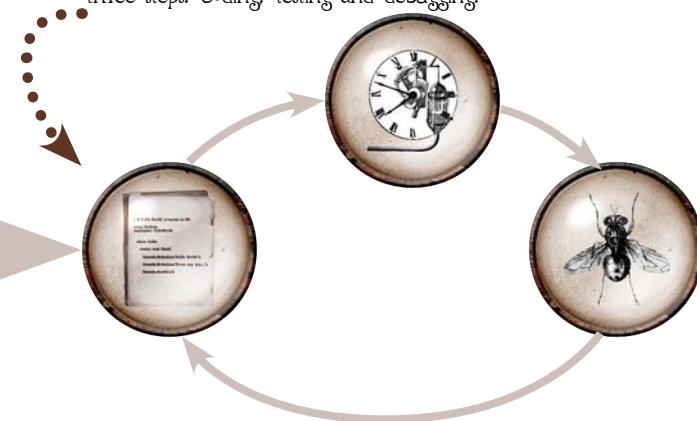
that Visual Studio provides for finding and fixing errors, and then we'll look at the tools for deploying your applications once they're finished.



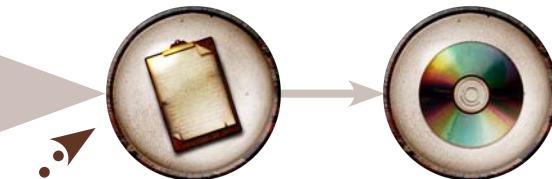
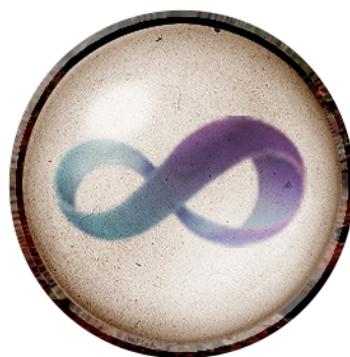


FITTING IT IN...

Remember the development process diagram from Chapter 1? In this chapter, we're going to expand on Step 3, creating the source code, and Step 4, turning the source into an executable.



We'll expand the development process to three steps: coding, testing and debugging.



And we'll expand deployment into two steps: specifying the build configuration and deployment.



TASK LIST

In the last chapter we looked at how to make the Visual Studio IDE suit your way of working and the support that Intellisense and the Code Editor provide while you're writing code. In this chapter, we'll complete our examination of the Visual Studio IDE by looking at two other functions you'll perform as a developer: finding & fixing errors in your code, and deploying finished applications.



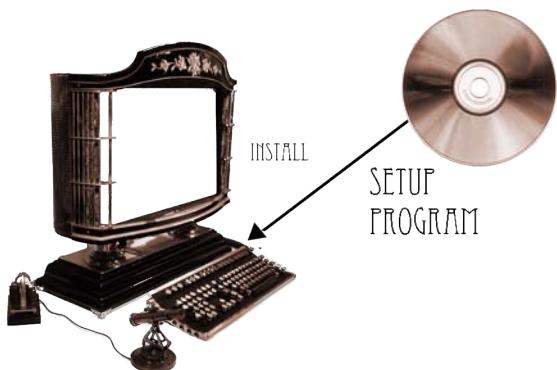
ERRORS AND EXCEPTIONS

You don't deploy until you're as sure as you can be that the application is working as it's designed to. (Or at least you shouldn't.) So we'll begin by examining the tools Visual Studio provides for tracking down and fixing those pesky bugs.

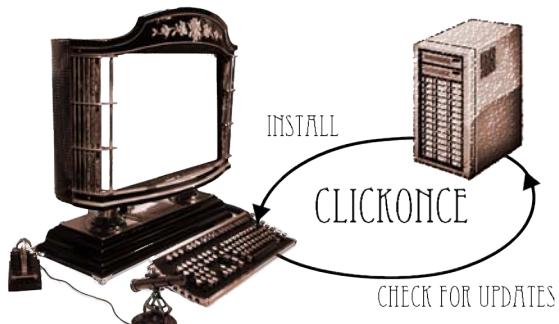


DEPLOYMENT

After you've fixed the mistakes you know about, you need to let your application's users have a crack at finding the rest, so after we've examined the testing and debugging process, we'll look at two of the ways you can get your finished application out to the people who will be using it, and the deployment housekeeping that's necessary before you start handing it out.



We'll look at the two primary methods of deploying .NET applications: ClickOnce and Setup programs.





ERRORS & EXCEPTIONS

Under the general heading of "things that can go wrong with your program", there are two that you need to worry about as part of the development process.



BUGS

BUGS are the things you need to try to find before you deploy. We'll concentrate on fixing bugs (debugging) in this chapter. There are three types of bugs that you need to worry about:



SYNTAX ERRORS are the typos of the programming world. They're easier to avoid now that we have Intellisense, but they can still happen.

Messagebox("Hello")

The editor underlines syntax errors (here "Messagebox" should be "MessageBox") with a squiggly red line.



COMPILATION ERRORS are halfway between syntax errors and logical errors. Just as the spell-checker in a word processor won't know that you meant to type "through", not "though", but the grammar-checker probably will, the editor won't catch these errors, but the compiler will.



LOGICAL ERRORS occur when the computer does what we told it to, but we didn't tell it what we intended. These kind of errors are a *lot* easier to make than you might expect and can be hard to track down.



EXCEPTIONS

EXCEPTIONS are things that happen at runtime, like a hardware error, that are outside your control. (But failing to handle them is a bug.) We'll discuss exception handling in Chapter 7.



LET'S SQUASH SOME BUGS

We've seen that Visual Studio provides several tools to help you avoid bugs. It'll probably come as no surprise that it also helps you track them down. So let's explore them by creating a silly little program that we can break at our pleasure.

1

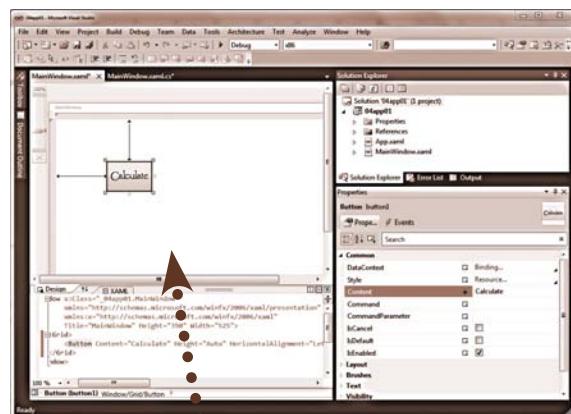
CREATE A NEW WPF APPLICATION SOLUTION

You can call it anything you like, but SillyApplication might be a good choice. Peek back to Chapter 2 if you've forgotten the details for doing this.

2

CREATE A BUTTON

When Visual Studio finishes creating the Solution and Project for you, it will open up the WPF/XAML Design and Code Editor for the MainWindow. Drag a Button onto the form from the Toolbox and set its Content property to "Calculate". You can set any other properties you like, of course.



Here's my version of
the window

3

CREATE THE BUTTON1_CLICK EVENT

Double-click the button or choose Click in the Events tab of the Properties window to create code that will execute when the button gets clicked. Visual Studio will write all the standard code and bring the Code Editor tab to the foreground.

BUG SQUASHING, CONTINUED...

One of the applications our chefs, Neil & Gordon, need us to write is a kitchen calculator. If you cook and you don't use the metric system, you know how tedious it can be to translate between cups and tablespoons and teaspoons.

We can't make a real calculator yet—you'll need to work through a few more chapters first—but we can make a start by simply displaying the number of tablespoons in a cup. (I did warn you that this was a pretty silly application. Don't worry; it's complicated enough to break.)

4

WRITE SOME CODE

Tell Visual Studio to add an event handler for the button by double-clicking it on the design surface. Add the following code to the `Button1_Click` event. I've given you a sneak peak at what the code means, but don't worry about it. You'll find out what all this stuff means in the next chapter.

These are
VARIABLES. They're
placeholders for values
that can change.

This is an EXPRESSION. In English, it means something like "set the value of result to the current value of cups multiplied by the current value of tablespoons". In other words, it sets the variable named `result` to the number of tablespoons in the specified `cups`.

In these two lines
we're assigning values
to the variables.

```
Dim tablespoons As Int32
Dim cups As Int32
Dim result As Int32

cups = 1
tablespoons = 16
result = cups * tablespoons

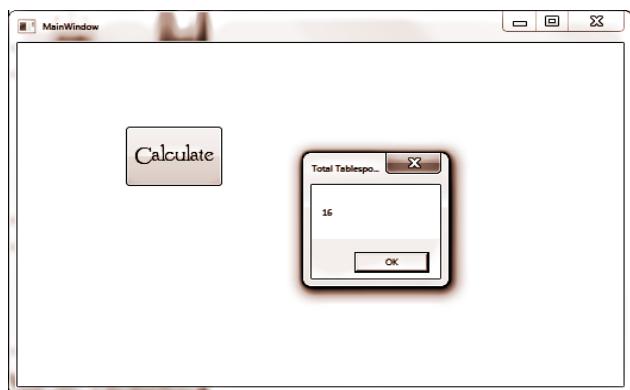
MessageBox.Show(result.ToString(), "Total Tablespoons")
```

As you'll find out in the next chapter, Visual Basic is really picky
about whether a variable is a number or text or something else.
Since the `tablespoons` variable contains a number, we have to call
`ToString()` to turn it into text.

5

RUN IT

Once you've entered the code for the click event, press F5 to run the application, and, once the window displays, click the Calculate button. You should see a screen like this one.



6

BREAK IT!

Now that we've got our application working, let's introduce some bugs so we can see what Visual Studio does about them.

SYNTAX & COMPILATION ERRORS

Remember that syntax errors are basically typos, and compilation errors are like grammatical errors. In truth, there aren't many errors that the Editor won't catch, and they're treated the same way, so you don't need to worry too much about the difference. These are probably the easiest kind of bug to spot and fix, so let's start by introducing one into our code. Change the word "MessageBox" to "MsgBox". Visual Studio will underline the word with a blue squiggly line, and if you hover the mouse over it, you'll see an Intellisense message:



What happens if you press F5 to run the application without correcting the error?

MORE BUGS

When you try to compile and run an application that has either syntax or compilation errors, Visual Studio displays this error box. You'll hardly ever want to say yes here...



THE ERROR LIST

Visual Studio will display both syntax and compilation errors in the Error List window as soon as it spots them. By default, the Error List displays at the bottom of the Visual Studio window. If you don't see it at all, choose Error List from the View menu or press Ctrl+W, E.

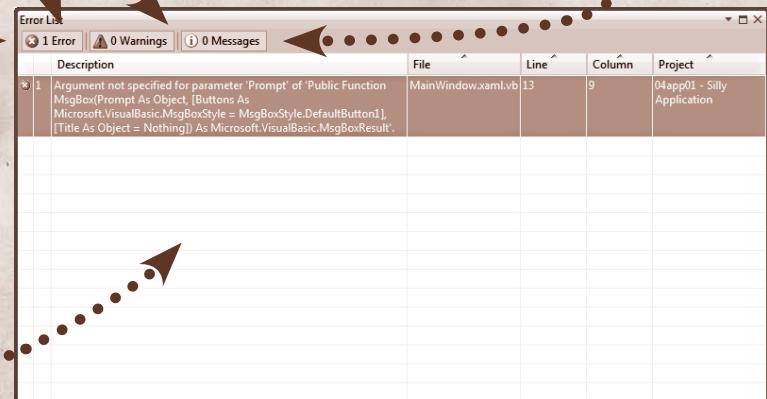
Errors are problems that prevent the code from running. You need to fix them right away. •••••

Warnings are bugs waiting to happen. They won't stop the code from running, and they're usually a result of "haven't gotten there yet", but you'll want to fix them up before you deploy.

Messages are, well, everything else. You won't see a lot of them.

These are actually buttons. •••••
Click them to turn the display of that category of item on and off.

You can double-click on a line in the Error List and Visual Studio will take you right to it, opening the file if necessary. •••••



BREAKING ON ERRORS

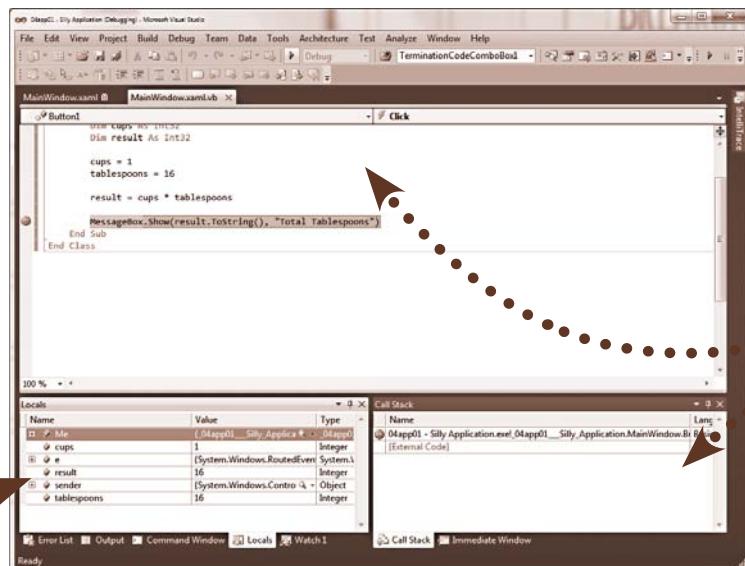
Double-clicking a line in the Error List window will take you to the line (and even the position within the line) where Visual Studio found the error. But one of the infuriating things about programming is that Visual Studio may not find the error where the error is. How can that happen? Well, have you ever gone to the store without your wallet? You discover the problem at the store, but the problem actually happened back at home. It works the same way with Visual Studio—it can't know that you need your wallet until you go to pay for something.

When something like that happens, the Error List may not be enough to help you find and

fix the problem. And if you have a logical error, Visual Studio probably won't find an error at all. To track down complex problems, you'll need to examine what's going on inside the program while it's running. The IDE lets you do that in BREAK MODE. There are a couple of ways to enter break mode, but you'll usually get there by setting a breakpoint, which is just a signal to tell Visual Studio "stop here and let me see what's going on."

Once you set a breakpoint (we'll set one on the next page) and run the application, it will execute normally until the debugger hits the line with the breakpoint. At that point, Visual Studio will enter break mode and, by default, display these windows:

The Locals window shows you the value of all the things you've declared in your application at this point.



The Watch window is like the Locals window, but it shows the value of specific variables you choose, and they don't have to be local to the procedure.

This is the standard editor, but notice the little lock symbol on some of the tabs. You can't change them, and there are limitations on the changes you can make to the current file without restarting the application.

It's common in any application for procedures to call other procedures. The Call Stack window shows you how you got where you are.

The Immediate window lets you execute statements or evaluate expressions while in Break Mode.



GIVE BREAK MODE A TRY...

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), and we'll be using it throughout the rest of the book, but let's take a quick tour of what you have in store...

1

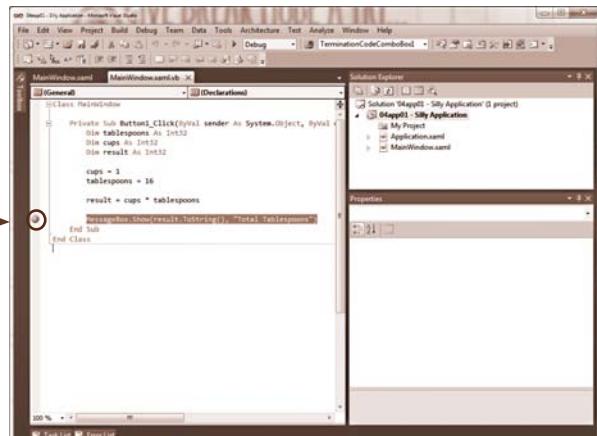
Correct the call to a MessageBox in the Button1_Click event handler:

```
MessageBox.Show(result.ToString(), "Total Tablespoons")
```

2

Set a breakpoint on the MessageBox line by clicking in the grey bar on the left side of the editor window.

Click here to set the breakpoint.
Visual Studio will display a round
circle and highlights the line.



ON YOUR OWN

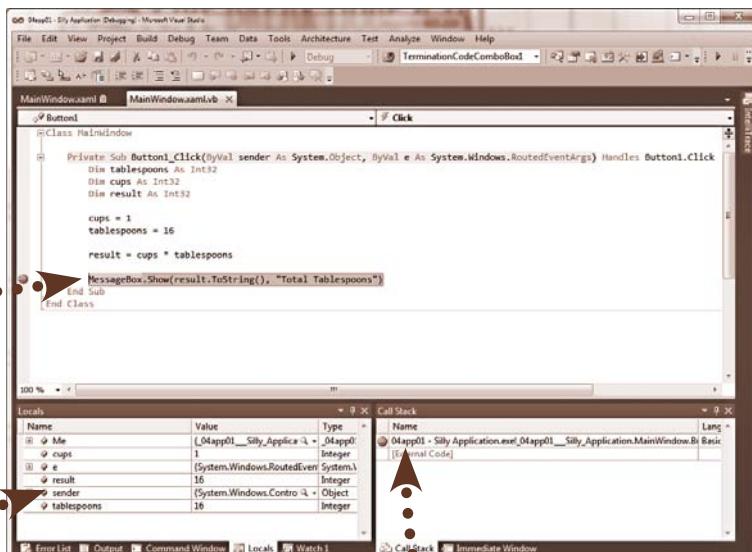
There are three other ways to set a simple breakpoint. Can you find them? (Hint: One is a keyboard shortcut; you'll see it once you find the right menu.)

3

Run the application by pressing F5, and click the Calculate button. When Visual Studio gets to the line where you set the breakpoint, it will enter break mode:

The editor shows the line that's about to be executed in yellow.

The Locals window shows the value of all the variables in the procedure.



The Call Stack is populated.

4

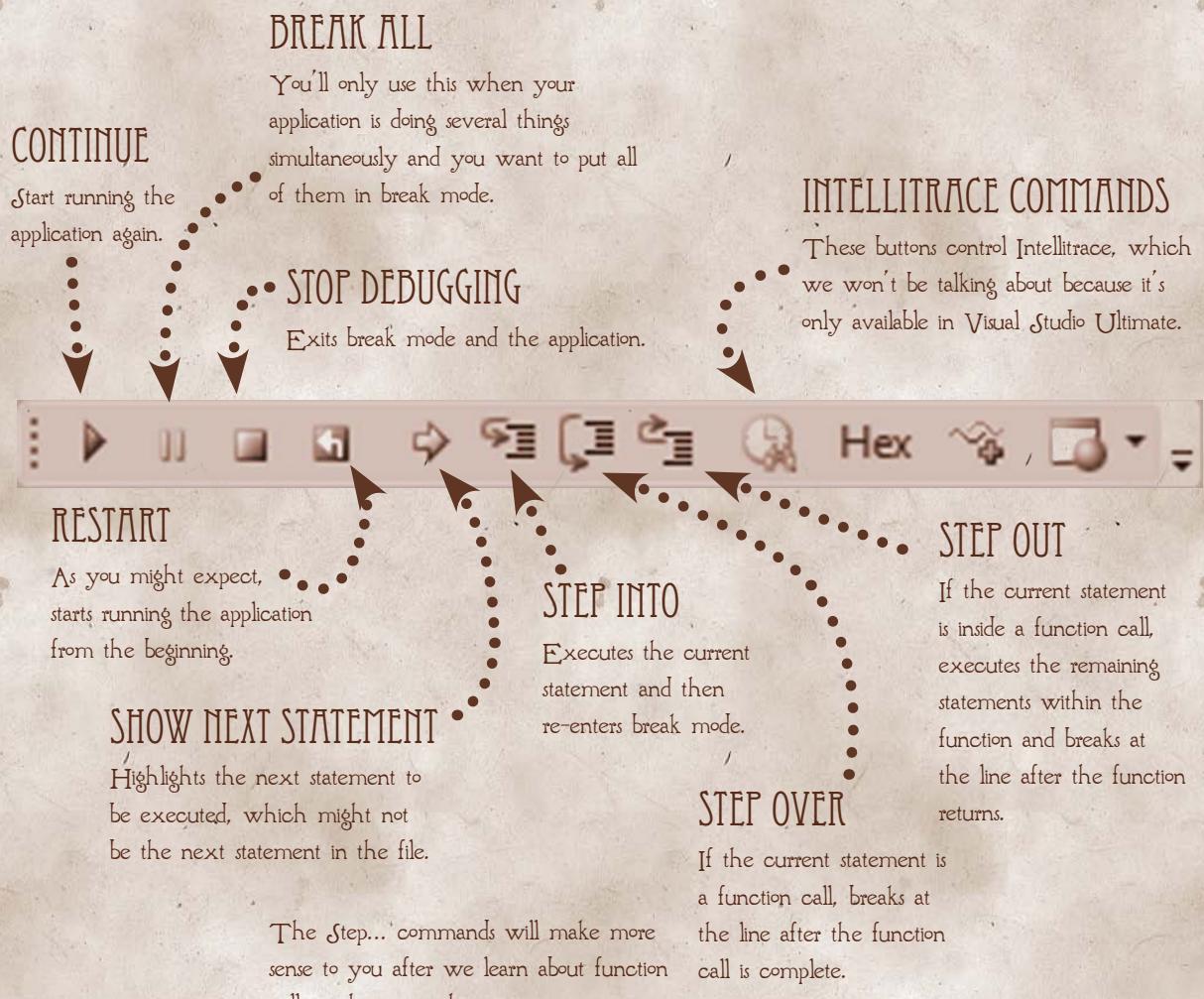
In break mode, the Visual Studio debugger will display the value of any variable as a Data Tip; just hover the mouse over the variable.

result = cups * tablespoons
cups | 1

To keep the Data Tip open after you move the mouse, click the pushpin.

BREAK MODE COMMANDS

The commands that are available to you in break mode are available from the Debug menu, from the context menu that's displayed when you right-click a line, or from the Debug toolbar:



TRYING THE COMMANDS

Let's carry on with our example by trying out the commands on the Break Mode toolbar.

5 Stop the application if it's still running, and then remove the breakpoint on the MessageBox call (just click in the grey bar again). Set a new breakpoint on the line that sets the value of cups.

6 Now run the application again, and click the Calculate button. Visual Studio will enter break mode at the new line. Notice that all the variables now have a value of 0, because the lines of code that set their values haven't yet been executed.

7 Click the Step Into button, press F11, or choose Step Into from the Debug menu to execute the current statement. Visual Studio will re-enter break mode at the next one.

Cups has a value now because that line has been executed.



```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.Windows.RoutedEventArgs) Handles Button1.Click
    Dim tablespoons As Int32
    Dim cups As Int32
    Dim result As Int32
    cups = 16
    result = cups * tablespoons
    MessageBox.Show(result.ToString(), "Total Tablespoons")
End Sub
```

Name	Type	Value
Me	System.Windows.Controls.Control	0x0
cups	Integer	1
result	Integer	0
sender	Object	0
tablespoons	Integer	0

Name	Lang
0x0	[External Code]

TAKE A BREAK

The debugging tools are pretty easy to use, which is a good thing since you need to concentrate on finding the problem, not using the tools. Now that we've explored them, why don't you take a break before completing the Review and moving on to deployment?





REVIEW

Can you match each of these images to its description?



STEP OVER

Break at the next line, no matter where it is.



STEP INTO

Run a function without stepping through it.



STEP OUT

Step into a function call.



SYNTAX ERROR

Visual Studio won't run the application.



LOGICAL ERROR

The editor is showing a squiggly line.



COMPILATION ERROR

The application runs, but it doesn't give the right results.



DEPLOYMENT

Your application may be a better mousetrap, but people aren't going to beat a path to your door unless they can install it. In a corporate environment deployment can (and usually does) involve a whole lot more than simply installing an application; it might include implementing services like a database, preparing policies documentation, and running training courses. But from a purely programming perspective, deployment is usually restricted to preparing a setup program. As always, Visual Studio offers you some choices:



CLICKONCE

ClickOnce is a special kind of deployment for .NET Framework applications. Although it can be distributed on disk like a Setup program, ClickOnce applications are typically installed from a network share. Because they can be installed without administrator-level permissions and can be configured to automatically search for and install updates, ClickOnce applications are ideal for applications that are frequently updated or extended in a corporate environment.



SETUP PROJECTS

If you've used Windows very much, you've probably used a lot of setup programs. In fact, you used one when you installed Visual Studio. Setup programs are the classic way to distribute an application. Setup programs don't require network connectivity, as ClickOnce applications can. That can make distribution simpler and is an obvious choice for retail application. However, the user will need administrator-level permission to install the application, and updates need user intervention.



MAKE A NOTE

Setup projects are only available in Visual Studio Basic edition and above. If you're running Visual Studio Express, you won't be able to do the exercises in the second half of this section. Just read through them to get a sense for what you're missing. (You might decide they're not something you care about.)



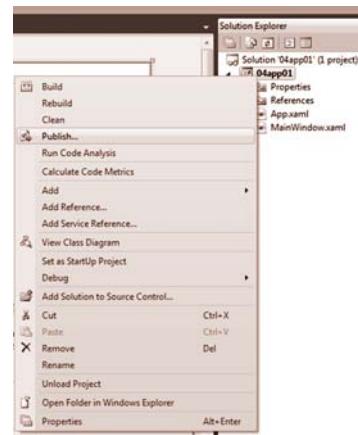
PUBLISHING WITH CLICKONCE

ClickOnce deployment isn't just easy for the user; it's easy for the programmer, as well. To see just how easy, let's walk through deploying our silly little application using ClickOnce.



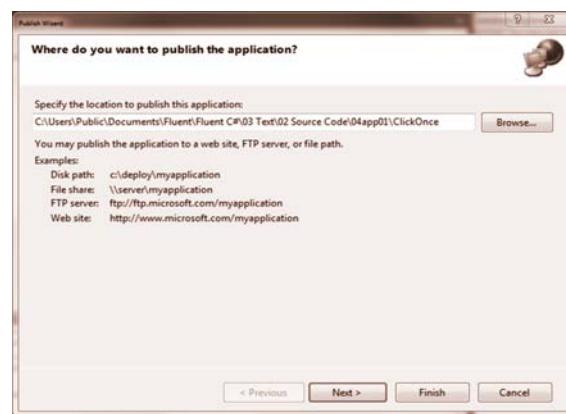
Right-click the project in the Solution Explorer, and choose Publish from the context menu, or choose Publish from the Build menu.

(If you're still running the debugger from the last exercise, you'll have to click Stop Debugging first.)



After you choose Publish, Visual Studio will start the ClickOnce Publishing Wizard. The first page asks for the location where you want the application to be published. In a corporate environment, this would be the network share from which users install the application. For now, just put it in a folder inside your main Solution folder. (Click the Browse button to open a standard File dialog that will help you navigate to the publishing location.)

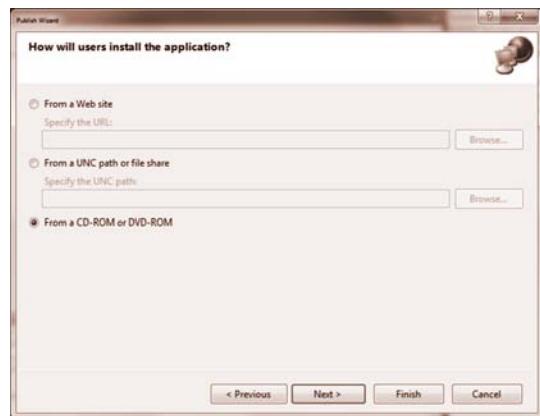
Once you've chosen the location, click the Next button.



3

Next, the wizard will ask you how you want users to install the application. For now, choose the last option, which creates a self-contained installation, but you could choose a Web URL or a network share on this page.

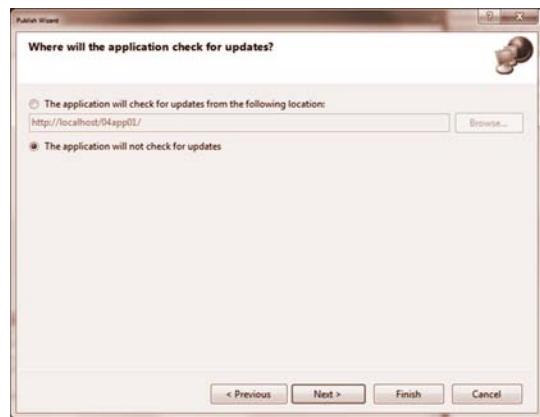
After you've made your choice, click the Next button.



4

On this page, the wizard asks for the location to which you would post updates. You would typically choose a folder different from the one to which you initially publish, but we're not going to be updating our silly little application, so choose the second option, not to check for updates.

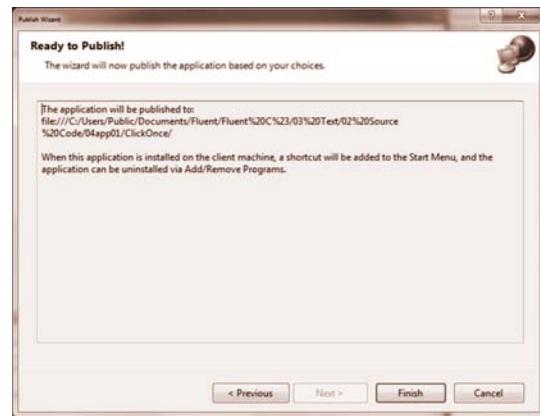
After you've made your choice, click the Next button.



5

And that's it. Just click Finish on this last page of the wizard, and Visual Studio will create all the files necessary to install the application and move them to the locations you've specified. How easy was that?

If you want to test it, just navigate to the location you specified in Step 2 and click the setup program. You can uninstall the application from the Control Panel in the normal way.





CREATE A SETUP PROGRAM

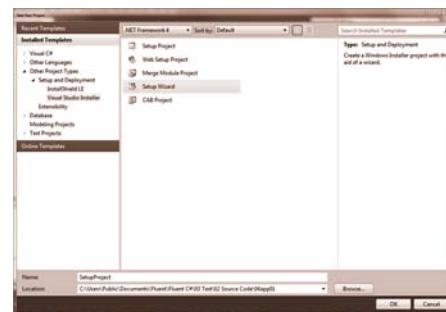
It's hard to imagine a deployment technique that's easier than the ClickOnce Publish wizard. Creating a Setup Project is a little more complicated, but not much, and only because it's far more flexible and there's a Setup Wizard that will do most of the basic configuration for you.

Remember, setup projects aren't available in the Express edition of Visual Studio. But if you're running Visual Studio Basic or above, let's try it out:



Add a new project to the solution, either from the context menu in Solution Explorer, or from File menu. In the New Project dialog, select Setup Wizard from the Other Project Types, Setup and Deployment category. You can call the project anything you like; I've called it SetupProject.

Once you've specified the project name, click the OK button.



Once you've clicked OK, Visual Studio will display the first page of the Setup Project Wizard.

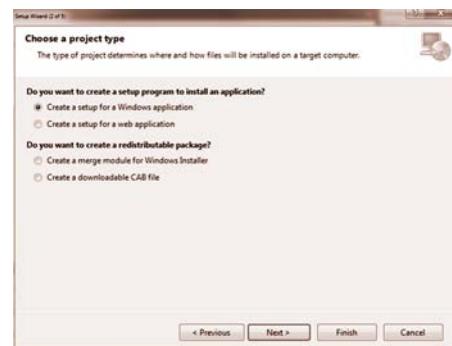
There's nothing to do here, so just click the Next button.



3

On this page of the Setup Project Wizard, you need to specify the type of application you'll be deploying. We're creating a Windows application, so the default is correct. Web applications are used in ASP.NET, and redistributable packages are used for application components. We'll talk about those a little later.

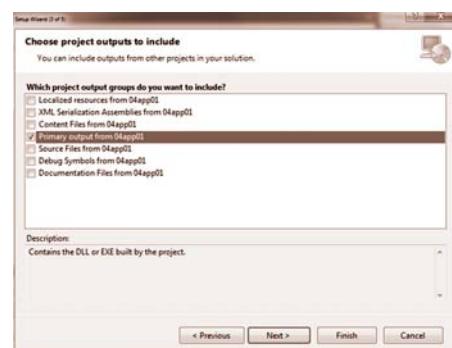
After checking that “Create a setup for a Windows application” is selected, click the Next button.



4

On this page, you specify what compiled components you want to deploy. Ours is a simple project, so you only need to check the Primary output box, as shown.

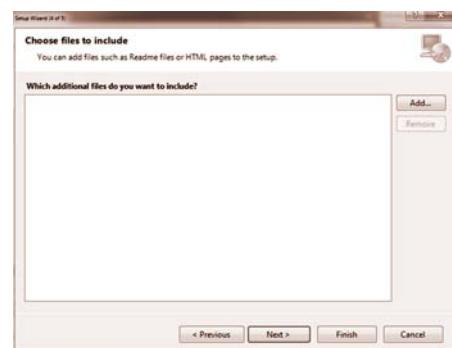
Once you've checked the correct box, click the Next button.



5

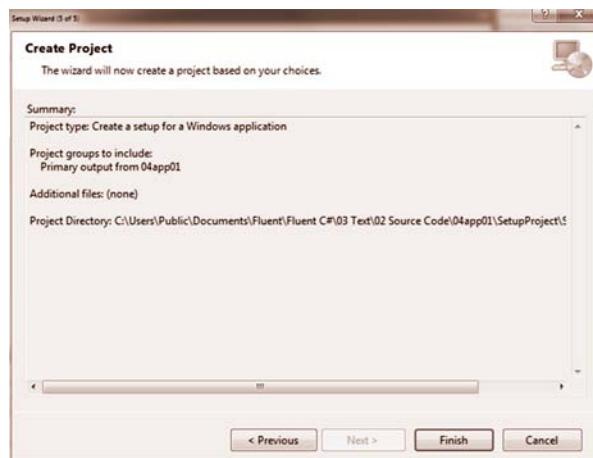
This page lets you include files that aren't part of the solution in the setup program. For example, you might have licensed a typeface for distribution with your application. You would specify it here by clicking the Add button, which opens a standard File dialog.

We're not including anything tricky, so just click the Next button, and move on to the next page (of the wizard, and the book!)



6

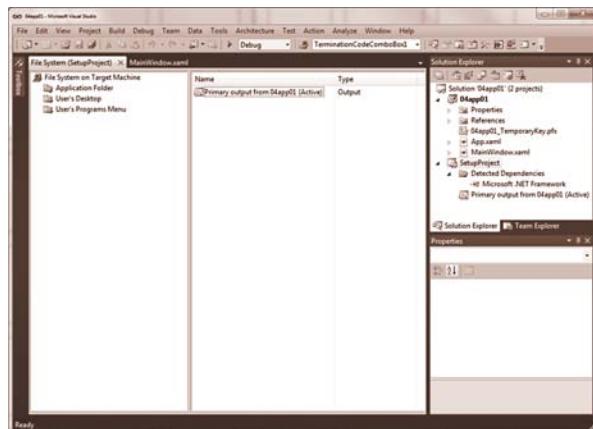
The last page of the wizard just confirms the choices you've made and gives you a chance to go back and fix any problems. If everything's okay, click Finish and Visual Studio will create the Project for you.



7

Once the Project is created, it will display in the Solution Explorer, and Visual Studio will open the File System Designer in the IDE.

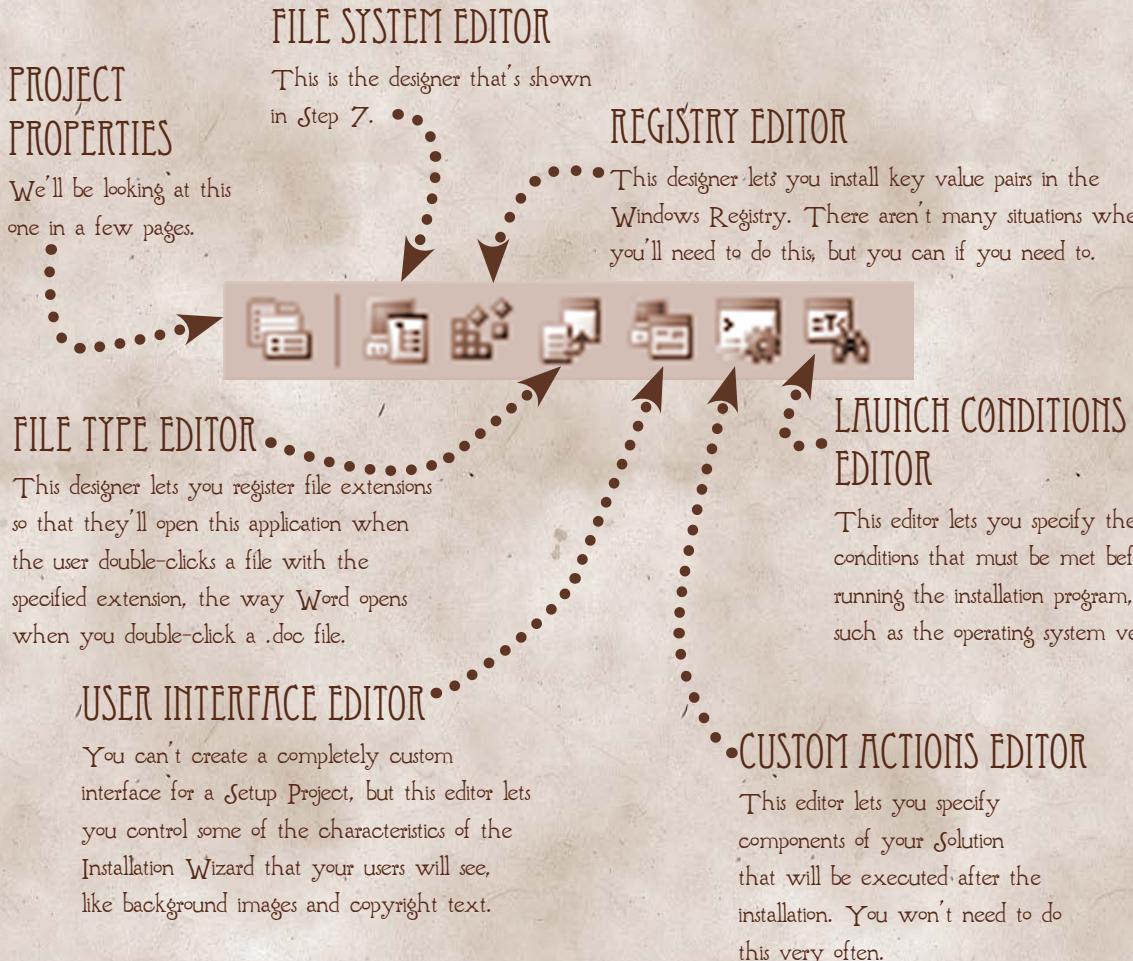
The File System Designer lets you specify the files that are placed on the user's system when they install the application. In this image, I have Application Folder selected, and you can see that the only thing to be placed in that folder will be the Primary output, which in this case is the application executable.



At this point, all you have to do is build the Setup Project, and you're ready to deploy the application by simply copying the contents of the output directory to a DVD or network share. Not really much more difficult than using the Publish wizard, was it? But as is often the case with programming, if you want more control you have to deal with more complexity, and in this case the added complexity of a Setup Project buys you the power to configure precisely how the Setup program will work. (Well, within some limitations.)

SETUP DESIGNERS

Like any Project in Visual Studio, a Setup Project has a number of custom designers that help you build various components. As we've seen, the File System Designer is displayed when Visual Studio first creates the project. The easiest way to display the other designers is from the toolbar that's displayed in the Solution Explorer when the Project is selected.





ADD A SHORTCUT

The Setup Wizard does a pretty good job of setting up a basic Setup Project, but there's one thing it doesn't do that you'll almost certainly want to: add a shortcut to the user's Desktop or Programs menu. Let's add that to our Project:

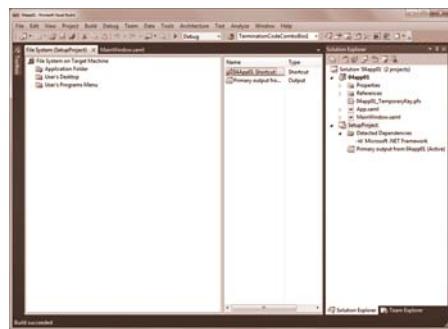
1

In the File System Designer, select Application Folder in the left pane, and right-click on the "Primary output..." item in the right pane. Choose "Create Shortcut..." from the context menu. (The option is also available from the Action menu, but only when the Primary output item is selected.)



2

Visual Studio will add the shortcut to the file list in the right pane. You'll probably want to rename it, as I have. You can do that by clicking on the name in the designer or by changing the (Name) property in the Properties window.



3

In the File System Editor, drag the newly-named shortcut from the Application folder to the User's Desktop folder. (Just drag it from the right pane onto the folder icon in the left pane.)

That's it. Now your setup program will put a shortcut to your application on the desktop when it's installed.



TRY IT OUT

Probably time to test our setup program, don't you think? Here's what you need to do...

- 1** Build the Solution.
- 2** Navigate to the folder where you put the Setup Project. If you accepted the defaults, it will be inside the Solution folder.
- 3** Double-click either of the files that you'll find inside the Debug folder, and your application will install.

You can also run the Setup project from within Visual Studio. It works just fine.



ON YOUR OWN

The setup program created by the Setup Wizard is perfectly functional, but it's not very attractive. If you like, play around with the User Interface editor (you can't break anything). To change some of the text and maybe add a bitmap. You'll find that you can't do too much, but then, the average installation program doesn't need to do too much, either!

ONE LAST THING...

Did you notice that the Setup program was in a folder named Debug? Did you wonder why? It has to do with the build configuration. A build configuration is a group of settings that tell the compiler how to handle the compilation process. You can create your own configurations if you need fine control, but you can usually use one of the two default configurations defined by Visual Studio.

DEBUG BUILD CONFIGURATION

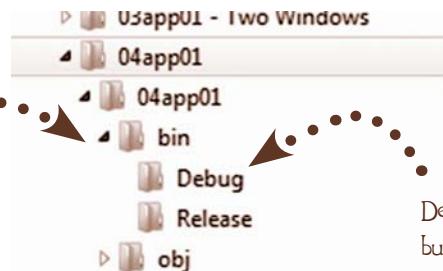


When you create a new project, this configuration will be chosen by default. The compiler will include all the symbolic information the debugger needs to help you track down and squash those pesky things. It also compiles your code more or less the way you wrote it, without optimizing anything.

RELEASE BUILD CONFIGURATION



Once you've fixed all the bugs you can find and you're ready to deploy your application, you should switch to the Release configuration. When the compiler builds a release version, it omits all the stuff the debugger needs, and it optimizes your code to make it smaller, faster and more efficient. (But I'll warn you now, you probably won't notice the speed improvement.)

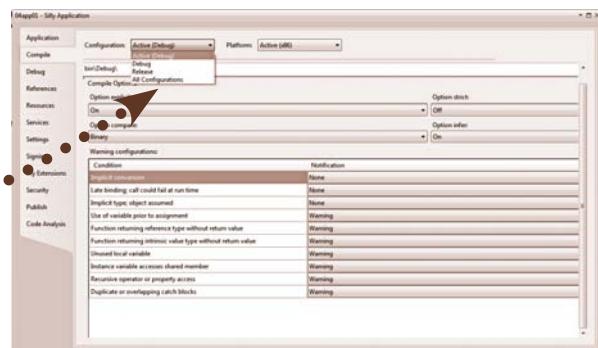


Visual Studio will create a folder for each build configuration. You'll find the compiled output in the folder of the active build.

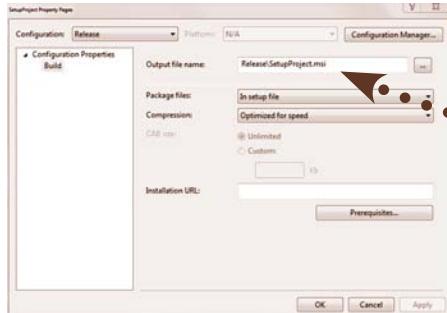
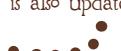
Debug and Release are the build configurations you get by default.

SETTING THE CONFIGURATION

You set the build configuration on the Project Designer or Project Properties dialog. Just select the configuration you want from the list.



For a Setup project, the Output Directory is also updated.



TAKE A BREAK

There's a lot more to learn about deployment and setup projects (there's always a lot more to learn), but we've seen enough to handle 90% of your applications, so take a break before we finish up the chapter.





REVIEW

Before we move on to the nitty-gritty of VB syntax, let's take a moment to review what we've learned about deployment methods.

THE PROS

List three things that are good about each deployment method:

ClickOnce

- ①
- ②
- ③

Setup Project

THE CONS

List three drawbacks of each deployment method:

ClickOnce

- ①
- ②
- ③

Setup Project

AND THE WINNER IS...

Pick a couple of examples from the list of applications you'd like to write, and choose the best deployment method for each.

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

①

②

③

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



THE LANGUAGE, PART 1: NOUNS



5

Here we are five chapters into a book that has Visual Basic in the title, and you discover that there are only three chapters on the language itself. False advertising? No, not really. Visual Basic is part of the .NET Framework. The .NET Framework, and in particular the Framework Class Library (FCL), is big and complex. The language itself, the mechanics of the syntax, is the easy bit.

And the best part is that you already know most of what you need. After all, Visual Basic is just a dialect of English with very strict syntax rules and a limited vocabulary, and you already speak English.

Just like spoken English, Visual Basic has nouns and verbs, sentences and paragraphs. But it doesn't have tenses or irregular verbs, so you never have to worry about anything as complicated as "I am but you are" or "he would be were he not". And just like spoken English, once you learn the basic grammatical rules, you'll be able to concentrate on what you want to say, and not the mechanics of how to say it.

In this chapter we'll cover Visual Basic sentences, paragraphs and nouns and also look at how you modify them, in the same way you use modifiers like adjectives and adverbs in spoken English. We'll look at expressions and commands, the Visual Basic equivalent of verbs, in the next two chapters. And then we'll spend the rest of the book learning the vocabulary you need to fit into this very simple syntax.



LANGUAGE ELEMENTS

You've probably seen diagrams of the "Parts of Speech" in English. Here's the equivalent for the Visual Basic dialect.

LANGUAGE ELEMENTS

Comments are ignored by the compiler but really important for the programmer.

COMMENTS

DIRECTIVES

These are instructions to the compiler and the .NET Framework.

Visual Basic statements are the equivalent of sentences and paragraphs. We'll look at the syntax of both forms in this chapter.

STATEMENTS

We'll look at these statement components in the next two chapters.

These are the nouns of the language.

DECLARATIONS

COMMANDS

EXPRESSIONS



TASK LIST

In this chapter, we'll begin our examination of Visual Basic syntax with the basic syntax of a statement, one specific kind of statement (the kind that creates a noun), the comments that help us understand what we're doing, and the directives and attributes that tell the compiler and the .NET Framework (as opposed to the computer) what to do.

STATEMENTS

do it.

Like sentences and paragraphs, statements are the basic building block of a Visual Basic program, so we'll start by looking at how you build them.

DECLARED ELEMENTS



It's pretty unusual to have a sentence without a noun in English ("Think!"), and it's almost impossible to create a statement that doesn't use one in Visual Basic, so after we learn basic statement syntax, we'll examine the type of statement, called a **DECLARATION**, that creates the nouns you'll be working with.

COMMENTS

'um...

It's all very well for me to talk about Visual Basic being a dialect of English, but reading a program isn't like reading a book. **COMMENTS** are how you document what you're doing in your code, so you'll understand it when you come back six months (or six hours) later.

DIRECTIVES

#

Most of the code you write will involve telling the computer what to do, but in the last part of the chapter you'll find out how you can also boss around the compiler using **DIRECTIVES**.



STATEMENT SYNTAX

In regular English, a sentence is some combination of nouns, verbs and modifiers set off with a period, exclamation point or question mark.

The Visual Basic equivalent of a sentence is a STATEMENT: some combination of declared elements, commands and expressions treated as a single unit of work.

IN ENGLISH

SIMPLE STATEMENTS.

The period marks the end of the sentence. In Visual Basic, a statement usually ends at the end of the line.

Something something.

IN VISUAL BASIC

something something

MULTI-LINE STATEMENTS.

Both a sentence in English and a statement in Visual Basic can extend over multiple lines, but in Visual Basic you usually have to use the LINE CONTINUATION CHARACTER ("_") to tell the compiler to keep reading.

Something
something.

something
something

An underscore (capital-dash on your keyboard) is the line continuation character.

MULTI-STATEMENT LINES.

In English, you can put multiple sentences on a single line if you separate them with a period. In Visual Basic, you can put multiple statements on the same line if you separate them with a colon.

Something something. Something.

The colon lets you combine statements on a single line.

something something(:)something

PARAGRAPHS.

In English, a paragraph is set off by some typographical convention like an indent or an extra space. In Visual Basic, the equivalent of a paragraph is a pair of statements: *Something...* *End Something*. Not all statements support this syntax. We'll see the ones that do as we go along.

Something something.
Something.
Something.

• • • "Something" and "End Something" have to match.

Something
Something
End Something.



PUT ON YOUR THINKING HAT

Here's some moderately complex code. Don't worry about what it does; just count the number of statements it contains.

```
Imports System.IO
```

```
Namespace FluentLearning
```

```
    Public Class MyClass
```

```
        Inherits MyOtherClass
```

```
        Implements IPrintable
```

```
    Private myArray _
```

```
        as ArrayList
```

```
    Sub New()
```

```
        myArray = New ArrayList()
```

```
        For x = 1 to 100
```

```
            myArray.Add(x)
```

```
        Next x
```

```
        myArray(5) = 325
```

```
        For Each y As Int32 In myArray
```

```
            Console.WriteLine(myArray(y))
```

```
        Next y
```

```
    End Sub 'New
```

```
End Class
```

```
End Namespace
```



MAKE A NOTE

There are some exceptions to the rule that multi-line statements need a line continuation character, and we'll look at them as we go along. But it's never wrong to use one, so you don't have to memorize the exceptions.



HOW'D YOU DO?

There are 15 statements in this code.

```
1 Imports System.IO  
2 Namespace FluentLearning  
3 Public Class MyClass  
4     Inherits MyOtherClass  
5     Implements IPrintable  
6     Private myArray _  
        as ArrayList  
7     Sub New()  
8         myArray = New ArrayList()  
9         For x = 1 to 100  
10            myArray.Add(x)  
11        Next x  
12        myArray(5) = 325  
13        For Each y As Int32 In myArray  
14            Console.WriteLine(myArray(y))  
15        Next y  
16    End Sub 'New  
17 End Class  
18 End Namespace
```

Did you catch the line continuation character?
These two lines are one statement.

Extra Credit: Read through the code. You'll know what all this means in a couple chapters, but can you make a guess at it now?

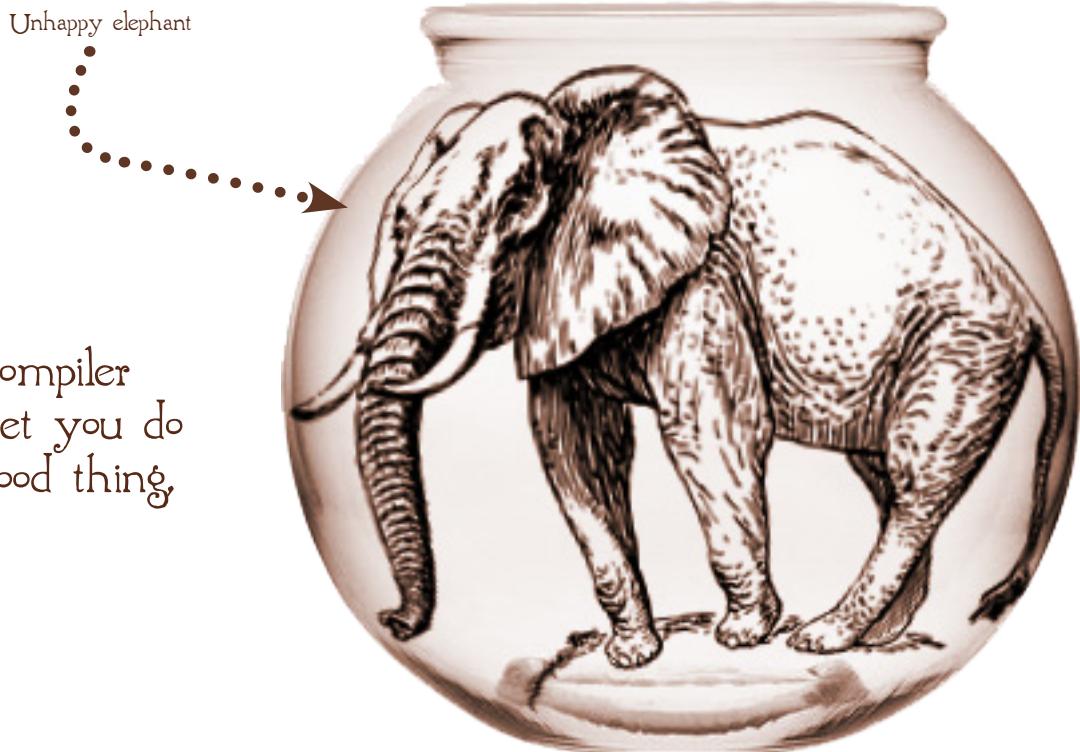


DECLARED ELEMENTS

You wouldn't ask your pet elephant to live in a goldfish bowl, would you? I certainly hope not. (The world doesn't need more unhappy elephants.) Visual Basic won't let you do anything so unkind. DECLARED ELEMENTS are the nouns of Visual Basic, the things you will be manipulating in your code. When you create one, you have to tell the compiler what type of information it's going to hold, and it won't let you put an elephant in a goldfish-holder, or vice versa.

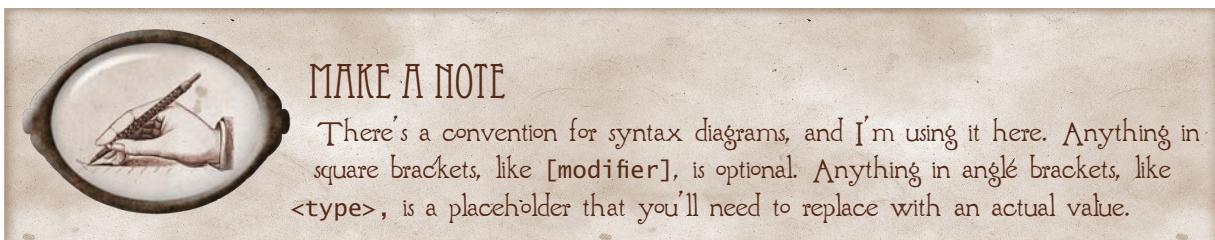
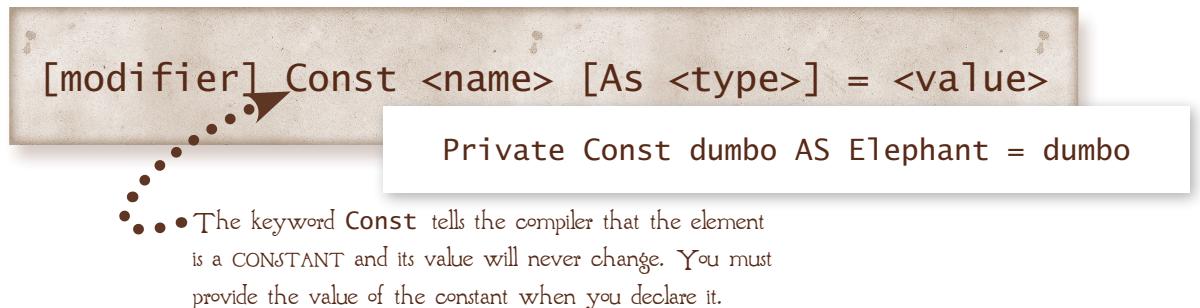
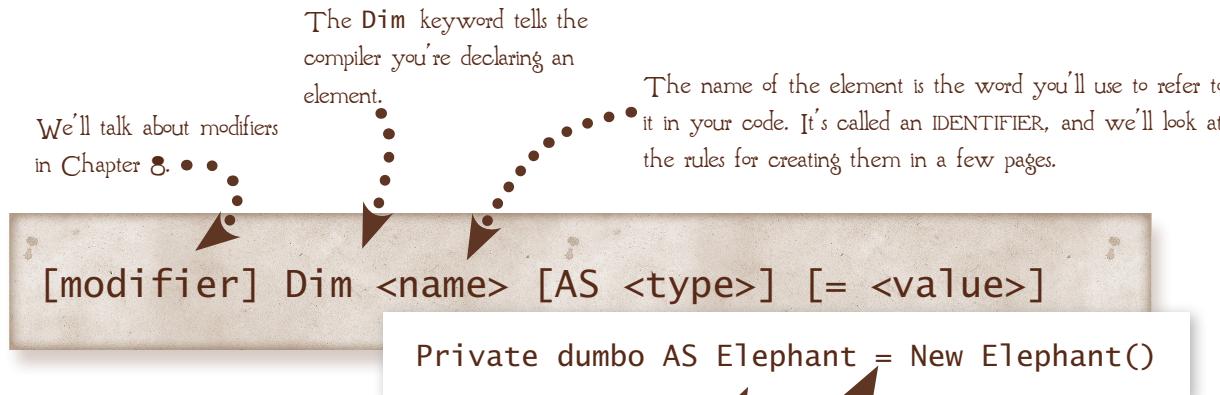
There are two types of declared elements in Visual Basic: VARIABLES that can contain different elephants (but only elephants) over the course of the application, and CONSTANTS that always have to hold the same elephant. (Of course, you're probably not going to be working with elephants. We'll examine the .NET Framework type system, which defines the common data types and lets you define new ones, in Chapters 10 and 11.)

The compiler
won't let you do
this. Good thing,
too!



DECLARING ELEMENTS

With very few exceptions, everything you do in VB is a statement or part of a statement. To create a declared element, you use a particular kind of statement called a declaration statement. (No surprise there, right?)





PUT ON YOUR THINKING HAT

Time to create some declaration statements (and find some problems).

The .NET type `Char` holds a single character. Write the statement that declares an element named `theChar` that holds a `Char`:

Write the statement that declares an element named `myChar` that contains a `Char` and has the initial value of 'A':

An `Int32` holds an integer (whole number) between -2,147,483,648 and 2,147,483,647. Write the statement that declares an `Int32` element named `myConst` with the constant value of 5000:

What's wrong with the following declaration statements?

```
Const myVar = 5
```

```
Const myConstChar As Char
```

```
Dim theNumber
```

```
Const myChar As Char = 5
```

```
MyChar As Char = 5
```



HOW'D YOU DO?

Please don't worry if you didn't get all these right. The point here is to learn. You're not supposed to already know the answers. (Besides, one of the questions was a trick.)

The .NET type `Char` holds a single character. Write the statement that declares an element named `theChar` that holds a `Char`:

`Dim theChar As Char`

Write the statement that declares an element named `myChar` that contains a `Char` and has the initial value of 'A':

`myChar As Char = 'A';`

This was the trick question. You have to enclose strings in double quotes and characters in single quotes so that the compiler can distinguish them from identifiers.

An `Int32` holds an integer (whole number) between -2,147,483,648 and 2,147,483,647. Write the statement that declares an `Int32` element named `myConst` with the constant value of 5000:

`Const myConst AS Int32 = 5000;`

What's wrong with the following declaration statements?

`Const myVar = 5`

Nothing. This is valid, but a bad idea in most situations.

`Const myConstChar As Char`

You have to provide a value when you declare a constant

`Dim theNumber`

The type of the variable isn't defined

`Const myChar As Char = 5`

A `Char` accepts a character, not a number

`MyChar As Char = 5`

The `Dim` keyword is missing



Older, pre-.NET versions of Visual Basic allowed, even encouraged, variables that could accept different types of values over time. It's usually a very, very bad idea for reasons that we'll examine in a few pages, but there are a couple of situations where it's necessary.

IDENTIFIERS

In Visual Basic, an IDENTIFIER is the name of something. That "something" could be any programming construct, including variables (`myVar`), constants (`myConst`), types (`Int32`), and methods (`DoSomething`).

The rules for identifiers are simple:

An identifier must begin with a letter or an underscore ('_').

An identifier can contain letters, numerals or underscores after the first character.

An identifier cannot contain spaces.

An identifier cannot be a Visual Basic keyword.

An identifier cannot be more than 1023 characters long.

I have no idea why you'd
ever be tempted to create a
thousand-letter identifier.



A KEYWORD is an identifier that has meaning to the compiler, like `const`. You can find a complete list of keywords on MSDN; search for "Visual Basic Keywords".



PUT ON YOUR THINKING HAT

Indicate whether the following identifiers are valid or not, and if not, why.

ReallyReallyLongName

My Name

1MoreIdentifier

Char

Name3

HUNGARIAN CAMELS

Visual Basic gives you a lot of leeway in naming your identifiers, but it's considered best practice to abide by some basic conventions. Most important of these is that the identifier should be meaningful. `IHaveACrushOnJim` is not a good idea for the name of a constant that represents the name of your elephant. (Unless, of course, your elephant is sitting on Jim.) Beyond that, there are some common conventions that you should know about:

HUNGARIAN NOTATION

Invented by and named for Charles Simonyi, a Hungarian-American software architect, Hungarian notation puts the type of an identifier at the beginning of the name: `intMyInteger`. A lot of programmers swear by Hungarian notation (I'm not one of them), so you'll see a lot of it in sample code, but Microsoft no longer recommends it.

`elephantDumbo`

CAMELCASE

Camel-cased identifiers begin with a lowercase letter, but capitalize the first letter of any other words contained in the identifier: `myCamelCasedIdentifier`. Microsoft recommends camel casing for private identifiers (we'll find out about private identifiers in Chapter 8) and the values you pass to methods. (These are called PARAMETERS, and we'll talk about them in the next chapter.)

`myElephant`

PASCALCASE

Pascal case is like camel case, but the whole thing begins with a capital letter: `MyPascalCasedIdentifier`. Pascal case is the recommended default convention. Use it for everything except private identifiers and parameters.

`MyElephant`



HOW'D YOU DO?

ReallyReallyLongName

Valid

My Name

Invalid - contains a space

1MoreIdentifier

Invalid - begins with a number

Char

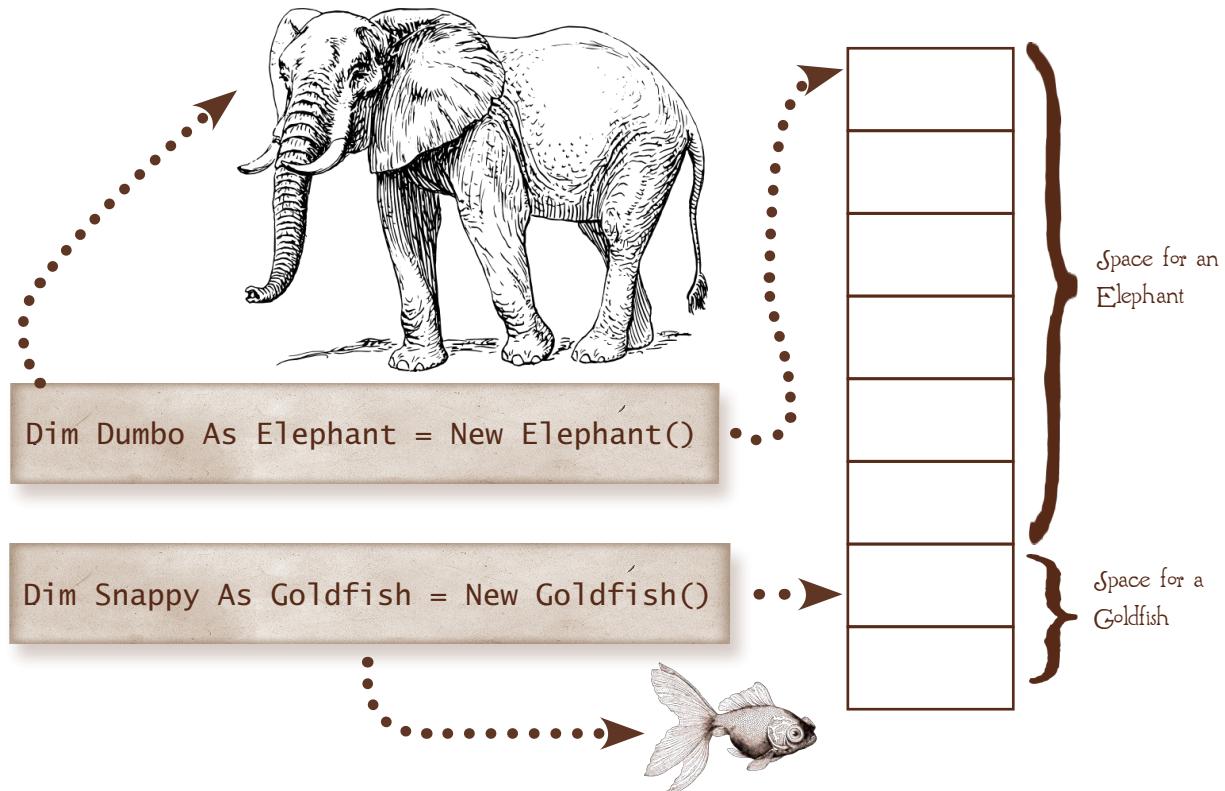
Invalid - keyword

Name3

Valid

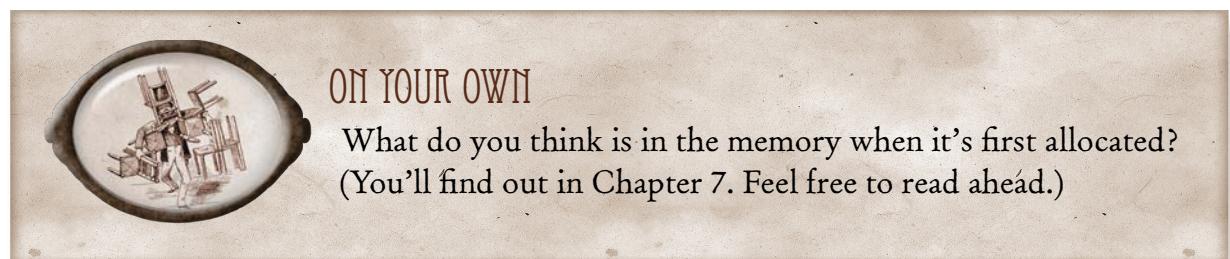
MAKING ROOM

When you create a declared element, .NET will allocate some memory to store it. The amount of memory allocated is determined by the type of element you've defined. Actually, as we'll see in Chapter 8, it's a little more complicated than that, but one of the great things about .NET is that for the most part you don't have to think about what's going on behind the scenes.



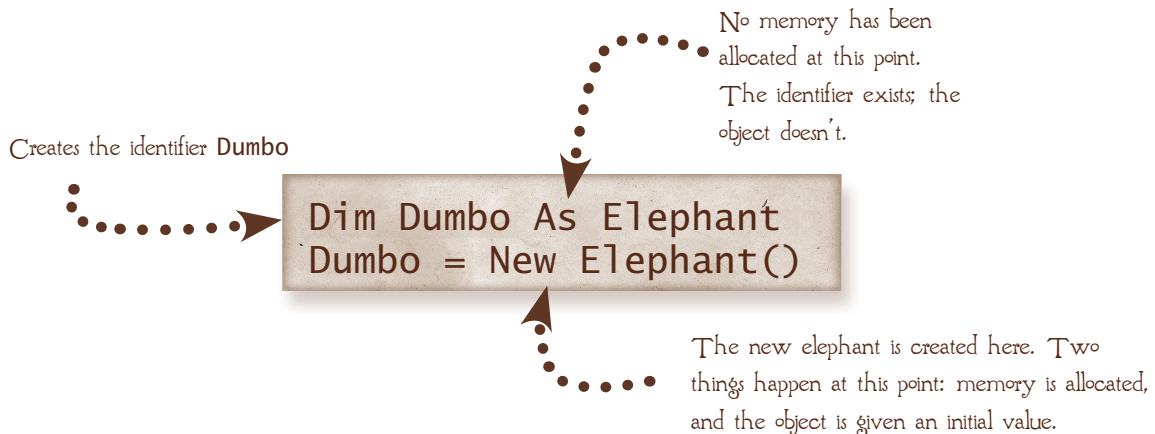
ON YOUR OWN

What do you think is in the memory when it's first allocated?
(You'll find out in Chapter 7. Feel free to read ahead.)



DECLARING OR CREATING?

A declared element has two components: the identifier and the value. The identifier is created when the element is declared. The element itself is created when you assign it a value.



ON YOUR OWN

You've seen that you can create the identifier and initialize it in one statement. There are actually three things that happen when this line gets executed. Can you work out what they are?

`Dim MyInt As Integer = 5`

- ①
- ②
- ③

NEWO SUGAR

When you create and initialize a variable, three things happen: The identifier is declared, the object is created, and it is given a value. `myInt As Int32 = 5` does all three steps in a single statement, but you can also do them separately using the `New` keyword.

DO EACH STEP
SEPARATELY...

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

Dim Letter As Char
Letter = New Char()
Letter = 'A'

...IN TWO STEPS...

Dim Letter As Char
Letter = 'A'

...OR ALL AT ONCE

Dim Letter As Char = 'A';

This only works
for objects that
have simple values,
like numbers.

The `New` keyword creates a new object and invokes its `CONSTRUCTOR`, which is the bit of code that creates an object. (We'll be looking at constructors in detail in Chapter 8.) The constructor in our example is `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.

WORDS FOR THE WISE

The ability to wrap the declaration, creation and initialization of a variable in a single line is called **SYNTACTIC SUGAR**. Syntactic sugar is something the compiler does for you. In this case, it's turning a single statement into three. Every language provides different sugar, which is one reason people develop such strong language preferences—the sugar a language provides makes it easier to perform certain types of tasks.



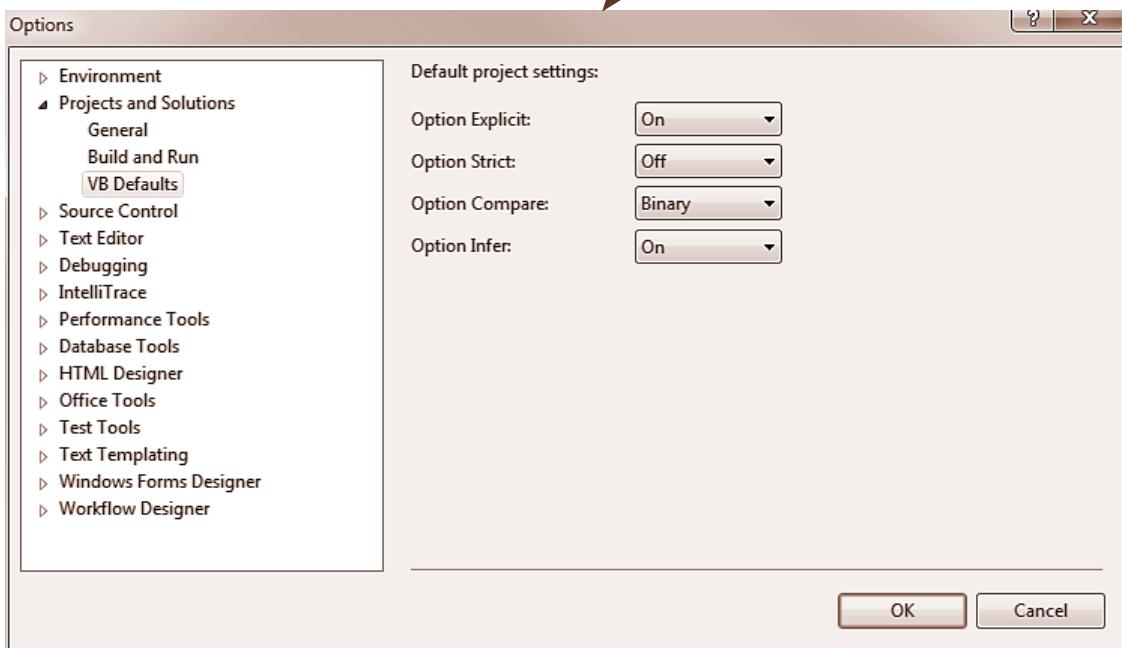
VESTIGES OF VB

As you probably know, Visual Basic 2010 is only the latest in a long line of versions of Visual Basic. The tenth, to be exact. (The first version of Visual Basic was released in 1991.)

The move from "classic" Visual Basic to Visual Basic .NET in 2003 was a huge change. There were changes to the language itself (they're just similar enough to be really confusing), and with the integration of the .NET Class Library, and the changes that entailed...to say the learning curve was steep is a vast understatement. (And I speak as one who still has occasional bouts of altitude sickness.)

It was never going to be easy to upgrade (many programmers still haven't), but Microsoft did what they could to ease the transition. Among other things, they set defaults in Visual Studio to make VB .NET behave as much as possible like earlier versions. Unfortunately, several of those default settings aren't considered best practice for new development, so let's take a minute to fix that up before we go any further.

The options are set in the dialog
displayed when you choose Options
from the Tools menu.



OPTION EXPLICIT

This requires you to declare a variable before you reference it. Unless it's On, which is the default, you'll get yourself in trouble when you misspell a variable name.



```
Dim theVar As Integer  
tehVar = 5
```

If you turn off Option Explicit, VB will silently interpret this as a new variable.



OPTION STRICT

Option Strict prevents three types of subtle errors: narrowing conversions (trying to put an elephant in a goldfish bowl), late binding (trying to put an elephant in a variable defined as an Object, and implicit typing that results in an Object. (We'll talk about that one in a few pages.) Because a lot of legacy Visual Basic programs relied on these features, it defaults to Off. You should change it to On.



```
Dim MyPet As Goldfish  
MyPet = New Elephant()
```

Option Strict prevents these kinds of errors, which are easier to make than you might expect.



OPTION COMPARE

This option controls how the compiler decides two strings are equal. We'll talk about string comparisons in detail in Chapter 10, but basically this option controls things like whether "Hello" is equal to "hello" (case-insensitivity). The default is Binary, and that's usually the best choice.

OPTION INFER

This option controls whether the compiler is allowed to infer the type of a variable from a constant declaration. The default is On, which is usually the best choice, but we'll talk about it some more on the next page.



ON YOUR OWN

Display the VB Defaults section of the Options dialog, and change Option Strict to On.

TYPING WITHOUT A KEYBOARD

I said that **Option Infer**, which defaults to **On**, allows you to let the compiler infer the type of a variable from a constant. Let's take a closer look at how that works. If, instead of specifying the type, you omit the **As** clause, the compiler will work out the type from the value you provide.

```
Dim GuessWhat = "This is a string"  
Dim SomeNumber = 0
```

You have to provide
a value if you don't
provide a type.

Getting the compiler to decide what type of thing a variable contains can be convenient—there are some things that would be tedious to do without it—but you shouldn't use it unless there's a good reason to. To find out why, let's try a little experiment.

Create a new WPF Solution and add a button to the window with the text “Try Me”.



Double-click the button and add the following lines to the procedure:

```
Dim GuessWhat = 0  
GuessWhat = 4.5
```



As soon as you move the cursor off the line, the Code Editor underlines the value with a red squiggly line. What's the error message?

When you declare an element without declaring its type, you've used **IMPLICIT TYPING**. The variable is still a specific type; it's just that the compiler works out what the type is. Unfortunately, it may not come to the conclusion you wanted it to.

So far, we've declared all our numeric variables as `Integer`, but the .NET Framework defines several different types for storing numeric data, and it makes a distinction between integers and floating point numbers. (In computer-speak, **FLOATING POINT NUMBERS** are those that have a fractional component, like `4.5`.)

Visual Basic, like most of the .NET languages, is **STRONGLY TYPED**. That means that once you declare a variable of being type X, you can't go assigning type Y values to it willy-nilly. In our example, the compiler decided that 0 was an integer, and it's not going to let you assign `4.5`, a floating point number, to an integer, because you'd lose information.

In those few situations where you need to, you can usually get around strong type constraints simply by telling the compiler that it's what you really mean to do. (This is called **CASTING**, and we'll see how to do this when we discuss data types.) But strong typing in general is a good thing, because it keeps you from making a lot of silly, difficult-to-find mistakes. (And yes, dear reader, that is the voice of experience!)

TAKE A BREAK



Don't worry; we'll be creating a *lot* of objects in this book, so you'll get a whole lot more practice working with declaration statements. For now, why don't you take a break before you do the final declaration exercise and we move on to comments?



REVIEW

Here are some statements that contain identifiers with problems. Can you identify the problem and provide a better version?

My.Dog.Spot

My String

elephantMine

reallylongidentifier

Write the following declaration statements. Declare all of them as Int32:

- A variable called MyNumber with a value of 5
- A constant called MyConstant with a value of 0
- An implicitly declared variable called MyInt with a value of 10

Write the three lines of code that are equivalent myVar As Int32 = 10:



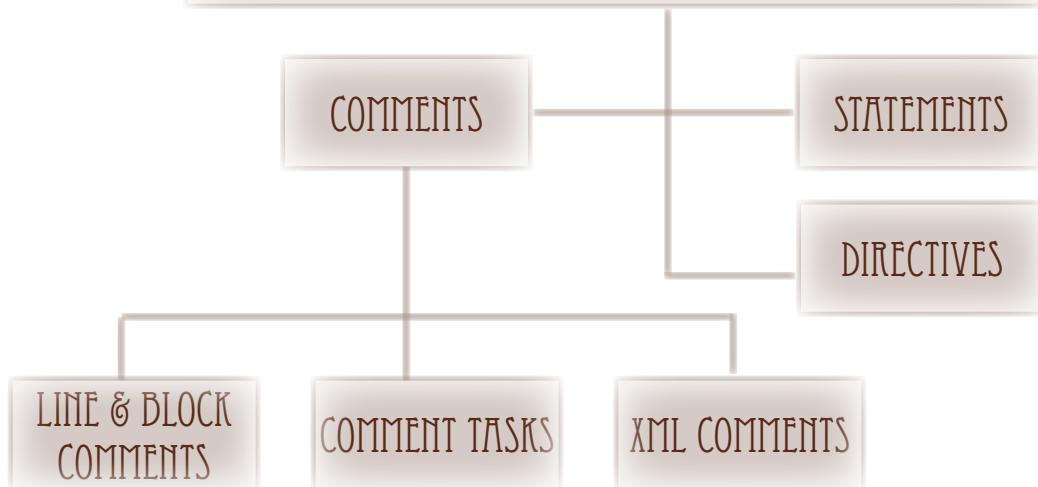
EXPLAIN YOURSELF

Has this happened to you? You’re cleaning out your wallet or your planner, and you find a business card for Suzie Somebody at Widgets, Incorporated. It’s nice that you haven’t lost the card, but you have no idea who Suzie is or why, where or when you acquired it. Nothing written on the back. You don’t even know what a widget is, much less why you thought you needed one.

If it hasn’t happened to you yet, give yourself a year or two. It will. And it can happen with code, too. It might make perfect sense while you’re writing it, but come back in six months when you need to track down a bug, and it will be gibberish. And if you didn’t write the code in the first place? You can spend more time figuring out what the code does than you do fixing it.

The secret is to do the equivalent of writing “Joe’s date at Sam’s wedding” on the back of that business card: Comment your code.

LANGUAGE ELEMENTS



COMMENTING

As we'll see, the Visual Basic compiler understands several different types of comments. The simplest are line comments, which are simply ignored by the compiler.

The most basic type of comment, a **LINE COMMENT**, begins with a single quote ('') and ends at the end of the line:

```
' This is a comment  
    MyInteger As Int32      'This is also a comment
```

As you can see, the single quote comment character can come anywhere on a line. The only rule is that everything after the comment will be ignored. You can type anything you like after the quote; the compiler doesn't care. But you can't continue the comment onto the next line, even if you use the line continuation character:



"cause an error" isn't valid Visual Basic

'This will _
cause an error.



```
' This will  
' NOT cause an error.
```

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.

There is one more rule for comments: You cannot use them after a line continuation character:



```
MyInteger As _ 'This will cause an error '
Int32
```

You can't put ANYTHING after a line continuation character



MAKE A NOTE

Visual Studio has a handy keyboard shortcut for commenting several lines: Select the lines you want to comment out, and press Ctrl+K, C. To uncomment the lines, select them and press Ctrl+K, U.

It's easy to remember: Ctrl+K is used for most editing shortcuts in Visual Studio, so start with that and then press "C" for comment or "U" for uncomment.

TASK LIST COMMENTS

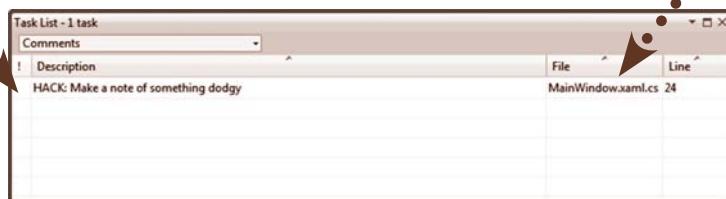
Basic comments are general-purpose "ignore this" indicators to the compiler. The next type of comment, the Task List comment, is ignored by the compiler, but not by Visual Studio.

Task List comments begin with a single quote like a line comment, but the first word is a TOKEN, a word that has meaning to Visual Studio.

The token isn't case-sensitive (Hack or hack would be just fine), and the colon isn't required, but customary.

'HACK: Make a note of something dodgy

When a comment begins with a Task List token, it will be displayed in the Task List window.



Double-clicking on the item in the Task List will take you right to the comment in the code.

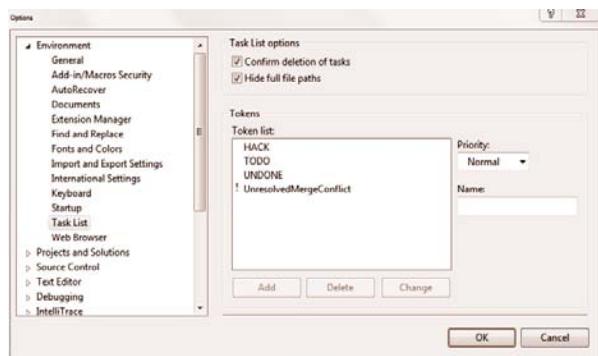


WORDS FOR THE WISE

In programming slang, a HACK is code that works but is ugly or poorly structured. The term is said to have originated with the mathematician John Nash. It isn't related to the term HACKER in the sense of someone who breaks into other people's computers. It is, however, related to KLUDGE, which is used to refer to a quick-and-dirty solution to a problem. A hack can sometimes be cool. A kludge never is.

ADDING TASK LIST TOKENS

By default, Visual Studio only knows a few tokens, but you can add more through the Options dialog available from the Tools menu.



 **ON YOUR OWN**

- ① In the Project you created for this chapter, display the Task List Window by choosing Task List from the View menu or pressing **Ctrl+W, T**. Make sure the drop-down is showing Comments, not User Tasks.
- ② In the editor, add a ToDo comment before the click handler:

```
'Hack: Make a note of something dodgy
Private Sub Button1_Click ...'
```
- ③ As soon as you move your cursor off the comment line, the comment should show up in the Task List window. If it doesn't, make sure the window is showing Comments, not User Tasks, and that you spelled "Todo" correctly. (Capitalization doesn't count.)

XML COMMENTS

Like Task List comments, XML comments are recognized by Visual Studio. They allow you to provide Intellisense support for the code that you write. XML comments can also be used to create documentation files and help files directly from your code using tools like Sandcastle, which is available on the CodePlex site.

XML comments begin with three single quotes, and, like any XML, surround their elements with `<tag>...</tag>` pairs.

XML comments are used to describe methods (like our event handler) or properties (we'll talk about those starting in Chapter 8). You can only use them before these elements.

As soon as you type three quotes, the code editor will add an outline of the elements it thinks you should document. You can add additional elements if you like.

You'd fill the description in here, using as many lines as you need.

There are a dozen or so tags that have meaning in XML comments. Search MSDN for "Recommended Tags" for a list.

The contents of an XML Comment tag can span multiple lines, but each line must begin with the triple slash.

```
Class' MainWindow
  ''' <summary>
  ''' <param name="sender"></param>
  ''' <param name="e"></param>
  ''' <remarks></remarks>
Private Sub Button1_Click(ByVal sender As System.Object,
    ByVal e As System.Windows.RoutedEventArgs) _
    Handles Button1.Click
    ...
End Sub
End Class
```



XML MAGIC AT WORK

The standard tags defined for XML comments help structure your documentation, which can be surprisingly helpful when you're working in a team. And because Visual Studio knows about the tags, it will prompt you for the documentation you should include, and use that documentation itself. Let's try it:

- ➊ In the chapter Project, add a line after the ToDo comment, and type three single quotes. Visual Studio will add an XML Comment stub:

```
'Hack: Something dodgy
''' <summary>
'''
''' </summary>
''' <param name="sender"></param>
''' <param name="e"></param>
''' <remarks></remarks>

Private Sub Button1_Click ...
```

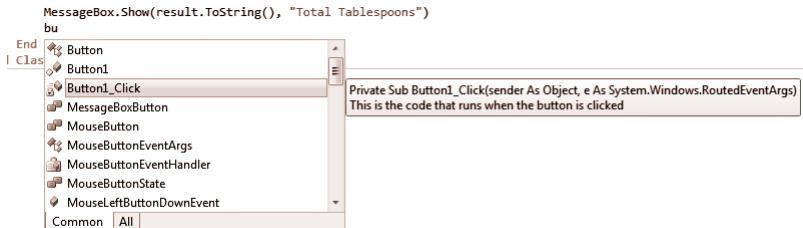
Visual Studio will add some basic documentation where it can.

- ➋ Add a description of the code to the line between the `<summary>` and `</summary>` tags:

```
'Hack: Something dodgy
''' <summary>
''' This is the code that runs when the button is clicked
''' </summary>
''' <param name="sender"></param>
''' <param name="e"></param>
''' <remarks></remarks>

Private Sub Button1_Click ...
```

- ➌ Now place your cursor on a new line anywhere inside the procedure, and type "bu" to trigger Intellisense. Cursor down to `Button1_click`. Like magic, your comment appears:



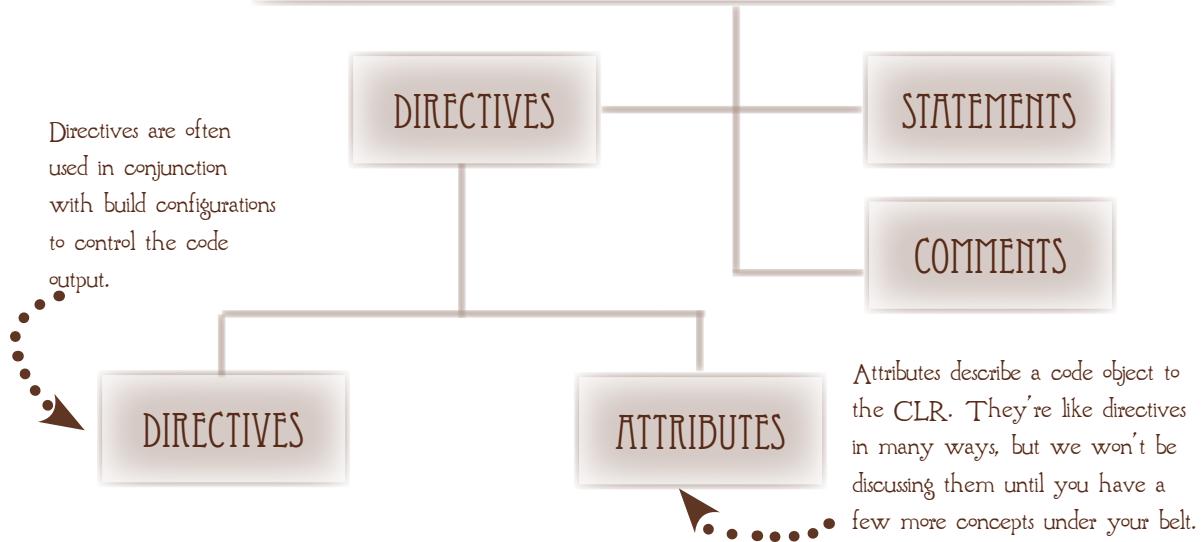


DIRECTIVES

Although Task List and XML comments are understood by Visual Studio, all comments are ignored by the compiler. That doesn't mean you can't give the compiler directions,

however. The compiler understands a small set of DIRECTIVES that you can use to control what code it includes in the output. ATTRIBUTES, on the other hand, are included in the final output and used to describe a code object.

LANGUAGE ELEMENTS



WORDS FOR THE WISE

In computer science, a PREPROCESSOR is a program that prepares input for another program. In older languages, directives were handled by a preprocessor that parsed them before passing the source code onto the compiler proper. Visual Basic doesn't use a preprocessor, but directives are still usually called PREPROCESSOR DIRECTIVES, even in the Microsoft documentation.

USING DIRECTIVES

The directives that are used most often are the `#const` directive that defines a value and the `#if...#else...#endif` directive that controls what code is included in the output based on whether an expression is true or false. (The `#else` clause is optional.) The two directives are almost often used together. Let's see how that works:

```
#CONST DEBUG = True
Private Sub Button1_Click ...
#If DEBUG = True
    MessageBox.Show("Debug is defined.")
#else
    MessageBox.Show("Debug isn't defined.")
#endif
End Sub
```

If DEBUG isn't
defined, this code is
produced instead.

If DEBUG is defined,
this is the code
produced. It's just
as though the
other code
weren't even
there.

```
Private Sub Button1_Click ...
    MessageBox.Show("Debug is defined.")
End Sub
```

```
Private Sub Button1_Click ...
    MessageBox.Show("Debug isn't defined.")
End Sub
```

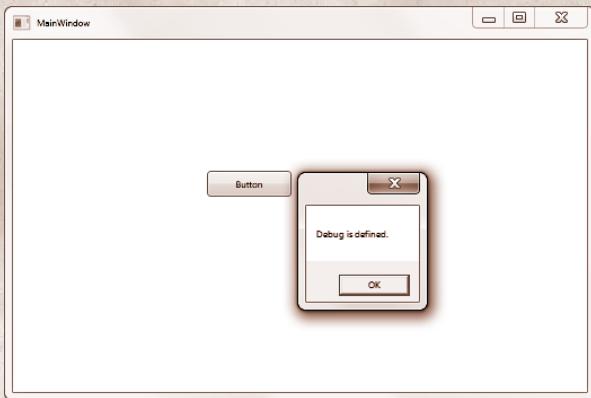


ON YOUR OWN

Why don't you try using the Debug constant, just to get a sense for how directives work?

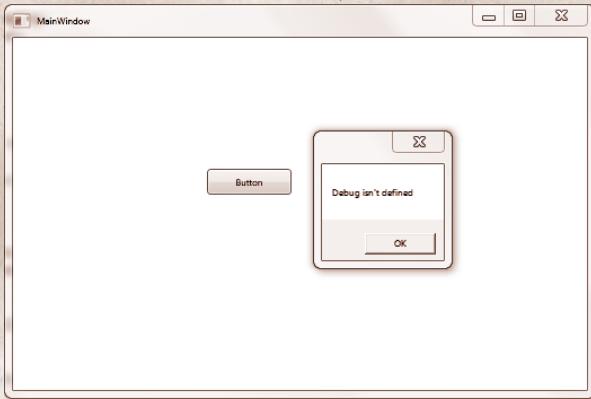
1

Add the code on the previous page to the chapter project, and run it to make sure that “Debug is defined” is displayed in the MessageBox.



2

Change the #CONST DEBUG statement to define DEBUG as false and rerun the application to make sure that “Debug isn't defined” is displayed.



DEFINING GLOBAL DIRECTIVES

Two constants are used by convention during development: DEBUG is used to control output, usually to perform additional checks or display more detailed information than would be required in a released application, and TRACE is typically used to include code that outputs information about the state of the application at a given point in time. Visual Basic provides a simple way to define these constants on a project-wide basis via the Advanced Compile Settings dialog that is displayed when you click the Advanced Compile Options button on the Compile tab of the Solution Properties dialog:

Click here to display the dialog.



The constants are controlled here.

ON YOUR OWN



Check that the DEBUG constant is defined in the Solution Properties dialog, and then change the #if directive to read simply #if DEBUG. Does the application still work as expected?

TAKE A BREAK



You're almost finished with the first pure language chapter. Was it what you expected? Why don't you take a break now, before completing the Review, and then we'll move on to expressions?



REVIEW

Write a single statement that declares an Integer and sets its initial value to 200:

Rewrite the declaration as three separate statements:

Which of the following are legal identifiers in Visual Basic?

myIdentifier
12Things

Some Elephant
#Items

What are the two types of comments recognized by the Visual Basic compiler, and how do you begin each one?

What character do you use to begin a directive?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



THE LANGUAGE, PART 2: TRANSITIVE VERBS

6

In the last chapter you saw how to create the nouns of Visual Basic: declared elements. In this chapter, we'll look at one of the two kinds of verbs: the EXPRESSIONS that are roughly (very roughly) equivalent to transitive verbs. (If it's been awhile since you studied grammar, a transitive verb is one that takes both a subject and an object: "He bought the book.")

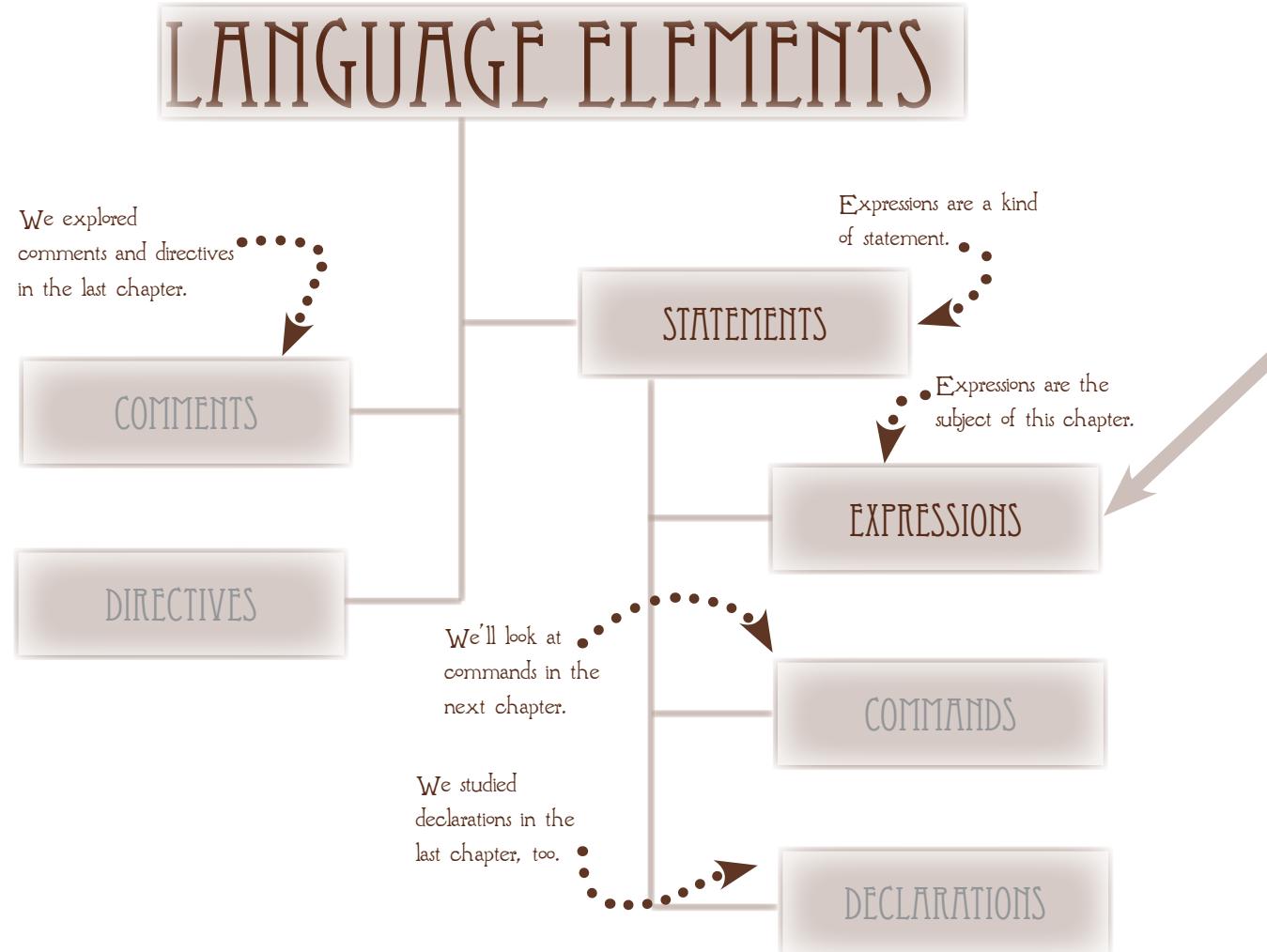
Many people equate programming expressions to mathematical formulae. That's not wrong exactly, but expressions are a lot more than that. Of course you can say, " $x = 1 + 1$ " in Visual Basic, and you'll use an expression to say it, but you'll also use expressions to say things like, "If Dumbo is an elephant, give him some peanuts" and "Greedy wants the biggest apple in the barrel".

By the way, programming doesn't require nearly as much mathematics as some people think. In fact, unless you're writing a mathematical application or doing certain kinds of graphics work, it requires hardly any at all. There is some arithmetic involved, but the computer will do all that for you. Personally, I find that comforting.



LANGUAGE ELEMENTS

We're continuing our study of the Visual Basic parts of speech in this chapter.



IN A NUTSHELL

EXPRESSIONS

We'll explore
method calls in
Chapter 8.

LINQ expressions are primarily used for data handling, things like, "show me all the vegetable providers who sell artichokes".

We won't be studying LINQ in this book.

SPECIAL

LINQ

```
from e in myColl  
select e
```

LAMBDAS

```
Function(x) Return x + 1
```

A lambda is an anonymous expression, roughly equivalent to declaring an anonymous variable. We'll see an example of lambda expressions when we study WPF binding in Chapter 22.

METHODS

```
MessageBox.Show()
```

LITERAL

```
a = x + 1
```

IDENTIFIERS

```
myIdentifier
```

Yes, names like `myIdentifier` really are considered expressions in Visual Basic because the compiler has to do something to resolve a name. There really isn't anything more for us to learn about these.



TASK LIST

In the last chapter, we examined comments, directives, and one kind of Visual Basic statement: the declaration statements that create the nouns you'll work with. In this chapter, we'll look at one of the two remaining types of statements: expressions.



BASIC EXPRESSIONS

What in spoken English is generally called an “equation” in Visual Basic is called a **LITERAL EXPRESSION**. In addition to basic arithmetic, Visual Basic literal expressions include text manipulation (combining strings, for example) and an important category of logical formulas called Boolean expressions. We’ll look at all of these in this chapter.



OBJECT EXPRESSIONS

Visual Basic is an object-oriented language, which means that it models what the application needs to do as a set of programming constructs (technically called **TYPES**, of which objects are one kind) that have **STATE** (they know things) and **BEHAVIOR** (they can do things). We’ll be looking at exactly what this means and how to do it throughout the rest of the book, and we’ll get started here by looking at how to get at the behavior that types expose: by calling methods.



MAKE A NOTE

You’re going to run into a lot of new concepts in this chapter, and I won’t be explaining all of them right away. (It’s a problem: just about every aspect of programming is related to just about every other aspect.)

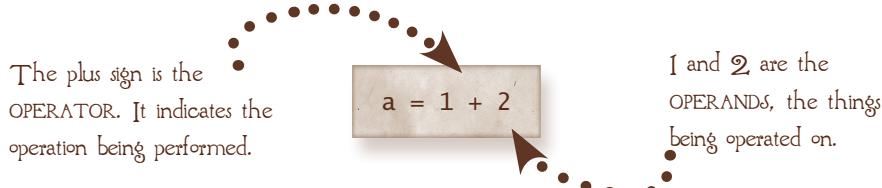
Don’t panic. You don’t need to understand everything about everything right away. Just concentrate on figuring out how expressions work, and the other bits will fall into place in a chapter or two.



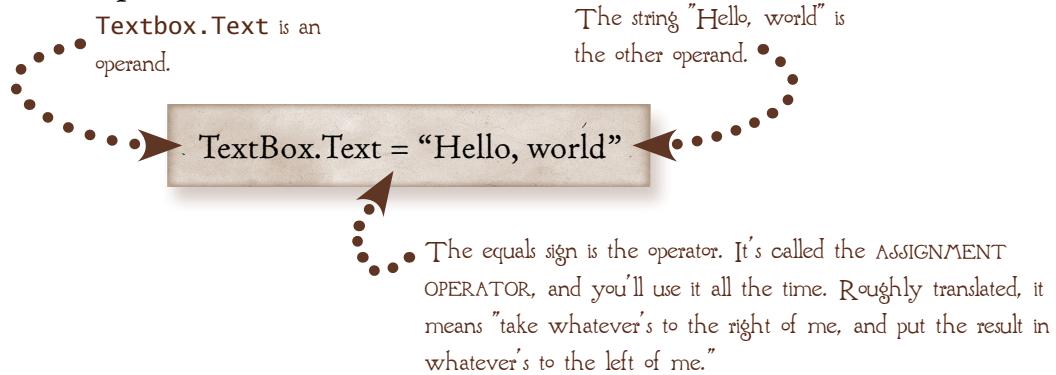
BASIC EXPRESSIONS

Like transitive verbs in spoken English, a Visual Basic expression performs some action on some thing. In English, the action is called the "verb", and the thing it acts on is the "object". In Visual Basic, the equivalent of the verb is the OPERATOR, and the object is the OPERAND. These are the same terms used in mathematics, but Visual Basic expressions don't often look like mathematics (unless, of course, you're actually doing mathematics). Let's look at some examples:

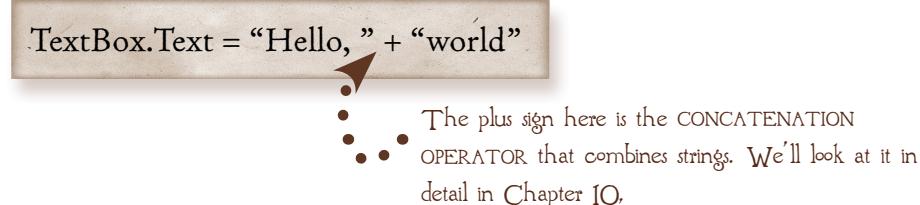
This is probably the kind of thing you think of when you hear the word "equation", and by extension, "Visual Basic expression":



But Visual Basic expressions can do a lot more. This isn't a mathematical equation, but it is a valid Visual Basic expression:



In Visual Basic, operands don't always operate on numbers. Here's another example:



EXPRESSIONS VS. EQUATIONS

Many Visual Basic expressions look just like mathematical equations: They have an operand on either side of an operator, and the operators look like the ones you used for arithmetic. But it's important to keep sight of the fact that Visual Basic expressions are NOT mathematical equations.

- Expressions don't always have two operands, although they always have at least one:



You probably don't think of this as an expression, but it is: NEGATION. The operator is the minus sign.

- Operators don't always look like mathematical symbols. Some of them are words, and the most important of these, new, you've already used:

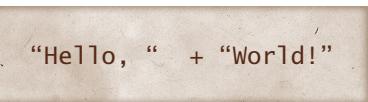


This probably looks familiar by now.



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 redefine the meaning of true and false when you start building your own classes (which we'll do in Chapter 8.)

- Operands aren't always numeric values, even when the operators look like arithmetic:



The operands here are, of course, strings of characters, not numbers. The plus sign, in this situation called the CONCATENATION OPERATOR, sticks the string to its right at the end of the string to its left: "Hello, World!!" We'll examine strings and how to manipulate them in detail in Chapter 10.



PUT ON YOUR THINKING HAT

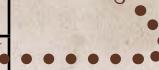
Let's get our feet wet with expressions by writing some expressions that map to simple mathematical equations. The operators used to create basic mathematical formulas in Visual Basic are shown in the table. Translate each description into a statement that contains a literal expression.

The MODULUS is the remainder of a division. $7 \bmod 2$, for example, is 1 (0.5 rounded to the nearest whole number).



+	Addition
-	Subtraction
*	Multiplication
/	Division
Mod	Modulus
=	Assignment
\	Integer division
^	Exponentiation

INTEGER DIVISION returns only the integer portion of the result, throwing away any remainder.



Assign the result of 30 divided by 15 to x:

Assign what's left over after you divide 33 by 2 to z:

Assign the result of 15 divided by 3 to q:

Add 2 to y, and assign the result to y:

Add 1 to the value of x, and assign the value to x:



HOW'D YOU DO?

Assign the result of 30 divided by 15 to x:

$x = 30 / 15$

Assign what's left over after you divide 33 by 2 to z:

$z = 33 \text{ Mod } 2$

Assign the result of 15 divided by 3 to q:

$q = 15 / 3$

Add 2 to y, and assign the result to y:

$y += 2$



This one was a trick question. You weren't wrong if you said $y = y + 2$, but Visual Basic has some special operators that perform an operation and assignment in one step. They save a lot of typing, and are sometimes clearer than spelling things out. (But spelling things out is always okay, too.) The syntax for these operators is $x <\text{op}>= y$, where $<\text{op}>$ is any of the operator symbols.

Add 1 to the value of x, and assign the value to x:

$x += 1$



If you said $x - x + 1$ you were also right.

WHAT HAPPENS WHEN?

What's the result of this equation?

$$2 + 3 * 4$$

Did you say 20 or 14? Depends on whether you did the multiplication or the subtraction first. If you typed the equation into a calculator, you probably said 20, because calculators always evaluate left to right. But in Visual Basic the answer is 14, because Visual Basic evaluates expressions according to a specified precedence, and multiplication precedes (i.e., has a higher precedence than) subtraction.

You can memorize the order of precedence, but best practice is to use parentheses to disambiguate an equation. The part of the equation inside the parentheses is always executed first:

This is 20.



$$(2 + 3) * 4$$

This is 14.



$$2 + (3 * 4)$$

PUT ON YOUR THINKING HAT



In practice, you don't use the VB compiler as a calculator very often (you might build a calculator application, but that's a different issue). You'll usually build your equations from variables, and the logic of the situation will tell you the order in which things need to happen. But just to be sure you've got the concept down, here are some practice equations. Can you add parentheses to these formulas so that they return the correct values?

$3 + 5 * 7$ (return 56, not 38)

$5 \bmod 3 * 2$ (return 4, not 5)

$6 / 3 + 2$ (return 1, not 4)

HINT: To check your results, display the Immediate window (from the Windows sub-menu on the Debug menu), and precede each equation with a question mark, which is a shortcut for the Debug.Print command: ? $3 + 5 * 7$



HOW'D YOU DO?

Did you try the Immediate window? Cool, isn't it? You can also use it when you're in break mode when you're debugging, and in break mode you can reference any variables that are in scope. It's really useful when the variables contain the correct values, but they're just not behaving the way you expect them to.

Make $3 + 5 * 7$ return 56:

By default, multiplication occurs before addition, so without any parentheses, this equation returns 38. To return 56, you need to force the addition to occur first:

$$(3 + 5) * 7$$

Make $5 \bmod 3 * 2$ return 4:

By default, multiplication has higher precedence than modulus so this equation returns 5. To return 4, you need to force the modulus to occur first:

$$(5 \bmod 3) * 2$$

Make $6 / 3 * 2$ return 1:

Division and multiplication have the same precedence, so by default they're evaluated left to right. Again, you can use parentheses to control things:

$$6 / (3 * 2)$$



UNDER THE MICROSCOPE — OPERATOR PRECEDENCE

For reference, here's the order of precedence for Visual Basic operators. Don't try to memorize it. If you need to check it in the future, use MSDN.

^

Unary negation

*, /

\

Mod

+, -

&

And then...

<<, >>

=, <>, <, <=, >=

Not

And, AndAlso

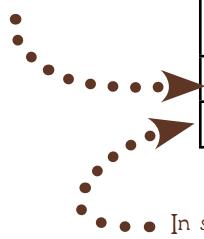
Or, OrElse

XOr

BEGINNING BOOLEANS

An important type of expression, one you'll probably use a lot in your work, is a Boolean expression. Boolean expressions use relational or logical operators and return the values true or false. (Be careful of the capitalization.) Of the two, the relational operations are the simplest. They return true or false based on the relationship of two numeric values:

This symbol can be used for either comparison or assignment. In practice, it's not usually confusing, but make a note of it.



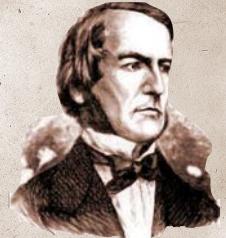
op	Meaning	True Example	False Example
<	"less than"	$1 < 2$	$2 < 1$
\leq	"less than or equal to"	$2 \leq 2$	$4 \leq 2$
$>$	"greater than"	$2 > 1$	$2 > 2$
\geq	"greater than or equal to"	$3 \geq 3$	$1 \geq 2$
$=$	"is equal to"	$1 = 1$	$1 = 2$
\neq	"is not equal to"	$1 \neq 2$	$1 \neq 1$

In symbolic logic, \neq is used to mean "not equal to", and if you've studied logic at all, this might seem odd. Think "either less or greater than, but not equal to".



A BIT OF HISTORY

Boolean expressions and the Boolean values true and false are named after George Boole, the Victorian mathematician and logician who developed symbolic logic in the 1840s.



BEING LOGICAL

The second category of Boolean operators are the logical operators that compare Boolean values.

op	Meaning	True Example	False Example
And	both are true	True And True	True And False
Not	not	Not False	Not True
Or	either is true	True Or anything	False Or False
Xor	“exclusive or” — only one is true	True XOr False	True XOr True
AndAlso	“conditional and” — stop if the first operand is False	True AndAlso True	False AndAlso anything
OrElse	“conditional or” — stop if the first operand is True	True OrElse anything	False OrElse anything

CONDITIONAL OPERATORS & SIDE EFFECTS

To understand the conditional operators, consider the following snippet of code:

```
MyNumber x = 1;  
MyNumber y = 5;  
Boolean a;  
  
a = (x < y) XOr (x.Add(1) < y);
```



Assume that the `MyNumber` class has a method called `Add()` that adds the value provided to its current value.

What's the value of `x` after this code executes? You might expect it to be `2`, because the `Add()` method adds `1` to the initial value. But `XOr` is a conditional `or`. It doesn't evaluate the expression to the right (`x + 1 < y`) unless the result of the expression to the left is `False`. Since `x` is indeed smaller than `y`—`(x < y)` is `True`—the compiler never looks at the increment operator.

Confused? Probably; it's confusing. Calling `Add()` in that equation is called a **SIDE EFFECT**. (Means the same thing in programming as it does in spoken English.) Side effects are inherently confusing, and you should avoid them. They'll happen by accident sometimes (and I can almost guarantee you won't enjoy tracking down that bug!), but it's considered very poor practice to create them on purpose. They're never necessary and it doesn't matter how many comments you write, they're just too hard to understand.



PUT ON YOUR THINKING HAT

Time to give your new Boolean expression-writing skills a whirl. Translate each of the questions below into a valid Visual Basic expression. (Checking the operator tables isn't considered cheating.)

Does x have the same value as y?

Are both x and y true?

Is only one of x or y true?

Do x and y have different values?

Is x bigger than y?

Is either x or y true?

Is y smaller than or equal to x?



HOW'D YOU DO?

Does x have the same value as y ?

$x = y$



As it stands, this is actually an assignment statement, but in practice you'd put it in a test ("if x is equal to y " or assign it to a Boolean variable, so the meaning would be clearer).

Are both x and y true?

$x \text{ And } y$

Is only one of x or y true?

$x \text{ Or } y$

If you got this one wrong, look at the question again.



The exclusive or returns true if one, and only one, of the operands is true.

Do x and y have different values?

$x <> y$

Is x bigger than y ?

$x > y$

Is either x or y true?

$x \text{ Or } y$

Is y smaller than or equal to x ?

$y <= x$



ON YOUR OWN

Can you match the following terms to their definitions?

Boolean Value

A symbol that acts on one or two values.

Operator

An operator that adds 1 to the value of a variable and assigns the result to that variable.

Increment Operator

A logical operator that doesn't evaluate the second operand if the first operand is sufficient to determine the result.

Operand

An expression that returns true or false.

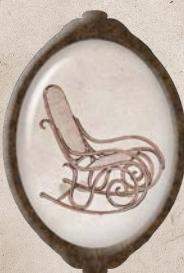
Conditional

True or false. Sometimes called a "logical value".

Operator

Boolean
Expression

A value that is evaluated by an operator.



TAKE A BREAK

Boolean logic can seem really strange until you get the hang of it. The symbols look weird, and "x and y" doesn't mean the same thing that it means in English. (If I ask you to give me all the clients in New York and California, I mean either, not both—a logical or, not an and.)

But why don't you take a break before you move on to the Review? You might find you understand them better than you think you do.



REVIEW

Circle the four operators in the following statement. What type of value does the expression return?

$x = (y * 3) >= 15) \text{ And } (y < 5)$

Given the declarations and initial values shown, rewrite each of the following statements so that they return true instead of false:

```
Integer x = 1;  
Integer y = 5;  
Boolean a = True;  
Boolean b = False;
```

$x < x$

$(x \leq x) \text{ Or } b$

$x > y$

$a \text{ And } b$

$(x > y) \text{ Or } b$

OBJECT EXPRESSIONS

I've said (and it probably wasn't news) that Visual Basic is an object-oriented language and that objects are a programming construct that have STATE (things they know) and BEHAVIOR (things they do). You've seen that you create an object using a declaration statement, and that when you do, you must specify what type of object it is.

As we'll see, you can define several different kinds of objects (called TYPES) in Visual Basic, but the one you'll probably work with most often is the OBJECT. You already know how to create an instance of an object via a declaration statement. We'll be looking at how to define objects beginning in Chapter 10. Right now, we'll look at how you perform two important kinds of object expressions: member access, which gives you access to an object's state, and method calls, which invoke its behavior.

The diagram is titled "ANATOMY OF AN OBJECT". It features a classical oval portrait of Vitruvian Man on the left. Three arrows point from three statements on the left to a screenshot of the Visual Studio Class Designer window on the right. The first arrow points to the statement "An object's state is represented by fields and properties." The second arrow points to the statement "An object's behavior is represented by its methods and events." The third arrow points to the explanatory text about the Class Designer.

ANATOMY OF AN OBJECT

An object's state is represented by fields and properties.

An object's behavior is represented by its methods and events.

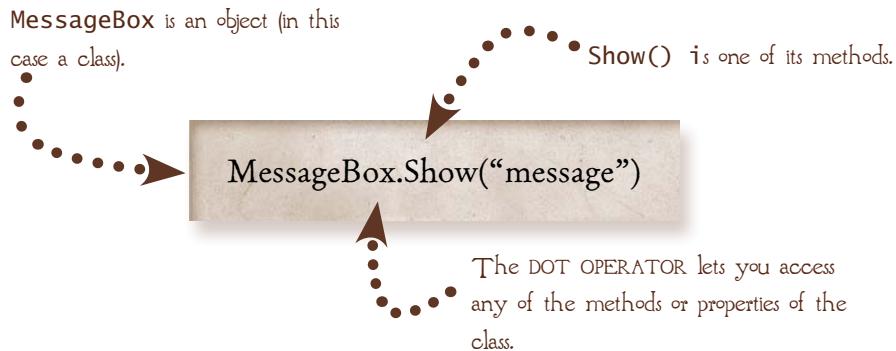
The things an object can know and do are defined in its class declaration. This is what a class looks like in the Visual Studio Class Designer. (We'll be working with the Class Designer a lot in the chapters to come.)

Class1
Class

- Fields
 - fieldOne As Integer
 - fieldTwo As Integer
- Properties
 - PropertyOne As Integer
 - PropertyTwo As Integer
- Methods
 - DoSomething(numericValue As Integer)
 - DoSomethingElse(numericValue As Integer, logicalValue As Boolean)
- Events
 - PropertyOneChanging As EventHandler
 - PropertyTwoChanging As EventHandler

DOTS TO MEMBERS

So far, we've looked at literal expressions and operators that look (more or less) like the equations you remember from school. But there's a very important kind of expression that is hardly recognizable as one. It's called MEMBER ACCESS (the fields, properties, methods and events of a class are called its members), and the operator is a period ".", commonly referred to as the "dot operator".



ON YOUR OWN

Here's the class diagram from the previous page. Using the dot operators, write the expression (not a statement) to access each of its properties.

Class1
Class

- Fields
 - fieldOne As Integer
 - fieldTwo As Integer
- Properties
 - PropertyOne As Integer
 - PropertyTwo As Integer
- Methods
 - DoSomething(numericValue As Integer)
 - DoSomethingElse(numericValue As Integer, logicalValue As Boolean)
- Events
 - PropertyOneChanging As EventHandler
 - PropertyTwoChanging As EventHandler

METHOD SIGNATURES

You've already made some method calls, which simply means that you've gotten an object (in our examples, a `MessageBox`) to do something it knows how to do. We'll look at how to build methods starting in Chapter 8, but let's start now by exploring the **SIGNATURE** of a method, and what that means for how it's used.

This is the **SIGNATURE** of a method. The signature is defined by the method name and the number and type of its parameters.

Methods can either be **FUNCTIONS** that return a value or **SUBs** (short for "subroutine" that don't).

A **PARAMETER** specifies a value (either a literal value or a variable) that you pass to the method to work with. The method signature defines the type and name of each parameter.

Parameters are given names (conventionally in `camelCase`). The names are used to refer to the parameters within the method, but you don't need to use the same names for variables you use as **ARGUMENTS** (the values you pass to the method), and they aren't technically part of the signature.

[Modifier] <Sub/Function> <MethodName>([<parameterName> As <type>], ...) [As <type>]

Method names conventionally use `PascalCase`.

If a method needs more than one parameter, they're separated by commas.

Functions return a value that can be assigned to a variable of the type specified by `RETURN TYPE`.

Modifiers determine where the method can be seen and how it can be modified. We'll examine them in Chapter 12.

WORDS FOR THE WISE



The data that's passed to a method is called two different things, depending on the context. The abstract definition is a **PARAMETER**. But the actual values passed are **ARGUMENTS**. The distinction isn't critical, and people tend to use them interchangeably, but it is a valid difference: parameter is the description of the value, argument is the value itself.

METHOD OVERLOADING

Visual Basic, and indeed all the .NET languages, allows you to define multiple methods with the same name, distinguished by their signatures. This process is called **OVERLOADING**, and the methods are called **OVERLOADS**. It saves you thinking up or remembering different method names for methods that vary only by their signature. Here's how the process works:

Different classes can define methods with the same name. These aren't overloads—the simple fact that they belong to different classes distinguishes them.

The `MessageBox` class defines 12 different overloads of the `Show()` method. Here are examples of two.

```
MessageBox.Show("message")
MessageBox.Show("message", "caption")  
  
Window MyWindow = new Window();
MyWindow.Show()
```

The `MessageBox.Show()` overloads are distinguished by their signatures:

```
Public Shared Function Show(string message) As MessageBoxResult
Public Shared Function Show(string message, string caption) As MessageBoxResult;
```

The parameters are given the same name, but that isn't a requirement. Remember, the name isn't technically part of the signature; it's the type that distinguishes them.

It's really not a good idea to have overloads with different return types anyway. It's too confusing.

The return type (`MessageBoxResult` in this example) is not technically part of the signature, so you can't define two methods with the same name that are distinguished only by the values they return.

PUT ON YOUR THINKING HAT

The `Integer` exposes a method named `Parse()` that accepts a `String` value like "23" and turns it into a number, 23. What do you think the signature of this method is?



CALLING METHODS

When you put something like `MessageBox.Show("my message")` in your code, you're technically doing something called INVOKING the method, but we commonly say that you're CALLING the method, and the syntax is referred to as a METHOD CALL. You use the dot operator to call a method. (But you already knew that.)

You've used the `Show()` method of the `MessageBox` class several times now, and you've seen the support that Intellisense gives for calling the method. Let's have another look at that:

The diagram illustrates the Intellisense interface for the `MessageBox.Show()` method. A tooltip window is open below the method name. The tooltip contains the following information:

- These numbers tell you how many overloads are defined for the method. In this example, you're looking at the second of twelve.
- These are the parameters that define the signature for this version of the method.
- This is the return type of the method, `MessageBoxResult`, another class. We've been ignoring it, which is always an option.

MessageBox.Show()

▲ 1 of 12 ▼ **Show(messageBoxText As String) As System.Windows.MessageBoxResult**

Displays a message box that has a message and that returns a result.

messageBoxText: A *System.String* that specifies the text to display.

Remember that the parameter names are used inside the method to refer to the values you've passed. You don't have to use them when you call the method.

UNDER THE MICROSCOPE

Symbolic operators like “+” and “/” are actually syntactic sugar. The compiler translates them into method calls like

- `Int32.Add(Int32 x, Int32 y)`

You can always call the method directly if it seems easier or makes your intentions clearer. (But that doesn't happen very often.)

HOW'D YOU DO?

Here's the signature of the `Int32.Parse()` method:

`Function Parse(s As String) As Integer`

The method is a function because it returns a value.

The method name is `Parse`.

The value the method returns is an `Integer`.

It takes a single parameter: a `String` that's given the name `s`.

UNDER THE MICROSCOPE—SHARED METHODS

Look at the example method calls on page 174 again. Do you notice something different about the way `MessageBox.Show()` and `Window.Show()` methods are called? `MessageBox.Show()` is called directly on the class itself, while `Window.Show()` is called on an instance of the method.

`MessageBox.Show()` is a STATIC METHOD, which in Visual Basic is declared with the Shared modifier. We'll explore modifiers in detail in Chapter 12, but for right now, you should know that you don't need to instantiate a class in order to call a static method. You just specify the class name as though it were an instance and pass whatever parameters are required.

Static methods are typically used for utility operations when all the information required to perform whatever the method does can be passed as parameters. (Static methods can't reference the object's state, because there's no instance to store that information.)

The `MessageBox.Show()` method is pushing the limits of parameter passing. Some of the overloads require eight or nine parameters, and passing that many parameters quickly gets tedious. When you design your own classes, you'll need to think carefully about what's reasonable for your situation.

TYPE NAMES

One last note about object expressions before we move on: Have you noticed that MSDN talks about the `Int32` (and sometimes I do, too), but Visual Basic calls it an `Integer`? As we'll see when we examine the .NET Base Class Library, both of these are types defined by the .NET Framework itself, and like all types in the .NET Framework Library, they have PascalCase names.

But Visual Basic also defines keywords for the most common types and, like all keywords, they have no capitalization at all. The Visual Basic keywords are the same as those used in most older versions of VB, so presumably the designers thought it would be more comfortable for people who were used to them.

Visual Studio gives preference to the Visual Basic keywords, particularly in Intellisense but I prefer the .NET Framework type names. (I program in several different languages, so it's easier for me to only remember a single set of names.) But the keywords are just synonyms. At compile time, they're translated into references to the .NET Framework types. It doesn't matter which one you use and you can even use a combination of both without problem.

Visual Basic type name	.NET type name
Date	DateTime
Integer	Int32
Long	Int64
Short	Int16



TAKE A BREAK

Well, that about does it for expressions. There are some complications, and we'll look at those as we go along, but for now, why don't you take a break before completing the Review and moving on to the last category of Visual Basic language components: commands?



REVIEW

When you're passing multiple arguments to a method, how do you separate them?

What does the Mod operator do?

What's the difference between Integer and Int32?

The = symbol can be used in two situations. What are they?

In the context of programming, what does the term "side-effect" mean?

What does the dot operator do?

What's the difference between a parameter and an argument?

What's the difference between Or, XOr and OrElse?

When can you call a method without instantiating a class?

Why is it a good idea to use parentheses to explicitly define operator precedence?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



THE LANGUAGE, PART 3: INTRANSITIVE VERBS



In the last chapter we explored Visual Basic expression syntax and had our first look at working with the objects of object-oriented programming. In this chapter, we'll finish up our examination of the language syntax by examining the intransitive verbs of the language: commands.

There are two important types of commands in Visual Basic, and we'll examine both of them in this chapter. Control-of-flow commands determine what code executes when (and how often). Exception-handling commands tell your application what to do when things don't go quite as expected, which of course things don't, more often than not.

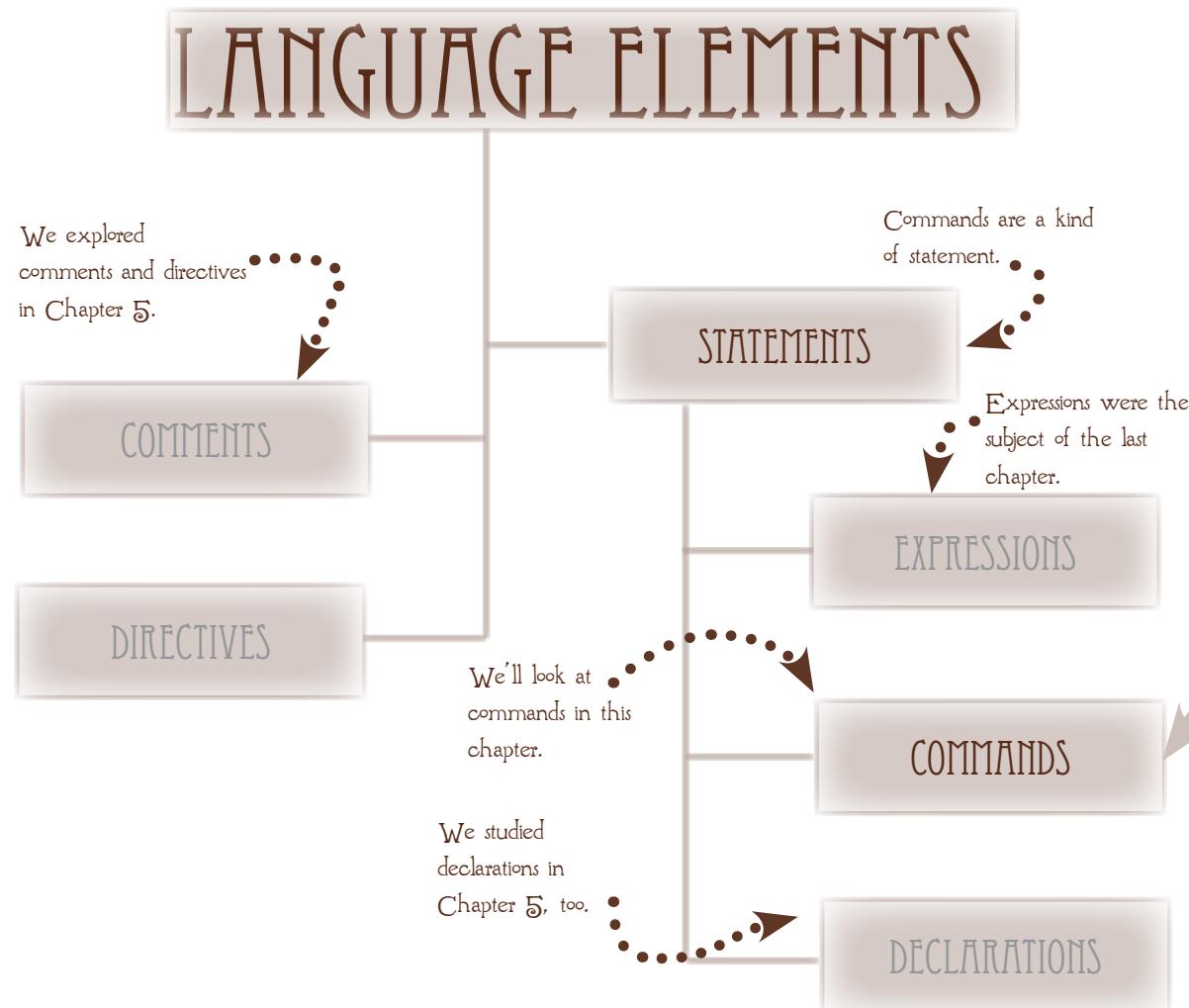
Although it's not unknown for a program to start execution at line one and continue, line by line, to the end, most applications need to perform different actions based on a condition or perform a set of actions multiple times. The control-of-flow commands (combined with the Boolean logic you learned in the last chapter) are what make that possible.

Exception-handling commands, on the other hand, make your code resilient to the unexpected. Whether it's an argument of the wrong type or a piece of hardware that isn't where it's supposed to be, stuff happens, and exception handling is how your applications cope.

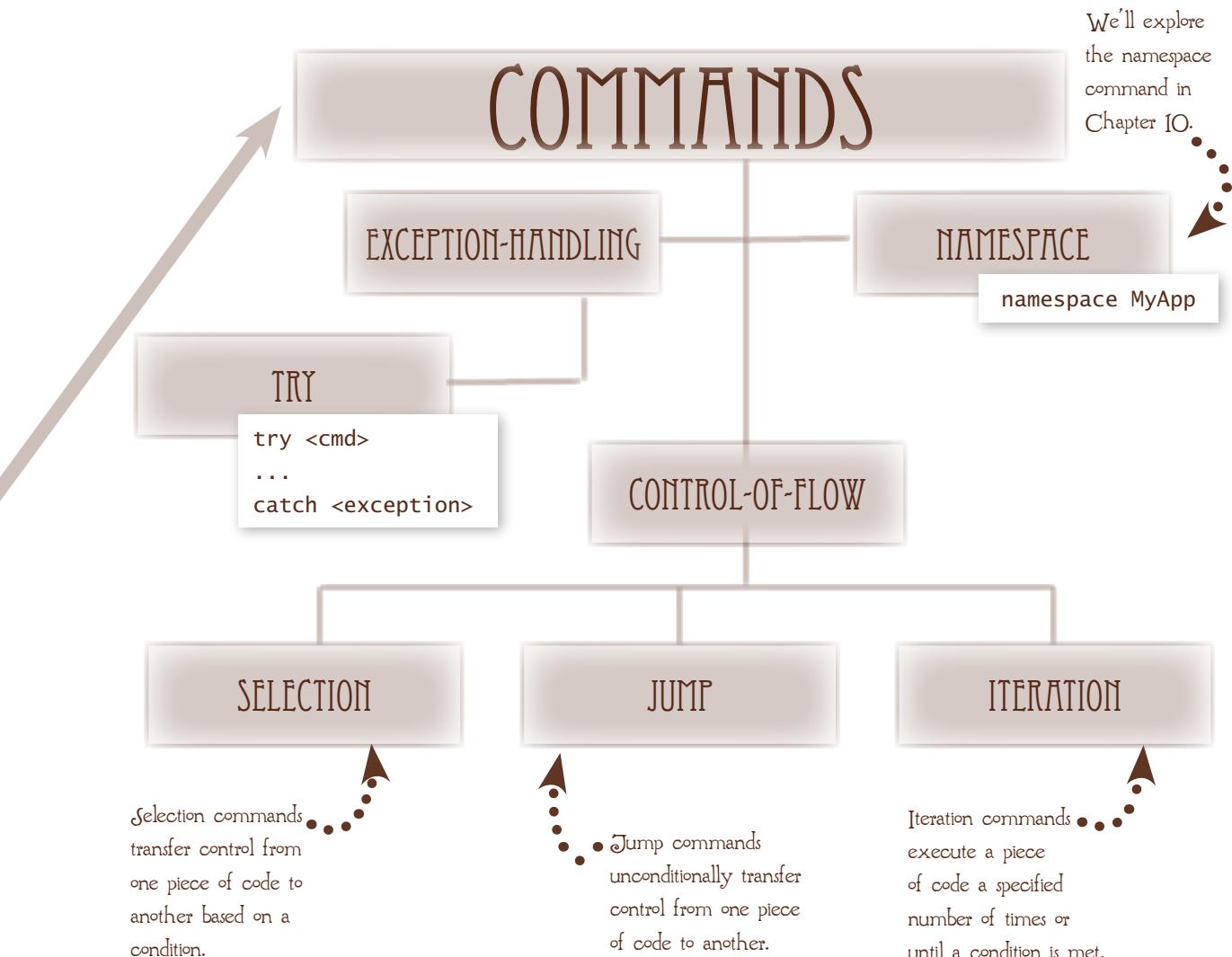


LANGUAGE ELEMENTS

We'll complete our study of the Visual Basic parts of speech in this chapter.



IN A NUTSHELL





TASK LIST

In this chapter we'll get dictatorial and learn how to issue commands to our applications.



CONTROL-OF-FLOW COMMANDS

By default, code begins at the first line and continues, line by line, to the end. But that's hardly ever what you really want to have happen, and in this chapter we'll learn how to control the execution flow using this important category of commands.



EXCEPTION-HANDLING COMMANDS

You probably remember from Chapter 4 that exceptions are problems that you can anticipate but can't avoid. They include things like a missing disk or the weird and wonderful things users tend to do when the application asks them a question. We'll look at what you can do about these situations when we examine exception handling and the `try...catch` command.

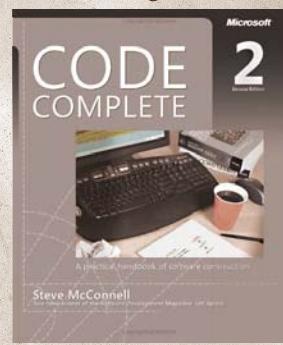


FURTHER STUDY

The .NET Framework has a very clean (some might almost say elegant) mechanism for handling the unexpected in programming and we'll explore it here. The principles are simple.

The practice is less so.

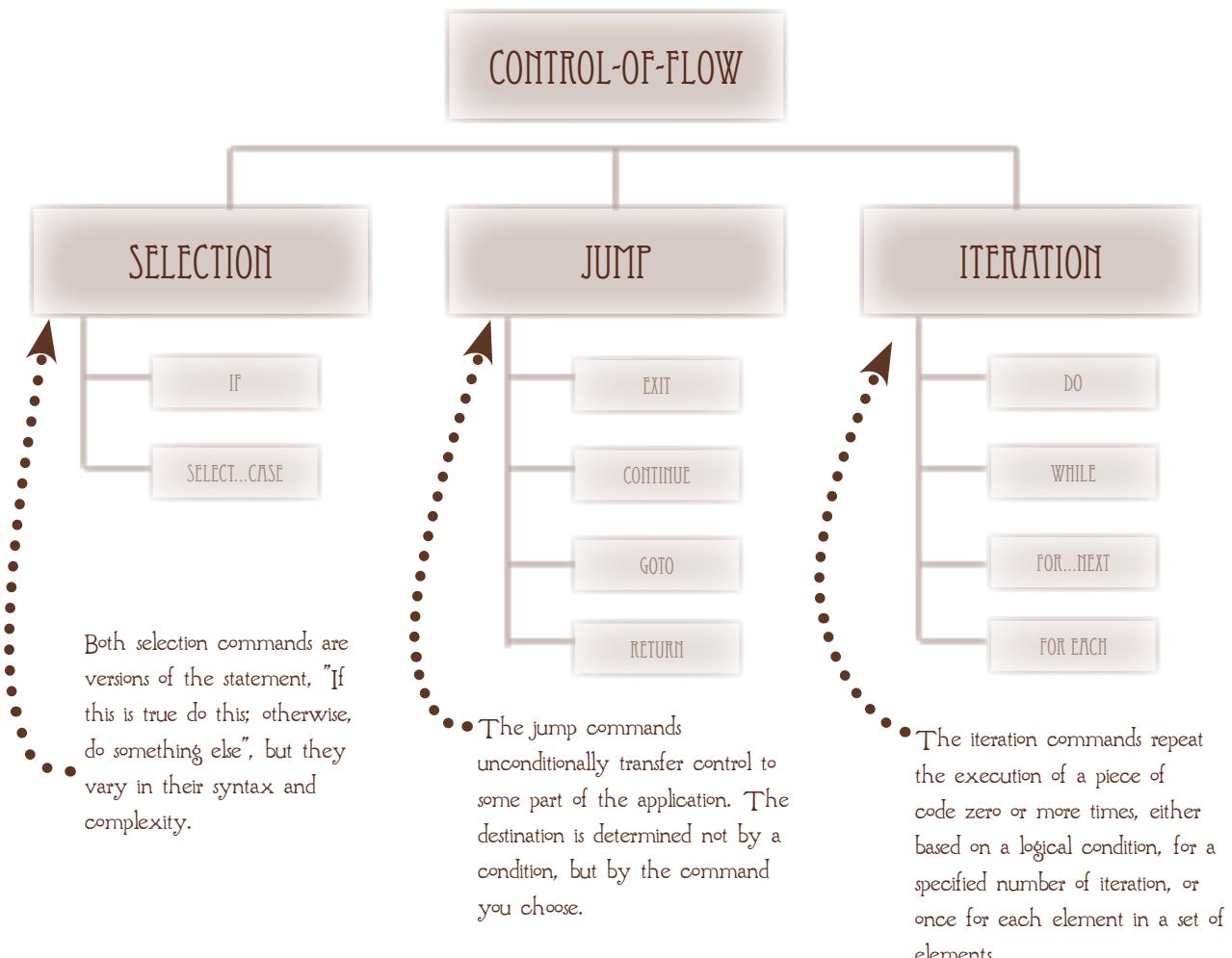
And it's important. The skill with which you protect your application against exceptions is a measure of your ability as a programmer. If you want to explore the subject in more detail, a good place to start is Steve McConnell's *Code Complete*, an excellent resource for defensive programming, and many other aspects of high-quality software construction.





CONTROL-OF-FLOW

The control-of-flow commands control how execution flows from one part of the application to another. (They "control the flow", hence the name.) This set of commands can be divided into three categories. The **SELECTION COMMANDS** decide whether to execute a piece of code based on a logical condition. The **JUMP COMMANDS** transfer control from one place in the application to another. Finally, the **ITERATION COMMANDS** execute a piece of code zero or more times, based on a condition.



SELECTION STATEMENTS

Selection statements let you selectively execute code based on some condition. Visual Basic supports two primary selection statements, If and Select Case, and two selection functions, Choose and Switch. Let's start with the simplest: the if statement.

The If keyword starts the command and it's followed by any Boolean expression.

Then is optional, but customary.

```
If <Boolean expression> [Then]  
    [<statement if true>]  
    [ElseIf <Boolean expression> [Then]  
        [<elseif statements>]  
    Else  
        [<statement if false>]]  
End If
```

The Else and ElseIf clauses are optional. You can have multiple ElseIf blocks but only a single Else.

```
If x > y Then  
    x += 1  
    y = x  
ElseIf x < y Then  
    x = y  
Else  
    z = x  
End If
```



MAKE A NOTE

The If command and the #If directive have almost identical syntax, but they behave very differently. The directive controls the code that the compiler includes in the output. The command controls execution. All of the code involved in an If command is included in the output.



PUT ON YOUR THINKING HAT

To get some practice with the `if` statement, try translating these situations into code:

If `x` is greater than `y`, increment `x`, otherwise set `y` equal to 12:

If `a` is equal to 5, then set `x` equal to 5; otherwise, if `x` is equal to 12, set `y` equal to 21, or if it's not, set `y` equal to `x`:



HOW'D YOU DO?

If x is greater than y , increment x , otherwise set y equal to 12:

If $x > y$ Then

$x += 1$

Else

$y = 12$

End If

If a is equal to 5, then set x equal 5; otherwise, if x is equal to 12, set y equal to 21, or if it's not, set y equal to x :

If $a = 5$ Then

$x = 5;$

ElseIf $x = 12$ Then

$y = 21$

Else

$y = x$

EndIf

Did you find this one a little confusing? Not surprising: it is a little confusing. If statements can be nested as deeply as you like, but they can be hard to keep track of. The switch statement that we'll look at next is sometimes a better option.



MAKE A NOTE

There's also a simpler, single line syntax for the If statement:

If <expression> Then <statement>

This version is useful for simple statements that are unlikely to change, but using it really isn't considered best practice, mostly because if it does change, rewriting it using normal syntax is tedious and error-prone.

THE SELECT...CASE COMMAND

The Select...Case statement compares the condition value to a series of constant values. It's much easier to understand than a complex If statement, but the downside is that the Select...Case isn't quite as flexible: You can only use a single expression for comparison, unlike the ElseIf clause that allows you to check something completely different.

The Case keyword is optional, but customary.

```
Select [Case] <value>
Case <expressionlist>
  <statements>
[Case <expressionlist>
  <statements> ]
[Case Else
  <statements> ]
End Select
```

The expression list can be a single value, multiple values separated by commas, or a range expressed as: <value> To <value>

The test value can be a constant, a variable or an expression that evaluates to one of the elementary data types like Integer or Char.

```
Select Case x
Case 1
  y = x
  z = x
Case 2 To 5
  y = x
  z = x + 1
Case 3, 7, 9
  z = 0
Case Else
  z = 1
End Select
```



PUT ON YOUR THINKING HAT

For each of the statements below, decide whether it would be easiest to write as an If or a Case statement, and then write the statements:

If a is equal to 3, set a to 5. If a is equal to 10, set a to 15. If a is equal to 0, do nothing. If it's equal to anything else, display a MessageBox.

If x is equal to 0, increment x. If y is equal to 5, decrement x. If x is equal to y, set x equal to 10.



HOW'D YOU DO?

Here are my choices. If you made different choices, you aren't necessarily wrong. There are very few absolute rules in programming. Remember, it depends on...it depends.

If a is equal to 3, set a to 5. If a is equal to 10, set a to 15. If a is equal to 0, do nothing. If a is equal to anything else, display a MessageBox:

```
Select Case a  
Case 3  
    x = 5  
Case 10  
    x = 15  
Case Else  
    MessageBox.Show("something")  
End Select
```

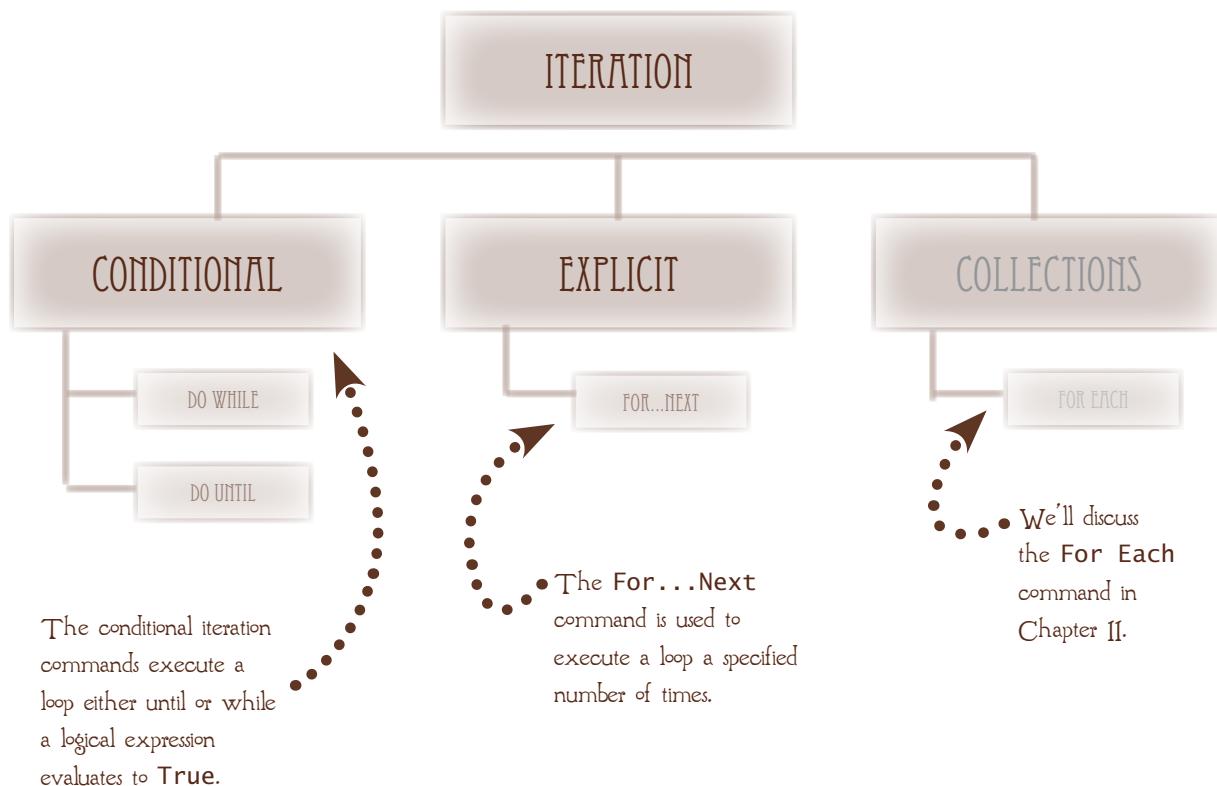
If x is equal to 0, increment x. If y is equal to 5, decrement x. If x is equal to y, set x to 10:

```
If x = 0  
    x += 1  
ElseIf y = 5  
    x -= 2  
ElseIf x = y  
    x = 10  
EndIf
```

You might have thought a `Select...Case` would be better here, but remember that `Select` can only evaluate a single test variable.

ITERATION COMMANDS

The next category of control-of-flow commands are **ITERATION STATEMENTS** (also called **LOOPS**) that cause a block of code to be executed more than once. There are three categories of iteration statements: those that execute a statement block zero or more times based on a Boolean expression, those that execute a statement block a specific number of times, and those that execute a statement block for each element in a collection of elements. (We'll discuss the last category in Chapter II when we look at .NET Framework collection types.)



CONDITIONAL LOOPS

All kinds of conditional looping in Visual Basic are handled with the **Do...Loop** command, which makes remembering the syntax easy but can sometimes make sorting out the logic a little tricky. The **Do...Loop** always decides whether to execute a set of statements based on a Boolean expression, but it can execute them **While** the expression is true, or **Until** the expression is true, and it can perform the test before or after it executes the statements the first time. Let's start with the most common version:

The condition is a Boolean expression.
(Remember that constants like **True** or variables declared as Booleans are a kind of expression to the compiler.)

**Do While <condition>
[statements]
Loop**

The statements inside the loop are executed if and as long as the condition is **True**.

When execution reaches this point, it returns to the top of the loop and the condition is re-evaluated. If the condition is still true, the statements inside the loop are executed again (and again, and again, while the condition is **True**).

If you use the **Until** keyword instead of **While**, the statements are executed as long as the condition is **False**.

**Do Until <condition>
[statements]
Loop**

You can also put the tests at the end of the loop instead of the beginning. The logic is basically the same, but the statements inside the loop are always executed at least once before the condition is tested:

**Do
[statements]
Loop While <condition>**

**Do
[statements]
Loop Until <condition>**



PUT ON YOUR THINKING HAT

If x is 3, y is 7 and z is 0, what's the value of z at the end of each of these loops? In other words, how many times is each loop executed?

Do While $x < y$

$y -= 1$

$z += 1$

Loop

Do Until $x > y$

$x += 1$

$z += 1$

Loop

Do

$z += 1$

Loop While $x > y$

Do

$y -= 1$

$z += 1$

Loop Until $x > y$

Do While $x < y$

$z += 1$

Loop



BEST PRACTICE

Some programmers rely on the fact that the Visual Basic compiler converts any non-zero numeric value to True when they write loops, so you might run across things like Do While MyInteger. Don't do this. It's confusing, unnecessary, and may very well break in a future version of Visual Basic. Instead, use

Do While MyInteger $<> 0$



HOW'D YOU DO?

```
Do While x < y  
    y -= 1  
    z += 1  
Loop
```

This loop executes four times, so $z = 4$.

```
Do Until x > y  
    x += 1  
    z += 1  
Loop
```

This loop executes five times, so $z = 5$.

```
Do  
    z += 1  
Loop While x > y
```

Even though x is less than y and stays that way, this loop executes once before the test, so $z = 1$.

```
Do  
    y -= 1  
    z += 1  
Loop Until x > y
```

This loop five times before the condition is **False**, so $z = 5$.

```
Do While x < y  
    z += 1  
Loop
```

This one was a trick. Because neither x nor y is changed inside the loop, this executes until z exceeds the maximum value of an **Integer**, at which point the program fails.

EXPLICIT ITERATION

The Visual Basic For...Next command is most often used to execute a block of statements a specific number of times. But although there are several options, unlike the Do...Loop command, there's really only one version, and it's pretty straightforward.

The counter has to be a numeric value. If you declare it inside the **For** statement with the **As <type>** option, it will be scoped to the statement. This is a good idea because it eliminates the possibility of accidentally changing a variable value.

The start value sets the initial value of the counter. If you use a variable that you've declared outside the **For...Next** statement, it will be set to this value at the beginning of the loop.

The end value tells the **For...Next** statement when to quit.

It must be a positive value greater than the start value

By default, the counter will be incremented by 1 each time through the loop, but you can increment by a different value by specifying the step.

```
For <counter> [As <type>] = <start> to <end> [Step <step>]  
    <statements>  
    Next [<counter>]
```

```
For x As Integer = 1 to 200 Step 5  
    y += 1  
    Next x
```

When execution reaches the **Next** statement, it adds the step value to the counter (or 1 if no step is specified) and returns to the top of the loop if counter is equal to or less than end.

You don't need to specify the counter variable again at the end of the loop, but it makes it clearer, particularly if you have loops nested inside one another.



PUT ON YOUR THINKING HAT

Write a snippet of code that uses a **For...Next** command to display a **MessageBox** containing the value of the counter twice with the values 0 and 5. Declare the counter inside the statement.



HOW'D YOU DO?

There are lots of ways to solve this one. (There almost always are.) Here are two possibilities, one simple (generally simplest is best) and one complicated (but sometimes necessary):

```
For x As Integer = 0 to 5 Step 5  
    MessageBox.Show(x.ToString())  
Next x
```

OR

```
For x As Integer = 0 to 5  
    If x = 0 or x = 5  
        MessageBox.Show(x.ToString())  
    End If  
Next x
```

Any statement can be nested inside a conditional or explicit loop. You just need to make sure that it's completely inside the loop. The `End <whatever>` can't be after the `Next`.



TAKE A BREAK

The iteration commands are really important; you'll use them a lot. But they're also pretty simple. So why don't you take a break before you complete the Review and move on to the jump commands that transfer control unconditionally?



REVIEW

Can you decide which command would be simplest to use in each of the following conditions?

A statement block should be executed at least once and repeated until a condition is true.

A variable should be set to one value if the condition is true or another value if it's false.

Different blocks of code should be executed based on a constant value of a variable.

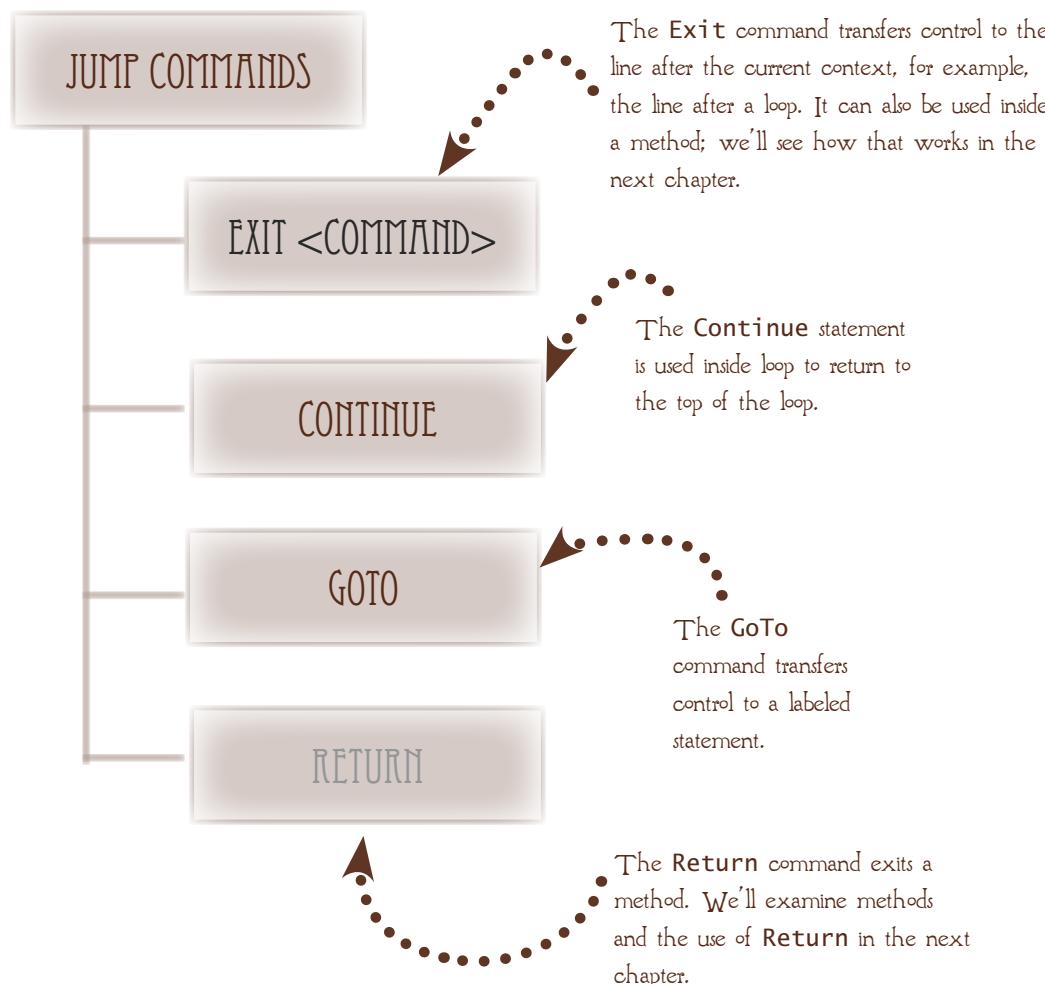
A statement block should only be executed if and until a condition is true.

A statement block should be executed a fixed number of times.

A statement block should only be executed if a condition is true.

JUMP STATEMENTS

All of the control-of-flow commands we've examined so far rely on the evaluation of a Boolean expression. Even the `For` statement evaluates the counter value for equality. The jump commands that we'll consider next are unconditional. They just do it.



LET ME OUT!

The `Exit` command is used inside loops to exit them unconditionally. (It can also exit methods. We'll see how that works in the next chapter.) All of the jump statements we'll be examining should be used with caution, because whenever there is more than one way out of a structure things start to get confusing. But used correctly, they can make your code more efficient and much easier to understand. Let's look at an example of the kind of problem the `Exit` statement can solve:



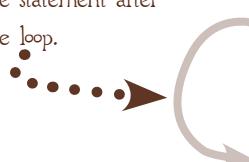
```
For x As Integer = 1 to 100
    a += x
    If (a > b) Then
        a.DoSomething()
    End If
Next x
```

To understand what's wrong with this code, ask yourself some questions:

- How many times does this loop get executed if `b` is equal to 5? How many times does it get executed if `b` is equal to 100?
- Does the value of `b` affect the value of `a` at the end of the loop?

Now compare a version that uses `Exit` and ask yourself the same questions:

If `a` is larger than
`b`, execution is
transferred directly to
the statement after
the loop.



```
For x As Integer = 1 to 100
    a += x
    If (a > b) Then
        Exit For
    End If
    a.DoSomething()
Next x
```

Notice that you use
the keyword for the
structure you want to
leave after `Exit`.

In the second example, the loop is exited as soon as `a` is greater than `b`. If `b` has a low value, that can save a lot of iterations, and in complex code could produce a noticeable increase in performance. Also, because the number of iterations through the loop is controlled by the value of `b`, the value of `a` isn't always going to be 100. (Can you figure out what it will be?)

(SKIP AND) CONTINUE

The `Exit` command transfers execution to the first line after the loop that contains it. The `Continue` command has an almost opposite effect: Instead of transferring control down and out, it transfers it up and in. The `Continue` statement transfers controls to the top of the loop statement, skipping any other statements in the block. Let's see an example:

```
For x As Integer = 1 to 100
    If a <> y Then
        a += x
    If a <> z Then
        If a <> q Then
            a.DoSomething();
        End If
    End If
    End If
Next x
```

Deeply nested structures are intrinsically hard to understand because they require keeping track of too many things at once.

In this example, in order to understand how the code gets to `a.DoSomething()`, you have to keep track of `y`, `z` and `q`.

You can forget about `y` after this line.

```
For x As Integer = 1 to 100
    If a = y Then
        continue . You can forget about
    End If      z and q after this line.
    a += y
    If a = z Or a = q Then
        Continue For
    End If
    a.DoSomething()
Next x
```

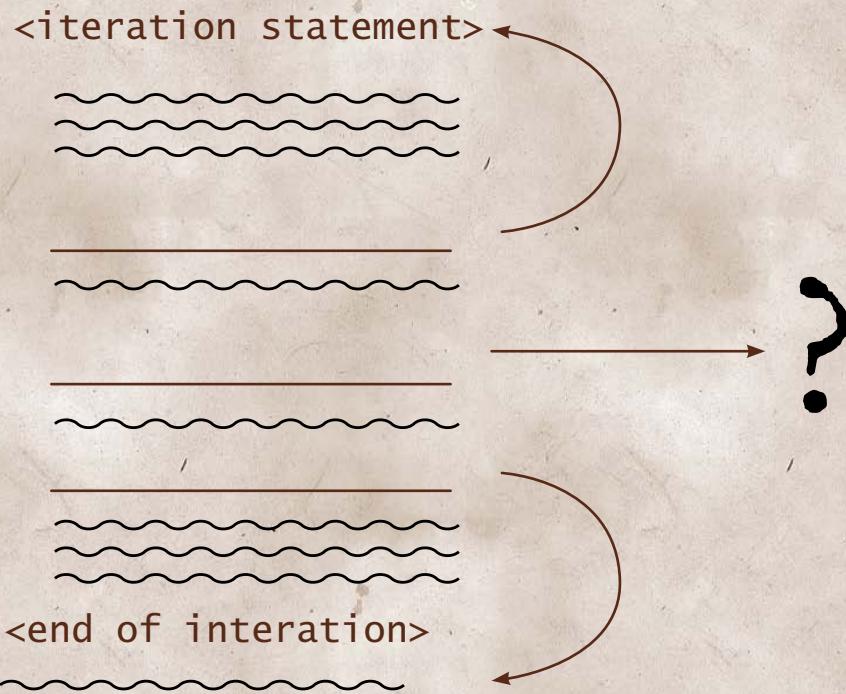
By rewriting the loop using two `Continue` commands and combining two of the Boolean expressions using a logical or, by the time you get to `a.DoSomething()`, you only have to think about `a`.

Unlike the example on the previous page, the first code snippet here isn't wrong; it's just not as good as it could be. The loop will still execute 100 times, but in this case, that's exactly what you want. The `continue` command is often used in this way, to reduce the complexity of your code by eliminating nested `If` statements.



PUT ON YOUR THINKING HAT

Can you label each arrow in the diagram below with the correct statement? (We haven't looked at the one that goes...somewhere...yet.)



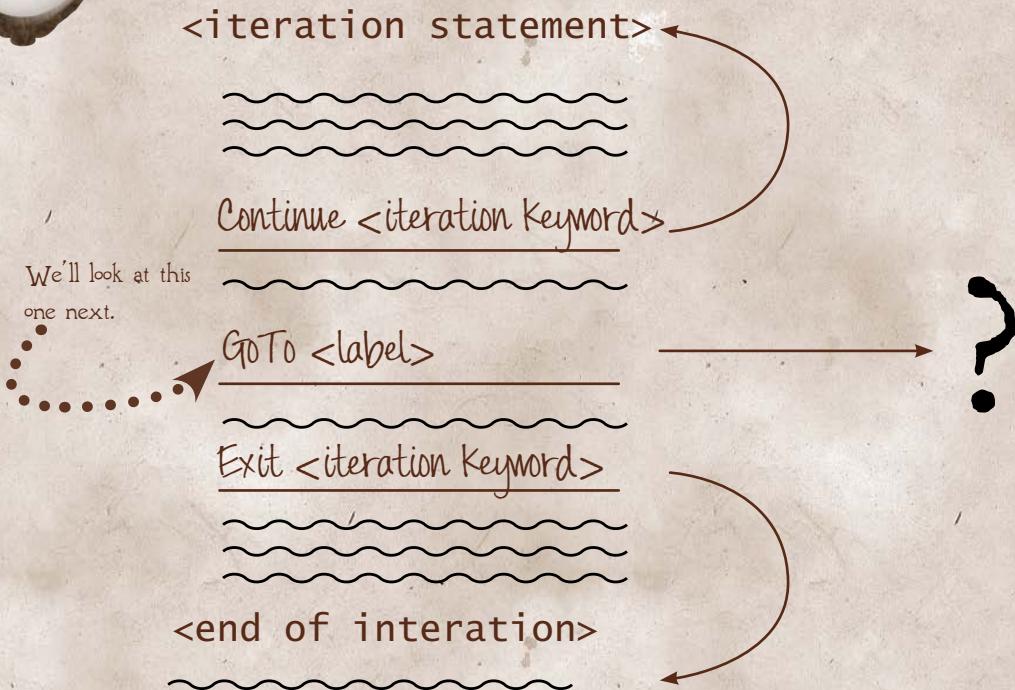
Remember that you can add the control variable after a **Next** statement. Doing so is optional, but a really good idea. With the **Continue** and **Exit** statements, adding the keyword for the controlling loop isn't optional. What would the actual syntax be in each of the following situations?

To transfer control to the top of a **For** loop, you'd use:

To unconditionally leave a **While** loop, you'd use:



HOW'D YOU DO?



Remember that you can add the control variable after a **Next** statement. Doing so is optional, but a really good idea. With the **Continue** and **Exit** statements, adding the keyword for the controlling loop isn't optional. What would the actual syntax be in each of the following situations?

To transfer control to the top of a **For** loop, you'd use:

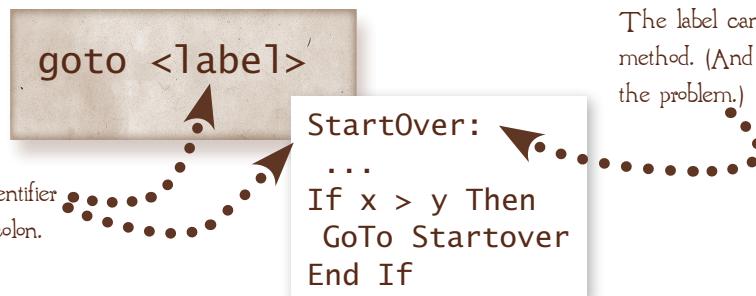
Continue For

To unconditionally leave a **While** loop, you'd use:

Exit While

GOTO....WHEREVER

The last of the jump statements is `GoTo`. Now the `GoTo` statement gets a lot of bad press, and you will often hear people say that it's the sign of poorly designed code. (It often is.) But you may also hear it said that `GoTo` is "evil" and its use should be avoided at all cost. That's not entirely fair. It's true that `GoTo` is a heavy-handed tool and, like a chainsaw, you should use it with caution. But you wouldn't trim a tree with a carving knife, and there are some situations (admittedly, only a few) where `GoTo` is an effective tool. The syntax is simple:



PUT ON YOUR THINKING HAT



There's really only one situation in which the `GoTo` is useful: It can eliminate a whole bunch of repetitive tests in deeply nested loops. To get a sense of how that works, can you rewrite this snippet without using a `GoTo`?

```
For x As Integer = 1 to 10
    While y < x
        Do
            If a = b Then
                GoTo EndOfLoops
            EndIf
            a += y
        Loop Until q = r
        y.DoSomething()
        y += 1
    End While
Next x
EndOfLoops:
...

```



HOW'D YOU DO?

Here's a solution without GoTo. (I think.) It's so complicated, I'm sure I got confused somewhere.

```
For x As Integer = 1 to 10
    While y < x
        Do
            If a = b Then
                Exit Do
            End If
            a += y
        Loop Until q = r
        If a = b Then
            Exit While
        Else
            y.DoSomething()
            y += 1
        End If
    End While
    If a = b Then
        Exit For
    End If
Next x
```

Here's what we started with:

```
For x As Integer = 1 to 10
    While y < x
        Do
            If a = b Then
                GoTo EndOfLoops
            EndIf
            a += y
        Loop Until q = r
        y.DoSomething()
        y += 1
    End While
Next x
EndOfLoops:
...
```



TAKE A BREAK

That's it for the control-of-flow statements. Why don't you take a break before we move on to exception handling?



EXCEPTION HANDLING

When we explored the support that Visual Studio provided for debugging applications in Chapter 3, I said that there were bugs that you need to try to eliminate before your application is deployed, and that exceptions were things that happen at runtime you can predict but not always avoid.

Stuff happens. We all know that. It's the mark of a well-designed and well-implemented application that it doesn't pitch a fit when something happens that it wasn't expecting.



PUT ON YOUR THINKING HAT

Here's a list of things that can go wrong when you're programming. Which ones do you think should be handled as exceptions?

An external file can't be found.

A method call is missing a parameter.

The application asked for a number, but the user entered a string.

A variable passed to a method call contains the wrong type of data.

The application doesn't do what you meant to tell it to do.



HOW'D YOU DO?

Some of these could be either exceptions or bugs, depending on the larger context of the application, so don't worry if your answers don't exactly match mine. The point is just to think about what kind of thing you need to anticipate with exception handling.

An external file can't be found.

Exception



This is the classic example of an exception.

A method call is missing a parameter.

Bug



This is a bug, and you'll catch it in the editor.

The application asked for a number, but the user entered a string.

Exception



- You have some control over this in the way you design the application's interface, but expect it to happen anyway.

A variable passed to a method call contains the wrong type of data.

Exception



- Even though you declare the type of data you expect when you define a method, sometimes the wrong value can slip in anyway.

The application doesn't do what you meant to tell it to do.

Bug

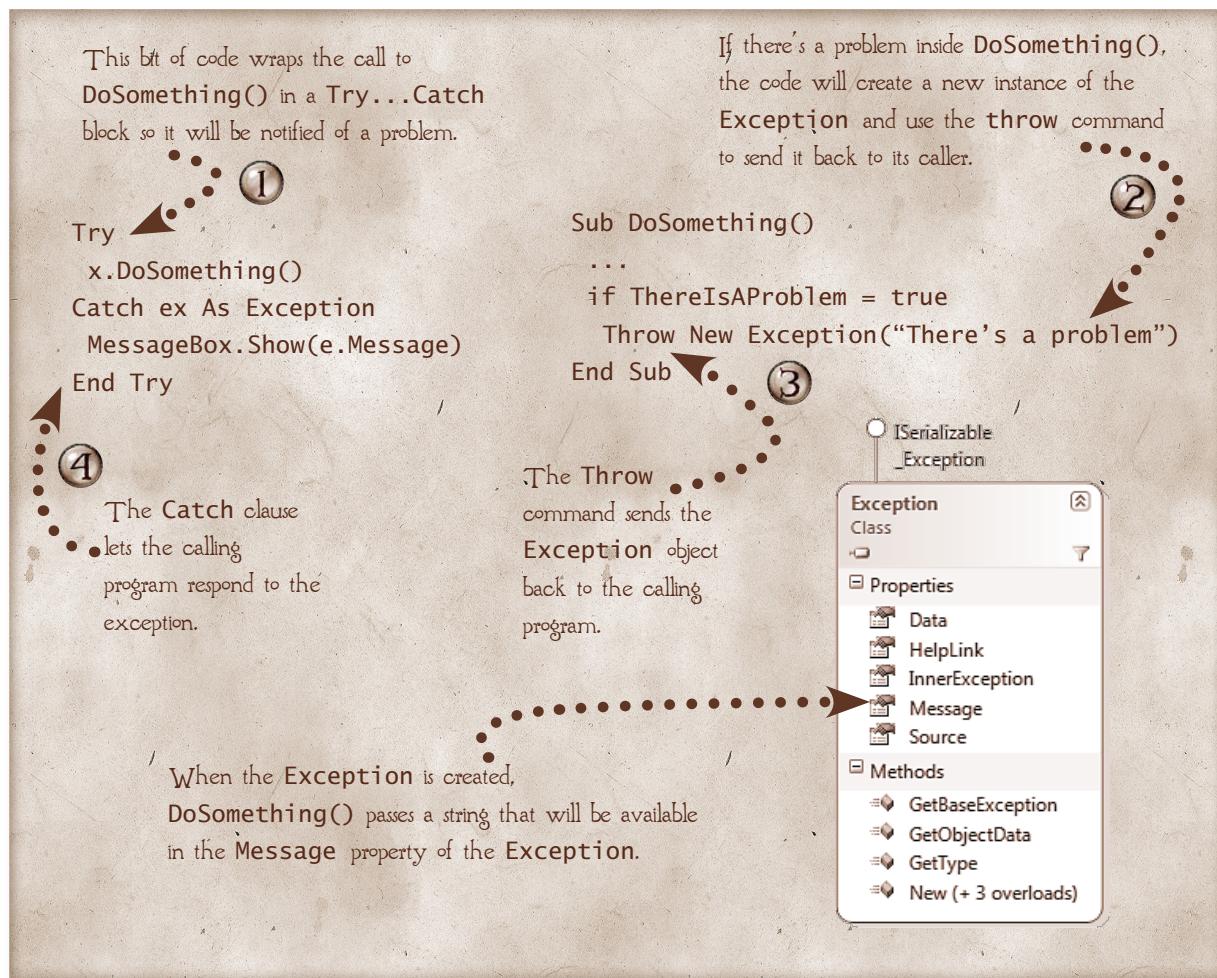


This is probably a logical error.

STRUCTURED EXCEPTIONS

Visual Basic and the .NET Framework provide an exception-handling mechanism called STRUCTURED EXCEPTION HANDLING that makes it much simpler to deal with exceptions, and much more likely that they'll be handled in a consistent way. (Unfortunately, you don't always have to use it, so it does still require some discipline. The days of ugly Windows error dialogs aren't completely gone.)

.NET structured exception handling relies on a set of classes (`System.Exception` and its children) and two Visual Basic commands: the `Try...Catch...Finally` command that lets you handle problems gracefully and the `throw` command that generates an `Exception` when your code encounters a problem.



TRY...CATCH...FINALLY

Any statements that might generate an exception are wrapped inside the Try block.

Exceptions, if they occur, are handled inside the Catch block.

You can have more than one Catch block to handle different kinds of exceptions. They act sort of like a Select...Case statement. Once an Exception type is matched, all the other Catch blocks are ignored.

The Finally block is optional, but if it's present, the code it contains is always executed, whether or not an exception occurs.

The statements inside the finally block are used to clean up things whether or not there's an oops.

```
Try  
<statements>  
Catch <name> As <type>  
<statements>  
[Catch...]  
[Finally  
<statements>  
]  
End Try
```

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 "ex".

The specific type of exception to be handled is specified in the Catch clause.

```
x.Open()  
Try  
x.DoSomething()  
Catch ex As Exception  
Throw ex;  
Finally  
x.Close()  
End Try
```

You can rethrow the exception within the catch block to pass it up the line.

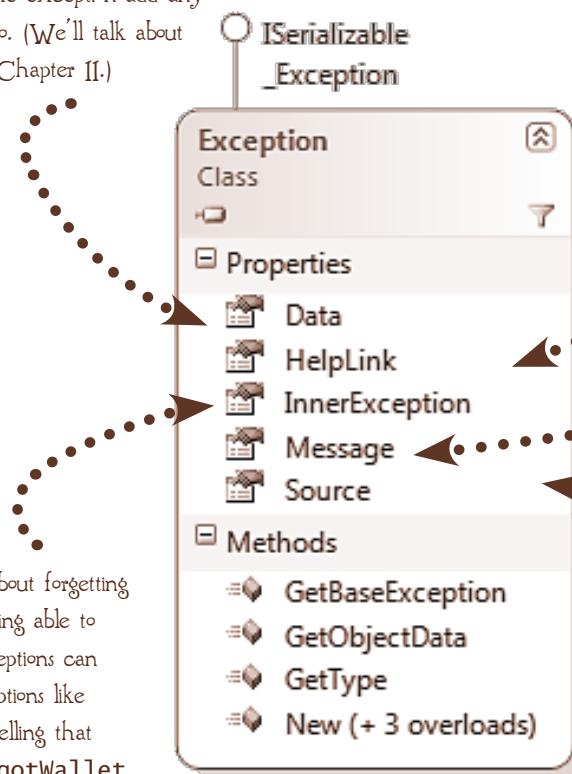
In this example, x needs to be closed whether or not an exception occurred, so the x.Close() statement is in the finally block.

THE EXCEPTION CLASS

You learned how to call methods and use the dot operator to access the members of a class in the last chapter. (And it's okay if you don't feel very confident about that statement yet. Using classes really is as simple as the dot operator and passing arguments. Well, okay, *almost* that simple.)

But you haven't done much work with a real live class yet, so let's have a look at the **Exception** class. Here's a partial class diagram that shows some of the important members:

The **Data** property lets the code that throws the exception add any data it needs to. (We'll talk about dictionaries in Chapter 11.)



The **HelpLink** property is used to link the **Exception** to an external help file.

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.

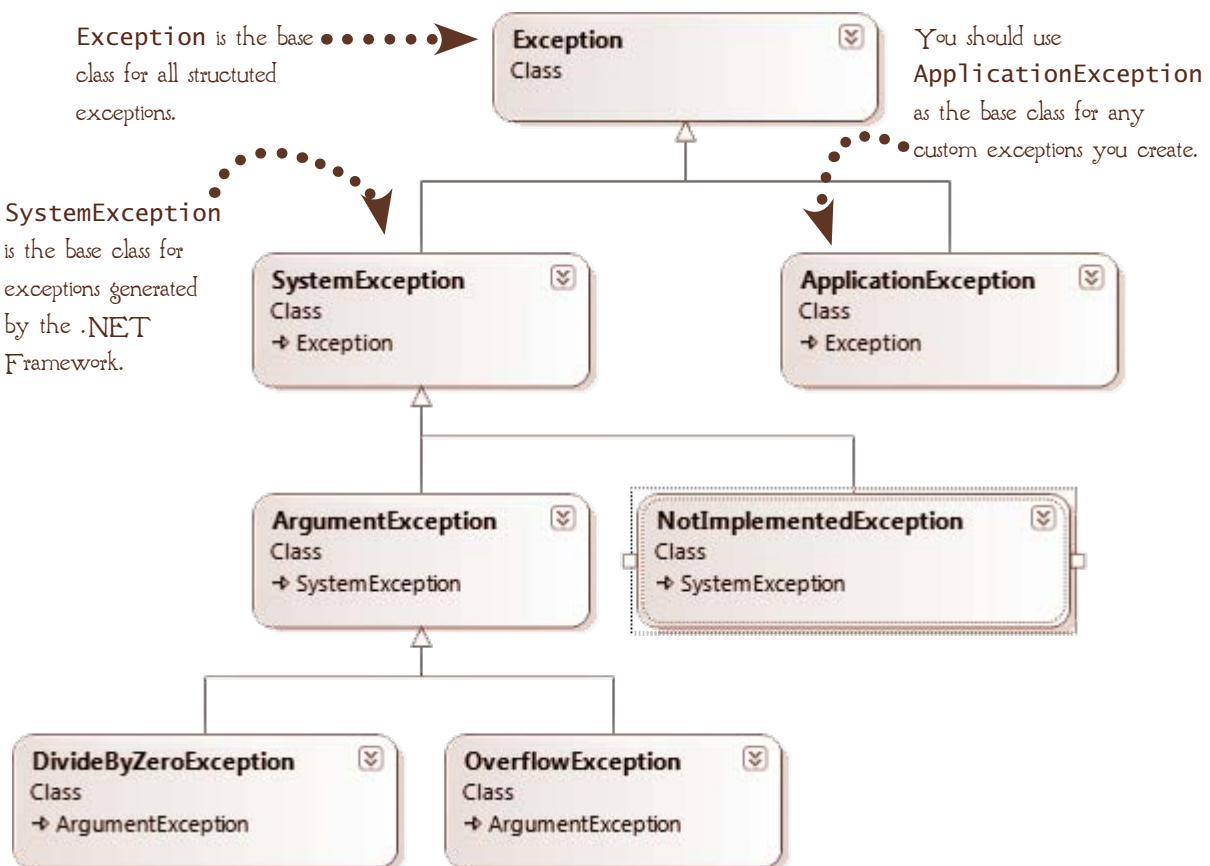
Remember the story about forgetting your wallet and not being able to pay at the store? Exceptions can often cause other exceptions like that. If you were modelling that example in code, `ForgotWallet` would be the **InnerException** (it happened first) inside the `InsufficientFunds` exception.

The **Source** property is the object, routine or application that generated the **Exception**. Again, this may be displayed to the user, so be kind (and check your spelling).

EXCEPTION'S CHILDREN

There are hundreds, perhaps bazillions, of exception classes defined in the .NET Framework library. Most of them have the same properties and methods, but the class names themselves help you (and your code) understand what's gone wrong. (We'll talk about designing and implementing your own classes in detail starting in the next chapter.) For now, you need to know that classes can inherit from one another in a hierarchy and that you can always treat a child class as though it were the parent, but the reverse is not true. That means that whatever you catch, you can just treat it as an `Exception`.

The diagram below shows a few of the exception classes you're likely to work with. The arrows show that, for example, `SystemException` inherits from `Exception`, and `OverflowException` inherits directly from `ArgumentException`, indirectly from `SystemException` and, ultimately `Exception`.





PUT ON YOUR THINKING HAT

I've been talking, talking, talking at you. Time for you to go to work.

Assume that `MyObject.MyMethod()` can throw one of two exceptions: a base `Exception` or a `ParseException`. Write a code snippet that wraps a call to `MyMethod()` in a `try...catch` block, displays the `Message` property of a `ParseException` in a `MessageBox` and rethrows a general `Exception`.

The exception classes expose a `Message` property that contains a user-readable description of the problem that occurred, but the names themselves also tell you a great deal. Under what circumstances do you think each of these exceptions might be thrown?

`DivideByZeroException`

`ArgumentException`

`OverflowException`

`NotImplementedException`



HOW'D YOU DO?

You didn't cheat if you looked up the exceptions in MSDN.

Assume that `MyObject.MyMethod()` can throw one of two exceptions: a base `Exception` or a `ParseException`. Write a code snippet that wraps a call to `MyMethod()` in a `try...catch` block, displays the `Message` property of a `ParseException` in a `MessageBox` and re-throws a general `Exception`.

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.

```
Try
    MyObject.MyMethod()
    Catch pe As ParseException
        MessageBox.Show(pe.Message)
    Catch ex As Exception ex
        Throw ex
    End Try
```

DivideByZeroException

The divisor is zero.

In mathematics, division by zero is undefined. In the world of computers, it's an exception.

Even if the argument is the right type, it can still contain the wrong data. The code might have expected a `String` date in the form "mm-dd-yyyy" and get "today", instead.

ArgumentException

An argument passed to the method is invalid.

OverflowException

The result of division is too large to fit in the variable to which it's being assigned.

Remember the elephant and the goldfish? This is the exception that gets thrown when an expression returns an elephant, but it was expecting a goldfish.

NotImplementedException

The method has not been implemented.

This is an interesting one. It's thrown when a method (or some other routine) has been defined but not yet implemented. When a Visual Studio Designer is creating code for you, it will add a `NotImplementedException` any place that code is required, but the Designer can't know what that code needs to do.



MAKING EXCEPTIONS

The application of structured exception handling, the wheres and whys of it, can be quite complex (and sometimes contentious) but the implementation isn't. Let's give it a try:



Start by creating a new WPF solution and adding a button to the main window.



Add the following code to the button1_Click event handler:

```
Dim a As String  
Dim x As Integer  
  
a = "1"  
x = Int32.Parse(a)  
  
MessageBox.Show(x.ToString())
```



Run the application by pressing F5, just to make sure there aren't any problems with this code.



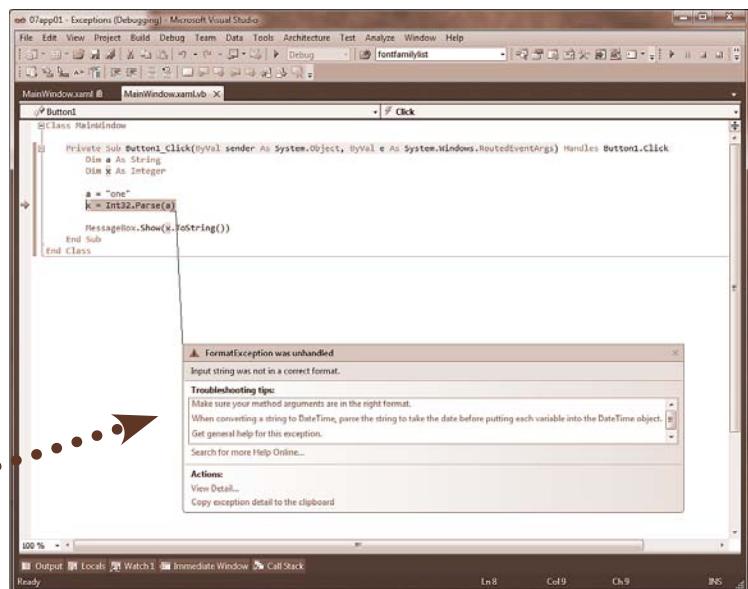
Change the value of a from “1” to “one”:

```
a = "one";
```



Rerun the application. When you click the button, the application will fail with an error message:

In a release version
of the application,
Windows would
display a version of this
error to your users. Not
very helpful, is it?

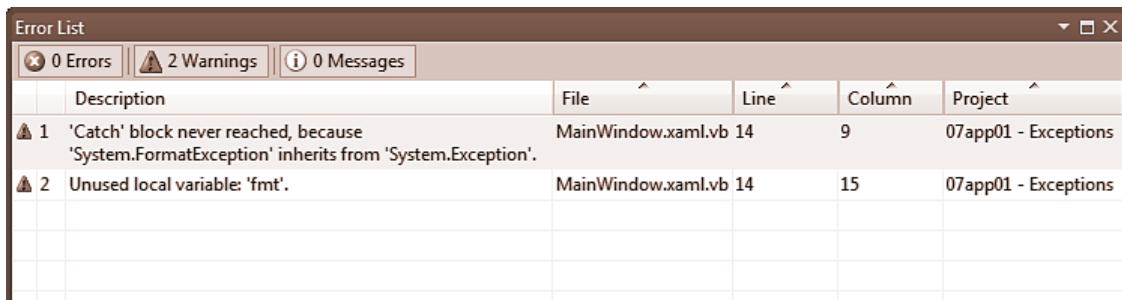


Close the error dialog and stop the application. Add a Try...Catch block to handle the exception with a little more grace. My version is on page 216, but try doing it on your own before you peek.



In your Try...Catch statement, you probably used ex As Exception in the catch block, which catches the base System.Exception class and any of its descendants. But if you remember, the syntax of Try...Catch allows multiple catch blocks that declare different exception types. Let's try that now.

Add a Catch clause to handle the generic exception and the FormatException being generated by our error. If you add after the general exception Catch block, Visual Basic will display the following error:

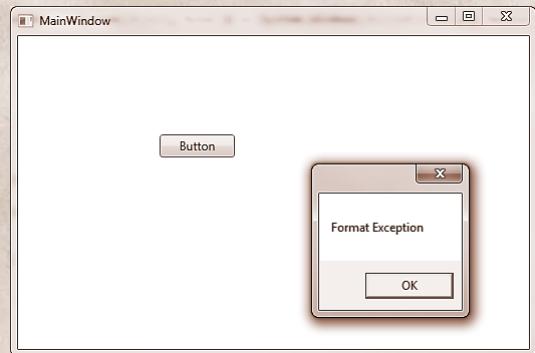


PUT ON YOUR THINKING HAT



Can you figure out what the problem is? Remember, the Try...Catch block works like a Select...Case statement in that it finds the first match and then quits.

Fix the problem (if you get stuck, my version is on the next page), and run the application to verify that it works the way it should now.





HOW'D YOU DO?

To fix the catch block:

Catch blocks must be ordered from most to least specific. To fix the problem, just swap them.

Step 5 code:

```
Dim a As String
Dim x As Integer

a = "one"

Try
    x = Int32.Parse(a)
Catch fmt As FormatException
    MessageBox.Show("Format Exception")
Catch ex As Exception
    MessageBox.Show("The String does not contain a number.")
End Try

MessageBox.Show(x.ToString())
```



ON YOUR OWN

What class do you think would be best to throw in each of the following situations? HINT: The easiest way to get a list of exceptions is to search for SystemException in the Object Browser, and then expand the Derived Types node.

The argument passed to a method is of the correct type but in the wrong format.

An operation has taken longer to complete than the time allowed for it. (This is referred to as a TIMEOUT, not to be confused with the recommended response to an unruly child.)

One of the arguments passed to a method is invalid.

There isn't enough memory available to complete an operation.

Something has gone wrong with an attempt to print.



TAKE A BREAK

That's it for Visual Basic commands and our discussion of the language syntax. Don't worry, though; you'll get lots more practice working with language as we go through the rest of the book.

For now, though, why don't you take a break before finishing up the chapter Review?



REVIEW

Which command would be appropriate in each of these situations?

To skip the remaining statements in a loop:

To repeat a statement block as long as a Boolean expression remains true:

To execute a statement block a fixed number of times:

To return one value if a Boolean expression is true but a different value if it's false:

To capture and handle an exception:

To exit a loop:

To execute a statement block while a Boolean expression remains true:

To execute one block of code based on the value of a variable:

To execute a statement or statement block based on a Boolean expression:

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



CLASSES IN THE .NET FRAMEWORK

8

We've been working with classes for the last several chapters. You've seen that they are programming constructs that have state (fields and properties) and behavior (methods and events), that they descend from one another in a hierarchy, and that you can access their members with the dot operator.

But you may not think you know what any of that actually means yet, and that's okay. We've only looked at these things very quickly so far, because we were concentrating on other parts of the application development process. In this chapter, we'll start looking at exactly what classes are and how you go about creating and using them.

ON YOUR OWN



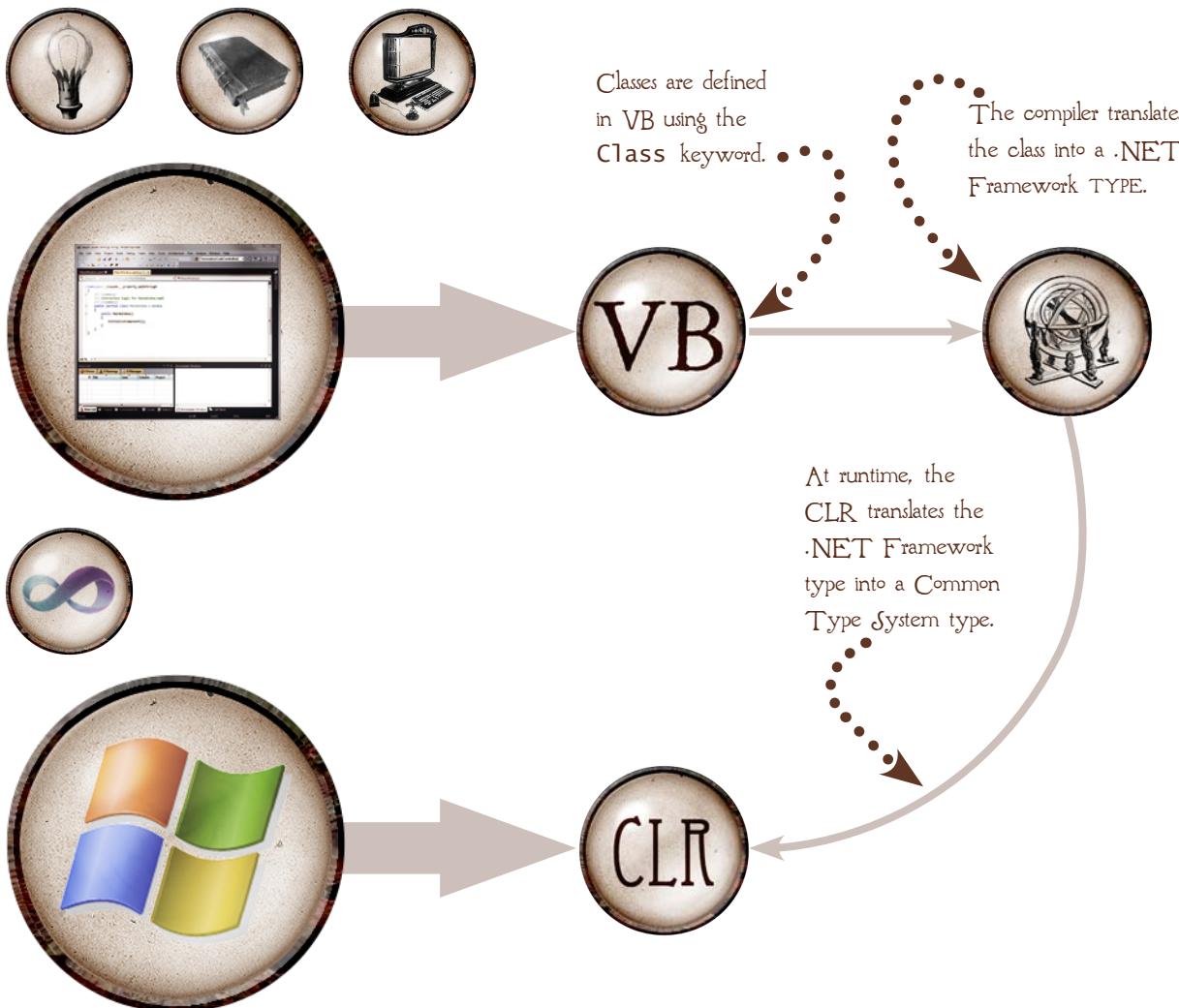
Objects INSTANTIATE classes. That is, they are INSTANCES of them. The statement below contains both a class and an object. Which do you think is which?

```
Dim Message As String
```



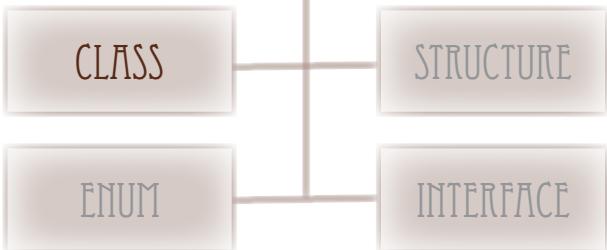
FITTING IT IN

Classes are part of the .NET Framework type system that also includes interfaces, structures, and enumerations. They're implemented using Visual Basic type and member keywords (of which `Class` is one) and translated by the compiler into .NET Framework types and by the CLR into types in the Common Type System (CTS). In this chapter, we'll examine the member keywords as they relate to classes. In the next chapter, we'll look at the remaining .NET Framework types.



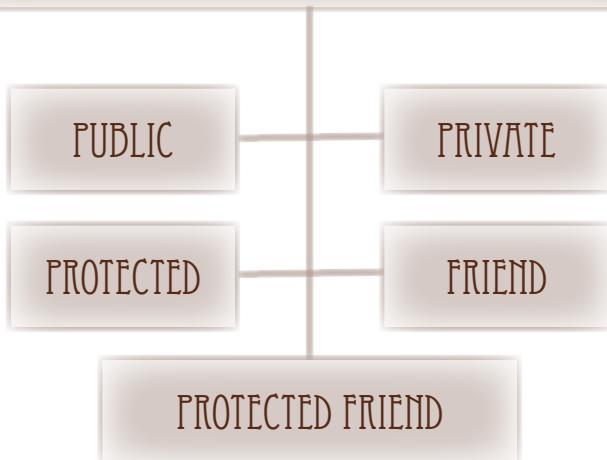


TYPE KEYWORDS



The type keywords are used to tell the compiler what kind of CLR type you're defining.

ACCESS MODIFIERS



We'll discuss the other CLR types in the next chapter.

Access modifiers control where a type or member is visible and whether it can be changed by child classes.

DISSECTING CLASSES

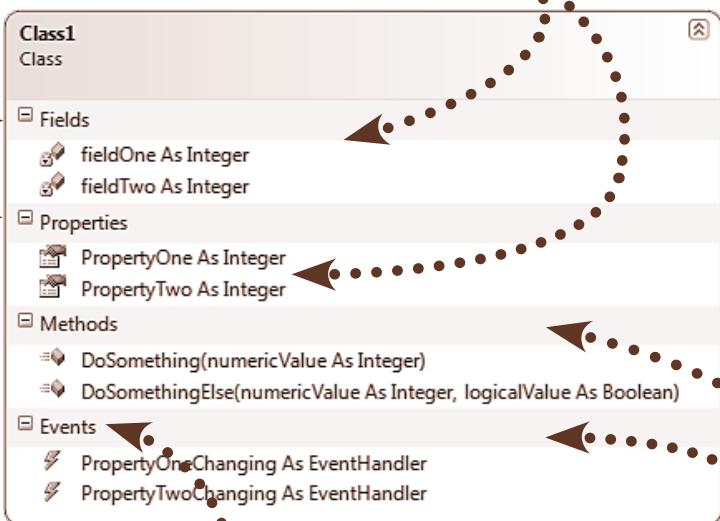
FIELDS are simple variables. You usually won't let these be accessed outside the class itself. Instead, you'll wrap the field in a property so that you can control what happens to it.

PROPERTIES contain information, like fields, but you can control how that information is changed. Properties are often, but not always, made available outside the class.

METHODS are routines that accept parameters and may or may not return a value. `MessageBox.Show()` and `Int32.Parse()` are examples of methods that we've already used.

So far, all the code we've written was in the event handler for the `Button.Click` event.

Fields and properties represent the state of the class, the things it knows about like the name of an elephant or the color of a fish.



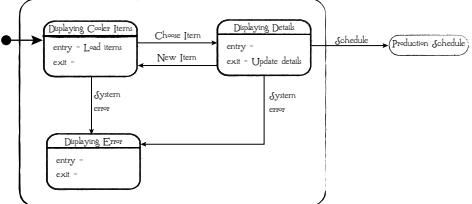
METHODS and EVENTS represent the behavior of a class: the things it can do and respond to.

EVENTS are things that happen while an application is running, like a user clicking a Button or a property being changed. The class PUBLISHES the event. Other classes can SUBSCRIBE to the event by writing code to handle (respond to) it.

AN EXAMPLE...

Way back in Chapter 1, we built a UML Use Case diagram for one of the applications that Neil Gordon needs. Gordon wants to be able to see the dessert components that are available so that he can decide on the dessert menu and production schedule for the following day.

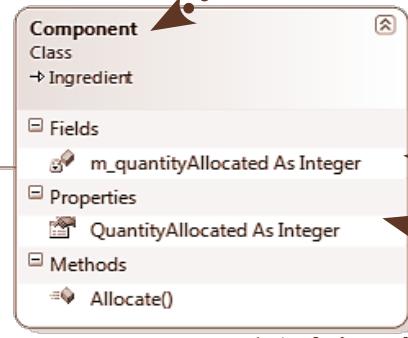
Here's another kind of UML diagram: the Class diagram. This one shows two of the classes that we'd need to build the application for Gordon. I built the diagram in the Visual Studio Class Designer, which, like all of Visual Studio's designers, lets us think about the problem visually and also builds a lot of the code behind the scenes.



We won't be implementing `Recipe` in this chapter, but we'll need it later for the `Component` class.



The `Component` class **INHERITS** from the `Ingredient` class, that is, a `Component` is AN `Ingredient`. That's shown by the arrow and class name here.



If the base class is shown on the diagram, inheritance is also shown with an open arrowhead like this.

Each kind of member is shown in a separate area within the class box. The areas are called **COMPARTMENTS**.

We won't look at implementing events in this chapter; we'll look at how that all works in Chapter 14 when we look at the Observer pattern.

We'll look at inheritance in detail in Chapter 12. For now, you need to know that the child class will have all the members of the parent or base class (and the base class of its base class, and so on) and can add new ones.



TASK LIST

The .NET Framework supports types other than classes, but the other types expose the same kinds of members and are implemented in similar ways, so in this chapter we'll begin our examination of the type system with the ubiquitous class type—how they're created, how they're implemented, and how they communicate.



CLASS DESIGNER

We'll begin by examining the Visual Studio Class Designer, available in Standard editions of Visual Studio and above, that lets you create a UML Class diagram and then writes the basic code for you.



CLASS DEFINITIONS

After the Class Designer has done the grunt work, we'll look at how VB implements a class using the `class` keyword and specifying inheritance.



FIELDS AND PROPERTIES

The state of a class is represented by its fields and properties, so after we look at the `class` statement, we'll look at how these are implemented in VB code.



METHODS

Once we've learned how to represent STATE, we'll turn to BEHAVIOR. We'll look at the most basic of code constructs, the method, and in particular two categories of methods, instance methods and constructors.



THE CLASS DESIGNER

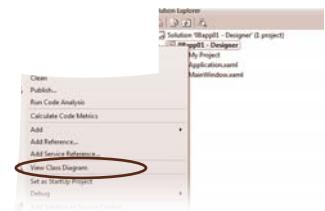
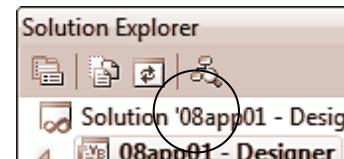
Well, now that we know what we're doing in this chapter, let's get started with a walk-through. We'll use the Visual Studio Class Designer to create a UML Class Diagram and the rough outline of our code.



Create a new WPF Project, add a button to the MainWindow, save it and close the Designer and Editor windows. (We'll come back to them when we start looking at code in the next section.)

Next we'll add a class diagram to the project. Like almost everything in Visual Studio, there are several ways to do it.

- Any of the techniques that you've learned to add a new item will create a blank diagram.
- Selecting one or more class files in the Solution Explorer (those are the ones that end in .vb) and choosing View Class Diagram from the context menu or the Solution Explorer toolbar will add a diagram for the selected items.
- Selecting a Project and choosing View Class Diagram from the context menu or the toolbar will add a diagram that contains all the classes in the Project.



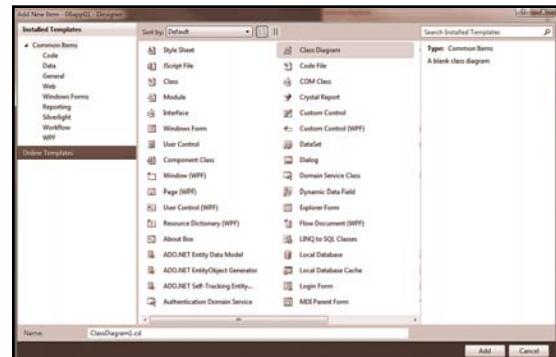
MAKE A NOTE

If you're using the Express version of Visual Studio, you won't have access to the Class Designer. You can just read through these exercises to decide whether you think you'll actually miss what you're missing. Personally, I think the Class Designer is a great tool, worth upgrading for, but lots and lots of people do excellent work without it.

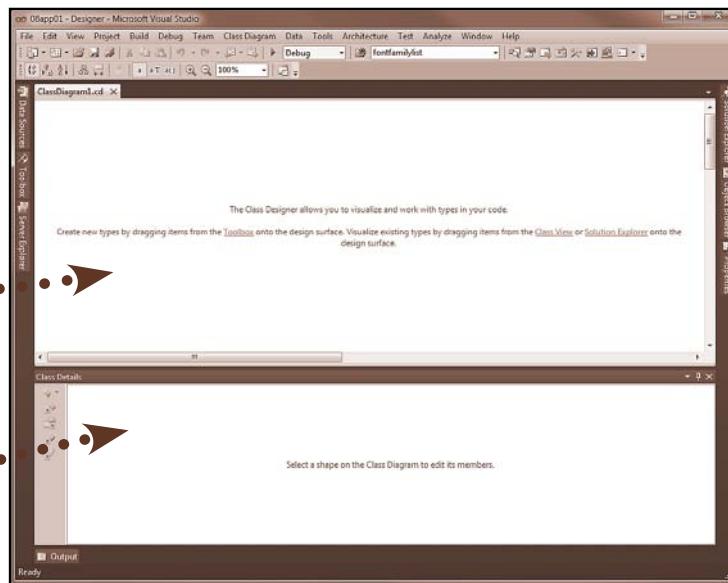
THE CLASS DESIGNER WINDOW



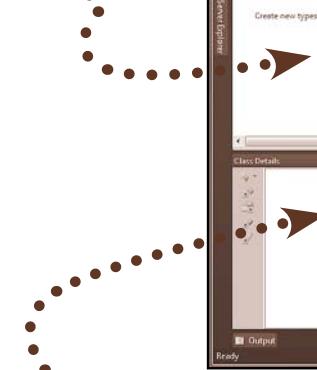
We want to start with an empty diagram, so add a new item using any of the techniques you know, and choose Class Diagram from the dialog. You can call it whatever you like.



After you add the class diagram to the project, it should open in a tab like this:



Just like the WPF
Designer, you'll drag and drop on this surface.



You'll create class

members in the Class Details



Not all versions of Visual Studio have the same defaults,
so if you don't see the Class Details at the bottom, choose
Other Windows, Class Details from the View menu.

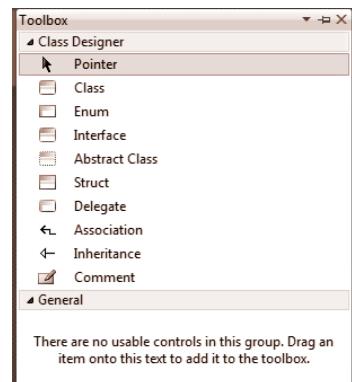
ADDING CLASSES

When you have a class diagram displayed, the Toolbox displays the items that you can drag and drop onto the diagram surface. The items available are all the types that the Class Designer knows how to create.

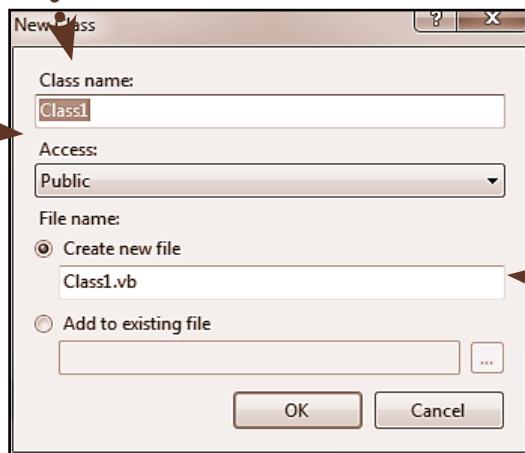


Drag a Class item onto the diagram surface from the Toolbox. Visual Studio will display the New Class dialog.

This is the name of the class, the identifier by which it will be referenced in your code. It must comply with the rules for identifiers.



Access controls whether the type can be seen outside the type that contains it. (We'll talk about access modifiers in more detail in a few pages.) Most of your classes will have **public** access, which means that they can be seen & instantiated anywhere.



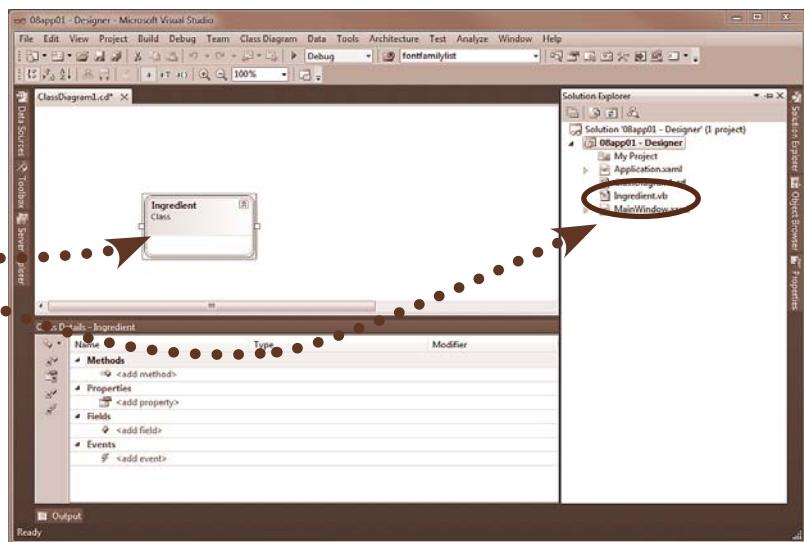
Best practice is to put all of your classes in separate code files. The file name defaults to the name of the class, but you can change it to any valid Windows file name.



Set the name of the new class to **Ingredient**, leave the remaining items on the dialog at their default values, and then click OK.

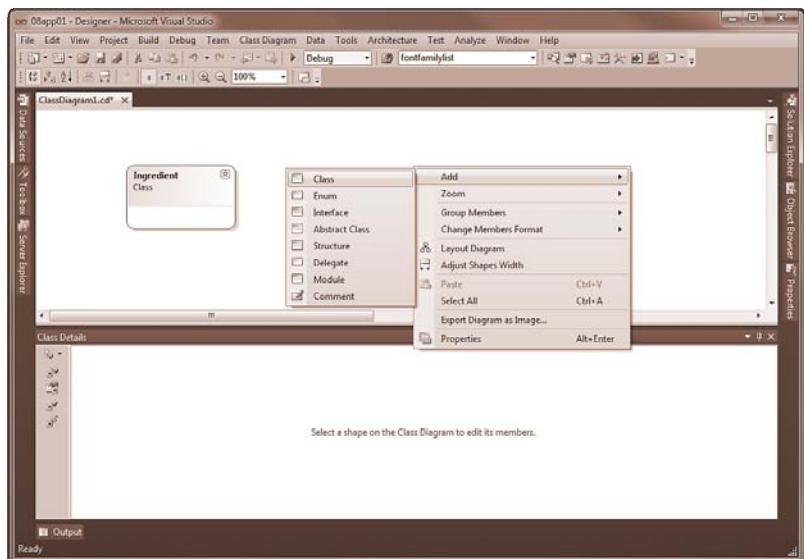
ADDING MORE CLASSES

When you click OK on the New Class dialog, Visual Studio will add a class to the diagram and a class file to the project. The class doesn't have any members yet, so there are no compartments shown.



Once you've added the first class to the diagram, you can use the context menu on the design surface (NOT the class you've added) to add new classes.

Using either
the context menu
or the Toolbox, add the
other two classes we need for
our diagram, Component, and
Recipe accepting the defaults
for everything but the Class
Name.



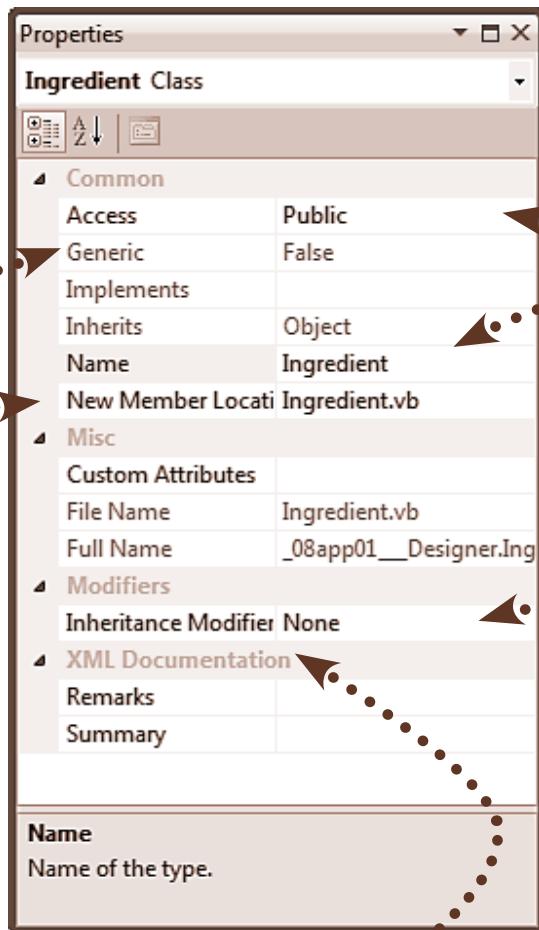
CLASS PROPERTIES

Like any Visual Studio Designer, the Class Diagram Designer Properties window shows you the properties of the selected objects. Unfortunately, in Visual Studio 2010, not all of these properties can be set in the Designer itself, but changes you make in code will be reflected in the Properties window.

Properties that are grey in the Properties window can only be changed in code. We'll talk about most of these when we look at the code for declaring a class.

New Member Location defines the file where members will be defined. It isn't retroactive—it doesn't move existing member definitions to the file you specify here—but any new definitions will be placed in the file you specify here.

The definition of a class can be spread across multiple code files. As we'll see, that's really handy when you're working with Designers, but it's not something you'll do often otherwise.



We'll talk about Access modifiers in just a minute.

You can change the name of the class here, but it won't change the name of the code file.

Inheritance Modifiers control whether and how new classes can inherit from this class. We'll discuss them in Chapter 12.

If you fill in the XML Documentation properties here, Visual Studio will add them to the class file so that they'll be available to Intellisense.

ACCESS MODIFIERS

Statements that create things, like the class statement that defines a class and its members, take an ACCESS MODIFIER that controls their VISIBILITY, that is, what other code can create instances of them or reference them, and change their values. There are five levels of accessibility, from "whoever wants to" to "nobody but me". They're shown in the table below.

TYPES DECLARED WITH THIS KEYWORD	CAN BE ACCESSED BY
Public	Any code
Protected	The type that declares them, or by types that inherit from that type
Private	The type that declares them
Friend	The assembly in which they're declared
Protected Friend	The type that declares them, types that inherit from that type, or the assembly in which they're declared

These two modifiers are only rarely used.

An assembly is, roughly, an executable (.exe) or dynamic link library (.dll) that runs in the CLR. It's actually a little more complicated than that, but you don't need to worry about those details at this point.



PUT ON YOUR THINKING HAT

In the diagram below, the `TestAccess` class itself is declared as `public`, so all of the other classes shown can reference it. But what members of `TestAccess` can each of the other classes see?



WindowClass

TestAccessAssembly

TestAccessWindowChild

TestAccessChild

MyPublic,

MyProtected

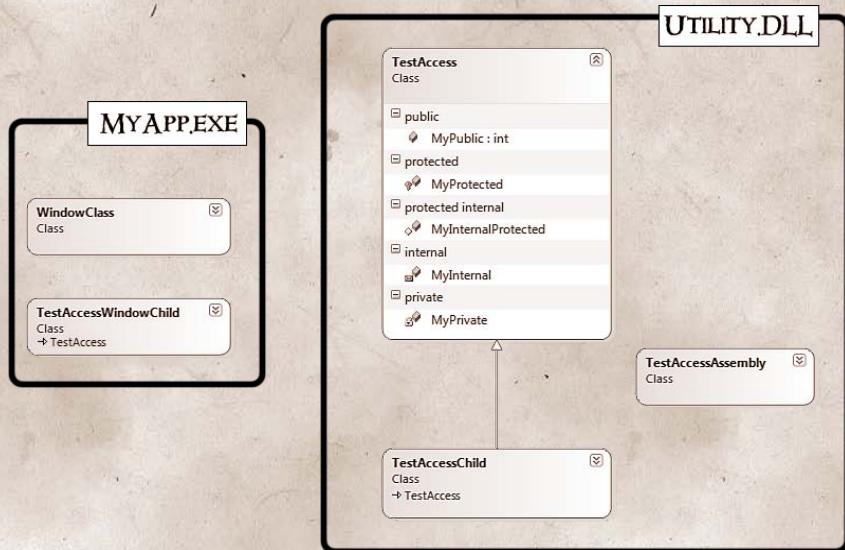
MyInternalProtected

MyInternal

MyPrivate



HOW'D YOU DO?



Protected members are visible to a type's children, in any assembly.

Everyone has access to public members.

	WindowClass	TestAccessWindowChild	TestAccessAssembly	TestAccessChild
MyPublic	✓	✓	✓	✓
MyProtected		✓		✓
MyInternalProtected		✓	✓	✓
MyInternal			✓	✓
MyPrivate				

MyPrivate can't be seen outside the TestAccess class.

Even though **TestAccessWindowChild** inherits from **TestAccess**, it can't see **MyInternal** because it's in a different assembly.

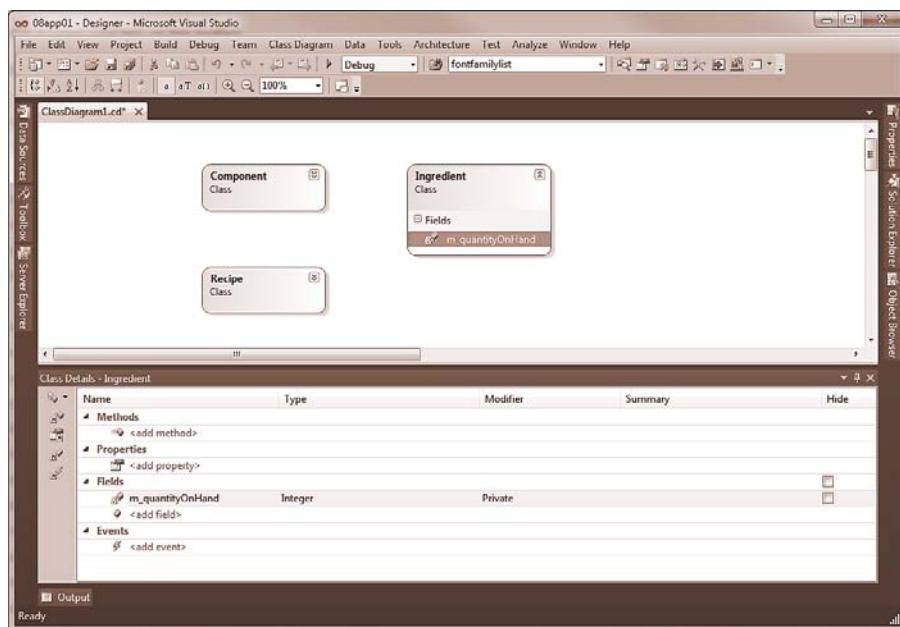
ADDING STATE MEMBERS

Now that we have our classes set up, it's time to start adding some members to them. We'll start with the **Ingredient** class. You can add class members in the Diagram pane or the Class Details pane of the Class Designer. I find it easier to use the Class Details pane, but let's try the Diagram pane so you can decide for yourself.



Right-click the **Ingredient** class in the Diagram pane and choose Add, Field. Visual Studio will add the field to both the diagram and the Class Details pane.

Change the name of the field to **m_quantityOnHand**.



You'll see in the Class Details pane that Visual Studio has set the field to its default values, an Integer (which, you'll recall, is a .NET Framework Int32) with Private accessibility. That's what we need here, so you don't need to change anything.

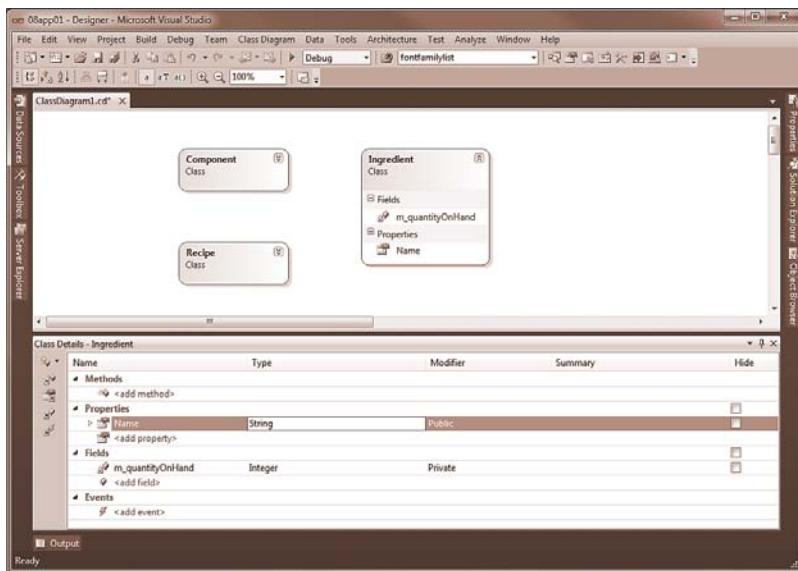
USING CLASS DETAILS

Now let's add a property to the Ingredient class using the Class Details pane.



In the Class Details pane, click on <add property> in the Properties section. Type "Name", and press the Tab key to move to the next field. By default, Visual Studio will set the property type to Integer.

Type "str" to bring up Intellisense, and set the property type to String.



A BIT OF CONTROVERSY

There's some debate over how to name private member fields that have corresponding public properties. I personally use the convention of prefixing the field with "m_" which is short for "member variable", so the backing field for a property called Name would be m_Name. Some people (including some people who write MSDN standards) recommend just the underscore: _Name. I think that naked leading underscore is too easy to overlook, but ultimately, which convention you choose is less important than choosing one and sticking to it.

ESTABLISHING INHERITANCE

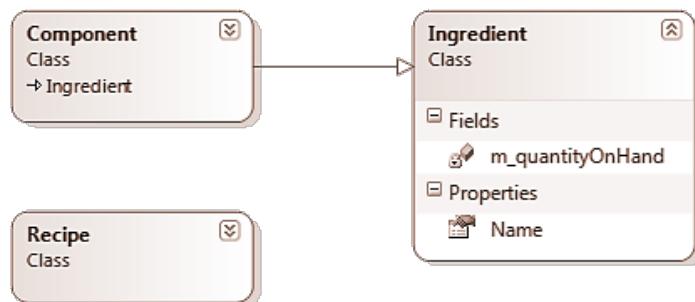
One last thing to do before we move on to the code. In our application, a Component like pastry cream is a special kind of Ingredient. A component knows and does everything an ingredient knows and does, but it has some extra functionality—it has a recipe, and it can be allocated to a dessert. In the application class hierarchy, we'll represent that kind of "IsA" relationship by having the Component class inherit from the Ingredient class. (And we'll discuss how to make those decisions starting in Chapter 12.)



In the Class Designer Toolbox, select the Inheritance item. (Don't try to drag it onto the design surface. That's what you usually do, but it won't work here. When you select Inheritance and move your cursor onto the Diagram pane, the cursor will show a little arrow with a white arrowhead to its right. That's how you know you're on the right track. If you choose Association by mistake, the arrow will be black.)

With Inheritance chosen in the Toolbox, click on Component and then click (or drag to)

Ingredient. Visual Studio will add the Inheritance arrow to the diagram and also to the Component class.



TAKE A BREAK

We're through with the Class Designer for the moment, so take a short break before you do the Review exercise and we tackle the Visual Basic version of a class definition.





REVIEW

Our model needs one more class, `Recipe`. We won't be adding any members to it, but we'll need it for a member of the `Component` class. Add it to the diagram.

The `Ingredient` class needs one more property, an Integer called `QuantityOnHand`. Add it to the class using either the Class Details pane or the Diagram pane, whichever you prefer.

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. What property was changed?



CLASS DEFINITIONS

Now that we've got our classes defined (at least roughly), let's look at the code that Visual Studio has generated for us. Open the Ingredient.vb file in the Solution Explorer. It should look like this:

```
Public Class Ingredient
    Private m_quantityOnHand As Integer

    Public Property Name As String
        Get
            End Get
        Set
            End Set
        End Property

    End Class
```

The `End Get` will be underlined. If you hover your mouse over the line, Visual Studio will tell you that "Property 'Name' doesn't return a value on all paths. A null reference exception could occur at runtime when the result is used." For now, let's throw a `NotImplementedException` just to keep our application relatively clean:

```
Get
    Throw New NotImplementedException()
End Get
```

The `Class` statement wraps the declaration.

- The field is a simple declaration statement. Notice that the Class Designer doesn't add the optional `Dim` keyword.

Here's the property, `Name`. There are two parts to the property definition: the `Get` clause contains code that will be executed when the value is retrieved, and the `Set` clause is executed when the value is changed. We'll look at how that all works in a few pages.

THE CLASS STATEMENT

The VB Class statement is used to define a .NET Framework class type. Here's the syntax:

Any of the access modifiers can be used with the `Class` statement. (Check back to page 232 if you've forgotten them.) If you don't specify the modifier, the class will be `Friend`, which is hardly ever what you want.

You really should be explicit about the access modifiers even if you do want the default, because not everything has the same defaults, and it's easy to muddle them up.

If you want to split the class definition into multiple files, at least one of them must use the `Partial` keyword.

You can name your class anything you like as long as it complies with the rules for identifiers. By convention, classes use PascalCase.

```
[modifier] [Partial] Class <name>
[Inherits <classname>]
[Implements <interface>[, ...]]
```

A class can inherit from one (and only one) base class. If you don't specify a base class here, the class will inherit from the .NET Framework `System.Object` class.

We'll talk about `interfaces` in the next chapter. A class can only inherit from one class, but it can implement multiple interfaces. They're separated by commas.

Notice that `Inherits` and `Implements` are separate statements on separate lines.



PUT ON YOUR THINKING HAT

Write the statement that defines a class named `MyClass` that inherits from `MyBase`.

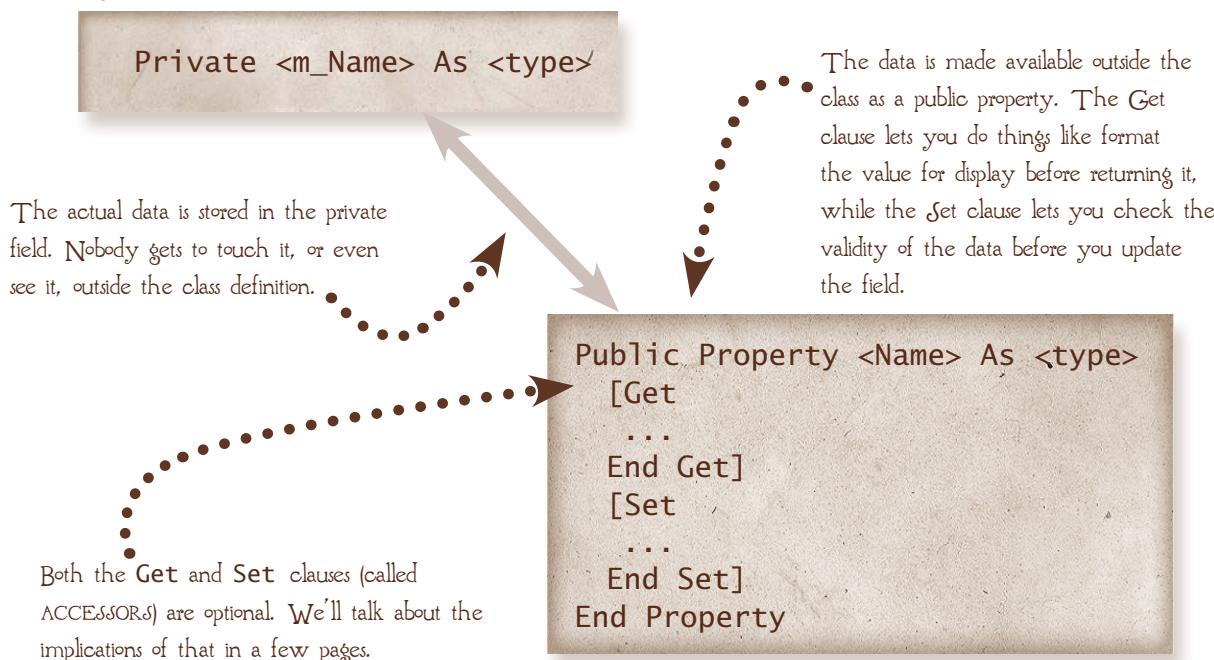
Write the statement that defines a class named `NewClass` that inherits from `Object`.



FIELDS AND PROPERTIES

I bet you'd tell just about anybody the name of your bank, but even your close friends probably don't know exactly how much you have in your checking account, and if you let more than one or two people have access to the account itself, you're far more trusting than I am. (Or you live in a commune.)

When you transfer this principle to programming, it's called **Information Hiding**, and we'll look at it again in Chapter 12. Access modifiers are a big part of information hiding—best practice dictates that everything is on a need-to-know basis—but in the context of a class definition, it's also implemented by the combination of a private field, called a **backing field**, and a public property that allows you to control how information is seen or changed. The basic pattern looks like this:



MAKE A NOTE

Properties don't have to be public and backing fields don't have to be private, but that's the usual pattern.

IMPLEMENTING A PROPERTY

In the classic property pattern, the `Get` and `Set` clauses of the property statement, called `get` and `set` accessors respectively, transfer data back and forth between the backing field and the rest of the world. The property code pattern simply lets you wrap some code around the data to make sure nobody does anything to it that they shouldn't.

When it's compiled, the property statement is actually translated into two distinct methods: `get_<PropertyName>` and `set_<PropertyName>(<type> value)`.



But all of that is handled by the compiler, so you don't need to worry too much about it, except that it means that within the set accessor, you can access the new data using the parameter name `value`:

`m_PropertyOne = value`

HOW'D YOU DO?

Write the statement that defines a public class named `MyClass` that inherits from `MyBase`.

```
Public Class MyClass  
Inherits MyBase  
End Class
```

Write the statement that defines a private class named `NEWCLASS` that inherits from `Object`.

```
Private Class NewClass  
End Class
```

Classes inherit from `Object` by default, so you don't need to specify it. (But it wouldn't be wrong if you did.)



PUT ON YOUR THINKING HAT

Try writing the property procedure for the `QuantityOnHand` property. In the get accessor you simply want to use the `Return` keyword to return the value of the backing field, and in the set accessor, you want to make sure that the value is greater than 0 before you set it. (For now, you can ignore bad values, but for extra credit, try throwing an exception.)



HOW'D YOU DO?

This was a tough one, so don't worry if you struggled with it a little, and didn't get it quite right. After all, you've never seen this before.

Public Property QuantityOnHand As Integer

Get

 Return m_quantityOnHand;

End Get

Set

 If value >= 0

 m_quantityOnHand = value

 Else

 Throw New Exception("Value must be greater
than 0")

 End Set

End Property

The get accessor uses
Return to pass the
value of the backing
field back to the
calling program.

The set accessor checks that value is
greater than zero, and then uses a simple
assignment statement to update the value of
the backing field.

AUTO-IMPLEMENTATION

Let's face it; it's all very well to know, intellectually, that wrapping a state value in a property is safer, more flexible, and generally deemed to be best practice. But even when the Class Designer builds the basic code structure, it's still a nuisance to create the backing field and write simple code around it. If you're rigorous, you'll find yourself writing the same basic structure over and over again.

Well, you don't have to, not since Visual Basic introduced some syntactic sugar for just this situation.

Public Property <name> As <type> [= value]

You can even give the property a default value using this optional syntax.



When you use this syntax, the compiler will create an anonymous backing field (which means you can't see it) and build the get and set accessors for you. If at some point in time you need to add some code to either accessor, you can simply add an explicit backing field and replace the auto-implemented syntax with normal syntax. And the great news is that the code that references the property doesn't have to change at all.

Unfortunately, when you define a property in the Class Designer, Visual Studio will always use full property syntax for the properties you create there. But you can change the code file to use auto-implementation, and the Class Designer will be able to manipulate them.



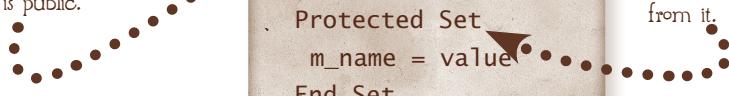
PUT ON YOUR THINKING HAT

The `Name` property of the `Ingredient` class doesn't have any restrictions that we know of. Replace the standard property syntax created by the Class Designer with an auto-implemented property.

ACCESSOR ACCESS

In the last section we looked at how modifiers can be used to control access to a type's members. By default, the get and set accessors of a property have the same access as the property itself. That is, if a property is public, any other code can both read and set the value of the property. But what if you want any code to be able to read the value (public accessibility), but only the type and its children to be able to set it (protected accessibility)? You can do that by specifying an access modifier for one or both of the accessors.

Because the get accessor doesn't specify accessibility, it inherits the accessibility level of the property, which in this case is public.

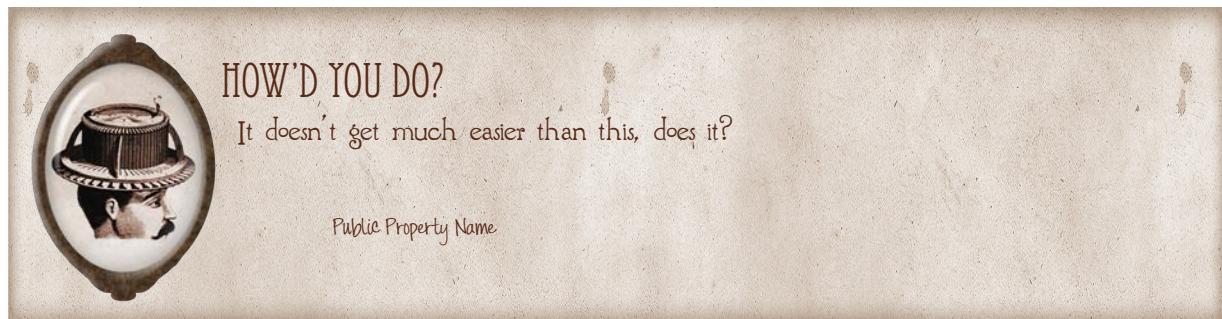


```
Public Name As String  
Get  
    Return m_name  
End Get  
Protected Set  
    m_name = value  
End Set  
End Property
```

The set accessor, by specifying protected accessibility, restricts changing the value to the type itself and the classes that inherit from it.

The modifiers to the get and set accessors can be more restrictive than the property itself, but not less restrictive. You couldn't, for example, declare a private property with a public get accessor.

You can't use access modifiers on auto-implemented properties; specifying different accessibility for the get and set accessors is strictly DIY.



READ OR WRITE

Did you notice that the syntax diagram on page 241 shows both the get and set accessors as optional? You must have at least one accessor, but you don't have to have them both. A property that has only a get accessor is called **READ-ONLY**, while a property that has only a set accessor is a **WRITE-ONLY** property.

Read-only properties are quite common. For the sake of efficiency, some properties of a type may be **IMMUTABLE**; they cannot be changed once the object is created because doing so would be slower than creating a new object. Other properties are the result of a calculation. The area of a rectangle, for example, is the product of its length and width. It makes sense to expose `Rectangle.Area`, but if you allowed users to change the value, how would the system calculate the `Length` and `Width` properties? In situations like that, it makes sense for `Area` to be read-only.

Write-only properties are fairly rare. They're sometimes used when doing low-level hardware manipulation, or they can be used in inter-process communication where you might, for example, set a property that tells a separate process, "I'm finished; go do what you have to do."

In either case, the syntax is simple. You simply omit the get or set accessor in a full property declaration. (You can't omit an accessor with auto-implemented properties. They always have both.)

PUT ON YOUR THINKING HAT

We didn't add any members to the `Component` class when we were working with the Class Designer. Open up the `Component.cs` file and add a read-only public property called `QuantityAllocated` with a backing field called `m_quantityAllocated`. Both should be declared as `Integer`.

Why do you think the `QuantityAllocated` should be read-only? (Hint: Look at the diagrams on page 225. What does the `Allocate()` method need to do?)



HOW'D YOU DO?

Here's what your class definition should look like:

Public Class Component
Inherits Ingredient

Private m_quantityAllocated As
Integer

Public QuantityAllocated As Integer

Get

 Return m_quantityAllocated

End Get

End Property

End Class

The Allocate() method of this class will need to change quantityAllocated based on the recipe chosen. It may also need to interact with the Production Schedule application. The property accessors should only manipulate the property itself, so this functionality is best left to a method.



TAKE A BREAK

We've finished our discussion of properties for the time being (we'll look at them again when we discuss principles in Chapter 11), so take a break before you do the Review and we move on to methods.



REVIEW

When can you use an auto-implemented property?

What is the private field that stores the data for a public property usually called?

Can you think of an example of a property that should be read-only? Describe it.

Write the code that declares an `Integer` property called `Counting` that can only be set to a number greater than its current value.



METHODS

Let's start our discussion of methods by going over some of the things you've already learned about them. In the sentences below, circle the correct option from the choices in parentheses. (It's okay if you've forgotten some of this. We'll be looking at this in more detail now, and that should help it stick.)

A method that returns a value is a (Function/Module).

Method parameters are enclosed in (parentheses/curly braces).

A class can declare (one/more than one) method with the same name.

The keyword for a method that doesn't return a value is (Nothing/Sub).

The return type, name and parameters of a method define its (membership/signature).

The statements to be executed by a method are enclosed in (a statement block/parentheses).

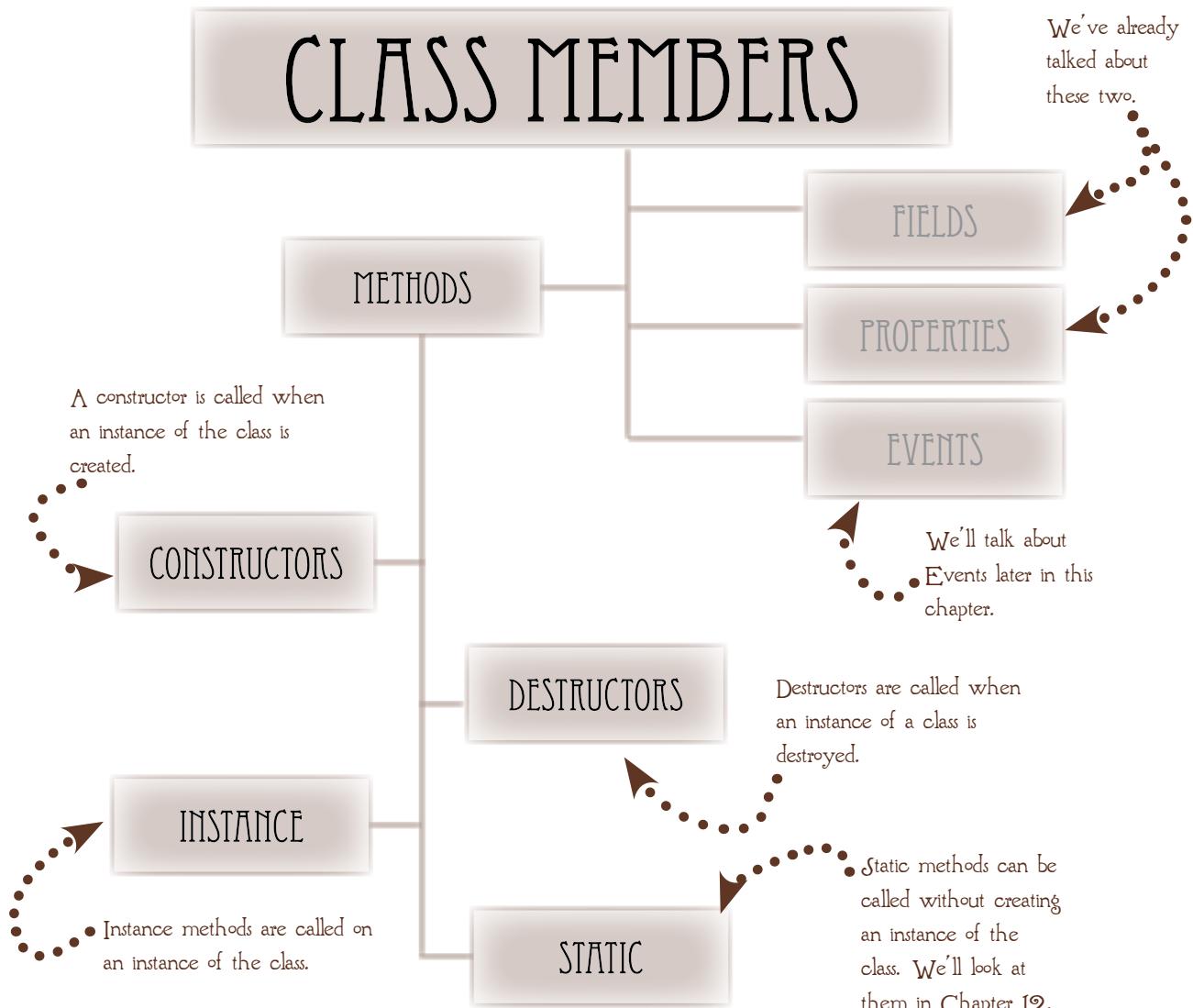
The keyword to indicate the value a method should return to its caller is (Return/Exit).

By convention, method names are given (camelCase/PascalCase).

Inside the method, parameters are referred to by (their parameter name/the keyword value).

METHOD CATEGORIES

To the Visual Studio Class Designer (and, as we'll see, the .NET CLR), a method is a method. But from a programmer's point of view, methods fall into four distinct categories. Almost all the methods you write will be instance methods, but it's important to know how to create the others if you need them.





CREATING METHODS

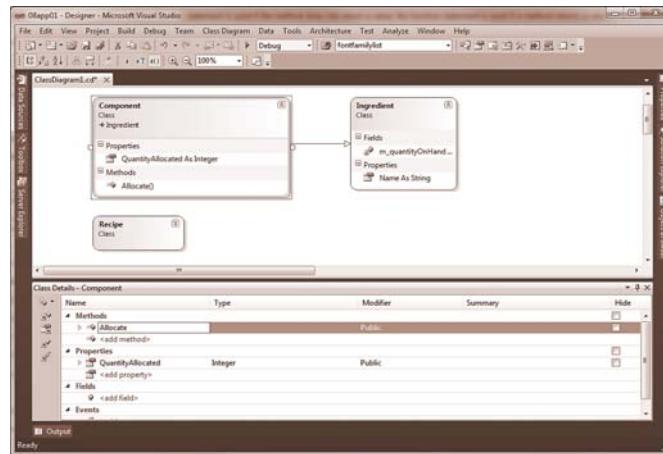
Like any class or member, you can create instance methods either directly in code or by using the Class Designer. (Of course, you'll always have to implement the behavior of the method in code, because the Class Designer doesn't know how to do that.) Let's start by creating an instance method in the Designer, and then we'll look at the implementation in code.



If it isn't already open, load the chapter solution into Visual Studio and open the class diagram we've been working with.



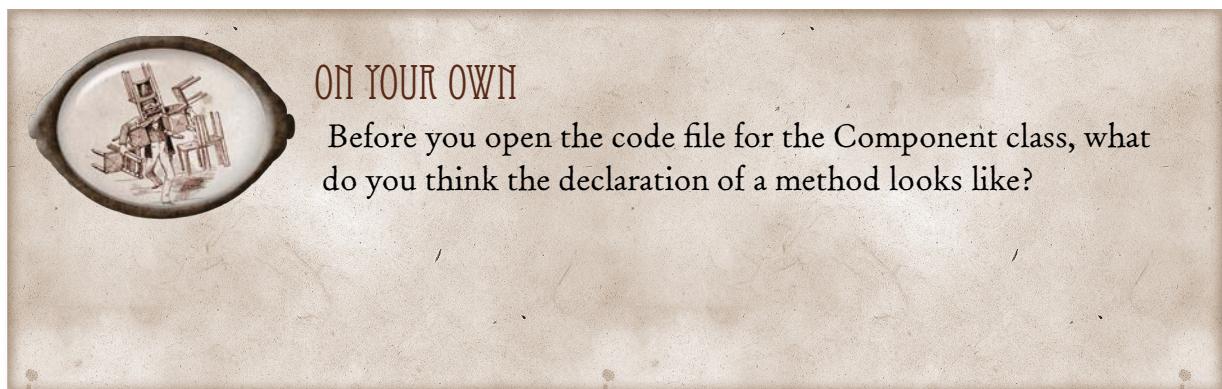
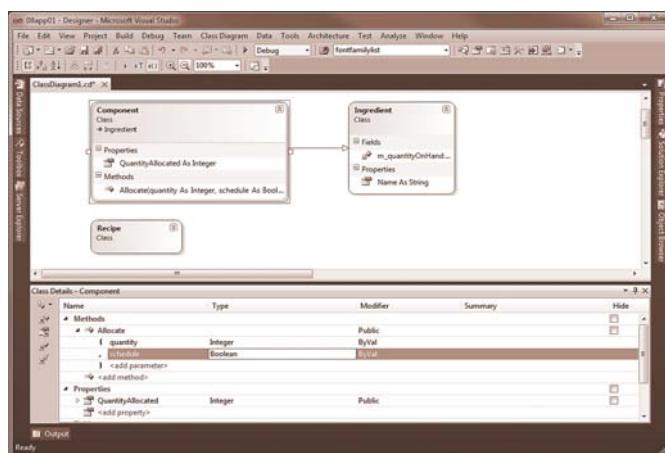
Select the Component class in the Diagram pane and add a method named `Allocate` by clicking on <add method> in the Class Details pane. You'll also need to add an Integer `QuantityAllocated` property.



ADDING PARAMETERS



The `Allocate()` method is `public` and doesn't return a value, so the defaults for type and modifier are correct. But it does have two parameters: an `Integer` called `quantity` and a `Boolean` called `schedule`. Click on the triangle just to the left of the method name in the Class Details pane to open the parameters list for the method, and then add the parameters:





ANATOMY OF A METHOD

The syntax for a method declaration is almost identical to the syntax for declaring a variable, class or property:

If you don't specify an access modifier, the default is private, but you should always specify the modifier (if only so you won't have to remember that).

The name of the method must comply with the rules for identifiers.

You can declare as many parameters as you need or none, if you don't need any. Each one takes the format <name> As Type, and it should be separated by commas.

```
[modifier] <Sub|Function> <Name> ([param As type,  
...])  
// implementation  
End <Sub|Function>
```

```
Public Sub MyMethod()  
...  
End Sub
```

Use Function if the method returns a value, Sub if it doesn't.
(But you knew that, right?)

You can use almost all the statement types you've learned (and some others you haven't seen yet) inside the method. You can't create types or members within a method, but you can instantiate types using a variable declaration.



MAKE A NOTE

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, which makes sense...how would you call `MyClass.X()` if you can't see `MyClass`?

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

BUILDING ALLOCATE()

We're not doing a full implementation of the Scheduling application, but let's rough out the functionality. The `Component.Allocate(quantity, schedule)` method needs to update the values of `QuantityAllocated` and `QuantityOnHand`. (`Component` inherits both of these properties from its base class, `Ingredient`.) If the Boolean `Schedule` parameter is `true`, the method should make a call to the scheduling component. We don't have a scheduling component, so we'll throw a `NotImplementedException`. Let's take a look at one way to implement it:

You should always check that the arguments you received are valid before you do anything else.

We haven't defined a backing field for `QuantityAllocated` (presumably we'll use an auto-implemented property), but even if we had, it's best practice to go through the property accessors so that we catch any code they implement.

```
Public Sub Allocate(quantity As Integer, schedule As Boolean)
    If (quantity < 0)
        Throw New ArgumentException("quantity must be greater than 0")
        Return
    End If
    Me.m_quantityOnHand -= quantity
    Me.QuantityAllocated += quantity
    If schedule = True
        Throw New NotImplementedException("Not Yet Built");
    End If
End Sub
```

The keyword `Me` refers to the current instance of the object. It's not strictly necessary here, because VB can find the instance variable `m_quantityOnHand`, but it's a good idea to always use it, if for no other reason than it means you don't have to think about whether it's actually required.

We're not returning a value from this method, so an explicit `Return` statement isn't necessary. The method will return to the caller when it runs out of statements to execute.

ME WHO?

In the example on the previous page, I used the `Me` keyword to avoid a programmer error (as opposed to a programming error). But `Me` isn't always optional; you'll need to use it whenever you refer to the instance of the object from within the method. Say, for example, that once the scheduling component of our dessert application is implemented, it exposes a method called `Schedule` that takes as an argument the component to be scheduled. The method signature would look like this:

Public Function Schedule(component As Component)

Inside the `Allocate()` method, you could call the `Schedule()` method and pass the current `Component` using the `Me` keyword:

Schedule(Me)



BEST PRACTICES — BUILDING METHODS

You should try to keep the method body to less than 50 statements. More than that and it becomes too difficult to read.

You should try to keep the number of parameters to less than 6. After 3 or 4, the method becomes tedious to call. After 6 or 7, it's almost impossible to keep everything straight, even with the help of Intellisense. (In the next chapter we'll look at `Structures`, which are a good way to avoid the endless parameter list.)

As a general rule, every method should have a single exit point. Some people will argue that it's an inviolable rule, but you've already seen me break it by adding a `Return` statement for invalid arguments. My rule is "aim for one return, unless adding more makes the code easier to understand". In my world, "easier to understand" is always a higher priority than following conventional heuristics. To get a sense of my reasoning here, why don't you try re-writing the `Allocate` method with a single exit point at the end of the sub.



PUT ON YOUR THINKING HAT

You know that a class can expose more than one method with the same name. Called OVERLOADS, these methods must have different signatures which, for the purpose of overloading, means that they have different parameter lists. Write an overload of the `Allocate()` method that doesn't take a `schedule` parameter.

You can use the Class Designer to "stub out" the method, or write it from scratch, whichever seems easier to you, but try to write it without looking back at my sample.



HOW'D YOU DO?

Did you decide to use the Class Designer or did you write everything from scratch? Either way, your code should look something like this:

```
Public Sub Allocate(quantity As Integer)
    If quantity < 0
        Throw New ArgumentException("quantity must be greater than 0")
    Return
End If
Me.Quantity -= quantity
m_quantityAllocated += quantity
End Sub
```



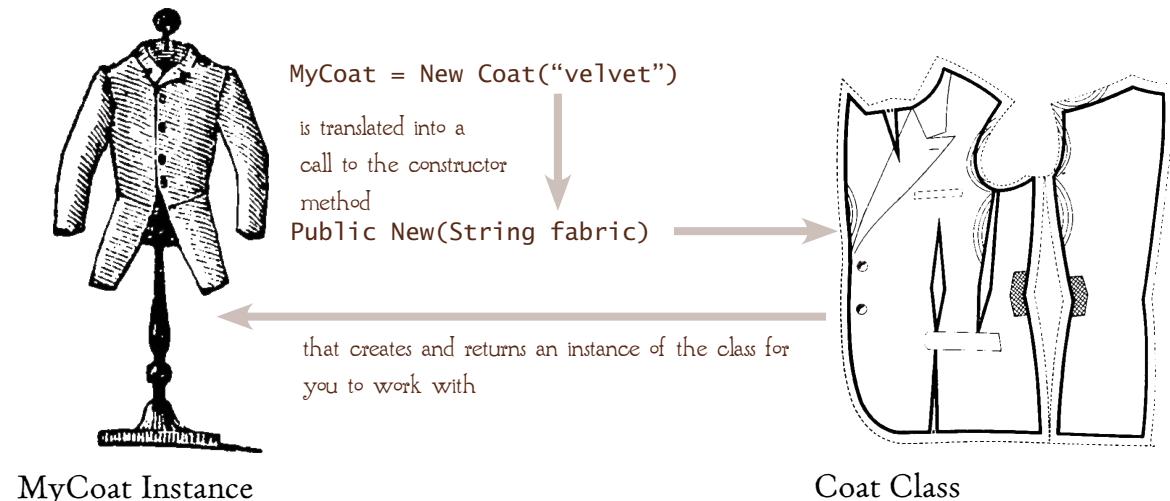
DEFAULT CONSTRUCTORS

Like any method, constructors can be overloaded. A constructor that takes no parameters is called the DEFAULT CONSTRUCTOR. If (and only if) you don't define any constructors in your class definition, the compiler will create a default constructor for you. Compiler-generated default constructors are convenient, but they can come back to bite you.

Say, for example, that you release a class called `Recipe` that doesn't define any constructors, and people use it with the compiler-generated default constructor: `New Recipe()`. Later on, you decide it would be nice to add a constructor that takes the `Name` property as a parameter, so you revise the class and add the overload. All the code that uses the `Recipe()` syntax is now broken because the compiler no longer generates the default for you. The solution? Always define the default constructor yourself, even if it doesn't do anything.

CLASSES, INSTANCES & CONSTRUCTORS

A class definition is like a sewing pattern that provides the instructions for constructing a garment. When you use a class, you **INSTANTIATE** it (create an instance) by using the **New** keyword with something that looks like a method call. It looks like a method call because it *is* a method call, a call to a special kind of method called a **CONSTRUCTOR**.



MAKE A NOTE

Constructors are just a kind of method and are created like them. But there are a few additional rules:

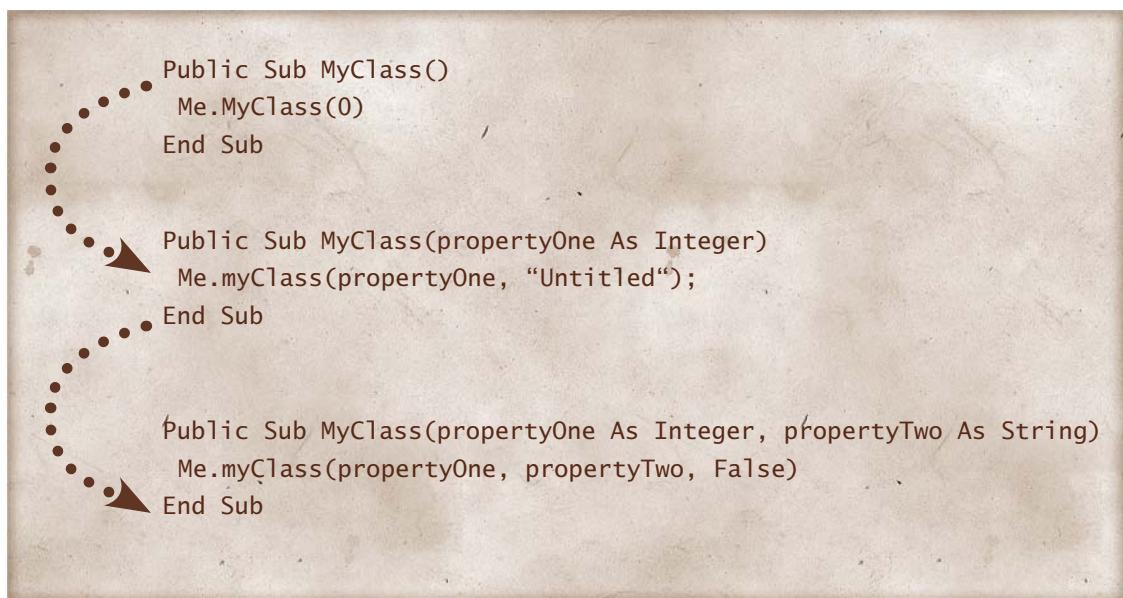
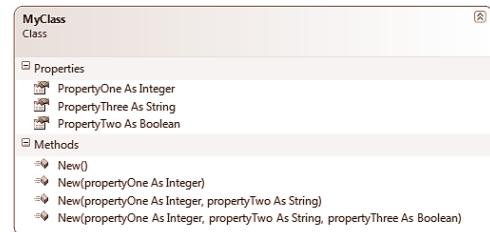
- Constructors must always be named **New()**.
- Although they return an instance of the class, a constructor is always declared as a **Sub**.

CONSTRUCTING CONSTRUCTORS

Best practice dictates that constructors should do very little. In fact, most well-designed constructors only set the class properties to their initial values using the property accessors to ensure that any necessary validation takes place.

The reason you should use the property accessors is so that you don't duplicate code. Duplicating code is always bad. (Or at least, in all my years of programming I've never seen a good reason to do it. If you think of one, email me.) And it applies to constructors just as much as it applies anywhere else in an application.

There's a conflict here: On the one hand, it's a kindness to the users of your class to be able to set only the basic properties of the object when it's instantiated, and that usually means several overloads of the constructor. On the other hand, you don't want to duplicate the code that sets those properties within the various method calls. The solution is to use a design pattern called CONSTRUCTOR CHAINING: the versions of the constructor with fewer parameters call the versions with more, providing reasonable defaults for the missing arguments:





PUT ON YOUR THINKING HAT

The `Ingredient` class in our sample application is a good place to try out your constructor-writing skills. Add two constructors, one that lets the user set the `Name` property, and one that takes no parameters at all (in other words, a default constructor).

(Answer on page 265.)

SCOPE

When you create a variable with a declaration statement, .NET allocates enough memory to store the specified type's properties and makes the identifier you've declared available for use. But declared elements don't last forever, and you can't access them from anywhere. The rule is simple: A variable is only visible within the context in which it's defined. It's a simple rule, so let's look at a simple example:

The Public

keyword can only be applied to types and members, not variables.

 Even though they're declared in the same class definition, `MySentence` can't see the variable `Sentence` because it's declared in a different scope.

```
Public Class MyClass  
    Public Sub MyMethod()  
        Dim Sentence As String = "My name is "  
        End Sub  
  
    Public Function MySentence(string Name)  
        Return Sentence += Name  
    End Function  
End Class
```

The variable `Sentence` is defined within `MyMethod`. That is its SCOPE. It cannot be declared `Public`.

 The calling code can't see the variable `Sentence` declared inside `MyMethod`.

```
MyClass MakeAName = New MyClass(),  
MessageBox.Show(MyClass.Sentence)
```



HOW'D YOU DO?

The default constructor just passes a default value to the overload.

```
Public Sub New()
    Me.Ingredient("No Name")
End Sub
```

The overload uses the set assessor of the public property to set the Name.

```
Public Sub New(String name)
    Me.Name = name
End Sub
```



PUT ON YOUR THINKING HAT

Scope really is a pretty simple principle, but there are a couple of places you can manage to confuse yourself. Let's take a look at the two most common:

```
Public Class MyClass
    Private x As Integer = 5
```

```
Public Sub DoSomething()
    Dim x As Integer = 0
    MessageBox.Show(x.ToString())
End Sub
```

What gets displayed in the MessageBox?

```
Public Sub DoSomethingElse()
    For y As Integer = 0 to 5
        x +=, 1
    Next y
    MessageBox.Show(y.ToString())
End Sub
End Class
```

Why does this line fail?





HOW'D YOU DO?

```
Public Class MyClass  
    Private x As Integer = 5  
  
    Public Sub DoSomething()  
        Dim x As Integer = 0  
        MessageBox.Show(x.ToString())  
    End Sub  
  
    Public Sub DoSomethingElse()  
        For y As Integer = 0 to 5  
            x += 1  
        Next y  
        MessageBox.Show(y.ToString())  
    End Sub  
End Class
```



The `MessageBox` will display 0 because the variable declared inside the method hides the variable declared outside of it. If you wanted to display 5, you would need to use the `this` keyword, but I'm sure it will come as no surprise that it's really best practice to avoid using duplicate variable names if there's any chance of confusion.

This line will fail because `y` is declared in the context of the `for` statement, not the method itself. When the `for` statement exists, `y` goes out of scope.



BEST PRACTICE — VARIABLE NAMES

Beware! The variable names `a`, `b`, `c` and `x`, `y`, `z` are dangerous. It's customary to use them as the control variable of a loop, as I've done here, but they don't tell you anything but that they're a control variable. If you're going to use them for any other purpose, you should choose a more descriptive name (even if it's only `loopCounter`).



UNDER THE MICROSCOPE—GARBAGE COLLECTION

You know that when you declare a variable, .NET will allocate the memory for it. When the variable goes out of scope, that memory is no longer needed and is available for reuse, a process called garbage collection. Notice that I said “available” for reuse, not “is reused”.

In older programming languages, the memory would be reclaimed immediately, but the .NET Framework uses a more sophisticated process: Non-Deterministic Garbage Collection. Instead of stopping everything while the system reclaims memory it may never need (by default, a .NET Framework application has 2 gigabytes of memory available for variables; that’s a lot of variables), the garbage collector keeps track of what can be reclaimed when needed, and only reorganizes things when it has to. It’s also pretty smart about doing its housekeeping when the application isn’t doing anything else (like when it’s waiting for user input).

In the vast majority of cases, garbage collection works just fine, and you don’t have to think about it, but the .NET Framework does give you some control over the process when and if you should ever need it. Check MSDN for “garbage collection” for details. (When and if you should ever need them.)



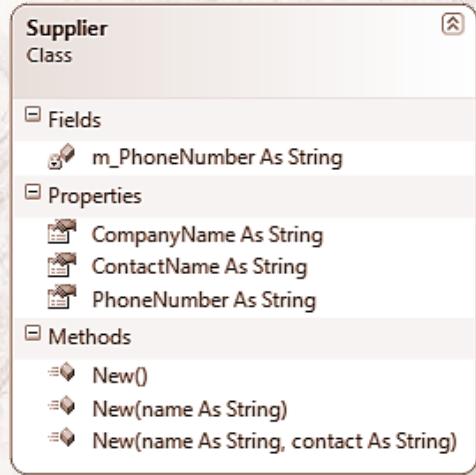
TAKE A BREAK

We’ll be looking at other aspects of methods and other categories of methods as we go along, but that’s it for now. Why don’t you take a short break before you do the final Review exercises?



REVIEW

Here's a class diagram for a simple class that represents suppliers of ingredients for Neil & Gordon's cafe. Write the class code.



Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

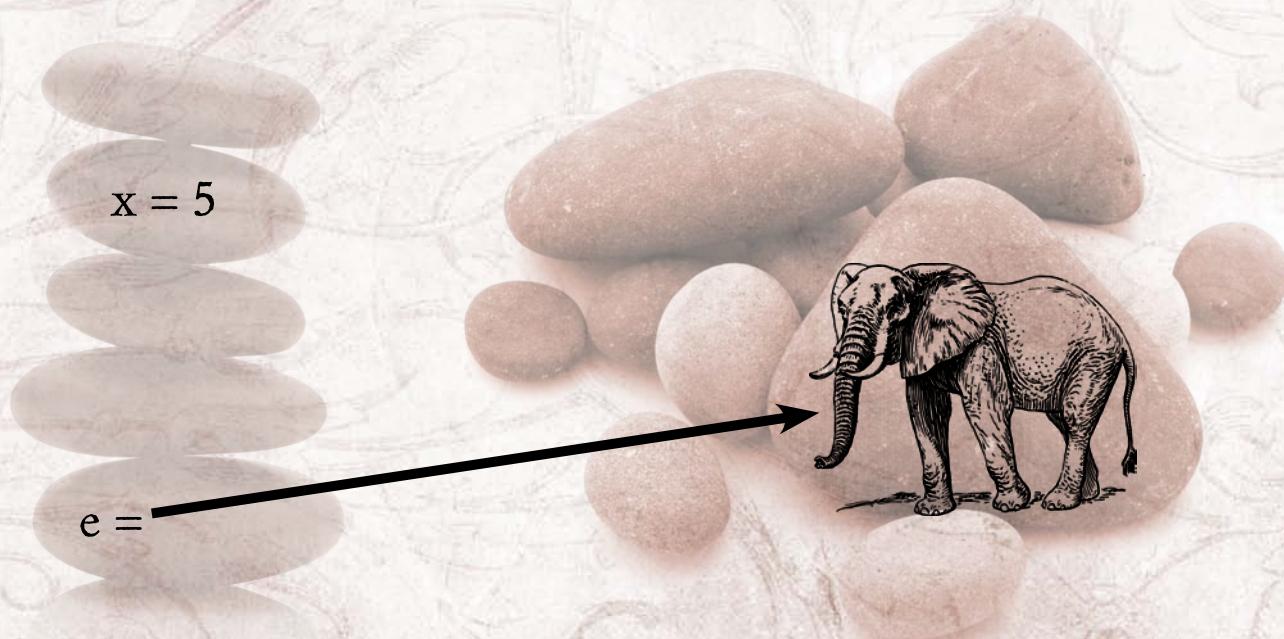
Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



OTHER FRAMEWORK TYPES

9

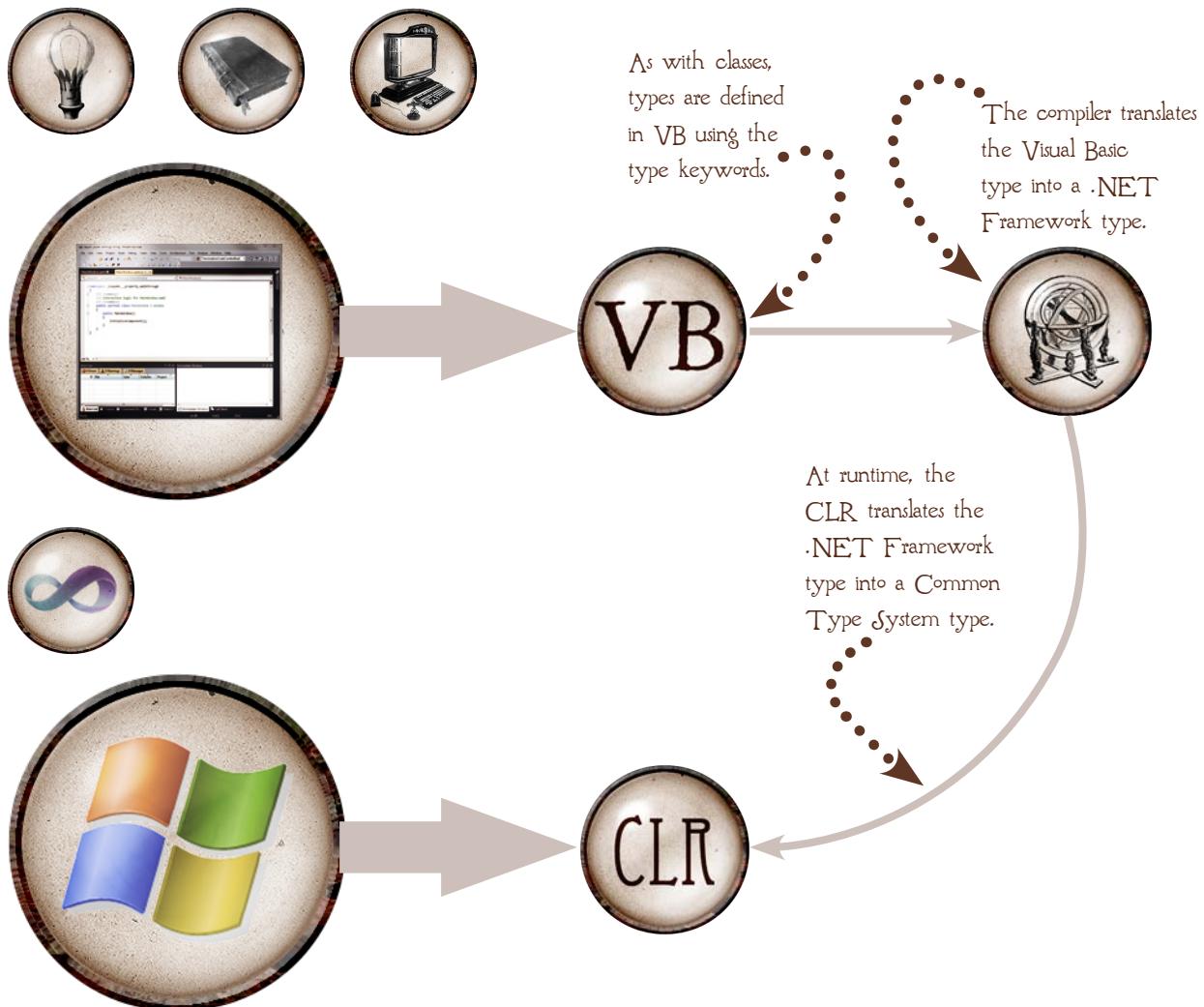
In the last chapter we looked at how you define .NET Framework classes. Classes are the most common of Framework types, and in many ways the most complex, but they're not the only ones. In this chapter, we'll look at the remaining types: Structures, which are like baby classes; Enumerations, which are simple lists of named values; and Interfaces, which are like the definition of a class without any implementation.



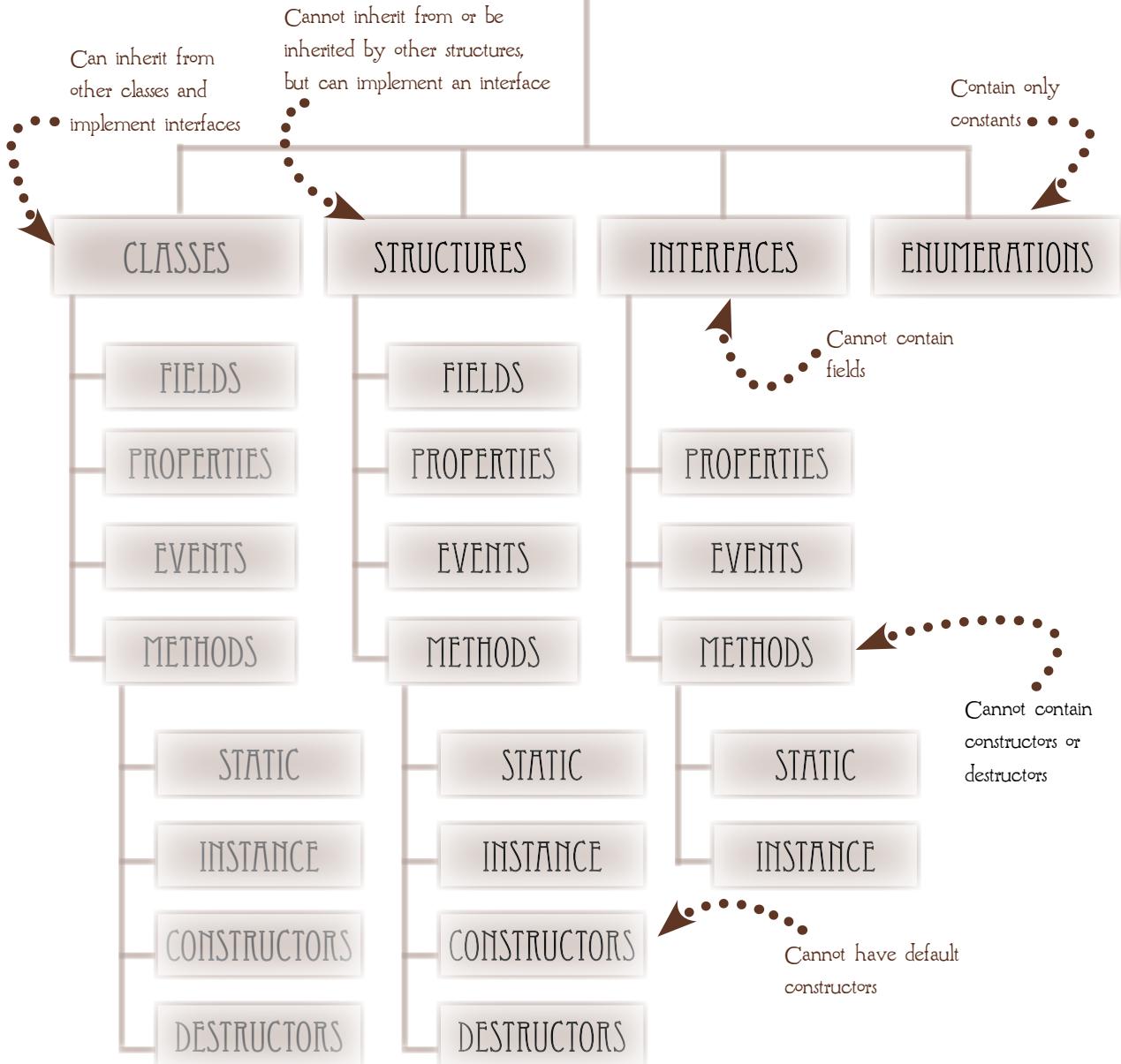


FITTING IT IN

Like classes, structures, enumerations and interfaces are implemented using Visual Basic type and member keywords (many of which you already know) and are eventually translated into .NET Framework CTS (Common Type System) types. In this chapter we'll look at these remaining types and how they fit into the CTS.

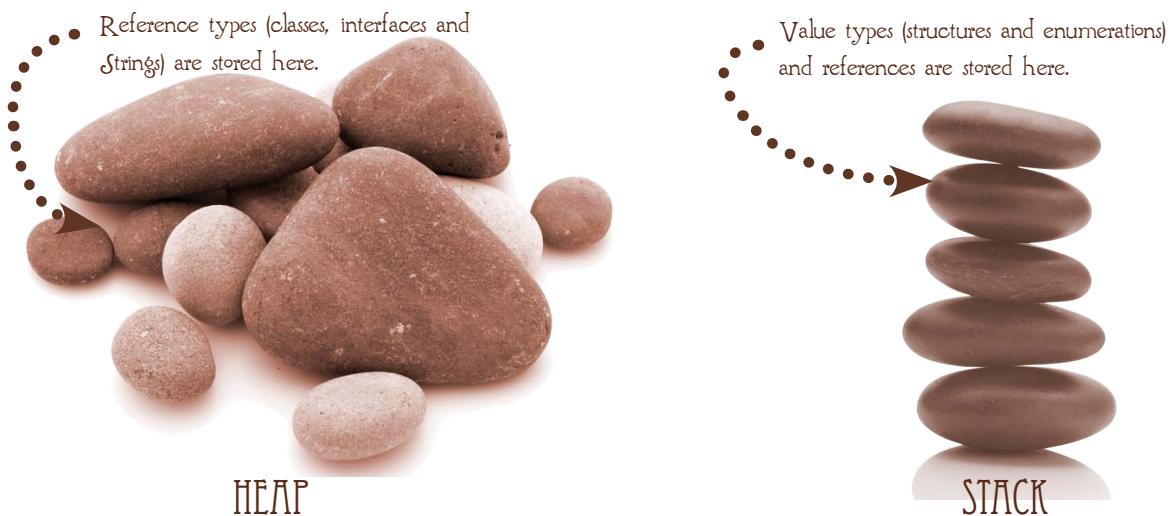


.NET TYPES



REFERENCE & VALUE TYPES

We've seen that when you instantiate an object, the .NET Framework will allocate memory to store it, and I've talked about how the garbage collection process collects the memory used by your classes. But we've been a bit vague about the whole thing, which is okay when you're only working with classes, but things get just a little more complicated when you're working with other types. Not much, but a little.



It all works fine and you don't have to think about it very much with three exceptions:

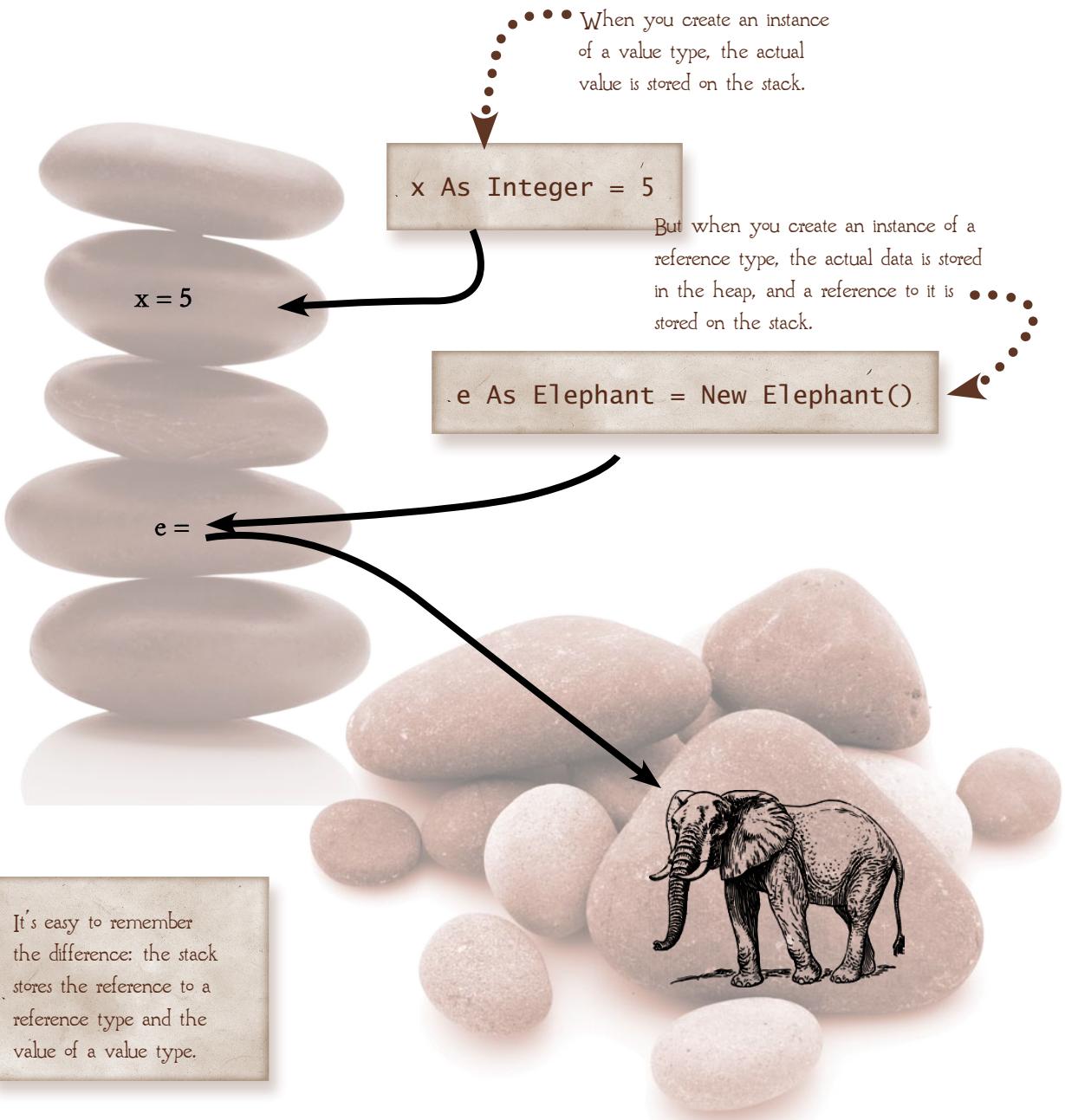
- When you treat a value type as a reference type
- When you pass an argument to a method
- When you compare two variables for equality

We'll look at these situations in detail in this chapter.



UNDER THE MICROSCOPE

Neither of the two .NET Framework value types—structures and enumerations—are allowed to specify a base class from which they inherit. There's actually a little magic going on under the hood: The .NET Framework type System. `ValueType`, itself a reference type, contains the magic that causes its children to be stored directly on the stack. Don't worry about it—it just works.





TASK LIST

In this chapter we'll finish up our initial examination of .NET Framework types by looking at the remaining types, and, once that's done, we'll take a closer look at the implications of value and reference types.



STRUCTURES

In .NET, a Structure is nothing more than a lightweight class that's stored as a value type. Because they're so much like classes, we'll start the chapter by looking at how to create them in the Class Designer and in code.



ENUMERATIONS

Enumerations are the second value type in the .NET Framework. Enumerations are simply lists of constants that make your system easier to use or understand. (After all, would you rather remember "Red" or "#FF0000"?)



INTERFACES

Interfaces represent things that multiple classes (that aren't necessarily related by inheritance) can do. We won't be looking at the role that interfaces play in object design until Chapter 12 when we examine the principles of Object-Oriented Design, but in this chapter you'll get a taste of what they can do when we look at how they're implemented.



WORKING WITH TYPES

Finally, once we've examined all the basic types that the .NET Framework supports, we'll look at how you can treat one type as another (CASTING), what happens when you treat a value type as a reference type (BOXING), & what happens when you pass value and reference types as arguments,



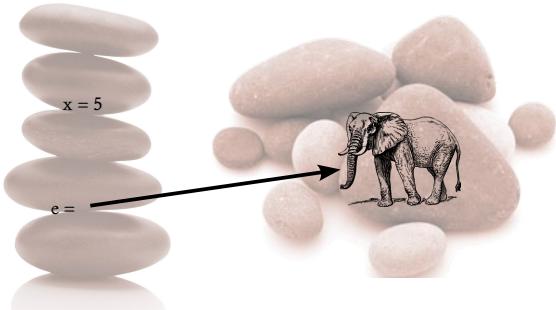
STRUCTURES

There are other considerations, as we'll see, but it isn't wrong to think of structures as the "value-type version of a class". So why do you care, and how do you decide whether to use a structure or a class?

Let's look at that stack and heap diagram again. You can see that when a value type is created, the runtime only needs to allocate memory once, from the stack, but when a reference type is created, it must be allocated from both the stack and the heap. It stands to reason, I think, that two memory allocations are going to take a bit longer than one. (Microsoft's own estimates

are that creating a reference type can take as much as twenty times longer.) If you're dealing with an object or two or ten, you'll never see it. But what if you're dealing with thousands or tens of thousands of objects? Those nanoseconds start to add up.

On the other hand, objects tend to get passed around—in to method calls as arguments and out of them as return values, as part of structured exception handling, and as events. Reference types only pass the reference while value types pass the actual values. Passing a lot of data around can be time-consuming, too, so value types are usually restricted to fairly small objects. In fact, best practice dictates that they be less than 16 bytes.



PUT ON YOUR THINKING HAT

Based on what you've learned so far about structures and classes, can you choose which type each of the following statements applies to?

CLASS

Stored in the heap

Faster to allocate

Can be of any size

Can be faster to pass to a method

Stored on the stack

STRUCTURE



CREATING STRUCTURES

Time to get our hands dirty and create a structure. We'll stub it out in the Class Designer and then complete the code in the Editor.



Create a new Class Library project. You can close the `Class1.VB` file that Visual Studio creates by default; we won't be using it.



Add a new class diagram to the project. From the Toolbox, drag a Struct onto the Diagram pane. The Designer will display the New Struct dialog. It's almost identical to the New Class dialog we worked with in the last chapter. Change the name to `FirstStructure` and click the OK button.



HOW'D YOU DO?

- Stored in the heap
- Faster to allocate
- Can be of any size
- Can be faster to pass to a method
- Stored on the stack

CLASS

- ✓
- ✓
- ✓
- ✓

STRUCTURE

- ✓
- ✓
- ✓



The Class Details pane for a structure looks just as it did when you added a class, too. (Visual Studio doesn't even change the name!)

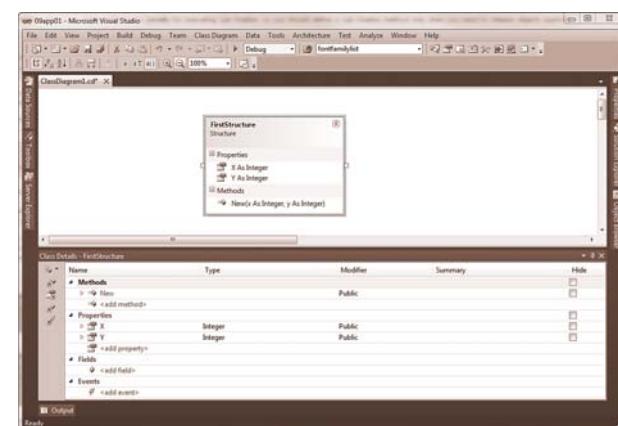
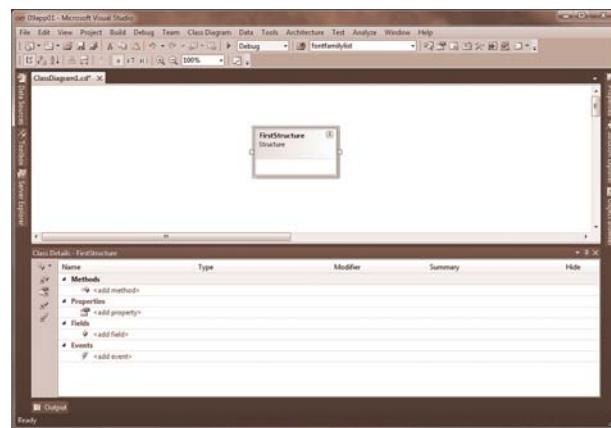
That's because the main difference between a class and a structure when you're creating the definition is that a structure always inherits directly from `System.ValueType`.



Add a couple of properties to the structure called `X` and `Y`, both `Integers`, and a constructor that takes those same properties as parameters.



There's a potential gotcha here. The Class Designer will let you create a default (parameterless) constructor. But default constructors aren't allowed in structures, and the compiler will complain if you try to build the file.



PUT ON YOUR THINKING HAT

Structure members are implemented in the same way as class members. Open `FirstStructure.vb` and write the code to auto-implement the two properties and initialize them in the constructor.



HOW'D YOU DO?

Are you surprised that you already knew how to implement a Structure? Well done!

```
Public Structure FirstStructure
    Sub New FirstStructure(Int32 x, Int32 y)
        Me.X = x
        Me.Y = y
    End Sub

    Public Property X As Integer

    Public Property Y as Integer
}
```

[modifier] Structure <name>
[Implements <interface> [, ...]]

► <data member>
[<methods>]
End Structure

Public Structure SecondStructure

- Structures must implement at least one member to store data (like a property or a field, for example) and may have methods as well.

Structures can implement interfaces, but it isn't done very often and should be done with care because of the boxing issues we'll discuss later in this chapter.

INSTANTIATING STRUCTURES

As I said, structures don't allow you to explicitly define a default constructor. The compiler will create one for you, of course, but it's unusual to actually call it. Instead, you'll just assign the value properties directly, which is clearer and marginally faster:

```
a As FirstStructure = New FirstStructure()
```

These are equivalent statements.

```
a As FirstStructure  
a.X = 5  
a.Y = 2
```



Because structures can be instantiated this way, without explicit initialization, it's important that the properties have reasonable default values. Since the properties of a structure should themselves be value types, it's not often an issue, but it is something to be aware of.

But in fact it's best practice to provide a constructor overload that allows the structure to be created and initialized in a single step, as we did with our example:

```
a As FirstStructure = New FirstStructure(5, 1)
```

If you're using a constructor overload that takes parameters, you must use `New`.



TAKE A BREAK

Yes, structures really are that simple, and we've finished our examination of them for now. Why don't you take a break before completing the Review and moving on to enumerations?



REVIEW

Given the property and usage descriptions below, would you choose to implement each type as a class or a structure?

The `Cup` type has two properties, `Tablespoons` and `Teaspoons`, both `Integers`, and a single method, `Convert()`.

The `Teaspoon` type inherits from the `Measurement` type. It has a single property, `Value`, an `Integer`, and two methods, `AsCups()` and `AsTablespoons()`.

The `Ingredient` type has 32 properties of type `String` and `Integer`, and 12 methods.

Implement a structure called `NewStruct` with one property called `NewValue` and one constructor. Use the class diagram and the editor as you wish.



ENUMERATIONS

The other value type in the .NET Framework is the enumeration. Unlike structures and the interfaces we'll look at in the next section, enumerations don't look anything like classes; they're simply sets of constant values that are assigned meaningful names.

Let's look at an example from that Scheduling application Neil & Gordon want us to write. It might define a method called, say, `AddToSchedule`, that takes the `Component` to be scheduled and the day of the week as parameters. There are three ways we might define the method:

```
Public Function AddToSchedule(item As Component, day As String)
```

The problem with this version is that you can't tell from the signature whether the method is expecting "Monday", "Mon" or "M". (Good programmers live in fear of "you know because you know".)

The problem with this version is that days don't really translate to numbers very well. Is 1 Sunday or Monday? Depends on when you start counting.

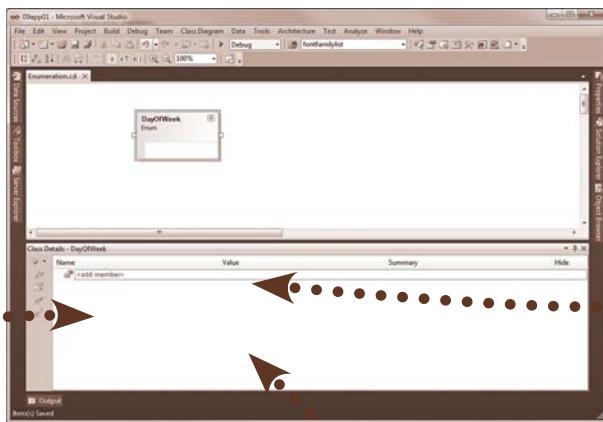
```
Public Function AddToSchedule(item As Component, day As Int32)
```

```
Public Function AddToSchedule(item As Component, day As DayOfWeek)
```

Using an ENUMERATION, `DayOfWeek` eliminates the possibility of confusion, and the method will get exactly what it needs. Easier for everyone involved.

CREATING ENUMERATIONS

When you add an enumeration to the Class Designer, Visual Studio first displays the New Enum dialog that allows you to specify a name, and then shows only one section in the Class Details Pane: Add Member:



The Details pane lets you select a value but not a type. The type is defined for the enumeration as a whole, not its individual members. By default, it's **Integer**.

- Unless you specify otherwise, the first member defined for the enumeration will have the value of zero, and each subsequent member will increase by one.

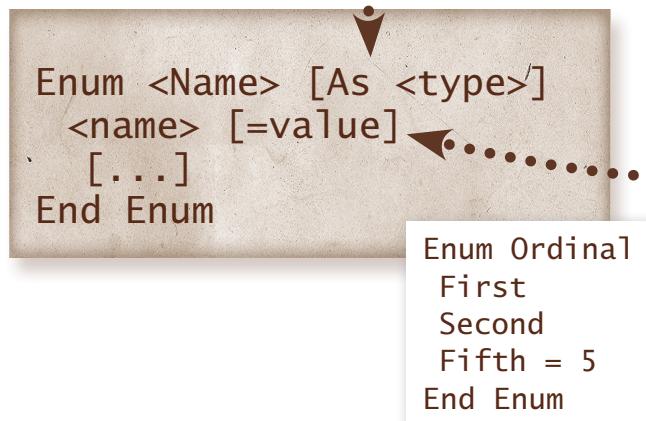
The exact value of an enumeration usually doesn't matter because they're treated as symbols, but you can specify a number if it has some meaning for your code.



ENUMERATION SYNTAX

The syntax diagram below looks really confusing, doesn't it? And yet, you've seen that enumerations are simple to create and use. Like any statement that defines a type, enumerations must begin with the type keyword, which in this case is `Enum`, followed by the name of the type, which must follow the rules for identifiers.

- All of the members of the enumeration must be of the same numeric type. By default, that type is `Integer`, and most of the time that's fine because you don't usually care what the actual value is. But if for any reason you want to specify a different type, you do so by specifying the type name after the name, as shown in the example.



After the identifier and the optional data type specification, you list the symbolic members of the type, each on a separate line. Again, you won't usually care what the value of the member is because it's used symbolically, but if you do, you can specify the value.

MAKE A NOTE



It would be reasonable to think that if you build an enumeration and specify it as a parameter, you needn't worry about anybody passing an invalid argument. Unfortunately, there's a way around the restriction. The .NET Framework lets you cast any `Integer` to your `Integer` enumeration:

```
CType(215, DaysOfWeek)
```

That statement example will make it through the compiler just fine, despite the fact that the maximum value of the `DaysOfWeek` enumeration is 6. You'll still have to check for invalid arguments. Sigh.

ENUM METHODS

In an amazing bit of compiler sleight-of-hand, the enumerations that you create using the `Enum` keyword are actually instances of a `System.Enum` class. (Don't let the fact that they're instances of a class confuse you. `System.Enum` descends from `System.ValueType`, which makes them value types, not reference types.)

As you know, members are defined in the type, not the instance, and since enumerations are instances, you can't add members to them. But you can reference the members of `System.Enum` itself, and it has some useful ones:

<code>GetName(type, value) As String</code>	Retrieves the name of the constant with the corresponding numeric value inside the specified enumeration
<code>GetNames(type) As String()</code>	Gets an array of all the constant names in the specified enumeration
<code>IsDefined(type, value) As Boolean</code>	Returns a Boolean indicating whether the numeric value provided is defined in the specified enumeration
<code>Parse(type, string)</code>	Returns the numeric value of the specified constant name in the specified enumeration

The constant names in an enumeration must comply with the rules for identifiers, so they're not always as useful as you might like—you wouldn't want to show the user "DoSomething" with no spaces, for example—but it's surprising how often choosing the names carefully can save you a lot of work.

Notice that all of these methods require the enumeration type as a parameter. They're actually static methods that are available from the class itself, not its instances. You can pass the type argument by passing the name of the enumeration wrapped in the `GetType()` operator, which returns an instance of the `System.Type` class that represents the type itself.

```
ConstantName As string= Enum.GetName(GetType(DayOfWeek), 0)
```



PUT ON YOUR THINKING HAT

Write a method that accepts a `DayOfTheWeek` as an argument and displays the name of the day in a `MessageBox`. (Hint: You can use the example of `GetName` on the previous page to help you out with the syntax.)



HOW'D YOU DO?

Here's my version. It's okay if yours looks a little different, as long as it's understandable and it works.

```
Public Sub ShowDay(DayOfWeek day)
    If day > 6
        Throw New ArgumentException("Invalid day argument.")
    Return
End If

DayName As String =Enum.GetName(GetType(DayOfWeek), day)
MessageBox.Show(DayName)
End Sub
```

Remember, there's no guarantee that the value passed will actually be a valid member of the enumeration.

You don't have to put the `String` returned by `GetName()` in a separate variable; it just makes the code a little easier to read.



TAKE A BREAK

That's it for enumerations. They're simple (but useful) types. Why don't you take a break before you complete the Review on the next page and we move on to the last of the .NET Framework types: Interfaces?



REVIEW

Write the statement that defines each month in the year, assigning January to 1, not the default of 0:

List three benefits of using enumerations as method parameters:

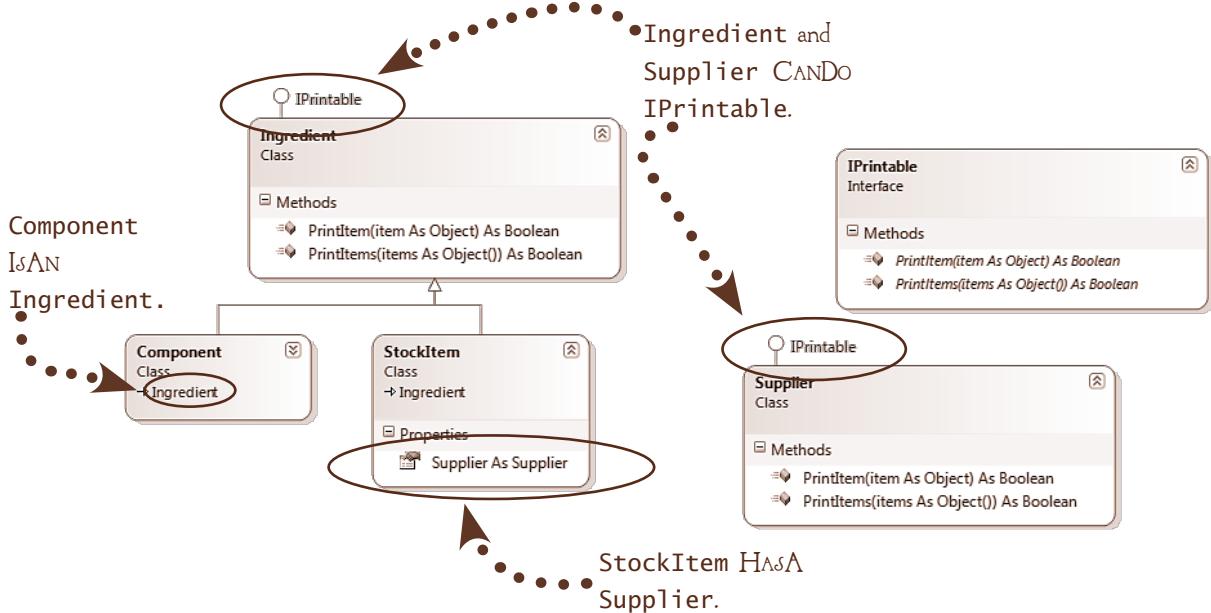


INTERFACES

There are three primary relationships that can exist between types in your application: IsA, HasA and CanDo. We'll be looking at these relationships in detail in Chapter 12, but let's take a quick look now so that we can put interfaces in perspective.

A class might be a specialized type of another class, like `Component` and `StockItem` are specialized versions of their base class `Ingredient` in the diagram below. We call this an IsA relationship, and it's implemented using inheritance. Alternatively, a class might contain another, as the `StockItem` class contains a `Supplier`. This is a HasA relationship, and it's implemented using fields and properties.

You've already seen and implemented IsA and HasA relationships, but there's a third kind, the CanDo relationship. It's implemented using the last of the .NET Framework types we'll examine in this chapter, the INTERFACE. In the example below, the `Ingredient` and `Supplier` classes have no obvious relationship other than one of the children of `Ingredient` HasA `Supplier`. But it's reasonable that both of them might be able to print themselves, and so they both implement the `IPrintable` interface, which is shown on the diagram using what's affectionately referred to as "lollipop notation".



INTERFACES & CLASSES

The syntax for defining an interface is almost identical to that for defining classes, but with two differences:

- An interface can contain only member definitions, not implementation. Essentially, an interface defines something a class (or other interface) can do but leaves the doing of it up to the class itself.
- Interfaces cannot define fields, constructors, destructors or other types.

```
Interface <Name>
[Inherits <interface> [,...]]
...
End Interface
```

• By convention, interface names begin with I and are usually adjectives that describe behavior (IPrintable).

```
Interface IMyDoable
Public Property MyProperty As Integer
Public Sub ShowHello()
End Interface
```

```
Public Class MyClass
Implements IMyDoable
```

```
Private m_myProperty As Integer
Public Property MyProperty As Integer
Get
    Return m_myProperty
End Get
Set
    m_myProperty = value
End Set
End Property

Public Sub ShowHello ()
    MessageBox.Show("Hello!")
End Sub
End Class
```

ON YOUR OWN

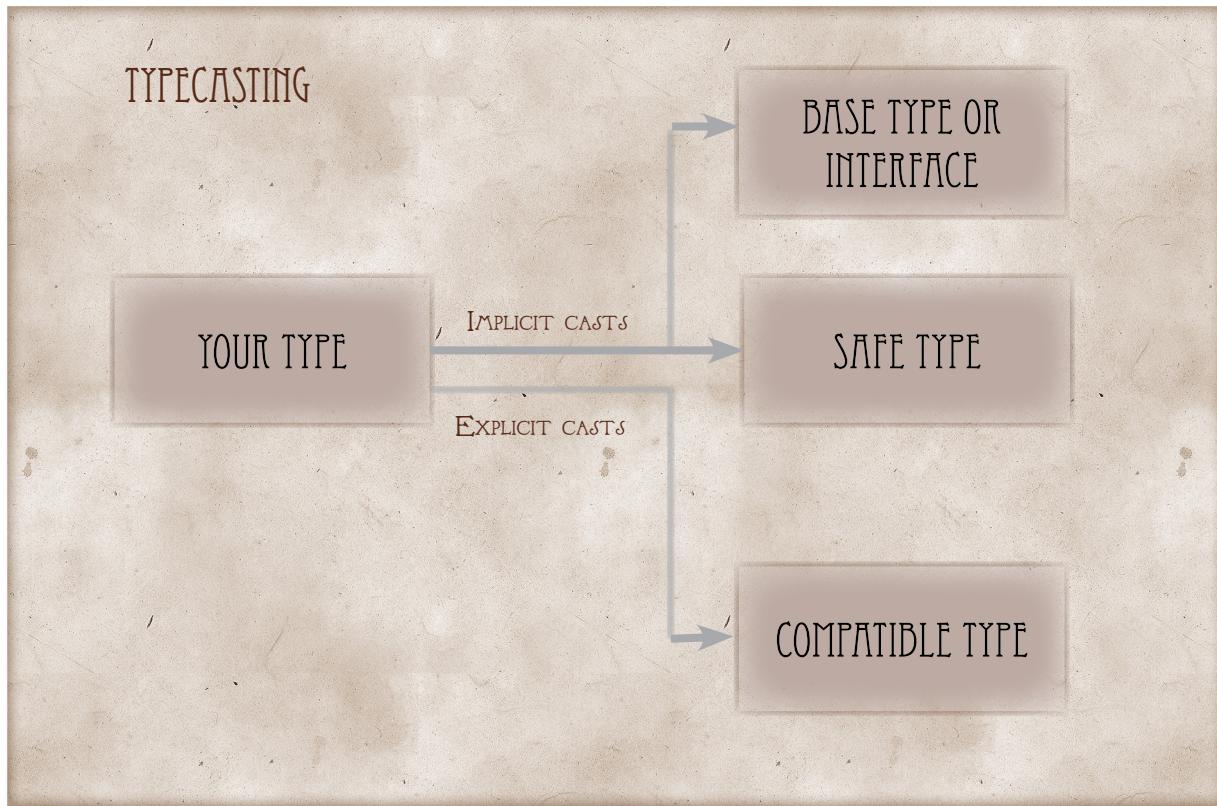
You may not realize it, but you already know how to create an interface in the Class Designer. Give it a try: Create the IPrintable interface and define its two methods.



WORKING WITH TYPES

Most of the time your code will instantiate types and manipulate their values and behavior.

But sometimes you'll need to step back and manipulate the types as types. We've already seen one example of this when we needed to use the `GetType()` operator to pass our enumeration to the methods of the `System.Enum` class. 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. In this section we'll look at casting, and also you can use REFLECTION to determine information about your types at runtime.



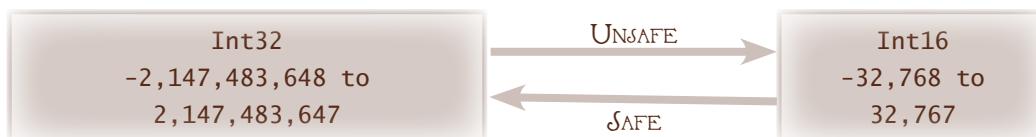
IMPLICIT CASTING

Implicit casting takes place when the compiler silently converts from one type to another without you needing to do anything in your code. There are two situations in which VB will help you out like this: Any type can be implicitly cast to its base class or one of the interfaces it implements, and numeric values will be implicitly cast if the destination type is larger (e.g., can contain more values) than the source type.

SAFE CASTS

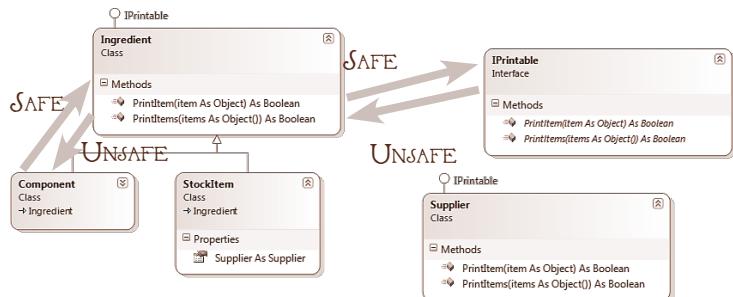
We'll look at the numeric data types that the .NET Framework exposes in the next chapter, but you've already seen `Int32` and `Int16`, both of which hold whole numbers. Since there's no possibility of losing data if you put the value of an `Int16` into an `Int32`, the cast is considered "safe" and the compiler will implicitly cast it for you.

The reverse isn't true, however. `Int32` can hold values that can't be contained in an `Int16`, so the compiler will complain if you try to treat an `Int32` as an `Int16` without stating your intentions. (We'll see how to do that in a minute.)



INHERITANCE CASTS

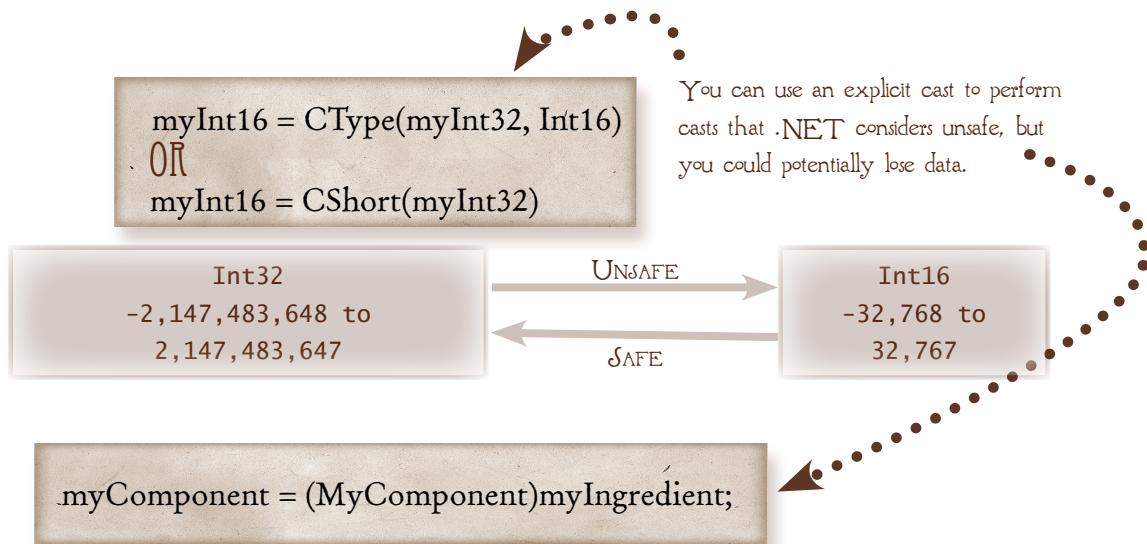
Remember that inheritance is usually an instance of an `IsA` relationship, while interface implementation is usually an instance of a `CanDo` relationship. In our sample class hierarchy, for example, both `StockItem` and `Component` inherit from `Ingredient`. That means that they expose all of the members that are defined for their parent. Similarly, both `Ingredient` and `Supplier` implement the `IPrintable` interface, so they expose all its members. In both cases, it's safe to treat the derived type as the base type or interface, but the reverse isn't true.



EXPLICIT CASTS

Like implicit casts, explicit casts are also performed between numeric types and between base and derived classes, but an explicit cast is used when going the opposite direction: from a larger value to a smaller value (Int32 to Int16, for example), or from a base class or interface to a derived class.

You specify an explicit cast either by passing the name of the desired type to the CType() function or by calling one of the intrinsic Visual Basic conversion functions. Be aware: even with explicit casting, these remain unsafe conversions, and can result in either data loss (in the case of numeric casts) or exceptions (in either case).



MAKE A NOTE

Visual Basic exposes dedicated conversion functions like CShort() and CStr() for all the core data types, but you can always use the CType() method. As you've probably worked out by now, I don't like memorizing things (like the abbreviation for a dedicated conversion function), so I almost always use CType(), and that's what you'll see in the examples in this book. But there's no discernable difference between the functions, so you can use whatever you find more convenient. For more information and a list of the available Visual Basic functions, search for "type conversion functions" on MSDN.

ERROR-FREE CASTS

Visual Basic supports two other mechanisms for casting that let you avoid exceptions when casting objects. The `Is` operator returns a Boolean value indicating whether the source type is the destination type, or derived from the destination type either through inheritance or interface inheritance.

```
If sourceObject Is DestinationType Then  
    destinationObject = CType(sourceObject, DestinationType)  
    //do something  
Else  
    Throw New ArgumentException("Please pass a DestinationType.")  
End If
```

It's common to use the `Is` keyword in a pattern like the one shown—test then cast. The `TryCast()` function is a slightly more efficient way of achieving the same objective. If the source object can be cast to the destination object, the `TryCast()` function does so. If it can't, rather than throwing an exception, it returns the value `Nothing`, which is the value assigned to an object variable that does not refer to an object:

```
destinationObject as Destination = TryCast(sourceObject, DestinationType)  
If destinationObject <> Nothing Then  
    //do something  
Else  
    Throw New ArgumentException("Please pass a DestinationType.")  
EndIf
```

PUT ON YOUR THINKING HAT



Using the sample class hierarchy for this chapter, write the code to safely perform each of these conversions. Assume that in each case you've declared variables of each type named `MyWhatever`.

Supplier to `IPrintable`

Component to `Ingredient`

StockItem to `IPrintable`

`Ingredient` to Component



HOW'D YOU DO?

Supplier to IPrintable:

```
myIPrintable = mySupplier
```

Supplier implements **IPrintable**, so this is a safe cast and can be done implicitly. (But it wouldn't be wrong to cast it explicitly.)

StockItem to IPrintable:

```
myIPrintable = myStockItem
```

Even though **IPrintable** is implemented by the base class of **StockItem**, not **StockItem** itself, this is still a safe cast.

Component to Ingredient:

```
myComponent = myIngredient
```

Component is the base class for Ingredient, so this is also a safe cast. But (Component)MyIngredient wouldn't be wrong, and might make your intentions clearer.

Ingredient to Component:

```
myComponent = CType(MyIngredient, Component)
```

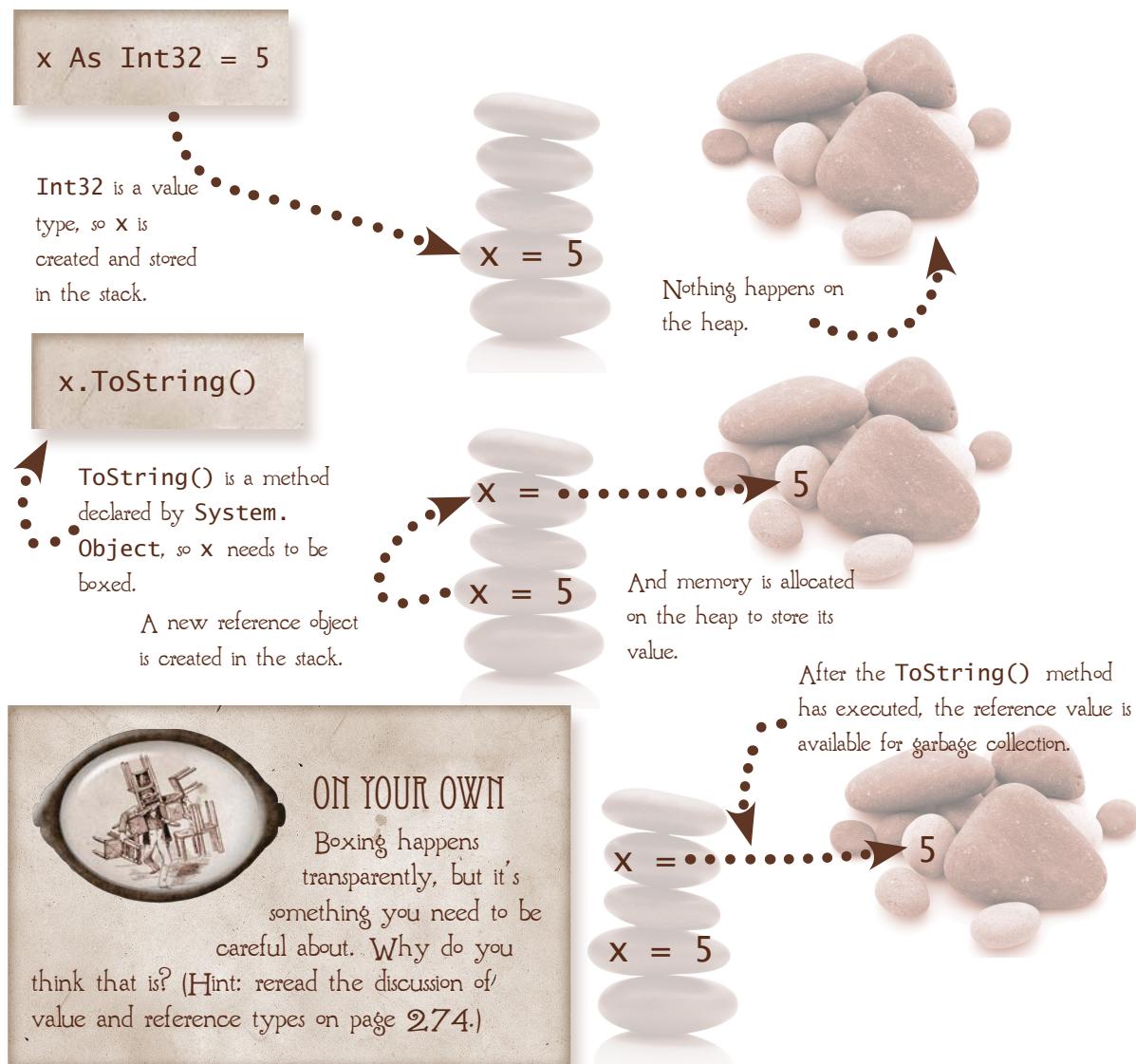
or

```
myComponent = TryCast(MyIngredient, Component)
```

This is an unsafe conversion, so it must be done explicitly. The first version is safe if you're absolutely sure that MyIngredient really is a Component (which generally means you've only just declared it). The second is always safe, but you need to test MyComponent for null values before you use it.

BOXING & UNBOXING

You know that VB will implicitly cast any type to one of its ancestors or an interface that it or one of its ancestors implements. Sometimes this means converting a value type to a reference type and vice versa, a process called BOXING and UNBOXING. It can happen whenever you cast a value type to an interface or to the base class of all .NET Framework types, `System.Object`.





ARGUMENTS, REVISITED

Let's do a little experiment to explore one more difference between value and reference types. Try to build the project just from the instructions, but if you run into trouble, my version is on page 302.



Create a new WPF project and add a button to it.



In the code file, define a class called `Elephant` within the `MainWindow` class. (Remember, classes can have classes as members.) Add a single auto-implemented property to `Elephant` called `Name`.



Add two methods to the `MainWindow` class (not the `Elephant` class). Call them both `ChangeMe`. The first should accept an `Int32` and, in the body, increment its value. The second should accept an `Elephant` and change its `Name` property to "Sally" in the body.



In the `button1_Click` event handler, create an `Int32` variable and an `Elephant` variable. Set the `Int32` to 5 and the `Elephant.Name` property to "George".



Still in the `button1_Click` event handler, call the `ChangeMe()` method for each of the variables you created, and display the results in a `MessageBox`.



PUT ON YOUR THINKING HAT

Before you run the application, what do you think the values displayed in the MessageBox will be?

Elephant.Name:

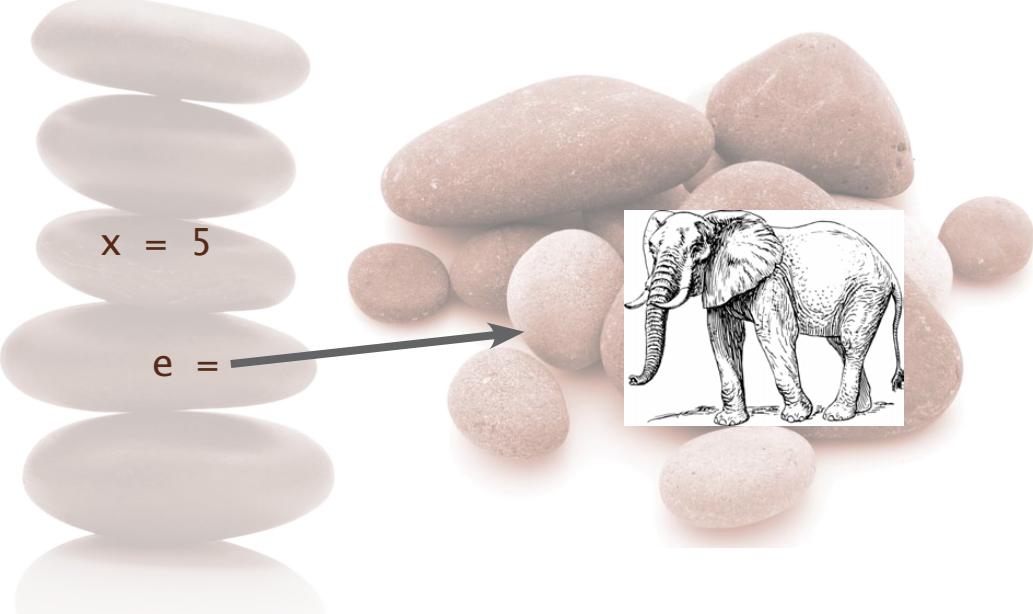
Int32



Now run the application and click the button. Did the MessageBox display what you expected?

It might help you understand what happened if I tell you that, by default, when you pass a variable as an argument to a method, it's pass by value—what the method actually receives is a copy of what's on the stack, and when the method ends, the copies passed to it are discarded.

So what's on the stack for each of our two variables?



WALK-THROUGH CODE

Here's what my version of the walk-through code looks like. Yours doesn't have to match exactly, as long as the basic elements are in place.

```
Public Partial Class MainWindow
Inherits Window

    Public Class Elephant
        Public Property Name As String

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub button1_Click(sender As Object, e As RoutedEventArgs)
            Dim MyInt As Integer
            Dim MyPet As Elephant = New Elephant()

            MyInt = 5
            MyPet.Name = "George"

            ChangeMe(MyInt)
            ChangeMe(MyPet)

            MessageBox.Show("My pet's name is " +
                           MyPet.Name + " " + MyInt.ToString())
        End Sub

        Private Sub ChangeMe(Int32 x)
            x += 1
        End Sub

        Private Sub ChangeMe(Elephant x)
            x.Name = "Sally"
        End Sub
    End Class
```

PASSING BY REFERENCE

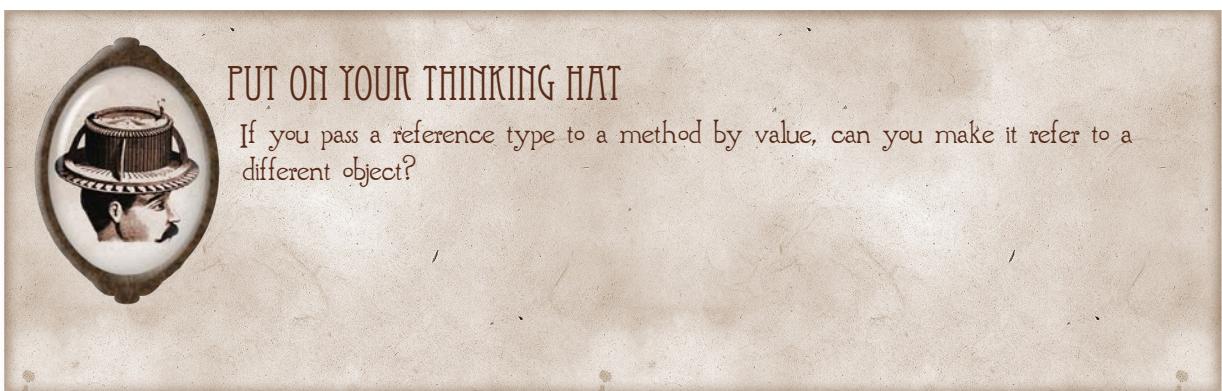
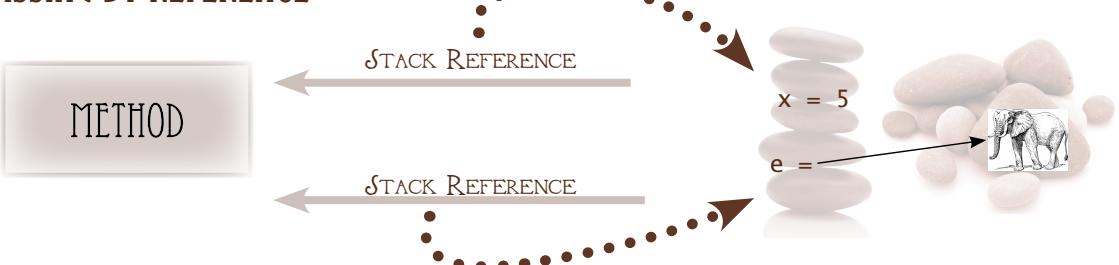
When you pass a variable as an argument to a method, the method receives a copy of the value on the stack. In the case of value types, it receives a copy of the value. In the case of reference types, it receives a copy of the reference to the values in the heap (in other words, reference by reference and value by value).

The arguments are discarded when the method exits, but changes made through the reference (that is, to values in the heap) persist. That's why the change to the `Name` property was maintained, but the change to the value of the `Int32` variable wasn't.

PASSING BY VALUE



PASSING BY REFERENCE





IN A NUTSHELL—ARGUMENTS

You don't need to worry too much about the details of what's happening with the heap and the stack. Here are the important points:

- Value types passed by value CANNOT be changed.
- The properties of reference types passed by value CAN be changed.
- The object referenced by a reference type passed by value CANNOT be changed.
- The properties of value types passed by reference CAN be changed.
- The object referenced by a reference type passed by reference CAN be changed.



HOW'D YOU DO?

No, you can't. Just like passing a value type, the changes you make to what's on the stack (the reference) will be discarded when the method exits.

```
void DoSomething(Elephant x)
{
    y = new Elephant();
    x = y;
}
```



This line has no real effect because the argument passed to this method will still refer to the original object when the method exits.

REFERENCE SYNTAX

The Visual Basic keyword for passing a value by reference is `ByRef`. Both the method definition and the method call must use the keyword:

```
Sub DoSomething(ByRef x As Elephant)
    y = New Elephant()
    x = y
End Sub
```

`DoSomething(ByRef MyPet)`

This line works now,
because the change will
be preserved.

These must
match.



MAKE A NOTE

The `ByRef` modifier is part of the signature, so you can declare two versions of a method differing only by whether they can change their parameters. But it doesn't seem like a very good idea, does it? No, I don't think so either.



TAKE A BREAK

And that's it for creating .NET Framework types, at least for now. We'll be coming back to a lot of the subjects we've discussed in the last couple of chapters as we move through the book, particularly in the "Principles" section, but for now, take a break before you come back to finish the final Review.



REVIEW

List at least three differences between classes and structures:

One of the following is NOT a good reason to use an enumeration. Which one?

- Enumerations make passing parameters more reliable.
- Enumerations make a list of options easier to remember.
- Enumerations guarantee that the argument passed to a method will be valid.
- You can (sometimes) avoid a lot of code by using the constant names.

The statement `Property MyProperty As Integer` can mean two different things depending on where it's used. What two types allow the statement, and what does it mean in each case?

Using the example class hierarchy from this chapter, what code would you need to pass an instance of `Component` to a method with the signature `DoSomething(Ingredient)`?

What keyword do you need to use to persist changes made to an argument once a method exits? Where must you use it?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

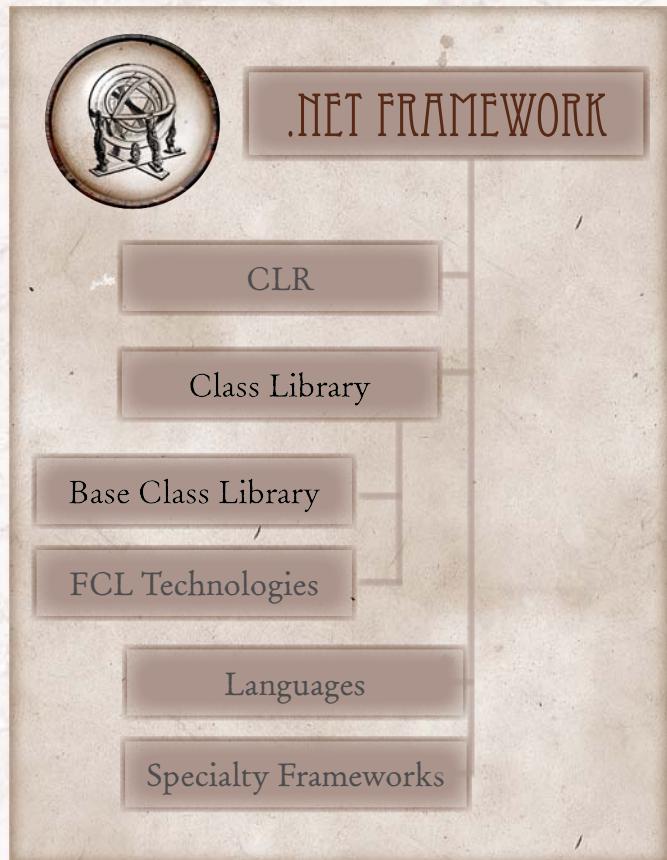
Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



THE CLASS LIBRARY, PART 1

10

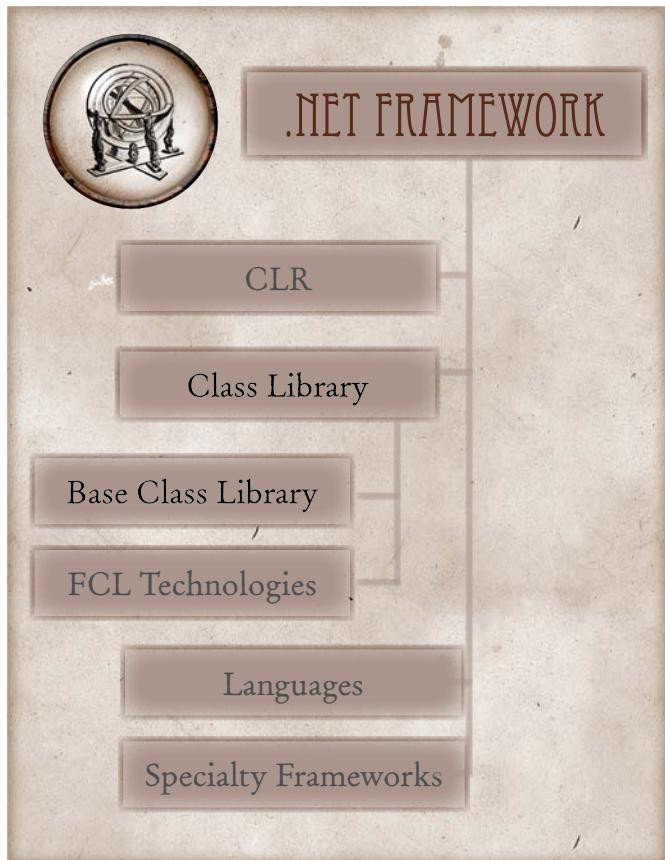
In the last couple of chapters you've learned the basics of how to create .NET Framework types and members. Now it's time to back up a bit and look at the vast library of types that are provided for your use in the Framework Class Library (FCL). In this chapter we'll look at the core data types that will help you manipulate numeric data, character data, and dates and times. In the next chapter we'll move on to types that let you handle multiple instances of an object and a special construct called a generic that lets you specify a type at runtime.





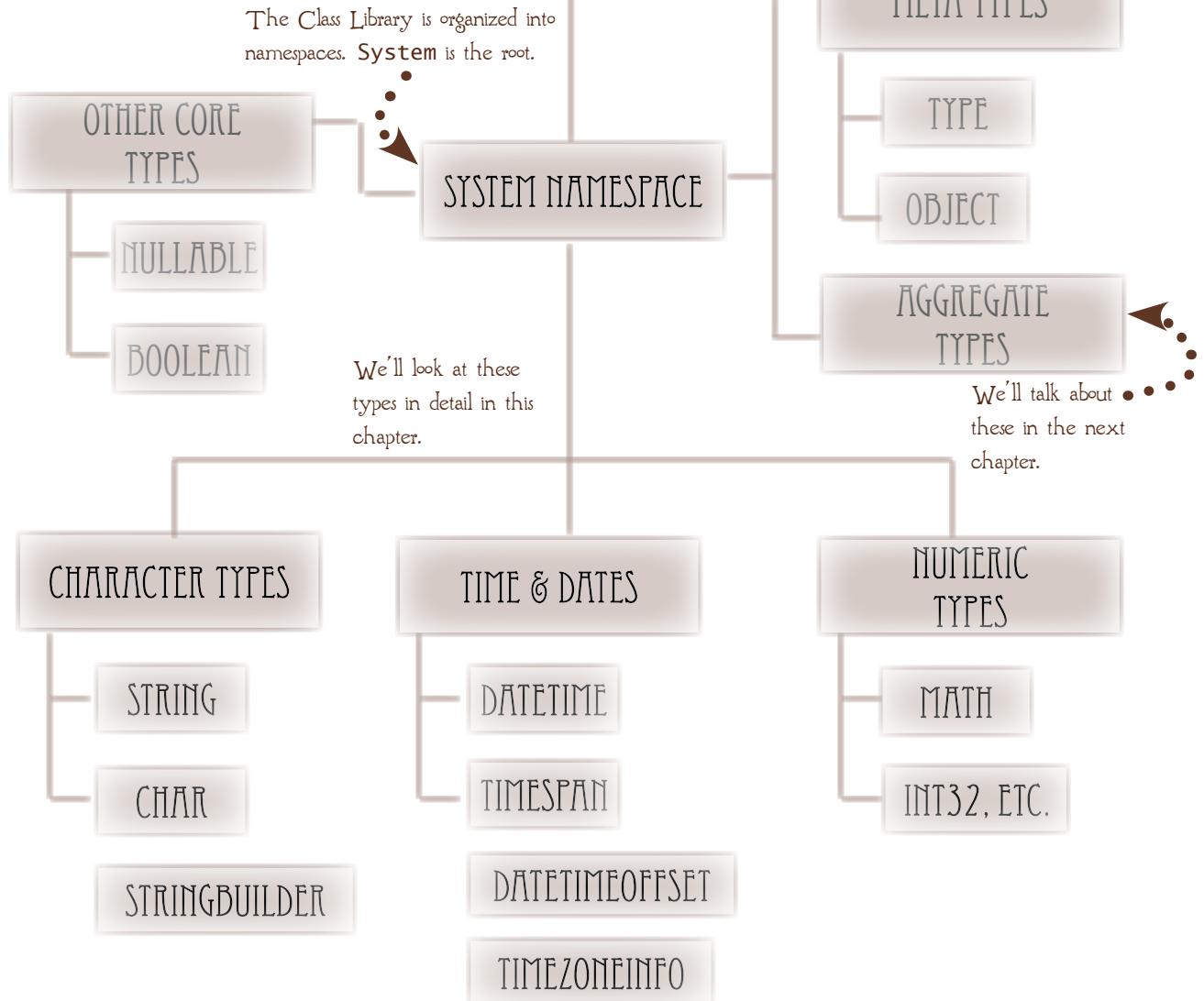
FITTING IT IN

Like classes, structures, enumerations and interfaces are implemented using Visual Basic type and member keywords (many of which you already know) and are eventually translated into .NET Framework CTS (Common Type System) types. In this chapter we'll look at these remaining types and also at how they fit into the CTS.



.NET FRAMEWORK

CLASS LIBRARY



AN EXAMPLE...

To get a sense of just how much functionality the .NET Framework Class Library provides for you, take a look at this kitchen calculator and the list of classes that are involved in creating it.

WINDOW—The **Window** class acts as a container. It displays the main calculator window, the title bar, and the **Minimize**, **Maximize** and **Close** buttons. It provides the functionality to move, resize and close the application.

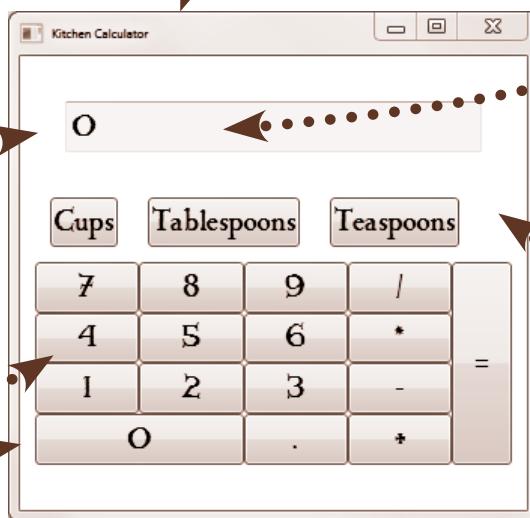
TEXTBOX—The output of the calculation is shown in a **TextBox**. It displays text in the typeface and size you choose and provides all the standard Windows editing functions like cut, copy and paste. It even lets you spell check its contents.

BUTTON—These widgets are instances of the **Button** class. Buttons know how to display themselves and how to respond to click events.

GRID—The **Grid** knows how to display its contents in rows and columns, and can resize and rearrange them when the window size changes.

STRING—Among other things, the **String** class knows how to create itself from a set of characters, compare itself to another string, and find a single character inside itself.

INTEGER—It knows how to perform basic mathematical functions on itself and display itself as a string.



RADIOBUTTON—The three buttons at the top of the form are instances of the **RadioButton** class. They know that they're in a group, and when the user clicks one of them, it not only stays clicked, it "unclicks" all the other buttons in the group.

Can you imagine writing all this from scratch?



TASK LIST

In this chapter, we'll start by looking at how the FCL is organized and a couple of ways to find your way around it. Then we'll examine the types that support three core categories of data: numeric, character, and chronological.



NAMESPACES

We'll begin by looking at namespaces, the primary way of organizing types into logical groups in the .NET Framework, and how you can access and create them.



THE OBJECT BROWSER

We've already used the Object Browser, but in this chapter we'll look at it in much more detail—it's one of the best tools for finding your way around the class library.



NUMERIC DATA

So far we've primarily used the `Integer` class for storing numeric data, but the FCL provides almost a dozen (eleven, to be exact) types for storing different kinds of numeric data, which we'll examine here. We'll also take a quick look at the `Math` class, which provides support for trigonometric, logarithmic and other advanced mathematical functions.



CHARACTER DATA

After we've finished looking at numbers, we'll look at text, specifically the `String`, `Char` and `StringBuilder` classes that let you store and manipulate character data.



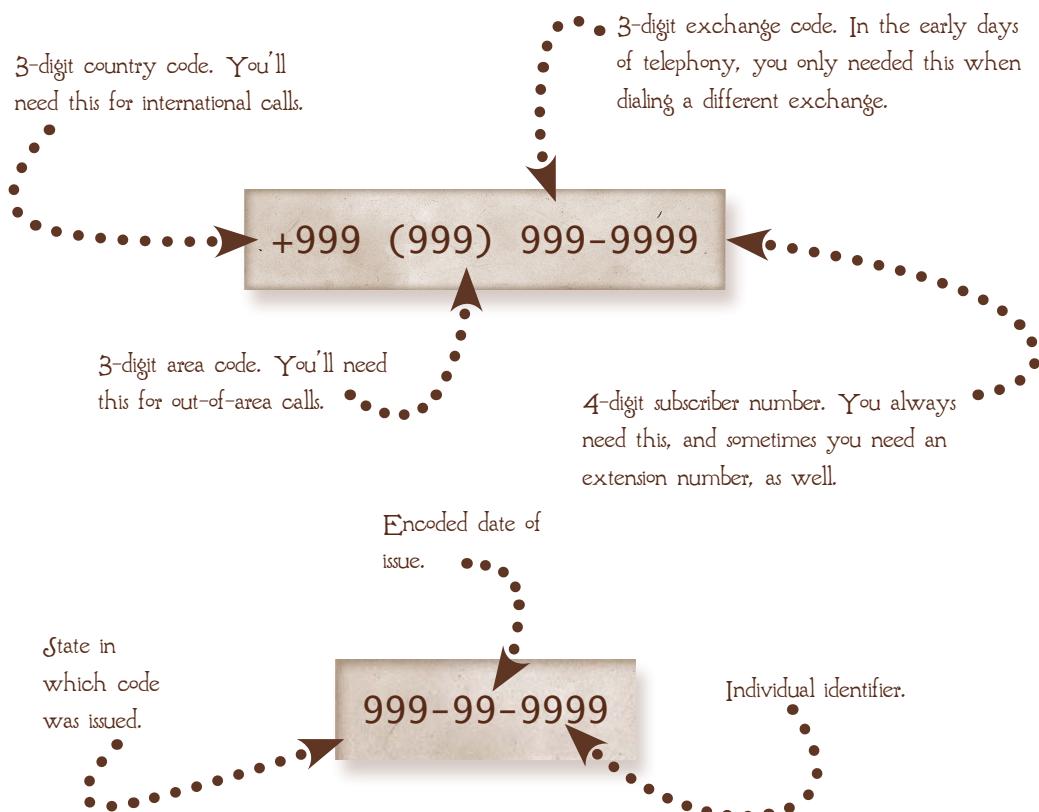
CHRONOLOGICAL DATA

Most people tend to think of dates and times as numbers, and it's true that they're often represented that way, but time is a tricky thing, and the FCL provides some types that make it, if not easy, at least easier to manipulate.

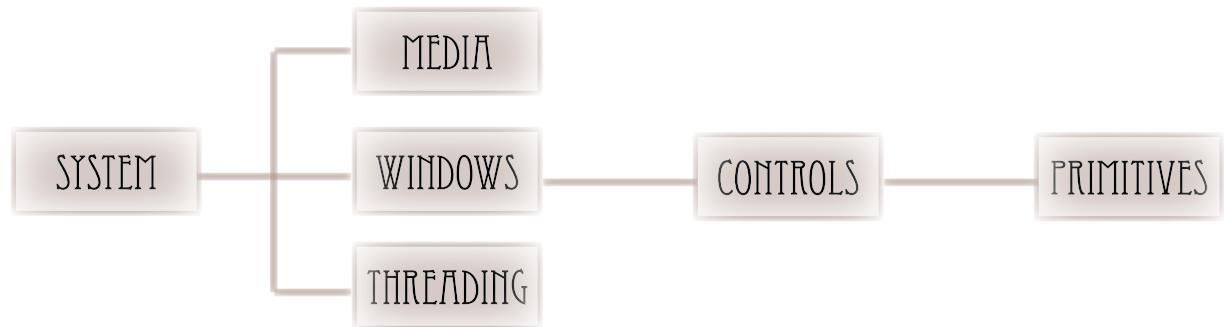


NAMESPACES

Here are two identification codes you're probably used to seeing. The first is a North American telephone number, the second a US Social Security Number. In both cases, we don't usually think too much about what the components of the final code mean; we just use them as black boxes. But they do have meaning, and there are times when it's useful to decode them.



There are two ways of identifying a class in the FCL and, like a phone number, sometimes you can ignore parts of them, and sometimes you can't. NAMESPACES are used to organize types into logical groups. This is how you'll refer to a type in your code, just like the phone number you use to call someone.



THE OBJECT HIERARCHY defines the capabilities of the type. You'll need it to understand where the type lives in the hierarchy, because it inherits the capabilities of all its ancestors.

This is the way
hierarchies are displayed
in MSDN.
•

System.Object
System.Windows.Threading.DispatcherObject
System.Windows.DependencyObject
System.Windows.Media.Visual
System.Windows.UIElement
System.Windows.FrameworkElement
System.Windows.Controls.Control
System.Windows.Controls.ContentControl
System.Windows.Controls.Primitives.ButtonBase
System.Windows.Controls.Button

The class hierarchy and namespace
hierarchy aren't related. `Button`,
for example, inherits from types
in the `Threading`, `Media`
and `Controls.Primitives`
namespaces, but `Button` itself is in
the `Controls` namespace.



USING NAMESPACES

There are two things you need to do in order to use the types declared in a given namespace. First, you have to tell the compiler about the namespace by setting a reference, and then you tell the code file about it with the `using` directive.

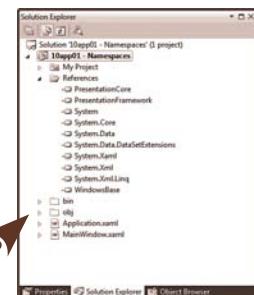
(Technically, the `using` directive isn't necessary, but as we'll see, it saves a lot of typing.)

To get our feet wet, let's create a very cool version of "Hello, World!" using the `System.Speech` namespace.

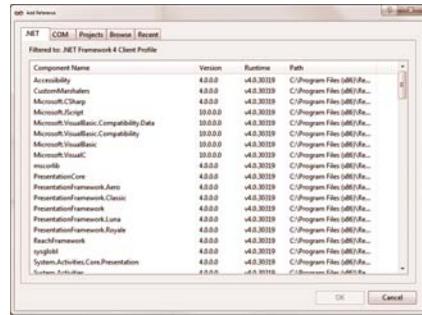


Create a new WPF project and add a button to the window.

When you create a project from a template, Visual Studio will add references to the namespaces you're most likely to use. (That's part of what a template does.) You can see what references are set in the Solution Explorer. Here are the ones that are added for a WPF project.

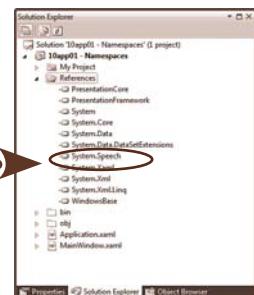


We'll need a reference to the `System.Speech` namespace. You add references from the Add Reference dialog, available from the Project menu and the Reference node context menu in the Solution Explorer. Open it now, and display the .NET tab.



Find `System.Speech` in the list, select it and click OK. Visual Studio should add it to the list in the Solution Explorer.

You may need to click the Show All Files button in the Solution Explorer before you see this.





Now let's take it for a test run. Double-click the button you added to your window and add the following lines to the event handler:

```
Dim sr As System.Speech.Synthesis.SpeechSynthesizer  
sr = New System.Speech.Synthesis.SpeechSynthesizer()  
sr.Speak("Hello world!")
```

Run the application and click the button. Not the greatest artificial voice in the world, but hey, what do you want in three lines of code? The `System.Speech` namespace actually provides a great deal of control over how speech is produced, including selecting different voices and emphasizing words. It also supports speech recognition, so you can talk back to your applications. But unfortunately we don't have space to explore its coolness here.



We got the application to talk to us in three lines of code, but they were three very long lines. Even with Intellisense, wading through all those nested namespaces is a nuisance. The `Imports` statement can save us all that.

Imports [<alias> =] <namespace>

```
Imports Speech = System.Speech.Synthesis
```

```
sr = New SpeechSynthesizer()
```



PUT ON YOUR THINKING HAT

Can you figure out what you need to add to the end of the list to avoid typing `System.Speech.Synthesis` every time? Try it, and then delete the namespaces from your event handler. Everything still work?



HOW'D YOU DO?

Here are the details of the syntax, in case things didn't quite work out as planned...

`Imports System.Speech.Synthesis;`



IN A NUTSHELL — THE IMPORTS DIRECTIVE

- You must add a namespace reference to the project before you can reference it in the `Imports` statement.
- The optional alias allows you to create a nickname for the namespace. You'll need to do this if you're using two (or more) namespaces that have types with the same name.
- Unlike almost everything else in Visual Basic and the .NET Framework, the `using` directive has FILE SCOPE: It doesn't apply to a type; it applies to a physical code file. If you split your class across multiple files using the `partial` modifier, you'll need to add the `using` directive to each of them.

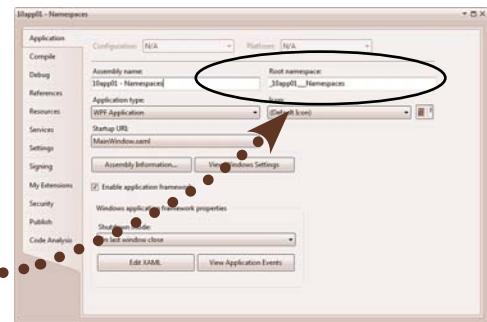
CREATING NAMESPACES

Now that you've seen how to use namespaces, how do you create them? By using the Namespace statement, of course:

```
Namespace <name> ◀••••••••  
‘types and other namespaces  
End Namespace
```

The name of the namespace must be a valid identifier with the exception that namespaces can be nested using the dot operator:
`NameSpace.SubNamespace`

Every object in a .NET Framework application is contained in a namespace, either explicitly or implicitly. By default, the types you define in your application will be enclosed in a default namespace named after the Project. You can change the namespace by editing the code. (Be sure to use Refactor, Rename from the context menu so it gets changed everywhere.) You can also change the default namespace in the Project Designer.



Changing the default namespace affects new items, but it doesn't change the namespace that Visual Studio creates for the items (like `MainWindow`) that it adds when you first create the project. You'll have to change those manually.



TAKE A BREAK

There's more to be learned about namespaces (there's always more to be learned about everything), but that's enough to see you through the vast majority of your programming projects. Why don't you take a break before completing the Review and we move on to finding our way around the FCL?



REVIEW

Are the following statements true or false?

Namespaces define the capabilities of a type.

Namespaces group types into logical sets.

You must declare a namespace with the `using` directive.

You must set a reference to a namespace before using it.

A `using` directive applies only to the file in which it is declared.

Namespaces can be nested.

The `Image` class is defined in the `System.Drawing` namespace. What two things must you do to use `Image` in your code?



THE OBJECT BROWSER

As anyone who's climbed the .NET Framework learning curve will tell you, one of the hardest things about learning to use it effectively is figuring out exactly what's there. It's huge. Really, really, huge. And Microsoft adds new components with every release of the Framework (and sometimes between releases).

The good news is that you don't have to learn it all, and you don't have to learn it all at once. As fun as our little speaking Hello, World application might have been, until you need a speaking application, there's no reason to have to learn the ins and outs of the speech synthesizer. It'll be there when (and if) you need it.

The other piece of good news is that Visual Studio provides a good tool for exploring the FCL. Called the Object Browser, it displays quite useful information about all the types in your project and all the types in the FCL. Best of all, it's an easy gateway to the MSDN help system (which has a learning curve all of its own.)

ON YOUR OWN



You've used the Object Browser a little bit in earlier chapters, but now that you know a little more about how the Framework Library is organized, poke around a little more. You can't hurt anything and it won't make any changes to your application.

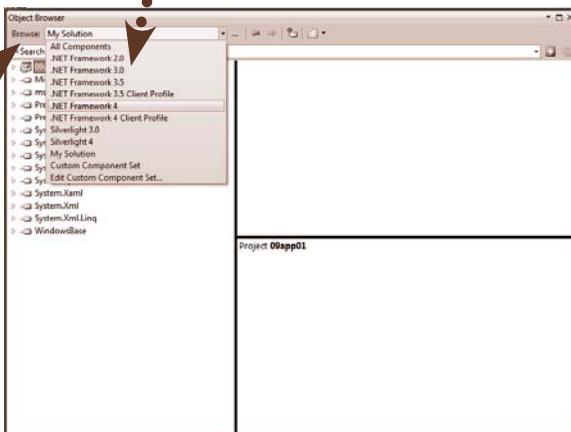
Why don't you look at the types `System.Speech` namespace and think about how they might be used to refine out little application?

THE BROWSER WINDOW

When you first show it, the Object Browser will display the namespaces you've told it you're interested in—the Solution itself, and the namespaces to which you've set references.

Use the Browse ComboBox to choose a different set of namespaces.

The .NET Framework <x> option gives you a list of all the namespaces in the FCL. (I told you it was huge, didn't I?)



Select a type here...

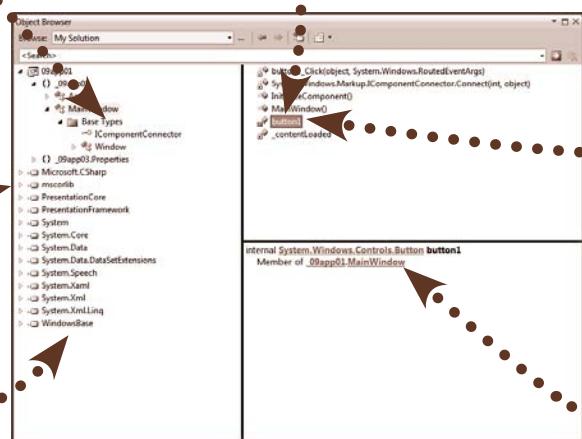
...and its members will display here.

You can expand the nodes in this pane to see what types are declared in a namespace.

Select a member here...

...and its details display here. If you've defined XML comments, they'll show up here, too.

These are the references set by default, plus the reference to Speech that we set earlier.

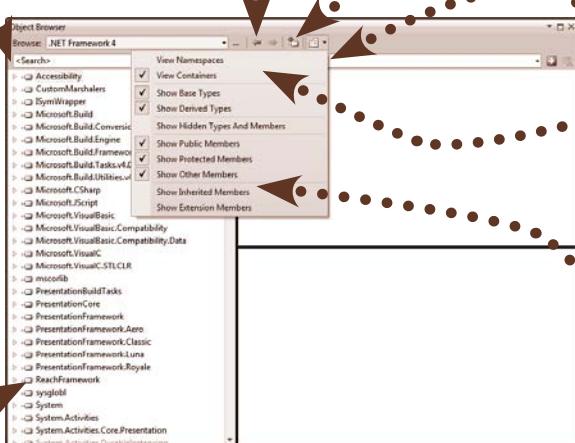


The search box lets you find a particular type quickly. You don't need to type the whole namespace; it will search for just a type name and return all matches.

Select any namespace, type or member and press F1 to open help for that item.

These toolbar buttons work just like Back and Forward in Internet Explorer and Windows Explorer.

If you select a namespace in the list, this button will let you quickly add a reference to it.



Like almost everything in Visual Studio, you have a lot of control over what's displayed in the Object Browser.

You'll probably want to choose View Namespaces, not Containers (the default), because that's how you'll reference the types you're looking for.

Show Inherited Members is handy when you're exploring because it shows you everything the type can do, not just what it declares itself.

PUT ON YOUR THINKING HAT



Set the Object Browser to display the entire .NET Framework 4 library by namespace, and look up the Summary of the following types and members:

System.Int32

System.String

System.Windows.Media.Brush

System.String.Compare(String, String)

Hint: This one's a member; look for it in the top-right pane.



HOW'D YOU DO?

`System.Int32`

“Represents a 32-bit signed integer.”

`System.String`

“Represents text as a series of Unicode characters.”

`System.Windows.Media.Brush`

“Defines objects used to fill the interiors of graphical shapes such as rectangles, ellipses, pies, polygons, and paths.”

`System.String.Compare()`

“Compares two specified `System.String` objects and returns an integer that indicates their relative position in the sort order.”



ON YOUR OWN

Look at the members of the `System.String` type in the Object Browser. Based on the method summaries, which method do you think you'd use to find the position of “W” in the String “Hello, World”? Try it.



NUMERIC DATA

VB and the .NET Framework support eleven different types for manipulating numeric data, but in practice one type for each kind of number is really efficient, and you'll almost always use it unless you have a very good reason not to: `Integer` for integers, `Double` for floating point numbers, and `Decimal` for currency values.

ON YOUR OWN



All of the numeric values are defined as structures in the `System` namespace. Using the Object Browser and MSDN, complete the following table of numeric types. There are useful (but incomplete) tables in the MSDN library—search for “Types Reference Tables”.

.NET Type	VB Keyword	Minimum Value	Maximum Value	Default Value
SByte				
Byte				
Int16				
UInt16				
Int32				
UInt32				
Int64				
UInt64				
Double				
Single				
Decimal				

INTEGER EFFICIENCY

There are two principles that underlie the efficient use of integers:

- Any given CPU will operate slightly faster if the integer size matches the register size.
- Multiplication is about twice as fast as division.

From these two principles, we can derive some basic heuristics:

USE AN INTEGER UNLESS YOU HAVE A GOOD REASON NOT TO

Most modern desktop computers have 32-bit processors, so a 32-bit integer will provide the best performance. But, of course, this is subject to change over time.

REPLACE DIVISION WITH MULTIPLICATION IN TIME-CRITICAL SECTIONS OF CODE

Don't obsess over this one, but if you're going to perform the same operation 10,000 times, it can make a difference:

```
For x As Integer = 1 To 10000
    a = b / 123
Next X
```

```
c = 1 / 123
For x As Integer = 1 To 10000
    a = b * c
Next X
```

This version will run
about twice as fast.

And one final tip: You may have noticed that the unsigned integers (UInt32, etc.) are not CLS-compliant. They'll work fine in a pure Visual Basic application running on a Windows operating system, but you may run into problems if you're using multiple languages or porting to a different version of the .NET Framework. You might not care, but these things do have a way of coming back to haunt you. My advice would be to avoid the unsigned integers without a very good reason.

FLOATING POINT NUMBERS

The floating point types `Single` and `Double` are used to represent numeric values with a fractional component (rational numbers for you math geeks out there). As with using `Integer` values to represent values, the `Double` is the go-to floating point type. It's faster and more accurate. Wait a second. "More accurate"? There's a "less accurate"? Well, here's the tricky thing about floating points: they're only approximations of a value.

Do you know the value of pi? 3.141592653589...yada yada. The value has been calculated to over a trillion digits. In the world of computers, we just don't have that many to work with. So, like pi, which is usually truncated to 3.14159, we pick a certain number of digits (called the precision) and arbitrarily decide that the rest don't matter. (In the .NET Framework, a `Double` value maintains 15 decimal digits of precision, while a `Single` maintains 7.)

Because floating point numbers are approximations, using them can be problematic. Operations that result in the same decimal real value may result in different floating point values. Setting the value of a floating point number in different ways (say, by assigning a literal or by assigning a value using the `Parse()` function) may result in different floating point values. Any given operation may result in different values on different versions of the .NET Framework.

Does this mean you should avoid them? No, of course not. It does mean that if you're going to do anything terribly complex, you may need some number theory that's beyond the scope of this book. (.NET Framework floating point numbers comply with the standards set by the IEEE 754 standard. If you're interested, there are hundreds of sites on the Web with details.)

IN A NUTSHELL — FLOATING POINT NUMBERS

For general use, there's only really one technique you need to remember:

Don't compare floating point numbers for equality. Instead, determine whether the difference between them is less than a specified amount. Instead of

If `MyFloat1 = MyFloat2`

use

If `(MyFloat1 - MyFloat2) < 0.0000001`

THE DECIMAL TYPE

Like a floating point number, the .NET Framework `Decimal` is also used to store real numbers, but it is less likely to suffer from the rounding errors that plague floating points. Unfortunately, it's also huge, and not as efficient as floating points. It is most often used to represent currency values, and it exposes a set of methods that are useful in that context: `Ceiling()`, `Floor()` and `FromOACurrency()`.

In addition, the `Decimal` type has four different versions of the `Round()` method that provide precise control over how rounding occurs, and a `Truncate()` method that ignores the fractional component completely:

`Round(value)` returns the nearest integer value. If `value` is halfway between two integers, the even number is returned. This is known as Banker's Rounding, or Rounding To Even.

`Round(value, places)` returns the value rounded to the specified number of decimal places, using Rounding To Even.

`Round(value, MidPointRounding)` returns the value rounded to the nearest integer using the specified member of the `MidPointRounding` enumeration, which has two values: `ToEven` and `AwayFromZero`.

`Round(value, places, MidPointRounding)` rounds `value` to the specified number of decimal places, using the specified rounding method.

`Truncate(value)` simply discards the fractional component.

INITIALIZING DECIMAL VALUES

Like any value type, `Decimals` can be initialized by assigning them a value directly. Constructors exist for `Single`, `Double`, `Int32`, `UInt32`, `Int64` and `UInt64` values. The `Decimal` also supports a constructor that allows you to specify each component of the number individually, which can sometimes be clearer:

```
Decimal(Int32 lowBits, Int32 midBits, Int32 hiBits,  
        Boolean isNegative, Byte Scale)
```

MATH CLASS

We've seen that all of the numeric types support the basic arithmetic operators, and that the `Decimal` type exposes some functions that are useful for financial calculations. The .NET Framework also has a class that exposes static methods and a couple of static fields that you can use to perform more advanced calculations.

We've seen a few static members before, but just to remind you: Because the members of the `Math` class are static, you don't need to instantiate it; you can simply call these methods using the name of the class: `Math.Pi`.

Most of the methods of the `Math` class are moderately advanced mathematical functions like `Log10()`, which returns the logarithm base 10 of the provided number, and `Cosh()`, which returns the hyperbolic cosine. I'm going to assume that if you need one of these functions, you'll know how to use it. But `Math` also provides a couple of methods that are generally useful when you're comparing numeric values.

`Max()` and its partner `Min()` return the greater or lesser of two values. They can save some typing and processing time:

`a = Max(x, y)`

IS EQUIVALENT TO

```
If x > y Then  
    a = x  
Else  
    a = y  
End If
```

The `Math` class also exposes the same four `Round()` methods that are exposed by the `Decimal` type. There are overloads for `Decimal` and `Double` types, while its partner the `Truncate()` method, which is also overloaded to accept either a `Decimal` or a `Double`, simply discards the fractional portion of the number, without rounding.



TAKE A BREAK

The numeric data types in the .NET Framework are straightforward; the trick is simply knowing what's available to you. Now that you've seen an overview, take a break before you move on to the Review and the next type of data: characters and text.



REVIEW

What .NET Framework type should you use to represent each of the following types of numeric data, unless you have a good reason not to?

Whole Numbers

Real Numbers

Currency

In what class would you look for a static method to perform an advanced mathematical function?

List three reasons you should avoid using unsigned numeric types if possible:



CHARACTER DATA

Except for a few idiosyncrasies, numeric data is represented and manipulated in VB and the .NET Framework more or less like it's represented and handled in the real world. The same can't be said for character data. As you probably know, computers store data as zeros or ones. (Actually, they store data in a bunch of tiny little capacitors, but we think of them as zeros and ones.) In order to represent an alphabet, the individual characters must be ENCODED.

In the .NET Framework, characters are encoded using the Unicode standard, which has replaced the older ASCII encoding as an industry standard. The basic units of encoding character data in the .NET Framework are `System.Char` (I pronounce it "char" like burnt wood, but some people say "care" like "character"), which represents a single character, and `System.String`, which represents an array of characters. (We'll discuss the `Array` type in the next chapter. It's the most basic of a group of classes that allow you to treat zero or more instances of an object as a single entity.)

PUT ON YOUR THINKING HAT



The `System.Char` structure exposes a large set of static and instance methods for manipulating characters. Using the Object Browser, can you write the statements and method calls that perform the following operations? (Remember, you can press F1 while a method name is selected to get more information about it.)

Create an instance of a `Char` named `MyChar` and set its value to 'x'.

Determine whether `MyChar` contains a digit.

Convert `MyChar` into a lowercase letter.

Determine whether `Char1` or `Char2` comes first in the alphabet.



HOW'D YOU DO?

In VB, character literals are delimited by single quote characters. Be careful not to use the double quote ("), because that makes your character a **String**.

Create an instance of Char called MyChar and set its value to 'x'.

MyChar As Character = 'x'

Determine whether MyChar contains a digit.

Char. IsDigit(MyChar)

Because **IsDigit()** and **ToLower()** are static methods, you call them on the class itself, not the instance.

Convert MyChar into a lowercase letter.

Char. ToLower(MyChar)

Determine whether Char1 or Char2 comes first in the alphabet.

Char1. CompareTo(Char2)

CompareTo() is an instance method, so you use normal method syntax rather than calling the method on the class.

You could also have written this as Char2. CompareTo(Char1).

MAKE A NOTE



A Char represents a single logical character, but you should be aware that a single logical character might map to more than one Unicode character. Accented characters, for example, require two characters. This is primarily an issue when you need to compare characters from different languages. If this applies to you, you'll find lots of useful examples on MSDN. Search for "Unicode".

STRINGS



In the computer world, a sequence of characters is called a **String**. (Think "string of beads" not "piece of string", and that might make more sense to you.) In the .NET Framework, strings are represented by the **System.String** structure. Like any value type, you can instantiate a **String** directly, or by calling one of its constructors.

MyString As String = "Hello"

Notice the single quotes; this one
is a **Char**.

MyString As String = New String('a',10)

The **String** structure has 8 different constructors that allow you to instantiate it in different ways. This one creates a **String** that contains the character argument the specified number of times, so in the case, **MyString** will contain "aaaaaaaaaa".

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.

NON-TEXT TEXT

Sometimes you need a **String** to contain characters that either can't be entered directly, like a tab or a new line, or that have special meaning, like the double-quote character (""). In order to do so, you have to **ESCAPE** the characters, that is, tell the compiler to treat them as characters, not VB symbols.

In order to include a quote mark in a string, just use two:

MyString As String = "John said, ""Hello, World!"""

For non-printing characters, you can use a set of Visual Basic constants:

MyString As String = "John said" + vbCrLf + "Hello, World!"

This string will
print on two lines.

VB defines the following constants:

vbCrLf carriage return & line feed

vbNewLine new line

vbCr carriage return

vbTab tab

vbLf line feed

vbBack Backspace

COMPARING STRINGS

The value of a `String` is the text it contains, and in most cases comparisons work exactly as you'd expect them to:

```
String1 As String = "Hello"  
String2 As String = "Hello"  
TheSame As Boolean = String1 = String2
```

The variable
TheSame here will
be True.

But what if `String1` contains “Hello” and `String2` contains “hello”? Are they equal? By default, the answer is no, because the `String` structure performs an **ORDINAL STRING OPERATION**, which means, in effect, that it compares the numeric values of each `Char` in the string. You may have noticed that Windows Explorer sorts “File 20” before “File 5”. That’s because it does an ordinal comparison, and “2” is less than “5”.

But the `String` structure exposes overloads of most of its methods that accept a member of the `StringComparison` enumeration as a parameter. The `StringComparison` enumeration allows you to specify precisely what you want to have happen, and best practice dictates that you should always use one of these overloads, just to make your intentions perfectly clear. The `StringComparison` enumeration contains the following members:

<code>CurrentCulture</code>	Performs case-sensitive operations based on the culture set in Windows
<code>CurrentCultureIgnoreCase</code>	Performs case-insensitive operations based on the culture set in Windows
<code>InvariantCulture</code>	Performs case-sensitive operations using an invariant culture
<code>InvariantCultureIgnoreCase</code>	Performs case-insensitive operations using an invariant culture
<code>Ordinal</code>	Performs case-sensitive ordinal operations
<code>OrdinalIgnoreCase</code>	Performs case-insensitive ordinal operations

```
TheSame As Boolean = String1.Equals(String2, StringComparison.InvariantCultureIgnoreCase)
```

STRING OPERATIONS

Like the `Char` structure, the `String` structure exposes methods to help you manipulate them. Most of them work the way you'd expect them to. For example, what would you expect the value of `IsIn` in the following snippet to be?

```
MyString As String = "Hello"  
IsIn As Boolean = MyString.Contains("l")
```

No prizes for saying `IsIn` is True.

But one very common operation deserves special consideration. The `Concat()` methods, all of which are static, simply slap one argument on the end of the other. ("Concat" is short for "concatenate". Programmers never use a simple word like "combine" if they can avoid it.) There are several overloads of `Concat()` that allow you to combine objects of various types, but most often you'll be combining one `String` with another, and you can use the `+` operator as a shorthand for that operation. For example, the following statement sets `MyString` to "Hello, World!":

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



PUT ON YOUR THINKING HAT

Using the Object Browser or MSDN (or both) can you figure out what method calls you would use to perform the following string operations?

Determine whether `String1` contains the letter "a".

Determine whether `String1` begins with the word "Hello".

Make `String1` 12 characters long by inserting spaces at the beginning.

Insert " the " at the sixth position in `String1`.



HOW'D YOU DO?

Determine whether `String1` contains the letter “a”.

`String1.Contains("a")`

Determine whether `String1` begins with the word “Hello” .

`String1.StartsWith("Hello")`

Make `String1` 12 characters long by inserting spaces at the beginning.

`String1.PadLeft(12, ' ')`

• You could
also have used

`PadLeft(12)`.

Insert “ the ” at the sixth position in `String1`.

`String1.Insert(5, " the ")`

• Remember, computers usually count
from 0.

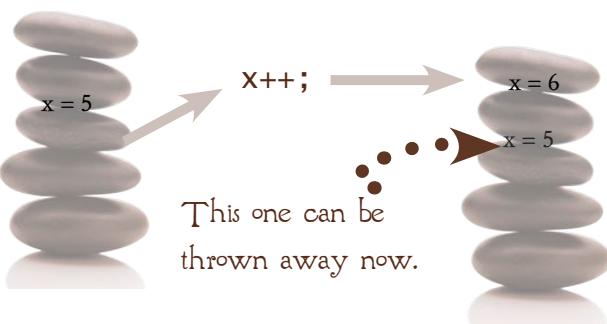


MAKE A NOTE

We'll come back to some other `String` manipulation operations, like replacing characters and finding the character at a specific position, once we've examined the `Array` type in the next chapter.

IMMUTABILITY

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. The Framework behaves this way because most value types are very small, and this behavior is faster. It all happens automatically, and with one exception, you can forget I ever mentioned it. The exception, of course, is Strings, which can sometimes be very, very big indeed. If you're doing a lot of manipulation on a single string, constantly allocating new objects can be expensive and result in a noticeable lag in your application's performance. The `StringBuilder` class is a reference type, and like all reference types, it is **MUTABLE**. (Its contents can be changed without creating a new instance.) When you're performing multiple append, insert, remove or replace operations, it's faster to use the `StringBuilder` class to perform the manipulation and then generate a `String` (once) using `StringBuilder.ToString()` when you've finished.





ON YOUR OWN

Replace the button event handler code in our chapter sample with each of the following code snippets and compare the execution time on your machine. Big difference, isn't there?

USING STRING:

```
String1 As String = "s"  
For x As Integer = 0 To 50000  
    String1 += "s"  
Next x  
MessageBox.Show("done")
```

USING STRINGBUILDER:

```
sb As StringBuilder = New StringBuilder()  
For x As Integer = 0 to 50000 /  
    sb.Append("s")  
Next x  
sb.ToString()  
MessageBox.Show("done")
```



REVIEW

What kind of object would you be working with to use the statement?

```
MyVariable = 'x'
```

What method call would you use to convert a Char to lowercase?

If you needed to perform several operations on a String before displaying it to the user, what class would provide better performance than manipulating the String itself?

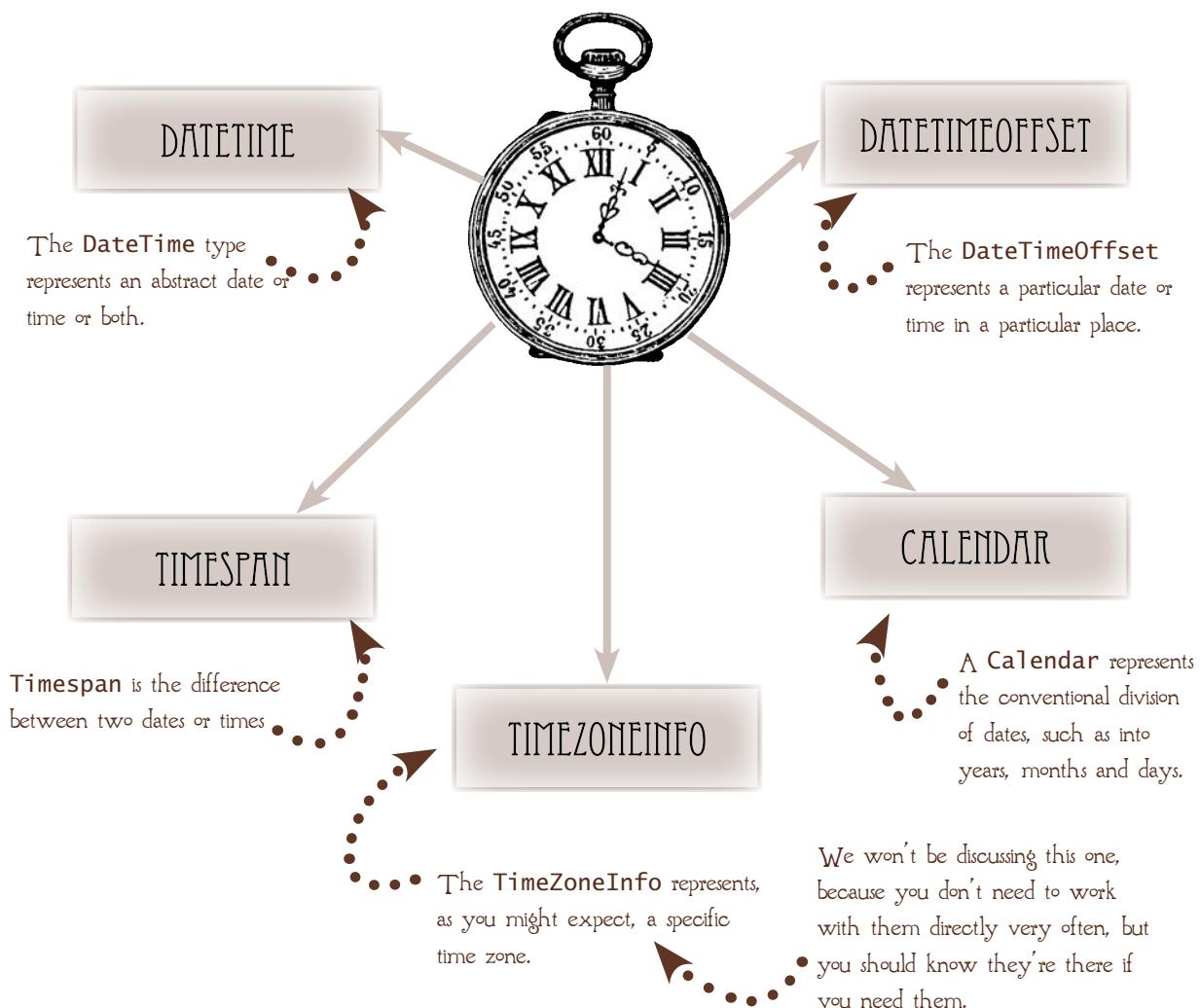
What method call would you use to find out whether a String ends with the letter 'x'?

There are (at least) two ways to combine "Hello, " and "World". What are they?



CHRONOLOGICAL DATA

What time is it? Unless you live in roughly the same part of the world as I do, it's not the same time for you as it is for me. And do you recognize daylight savings time where you live? How many hours between 3 am and 4 pm? The .NET Framework provides a set of classes to help you manipulate dates, times and time periods. (But I warn you, even with the excellent support the .NET Framework provides, handling dates and times is always tricky.)



WHAT DO YOU CARE ABOUT?

There are two structures that can be used to represent dates and times in the .NET Framework: `DateTime` and `DateTimeOffset`. The primary difference between them is in their awareness of universal time.

While the `DateTime` structure allows you to specify that the value is in either UTC or local time (as defined by the Windows machine on which the application is being run), it defaults to a value of "Unspecified", which makes the time zone anybody's guess. The `DateTimeOffset` structure, on the other hand, always records the offset of the value from UTC in hours and minutes. (By default, the offset is zero.) That's not exactly the same as a time zone, because different time zones can have the same offset from UTC, and the UTC offset of any location can change because of daylight savings time. But UTC offset does unambiguously identify a date and time, irrespective of location.

Although Microsoft recommends that `DateTimeOffset` be considered the default time structure for new applications, it isn't quite that simple. Some times are abstract. St. Swithin's Day is the 15th of July, end of discussion, not the 15th of July in Great Britain and the 16th of July in Tonga, despite the fact that Tonga is a full day ahead of Greenwich. Most servers and services outside the .NET Framework (SQL Server is a prime example) only understand the simpler `DateTime` structure. This is also true if you're working with a legacy application, since the `DateTimeOffset` structure is a recent addition to the .NET Framework. Finally, you may simply not care. Many applications only run at a single location (or at least in a single time zone); time zones simply aren't an issue.

In these situations, using a `DateTimeOffset` only complicates programming (mostly because you'll need to convert the `DateTimeOffset` into a `DateTime` to perform many basic operations), and programming is sufficiently complicated without anyone's help. So, before you assume that `DateTimeOffset` is newer, and therefore better, consider what you actually need.

WORDS FOR THE WISE



UTC stands for "Universal Time, Coordinated", a compromise between "Coordinated Universal Time" in English and "Temp Universel Coordonné" in French. It is the international civil standard for measuring time.

If you were born before 1972, when UTC was adopted, you may be more familiar with Greenwich Mean Time (GMT). If you're a pilot, sailor, or in the military, you may know Zulu time. Unless you're involved in precise astronomical observations (in which case, you undoubtedly know more about the subject than I do), you can consider UTC, GMT and Zulu as equivalent.

THE DATETIME STRUCTURE

The `DateTime` structure is the most basic of the types that deal with dates and times. The `DateTime` structure, like most time-related structures in the .NET Framework, measures a specific moment in time as the number of ticks since midnight, January 1, 0001 CE. A tick is 100-nanoseconds; there are 10 million ticks per second.

The `DateTime` structure exposes a number of constructors that allow you to specify the year, month, day, hour, minute and second in various combinations, the number of ticks (all as `Int64` structures) and optionally a `Calendar` or `DateTime.Kind`. By default, the `DateTime` uses the Gregorian calendar that is in use throughout most of the Western World, but the .NET Framework implements others, including the Hebrew, Japanese and Persian calendars. (If you need to work in an alternate, or multiple, calendars, search MSDN for `System.Globalization.Calendar`.)

The `DateTime.Kind` property is defined as a member of the `DateTimeKind` enumeration, which contains the values `Unspecified`, `Local` (as in "local time", as defined by the Windows operating system) and `Utc`. By default, the `Kind` property will be set to `Unspecified`. The `SpecifyKind()` method can be used to change the property (but not the value), and you can convert values using `ToUniversalTime()` and `ToLocalTime()`, but if you're really concerned about time zones, you should use the `DateTimeOffset` structure instead, because `DateTime`'s support is incomplete. (It doesn't, for example, allow for daylight savings times when performing operations such as addition and subtraction.)

Although the `DateTime` structure can capture both dates and times, it's common for an application to be interested in only one of those components. You can simply ignore the other part, but you must be careful about doing so consistently. If you're only interested in the date, for example, you should use `Date1.Date = Date2.Date` for comparisons, not `Date1 = Date2`, because either of the variables might contain random times.



PUT ON YOUR THINKING HAT

Using the Object Browser or MSDN for constructor details, can you write statements to create and initialize a variable called `Date1` to the following values?

January 1, 2012

1 am, March 13, 2010

3 pm, April 20, 1972 in UTC



UNDER THE MICROSCOPE...

The ability to store dates across a span of almost 10,000 years should protect us from the possibility of future "Millennium bugs", but because the earliest possible date is January

1, 0001, you'll run into problems if you're dealing with historical data, ("historical" in the real-world sense, not the computer sense of "anything before today"). You can't record the date of the Battle of Thermopylae (353 BCE) or the beginning of the Roman Republic (509 BCE) using normal techniques.

It's not an issue very often. (In my entire career, I've had to deal with the problem just once.) But you should be aware of the issues. One alternative is to include the appropriate date field (almost certainly a `DateTime`) as a property of a new class that also exposes an `Era` property and create methods that accommodate eras. (If you're interested, why don't you try doing it?)

HOW'D YOU DO?



January 1, 2012

```
Date1 As DateTime = New DateTime(2010, 1, 1)
```

1 am, March 13, 2010

```
Date1 As DateTime = New DateTime(2010, 3, 13, 1, 0, 0)
```

Times are expressed
using a 24-hour
clock

3:15 pm, April 20, 1972 in UTC

```
Date1 As DateTime = New DateTime(1972, 4, 20, 15, 15, 0, DateTimeKind.Utc);
```

...ANCIENT DAYS

Even in the current era in Europe, you can run into calendar issues when working with history. The Gregorian calendar was first proposed in 1582 but not fully adopted until it was finally adopted by Greece in 1923. Because of the change from the Julian to the Gregorian calendar, certain dates simply never occurred in the adopting countries. October 21, 1582 is a valid date in The Netherlands, for example, but not in Spain. The .NET Framework representation of dates (even in the `DateTimeOffset` structure) simply isn't sophisticated enough to deal with these issues.

Be aware, too, that programmers tend to use the word "Julian" incorrectly to refer to a format rather than a date. 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 (July 4, 1776 as opposed to 4 July, 1776).



PUT ON YOUR THINKING HAT

Assuming a variable called `Date1` initialized to January 1, 2010, can you write the statements to perform the following operations? As always, you can use the Object Browser to explore what `DateTime` knows and does, and press F1 on a specific member to get more details.

Set `Date1` to the current operating system date, ignoring the time.

Add 3 days to the value of `Date1`.

Display the name of the day of the week in a `MessageBox`. (Hint: Use the static `Enum.GetName()` method to retrieve the string representation of the `DayOfWeek` enumeration.)

In a `MessageBox`, display the number of days in the month contained in `Date1`.

Determine whether `Date1` is earlier than the current operating system date.



HOW'D YOU DO?

Set Date1 to the current operating system date, ignoring the time:

```
Date1 = DateTime.Today
```

Add 3 days to the value of Date1:

```
Date1.AddDays(3)
```

Display the name of the day of the week in a MessageBox:

```
MessageBox.Show(Enum.GetName(typeof(DayOfWeek), dt.DayOfWeek))
```

This one was tricky.
Give yourself a gold star
if you got it right.



In a MessageBox, display the number of days in the month contained in Date1:

```
MessageBox.Show(DateTime.DaysInMonth(dt.Year, dt.Month).ToString())
```

Determine whether Date1 is earlier than the current operating system date, including the time:

```
(Date1 < DateTime.Now)
```

TIMESPANS

It's one of those paperclip things: obvious once you've seen the answer, but less than intuitive: The difference between two dates is not a date.

How many hours between noon and 6 pm? Six, of course. Silly question. But until the release of the .NET Framework, it was also a thorny one, because most programming languages didn't expose a data type that could represent a period of time. Because our system of arithmetic is decimal, and our system of timekeeping isn't, you had to write a whole series of complicated methods to convert and round between 60 seconds, 60 minutes, and 24 hours. Personally, I'd rather go to the dentist.

All that tedium goes away with the .NET Framework `TimeSpan` structure. Like all the date and time structures in the .NET Framework, the `Timespan` stores information as 100-nanosecond ticks, but it knows how to convert them into milliseconds, seconds, minutes, hours and days. It can even do that tedious conversion into decimal fractional units—1 hour and 30 minutes is 1.5 hours, but 1 hour, 37 minutes and 52 seconds is, um....

PUT ON YOUR THINKING HAT

Can you figure out what statements you'd write to perform the following operations using a `Timespan` and a `DateTime`?



Create a new `TimeSpan` called `Span1`, initializing it with 1 hour, 37 minutes and 52 seconds.

Set a `Double` called `TheHours` to the fractional number of hours in `Span1`.

Create a new `TimeSpan` called `Span2` from 3 days.

Set `Span2` to the difference between two `DateTimes` called `Date1` and `Date2`.

Set `TheHours` to the number of hours in `Span2`.



HOW'D YOU DO?

Create a new `TimeSpan` called `Span1`, initializing it with 1 hour, 37 minutes and 52 seconds:

```
Span1 As Timespan = new TimeSpan(1, 37, 52)
```

Set a Double called `TheHours` to the fractional number of hours in `Span1`:

```
TheHours = Span1.TotalHours()
```

You could also have used

Create a new `TimeSpan` called `Span2` from 3 days: `New TimeSpan(3, 0, 0);`

```
Span2 As Timespan = TimeSpan.FromDays(3)
```

Set `Span2` to the difference between two `DateTime` values called `Date1` and `Date2`:

```
Span2 = Date1 - Date2
```

Set `TheHours` to the number of hours in `Span2`:

```
TheHours = Span2.Hours
```

THE DATETIMEOFFSET

The `System.DateTimeOffset` structure is essentially a `DateTime` with expanded capabilities. It replaces the `DateTime.Kind` property with an `Offset` property, a `TimeSpan` that represents the offset between the `DateTime` value of the `DateTimeOffset` and UTC (with zero representing UTC itself).

The `DateTimeOffset` structure plays nice with `DateTime`. Two versions of the `DateTimeOffset` constructor accept a `DateTime` as an argument, either by itself, in which case the `Offset` property is determined by the `DateTime.Kind`, or with an explicit `Offset`. The `DateTimeOffset` also exposes a property named, appropriately enough, `DateTime`, that returns a `DateTime` with the `Kind` property set to `Unspecified`.

The screenshot shows the Microsoft Visual Studio Object Browser for the `DateTimeOffset` structure. At the top, it lists several interface implementations: `IComparable`, `IFormattable`, `ISerializable`, `IDeserializationCallback`, `IComparable<DateTimeOffset>`, and `IEquatable<DateTimeOffset>`. Below this, the `DateTimeOffset` structure is defined as a `Struct`. The `Fields` section contains `MaxValue` and `MinValue`. The `Properties` section lists numerous properties including `Date`, `DateTime`, `Day`, `DayOfWeek`, `DayOfYear`, `Hour`, `LocalDateTime`, `Millisecond`, `Minute`, `Month`, `Now`, `Offset`, `Second`, `Ticks`, `TimeOfDay`, `UtcDateTime`, `UtcNow`, `UtcTicks`, and `Year`. The `Methods` section lists various methods such as `Add`, `AddDays`, `AddHours`, `AddMilliseconds`, `AddMinutes`, `AddMonths`, `AddSeconds`, `AddTicks`, `AddYears`, `Compare`, `CompareTo`, `DateTimeOffset (+ 5 overloads)`, `Equals (+ 2 overloads)`, `EqualsExact`, `FromFileTime`, `Parse (+ 2 overloads)`, `ParseExact (+ 2 overloads)`, `Subtract (+ 1 overload)`, `ToFileTime`, `ToLocalTime`, `ToOffset`, `ToString (+ 3 overloads)`, `ToUniversalTime`, `TryParse (+ 1 overload)`, and `TryParseExact (+ 1 overload)`.



PUT ON YOUR THINKING HAT

The `DateTime` returned by the `DateTime` property of a `DateTimeOffset` has a `Kind` property of `Unspecified`. Given a `DateTimeOffset` with an unknown `Offset` value, how would you create a `DateTime` with the UTC value and the appropriate `Kind` property?



HOW'D YOU DO?

It was a trick question. The `DateTime` property returns a `DateTime` with the `Kind` property set to `Unspecified`, but the `DateTimeOffset` exposes another property, `UtcDateTime`, that does exactly what you need, so you only need a single line of code.

```
Date1 As DateTime = DateTimeOffset1.UtcDateTime
```

Don't feel bad if I tricked you. Show me a programmer who hasn't spent hours (or days) writing a method to do something that could have been done with a single method call, if only they'd known that the method existed, and I'll show you a programmer who hasn't been at this more than a week or two.

That's why it's so important to learn your way around the Object Browser and the .NET Framework library documentation. Before you write a method that performs some kind of general manipulation of a .NET Framework class, you always need to go exploring to see what's already available to you. The library is so big, and it changes so quickly, you can never know it all...



TAKE A BREAK

We've almost finished this chapter, so why don't you take a break before you complete the section and chapter Reviews?



REVIEW

Even with all the support provided by the .NET Framework, dealing with dates and times is complex, or at least it can be. Before we move on, let's go over the basic principles:

What are the three primary structures used for representing dates and times in the .NET Framework?

What's the primary difference between a `DateTime` and `DateTimeOffset`?

Date and time information is stored in what data type?

What type would you use to store an abstract date, like Halloween?

What type would you use to represent a period of time, regardless of the date?

What is the civil (as opposed to military or aviation) standard used for representing time?

What determines "local time" on any given system?



REVIEW

What two steps are required to use a namespace in a code file?

Write the statement that allows you to reference the types in the `System.Windows` namespace:

What .NET Framework type should you use to represent each of the following types of data, unless you have a good reason to use something else?

Integer

Floating Point

Currency

Under what conditions should you use the `StringBuilder` class? Why?

List three reasons you might choose a `DateTime` over a `DateTimeOffset`:

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

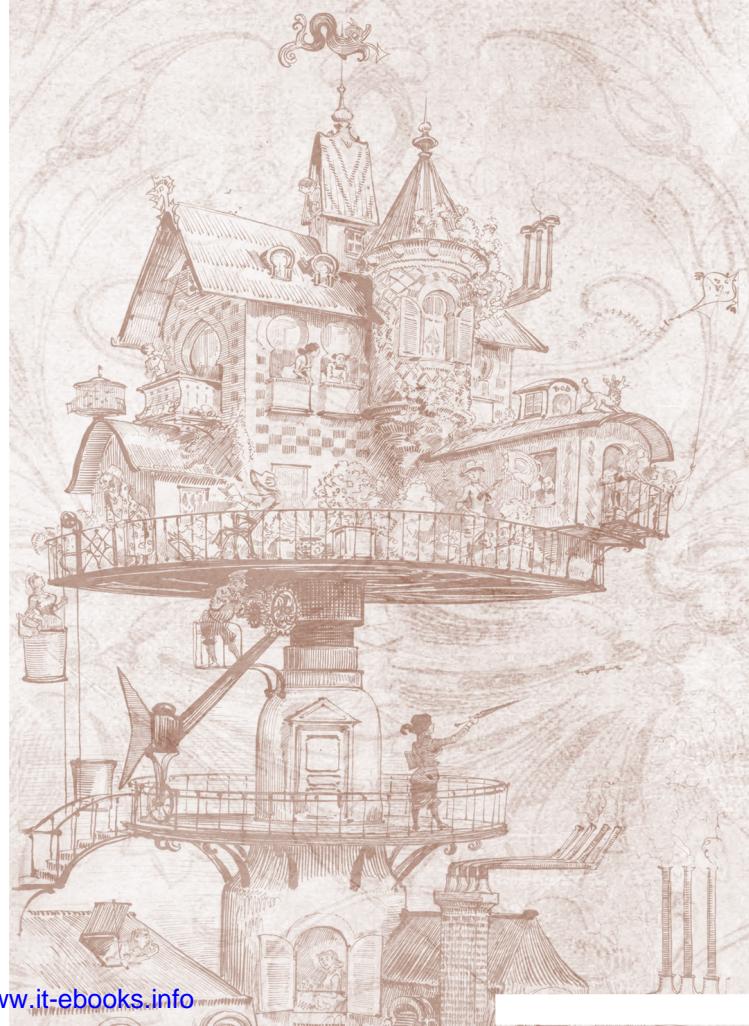
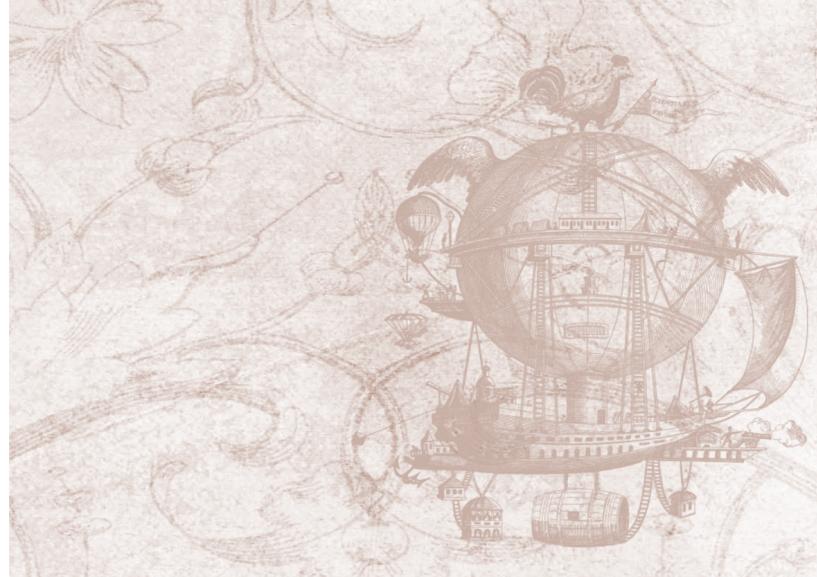
1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



THE CLASS LIBRARY, PART 2

II

The .NET Framework types we discussed in the last chapter manage single items of data, even though the structure of that data can be complex. In this chapter, we'll concentrate on the types that represent more than one element and allow those elements to be manipulated either as a group or as single entity.

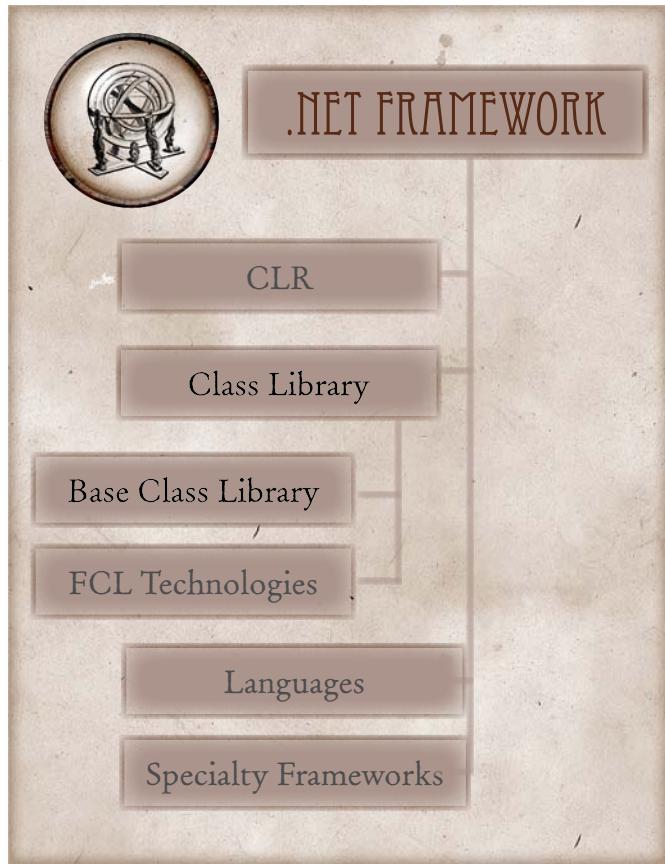
The types we've examined so far are like a single book. In this chapter, we'll look at the .NET Framework types that let you manipulate an entire bookcase.





FITTING IT IN

In this chapter we'll finish our exploration of the .NET Framework class library.



.NET FRAMEWORK

CLASS LIBRARY

We discussed
these types
in the last
chapter.



MAKE A NOTE

There are other aggregate types in the FCL that are useful in special situations, like concurrency. The principles of their use are the same as the more general aggregates we'll be discussing here, so you can learn about them when (and if) you need them.

AN EXAMPLE...

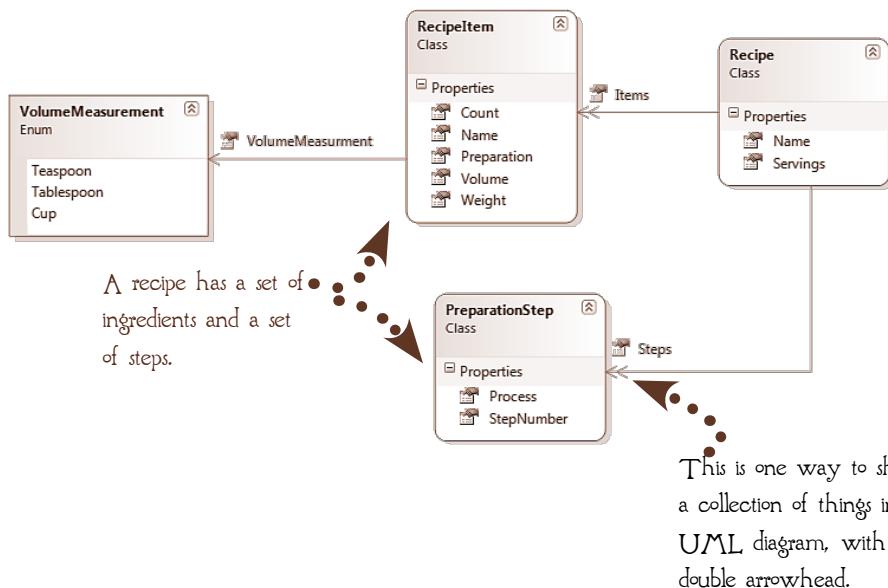
In the last couple of chapters we've looked at some simple hierarchies for handling ingredients and components for Neil & Gordon's cafe. But you may have noticed something important has been missing: the recipes themselves. Whether or not you've done much cooking, you've probably seen a recipe at some point. Most of them consist of a list of ingredients, and a list of steps to make the recipe. The number of steps and the number of ingredients varies from recipe to recipe—a soup might have dozens of ingredients but only one step, while a classic baguette has four ingredients (flour, water, salt and yeast) but a lengthy preparation method.

Since we can't know in advance how many ingredients or steps are required in our recipe object, we need some way to treat those "sets" as a group—zero or more instances of another object. In this chapter, we'll learn how to do that, using the objects that the .NET Framework provides for storing and manipulating sets of objects as a unit.

Buttermilk Bread

1/2 cup warm water 1 tab active dry yeast 1 tsp sugar 1-1/2 cups buttermilk	2 tabs unsalted butter, melted 3 tabs honey 1 tab salt 6 cups unbleached AP flour
--	--

Proof yeast with sugar and water.
 In a small bowl, combine buttermilk, butter, honey, and yeast mixture.
 Place flour in bowl of stand mixer fitted with paddle attachment. Mix in buttermilk mixture on slow speed until a shaggy dough is formed.
 Using dough attachment, knead on medium speed for about 5 minutes, until smooth and satiny.
 Place dough in a greased bowl, lightly spray top with oil, and let rise, covered, until doubled, about 1 hour.
 Shape dough and place in greased loaf pans. Spray lightly, cover, and let rise until it just reaches the top of the pans.
 Brush with egg glaze.
 Bake at 375°F for about 45 minutes.





TASK LIST

In this chapter, we'll start by looking at the most basic type of set, the **Array**, in order to get a sense of how this whole "set" thing works, and then move on to some of the specialized sets provided by the .NET Framework, and the **GENERICs** that make it possible to create strongly typed sets without reimplementing the world.

ARRAYS



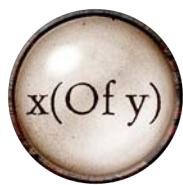
We'll begin by looking at the simplest kind of set, the **Array**, and the way it's directly supported by the VB compiler, and a special kind of procedure called a **Predicate** that allows you to specify criteria at runtime.

SPECIALIZED SETS



After we've explored how sets work in general, we'll move on to some of the specialized set types, including those that keep their members ordered in a particular way or allow their members to be accessed by a key. In practice, you won't use these set types very often, so we'll only take a quick look at them.

GENERICs



After we've seen how the various types of sets work, we'll take a look at the concept of **GENERICs**, a special way of specifying type parameters that allow you to defer specifying an object type until the class is instantiated.

WORDS FOR THE WISE



Computer science defines several types of sets that are structured in specific ways. The most common term for referring to these objects that contain zero or more instances of another object is "collection", but I'm using the word "set" because the .NET Framework has a specific kind of set called a **Collection**. But when you run across the word "collection" elsewhere (including in MSDN), bear in mind that the writers aren't necessarily referring to a **Collection** type, but using the word as a general term to refer to any kind of set.



ARRAYS

The most basic way of representing a set of objects is an **ARRAY**, which is simply a set of objects of the same type that can be accessed individually or as a group. **Array** is a .NET Framework class, with methods and properties that you can use, but it's defined as abstract, which means you can't instantiate it directly. (We'll look at abstract classes in detail in the next chapter.) Instead, you'll use some special syntactic sugar provided by the VB compiler. We'll look at that in a minute. But first, let's look at the nature of an array.

Arrays are defined by their dimensions. The .NET Framework supports three types of arrays. **SINGLE DIMENSIONAL** arrays, the kind you'll probably use more often, contain a simple set. **MULTIDIMENSIONAL ARRAYS** contain more than one set of members, like a grid or a cube. Finally **JAGGED ARRAYS** are arrays of arrays. Jagged arrays are an efficient way of storing values that have a lot of missing elements.

This single-dimensional array
represents a simple menu.

0	Lobster Bisque
1	Roast Lamb with Asparagus and Sautéed Potatoes
2	Mixed Green Salad
3	Pear Bombe with Tuiles

This is a multidimensional array that allows you to store two dishes for each course. Multidimensional arrays can be of any number of dimensions; this one just happens to have two.

0	Lobster Bisque	Vegetable Plate
1	Roast Lamb	Polenta with Roasted Squash
2	Mixed Green Salad	Mixed Green Salad
3	Pear Bombe	Pear Bombe

This example of a jagged array eliminates the duplicate values in the one dimensional sample. With a jagged array, the array members may have any number of members.

0	Lobster Bisque	Vegetable Plate	
1	Roast Lamb	Polenta	
2	Mixed Greens	Chicken Elizabeth	
3	Pear Bombe		

Every item in the array has an INDEX, starting with zero. The index isn't part of the array, really, just a way to reference the items it contains.

CREATING ARRAYS

In VB, parentheses are used to declare an Array and, as we'll see, to reference their elements. You specify the type of elements the Array contains when you declare it. All of the elements must be the same type, but you can use the base class for all elements in the .NET Framework, Object, to create an Array that can contain mixed types.

<name>(<size>) As <type>

FirstArray(5) As Integer

You can specify the size of the array when you declare it or leave the parentheses empty and declare the size later.

This syntax tells the compiler to create a type derived from the abstract `Array` class with five elements. It doesn't actually create an instance of `Array`.

<name>()() As <type>

FirstArray()() As Integer

Jagged arrays are declared using multiple sets of parentheses, one for each dimension.

Commas are used to declare multidimensional arrays, one comma for each extra dimension. (This example is two-dimensional.)

Like any reference type, you can instantiate the array using the `New` keyword and provide the number of elements as a parameter if the size hasn't already been declared.

MyArray() As Integer = New Integer(5)



MAKE A NOTE

Arrays are immutable, like `Strings`. It is possible to change the size of single-dimension arrays using the static `ReDim()` method, but it's a very expensive operation. (There's no `Array` equivalent to the `StringBuilder` class. If you expect the size of your set to change, you'll probably want to use one of the other .NET Framework set types that we'll discuss in this chapter.)

INITIALIZING ARRAYS

You can initialize the elements of an Array when it's declared by enclosing the element values in braces. Notice that you add the parentheses to the type, not the name:

```
Weekend As New String() = {"Saturday", "Sunday"}
```

The type and length specifications aren't necessary when you initialize an array using this syntax, but it isn't wrong to use them. The following statements would also be correct, but beware that the compiler will infer the type from the values, and that (as we've seen) can sometimes go wrong:

```
Dim Weekend() As String = {"Saturday", "Sunday"}  
Dim Weekend() = {"Saturday", "Sunday"}
```

For multidimensional arrays, you can nest the braces:

```
Dim MyArray = {{'a', 'b'}, {'c', 'd'}, {'e', 'f'}};
```



Notice the commas that separate each array
and the elements within the arrays.

When you're working with jagged arrays, you must use the New keyword to directly initialize the child arrays:

```
Dim Menu = { _  
    New String() = {"a", "b"}, _  
    New String() = {"one", "two", "three"}, _  
    New String() = {"x", "y", "z", "omega"} _  
}
```



You still use commas to separate the
array elements.

REFERENCING ARRAY ELEMENTS

Parentheses are used when initializing an array in the declaration statement and to reference elements within the array. You simply use an integer value to specify which element you want:



The syntax is similar to the syntax you used to declare the **Array**, but this time it refers to a specific element, not the length of the array.

```
MyArray(3) = "Hello"  
MyArray(1, 4) = "World"  
MyArray(1)(5) = "Lobster Bisque"
```

The tricky bit—at least until you learn to count from zero—is that the first element of an array is element zero, not element one, and the last element is one less than the length of the array:

```
Dim Weekend(2) As String  
Weekend(0) = "Saturday"  
Weekend(1) = "Sunday"
```



PUT ON YOUR THINKING HAT

Write the statement that declares a single-dimension Array of Strings called **Week** that is initialized with the days of the week:

Write the statement that sets the fourth element of **Week** to “Hello”:

Write the statement that declares and initializes a two-dimensional array called **Menu** based on the example on page 356:



HOW'D YOU DO?

Write the statement that declares a single dimensional Array of Strings called Week that is initialized with the days of the week:

```
Dim DaysOfWeek As String() = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
```

Write the statement that sets the fourth element of Week to "Hello":

Week(3) = "Hello"



Weird, isn't it? But the fourth element is referenced as 3, because the numeric value you provide is the offset from the first element: the fourth element is three past the first; the first element is zero past the first. (Read that last bit again.)

Write the statement that declares and initializes a two-dimensional Array called Menu based on the example on page 356:

```
Dim Menu As String(,) = {{"Lobster Bisque", "Vegetable Pate"}, _  
    {"Roast Lamb", "Polenta with Roasted Vegetables"}, _  
    {"Mixed Green Salad", "Mixed Green Salad"}, _  
    {"Pear Bombe", "Pear Bombe"} _  
}
```

THE FOR EACH...NEXT COMMAND

Back in Chapter 7 we looked at iteration commands that allow you to execute a statement block multiple times. At that time we skipped one, the For Each command, that executes a statement block once for each element in an array or other set type.

```
For Each <counter> [As <type>] In <set>
  ...
  Next [<counter>]
```

The For Each...Next statement is roughly equivalent to a For...Next statement but much simpler, because you don't have to worry about how many elements are in the set or index into it:

```
For Each currentDay As String In DaysOfWeek
  MessageBox.Show(currentDay);
Next currentDay
```

Both of these snippets display each element of the Array in a MessageBox, but the syntax of foreach is much simpler.

```
For x As Integer = 0 To DaysOfWeek.Length/
  MessageBox.Show(DaysOfWeek[x]);
Next x
```



PUT ON YOUR THINKING HAT

How would you write a For Each statement that did something for every element in an Array of Int32s called ValidDates?

THE ARRAY CLASS

The `System.Array` class is declared as abstract. As we'll see in the next chapter, an abstract class is one that is only intended to be used as a base class, not instantiated directly. (Most abstract classes don't implement all of the members they define.) But the `Array` class doesn't even allow you to create classes that explicitly derive from it, courtesy of some behind-the-scenes .NET Framework magic.

In most circumstances, you'll rely on the syntactic sugar the VB compiler provides for creating instances of an `Array`, but the class also exposes static `CreateInstance()` methods that return an instance of an `Array` of a specified type. About the only time you'll need to use `CreateInstance()` is when you don't know the type of the array until runtime, and that doesn't happen very often.

`CreateInstance()` aside, the `Array` class exposes several properties and methods that you'll find yourself using a lot. Many of them are straightforward, such as the `Length` property that returns the number of elements in the array, while some require the use of a PREDICATE that

specifies a criterion that an element must meet, so let's take a look at how to build and specify a predicate.

You probably remember from English class that the predicate of a sentence is the part that modifies the subject. In the sentence "Neil is a chef", "is a chef" is the predicate that modifies the subject "Neil". Predicates in computer programming mean something a little different. The term "predicate" in this context derives from formal logic and means (roughly) "something that is true or false about the subject".

In the .NET Framework, predicates are implemented as methods that return a `Boolean` value indicating whether or not the supplied argument satisfies a particular condition. They're used by methods in the `Array` class that search the elements of the array. The description of predicates actually makes them sound more complicated than they are, so let's look at an example, using the static `Find()` method that accepts the one-dimensional `Array` to search and the predicate that defines what to search for, and returns the first matching element:

```
Shared Function IsWeekend(String theDay) As Boolean
    If theDay = "Saturday" Or theDay = "Sunday" Then
        Return True
    Else
        Return False
    End Function

    Dim MyDays = {"Monday", "Saturday", "Friday", "Wednesday", "Sunday"}
    String FirstWeekend = Array.Find(MyDays, AddressOf IsWeekend)
```

The predicate is just a regular method that returns a Boolean. This one works with arrays of Strings.

When you pass the predicate to the `Find()` function, you use only the name, not the signature.



HOW'D YOU DO?

How would you write a foreach statement that did something for every element in an Array of Int32s called `ValidDates`?

For Each `x` As Integer in `ValidDates`

...

Next `x`



It doesn't matter what you called the control variable here, as long as it's a valid identifier and not a keyword.



PUT ON YOUR THINKING HAT

Using the Object Browser and MSDN, can you write the following statements using the members of the `Array` class?

Set an Integer called `NumElements` to the number of elements in the array `MyDays`:

Sort the Array called `Names` using the default comparison method for Strings:

Find the last instance of "Hello" in an array of Strings called `Words` (you'll need to define the predicate as well):



HOW'D YOU DO?

Set an Int32 called NumElements to the number of elements in the array MyDays:

```
NumElements = MyDays.Length
```

Sort the Array called Names using the default comparison method for Strings:

```
Array.Sort(Names)
```

Find the last instance of "Hello" in an Array of Strings called Words, (you'll need to define the predicate as well):

```
Function IsHello(String test)
    If test = "Hello" Then
        Return True
    Else
        Return False
End Function
```

```
Array.FindLast(Words, IsHello)
```



TAKE A BREAK

Now that we've gotten a taste of how to handle sets of objects using the Array class, why don't you take a break before we move on to some of the specialized set types provided by the .NET Framework?



SPECIALIZED SETS

The `Array` class is supported by the VB compiler, its elements are simple to access, and it provides some basic methods for manipulating its contents. But the `Array` is a value type, and as such is immutable, which can make operations such as sorting elements or resizing the array an expensive operation. The solution, of course, is to use a reference type when you need to perform operations like these, and the .NET Framework provides several.

Most of the .NET Framework set types come in two versions, one that manipulates elements as `Object` types, and one that allows elements to be strongly typed through the use of generic parameters that are specified when the class is instantiated. We'll look at the types that use `Object` elements in this section and examine the strongly typed `GENERIC` versions in the next.

THE ARRAYLIST CLASS

You can think of the `ArrayList` class as an `Array` that contains reference types, but it does have slightly different capabilities. The `ArrayList` is always one-dimensional (as, in practice, most `Arrays` are), and its elements are always instances of the `Object` type, so you must cast them, and value type elements will be boxed.

On the other hand, the size of an `ArrayList` isn't fixed, and you can insert elements at any position without re-creating the set. Because the elements are `Objects`, you can search the set without using a predicate.



MAKE A NOTE

The non-generic version of the specialized sets requires a reference to the `System.Collections` namespace. This won't be added to your source files by default, so if you want to play with these types, you'll need to add the statement to the top of your source file:

```
using System.Collections;
```

THE SET INTERFACES

Here's a partial class diagram of the `ArrayList`. As you can see, the `ArrayList` implements several interfaces but descends directly from `System.Object`. These interfaces—`ICollection`, `IList` and `IEnumerable`—define the basic functionality of set types in the .NET Framework. They're implemented by most of the set types, including the basic `Array` we looked at in the last section.

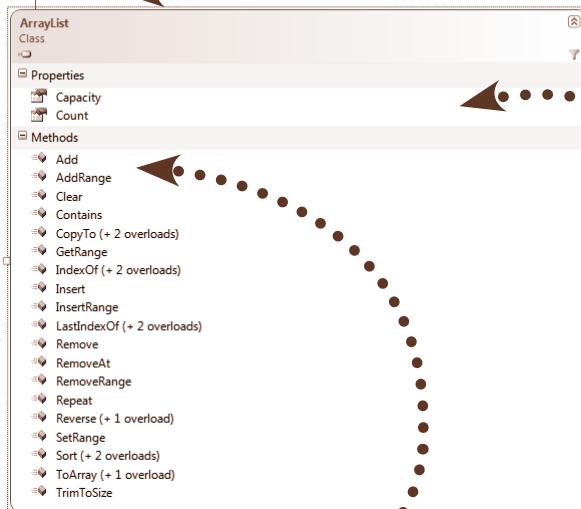
Internally, the `ArrayList` stores its data as a basic `Array`, but the behavior it exposes makes it dynamic: You can insert and delete items at any position, and the `ArrayList` will take care of resizing and reordering the data for you. But be aware that this capability doesn't come for free. The `ArrayList` does the memory allocations for you, but they're still being done, and they still take time.

The `ICollection` interface extends `IEnumerable`, adding, among other things, the `Count` property and the `CopyTo()` method that creates a standard `Array` from the `ArrayList`.

`IList` extends `ICollection` and allows elements to be accessed by their index.

The `IEnumerable` interface allows the `For Each` statement to be used with an `ArrayList`.

`ICloneable` allows copies of the object to be created.



When you `Add()` a new item to an `ArrayList`, it will expand the `Capacity` for you as necessary.

The `Capacity` of an `ArrayList` is the number of items it *can* currently hold. The `Count` is the number of items it *does* currently hold.



PUT ON YOUR THINKING HAT

Using the class diagram, the Object Browser and MSDN, can you write statements to perform the following functions?

The default constructor creates an `ArrayList` with a capacity of 4. If you know in advance that you'll have 10 items, what constructor will be more efficient? Write the statement to declare and initialize an `ArrayList` called `MyAL`:

Set the third item in `MyAL` to "Hello":

Insert "World" as a new item at the end of the `MyAL`:

Remove the fourth item in `MyAL`:

Determine whether `MyAL` contains "Hello":

Insert "Hello, World" at the seventh position in `MyAL`:



HOW'D YOU DO?

The default constructor creates an `ArrayList` with a capacity of 4. If you know in advance that you'll have ten items, what constructor will be more efficient? Write the statement to declare and initialize an `ArrayList` called `MyAL`:

```
ArrayList MyAL = New ArrayList(10)
```

Set the third item in `MyAL` to "Hello":

```
MyAL(2) = "Hello"
```

Give yourself a gold star if you remembered that the first item is zero, not one!

Insert "World" as a new item at the end of the `MyAL`:

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

You could also have used something like

```
MyAL(MyAL.Count - 1) = "World"
```

but that would be far more complicated...

Remove the fourth item in `MyAL`:

```
MyAL.RemoveAt(3)
```

If you used the `Remove()` method, go back and check the difference between `Remove()` and `RemoveAt()`. But don't feel bad; it's a common mistake.

Determine whether `MyAL` contains "Hello":

```
MyAL.Contains("Hello")
```

A lot easier than the `Array` version,
isn't it?

Insert "Hello, World" at the seventh position in `MyAL`:

```
MyAL.Insert(6, "Hello, World")
```

LIFO, FIFO, LINKS & KEYS

Both the `Array` and the `ArrayList` contain elements that can be accessed in any order. You can sort them, of course, or add the elements in a particular order, but by default the elements are stored randomly. The .NET Framework also supports classes that allow their elements to be accessed in particular ways:

The diagram illustrates various .NET collections using stones and keys.

Stack (LIFO): A stack of stones representing Last In, First Out (LIFO) order. Annotations explain `Push()` and `Pop()` operations, and note that only the top element can be accessed.

Dictionary / Hashtable (Key/Value Pairs): A table with keys (keys) and values (stacks of stones). Annotations explain that keys must be unique and hash codes don't have to be.

LinkedList (Links): A linked list of stones where each node knows about its neighbors. Annotations show `Node.Previous()` and `Node.Next()` methods.

Queue (FIFO): A queue of stones representing First In, First Out (FIFO) order. Annotations explain `Enqueue()` and `Dequeue()` operations.



PUT ON YOUR THINKING HAT

Looking back at the descriptions of the different kinds of set types, can you match each of these real-world examples to its programming equivalent?



A line of people expect to be served in the order they arrived.

It's like a:

A rolodex has one card for every phone number. It's like a:



A dictionary has a single entry for a word, but a word can have multiple meanings. It's like a:



The last plate added to the top of a stack of dishes is the first one used. It's like a:



The items on a check list can be performed in any order. It's like a:



ON YOUR OWN

Look at that list of applications you'd like to write. What kind of set types do you think you'll need when you implement them? Make a note of what data you'll need to store, using what set type, and why you think that type is the best.



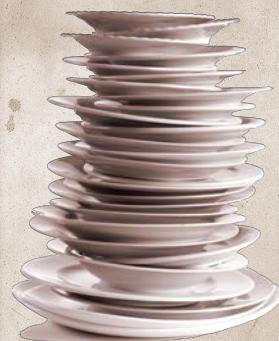
HOW'D YOU DO?

A line of people is like a Queue



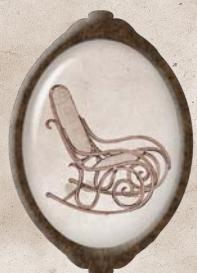
A Rolodex is like a Dictionary,
while a dictionary is like a
Hashtable.

Don't worry too much if you got
these two the other way around.
The difference is a subtle one, and
in practice not often critical.



A stack of dishes is
like a Stack.

A check list is
like an Array or
ArrayList.



TAKE A BREAK

There's just one more type of set to study, so why don't you take a break before you complete the Review and we move on to generics?



REVIEW

LIFO stands for:

It's represented by a:

FIFO stands for:

It's represented by:

The main difference between a **Hashtable** and a **Dictionary** is that the key of a **Dictionary** must be:

It's generally more efficient to define the **Capacity** of an **ArrayList** when you instantiate it because:



GENERICs

We've seen that the basic `Array` type is strongly typed and supported by the language, while the specialized sets such as `ArrayList` and `Stack` are more flexible, but treat all their elements as `System.Object`, which requires casting and possibly boxing. The final group of set types in the .NET Framework provide the best of all possible worlds: They provide the flexibility of the reference-type sets but allow you to specify the type at runtime.

PROS AND CONS

TOO HARD
Arrays are strongly typed, but immutable, and their behavior is not very rich.

Specialized Sets provide a rich set of behaviors, but they're weakly typed.

TOO SOFT

Generic sets provide strong typing, efficient performance, and rich behavior.

JUST RIGHT!

INSTANTIATING A GENERIC

You know how to use standard parameters that allow you to specify a value at runtime, like the `String` to be displayed in the `MessageBox` or the `Integer Capacity` of a new `ArrayList`. GENERICS are also a kind of parameter, but instead of specifying a value, they specify a type.

The `ArrayList` contains instances of `System.Object`, and it always casts whatever you pass to it to that type. We've seen that that's time-consuming and error-prone because there's nothing to prevent you from adding objects of different types to the set, and that will come back to bite you. `List(of T)` is the generic equivalent of the `ArrayList`. It can do almost everything `ArrayList` can do, and some more besides, but because you specify the type at runtime, it doesn't have to cast everything to `System.Object`.

Notice the `(of T)`. That's the syntax for a generic parameter. The parentheses are required to specify a generic. The "T" is a convention. (It stands for "Type".) At runtime, you treat a generic parameter like any parameter—replace it with an appropriate value. The only difference is that "appropriate value" in this case means the name of a type:

You specify the type inside
parentheses. •••••

Notice that you need the parentheses
to call the constructor. You can pass
standard parameters just like you would
to a non-generic constructor. •••••

MyList As List(of String) = New List(Of String)(5)

You need to specify the type ••••• again when you instantiate the generic class.



PUT ON YOUR THINKING HAT

Below is a list of the generic set types you're most likely to use. They're all defined in the `System.Collections.Generic` namespace. Can you connect each generic set type to its description? (You can, always, use the Object Browser and MSDN to help.)

Dictionary(of T)

The elements in this set type know about the ones that precede and follow them.

HashSet (of T)

This type allows random access like an array but stores elements in order.

LinkedList(of T)

This type allows elements to be accessed in LIFO order.

List (of T)

This set type is roughly equivalent to the non-generic `ArrayList`.

Queue (of T)

The elements in this set type are accessed in FIFO order.

SortedDictionary (of T)

This type keeps key/value pairs in order.

SortedList (of T)

This type is like a `Dictionary`, but the `Add()` method takes just the value and calls `GetHashCode()` to generate the key.

SortedSet (of T)

This type is like a `SortedDictionary`, but you have to provide a custom routine to compare the keys.

Stack (of T)

This type allows elements to be retrieved by their key value.

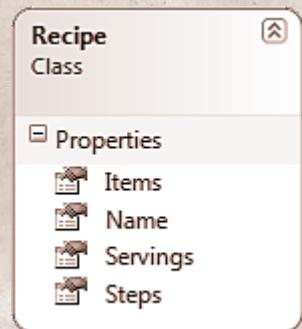


PUT ON YOUR THINKING HAT

Below is the class diagram for our `Recipe` class from the beginning of the chapter. As you can see, the `Recipe` contains two sets: one called `Items` that represents the ingredients in the recipe, and one called `Steps` that represents the preparation steps.

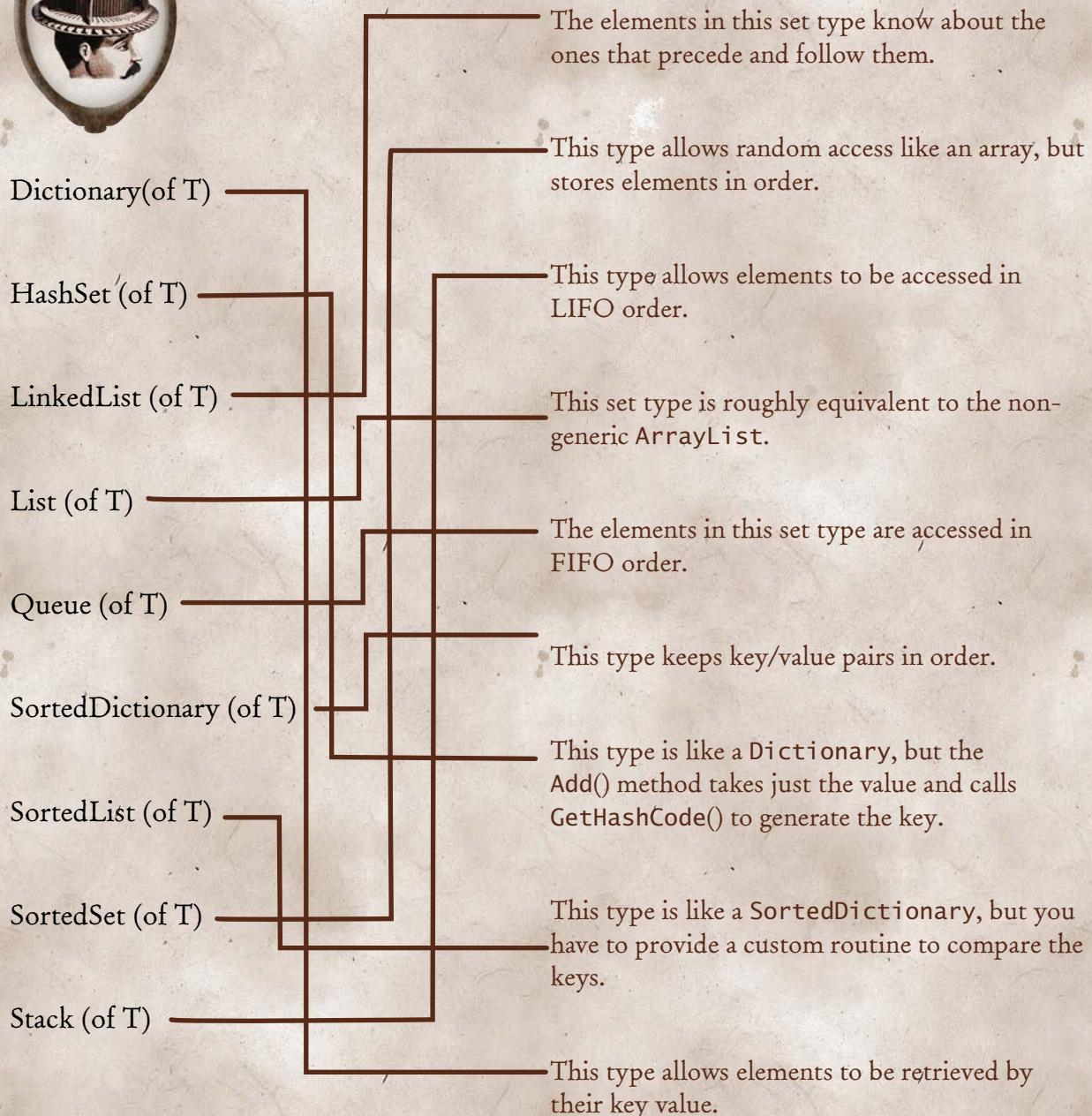


What set types would you use to represent the `Items` and `Steps` sets in a `Recipe`? Why?





HOW'D YOU DO?





HOW'D YOU DO?

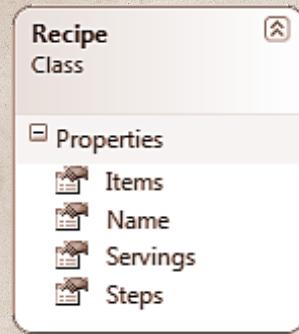
Remember, there are very few absolutes in programming. It doesn't really matter if you came to a different conclusion than I did, as long as you can defend your choice.



What set types would you use to represent the **Items** and **Steps** sets in a **Recipe**? Why?

Because the order of the ingredients is arbitrary, and they're not guaranteed to be unique (butter might be required several times in a single recipe, for example), I'd use a List(of T) for the **Items** set, which was the initial choice.

For the **Steps**, which are ordered, I'd use a Dictionary(of T).



TAKE A BREAK

Well done! We've finished the chapter, so take a break before we move on to the final Review?



REVIEW

What statement would you use to declare a five-element array of recipes?

How would you reference the fourth element in an array of integers?

Write the statement to loop through a `Dictionary<Recipe>` using `foreach`:

There is a non-generic equivalent to an array that takes instances of `System.Object`, and a generic version that is strongly typed. What are they?

What do LIFO and FIFO stand for, and what generic types represent them in the .NET Framework?

What's the most common type used to represent a Key/Value pair?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?

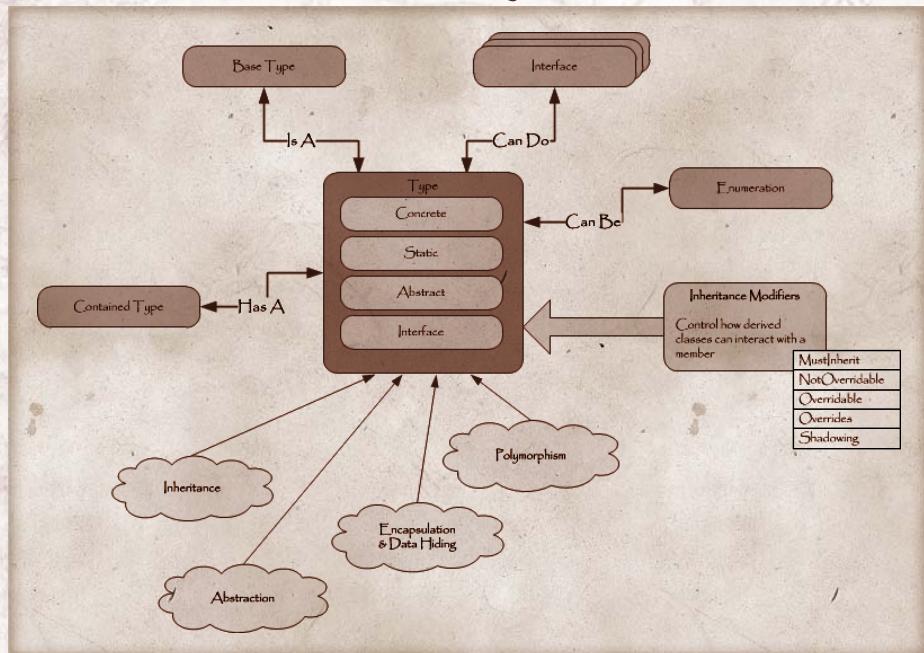


OOD

12

In the last eleven chapters, you've learned about VB syntax and the .NET Framework types that represent basic data. You know (in theory) how to define a type in code or by using the Visual Studio Class Designer. But if you're like most people, the idea of starting from a problem description and working out how to represent it as .NET Framework types is probably still pretty scary. How do you decide how the problem should be divided into classes? How do you choose between classes, structures, or something else? How do you know what properties and methods they need to expose? And how are those classes going to interact to actually do something?

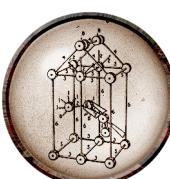
Every application you build is going to be different. That's what makes application design interesting. But you don't have to start with a blank slate. There are rules and principles that will help you get started. Collectively, they're part of OOA&D (Object-Oriented Analysis & Design). In this chapter we'll begin our examination of these guidelines by looking at how classes can interact based on their relationships and the basic principles of OOP (Object-Oriented Programming) and OOD (Object-Oriented Design).





FITTING IT IN

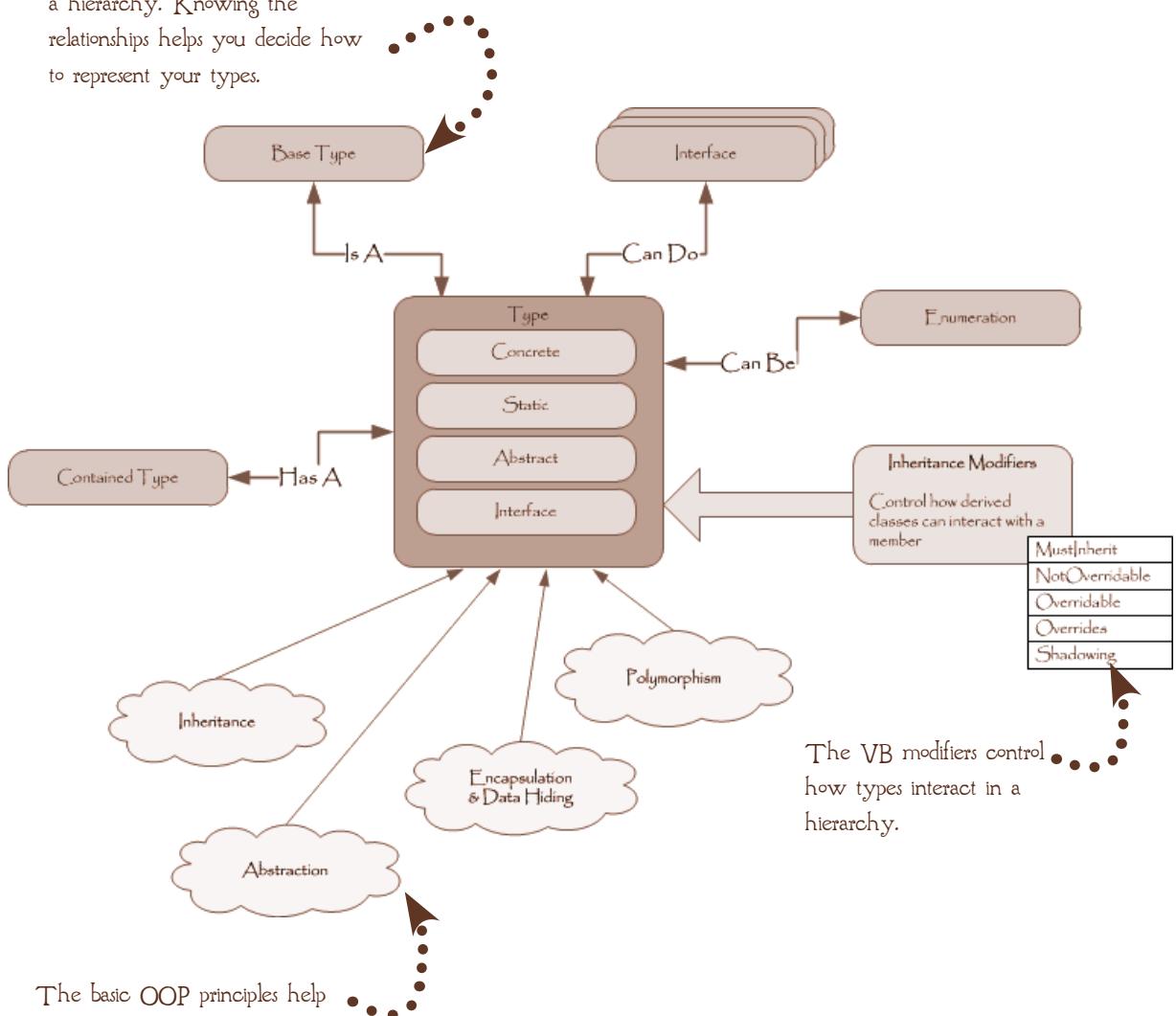
In this chapter, we'll begin our exploration of the principles and patterns that will help you to not just program, but program effectively and well.



We'll talk about
architectures and patterns
in Chapter 14.

In this chapter we'll explore the
principles of Object-Oriented Analysis
& Design (OOA&D).

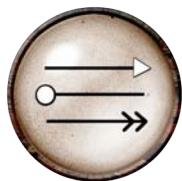
OOA&D helps define the relationships between classes in a hierarchy. Knowing the relationships helps you decide how to represent your types.





TASK LIST

In this chapter, we'll start by looking at the three kinds of relationships that can exist between types and then back up a bit to examine the fundamental principles of object-oriented programming. Finally, we'll examine the VB modifiers that allow you to control how the types in your application interact.



TYPE RELATIONSHIPS

We'll begin by exploring the four types of relationships that can exist between types—IsA, HasA, CanDo and CanBe—and how they're represented in the .NET Framework.



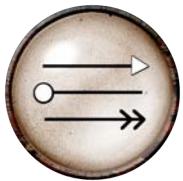
OOP PRINCIPLES

There are four basic principles that are fundamental to object-oriented programming: **ABSTRACTION**, the elimination of unnecessary or irrelevant detail; **INHERITANCE**, the ability of types to inherit behavior from a base; **ENCAPSULATION**, the hiding of implementation details, and **POLYMORPHISM**, the ability of derived classes to be substituted for their ancestors. We've already seen examples of all of these principles, but in the second section of this chapter, we'll examine the principles themselves and how to implement them in your class hierarchies.



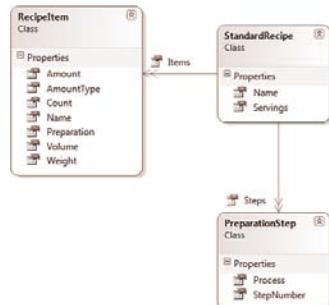
TYPE MODIFIERS

We've seen examples of modifiers like the `private` keyword that hide type members. VB and the .NET Framework also expose a set of modifiers that control how types interact and inherit from one another. In the last section of this chapter, we'll look at how these work.



TYPE RELATIONSHIPS

Here's the latest version of our Recipe class hierarchy. It's a good start, but when we show it to Neil and Gordon, they explain that for baked goods, they need to store not just the basic recipe, but a formula that expresses each ingredient in terms of bakers' percentages.



ON YOUR OWN

Here's one way to fix the Recipe hierarchy, using a separate class definition for baked goods and everything else. This design captures all



the requirements, but it's not a good design. Before we start looking at the object relationships that can help us improve it, take a few minutes to think about what's wrong. (Hint: In programming, certain kinds of laziness are actually best practice.)

WORDS FOR THE WISE

Professional bakers use a formula expressed in bakers' percentages, with ingredients expressed as the percentage of flour (which is always 100%). Expressing a formula in this way tells the baker a lot about how the dough will behave and makes it easy to adjust the yield.



KINDS OF RELATIONSHIPS

There are four basic categories of relationships that can exist between types in a hierarchy (assuming a relationship exists between the types at all—not every type is directly related to every other). They are:

ISA RELATIONSHIPS

An IsA relationship exists when one thing is a specific example of a more general kind of thing. A Mercedes-Benz IsA car. So is a Yugo. Lions and tigers both have IsA relationships with cat. IsA relationships are usually best modelled by collecting the common characteristics in a base type (usually, but not always, a class) and using inheritance for the more specialized types.

HASA RELATIONSHIPS

A HasA relationship exists when a type contains one or more instances of another. A car HasA steering wheel. A cat HasA tail. HasA relationships are usually modelled by exposing the instances of one type as properties of the type that owns it. (Technically, this is called OBJECT COMPOSITION.)

CANDO RELATIONSHIPS

CanDo relationships are behaviors that multiple types have in common. It is often the case that the types aren't related in any other way. CanDo relationships are usually modelled as interfaces. For example, all of the set types we looked at in the last chapter CanDo the things defined by the `IEnumerable` interface.

CANBE RELATIONSHIPS

CanBe relationships represent mutually exclusive options. For example, a car CanBe manual or automatic, but not both. CanBe relationships are implicit in type inheritance (a cat CanBe a lion or tiger, but not both). For properties, they are usually best modelled as an enumeration.



Oops. One of my editors has just pointed out to me that there *are* cars that are both manual and automatic. Who knew? Well, he did, obviously.



PUT ON YOUR THINKING HAT

Using the guidelines on the previous page, let's improve our class hierarchy. Start by labelling some of the relationships, and then create a new UML diagram showing any new classes that you think are necessary. (You can use the Class Designer or just sketch it on a piece of paper.)



BakedGoodRecipe _____ collection of FormulaItems.

Amount _____ one of Teaspoon, Tablespoon, Cup or Weight.

BakedGoodRecipe _____ kind of Recipe.

StandardRecipe _____ kind of Recipe.

Both StandardRecipe and PreparationStep _____ printing.

StandardRecipe _____ collection of RecipeItems.



HOW'D YOU DO?

Did you identify the relationships correctly? You really need to understand the problem you're modelling to do this, so if in doubt, ask your client. (Or in this case, ask a cook.)



BakedGoodRecipe HasA collection of FormulaItems.

Amount CanBe one of Teaspoon, Tablespoon, Cup or Weight.

BakedGoodRecipe IsA kind of Recipe.

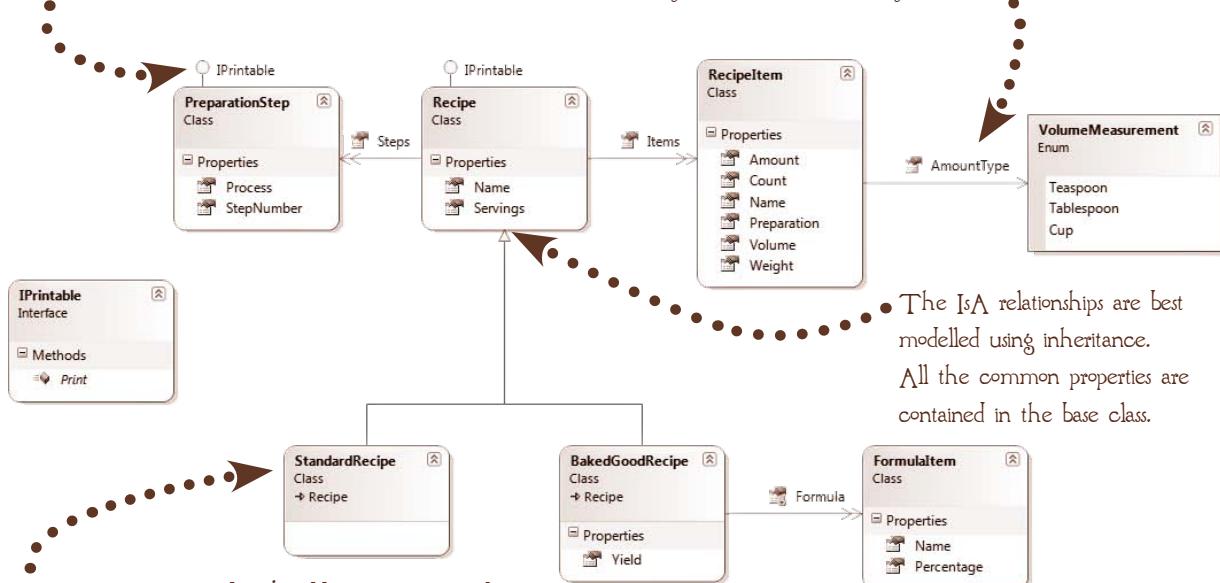
StandardRecipe IsA kind of Recipe.

Both StandardRecipe and PreparationStep CanDo printing.

StandardRecipe HasA collection of RecipeItems.

Here's my version of the revised hierarchy. Your version probably has some differences in names and properties, but the structure should be more or less the same. (Or if it isn't, you should be able to explain why yours is a better version. Remember, there are no absolute "right" answers in programming.)

The CanDo relationship has been extracted as an interface I've called **IPrintable** that's implemented by **Recipe** and **PreparationStep**.



StandardRecipe doesn't add any new members to the ones defined by the base class, but it will probably have a different implementation of the **Print()** method than **BakedGoodRecipe**, so I've kept it separate. Did you make a different decision?

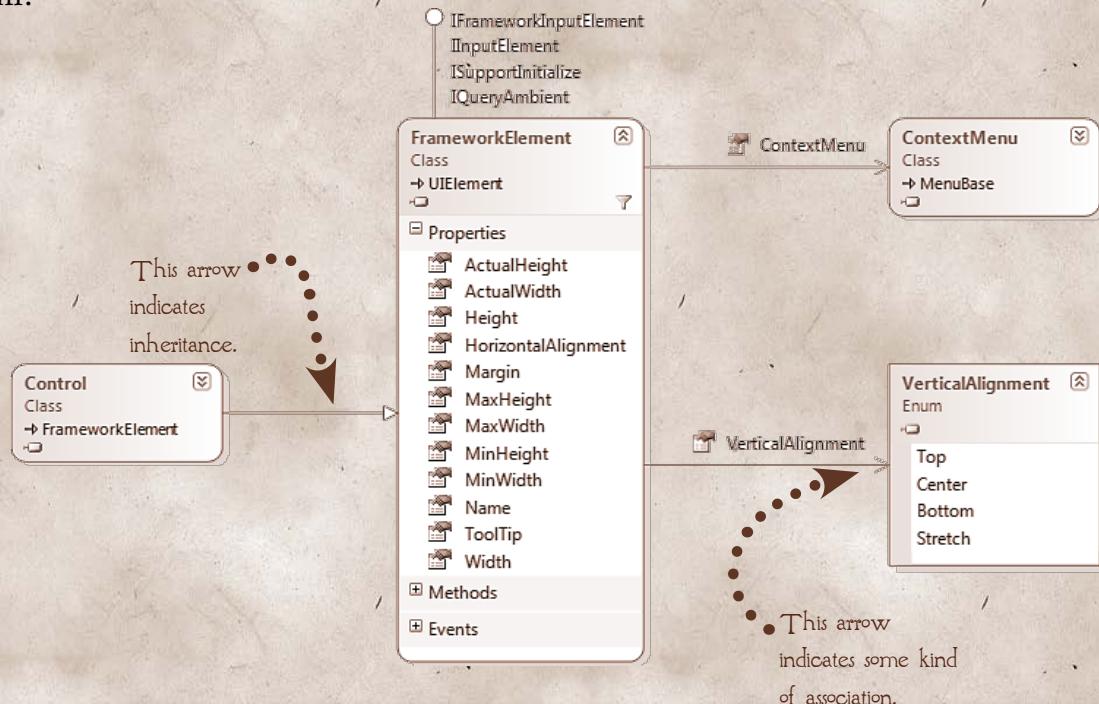


REVIEW

What are the four types of relationships between types in a hierarchy, and how is each one usually represented?

- _____ is usually represented using _____

This diagram has an example of IsA, HasA, CanDo and CanBe relationships. Can you find them?





OOP PRINCIPLES

Most design, in programming or any other field, doesn't start with a completely blank whiteboard. Even when presented with an entirely new problem (which is rare), a good designer will apply the principles of the field. You'll do the same—these principles are the most important tools in your toolbox.

We've already looked at one of them: the identification of object relationships that helps you decide how you represent them. In this section, we'll examine the four basic principles of Object-Oriented Programming: inheritance, abstraction, encapsulation, and polymorphism. Let's start by looking at how the boys use these principles when they develop a new recipe:

INHERITANCE

When I developed our thyme ice cream, I took our basic recipe and added herbs.

ENCAPSULATION

I'm not a chemist. I don't worry about how baking soda is formulated, just how to use it.

ABSTRACTION

When I write up the recipe for the staff, I don't include all the details, just what they need to do to make it.

POLYMORPHISM

Our Angels & Demons cake is a classic Devil's Food cake, but we use meringues instead of chocolate cake layers.



INHERITANCE

The ability for a type to inherit state and behavior from another type.

In the .NET Framework, classes can only inherit from a single base class. Some languages support MULTIPLE INHERITANCE which allows multiple base classes. It seems like a good idea but in practice tends to be very confusing. Interfaces provide much the same functionality in a conceptually simpler way.



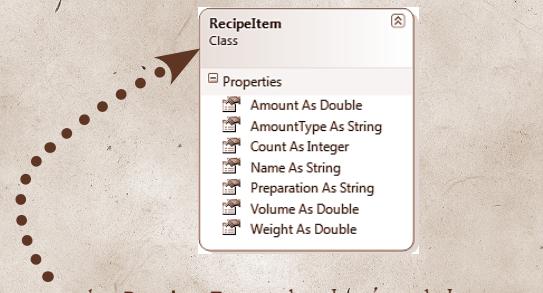
StandardRecipe
BakedGoodRecipe
both inherit from
Recipe.

ABSTRACTION

The simplification of an object model by including only necessary details and working at the most appropriate level of detail.

The word "Abstraction" is used in several ways in the programming world. You can also "abstract out" the common members of classes to define a base class.

```
List<Recipe> myList = new List<Recipe>();
...
myList[3] = new BakedGoodRecipe();
MessageBox.Show(myList[3].Name);
```



The RecipeItem class doesn't include nutritional information because it isn't relevant.

Abstraction also lets you work with the simplest form of an object. If you don't care about the **Formula** property of a **BakedGoodRecipe**, you can just treat it as a **Recipe**. (This is also an example of polymorphism. Do you know why?)

POLYMORPHISM

The ability of a type to be treated as any of its base types or any of the interfaces it inherits.

```
PreparationStep theStep = new PreparationStep();
Recipe theRecipe = new Recipe();

theStep.Print()
theRecipe.Print()
```



Both these classes implement **IPrintable**, so they both have **Print()** methods, but the methods do different things.

ENCAPSULATION

Also called “information hiding”, hides the implementation details of a type from the users of the type.

```
MessageBox.Show("Hello, World!")
```



You don't know how the **MessageBox** class displays itself when you call **Show()**. You don't have to. (You probably don't want to.)



PUT ON YOUR THINKING HAT

Let's start with abstraction. If you were building the model for Neil & Gordon's event planning system, which of the following members do you think should be included in the **Event** class?

EventDate

ClientBirthday

WallColor

NumberOfGuests

Venue

ParkingLotSize

Client

Menu



HOW'D YOU DO?

EventDate
WallColor
Venue
Client

• WallColor might be useful for an interior decorator, but not an event planner.

ClientBirthday
NumberOfGuests
ParkingLotSize
Menu

• ClientBirthday might be a good thing to store somewhere, but it's more properly part of a Client class, not the Event class.

This might be useful information for choosing a venue, but once you've made the choice, it's irrelevant.



MAKE A NOTE

The most common mistake that novice designers tend to make with abstraction is to include details “because they might be useful later”. In the world of programming, “later” doesn’t happen nearly as often as we expect it to. Before you include anything that isn’t immediately useful, you need to compare

- A) The chances that the “thing” will actually be needed
- B) The cost of adding the “thing” then to
- X) The cost of adding it now
- Y) The cost of having a bunch of extraneous stuff cluttering up your classes



PUT ON YOUR THINKING HAT

We'll talk about encapsulation in more detail in the next chapter, but let's look at an example. What's wrong with the following class declaration? Make the class a bit safer by changing the declaration. (Don't worry about implementation at this point.)

Class FinancialDetails

```
public Property BankName As String  
public Property AccountNumber As String  
public AccountBalance As Decimal  
End Class
```



HOW'D YOU DO?

Here's one way to fix the class declaration to make it safe. You might have done things a little differently, and that's fine, as long as you gave the class itself, and not the calling program, control over how and when the AccountBalance gets changed.

Class FinancialDetails

```
Public Property BankName As String  
Public Property AccountNumber As String  
Public AccountBalance As Decimal  
End Class
```

Would you REALLY let just anybody change the balance in your bank account? I certainly wouldn't.

In the next section, we'll learn some better options for making these public, too.

By requiring that changes to the account balance be made through functions, you can control what happens.

Class FinancialDetails

```
{  
    Public Property BankName As String  
    Public Property AccountNumber As String  
    Public Property Decimal AccountBalance  
        Get  
        ...  
    End Get  
    End Property
```

```
    Public Sub MakeDeposit(Decimal amount)  
        ...  
    End Sub
```

```
    Public Sub WithdrawCash(Decimal amount)  
        ...  
    End Sub  
End Class
```

At the very least, you'll need to make AccountBalance read-only.

POLYMORPHISM & CASTING

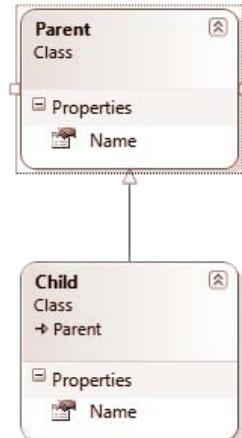
Polymorphism isn't the same as casting. When you cast a type to another, it behaves like the new type. The code that's executed is that of the type to which it is cast, not the original type.

Let's look at an example: Here's a simple class hierarchy with two classes, each exposing a single member, a property called Name. The Parent class returns the String "Parent" in the get accessor, while the Child class returns the String "Child". (We'll examine redefining inherited members in detail in the next section.)

```
Public Class Parent  
    Public Property Name As String  
        Get  
            Return "Parent"  
        End Get  
    End Property  
End Class
```

The **Shadows** keyword here lets us redefine the behavior of the property. We'll look at it in the next section.

```
Public Class Child  
    Inherits Parent  
    Public Shadows Property Name As String  
        Get  
            Return "Child"  
        End Get  
    End Property  
End Class
```



ON YOUR OWN



`myChild As Child = New Child()` What does this statement display?
`myParent As Parent = myChild`

`MessageBox.Show(myParent.Name)`

TYPEOF...IS

We've seen that you can cast from a type to one of its ancestors or an interface it implements by using the `CType()` function (or one of the dedicated conversion functions):

```
CType(oldType, NewType)
```

We've also seen that the `Object.GetType()` method (inherited by every type in the .NET Framework) returns an instance of the `Type` class, so you can do something like this to check whether two objects are of the same type:

```
If MyStandardRecipe.GetType() = MyRecipe.GetType()
```

This test certainly has its place, but it compares the types of two objects, and it's an absolute test. What if you wanted to test whether an object is a certain type but don't have an instance of that type lying around? You could use the `Name` property of the `Type` object returned by `GetType()`:

```
If MyStandardRecipe.GetType().Name = "StandardRecipe"
```

That's a little more useful, but it still doesn't tell you whether your object can be safely cast to the desired type. That's where the `TypeOf...Is` operator comes in:

```
If TypeOf MyRecipe Is BakedGoodRecipe
```



Notice that you're using the type name here as an identifier, not a string. Because type identifiers are known to the editor, that makes it a lot less prone to typos.

Even better, the `TypeOf...Is` operator returns `True` if the specified object has an `IsA` relationship with the specified type, which as you know, isn't quite the same as being an instance of the type. It can be used to test an object before casting:

```
If TypeOf MyStandardRecipe Is Recipe Then  
    Recipe myDish = CType(MyStandardRecipe, Recipe)
```



PUT ON YOUR THINKING HAT

The `TypeOf...Is` operator can simplify your coding and safely deal with casts. To get a feel for how this works, can you rewrite the following code using it?

```
Function DoSomething(thing As Object)  
    If thing.GetType().Name = "Parent" Or _  
        thing.GetType().Name = "Child" Then  
        Parent newThing = CType(thing, Parent)  
    Else  
        Try  
            ISomething newThing = (ISomething) thing  
        Catch  
            Throw New ArgumentException("Can't use thing.")  
    End Function
```



HOW'D YOU DO?

Here's my version. You might have structured the code a little differently. If you did, compare your version to mine and decide which you think is going to be easier to understand six months from now.

The `TypeOf...Is` operator replaces checking for any descendant of `Parent` and avoids comparing the type name to an (error-prone) string.

Using `As` here eliminates the complexity of a `Try...Catch` statement block.

```
Function DoSomething(thing As ISomething)
  If TypeOf Thing Is Parent Then
    Parent newThing = CType(thing, Parent)
  Else
    ISomething newThing = Thing As ISomething
    If interfaceThing = Nothing
      Throw New ArgumentException("Can't use thing.")
    End If
  End Function
```



TAKE A BREAK

That's it for the four basic principles of OOP and the other casting operators. Why don't you take a break before you complete the Review on the next page?



REVIEW

Representing an IsA relationship by defining all the common characteristics of related types in a base class is an example of the OOP principle of _____.

The _____ operator returns null if the specified object isn't compatible with the specified type.

The OOP principle of _____ is often called “information hiding”.

Casting is how the OOP principle of _____ is implemented in the .NET Framework.

Only including relevant information is one meaning of the term _____ in the programming world.

The _____ operator returns a Boolean value.



TYPE MODIFIERS

We've seen that modifiers like `Public` and `Private` control the visibility of a type member. VB exposes another set of modifiers and keywords that control how types interact within a class hierarchy.

SHARED

The `shared` keyword declares a member that belongs to the type itself, not an instance of the type. These are referred to as **STATIC MEMBERS**. By the way, VB doesn't support the `SHARED` keyword at the class level, but you can achieve the same thing using a VB-specific file called a **MODULE**. We'll see how to do that in a few pages.

`Public Class Recipe`

```
  Public Function Shared TabsToTsp(tabs As Integer)
    Return tabs * 3
  End Function
End Class
```

`asTsp = Recipe.TabsToTsp(15)`

Static members are called
directly on the type.



NOTOVERRIDABLE

The `NotOverridable` keyword, which only applies to classes, indicates that the class cannot be used as the base for any other class. Classes that can't be inherited are called **SEALED CLASSES**.

`Public NotOverridable Class Recipe`

`...`
`End Class`



`Public Class NewRecipe`

`Inherits Recipe`

`...`
`End Class`

You can't do this... •••••



MUSTINHERIT/MUSTOVERRIDE

The `MustInherit` keyword declares a class all of whose members must be implemented by an inheriting class, while `MustOverride` declares a member that must be implemented by an inheriting class. Types and members marked with these keywords are called **ABSTRACT**.

```
Public MustInheritClass Recipe  
Public MustOverride Sub DoSomething()  
End Sub  
End Class
```

The abstract definition doesn't include any implementation code.

```
Public Class NewRecipe  
Inherits Recipe  
Public Overrides Sub DoSomething()  
    MessageBox.Show("Something!")  
End Sub  
End Class
```

The defining class must include an implementation and use the `Overrides` keyword.

OVERRIDABLE

While the `MustInherit` keyword indicates a class or member that must be implemented by the child class, members declared as `Overridable` can be overridden, but need not be. They're known as **VIRTUAL** members.

```
Public Class NewRecipe  
Inherits Recipe  
End Class
```

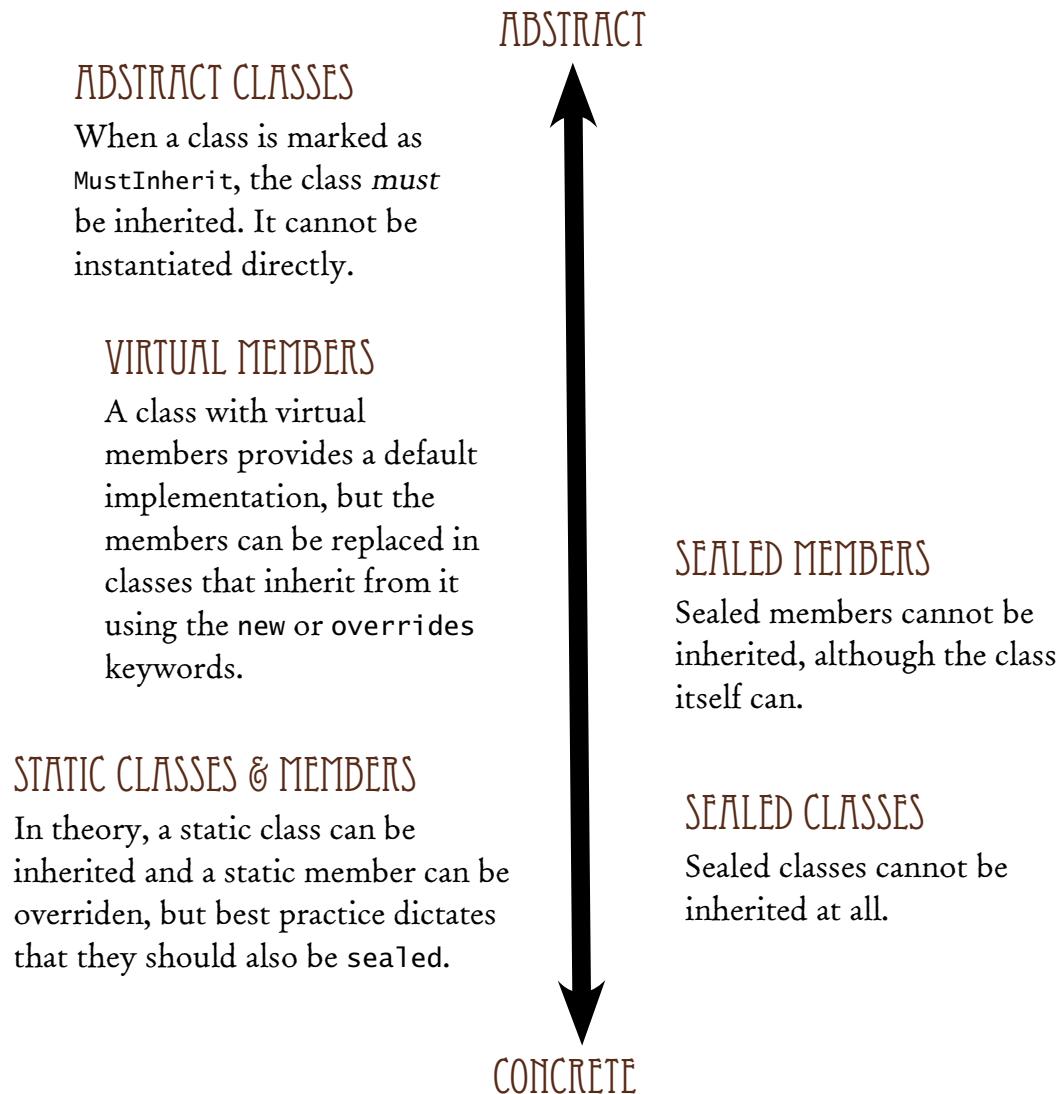
The inheriting class can provide a new implementation, but it doesn't have to.

```
Public Class Recipe  
Public Overridable Sub DoSomething()  
    MessageBox.Show("Something!")  
End Sub  
End Class
```

```
Public Class RecipeChild  
Inherits Recipe  
Public Overrides Sub DoSomething()  
    MessageBox.Show("Something Else!")  
End Sub  
End Class
```

TYPES OF TYPES

The `abstract`, `virtual` and `sealed` modifiers let you place types and members along a continuum from the completely abstract class that provides only the definition of a class and no implementation, to the completely concrete sealed class that implements all of its members and cannot be inherited.



MODULES

Some .NET languages like C# allow you to declare an entire class as static. Visual Basic doesn't, but you can create a **MODULE**, which behaves in much the same way.

Modules are like classes—they can contain variables, constants properties, methods and events, and they're translated into reference types by the compiler—but there are some important restrictions:

- **MODULES CAN ONLY BE DECLARED AT THE NAMESPACE LEVEL.**

That means that you can't declare a module inside a class or other module. In practice, modules are almost always created in their own source file.

- **MODULES DO NOT SUPPORT INHERITANCE.**

You can't use the **Inherits** keyword inside a module, and other types can't inherit from them.

- **MODULES CANNOT BE INSTANTIATED.**

In other words, you can't declare a variable **As MyModule**. Instead, you can simply reference the module's members directly:

```
MyResult = MyModule.Method()
```

- **ALL OF THE MEMBERS OF A MODULE ARE SHARED.**

You don't need to use the **SHARED** keyword when you declare the members of the module. (In fact you can't.) All of the module's members will be implicitly shared.



PUT ON YOUR THINKING HAT

I bet you can guess the syntax for declaring a module. Why don't you try it? Write a module that declares the constants for the number of teaspoons per tablespoon (**3**), tablespoons per cup (**16**) and fluid ounces per cup (**8**).

What statement would you need to include in a source file in order to reference these constants?



HOW'D YOU DO?

It doesn't matter if you chose different names for your module and variables, as long as the names you chose are valid identifiers and make some sort of sense.

Write a module that declares the constants for the number of teaspoons per tablespoon (3), tablespoons per cup (16) and fluid ounces per cup (8).

```
Public Module CookingConversions  
    Constant TspPerTab As Integer = 3  
    Constant TabsPerCup As Integer = 16  
    Constant OzsPerCup As Integer = 8  
End Module
```



Did you guess right? The basic pattern for declaring a module is the same as a class or a method.

Modules can only be declared as Public or Friend. By default they're Friend, which is hardly ever what you want.

```
[Public | Friend],Module <Name>  
...  
End Module
```

CREATING MODULES

You probably won't find yourself creating modules all that often, but they're useful in certain situations:

- WHEN THERE SHOULD ONLY EVER BE A SINGLE INSTANCE OF THE OBJECT

This is called 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.

- TO GROUP GENERAL-PURPOSE CONSTANTS AND METHODS

As an example, the .NET Framework `System.Math` class provides constants like `Pi` and general mathematical functions like calculating the natural logarithm of a number. In our cooking example, constants like the number of teaspoons in a tablespoon or a cup might be good candidates for a static class.

- TO CONTAIN GLOBAL VARIABLES

As you've seen, one of the basic principles of OOP is encapsulation, the hiding of information from outsiders. It's a principle for a good reason, but 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.

- TO CREATE EXTENSION METHODS

Extension methods are a special way of extending the functionality of an existing class (including a class in the Framework Class Library) without having access to the original source code. We'll look at them at the end of this chapter.



PUT ON YOUR THINKING HAT

Based on the guidelines above, which of the following would be good options for static classes or members?

- A class that models the nutritional qualities of an ingredient
- A set of constants and methods for converting between teaspoons, tablespoons, cups and liters
- A class that manages user configuration preferences, like the typeface or window layout
- A class that represents a log-in dialog that controls user access to the application



HOW'D YOU DO?

Here are my answers, but be aware that these examples are out of context, and only the first two are clear-cut. You may have imagined the role of the others in an application differently...

No A class that models the nutritional qualities of an ingredient

This is a classic example of a concrete class. The class might define static methods, but it's difficult to imagine a situation in which the class itself should be static.

Yes A set of constants and methods for converting between teaspoons, tablespoons, cups and liters

And this is a classic example of a static class. You only need one instance of these constants, and the methods are probably going to be useful to multiple other classes that might not be related in any other way.

Yes A class that manages user configuration preferences, like the typeface or window layout

This one could go either way. Most applications only have a single user at any one time, which would make this a candidate for a singleton class that only has a single instance, which is what I've chosen. But you might want to accommodate multiple (consecutive) users with a concrete class.

Yes A class that represents a log-in dialog that controls user access to the application

In every application I've ever seen, there can only be a single log-in dialog displayed at any one time, which makes this a classic singleton. Can you imagine a situation in which there's a need for more than one instance?

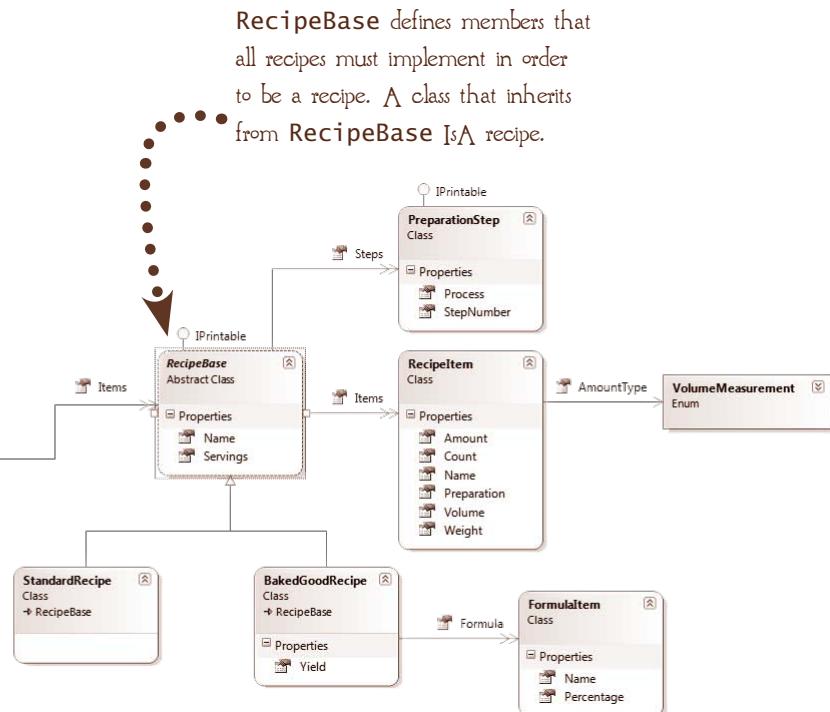
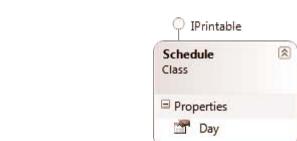
ABSTRACT CLASSES & INTERFACES

An abstract class (implemented in VB using the `MustInherit` keyword) is very like an interface. It defines the members that must be implemented by an inheriting class but doesn't usually provide any implementation. So how do you decide which to use? In practice, it isn't that difficult. Just remember that interfaces represent a `CanDo` relationship, while classes (abstract or otherwise) represent an `IsA` relationship.

Let's look at an example. Remember that weekly production schedule that Neil and Gordon need to prepare? I've added it to the basic Recipe class hierarchy. The only relationship the Schedule class has to the other classes in the hierarchy is that it HasA collection of Recipe items, but, like a Recipe, it can print itself.

`IPrintable` is clearly an interface. It's something that otherwise unrelated classes

CanDo.

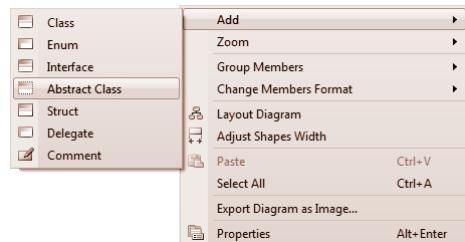


`RecipeBase` defines members that all recipes must implement in order to be a recipe. A class that inherits from `RecipeBase` IsA recipe.

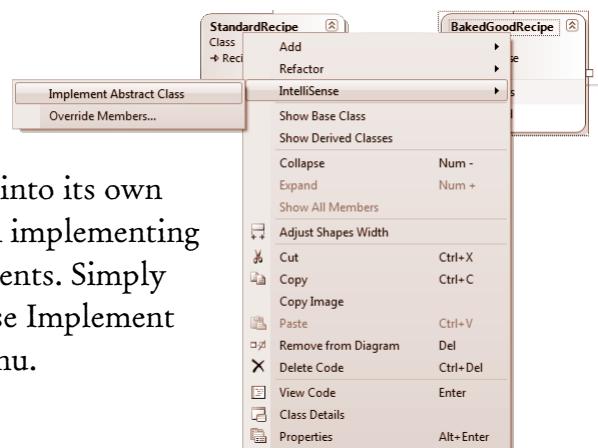
MAKE A NOTE

By convention, an abstract class should have "Base" at the end of its name, as in `RecipeBase` (or, as we'll see, .NET Framework classes like `ButtonBase`). Classes that descend from the abstract class should have the object appended to their names: `StandardRecipe`, or `ToggleButton`.

ABSTRACT SYNTAX



You can add an abstract class from the Class Designer, but you can't define abstract members. You have to do that in code.



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.

As you might expect, the code for implementing an abstract class or abstract member looks very like the code for an interface. Only the signature is defined, without any implementation, marked with the `MustInherit` or `MustOverride` keyword.

Abstract classes can contain concrete code. This is an auto-implemented non-abstract property.

```
Public MustInherit Class myClass
    Public Property myProperty As Integer
    Public MustOverride Property myAbstractProperty
        Public MustOverride Sub DoSomething()
    End Class
```

The compiler will complain if you try to put code in an abstract method declaration.

Only the `MustOverride` keyword sets this property apart from the concrete version.



PUT ON YOUR THINKING HAT

Given the descriptions below, which of these do you think should be implemented as an abstract class, and which should be implemented as an interface? Hint: If the descriptions don't seem clear to you, think about where the types might sit in a hierarchy, specifically, whether the members might be needed in more than one limb of the tree.

	Abstract Class	Interface
A set of members that describes the characteristics of all menus	_____	_____
A set of members that describes how content is scrolled in a control	_____	_____
A set of members that describes how buttons (standard buttons, radio buttons, etc.) behave	_____	_____
A set of members that describes how a control that allows a user to enter and edit text behaves	_____	_____
A set of members that describes how controls that can represent a range of values, like a ScrollBar or ProgressBar, behave	_____	_____
A set of members that describes the behavior of a control that can contain other controls	_____	_____



HOW'D YOU DO?

Did any of these descriptions seem suspiciously familiar? They're all taken from the `System.Windows.Forms` namespace that contains the types used to create Windows Presentation Foundations applications, and I mentioned the `ButtonBase` class as an example of an abstract base class. I've given you the name of the actual type in the FCL if you want to explore further.

A set of members that describes the characteristics of all menus

Abstract Class

MenuBase

Interface

IScrollInfo

A set of members that describes how buttons (standard buttons, radio buttons, etc.) behave

Abstract Class

ButtonBase

TextBoxBase

A set of members that describes how a control that allows a user to enter and edit text behaves

Abstract Class

RangeBase

IContainerControl

A set of members that describes how controls that can represent a range of values, like a ScrollBar or ProgressBar, behave

A set of members that describes the behavior of a control that can contain other controls

SEMI-ABSTRACT CLASSES

VB doesn't restrict you to "reimplement everything" (abstract) classes or "reimplement nothing" (concrete) classes. The `Overridable` keyword allows (but doesn't require) a derived class to reimplement the method. This allows you to provide a default implementation of your methods. If the derived class doesn't redefine the method using the `Shadows` or `Overrides` keyword, the implementation provided by the base class is used.

The `Shadows` and `Overrides` keywords can be tricky. Here's how MSDN defines them:

- `Shadows` is used to hide an inherited member from a base class member
- `Overrides` is required to extend or modify the abstract or virtual implementation of an inherited member

Doesn't really tell you much, does it? What's the difference between "extend or modify" and "hide"?



ON YOUR OWN, PART 1

Let's look at an example. Here's a simple class diagram. The `myBase` class defines a method called `GetClassName` that returns (not surprisingly) the name of the class as a string. The `myOverride` class redefines the method using the `override` keyword, and the `myNew` class redefines it using the `new` keyword. `myNothing` doesn't redefine the class at all. What do you think each of these child classes returns in the following situations?

x As MyBase = New myBase()

x As MyOverride = New myOverride()

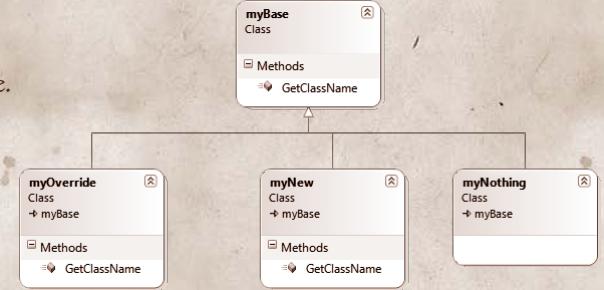
x As MyNew = New myNew()

x As MyNothing = New myNothing()

x As MyBase = New myOverride()

x As MyBase = New myNew()

x As MyBase = New myNothing()





ON YOUR OWN, PART 2

Try it out. Define the classes shown in the diagram on the previous page, implement the classes (don't forget to mark myBase, GetClassName() as Overridable) and build a simple screen that displays the results in a MessageBox when the button is clicked.



x As MyBase = New myBase()

x As MyOverride = New myOverride()

x As MyNew = New myNew()

x As MyNothing = New myNothing()

x As MyBase = New myOverride()

x As MyBase = New myNew()

x As MyBase = New myNothing()

Were the results what you expected? Based on what actually happened, how would you define the new and override keywords?

Shadows:

Overrides:

EXTENDING THE DEFAULT

Despite what the MSDN documentation says about the `Shadows` and `Overrides` keywords, neither one actually *extends* the default implementation provided by the base class, at least not in the sense of "do that, and do this too". But extending the implementation in that sense is a perfectly reasonable and sometimes necessary thing to do, and the `MyBase` keyword allows you to do just that.

1

```
Public Class MyParent()
    Public Overridable Sub SayHello()
        MessageBox.Show("Hello")
    End Sub
End Class
```

2

```
Public Class MyChild()
    Inherits MyParent
    Public Overrides Sub SayHello()
        MyBase.SayHello()
        MessageBox.Show("World")
    End Sub
End Class
```

3

```
x As MyChild = New MyChild()
x.SayHello()
```

Call the method of
the base class.

This code that calls the
`MyChild.SayHello()` method
displays two `MessageBox`
windows.

BASE CONSTRUCTORS

Can you guess what the syntax is to call the constructor of a base class? That's right; you just call `MyBase.New()`. Of course, you can pass any arguments you need to the call, as well:

```
Public Class MyClass  
Inherits MyBase  
  
Public Sub New()  
    MyBase.New()  
    ...  
End Sub  
End Class
```

```
Public Class MyClass  
Inherits MyBase  
  
Public Sub New(value As Integer)  
    MyBase.New(value)  
    ...  
End Sub  
End Class
```

ON YOUR OWN

Can you think of three reasons why it's better to call the constructor of the base class rather than repeat the code in your derived class?

Can you think of a situation in which you wouldn't want to call the base class constructor?

PREVENTING INHERITANCE

At the opposite end of the inheritability spectrum from the abstract classes and members that must be inherited are sealed classes. The syntax to seal a class is straightforward. Just mark a class with the `NotOverridable` keyword and other classes can't inherit from it:

```
Public NotOverridable Class mySealed
```

Classes are easy, but those tricky keywords `new` and `override` get involved when you want to seal individual methods within a class. Here's how that works:

```
Public Class myBase
    Public Overridable Sub DoSomething()
        ...
    End Sub
    Public Sub DontTouchMe()
        ...
    End Sub
End Class
```

A non-virtual method is effectively sealed, but...

...it can be redefined using the `Shadows` keyword, and...

```
Public Class myNew
    Inherits myBase
    Public Shadows Sub DontTouchMe()
        ...
    End Sub
End Class
```

```
Public Class mySealed
    Inherits myBase
    Public NotOverridable Overrides Sub DoSomething()
        ...
    End Sub
End Class
```

...a virtual method can only be sealed for further inheritance when combined with `Overrides` (not `Shadows`).



PUT ON YOUR THINKING HAT

It might seem that only being able to mark a method as `NotOverridable` when you use the `Overrides` 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?



HOW'D YOU DO?

The definition of `DontTouchMe()` on the previous page is sealed because it's non-virtual. In order to allow it to be inherited (as opposed to replaced) you'd need to add the `Overridable` keyword:

```
Public Class myNew
    Inherits myBase
    Public Shadows Overridable Sub DontTouchMe()
        ...
    End Sub
End Class
```



TO SEAL OR NOT TO SEAL...IS THAT REALLY THE QUESTION?

You can get yourself in a terrible muddle over when to seal, what to seal, when to virtualize, and whether to override or replace functionality. And, as always, the answers aren't easy. In fact, there's a huge debate in the professional development community over sealing. We call these "religious debates" (and they happen a lot in the programming world) because people get dogmatic and throw around words like "evil". If you're interested in the great debate, here are two blogs with well-presented arguments to get you started (although they use C# examples):

Chris Sells at www.sellsbrothers.com/Posts/Details/12564

Eric Lippert at blogs.msdn.com/b/ericlippert/archive/2004/01/22/61803.aspx

My advice? Relax. Here are the rules of thumb I use:

- When you're designing a hierarchy it will be obvious that some classes are intended to be base classes.
- If an obvious base class can or should provide some default functionality, make it a concrete class with virtual methods; otherwise, make it abstract.
- Leave your remaining concrete classes unsealed so that they can be extended with new functionality, but leave their methods sealed (non-virtual).

My reasoning for that last point might need some explanation--as Eric Lippert points out, a virtual method can be dangerous when the class gets modified (and it will). Unsealing a class later if you need to won't have any side effects. But sealing it later when you discover you were wrong could break other code. So making the class inheritable but the methods non-virtual is the path of least resistance.

EXTENDING CLOSED CLASSES

There's one last twist to working with classes in .NET: extension methods. EXTENSION METHODS behave as though they were standard instance methods, but they're defined in a separate module. They allow you to add new methods to a class without recompiling or inheriting from it, so you can extend the functionality of sealed classes and classes, like those in the BCL, that you can't or shouldn't change directly. Let's look at how they work:

① Create a Public Module class to contain the methods.

② Import the System.Runtime.CompilerServices namespace.

③ Mark each method with the `<Extension()>` attribute and declare it `Public`:

```
<Extension()
Public Sub MyMethod...
```

④ The first parameter of the method must be the name of the type the method extends. You don't actually pass this argument to the method when you call it.

⑤ To use the extension, simply Import the module file, and then you can call the method just as though it were an instance method. (Remember, you omit that first parameter specifying the class.)



PUT ON YOUR THINKING HAT

Based on the steps outlined above, write an extension method (including the Imports statement) that declares an extension method called `PrintRecipeCard()` on the `Recipe` class. Don't worry about the implementation, and assume that the method takes no arguments, and returns nothing.

After you've written the extension, write an example of calling the extension method (again, including the Imports statement).



HOW'D YOU DO?

How's you do? Dd you get all the Imports and the declarations correct?

The attribute must
be included for every
method in the module.

Module RecipeExtensions

<Extension()>

Public Sub PrintRecipeCard(theRecipe As Recipe)

...
End Sub

End Module

You need to import this
Imports System.Runtime.CompilerServices namespace in order to have
access to the Extension()
attribute.

This first parameter identifies the class the
method extends.

To call the method, you'll need to reference the namespace:

Imports RecipeExtensions

Then it behaves just like a normal instance method:

Dim myRecipe As Recipe = New Recipe() Omit that special first parameter.
myRecipe.PrintRecipeCard()



UNDER THE MICROSCOPE

You won't create extension methods nearly as often as you'll use the ones created for you in the BCL. But it's good to know that they're there when you need them. But there are a few things to be aware of:

You cannot redefine a method using extension methods. The compiler will always give precedence to the instance method, so your override will simply never be called.

You can define a method with a different signature, but you can't use the base keyword and you can't reference private or protected members of the class you're extending.

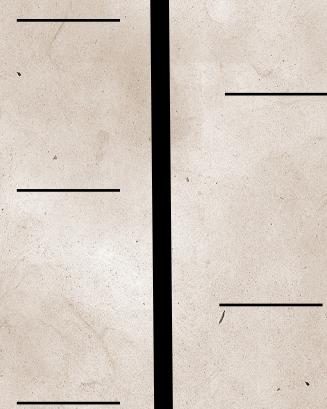


REVIEW

Now that you've seen the keywords, can you place each of the snippets below on the continuum from most abstract to most concrete?

- ① Public Sub DoSomething()
- ② Public MustInherit MyBase
- ③ Public MustInherit Sub DoSomething()
- ④ Public Overridable Sub DoSomething()
- ⑤ Public NotOverridable MyClass
- ⑥ Public NotOverridable Overrides DoSomething()

ABSTRACT



TAKE A BREAK

Wow, this chapter was a doozy, wasn't it? Don't get too worried. A lot of this won't be useful to you for awhile, but it's important that you know it's there. Why don't you take a break before you complete the final Review, and move on to the next chapter on programming principles?



REVIEW

Can you match the terms and keywords with their definitions?

MustInherit

This keyword returns a Boolean value indicating whether a variable can be cast to the specified type.

Inheritance

This keyword allows a method to be overridden in an inheriting class.

Is

This term indicates the ability of one type to inherit state and behavior from another.

Abstraction
NotOverridable

This keyword returns a variable cast to the specified type, or `null` if that's not possible.

Encapsulation
Shared

This keyword marks a method or class that must be inherited.

Polymorphism
As

This term refers to the ability of a type to be treated as one of the types from which it derives.

Overridable

This term refers to the general principle of leaving out unnecessary details.

This keyword marks a method as belonging to the entire class, not an instance.

This term refers to the ability to use a method or class without knowing the details of its implementation.

This keyword prevents a class or method from being inherited.

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



PROGRAMMING PRINCIPLES

13

In the last chapter, we looked at some of the theory behind designing object-oriented applications, specifically how the relationships between types can be defined and implemented. It's probably obvious that understanding the mechanics of an object-oriented design in that way is necessary, but it's only part of the picture.

The design principles we'll look at in this chapter address a basic fact of the programming life: It's gonna change. Nobody is ever going to ask an architect to insert three floors in the middle of the Empire State Building, but it happens to programmers all the time. So think of this chapter as a kind of self-defense course for programmers. The principles we'll explore will help you build an object model that's resilient to change (which is programmer-speak for "it won't break more when you fix it".)

Common wisdom says that any given application will spend four times as long being changed as it will being developed in the first place, and studies have shown that 80% of that maintenance effort will be spent modifying the behavior of the application, not fixing errors.

Yikes. Better be ready for it, hadn't we?



WORDS FOR THE WISE

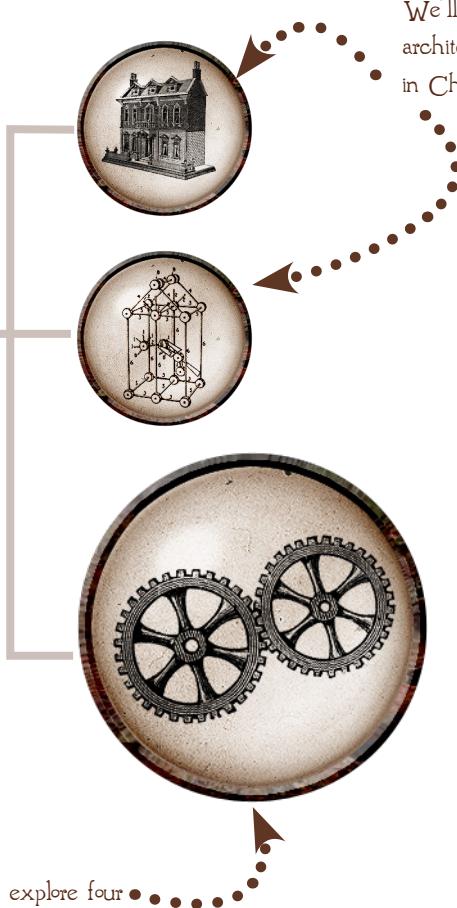
"There's no difference between theory and practice in theory. There's a lot of difference between theory and practice in practice."

Variously attributed to the inimitable Yogi Berra, the computer scientist Jan van de Snepscheut and Albert Einstein.



FITTING IT IN

In the last chapter we examined the basic principles of Object-Oriented Analysis & Design. In this chapter we'll continue that exploration by examining four of the classic design principles.



We'll talk about architectures and patterns in Chapter 14.

In this chapter we'll explore four classic design principles.



SMELLY CODE

One of the best ways to learn to write good code is to learn how to recognize bad code. Code that doesn't work is obviously bad, but there are other hallmarks. Most programmers call these **CODE SMELLS**. There are lots of them (there's an infinite number of ways for anything to be not-quite-right), and every theorist seems to list them differently, but here are a few to get you started:

RIGIDITY

If changing one piece of code requires changes to another piece, which might then require changes to a third piece, and...the code is rigid. And smelly.

FRAGILITY

Cynical programmers (that is, anybody who's done it for a living) say that there are always three bugs in any application: the one you know about, the one you don't know about, and the one you're going to create when you fix the one you know about. Fragile code breaks in unexpected ways, in areas that don't seem to be related to what you're working on.

IMMOBILITY

A common situation: You're modifying an application, and you want to reuse what was supposed to be a general-purpose bit of code. But the snippet you need is so tightly entrenched that the only way to reuse it is to cut and paste. You're looking at code that's immobile. That's a smelly waste of time.

REDUNDANCY

Immobile code is the primary cause of this design smell. When you're programming, some kinds of lazy are good. If there's any bit of code that appears in more than one place, you've got a smelly problem, because if the snippet needs to change (and it will), you need to change it everywhere. How do you spell "boring and error-prone"?

OPACITY

I really wish I had money for every time I've seen a comment like, "Don't know how this works, but it does" in a released application. I don't need to tell you that this is unacceptable, right? If you don't understand it, how are you going to know how to change it? You should write code that makes comments seem unnecessary. (But comment everything anyway.) If you need the comment to understand the snippet, then your code is opaque, and opacity smells. Really, really badly.

IF IT AIN'T BROKE...

On a slow news day, the American press is fond of regaling us with stories of military overspending. Remember the \$640 toilet seat or the \$7,600 coffee maker? Now I'm not privy to the Department of Defense's procurement procedures, but I'd bet I know how those things happen: When you're not certain what constitutes "good enough", the safe bet is to substitute "best", and that almost always means "most expensive you can find". It's not unreasonable, really, but it can (and all too frequently does) result in embarrassing over-spending.

The same thing applies to software design. The principles we'll explore in this chapter come at a price, in initial design, development time, and application complexity. That price isn't always justified. (Would you buy a Formula 1 race car just to go to the grocery store? Well, you might if you're a gear-head, but could you justify the expense to your spouse or worse, to the IRS?)



So what's a poor programmer to do? When you're evaluating your design, ask yourself two questions:

- Does this class (or class member) violate one of the design principles we're about to discuss?
- Does it matter?

That second question is the critical one. For example, most of the principles we'll be exploring in this chapter are aimed at reducing the cost of maintenance. But a surprising number of applications aren't maintained; they're replaced.

Even when you're developing an application that you expect to be in service for some time, the added complexity of implementing all the principles rigidly may not be justified, because some things just don't change. So, here's a way of going forward. It's not ideal, but it works in the real world.

- ① Review your initial design in light of the design principles.
- ② If you find that a class violates a principle, consider how expensive fixing it now would be. If it's a simple change, you should implement it now.
- ③ If bringing the class into compliance with the principle is going to add a lot of complexity, consider how likely it is to change. If the class seems to you to be volatile, implement the change.
- ④ If, on the other hand, the class doesn't seem particularly volatile, document the changes required, but don't implement them until (and unless) the class actually does change during maintenance.



TASK LIST

We'll be looking at four design principles in this chapter. Lots and lots of authors have put forward lots and lots of principles, but these four are generally considered the most important, and many of the others are variations or refinements of these basics.



THE SINGLE RESPONSIBILITY PRINCIPLE

This principle states that any given object should do one thing and one thing only. The compliance of an object to the Single Responsibility Principle is measured by its COHESION.



THE OPEN/CLOSED PRINCIPLE

This principle states that objects should be open for extension but closed for modification. Basically, that means that you should be able to add new behavior without changing the old stuff. This trick is most often implemented by having a client object talk to an interface or abstract class rather than a concrete one.



THE LISKOV SUBSTITUTION PRINCIPLE

This principle states that any object must be completely substitutable for its base class. The LSP is a refinement of basic polymorphism. Complying with it means, among other things, that you don't create classes (called degenerate classes) that are less capable than their parents.



THE LAW OF DEMETER

The Law of Demeter, also known as the Principle of Least Knowledge, states that any given object should only directly reference its own members and members of the objects that it instantiates.

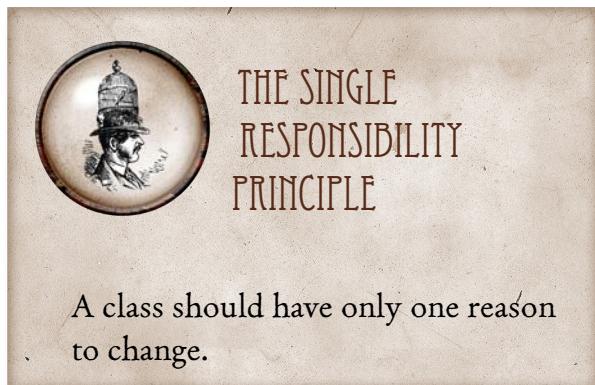
SINGLE RESPONSIBILITY

The Single Responsibility Principle (SRP) is all about doing one thing and doing it well. The theory is that if a class is responsible for more than one thing, it has more than one reason to change. That makes the class harder to understand and harder to maintain.

By itself, that seems pretty obvious, right? You wouldn't try to make a class that represents lions and elephants. But in practice, it's one of the hardest things a designer does. People are really good at finding the connections between things; not so good at deconstructing them. Lions and elephants are both animals, they're both mammals, they're likely to be found in the zoo, or on the savannah...

Now that's a slightly silly example. But what about a recipe? It's something that provides instructions for preparing something in the kitchen. But in the context of a computer application, it's also something that displays itself on the screen and formats itself for the printer, and perhaps provides guidelines for serving. The first cut of your class design is likely to include all those things, because those are all things a recipe does. But all those responsibilities are different reasons to change. Maybe the restaurant buys new plates, so the serving instructions change, or they decide to make recipes available on its Web site (so the display changes from WPF to Silverlight), or...the SRP tells you that while yes, those are all things a recipe does, they probably shouldn't all be modelled in the same class.

Fortunately, in addition to just asking ourselves why something might change, we also have a tool related to the SRP that makes the task of checking responsibilities much easier. It's the idea of COHESION. There's a continuum of cohesion, from chaotic at one end to functional at the other, and those levels are useful ways to think about how you design your methods and classes. Keep reading to find out more...



"Every good chef has a suite of knives. Yes, they all cut, but trying to use a bread knife to bone a chicken is just making hard work for yourself."



LEVELS OF COHESION

At the class level, the application of the Single Responsibility Principle is usually obvious, but it's surprisingly easy to violate it when designing methods. One of the best tools for determining what (or rather, how much) a method should do is to evaluate the method based on its level of cohesion.



FUNCTIONAL COHESION

The most cohesive methods are those that perform a single function, like a knife that's intended only for boning or cutting bread. Functionally cohesive methods epitomize the Single Responsibility Principle.



SEQUENTIAL COHESION

Methods that perform a series of operations that use the same data and must be performed at the same time, usually in a specific order, display sequential cohesion. The operations themselves should be implemented as separate functionally cohesive methods, so that the sequentially cohesive method acts like a train—a locomotive and a set of cars of different types, all going to the same place at the same time.



TEMPORAL COHESION

Temporally cohesive methods perform operations that don't necessarily share the same data or need to be performed in a specific order but must be performed at the same time.



COMMUNICATIONAL COHESION

Methods that display communicational cohesion only share the same data. A method that displays this kind of cohesion can be a convenient wrapper for groups of operations that are frequently performed together, but you should examine it with suspicion. If the method does anything other than call functionally cohesive methods, it's probably poorly designed.



CHAOTIC COHESION

Groups of unrelated random operations display chaotic cohesion, which simply means that they show no cohesion at all. You'll sometimes see odd methods that accept parameters that define what operation they should perform. These are badly designed, and should be fixed immediately. Do not pass go, and do not collect \$200 until you've done so.



PUT ON YOUR THINKING HAT

Can you match the description of each method on the right to the level of cohesion it displays on the left?

FUNCTIONAL

The `GetIngredientNutrition()` method calculates the USDA nutritional values for a specific Ingredient.

SEQUENTIAL

The `RecalculateRecipe()` method accepts a number of servings, and updates the amount of each ingredient in a recipe with the new amounts required, and the `NutritionalData` structure of the recipe with the new values.

TEMPORAL

The `Recipe()` constructor performs all the operations needed to initialize a new instance of the `Recipe` class.

COMMUNICATIONAL

The `ConvertAndCalculate()` method converts all ingredients to metric measurements and recalculates the nutritional data.

CHAOTIC

The `GetPrintableVersion()` method accepts the number of servings as an argument and then calculates the ingredient amounts and nutritional data and returns a formatted `FixedDocument` object.



PUT ON YOUR THINKING HAT

No, that wasn't an editing mistake. The descriptions of the functions are in order of cohesion. Did you have the courage of your convictions, or did you think you'd gotten it wrong when you started drawing all those horizontal arrows?

FUNCTIONAL

This method does only one thing. It's functionally cohesive, and demonstrates the SRP.

The `GetIngredientNutrition()` method calculates the USDA nutritional values for a specific ingredient.

SEQUENTIAL

Because the ingredient amounts and nutritional data must be synchronized, this is a valid sequential operation.

The `RecalculateRecipe()` method accepts a number of servings, and updates the amount of each ingredient in a recipe with the new amounts required, and the `NutritionalData` structure of the recipe with the new values.

TEMPORAL

Constructors are the classic example of methods that display temporal cohesion.

The `Recipe()` constructor performs all the operations needed to initialize a new instance of the `Recipe` class.

COMMUNICATIONAL

This method uses the same data, the `Items` collection of the `Recipe` but the two operations aren't otherwise related. It should be re-designed.

The `ConvertAndCalculate()` method converts all ingredients to metric measurements and recalculates the nutritional data.

CHAOTIC

Calculating amounts and formatting the recipe for printing are completely unrelated activities. This is a bad method!

The `GetPrintableVersion()` method accepts the number of servings as an argument, and then calculates the ingredient amounts and nutritional data and returns a formatted `FixedDocument` object.





PUT ON YOUR THINKING HAT

Here's a possible implementation of the chaotic method from the last exercise. Can you fix it?

```
Public Function GetPrintableVersion(servingsReqd As Integer) As FixedDocument
    'Update servings & data
    If servingsReqd <> Me.Servings Then
        Dim servingsModifier As Integer = Me.Servings / servingsReqd
        For Each ingr as RecipeItem in Me.Items
            ingr.Amount *= servingsModifier
            ingr.GetIngredientNutrition()
        Next ingr
    End If

    'create fixed document
    Dim fd as FixedDocument = New FixedDocument()
    fd.DocumentPaginator.PageSize = (96 * 8.5, 96 * 11)
    PageContent pc = CreateFormattedRecipe()
    fd.Pages.Add(pc)
    Return fd;
End Function
```

servingsModifier
contains the amount for a
single serving.

I'm cheating here because laying out a WPF
FixedDocument is pretty verbose, and you don't need
to understand it for the exercise. If you're interested, search
for "WPF Documents" in MSDN.

Here's the snippet that calls the method. You'll need to fix it, too:

```
myRecipe.GetPrintableVersion(6)
```



PUT ON YOUR THINKING HAT

Did you recognize the RecalculateRecipe() method from the first exercise?

```
Public Sub RecalculateRecipe(Integer servingsReqd)
    'update servings & data
    If servingsReqd <> Me.Servings Then
        Dim servingsModifier As Integer = Me.Servings /
servingsReqd
        For Each ingr As RecipeItem In Items
            ingr.Amount *= servingsModifier
            ingr.GetIngredientNutrition()
        Next ingr
    Then
End Sub

Public Function GetPrintableVersion() As FixedDocument
    Dim fd As FixedDocument = new FixedDocument()
    fd.DocumentPaginator.PageSize = (96 * 8.5, 96 * 11)
    Dim pc As PageContent = CreateFormattedRecipe()
    fd.Pages.Add(pc)
    Return fd
End Function
```

And here's the revised snippet that replaces the original method call:

```
myRecipe.RecalculateRecipe(6)
fd As FixedDocument = GetPrintableVersion()
```



REVIEW

Can you write a definition (in your own words, please) for each of the terms below and rank the levels of cohesion from most cohesive (1) to least cohesive (5)?

COMMUNICATIONAL

CHAOTIC

FUNCTIONAL

SEQUENTIAL

TEMPORAL



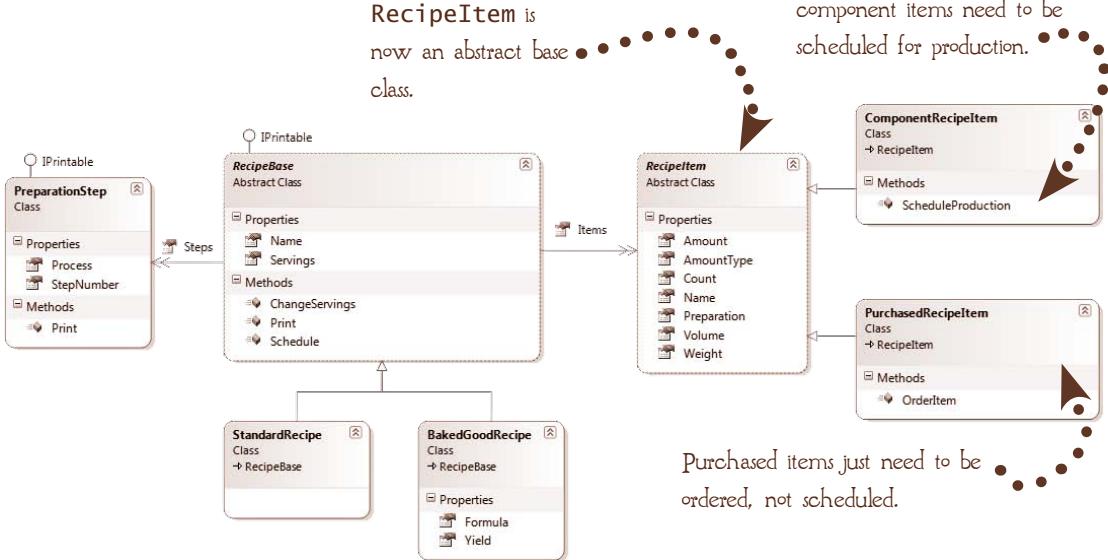
TAKE A BREAK

That's the first principle, SRP, in a nutshell. Why don't you take a break before we move on to the next one: the Open/Closed Principle?

Open/Closed

The usual definition of the Open/Closed Principle states that "software entities should be open for extension but closed for modification". Sounds great, but what does that mean, exactly? It means, essentially, that extending the functionality of an entity (class or member) shouldn't require changes to any other code, whether in that entity or another.

Like many principles, OCP can be easier to understand in its breach, so let's look at an example of code that isn't immune from changes. You might remember that some of the recipe components are made in-house, and others are purchased. So what happens when our chefs need more of something? Here's a version of our class diagram that reflects the two types of ingredients:





PUT ON YOUR THINKING HAT

On the face of it, the new class hierarchy isn't a bad design. The common properties are abstracted into a base class, and the behavior that's unique to each type of `RecipeItem` is represented by separate classes.

But what happens when we need to create a `Restock()` method that handles any kind of `RecipeItem`? Here's one way to handle it:

```
Public Sub Restock(ingr As Ingredient)
    If ingr Is ComponentRecipeItem Then
        ingr.ScheduleProduction()
    Else
        ingr.OrderItem()
    End If
End Sub
```

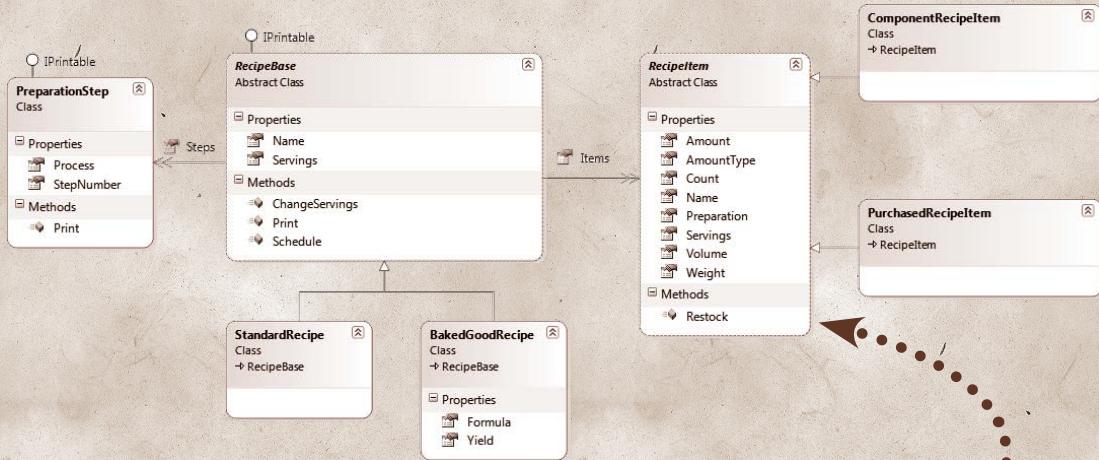
Again, on the face of it, this isn't bad code, and it will work just fine. But it violates the Open/Closed Principle. Can you see how? (Hint: What happens when you add a new type of `RecipeItem`?) How would you fix it?



HOW'D YOU DO?

The problem with the design is that extending the application by adding a new type of `RecipeItem` requires modifying the existing `Restock()` method. So while it's open for extension, it's also open for modification.

Did you think of a way to fix the problem? Here's one way:



The `Restock()` method has been moved to the abstract base class. (It could also have been an `Interface`, but semantically an abstract class seemed more reasonable. (You're free to disagree.) The important point is that each type of `RecipeItem` (including new ones) can implement the behavior as it sees fit, and that extending the behavior of the `RecipeItem` classes won't affect `RecipeBase`.

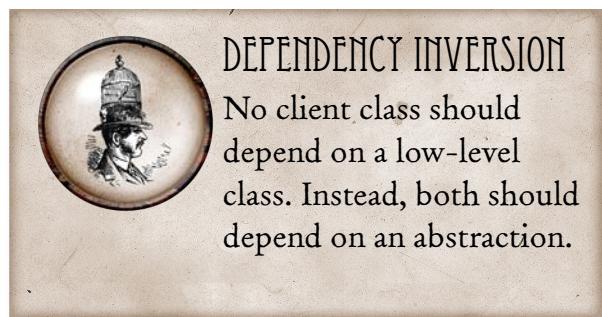


UNDER THE MICROSCOPE

At its most extreme, the OCP requires that you should be able to extend functionality without even recompiling the entity, which is possible using Dependency Inversion, but in reality not all that common because of the design complexity required.

DEPENDENCY INVERSION

DEPENDENCY INVERSION is one of the most effective ways of ensuring that your class hierarchies are open for extension but closed for modification. There are three basic guidelines for implementing dependency inversion. (They're sometimes described as "rules", but no design principles are, or should be, unbreakable, and these perhaps more than most!)



DEPENDENCY INVERSION

No client class should depend on a low-level class. Instead, both should depend on an abstraction.

1

BEWARE OF CLASS MEMBERS DEFINED AGAINST A CONCRETE CLASS

Whether you're considering properties or variables, be sensible about this one—it's impossible to implement completely. Somewhere or other, you have to create a `String` or `Integer` (which are concrete classes, after all) or you'll never be able to do anything! But implementing a concrete class makes you vulnerable to changes made to it, and that leads to fragile code, so... beware.

2

BEWARE OF INHERITING FROM A CONCRETE CLASS

Well, obviously, every .NET class descends ultimately from `Object`, which is concrete, but the intention of this rule is that you always implement interfaces rather than deriving subclasses. As with the first guideline, this needs to be applied with caution or you'll be losing the benefits of OOP, but it does make you think carefully about the role interfaces should play in your hierarchy.

3

BEWARE OF OVERRIDING INHERITED METHODS

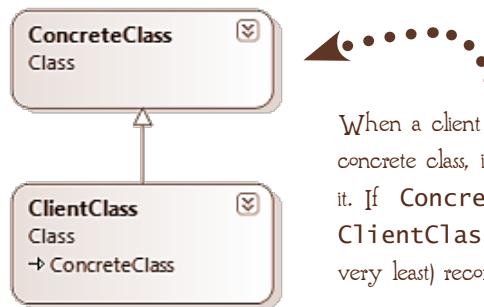
The theory behind this one is that if you're overriding the method of an abstract class, it wasn't really abstract to begin with. That is often true, and you should rethink your hierarchy. But if your abstract class is providing default behavior, it's sensible to override it where appropriate.

PROGRAMMING TO INTERFACES

The basic technique for implementing dependency inversion using interfaces is called PROGRAMMING TO INTERFACES, and it's always a good way to protect yourself (and your classes) from change. I've actually seen it described as "a fundamental principle of OOP". It's a good practice, but I personally doubt it's fundamental, not in the way that, say encapsulation is.

The thing to be careful of with programming to interfaces is a cluttered class hierarchy. You should beware of any interface that is only implemented by one or two concrete classes. It's probably just added complexity (otherwise known as "noise").

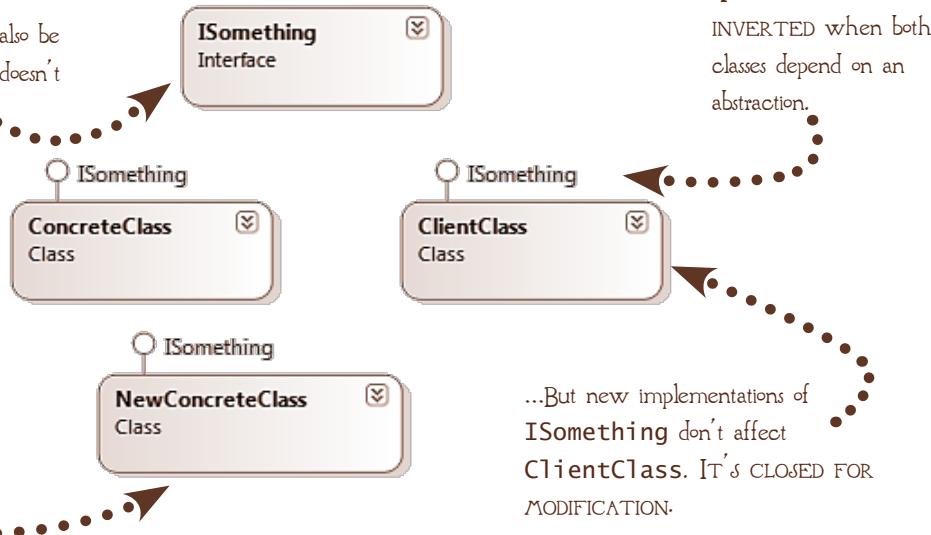
THIS IS HOW THINGS ARE USUALLY DONE:



When a client class instantiates a concrete class, it is dependent upon it. If **ConcreteClass** changes, **ClientClass** needs to be (at the very least) recompiled.

THE SAFE WAY:

ISomething could also be an abstract class; that doesn't break the pattern.



The DEPENDENCY IS INVERTED when both classes depend on an abstraction.

New classes implementing ISomething can extend the functionality of ClientClass...

...But new implementations of ISomething don't affect ClientClass. IT'S CLOSED FOR MODIFICATION.



PUT ON YOUR THINKING HAT

Knowing when to apply a design principle or pattern is just as important as knowing how, so here are some situations to think about. Assume that in each case your initial design has the `ClientClass` defining a property of the type `ConcreteClass`, as shown in the diagram. In which of the following situations do you think dependency inversion should be applied?



1

This is the first time your client (the person you're developing for, not the client class) has been involved in developing a computer application, and every time you show them a sketch of the user interface for `ConcreteClass`, they think of new variations.

2

You're building a new user interface for an existing application. The class hierarchy hasn't changed significantly since the application was first deployed.

3

You're designing a new application using the WPF platform, but the client has mentioned that they might want to move it to the Internet sometime in the future.

4

Your client is a product distributor. The existing application has separate forms for each product category.



MY VIEW

I've always thought that dependency inversion isn't very well named. (It sounds like the newest fitness fad, doesn't it? Or maybe something vaguely sexual. Or both.) The "dependency" part is clear enough, but how is it "inverted"? Well, it isn't, not really. More like displaced. When you implement dependency inversion, the low-level classes don't become dependent on the high-level classes, which is my understanding of the word inversion, but rather both objects become dependent on a third. Unfortunately, if you talk about "dependency displacement", nobody will know what you mean, so I guess we'll just have to accept the conventional name.



HOW'D YOU DO?

Remember, these are judgement calls. It doesn't matter if you made different decisions than I did. What matters is that you think through the issues involved.

1

This is the first time your client has been involved in developing a computer application, and every time you show them a sketch of the user interface for `ConcreteClass`, they think of new variations.

I'd implement dependency inversion here. It's hard to translate the real world into a computer application, and the client is almost certain to come up with additional changes once they start working with the initial release.

2

You're building a new user interface for an existing application. The class hierarchy hasn't changed significantly since the application was first deployed.

The class hierarchy has been stable for awhile, and there's no particular reason to think it will change in the future. I'd leave it as it is.

3

You're designing a new application using the WPF platform, but the client has mentioned that they might want to move it to the Internet sometime in the future.

This is a trick question. Designing a class hierarchy for multiple platforms can be tricky. It usually requires a lot more than simply implementing dependency inversion. Since "might" and "sometime in the future" are fairly vague, I'd probably assume a single UI for now.

4

Your client is a product distributor. The existing application has separate forms for each product category.

This is a classic example of a good place for the dependency inversion, and the multiple forms in the existing application are a good example of why. Whenever a client sells something, you can expect that "something" to change in time. That said, if you're building a "quick and dirty" to get them over a crisis while something more resilient is being designed, then do it quick and dirty. (You could figure that out for yourself, right?)

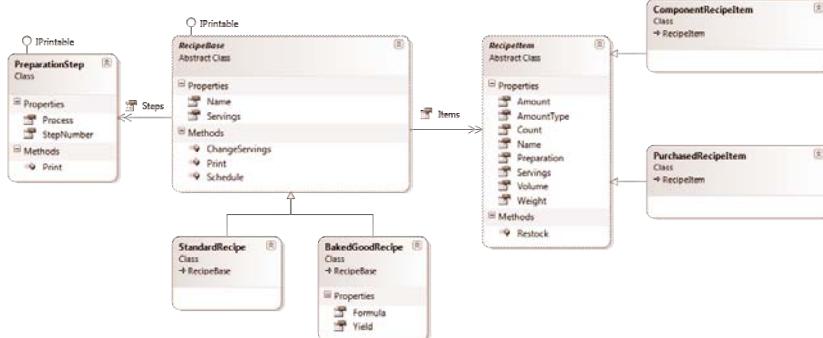
LISKOV SUBSTITUTION

The LISKOV SUBSTITUTION PRINCIPLE (LSP) is named for the woman who formalized it, Barbara Liskov. At first glance, the LSP seems trivial. By definition, subtypes in an object-oriented language like VB are substitutable for their base types; it's just part of the language.

But LSP isn't about the syntax, it's about the semantics. To understand what that means, let's look at our Recipe class hierarchy again.

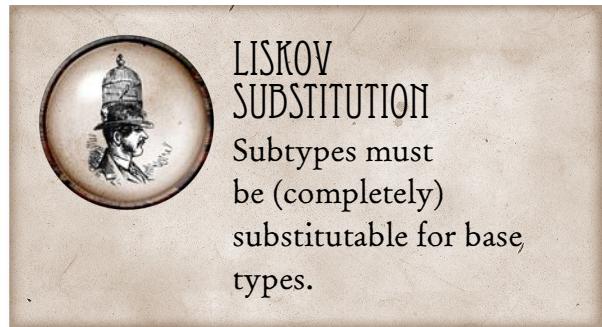
The ComponentRecipeItem and PurchasedRecipeItem types both inherit a Restock() method from their parent, RecipeItem. The two classes do very different things within the method, but from the point of view of a client class like RecipeBase, both methods mean the same thing: "organize to get some more of yourself in stock". They're semantically identical.

But look at the StandardRecipe and BakedGoodRecipe, which both inherit a Servings property



that doesn't tell you anything about how many people (or teenagers) it will feed. These two classes violate the LSP.

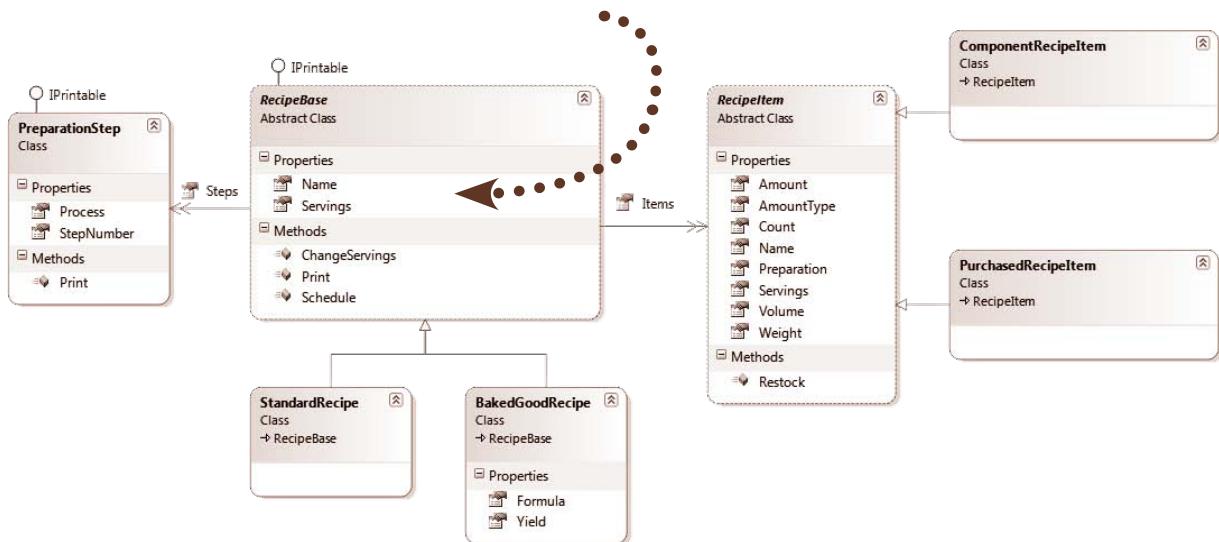
Why is that a bad thing? Because in order to understand the **Servings** property, the user has to know what type of **Recipe** it belongs to. Worse, the application needs to know what kind of **Recipe** it's manipulating, and applications are just way too stupid to do that.



IS A... REALLY?

The LSP is in many ways a refinement of the IsA relationship between classes. Unfortunately in programming, as in life, there are an infinite number of ways for something to be wrong, and there seems to be no end of ways that a subtype can be almost, but not quite, an instance of its parent. But here are a couple of the most common problems to look out for:

BEHAVIOR ISN'T TRULY POLYMORPHIC
This is the classic situation that we just discussed.



```
Sub ChangeServings(Integer desiredServings)
{
    'do nothing
}
```

DEGENERATE CLASSES

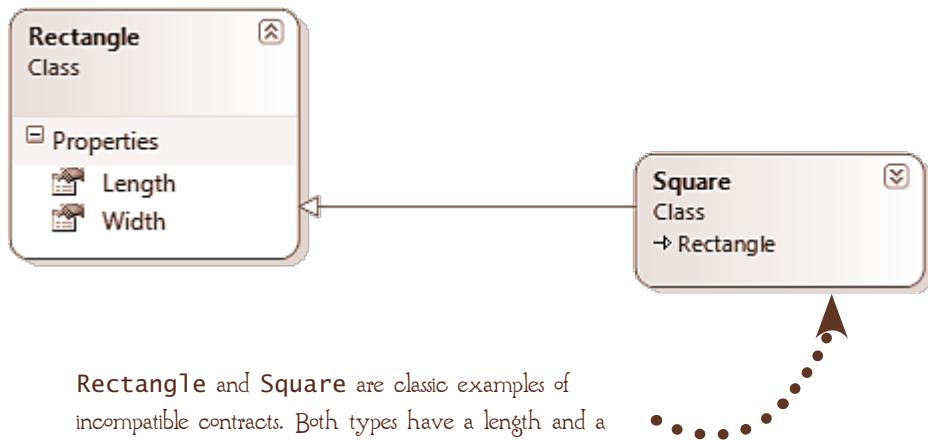
Because Servings has a different meaning, and also because baked goods can't just be doubled or halved (leavenings like yeast or baking powder don't work like that), **BakedGoodRecipe** overrides the implementation of **ChangeServings** to do nothing. This is called a degenerate class, and is usually symptomatic of a breach of LSP.

INCOMPATIBLE CONTRACTS

Classes have incompatible contracts if there are rules that apply to one but not the other (again, a semantic, not syntactic difference).

MAKE A NOTE

Another way of thinking about the LSP is that it doesn't matter that a property or method is meaningful for each of its subtypes; it has to have the **SAME** meaning.



Rectangle and **Square** are classic examples of incompatible contracts. Both types have a length and a width, but **Square** has an extra requirement: The length and width must be equal.

PUT ON YOUR THINKING HAT

We've seen at least two ways in which the **Recipe** class hierarchy violates LSP. How would you fix it?

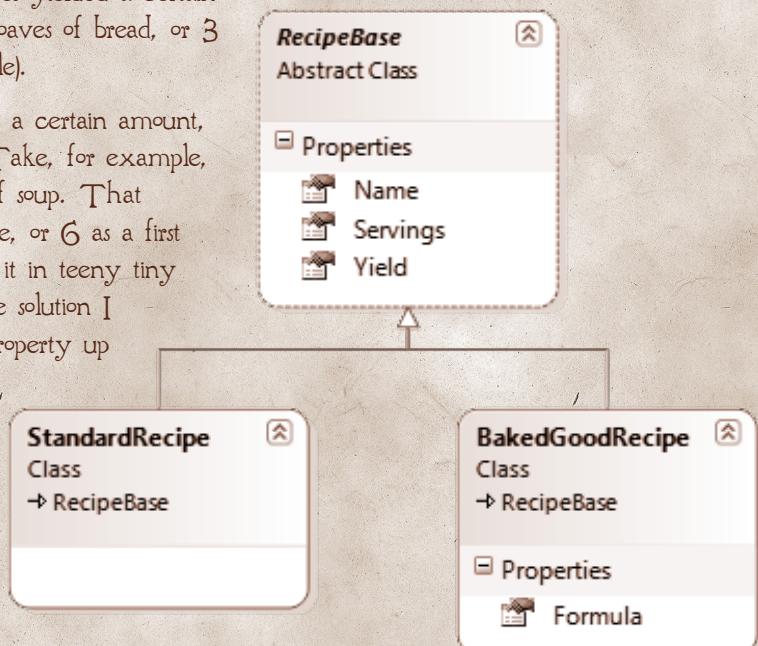


HOW'D YOU DO?



There are, of course, lots of ways to solve the problems with the Recipe hierarchy, but here's what I'd do: The problem is a semantic one, so I looked at the semantics of classes. Originally the Yield property belonged only to the BakedGoodRecipe class, the idea being that instead of a specific number of servings, these recipes yielded a certain number of items (2 loaves of bread, or 3 dozen cookies, for example).

But of course any recipe yields a certain amount, and that's useful information. Take, for example, a recipe that yields 2 quarts of soup. That might serve 4 as a main course, or 6 as a first course, or 12 if you're serving it in teeny tiny cups as part of a starter. So the solution I chose is to move the Yield property up to RecipeBase, so that all Recipes can now specify both Yield and Servings.



MY VIEW



The solution probably seems obvious, doesn't it? It does to me, in retrospect. You might even feel like I set you up. But here's the reality: I'm a dedicated cook, and I have a couple of decades' experience as an application designer, but it took me two days of pondering to come up with this solution. Lots of things are obvious in retrospect. Remember the paperclip?

LAW OF DEMETER

Unfortunately, the LAW OF DEMETER has nothing to do with the Goddess of the harvest. It's named for the Demeter Project at Northwestern University where it was developed. It's also known as the PRINCIPLE OF LEAST KNOWLEDGE (but that's not nearly as much fun, is it?)



LAW OF DEMETER

An object should only reference its own members, and the members of objects it instantiates directly.

Like the OCP, the Law of Demeter addresses the issue of coupling between classes. It states, essentially, that you should only talk to the objects you instantiate directly, not the objects they instantiate. Let's look at an example:

MyClass has a property
that's an instance of
MyFriend.



MyFriend has a property that's an instance of MyFriendsFriend.



According to the Law of Demeter, it would be legal to do any of these things in MyClass:

```
Dim aFriend = New MyFriend()
aFriend.aFriendlyProperty = 3
aFriend.aFriendsFriend = new aFriendsFriend()
```

but you shouldn't do this:

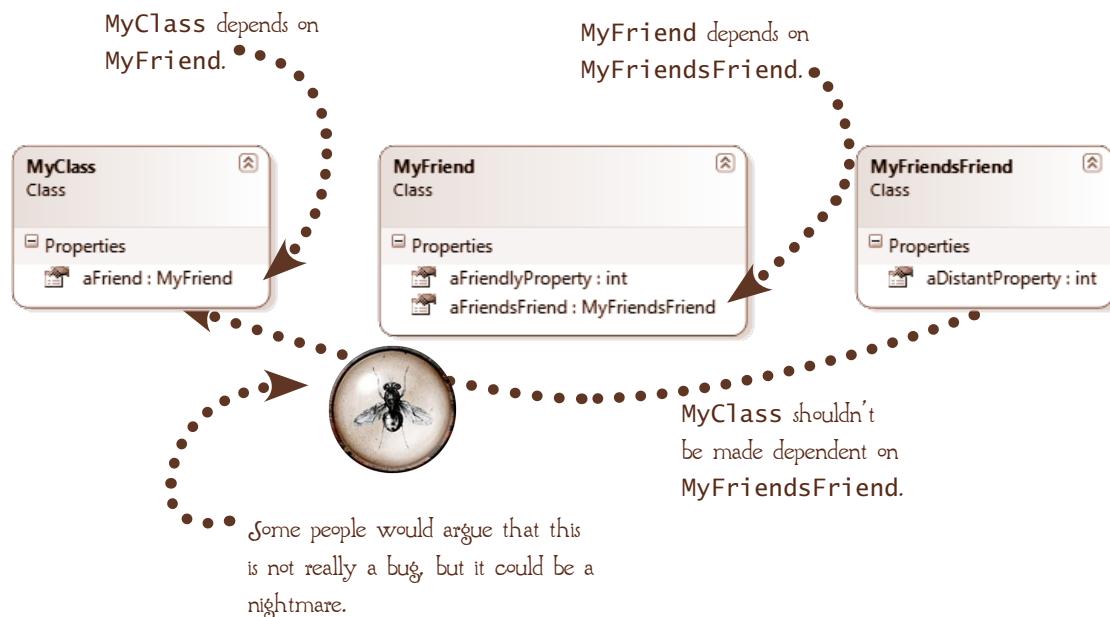
```
aFriend.aFriendsFriend.aDistantProperty = 5
```

AND THE PROBLEM IS...

...fragility, of course. Let's say that after the application has been in use for some time (long enough that you've forgotten the details of how it works—say a week), you discover that `aDistantProperty` really shouldn't be a single property at all; it should be a collection. That's a major change, and you know you'll have to update all the classes that instantiate `MyFriendsFriend`. Ugly, but it can't be helped.

Visual Studio has a handy tool that will help you out here; just highlight `aDistantProperty` and select Find All References from the context menu. But Find All References has some limitations: It won't work across languages (what if `MyClass` were written in C#?), across projects that target different versions of the .NET Framework, or across different Solutions. A useful tool, but not perfect.

So, the first problem is that you might not find the reference to `aDistantProperty` in `MyClass` until it breaks. And by the time it breaks, you may have moved on to your next project. And even if you do find that second-degree reference, it's increased the complexity of an already difficult task because whenever you make a significant change to functionality, you need to understand the class you're changing and all the classes that are affected by that change.



DEMETER DOS AND DON'TS

We've seen one example of a violation of the Law of Demeter, referencing the property of a property. (Got that?) But the principle (and like everything discussed in this chapter it *is* a principle, not a law) can be elaborated to some specific situations:

YOU CAN REFERENCE:

- The properties of your properties:

```
Me.aFriend.aFriendlyProperty = 3
```

- The properties of instance variables:

```
Dim aFriend = New aFriend()  
Me.aFriend.aFriendsFriend = New MyFriendsFriend()
```

- The properties of parameters:

```
Sub DoSomething(MyFriend x)  
    x.aFriendlyProperty = 4  
End Sub
```

- The properties of static classes:

```
Fonts.SystemFontFamilies
```

YOU SHOULDN'T REFERENCE:

- Anything else



PUT ON YOUR THINKING HAT

Looking back to the complete Recipe hierarchy shown on page 446, there are three things you can do when a class needs to reference `MyRecipe.Items(1).Amount`, which on the face of it is a violation the Law of Demeter. (Hint: One involves adding some behavior to the `MyRecipe` class.)

Can you think of the three possibilities?



HOW'D YOU DO?

Here are three ways I can think of to deal with the issue. Did you think of others?

1

INSTANTIATE A VARIABLE WITHIN THE REFERENCING CLASS

```
Sub DoSomething(RecipeBase recipe)
    Dim current As RecipeItem = recipe.Items[0];
    MessageBox.Show(current.Amount.ToString());
End Sub
```

Because `current` is instantiated within the method, it's not a violation of the Law of Demeter to reference its properties.

2

EXPOSE THE PROPERTY FROM RECIPEITEM

This can be done by defining a property that returns the second-hand property (but that's tricky when you're working with a collection property), or define a method that returns the property:

```
Public Class RecipeItem
    Function GetAmount(int itemNumber) As Integer
        Return Me.Items(itemNumber).Amount;
    End Function
End Class
```

This is a reasonable solution when you need to do something besides just reference the property.

3

IGNORE IT

Like any design principle, the Law of Demeter is something to consider, but not something that you absolutely have to obey. In other words, it's like the speed limit (breaking it can have serious consequences), not like the laws of thermodynamics (can't be broken this side of a sci-fi novel). Yes, violating the Law of Demeter can cause maintenance nightmares. But instantiating a variable for one line of code, as in our first example, seems to me to be just clutter (and doesn't really resolve the issue), and while the second example clearly *does* resolve the violation, it can clutter up your classes with a bunch of wrapper functions of limited use.



REVIEW

Let's start off easy. Can you match each of these definitions to the principle or strategy it describes?

Single Responsibility

A class should be open for extension but closed for modification.

Degenerate Class

No client class should depend on a low-level class. Instead, both should depend on an abstraction.

Dependency Inversion

This phrase is used to describe a class that overrides one of the methods it inherited from its base with an empty method.

Open/Closed Principle

A class should have only one reason to change.

Liskov Substitution

An object should only reference its own members and the members of objects it instantiates directly.

Levels of Cohesion

Subtypes must be (completely) substitutable for base types.

Law of Demeter

This is a useful way of thinking about how well a method complies with the Single Responsibility Principle.



MORE REVIEW

Here are some code snippets that violate one or more of the principles discussed in this chapter. Can you identify which principle each violates and come up with a solution? (Assume, for the sake of the exercise, that the solution is warranted.)

```
Public Overrides Sub DoSomething()  
End Sub
```

```
Public Class MyClass  
    Public Property BakedGoodsRecipe As Recipe  
End Class
```

```
Public Sub DoSomething()  
    DisplayCurrentTime()  
    CalculateSalary()  
    UpdateCustomer()  
End Sub
```

```
Public Sub DoSomething()  
    CurrentValue As Integer = Me.CurrentFriend.FriendsFriend.DistantProperty  
End Sub
```

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a C# programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



PATTERNS

14

So what, exactly, is a design pattern? Basically, it's a solution to a common problem. Sometimes patterns are expressed as class diagrams, but they're more often a diagram that describes a "who does what" scenario at a fairly high level.

The idea of a design pattern was first developed by the architect Christopher Alexander, whose books *The Timeless Way of Building* and *A Pattern Language* proposed patterns for common architectural problems. It was applied to software engineering by Gamma, Helm, Johnson and Vlissides (familiarly known as the Gang of Four) in their influential *Design Patterns: Elements of Reusable Object-Oriented Software*.

In the last chapter we looked at a few of the principles that theoreticians have proposed for designing better, more robust software. Principles, of course, apply generally to any programming situation. Patterns, on the other hand, are general solutions to specific problems. In this chapter we'll tackle a few of the patterns that are important in .NET development and also look at some software architectures that aren't technically patterns, but behave a bit like them.

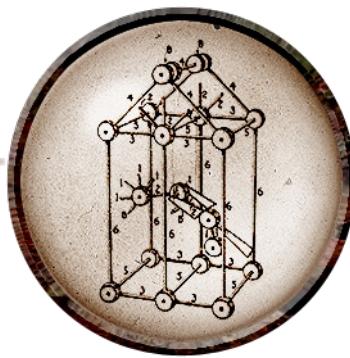
The elements of this language are entities called patterns. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

-Christopher Alexander,
A Pattern Language

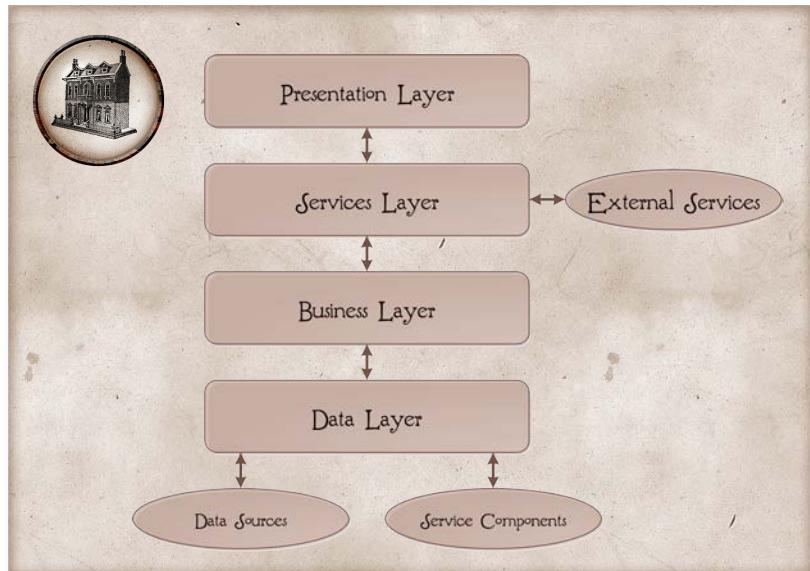


FITTING IT IN

In the last chapter we'll complete our exploration of the principles of software development by examining some of the most important design patterns, and also take a quick look at the architectural patterns that you can use to organize complex applications.



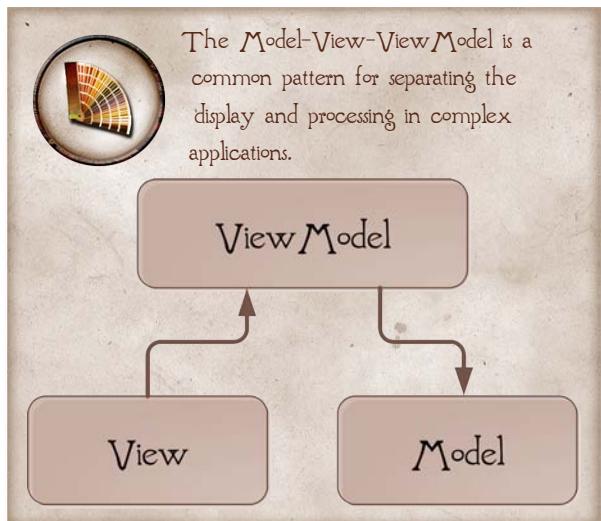
Architectural patterns define how the different parts of an application work together.



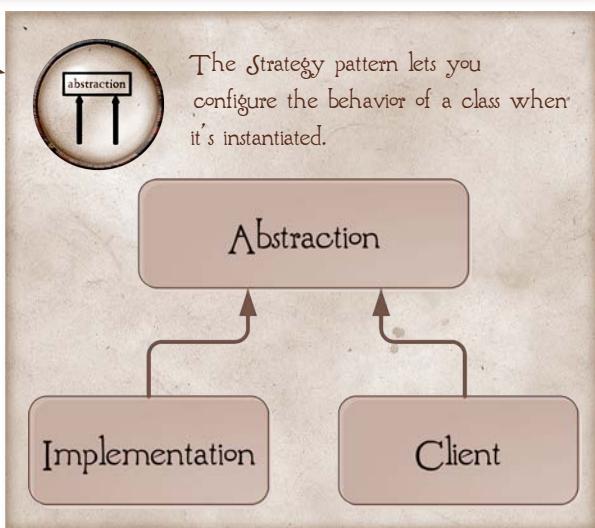
The Observer pattern is used for .NET events.



The Model-View-ViewModel is a common pattern for separating the display and processing in complex applications.



The Strategy pattern lets you configure the behavior of a class when it's instantiated.



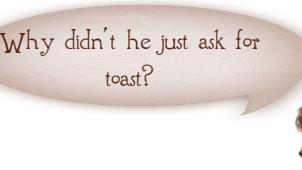
WHY PATTERNS?

- PATTERNS ARE PROBLEMS SOMEBODY ELSE HAS ALREADY SOLVED.



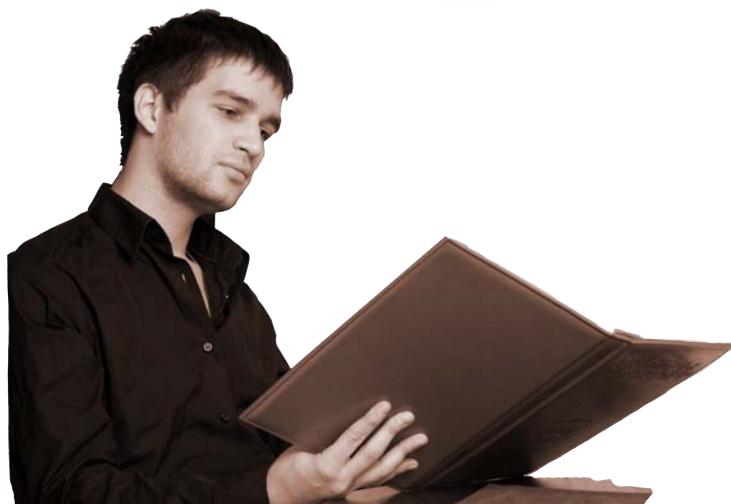
When I'm developing a new recipe, it may look like I'm just randomly throwing things in, but whether it's a chowder or a new kind of fruit pie, there's always a classic technique that I use to structure the dish.

- PATTERNS ARE STRUCTURES YOU CAN WORK WITHIN TO SOLVE YOUR PROGRAMMING PROBLEMS.



PATTERNS GIVE YOU A WAY TO TALK ABOUT YOUR PROGRAMS.

Why didn't he just ask for toast?



I'd like you to take a slice of bread and put it in front of direct heat until it's just lightly brown. Then turn it over and do the same thing on the other side. And could I have some butter and jam to go with it, please?





TASK LIST

There are many more design principles than the few we looked at in the last chapter, and there are more design patterns than we'll be able to cover here. (Just do a quick search for "design pattern" in your favorite search engine, or check out Microsoft's Patterns & Practices Web site.) I've picked a few that you'll see a lot when you're working with the .NET Framework. They should give you a taste for what design patterns are and how they work.



THE STRATEGY PATTERN

We'll start this chapter off by exploring the Strategy pattern (which looks a lot like programming to interfaces) and how it can help us refine our type hierarchies.



THE OBSERVER PATTERN

Next we'll examine observers, subjects, and the events that bind them. The Observer pattern is one of the most important in the .NET platform.



ARCHITECTURAL PATTERNS

Design patterns typically address problems in the way that objects interact. Architectural patterns, while not technically design patterns, address similar types of problems with the way application components interact. We'll examine the Layered Architecture, one of the most sophisticated and flexible, in some detail.



PRESENTATION PATTERNS

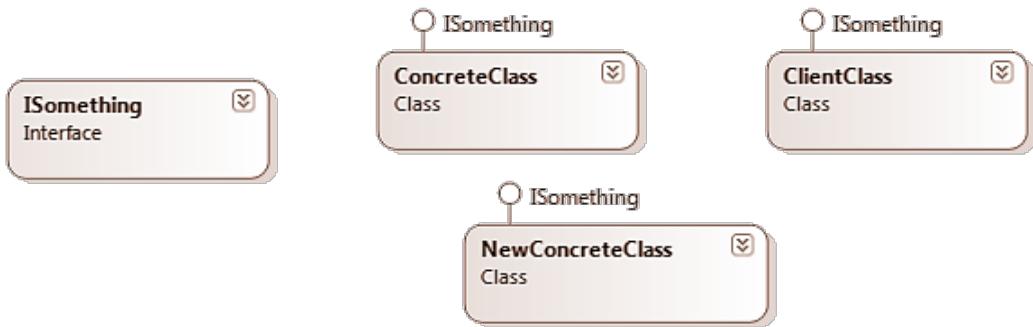
Finally, we'll return to design patterns with a look at the Model-View based patterns that are used to structure the user interface.



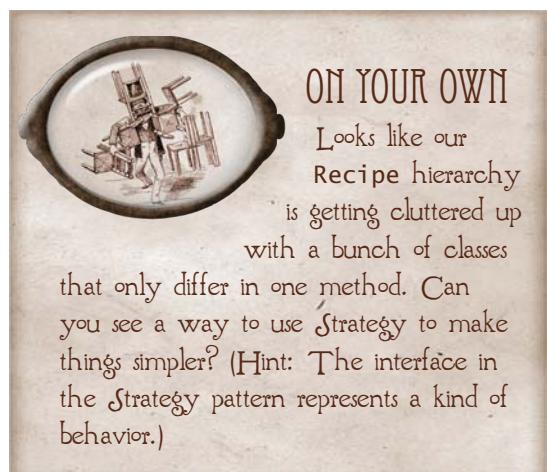
THE STRATEGY PATTERN

I have a surprise for you. (I hope it's a good one.) You already know the first of the design patterns we're going to be exploring.

When we were examining DEPENDENCY INVERSION in the last chapter, we saw that it's often implemented by a technique called "programming to interfaces", which looks like this:

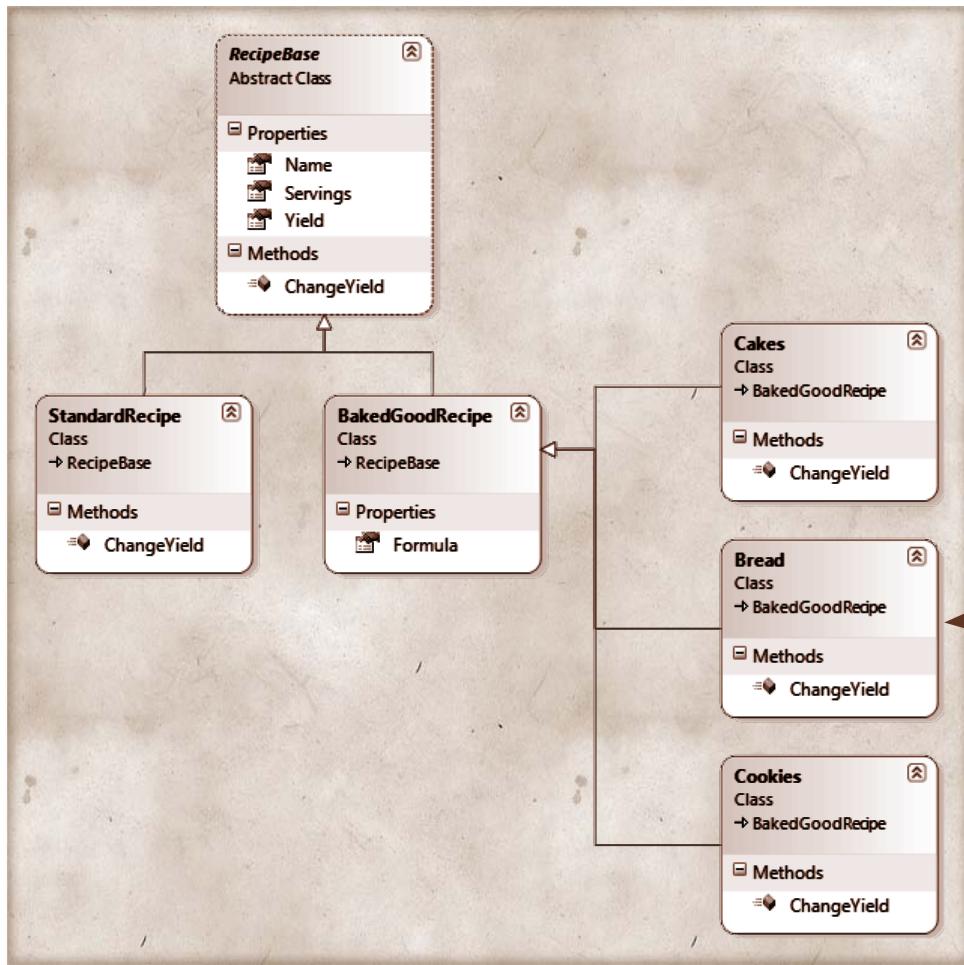


That's the basic structure of the STRATEGY PATTERN. There is one little twist, though: The pattern is technically only the STRATEGY when it represents some behavior of the ClientClass (not just an object the ClientClass works with).



USING STRATEGY

We've learned from our chefs that different categories of baked goods change their yield using different methods. If we use inheritance for all the different possibilities, we're going to have a problem:

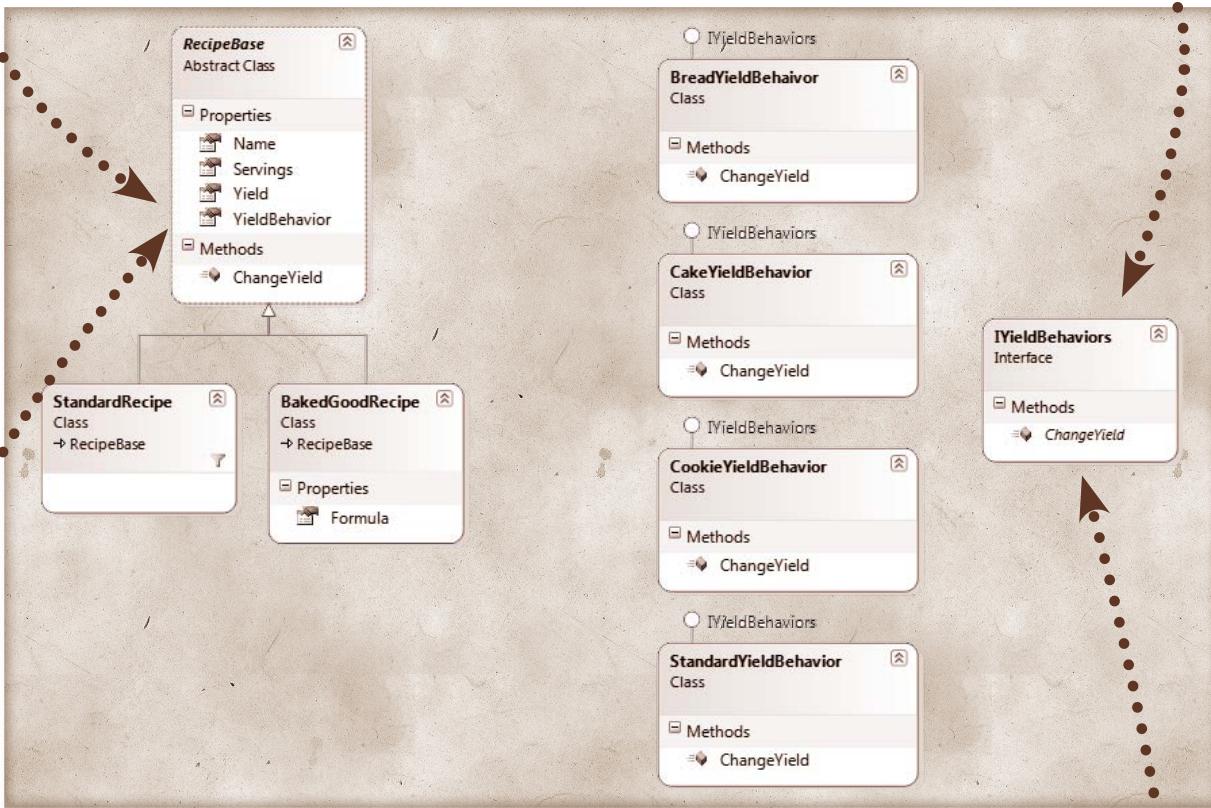


Using classical inheritance, we'd create a lot of classes that differ only slightly. It's easy to see how the class hierarchy could get really cluttered really quickly (there are different categories of cakes and bread, for example, each of which would need a different class).

The other problem here is that if the semantics of the **ChangeYield()** function changes (say, for example, you decide that you need another parameter), you have to fix it in lots of different places. That's potentially a lot of repetitive work. Who needs that?

Using the Strategy pattern, we still have as many classes (in fact, we have an extra type), but we gain two things: flexibility and a simple mechanism for reusing behaviors.

- The `RecipeBase` class now has a property called `YieldBehavior` that takes an instance of `IYieldBehavior`. Because it's declared in the base, it will be inherited by each of the concrete recipe types.



- The classes that inherit from `RecipeBase` can decide at runtime what their `YieldBehavior` is. The appropriate behavior can be assigned in the constructor and changed at any time.

Initially we've added an extra interface class to the hierarchy, so it probably doesn't seem any simpler. But think about what happens when, say cupcakes need to reuse the `CakeYieldBehavior`? No new code, just an assignment. That's the real power of the *STRATEGY* pattern.

STRATEGY DEFINED



STRATEGY

Define a family of algorithms, encapsulate each one, and make them interchangeable. Allow the algorithm to vary independently from the clients using it.

THE PROS

- Code reuse: Multiple types can use the same algorithm.
- Simplified hierarchy: Strategy eliminates classes that vary only by behavior.
- Runtime flexibility: Strategy allows objects to change their behavior at runtime.
- Simpler maintenance: If an algorithm changes, it need only be changed in the behavior class that represents it.

THE CONS

- If algorithms aren't being changed or reused, Strategy can add more complexity to the class hierarchy than it removes.



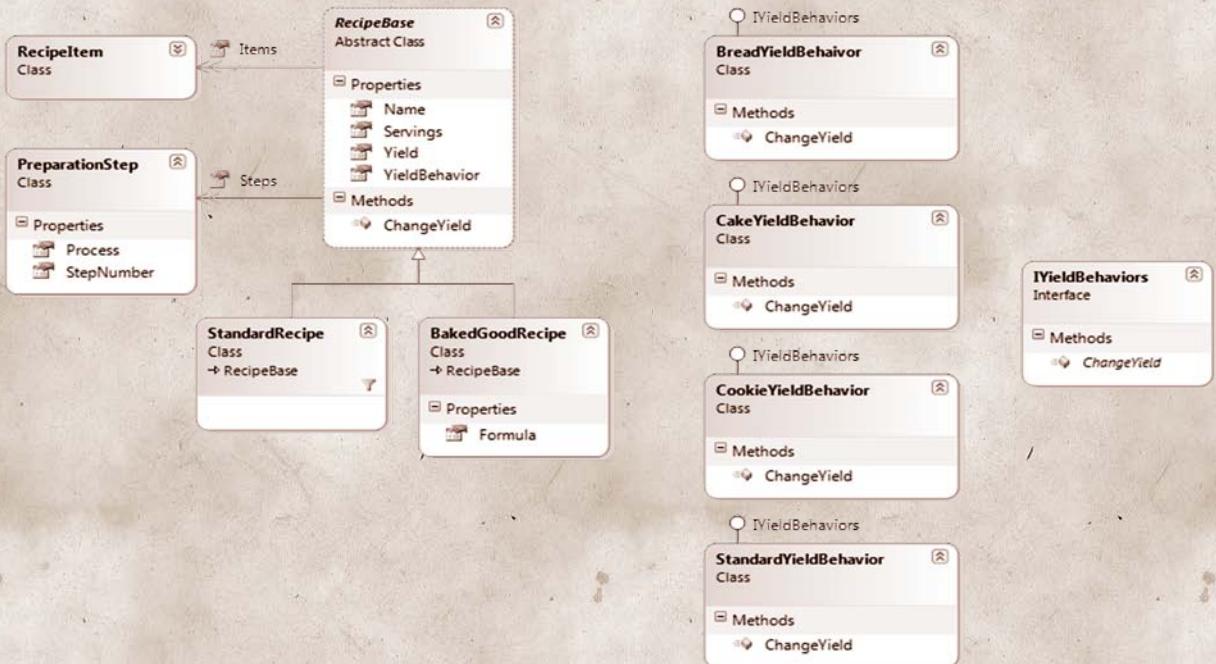
TAKE A BREAK

The Strategy pattern is pretty simple once you understand it, but it's fundamental to modern OOA&D. Why don't you take a break before completing the exercise and we move on to a pattern that's very important in the .NET Framework: the Observer.



PUT ON YOUR THINKING HAT

Most recipes share the same preparation method, so the `PreparationSteps` collection in our recipe hierarchy could also benefit from the Strategy pattern. Can you change the hierarchy so that recipes have a `PreparationMethod`. (For the exercise, create a method class for soup and one for bread.)

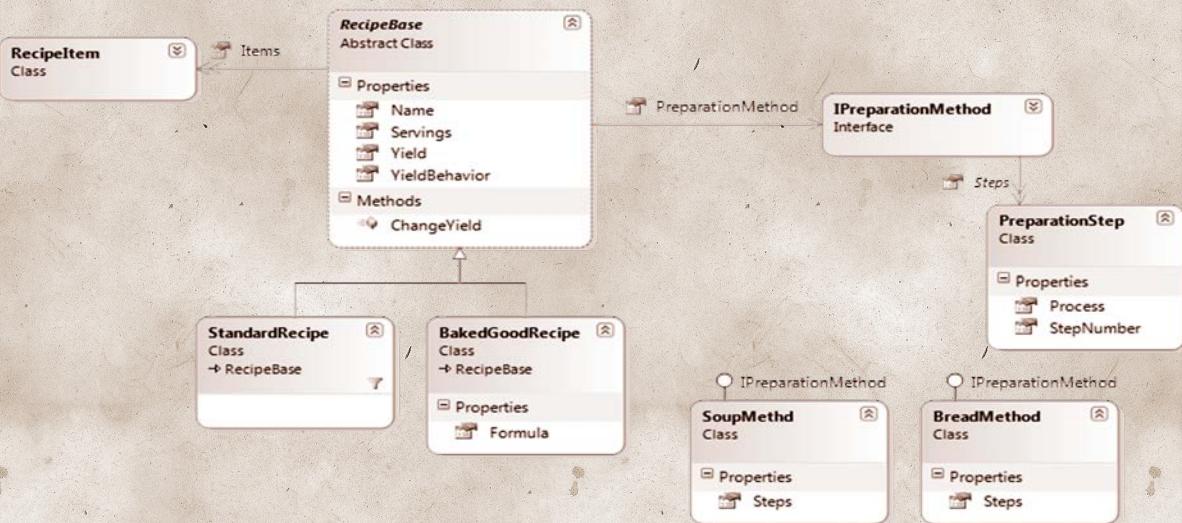


Write a constructor for `BakedGoodRecipe` that uses the preparation method class for bread.



HOW'D YOU DO?

It doesn't matter if you chose different names for the objects in your hierarchy.



```
public void BakedGoodRecipe()
{
    this.PreparationMethod = new BreadMethod();
}
```



MAKE A NOTE

Strategy is useful in three situations:

- Multiple classes implement the same behavior.
- Sibling classes vary only by behavior.
- A class can change its behavior at runtime.

THE OBSERVER

Think about the way a restaurant works:

How do you think you could translate this to a class hierarchy?

The kitchen CanDo the things listed on the menu.



The customer lets the kitchen know what to do by telling the waiter, who passes the order on to

the kitchen



The customer wants one or more of the items on the menu.



When the order is ready,
the kitchen tells the waiter,
who delivers it to the customer.

Now, the interesting thing here (from an OOA&D point of view, which is quite different from that of a restaurant critic) is that the kitchen doesn't know about the customer, and the customer doesn't know much about the kitchen except what's on the menu. The waiter knows about the kitchen and the customer, but doesn't need to know how the food is prepared.



The customer only knows what the kitchen can do but needs to know when their food is ready.



The kitchen doesn't know anything about customers, it just does stuff and lets the waiter know when it's done.

THE LOGICAL OBSERVER

Some design patterns, like the Strategy pattern we've been exploring, do seem to imply a canonical structure. The **OBSERVER**, on the other hand, is a pattern that's more about the way objects interact than about the way they're structured, and there are several ways to implement it.

Let's start by looking at the interactions the Observer pattern represents:

Logically, the Observer pattern is quite simple. One class, called the **Observer**, wants to know about changes in another class, the **Subject**, just as our restaurant customer wants to know about a change in the kitchen (the food's ready).



This actually requires three different processes:



REGISTRATION

The Observer tells the Subject that it's interested in the changes.

I'd like a steak. ①



NOTIFICATION

The Subject notifies all the Observers that the change has occurred.



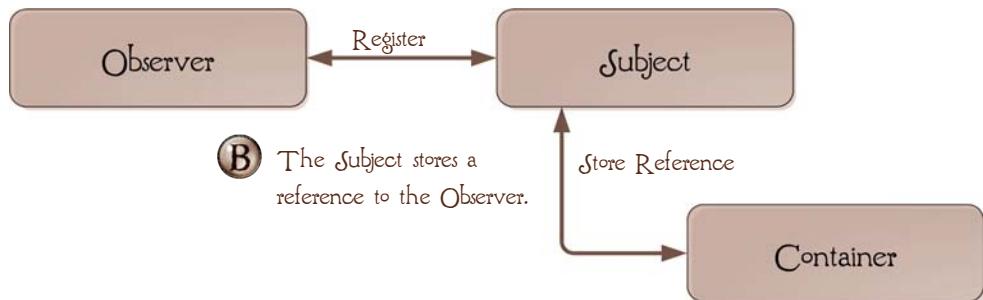
UNREGISTRATION

The Observer tells the Subject that it's no longer interested.

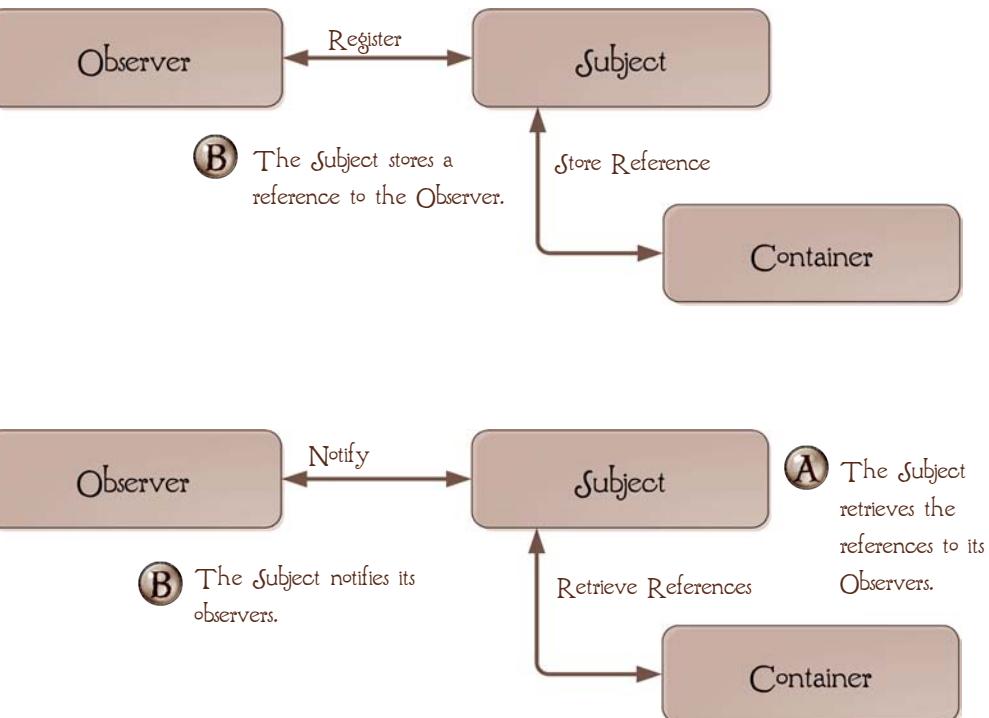


I

- A** The Observer notifies the Subject of its interest.

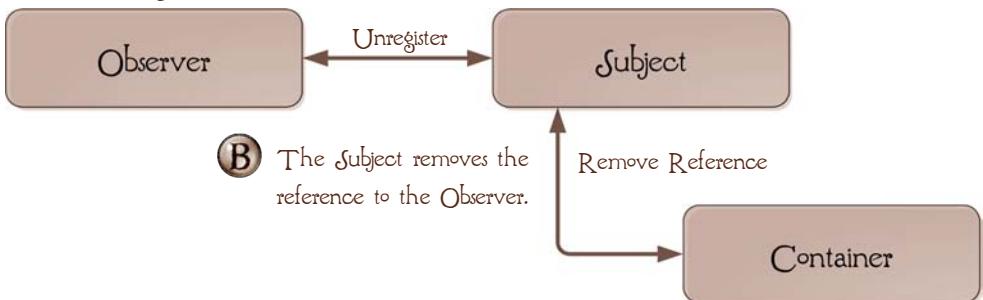


2



3

- A** The Observer notifies the Subject that it is no longer interested.



OBSERVER OPTIONS

There are three standard ways to implement the Observer pattern. Don't worry about how they're implemented; .NET has a built-in solution, which we'll look at in a page or two.

THE OBVIOUS SOLUTION

1

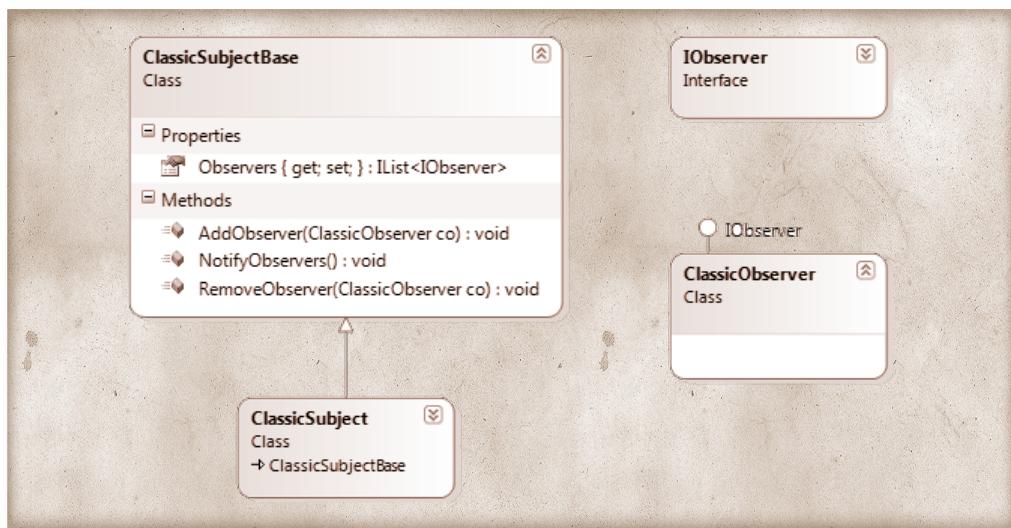
The most obvious way to solve the problem is to store a collection of observers in the subject.



THE CLASSIC SOLUTION

2

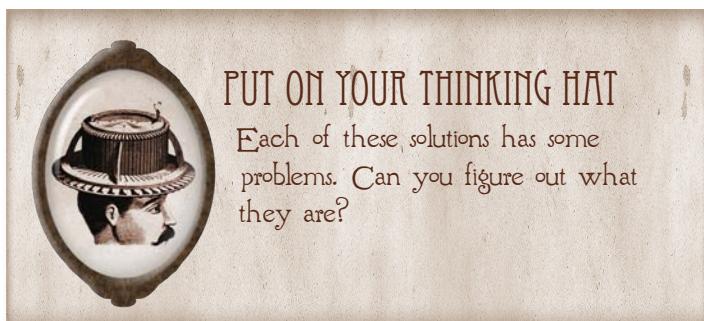
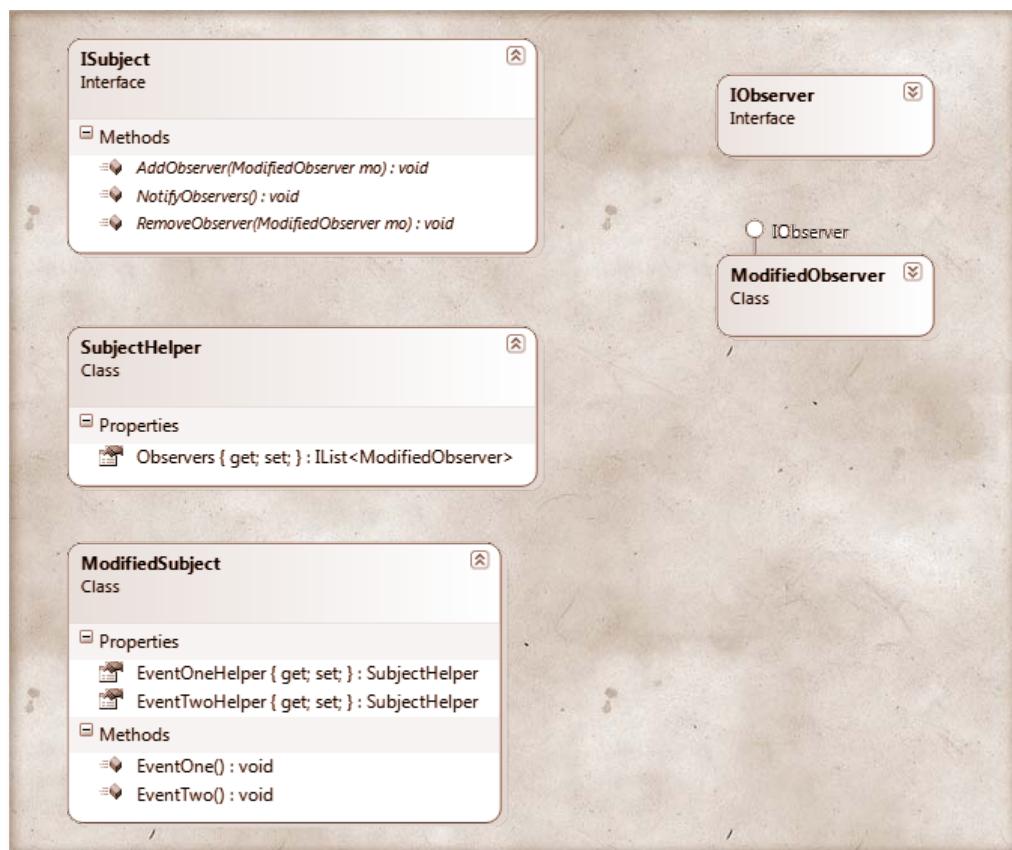
The classic solution described by the Gang of Four decouples the subject from the observer by using an interface and having all subjects derive from a base class.





THE MODIFIED SOLUTION

The Modified Observer pattern is most often used in languages (like VB) that only support single inheritance.



PUT ON YOUR THINKING HAT

Each of these solutions has some problems. Can you figure out what they are?



HOW' YOU DO?

This was a hard one. Don't feel bad if you missed some of the problems. I probably missed some, too.

THE OBVIOUS SOLUTION

The basic problem here is that the classes are too tightly coupled.

- Only instances of `ObviousObserver` can be notified.
- Any instance that cares about one change will be notified of all changes.
- It isn't possible to reuse one class without including the other.

THE CLASSIC SOLUTION

The classic solution solves the problem of requiring that the observer be of a specific type, but it has problems of its own.

- All subjects need to descend from `ClassicSubjectBase`. That complicates the class hierarchy when only single inheritance is supported.
- Any instance that cares about one change will be notified of all changes.
- The code to implement this solution is complicated, and it's rarely obvious how it works. That makes the classes hard to understand (and therefore, to maintain).

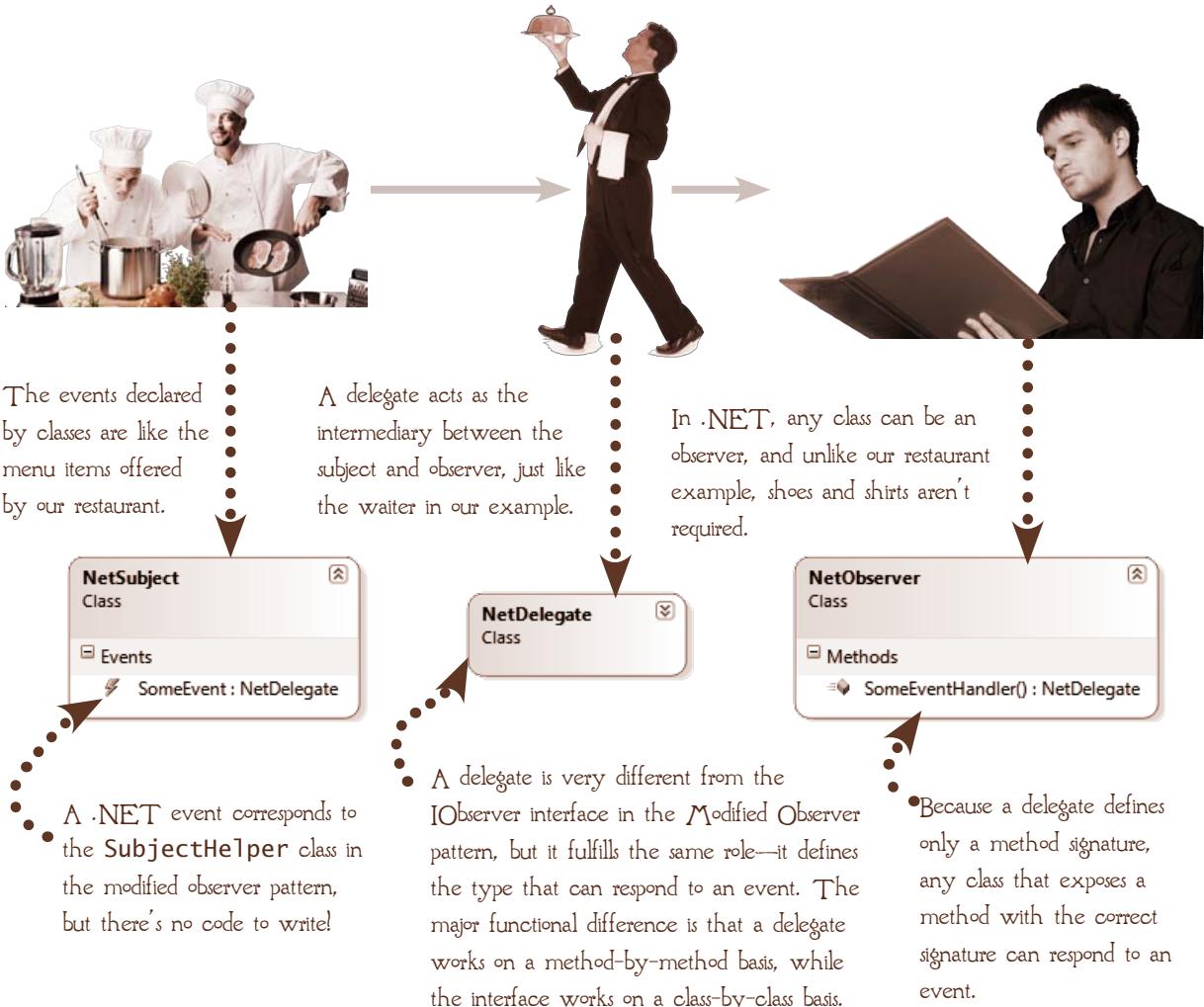
THE MODIFIED SOLUTION

The Modified Observer pattern is most often used in languages (like VB) that only support single inheritance. It solves most of the problems in the obvious solution, but some remain.

- The pattern has moved from the two simple classes in the obvious solution to three classes and two interfaces. That's a lot of complexity.
- The code is still fairly complicated and hard to understand, and there's more of it (although the interfaces and `SubjectHelper` class only need to be written once).

THE .NET SOLUTION

The Modified Observer pattern solves the major problems posed by the Observer, but it's complicated and can be tricky to implement. Fortunately, .NET (along with some other development platforms) implements some constructs that make implementation quite straightforward. Let's see how the .NET solution maps to our original example:



EVENT OBJECTS

Defining an event is a whole lot easier than implementing any of the Observer patterns directly, but it does require four things.

AN ARGUMENT CLASS

The second argument passed to an event handler is an instance of `System.EventArgs` or a class that descends from it. `EventArgs` itself doesn't pass any information, but the Framework defines dozens of classes that derive from it (everything from `AccessKeyEventArgs` to `XsltMessageEncounteredEventArgs`). So if you want to pass some information, you might be able to use one of them, or you can define your own class that inherits `System.EventArgs`. By convention, the argument class should be named `<EventName>EventArgs`.

AN EVENT DELEGATE

If you're using an `EventArgs` class that's defined by the Framework, there will be a delegate defined for it as well. The handler will be named `<EventName>Handler`, and you should follow the same convention when you create a delegate for a new argument type.

The delegate defines the signature of methods that can respond to the event. All delegates have the same basic pattern: they are declared as a `Public Sub` and have two arguments. The first argument is an `Object` named `sender` and represents the instance that is raising the event. The second argument is named `e` and represents an instance of the appropriate event argument class.

AN EVENT DECLARATION

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

```
Public Event <EventName> As <DelegateName>
```

(VB also supports another syntax, a CUSTOM EVENT, but we'll talk about that in Chapter 19.)

A METHOD TO RAISE THE EVENT

Finally, you need a method that will actually raise the event. By convention the event-raising method is named `On<EventName>`, is protected and virtual, and takes the event arguments as a parameter. Inside the handler, you use the `RaiseEvent` keyword with the event name, passing the appropriate arguments:

```
Protected Overridable Sub On<EventName>(<EventName>Args e) ,  
    RaiseEvent <EventName>(Me, e)  
End Sub
```

HANDLING EVENTS STATICALLY

VB supports two ways for a class to subscribe to an event. Static handlers are created using the `Handles` keyword and are registered for the lifetime of the class. This is the technique Visual Studio uses when you create an event handler via a Designer property window. The other technique, which uses the `AddHandler` and `RemoveHandler` keywords, allows you to add and remove event handlers dynamically at runtime. We'll look at dynamic event handlers in a bit. Let's start with what we've been doing all along.



DECLARE THE VARIABLE USING `WithEvents`

When you use WPF as we've been doing, you don't need to explicitly create variables to represent the objects in your UI (although you can). But if you're creating them in code, you need to use the `WithEvents` keyword:

```
Dim WithEvents MyButton As Button
```



REGISTER THE EVENT HANDLER USING THE `HANDLES` KEYWORD

The `Handles` keyword comes at the end of the handler definition and is followed by one or more events:

```
Public Sub <Instance>_<Event>([args]) Handles <event>[, <event>]  
    'do something  
End Sub
```



```
Public Sub MyButton_Click(ByVal sender As Object, _  
    ByVal e as RoutedEventArgs) Handles MyButton.Click
```

By convention, the handler is named `<Instance>_<Event>`.



PUT ON YOUR THINKING HAT

You wouldn't ordinarily handle an event in the same class that raises it, but let's do it just to get some experience writing events and handlers. You can declare the event (use the base `EventArgs` argument type) and the handler (have it display a `MessageBox`) in the `MainWindow` class, and then call the event-raising method from inside a `Button_Click` event handler.



HOW'D YOU DO?

Remember, you wouldn't normally put all the pieces in one file like this...

```
Public Partial Class MainWindow  
    Inherits Window
```

```
    Public Sub New()  
        InitializeComponent()  
    End Sub
```

Here's the event declaration.

```
    Public Event MyEvent As EventHandler
```

```
Protected Overrides Sub OnMyEvent(ByVal e As EventArgs)  
    RaiseEvent myEvent(Me, e)  
End Sub
```

```
Public Sub MyHandler(Object sender, EventArgs e) Handles Me.MyEvent  
    MessageBox.Show("Hey; the event worked!")  
End Sub
```

```
Private Sub button1_Click(ByVal sender AS Object,  
    ByVal e As EventArgs) Handles button1.Click  
    OnMyEvent(new EventArgs())  
End Sub
```

```
End Class
```

Creating a single method to raise the event makes it easier to be consistent about how you handle it. It's common, for example, to actually raise two events when a property changes: one before the property is changed and one after. You'd raise both of them in this `On<Property>Change` method.

The `Handles` keyword associates the method with the events and objects specified.

DYNAMIC EVENT HANDLERS

When you use the `Handles` keyword to associate a method with an event raised on an object instance, the association exists for the life of the class. They're called **STATIC METHODS**, although they're not quite static in the sense of a static member. They're static because they don't change. The alternative is to add and remove handlers at runtime.

Why would you want to do this? Well, honestly, you won't very often. Sometimes you're creating (or changing) your UI on the fly, so there's no object to specify in the `Handles` clause. Or, theoretically, you might want to either change the method that responds to the event or only have the method respond sometimes, but not always. (Theoretically. I've personally never seen it, but that doesn't mean there aren't programs and programmers that do this for every good reason.) In any event, the syntax is straightforward.

TO ADD AN EVENT HANDLER

Use the `AddHandler` statement, passing it the event (on a specific object) and the address of the handler:

```
AddHandler <Object>.<Event>, AddressOf <Handler>
```

```
    AddHandler Button1.Click, AddressOf Me.ButtonClickHandler
```

TO REMOVE AN EVENT HANDLER

The syntax is almost identical, but use the `RemoveHandler` statement:

```
RemoveHandler <Object>.<Event>, AddressOf <Handler>
```

```
    AddHandler Button1.Click, AddressOf Me.ButtonClickHandler
```



PUT ON YOUR THINKING HAT

Rewrite the last exercise to add the `Me.MyEvent` handler dynamically instead of statically. You can call the `AddHandler` method inside the `New()` method.



HOW'D YOU DO?

The changes are pretty simple.

```
Public Partial Class MainWindow  
    Inherits Window  
  
    Public Sub New()  
        InitializeComponent()  
        AddHandler Me.MyEvent, AddressOf MyHandler  
    End Sub  
  
    Public Event MyEvent As EventHandler  
  
    Protected Overridable Sub OnMyEvent(ByVal e As EventArgs)  
        RaiseEvent myEvent(Me, e)  
    End Sub  
  
    Public Sub MyHandler(Object sender, EventArgs e)  
        MessageBox.Show("Hey, the event worked!")  
    End Sub  
  
    Private Sub button1_Click(ByVal sender AS Object,  
        ByVal e As EventArgs) Handles button1.Click  
        OnMyEvent(new EventArgs())  
    End Sub  
End Class
```

••• AddHandler replaces the
Handles clause, so...

...be sure you remove
it from the method
declaration.



REVIEW

What problem does the Observer pattern address?

The event pattern in .NET provides built-in support for which version of the Observer pattern?

How do the classic and modified versions of the Observer pattern differ?

Under what circumstances would you create a custom event argument class? What about a custom delegate type?

What's the purpose of the `On<Event>` method? Why is it a good idea to centralize this behavior in a single method?



ARCHITECTURES

Application architectures aren't technically design patterns, but they serve the same purpose: They are accepted ways of organizing your application that address particular situations. The difference is that while design patterns typically address the interaction of objects within an application, application architectures address larger components: the various executables, libraries and services that interact to provide your application's functionality.

MONOLITHIC APPLICATIONS

By now you've probably gotten the idea that the .NET Framework provides a lot of functionality. You're right, it does, enough that a lot of applications don't need anything else. There's absolutely nothing wrong with this design, but even when everything's going to be compiled into a single executable, it's useful to think about your application as a series of roles (displaying things to the user, manipulating data, and so forth) and organize the code accordingly. We'll look at one way to do this in the next section.

CLIENT/SERVER APPLICATIONS

A special category of N-tiered applications is one that talks to some kind of back-end service like a database engine or a web service. That back end isn't part of your application and may not be (in fact, usually isn't) part of the development project.

The Framework servers like SQL Server and SharePoint Server are often used in this situation, and the client application (the bit you're writing) often provides an interface to whatever the server provides in a form that's more comfortable to your users.

N-TIER APPLICATIONS

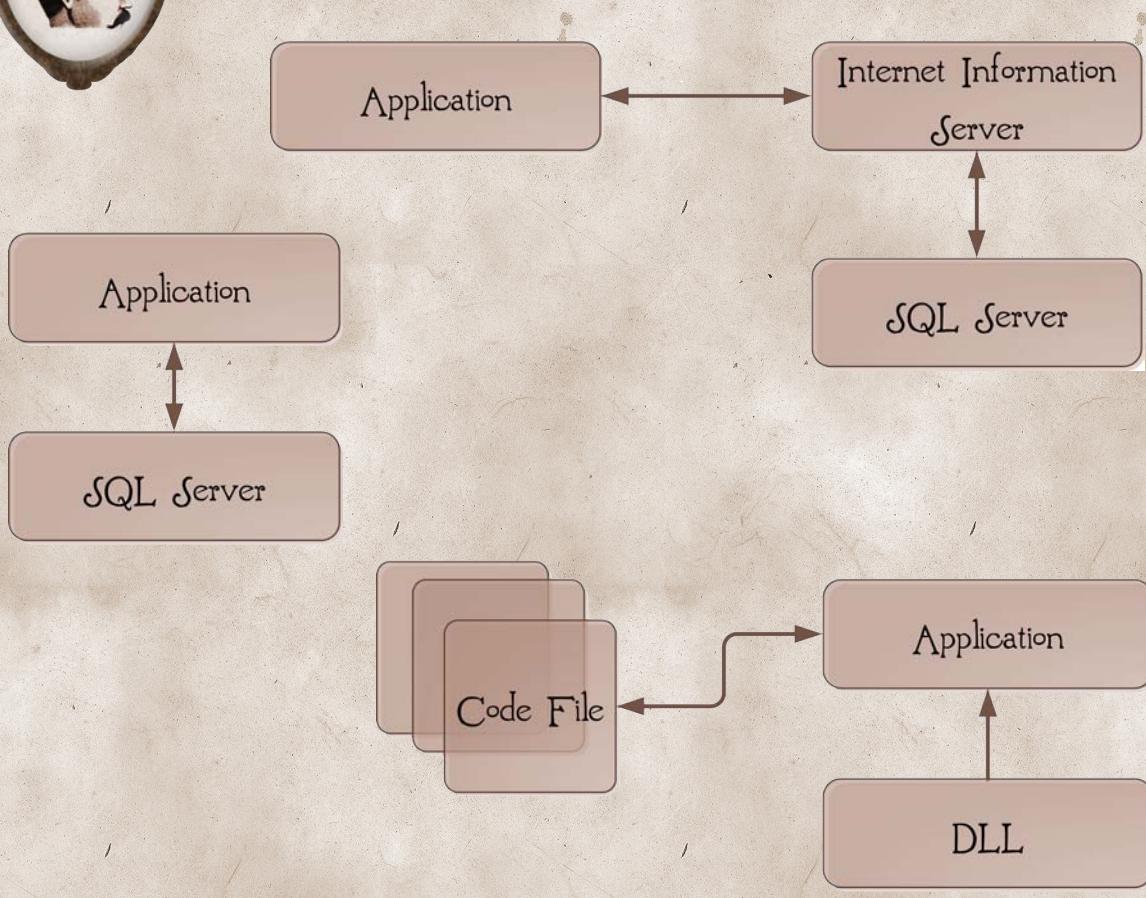
When an application is divided into different executable components, like the client and server in a Client/Server application, each component is called a tier. So, technically, the Client/Server architecture is two-tiered.

As applications grow in size and complexity, the number of tiers often increases. "N-tiered application" can be used to refer to any application that has more than one tier but is generally reserved for those that have more than two.



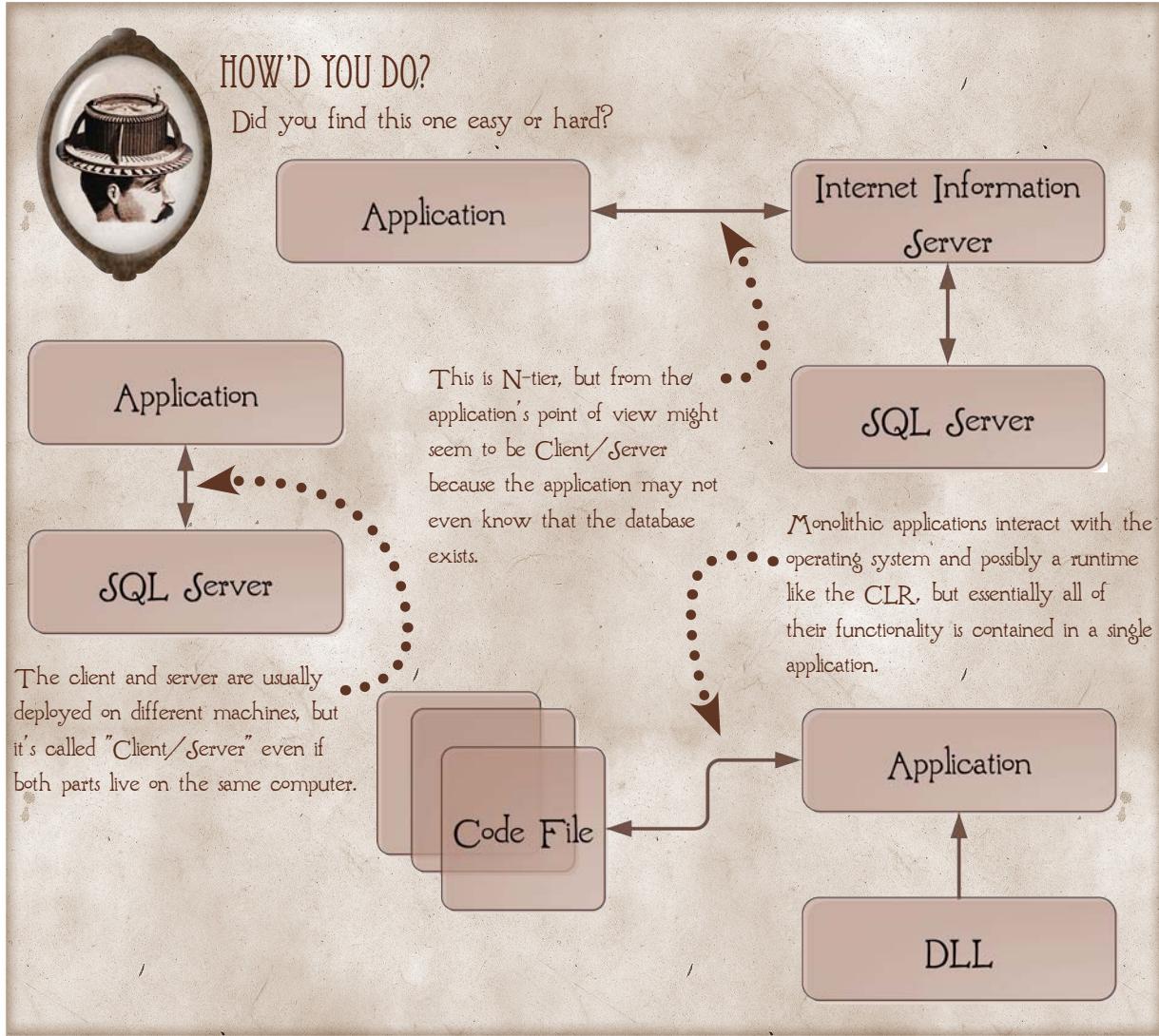
PUT ON YOUR THINKING HAT

Physical architectures are pretty straightforward. Can you label each of the examples below?



ON YOUR OWN

Looking back at that list of applications you want to write, what architectures do you think would be appropriate for each of them?



WORDS FOR THE WISE

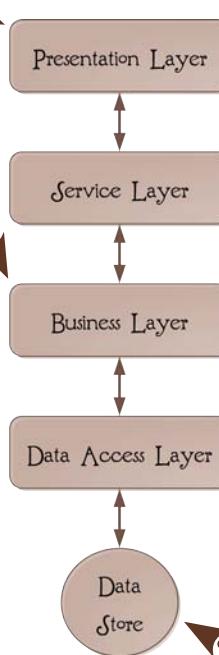
The user interface of an application is called the FRONT END; everything else is the BACK END. Back-end services might be called a SERVER or a SERVICE. Technically, a SERVICE communicates with its clients using standards-based interfaces that are discoverable at runtime in a SERVICE-ORIENTED ARCHITECTURE.

LOGICAL LAYERS

The application architectures (monolithic, N-tier, etc.) refer to the physical separation of components. The layered architecture refers to the *logical* separation of components. Not every application needs every layer, of course. It depends on what you're doing. An application that doesn't have a user interface doesn't have a Presentation Layer; one that doesn't access data doesn't have a Data Access Layer. (That may seem obvious, but it's surprising how many people seem to think there's something wrong if their application design doesn't include every layer. Hey, none of us is born knowing this stuff.)

The PRESENTATION LAYER is responsible for the way your application interacts with real people. It's what most of us think of as the "user interface".

The BUSINESS LAYER contains the core functionality of your application. Those tricky algorithms you're developing, and the logical data entities (which might be very different from the way data is stored) live here.



- The SERVICE LAYER is generally only implemented in applications that deal with other software in addition to real people. If the Presentation Layer is the "user interface", the Service Layer is the "computer interface".

The DATA ACCESS LAYER (DAL) is responsible for the communication between your application and wherever the data is stored (wherever and however that might be). The DAL doesn't do anything with the data; it just loads and saves it.

In the layered architecture paradigm DATA STORES—things like relational databases or XML files—aren't considered part of the application. You can think of them as tools used by the DAL.

WORDS FOR THE WISE



Confusing the logical and physical layers of a complex application can lead to, well, confusion. Best practice is to talk about physical tiers and logical layers. That way, you can say things like, "The business layer is split across multiple tiers" and it makes sense.

Unfortunately, people aren't always careful about making the distinction, so when you're reading other authors, be sure you know what they mean when they use one (or both) of these terms.

THE DATA ACCESS LAYER

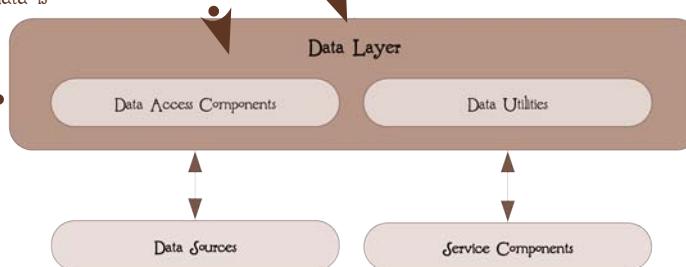
The DAL is responsible for persisting data, which is what programmers call loading and saving it someplace that won't disappear when the user turns off the computer. (Remember that the actual data stores aren't part of the layer.) When you separate the DAL from the rest of the application, you can easily change stores at any time. (And data storage technology tends to be pretty volatile—it's an XML file today and a SQL Server database tomorrow.)

Data Access Components are responsible for actually talking to the data store. They insulate the rest of the application from the details of how the data is physically stored.

The FCL provides a set of related technologies that make building the DAL a lot easier.

Data Utilities are usually private to the DAL. They're exactly what they sound like: utilities that help the data access components do their job. For example, a utility that translates a string value returned from a text file into a .NET numeric type would be considered a data utility.

Sometimes the DAL talks to the Service Layer of another application instead of talking directly to the data store. That doesn't change its logical role in the architecture.



ADO.NET CORE SERVICES

ADO.NET knows how to talk to several popular relational databases, XML, and any other data stores for which a OLE DB or ODBC provider exists (and that's a lot of them). It represents data in DataSet and DataTable objects that closely resembles the native structure of the data.

ADO.NET ENTITY FRAMEWORK

The ADO.NET Entity Framework builds on the core services provided by ADO.NET to represent data in a way that more closely resembles the OO paradigm. For example, The Entity Framework would represent RecipeItems as a Collection property of the Recipe object instead of a separate entity.

ADO.NET DATA SERVICES

ADO.NET Data Services allows an Entity Framework data model to be (easily) exposed to other applications over HTTP.

THE BUSINESS LAYER

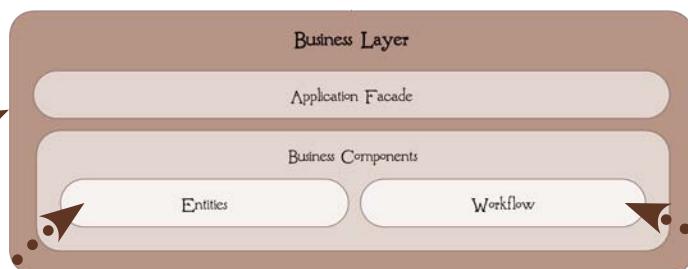
The Business Layer represents the smarts of your application. Oh, there may be some tricky stuff in the DAL (and there's almost always tricky stuff in the Presentation Layer), but applications hardly ever just retrieve data from the DAL and show it to the user, and all the logic that processes the data and makes sure that business rules are enforced lives here.

Most of the functionality of the business layer lives in the **BUSINESS COMPONENTS** that implement the logic of the applications.

These are collectively known as **BUSINESS RULES**, and the **BUSINESS LAYER** is where they're enforced.

The Business Layer is responsible for:

- Handling communication between the DAL and the Presentation and Service layers
- Enforcing data integrity and consistency
- Making sure things happen in the right order



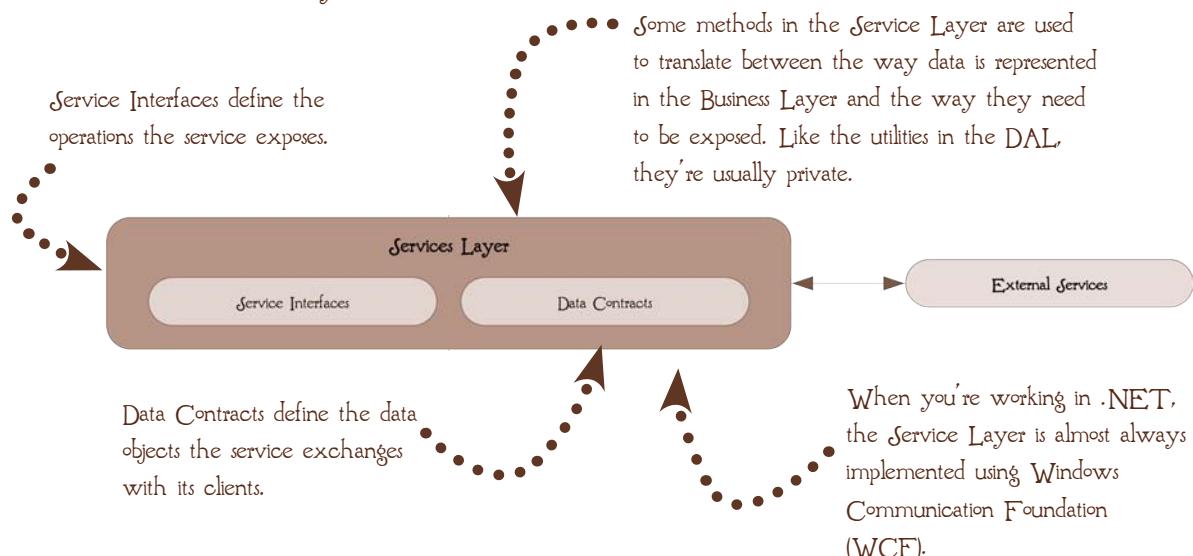
Business Entities are the objects you write to represent the objects that your application models.

BUSINESS WORKFLOWS are processes that need to be performed in a specific order. Workflows can get pretty complicated. If you're writing them, you'll want to check out the Windows Workflow technology.

An **APPLICATION FACADE** makes the components defined in the business layer simpler for the Presentation Layer to handle. For example, the Application Facade might combine several distinct operations into a single `AddRecipe()` method. Not all applications (in fact, not many applications) need an Application Facade.

THE SERVICE LAYER

You can think of the Service Layer as the user interface for users who aren't people. It exposes the functionality of your application as a service, which means that it has a standards-based interface. That interface is defined in this layer.



WORDS FOR THE WISE:



Internet services are a **HUGE** topic, and unfortunately we don't have room to explore them here. But to get you started, here are some basic terms you can check out in MSDN or your favorite search engine:

RESTful - REST stands for REpresentational State Transfer, and a RESTful service is one that conforms to the REST constraints. In essence, REST translates the basic Client/Server two-tiered architecture to the Web.

SOAP - SOAP stands for "Simple Object Access Protocol". It's a protocol for exchanging structured data over the Web.

SOA - SOA stands for "Service-Oriented Architecture". It refers to structuring applications as a set of interacting services.

And, of course, don't forget to check out WCF, which makes exposing your application as a service (almost) trivial.



PUT ON YOUR THINKING HAT

Here's a description of some functions that your application might implement. Can you assign them to the correct architectural layer?

The `IsValid()` method that ensures all the data is valid before calling `SaveChanges()`.

The `GetRecipe()` method combines two different ADO.NET `DataTable` objects into a single .NET object.

The `FormatForWeb()` method translates a .NET object into XML.

The `LoadData()` method populates an ADO.NET `DataSet` from an XML file.

The `CheckCredit()` method verifies that a customer is in good standing before allowing an order to be placed.



HOW'D YOU DO?

The `IsValid()` method ensures all the data is valid before calling `SaveChanges()`.

Validity is determined by business rules, so this one belongs in the Business Layer.

The `GetRecipe()` method combines two different ADO.NET `DataTable` objects into a single .NET object.

Did you put this one in the DAL? Many people would, but data manipulation is done in the Business Layer. The DAL only saves and retrieves it. Remember that the whole point of separating the DAL from the business logic is so that you can unplug one data store and replace it with another. You don't want to have to duplicate that manipulation code, right?

The `FormatForWeb()` method translates a .NET object into XML.

The name of this was probably a give-away. Anything that talks to other software, and particularly anything that talks across HTTP, lives in the Service Layer.

The `LoadData()` method populates an ADO.NET `DataSet` from an XML file.

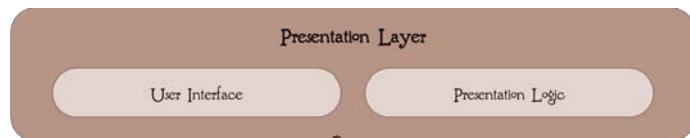
Loading data is exactly what the DAL is intended to do, so that's where this function should live.

The `CheckCredit()` method verifies that a customer is in good standing before allowing an order to be placed.

This involves both a business rule ("customer must be in good standing") and the enforcement of a process ("credit check before order placement"), so it belongs in the Business Layer.

THE PRESENTATION LAYER

The Presentation Layer is responsible for the way your application interacts with real people. It includes the actual interface (the widgets that the user sees) and what's called the presentation logic, which is just geek-speak for "what happens when the user clicks this button."



.NET gives you a lot of options for building a user interface. We'll be examining Windows Presentation Foundation (WPF) in this book, but you might also want to explore some of these others.

WINDOWS FORMS (WINFORMS)

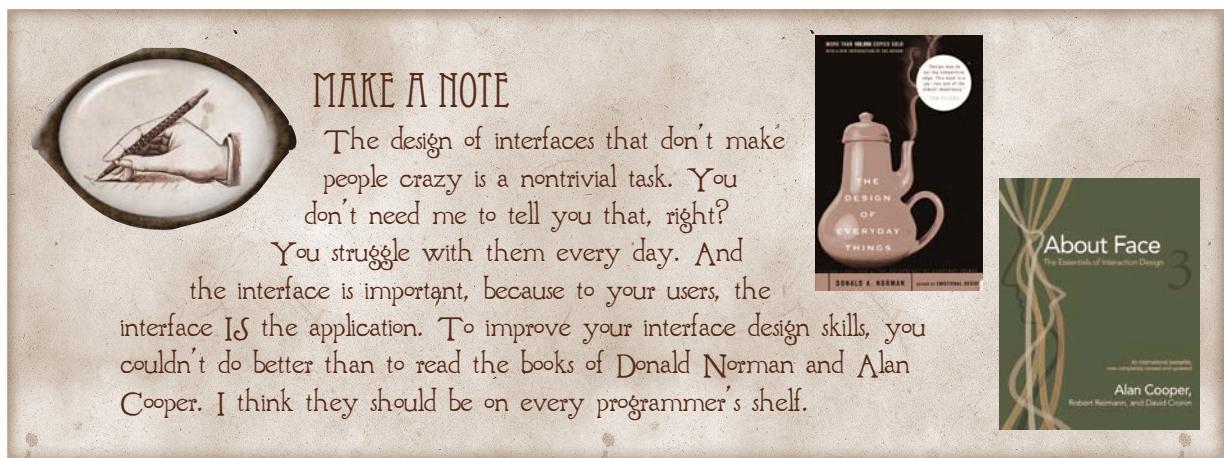
This is an older technology for writing desktop applications. If you're working on an existing application, this is probably what you'll use.

ASP.NET WEB FORMS

This is (more or less) the Web-based version of the WinForms toolkit. Again, you'll probably use it if you're working on existing applications, but for new projects, you'll want to look at other options.

SILVERLIGHT

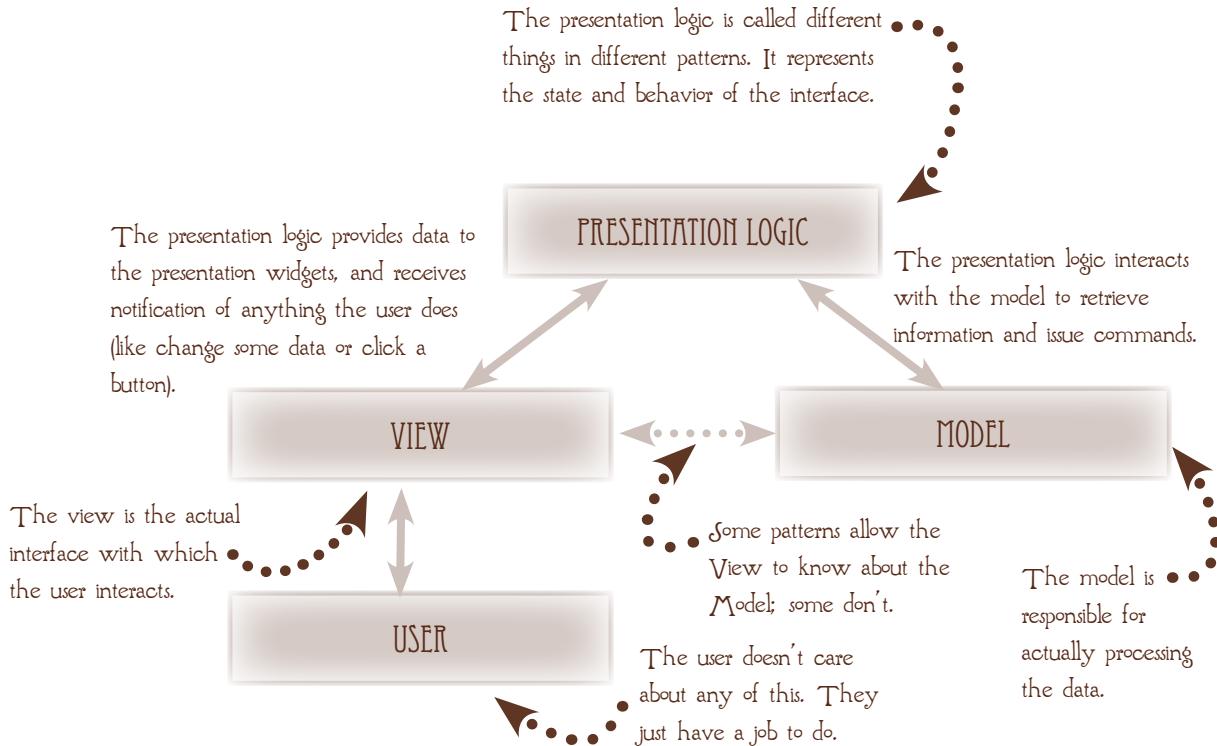
Silverlight is closely related to WPF. It runs over the Web (as a replacement for Web Forms), on the desktop (with some limitations), and on mobile devices like Windows Phone. If you need to deploy your application on multiple platforms, Silverlight is probably your best option.





PRESENTATION PATTERNS

There are several common patterns for the Presentation Layer, and each one has a number of variations, but they all have the same basic goal: to separate the widgets the user sees from the logic that maintains the data those widgets display or responds to interactions the user has with the widgets. This is often described as separating state and behavior from presentation.



ON YOUR OWN

When you separate the logic behind an interface from the interface itself, it becomes much easier to do a couple of things. Do you know what they are? (Hint: both words begin with "re".)

PATTERN PROS...

All of the Presentation Layer patterns separate the Presentation Layer into two components, one responsible for display, the other responsible for processing. This is a general technique known as separation of responsibilities, which is, of course, closely related to the Single Responsibility Principle we examined in the last chapter. The separation of responsibilities has a couple of immediate benefits:

THE VIEW CAN BE DEVELOPED INDEPENDENTLY

Although every programmer should have a good grasp of interface and interaction design, very few of us are graphic designers. It's a whole different skill set. Separating the Presentation Logic from the View allows everybody to concentrate on what they're best at.

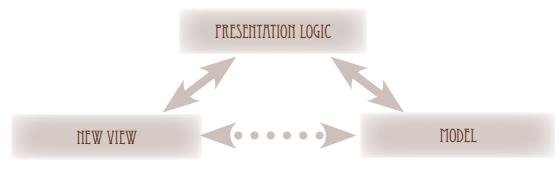
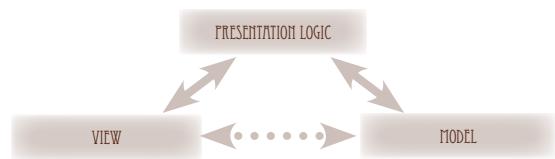
THE VIEW CAN BE REPLACED

The interface of an application changes more often than most of the logic behind it. Think about Microsoft Office, for example. Every new release brings a slightly different interface, but do you really think that the basic algorithm for, say, assigning a font to a paragraph, changes with every version? Probably not. By separating the presentation logic from the View, you can refine or replace the interface widgets without impacting anything else in the application.

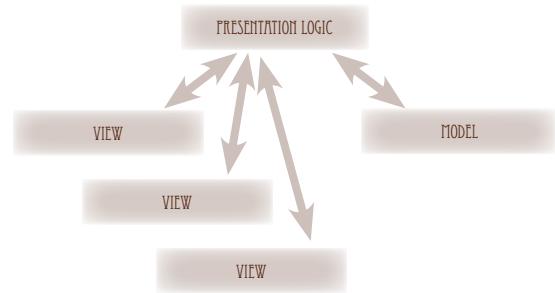
THE PRESENTATION LOGIC CAN BE RE-USED

In our recipe application, we might want to allow users to view the recipes on file in several different ways. One View might show the details of a single recipe, for example, while another might list all recipes on file for a given category (all the soups, for example), and another might allow the user to search for a recipe based on ingredients.

This is a pretty common scenario, and the presentation patterns allow the logic behind the View to be reused without change, which is always a good thing.



• You can just swap one View for another.



...AND CONS

THE MODEL BECOMES MORE COMPLEX

The most obvious downside of the presentation patterns is that they add complexity to the application. Not only are you adding additional classes to the model, which almost always adds a certain amount of complexity, but the interaction between the three components of the models can be fairly complex in and of itself.

THE PATTERN IN USE ISN'T ALWAYS DISCOVERABLE

While any design pattern can be a little obscure (not to mention scary) when you first start working with it, once you gain a little bit of experience, you can generally spot when one is being used just by inspecting the code. If you see that a constructor is instantiating an interface, and the interface defines a behavior or two, for example, you know you're working with `Strategy`.

That's less true of the presentation patterns. Although file naming conventions help—if a project contains classes named `RecipeModel` and `RecipeSearchView` it's a pretty good hint that you're working with one of the patterns—they don't tell the whole story. All three patterns use Models and Views, so which one are you working with? Because, as we'll see, the way you modify an application will be different depending on the specific pattern in use.

SOME THINGS ARE SIMPLY DIFFICULT TO DO

While they differ in detail, all of the design patterns enforce a certain level of ignorance between the components. From a theoretical standpoint that's a good thing, because it means the classes are loosely coupled. From a practical standpoint, it can make some things (sometimes very simple things) difficult to implement.

Exactly what's going to be tricky depends on the interface technology you've chosen and the pattern you're implementing, but in general, it's difficult to keep the presentation logic completely ignorant of the view. For example, something as simple as displaying a `MessageBox` to the user from within a validation routine can get ugly, because the validation routine isn't supposed to know about the user interface.

Fortunately, these problems are well understood, and the development community has created solutions to most of them, usually in the form of a "helper" routine that does whatever tricky thing you need to do without breaking the pattern. Visual Studio even has project templates for `MVC` Web applications and `MVVM` WPF applications that already implement the basic helpers, so the benefits of replacement and reuse can easily outweigh the implementation difficulties. In fact, once you understand the use of the helper classes (and there are hundreds of tutorials and forums responses that will help you do just that), you may not find the patterns any more difficult to implement than a standard interface.

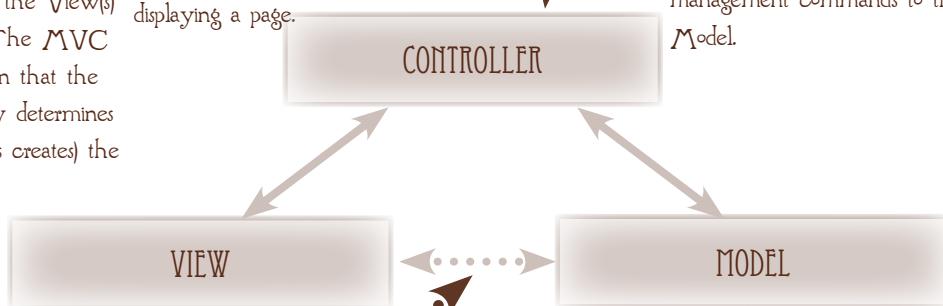
MODEL-VIEW-CONTROLLER

The Model-View-Controller (MVC) pattern is the granddaddy of all presentation patterns. Most of the other presentation patterns, including the MVP and MVVM patterns we'll be looking at here, descend, directly or indirectly, from MVC. MVC is a good match for the disconnected architecture of the Web, and it's extremely popular in ASP.NET development.

In most of the presentation patterns, the presentation logic is decoupled from the View(s) that rely on it. The MVC pattern is unique in that the Controller actually determines (and in some cases creates) the View.

This works really well for thin-client applications (thin being geek-speak for "dumb"), where the Controller receives a request and responds by displaying a page.

In MVC, the presentation logic component is called the Controller, and it's well named, because it is actually in control of both the View and the Model. An MVC Controller responds to user input, creates Views, and issues data management commands to the Model.



DATA BINDING is a mechanism for allowing an interface widget to set a property based on some data. We'll discuss the WPF version of data binding in Chapter 22.

In a connected environment like a desktop application, the Model is allowed to communicate directly with the View, typically by way of a data binding.

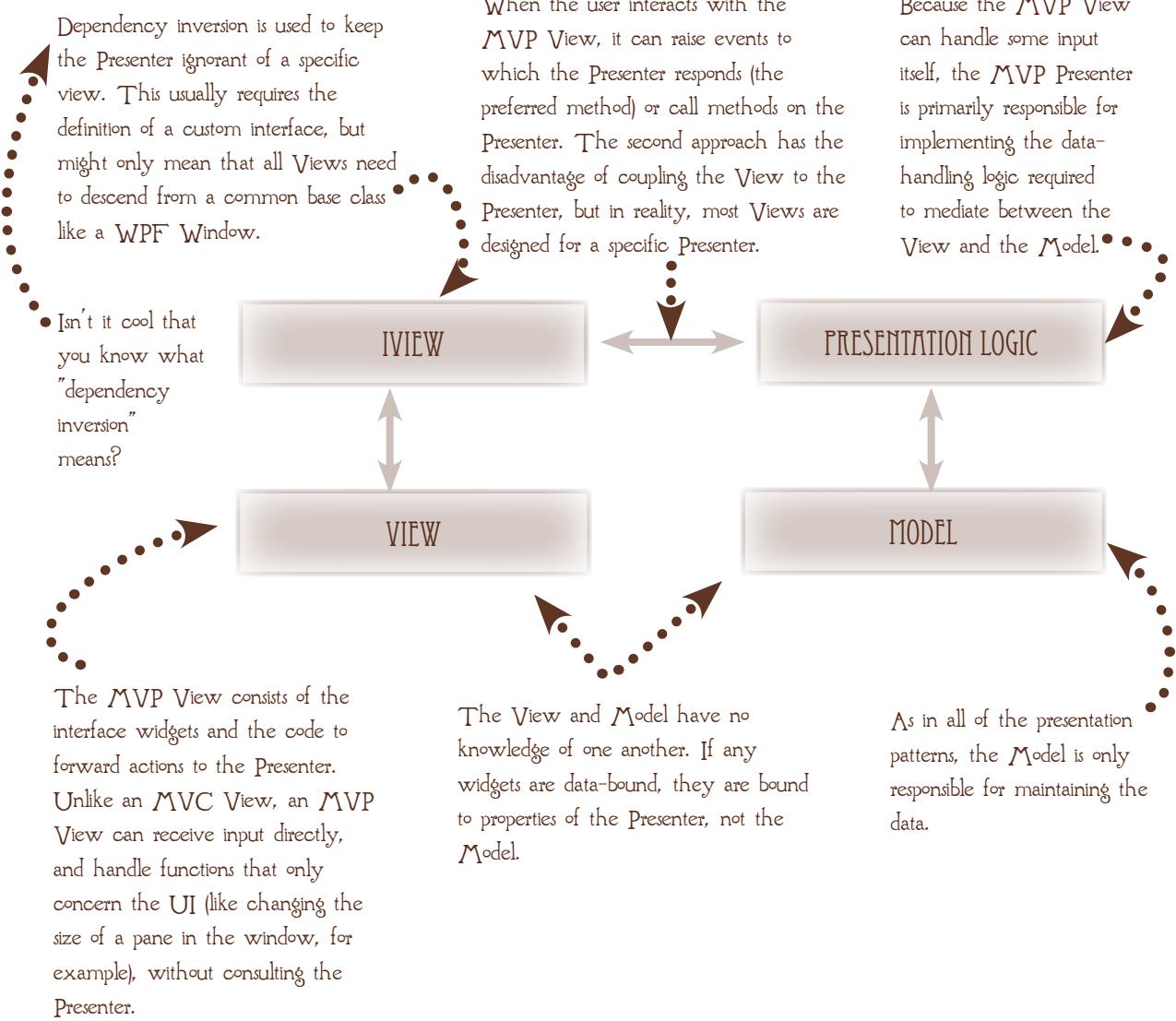
The Model is responsible for loading, saving, and updating data.

ON YOUR OWN

.NET defines a framework for developing ASP.NET MVC applications in the System.Web.Mvc namespace. If you'll be developing Web applications using ASP.NET, be sure to check out the tutorials you can find at www.asp.net.

MODEL-VIEW-PRESENTER

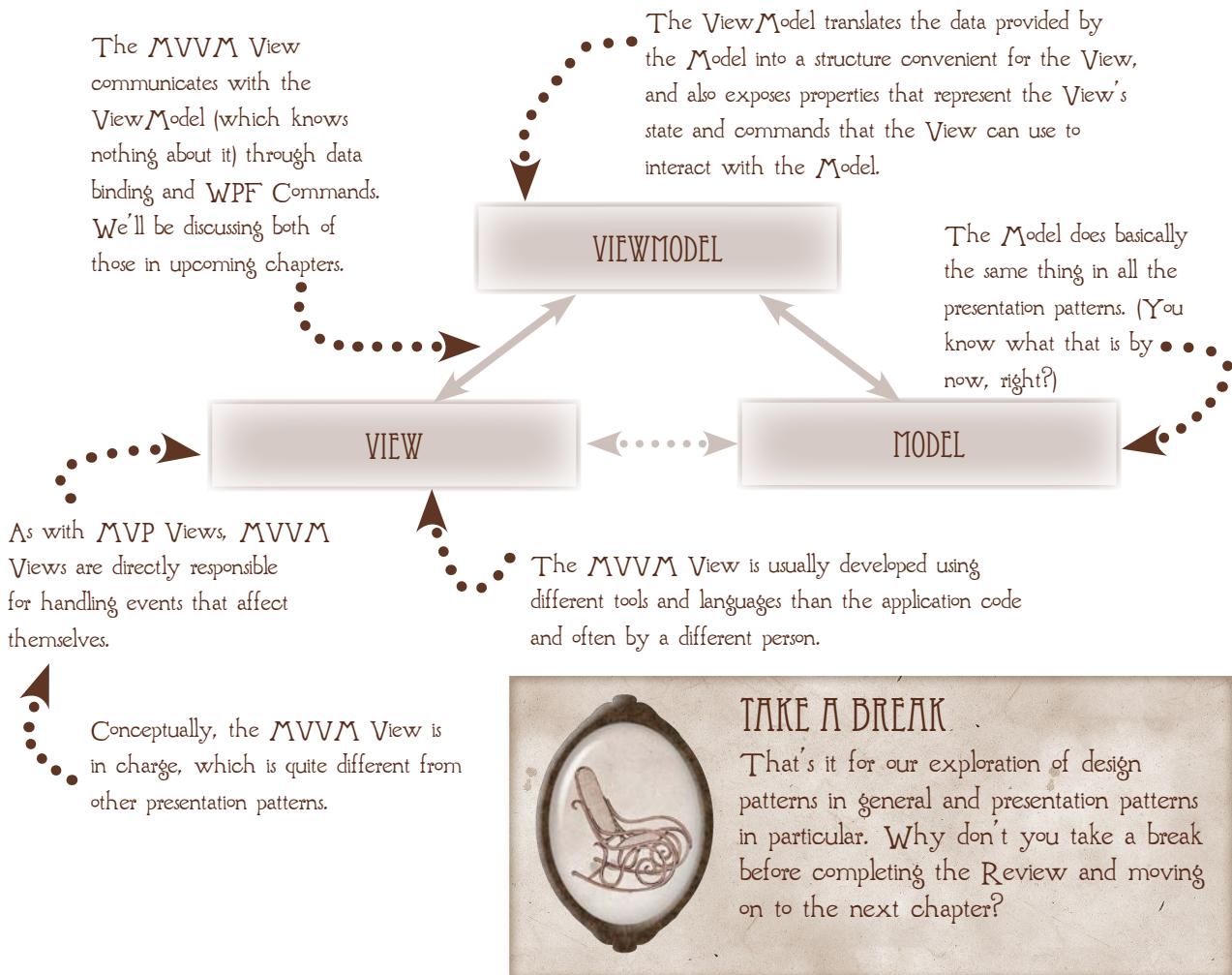
The Model-View-Presenter (MVP) pattern is a direct descendant of MVC and is a better match for platforms where the interface widgets themselves have significant functionality such as Windows desktop applications or Rich Internet Applications written using a technology such as Silverlight.



MODEL-VIEW-VIEWMODEL

The *Model-View-ViewModel (MVVM)* presentation design pattern descends from *MVC* by way of Martin Fowler's Presentation Model. (We won't be exploring the Presentation Model, but you can read Fowler's original description of it at www.martinfowler.com.) Although it can be applied in other environments, the *MVVM* pattern was developed specifically with *WPF* in mind, which isn't surprising since it was first described by John Grossman, one of the architects of *WPF* and *Silverlight*.

MVVM is the most loosely coupled of all the presentation models we've explored. The *View* only knows the *ViewModel*, the *ViewModel* only knows the *View*, and the *Model* is lonely indeed.



TAKE A BREAK

That's it for our exploration of design patterns in general and presentation patterns in particular. Why don't you take a break before completing the Review and moving on to the next chapter?





REVIEW

What three objects are required to define a .NET event?

Which of the presentation patterns is particularly well suited to Web applications?

What are the three primary benefits of using design patterns? Can you think of any other benefits?

Why is it not surprising that WPF is particularly well suited to the MVVM pattern?

What problem does the Strategy pattern address?

What's the difference between an application's architecture and the logical layers it contains?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



XAML



15

Well, we've learned about the .NET environment, the VB programming language, and we've taken a quick look at some of the principles and practices that will help you make your programs efficient and resilient. Now it's time to return to the .NET Framework and get our hands dirty with one of the FCL technologies.

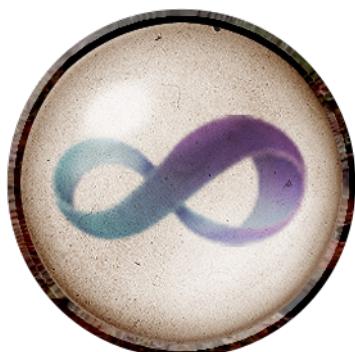
We're going to look at Windows Presentation Foundation (WPF), one of the two .NET technologies that you can use to develop applications for the Windows desktop.

Now, before you stop reading because you're not interested in Windows desktop applications, let me assure you that the reason I chose WPF as the technology to explore is that the concepts and techniques you'll learn in studying WPF will be useful in most of the other .NET technology areas. Silverlight, for example, which is used for developing Rich Internet Applications and Phone 7 apps, is basically a subset of WPF, and the integration of an XML-based language with VB is fundamental to most of Microsoft's new technologies.

It's that XML and code integration that we're going to examine in this chapter. XAML (pronounced "zammel", which rhymes with "camel") is the declarative language that's used to describe the layout of a WPF window. It's based on XML (XAML is actually an acronym for "Xml Application Markup Language") and allows you to separate the appearance of an application (defined in the XAML) from its behavior (defined in code-behind.)



FITTING IT IN



.NET PLATFORM COMPONENTS

.NET SERVERS

SQL Server

IIS

BizTalk Server

SharePoint Server

.NET TOOLS

Visual Studio

Expression Studio

Framework SDK

.NET FRAMEWORK

Specialty Frameworks

Compact

Micro

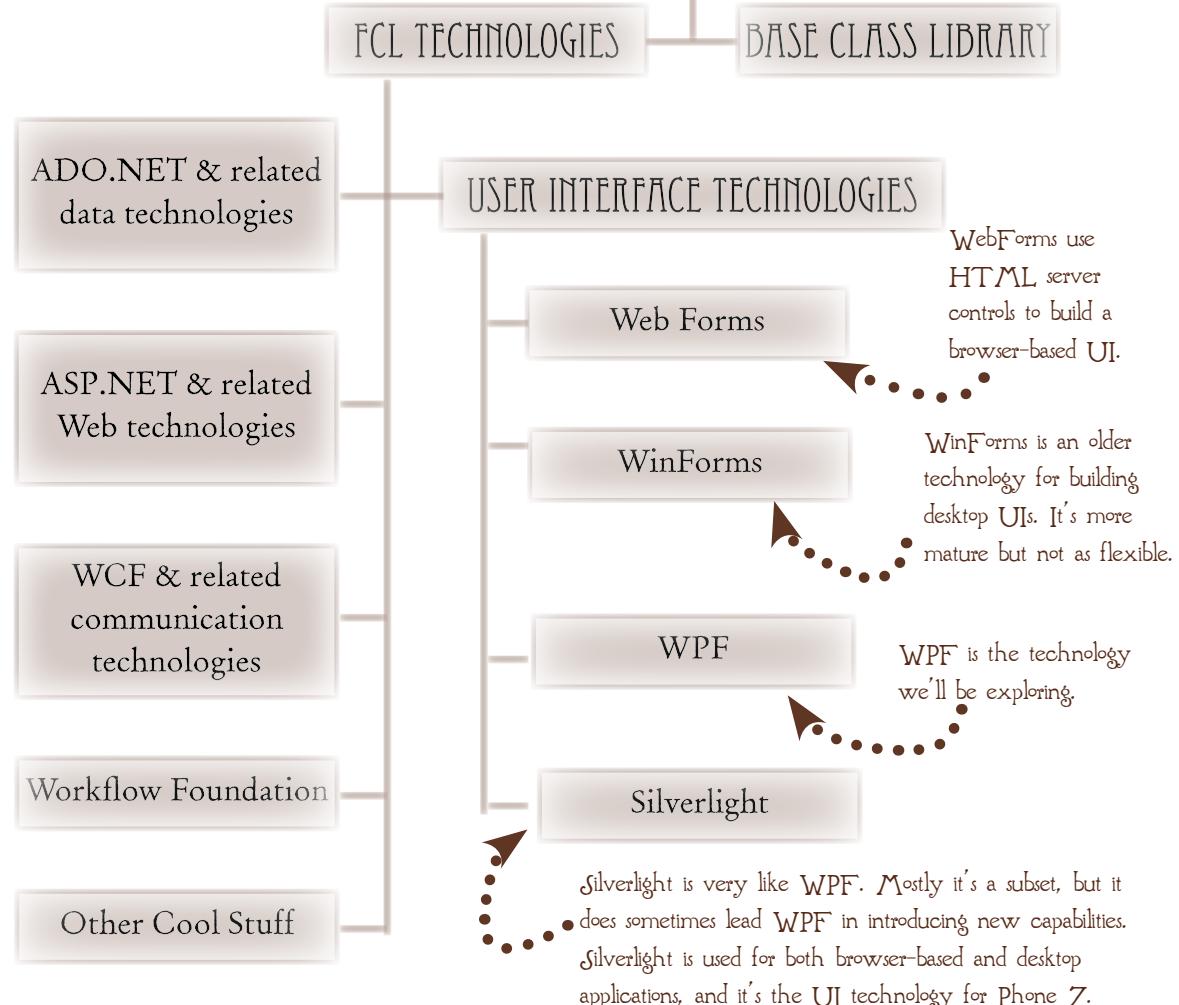
Windows Phone

Class Library

CLR

Languages

CLASS LIBRARY



WHAT?? ANOTHER LANGUAGE?



In vino veritas

- Truth in wine

Semper fidelis

- Always faithful

Carpe diem

- Seize the day

Cogito ergo sum

- I think, therefore I am

Caveat emptor

- Buyer beware

I'm glad you asked. Just as a well-educated English speaker is expected to know at least a few common phrases in Latin, as a programmer, whatever language or type of application you specialize in, you'll need to know the basics of XML (which is trivially easy) and XAML is becoming a pervasive part of .NET Framework programming.

To be fair, I need to say that it's perfectly possible to write a WPF application without writing a single line of XAML. But splitting the appearance and behavior into two separate files has several advantages, and we'll see some of them later in this chapter.

A SNEAK PEEK

Here's a sample of what we're going to be learning in this chapter. It's the **XAML** from the application we wrote in Chapter 2 that describes a simple screen with one button.

Like any XML language, XAML elements (called tags) are wrapped in angle brackets. The `<Window>` element is the outermost element in this snippet.

Attributes are defined in the opening tag using `AttributeName = "value"` syntax.
This attribute sets the window title.

```
<Window x:Class="_03app01_Two_Windows.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="A Prettier Window" Height="270" Width="441">
    <Grid Background="Black">
        <Button Height="74" HorizontalAlignment="Left" Margin="91,71,0,0"
            Name="button1" VerticalAlignment="Top" Width="229"
            FontFamily="Windlass" FontSize="24" Click="button1_Click">
            Click Me, Please
        </Button>
    </Grid>
</Window>
```

Elements can be nested inside other elements. This `<Grid>` element is nested inside the `<Window>`.

This is the closing tag for the `<Window>` element. It begins with a slash.





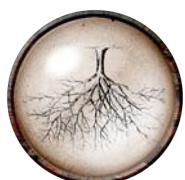
TASK LIST

In this chapter we'll explore the declarative XAML language and how it fits in the development process.



XAML FUNDAMENTALS

XAML is a specialized form of XML, and the rules of its syntax conform to XML rules. We'll start by looking at exactly what those rules are and how to use them to make your XAML code as readable as possible.



WPF TREES

XAML, like XML, is intrinsically hierarchical. In a WPF application, there are actually three different hierarchies: the **OBJECT TREE** that you build in your XAML markup, the **LOGICAL TREE** that models how elements are nested, and the **VISUAL TREE** that represents how elements are displayed to the user. Most WPF classes expose **Parent** and **Children** properties related to their position in the trees.



XAML AND VB

For the most part, Visual Studio handles combining the XAML and the code you write for you, but it's a good idea to know what's happening behind the scenes, so we'll take a quick look at how the XAML and code-behind files get combined at design-, compile-, and runtime.



XAML FUNDAMENTALS

Let's start our examination of XAML with a comparison of XAML markup with something you're already familiar with: VB. The following code snippets produce the same window:

```
<Window x:Class="MainWindow" Title="My  
Window"  
Name="MyWindow">  
  
<Grid name="LayoutRoot">  
  
    /  
  
    <!-- The button is inside the grid -->  
    <Button name="FirstButton" Height="74"  
        Width="229">  
        Click Me, Please  
    </Button>  
  
    </Grid>  
</Window>
```

```
Dim MyWindow As MainWindow  
MyWindow = New MainWindow()  
MyWindow.Title = "My Window"  
  
Dim LayoutRoot As Grid  
LayoutRoot = New Grid()  
MyWindow.Content = LayoutRoot  
  
'The button is inside the grid  
Dim FirstButton As Button  
FirstButton = New Button()  
Button.Height = 74  
Button.Width = 229  
Button.Content = "Click Me, Please"  
  
LayoutRoot.Children.Add(FirstButton)
```



PUT ON YOUR THINKING HAT

If VB can do everything XAML can do, and a lot more besides, can you think of some reasons why using both languages in the application development process is a good idea?



HOW'D YOU DO?

Did you think of some reasons? Were you convinced by them? Here's a list of a few you might not have thought of:

XAML IS OFTEN MORE CONCISE

If you compare the snippets on the previous page, you'll see that the XAML version of the window is much less verbose. (But that's not always true.)

XAML IS DECLARATIVE

"Declarative" doesn't necessarily mean better, but it's often easier to understand (and write). Imagine the difference between saying, "I want an electrical outlet on that wall" and actually wiring it.

XAML CLEARLY EXPRESSES THE WPF COMPOSITION MODEL

As we'll see when we explore logical and visual trees, WPF interfaces are composed. Things don't just float around on their own; a String is contained in a Button that's contained in a Grid that's contained in a Window; the hierarchical structure of XAML makes that much, much clearer than procedural code.

XAML IS TOOLABLE

The boffins at Microsoft talk about XAML being "toolable". What they mean is that because XAML is plain old XML, it can be easily parsed and manipulated by any application. A great example of that is Microsoft's Expression Blend, a designing and prototyping tool aimed at designers rather than developers (for which read, "it will feel familiar to people used to working in Adobe Photoshop").

XAML SUPPORTS SEPARATE WORKFLOWS

Programmers, as a class of people, aren't known for their graphic design skills; it's a different profession. But once you have separate tools for designing the graphics and implementing the behavior, it becomes trivially simple to let different people perform the tasks they're trained to do. Doesn't it make sense to let the graphic designers design and the programmers program? XAML makes that easy. And because many graphic designers are familiar with another XML-based language, HTML, many of them find working directly in XAML quite comfortable.

SYNTAX BASICS

If you're familiar with XML or HTML, the syntax of XAML will already be familiar, but it adds some extensions to basic XML and uses some special vocabulary. If you're not familiar with XML, don't worry. It's really, really easy.

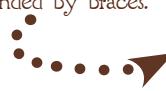
OBJECT ELEMENTS

The basic unit of VB is a statement. The basic unit of XAML is an object element. Just as in VB you can write single- or multi-line statements, you can write single- and multi-line object element declarations:

This syntax is used for elements that don't contain any other elements. It's roughly equivalent to a single-line VB statement.



This syntax is used when other elements are nested inside the object. It's roughly equivalent to a VB statement block surrounded by braces.



ATTRIBUTES AND PROPERTIES

In VB, you assign a value to the property of an object using syntax like `Object.Property = value`. In XAML, you have two options. You can either set the property as an attribute that's declared in the opening tag, or you can define the property as a separate nested element:

This is ATTRIBUTE SYNTAX. The values of the attributes are contained in quotation marks, and multiple attributes are separated by white space.

```
<Button Height="74" Width="229" />
```

This is called PROPERTY ELEMENT SYNTAX. For some complex attributes, it's a lot easier to write and understand than attribute syntax.

```
<Button Width="229">
  <Button.Height>74</Button.Height>
</Button>
```

Notice that you can combine property element and attribute syntax.

CONTENT PROPERTIES

Did you notice that in the `Button` property element example, we had to specify the `Height` property as `Button.Height`?

When we used attribute syntax, we only had to specify the property name.

```
<Button Width="229">  
  <Button.Height>74</Button.Height>  
</Button>
```

Property element syntax required the object name and the property name.

Repeating the object name like that makes it clear that you're referring to a property, not some other element named `Height`. But every WPF element can define a single property, called the content property, that can simply be placed between the opening and closing tags of the object. For the `Button`, the content property is whatever's displayed inside the `Button`, usually a string:

```
<Button>  
  Click Me, Please  
</Button>
```

The content property doesn't need to be enclosed in tags (although you can) and some elements, like the WPF `Button` can even interpret strings directly without enclosing them in quotes.

The tricky bit about using the content property is knowing what it means. Technically, the content property is any object that will be displayed by the `ContentPresenter` of the control's template. (We'll examine control templates in Chapter 21.) It's usually intuitive, though, like the `Button`. I mean, what else would you expect the control to do with "Click Me, Please" but display it?



PUT ON YOUR THINKING HAT

Time to get your feet wet. Can you translate the VB snippet below into XAML and the XAML code into VB? (Don't worry about what all the elements are. We'll be examining them in later chapters.)

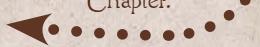
```
Dim MainWindow As Window = New MainWindow()
MainWindow.Title = "Sample"
MainWindow.Height = 300
MainWindow.Width = 500

Dim LayoutRoot As DockPanel = new DockPanel()

Dim MainMenu As Menu = New Menu()
Dim FileMenuItem As MenuItem = new MenuItem()
FileMenuItem.Header = "File"

MainMenu.AddChild(FileMenuItem)
LayoutRoot.Children.Add(MainMenu)
MainWindow.Content = LayoutRoot
```

Notice that you call AddChild() on the Menu, but Add() on the LayoutRoot. You'll find out why when we examine WPF controls in the next Chapter.



```
<Window x:Class="MainWindow" Height="Auto" Width="Auto">
<Grid name="LayoutRoot">
<Button Height="50" Width="75">
    Cancel
</Button>
</Grid>
</Window>
```



HOW'D YOU DO?

Remember, there are different ways to express the same thing in both VB and XAML, so don't worry if your code is a little different than mine, so long as the syntax is correct.

```
Dim MainWindow As Window = New MainWindow()  
MainWindow.Title = "Sample"  
MainWindow.Height = 300  
MainWindow.Width = 500  
  
Dim LayoutRoot As DockPanel = new DockPanel()  
  
Dim MainMenu As Menu = New Menu()  
Dim FileMenu As MenuItem = new MenuItem()  
FileMenu.Header = "File"  
  
MainMenu.AddChild(FileMenu)  
LayoutRoot.Children.Add(MainMenu)  
MainWindow.Content = LayoutRoot
```

Did you have a problem figuring out what to call the variables here? Unlike VB variables, XAML elements don't have to have names unless you're going to reference them in code, but it's a good idea to give them one using the name attribute.

```
<Window x:Class="MainWindow"  
Height="Auto" Width="Auto">  
  <Grid Name="LayoutRoot">  
    <Button Height="50" Width="75">  
      Cancel  
    </Button>  
  </Grid>  
</Window>
```

```
<Window x:Class="MainWindow"  
  Title="Sample"  
  Height="300" Width="500">  
  <DockPanel Name="LayoutRoot">  
    <Menu Name="MainMenu">  
      <MenuItem Name="FileMenu"  
        Header="File" />  
    </Menu>  
  </DockPanel>  
</Window>
```

This one was a little tricky. Did you get it right? You don't need to explicitly add the MenuItem to the Menu because that's implicit in the structure of the XAML.

```
Dim myWindow As MainWindow  
myWindow = New MainWindow()  
  
Dim LayoutRoot As Grid = New Grid()  
  
Dim myButton As Button = New Button()  
myButton.Height = 50  
myButton.Width = 75  
  
LayoutRoot.Children.Add(myButton)  
myWindow.Content = LayoutRoot /
```

XAML COLLECTIONS

In the last exercise we saw that XAML makes it unnecessary to explicitly add elements to their containing elements because the ownership hierarchy is implicit in the way the elements are nested. XAML also supports a special syntax for adding elements to properties that are collections in a similar way.

This is called **COLLECTION PROPERTY SYNTAX**, and like a lot of things in XAML, you don't need to worry too much about it. It just works.



```
<Element>
  <Element.CollectionProperty>
    <Item />
    <Item />
    ...
  </Element.CollectionProperty>
</Element>
```

The XAML compiler is smart enough to know that the items nested inside a collection property are children of the collection.



PUT ON YOUR THINKING HAT



One of the WPF elements that has collection properties is the `Grid` layout panel. (We'll look at layout panels in detail in the next chapter.) Like other panel controls, the `Grid` can contain multiple children, and it organizes those children into rows and columns. The rows are defined as `<RowDefinition>` elements in the `<Grid>` `RowDefinitions` collection, and columns are `<ColumnDefinition>` elements in the `<Grid>` `ColumnDefinitions` collection. Using collection property syntax, can you write the XAML that declares a `<Grid>` element with three rows and three columns? Don't worry about any other attributes or properties.



HOW'D YOU DO?

Be careful about how your elements are declared. `RowDefinitions` (plural) is a collection property. `RowDefinition` (singular) is a member of the collection.

Because we're not specifying any attributes, the single-tag syntax works fine for `RowDefinition`.

You need to be careful that nested elements are completely enclosed in their parent. Intellisense will help you with this.

`<Grid.RowDefinitions>` is the collection property. It's declared using standard XAML property element syntax.

```
<Grid>
  <Grid.RowDefinitions><!--
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
</Grid>
```

The `<RowDefinition>` elements are the members of the collection and use collection property syntax.



UNDER THE MICROSCOPE

We talked about how text content properties don't need to be enclosed in quotation marks, but have you noticed that XAML is really smart about interpreting value types? In VB, 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.

Behind the scenes, XAML has a bunch of converters that handle various types of data, and you can write your own if you really need to, but most of the time you can ignore all that. XAML just works. Don't you love that?

XAML ROOTS & NAMESPACES

Like any well-formed XML, XAML is hierarchical, and there must be a single top-level element that contains all the others. It's called the **ROOT ELEMENT**, and it has some basic requirements.

The root element of a XAML file must:

- Be a `<Window>`, `<Page>`, `<Dictionary>` or `<Application>` element.
- Define the `x:Class` attribute that ties the XAML to the code-behind file.
(We'll talk about this one in a minute.)
- Contain a reference to the presentation and XAML namespaces.

Here's the XAML syntax for declaring a namespace:

```
xmlns:[id]=""<namespace>"
```

Visual Studio will add the XAML namespace declarations for you. They are:

Because this declaration doesn't have an identifier, it's the default namespace for the file.

The presentation namespace declares all the WPF elements like `Window` and `Button`.

```
xmlns="http://schemas.microsoft.com/winfx/2008/xaml/presentation"  
xmlns:x="http://schemas.microsoft.com/winfx/2008/xaml"
```

The `x` identifier is used by default to refer to a namespace that defines the XAML language.

You don't really need to worry about what these mean and do. Visual Studio will add them for you, and if you're working outside Visual Studio, well, cut and paste is a good thing...

DECLARING NAMESPACES

In addition to the two namespace declarations that are required in a XAML root element, you can reference the namespaces you create in your code-behind, or .NET FCL namespaces. The syntax for referencing these namespaces is a little different:

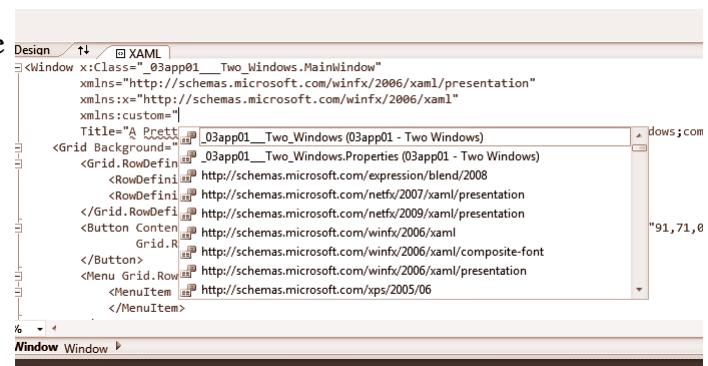
clr-namespace: is a special token that has meaning to the XAML compiler. It links the XAML namespace to the sort of namespaces you've used in your VB applications.

`xmlns:[id]=""clr-namespace:<name>[;assembly]"`

The ID isn't really optional, because by convention the default namespace (the one without an ID) is set to the WPF schema, and you really should leave the convention alone because it's what Visual Studio provides for you, what other tools expect, and what lets you use, for example, `<Button>` instead of, say, `<WPF:Button>`, which would be a real nuisance.

The specification of an assembly is only required if the namespace isn't in the same .NET assembly as the declaration.

In practice you don't need to worry too much about XAML namespace syntax. It's important to add declarations for FCL classes and classes in your project you want to reference, but Visual Studio Intellisense presents you with a list of available namespaces as soon as you type the equals sign, and once you choose one, it writes the correct declaration for you. Again, it just works.



XAML SYNTAX ONE MORE TIME

XAML stands for _____.

To set the `Width` property of a `<Window>` element using attribute syntax, you'd write the following XAML:

To do the same thing using property syntax, you'd write the following XAML:

By convention, the default namespace of a XAML file refers to which of the following namespaces?

- The XAML language namespace
- The WPF namespace
- The local assembly

It isn't necessary to explicitly add children to elements in XAML because:



TAKE A BREAK

There's lots more to learn about XAML, and we'll look at some refinements in later chapters, but now you've seen the basic syntax. It's certainly a lot easier than VB, isn't it?

Why don't you take a break before you do one final Review and we move on to WPF logical and visual trees?

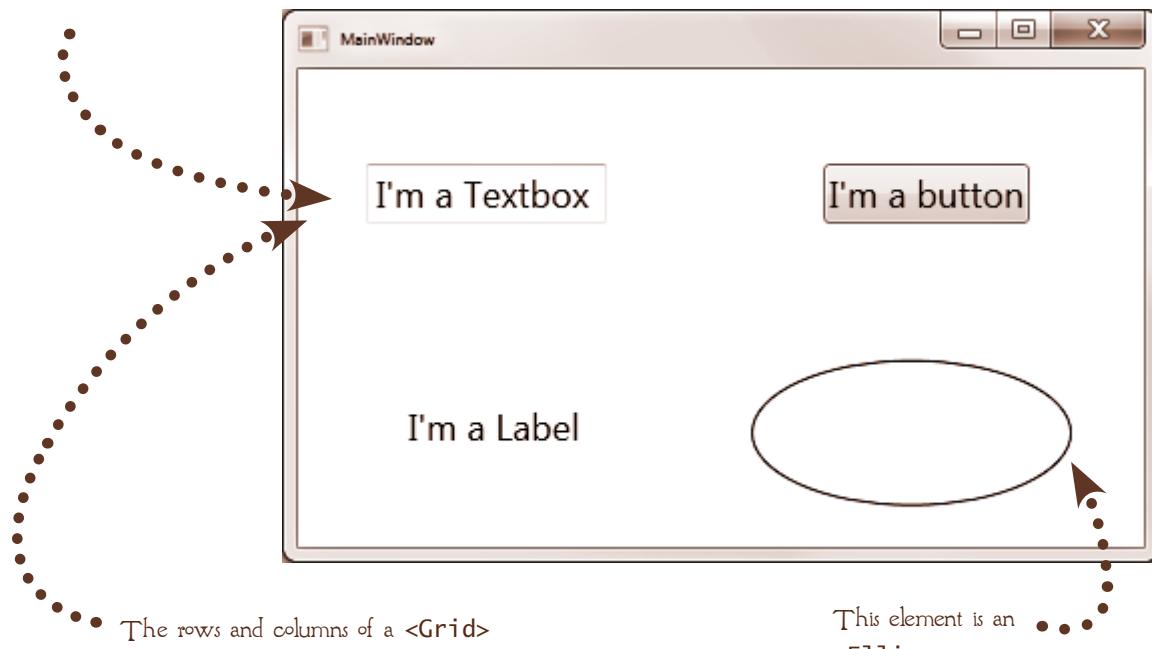


REVIEW

Here's a WPF `<Window>` that contains a `<Grid>` that uses two rows and two columns to lay out its children. Can you write the basic XAML to build it?

The syntax to specify which row and column of a `Grid` an element should be displayed in is

`Grid.Row="x" Grid.Column="x"`



The rows and columns of a `<Grid>` are counted from zero, so this `<TextBox>` is in 0, 0.

This element is an `<Ellipse>`.

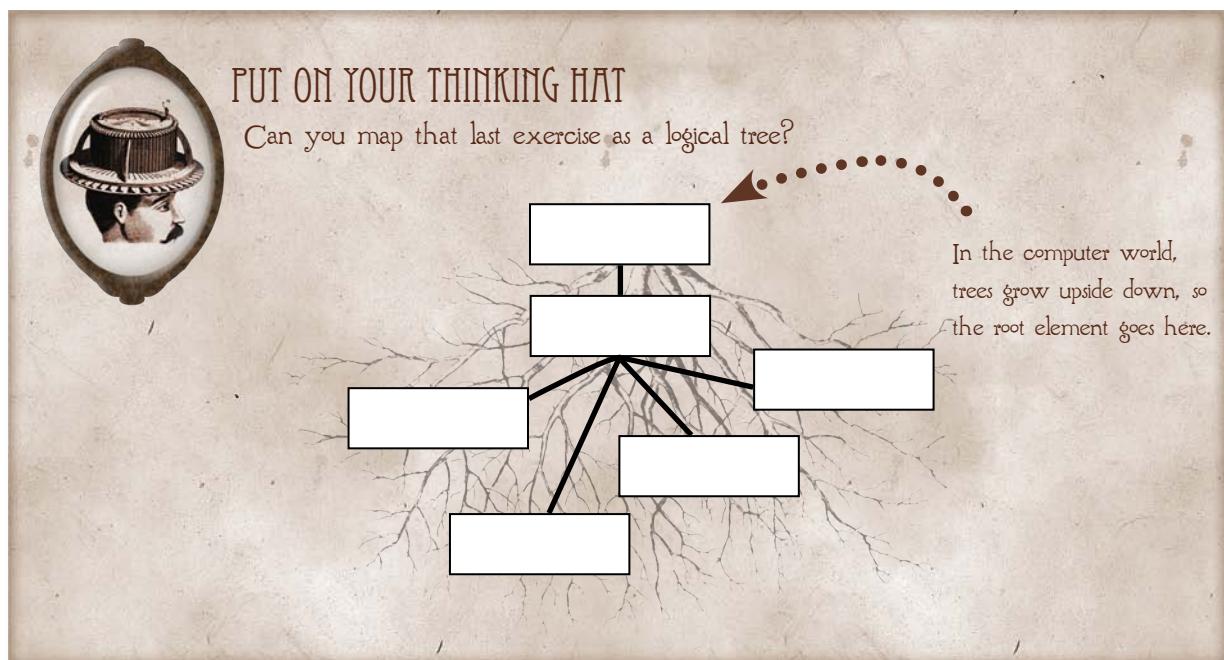
Now, for extra credit, create a new WPF project in Visual Studio and duplicate the window. Does your XAML match the code Visual Studio created?

WPF TREES

We've talked about how XAML elements are hierarchical. Programmers find it useful to think of hierarchies as trees, and in WPF there are three. The **OBJECT TREE** is probably the simplest to understand: It represents the way elements are nested in the XAML markup (or in code).

The **LOGICAL TREE** is similar to the object tree, and probably best understood by example. Logically, the parent of the items in a `ListBox` is the `<ListBox>` itself, and that's what the logical tree reflects. But in the object tree, the items are contained in the `ListBox.Items` collection. In other words, the logical tree omits most of the interim objects.

Finally, the **VISUAL TREE** contains all the elements that are displayed on the screen. Almost all of the WPF controls you'll work with are actually composed of multiple visual elements—the default `Button`, for example, includes a `Border`, a `ContentPresenter` and a `TextBlock`. (As we'll see in Chapter 18 when we examine control templates, you have a lot of control over the elements that compose a WPF control, so this isn't an absolute.)

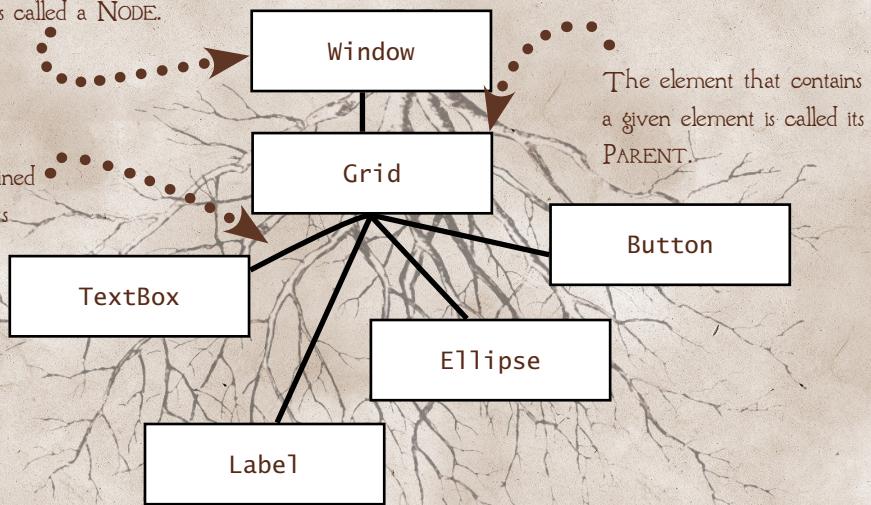


HOW'D YOU DO?



Each element in the tree is called a NODE.

Elements that are contained in a given element are its CHILDREN.



MAKE A NOTE



We're discussing logical and visual trees because you'll run across trees a fair bit in the WPF documentation, and I don't want you to be afraid of them. But in reality, you won't need to manipulate them very often unless you're creating your object tree in code or in a few special data-binding situations that we'll discuss in Chapter 22.

TRaversing THE FAMILY

Almost all of the WPF controls expose a `Parent` property that returns the element that contains them. Only a few WPF controls can contain multiple children, and those that do (they all derive from `Windows.Controls.Panel`) expose a `Children` collection. You can use these properties to navigate up and down the logical tree.

The `LogicalTreeHelper` class also exposes a static method, `FindLogicalNode()`, that returns the element with the specified name. But there's a gotcha here. You have to set the `Name` property explicitly, either as an attribute in XAML (`Name="MyButton"`) or in code (`MyButton.Name = "MyButton"`).



PUT ON YOUR THINKING HAT

The `Parent` and `Children` properties and the `LogicalTreeHelper`.
`FindLogicalNode()` method work exactly the way you'd expect them to. Can
you write the VB code that gets the following element references?

The element that contains the variable `LayoutRoot`

The element named `SomeElement`

The collection of elements contained by the element named `MyContainer`

The first child of the variable `MyGrid`



HOW'D YOU DO?

This was a real test of how well you mastered basic C# syntax.

The element that contains the variable `LayoutRoot`

`LayoutRoot.Parent`

You might have been able to use `FindLogicalNode("LayoutRoot")`, but only if the `Name` property has been set.

The element named `SomeElement`

`FindLogicalNode("SomeElement")`

Remember, in VB, you have to wrap a string in quotation marks.

The collection of elements contained by the element named `MyContainer`

`FindLogicalNode("MyContainer").Children`

Doing this in one line is tricky. Give yourself a gold star if you did it this way. You could also have called `FindLogicalNode()` to get a reference, and then called the `Children` property to get the collection.

The first child of the variable `MyGrid`

`MyGrid.Children[0]`

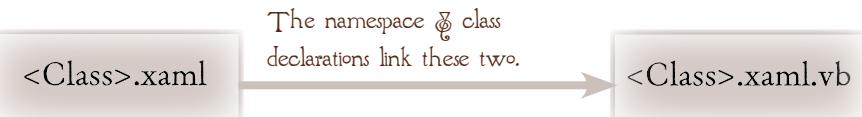
You get another gold star if you remembered how to reference an item in a `Collection` and that the first item is zero. If you didn't, don't feel bad...it takes a while for the idea that computers count from zero to sink in.



XAML & VB

Back in Chapter 1, we saw that .NET source code is compiled into *MSIL*, linked into an executable, and at runtime the CLR translates the *MSIL* into CPU-specific code. But *XAML* isn't a programming language in the sense that C# and Visual Basic are, so there are some more steps to the process. Let's take a quick look at how the process works:

1



2

The XAML file is compiled into BAML.

At compile-time, Visual Studio generates a .g.cs file that contains the InitializeComponent() method.

<Class>.xaml.vb

<Class>.g.vb

Both the .cs and the .g.cs files are compiled into MSIL.

3

BAML

The BAML file is embedded in the MSIL.

MSIL



WORDS FOR THE WISE

BAML stands for BINARY APPLICATION MARKUP LANGUAGE. To create the BAML, the XAML is parsed, tokenized and converted to binary format that's smaller and faster to load than the original XAML. You should know it exists and how it fits into the application development process, but don't worry about the exact format or contents.

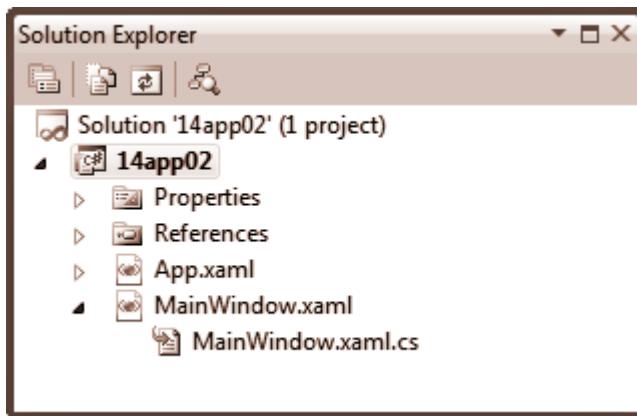


CREATE SOME XAML

You've actually been doing this in every exercise in the book, but we've been concentrating on other things, so let's create a new project and look at the files that you've actually been creating.



Create a new WPF application in Visual Studio. In the Solution Explorer, expand the `MainWindow.xaml` node. You'll see the `MainWindow.xaml.cs` file that contains the VB code-behind.



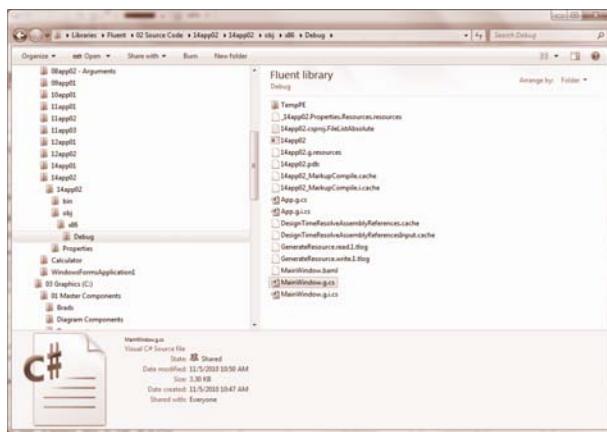
Drag a `TextBox` element from the Toolbox onto the XAML designer, and double-click the button to add an event handler. (You don't need to add code to the handler, but of course you can if you want to.)



Build the solution.



In the Windows Explorer (not the VS Solution Explorer), navigate to the folder where you put your application, and then to /obj/x86/Debug folder beneath it. Find the `MainWindow.g.vb` file.

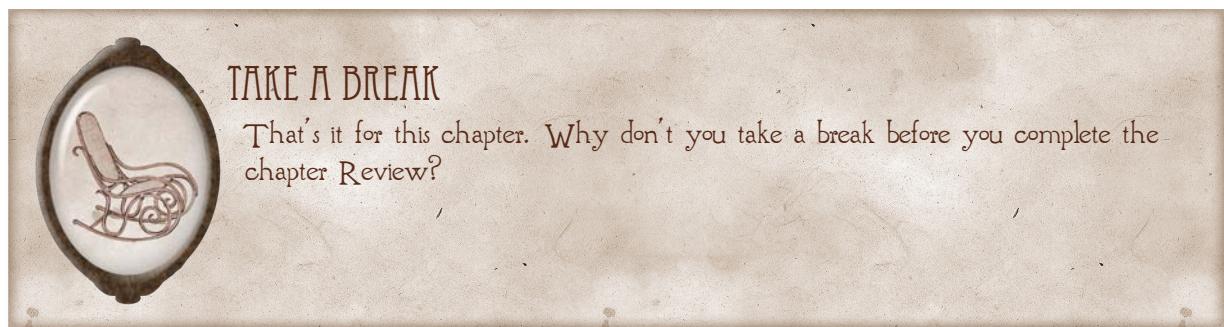
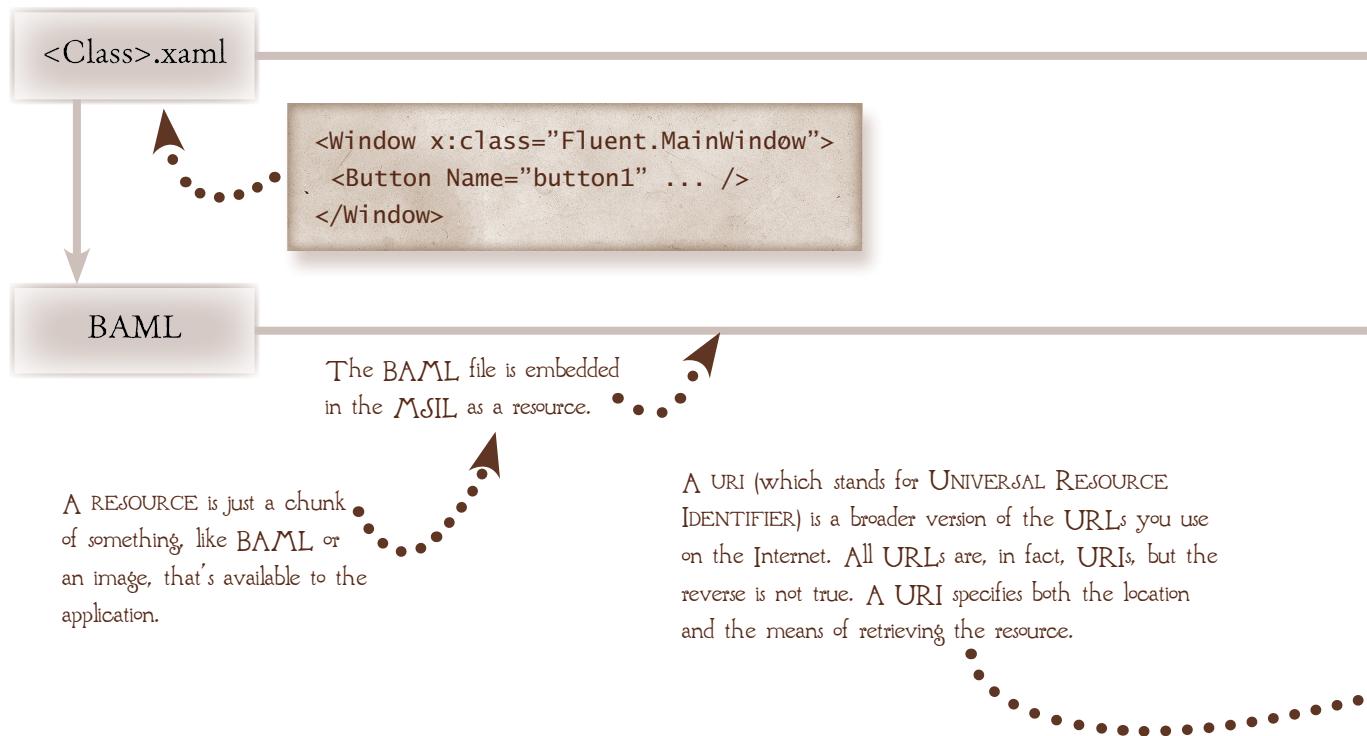


Double-click `MainWindow.g.vb` to open it in Visual Studio. The file contains a partial class definition of `MainWindow` that adds two methods, `InitializeComponent()` and `Connect()`.

What do you think they do?

BAML & VB IN DETAIL

The `.g.cs` file that's generated (the `".g"` stands for "generated") by Visual Studio extends the class declaration to include an implementation of the `IComponentConnector` interface and implements the interface's two methods: `InitializeComponent()` and `Connect()`. Here's how it works in detail.



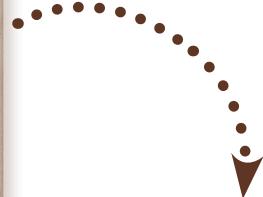
```

Public Partial class MainWindow
Inherits Window

Public Sub New()
    InitializeComponent()
End Sub
End Class

```

The class constructor calls InitializeConstructor().



<Class>.xaml.vb

<Class>.g.vb

MSIL

```

Public Partial Class MainWindow
Inherits Window
Implements IComponentConnector

Public Sub InitializeComponent()
    System.Uri resourceLocator = new System.Uri(...)
    LoadComponent(Me, resourceLocator)
End Sub

Sub Connect(int ConnectionID, object target)
    Me.button1.Click += new RoutedEventHandler(this.button1_Click)
    // code to create variables and connect events
End SUb

}

```

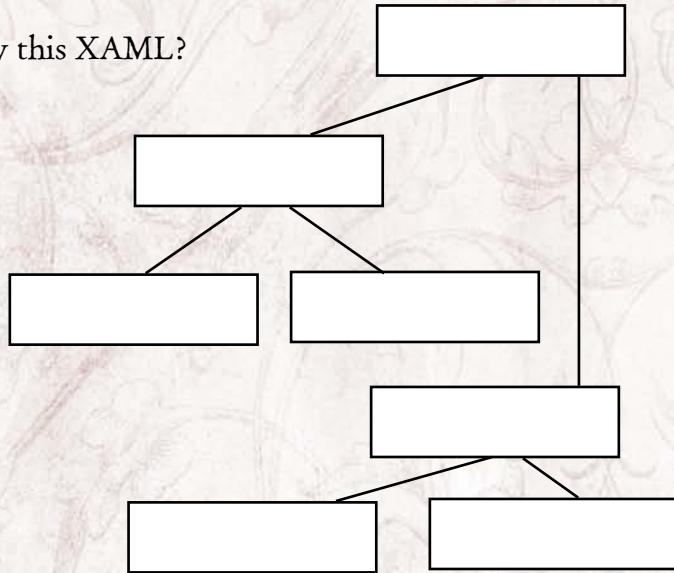
InitializeComponent() loads the BAML resource and then calls the LoadComponent() method, which in turns calls Connect() to connect the event handlers and names.



REVIEW

Can you draw the logical tree created by this XAML?

```
<Window Name="MainWindow">
<Window.ContextMenu>
  <ContextMenu>
    <MenuItem Header="Item1" />
    <MenuItem Header="Item2" />
  </ContextMenu>
</Window.ContextMenu>
<StackPanel Name="LayoutRoot">
  <Label>Say Something</Label>
  <Button>Do Something</Button>
</StackPanel>
</Window>
```



In the XAML markup above, find examples of attribute, property and collection syntax.

You have to do three things to make sure that the CLR can connect XAML and the code-behind. Can you pick them from the list below?

Call the `Connect()` method in the class constructor.

Make sure both files exist in the same namespace.

Set the `Name` property of objects in code.

Set the `Class` attribute of the XAML root element.

Give XAML elements and variables in the code the same name.

Include a call to `InitializeComponent()` in the class constructor.

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



WPF CONTROLS

16

Now that you've learned how to write XAML to define your user interface, it's time to look at the widgets that WPF provides for building your applications. Like any object, WPF controls provide state and behavior, but what sets WPF apart from most UI platforms is that their appearance is only a default.

Microsoft calls WPF controls "lookless," which is a little misleading. It's not that they don't have a look, but rather that their look can be completely changed. As we'll see in Chapter 21 when we examine control templates, you can completely change the structure and appearance of any WPF control. Want a 3-dimensional image of an elephant that turns into a mouse when it's selected and behaves like button? No problem. The WPF Button control can do that. Want a combobox that opens right to left and displays checkboxes and images? Easy.

But before you can start restyling and rearranging the appearance of your WPF widgets, you need to understand the basic set of behaviors they offer, and that's what this chapter is about. We'll look at how the Control object hierarchy is organized, and take a quick spin through all the basic control types: panels like the Grid that arranges multiple elements, Content controls like the Label and Button that contain a single element, and Item Controls like the ListBox that present a collection of elements.

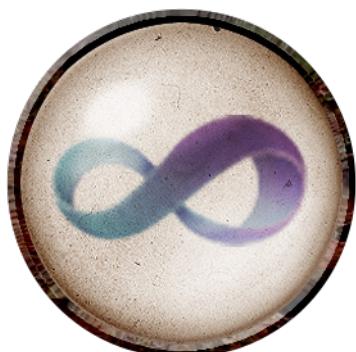
MAKE A NOTE



This is a really long chapter; the longest in the book. Don't feel like you need to finish it in one sitting. Take breaks whenever you feel tired or find your attention wandering.



FITTING IT IN



.NET PLATFORM COMPONENTS

.NET SERVERS

SQL Server

IIS

Biztalk Server

SharePoint Server

.NET TOOLS

Visual Studio

Expression Studio

Framework SDK

.NET FRAMEWORK

Specialty Frameworks

Compact

Micro

Windows Phone

Class Library

CLR

Languages

CLASS LIBRARY

FCL TECHNOLOGIES

BASE CLASS LIBRARY

USER INTERFACE TECHNOLOGIES

Window Presentation Foundation

Other UI Technologies

System.Windows

WPF is defined in
the `System.Windows`
namespace.

FrameworkElement

Panel controls like the
`Grid` and `DockPanel`
can contain multiple child
elements.

Panel Controls

Items controls like the
`ListBox` and `Menu`
contain a single collection of
items.

Range controls like the
`ProgressBar` and
`ScrollBar` display a value
within a specified range.

Control

Content controls display
a single item of content.
Examples include the
`Label` and `Button`.

Items Controls

Content Controls

Range Controls

Other Controls

BUILDING BLOCKS

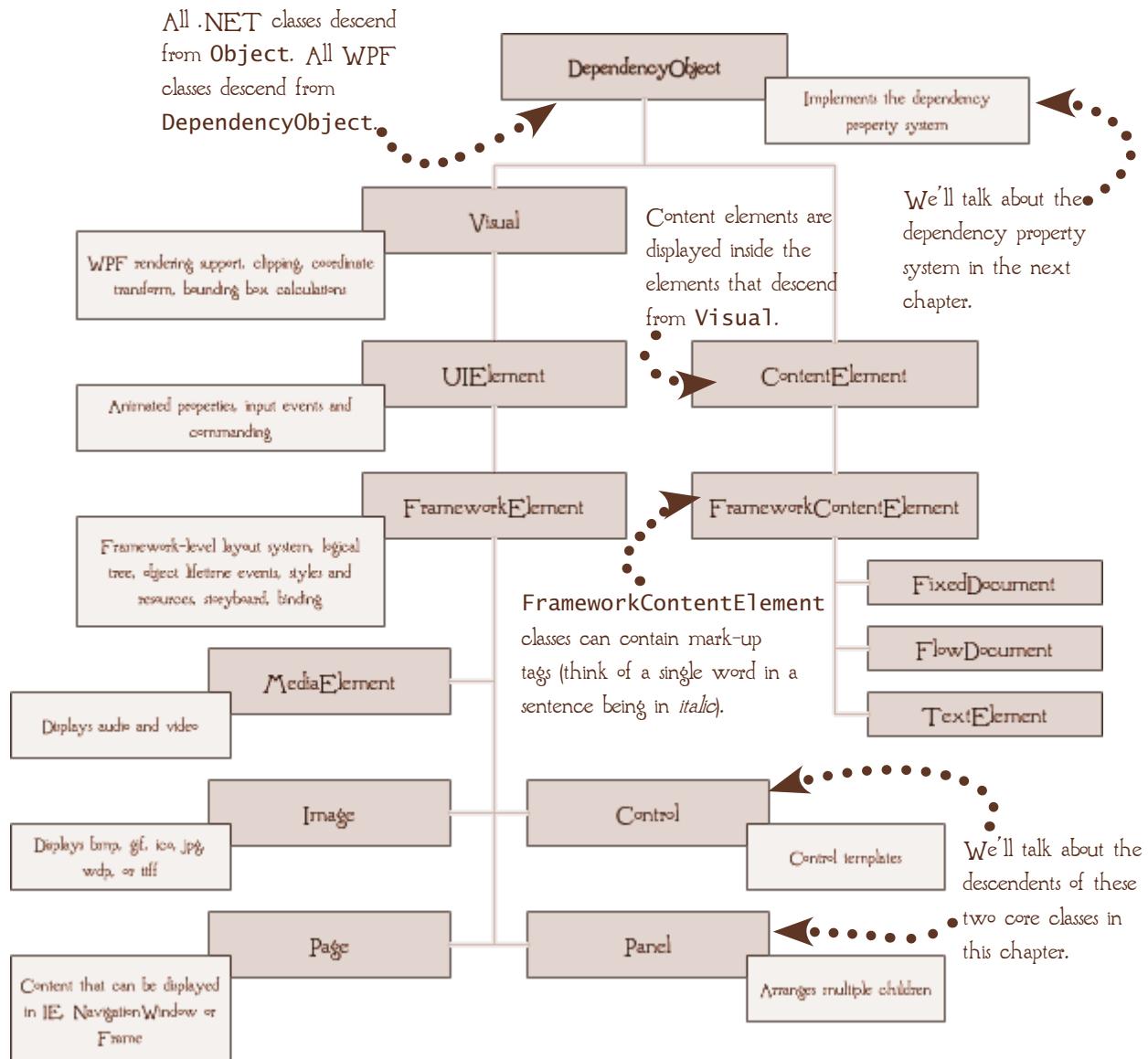
Just like our chefs can take basic components and combine them to make an infinite number of dishes, you'll use the basic widgets provided by the WPF to build almost any kind of applications, at least any kind of application that has a Windows user interface.

"Give us some flour, sugar and eggs,
and we can make you some bread,
or sweet rolls or cookies, or..."



A ROAD MAP

There are a bazillion classes in WPF, and it's easy to get lost without a map, but fortunately we have one in the WPF class hierarchy. Once you know where a given class sits on the tree, you know a lot about how the class works and what it does.





TASK LIST

In this chapter we'll explore some of the basic kinds of control widgets. We won't go into a lot of depth—you could spend the rest of your life exploring WPF—but we'll see enough to get you started.



WPF PANELS

Panel controls are the basis of most of your layouts because they're the widgets that know how to arrange multiple children. You've been using the Grid, but there are other panel controls that lay out their children in various ways, and we'll look at three of the most important: the Canvas, DockPanel and Grid.



CONTROL CLASSES

The `System.Windows.Control` class is the base for most of the widgets you'll use to build your interface. After we've looked at the panel options, we'll explore the way these classes are organized.



CONTENT CONTROLS

Content controls are an important branch of the `Control` class hierarchy. This branch includes everything from the `Window` to the `Button`. In theory, a content control can contain a single element of content, but as we'll see, that's a little misleading since the content can be another content control (which contains another element, which contains another element, and so forth) or even a panel that contains as many elements as you need.



ITEMS CONTROLS

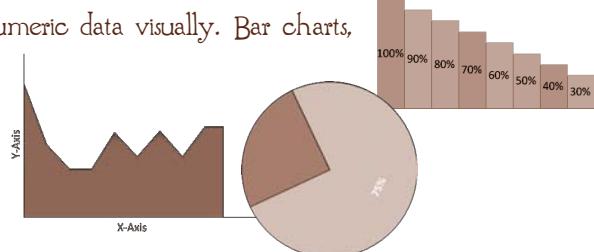
Finally, we'll examine the other important branch of the `Control` class hierarchy: the items controls that contain a collection of items. The items controls group includes the `ListBox`, `ComboBox` and `TabControl` that are so useful for managing the real estate of your application, and we'll see how some of those work.



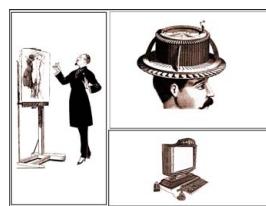
WPF PANELS

There are lots of ways to present numeric data visually. Bar charts, pie charts and line charts are all valid methods, and they each emphasize different aspects of the information.

In the same way, WPF panels can display the same set of widgets, but they're each useful in different situations.



The **Grid** organizes its contents in rows and columns.



The **StackPanel** stacks its contents, either horizontally or vertically.



We won't be discussing these two.

The **DockPanel** arranges its elements relative to the edges of the panel.



The **WrapPanel** organizes its contents either horizontally or vertically, and wraps them from one line to another.

The **Canvas** places its elements at specified coordinates relative to the edges of the panel.



ON YOUR OWN

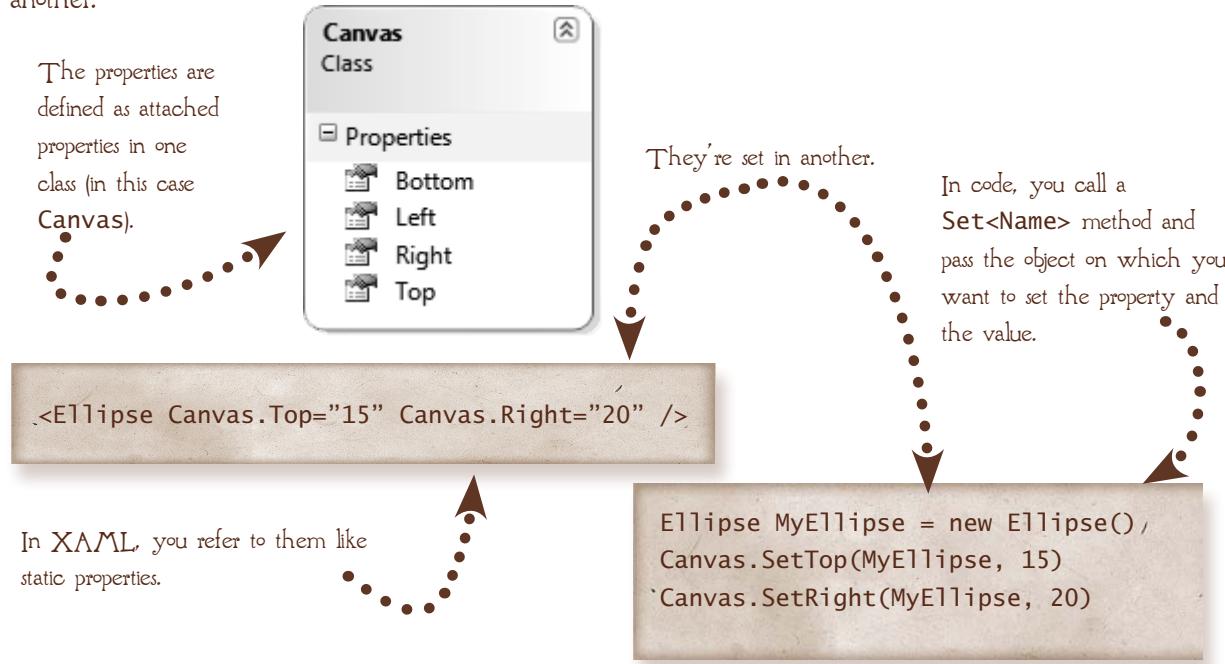
Can you think of situations in which each of the panels might be useful as a layout root?



ATTACHED PROPERTIES

You've been working with the properties of objects for chapters and chapters, and you're probably pretty familiar with the way they work: A property is just a bit of state with a name that you can access. But WPF adds a lot of functionality to the basic concept of a property (like the ability to be computed) and we'll look at that when we study dependency properties in the next chapter. But one aspect of the WPF property system is attached properties, and they're important for (among other things) working with panels.

The theory behind attached properties is a little complex, and frankly, not all that important for our purposes. In practice, they're easy to use. Basically, attached properties are defined in one class but set in another:



MAKE A NOTE

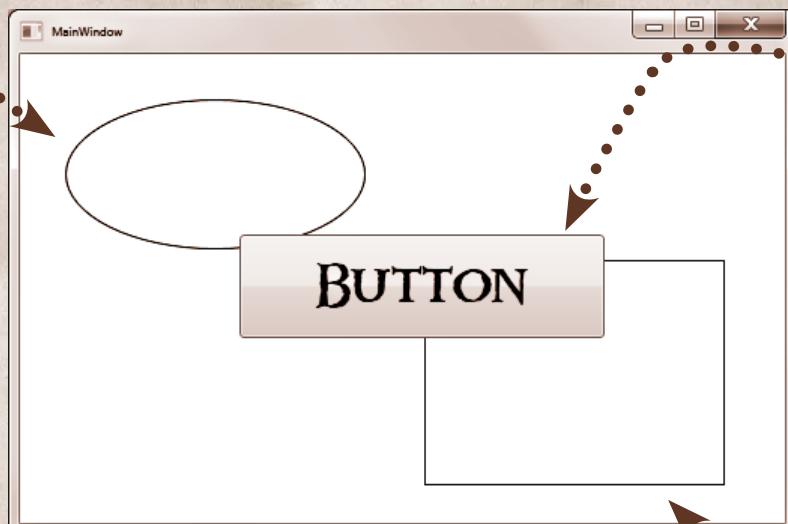
Panels aren't the only place attached properties are used in WPF. For example, ToolTips (those little boxes that open up when you hover over something) are implemented using attached properties. But panels are where you'll use them most often.



PUT ON YOUR THINKING HAT

In many ways, the Canvas is the simplest of the panels to understand. You specify a position relative to one of the sides of the panel, and that's where the elements sit. End of discussion. To see just how simple the Canvas is to use, write the XAML to reproduce this window:

The **Ellipse** is 30 from the left, 30 from the top, and has a **Height** of 100, a **Width** of 200 and a **Stroke** of Black.



The **Button** is 145 from the left, 120 from the top, 69 high and 243 wide.

The **Rectangle** is 40 from the right, 25 from the bottom, 150 high and 200 wide. Its **Stroke** is Black.

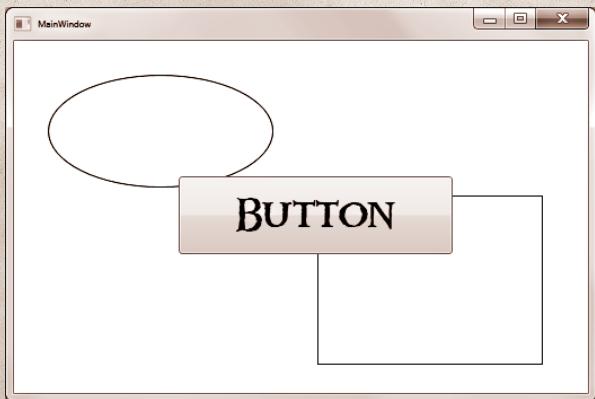


HOW'D YOU DO?

Did you remember to put the values in quotation marks?

```
<Canvas>
  <Ellipse Canvas.Left="30" Canvas.Top="30" Height="100" Width="200" Stroke ="Black" />
  <Rectangle Canvas.Right="40" Canvas.Bottom="25" Height="150" Width="200" Stroke="Black" />
  <Button Canvas.Left="145" Canvas.Top="120" Height="69" Width="243" Content="Button"/>
</Canvas>
```

Don't forget the closing tag!



Did you work out how to specify the **Height**, **Width** and **Stroke** properties? Well done if you did!

It's okay if you forgot the **Content** property; it wasn't part of the exercise.



UNDER THE MICROSCOPE—UNITS IN WPF

The default unit of measurement in WPF is the Device Independent Pixel, an abstract unit equal to $1/96$ th of an inch (in other words, there are 96 dips to an inch, the pixel resolution of most monitors). Using an abstract unit like this allows WPF graphics to scale correctly when the user changes the hardware dpi setting of their monitor. References to device independent pixels are stored as **Double** values.

You can specify other units of measurement in XAML by using a **QualifiedDouble**, which is simply a numeric value with a unit specifier, for example, "**12in**" for inches or "**12cm**" for centimeters. The available qualifiers are "**px**" (device independent pixel), "**in**" (inch), "**cm**" (centimeter) and "**pt**" (point).



ON YOUR OWN

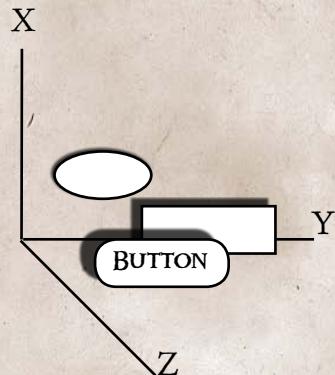
Create the window from our last exercise in Visual Studio, and try these experiments.

The Button appears on top of the Ellipse and the Rectangle. What happens if you change the order of the elements inside the Canvas?

In the Properties window, the Button shows a property called `ZIndex` that takes a numeric value. Try setting that value. What object defines `ZIndex`? (Hint: Look at the XAML.)

Try playing around with the `ZIndex` values of the various objects. How does `ZIndex` work?

Run the application and resize the window. What happens? Why doesn't the Rectangle behave the same way as the Ellipse and the Button?



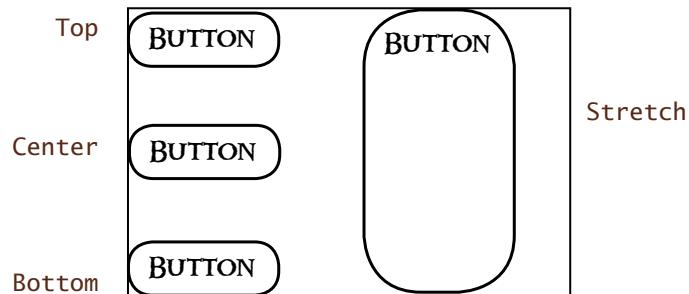
PANEL POSITIONING AND SIZING

Canvas is the only one of the standard panels that sizes and positions its children absolutely. The remaining panels adjust their children based on available space and the layout properties that are defined in **FrameworkElement**.

VERTICAL ALIGNMENT

The **VerticalAlignment** property accepts an enumeration whose values are:

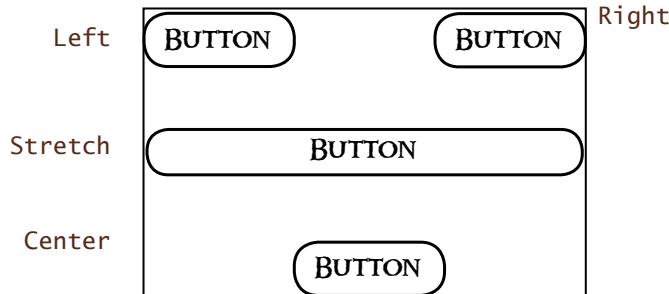
Top
Center
Bottom
Stretch (default)



HORIZONTAL ALIGNMENT

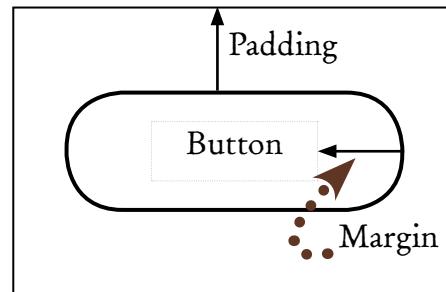
The **HorizontalAlignment** property accepts an enumeration whose values are:

Left
Center
Right
Stretch (default)



MARGIN & PADDING

The distance between elements is controlled by two properties: Margin determines the distance between an element and its container, while Padding determines the distance between an element and its children.



HEIGHT & WIDTH

FrameworkElement exposes eight (!) properties to determine the height and width of an element:

- Height & Width are actually the size an element would like to be, if there's enough room.
- MinHeight & MinWidth are the smallest size the element can have and still display itself.
- MaxHeight & MaxWidth are the largest size the element can reasonably have.
- ActualHeight & ActualWidth are the sizes assigned by the panel.

The ActualHeight and ActualWidth properties are set by the panel and are read-only. The others can be set using any of the unit qualifiers ("in", "cm" and so forth) or with the special value Auto. The Auto value generally means that the element will expand to fill the available space, but as we'll see in Chapters 20 & 21, this is actually controlled by the control template and style.

SPECIFYING MARGINS AND PADDING

The Margin and Padding properties are actually instances of the Thickness class that has four properties: Left, Right, Top, Bottom. There are several ways to specify a Thickness:

```
<Element Margin="Left&Right, Top&Bottom" />
```

or

```
<Element Padding="Left, Top, Right, Bottom" />
```

or

```
<Element>
  <Element.Margin>
    <Thickness Left="left" Right="right" Top="top" Bottom="bottom" />
  </Element.Margin>
</Element>
```



PUT ON YOUR THINKING HAT

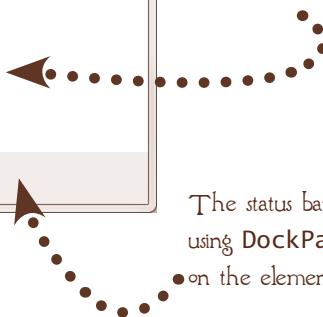
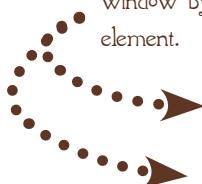
Using the alignment and sizing properties, how would you specify a TextBox element that's centered vertically, 20 pixels from the left and right, and at least 100 pixels wide?

THE DOCK PANEL

By default, Visual Studio sets up a new window with a Grid as the layout root. I almost always change that to a DockPanel, which seems to me perfectly suited to the conventional layout of any window, with a menu bar and a toolbar at the top, a status bar at the bottom, and a client area filling the remaining space.



The menu and toolbar are docked to the top of the window by setting `DockPanel.Dock="Top"` on each element.



Setting the `LastChildFill="True"` property on the `DockPanel` itself causes the last element to take up any remaining space.

The status bar is docked to the bottom using `DockPanel.Dock="Bottom"` on the element.



HOW'D YOU DO?

• The `VerticalAlignment` trumps the top and bottom margin, so we can just set it to 0.

```
<TextBox VerticalAlignment="Center" Margin="20,0" MinWidth="100" />
```

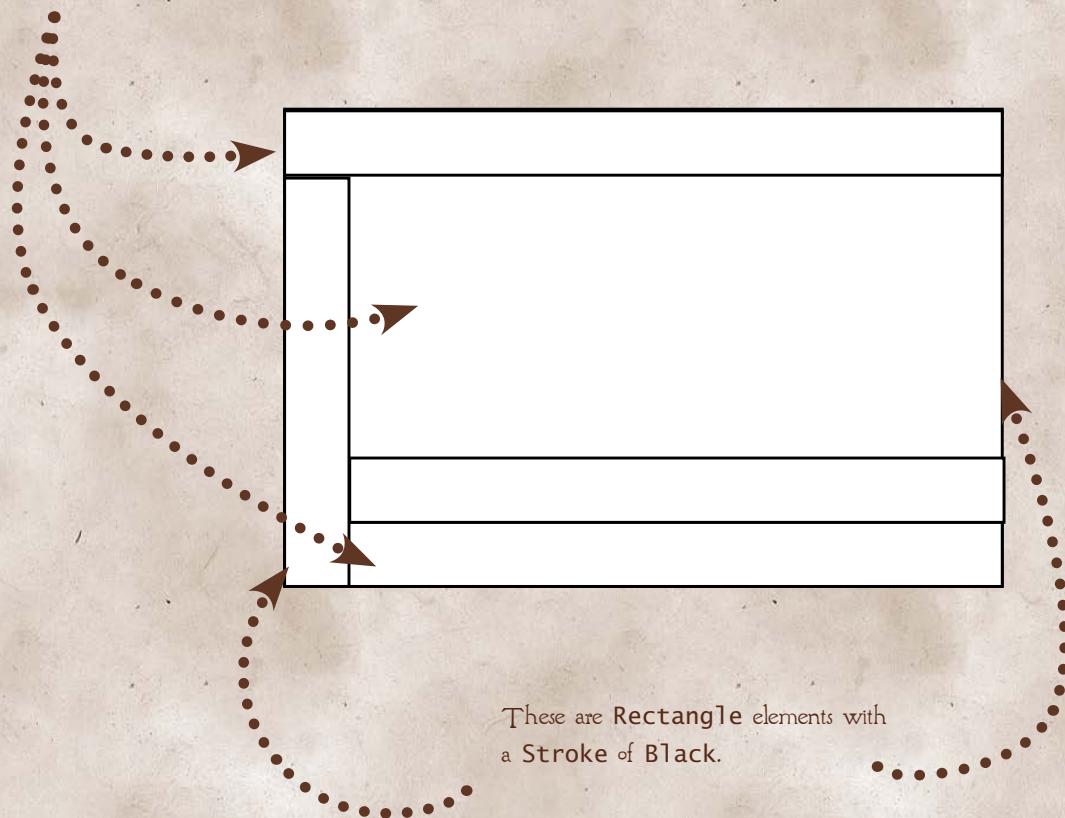


PUT ON YOUR THINKING HAT

It's probably obvious by now how you can use the Dock attached property to arrange the children of a DockPanel, but look carefully at the example below. The TextBox at the very top runs all the way to the edges of the panel, but the TextBox at the bottom is interrupted by the Rectangle at the left.

Try building the window in Visual Studio. Can you work out what controls how the child elements are stacked?

These are **TextBox** elements.



These are **Rectangle** elements with
a **Stroke of Black**.



HOW'D YOU DO?

The DockPanel allocates space on a strictly first-come, first-serve basis. The first child element receives the full width or height of the panel. Subsequent child elements are allocated the full width or height of the remaining space. Elements docked to the same side are stacked.

The first element gets all the available side.

Subsequent elements are allocated space from what's left over.



If `LastChildFill` is set to "False", the last element behaves like any other. But remember, the default values for `HorizontalAlignment` and `VerticalAlignment` are "Stretch", so if you don't set them, the element will fill the available space anyway.

Elements docked to the same side are stacked.



MAKE A NOTE

Unlike Canvas, the `ZIndex` property has no effect on the arrangement of elements in the DockPanel—only the order in which the elements are added controls their arrangement.

THE GRID

The **Grid** is the default layout root when you create a window in Visual Studio. Like a spreadsheet, the **Grid** divides the available space into **rows** and **columns**. Those **rows** and **columns** are defined using the **Grid.RowDefinitions** and **Grid.ColumnDefinitions** collections:

You add one **RowDefinition** element for each row and a **ColumnDefinition** for each column in the **Grid**.

These **ColumnDefinition** elements assign two-fifths of the available height to the first column, and three-fifths to the second.

```
<Grid>
<Grid.RowDefinitions>
  <RowDefinition Height="20"/>
  <RowDefinition />
  ...
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="2*"/>
  <ColumnDefinition Width="3*"/>
  ...
</Grid.ColumnDefinitions>
</Grid>
```

The **Height** property of a **RowDefinition** and the **Width** property of a **ColumnDefinition** can be specified using any of the qualified doubles, as "Auto", or as a proportional value using the star syntax.



PUT ON YOUR THINKING HAT

The **Grid** panel defines **Row** and **Column** attached properties that allow you to assign a child element to a specific cell. In addition, it defines **RowSpan** and **ColumnSpan** attached properties that allow an element to stretch across more than one cell.

Can you re-create the diagram on the previous page using a **Grid**? (Hint: Like most things in the .NET world, the rows and columns in a **Grid** start at zero.)

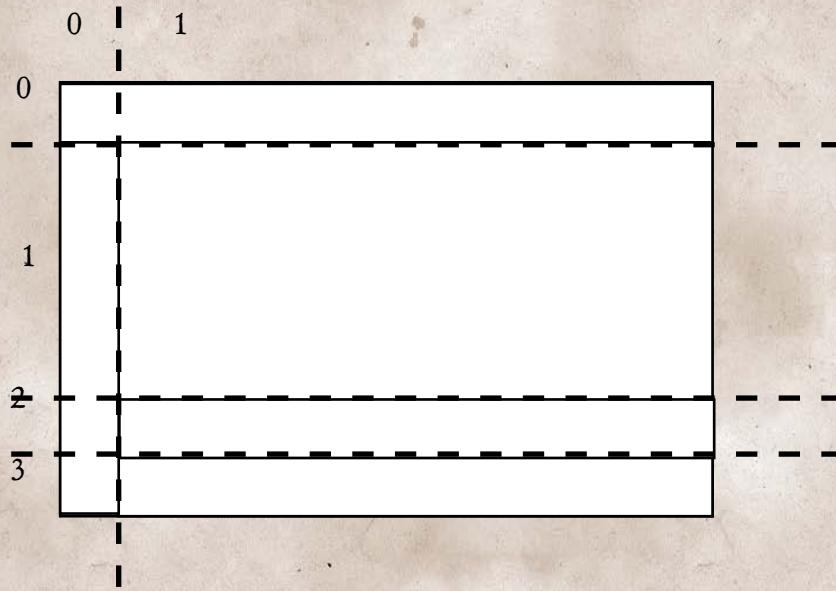


HOW'D YOU DO?

It doesn't matter what actual values you assigned to the heights and widths, as long as you got the basic structure right.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="40" />
    <RowDefinition Height="*" />
    <RowDefinition Height="40" />
    <RowDefinition Height="40" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="40" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Rectangle Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"/>
  <Rectangle Grid.Row="1" Grid.Column="0" Grid.RowSpan="3" />
  <Rectangle Grid.Row="2" Grid.Column="1" />
  <Rectangle Grid.Row="3" Grid.Column="1" />
</Grid>
```



THE GRIDSPLITTER 1-2-3

Spreadsheets like Microsoft Excel are obviously grids, but even things like Windows Explorer have identifiable grid components. Most of those windows allow you to change the relative size of the panes by clicking and dragging. The tool for adding that functionality in WPF is the `GridSplitter`. Most discussions of the `GridSplitter`, including those on MSDN, make it sound a lot more difficult to use than it actually is, because they set a bunch of properties that are largely irrelevant for the most usual case. So, here's my three-step process for making a row or column of a grid resizeable using the `GridSplitter` control:



CREATE A ROW OR COLUMN DEFINITION TO CONTAIN THE GRIDSPLITTER

Set the `Width` property of the row or column to the width of the splitter. You can make it anything you want, but "5" is usually a good place to start. You can put the control in a row or column with other elements, but making sure the `GridSplitter` is displayed correctly can get tricky. Why make things more difficult for yourself?



ADD THE GRIDSPLITTER TO THE GRID

You'll use the same syntax you'd use for any other element, setting the `Row`, `RowSpan`, `Column` and `ColumnSpan` properties so that they sit where you want them to. Be sure to explicitly set the `Width` property. By default, the `GridSplitter` has a `Width` of zero, which makes it impossible to grab with the mouse.



SET THE RESIZEBEHAVIOR PROPERTY

You'll almost always want to set `ResizeBehavior` to "`PreviousAndNext`", which tells the `GridSplitter` to resize the rows or columns on either side of it.

MSDN will tell you to set `HorizontalAlignment` or `VerticalAlignment` to control how the `GridSplitter` behaves, but that's a bit obscure. If you set `ResizeBehavior` explicitly, the code becomes self-explanatory, which is always better than "this property does more than you think, and you only know that because somebody told you."



PUT ON YOUR THINKING HAT

If you haven't already done so, create the Grid on the previous page in Visual Studio. (You'll probably want to add a `Stroke` to the `Rectangle` elements so you can see them.) Then add a `GridSplitter` that divides the first and second column, from row 1 through 3. (Leave the first row spanning the whole window.)



HOW'D YOU DO?

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="40" />
    <RowDefinition Height="*" />
    <RowDefinition Height="40" />
    <RowDefinition Height="40" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="5" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Rectangle Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"/>
  <Rectangle Grid.Row="1" Grid.Column="0" Grid.RowSpan="3" />
  <Rectangle Grid.Row="2" Grid.Column="2" />
  <Rectangle Grid.Row="3" Grid.Column="2" />
  <GridSplitter Grid.Row="1" Grid.Column="1" Grid.RowSpan=3"
    Width="5" ResizeBehavior="PreviousAndNext" />
</Grid>
```



UNDER THE MICROSCOPE

Another useful property of the `GridSplitter` is `ShowsPreview`. `ShowsPreview` determines whether the rows or columns controlled by the `GridSplitter` are actually changed as the user drags it, or simply shows a preview of the change until the user releases the mouse.

`ShowsPreview` defaults to `False`, which is fine for most situations, but if there's a noticeable delay in updating the display, you'll probably want to change it to `True`.

NESTING PANELS

You've seen that the WPF panel controls are responsible for displaying a collection of child elements. But has it occurred to you that those child elements can, in turn, be panels? Look at the window below. It's a prototype of what the Recipe system might look like, and it has panels inside panels inside controls...and that's pretty typical of how a real-world window will be built.

In this cell of the **Grid** is
a **ScrollView** with a
StackPanel nested inside it.

The layout root is a
DockPanel, and the
Menu, **Toolbar**,
StatusBar and
TreeView are
docked to its edges.

This is a
TreeView docked
to the left of the
Window.

This is actually a
Grid nested inside a
ScrollView, nested
inside a **Grid**, nested
inside the **DockPanel**.
Whew!

The last child of the **DockPanel** is
a **Grid** with 4 rows and 3 columns.

A **StackPanel** just stacks
up its children.

This **Label**
spans both
columns.



A **ScrollView** displays a scrollbar
if the contents don't fit.

TAKE A BREAK

That's it for the first section. Panels can get complicated, as you can see from this example, but they're really easy to use, and you'll get lots of practice as we move through the rest of the book. For now, why don't you take a break before you complete the Review and we move on to the WPF control classes?





REVIEW

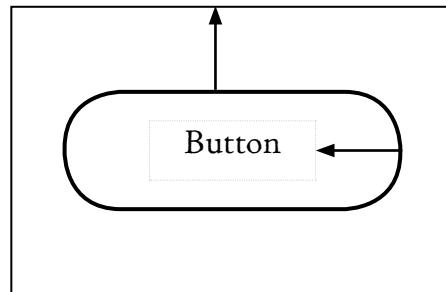
How does each type of standard panel control the relative position of its children?

Grid:

DockPanel:

Canvas:

Label each of the arrows on this diagram with the property that controls that space.



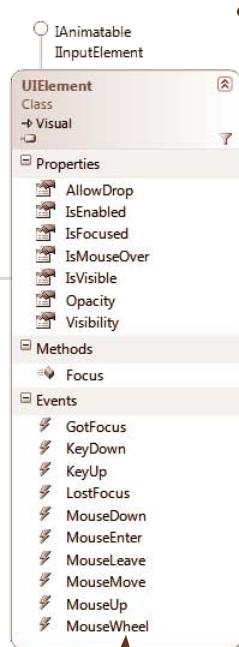
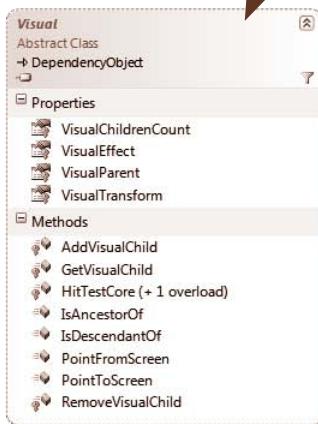
The layout of the Windows Explorer is a common structure used in many applications. What WPF panels would you use to build it? Don't worry about the contents, just the panel controls you'd use to create the panes. (Hint: One of the panels is a standard panel we didn't discuss, but it's illustrated on page 537.)



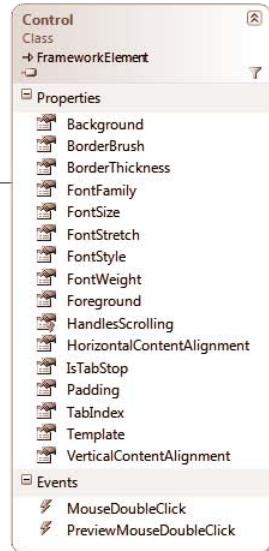
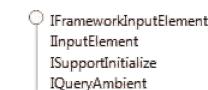
CONTROL CLASSES

There's a road map to the core WPF classes on page 535. Before we get into the details of the classes that descend from `Control`, let's take a quick look at `Control` itself and its ancestor classes.

`Visual` exposes the properties and methods that control display.



`FrameworkElement` exposes the methods and properties that support the layout system and lifetime events like `Initialized` and `Loaded`.



`UIElement` exposes all the methods and properties that allow a WPF element to interact with the user via the keyboard, mouse, stylus or touch screen. There are a *lot* more than are shown here—this is just a sampling.

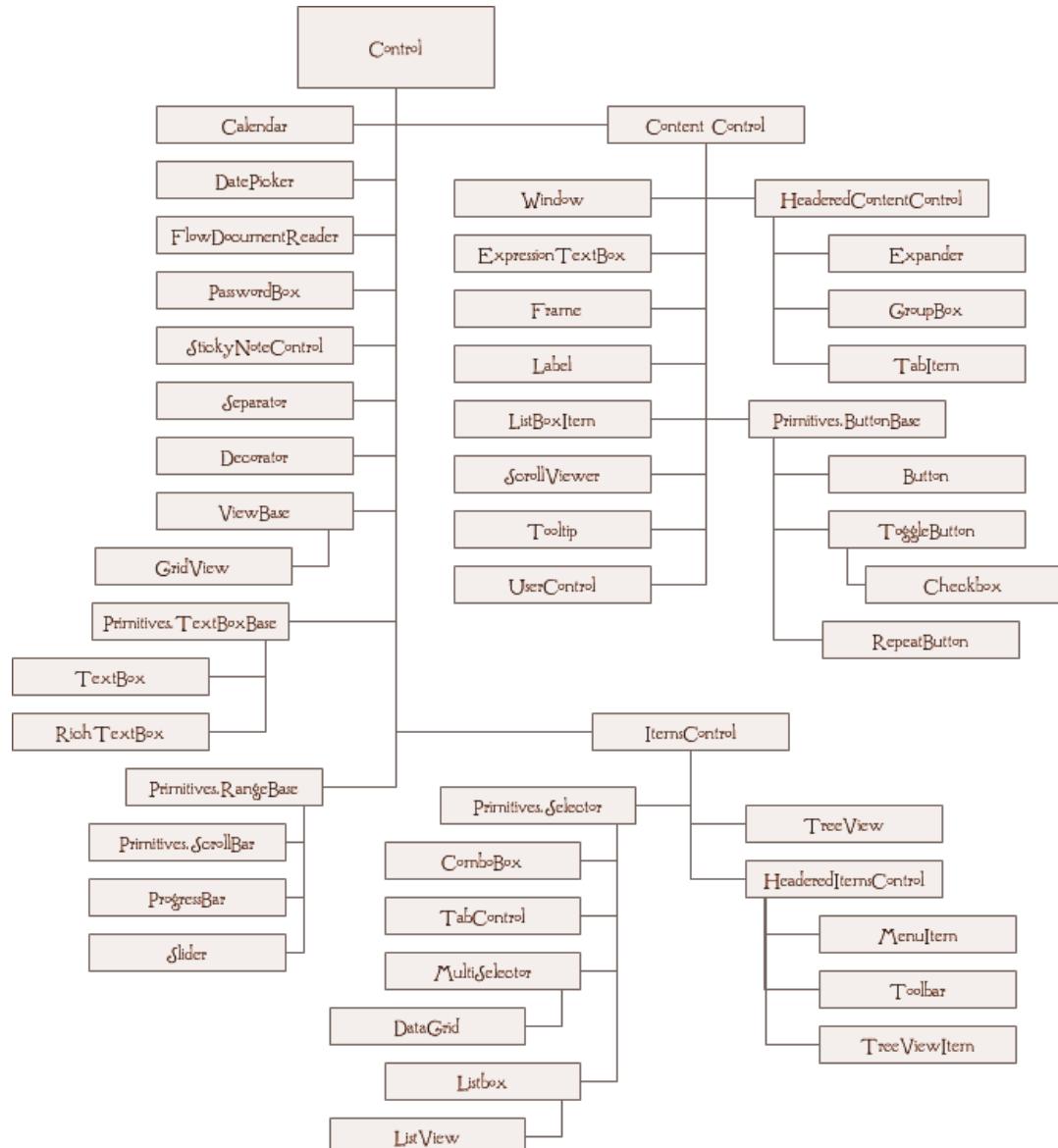


Most of the widgets you'll use to build your application interface descend directly or indirectly from `Control`, (including the ones we'll be discussing the rest of this chapter). The `Control` class adds format and layout properties and also support for styles and templates, which we'll discuss in Chapters 20 and 21.



THE CONTROL FAMILY

Remember when I said that most of the widgets you'll use to build your applications will descend from `Control`? Here's a (partial) hierarchy. There sure are a lot of them, aren't there? But it's not as bad as it could be, because of the way the hierarchy works. Keep reading...





PUT ON YOUR THINKING HAT

Knowing the basic capabilities provided by `FrameworkElement` and `Control` gives you a head start on learning to use any of the WPF controls. Take the partial class diagrams below, for example. The `DatePicker` (which is like a combobox for displaying dates) and the `Calendar` (which is exactly what it sounds like), only expose properties and events that pertain to their own behavior. All the stuff that's common to all widgets is somewhere up the tree.

Using the class diagrams, can you write these VB and XAML snippets?

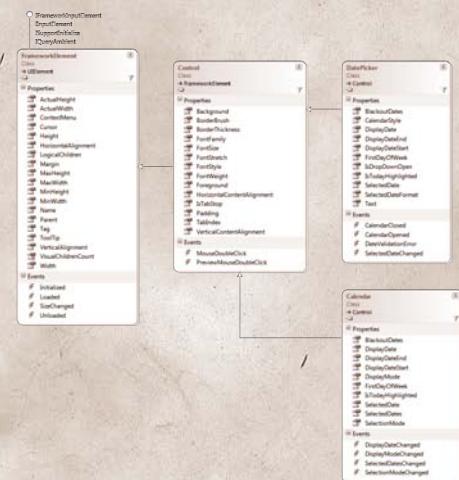
In VB, write an if statement that displays a `MessageBox` if the actual height of `MyDatePicker` is less than the requested height.

In XAML, define a `Calendar` that displays its text using the Times New Roman `FontFamily`.

In VB, use a `MessageBox` to display the `Name` property of `MyCalendar`.

In VB, write a function that changes the background color of any control to the `Beige` static property of the `Colors` class.

In XAML, define a `DatePicker` that has a `Foreground` color of `AliceBlue`.





HOW'D YOU DO?

Were you surprised that you knew so much about these two controls?

In VB, write an if statement that displays a MessageBox if the actual height of MyDatePicker is less than the requested height.

```
If MyDatePicker.ActualHeight < MyDatePicker.Height Then  
    MessageBox.Show("Didn't get as much as I asked for")
```

In XAML, define a Calendar that displays its text using the Times New Roman FontFamily.

```
<Calendar FontFamily="Times New Roman" />
```

In VB, use a MessageBox to display the Name property of MyCalendar.

```
MessageBox.Show(MyCalendar.Name)
```

In VB, write a function that changes the background color of any control to the Beige static property of the Brushes class.

```
Public Sub ChangeBackground(Control ctrl)  
    ctrl.Background = Brushes.Beige  
End Sub
```

The Brushes class is implemented in **System.Windows.Media**. It defines the standard colors as static properties. XAML knows what you mean if you just refer to the color name, but in C#, you have to reference the class itself.

In XAML, define a DatePicker that has a Foreground color of AliceBlue.

```
<DatePicker Foreground="AliceBlue" />
```



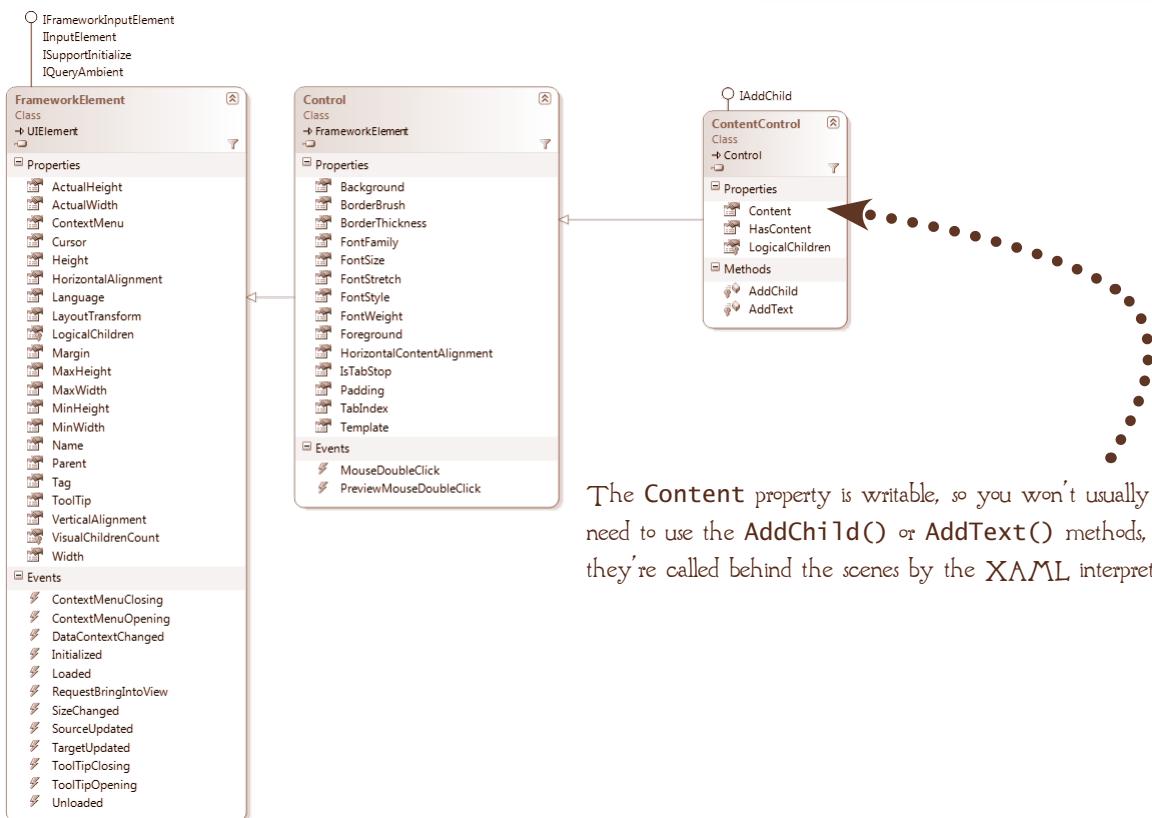
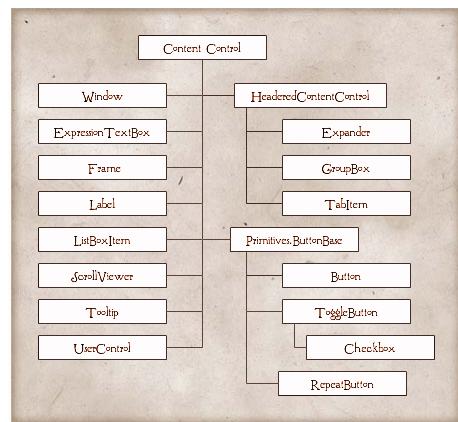
ON YOUR OWN

Do you know (or can you guess) what all the members of the classes shown in the diagram represent? If there are any that seem obscure, check them out in MSDN.



CONTENT CONTROLS

One of the most important branches of the WPF control class hierarchy is represented by the content controls that descend, appropriately enough, from `ContentControl`. Unlike a panel, which can contain multiple children, a content control can contain a single piece of content, but that content can be of any type (even a panel).



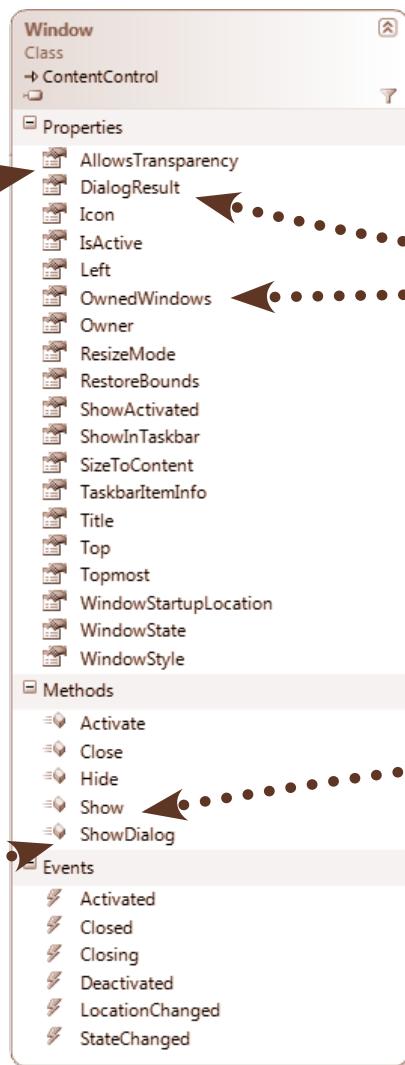
The `Content` property is writable, so you won't usually need to use the `AddChild()` or `AddText()` methods, but they're called behind the scenes by the XAML interpreter.

THE WINDOW

One very important, and slightly odd, content control is the `Window`. How is it odd? Glad you asked. The `Window` is odd because it's the only control that can't be hosted inside another control. That makes sense if you think about how UIs work (how could you put a `Window` inside a `Button`?) but it does violate strict OOP since it adds restrictions that don't exist in the parent (`ContentControl`).

`AllowsTransparency` lets you create nonrectangular windows. How cool is that?

The `ShowDialog()` method opens a window modally, which means that the user can't interact with the application until that window is closed. (The `MessageBox` is a modal window.) Such windows are called dialogs, and along with the `DialogResult` property, they let you create a dialog window that asks the user a question.



To return a result from a `Window` opened with `ShowDialog()`, you set the `DialogResult` property before you call `Close()`.

`OwnedWindows` and `Owner` let you create applications with multiple windows.

The `Show()` method displays a "normal", nonmodal window that is displayed in the Windows taskbar.

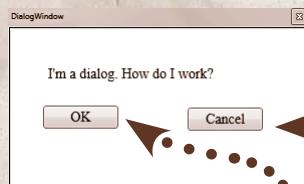


PUT ON YOUR THINKING HAT

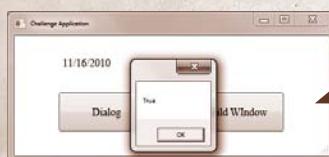
Here's a real challenge for you. Can you create the little application shown below? This is a pretty typical programming situation: You know what you need to do, and you know it's possible, but you're not entirely sure how to go about it. So you look at the objects, examine their properties and methods, maybe check around for some sample code somewhere. (Looking at sample code isn't cheating. Every programmer

I know looks at sample code A LOT. That's why there are so many programming "cookbooks" on the market.)

When the window opens, it displays `DateTime.Today` in this label. (Hint: Set the **Content** in the window constructor.)



The window opened with `ShowDialog()` has a **ToolWindow** style. It doesn't display in the Windows task bar, can't be re-sized and always stays on top of any other window.



These buttons set the **DialogResult** to **True** or **False**, respectively, and then close the window. The **DialogResult** is displayed in a **MessageBox** by the main window.



Clicking one of the buttons opens another window.



Clicking the button displays the window's **Owner.Title** in a **MessageBox**.



This is the main application window. It has a **ThreeDBorderWindow** style and displays a resize grip. It always opens in the center of the screen.

This is a resize grip
The child (owned) window has a **SingleBorderWindow** style & can't be resized.



HOW'D YOU DO?

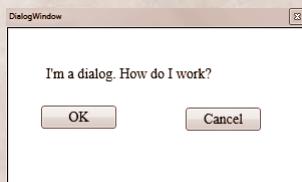
I warned you this one would be a challenge, so don't feel bad or stupid if it took you a while to work it all out.

MAIN APPLICATION WINDOW



Other properties are set here.
<Window x:Class="_15app07.MainWindow"
...
WindowStyle="ThreeDBorderWindow"
WindowStartupLocation="CenterScreen"
ResizeMode="CanResizeWithGrip">

DIALOG WINDOW



<Window x:Class="_15app07.DialogWindow"
...
WindowStyle="ToolWindow"
ResizeMode="NoResize"
ShowInTaskbar="False"
Topmost="True">

CHILD WINDOW



<Window x:Class="_15app07.ChildWindow"
...
ResizeMode="NoResize"
WindowStyle="SingleBorderWindow">

THE MAIN WINDOW

The main window displays the date when it opens. That code goes in the constructor.

```
Public Sub MainWindow()
    InitializeComponent()
    Me.TodaysDate.Content = DateTime.Today
End Sub
```

HANDLING THE DIALOG

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 will return a result.)

You could also have declared this using the syntactic sugar `bool?` which is what Intellisense shows you as the return value from `ShowDialog()`.

The buttons on the dialog need to set the `DialogResult` using code like this.

```
Private Sub CancelButton_Click(object sender,
    RoutedEventArgs e)
    Me.DialogResult = False
    Me.Close()
End Sub
```

```
Private Sub OpenDialog_Click(object sender,
    RoutedEventArgs e)
    Dim dw As DialogWindow
    dw = New DialogWindow()

    Dim result As Nullable(Of Boolean)
    result = dw.ShowDialog()
    MessageBox.Show(result.ToString())
End Sub
```

```
Private Sub OkButton_Click(object sender,
    RoutedEventArgs e)
    Me.DialogResult = True
    Me.Close()
End Sub
```

The code for the child window is on the next page...



HOW'D YOU DO, CONTINUED...

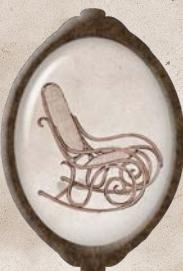
Here's the last bit: the child window.

To open a child window, you simply set the `Owner` property before you open it.

Finally, the button on the child window simply displays the `Owner.Title` property:

```
Private Sub OpenChild_Click(object sender,
    RoutedEventArgs e)
    Dim cw As ChildWindow
    cw = New ChildWindow()
    cw.Owner = Me
    cw.Show()
End Sub
```

```
Private Sub ShowOwner_Click(object sender,
    RoutedEventArgs e)
    MessageBox.Show(Me.Owner.Title)
End Sub
```



TAKE A BREAK

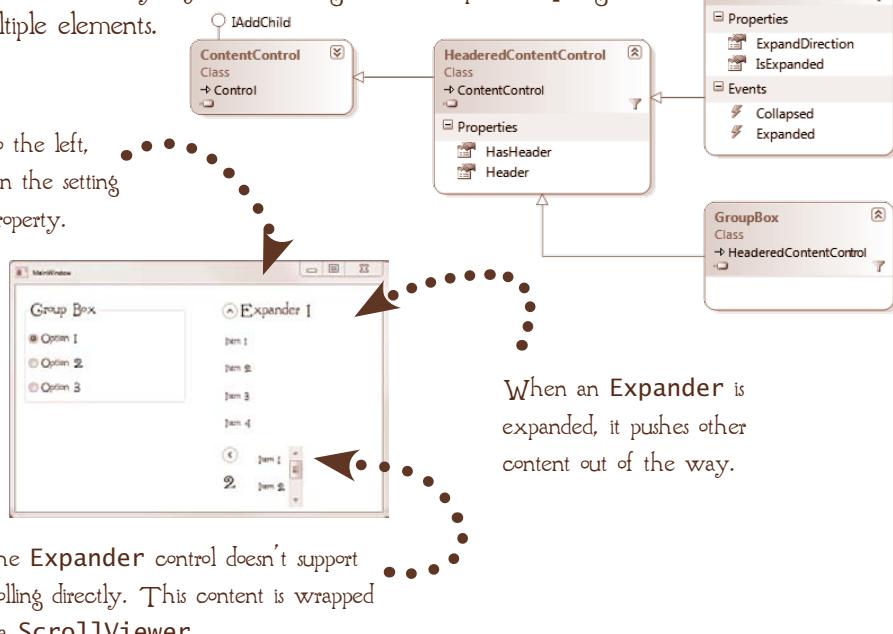
Whew, that was a big exercise, wasn't it? Probably took you a while to put together. I've been doing this since before .NET was officially released, and it took me a while, so don't feel like you're struggling. But after all that hard work, why don't you take a break before we move on?

HEADEDERED CONTENT CONTROLS

Another important branch of the content control class hierarchy is the controls that descend from `HeaderedContentControl`. Headered content controls are controls that have both a single item of content, and a single header. One of the cool things about headered content controls is that, like the content itself, the header can be any object, including, for example, an `Image` or a panel that itself contains multiple elements.

`Expander` controls can open to the left, right, top or bottom, depending on the setting of the `ExpandDirection` property.

A `GroupBox` only contains its content in a box, but it's commonly used to control the behavior of a set of `RadioButton` controls. (We'll discuss those in the next section.)



PUT ON YOUR THINKING HAT

Can you re-create the example window? Here are some hints:

- The layout root is a two-column `Grid`.
- The two `Expander` controls are in a `StackPanel` in the right column of the `Grid`.
- The first `Expander` contains a set of `Label` controls inside a `StackPanel`.
- The second `Expander` (its `Header` is `②`) contains a `ScrollViewer`. The `ScrollViewer` contains a `StackPanel`.



HOW'D YOU DO?

It doesn't matter if your layout is a bit different from mine, as long as the basic structure is the same. (And even that can be different, as long as the window looks the same. It's the result that counts.)

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="232" />
    <ColumnDefinition Width="146*" />
  </Grid.ColumnDefinitions>
  <GroupBox Header="Group Box" Height="135" Margin="10,10"
    Name="groupBox1" Width="200">
    <StackPanel>
      <RadioButton FontSize="18" IsChecked="True">Option 1</RadioButton>
      <RadioButton FontSize="18">Option 2</RadioButton>
      <RadioButton FontSize="18">Option 3</RadioButton>
    </StackPanel>
  </GroupBox>
  <StackPanel Grid.Column="1" >
    <Expander Header="Expander 1" HorizontalAlignment="Left" Margin="20,10,0,0"
      Name="expander1" IsExpanded="False">
      <StackPanel>
        <Label FontSize="14">Item 1</Label>
        <Label FontSize="14">Item 2</Label>
        <Label FontSize="14">Item 3</Label>
        <Label FontSize="14">Item 4</Label>
      </StackPanel>
    </Expander>
    <Expander Header="2" Margin="10,10" Name="expander2" ExpandDirection="Right"
      IsExpanded="False" Padding="10,0" >
      <ScrollViewer Height="75" >
        <StackPanel>
          <Label FontSize="14">Item 1</Label>
          <Label FontSize="14">Item 2</Label>
          <Label FontSize="14">Item 3</Label>
          <Label FontSize="14">Item 4</Label>
        </StackPanel>
      </ScrollViewer>
    </Expander>
  </StackPanel>
</Grid>
```

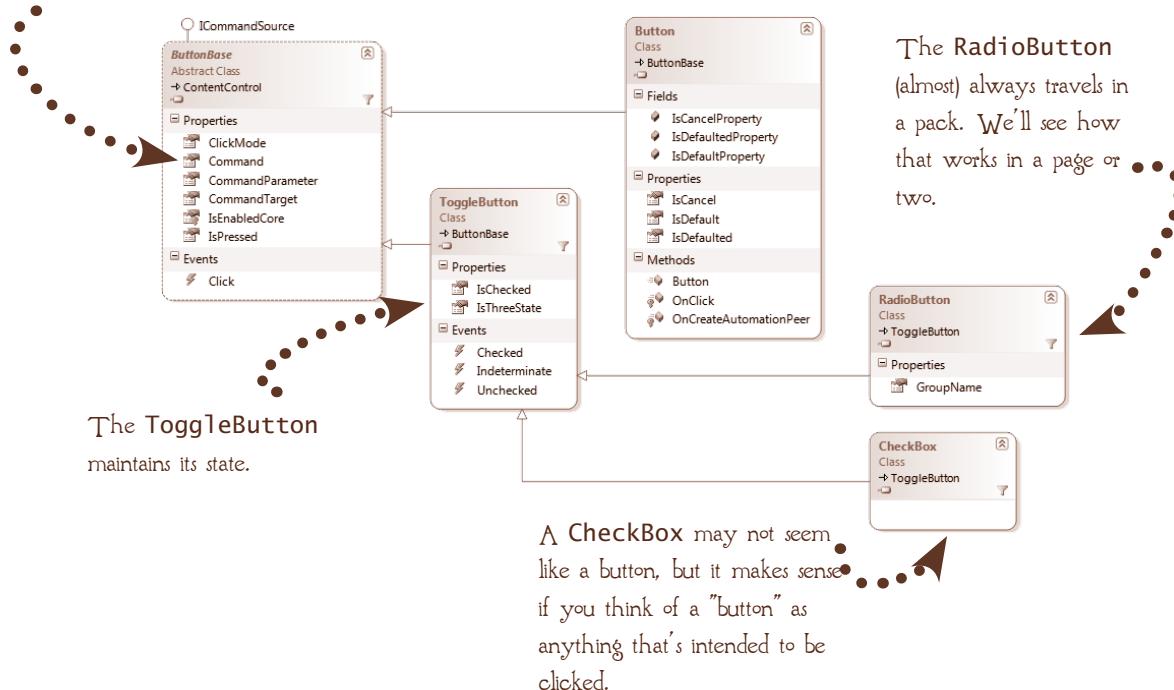
You haven't seen the
RadioButton control. Did
you get the syntax right?
Well done if you did!



BUTTONS

The final important branch of content controls is the buttons. Of course, you've been using the basic `Button` all through the book, and what you've been doing—adding code to the `Click` event—is the most common use of any kind of button. But it's time to put buttons in context. Here's the class hierarchy:

We'll talk about `Command` objects
in Chapter 18.



BUTTON CONTENTS

We've seen that the distinguishing characteristic of a `ContentControl` is that it defines a property called `Content`. The property is defined as an `Object`, and I've said that that means it can be anything. Let's use the familiar `Button` control to explore the implications of that.



Here's a basic button. Its content is defined (in attribute syntax) as `Content="I'm a basic button"`. You've created a lot of these.

This is still a basic button. Its content is defined using property syntax:

```
<Button>
  <Image Source="Binoculars.png" />
</Button>
```

It's kind of cool, but what does it do? We should add a label...



That's better. Now, I could have added the label to the image, but that's really doing it the hard way. And what if you want to reuse the image with different labels? Instead, the direct content of the `Button` is a `StackPanel` that contains the `Image` and the `Label`:

```
<Button>
  <StackPanel>
    <Image Source="Binoculars.png" />
    <Label>More Details</Label>
  </StackPanel>
</Button>
```

RADIO BUTTON GROUPS

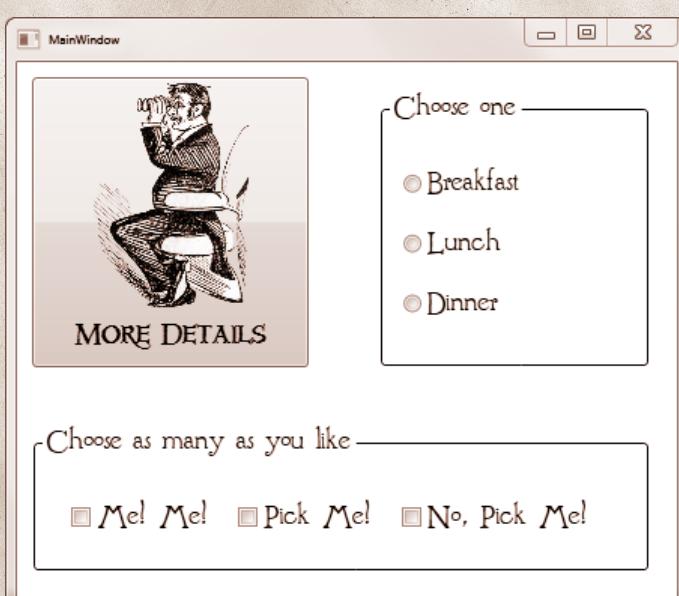
If you look back to the partial class diagram of the `RadioButton` on page 565, you'll see that the class adds only one property to the `ToggleButton` from which it descends: `GroupName`, which is defined as a `String`. Unlike other types of buttons that operate independently, `RadioButtons` interact within a group—only one button in the group can be selected at any given time.

There are two ways to create a `RadioButton` group. You can either nest them in the same container (usually, but not always a `GroupBox`), or you can explicitly assign the `GroupName` property to the same value.



PUT ON YOUR THINKING HAT

Re-create this window, using any image you like. When you do, and you run it, test the behavior of the `RadioButton` and `CheckBox` objects. Both of them are in `StackPanels` inside a `GroupBox`, but they behave differently. How?





HOW'D YOU DO?

You may have used different layout objects than I did, and your sizing may be different. That's fine.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="231" />
    <ColumnDefinition Width="200" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="222" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Button Height="193" HorizontalAlignment="Left" Margin="10,10,0,0"
Width="184">
    <StackPanel>
      <Image Source="Binoculars.png" Height="150"/>
      <Label HorizontalAlignment="Center" >More Details</Label>
    </StackPanel>
  </Button>
  <GroupBox Grid.Column="1" Margin="10,10,10,19" Header="Choose one"
    Padding="10,10" BorderBrush="Black">
    <StackPanel>
      <RadioButton>Breakfast</RadioButton>
      <RadioButton>Lunch</RadioButton>
      <RadioButton>Dinner</RadioButton>
    </StackPanel>
  </GroupBox>
  <GroupBox Grid.Row="1" Grid.ColumnSpan="2" Padding="10,10"
Margin="10,10" Header="Choose as many as you like">
    <StackPanel Orientation="Horizontal" >
      <CheckBox Margin="10,0">Me! Me!</CheckBox>
      <CheckBox Margin="10,0">Pick Me!</CheckBox>
      <CheckBox Margin="10,0">No, Pick Me!</CheckBox>
    </StackPanel>
  </GroupBox>
</Grid>
```



TAKE A BREAK

Why don't you take a quick break before you complete the Review and we finish up the chapter with an examination of items controls?



REVIEW

How does the `Window` class differ from other content controls?

There are two ways to create a set of `RadioButtons`. What are they?

What property distinguishes content controls?

What are the two methods you can use to open a `Window`?

What property distinguishes a headered content control?

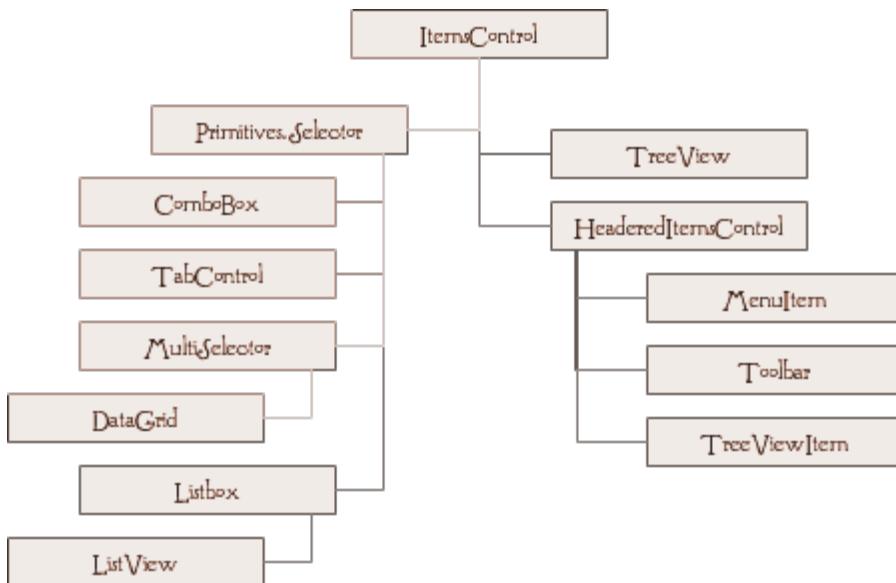
What happens when you nest checkboxes in a group box?

How do you nest multiple objects in a content control?



ITEMS CONTROLS

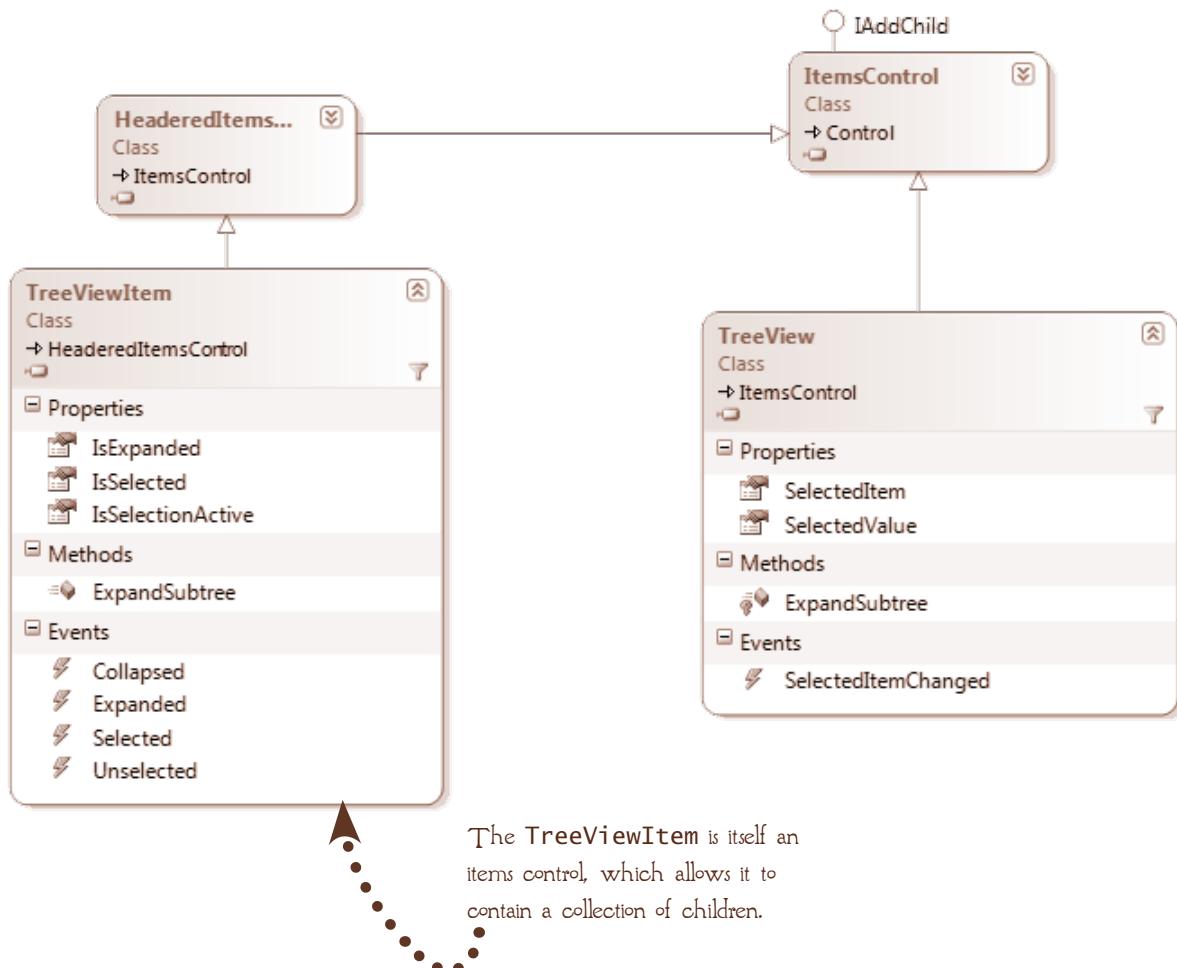
The other important branch of the control hierarchy is the classes that descend from `ItemsControl`. Like a content control, an items control takes a single element of content, but that item is a collection, the `Items` property, which is a collection of `UIElement` objects. (`UIElement` is the base class for `FrameworkElement`.)





THE TREEVIEW

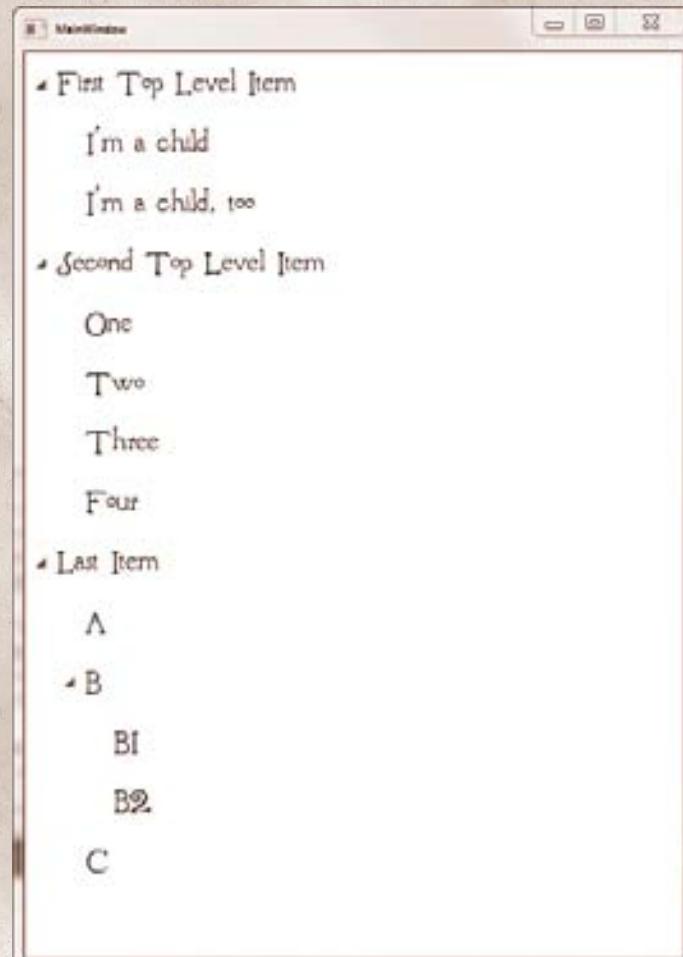
The TreeView, which displays hierarchical data, is the only common control that descends directly from ItemsControl. (Most descend by way of the Selector class.) Like many of the items controls, the TreeView redefines the Items collection to be a collection of a specific type, in this case TreeViewItem, which is itself an items control, by way of HeaderedItemsControl.





PUT ON YOUR THINKING HAT

Using XAML collection syntax, can you work out how to specify the collections that create the TreeView shown? (If you've forgotten how collection syntax works, check page 513). Hint: You specify the text shown by the TreeView using the **Header** property.



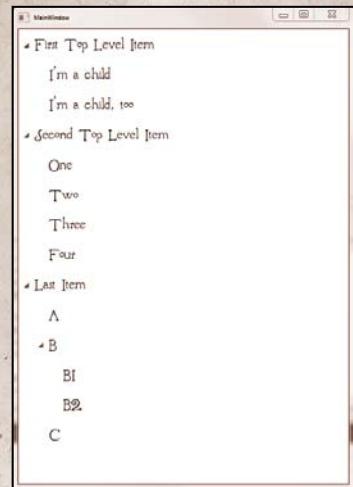


HOW'D YOU DO?

Was it a poser, or did you find it easy to figure out? Don't give up if it was a bit difficult. I promise it will get easier with practice.

Did I catch you? Collection syntax is how you specify the content of a panel, too.

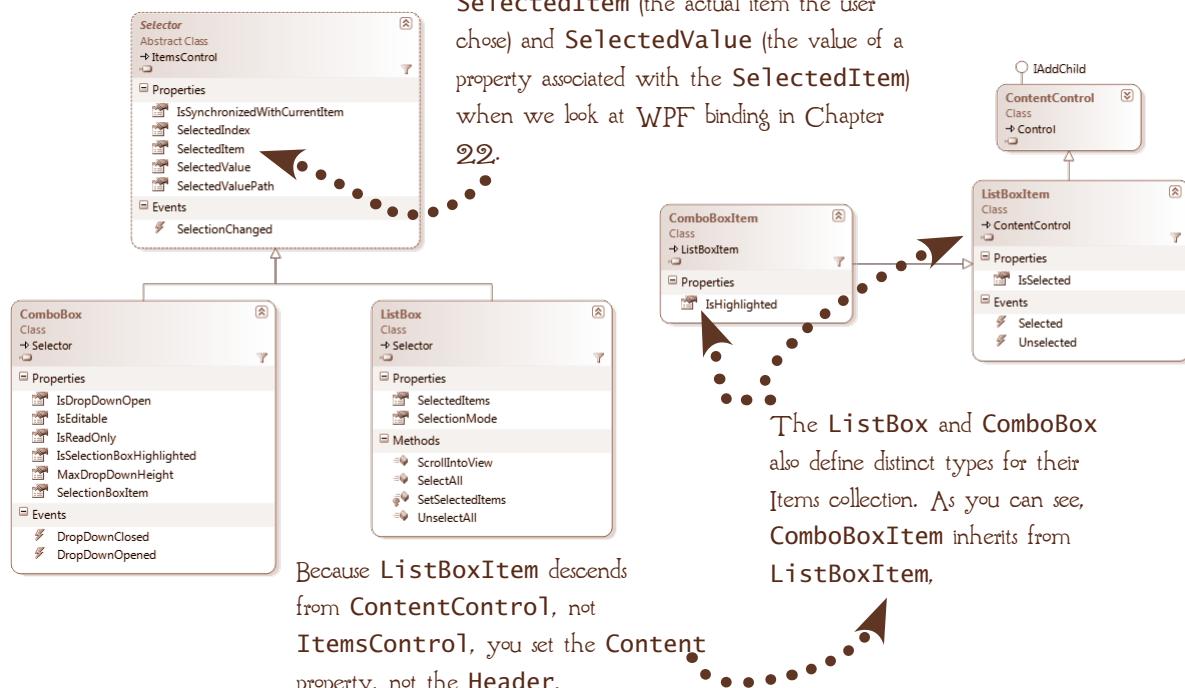
```
<TreeView FontFamily="Brandywine"  
FontSize="24">  
    <TreeViewItem Header="First Top Level Item">  
        <TreeViewItem Header="I'm a child" />  
        <TreeViewItem Header="I'm a child, too" />  
    </TreeViewItem>  
    <TreeViewItem Header="Second Top Level Item">  
        <TreeViewItem Header="One" />  
        <TreeViewItem Header="Two" />  
        <TreeViewItem Header="Three" />  
        <TreeViewItem Header="Four" />  
    </TreeViewItem>  
    <TreeViewItem Header="Last Item">  
        <TreeViewItem Header="A" />  
        <TreeViewItem Header="B">  
            <TreeViewItem Header="B1" />  
            <TreeViewItem Header="B2" />  
        </TreeViewItem>  
        <TreeViewItem Header="C" />  
    </TreeViewItem>  
</TreeView>
```



The `TreeViewItem` collections can be nested as deeply as you need them to be.

THE LISTBOX AND COMBOBOX

The **ListBox** and **ComboBox** controls are two of the most commonly used items controls. They both descend from the **Selector** class, which adds properties that allow the user to select items in the control.



PUT ON YOUR THINKING HAT



We've been working in **XAML** a lot in this chapter, so let's do something in **VB**. Can you write an application that displays a **ListBox** with three or four items, selects the second item in the list when the window is loaded, and then displays the **SelectedIndex** in a **MessageBox** when the user chooses something else? (Hint: You'll need to add an event handler. Do you remember how to do that from the Properties window?)

When you run the application, it will display the **MessageBox** twice. Do you know why?



HOW'D YOU DO?

The application displays the MessageBox when you set the SelectedIndex in code. We'll find out how to fix that in the next chapter.

THE XAML

There's nothing too tricky here, as long as you didn't follow the TreeView sample too closely—ListBoxItem is a content control, so you can either set the Content property explicitly using attribute syntax, or implicitly using content syntax, as I've done here.

```
<Grid>
    <ListBox Name="SampleListBox" Margin="10,10">
        <ListBoxItem>First Item</ListBoxItem>
        <ListBoxItem>Second Item</ListBoxItem>
        <ListBoxItem>Third Item</ListBoxItem>
        <ListBoxItem>Fourth Item</ListBoxItem>
    </ListBox>
</Grid>
```

SELECT THE SECOND ITEM

The best place to do this is in the window constructor, which should look like this:

```
Public Sub New()
    InitializeComponent()
    Me.SampleListBox.SelectedIndex = 1;
End Sub
```

Did you remember to
count from zero?



RESPOND TO A SELECTIONCHANGED EVENT

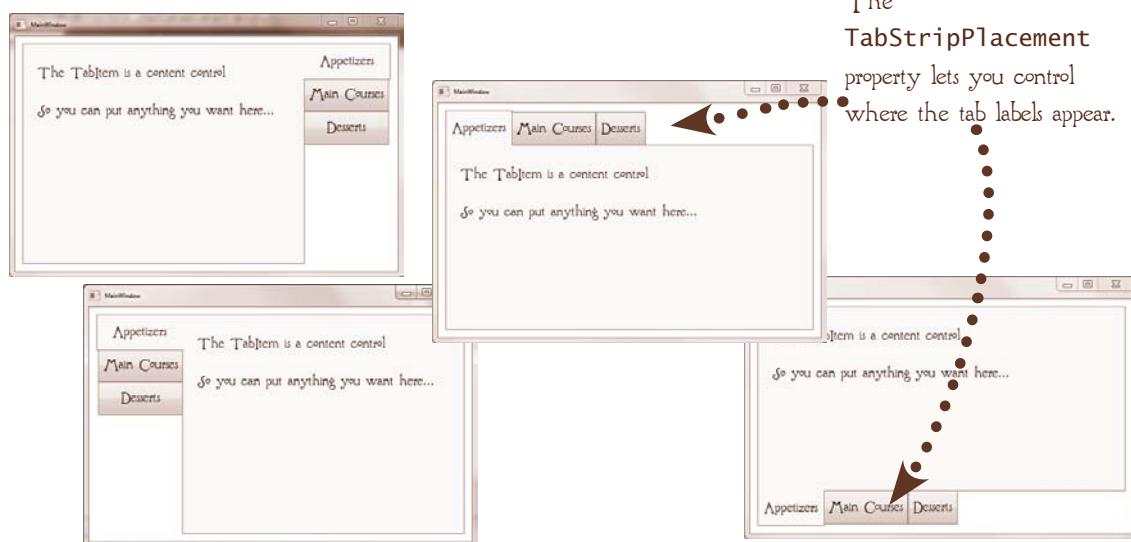
This one is easy if you do it from the Properties window. Otherwise, the syntax for the method is tedious:

```
Private Sub SampleListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
    Handles SampleListBox.SelectionChanged
    MessageBox.Show(this.SampleListBox.SelectedIndex.ToString())
End Sub
```

TAB CONTROLS

Before we finish up the chapter with an exercise that combines all we've learned, let's take a quick look at the `TabControl`, which is another useful mechanism for managing your screen real estate. Modern applications have largely replaced the older MDI structure that uses child windows with tabbed windows. The only new property you need to deal with is `TabStripPlacement`. You already know everything else you need to use this control. Are you impressed? I am!

The `TabStripPlacement` property in `TabItem` is read-only.



The `TabStripPlacement` property lets you control where the tab labels appear.

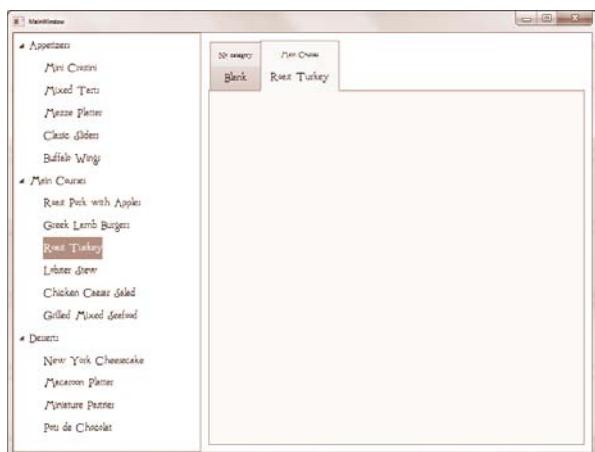
PUTTING IT ALL TOGETHER

We'll finish up this chapter with an extended exercise that uses what we've learned to rough out a real application component. Here's what it's going to look like:

When it first opens, the application will display a list of recipes in a TreeView and a TabControl with a single blank tab.



When the user double-clicks a recipe in the TreeView, we'll open up a new tab. (We won't worry about the contents of the tab right now, just the header.)





PUT ON YOUR THINKING HAT

Before we jump into coding, take some time to think about what we need to do. Can you answer these questions?

The main window has a Treeview on the left and a TabControl on the right. When the window is resized, the width of the TreeView remains the same. What panel type do you think will be best to use as the layout root?

The TabItem has two items in its Header. How will you do that? (Hint: the Header property is defined as an Object.)

When an item in the list of recipes is double-clicked, a new tab will open. You'll need to put that code in an event handler. What event do you think you should respond to? (Hint: Check the available events in the Properties window when the TreeView is selected.)

What property can you use to get a reference to the item the user double-clicked?

The tab header lists both the recipe name and the type of recipe (e.g., “Appetizers” or “Desserts”). What property can you use to get a reference to the type of recipe?

Our recipe hierarchy has two levels. You'll only want to add a new tab if the user double-clicks a second-level item (e.g., “Mini Crostini”, not “Appetizers”). How do you think you can check that? (Hint: Remember the GetType() method?)

To add the new TabItem, you'll need to create it and then add it to the TabControl's item collection. How do you think you'll do that?

It will be convenient for the tab you've just added to be selected in the TabControl once you add it. That doesn't happen automatically. What property of the TabItem will you set to select it?



HOW'D YOU DO?

Here's the way I'm going to approach the problem. But remember, there are very few absolutes in programming. If you answered these questions differently, can you justify your answers?

What panel type do you think will be best to use as the layout root?

I'm going to use a DockPanel, but other panels would work, too.

The TabItem has two items in its Header. How will you do that?

There are other ways, but the easiest is probably to put the two Labels in panel. I'm going to use a StackPanel.

What event do you think you should respond to?

We'll use the MouseDoubleClick event.

What property can you use to get a reference to the item the user double-clicked?

We can use SelectedItem.

What property can you use to get a reference to the type of recipe?

The Parent property of the selected property will give us the reference, but we'll have to cast both objects to TreeViewItem.

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?

You can test for Parent.GetType().Name == "TreeView"

To add the new TabItem, you'll need to create it, and then add it to the TabControl's item collection. How do you think you'll do that?

We can call the Add() method of the Items collection.

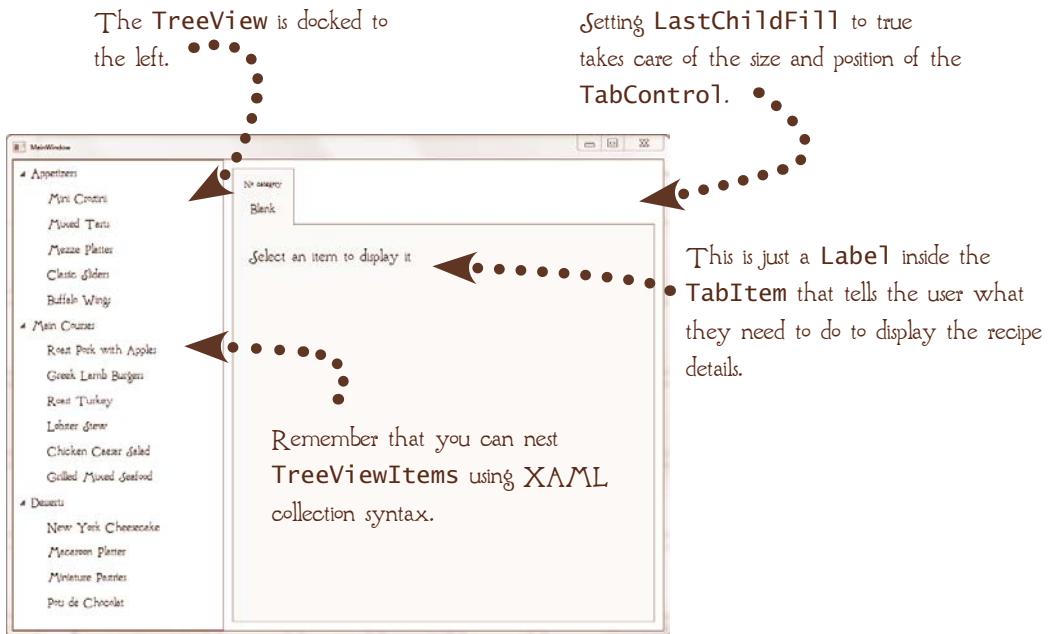
What property of the TabItem will you set to select it?

We just need to set IsSelected to true.



BUILD THE UI

We'll start by building the basic window that contains a DockPanel, a TreeView containing several items at two levels, and a TabControl with one TabItem. If you'd like to use something other than recipes in your sample, feel free (you could go back to that list of applications you want to write), but for our purposes, you'll want a two-level hierarchy with details to be displayed in tabs.



PUT ON YOUR THINKING HAT

Create a new Visual Studio WPF project and build the basic window as shown.





HOW'D YOU DO?

Here's the basic structure of my version of the XAML. Your fonts and sizes will be different, and if you used a different hierarchy, that's just fine.

```
<DockPanel LastChildFill="True">
    <TreeView Name="RecipeList" DockPanel.Dock="Left" Width="250"
        FontSize="18" MouseDoubleClick="TreeView_MouseDoubleClick">
        <TreeViewItem Header="Appetizers">
            <TreeViewItem Header="Mini Crostini" />
            <TreeViewItem Header="Mixed Tarts" />
            <TreeViewItem Header="Mezze Platter" />
            <TreeViewItem Header="Classic Sliders" />
            <TreeViewItem Header="Buffalo Wings" />
        </TreeViewItem>
        <TreeViewItem Header="Main Courses">
            ...
        </TreeViewItem>
        <TreeViewItem Header="Desserts">
            ...
        </TreeViewItem>
    </TreeView>
    <TabControl Margin="10,10" Name="RecipeTab">
        <TabItem>
            <TabItem.Header> Remember the name; it's
                / <StackPanel> important!
                <Label FontSize="12" HorizontalAlignment="Center"
                    Height="25">No category</Label>
                <Label FontSize="18" HorizontalAlignment="Center"
                    Height="35">Blank</Label>
            </StackPanel>
            </TabItem.Header>
            <Label>Select an item to display it</Label>
        </TabItem>
    </TabControl>
</DockPanel>
```

It's important to name the elements that we'll refer to in code, but it's fine if you chose different names, as long as they make sense to you.

The name of this method will be different, depending on when you created the handler. I created mine before I set the element name.

You'll have the second-level items here. I didn't have room, so I've left them out of the sample.



PUT ON YOUR THINKING HAT

We only have a single method to write: the `MouseDoubleClick` event handler. But before we jump into coding, let's spend some time thinking about what needs to be done, and in what order. In the list below, three of the action items are unnecessary. Can you spot the ones that you won't need and put the rest in order?

Get a reference to the item that was double-clicked.

Add two labels to the header `StackPanel`.

Add the new tab to the `TabControl`.

Create a new `TabItem`.

Select the new tab.

Create a `StackPanel` for the header.

Create a `Label` containing the recipe name.

Create a `TabControl`.

Set the `Header` property of the `TabItem` to a `StackPanel`.

Check that the selected item is in the second level of the hierarchy.

Get a reference to the parent of the item that was double-clicked.

Select an item in the `TreeView`.

Set the `FontFamily` and `FontSize` of the `Window`.

Create a `Label` containing the recipe category name.



HOW'D YOU DO?

Here's the order I'd do the items, but notice that not all of them need to be done in a particular order. For example, while you have to add the two labels to the StackPanel in a particular order to get them to display correctly, you can create the labels in any order.

- 1 Get a reference to the item that was double-clicked.
- 8 Add two labels to the header StackPanel.
- 10 Add the new tab to the TabControl.
- 4 Create a new TabItem.
- 11 Select the new tab.
- 5 Create a StackPanel for the header.
- 7 Create a Label containing the recipe name.
- X Create a TabControl.
- 9 Set the Header property of the TabItem to a StackPanel.
- 2 Check that the selected item is in the second level of the hierarchy.
- 3 Get a reference to the parent of the item that was double-clicked.
- X Select an item in the TreeView.
- X Set the FontFamily and FontSize of the Window.
- 6 Create a Label containing the recipe category name.



PUT ON YOUR THINKING HAT

Okay, it's time to write the code. The list of action items is below, with some notes to get you started. You might also want to check back to that first exercise to remind yourself of the methods and properties you'll need to use.

- ① Get a reference to the item that was double-clicked.
Remember, you'll have to cast the `SelectedItem` property to a `TreeViewItem`.
- ② Check that the selected item is in the second level of the hierarchy.
If it isn't, you can just return. (But it would be better to tell the user what's going on.)
- ③ Get a reference to the parent of the item that was double-clicked.
- ④ Create a new `TabItem`.
- ⑤ Create a `StackPanel` for the header.
- ⑥ Create a `Label` containing the recipe category name.
Remember to set the `FontSize` and `Height` properties. Check the XAML for the values.
- ⑦ Create a `Label` containing the recipe name.
- ⑧ Add two labels to the header `StackPanel`.
Remember to add them in the right order, recipe name first.
- ⑨ Set the `Header` property of the `TabItem` to a `StackPanel`.
- ⑩ Add the new tab to the `TabControl`.
- ⑪ Select the new tab!



HOW'D YOU DO?

Here's my version. As long as yours works, don't worry if your code's a little different...

```
Private Sub TreeView_MouseDoubleClick(sender As Object, e As MouseButtonEventArgs)
    'get a reference to the item the user double-clicked
    tv As TreeView = CTypeRecipeList.SelectedItem, TreeView)

    'if the parent of this item isn't a TreeViewItem, it's a first-level object,
    'so we don't want to do anything.
    If tv.Parent.GetType().Name = "TreeView" Then
        Return
    End If

    'get a reference to the parent
    parentTv As TreeViewItem = CType(tv.Parent, TreeViewItem)

    'create the new tab
    NewTab As TabItem = New TabItem()

    'create the header
    RecipeHeader As StackPanel = New StackPanel()

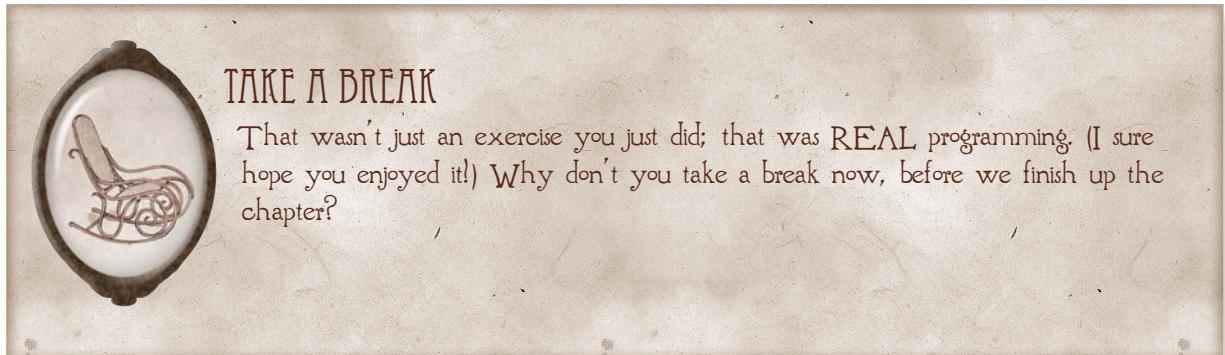
    'Create the category label
    Category As Label = New Label()
    Category.Content = parentTv.Header
    Category.FontSize = 12
    Category.Height = 25
    Category.HorizontalContentAlignment = System.Windows.HorizontalAlignment.Center
```

```
'Create the recipe name Label
    RecipeName As Label = New Label()
    RecipeName.Content = tv.Header
    RecipeName.FontSize = 18
    RecipeName.Height = 35
    RecipeName.HorizontalContentAlignment = System.Windows.HorizontalAlignment.Center

'Add the labels to the header StackPanel
    RecipeHeader.Children.Add(Category)
    RecipeHeader.Children.Add(RecipeName)

'Add the header to the new tab
    NewTab.Header = RecipeHeader

'Add the tab to the Tabcontrol and select it
    RecipeTab.Items.Add(NewTab)
    NewTab.IsSelected = True
End Sub
```



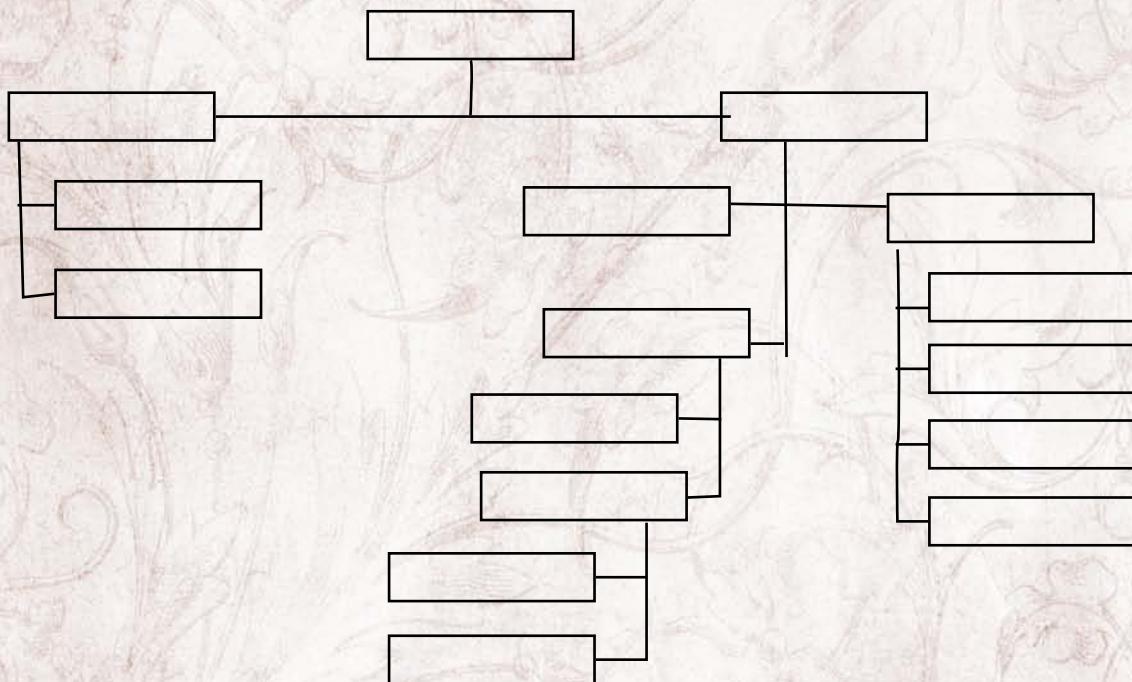
TAKE A BREAK

That wasn't just an exercise you just did; that was REAL programming. (I sure hope you enjoyed it!) Why don't you take a break now, before we finish up the chapter?



REVIEW

You've worked really hard in this chapter, but this should be pretty easy. Can you put each of the classes in the list into their place in the class hierarchy?



Framework Element

Control

Calendar

ContentControl

Window

Button

TabItem

Label

ItemsControl

TreeView

TabControl

Panel

DockPanel

Grid

StackPanel

Selector

ListBox

ComboBox

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



DEPENDENCY PROPERTIES

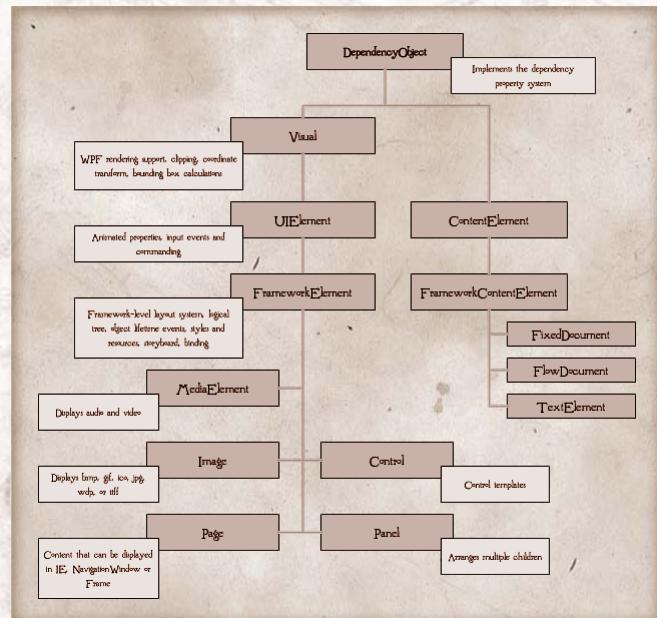
17

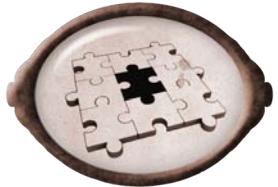
In the last chapter you started to get your teeth into some real WPF programming projects using XAML and VB. Using XAML to define an application's interface declarative certainly sets WPF programming apart from older, code-only platforms. But as cool as declarative programming is, it's perfectly possible to ignore XAML completely and write a WPF application completely in VB.

The same isn't true of the subject of this chapter: dependency properties. Take a look at the WPF class hierarchy from the last chapter. Every WPF class descends from `DependencyObject`; in a sense that's what makes a class part of WPF. And what does `DependencyObject` do? It implements the dependency property system.

Unlike standard .NET properties that only change their values when a new value is explicitly assigned to them, dependency properties can inherit their values from their parents in the logical tree, change their values over time, and notify their world when their values change. A dependency property definition can even be shared across multiple classes. (The attached properties like `Grid.Row` that you've been using are a special kind of dependency property.)

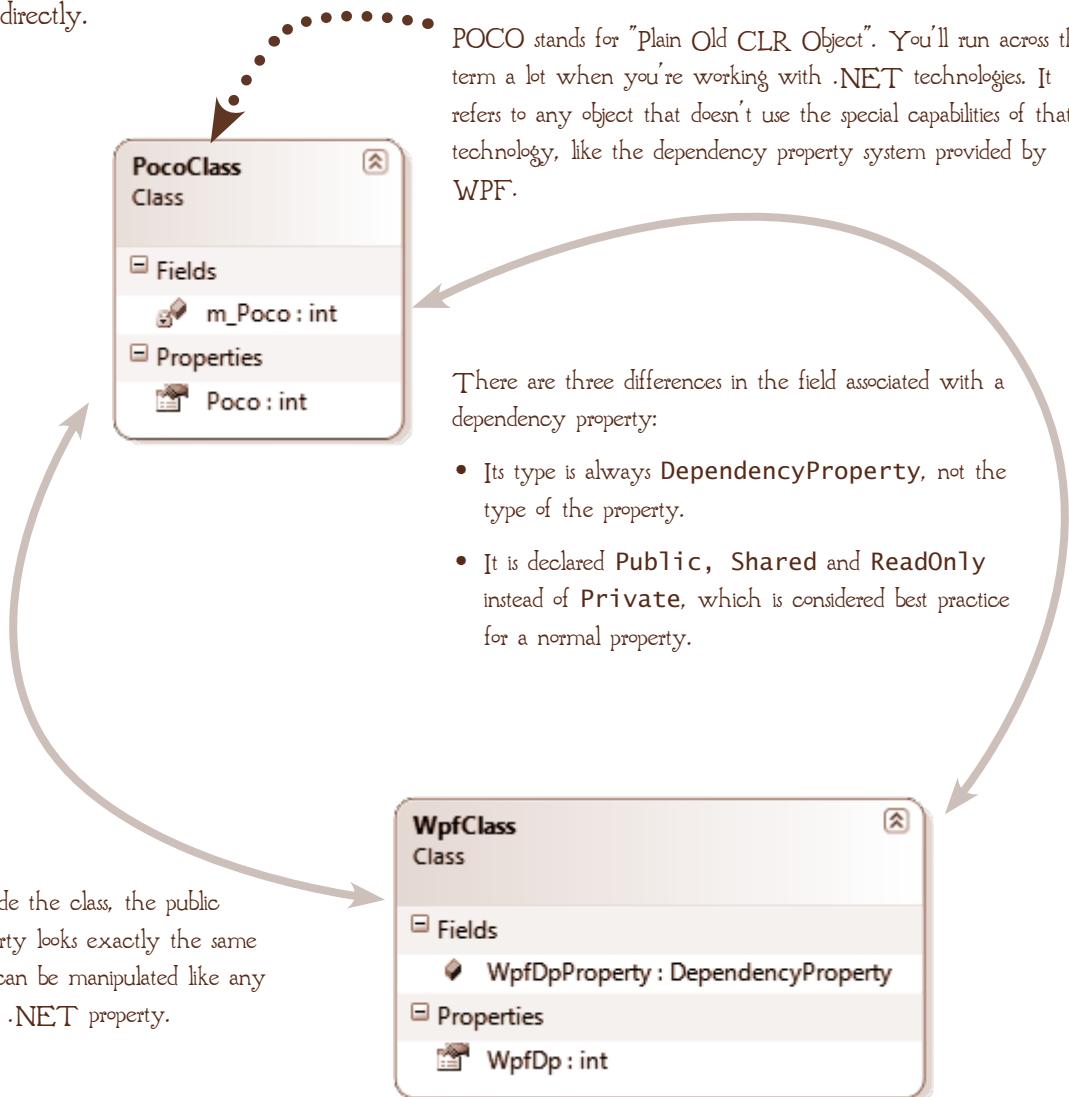
Of course, you can do all this with standard .NET properties, too, but why write code you don't have to?





FITTING IT IN

We've seen that .NET Framework types have four kinds of members: fields, properties, methods and events. Dependency properties don't change that. Just like regular .NET properties, dependency properties are implemented using fields and properties. The magic happens when you register the property with the dependency property system and use WPF `GetValue()` and `SetValue()` to change the value of the backing field instead of setting the value directly.



```

Public class PocoClass
    Private m_Poco As Integer;

    Public Property Poco() As Integer
        Get
            Return m_Poco
        End Get
        Set
            m_Poco = value
        End Set
    End Property
End Class

```

The `SetValue()` and `GetValue()` methods are used to keep the value of the standard property synchronized with the dependency property.

```

Public Class WpfClass
    Public Property WpfDp() As Integer
        Get
            Return CType(Getvalue(WpfDpProperty, Integer)
        End Get
        Set
            SetValue(WpfDpProperty, value)
        End Set
    End Property

    Public Static ReadOnly WpfDpProperty As DependencyProperty = _
        DependencyProperty.Register(...)
End Class

```



WORDS FOR THE WISE

Remember that a private field that stores the values associated with a standard .NET property is called a BACKING FIELD. In WPF, the field associated with the property is the dependency property, and it is said to WRAP the standard .NET property, not BACK it. It's a difference in perspective that's reflected in the terminology.



TASK LIST

In this chapter we'll begin our exploration of how WPF extends the way that .NET objects can communicate and interact with the dependency property system.



CALCULATING DEPENDENCY PROPERTY VALUES

As we'll see, dependency properties can do a lot of cool things, but the one you'll rely on most often is their ability to calculate their values, so we'll begin by exploring exactly how that happens.



DEPENDENCY PROPERTY CLASSES

Like any set of .NET objects, the class hierarchy defines how dependency properties function and interact, so after we've explored how they arrive at their final values, we'll take a quick look at the class hierarchy that defines them.



CREATING DEPENDENCY PROPERTIES

There are several steps to creating a dependency property in the classes you define, and, frankly, the syntax is pretty complex. Fortunately, Visual Studio includes a code snippet that does (almost) all the heavy lifting. We'll see how that works in the next section of the chapter.

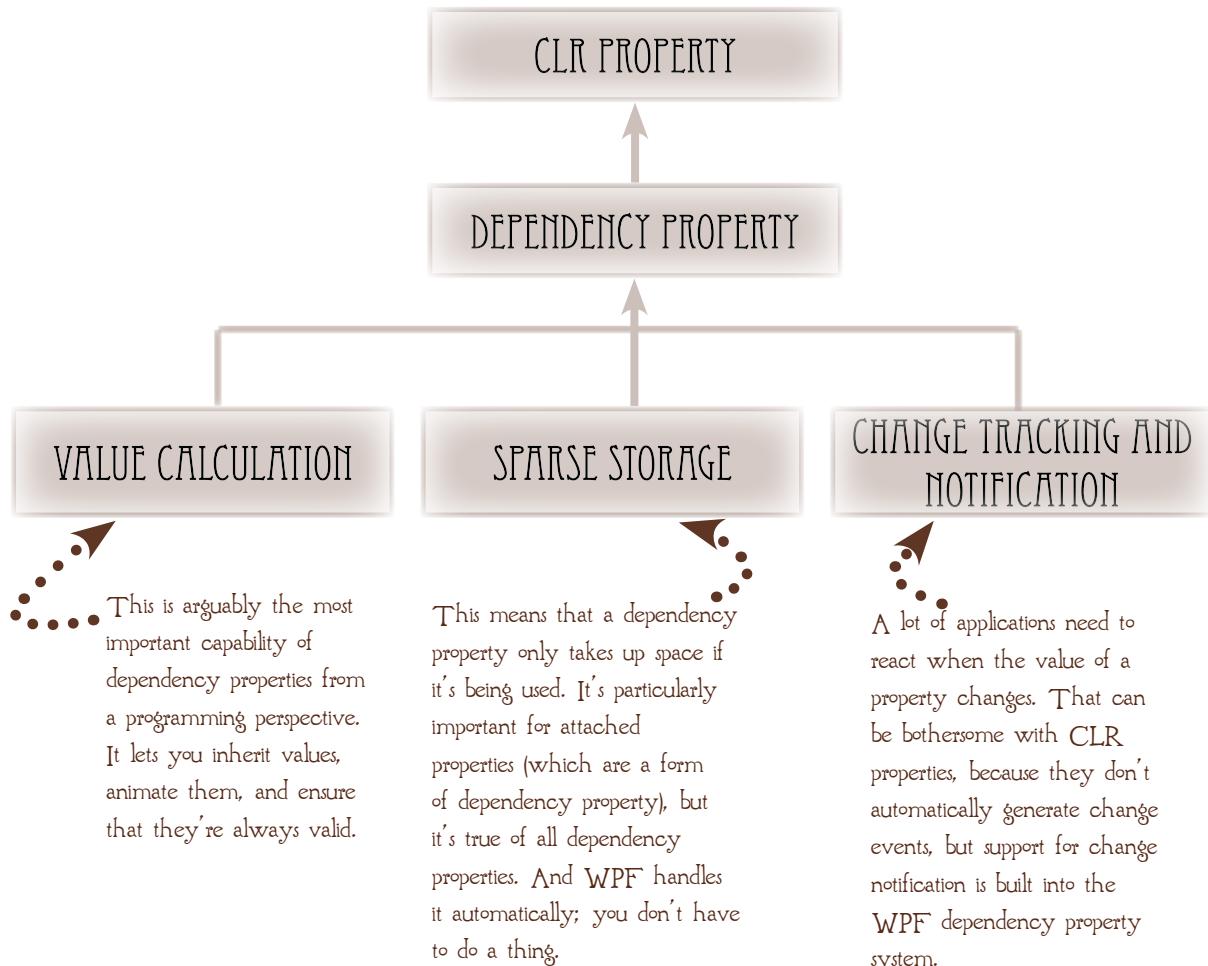


DEPENDENCY PROPERTY CALLBACKS

At the end of the chapter, we'll come full circle when we explore how you can control the calculated value of a dependency property using callback methods. A **CALLBACK** is rather like an event handler—it's a method that's called at specific points during processing. Dependency properties support three callbacks, and we'll finish the chapter by examining them.

DEPENDENCY PROPERTIES

Dependency properties "extend the functionality of CLR properties". Fine. What does that mean, exactly? Do they do things that you can't do with a CLR property? No, not really. It's just that dependency properties encapsulate a bunch of useful functionality that can be tedious to implement using CLR properties, and because they're part of the WPF infrastructure, other components of WPF know how to work with them. So, for example, you can bind a dependency property value to another object (when a value is bound, it means that it derives its value from another value). We'll see how to do that in Chapter 22 or animate its value using a WPF Storyboard (we'll take a quick look at animation in Chapter 20).





CALCULATING DP VALUES

Dependency properties go through an entire sequence of processes to calculate their final value. In most cases, this is one of those WPF things that usually "just works", but here's the process in outline, so you'll understand what happens when you run across something weird:



- ① The process begins by determining a base value for the property. The base property itself is calculated from a series of sources. (Keep reading.)
- ② The value of a property can be based on an expression, like the value of a field in a data source. (We'll talk about that in Chapter 22.) An expression takes precedence over the base value.
- ③ When a value is ANIMATED, its value will change over time. (We'll talk about animations in Chapter 20.) The animation is applied to the result of Step 2, so it takes precedence over the expression and the base value.
- ④ If a validation callback was specified when the property was registered (we'll see how to do that at the end of this chapter), it will be called with the result of Step 3.
- ⑤ Finally, the coerce value callback, if it was specified, is applied to the result of Step 4 in order to make sure that the final value falls within an acceptable range. We'll explore coerce value callbacks at the end of this chapter, too.



WORDS FOR THE WISE

The diagram on this page is called a PROCESSING PIPELINE. You'll see that term from time to time in MSDN. It just means "the order in which stuff happens."

THE BASE VALUE

The first step a dependency property performs in calculating its final value is to determine its base value. Like a standard .NET property, the process starts with a default value, but a dependency property has a whole bunch more steps (most of which we won't talk about for a chapter or two. Sorry.)

At the lowest level is the default value that's declared when the dependency property is created.

Determine Base Value

- ① Default Value
- ② Inherited Value
- ③ Theme Setters
- ④ Theme Triggers
- ⑤ Style Setters
- ⑥ Template Triggers
- ⑦ Style Triggers
- ⑧ Local Value

WPF THEMES are just like Windows themes—they are a collection of visual characteristics that are applied application-wide. We'll talk about them in Chapter 20.

A WPF STYLE defines the appearance of a particular type of control. We'll discuss styles in Chapter 20.

A TEMPLATE defines how a control is composed. We'll discuss templates in Chapter 21.

Values that are inherited from a parent, like the `FontFamily` that's set for the `Window`, take precedence over default values.

A SETTER defines the value of a property as part of a style or template. We'll explore setters in Chapter 20.

A TRIGGER changes the value of a property based on some condition, like displaying negative values in red. We'll explore triggers in Chapter 20.

When you set a property value explicitly, it sets that LOCAL VALUE and trumps any other for the base value.

AN EXAMPLE

This all seems really complicated, doesn't it? Relax. In practice, using dependency properties is pretty simple. Let's look at an example:

Don't worry about how this style is defined.

We'll look at how styles work in Chapter 20.



```
<Window ...  
    FontStretch="Expanded">  
    <StackPanel>  
        <Label FontStretch="Medium">  
            <Label.Style>  
                <Style TargetType="Label">  
                    <Setter Property="FontStretch" Value="Condensed" />  
                </Style>  
            </Label.Style>  
            Hello, Dependency!  
        </Label>  
    </StackPanel>  
</Window>
```



And the final value of the `Label.FontStretch` property? It's `Medium`, the local value, which always trumps any other source.



PUT ON YOUR THINKING HAT

Let's try a little dependency property experiment.

By default, when you create a new WPF window, the `FontFamily` attribute will be set to the default system font. That's `Tahoma` on XP and `Segoe UI` (which, by the way, is pronounced "see-go") on Vista and Windows 7. The `FontSize` defaults to `8pt`, and the `Background` to `White`.

These aren't bad choices. But they're not the only ones, and they may not be the most appropriate for your application. So, suppose you want your text to display in `Times New Roman` at `14` point in dark brown on a pale yellow background. Because dependency properties inherit their values, you should be able to set the font properties for the `Window`, right? Here's a simple example:

```
<Window ...  
    FontFamily="Times New Roman" FontSize="14"  
    Foreground="SaddleBrown" Background="CornSilk" >  
    <StackPanel>  
        <Label>Hello, properties!</Label>  
        <TextBlock>How are you doing today?</TextBlock>  
    </StackPanel>  
</Window>
```

Before you create the application, take a minute to think about what you expect to have happen:

What text and brush values do you think will be displayed in the Properties window for the `Label` and `TextBlock` controls? How do you expect them to display? Will they use the values set for the `Window`, or the default properties? Will both controls behave the same?

Okay, now create the `Window` and see what really happens.



HOW'D YOU DO?

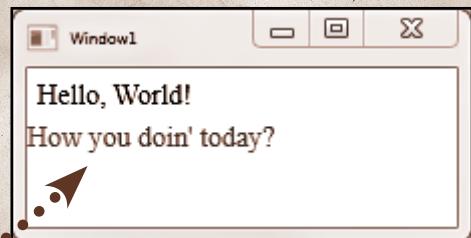
Okay, so dependency property value calculations don't *always* work the way we expect them to. Usually, but not always. So what happened and why?

Foreground, Background, `FontFamily` and `FontSize` are all dependency properties, but while the `TextBlock` inherits all the values from the `Window`, the `Label` only inherits the `FontFamily` and `FontSize` properties.

The reason for the inconsistent behavior is that all of the standard WPF controls have implicit default styles, and the default style for a `Label` sets the `Foreground` property. If you look back to the diagram of how values are calculated, you'll see that styles have a priority of 5 or 7, while inherited values only have a 2, so the style settings win.

Even once we've explored WPF styles in Chapter 20, you'll occasionally be surprised by this kind of thing. (And yes, once more, that's the voice of experience.) So what's a poor programmer to do? Take control. Either define your own control styles (probably the best solution in the long run) or set the value explicitly.

We'll look at WPF styles in detail in Chapter 20. Basically, they're just property settings that are grouped together and can be applied to more than one object.



TAKE A BREAK

The final value of a dependency property can be determined by a lot of things, and as we've just seen, every once in a while that final value won't be what you were expecting. But those occasions really are pretty rare, and now that you've seen the pipeline, you know how to address them. So for now, why don't you take a break before completing the Review?



REVIEW

What does POCO stand for?

What does it mean to say that a dependency property “wraps” a POCO property?

What value always trumps the base value calculation?

What are the values of the `FontFamily` properties for the two labels in the following snippet?

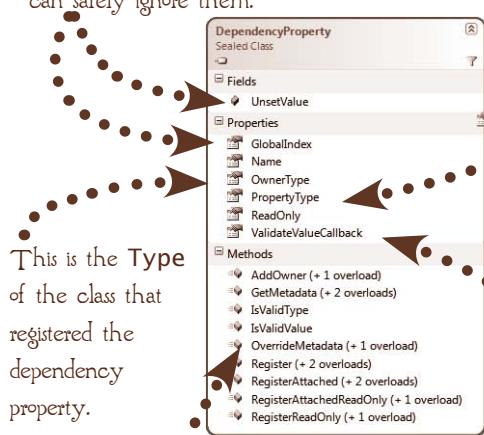
```
<Window ...  
    FontFamily="Times New Roman">  
    <StackPanel>  
        <Label Name="Label1">Hello</Label>  
        <Label Name="Label2" FontFamily="Tahoma">World</Label>  
    </StackPanel>  
</Window>
```

DEPENDENCY PROPERTY CLASSES

The keys to the WPF dependency property system are the `DependencyProperty` and `PropertyMetadata` classes that encapsulate all the functionality required to extend the behavior of a POCO property. Most of the members of these classes are self-explanatory. They're all read-only values that are established when the dependency property is registered with the system by calling one of the `Register...()` methods. We'll look at those at a minute, but first let's look at the classes themselves.

`UnsetValue` and `GlobalIndex`

are used internally by the dependency property system. You can safely ignore them.

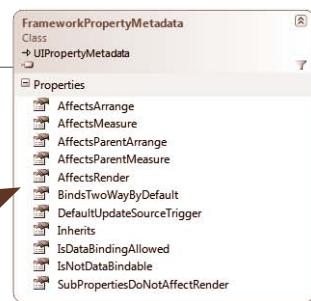
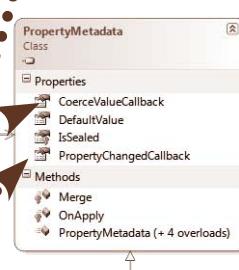


This is the Type of the class that registered the dependency property.

A CALLBACK is a method that gets called whenever something occurs. If you specify values for these three properties, they'll be called by the system whenever the value of the property is changed. We'll talk about callbacks later in this chapter.

You can override the `DefaultMetadata` values using this method, but only in a static constructor for a class that inherits the property. (You'll hardly ever do this.)

This is the Type of the CLR property wrapped by the dependency property.



Most of these properties are used by panels to display their contents correctly. The exceptions are the binding properties, which we'll discuss in Chapter 22, and the `Inherits` property, which determines whether the property can inherit values from its parent.

MORE CLASSES

Did you notice something odd about the `DependencyProperty` and `PropertyMetadata` classes? They descend directly from `Object`, not one of the WPF classes like `FrameworkElement`. I think of them as sitting outside the system in order to manipulate it. (I'm sure the architects of WPF would have a more elegant description.)

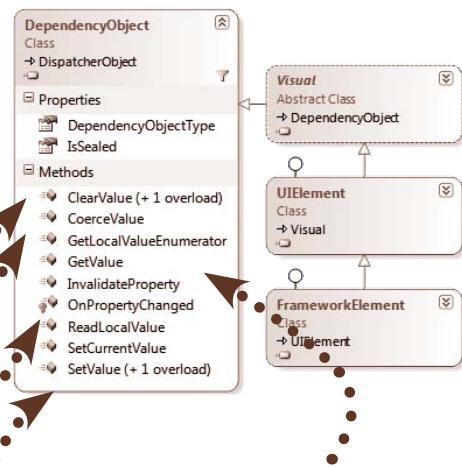
But the WPF classes you create will descend, more or less directly, from `FrameworkElement`, and one of the ancestors of `FrameworkElement`, the `DependencyObject`, provides some important functions for creating and working with dependency properties. In fact, `DependencyObject`, which as you can see sits quite high in the WPF class hierarchy, specifically encapsulates the functionality of objects that participate in the dependency property system.

A `DependencyObjectType` links a specific CLR type to the dependency property system. It's really just glue that you don't need to worry about.

`ClearValue()` clears any local values that have been explicitly set, but not values applied further up the calculation chain.

You'll only have to call `CoerceValue()` explicitly in very rare situations.

Don't try to use `OnPropertyChanged` to respond to a property change event. Instead, use the `PropertyChangedCallback`.



`SetValue()` and `GetValue()` are used inside the CLR property definition to make sure the backing property stays synchronized with the dependency property system.

`InvalidateProperty()` forces the value of the property to be recalculated.



CREATING DEPENDENCY PROPERTIES

You don't have to define the properties of your WPF classes as dependency properties. Plain Old POCO properties work just fine, but they don't take advantage of the things that make WPF such a joy to work with. Would you go to a five-star restaurant and only have a glass of water?

It's really not that difficult to make the magic happen. There are only four steps, and two of them are optional:



CREATE THE CLR PROPERTY

This is just a basic CLR property, but you must use `GetValue()` and `SetValue()` within the `Get` and `Set` accessors to make sure the CLR property works within the dependency property system.



CREATE THE CALLBACKS

This one's optional, but if you need to coerce or validate the property values or want something to happen when the property changes, you'd create callback methods for those purposes. We'll see how callbacks work at the end of this chapter.



CREATE THE METADATA

In practice, you'll almost always do this right inside the call to the `Register()` function (in Step 4), but you can create the metadata object separately if you really need to.



REGISTER THE PROPERTY

The final step is to call one of the static `Register...()` methods of the `DependencyProperty` class. There are four versions of the method (each of which has several overloads) depending on what kind of dependency property you want to create: a standard dependency property, a read-only dependency property, or a standard or read-only attached property.

When you register the dependency property, the name you specify must be the name of the CLR property plus the word "Property". There's nothing that enforces this convention, but if you don't follow it, WPF can break in surprising and apparently unrelated places.



WALK-THROUGH

The four steps to creating a dependency property are pretty standard, so it will probably come as no surprise that Visual Studio Intellisense can stub out the code for you. Let's walk-through using Visual Studio to define a basic dependency property.



CREATE A WPF APPLICATION IN VISUAL STUDIO

You can name it anything you like.



ADD A DEPENDENCY PROPERTY SNIPPET TO MAINWINDOW.XAML.VB

Inside the class declaration, either right-click and select Insert Snippet...or press Ctrl-K, X to open the Intellisense menu. Select WPF and press Enter and then select Add a Dependency Property and press Enter. Visual Studio will insert the stub for you:

This is the CLR
property, Step 1.

```
17pp01 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
MainWindow.xaml MainWindow.xaml.vb X
Properties Solution Explorer Object Browser
Public Property Prop1 As String
Get
    Return GetValue(Prop1Property)
End Get
Set(ByVal value As String)
    SetValue(Prop1Property, value)
End Set
End Property

Public Shared ReadOnly Prop1Property As DependencyProperty = _
DependencyProperty.Register("Prop1", _
GetType(String), GetType(MainWindow), _
New FrameworkPropertyMetadata(Nothing))
End Class
```

Notice that the accessors call `GetValue()` and `SetValue()` to keep the backing property in sync. That's all you should do inside the property accessors.

Here's Step 4, the call to `DependencyProperty.Register()` that creates the dependency property itself.

The dependency property has the same name as the CLR property with the word "Property" appended to it.

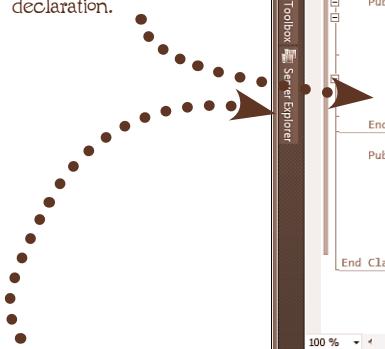
The metadata, Step 3, is created inside the `Register...` method call. This is common practice.

2

CHANGE THE PROPERTY TYPE TO INTEGER

The Intellisense stub defaults to a `string`, so we'll need to change that. Change "String" to "Integer" and move the cursor off the line. As soon as you do, Visual Studio will update the get accessor and the call to `Register()`.

`SetValue()` returns an `Object` that must be cast. Visual Studio updates the get accessor for you when you change the declaration.



```
MainWindow.xaml MainWindow.xaml.vb*
```

```
Class MainWindow
    Public Property Prop1 As Integer
        Get
            Return GetValue(Prop1Property)
        End Get
        Set(ByVal value As Integer)
            SetValue(Prop1Property, value)
        End Set
    End Property
    Public Shared ReadOnly Prop1Property As DependencyProperty = _
        DependencyProperty.Register("Prop1", _
        GetType(Integer), GetType(MainWindow), _
        New FrameworkPropertyMetadata(Nothing))
End Class
```

Remember, this is all your get and set accessors should ever do.



`.PropertyType` is one of the parameters to the `Register()` method. (We'll look at it in detail in a minute.) Visual Studio updates it for you, too.



MAKE A NOTE

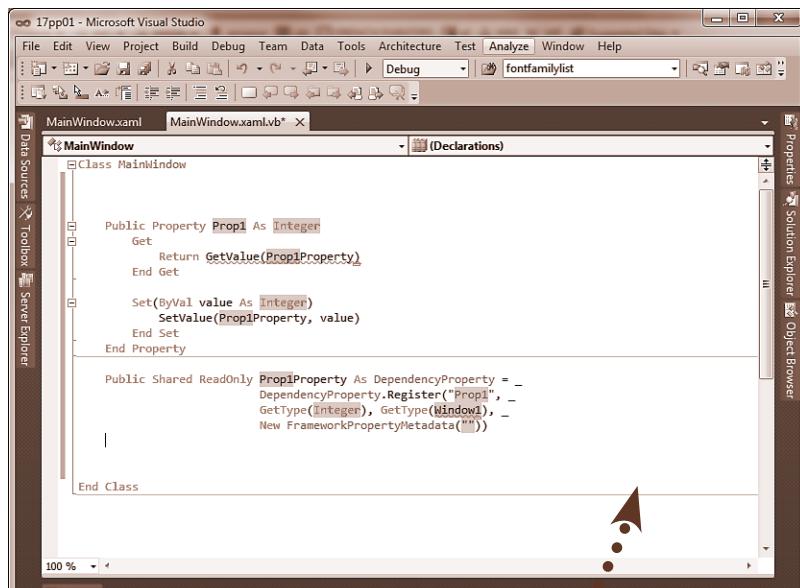
It's a bad idea to do anything other than call `GetValue()` and `SetValue()` within the CLR property definition, because it's not guaranteed to be called. Besides, a dependency property offers different mechanisms for doing most of the things you'd ordinarily do here, like checking that new values are acceptable.

3

CHANGE THE DEFAULT VALUE

The stub that Visual Studio adds for a dependency property creates the metadata inside the call to the `Register()` method, and it uses the version of the constructor that accepts a default value for the property. Because the property type is `String`, the default value is `Nothing`. If that's not what you want, you'll need to manually update the default value.

I've used an empty string here, but of course you can use any valid value for the underlying type.

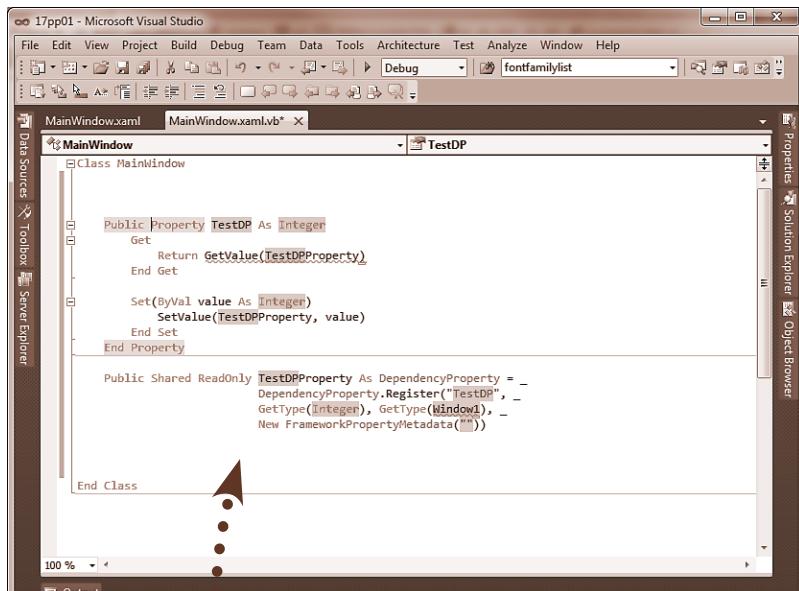


If you don't need to provide a specific default value, you can call the default (parameterless) constructor, which will set the default to `UnsetValue`. But you'll almost always want to provide a default, since the dependency property system uses `UnsetValue` for other things.

4

CHANGE THE PROPERTY NAME

You can name your property with any valid VB identifier. I've used `TestDP` in this example. Visual Studio will update the stub for you once you leave the line, just as it did when you changed the property type.



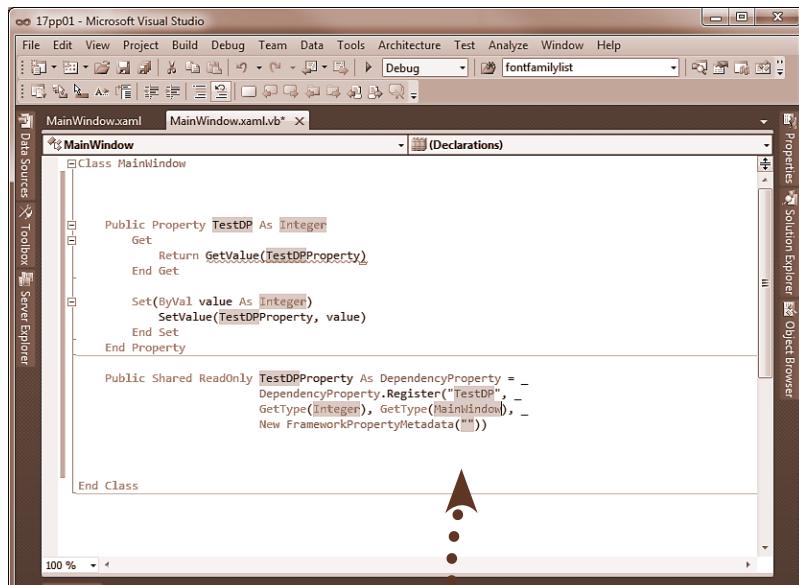
```
Module 17pp01 - Microsoft Visual Studio
File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help
MainWindow.xaml MainWindow.xaml.vb* fontfamilylist
MainWindow Class MainWindow
Public Property TestDP As Integer
    Get
        Return GetValue(TestDPProperty)
    End Get
    Set(ByVal value As Integer)
        SetValue(TestDPProperty, value)
    End Set
End Property
Public Shared Readonly TestDPProperty As DependencyProperty = _
    DependencyProperty.Register("TestDP", _
    GetType(Integer), GetType(MainWindow), _
    New FrameworkPropertyMetadata())
End Class
```

- Visual Studio updates the name of the dependency property to match the POCO property name plus "Property". You should always follow this convention.

5

SPECIFY THE OWNER CLASS

There's one last change that's required to declare a basic dependency property: You must specify the class that owns the property. This will almost always be the class in which you're declaring the property. (In fact, I haven't ever seen a situation that required anything else.)



This is the last change you have to make to define a basic dependency property.

REGISTERING A DP

Look back at the class diagram of `DependencyProperty` on page 602. Notice anything missing? There's no constructor. You never actually create an instance of the `DependencyProperty` class. Instead, you just declare it and then call one of the versions of the `Register...()` methods. WPF takes care of all the rest.

There are 10 different versions of the `Register()` method, 3 overloads of `RegisterReadOnly()`, 3 of `RegisterAttached()`, and 2 each of `RegisterReadOnly()` and `RegisterAttachedReadOnly()`. The signature of each version is shown below.

- ① `Register(<name>, <propertyType>, <ownerType>)`
- ② `Register(<name>, <propertyType>, <ownerType>, <PropertyMetadata>)`
- ③ `Register(<name>, <propertyType>, <ownerType>, <PropertyMetadata>, <ValidateValueCallback>)`
- ④ `RegisterReadOnly(<name>, <propertyType>, <ownerType>, <PropertyMetadata>)`
- ⑤ `RegisterReadOnly(<name>, <propertyType>, <ownerType>, <PropertyMetadata>, <ValidateValueCallback>)`
- ⑥ `RegisterAttached(<name>, <propertyType>, <ownerType>)`
- ⑦ `RegisterAttached(<name>, <propertyType>, <ownerType>, <PropertyMetadata>)`
- ⑧ `RegisterAttached(<name>, <propertyType>, <ownerType>, <PropertyMetadata>, <ValidateValueCallback>)`
- ⑨ `RegisterAttachedReadOnly(<name>, <propertyType>, <ownerType>, <PropertyMetadata>)`
- ⑩ `RegisterAttachedReadOnly(<name>, <propertyType>, <ownerType>, <PropertyMetadata>, <ValidateValueCallback>)`



PUT ON YOUR THINKING HAT

There are a lot of versions of Register...(), aren't there? Let's sort them out.

Which method do you think you would use in each of the following situations? You can just reference the number shown next to the signature on page 610.

To declare a read-only dependency property with a validation callback:

To declare an attached property with default metadata:

To declare a read-only dependency property without a callback:

To declare a dependency property with the default value of `false`:

To declare a read-only attached property with a validation callback:



HOW'D YOU DO?

Are the various versions of the method starting to sort themselves out in your head?

To declare a read-only dependency property with a validation callback:

```
RegisterReadOnly(<name>, <propertyType>, <ownerType>, <PropertyMetadata>,
    <ValidateValueCallback>)
```

To declare an attached property with default metadata:

```
RegisterAttached(<name>, <propertyType>, <ownerType>)
```

To declare a read-only dependency property without a callback:

```
RegisterReadOnly(<name>, <propertyType>, <ownerType>, <PropertyMetadata>)
```

To declare a dependency property with the default value of `false`:

```
Register(<name>, <propertyType>, <ownerType>, <PropertyMetadata>)
```

To declare a read-only attached property with a validation callback:

```
Register(<name>, <propertyType>, <ownerType>, <PropertyMetadata>)
```



TAKE A BREAK

There's still a little more to learn about dependency properties, but we've covered a lot, so why don't you take a quick break before we move on to callbacks and adding owners?

REUSING PROPERTIES

Okay, so you've seen how to create a dependency property from scratch, but imagine this: You're building a custom control, and you want it to have a `Background` property. Like other WPF classes that implement a `Background` property, you want it to be a dependency property that can inherit its value from its ancestors and can be animated or data-bound. Of course you could build it from scratch, but wouldn't it be easier to just "borrow" the dependency property that already exists?

You can. That's exactly what the `AddOwner()` method does. Just find a class that exposes the property, create a backing field, and add your class as an owner of the dependency property. Your `Background` property will behave exactly as though you had declared it yourself.

```
Public Shared <name> As DependencyProperty =  
<OwningClass>.<Property>.AddOwner(typeof(<newOwner>))
```

```
Public Shared BackgroundProperty As DependencyProperty =  
Control.Background.AddOwner(typeof(MyClass))
```

If you don't specify the `PropertyMetadata` when you call the `AddOwner()` method, the property system will generate the metadata by merging the metadata values for base classes that define the property. In practice, that means that the metadata will be the same as it was for the class from which you're borrowing the property, which is what you want.

If you want different metadata, or if you just want to be sure that your version of the property has exactly the metadata you want, you can use the version of the `AddOwner()` method that accepts new metadata as an argument.

```
Public Shared <name> As DependencyProperty =  
<OwningClass>.<Property>.AddOwner(typeof(<newOwner>),  
New PropertyMetadata(<...>));
```



PROPERTY CALLBACKS

There are three places where you can "hook" into the process by which the value of a dependency property is set. You do so by creating callback procedures and providing them to the property's `Register...()` method.

- ① The `ValidateValueCallback` is called here. It receives only the value of the property.

Validating happens on a class level, not an instance level. As we'll see, that means it can't be used in certain situations.

- ② The `CoerceValueCallback` is called here. It receives a copy of the object and the new value.

The `PropertyChangedCallback` usually raises an `OnPropertyChanged` event, but it might also do some other things. We'll look at WPF events in the next chapter.

- ③ The `PropertyChangedCallback` is called at the end of the process. It receives a copy of the object and an instance of the `DependencyPropertyChangedEventArgs`.

Now, it's important to understand that while any dependency property *can* implement behavior in these callbacks, they don't have to. This has implications not only for properties you create, but also when you're working with the dependency properties that are defined on FCL objects. Calling methods always has a little bit (or a lot) of overhead, so only those properties that need the callback behaviors implement them.

This doesn't usually cause any problems (the WPF architects were pretty smart about what might and might not be needed), but every once in a while you'll find that there isn't a property changed event. You can usually get around the problem using the `OverrideMetadata` method, but it happens so rarely that you really don't need to worry about it, except to know that the capability is there if you ever need it.

VALIDATION CALLBACKS

Let's look at how the callbacks actually work. Callbacks are just standard VB methods, and you can do anything you need to inside them. But the properties of the dependency property metadata and the value passed for the validation method aren't methods themselves, they're instances of delegate types, and so the callback methods have to conform to the signature specified for the delegate. (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.) Let's start with the `ValidateValueCallback` delegate, which defines a method that accepts an `Object` as a parameter and returns a `bool`:

By convention, validation callbacks are named `Validate<Property>` or `validate<Property>Callback`, where `<Property>` is, of course, the name of the CLR property the callback wraps.

`<MethodName>(value As Object) As Boolean`

The method must take a single argument, an `Object`. By convention, the parameter is named `value`, but you can call it anything you like and it will work just fine.

CREATING VALIDATION CALLBACKS

The validation callback receives a single argument that represents the value of the dependency property. Because the parameter is defined as an `Object`, you'll almost certainly have to cast it to the appropriate type:

```
Private Shared Function ValidateAmount(value As Object) As Boolean
    newValue As Integer = CInt(value)
    If <is valid> Then
        Return True
    Else
        Return False
    End If
End Function
```

The code that replaces `<is valid>` in the sample can, of course, be as complex as you need it to be.

The method should return `True` if the value is valid for the property, or `False` if it is not, but notice that you only have access to the value, not the property or the instance that contains it, so you can only do discrete validations. For example, you can test whether the `Amount` is a positive value (that's required for any `RecipeItem`), but you can't perform tests like "if the `Volume` is greater than zero, then `Weight` must be zero", because that requires access to other properties of the `RecipeItem` class.

COERCE VALUE CALLBACKS

Unlike validation callbacks, the method referenced by the `CoerceValueCallback` delegate is part of the `PropertyMetadata`, so it has access to the instance that contains the property. The delegate defines a method with the following signature:

As with any delegate, you can choose whatever you like for the method and parameter names.

The first parameter is the instance of the class that exposes the dependency property.

```
<MethodName>(d As DependencyObject, baseValue As Object) As Object
```

The second parameter is the base value of the property (the first step in the property calculation pipeline).

The method returns an **Object** that represents the adjusted property value.

CREATING COERCE VALUE CALLBACKS

A coerce value callback is most often used in two situations:

- When you want to restrict a property to a specific range (such as positive integers less than 10,000), but you want it to be possible to override that restriction (which isn't possible with a validation callback), or
- The valid range of values is dependent on the values of other properties.

The second situation is probably the most common (and it is surprisingly common). In our `RecipeItem` class, for example, some of the properties are mutually exclusive. A recipe might specify a single egg (`Count = 1`) or 57 grams of egg (`Weight = 57`), but it isn't sensible to do both. You could impose this restriction in a coerce value callback that looked something like this:

```
Private Shared Function CoerceCount(ByVal d As DependencyObject, _  
    ByVal baseValue As Object) As Object  
    RecipeItem theItem = CType(d, RecipeItem)  
    If theItem.Weight > 0 Then  
        theItem.Count = 0  
    End If  
    Return theItem.Count  
End Function
```

You need to cast the `DependencyObject` to the type you need, but you know it's a safe cast.

PROPERTY CHANGED CALLBACKS

The final callback type for dependency properties is the property changed callback, which is probably the one you'll use the most often, because it allows consumers of your class to hook into the change (as opposed to the other callbacks, which only allow operations from within the class itself). Like the coerce value callback method, the method reference by the `PropertyChangedCallback` delegate is declared as part of the `PropertyMetadata` and has access to the owning instance. The delegate defines a method with the following signature:

Just as with the coerce value callback, the first parameter is the instance that exposes the property.

```
Function <MethodName>(ByVal d As DependencyObject, _  
    ByVal e As DependencyPropertyChangedEventArgs)
```

Instead of receiving the value directly, the second parameter is an instance of `DependencyPropertyChangedEventArgs`, which exposes the old and new values, and a reference to the property itself.



CREATING PROPERTY CHANGED CALLBACKS

Unlike the other dependency property callbacks, the property changed callback doesn't participate in the value calculation pipeline. Instead, it is called after all of the calculations have been performed. Notice that the method is a function, and although both the old and new values are visible in the `DependencyPropertyChangedEventArgs` parameter, all of the properties are read-only, so you can't directly change the value.

You can, of course, directly set the property within the method, but that triggers another trip through the pipeline. That's generally a bad idea, because of course the method will be called again at the end of the new calculation and trigger yet another trip, and another trip, and...your application may get stuck in an endless loop. Although resetting the value of a property is questionable practice within a property changed callback method, there's nothing intrinsically wrong with performing other actions, such as setting the value of a different instance property.

The most common use of the callback, however, is to raise a property changed event. Because events can be handled by multiple methods both inside and outside the class, raising an event allows client classes to respond to the property change.

REGISTERING CALLBACKS

Now that we've seen what each kind of callback can do, let's explore how each kind of callback is registered, because the registration method has implications for how each of the callbacks is implemented and how it can be used.

The `PropertyChangedCallback` and `CoerceValueCallback` are passed as part of the `PropertyMetadata` constructor:

```
New PropertyMetadata(<defaultValue>,
<PropertyChangedCallback>,
<CoerceValueCallback>)
```

Because the constructor parameters are delegates, you'll need to create instances of the appropriate delegate type. You'll usually do that inside the constructor, because there's no need to have the delegate types hanging around:

```
New PropertyMetadata(<defaultValue>,
new PropertyChangedCallback(<method>),
new CoerceValueCallback(<method>))
```

If you want to pass a `CoerceValueCallback`, but not a `PropertyChangedCallback`, you can pass `null` to the `PropertyMetadata` constructor:

```
New PropertyMetadata(<defaultValue>,
null,
new CoerceValueCallback(<method>))
```

The `ValidateValueCallback` is passed directly to the `Register..()` method:

```
Register(<name>, <propertyType>, <ownerType>,
         new PropertyMetadata(...),
         <ValidateValueCallback>)
```

Like the callback parameters to the `PropertyMetadata` constructor, the `ValidateValueCallback` parameter of the `Register...()` method is a delegate, so you'll need to create a new instance, and you'll almost always do it inside the method call:

```
Register(<name>, <propertyType>, <ownerType>,
         new PropertyMetadata(...),
         new ValidateValueCallback(<method>))
```

You can't skip the `PropertyMetadata` parameter the way you can the `PropertyChangedCallback`, but you can use the default constructor:

```
Register(<name>, <propertyType>, <ownerType>,
         new PropertyMetadata(),
         new ValidateValueCallback(<method>))
```



MAKE A NOTE

I'm using the basic `Register()` method for these examples, but the other versions work exactly the same way.

A REGISTRATION EXAMPLE

A call to one of the `Register...` methods can get pretty dense, what with constructors embedded inside constructors. Here's a complete example:

- Dependency properties are declared `public, static` and `readonly`.

The property name always ends with "Property".

```
Public Shared ReadOnly TestProperty As DependencyProperty = _  
    Register("TestDp",  
        typeof(String),  
        typeof(MainWindow),  
        New PropertyMetadata("Nothing", _  
            New PropertyChangedCallback(OnTestDpChanged), _  
            New CoerceValueCallback(CoerceTestDp)),  
            New ValidateValueCallback(ValidateTestDp))
```

The first three arguments to the `Register...` method are the name of the CLR property, the type of the CLR property, and the owning class.

The `ValidateValueCallback` constructor is the final argument to the `Register...` method (not the `PropertyMetadata` constructor).

The first argument to the `PropertyMetadata` constructor is the default value.

The `PropertyChangedCallback` and `CoerceValueCallback` constructors are embedded in the `PropertyMetadata` constructor.

Whew! Gives you a new appreciation for the value of Intellisense, doesn't it?



PUT ON YOUR THINKING HAT

As you can see, registering a dependency property can get pretty complex. The trick, of course, whenever you're working with a complex programming topic, is to break it down and only worry about one (relatively simple) part at a time. Let's walk through building a complex registration using that technique. (Hint: Check MSDN for the overloads of the `PropertyMetadata` constructor to decide which one to use.)

1

Write a simple call to `Register()` that creates the dependency property to wrap an integer named `Amount` in the `RecipeItem` class. (Don't worry about writing the code for the CLR property, but remember the modifiers.)

2

Add a validation callback called `ValidateAmount`.

3

Add a property changed callback called `OnAmountChanged`.

4

Add a coerce value callback called `CoerceAmount`.



HOW'D YOU DO?

1

Write a simple call to `Register()` that creates the dependency property to wrap an integer named `Amount` in the `RecipeItem` class. (Don't worry about writing the code for the CLR property, but remember the modifiers.)

```
Public Shared AmountProperty As DependencyProperty = _  
    DependencyProperty.Register("Amount", GetType(Integer), _  
        GetType(RecipeItem), _  
        New PropertyMetadata())
```

2

Add a validation callback called `ValidateAmount`.

```
Public Shared AmountProperty As DependencyProperty = _  
    DependencyProperty.Register("Amount", GetType(Integer), _  
        GetType(RecipeItem), _  
        New PropertyMetadata(), New ValidateValueCallback(ValidateAmount))
```

3

Add a property changed callback called `OnAmountChanged`.

```
Public Shared AmountProperty As DependencyProperty = _  
    DependencyProperty.Register("Amount", GetType(Integer), _  
        GetType(RecipeItem), _  
        New PropertyChangedCallback(OnAmountChanged), _  
        New ValidateValueCallback(ValidateAmount))
```

4

Add a coerce value callback called `CoerceAmount`.

```
Public Shared AmountProperty As DependencyProperty = _  
    DependencyProperty.Register("Amount", GetType(Integer), _  
        GetType(RecipeItem), _  
        New PropertyMetadata(0, _  
            New PropertyChangedCallback(OnAmountChanged), _  
            New CoerceValueCallback(CoerceAmount), _  
            New ValidateValueCallback(ValidateAmount)))
```

You'll have to add a default value to pass both callbacks. Did you catch that? Well done if you did.



ON YOUR OWN

What type of object is stored in a variable declared as a delegate?

What would you call the dependency property that wraps a CLR property named `InventoryState`?

Which dependency property callback would you use to accomplish each of the following things?

Restrict the range of values

Validate a value based on other properties of the instance

Trigger an event that allows clients to respond to a value change



TAKE A BREAK

There's always more to know, but we've finished with dependency properties for the time being (we'll look at them again, of course). Why don't you take a break before you complete the Review and we start looking at events?

REVIEW

The methods to register objects with the WPF dependency property system all begin with:

Each of the objects below can be involved in the creation of a dependency property. Which are required, and which are optional?

ValidateValueCallback

Property Metadata

Call to Register() method

CLR property definition

CoerceValueCallback

PropertyChangedCallback

You can't perform checks that require access to other instance properties inside the ValidateValueCallback. Why not?

What category of snippets do you need to reference to get to the Intellisense "Define a dependency property" option?

What do you do if you want to associate a CoerceValueCallback with a property, but not a PropertyChangedCallback?

Under most circumstances, the PropertyChangedCallback only does one thing. What is it?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?

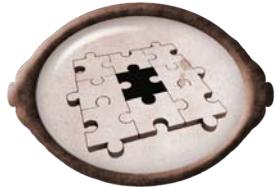


WPF INTERACTIONS

In the same way that WPF extends (adds functionality to the basic .NET property mechanism, it also extends the .NET event pattern that we explored in Chapter 14 by introducing ROUTED EVENTS.

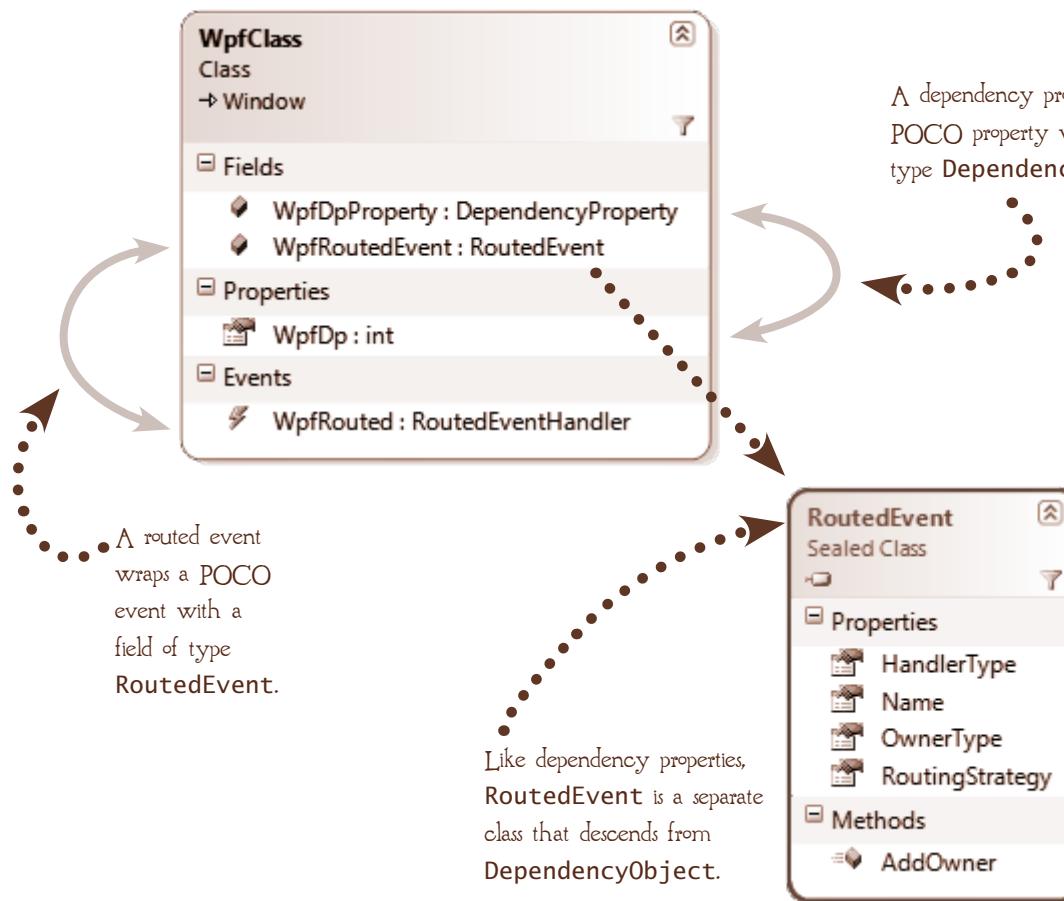
A standard .NET event can be handled by any object that registers a handler for it. WPF routed events can behave that way, too, but they can also travel up and down the logical tree, letting the event be handled by the most appropriate object in the logical tree. That's particularly important in WPF because of the way objects are nested. When you create a `Button`, for example, you want to set an event handler on the `Button` itself, not on every one of its visual children.

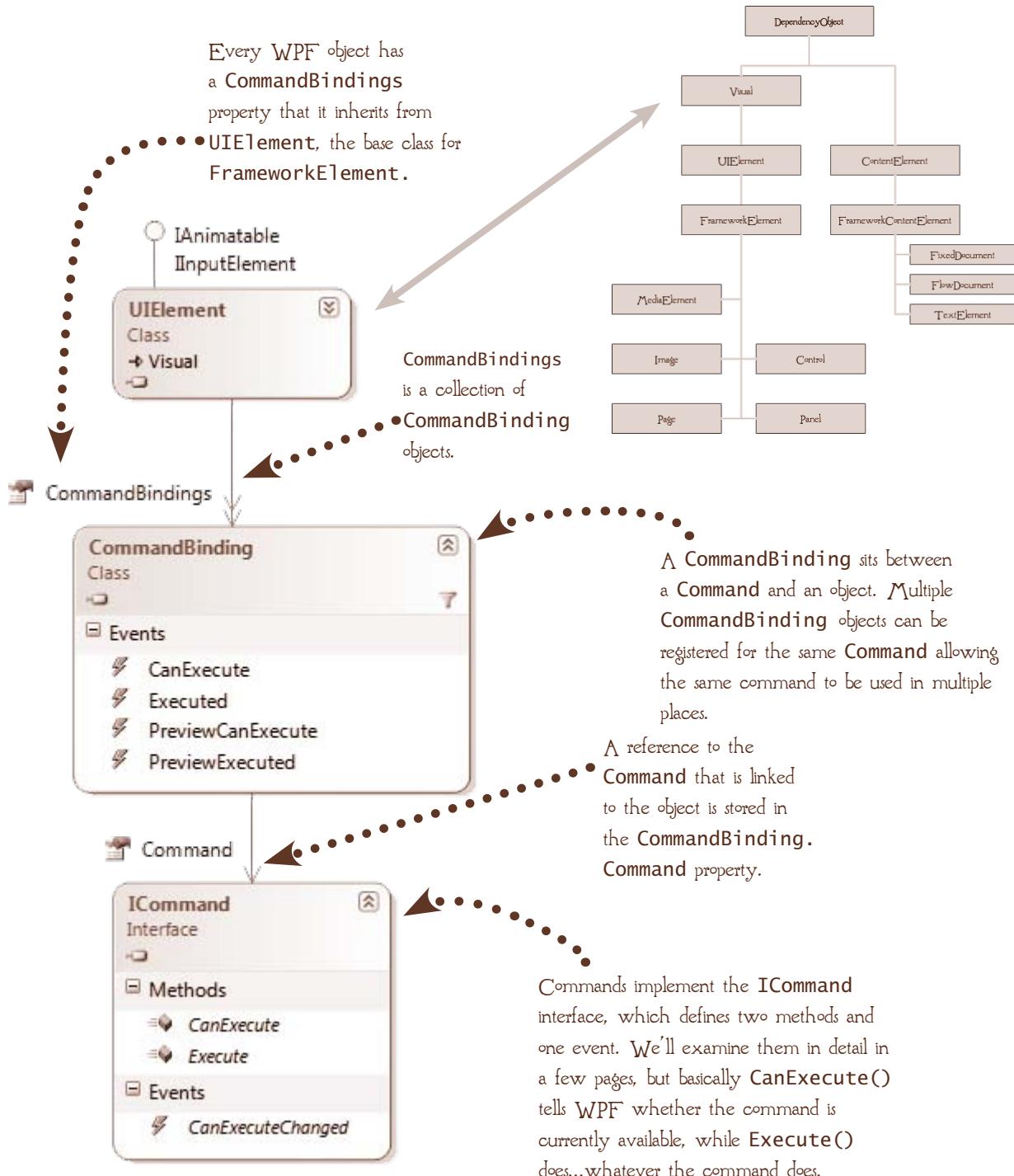
After we've explored the WPF event model, we'll examine WPF commands that let you encapsulate a set of actions in a single method and reference it from multiple places, another mechanism that's particularly useful given the way WPF models are nested. Finally, we'll look at the command libraries provided by the FCL that provide all the most common actions like Cut, Copy, Paste and Close.



FITTING IT IN

The mechanism for implementing WPF routed events is almost identical to the one you learned in the last chapter for dependency properties.







TASK LIST

In this chapter we'll explore the ways in which WPF objects can communicate and interact through the WPF property and event systems, and the special command classes that let you centralize application behavior.



ROUTED EVENTS

A POCO event allows interested parties to register their interest in an event by creating a handler for it. POCO events are called **DIRECT EVENTS** because the event is sent directly to the handlers, and that's the end of it. In WPF, events can be handled this way, but they can also bubble up from the object that raised the event to all its ancestors in the logical hierarchy, or tunnel down from the root element to the raising object. They can even move in both directions via `PreviewEvent/Event` pairs. At any level, the event can be handled or passed on to the next level.



COMMANDS

Most applications provide several different ways to perform common tasks. Think about basic text editing, for example. In most text editors, you can select a piece of text and copy it to the clipboard by selecting `Copy` from the `Edit` menu or from a context menu displayed when you right-click the selection, by clicking a button on a toolbar or ribbon, or by typing a keyboard shortcut (`Ctrl+C` by convention). In a standard .NET application, you have to write the code to actually perform the copy, and then write code for each event to call the copy. That's a lot of code. And then there are things like disabling the command if nothing's selected. A whole bunch more code. In WPF, all of this behavior is abstracted in a set of classes that center around the `ICommand` interface and the `CommandBindings` collection that links the object to the command it should execute. To make things even easier, the FCL defines a set of routed commands that include most of the standard operations performed by an application (including `Copy`, which is in the `ApplicationCommands` class).



ON YOUR OWN

Before we look at the functionality that WPF adds to the POCO event system, take a minute to review how events map to the Observer pattern that we explored in Chapter 14.



DECLARE EVENT

The first step, declaring the event and the handler delegate in the class that will raise it, isn't officially part of the Observer pattern. The VB code to declare an event named `MyEvent` that uses the default `System.EventArgs` is:

And the syntax for declaring a delegate for the handler is:

REGISTRATION

The first official step in the Observer pattern is for the observer to register interest in the event. In VB, there's some syntactic sugar to make this easy. To add a delegate method called `MyHandler` to the `MySubjectClass.MyEvent` event, you'd use the following syntax:

NOTIFICATION

The second step in the Observer pattern is to notify the observers when the event occurs. By convention, events are raised within a method named `On<Event>(<EventArgs> e)`. Given an event named `MyEvent` that uses `MyEventArgs`, the syntax for the method that raises the event would be:

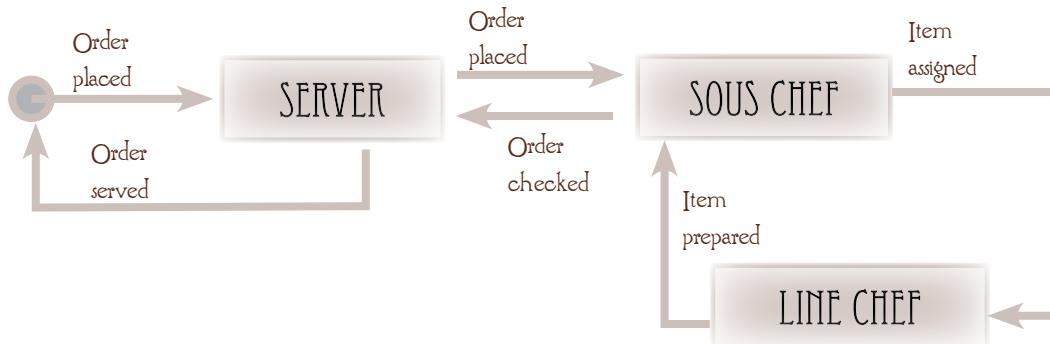
UNREGISTRATION

The final (optional) step in the Observer pattern is for the observer to let the subject know that it's no longer interested in the event. VB provides some syntactic sugar to unregister a handler. What operator does this?



ROUTED EVENTS...

Most applications are event-driven, and the POCO event system works pretty well. But compare it to this real-world situation:



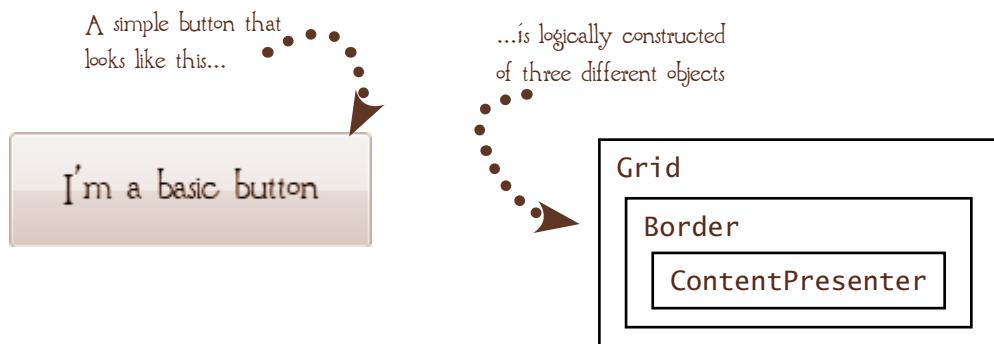
In a traditional kitchen, the sous chef (pronounced “sue chef”) checks orders before they are assigned to line chefs to be prepared and then checks the orders before they are served to customers.

If we were modelling this process, both the `OrderPlaced` and `ItemPrepared` events would be sent first to the `SousChef` class, which would have to raise additional events for the next step in the process.

Every `LineChef` would have to register a handler for the `ItemAssigned` event, and the `SousChef` class would have to register a handler for the `ItemPrepared` event of every `LineChef` (which means the `SousChef` would have to know about every instance of `LineChef`, which raises issues of its own).

Starting to look pretty complicated, isn't it?

...ADDRESS A PROBLEM



Now imagine the same situation applied to a WPF application.

The default structure of a WPF Button consists of a Grid that contains a Border, which in turn contains a ContentPresenter. When a user clicks the button, the mouse might actually be over any of these objects. So who raises the event? Do you *really* want to register the same event handler for the Button, the Grid, the Border and the ContentPresenter? No, I didn't think so.

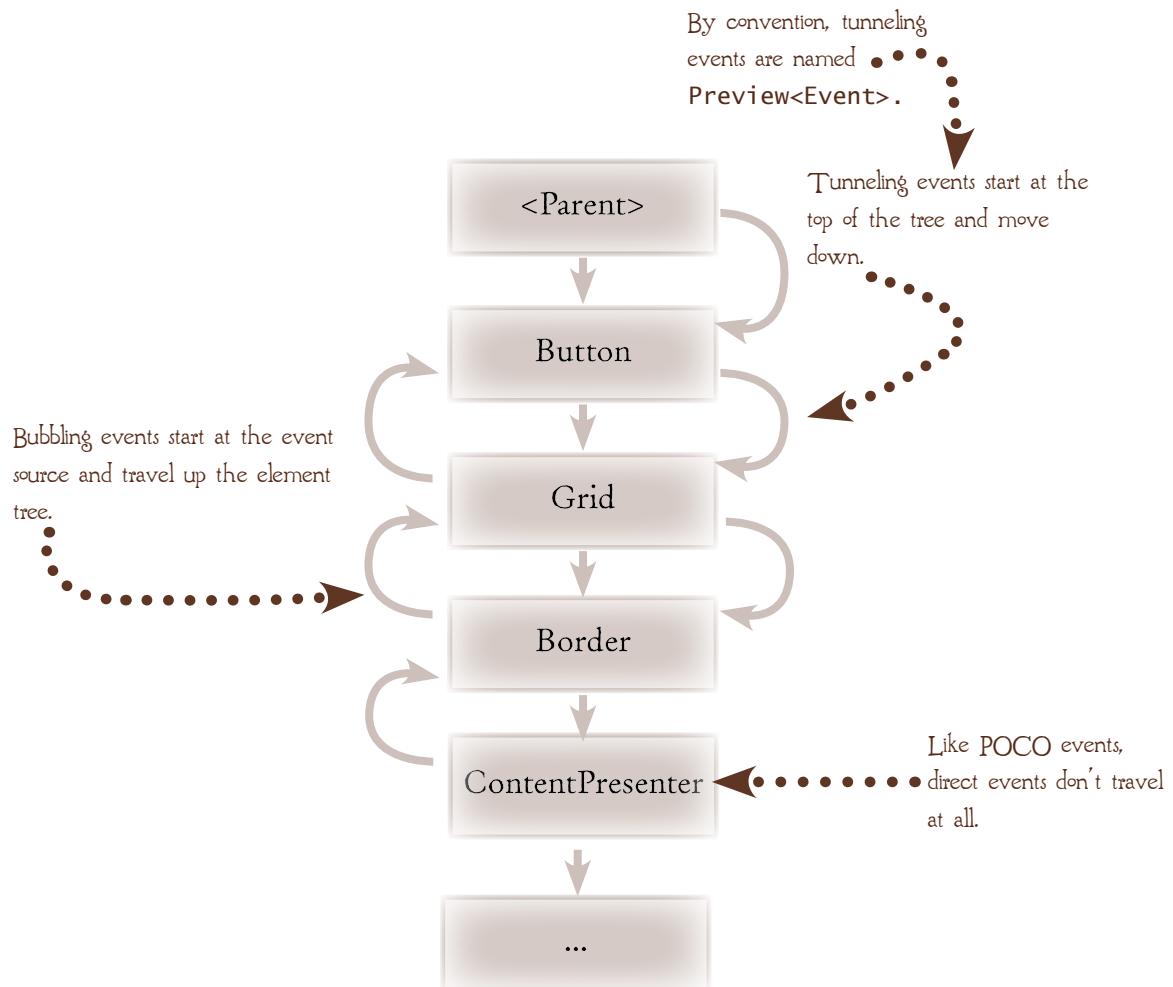
And what if the `Button.Content` is set to a `ListBox` that by default is made up of a Grid, a Border, a `ScrollViewer` and a `StackPanel` with all their individual components? Just how many times do you want to register that handler?

As we'll find out in Chapter 21, the situation is even more complex. A Button doesn't have to have this logical structure. It can even change its structure at runtime. So even if you were prepared to register all those event handlers, you can't know what objects might raise the event.

Of course, the architects of WPF thought of this problem, and they have a clean solution: routed events. A routed event can travel up and down the element tree, and it can be handled at the most appropriate level. Let the Button raise the Click event. That's sensible, efficient, and separates the logical action (the button was clicked) from the ugly details (the mouse was over a `ListItem` inside a `StackPanel` inside a `ScrollViewer` inside a...).

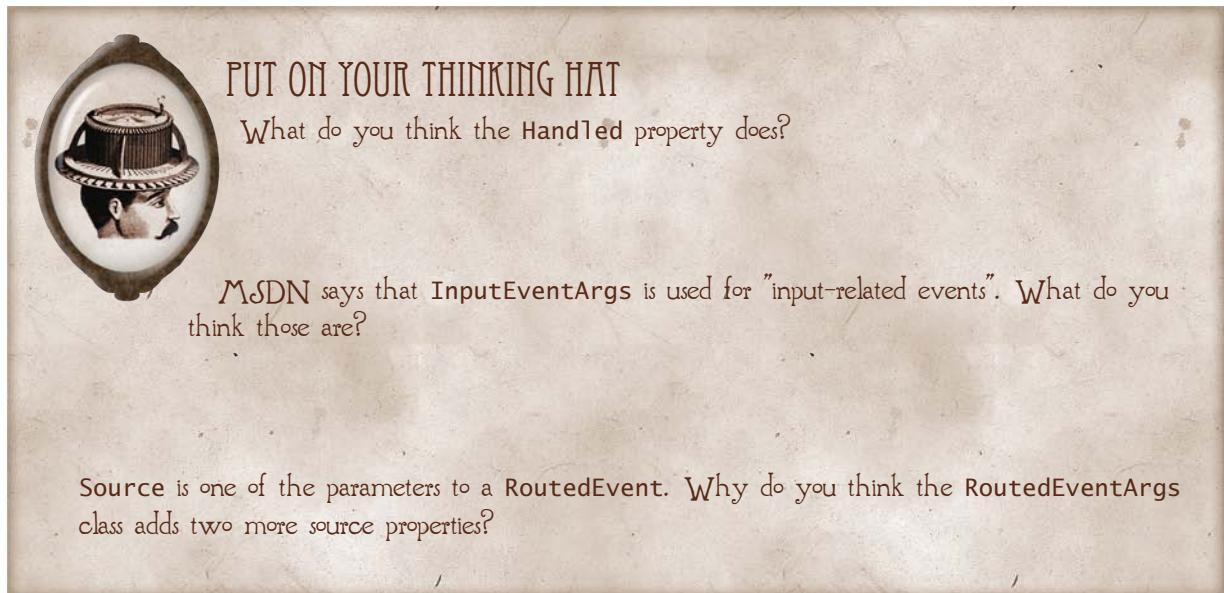
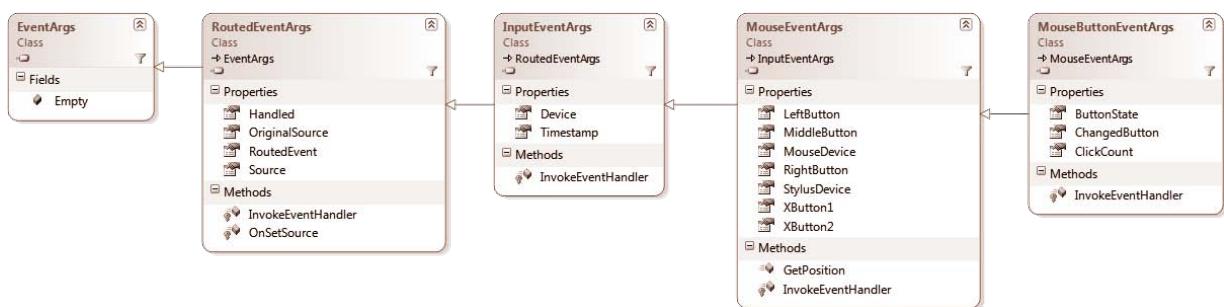
ROUTING STRATEGIES

Routed events can travel up or down the element tree, or they can behave more or less like a POCO event and go directly to a specific class. Events that start at the top of the tree and travel down are called tunnelling events, those that start at the event source and move up are bubbling events, and the ones that don't travel at all are called direct events.



EVENT ARGUMENTS

When we explored POCO events in Chapter 14, we saw that in .NET, an event always receives two parameters; the first is the object that raised the event, and the second is an instance of `System.EventArgs` or one of its descendants. That basic pattern doesn't change when you're working with routed events, but the second argument will be an instance of `RoutedEventArgs` or one of its descendants (`RoutedEventArgs` itself descends from `EventArgs`) and just as with POCO events, the argument can tell you a lot about the event. With routed events, it can also let you control how the event behaves. Here's one branch of the `RoutedEventArgs` hierarchy:





HOW'D YOU DO?

What do you think the `Handled` property does?

If `Handled` is set to true inside an event handler, the event will stop tunnelling or bubbling. (But there's a way around this, as we'll see.)

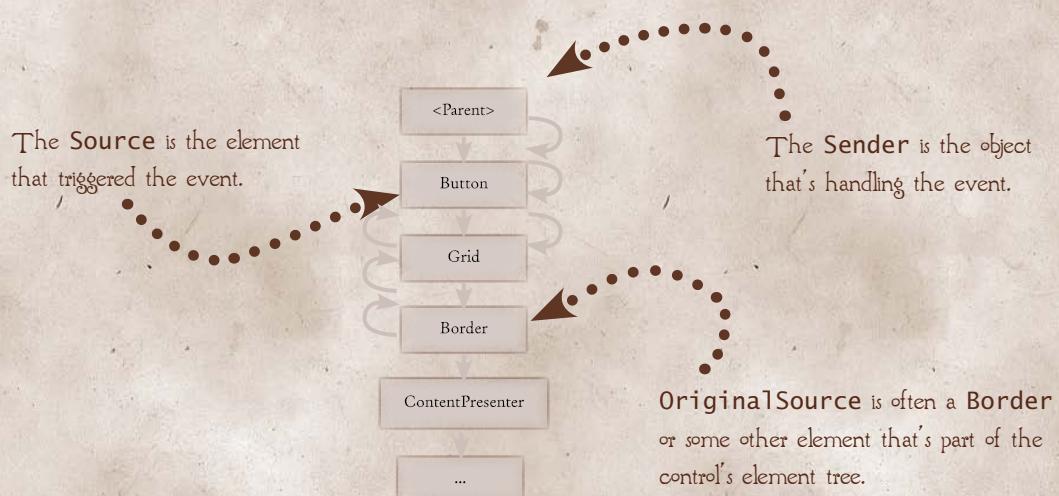
MSDN says that `InputEventArgs` is used for “input-related events”. What do you think those are?

Input events are those that are triggered by a real person interacting with the application via the keyboard, mouse, stylus, or anything else. (Do you want a Microsoft Surface machine as much as I do?)

`Source` is one of the parameters to a `RoutedEventArgs`. Why do you think the `RoutedEventArgs` class adds two more source properties?

Because the event can travel up and down the element tree, the element that originally triggered the event (`OriginalSource`), the element that sent the event (`Source`), and the element handling the event (`Sender`) may not be the same.

`OriginalSource` and `Sender` will be the same in most situations, but `OriginalSource` will often be some component of a composite control (and in WPF, almost all elements are composites).





EXPLORING ROUTEDEVENTARGS

In reality, you won't have to use the `Sender` parameter and `source` properties all that often, but when you need them, you really need them, so let's spend a little time playing with them. To test your new and improved version of the method:

O

CREATE A WPF PROJECT

You can call it anything you like.

I

CREATE AN EVENT HANDLER

We'll start by building a routine that will display the three parameters. Here's my version, but you can do something fancier. If you're feeling inspired, you might try building a `Window` to display the values in `TextBox` elements. The important thing is that the method have the same signature as mine. (This is the standard signature for any routed event handler.)

```
Private Sub ShowArguments(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim eventDetailsSB As StringBuilder = 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())

    MessageBox.Show(eventDetailsSB.ToString())
End Sub
```

2

TRIGGER THE EVENT

We'll look at different ways to trigger events in a minute, but for now, just set the `Loaded` event property of your main window (in the `Events` section of the Properties window in XAML view) to the name of your method.

3

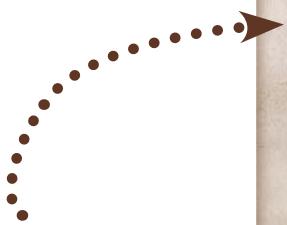
TEST IT

Run the application to make sure everything you've done so far is working properly. The `Sender`, `Source` and `OriginalSource` should all be the name of your `Window` class.

4

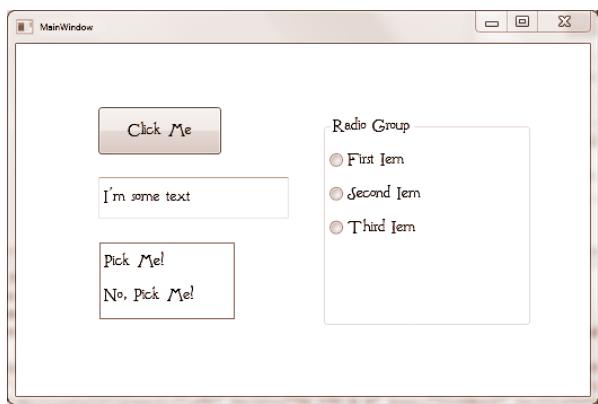
CREATE SOME WINDOW ELEMENTS

Next up, we'll add some elements to our window so that we can exercise our event handler. Add a Button, TextBox, ListBox and some RadioButtons (inside a GroupBox). Here's my version:



Make sure to set the Width of the Listbox so that it's bigger than the two items. (You'll see why in a few pages.)

```
<Window ... >
<Grid>
    <Button Content="Click Me" Name="button1" ... />
    <TextBox Name="textBox1" ... >I'm some text</TextBox>
    <ListBox Name="listBox1" ...>
        <ListBoxItem>Pick Me!</ListBoxItem>
        <ListBoxItem>No, Pick Me!</ListBoxItem>
    </ListBox>
    <GroupBox Header="Radio Group" Name="groupBox1" ...>
        <StackPanel>
            <RadioButton>First Item</RadioButton>
            <RadioButton>Second Item</RadioButton>
            <RadioButton>Third Item</RadioButton>
        </StackPanel>
    </GroupBox>
</Grid>
</Window>
```



5

ADD EVENT HANDLERS

You create handlers for routed events in exactly the same way that you create handlers for POCO events. You can do it in XAML by using the `Handles` clause or by calling the `AddHandler` statement. Let's start with the first option.

A

ADD A HANDLER IN XAML

When you set an event property in the WPF Designer, Visual Studio actually sets the `Handles` clause in the code behind. That's not a bad option, but it can be cleaner to set the property directly in the XAML. The syntax is simple. The events that are defined by an object are exposed as properties in XAML. To add the handler, you just set the property to the name of your handler.

Add the XAML to the opening `Window` tag to have the `ShowArguments()` method handle the `MouseDown` event. When you enter the tag, Intellisense will want to create a new event handler called `Window_MouseDown()` for you. Just ignore it and type "ShowArguments". Notice that you don't use the parentheses to reference the method in XAML.



BE AWARE, THIS IS STRANGE...

When you add the event property directly in the XAML like this, Visual Studio will update the Properties window, but it won't add the `Handles` clause to the method itself. That makes sense, because you wouldn't want the method called twice.

But what this means is that the WPF Designer is inconsistent. There are two ways to add an event handler: either by typing directly in the XAML or by using the Events tab of the Properties window. The two methods have different results, one results in adding a `Handles` clause, one doesn't. To make matters worse, you can't tell by looking at the Properties window which way the handler has been implemented; you have to inspect the XAML and the code.

This can bite you. It hasn't happened to me (which is surprising, really), but you can easily wind up with duplicate handlers. If, for example, having set the handler in the Properties Window, you then add it again in the XAML, the handler will be called twice. If you were to set the XAML to a different handler, Visual Studio would keep the original setting (in the Properties Window), and both would be called. Probably not what you want, and the only way you'll be able to track it down is by either looking at the Properties window (and why would you, if you're working directly in the XAML) or finding the `Handles` clause. My advice is to choose one technique or the other and stick with it. (Personally, I set all the event properties in XAML.)

5

ADD EVENT HANDLERS, CONTINUED

Now that we've seen the clean pure-XAML version (and the not-so-clean way the WPF Designer handles it), let's look at the other two, both of which you've seen.

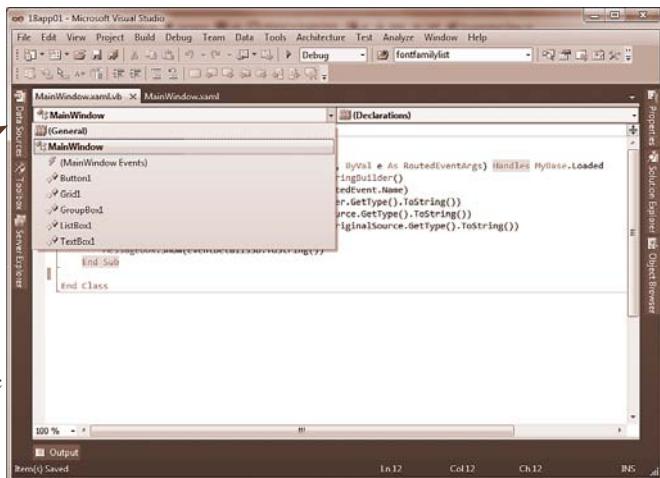
B

ADD A HANDLER USING ADDHANDLER

We've already seen how to do this, but it's a good time to review something about the Code Editor that I mentioned way back in Chapter 3 but we haven't had a reason to use yet. At the top of the Code Editor window are two combo boxes. The one on the left lists all the objects that have been declared in the current scope (called the **CLASS NAME COMBOBOX**), and the one on the right (called the **METHOD NAME COMBOBOX**) shows all the possible events. Just choose one from each, and the editor will stub out the handler for you.

This window displays the events available for the object you choose in the Class Name ComboBox.

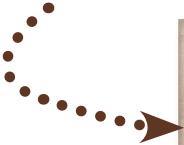
This ComboBox displays all the objects you've declared, either in the Designer or the code file.



Try it now. First, add a constructor to the window by selecting **MainWindow** in the Class Name ComboBox, and **New** from the Method Name ComboBox. Visual Studio will stub out the method for you. Add an **AddHandler** statement at the end of the method to add **ShowArguments()** as a handler for the **Listbox1.MouseDown** event.

The complete method is shown on the next page.

Do you remember what the `InitializeComponent()` method does?
(Check Chapter 15 if you've forgotten).



```
Public Sub New()
    ' This call is required by the designer.
    InitializeComponent()

    ' Add any initialization after the InitializeComponent() call.
    AddHandler ListBox1.MouseDown, AddressOf ShowArguments
End Sub
```



ADD A HANDLER USING THE HANDLES CLAUSE

You already know two ways to set the `Handles` clause: by using the Event tab of the WPF Designer Property Window, or by adding it directly in code. We've already added one event to the `ShowArguments()` method via the Property window, so let's add a second in code.

Add `Button1.Click` to the list of events the method handles. The first line of the method should look like this. (I've elided the parameters to save space.)

```
Private Sub ShowArguments(...) Handles MyBase.Loaded, Button1.Click
```

This one was added by
the Designer.

Add this one
now. Don't forget
the comma!

Notice that the Designer set the event to `MyBase.Loaded`. The `Loaded()` event is actually declared in the `FrameworkElement` class. You can override it for your window, but you must declare it as `Shows`.

6

TRY IT OUT

Before you run the application, think about what you expect to happen if Click and MouseDown are bubbling events (and they are). What do you think will happen when you click on each of these places?

- On the Window, outside any control
- On the Button
- On the ListBox
- Anyplace else
- On a ListBoxItem

In theory, the MessageBox should be displayed for the MouseDown on the Window, and then the Click event on the Button, right? And the ListBox should display the MessageBox for the MouseDown, and then (when the event bubbles up to the Window), it should be displayed again. The Window should catch the MouseDown everyplace else. At least, that's what I expected to happen when I was first learning about routed events.

Now run the application and try clicking different things. The results aren't going to be what you expect; that doesn't mean you've made a mistake. What happens when you click on each of these parts of the form? (Make sure you click on the ListBox background, as well as on the items it contains.)



UNDER THE MICROSCOPE

Event handlers aren't the only way to respond to an event. We'll examine triggers, which you can think of as a XAML version of event handlers, in Chapter 20.



LISTEN FOR HANDLED EVENTS

Well the `Window` caught the `MouseDown` when you clicked outside a control, and the `Button` caught the `Click` event (but the `Window` didn't get the `MouseDown`), and the `ListBox` only behaved as expected when you clicked on the background, not on a `ListBoxItem`. So what happened?

Routed events aren't broken; they do work the way I've described, but there's a little wrinkle we haven't seen yet. If you look back at the class diagram for the `RoutedEventArgs`, you'll see a `Handled` property. If that's set to true inside an event handler, the event still bubbles or tunnels, but most of the handlers that are defined for the event aren't called, because the event system assumes that the handlers aren't interested unless they tell it otherwise.

Controls like the `Button` and `RadioButton` are setting the `Handled` property to true, so our `Window`-level handler isn't getting called. In fact, they're not just handling the `MouseDown` event; they're converting it into an entirely different event (`Click`). That's a pretty common pattern in WPF—controls often consume low-level input events and replace them with something a little more useful.

If you do want to know about events that some low-level object is helpfully handling for you, you can use the `AddHandler()` method and pass true as the `HandledEventsToo` parameter. Let's try it out. Take out the event handler specification in the `Window` XAML tag and replace it with an `AddHandler()` call inside the constructor.

Now run the application and test the results. Closer to what we expected, isn't it? But did you notice that the `Button` is no longer clicking? It reports the `MouseDown`, but the `Click` event doesn't appear to be raised at all.



PUT ON YOUR THINKING HAT

What do you think is wrong with the `Button.Click` event? Don't spend a lot of time on the problem, but have a look at MSDN and check the forums to see if you can find the answer.



HOW'D YOU DO?

Did you find the answer? It's a bug in the Framework. Andrej Burger posted the solution on the MSDN forums (search for "Button does not click" if you want to read his post). If you interrupt processing by displaying a `MessageBox` or opening another Window (or, as Andrej found, setting a breakpoint so that Visual Studio opens the debugger), the `Click` event goes missing.

There's a lesson to be learned here. The vast majority of the bugs in your application will be yours. When you track most of them down, you'll probably feel a little silly. (That doesn't change with experience.) But sometimes it's not your fault. There ARE bugs in the Framework. Not a lot, and that shouldn't be your first (second, or third) guess, but it happens.



8

WORK AROUND THE BUG

Obviously, you can't fix bugs in the Framework itself, so you need to find a way to work around them. Sometimes that's easy; sometimes it's hard. Sometimes it requires completely re-thinking how you're doing something. I once spent a month redesigning an application to work around a bug in the way tab controls handled data binding in the first version of WPF. (I'd really like to have that month of my life back.)

This time it's easy, because what we're doing isn't very important. The problem is with interrupting the UI, so we just won't do that. We can replace the `MessageBox` with a `Console.WriteLine()` call:

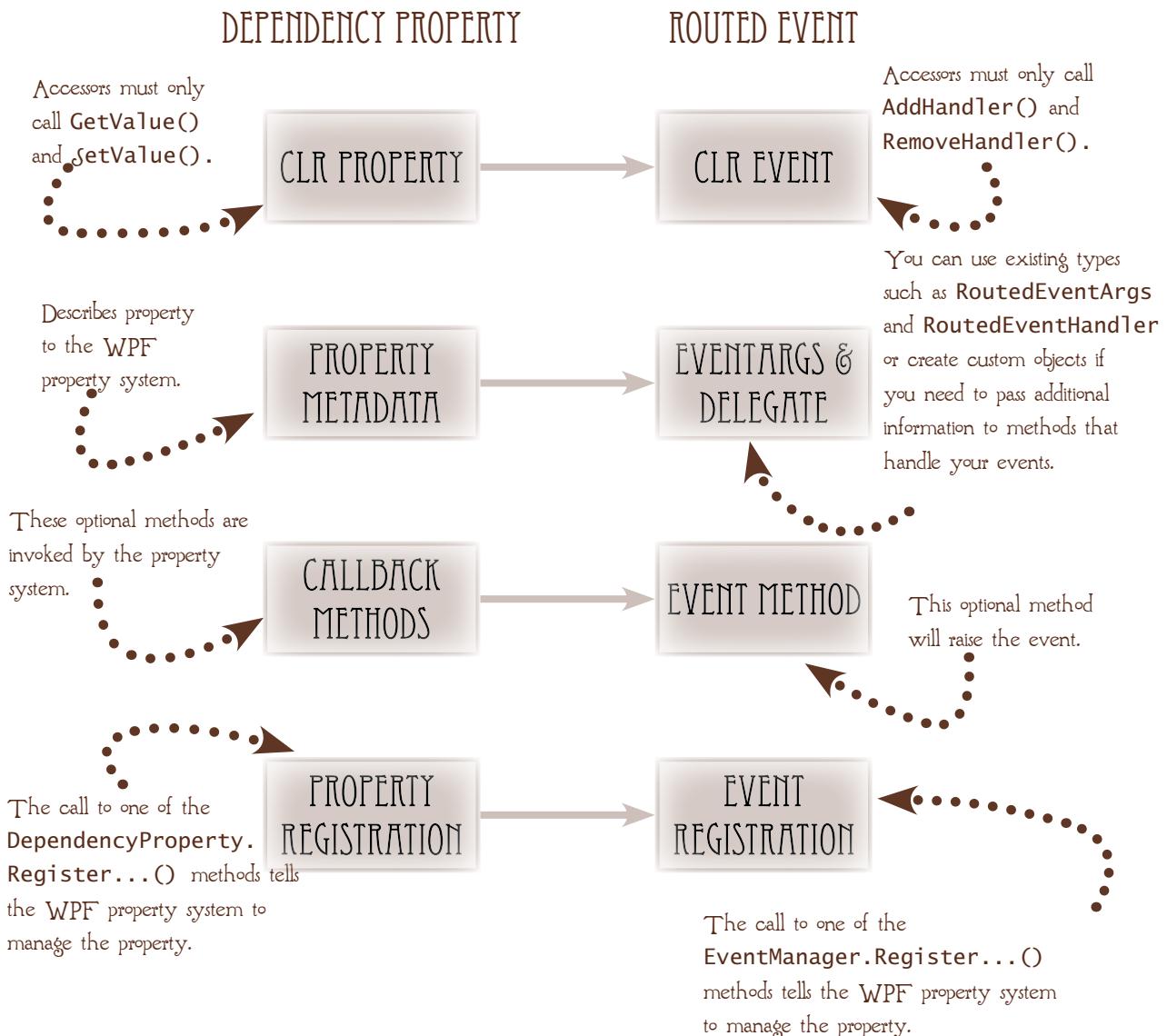
```
Console.WriteLine(eventDetailsSB.ToString())
```

Make that change to the `ShowArguments()` method, and rerun the application. It should work the way it's supposed to now, but you'll have to look in the Output window in Visual Studio to see the `EventArgs` details.

If the Output window isn't displayed automatically when you run the application, you can show it from the View menu or by pressing **Ctrl+W, O**. (Note: Many programmers routinely use `Console.WriteLine()` instead of the `MessageBoxes` we've been using to display internal information like this, but I personally find it no easier to call and a little cluttered to read.)

EVENT ELEMENTS

You've probably noticed that routed events are a lot like dependency properties: WPF takes the basic POCO object and adds some additional functionality. It probably won't come as a surprise that the process for creating one is very similar. You need to create the objects involved in a different order, but the components are the same:



POCO EVENTS REVISITED

Back in Chapter 14 we looked at the basic syntax for the `Event` statement that allows you to declare an event and specify the delegate class. Before we look at how to register routed events, let's look at the two other ways you can declare an event in VB.

DECLARING AN EVENT WITHOUT A DELEGATE

In the same way that you can use some syntactic sugar to get the VB compiler to write a backing field and the standard get and set accessors of a property, you can simply provide the parameters required by an event, and the compiler will write the delegate for you:

Event <Name>([<parameter[, ...]>])

```
Public Event SomethingHappened(EventId As Integer)
```

You'll see this syntax a lot in examples on MSDN and elsewhere, mostly I think because it's closest to the VB6 event declaration, and the original designers of .NET worked hard at minimizing the culture shock VB6 programmers face when making the switch.

But I don't really recommend that you use it. Yes, it saves some typing, but it doesn't conform to the conventional pattern of events in .NET, and in the long run that adds to the number of things you need to learn. You know how events work, but now you have to remember that there are some events that don't work like that. Ugly.



IN MY VIEW

My opinion of the version of the `Event` declaration that accepts parameters instead of declaring the event `As` a delegate type isn't a law. It's just my opinion. I don't like oranges, either. Doesn't make them bad.

In the spirit of fair play, I'll point out that advocates of this technique argue that declaring a new delegate type for every event (assuming you can't reuse one of the existing .NET delegates) can add needless clutter to your class design. And that's true. On the other hand, if you're using the same parameter list for several events, you're adding a bunch of conceptual overhead (not to mention typing) to your code.

What do you think?

CREATING CUSTOM EVENTS

The third alternative for declaring events in VB is to use the `Custom` keyword and an extended syntax that includes the event-raising method. This is the form you'll use when you create routed events:

The three clauses are like the `get` and `set` accessors of a property declaration.

To use the `Custom` keyword, you must declare the `Event As` a delegate type, either one you've created or one of the existing .NET delegate classes.

```
Custom Event <Name> As <Delegate>
    ►AddHandler(ByVal value As <Delegate>)
        Me.AddHandler(<Name>Event, value)
    End AddHandler

    RemoveHandler(ByVal value as <Delegate>)
        Me.RemoveHandler(<Name>Event, value)
    End RemoveHandler

    RaiseEvent(ByVal sender As Object, ByVal e as <Delegate>)
        Me.RaiseEvent(e)
    End RaiseEvent
End Event
```



EVENTS IN DEPTH

If you'd like to explore some of the more complex aspects of event handling in Visual Basic, you can't do better than to start with the article Ken Getz wrote for *MSDN Magazine* in 2005, "Discover a Series of Fortunate Event Handlers in Visual Basic". To find it, search for "fortunate event handlers" AND "Ken Getz" in your favorite search engine.

CREATING ROUTED EVENTS

You know that in order to register a dependency property with the WPF property system you call one of umpteen bazillion `Register..()` methods. It probably won't be a surprise that you use the same technique to register a routed event. The good news is that there's only a single version of the `RegisterRoutedEvent()` method:

```
Public Shared ReadOnly <ClrEventName>Event As RoutedEvent =  
    EventManager.RegisterRoutedEvent("<ClrEventName>", _  
        RoutingStrategy.<type>, _  
        GetType(<delegate>), GetType(<ownerClass>))
```

Routed events are declared `Public`,
`Shared` and `ReadOnly`.

This is a member of the
`RoutingStrategy` enumeration that has
the values `Tunnel`, `Bubble` and `Direct`.

The even better news is that Visual Studio has an Intellisense snippet that makes the whole process of creating routed events pretty painless. To insert the snippet, press `Ctrl+K, X`, then choose the `WPF` category (you'll probably have to scroll down to see it) and "Add a Routed Event Registration".

Visual Studio will add the code for the `Custom Event` declaration and the `RegisterRoutedEvent()` method. (You can see the result in the screenshot.) All you need to do is change the name of the event, the delegate type (if necessary—the `RoutedEventArgs` class is the WPF equivalent of the `EventArgs` delegate you used for POCO events), and the name of the owning class.

The `RoutingStrategy` argument defaults to `Bubble`, which is the most common, but it's an enumeration, so you can easily change it to `Direct` or `Tunnel` if that's what you need.

18app01 - Microsoft Visual Studio

File Edit View Project Build Debug Team Data Tools Architecture Test Analyze Window Help

Debug fontfamilylist

MainWindow.xaml.vb X MainWindow.xaml

MainWindow Event1

```
' This call is required by the designer.  
InitializeComponent()  
  
' Add any initialization after the InitializeComponent() call.  
AddHandler ListBox1.MouseDown, AddressOf ShowArguments  
End Sub  
  
Public Custom Event Event1 As RoutedEventHandler  
  
    AddHandler(ByVal value As RoutedEventHandler)  
        Me.AddHandler(Event1Event, value)  
    End AddHandler  
  
    RemoveHandler(ByVal value As RoutedEventHandler)  
        Me.RemoveHandler(Event1Event, value)  
    End RemoveHandler  
  
    RaiseEvent(ByVal sender As Object, ByVal e As System.Windows.RoutedEventArgs)  
        Me.RaiseEvent(e)  
    End RaiseEvent  
End Event  
  
Public Shared ReadOnly Event1Event As RoutedEvent =  
    EventManager.RegisterRoutedEvent("Event1", _  
        RoutingStrategy.Bubble, _  
        GetType(RoutedEventHandler), GetType(Window1))
```

100 %

Output

Ready Ln 28 Col 46 Ch 46 INS



PUT ON YOUR THINKING HAT

Add an event to the project we've been working on called `MyCustomEvent`. It should use a custom argument called `CustomArgs` that adds an int `MyProp`. Call the custom delegate `MyCustomHandler`. You can keep the default `RoutingStrategy.Bubble`.



HOW'D YOU DO?

It doesn't matter if you declared your objects in a different order, as long as the project will compile.

```
Public Class CustomArgs
    Inherits RoutedEventArgs
    Public Property MyProp As Integer
End Class

Public Delegate Sub MyCustomHandler(ByVal sender As Object,
    ByVal e As CustomArgs)

Public Custom Event MyCustomEvent As MyCustomHandler
    AddHandler(ByVal value As MyCustomHandler)
        Me.AddHandler(MyCustomEventEvent, value)
    End AddHandler

    RemoveHandler(ByVal value As MyCustomHandler)
        Me.RemoveHandler(MyCustomEventEvent, value)
    End RemoveHandler

    RaiseEvent(ByVal sender As Object, ByVal e As CustomArgs)
        Me.RaiseEvent(e)
    End RaiseEvent
End Event

Public Shared ReadOnly MyCustomEventEvent As RoutedEvent =
    EventManager.RegisterRoutedEvent("MyCustomEvent",
        RoutingStrategy.Bubble,
        GetType(RoutedEventHandler), GetType(MainWindow))
```



PUT ON YOUR THINKING HAT

Most of the events in any application are input events that are used to respond to things like keyboard input and mouse or stylus interactions. You'll often write handlers to respond to these events, but you'll rarely need to create them, because they're already exposed by the `UIElement` and `FrameworkElement` base classes. To get a sense of what's available to you, check the MSDN library for these two classes, and find the event you'd use in each of the following situations:

When the user moves the mouse over the control

When the control loses focus

When the control changes size

When the user types a character

When the user presses a mouse button, but before they release it

When the visibility of a control changes



HOW'D YOU DO?

When the user moves the mouse over the control

`MouseEnter`

When the control loses focus

`LostFocus`

When the control changes size

`SizeChanged`

When the user types a character

`KeyUp`

When the user presses a mouse button, but before they release it

`MouseDown`

When the visibility of a control changes

`IsVisibleChanged`



TAKE A BREAK

That's it for routed events for now. We'll be looking at them again in Chapter 20 when we examine triggers, but why don't you take a little break before we move on to commands?



REVIEW

Here's a real test for you. You know that events often come in tunnel/bubble pairs, and that by convention the name of the tunnelling version is prefaced with the word "Preview". You also know about the `Handled` property. Now, there's a convention in the Framework that if the `Handled` property of the tunneling event is set to true, the bubbling event isn't raised. So, here's your job: Write a pair of events, one tunneling and one bubbling, and then write the method that raises the events. Include the check that doesn't raise the bubbling event if the tunneling event is handled.

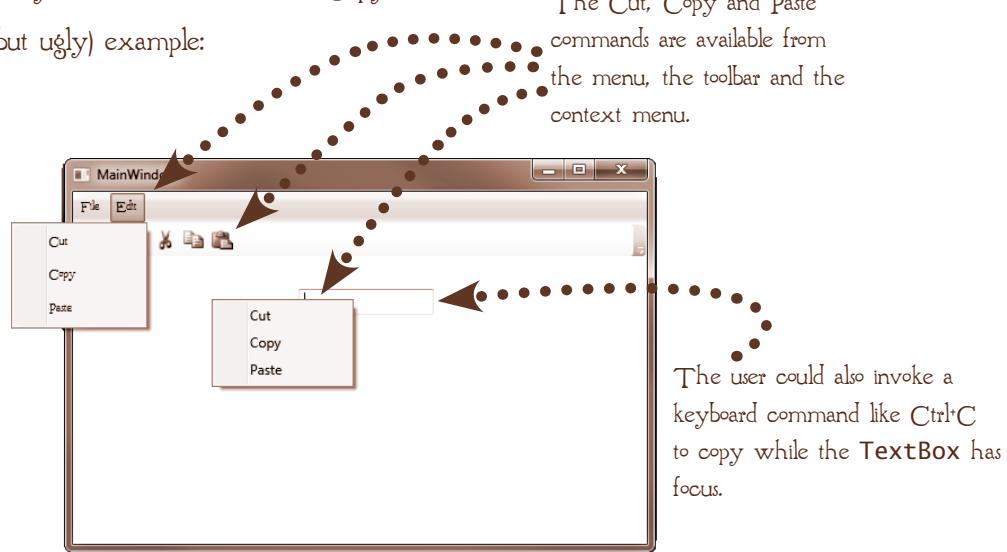


ROUTED COMMANDS

The third component in the suite of WPF interactions is the routed command. Unlike the other two, routed commands don't have an equivalent POCO object; they're a completely new abstraction. (Well, new to .NET. The concept of a command has been around in other development environments for years.)

Routed commands closely model the way we think about an application when we interact with it: We issue a Copy command. We might do it from the main application menu, from the toolbar, from a context menu or by pressing `Ctrl+C`, or by some other way that a clever interface designer has thought of, but whatever we actually do, what we think is "Copy".

Here's a typical (but ugly) example:

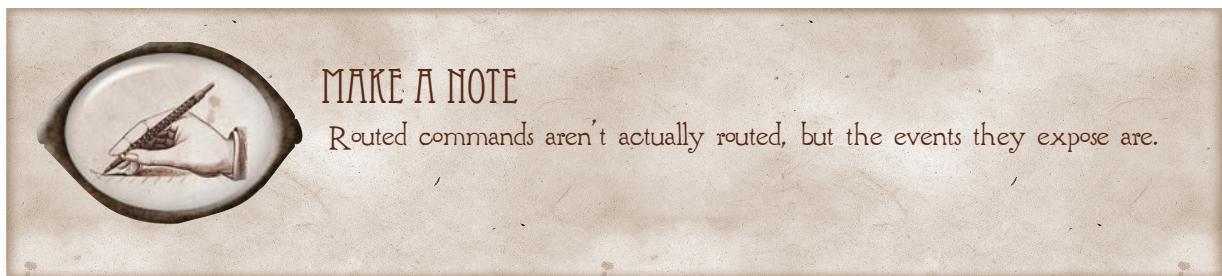
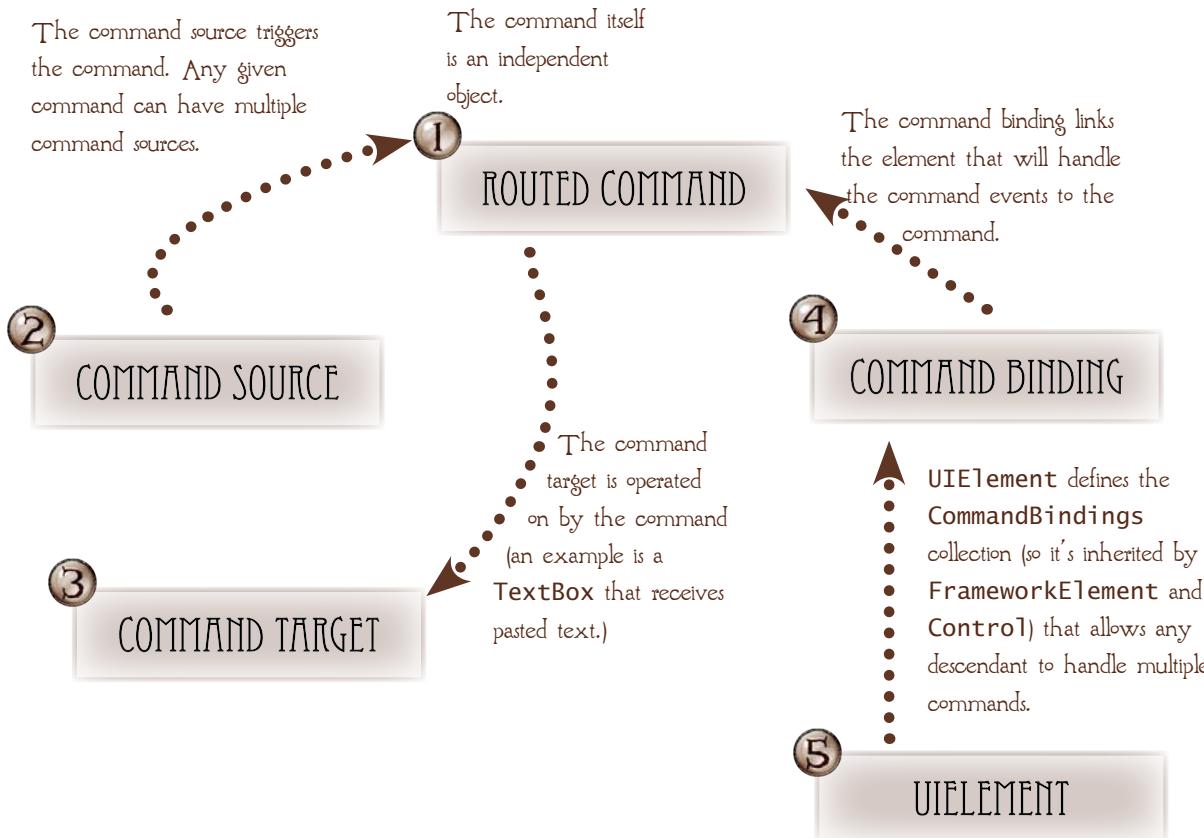


You can handle this situation with POCO events (programmers have been doing it forever), but it's tedious, because every object involved needs to respond to the event and pass it on to the code that performs the operation. The situation is complicated by the fact that in a good interface every object should be able to tell whether the action is actually valid at the moment. If the action isn't valid, the command should disable itself (no point trying to copy an empty selection, for example), which means that other objects that might effect that need additional events, and...did I mention tedious?

WPF routed events make it simpler because most of the events can be handled at the window level, but it's still pretty tedious. This is exactly the problem that routed commands address.

THE LOGICAL LEVEL

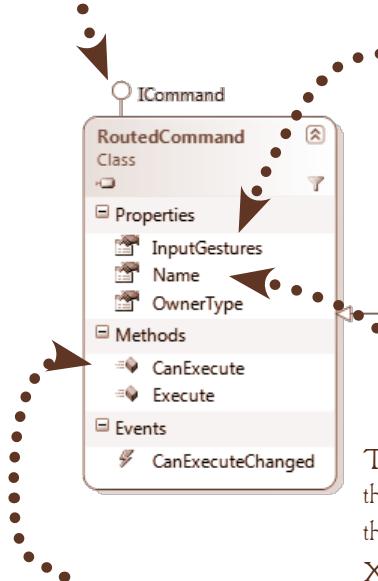
Routed commands are actually pretty simple to use, but the class diagram of all the objects involved is pretty complex (and frankly, a little scary), so let's begin by looking at how the pieces fit together:



FRAMEWORK COMMAND LIBRARY

Despite all the objects involved, routed commands are a lot easier to implement than routed events or dependency properties. They don't sit on top of POCO classes, so there's none of that messy registration or special requirements for accessors. Better yet, the Framework already defines dozens of the most common commands, complete with their keystroke shortcuts, that are ready to use "out of the box".

Technically, a routed command is any object that implements the **ICommand** interface. The Framework implements two: **RoutedCommand** and its descendant, **RoutedUICommand**.

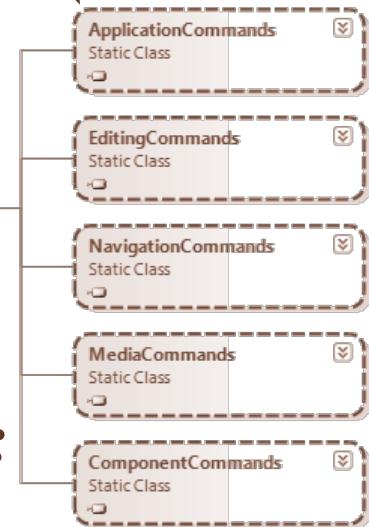


An **InputGesture** represents something like a control key combination (**Ctrl+C**) or a mouse click. Any given command can be associated with multiple **InputGesture** objects via this collection.

The **Name** is a string that you'll use to identify the command, mostly in XAML, but also in code.

The **CanExecute** command is typically used for enabling and disabling commands in the UI, while **Execute** is used to perform the action represented by the command.

The routed commands defined by the Framework are static classes.



The contents of the **Text** property will be displayed in the interface. As a bonus, the commands defined in the Framework will be localized based on the user's Locale setting in Windows.



PUT ON YOUR THINKING HAT

Time for some more research. You'll use the routed commands defined by the Framework in almost every application you write, so it's a good idea to be familiar with what's available. Can you answer the following questions regarding the Framework command classes? (The `EditingCommands` class is defined in the `System.Windows.Documents` namespace. The others are defined in `System.Windows.Input`.)

To populate File and Edit menus with the conventional commands (Save, Close, Cut, Copy, etc.), you'd use commands from which class?

What key gesture is defined for the Redo command?

If you were building a widget to display a slideshow, you might want to include some buttons to control the display. Which two commands could you bind to a button that paused the display?

Where would you look for commands to use in a word processing application?

Most of the commands in this class are supported by which three standard controls? (Hint: You'll have to look at one of the command properties. The description of the class only mentions one of them.)

What class and command would you use to represent moving the cursor left by one position?

What command would you use to allow the user to refresh the display of a Window? What gesture does it have?



HOW'D YOU DO?

Were you surprised at how many commands are available for you?

To populate File and Edit menus with the conventional commands (Save, Close, Cut, Copy, etc.), you'd use commands from which class?

ApplicationCommands

What key gesture is defined for the Redo command?

Ctrl + Y

If you were building a widget to display a slideshow, you might want include some buttons to control the display. Which two commands could you bind to a button that paused the display?

MediaCommands.Pause (if you have separate buttons) or *TogglePlayPause* (if one button performs both tasks).

Where would you look for commands to use in a word processing application?

EditingCommands

Most of the commands in this class are supported by which three standard controls?

RichTextBox, TextBox and PasswordBox

What class and command would you use to represent moving the cursor left by one position?

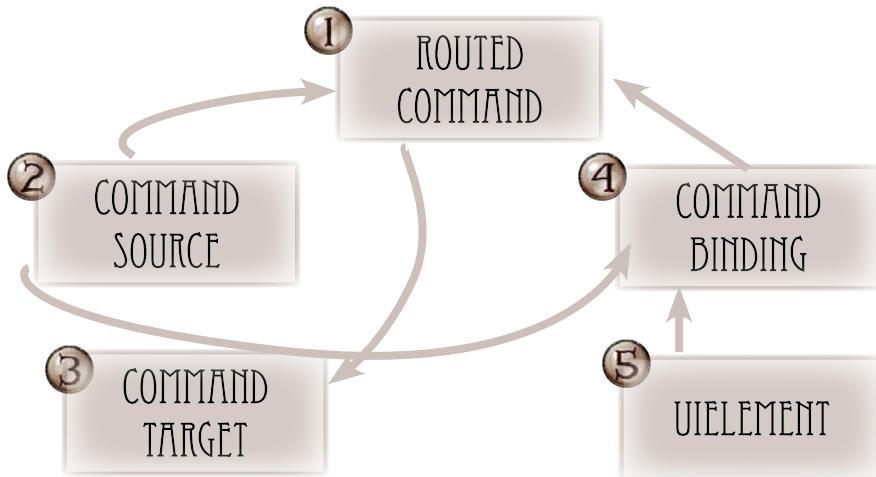
ComponentCommands.MoveLeft

What command would you use to allow the user to refresh the display of a window? What gesture does it have?

NavigationCommands.Refresh, F5

COMMAND OBJECTS, AGAIN...

Okay, now you know what commands are available in the Framework, but how do you implement them? Let's start by looking at the objects involved in a little more detail:



- ➊ The first object is, of course, the routed command itself. You can create your own command by implementing the `ICommand` interface or use one of the predefined commands.
- ➋ A command source implements the `ICommandSource` interface, which defines three dependency properties: `Command`, `CommandParameter` and `CommandTarget`. Many of the WPF widgets already implement this interface and you can do it yourself. (You already know how; it's just an interface with three properties.) Most command sources also handle the `CanExecuteChanged` event to enable and disable themselves if that's appropriate.
- ➌ The command target is the object that receives the effects of the command. If it isn't specified explicitly, the object that has focus will be the command target, and that's almost always what you want to have happen.
- ➍ The command binding specifies the methods to be executed when the `PreviewCanExecute`, `CanExecute`, `PreviewExecuted` and `Executed` events occur.
- ➎ Command bindings live in the `CommandBindings` collection property of the object that implements them.

HOOKING UP COMMANDS

There are a lot of objects involved in the commanding process, but there are only three steps to the process. You can perform the steps in any order.



SET THE COMMAND PROPERTY ON THE SOURCE

The `ICommandSource` interface exposes three properties: `Command`, `CommandParameter` and `CommandTarget`; any widget that can act as a control source will expose these as simple properties (they're actually dependency properties) that can be set in `XAML` or code:

```
<MyObject Command="Cut" />
```

or

```
MyObject.Command = ApplicationCommands.Cut;
```



You don't need to specify the class in `XAML` if you're using one of the predefined command classes, but you will if you've defined your own command.

In most situations that's all you'll need to do, but you can also set the `CommandParameter` and `CommandTarget` properties if you need to. The syntax is identical.



WRITE THE HANDLERS

You'll need two. The handler for the `CanExecute` event needs to match the signature for the `CanExecuteRoutedEventHandler`, which is:

```
Sub <MethodName>(sender As Object, e As CanExecuteRoutedEventArgs)
```

By convention, the method is named `<CommandName>CanExecute`. It should set the `CanExecute` property of the `CanExecuteRoutedEventArgs` to `true` or `false` to reflect whether the command is available.

The delegate for the `ExecutedRoutedEventHandler` is:

```
Sub <MethodName>(sender As Object, e As ExecutedRoutedEventArgs)
```

And by convention, it's named `<CommandName>Executed`. The handler performs the actual logic of executing the command.

C

CREATE THE COMMAND BINDING

The next step is to connect the command to the object that will handle it by adding a `CommandBinding` to the `CommandBindings` collection. You'll almost always bind the command to the top-level object (`Window` or `Page`), and you can do it in XAML or code:

Notice that you don't need to explicitly create delegates here; just pass the names of the appropriate methods (which must, of course, have the correct signature).



```
<Window ...>
<Window.CommandBindings>
  <CommandBinding Command="Cut"
    Executed="CutCmdExecuted"
    CanExecute="CutCmdCanExecute" />
</Window.CommandBindings>
...
</Window>
or
CutCmdBinding As New CommandBinding(
  ApplicationCommands.Cut, CutCmdExecuted, CutCmdCanExecute);
this.CommandBindings.Add(CutCmdBinding)
```



PUT ON YOUR THINKING HAT

Here's a very simple window with a single button. Connect the button to the `ApplicationCommands.Close` and create a `CommandBinding` for the `Window` to the `CanExecute` (which should just set the `CanExecute` property of `e` to `true`) and `Executed` (which should close the `Window`).

You'll need to write the methods in code, but you can do the binding in XAML.





HOW'D YOU DO?

Here's my version. Yours might be a little different, and that's fine as long as the application does what it needs to do.

Here are the critical bits of the XAML:

```
<Window ...>
    <Window.CommandBindings>
        <CommandBinding Command="Close"
            CanExecute="CloseCmdCanExecute"
            Executed="CloseCmdExecuted"/>
    </Window.CommandBindings>
    <Grid>
        <Button ... Command="Close"/>
    </Grid>
</Window>
```

And here are the two handlers, which can go anywhere in the class file (outside the constructor):

```
Sub CloseCmdCanExecute(ByVal sender As Object,
    ByVal e As CanExecuteRoutedEventArgs)
    e.CanExecute = true
End Sub

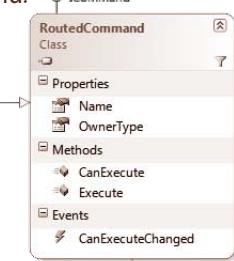
Sub CloseCmdExecuted(ByVal sender As Object,
    ByVal e As ExecutedRoutedEventArgs)
    Me.Close()
End Sub
```

ROUTED COMMAND CLASSES

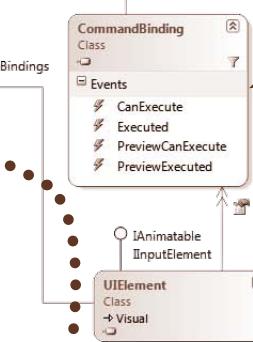
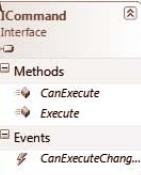
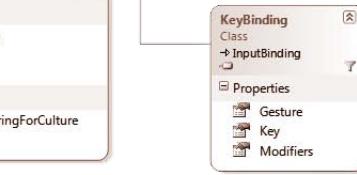
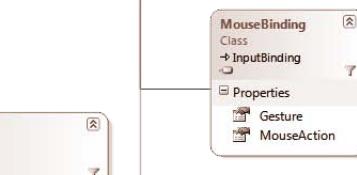
Now that you've seen that it's actually pretty easy to use routed commands, let's have a look at that scary class structure:

① Here's the command.

`ICommand` is the minimum requirement, but most commands descend from `RoutedUICommand`.



② To be a command source, the class needs to implement `ICommandSource`.

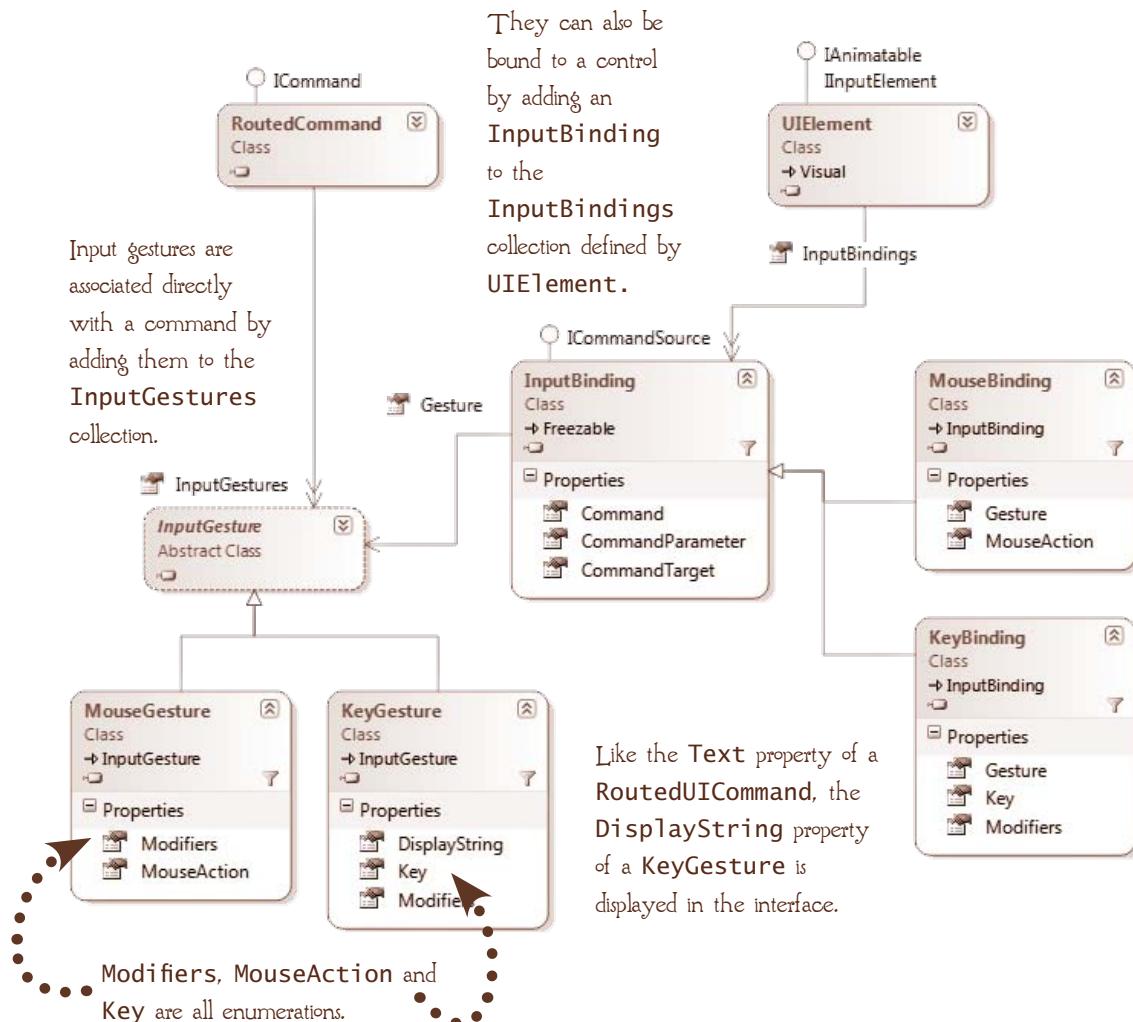


③ The command target only applies to `InputBindings`.

④ Because `CommandBindings` is implemented so high up in the WPF hierarchy, it's available to most WPF objects.

INPUT GESTURES & BINDINGS

The only classes in that scary diagram that we haven't worked with are the ones that provide keyboard and mouse input to a command: the KeyGesture, MouseGesture, KeyBinding and MouseBinding. An input gesture represents input from a device like the mouse or keyboard. MouseGesture and KeyGesture are the only two devices defined in the Framework (a stylus generates MouseGesture events), but in theory you could define other types of devices.



CREATING ROUTED COMMANDS

Because routed commands are first-class elements of the Framework rather than wrappers for CLR objects, they're much easier to create than dependency properties or routed events. In fact, they don't require any techniques you don't know already. But you do need to make some design decisions:

WHERE SHOULD THE COMMAND BE DEFINED?

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. But a property of which class?

SHOULD GESTURES BE ASSOCIATED WITH THE COMMAND?

If you're defining your own commands, you'll need to decide whether a gesture should be associated with the command (they're optional), and if so, what that gesture should be. Be careful not to step on any of the Windows conventions. If you redefine Ctrl+C to close the application instead of copy text, you're going to confuse people.

SHOULD GESTURES BELONG TO THE COMMAND OR THE BINDING?

Most input gestures are associated with the command itself so that they're available (and consistent) throughout the application. To assign an input gesture to a routed command, you simply create a new `MouseGesture` or `KeyGesture` object and add it to the `InputGestures` collection using normal VB syntax. (Both `RoutedCommand` and `RoutedUICommand` expose versions of their constructors that take the `InputGestureCollection` directly.)



TAKE A BREAK

That's it for routed commands. You don't need an exercise because you already know how to do what you need to do. So why don't you take a break before you finish up the chapter Review?



REVIEW

What property do you set in XAML to designate an object as a command source for the specified routed command?

Assuming the conventions are followed, which of these is a CLR event and which is a routed event identifier?

DoSomething

DoSomethingEvent

`UIElement` exposes two different binding collections. What are they? What does each do?

How do you register a routed event when you want to receive event notification even if the event is marked as handled?

To define a routed command, you declare a property. Most of your commands will be an instance of what Framework class?

Write a XAML snippet that binds a `Window` to the `Save` command, using `SaveCmd<Event>` for the handler names:

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



WPF GRAPHICS

19

Have you ever noticed that Windows applications tend to look very much alike? I won't say they're boring, but it's unusual to see the kind of variety you'll find on the Web, for example. There are a few standard layouts, and all the controls look almost identical.

It isn't that the designers of these applications lack the imagination to build innovative interfaces or graphic designs; it's that until the release of Windows Presentation Foundation (WPF), making significant changes to a control was really, really hard. Even changing the appearance of an application was difficult, because most of the standard controls only exposed a few appearance properties, so it was almost impossible to get a consistent appearance if you strayed too much from the standard set.

We've already seen that WPF makes it easy to combine, reorganize and restructure controls by simply changing the way they're defined in XAML. In Chapter 21, we'll look at control templates that make that even easier. But first, in this chapter, we'll look at the basic tools in your design toolbox: the pens that draw lines, brushes that fill areas, and the colors that define them both. Then we'll take a quick look at WPF typography, because it's rather different from working with fonts in most environments and finally at a few of the special effects that the Framework defines for you, like adding a drop shadow or blurring an image.



UNDER THE MICROSCOPE

WPF has extremely sophisticated 2- and 3-dimensional graphics capabilities.

If you're interested in pursuing more advanced graphics than we have space for here, the CodePlex open source site is a good place to start looking for sample applications.

But that's not what we're going to explore in this chapter. We're going to look, instead, at the graphic objects that control how your WPF interface looks.

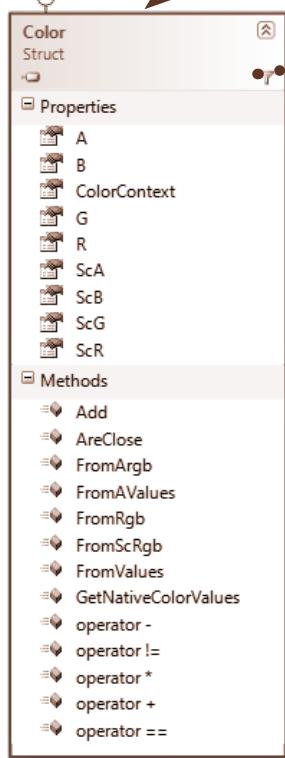


FITTING IT IN

You have complete control over the appearance of just about anything you'd want to display to the user in WPF. And it's easy; just set the appropriate properties.

In the next chapter, we'll look at a couple of ways to set the properties for all the elements in your application in one place, but we'll start by looking at the properties and the classes that expose them.

The most fundamental characteristic of anything we see is its color, which in WPF is represented by the `Color` structure.

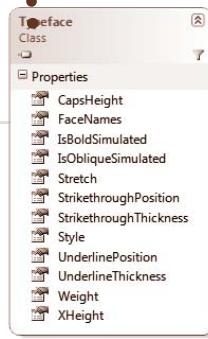
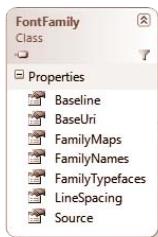


The text, background and border of this `TextBox` all have a `Color`.



This is just a standard WPF `TextBox`, but its appearance has been modified (a lot!).

Like a lot of things in WPF, typography is complex but easy to use. We'll explore the basics in this chapter.

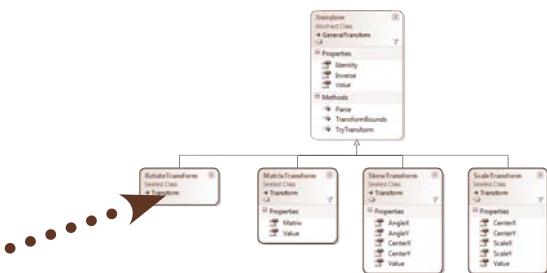


WPF Controls can be transformed, which in the context of graphics means they can be moved, sized or skewed. We'll look at the basics of the `Transform` object here, and in the next chapter we'll take a quick look at how to use transformations to animate things.

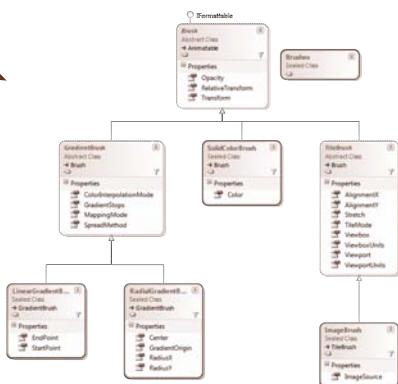
The `TextBox` has been skewed. Not something I'd recommend as a rule, but it can be useful in some situations.

The `TextBox` has a drop shadow, one of the effects defined in the Framework Library.

You apply color to an object with a `Pen` (for lines) or a `Brush` (for areas).



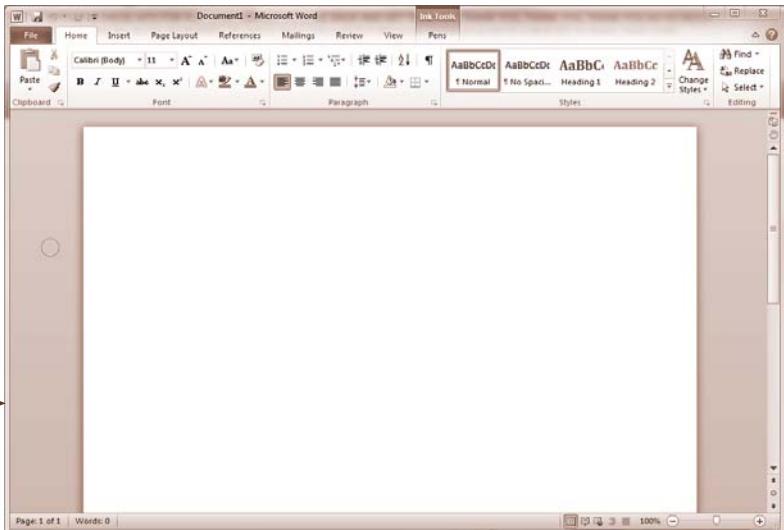
WPF supports a kind of graphic manipulation called a pixel shader. The bad news is that pixel shaders are written in a language called `HSL`, which makes VB look like baby talk. The good news is that two common effects, blurring and shadowing, are defined for you in the Framework.



AN UNFAIR COMPARISON

Both of these screens display text and a few controls. Which do you think is more attractive? Granted, while I think the Mindbloom site is seriously cool, you might not want to spend all day every day working with it, the way many people do with Microsoft Word. But my point here is that if you wanted to (and had the artistic skill), you could write a word processor in WPF that looked more like the Mindbloom site than Office.

This is Microsoft Word.
It's not ugly, and it's pretty
functional, but it's not all that
exciting, really.



This is the home page
of mindbloom.com. The
leaves and branches are a
kind of interactive todo list.
I think it's pretty cool.





TASK LIST

In this chapter we'll explore the basic building blocks you'll use to control the appearance of your WPF applications.



COLOR

We'll start by exploring the `Color` structure and two useful classes the Framework defines: `System.Windows.SystemColors` and `System.Windows.Media.Colors`.



BRUSHES

In WPF, a `Brush` is used to fill an area. But unlike an artist's brush, in WPF you aren't limited to a single color. You can load your brush with a gradient (a set of colors that blend into each other), or an image, and you can determine how the contents of the brush are displayed relative to the area they're filling.



PENS

A WPF `Pen` is used to draw a line, most often the outline of a shape. WPF pens can be solid or dashed, and you can specify the shape of the end and the corner. But you don't specify the color of a pen, you specify its `Brush`. That means all the capabilities of a WPF `Brush`, including gradients and images, and can be used to outline your controls.



TYPOGRAPHY

To typographers, type is considerably more complex than just specifying a typeface; they talk about leading and kerning and glyphs. WPF supports all of those things, but don't panic; we'll limit our exploration to the kinds of things you're likely to do in an application window.



EFFECTS

Finally, we'll explore the graphic effects that are defined by the .NET Framework and why we're not going to explore how to build them.

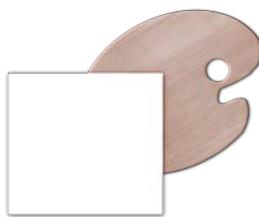


COLOR

In order to specify a color exactly, you need to use a color model, of which there are several. CMYK (Cyan, Magenta, Yellow, Black) is used by printers because those are the colors of ink from which all other colors are mixed. HSV (Hue, Saturation, Value) is used in many graphics programs because it's a fairly close match to how we perceive color and the way artists think about it. Computer monitors and most programming languages use some form of RGB (Red, Green, Blue), of which there are several. In the RGB model, the values for red, green and blue are specified as numbers, usually integer values between 0 and 255. Many versions of RGB, including the two supported by .NET, also support an alpha value, which determines the transparency of the color.

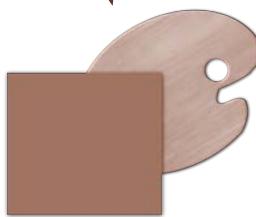
R = 255
G = 255
B = 255

Full amounts of all the colors result in white.



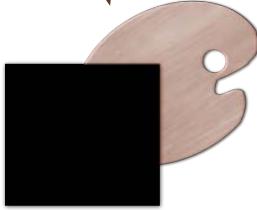
R = 127
G = 127
B = 127

Equal values of all colors result in grey.



R = 0
G = 0
B = 0

When 0 is specified for all the values, the result is black.

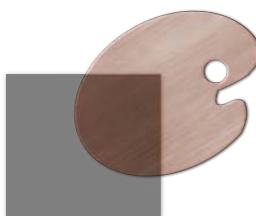
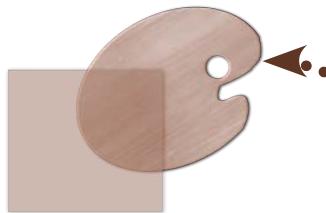
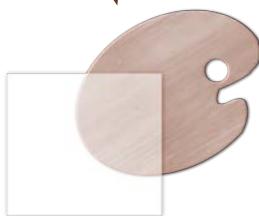


The values don't have to be equal, of course. For example, this would render a pure red (but I can't show you that here).

R = 255
G = 0
B = 0

A = 127
R = 255
G = 255
B = 255

The alpha value determines how transparent the color is.



Can you guess the ARGB values of these two examples?

THE COLOR STRUCTURE

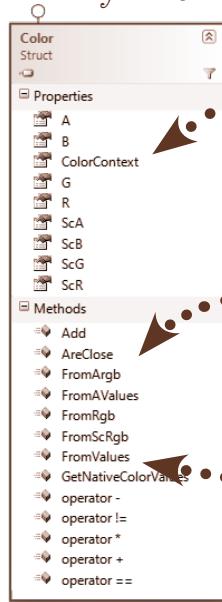
In .NET, colors are represented by the `Color` structure. The Framework also defines two classes that provide access to predefined color structures. The `System.Windows.Media.Colors` class contains a set of named colors based on the common "Internet-safe" color set. The `System.Windows.SystemColors` class defines static properties such as `ControlColor` and `ControlBrush` (we'll talk about brushes in a minute) that represent the Windows color scheme set by the user.

The structure
will keep these
synchronized for
you.

The A, R, G & B
properties let you specify
a color using integer values
between 0 and 255.

The ScA, ScR, ScG, &
ScB properties let you specify a
color using a `Single` between
0 and 1.

There are more values, and therefore more colors,
available using `ScRGB`. That's called a GAMUT.
But not every display device can display every color
you can define mathematically.



The `ColorContext` property lets
you specify an ICC or ICM Color
Profile. Color Profiles help you keep
colors consistent on different monitors and
printers.

The structure only has a default
constructor. The `From...` methods
let you specify the actual color of the
`Color`. The versions that don't take
an alpha value assume full opacity.

The `Color` structure supports basic
arithmetic operations. You won't need
them often, but they're useful in some
situations.

PUT ON YOUR THINKING HAT



Can you write the code to set a `Color` named `MyColor` to each of the following
colors?

A pure blue at 50% transparency

A purple with R=127, G=68 and B=182, fully opaque

The color named `CadetBlue` in the `Colors` class, which has
R=95, G=158, and B=160, with 75% transparency



HOW'D YOU DO?

A pure blue at 50% transparency

```
myColor.FromArgb(127, 0, 255, 0)
```

A purple with R=127, G=68 and B=182, fully opaque

```
myColor.FromRgb(127, 68, 182)
```

The color named CadetBlue in the Colors class, which has R=95, G=158, and B=160, with 75% transparency

```
myColor.FromArgb(190, 95, 158, 160)
```

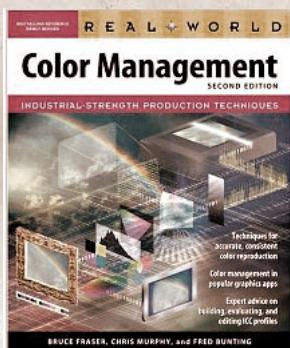
Did you say 75 here? You
need 75% of 255 (which
is actually 191.25. I rounded.)



MOVING ON

You probably learned somewhere that the primary colors are red, yellow and blue, and that you can mix any other color from those. That's only true for things that reflect light, like paint. When you're combining light, the primary colors are red, green and blue, which is why monitor colors are specified that way. That's only one complication when you're working with color. Printers and monitors display color differently, and they can display different colors. Different monitors or printers may display the same RGB values differently. (Monitors are notorious for not representing colors very well, and their colors shift over time.)

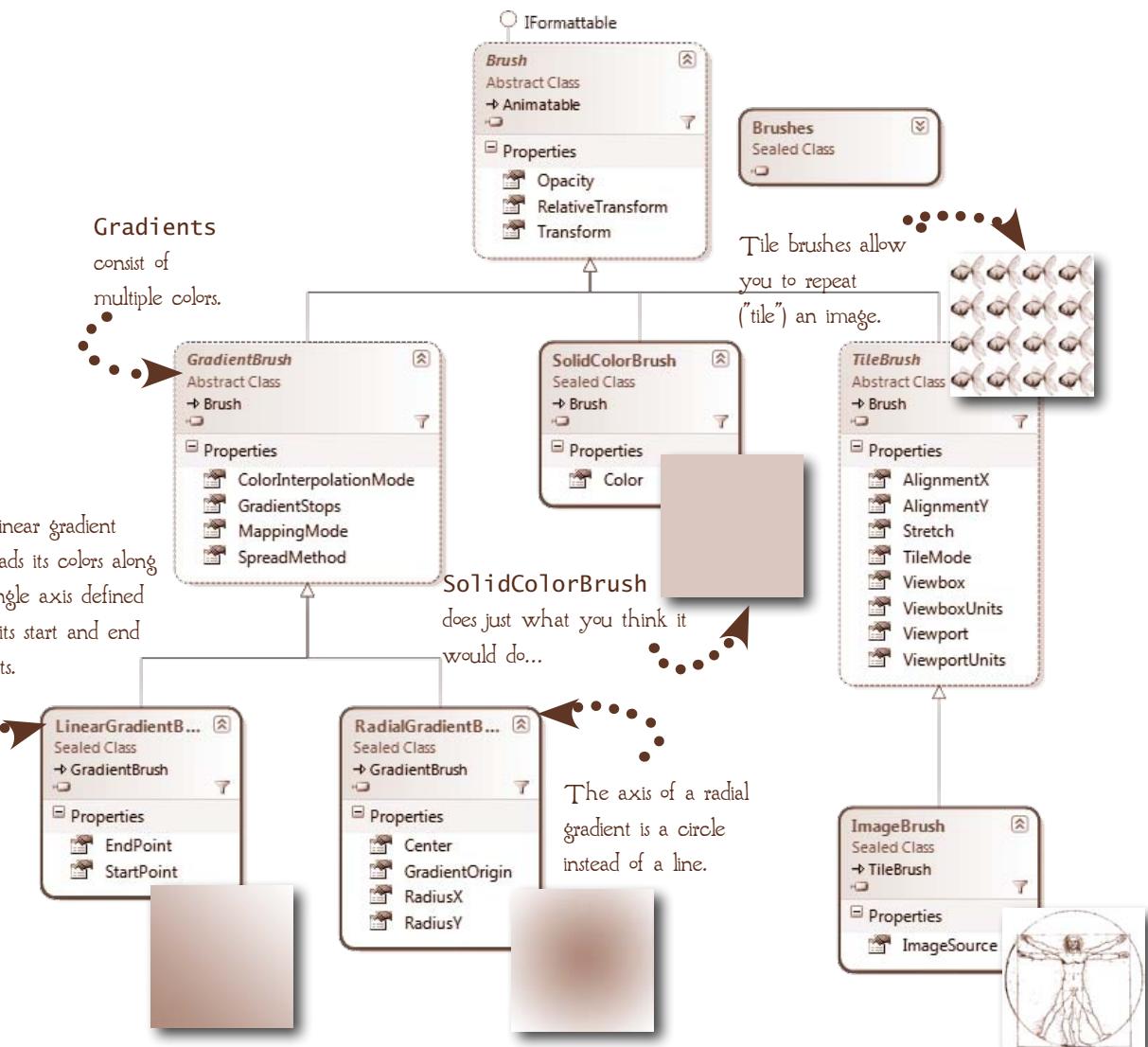
If you're interested in learning more about color theory, color models, gamuts and profiles, *Real World Color Management* by Bruce Fraser is an excellent (if somewhat daunting) resource.





BRUSHES

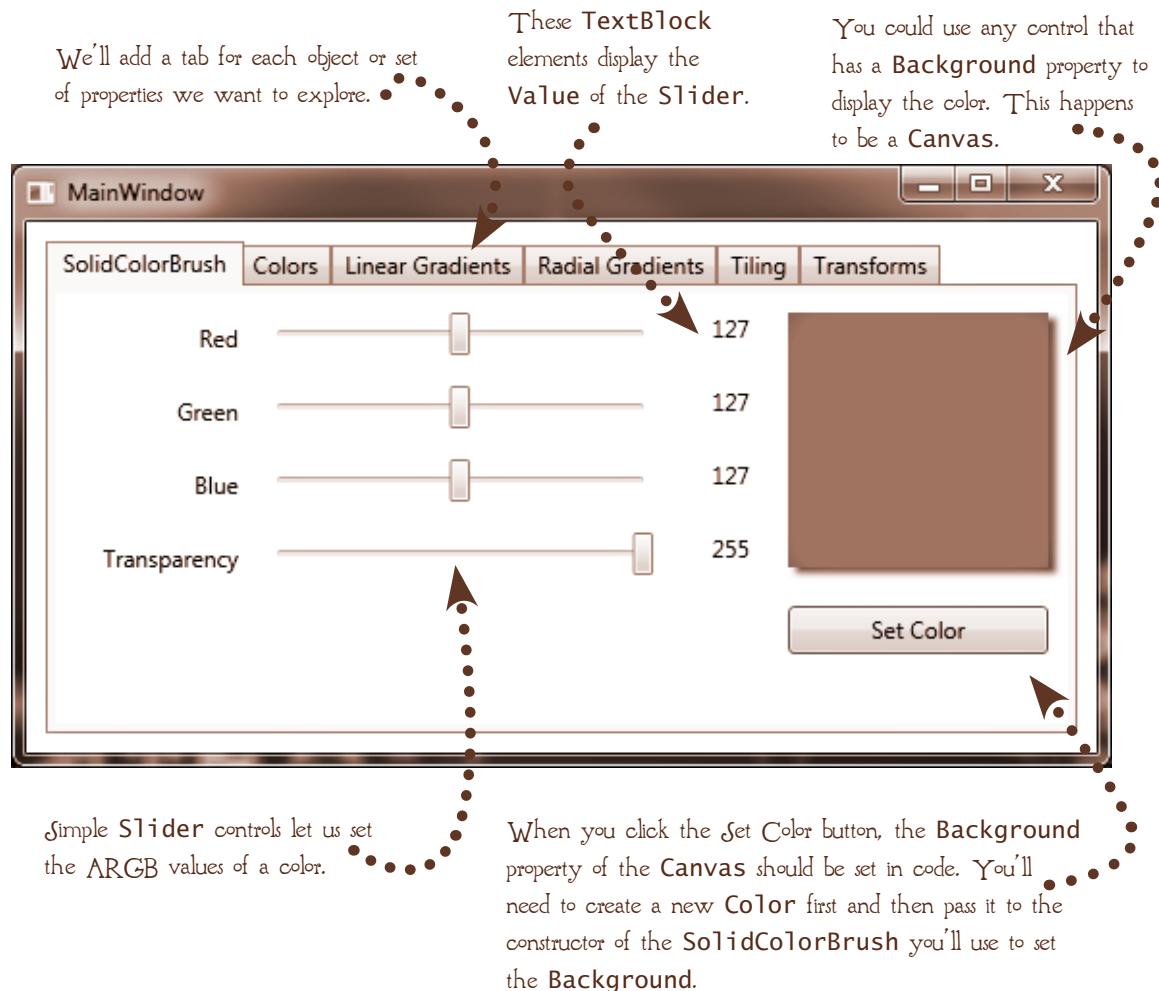
Painters these days don't necessarily use a brush to transfer paint from the palette to the canvas. (Come to think of it, they don't necessarily use a canvas, either.) But in .NET, if you're going to display a color, you need to load it into a Brush. On the other hand, .NET lets you paint with more than a single color. You can load a brush with multiple colors in a gradient, an image, or even video. At last, you can paint with stripes or plaids!



EXPLORING BRUSHES

There are a LOT of UI widgets in the Framework, and they have a LOT of properties that control how they work. While you can change most of the properties in the WPF Designer and all of them in code. When I'm exploring a new widget, I often build a little window that lets me play with the control's properties interactively. Let's do that with the graphic objects we're exploring in this chapter, so you can get a feel for how these "explorers" work. You might find the idea useful when you're working on your own.

Here's the basic structure of our Explorer. We'll start with the `SolidColorBrush` tab.





PUT ON YOUR THINKING HAT

Build the basic structure of the form, and implement the *Set Color* button. This tab uses some controls we haven't worked with before, so here are some hints to save you a lot of time searching MSDN:

- Individual tabs are defined with a `TabItem` element inside a `TabControl` element:

```
<TabControl>
  <TabItem Header="SolidColorBrush">
    ...
  </TabItem>
</TabControl>
```

- I used a 4-column, 5-row `Grid` within the `TabItem` to lay out the other controls, but you can use something else if you'd like.
- The `Slider` controls should be constrained to values between 0 and 255. You can do that by setting the `Minimum` and `Maximum` properties. You'll also want to set the `IsSnapToTick` property so that it always returns whole numbers. Remember to give these controls a name attribute so you can reference them in code.
- It isn't strictly necessary, but you can give the `Slider` controls default values by setting their `Value` property.
- The `Text` properties of the `TextBlock` controls are bound to the `Slider` controls. When you bind a control property, you're telling WPF to get its value from somewhere else, in this case from the `Value` property of the `Slider` to their left. We won't be exploring WPF binding until Chapter 22, so here's the XAML to make that happen:

```
<TextBlock ... Text="{Binding ElementName=RedSlider, Path=Value}" .../>
```

Be careful about . . . where the quotation marks and commas are in the binding syntax. It's easy to get them wrong, and WPF doesn't do a good job of identifying binding problems if you do.

This bit tells WPF that the "somewhere else" you want to get the value from is an Element.

This is the name of the element. I called the `Slider` for the red value "RedValue". You'll need to use whatever name you chose here (and, of course, it will be different for each `TextBlock`).

The `Path` attribute identifies the property that contains the value you want. In this case, you want `Slider.Value`, but you only need to specify the property name.



HOW'D YOU DO?

Here's the basic structure of my version of the window and my (verbose) version of the button event handler. As always, it doesn't matter if your code is a little different, as long as it works.

```
<Window ...>
  <TabControl Margin="10">
    <TabItem Header="SolidColorBrush">
      ...
      <Label Content="Red" Grid.Column="0" Grid.Row="0" .../>
      <Slider Name="RedSlider" Grid.Column="1" Grid.Row="0" Maximum="255"
        IsSnapToTickEnabled="True" Value="127" />
      ...
      <TextBlock Grid.Row="0" Grid.Column="2" HorizontalAlignment="Right"
        <Canvas x:Name="ColorSwatch" Grid.Row="0" Grid.RowSpan="4"
          Grid.Column="3" Margin="10" Background="#FF7F7F7F" />
        <Button x:Name="TestButton" Height="25" Margin="10" Grid.Row="4"
          Grid.Column="3" Click="TestButton_Click">Set Color</Button>
      ....
    </Grid>
  </TabItem>
  ...
</TabControl>
</Window>
```

This is repeated for each slider.

The grid is defined here.

It would have been a little more straightforward to bind the **Background** property to the values returned by the sliders, but because the individual components of a color aren't dependency properties, that's a little tricky. We'll see how to handle situations like this one in Chapter 22.

Did you work out the cast to byte? You could also have used `FromARGB()`, but the code would have been really long because of all these casts.

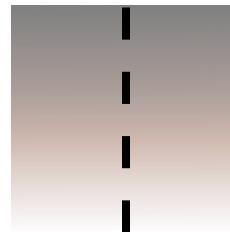
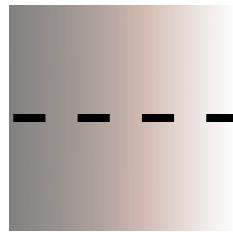
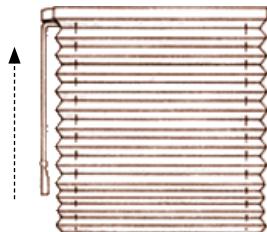
```
Private Sub TestButton_Click(sender As Object, e As RoutedEventArgs)
  newBkgd As Color = New Color()
  newBkgd.A = CType(TransparencySlider.Value, byte)
  newBkgd.R = CType(RedSlider.Value, byte)
  newBkgd.G = CType(GreenSlider.Value, byte)
  newBkgd.B = CType(BlueSlider.Value, byte)

  bkdbrush As SolidColorBrush
  bkgbrush = New SolidColorBrush(newBkgd)
  ColorSwatch.Background = bkdbrush
End Sub
```

GRADIENTS

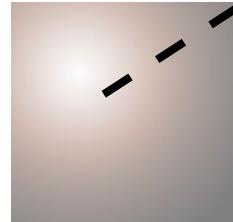
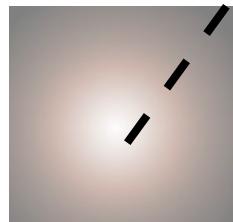
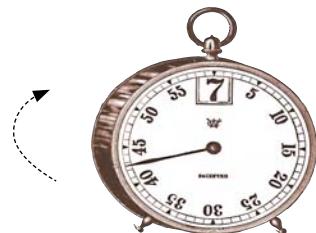
A color gradient (you'll also see it called a color ramp) consists of a set of colors, called **GRADIENT STOPS** (or just **STOPS**), that are given a position along an axis. Colors between the stops are blended. The abstract **GradientBrush** class in WPF defines the **GradientStops** collection and some properties that define how the gradient is created. **GradientBrush** is abstract, but the library defines the two basic kinds of gradients, the **LinearGradientBrush** and the **RadialGradientBrush**. The difference is in how they distribute the colors around their axis.

The colors defined by the stops on a linear gradient are distributed perpendicularly to the gradient axis, like a window shade.



The axis can run in any direction.

The colors defined by the stops in a radial gradient radiate from the center of the ellipse. The axis is like the minute hand of a clock.

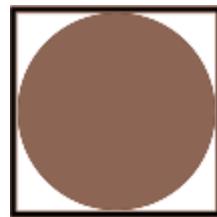


The origin of the axis doesn't have to be in the middle of the circle.

BOUNDING BOXES

The WPF layout engine treats graphic objects as if they were surrounded by an imaginary rectangle: the bounding box. It creates the bounding box based on the size and shape you specify for the object. Gradients are defined relative to that bounding box, not the actual shape.

This is the bounding
box.



SOME TEXT

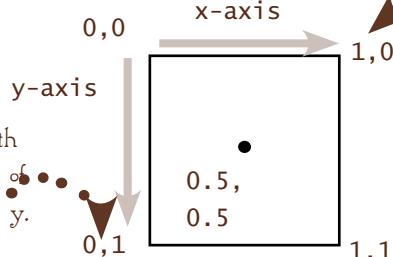


The bounding box is always
rectangular, even though
whatever it contains may not
be.

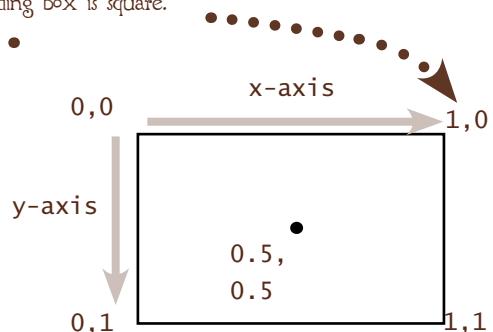
Yes, I know this doesn't make sense if
you're a mathematician. These aren't
Cartesian coordinates; they're mapping
coordinates.



The coordinates within the box range
from 0,0 in the upper left to 1,1 in the
lower right. That's true whether or not
the bounding box is square.

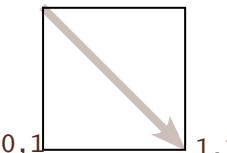


You specify a position with
a **Point**, which consists of
two double values: x and y.

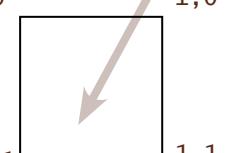


DEFINING A LINEAR GRADIENT

The axis of a `LinearGradientBrush` is defined by its starting and ending points relative to the bounding box. By default, the axis runs from `0,0` to `1,1`, which is top-left to lower right.

```
StartPoint = new Point(0,0);  
0,0  
1,0  
  
0,1  
1,1  
EndPoint = new Point(1,1);
```

But you can specify any starting and ending point you like, including values outside the the bounding box.

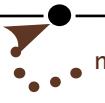
```
StartPoint = new Point(0.75,-0.25);  
0,0  
1,0  
  
0,1  
1,1  
EndPoint = new Point(0.25,0.75);
```

The position of a stop on an axis is defined relative to its origin as a double value. It doesn't matter which direction the axis points or whether the gradient is radial or linear; stops are always expressed this way.

A stop at the origin of the axis would have an offset of `0.0`.



A stop `25%` of the way along the axis would have an offset of `0.25`.



A stop at the end of the axis would have an offset of `1`.



```
new GradientStop(Colors.Green, 0.25);
```



PUT ON YOUR THINKING HAT

Can you declare a `LinearGradientBrush` in XAML and VB? Give it a `StartPoint` of `(0, 0.25)` and an `EndPoint` of `(1, 0.75)`, and add three stops: `Colors.Blue` at Offset `0.25`, `Colors.AliceBlue` at `0.5` and `Colors.White` at `0.95`.



HOW'D YOU DO?

Once you understand what the properties are, creating a `LinearGradientBrush` is pretty straightforward, but did you remember how to add the stops to a collection?

You can use collection syntax

in XAML to declare the gradient stops.



```
<LinearGradientBrush StartPoint="0,.25"
    EndPoint="1,.75">
    <GradientStop Color="Blue" Offset="0.25" />
    <GradientStop Color="AliceBlue" Offset="0.5" />
    <GradientStop Color="White" Offset="0.95" />
</LinearGradientBrush>
```

```
lgb As LinearGradientBrush = New LinearGradientBrush()
```

```
lgb.StartPoint = New Point(0, 0.25)
```

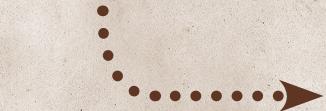
```
lgb.EndPoint = New Point(1, 0.75)
```

```
lgb.GradientStops.Add(new GradientStop(Colors.Blue, 0.25))
```

```
lgb.GradientStops.Add(new GradientStop(Colors.AliceBlue, 0.5))
```

```
lgb.GradientStops.Add(new GradientStop(Colors.White, 0.95))
```

If you apply this gradient to the background of a `Canvas` your gradient will look like this. (Only in color.)



But you need to call the `Add()` method on the `GradientStops` collection in code.



TAKE A BREAK

The next exercise is a big one. Why don't you take a break before you tackle it, and we'll move on to linear gradients when you get back?



PUT ON YOUR THINKING HAT

Let's add linear gradients to our brush explorer.

I chose to set my sliders to the range -2.0 to 2.0 in increments of 0.25. You can do whatever seems reasonable.

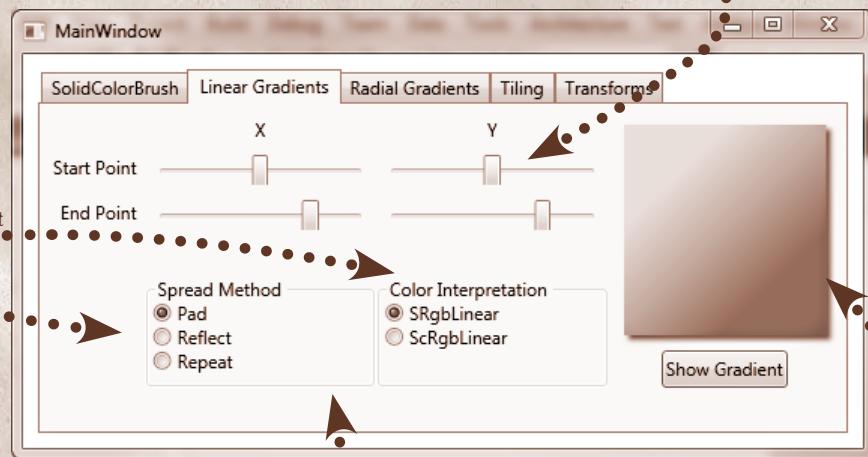
Set the **Minimum**,
Maximum and
TickFrequency
properties to do this.

Once you get your app running, play with these properties. Can you figure out what they do?

We haven't talked about these properties yet. In your code, you'll want to map the **Spread Method** **GroupBox** to the **SpreadMethod** property, which accepts a member of the **GradientSpreadMethod** enumeration, and the **Color Interpretation** **GroupBox** to the **ColorInterpretationMode** property, which accepts a member of the **ColorInterpretationMode** enumeration.

Before you peek at the solution on the next page, see if you can work out how to set these properties in your code. (Hint: You'll need to be explicit about `IsChecked = true` in your `if` statement.)

Instead of binding a **TextBox** to the **Slider**, you can turn on the **AutoToolTipPlacement** and **AutoToolTipPrecision** so that the current value will be displayed whenever you click the thumb. (That's the little thing you slide.)



Go ahead and define the basic gradient in the XAML. I picked three stops: **Bisque** at 0.25, **DarkSalmon** at 0.5 and **Chocolate** at 0.75—but you can set it up any way you like.

Remember to name your brush so you can change its properties in the event handler for the button. I called mine **LinearBrush**.

```
<Canvas ...>
<Canvas.Background>
<LinearGradientBrush
  x:Name="LinearBrush" ... />
...
```



HOW'D YOU DO?

You probably know by now that it doesn't matter if you chose a different way to lay out your window, right?

```
<Grid>
  <Grid.RowDefinitions>...</Grid.RowDefinitions>
  <Grid.ColumnDefinitions>...</Grid.ColumnDefinitions>
  <Label Grid.Column="1" HorizontalAlignment="Center">X</Label>
  <Label Grid.Column="2" HorizontalAlignment="Center">Y</Label>
  <Label Grid.Row="1" HorizontalAlignment="Right">Start Point</Label>
  <Slider x:Name="StartX" Grid.Column="1" Grid.Row="1" Margin="5"
    IsSnapToTickEnabled="True" Maximum="2" Minimum="-2"
    TickFrequency="0.25" AutoToolTipPlacement="TopLeft"
    AutoToolTipPrecision="2" />
  ...
  <GroupBox Grid.Column="1" Grid.Row="4" Grid.RowSpan="3"
    Header="Spread Method">
    <StackPanel>
      <RadioButton x:Name="PadMethod" IsChecked="True">Pad</RadioButton>
      <RadioButton x:Name="ReflectMethod">Reflect</RadioButton>
      <RadioButton x:Name="RepeatMethod">Repeat</RadioButton>
    </StackPanel>
  </GroupBox>
  <GroupBox Grid.Column="2" Grid.Row="4" Grid.RowSpan="3"
    Header="Color Interpretation">
    <StackPanel>
      <RadioButton x:Name="SRgbLinear" IsChecked="True">SRgbLinear</RadioButton>
      <RadioButton x:Name="ScRgbLinear">ScRgbLinear</RadioButton>
    </StackPanel>
  </GroupBox>

  <Canvas Name="LinearSwatch" Grid.Row="0" Grid.Column="3"
    Grid.RowSpan="6" Margin="10" >
    <Canvas.Background>
      <LinearGradientBrush x:Name="LinearBrush" >
        <GradientStop Color="Bisque" Offset="0.25" />
        <GradientStop Color="DarkSalmon" Offset="0.5" />
        <GradientStop Color="Chocolate" Offset="0.75" />
      </LinearGradientBrush>
    </Canvas.Background>
  </Canvas>
  <Button Name="ShowGradient" Grid.Row="6" Grid.Column="3"
    Click="ShowGradient_Click">Show Gradient</Button>
</Grid>
```

The Label and Slider elements are repeated for the other three sliders.

And here's my version of the event handler for the button. The theory here is fine, but there are some fiddly syntax issues. Did you sort them out?

You needed to remember to specify the `Value` property here, but we did that last time, too.

```
Private Sub ShowGradient_Click(sender As Object, e As RoutedEventArgs)
    LinearBrush.StartPoint = New Point(StartX.Value, StartY.Value)
    LinearBrush.EndPoint = New Point(EndX.Value, EndY.Value)

    If PadMethod.IsChecked = True
        LinearBrush.SpreadMethod = GradientSpreadMethod.Pad
    ElseIf ReflectMethod.IsChecked = True
        LinearBrush.SpreadMethod = GradientSpreadMethodReflect
    Else
        LinearBrush.SpreadMethod = GradientSpreadMethod.Repeat
    End If

    If SRgbLinear.IsChecked = True
        LinearBrush.ColorInterpolationMode =
            ColorInterpolationMode.SRgbLinearInterpolation;
    Else
        LinearBrush.ColorInterpolationMode =
            ColorInterpolationMode.ScRgbLinearInterpolation;
    End If
End Sub
```

NOW, ABOUT THOSE PROPERTIES...

The `ColorInterpolationMode` determines whether the gradient uses the sRGB or ScRGB color space for calculating the blending between stops. Because ScRGB supports fractional values, you may see a slightly smoother gradient with this version, but it very much depends on the colors you choose, the distance between stops, and the quality and resolution of your monitor.

The `SpreadMethod` property determines what happens when the gradient reaches the end of the gradient axis. The `Pad()` method, which is the default, simply extends the color of the last value to the edge of the bounding box. `Reflect()` mirrors the gradient stops (this method is particularly useful for depicting 3-dimensional shapes), and the `Repeat()` method starts over from the first stop.

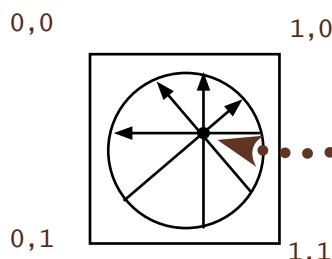


This is the brute force way of mapping a set of radio buttons to an enumeration. There are more elegant ways using data binding, but this works and it's clear.

RADIAL GRADIENTS

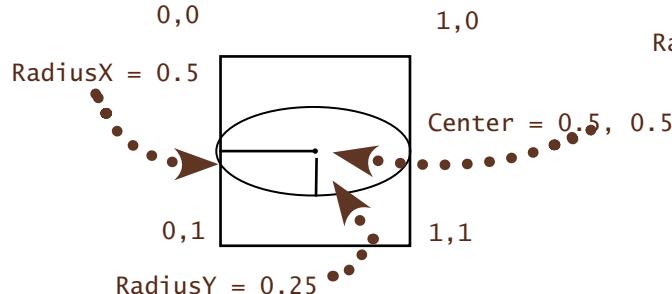
Like the `LinearGradientBrush`, the `RadialGradientBrush` blends colors along an axis that is defined relative to a bounding box. Gradient stops are defined in the same way for both brushes, but because the gradient is calculated as an ellipse, rather than a line, the axis is defined differently, and the class exposes additional properties to define the shape and position of the ellipse itself.

```
GradientOrigin = new Point(0.75, 0.25);
```



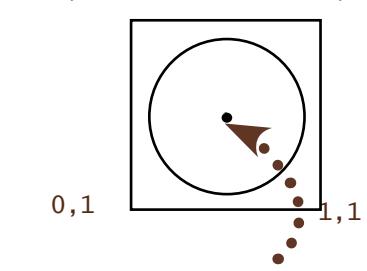
For a `RadialGradientBrush`, the axis is defined only by its origin, because it radiates in every direction.

The `RadiusX` and `RadiusY` properties define the size of the ellipse relative to the corresponding axis of the bounding box. Thus, a `RadiusX` of 0.5 will be half the length of the bounding box.



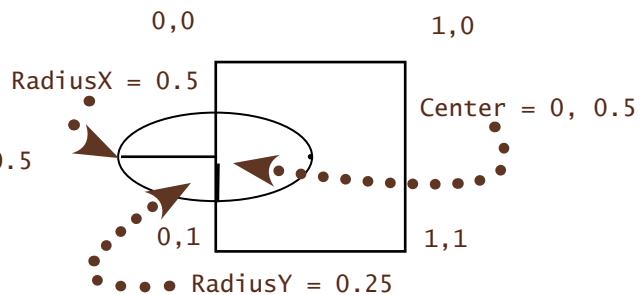
The center point of the ellipse, which need not be the center point of the bounding box, is also defined relative to the bounding box.

```
Center = new Point(0.5, 0.5);
```



```
Center = new Point(0.5, 0.5);
```

But the radius extends from the center of the ellipse, not from the center of the axis, so the ellipse can extend past the edges of the bounding box. (The parts that extend won't be displayed.)

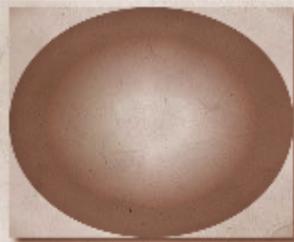




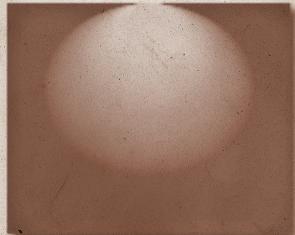
PUT ON YOUR THINKING HAT

Building the radial gradient tab for our brush explorer doesn't require any new skills. Stop for a second and think about that—when did you start working with this book? And now building an application that allows you to interactively define a radial gradient is old hat. Yea, you.

The XAML and VB code for this tab is really just a minor variation of the last exercise, so I'm not even going to show it to you. What I want you to do instead is use the radial gradient explorer tab to create some specific effects.



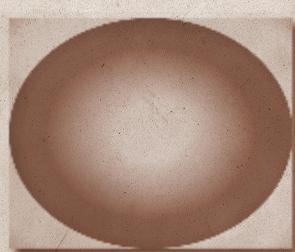
HOW'D YOU DO?



Center: (0.5, 0.5)
RadiusX: 0.5
RadiusY: 0.5
Origin: (0.0, 0.0)
SpreadMethod: Pad



Center: (0.5, 0.5)
RadiusX: 0.5
RadiusY: 0.5
Origin: (0.5, 0.0)
SpreadMethod: Pad



Center: (0.5, 0.5)
RadiusX: 0.5
RadiusY: 0.5
Origin: (0.5, 0.5)
SpreadMethod: Repeat

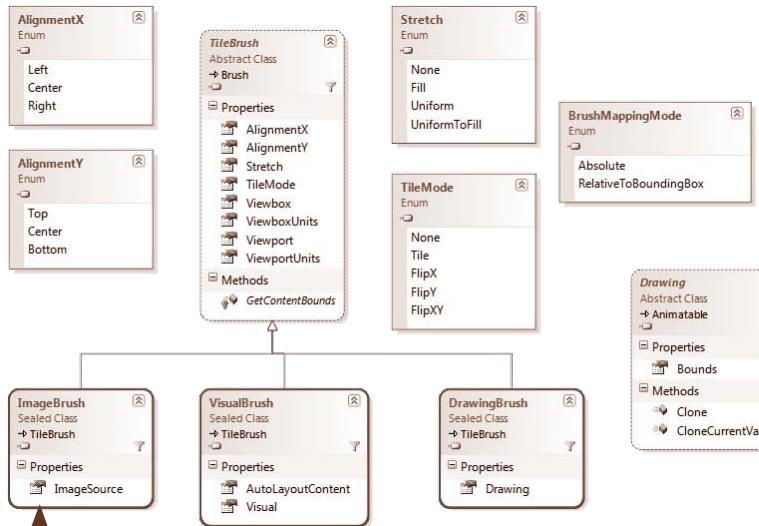
TAKE A BREAK



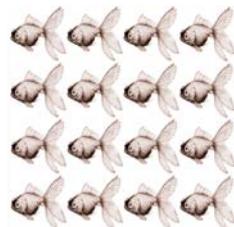
Gradients are another one of those WPF things that only get complicated if you think about them too much, aren't they? Well, why don't you take a break and not think about them at all for a while?

TILE BRUSHES

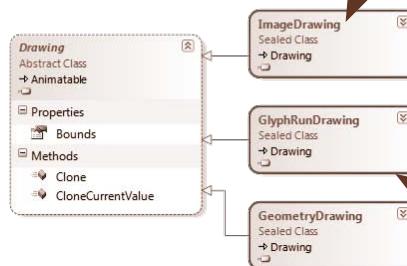
Our last, and very powerful, brush family descend from `TileBrush`, of which there are three: `ImageBrush`, `DrawingBrush` and `VisualBrush`. The tile brushes give you the ability to fit their content to its container, and will display it either as a single time or multiple times as a tile that fills the space. The tile brushes all work the same way, so we'll be concentrating on the `ImageBrush`, which is the one you're most likely to need in a standard application.



An `ImageBrush` is most often used to display a bitmapped image like a jpeg or png, but it can also display a `Geometry` or a 3-dimensional image built using the Direct3D API.



All of the tile brushes let you repeat their content across and down an area.



An `ImageDrawing` isn't as powerful as an `ImageBrush`, but it's slightly more efficient if you don't need the additional capabilities.

In WPF, a `Geometry` is a shape described by lines segments, arcs and points.

A GLYPH RUN is text.

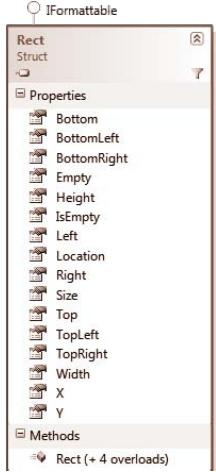
MAKE A NOTE

You'll notice some overlap in the kind of content that can be displayed by the `ImageBrush` and `DrawingBrush`. As a general rule, the `DrawingBrush` is less powerful, but also slightly more efficient, so although we'll only be exploring the `ImageBrush`, you should keep `DrawingBrush` in mind for situations when the `ImageBrush` seems to be slowing your application down.

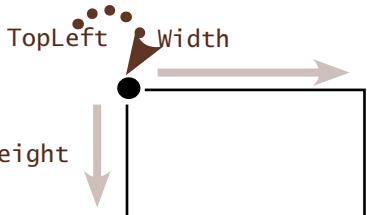
MORE IMAGINARY BOXES...

You've seen how WPF creates a bounding box around an object when it's calculating gradients. When you're working with tile brushes, you have two more imaginary boxes to consider: the Viewbox and the Viewport. But while the bounding box is maintained by WPF, you'll need to explicitly define the Viewbox and Viewport if you need to use them. (You often won't.)

Both of the imaginary boxes used by the `TileBrush` are defined as instances of the `Rect` structure, so let's look at that first:



A `Rect` is defined in terms of its `TopLeft` corner, a `Point`, its `Width` and its `Height` (both `Double`).



The `Width` and `Height` properties are represented as an instance of the `Size` structure:

`New Size(<width>, <height>)`

`TopLeft` defaults to `(0,0)`.

`New Rect(<size>)`

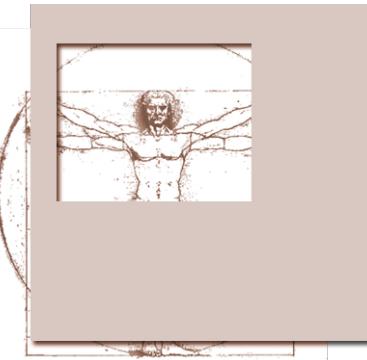
`New Rect(<topLeftPoint>, <bottomRightPoint>)`

`New Rect(<topLeftPoint>, <size>)`

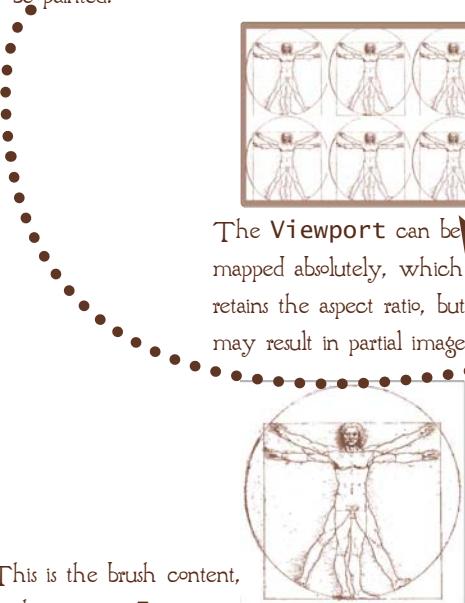
`New Rect(<topLeftPoint>, <vectorOffset>)`

`New Rect(<topPoint>, <leftPoint>, <bottomPoint>, <rightPoint>)`

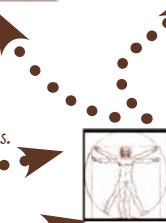
A **Viewbox** defines the part of the image to be displayed. You can use it to select a subset of the image. You won't use it all that often because unless you're using an image in more than one place, you'll resize it before loading it into memory.



A **Viewport** is used to display multiple copies of the brush content. It maps the content to the bounding box of the object to be painted.



The **Viewport** can be mapped absolutely, which retains the aspect ratio, but may result in partial images.



Or the **Viewport** can be mapped relative to the bounding box, which ensures that only full tiles will be displayed, but they might be distorted.

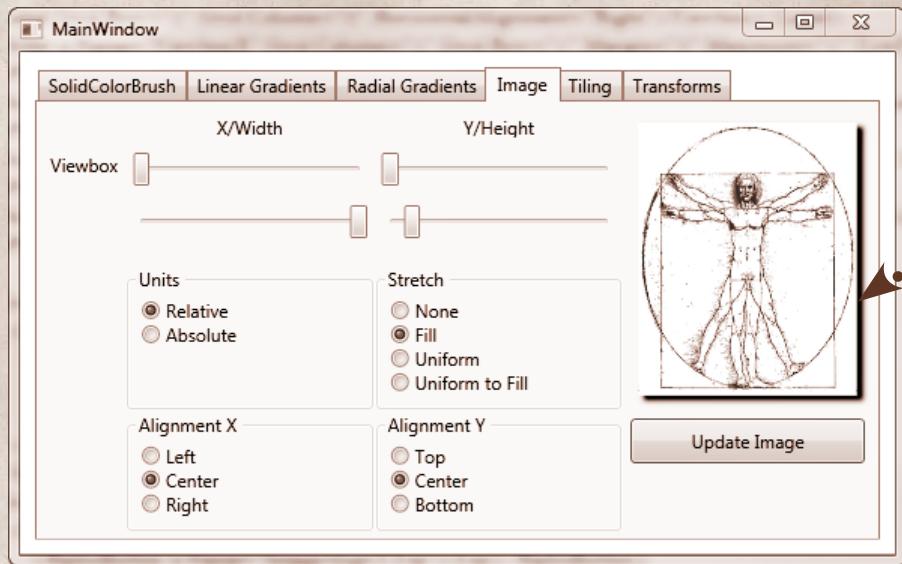
The **Viewport Rect** sits between the content and the bounding box.

This is the brush content, in this case an **Image**.

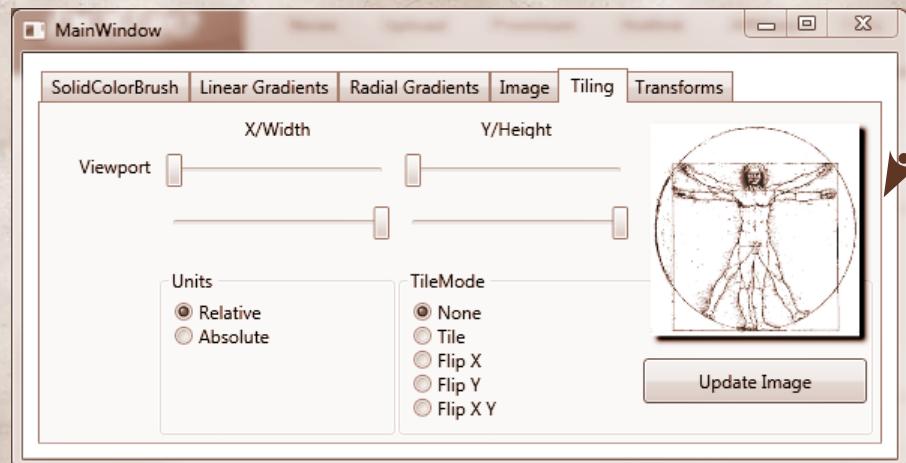


PUT ON YOUR THINKING HAT

Time to extend the brush explorer. You'll need two tabs; my versions are below. This time, I want you to use the explorer to figure out what each of the properties does. (It's okay to use MSDN for some guidance, but try to figure out what they do by experimentation.)



To get an image to play with, add it to your project, and then set the **ImageSource** property of the brush its name.



THE IMAGEBRUSH PROPERTIES

When the `Stretch` property is set to `Fill`, what does resizing the window do to the image?

How is this behavior different when the `Stretch` property is set to `Uniform`? My example image is square, so there's no difference between `Uniform` and `UniformToFill`. What happens when you use a rectangular image?

What does the `Stretch` property need to be in order to see the effects of the `AlignmentX` and `AlignmentY` properties?

What `Viewbox Width` and `Height` settings show just the upper left corner of the image?

THE TILEBRUSH PROPERTIES

What happens if you set the `TileMode` to `Tile` without changing the `Viewport`?

What happens if you set the `TileMode` to `None` with the `Viewport` set to its default values?

What does `FlipX` do? `FlipY`? `FlipXY`?



HOW'D YOU DO?

THE IMAGEBRUSH PROPERTIES

When the `Stretch` property is set to `Fill`, what does resizing the window do to the image?

Distorts to fill the space

How is this behavior different when the `Stretch` property is set to `Uniform`? My example image is square, so there's no difference between `Uniform` and `UniformToFill`. What happens when you use a rectangular image?

Resizes, but keeps aspect ratios.

What does the `Stretch` property need to be in order to see the effects of the `AlignmentX` and `AlignmentY` properties?

None

What `Viewbox Width` and `Height` settings show just the upper-left corner of the image?

0.5 for both

THE TILEBRUSH PROPERTIES

What happens if you set the `TileMode` to `Tile` without changing the `Viewport`?

Nothing

What happens if you set the `TileMode` to `None` with the `Viewport` set to its default values?

Only one small image displays

What does `FlipX` do? `FlipY`? `FlipXY`?

Flips every other tile



TAKE A BREAK

That's it for brushes. Why don't you take a break before you finish the Review and we move on to Pens (which are much simpler)?

REVIEW

When working with a `RadialGradientBrush`, you can add a gradient stop at 0.25 and set the `RadiusX` property to 0.25. Both values mean “one fourth of the way”, but of what? (It’s different in each case.)

What do the letters ARGB stand for?

What value should you give the `ViewportUnits` property of a `TileBrush` to ensure that the entire content is always displayed?

If a bounding box is twice as wide as it is long, what are the x and y coordinates of the lower-left corner?

What property do you set to control what colors are displayed if the gradient axis doesn’t reach the edge of the bounding box?

What’s the difference between ScRGB and SRGB?

What’s the shape of the bounding box around a circular object?

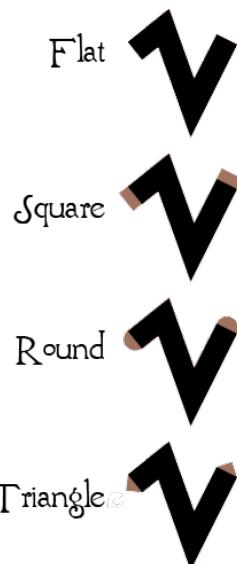
Our explorer tabs would benefit by a reset function that restores all the values on the tab to their default values. Add a Reset button to one or more of the tabs.



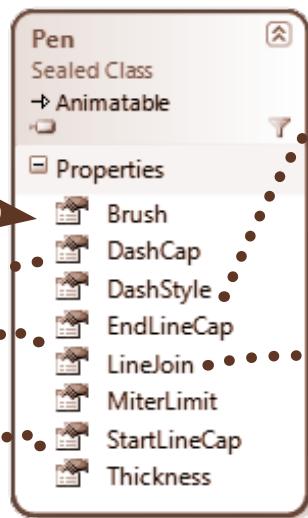
PENS

The WPF Pen is used to draw lines. I'm sure you'll be relieved to know that it's far less complex than the Brush. There's only one class (although you could derive your own if you need to) and a few properties.

You can use any kind of brush to draw the Pen. We'll use a gradient in the next exercise, but you can use solid brushes, images, whatever suits your purpose.



DashCap, StartLineCap and EndLineCap control the shape at the end of a line or dash. Unfortunately, you can't define your own cap styles to display, for example, arrowheads. You have to choose one of the styles defined in the PenLineCap enumeration.

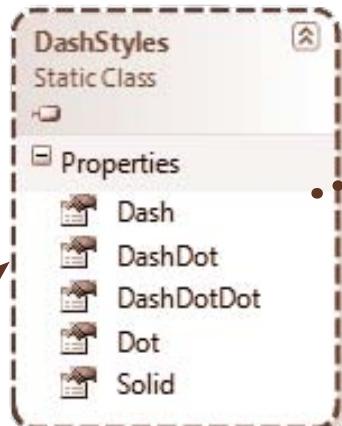


The **LineJoin** property controls what happens when a **Pen** turns a corner. Again, you can't define your own; you must use a member of the **PenLineJoin** enumeration.



Like the Colors class used for a SolidColorBrush, the Framework defines a set of useful dash styles you can use.

You can define your own dash styles. The Dashes collection of the DashStyle class contains a collection of Double values that represent a percentage of the Thickness of the Pen.



Solid (Default)

the lazy brown dog.

Dot (0,2)

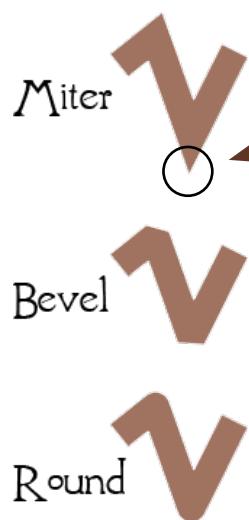
Dash (2,2)

the lazy brown dog., the lazy brown dog.

DashDotDot (2,2,0,2,0,2)

DashDot (2,2,0,2)

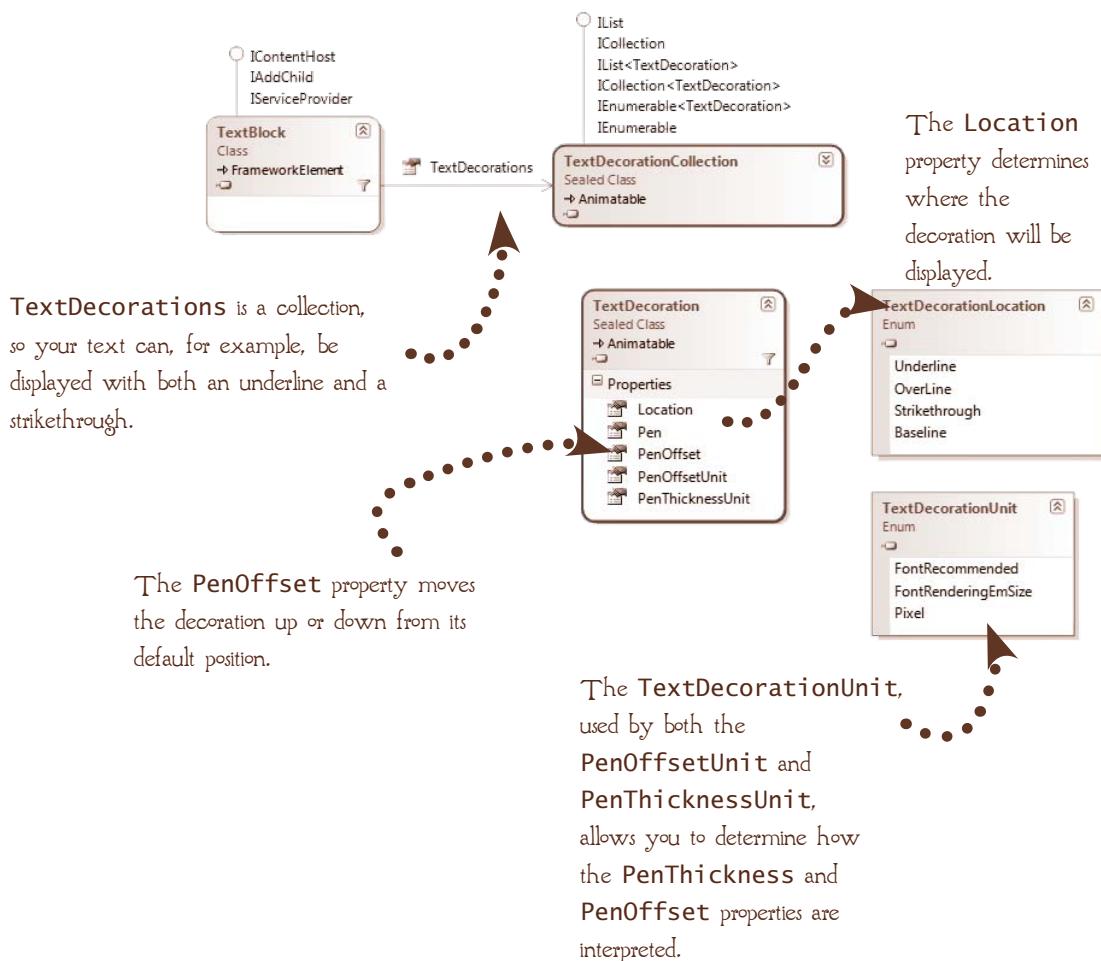
the lazy brown dog., the lazy brown dog.



The MiterLimit property controls how far the miter can extend. It's expressed as a percentage of the width of the Pen.

TEXT DECORATIONS

In WPF, pens are primarily used when you're defining graphic shapes, but you'll also use them when you want to "decorate" text with an underline or strikethrough using the `TextDecoration` class that has a `Pen` as a property. The `TextBlock` is one control that exposes a `TextDecorations` collection, and we'll use it to explore the properties of a `Pen`.





PUT ON YOUR THINKING HAT

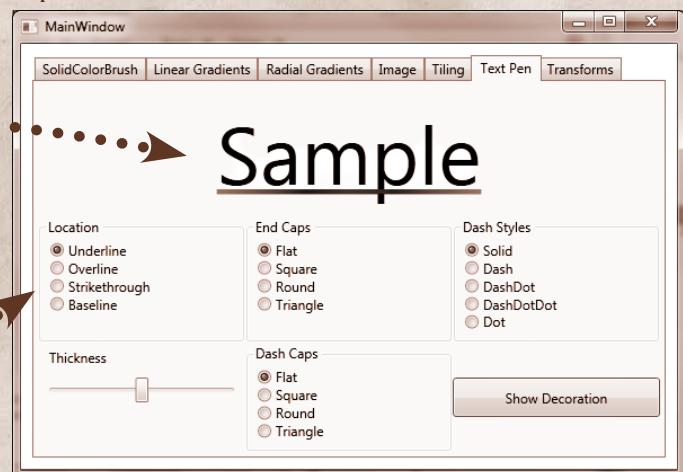
Back to our explorer. Can you duplicate (or improve on) my version?

This is a **TextBlock**. The **LinearGradientBrush** is declared in the **XAML** and displayed as an underline, because that's the default location for a **TextDecoration**. The equivalent VB to reference the pen would be:

```
TextBlock.TextDecorations(0).TextDecoration.Pen.Brush
```

The **XAML** syntax is in the solution on the next page, but see if you can work out how to do it without help.

In the code, remember that all of the controls except **Location** are properties of the **TextDecoration.Pen**, not the decoration itself.



Once you get the sample working, can you identify the location of each of these lines?





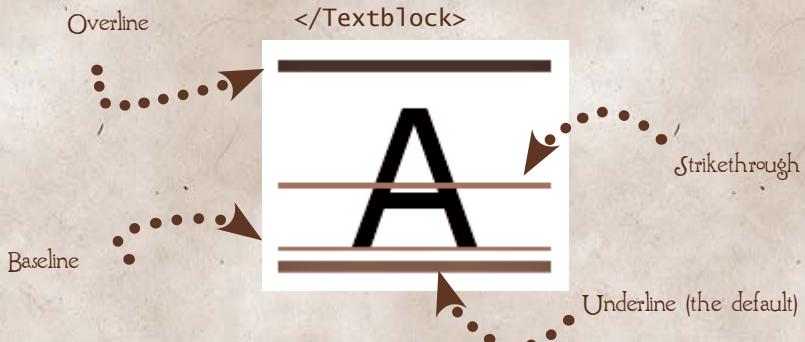
HOW'D YOU DO?

This one was a little trickier than our other examples because you were dealing with nested objects. Here's the XAML to set that initial gradient:

```
If endSquare.IsChecked = True  
    Decoration.Pen.StartLineCap = _  
        PenLineCap.Square  
    Decoration.Pen.EndLineCap = _  
        PenLineCap.Square  
End If
```

And here's a snippet to show you how to set up changing the pen's properties in the **Click** event handler of the **Button**.

```
<Textblock ...>  
    <Textblock.TextDecorations>  
        <TextDecoration x:Name="Decoration">  
            <TextDecoration.Pen>  
                <Pen>  
                    <Pen.Brush>  
                        <LinearGradientBrush>  
                            <GradientStop ... />  
                            <GradientStop ... />  
                            <GradientStop ... />  
                        </LinearGradientBrush>  
                    </Pen.Brush>  
                </Pen>  
            </TextDecoration.Pen>  
        </TextDecoration>  
    </Textblock.TextDecorations>  
</Textblock>
```



TAKE A BREAK

That was a tricky exercise and it probably took you some time, so why don't you take a break before we move on to exploring WPF text in more detail?



WPF TYPOGRAPHY

Just about anything you want to do typographically is possible in WPF. Swash capitals, ligatures, alternate glyphs, whatever you need, you can do. But this isn't a typography book; it's a book about VB programming. Don't worry if you don't know what any of those things are; we're only going to explore the basic text characteristics that are exposed by the widgets you're likely to be using. And the good news is that WPF makes controlling these simple things simple.

Most of the complexity of typography in WPF is that you have two sets of concepts to juggle. First is the properties of the text itself, and second is the widgets that best suit your purpose. We'll take them one at a time.

WHAT DOES IT LOOK LIKE?

Normal or *Italic*

Times New Roman or 

Condensed or Expanded

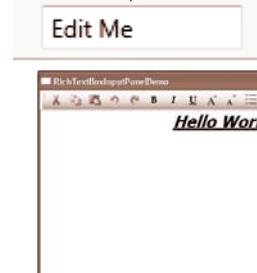
Medium or **Bold**

Or Some of Each??

This is a **FlowDocumentReader**, one of the widgets that display richly formatted text.

HOW IS IT BEING USED?

You've already used the **Label** and **TextBox**.

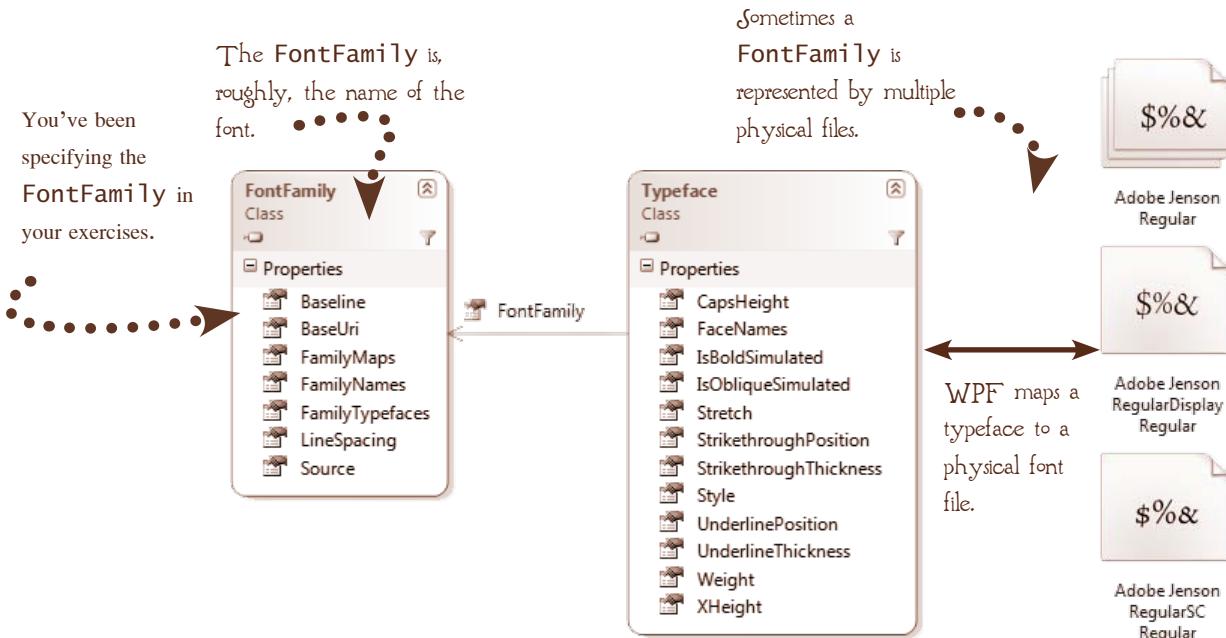

Label

This is a **RichTextBox**. It's like a baby Microsoft Word.



FONTS, FAMILIES AND FACES

Fonts in WPF are represented by two classes, the `FontFamily` that represents the basic font and the `Typeface` that represents a particular font family with a specific set of characteristics.



WORDS FOR THE WISE

A glyph is the shape of a letter or punctuation mark. What's important here is that the glyph is the shape, not the letter. Most alphabets, including the one we use in English, use multiple glyphs to represent the same letter. All of these glyphs represent the letter "a", but the individual marks are quite different.

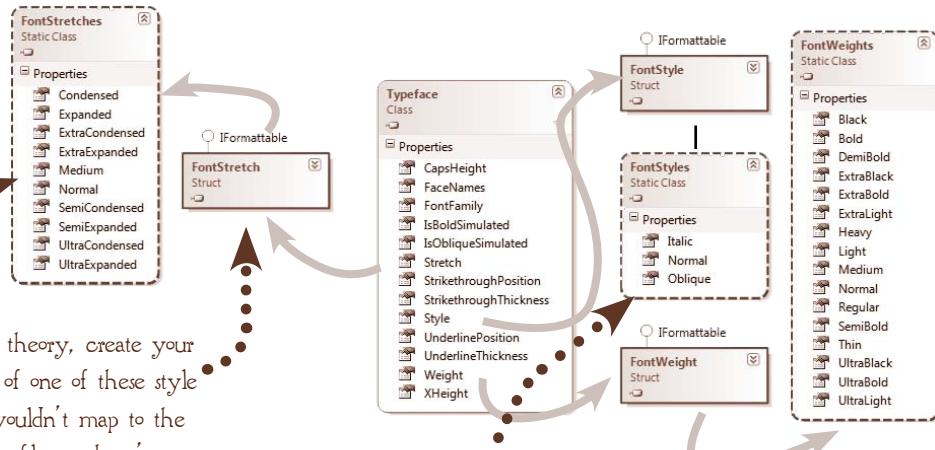
Why do you care? Because a font is a set of glyphs.

Diagrams on the right show various forms of the letter 'a' (A, a, à, á, à, à, à, à, à, à, à, à).

BASIC TEXT CHARACTERISTICS

Three of the four basic characteristics of text are represented in the `Typeface` class. The fourth, the `FontSize`, isn't, because any typeface can be displayed at any size.

You specify any of the characteristics using the properties of static classes defined by the Framework Library.



You could, in theory, create your own definition of one of these style types, but it wouldn't map to the layout of a font file, so there's really no point.

Most people don't make a distinction between oblique and italic, but they are quite different: The italic version of a font uses a different glyph.

a a a

Normal Oblique Italic

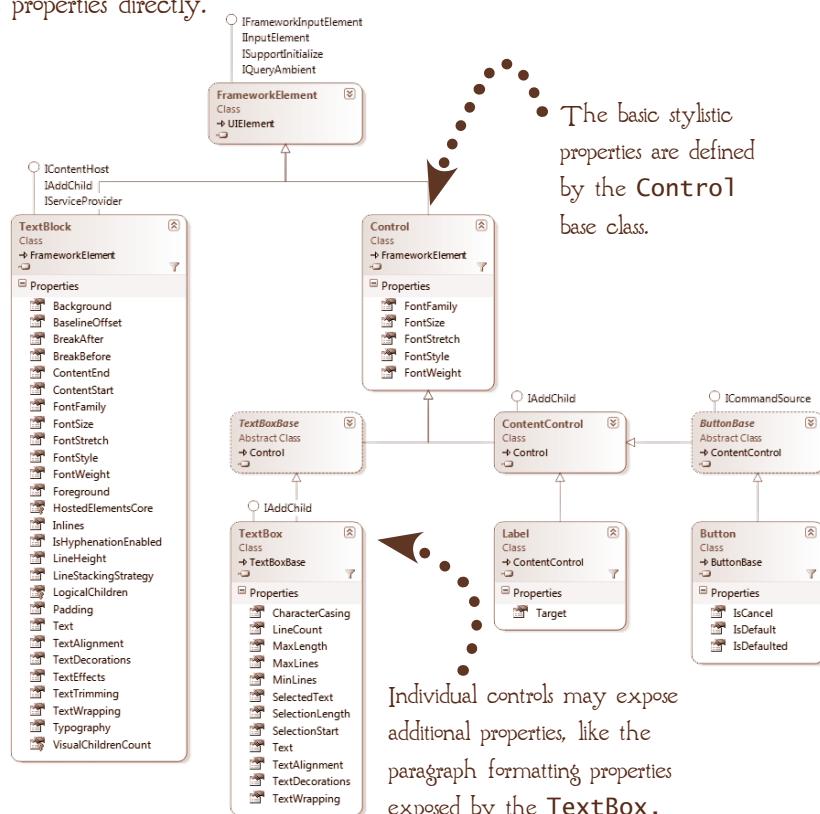
PUT ON YOUR THINKING HAT

Not all fonts have every version of every characteristic. Very few define all the possible weights, for example, and a font that defines an italic version usually doesn't include an oblique. What do you think WPF does if you specify a characteristic that isn't defined?

TEXT CONTROLS

Now, an odd thing about working with text in WPF is that you don't specify the Typeface directly. Instead, the stylistic properties are exposed on the widget you're using to display the text. The basic properties are exposed by the `Control` class, from which most of the UI widgets descend. Widgets like the `TextBlock` that we used to play with text decorations, which descend directly from `FrameworkElement`, expose the properties directly.

Some controls, most notably the `TextBlock`, expose a LOT of properties to give you fine control over how your text appears.



PUT ON YOUR THINKING HAT

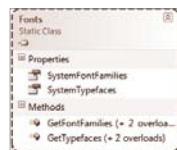
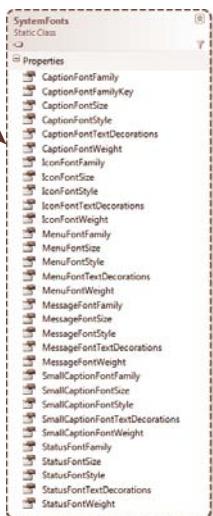
If you specify a typeface that doesn't physically exist in the font file, WPF will imitate it for you. This isn't always as successful as you might hope, but you'll know immediately if what you've chosen is ugly.



STATIC FONT CLASSES

It's considered polite for standard business applications to use the typefaces the user has chosen in the Windows control panel, but prior to WPF, that was a pretty tricky exercise. It was difficult enough just figuring out what fonts the user had installed. But the FCL makes it easy through two static classes: `System.Windows.SystemFonts`, which exposes the fonts the user has selected in Windows, and `System.Windows.Media.Fonts`, which, in addition to letting you load fonts from any location, also exposes static collection properties that tell you all the font families and typefaces in the default Windows font directory.

There's a set of ••••• properties for each type of display. I've only shown the full set for the first ones.



This is actually the list that Visual Studio pops up in Intellisense.

You can use these methods to query any location.

Unfortunately, to use these two classes effectively, you need to understand resources and data-binding, and we haven't looked at those yet. So for now, just remember that they're there when you need them, and we'll come back to them in the next few chapters.

MAKE A NOTE

WPF provides a mechanism for embedding fonts with your application. But just because it's technically possible, it doesn't follow that it's legal. Remember, the typographers who design those fonts and the foundries who distribute them have as much right to earn a living from their labors as you do. While there are many "free to use" fonts available on the Internet, not all of them are "free to distribute", and it's not always easy to track down the actual terms of use. The whole issue is so entirely complicated that I strongly recommend you stay away from font embedding except in extraordinary circumstances, and only when you're absolutely sure that you have the right to do so. (I consider "absolutely sure" as either a clear statement that the font has been placed in the public domain or a release, in writing, from the copyright holder. But that's just me.)

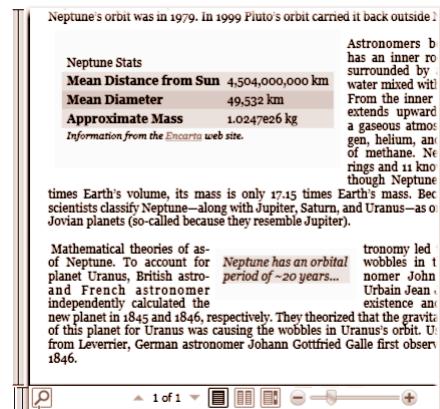
WPF DOCUMENTS

The stylistic properties are very easy to use, but there's a catch: Whatever values you specify will be applied to all of the text. So what do you do if you need to display mixed text? Only a single word might need to be bold, for example, or you might want to display only a single character in a different font. And what about embedding an image in some text? Most UI platforms don't let you do any of that. WPF does, through Documents, of which there are two kinds.

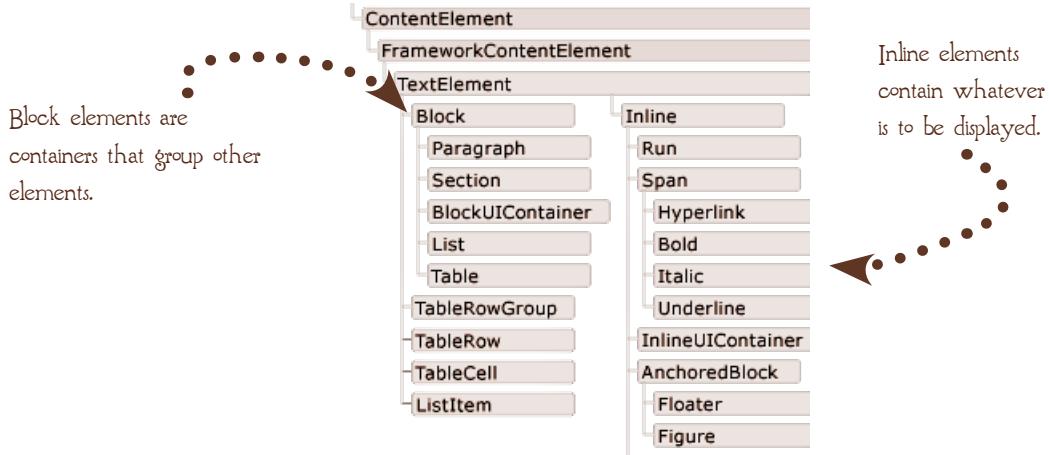
FIXED DOCUMENTS have, not surprisingly, a fixed layout. They're typically used to generate reports, like this catalog, and because that's a pretty specialized task, we won't be discussing them here.



FLOW DOCUMENTS, on the other hand, are used whenever you want to display so-called "rich content", including text with varying formats, lists, tables, or text mixed with other types of content such as images.



Although you can create very complex flow documents, the classes themselves are very simple. They're divided into two major groups, **Blocks** and **Inline**:

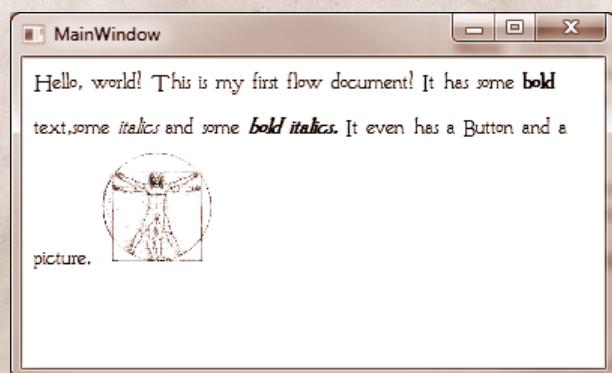
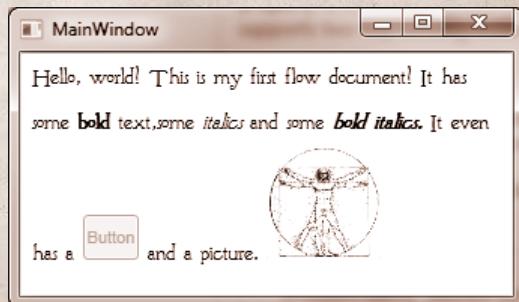




PUT ON YOUR THINKING HAT

To get a sense of just how simple flow documents are to use, can you pick the correct output from this snippet?

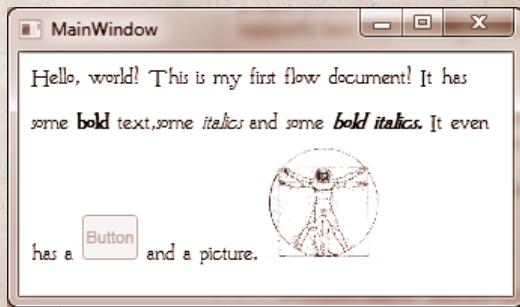
```
<FlowDocument>
    <Paragraph FontFamily="Brandywine" FontSize="18">
        Hello, world! This is my first flow document! It has some
        <Bold>bold </Bold> text, some <Italic>italics</Italic> and some
        <Bold><Italic>bold italics.</Italic></Bold>
        It even has a
        <InlineUIContainer>
            <Button Height="30">Button</Button>
        </InlineUIContainer>
        and a picture.
        <InlineUIContainer>
            <Image Source="leonardo.jpg" Height="75"></Image>
        </InlineUIContainer>
    </Paragraph>
</FlowDocument>
```



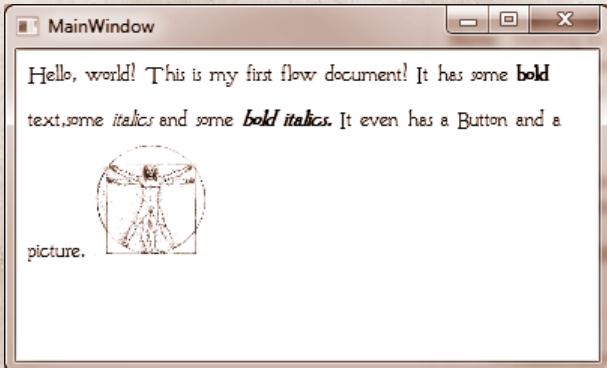


HOW'D YOU DO?

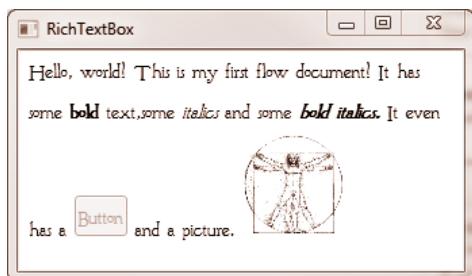
```
<FlowDocument>
  <Paragraph FontFamily="Brandywine" FontSize="18">
    Hello, world! This is my first flow document! It has some
    <Bold>bold </Bold> text, some <Italic>italics</Italic> and some
    <Bold><Italic>bold italics.</Italic></Bold>
    It even has a
    <InlineUIContainer>
      <Button Height="30">Button</Button>
    </InlineUIContainer>
    and a picture.
    <InlineUIContainer>
      <Image Source="leonardo.jpg" Height="75"></Image>
    </InlineUIContainer>
  </Paragraph>
</FlowDocument>
```



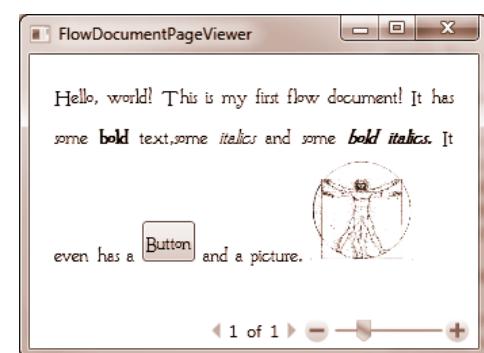
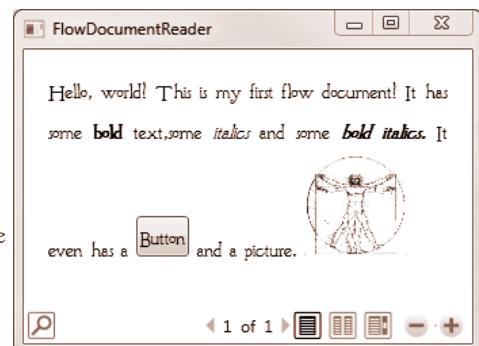
It's this one, of course. The
<InlineUIContainer> tag
displays the <Button> as a button.



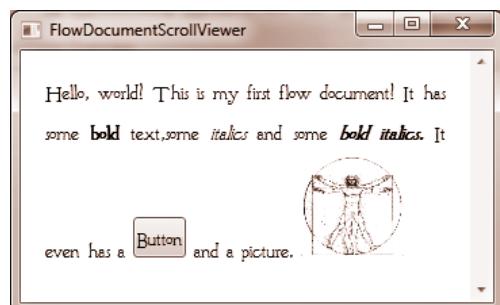
FLOW CONTROLS



A RichTextBox allows the text to be edited. It even exposes the conventional keyboard shortcuts like Ctrl+B for bold or Ctrl+I for italics. It even supports spellchecking.



The FlowDocumentPageViewer displays one page at a time, while the FlowDocumentScrollView displays its content as a single page with a scroll bar.



ON YOUR OWN

You don't need my help for this. To get a sense of what each of these flow controls looks like and how it works, wrap the flow document snippet from the last exercise in each of them and give them a whirl.



EFFECTS

Since version 3.5, the .NET Framework has provided direct support for off-loading the rendering of graphic effects to the GPU (the specialized Graphics Processing Unit that's optimized for this kind of rendering).

The library itself defines two effects that we'll have a quick look at here: `BlurEffect` and `DropShadowEffect`. There are effects libraries available on CodePlex and elsewhere, and you can even write your own. That's the good news. The bad news is that custom effects are written in HLSL

(“High Level Shading Language”) that’s part of DirectX and way outside of our scope. Here’s a snippet of HLSL. If you can figure out what it does, you already know a whole lot more about the subject than I do.

If you’re interested in doing low-level graphics programming, you’ll want to look at DirectX, Microsoft’s graphics toolset. Microsoft has an entire development center devoted to the subject, and there are dozens of books and blogs around; just do a search for DirectX in MSDN or your favorite search engine. (But I’ll warn you, graphics programming, unlike general programming, is very math-intensive.)

```
VS_OUTPUT VS_Skinning_Example(const VS_INPUT v, uniform float Len=100)
{
    VS_OUTPUT out;

    // Skin position (to world space)
    float3 vPosition =
        mul(v.vPosition, (float4x3) mWld1) * v.vBlendWeights.x +
        mul(v.vPosition, (float4x3) mWld2) * v.vBlendWeights.y +
        mul(v.vPosition, (float4x3) mWld3) * v.vBlendWeights.z +
        mul(v.vPosition, (float4x3) mWld4) * v.vBlendWeights.w;

    // Skin normal (to world space)
    float3 vNormal =
        mul(v.vNormal, (float3x3) mWld1) * v.vBlendWeights.x +
        mul(v.vNormal, (float3x3) mWld2) * v.vBlendWeights.y +
        mul(v.vNormal, (float3x3) mWld3) * v.vBlendWeights.z +
        mul(v.vNormal, (float3x3) mWld4) * v.vBlendWeights.w;

    // Output stuff
    out.vPosition = mul(float4(vPosition + vNormal * Len, 1), mTot);
    out.vDiffuse = dot(vLight, vNormal);

    return out;
}
```

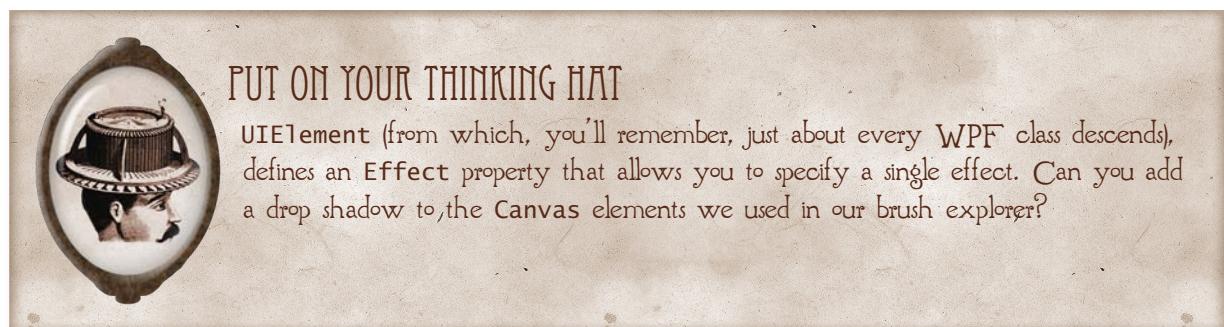
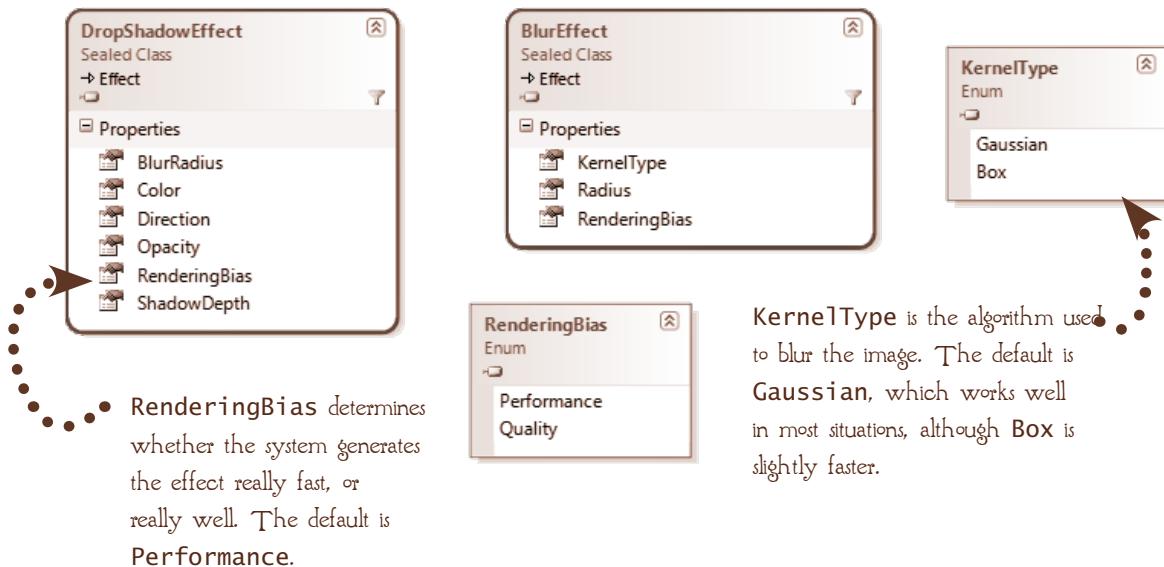


MAKE A NOTE

The FCL also defines a set of classes called bitmap effects that are software rendered (for which read “painfully slow”). They’re still there, but they’re marked obsolete, so you’ll get a nonfatal error if you use them.

FRAMEWORK EFFECTS

Okay, so custom effects and HLSL are so far out of our scope you can't even see them from here. But the Framework defines two effects that are useable (and useful) even for mathematically challenged programmers like me. Drop shadows are a common technique for setting off areas of the interface, and blurs can be very cool when combined with animation (which we'll discuss in the next chapter).

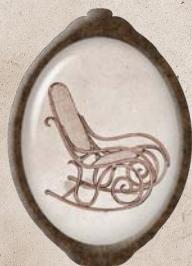




HOW'D YOU DO?

Did you have trouble finding the `Effect` property in Intellisense? (I sometimes do.) Here's my very simple version, and the result.

```
<Canvas ...>
<Canvas.Effect>
  <DropShadowEffect Color="DarkGray" />
</Canvas.Effect>
</Canvas>
```



TAKE A BREAK

Well, that's it for WPF graphics, at least for right now. We'll look at some of these elements in more detail in later chapters, but these are the basics. So, why don't you take a break before completing the final Review?



REVIEW

What control would you use to display read-only text with a decoration?

What are the ScRGB values for a pure blue?

What kind of brush would you use to display a picture in your UI?

What kind of brush would you use to make something look 3-dimensional?

Write a snippet that declares a radial gradient with three evenly distributed stops.

What brush would you use to repeat an image across an area?

What property do you set on a Pen to draw a dashed line?

What Rect property do you set to determine the size of a repeated image?

For layout and calculating gradients, WPF surrounds every object with an imaginary box.
What's it called?



REVIEW (CON'T)

What is the relationship between a font family and a typeface in WPF?

What is the difference between oblique and italic?

When would you want to use a flow document?

What are fixed documents primarily used for?

What two hardware-accelerated effects are defined by the .NET Framework?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

①

②

③

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



RESOURCES

20

In the last chapter we explored some of the rich graphics capabilities that WPF provides. We only skimmed the surface, of course, but you should have a sense of what the basic graphic objects like gradients and brushes are capable of. With the exception of writing custom effects in HLSL, none of the graphics classes are difficult to use.

But what if you have a beautiful radial gradient with, say, eight or nine gradient stops, all custom colors, that you want to use for every button in your application? You wouldn't want to define it in every `<Button>` tag; that would just be too tedious. Not to mention almost guaranteeing that the gradient would change at some point in the future, requiring you to change the code for every button in your application. Ouch.

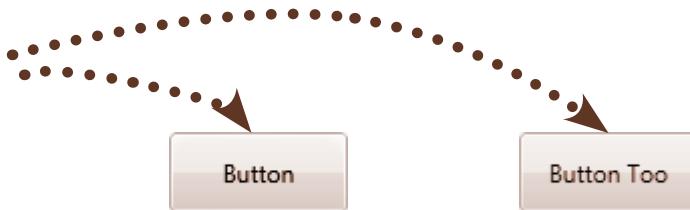
You could create a `Class` that declares the `Brush` as a property, but that adds to the complexity of your class hierarchy, and it means defining the brush in code, which is awkward and not a good option if you're lucky enough to have a graphic designer working on your application.

The best solution, in most situations, is to define the brush as a WPF resource. In WPF, a `Resource` is an object that is defined once, either directly in the `XAML` of the window or (more commonly) as a separate file, and used by multiple elements. The object could be a graphic object like our radial gradient, some `XML` data, or even just a simple value that you need to reference in several places. `Resources` are usually defined in `XAML`, but you can do it in code (if you really need to). `Resources` are also, by a strange coincidence, the subject of this chapter.

FITTING IT IN

You know that in OOP, objects have both state and behavior, and good programmers try very hard not to duplicate either of them. Using resources, we can avoid duplicating state or behavior between instances of classes.

These are both instances of the class `Button`.



STATE:

I'm painted with a silver gradient. ← I'm painted with a silver gradient.

I have a black border. ← I have a black border.

My `FontFamily` is Segoe UI. ← My `FontFamily` is Segoe UI.

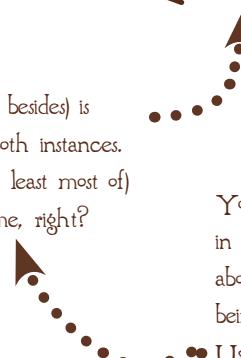
BEHAVIOR:

I look different when I'm pressed. ← I look different when I'm pressed.

I look different when I'm disabled. ← I look different when I'm disabled.

I look different when I'm selected. ← I look different when I'm selected.

All of this state and behavior (and a lot more besides) is defined at the instance level but shared by both instances. It's pretty common for you to want all (or at least most of) the buttons in your application to look the same, right?



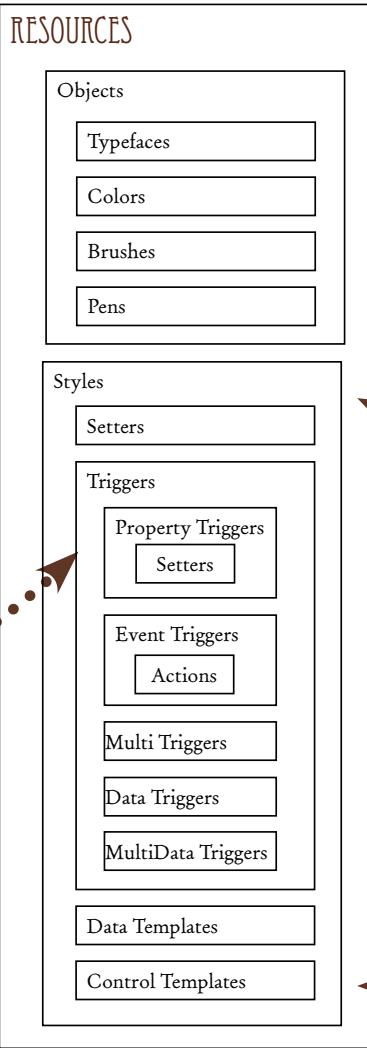
You haven't been defining most of these characteristics in your applications and probably haven't thought a lot about how or where they're defined. They're actually being defined in default style sheets provided by WPF.

Using what you learn in this chapter, you can take complete control over these characteristics (and more).

RESOURCES let you define objects that can be used anywhere.



TRIGGERS are a way of defining some behavior declaratively. They're not as powerful as, say, an event handler, but you can do a surprising amount without writing code.



STYLES let you assign a set of properties to an element or a type of element.



We'll be exploring CONTROL TEMPLATES in the next chapter and data templates in the chapter after that.



PUT ON YOUR THINKING HAT

Resources allow you to separate state and behavior in new and exciting ways. What benefits and drawbacks do you see to using this approach?



HOW'D YOU DO?

How many did you come up with? Did you think of some I haven't? Do you disagree with me about some of them?

BENEFITS

- Resources defined declaratively are easier to implement by non-programmers.
- Resources defined outside the element definition allow appearance and appearance-related behavior to be changed without re-compiling (if they're in a separate file).
- Resources defined outside the element allow different individuals or teams to work on code and appearance.
- Resources help enforce a consistent look and behavior throughout the application.
- Resources can improve performance by reducing the application's memory usage (but this is only true if resources are shared among multiple elements).

DRAWBACKS

- Multiple files can complicate managing and deploying the application.
- Property inheritance combined with resources can make tracking down display problems more complex. (You have some extra places to look.)



TASK LIST

In this chapter we'll explore the basic elements of WPF resources.



DEFINING RESOURCES

Our first step, of course, is to figure out how resources are declared and how to use them once they are. There are several options, which probably comes as no surprise at this point. We'll examine the most common ones.



STYLES

While simply declaring an object as a resource so that it can be reused is really useful, it's even better when you can combine a set of property declarations that will apply to all (or some) of the objects in your application.



PROPERTY TRIGGERS

Trigger elements are defined as part of a Style. They make it possible to change property values or even perform simple actions when something happens in the application. There are two kinds of triggers and we'll start by exploring the simplest type: property triggers that are invoked ("triggered") when the value of a property changes.



EVENT TRIGGERS & ANIMATION

The second type of trigger is an event trigger that is invoked when certain events (other than a change in property values) occur. WPF Resources and styles can't completely replace the code-behind you'll write (at least not in the immediate future), but both property and event triggers can perform simple actions, and the ones the Framework Library defines for you all relate to animation.



DEFINING RESOURCES

WPF Resources are simply elements that are defined inside a `ResourceDictionary`. Since both `FrameworkElement` and `FrameworkContentElement` define a `Resources` property of this type, you can define them pretty much wherever you need to.

Resources are often defined inside the `Application.xaml` file to make them available throughout the application, but you can define them inside any element using the same syntax.

```
<Application ...>
  <Application.Resources>
    <SolidColorBrush x:Key="MyBlueBrush"
      Color="Blue" />
  </Application.Resources>
</Application>
```

You can define any element you want inside a resource dictionary, but of course not everything makes sense.

If the resources are defined in a separate file, as they often are, you include them in your application by setting the `Source` property of the `ResourceDictionary`.

```
<ResourceDictionary Source="dict.xaml" />
```

```
<ResourceDictionary>
  ...
</ResourceDictionary>
```

An external dictionary is just a XAML snippet that wraps everything in a `ResourceDictionary` element.

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="First.xaml" />
      <ResourceDictionary Source="Second.xaml" />
      ...
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
  <SolidColorBrush ... />
</Application.Resources>
```

If you have multiple external files, you can include them in the **MergedDictionaries** collection of the **ResourceDictionary**.



You can define other resources along with the merged dictionaries, but it isn't very common practice.



PUT ON YOUR THINKING HAT ...

Add a resource dictionary called **FluentDictionary** to a new WPF application (it's available from the Add New Item dialog.)

Add a **SolidColorBrush** to **FluentDictionary**, any color you like. Give it the key "SolidBackground".

Add **FluentDictionary** to a **MergedDictionary** inside the project's **App.xaml** file.



HOW'D YOU DO?

Visual Studio adds some of the structure for you, but not all...

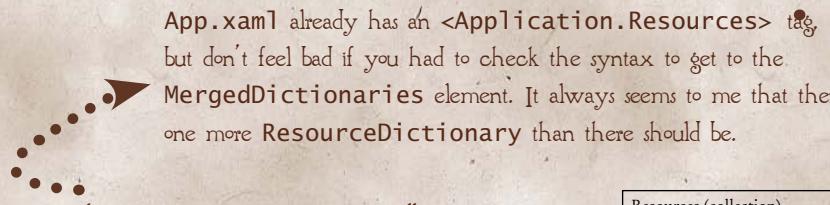
Here are the contents of `FluentDictionary.xaml`:

When you added the file to the project, Visual Studio created the `ResourceDictionary` tag for you. Good thing, too; who wants to type those namespaces?

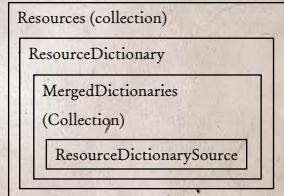
```
<ResourceDictionary ... >
  <SolidColorBrush x:Key="SolidBackground"
    Color="DarkTurquoise" />
</ResourceDictionary>
```

```
<Application ... >
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="FluentDictionary.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

I picked `DarkTurquoise`. You could have picked a different standard color or specified one using the hexadecimal values.


App.xaml already has an `<Application.Resources>` tag, but don't feel bad if you had to check the syntax to get to the `MergedDictionaries` element. It always seems to me that there's one more `ResourceDictionary` than there should be.

I'm wrong about that, of course. The `Resources` property is a collection, not a `ResourceDictionary`. The `<ResourceDictionary>` tag is an element in that collection, and the `MergedDictionaries` property contains another collection, with yet another `ResourceDictionary` contained in it.



REFERENCING RESOURCES

As we'll see, it's possible to define a style that applies to all elements of a given type automatically, but most of the time you'll have to reference the `ResourceKey` directly using one of the two XAML markup extensions that define how WPF treats the resource.

STATIC RESOURCES are evaluated once, when the XAML is first parsed at runtime. This is what you'll use most often.

```
<property>="{StaticResource <key>}"
```

```
<Label FontFamily="{StaticResource MainFont}" >
```

DYNAMIC RESOURCES are evaluated when the XAML is first parsed at runtime and change their values whenever the value of the resource changes.

```
<property>="{DynamicResource <key>}"
```

```
<Label FontFamily="{DynamicResource MainFont}" >
```



PUT ON YOUR THINKING HAT

Add the code to set the `MainWindow.Background` property of your application to the resource you defined in `FluentDictionary.xaml`. Do you think it should be a static or dynamic resource?



HOW'D YOU DO?

What did you decide?

```
<Window ...
```

```
    Background="{StaticResource SolidBackground}" >
```

The background color isn't going
to change, so there's no reason
to make it a dynamic resource.

Dynamic resources incur a small
performance penalty. These things add
up, so you shouldn't use them unless you
need to.



MAKE A NOTE—STATIC RESOURCES

The Framework Library defines several classes that expose static properties relating to the system configuration, including `SystemColors`, `SystemFonts` and `SystemParameters`. You'll often want these to be dynamic resources so they'll update if the user changes the values while the application is running. In order to do that, you need to reference the resource key version of the property (the `SystemColors.MenuBarColorKey` property instead of `SystemColors.MenuColor` property, for example).

Because these properties are static, you need to use another markup extension, `x:Static`:

```
{DynamicResource {x:Static SystemFonts.CaptionFontFamilyKey}}"
```

The `x:Static` extension is required whenever you're referring to a static property, field, enumeration value or constant.

`x:Static` is most often used in the context of resources, but you'll need it anywhere you reference one of these objects in XAML.

RESOURCES IN CODE

It's very unusual to need to create or even reference resources in code, and it's far more verbose than XAML, but you can if you need to.

TO CREATE A RESOURCEDICTIONARY

Nothing tricky here...

```
<rd> As ResourceDictionary = New ResourceDictionary()
```

TO ADD A RESOURCE TO A DICTIONARY

```
<rd>.Add(<ResourceKey>, <Object>)
```

- The `ResourceKey` and `Resource` are both `Object` types.

TO REFERENCE A RESOURCEDICTIONARY

All of these methods are defined on `FrameworkElement` and `FrameworkContentElement`, so they're available throughout WPF. The resources are always dynamic.

```
FindResource(<ResourceKey>)
```

or

```
TryFindResource(<ResourceKey>)
```

These methods return an `Object`, so you'll need to cast the object returned.

```
MyButton.Background = (Brush)this.TryFindResource("MyBrush")
```

- The `Try...` version returns a null if the resource isn't found, rather than throwing an error.



TAKE A BREAK

Feel free to play with creating and referencing resources in code, but you really only need to remember that it's possible, because you're not going to do it often (if ever). For now, why don't you take a quick break before we tackle styles?



BASIC STYLING

Resources are helpful, but defining individual resources can lead to some really verbose (not to mention tedious) code. The solution is to define a **Style**, which at its most basic is a resource that contains a collection of property values that can be applied all at once.

The **Style** contains a collection of **Setter** elements, each of which has a **Property** and a **Value**.

```
<Style x:Key="BoldItalicLabel" >
  <Setter Property="FontFamily" Value = "Arial" />
  <Setter Property="Foreground">
    <Setter.Value>
      <SolidColorBrush Color="Black" />
    </Setter.Value>
  </Setter>
</Style>
```

This is a simple example, but the **Value** property can be as complex as you need.

KEYED STYLES

Styles can be declared inside a resource dictionary or in the Resources collection of an element. In either case, you can give it a key and reference it using the same syntax you use to reference a Resource.

```
<ResourceDictionary>
  <Style x:Key="LabelStyle" >
    <Setter Property="Label.FontFamily" Value="Arial" />
  </Style>
</ResourceDictionary>

<Label Style="{StaticResource LabelStyle}">Hello!</Label>
```

TARGETED STYLES

If you add a **TargetType** to the **Style**, you don't need to keep specifying the element as part of the **Setter.Property** definition. You'll almost always do this when you declare a **Style** inside the Resources collection of an element.

```
<ResourceDictionary>
  <Style x:Key="LabelStyle" TargetType="Label">
    <Setter Property="FontFamily" Value="Arial" />
  </Style>
</ResourceDictionary>

<Label Style="{StaticResource LabelStyle}">Hello!</Label>
```

You can address properties directly, which saves a lot of typing.

IMPLICIT STYLES

If you declare a style's TargetType but don't give it a key, you don't need to keep specifying the element as part of the `Setter.Property` definition, and you don't need to apply it explicitly to elements of that type; it will become the default.

```
<ResourceDictionary>
  <Style TargetType="Label">
    <Setter Property="FontFamily" Value="Arial" />
  </Style>
</ResourceDictionary>

<Label>Hello!</Label>
```



In .NET 4.0, the `Background` property of an implicit style won't work for the `Window` if you're using a merged dictionary. Use a keyed style instead.



PUT ON YOUR THINKING HAT

Can you build a resource dictionary to define the visual appearance of this window? Remember to set up a merged dictionary inside `app.xaml` to get access to the dictionary.

The gradients used for the backgrounds of the `Window` and `Buttons` have the same `GradientStopCollection` but different start and end points. How can you avoid defining the stops twice?

There are two types of labels.



All of the text on the window has the same `FontFamily` and `FontSize`. How can you specify that only once? (Hint: Dependency property inheritance still works, even if the properties are set via a `Style`.)



HOW'D YOU DO?

The exact fonts and colors you chose aren't important. You probably chose different key names, too. That doesn't matter either.

MAINWINDOW.XAML

```
<Window ... Style="{StaticResource WindowStyle}">
<Grid>
    ...
    <Label ... Style="{StaticResource BasicLabel}">First Optional Item:</Label>
    <Label ... Style="{StaticResource BasicLabel}">Second Optional Item:</Label>
    <Label ... Style="{StaticResource RequiredLabel}">Required Item:</Label>
    <Label ... Style="{StaticResource RequiredLabel}">Second Required Item:</Label>

    <TextBox Grid.Row="0" Grid.Column="1">First Option</TextBox>
    <StackPanel Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2"
               Orientation="Horizontal" Height="35">
        <Button>Submit</Button>
        <Button>Cancel</Button>
    </StackPanel>
</Grid>
</Window>
```

Need the key because
of that nasty bug.

Need the keys here, too,
because there isn't a single
style.

We don't need to reference
styles here, because they're
applied implicitly.

- Omitting the key here makes this an implicit style that will be applied to both the buttons.

FLUENTDICTIONARY.XAML

```
<ResourceDictionary ...>
  <GradientStopCollection x:Key="Stops">
    <GradientStop Offset="0" Color="AntiqueWhite" />
    ...
  </GradientStopCollection>
```

If you figured this out without peeking, give yourself a couple of gold stars. It's tricky!

You need the key here because of the .NET 4.0 background bug.

```
  <Style x:Key="WindowStyle" TargetType="Window">
    <Setter Property="Background">
      <Setter.Value>
        <LinearGradientBrush GradientStops="{StaticResource Stops}" />
      </Setter.Value>
    </Setter>
    <Setter Property="FontFamily" Value="Arial" />
    <Setter Property="FontSize" Value="14" />
  </Style>
```

```
  <Style TargetType="Button">
    <Setter Property="Background" >
      <Setter.Value>
        <LinearGradientBrush GradientStops="{StaticResource Stops}"
          StartPoint="0, 0.5" EndPoint="0,1" />
      </Setter.Value>
    </Setter>
    <Setter Property="Margin" Value="5" />
    <Setter Property="Padding" Value="2" />
  </Style>
```

By setting the text characteristics here, they'll be inherited by the elements within the Window.

```
  <Style TargetType="TextBox">
    <Setter Property="Background" Value="AntiqueWhite" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="Margin" Value="10" />
  </Style>
```

Does this seem wrong to you? It works, but there's a lot of duplication, and we always want to avoid that. We'll look at a couple of ways to eliminate it on the next page.

```
  <Style x:Key="BasicLabel" TargetType="Label">
    <Setter Property="Margin" Value="5" />
    <Setter Property="HorizontalAlignment" Value="Right" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="FontWeight" Value="Black" />
  </Style>
```

```
  <Style x:Key="RequiredLabel" TargetType="Label">
    <Setter Property="Margin" Value="5" />
    <Setter Property="HorizontalAlignment" Value="Right" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="FontWeight" Value="Black" />
    <Setter Property="Foreground" Value="Chocolate" />
  </Style>
```

```
</ResourceDictionary>
```

STYLE HIERARCHIES

In the solution to the last exercise, I created two different styles for the `Label` elements, and pointed out that there was a lot of duplication. The `Style.BasedOn` property lets you avoid the duplication:

INSTEAD OF

```
<Style x:Key="BasicLabel" TargetType="Label">
<Setter Property="Margin" Value="5" />
<Setter Property="HorizontalAlignment" Value="Right" />
<Setter Property="VerticalAlignment" Value="Center" />
<Setter Property="FontWeight" Value="Black" />
</Style>

<Style x:Key="RequiredLabel" TargetType="Label">
<Setter Property="Margin" Value="5" />
<Setter Property="HorizontalAlignment" Value="Right" />
<Setter Property="VerticalAlignment" Value="Center" />
<Setter Property="FontWeight" Value="Black" />
<Setter Property="Label.Foreground" Value="Chocolate" />
</Style>
```

BASEDON LETS US DO THIS

You can specify the `TargetType` in a style that's based on another, but it must be the same type or a type descended from the original type.

```
<Style x:Key="BasicLabel" TargetType="Label">
<Setter Property="Margin" Value="5" />
<Setter Property="HorizontalAlignment" Value="Right" />
<Setter Property="VerticalAlignment" Value="Center" />
<Setter Property="FontWeight" Value="Black" />
</Style>

<Style x:Key="RequiredLabel" BasedOn="{StaticResource BasicLabel}" TargetType="Label">
<Setter Property="Foreground" Value="Chocolate" />
</Style>
```

Using the `BasedOn` property with a key is better, but it means that the parent style can't be implicit. Fortunately, the `BasedOn` property can also reference a type using the `x:Type` markup extension:

```
<Style TargetType="Label">
  <Setter Property="Margin" Value="5" />
  <Setter Property="HorizontalAlignment" Value="Right" />
  <Setter Property="VerticalAlignment" Value="Center" />
  <Setter Property="FontWeight" Value="Black" />
</Style>

<Style x:Key="RequiredLabel" BasedOn="{StaticResource {x:Type Label}}"
       TargetType="Label">
  <Setter Property="Label.Foreground" Value="Chocolate" />
</Style>
```

You have to specify the `TargetType` when you use `x:Type` to refer to an implicit type, and it has to be the same or a child type.

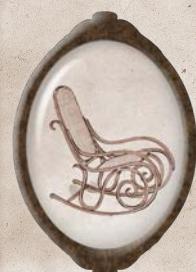
Notice that we need another set of braces when we use `x:Type`.



ON YOUR OWN

Edit your sample application to eliminate the duplicate definitions using `BasedOn`. Try to do it without looking at the examples here.

When you run the application to test your changes, put your mouse over one of the buttons. They'll change color, but not to the color you've chosen for your gradient. How do you think you can fix that?



TAKE A BREAK

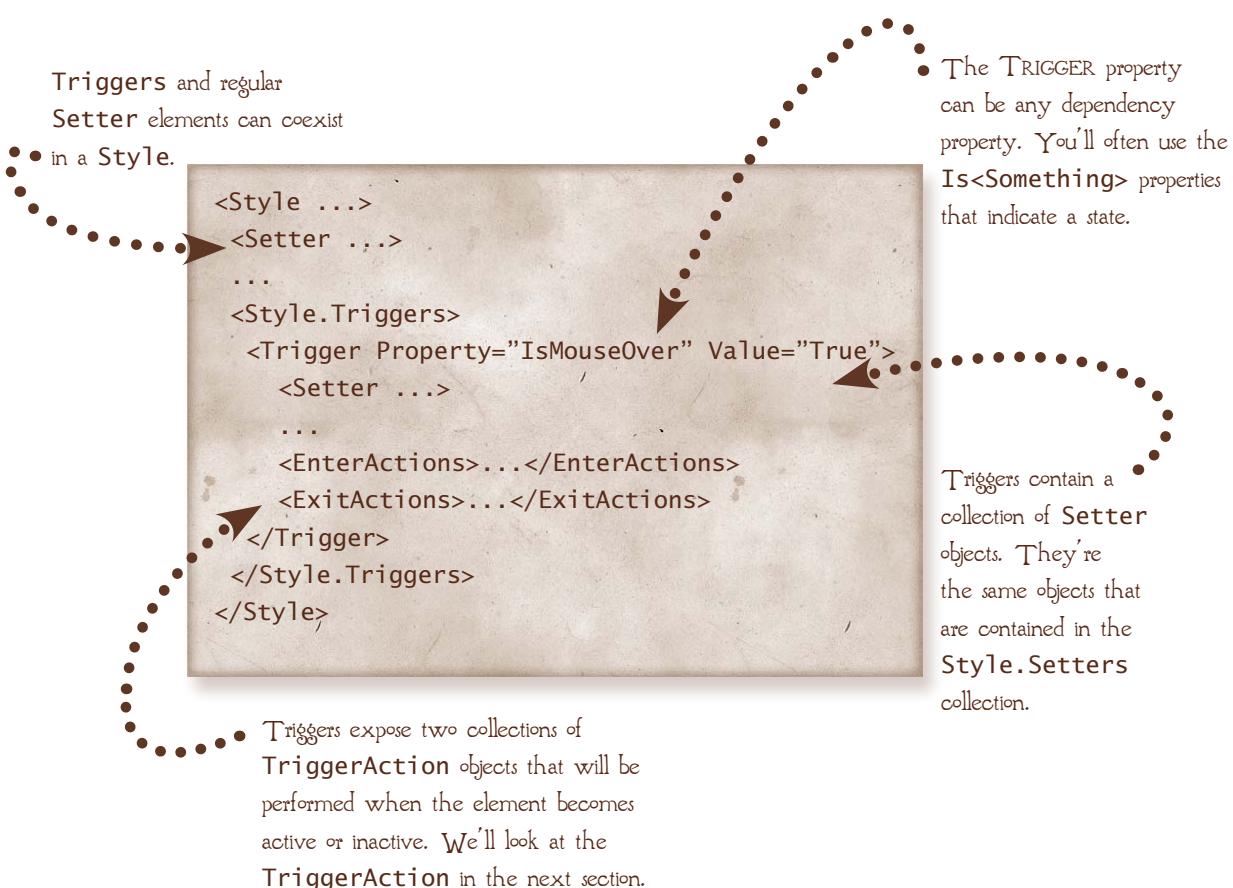
After you update your sample, why don't you take a break before we move on to triggers?



PROPERTY TRIGGERS

Basic styles have a lot of benefits: They help ensure visual consistency, they make it easier to hand off the graphics to a different team, and they reduce repetitive code. But they have one drawback: There's only one style per customer. A lot of elements, like the `Button`, actually have multiple appearances, depending on various conditions like whether they're selected or the user has moused over them. The solution to that is the `Triggers` collection that's exposed by the `Style` object.

There are three kinds of triggers. PROPERTY TRIGGERS, which we'll examine in this section, are applied when a property has a specified value. EVENT TRIGGERS, which we'll examine in the next section, apply their values when an event occurs. Finally, DATA TRIGGERS are like property triggers, but they're based on a bound value. We'll examine those in Chapter 22.





PUT ON YOUR THINKING HAT ...

Try out triggers by adding some to the `FluentDictionary` we've been working with.

- ① Add a trigger to the `TextBox` that sets the `FontStyle` to `Italic` when it has focus. The trigger property isn't `IsSelected`. What is it? (You can check MSDN or the object browser for the list of properties exposed by the `TextBox` class.)
- ② Add a trigger to the `Button` style that changes the `FontWeight` to `Bold` and the `FontStyle` to `Italic` when the button is pressed.
- ③ Write a trigger for the button that reverses the gradient when `IsMouseOver` is true. Hint: You need to change the starting and ending points, but you can use the same collection of stops. The trigger won't quite work. We won't be able to fix it until we look at control templates in the next chapter, but can you identify the problem? (You might not be able to if you have a really, really fast machine.)
- ④ Triggers are applied when the specified property is set to the specified value. What happens when the value changes? (To test this, try tabbing in and out of one of the `TextBox` elements on our sample window.)



HOW'D YOU DO?

- ① <Style TargetType="TextBox">
...
<Style.Triggers>
 <Trigger Property="IsFocused" Value="true">
 <Setter Property="FontStyle" Value="Italic" />
 </Trigger>
</Style.Triggers>
</Style>

- ② <Style TargetType="Button">
...
<Style.Triggers>
 <Trigger Property="IsPressed" Value="true">
 <Setter Property="FontWeight" Value="Bold" />
 <Setter Property="FontStyle" Value="Italic" />
 </Trigger>
 <Trigger Property="IsMouseOver" Value="true">
 <Setter Property="Background">
 <Setter.Value>
 <LinearGradientBrush
 GradientStops="{StaticResource Stops}"
 StartPoint="1,0.5" EndPoint="1,0" />
 </Setter.Value>
 </Setter>
 </Trigger>
</Style.Triggers>
</Style>

- ③ When the trigger condition is no longer true, the element will revert to its last style. You don't need to do anything to make this happen; WPF handles it for you automatically.

When you mouse over the button, the gradient will change briefly, but then it will revert to the Windows default. It's because another part of the WPF default **Button** style takes over. We'll see how to fix it when we look at templates in the next chapter. For now, you should remember that if a style isn't working the way you think it should, it may not be because you've done something wrong, only that there's something else going on.

MULTIPLE TRIGGER CONDITIONS

There's a lot to be said for a declarative programming language like XAML. It makes certain kinds of problems (like laying out a UI) much easier to think about and implement. But almost by definition, it's difficult to do any kind of conditional processing declaratively. Triggers help; they let you set properties based on the value of a single property, but only a single property. MultiTriggers extend the trigger paradigm by letting you evaluate multiple properties, but only using a logical AND.

```
<Style ...>
  <Setter Property="FontFamily" Value="Arial" />
  <Style.Triggers>
    <Trigger Property="IsEnabled" Value="false" />
      <Setter Property="FontWeight" Value="Light" />
    </Trigger>
  </Style.Triggers>
</Style>
```

A basic **Trigger** is a simple if statement. This one reads, "If the text is disabled, make it light. Otherwise make it normal." Or maybe "Make the **FontWeight** normal unless **.IsEnabled** is false," depending on how you think about it.

A **MultiTrigger** is a slightly more complex if statement, with the conditions combined with AND. This one reads, "If the text is enabled and selected, make it bold." Notice that the **MultiTrigger** can be combined with basic triggers and with plain Setters.

```
<Style ...>
  <Setter Property="FontFamily" Value="Arial" />
  <Style.Triggers>
    <Trigger Property="IsEnabled" Value="false" />
      <Setter Property="FontWeight" Value="Light" />
    </Trigger>
    <MultiTrigger>
      <MultiTrigger.Conditions>
        <Condition "IsEnabled" Value="true" />
        <Condition "IsSelected" Value="true" />
      </MultiTrigger.Conditions>
        <Setter Property="FontWeight" Value="Bold" />
      </MultiTrigger>
    </Style.Triggers>
  </Style>
```



PUT ON YOUR THINKING HAT

Can you rewrite the **MultiTrigger** example, including all the conditions implied by the other triggers and setters in the **Style**, in VB? How about translating this VB snippet into a **Style**?

```
If Me.IsEnabled Or Me.Focused Then
  Me.FontWeight = FontWeight.Bold,
Else
  Me.FontWeight = FontWeight.Normal
End If
```



HOW'D YOU DO?

You are, of course, entitled to disagree, but I think the XAML is more verbose, but easier to understand. Nested if statements are very, very complex and easy to mess up.

Me.FontFamily = SystemFonts.Arial

```
If Me.IsEnabled = True Then  
  If Me.IsSelected = True Then  
    Me.FontWeight = FontWeights.Bold  
  Else  
    Me.FontWeight = FontWeights.Normal  
Else  
  Me.FontWeight = FontWeights.Light
```

```
If Me.IsEnabled Or Me.Focused Then  
  Me.FontWeight = FontWeight.Bold  
Else  
  Me.FontWeight = FontWeight.Normal  
End If
```

```
<Style ...>  
  <Setter Property="FontFamily" Value="Arial" />  
  <Style.Triggers>  
    <Trigger Property="IsEnabled" Value="false" />  
      <Setter Property="FontWeight" Value="Light" />  
    </Trigger>  
  <MultiTrigger>  
    <MultiTrigger.Conditions>  
      <Condition "IsEnabled" Value="true" />  
      <Condition "IsSelected" Value="true" />  
    </MultiTrigger.Conditions>  
    <Setter Property="FontWeight" Value="Bold" />  
  </MultiTrigger>  
</Style.Triggers>  
</Style>
```

You can't represent a logical OR using a MultiTrigger. (Don't feel bad if you tried to; it was a trick question.)

You have two choices in this situation: Either do as I did here, and repeat the setters in each Trigger, or (better) create a keyed style that gets set inside the Trigger:

```
<Setter Property="Style"  
       Value="" />  
  
<Style>  
  <Setter Property="FontWeight" Value="Normal" />  
  <Style.Triggers>  
    <Trigger Property="IsEnabled" Value="true">  
      <Setter Property="FontWeight" Value="Bold" />  
    </Trigger>  
    <Trigger Property="IsFocused" Value="true">  
      <Setter Property="FontWeight" Value="Bold" />  
    </Trigger>  
  </Style.Triggers>  
</Style>
```



STYLE AND TRIGGER GOTCHAS

"It's right, but it doesn't work." Words to strike terror in any programmers heart. And you will say them far more frequently than, say, "The bear ate my ice cream." You've already seen one example, the button trigger that didn't quite work. Let's look at a few more:

- ➊ Add two triggers to the implicit style we created for the `Window`, one for `IsLoaded` and one for `IsEnabled`. Inside each trigger, set the `Background` property to a different value. Which background is displayed when you run the application?
- ➋ Change the order in which the triggers are declared. Now which background is displayed?
- ➌ Add another implicit `Button` style at the end of the `ResourceDictionary` (after the `TextBox` style). Set the `Background` property in a plain `Setter` (not a `Trigger`) to `Bisque`. When you run the application, which background does the button display?
- ➍ Now swap the two `Button` styles in the file. Which one is displayed?
- ➎ Add another implicit `Window` style to the end of the `ResourceDictionary`. Use a plain `Setter` to set the `Background` to `Aqua`. Now which background is displayed?

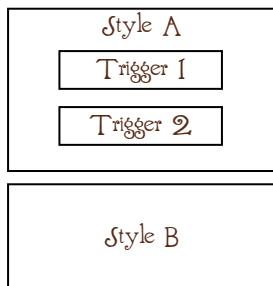


ON YOUR OWN

Can you work out the rules that determine the final value of a property when more than one `Setter` or `Trigger` applies?

COMBINING TRIGGERS & STYLES

Did you work out the rule that applies to both triggers and styles? Triggers beat styles, and whenever there are two or more of either of them, the last one declared wins.



AND THE WINNER IS....

Trigger 2. Even though style B takes precedence over style A because it is declared after it, triggers take precedence over styles, so the last applicable trigger is applied.

Now, the examples we played with on the last page were a little silly. You wouldn't define triggers for properties that aren't mutually exclusive, or two implicit templates for the same element in a single file. If you did, it would be fairly easy to track down the problem. But this kind of thing can easily happen when you're working with multiple `ResourceDictionaries`.

What's a poor programmer to do? WPF is doing exactly what you told it to do (you just didn't tell it what you thought you did), so classic debugging isn't going to help you, but here are some tips.

BE AWARE OF THE PROBLEM

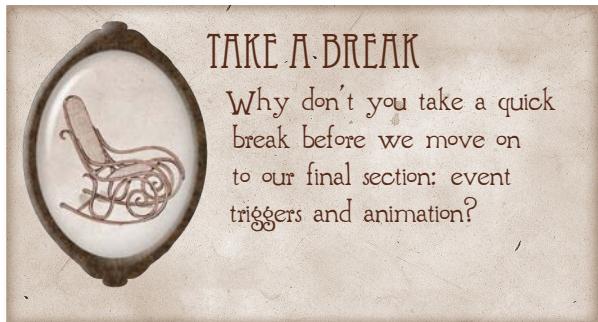
Just knowing how this kind of thing happens can help a lot.

MAKE IMPLICIT STYLES GENERIC

Your life will be easier if your implicit styles only set properties that really are valid for every instance of the element. I know the syntax for referencing a style is a bit ugly, and so it's tempting to choose the "most used" style as the default, but that will cause problems in the long run. (Yes, you're listening to the voice of experience.)

ORGANIZE STYLES FROM MOST-GENERAL TO MOST-SPECIFIC

Try to organize styles from most-generic (basic color and font resources, for example) to most specific (styles that only apply in a few cases), and when you're adding them to a merged dictionary, make sure you add them in that order. It's not a perfect solution, but it will help prevent styles from stepping on one another.





REVIEW

When would you reference a resource dynamically instead of statically?

What markup extension must you add to an element declaration inside a `ResourceDictionary`?

In order to make the dictionaries available to the entire application, the `ResourceDictionary.MergedDictionaries` collection is usually defined inside what file?

What property do you declare to base one property on another?

How do you create an implicit style?

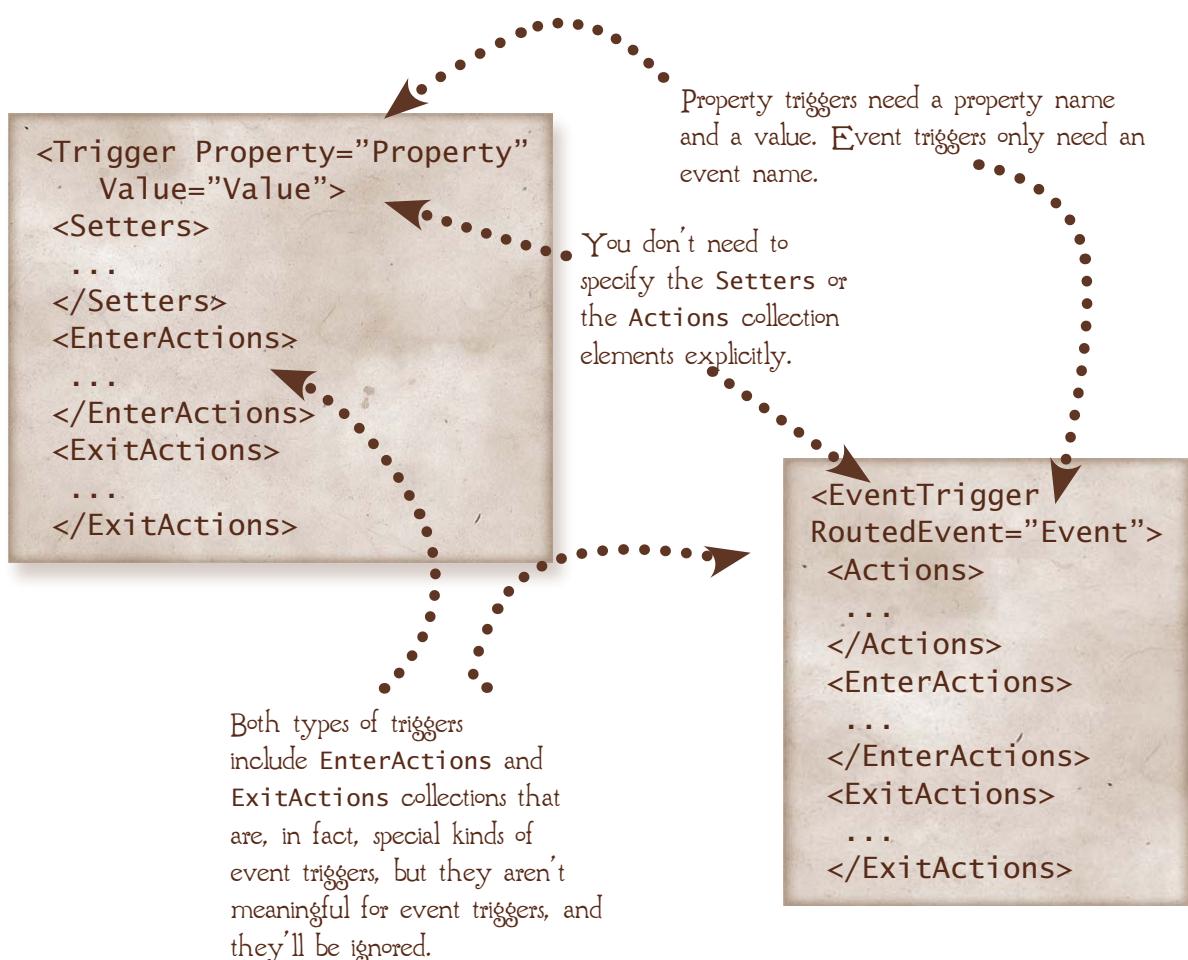
Textboxes that contain required information should be red and italic when they're not selected, pink when they're not enabled, and bold when they are focused and `CanUndo` is true. Write the style.



EVENT TRIGGERS

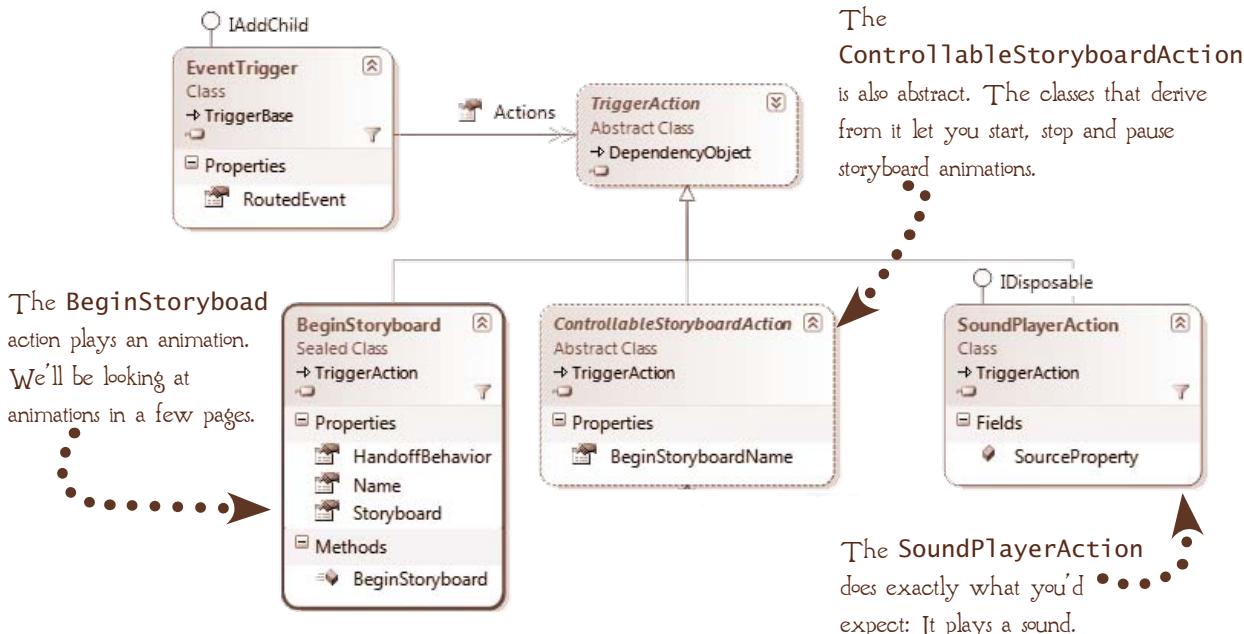
If a property trigger is activated by a change in a property value, what do you suppose activates an event trigger? You don't get a gold star for guessing an event. Specifically, a `RoutedEventArgs`. There's another difference, as well: Event triggers only execute actions.

Property triggers can execute actions via their `EnterActions` and `ExitActions` collections, but they're primarily used to set properties. The syntax for the two types of triggers is almost identical:



TRIGGER ACTIONS

The Actions collection of the EventTrigger accepts members of the abstract TriggerAction class. You can create your own actions by deriving from TriggerAction, but you'll most often work with one of the three trigger action classes defined in the Framework Library.



PUT ON YOUR THINKING HAT ...

We'll be looking at the **BeginStoryboard** trigger action and simple storyboard animations in a bit, but to get a sense of how event triggers work (it's easy), delete those duplicate definitions we added to **FluentDictionary** in the last exercise and then replace the **IsMouseOver** property trigger of the **Button** style with a **Click** event trigger.

You can set the **Source** property to any .wav file (search for *.wav in the Windows Explorer to find one, and then add it to your project).



HOW'D YOU DO?

Here's my version:

```
<Style TargetType="Button">
  <Setter Property="Background" >
    <Setter.Value>
      <LinearGradientBrush GradientStops="{StaticResource Stops}"
        StartPoint="0, 0.5" EndPoint="0,1" />
    </Setter.Value>
  </Setter>
  <Setter Property="Margin" Value="5" />
  <Setter Property="Padding" Value="2" />
  <Style.Triggers>
    <Trigger Property="IsPressed" Value="true">
      <Setter Property="FontWeight" Value="Bold" />
      <Setter Property="FontStyle" Value="Italic" />
    </Trigger>
    <EventTrigger RoutedEvent="Click">
      <SoundPlayerAction Source="giggle.wav" />
    </EventTrigger>
  </Style.Triggers>
</Style>
```

This EventTrigger has a single action, but just like setters in a property trigger, you can include as many as you need.



GOING FURTHER

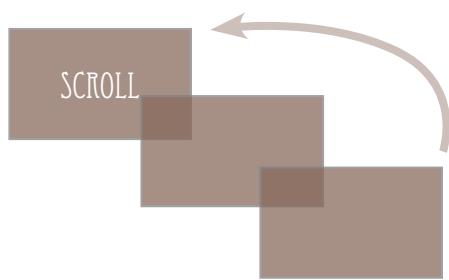
Under the covers, the SoundPlayerAction uses the lightweight SoundPlayer class, which only plays .wav files, and only plays them at the current speaker setting. If you need something more sophisticated, or the ability to play other types of media (including video), check out the MediaElement, which can be used from within a storyboard, which we'll discuss next.

WPF ANIMATIONS

The SoundPlayerAction is very simple, but surprisingly useful. The other two categories of event actions are used for animation. In WPF, the term "animation" doesn't have a lot to do with cranky hunters chasing wascally wabbits. A WPF animation changes the value of a dependency property over time, and unlike cartooning, this kind of animation can be really useful for making mainstream (and not-so-mainstream) interfaces responsive to the user. A few examples:

ANIMATION CAN CONTROL VISIBILITY

It's disconcerting to have things suddenly appear and disappear on the screen. By animating the opacity and size, you can fade things into and out of view smoothly and avoid startling people.

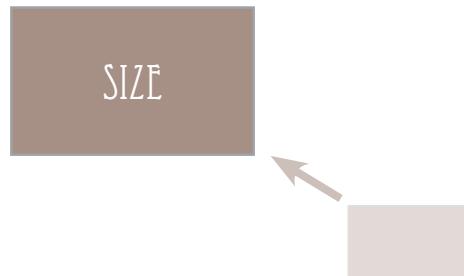


ANIMATION CAN MAKE MOVEMENT SEEM NATURAL

With the exception of slightly dubious experiments in quantum physics, things in the physical world don't get from one place to another without moving there. Things on a computer screen really shouldn't, either.

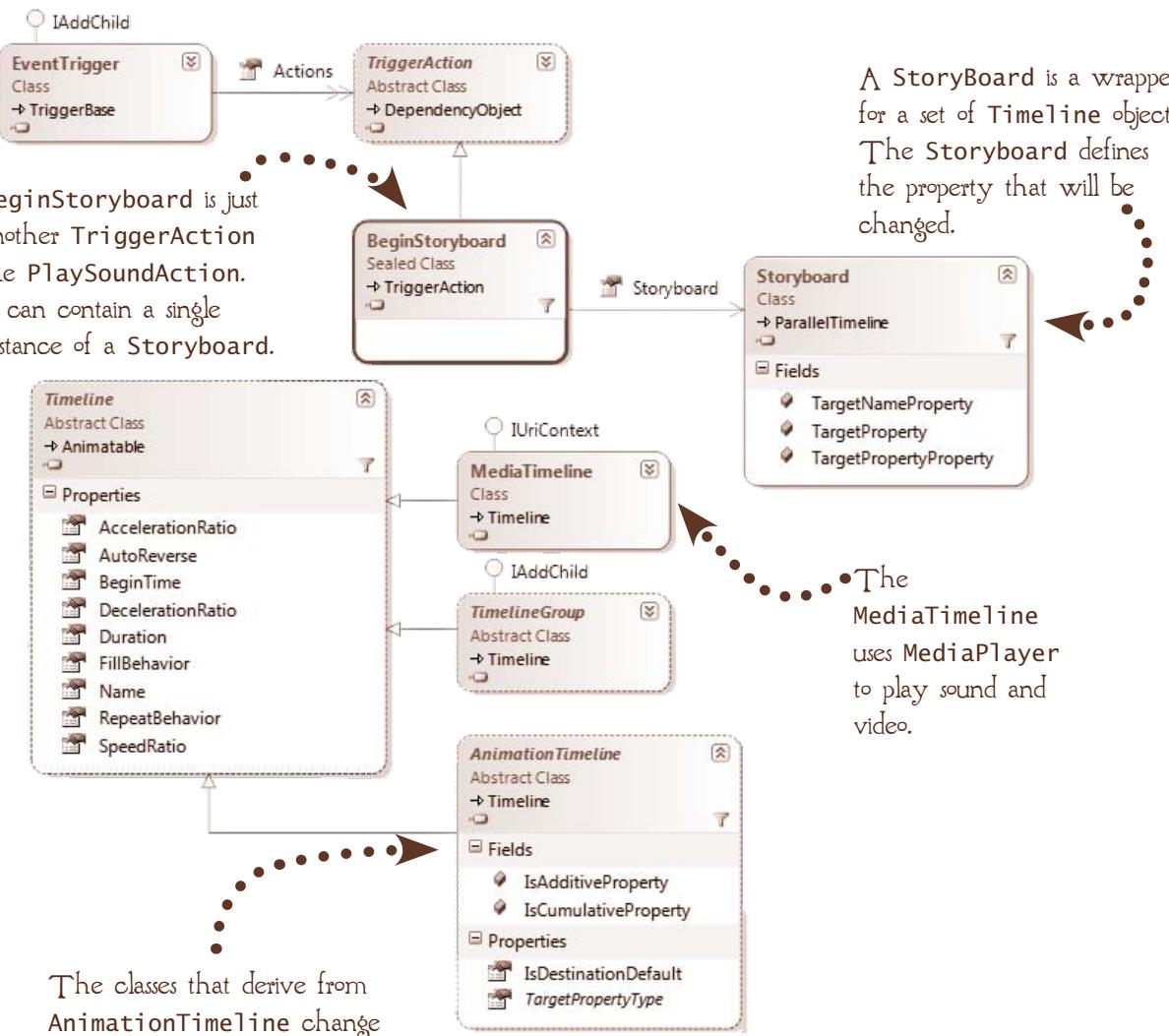
ANIMATION CAN SIMPLIFY PROGRAMMING

You can always change properties directly instead of animating them, but WPF storyboards make some changes, particularly ones involving a change of size and position, much, much easier. Wouldn't you rather say "make this half-again as big" than write the code to calculate "how big am I times 1.5?" and maintain that state between calls? Yeah, me too.



ELEMENTS OF AN ANIMATION

Like most things, you create a WPF animation declaratively, which is a real bonus for those of us who don't really enjoy linear algebra. (Personally, I'd rather clean the oven.) The catch is that there are quite a few objects involved. We'll take them one at a time.

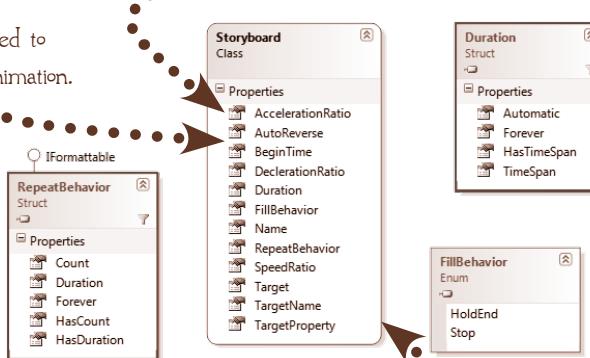


STORYBOARDS

You can, of course, use the WPF animation objects in code, but whenever you want to perform an animation in XAML, you must wrap the Timeline objects inside a Storyboard. Target, TargetName and TargetProperty are attached properties; you can specify them for the Storyboard as a whole, or for individual Timelines. (We'll discuss those next.)

The AccelerationRatio and DeclarationRatio allow you to define an animation that is a bit slower at the beginning or end. They're defined as a Double that represents the percentage of the Duration.

BeginTime can be used to delay the start of the animation.



In XAML, the Count property converter requires the character "x", so, for example, to specify two completes, you would set RepeatBehavior="2x"

The dependency property to be animated.

TimeSpan values are specified as Days.Hours:Minutes:Seconds. Fractions.



PUT ON YOUR THINKING HAT

Can you write the following declarations? If you want to try out your code snippets, remember that they usually live inside an event trigger, which in turn usually lives inside a style.

A Storyboard that targets the Background property of the element that is being styled.

A Storyboard that targets the Opacity property of a Label with the x:Name set to MyLabel.

A Storyboard that targets the Background property of the element being styled, and repeats three times over 3.5 seconds.

A Storyboard that targets the Width property of the element being styled, lasts 0.5 seconds, and then returns to the original value.

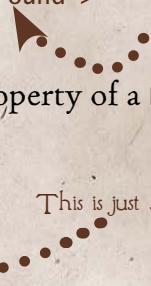


HOW'D YOU DO?

I didn't give you much to go on this time, so don't worry if you got some of these wrong or not-quite-right.

A Storyboard that targets the `Background` property of the element that is being styled:

```
<Storyboard TargetProperty = "Background">
```



You don't need to specify the `TargetName` or `Target` if the `Storyboard` applies to the object being styled.

A Storyboard that targets the `Opacity` property of a `Label` with the `x:Name` set to `MyLabel`:

```
<Storyboard TargetName = "MyLabel"  
TargetProperty = "Opacity" >
```

This is just standard XAML.



A Storyboard that targets the `Background` property of the element being styled and repeats three times:

```
<Storyboard TargetProperty="Background"  
RepeatBehavior="3x">
```

Did you remember
the "x"?



A Storyboard that targets the `Width` property of the element being styled, lasts 0.5 seconds, and then returns to the original value:

```
<Storyboard TargetProperty="Width"  
Duration="0:0:0.5" AutoReverse="True" >
```



`Duration` can be set to any of the values of the `Duration` structure, but is most often set to a `TimeSpan` in the format

`Days.Hours:Minutes:Seconds.Fractions`.

For example, a duration of 1 second (the default) would be specified as:

`Duration = "0:1.0"`

Be careful not to omit the value before the colon. A `DURATION` of `1.0` would be 1 day, not 1 second!

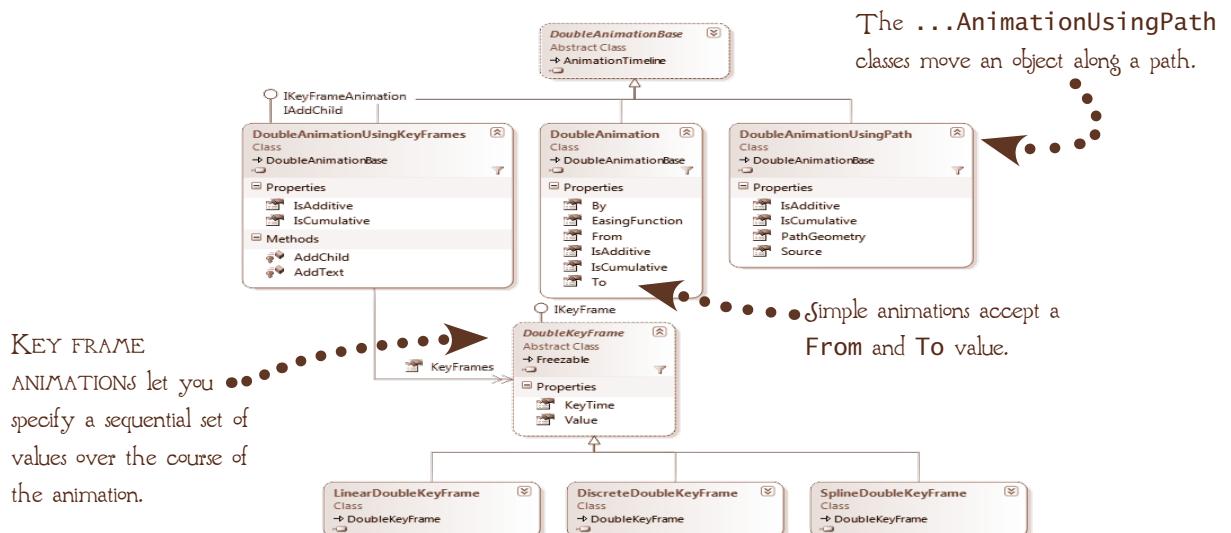
Did you find `AutoReverse`? You might have thought you'd use `FillBehavior` to control this, but `FillBehavior` controls whether or not you can change the property after the animation ends. `FillBehavior.HoldEnd` (the default) keeps applying the end value. If you want to be able to change the value, set `FillBehavior` to `Stop`.

TIMELINE CLASSES

A Storyboard lets you specify the dependency property to be animated, but you still need to specify how the property value will change. You do that using the descendants of AnimationTimeline in the System.Windows.Media.Animation namespace. The various classes let you specify discrete values over the animation, like the starting and ending values or positions along a path, and they'll calculate the intermediate values for you. These calculations can be tricky (did I mention linear algebra?), and they're unique to the type of value being calculated, so the namespace contains a set of classes for each of the following data types:

- Boolean
 - Int32
 - Object
 - Point
 - Byte
 - Int64
 - Thickness
 - Point3D
 - Char
 - Single
 - Color
 - Matrix
 - Decimal
 - Double
 - Size
 - Quaternion
 - Int16
 - String
 - Rect
- Quaternions represent rotation in 3D.

Because there are separate classes for every data type, the sheer number of them can be a little scary. (There are over 150 individual animation classes in the namespace.) But there are actually only a few types of operations, and once you know what they are, the classes sort themselves out.

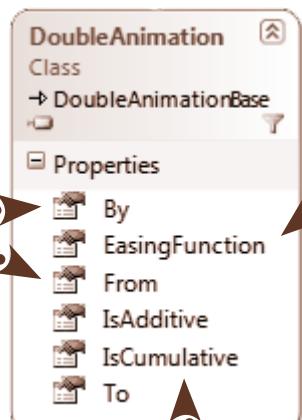


AN EXAMPLE

It's surprising, really, how many dependency properties are defined as doubles, so let's look at a simple `DoubleAnimation` example, just to get a sense of what you can do (and, of course, how to do it.)

By lets you specify a relative amount to be added (or subtracted, if it's a negative number) from the current value.

From and To let you set absolute start and end values for the property.



Remember that `DoubleAnimation` inherits the properties of `Timeline`, so it also exposes properties like `Duration` and `AccelerationRatio`.

You can think of an easing function as a more sophisticated version of the `AccelerationRatio` and `DecelerationRatio` that lets you do things like make an animation bounce.

`IsAdditive` and `IsCumulative` determine whether the animation starts at the "native" value of the property or from its current value after any previous animations have been applied.

GOING FURTHER

In truth, WPF animations have so many knobs and buttons to tweak that they can be difficult to code from scratch, and you'll need to do a lot of experimentation for anything very tricky. (If you're going to be doing a lot of animation, you might want to explore Microsoft Expression Blend, which has wonderful interactive tools for defining all kinds of sophisticated storyboards.)



PUT ON YOUR THINKING HAT

Here's a pair of simple animations that change the size of a button when the mouse enters and leaves it. Create a new project with a few buttons inside a `StackPanel`, and add the style to the `<Resources>` element of the `app.xaml` file, and then answer the questions below:

```
<Style TargetType="Button">
  <Style.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation Storyboard.TargetProperty="FontSize" To="36"/>
          <DoubleAnimation Storyboard.TargetProperty="Height" To="75"/>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.MouseLeave">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation Storyboard.TargetProperty="FontSize" />
          <DoubleAnimation Storyboard.TargetProperty="Height" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  <Style.Triggers>
</Style>
```

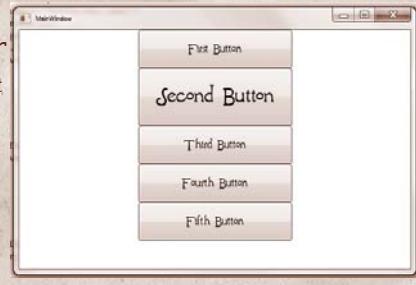
By omitting the `From` property, the animation starts with the current value of the property.

You can use the attached `TargetProperty` to specify a different target for each `Timeline`.



By omitting both `From` and `To`, the animation returns the property to its original value.

- The animation won't work unless you explicitly set the `Height` property of the buttons. Can you figure out why?
- What happens if you change the `Height` property of the `MouseEnter` animation to `By="25"` and then re-enter the button before it finishes returning to its original size? Can you think of situations in which this might be the behavior you want?
- Try adding an easing function to the `Height` animation of the `MouseEnter` `Storyboard`. The Framework defines several. (Intellisense will help you out here.) Which one do you think is most appropriate in this situation?
- When you set `TargetProperty` to an attached property, you need to wrap the property in parentheses: `Foreground. (SolidColorBrush. Color)`. Using this syntax, try adding a `ColorAnimation` that changes the text color.





HOW'D YOU DO?

There was a lot for you to figure out this time, wasn't there? In addition to giving you a chance to build some animations, this exercise gave you a taste of something programmers do all the time: Start with some example code that doesn't quite do what you want and change it into what you need, figuring out problems as you run across them.

The animation won't work unless you explicitly set the `Height` property of the buttons. Can you figure out why?

Don't feel bad if you didn't figure this one out. Remember when we talked about how WPF doesn't create storage for properties unless you explicitly set their value? That's what's going on here. If you don't explicitly set the `Width` property, it doesn't have a value, which WPF reports as `NaN` (short for "not a number"), and `DoubleAnimation` doesn't know what to do with something that isn't a number. You need to be careful that the properties you're trying to animate actually exist on the target element. One way to do that is to include them in the element template. We'll see how to do that in the next chapter.

What happens if you change the `Height` property of the `MouseEnter` animation to `By="25"` and then re-enter the button before it finishes returning to its original size? Can you think of situations in which this might be the behavior you want?

Because we're not setting the `From` property, the animation always starts from the current value. The original version of the animation specified an absolute value in the `To` property, and that's the maximum value for the animation. Setting `By` instead just adds `25` to whatever the current value is, so the button gets bigger and bigger and bigger.

This behavior is inappropriate for our example, but it isn't always. What if you wanted a user to be able to set the relative importance of a set of elements by clicking them? Making the element bigger each time it was clicked would be exactly what you wanted.

Add an easing function to the Height animation of the MouseEnter Storyboard. Which one do you think is most appropriate in this situation?

```
<DoubleAnimation Storyboard.TargetProperty="Height" To="75" >  
  <DoubleAnimation.EasingFunction>  
    <BackEase />  
  </DoubleAnimation.EasingFunction>  
</DoubleAnimation>
```



Each of the easing functions exposes some properties to control them, but they also work pretty well in their default configuration.

Aren't easing functions fun? I think the **BackEase** is best here, but that's a judgement call, and you might have preferred a different one.

When you set TargetProperty to an attached property, you need to wrap the property in parentheses: `(Foreground).(SolidColorBrush.Color)`. Using this syntax, try adding a **ColorAnimation** that changes the text color.

```
<ColorAnimation Storyboard.TargetProperty="Foreground.(SolidColorBrush.Color)"  
  To="Green" />
```

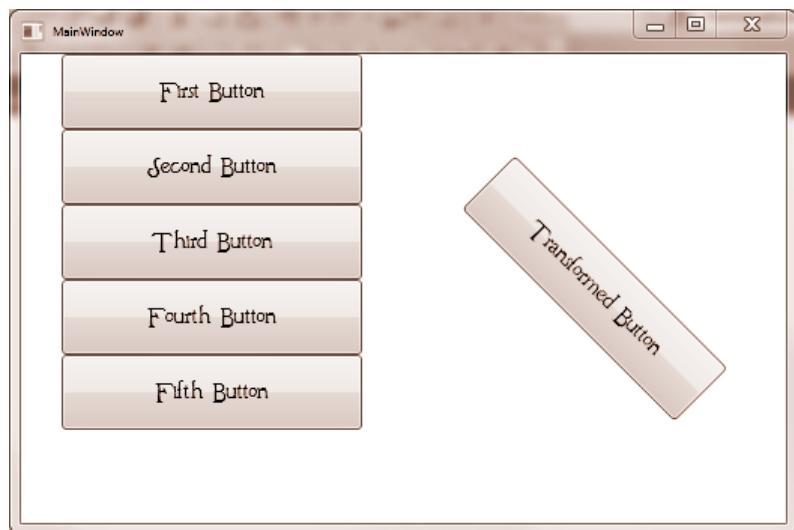
In this example, `(Foreground).(SolidColorBrush.Color)` works, but `(Foreground).(SolidColorBrush).(Color)` doesn't, because `Color` isn't an attached property. Sometimes it just takes a little trial and error to work out the exact syntax.

TRANSFORMATIONS

Even if your application doesn't involve wascally wabbits, you'll often want to move elements as part of an animation. As you've seen, you can change the `Width` and `Height` properties using the `DoubleAnimation`, but that only lets you change the size; it doesn't let you do things like rotate or skew elements. Also, changes to `Width` and `Height` always cause their container to rearrange its children, and that might not be what you want. The solution is the transform classes in the `System.Windows.Media` namespace. There are four basic transforms:

ROTATETRANSFORM

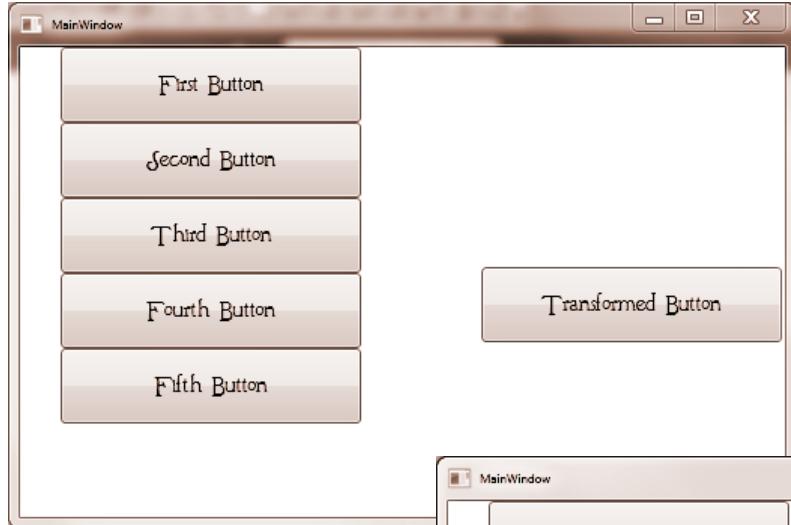
Rotates an element around a center point you can specify.



MAKE A NOTE



WPF also has a `MatrixTransform` class that lets you specify any combination of these four transforms as a 3×3 affine transformation, for those of you who like that kind of thing. (Get back to me after I finish cleaning the oven.)



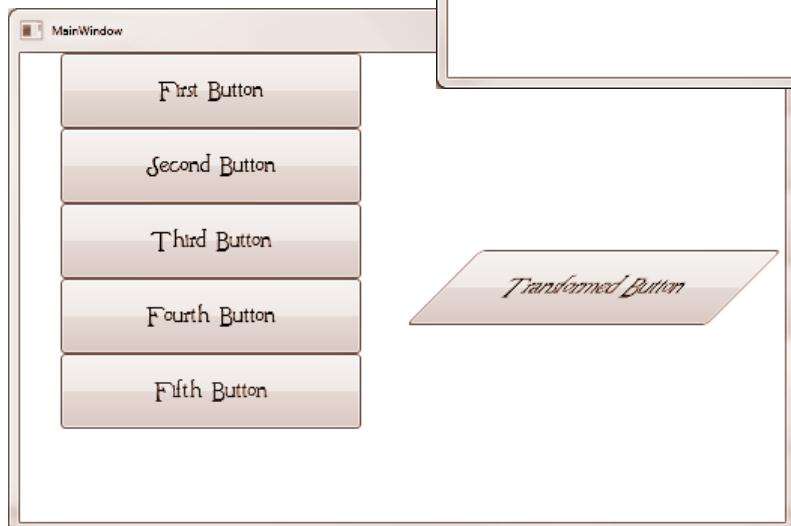
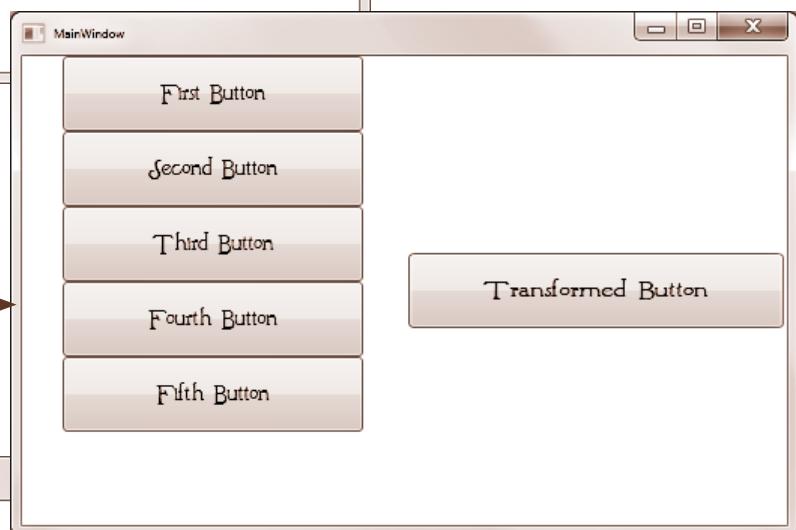
TRANSLATETRANSFORM

Moves an element horizontally or vertically.



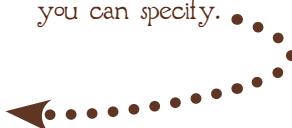
SCALETRANSFORM

Changes the size of an element.



SKEWTRANSFORM

Skews the element horizontally or vertically around a center point you can specify.

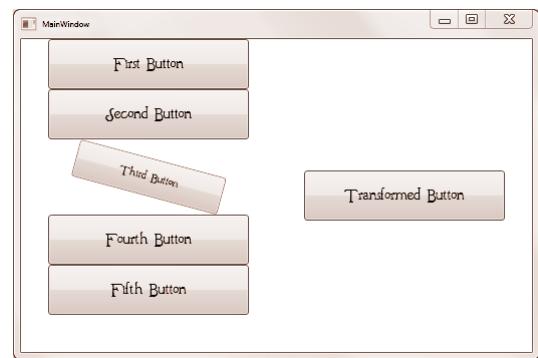


APPLYING TRANSFORMS

The `UIElement` defines two properties, `LayoutTransform` and `RenderTransform`, that you can use to transform your elements. You can set these properties to a `LayoutGroup` in order to apply multiple transforms.

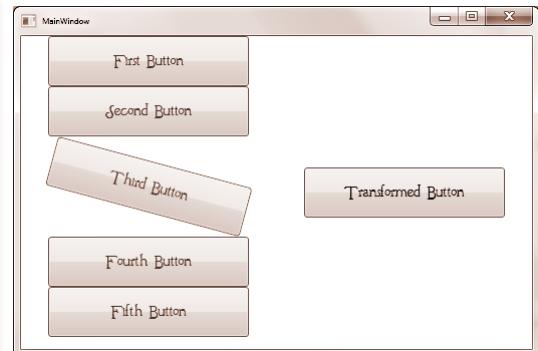
LAYOUT TRANSFORMS are applied before the panel arranges its children, so other elements move out of the way.

```
<Button>
    Third Button
    <Button.LayoutTransform>
        <RotateTransform ... />
    </Button.LayoutTransform>
</Button>
```



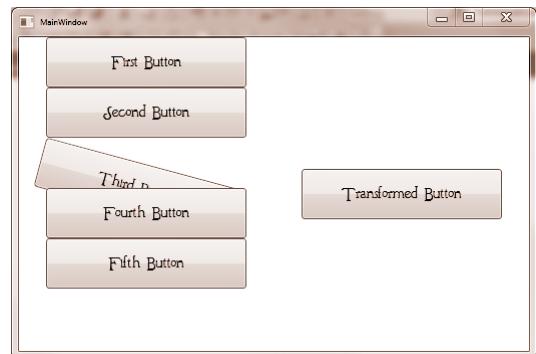
TRANSFORM GROUPS allow you to apply multiple transforms to an object.

```
<Button Width="200" Height="50">
    Third Button
    <Button.LayoutTransform>
        <TransformGroup>
            <RotateTransform CenterX="1" CenterY="1"
                Angle="15" />
            <ScaleTransform CenterX="1" CenterY="1"
                ScaleX="0.75" ScaleY="0.75" />
        </TransformGroup>
    </Button.LayoutTransform>
</Button>
```



RENDER TRANSFORMS are applied after the panel arranges its children, so the transformed element might overlap or be hidden by other elements.

```
<Button>
    Third Button
    <Button.RenderTransform>
        <RotateTransform ... />
    </Button.RenderTransform>
</Button>
```



PUT ON YOUR THINKING HAT



Add a second button to the form as shown in the examples, and make it wiggle when the mouse is over it. Here are some tips to get you started:

- Remember the NaN problem? In order to avoid it when you're animating the transform, give the button an empty transform when you declare it:

```
<RotateTransform />
```

You can also set the properties that will be shared by all the transformations in the animation:

```
<RotateTransform CenterX="100" CenterY="25"/>
```

- You don't want all the buttons to wiggle, so create a new button style with a key, and assign that style to the button as a **StaticResource**.
- You can set the **TargetProperty** in the Storyboard itself. You're going to be animating the **Angle** property, which is actually an attached property from **UIElement** where it's declared:

```
TargetProperty="(UIElement.RenderTransform).(RotateTransform.Angle)"
```

- You'll actually need four **DoubleAnimation** instances to wiggle the button. Rotate in one direction (I changed the angle 15 degrees, but do what seems right to you), return to the original (untransformed) values, and then rotate in the other direction and return.
- You'll need to adjust the **BeginTime** and **Duration** properties so that the animations run sequentially:

```
<DoubleAnimation Duration="0:0:0.25" ... />
```

```
<DoubleAnimation BeginTime="0:0:0.25" Duration="0:0:0.25" />
```

This animation begins after the previous one ends.





HOW'D YOU DO?

You should be proud of yourself if you got this one working without peeking. You're almost ready to start sending out resumes!

Here's my version of the Button element.

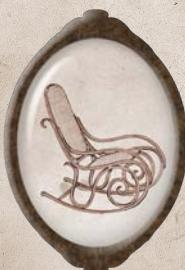
```
<Button Grid.Column="1" Width="200" Height="50"  
Style="{StaticResource Wiggle}">  
    Transformed Button ←•••••••••••••••••••••  
    <Button.RenderTransform>  
        ➤ <RotateTransform CenterX="100" CenterY="25"/>  
    </Button.RenderTransform>  
</Button>
```

Setting the default properties of the transformation saves some code in the event trigger and also ensures that WPF creates the `RenderTransform` property.

This is the button content.

Do you think it would have been clearer if I'd specified that explicitly?

```
<Button.Content>  
    Transformed Button  
</Button.Content>
```



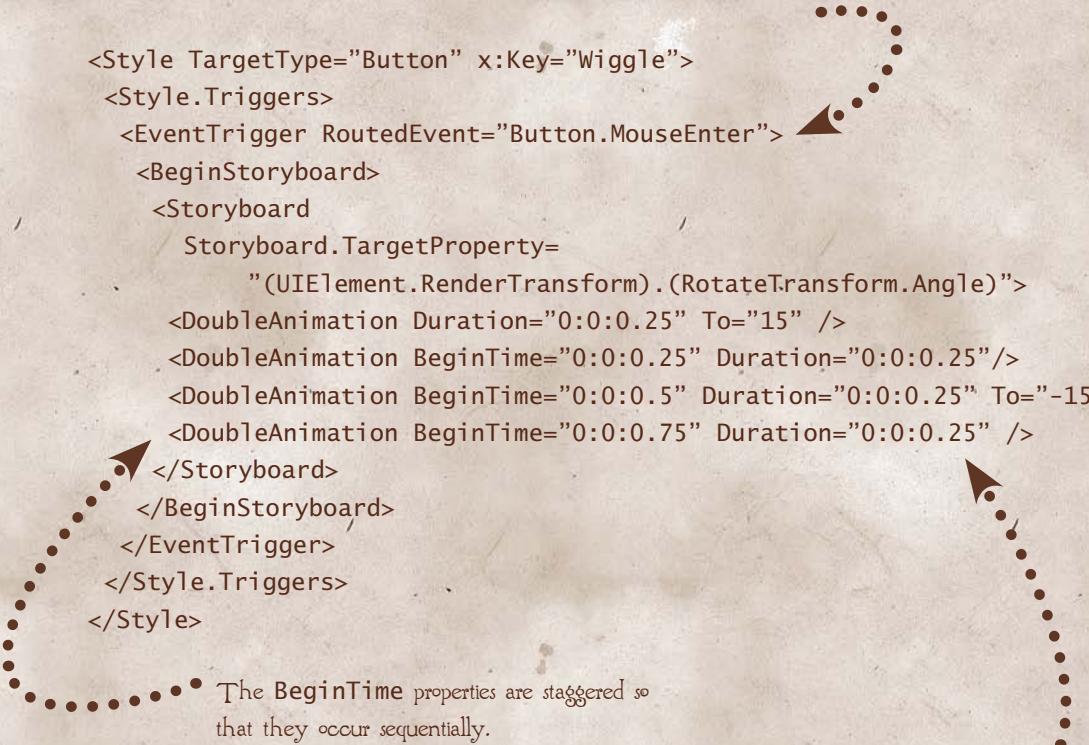
TAKE A BREAK

That was a difficult exercise, and it probably took you some time to work out. So why don't you take a break now before you complete the Review and we move on to completely redefining the appearance of a control using templates in the next chapter?

And the animation style:

```
<Style TargetType="Button" x:Key="Wiggle">
  <Style.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <BeginStoryboard>
        <Storyboard
          Storyboard.TargetProperty=
            "(UIElement.RenderTransform).(RotateTransform.Angle)">
          <DoubleAnimation Duration="0:0:0.25" To="15" />
          <DoubleAnimation BeginTime="0:0:0.25" Duration="0:0:0.25"/>
          <DoubleAnimation BeginTime="0:0:0.5" Duration="0:0:0.25" To="-15" />
          <DoubleAnimation BeginTime="0:0:0.75" Duration="0:0:0.25" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Style.Triggers>
</Style>
```

We know the `RotateTransform` property exists because it was declared in the XAML.



The `BeginTime` properties are staggered so that they occur sequentially.

You might have made the durations longer or shorter. That doesn't matter, as long as your `BeginTime` properties reflect the actual duration you chose.



REVIEW

There are three places you'd commonly define resources. What are they?

What are the two ways to reference a resource? When would you use each?

What's the difference between these two declarations:

```
<Style x:Key="MyStyle" TargetType="Button" >  
<Style TargetType="Button" >
```

What two collections allow you to put an `EventAction` inside a property trigger?

What logical operator combines the `Condition` elements in a `MultiTrigger`?

Why are there so many classes in the `System.Windows.Media.Animation` namespace?

What happens if you don't specify `By`, `To` or `From` in an animation timeline?

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



TEMPLATES



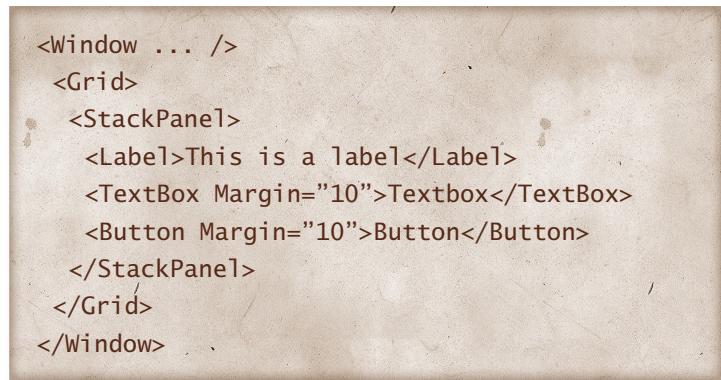
In the last chapter we saw how you can use resources, styles and triggers to simplify defining the appearance and visual behavior of a WPF control, but the basic structure of the elements remained the same—we changed the size and color of a button and made it move around, but it still looked like a button. In this chapter, we'll take control of that structure as well by using WPF templates to redefine the visual tree of any `FrameworkElement` or `FrameworkContentElement`.

Want your buttons to be triangular? No problem. Does your application really require an oval window? You can make that happen using templates.

We'll start by examining some fundamental elements you can use to build your new versions of controls, and then we'll look at the very simple process of redefining the control's appearance. We'll also explore the technique for making your new template reflect the control properties you set at design- and runtime. Finally, we'll look at a new way of specifying visual behavior, similar to a trigger but available only within a control template.

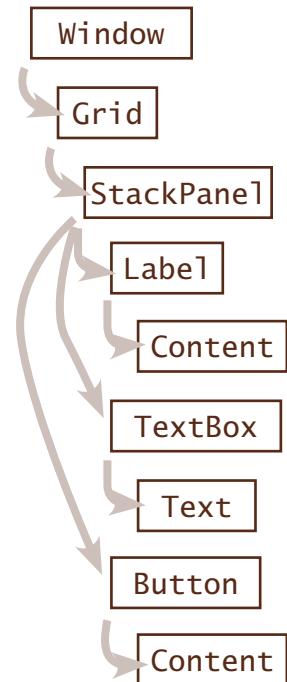
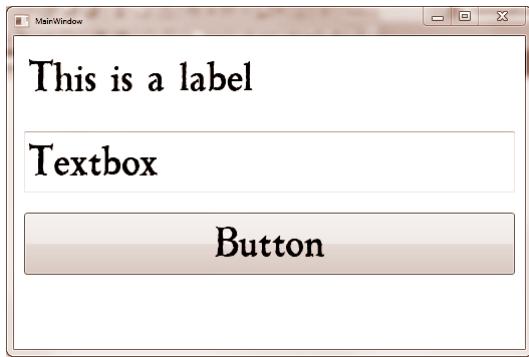
FITTING IT IN

Throughout the book, you've been building applications by defining a logical tree for the main window of the application.

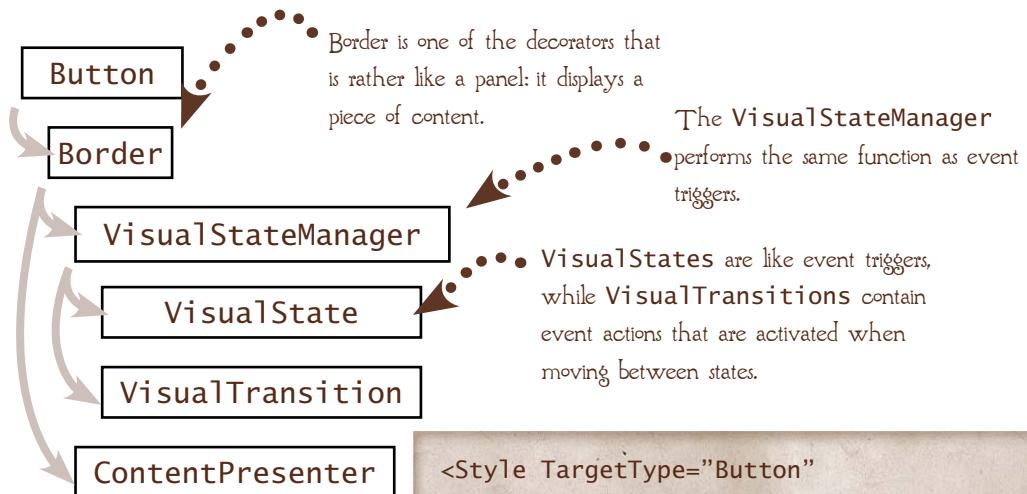


This XAML creates
this Window...

...from this logical tree.



But every element also has a visual tree that defines how the element itself is composed. The visual tree is in a XAML control template, which is defined as a resource or as part of the control's style.



PRESENTERS display the content of a control. They have their own template (called a ContentTemplate). There are templates throughout WPF.

```
<Style TargetType="Button">
  ...
  <Setter Property="Template">
    <ControlTemplate TargetType="Button">
      <Border ...>
        <VisualStateManager>
          <VisualStateGroup x:Name="CommonStates">
            <VisualState ...>
            <VisualTransition ...>
          </VisualStateGroup>
        </VisualStateManager>
        <ContentPresenter ... >
      </Border>
    </ControlTemplate>
  </Setter>
  ...
</Style>
```

TASK LIST

In this chapter we'll trace the entire process of redefining the visual tree and visual behavior of a WPF control, starting with some of the low-level visual components you can use to build up the control's templates.



BUILDING CONTROLS

You can use any WPF element inside a control template, but you'll often want to use fairly low-level components—you'll want a rectangle or a frame, not a `TextBox`. We haven't had a reason to explore these kinds of graphics, but we do now, so we'll start the chapter by taking a quick look at the WPF elements that expose this basic drawing functionality.



CONTROL TEMPLATES

Once we've worked out how to use the WPF drawing elements, we'll be ready to use them to build our control template, so we'll explore that next, along with how to find (and name) the control parts that have to be present for a control to function properly. We'll also examine a couple of special elements: the `ContentPresenter` that, well, presents the content of the control, and the `TemplateBinding` that links your template to the control's public properties.



THE VISUALSTATEMANAGER

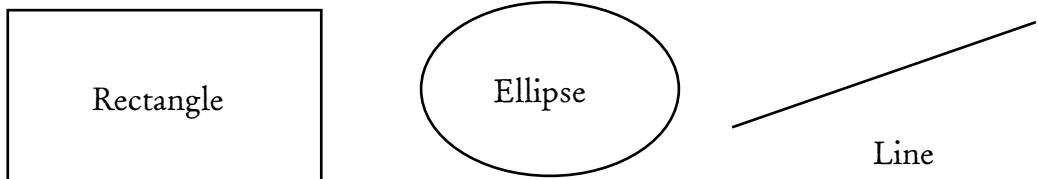
Control templates support a mechanism for defining a control's visual behavior that is similar to triggers but both more powerful and a little simpler to use (once you get the hang of it). The `VisualStateManager` is supported by the .NET 4.0 control contract, which also makes it a bit more robust. In this section, we'll learn how to support visual states, and we'll finally fix that button mouseover that didn't quite work in the last chapter.



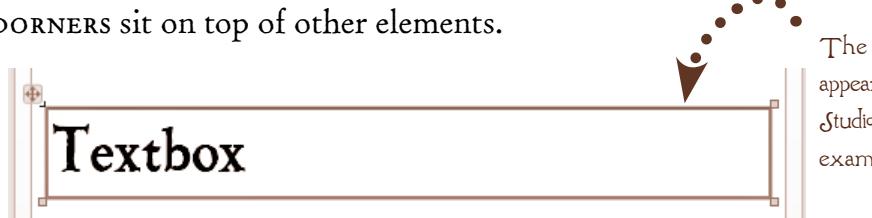
BUILDING CONTROLS

Like the logical tree of a `Window`, a control template can contain just about any Framework element, but you'll usually want to work at a lower visual level. You want "a rectangle with a dotted border", not a `TextBox`. WPF has three categories of elements that work at this level: **Shapes**, **Adorners** and **Decorators**. We've actually used some of these shapes in earlier chapters, but we never really explored them, so let's do that now.

SHAPES are exactly what you'd expect; they're basic geometric objects.



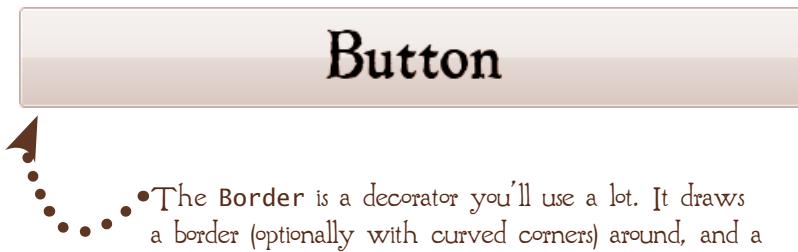
ADORNERS sit on top of other elements.



The bounding box that appears in the Visual Studio Designer is an example of an adorner.

We won't examine **Adorners** in this chapter, but we'll see an example in the next chapter.

DECORATORS display a single object in a particular way.

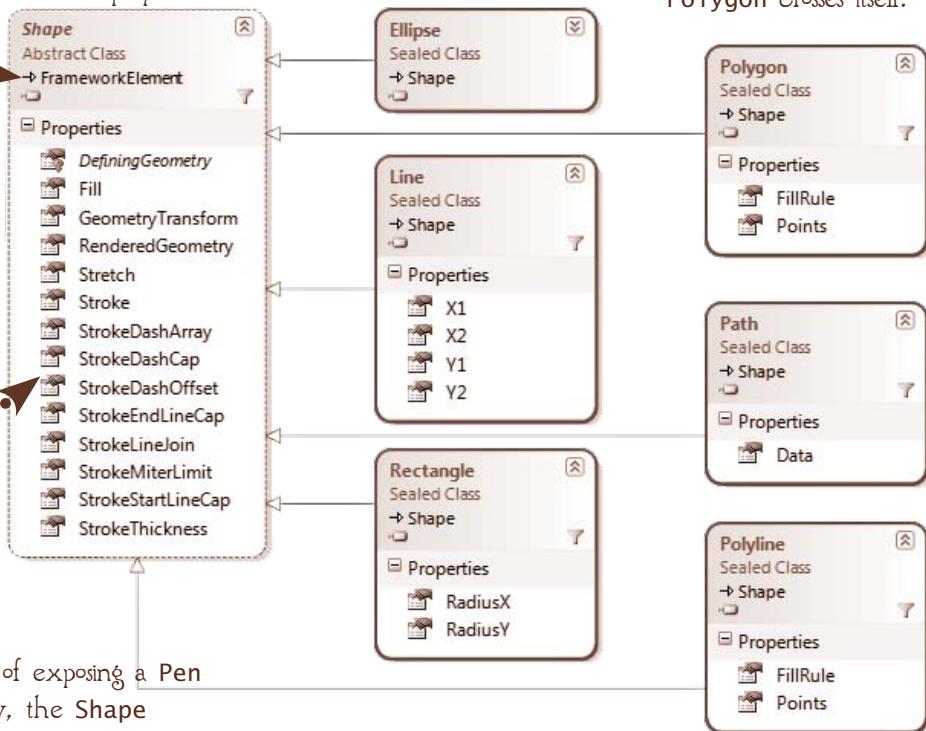


The **Border** is a decorator you'll use a lot. It draws a border (optionally with curved corners) around, and a background behind some content. The default template for a **Button** is just a **Border** and a content presenter.

DRAWING SHAPES

The `Shape` abstract class and its concrete descendants render a 2-dimensional drawing in the user interface. The drawing itself is defined using an even lower-level set of classes, `Geometry` and its descendants, that describe the shape mathematically.

Because `Shape` is a `FrameworkElement`, it inherits `Width`, `Height`, `RenderTransform` & `LayoutTransform` properties.



Instead of exposing a `Pen` property, the `Shape` exposes each of the pen properties individually. This might seem tedious, but in practice it's more efficient.

A `FillRule` defines what happens when the path that defines a `Polyline` or `Polygon` crosses itself.

The `Points` collection of a `Polyline` is specified in `XAML` as

`"x,y x,y x,y ..."`

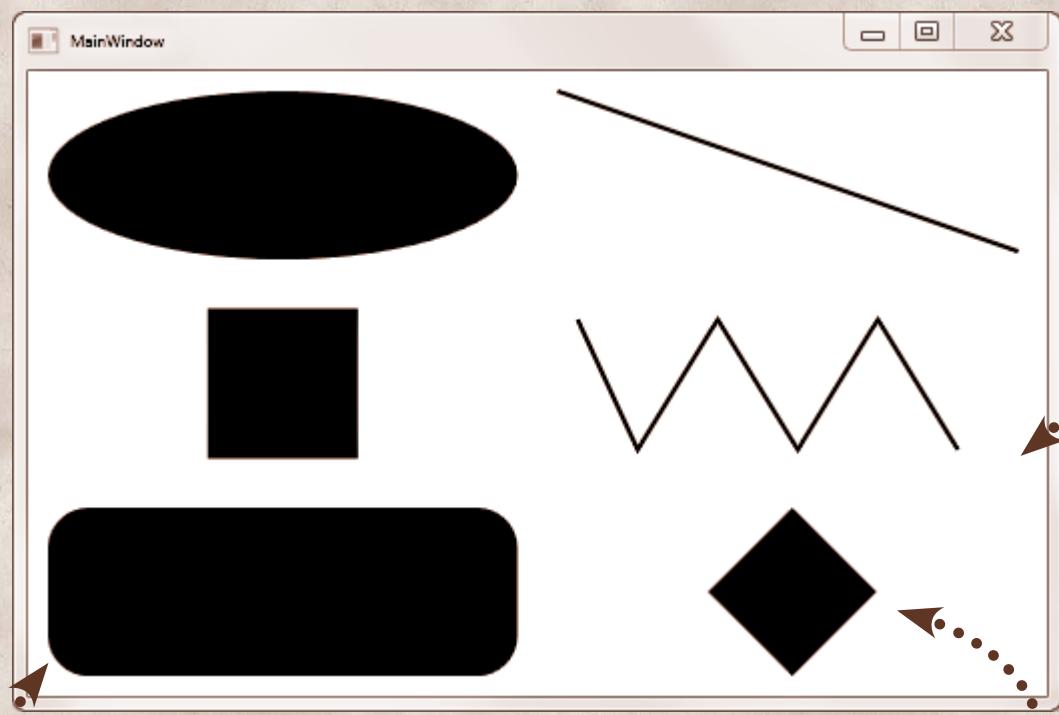


PUT ON YOUR THINKING HAT

Here's a Window that displays several shapes in a Grid. Can you duplicate it?

The point set I used here was

20,20 50,85 90,20 130,85 170,20 210,85



• Use the RadiusX and RadiusY properties of the Rectangle to create this. I used values of 20.

This is a Rectangle with a LayoutTransform.



HOW'D YOU DO?

Here's my version:

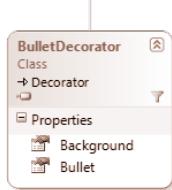
```
<Window ... >
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>

  <Ellipse Grid.Row="0" Grid.Column="0" Margin="10" Fill="Black"/>
  <Rectangle Grid.Row="1" Grid.Column="0" HorizontalAlignment="Center" Width="75"
    Height="75" Fill="Black" />
  <Rectangle Grid.Row="2" Grid.Column="0" Margin="10" Fill="Black"
    RadiusX="20" RadiusY="20" />
  <Line Grid.Row="0" Grid.Column="1" X1="10" Y1="10" X2="240" Y2="90"
    StrokeThickness="2" Stroke="Black" />
  <Polyline Grid.Row="1" Grid.Column="1" Stroke="Black" StrokeThickness="2"
    Points="20,20 50,85 90,20 130,85 170,20 210,85" />
  <Rectangle Grid.Row="2" Grid.Column="1" Margin="10" Fill="Black">
    <Rectangle.LayoutTransform>
      <RotateTransform CenterX="200" CenterY="200" Angle="45" />
    </Rectangle.LayoutTransform>
  </Rectangle>
</Grid>
</Window>
```

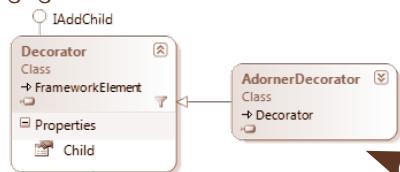
DECORATORS

We're not going to examine Adorners in this chapter (they work a lot like Decorators), but the **Decorator** classes, particularly **Border**, are used in almost every control template. You can think of a **Decorator** as halfway between a content control and a panel. Like a content control, a **Decorator** contains a single element in its **Child** property. Like a panel, the purpose of the **Decorator** is to display its content. But while panels specialize in arranging their content, a decorator manipulates it in some way.

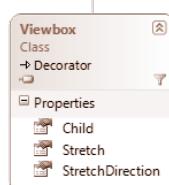
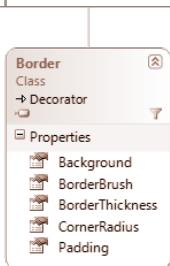
A
BulletDecorator
displays an image next
to its content.



InkPresenter is used to capture mouse
or stylus movements as paths, in, for
example, a drawing program.



The
AdornerDecorator
hosts Adorners.



The **Viewbox** decorator
works exactly like
the **Viewbox** of a
TileBrush.

PUT ON YOUR THINKING HAT

Can you add a third column to our example that contains decorators?



This is a **BulletDecorator** with
its **Background** set. The **Image** is
specified using property syntax.

This is a **Border** with a grey
Background and a black
BorderBrush.

This is a **Viewbox** with its
Stretch property set to Fill.



HOW'D YOU DO?

Your image will be different, of course, but here's my version:

```
<Window ...>
<Grid>
    ...
    <BulletDecorator Grid.Row="0" Grid.Column="2"
        Background="Gainsboro" Margin="10">
        <BulletDecorator.Bullet>
            <Image Source="Thinking Hat.tif" Width="50"/>
        </BulletDecorator.Bullet>
        <Rectangle Margin="10" Fill="Black"/>
    </BulletDecorator>

    <Border Grid.Row="1" Grid.Column="2" Background="Gainsboro" Margin="10"
        BorderBrush="Black" BorderThickness="10">
        <Ellipse Width="50" Height="50" Fill="Black"/>
    </Border>

    <Viewbox Grid.Row="2" Grid.Column="2" Stretch="Fill">
        <Image Source="Thinking Hat.tif" />
    </Viewbox>

    </Grid>
</Window>
```



TAKE A BREAK

Compared to some of the WPF elements we've used, the shape and decorator classes are pretty easy to use. What they do is straight-forward, and there aren't any tricky properties or structures to worry about. So why don't you take a break before we move on to building control templates with these and other objects?



CONTROL TEMPLATES

Now that we've seen some of the most common elements you'll use to define the visual tree of your control, let's look at the syntax for actually doing so using a control template.

Control templates can be defined as a resource or (more commonly) as part of a style. The syntax probably won't come as a great surprise to you.

You can define the control template of an individual element:

```
<Button>
  <Button.Template>
    <ControlTemplate TargetType="Button">
      <ControlTemplate.Triggers>
        ... property and event triggers ...
      </ControlTemplate.Triggers>

      ... Visual Tree ...

      <VisualStateManager.VisualStateGroups>
        ... Visual States & Transitions ....
      </VisualStateManager.VisualStateGroup>

    </ControlTemplate>
  </Button.Template>
</Button>
```

- Like a content control, a control template must have a single child element, which can in turn contain other elements.

- You must specify the target type, even within an element definition.

- Templates have a Triggers collection, just like a Style.

- We'll talk about the VisualStateManager later in this chapter.

But it's usually done inside a Style so it's available to multiple elements:

The control template is set inside a normal property setter.

You always need to specify the TargetType.

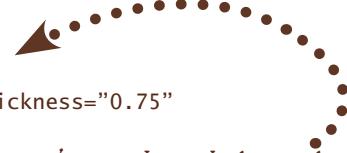
```
<Style TargetType="Button">
  ...
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="Button">
        ...
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

AN EXAMPLE



To give you a taste for how control templates work, here's a moderately complex example of a template that creates a round button with a shiny highlight. If you look at the XAML, you'll see that the most complex part of the definition is the three gradient brushes. The rest is just a Border that contains a Grid with two ellipses. I don't know about you, but I think that's a pretty cool result for very little effort.

```
<Button>
  <Button.Template>
    <ControlTemplate TargetType="{x:Type Button}">
      <ControlTemplate.Resources>
        <LinearGradientBrush x:Key="BackgroundGradient" EndPoint="0.5,1" StartPoint="0.5,0">
          <GradientStop Color="#FF000000" Offset="0.483"/>
          <GradientStop Color="#FFFFFF" Offset="1"/>
          <GradientStop Color="#FFE0E0E0" Offset="0" />
        </LinearGradientBrush>
        <RadialGradientBrush x:Key="CenterGradient" GradientOrigin="0.5,0.5">
          <GradientStop Color="#FF3F369F" Offset="1"/>
          <GradientStop Color="#FF1E1A49" Offset="0"/>
        </RadialGradientBrush>
        <LinearGradientBrush x:Key="HighlightGradient" EndPoint="0.5,1" StartPoint="0.5,0">
          <GradientStop Offset="1"/>
          <GradientStop Color="#FFFFFF" Offset="0"/>
          <GradientStop Color="#52FFFFFF" Offset="0.409"/>
        </LinearGradientBrush>
      </ControlTemplate.Resources>
      <Border x:Name="MetallicBorder" CornerRadius="100" Margin="5"
        Background="{StaticResource BackgroundGradient}" BorderThickness="0.75"
        BorderBrush="Black">
        <Grid>
          <Ellipse x:Name="ButtonCenter" Margin="15"
            Fill="{StaticResource CenterGradient}" />
          <Ellipse x:Name="ButtonHighlight" Margin="35,25,35,100"
            Fill="{StaticResource HighlightGradient}" Opacity="0.75"/>
        </Grid>
      </Border>
    </ControlTemplate>
  </Button.Template>
</Button>
```



I've used an absolute value for margin here. That's not a good idea when you don't know how big the button would be. A better approach would be to use columns with relative widths, but it complicates the code and isn't necessary in our example.

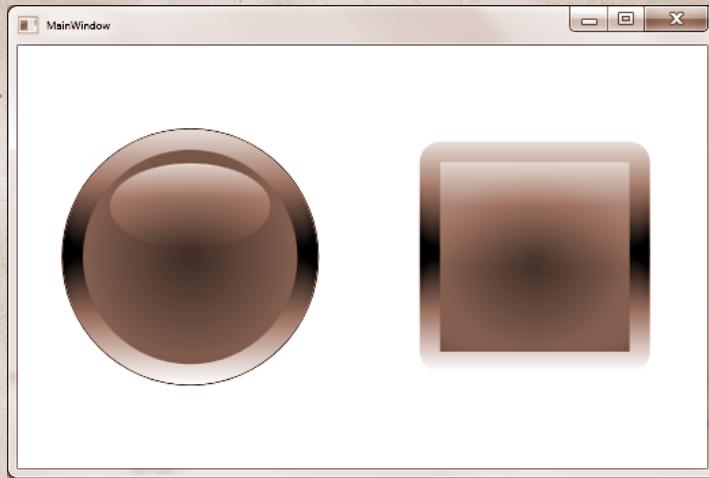


PUT ON YOUR THINKING HAT

Now it's your turn...



Create a new project to contain the sample Button and rectangular version. You'll want to move the brush definitions from the Resources section of the control template to `Window.Resources` so that you can reference them from both button templates.



Try changing the `BorderBrush` color of one of the buttons (the button itself, not the button's control template). What happens?



HOW'D YOU DO?

It doesn't matter if your button is a little different from mine, as long as it works and you're happy with it. (Oh, and changing the BorderBrush? It does nothing. Keep reading to find out why.)

```
<Window.Resources>
  <LinearGradientBrush x:Key="BackgroundGradient" EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Color="#FF000000" Offset="0.483"/>
    <GradientStop Color="#FFFFFF" Offset="1"/>
    <GradientStop Color="#FFE0E0E0" Offset="0" />
  </LinearGradientBrush>
  <RadialGradientBrush x:Key="CenterGradient" GradientOrigin="0.5,0.5">
    <GradientStop Color="#FF3F369F" Offset="1"/>
    <GradientStop Color="#FF1E1A49" Offset="0"/>
  </RadialGradientBrush>
  <LinearGradientBrush x:Key="HighlightGradient" EndPoint="0.5,1" StartPoint="0.5,0">
    <GradientStop Offset="1"/>
    <GradientStop Color="#FFFFFF" Offset="0"/>
    <GradientStop Color="#52FFFFFF" Offset="0.409"/>
  </LinearGradientBrush>
</Window.Resources>
<Grid>
  <Grid.ColumnDefinitions ... >
  <Button ... >
    <Button Grid.Column="1" Width="200" Height="200" VerticalAlignment="Center"
           HorizontalAlignment="Center">
      <Button.Template>
        <ControlTemplate>
          <Border x:Name="RoundedRectBorder" CornerRadius="15" Margin="15"
                  Background="{StaticResource BackgroundGradient}"
                  BorderThickness="0.75" BorderBrush="Black">
            <Grid>
              <Rectangle x:Name="RectCenter" Margin="15"
                         Fill="{StaticResource CenterGradient}" />
              <Rectangle x:Name="RectHighlight" Margin="15,15,15,75"
                         Fill="{StaticResource HighlightGradient}" Opacity="0.75"/>
            </Grid>
          </Border>
        </ControlTemplate>
      </Button.Template>
    </Button>
  </Grid>
```

The definition of the brushes hasn't changed,

I've just moved them to the window so that they can be referenced by both buttons.

Except for the Resources section, which I moved to the Window, the first button is identical to the example.

The only real change here is swapping Rectangle shapes for the Ellipse shapes in the first version.

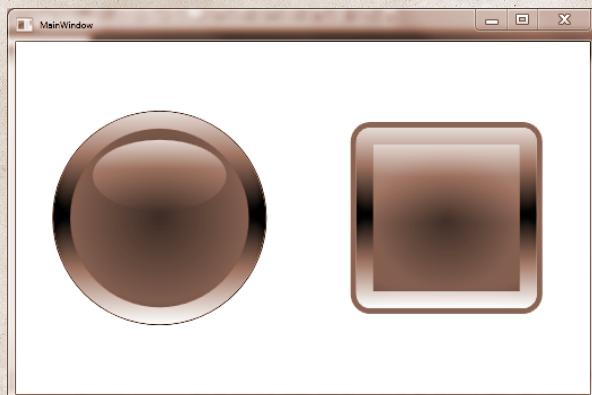
TEMPLATE BINDINGS

In the last exercise, changing the `BorderBrush` or `BorderThickness` properties of the `Button` has no effect on the display. In fact, none of the appearance properties except `Width` and `Height` have any effect. That's pretty rude, don't you think? Controls expose an entire set of properties that can be set to control their appearance, and a well-behaved control template will respect those property settings, or at least any that apply to the new visual tree. XAML exposes a markup extension for doing just that, and it's easy to use:



PUT ON YOUR THINKING HAT

The templates we've been working with set the `BorderThickness` and `BorderBrush` explicitly. Change them to use a template binding, and then set the public properties of the `Button` to see the changes reflected in the appearance.





HOW'D YOU DO?

Here's my version of the second button. (The syntax is the same for both.) You won't have chosen the same color and thickness, but the point is that a `TemplateBinding` will work with ANY values you might choose.

```
<Button BorderThickness="5" BorderBrush="Crimson" ... >
    <Button.Template>
        <ControlTemplate>
            <Border x:Name="RoundedRectBorder" CornerRadius="15" Margin="15"
                Background="{StaticResource BackgroundGradient}"
                BorderBrush="{TemplateBinding BorderBrush}"
                BorderThickness="{TemplateBinding BorderThickness}">
                <Grid>
                    <Rectangle x:Name="RectCenter" Margin="15"
                        Fill="{StaticResource CenterGradient}" />
                    <Rectangle x:Name="RectHighlight" Margin="15,15,15,75"
                        Fill="{StaticResource HighlightGradient}" Opacity="0.75"/>
                </Grid>
            </Border>
        </ControlTemplate>
    </Button.Template>
</Button>
```



MAKE A NOTE

To be truly well behaved, our new buttons should use the `Background` property to determine the color of the `CenterGradient`. But `Background` is a `Brush`, while a `GradientStop` is set using a `Color`, so we can't use a `TemplateBinding`. We have to use the more general WPF binding mechanism and a converter. We'll learn how to do that in the next chapter.

PRESENTING CONTENT

WPF has an entire set of classes called PRESENTERS that are like a `TemplateBinding`, but instead of presenting the value of a single property, they can display complex properties. For example, a `ContentPresenter` displays the `Content` property of a control, and that can get really complex. (Remember, the entire logical tree of a `Window` is contained in its `Content` property.) Presenters even have their own template properties (that might contain elements with their own template properties...it's never ending, really).

`ContentPresenter`, like all presenters, inherits from `FrameworkElement`, so you have access to all the formatting properties like `HorizontalAlignment` or `Height`, but you don't need to do anything special to include the actual content; the element handles that for you.



The contents of the control, as specified in the `Content` property, are inserted here.



PUT ON YOUR THINKING HAT

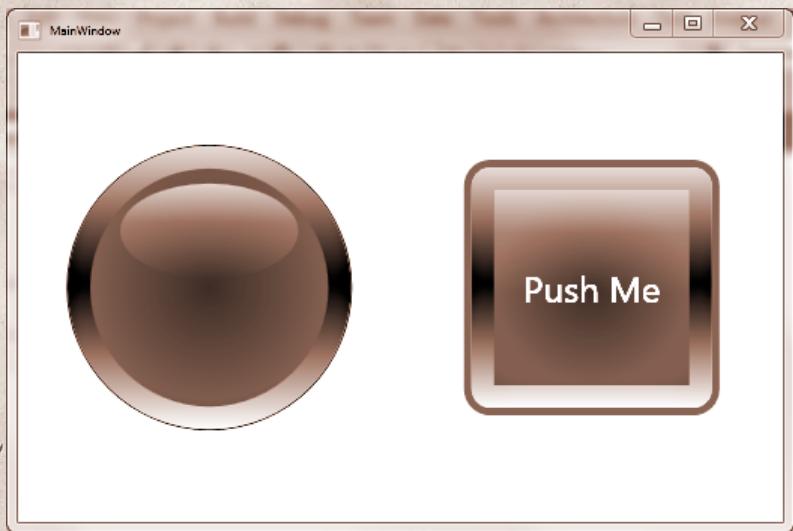
Add some text to one or both of the buttons (the buttons, not their templates) and a `ContentPresenter` element inside the `Grid` in the template to display it. You'll need to change the `FontSize` and `Foreground` properties of the button (again, the `Button`, not the template) in order for the text to be visible.



HOW'D YOU DO?

If your content isn't visible, check that you made the `FontSize` big enough to see and changed the `Foreground` from the default of Black.

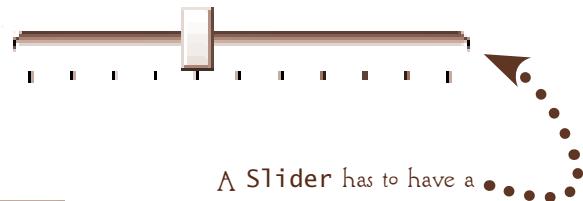
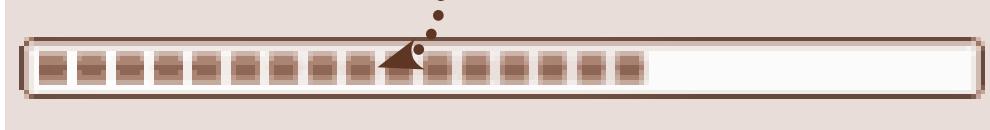
```
<Button Grid.Column="1" Width="200" Height="200" VerticalAlignment="Center"
    HorizontalAlignment="Center" Foreground="White" FontSize="24"
    BorderBrush="Crimson" BorderThickness="5">
<Button.Template>
    <ControlTemplate TargetType="Button">
        <Border x:Name="RoundedRectBorder" CornerRadius="15" Margin="15"
            Background="{StaticResource BackgroundGradient}"
            BorderBrush="{TemplateBinding BorderBrush}"
            BorderThickness="{TemplateBinding BorderThickness}">
            <Grid>
                <Rectangle x:Name="RectCenter" Margin="15"
                    Fill="{StaticResource CenterGradient}" />
                <Rectangle x:Name="RectHighlight" Margin="15,15,15,75"
                    Fill="{StaticResource HighlightGradient}" Opacity="0.75"/>
                <ContentPresenter HorizontalAlignment="Center"
                    VerticalAlignment="Center"/>
            </Grid>
        </Border>
    </ControlTemplate>
</Button.Template>
    Push Me
</Button>
```



CONTROL COMPONENTS

The beauty of the WPF template model is that you can make a control look almost any way you want it to, but all the functionality remains intact. If you were using most other UI platforms, you'd have to rebuild the control from scratch in order to do what we've been doing with a few lines of XAML. But the key word here is "almost". You might want your sliders to look like gauges (which would be cool, by the way), but there still has to be some way for the user to change the value. A TextBox isn't going to be too useful without some place to enter the text. A ComboBox...well, you get the idea.

A ProgressBar has to have something to indicate progress and a track-y thing to indicate the 100%. • • •



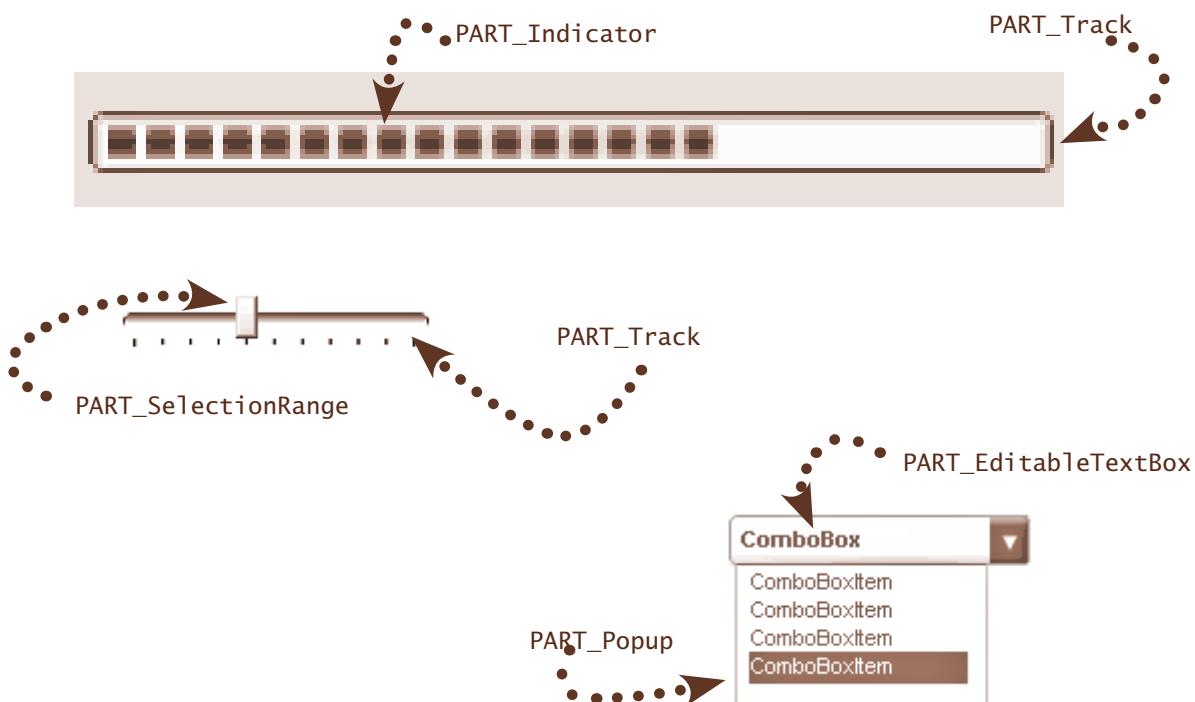
A Slider has to have a button-y thing that can be moved, and a track-y thing that it can move along.

A ComboBox has to have someplace to display the selected content, someplace to display the list, and a button-y thing to control whether the list is displayed.

CONTROL CONTRACTS

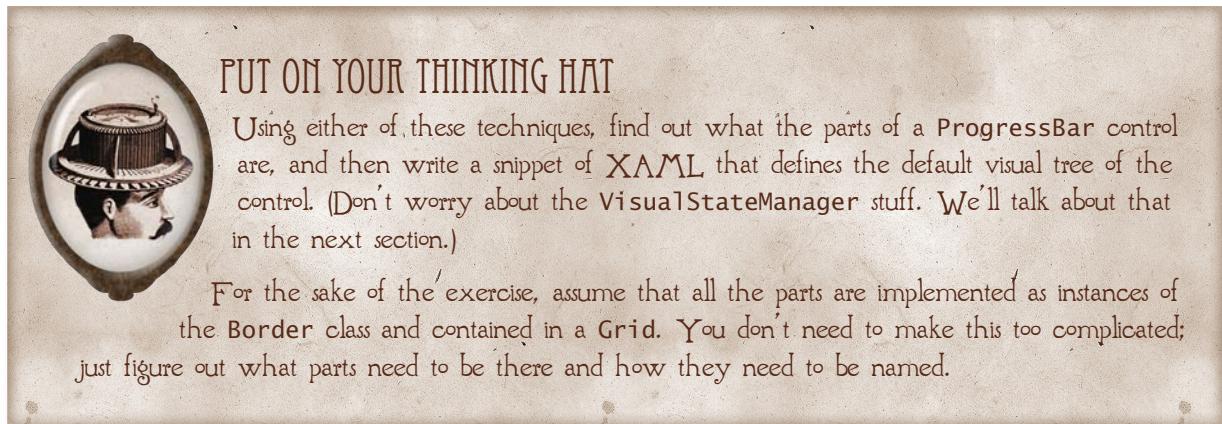
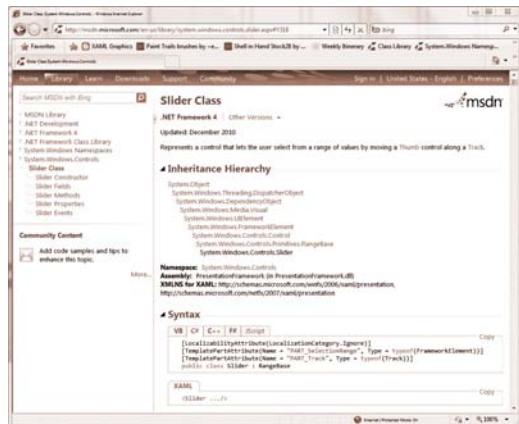
In earlier versions of the .NET Framework, the parts of a control that were required for it to function were implemented by setting the names of elements in the template to magic names that could be referenced by the functional code. It worked, but it was (to put it politely) painful. Just trying to figure out what controls need what parts and what those parts should be called could take the research skills of a PhD in Library Science. .NET 4.0 has made everybody's life easier by formalizing the ad hoc naming convention in a **CONTROL CONTRACT** that defines everything the template should include in order to function properly.

The control contract is defined using attributes. (We haven't worked with attributes since we looked at them back in Chapter 5, so you'll be forgiven for forgetting that they're a way of adding metadata to a code component.) Required control components, called **Parts**, are implemented using the **TemplatePartAttribute** that has two required parameters: **Name** and **Type**. The value of the **Name** property is the **x:Name** you must use in your control template, and the value of **Type** is the type of control it needs to be. (These are usually defined at a pretty high level.)



To make it even better, the contracts are actually documented in MSDN (Search for "Styles and Templates", with the quotation marks.) It should lead you to the Control Styles & Templates section of the Library:

But because the good people at MSDN have been known to move things around (usually the day after I find them), if all else fails, the control template attributes are included in the Class Library definition of the control:





HOW'D YOU DO?

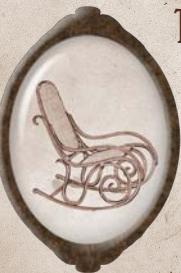
Here's my version.

```
<ControlTemplate TargetType="ProgressBar">
<Grid>
    <Border x:Name="PART_Track" />
    <Border x:Name="PART_Indicator" />
    <Border x:Name="PART_GlowRect" />
</Grid>
</ControlTemplate>
```



MAKE A NOTE

I've been referring to the parts that are specified in the control template as "required components", but in fact they aren't all necessarily required. The `ProgressBar` is a good example. The `PART_GlowRect` is purely decorative, and if you leave it out, the control will still work just fine. Even parts that are functional can sometimes be left out. A scrollbar, for example, doesn't need those little arrows that move the thumb by a small increment. The WPF controls are very smart about just ignoring parts that aren't included in the template.



TAKE A BREAK

Lots of WPF elements include templates. The control template we've explored in this section is one of the most common, and we'll look at data templates in the next chapter. But all of the templates work essentially the same way, so don't panic if you run across one you haven't seen before.

Now why don't you take a break before you complete the Review and move on to the final section of this chapter: the `VisualStateManager` that takes the place of triggers in the control contract.



REVIEW

What kind of Presenter do you use to display the Content property of a control?

How would you declare an ellipse with a dashed outline?

Control templates are typically placed in one of two places. What are those places, and when would you use each?

What's the difference between a Decorator and an Adorner?

Why is it customary to put a Grid with one row and one column inside a Border?

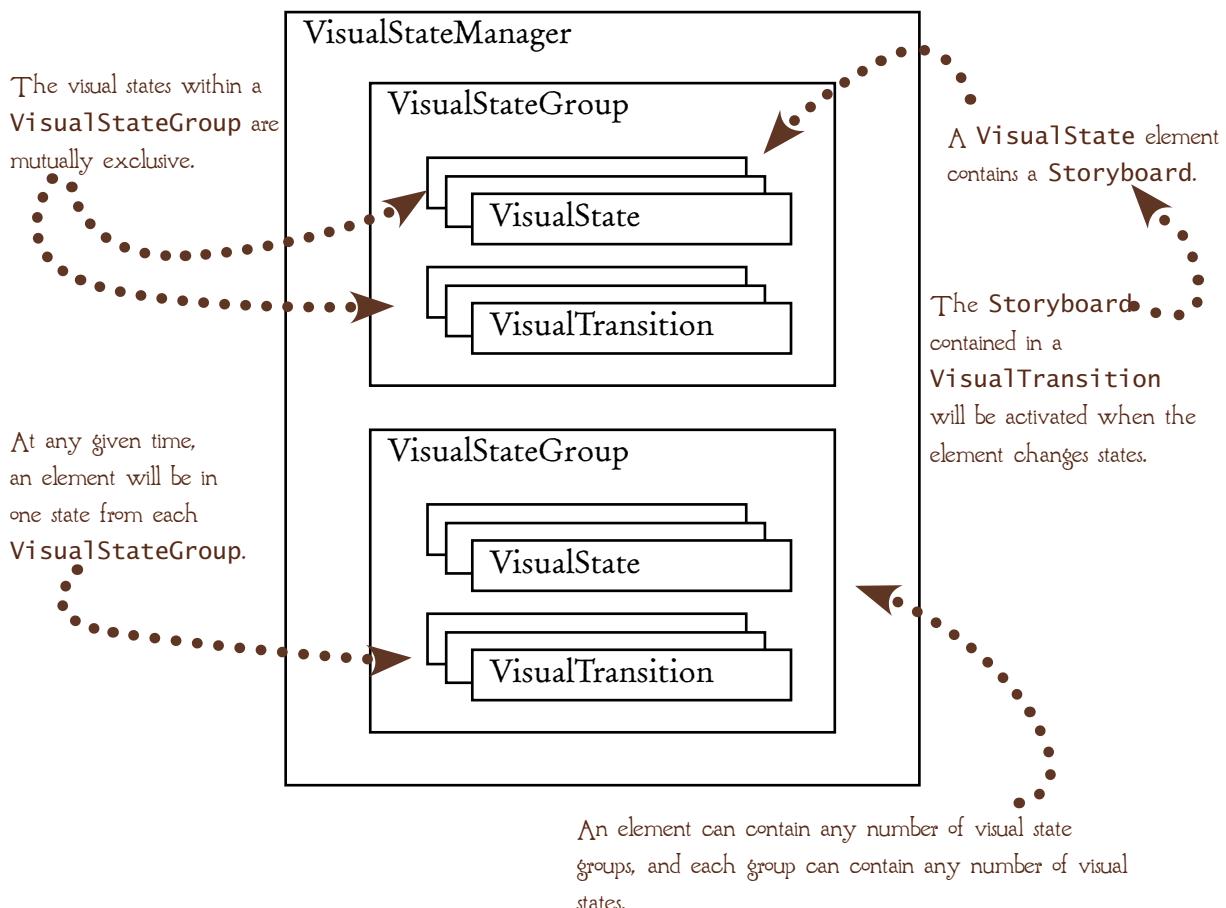
What element do you use to make a control template reflect the public properties of an element?

What happens if you don't include one of the parts specified by a control's contract?

VISUAL STATES

In the last chapter we used property and event triggers to control the visual behavior of a control. Inside a control template (and unfortunately only inside a control template) you have another, more powerful option: the `VisualStateManager`. Like control parts, the visual states that are supported by a control are defined by its control contract, using the `TemplateVisualState` attribute, which defines both the state and the state group:

```
[TemplateVisualState(Name="Normal", StateGroup="CommonStates")]
```





PUT ON YOUR THINKING HAT

The Button element has three state groups and a total of nine visual states, as shown below. Write a skeleton of a control template that defines each state. Don't worry about the Storyboard right now. You know how to do that, right?

Visual State Group	Visual State
CommonStates	Normal
	MouseOver
	Pressed
	Disabled
FocusStates	Focused
	Unfocused
ValidationStates	Valid
	InvalidFocused
	InvalidUnfocused



MAKE A NOTE

You can still use triggers inside a control template, and a lot of the samples you'll find on the Internet use this older technique. But when you're customizing controls that define visual states, you really should use the `VisualStateManager` instead because the controls are built to integrate with it, and you're likely to have more flexibility than you would with a pure-trigger solution. Of course, you can always use both, if you want to respond to an event or property change that isn't covered by the states defined by the control.



HOW'D YOU DO?

Did you figure out that you specify the group and state names using the `x:Name` markup extension, or did you have to peek?

```
<ControlTemplate TargetType="Button">
    ...
    Visual Tree ...
    ...
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal" />
            <VisualState x:Name="MouseOver" />
            <VisualState x:Name="Pressed" />
            <VisualState x:Name="Disabled" />
        </VisualStateGroup>

        <VisualStateGroup x:Name="FocusStates">
            <VisualState x:Name="Focused" />
            <VisualState x:Name="Unfocused" />
        </VisualStateGroup>

        <VisualStateGroup x:Name="ValidationStates">
            <VisualState x:Name="Valid" />
            <VisualState x:Name="InvalidFocused" />
            <VisualState x:Name="InvalidUnfocused" />
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</ControlTemplate>
```

The `VisualStateGroups` collection isn't exposed directly; it's an attached property.

Each group and each state is identified by name.



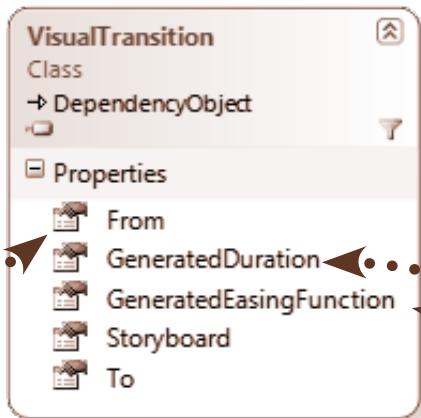
MAKE A NOTE

Unlike the parts defined in the control template, the visual states supported by a control are only documented in the "Control Styles and Templates" section of MSDN, not in the class library. At least, that's the case as I write this. It may be different tomorrow.

VISUAL TRANSITIONS

In addition to the `VisualState` storyboards, the `VisualStateManager` allows you to define how the control moves between states by adding `VisualTransition` elements to the `Transitions` collection of a `VisualStateGroup`. Like a `VisualState`, a `VisualTransition` can only contain a `Storyboard`, but the class also defines some additional properties:

The `From` and `To` properties define the visual states the control is transitioning between.



`GeneratedDuration` is functionally identical to the `Duration` property of a `Storyboard`. It defines the amount of time the transition takes, and like `Storyboard.Transition`, the default value is 1 second, which is usually too long.



`GeneratedEasingFunction` takes the same values as the `EasingFunction` of a `Storyboard`.



PUT ON YOUR THINKING HAT

The properties of the `VisualTransition` behave like the comparable properties of the `Storyboard`, and like the `Storyboard`, the `From` and `To` properties are optional. Can you match up combinations of property settings with the result?

	From	To
1	Specified	Specified
2	Not Specified	Specified
3	Specified	Not Specified
4	Not Specified	Not Specified

From any state to the state specified

From the specified state to the specified state

From any state to any other state

From the specified state to any other state

If more than one `VisualTransition` applies, what would you expect WPF to do?



HOW'D YOU DO?

	From	To
1	Specified	Specified
2	Not Specified	Specified
3	Specified	Not Specified
4	Not Specified	Not Specified

- ② From any state to the state specified
- ① From the specified state to the specified state
- ④ From any state to any other state
- ③ From the specified state to any other state

If more than one `VisualTransition` applies, what would you expect WPF to do?

Whenever you're working with a style, resource or template item, WPF uses the last one specified.



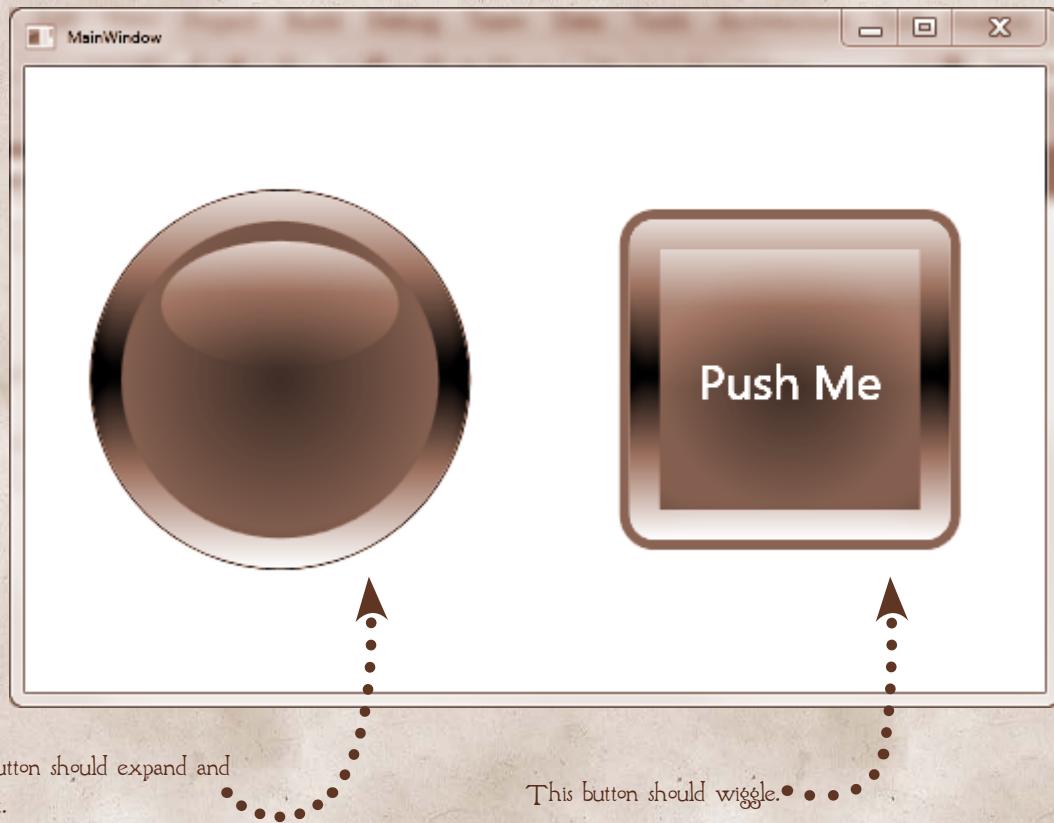
TAKE A BREAK

The `VisualStateManager` fits right in with the general pattern of styles and templates, and `XAML` in general, and that's the end of our exploration of the subject. Why don't you take a break before you finish up the chapter and move on to `WPF binding`?



ON YOUR OWN

At the end of the last chapter you wrote an event trigger to make a button wiggle. Using the same technique, add a `VisualStateManager` to the sample so that the rectangular button wiggles when the mouse moves over it. Add another to the round button, so that it expands a bit and then contracts and changes color.



This button should expand and contract.

This button should wiggle.



REVIEW

Working with our control template sample application, move the control templates into a resource dictionary as styles.

Create a control template for a `TextBox` that you think would blend with the button we developed in this chapter. Be sure to check whether there are any control parts you need to include.

Add visual state support to the `Button` control templates. Like control parts, you don't need to include all of them. Since our buttons aren't data-bound, you can certainly leave out the `ValidationStates` group, and you might decide some of the others don't apply.

Congratulations! You've finished the chapter. Take a minute to think about what you've accomplished before you move on to the next one...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



WPF BINDING

22

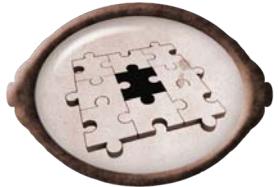
In the last chapter we used the `TemplateBinding` markup extension to set values in a control template to the values of the public properties of the object.

In general, "binding" simply means that the value of a property is retrieved from someplace else, and when you're working in WPF, the dependency property system does most of the work for you. You tell WPF what you want it to do, either in XAML or code, and it handles the how.

Template bindings are a particular case of binding. WPF provides a general mechanism through the `Binding` object and the `{Binding}` markup extension. You can bind the value of a dependency property to any .NET object, and you can manipulate the binding in pretty much any way you need to.

In this chapter we'll explore the basic concepts of WPF binding: the `source` of the value, the `target` that receives the value, and the `Binding` and `View` objects that link them. We'll also look at binding sets of items to an items control and some of the techniques you can use to format and navigate a set of items.

Binding is fundamental to WPF, and when you finish the chapter, you should understand the basics. But in WPF, binding can be a very, very large and complex subject, and I can't make you a database programmer in one short chapter. If you're interested in this area of application development, you'll need to learn database theory and ADO.NET, the .NET data access platform, and possibly Entity Framework, a tool for mapping relational databases to objects.



FITTING IT IN

The basic structure of a WPF binding is simple to use and understand. There are only three fundamental components: the **BINDING SOURCE**, a .NET object that provides the value, the **BINDING TARGET**, a dependency property that receives the value, and a **Binding** class that links the two.

The **BINDING SOURCE** can be any .NET object. It doesn't know anything about the binding.

The **Binding** class manages moving values between the source and target. A **Binding** points to a specific source, but it can be used by multiple targets.

Any dependency property can be a binding target.



If the binding source is a set of items like a collection or array, WPF will create a **CollectionView** that points the binding to a specific item in the set, and also provides the ability to navigate through the set and to organize it in various ways.



If the value provided by the binding source is of the wrong type or in the wrong format, you can add a converter to the **Binding** to translate it.

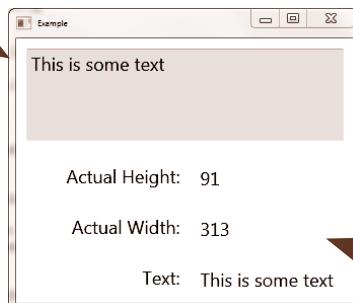
The `GetBindingExpression(<property>)` method will return a reference to the **BindingExpression**.

The binding expression links the **BINDING** object to the target. Because it's an independent object, you can reuse a **BINDING** with multiple targets, which is a surprisingly common thing to do.

THE PROBLEM...

To get a sense of what the WPF binding mechanism does for you, let's look at a simple example. Here's a simple window that displays the actual width and height of the `Border` object in the top pane.

This is just a `TextBox` element with a background fill. Its name is set to `Box`. It spans two columns and one row of a Grid.



These labels display the values of the `ActualHeight`, `ActualWidth` and `Text` properties of the `TextBox` element.

Without WPF binding, in order to change the values in the labels when the size of the window changes, you'll need to update them in code. The `Border` doesn't have a `SizeChanged` event, but the `Window` does, and we can update the values in a basic event handler.

```
Private Sub Window_SizeChanged(sender As Object, _  
    e As SizeChangedEventArgs)  
    WidthLabel.Content = Box.ActualWidth.ToString()  
    HeightLabel.Content = Box.ActualHeight.ToString()  
End Sub
```

Now, that's not too tedious, but what about the `TextLabel`? Well, `TextBox` exposes a `TextChanged` event, and we can catch that to update the label. But there's a problem: As soon as the application runs, it generates a null reference exception because `TextChanged` gets called when the `TextBox` is created, which happens before the label is created. So you need to check that the `Window`'s loaded:

```
Private Sub Box_TextChanged(sender As Object, _  
    e As TextChangedEventArgs)  
    If Me.IsLoaded Then  
        TextLabel.Content = Box.Text  
    End If  
End Sub
```

Starting to get a little complicated, isn't it? And what if you wanted to, say, bind a label to the background color of the `TextBox`? There's no event for that anywhere. (And `Color` doesn't have a reasonable text representation. What do you do about that?) The solution, of course, is to let WPF handle all these ugly details for you. Keep reading...

THE SOLUTION...

Compare the increasingly complex set of event handlers on the previous page to the same functionality implemented with WPF binding. All we have to do is specify the element and property we want, and WPF handles all the rest. I know which version I'd rather write.

This is our original version. We had to find change events we could use and track down some non-obvious bugs. And this is only three properties. What if you had twenty, or fifty? What about binding to a collection? It gets ugly fast.

```
Private Sub Window_SizeChanged(sender As Object, _  
    e As SizeChangedEventArgs)  
    WidthLabel.Content = Box.ActualWidth.ToString()  
    HeightLabel.Content = Box.ActualHeight.ToString()  
End Sub  
  
Private Sub Box_TextChanged(sender As Object, _  
    e As TextChangedEventArgs)  
    If Me.IsLoaded Then  
        TextLabel.Content = Box.Text  
    End If  
End Sub
```

Here's a version that uses WPF binding. All we need to do is add a markup extension to the XAML. (We could have done it in code instead, of course). That's all we ever have to do. In every situation. No hunting for an event we can hook on to, no tracking down weird and wonderful unrelated problems...

```
<TextBox x:Name="Box" .../>  
<Label Content="{Binding ElementName=Box, Path=ActualWidth}" .../>  
<Label Content="{Binding ElementName=Box, Path=ActualHeight}" .../>  
<Label Content="{Binding ElementName=Box, Path=Text}" .../>
```



TASK LIST

We'll start this chapter by exploring the basics of WPF binding. You've already done some basic binding in other contexts, so this might seem like old news, but then we'll move on to some more interesting stuff, including how to convert and format the data and how to bind to and navigate entire sets of data.



CREATING BINDINGS

Before we get into the complexities of WPF bindings, we need to get the basics down, so we'll start by looking at the basic objects involved and how to use them.



BINDING TO COLLECTIONS

Next we'll explore binding entire sets of data to items controls like the `ListBox`, `ComboBox` and `TreeView`. In order to use these controls effectively, you need to know how the data from the binding source is displayed, so we'll also examine data templates, which are just another form of the templates you learned how to build in the last chapter.



WORKING WITH COLLECTIONS

WPF provides some useful functions for working with collections of data. We'll start by exploring the Master-detail pattern that lets you display the details about an item selected in an items control, and then explore a few other useful tools in the binding toolbox, including how to filter, sort and group a set of data, and how to convert a value into a form that's more useful to you.



CREATING BINDINGS

You've already written one kind of binding expression when you used the `TemplateBinding` element in a control template, but the general binding syntax is a little different, so let's start with a simple little example. (We'll be expanding it as we go along.)

This is a `ListBox` we'll be binding later in the chapter.

The elements in this

column...

...are given the names in this column.

`TextSource:`

`TextLabel:` *bound value*

`FontLabel:` *bound value*

`GridLabel:` *bound value*

`Update Now`

We'll use the `TextBox` as the source for the labels.

We'll be binding these labels as we go along.

```
{Binding ElementName=<name>, Path=<property>}
```



PUT ON YOUR THINKING HAT

Ready to get started? Here's a class diagram of the `Binding` object. Using it and the example from a couple of pages ago, can you do the following?

1

Create a new project and build the basic form. Mine uses a `Grid` as the root element, with three columns and five rows, but you don't need to organize it that way if you have a better idea. Make sure you give the `TextBox` and the labels in the last column the names specified in the second column.

2

Set the binding for the `Label` named `TextLabel` so that it displays the contents of the `TextBox`.

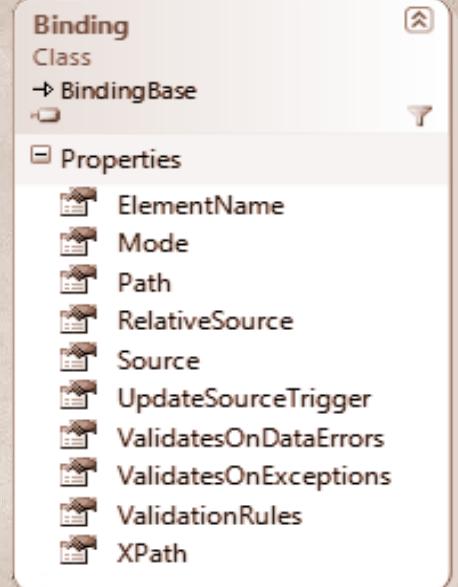
3

Set the binding for the `Label` named `FontLabel` so that it displays the `FontStyle` of `TextLabel`.

Leave the `GridLabel` and `Button` for the time being; we'll fix them later.

4

Run the application. When is the first `TextBox` updated? What property of the `Binding` object do you think you need to set to change this behavior?





HOW'D YOU DO?

It's only fair to tell you that WPF binding can get pretty complicated, but most of the time, it's just this easy.

I've left out the XAML that just formats the Window so we can concentrate on the Binding elements.

```
<Window ... >
<Grid ... >
  <Grid.ColumnDefinitions ... />
  <Grid.RowDefinitions ... />

  <ListBox ... />
  <Label ...>TextSource:</Label>
  <Label ...>TextLabel:</Label>
  <Label ...>FontLabel:</Label>
  <Label ...>GridLabel:</Label>

  <TextBox x:Name="TextSource" ... >Sample Text</TextBox>
  <Label x:Name="TextLabel"
        Content="{Binding ElementName=TextSource, Path=Text}" />
  <Label x:Name="FontLabel"
        Content="{Binding ElementName=TextLabel, Path=FontSize}" />
  <Label x:Name="GridLabel" ... >bound value</Label>

  <Button x:Name="UpdateButton" ... >Update Now</Button>
</Grid>
</Window>
```

There are other options, and we'll look at a couple in a few pages, but the easiest way to bind to an element is by name.

Here are the lines that perform the binding. You had an example of the first one. Did you work out the second one?



TAKE A BREAK

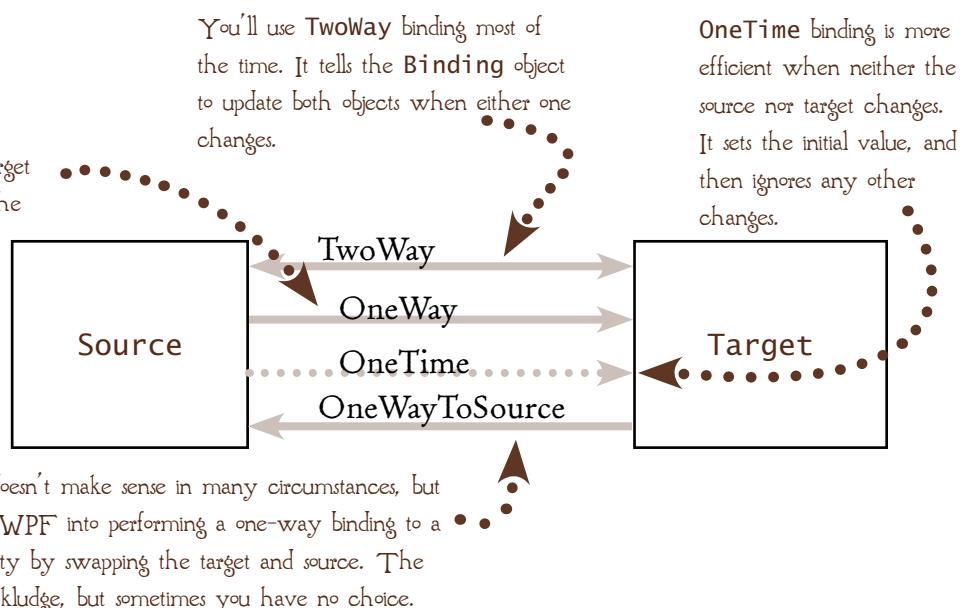
Bindings are a little complex, so why don't you take a break before we move on?

BINDING MODES

In the exercise, I asked you to think about what property you might use to control when a binding target is updated. The answer is the `UpdateSourceTrigger`, which takes a member of the `UpdateSourceTrigger` enumeration. `UpdateSourceTrigger` is one of two properties that control how the Binding behaves. The other is the `BindingMode`, which determines how the source and target interact. We'll start by exploring the second property: `BindingMode`.

The `BindingMode` property controls the effect that the target and source have on one another.

`OneWay` binding means that the target has no effect on the source.



`OneWayToSource` doesn't make sense in many circumstances, but it can be used to trick WPF into performing a one-way binding to a non-dependency property by swapping the target and source. The technique is, at best, a kludge, but sometimes you have no choice.

PUT ON YOUR THINKING HAT

One of the bindings in our example should really be `OneTime`. Can you decide which one and set the `BindingMode` property?

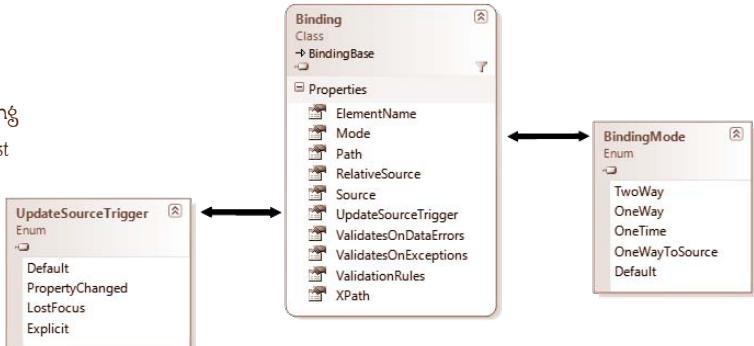


Answer on page 813

UPDATE TRIGGERS

The other property that controls how the Binding behaves is the `UpdateSourceTrigger` that does exactly what the name implies: It defines when an update of the source is triggered. Don't confuse the `UpdateSourceTrigger` with style or event triggers. Like static properties and static resources, the terms mean different things.

`UpdateSourceTrigger` only applies when the source is actually being updated, which means the `Mode` must be `TwoWay` or `OneWayToSource`.



When the `UpdateSourceTrigger` is set to `Explicit`, you have to call `UpdateSource()` on the `BindingExpression`. (Remember that WPF creates a `BindingExpression` whenever you link a `Binding` object to a property on the binding source. This allows you to reuse a `Binding` object for multiple source properties.

The `GetBindingExpression()` method returns an instance of the `BindingExpression` for you to work with in code.



Notice how the property is specified:
`<Class>.<`

```
BindingExpression be = TextLabel.GetBindingExpression(Label.ContentProperty)  
be.UpdateSource()
```



Once you have a reference to the `BindingExpression`, the `UpdateSource()` method tells the `Binding` to perform the deferred update.



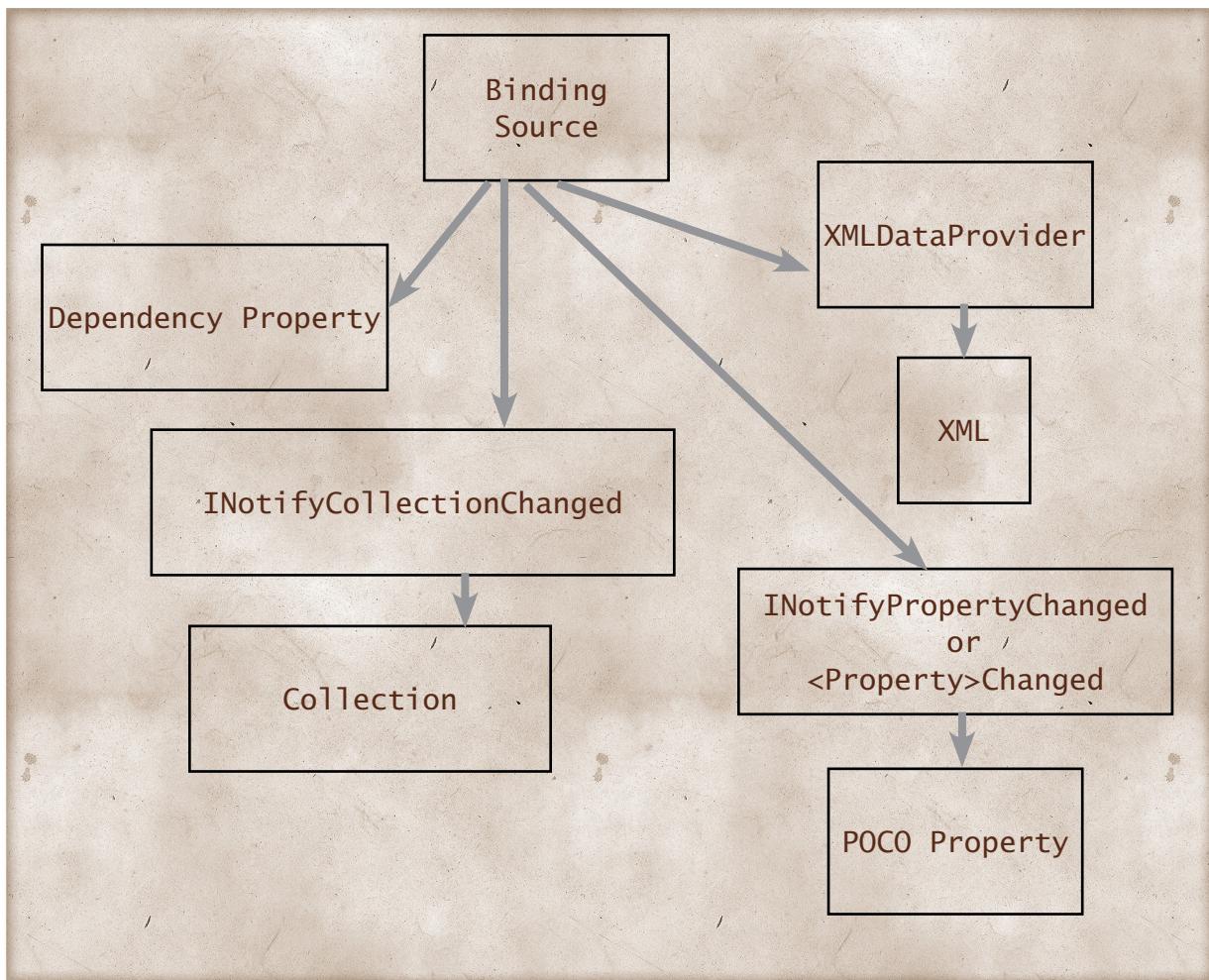
PUT ON YOUR THINKING HAT

`UpdateSourceTrigger.Explicit` is useful when you want to give the user a chance to change their mind before they commit any changes. Change the `TextLabel` binding to `Explicit`, and add an event handler to the button that calls `UpdateSource()`.

Answer on page 813

BINDING SOURCES

As you know, the binding source provides the values that are used by the target. While the target of a Binding can only be a dependency property, the source can be any CLR object. That said, in order for the binding to update properly, the target needs to provide some kind of change notification, which simply means the object needs to tell the world when its values change. Dependency properties do this automatically, as do the collections defined in the Framework Class Library. Change notification on XML data is implemented by wrapping the XML inside an XMLDataProvider element. For other POCO types defined by the FCL, you'll need to check that either the class implements `INotifyPropertyChanged` or `INotifyCollectionChanged` or the property you're interested in raises a `<Property>Changed` event.



DEFINING A BINDING SOURCE

To fully define a binding source, you need to specify two things: the object that provides the values, and the property of the object that contains the value you want. Of course, this is .NET, so there are several ways to do both of these things, and sometimes you don't need to do specify them at all...

Specifying the object can be optional. We'll see how that works in a few pages.

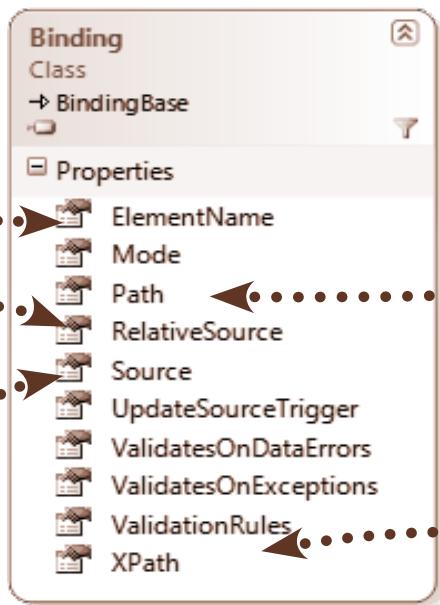
SPECIFYING THE OBJECT

You've already used `ElementName` to bind to a named element in the logical tree.

`RelativeSource` is used to set the source and target to the same element, or when you don't know the element's name. Read on...

Specifying the property is optional if you want the entire object.

SPECIFYING THE PROPERTY



The `Source` property can be used to bind to a resource or to a collection defined in code or XAML.

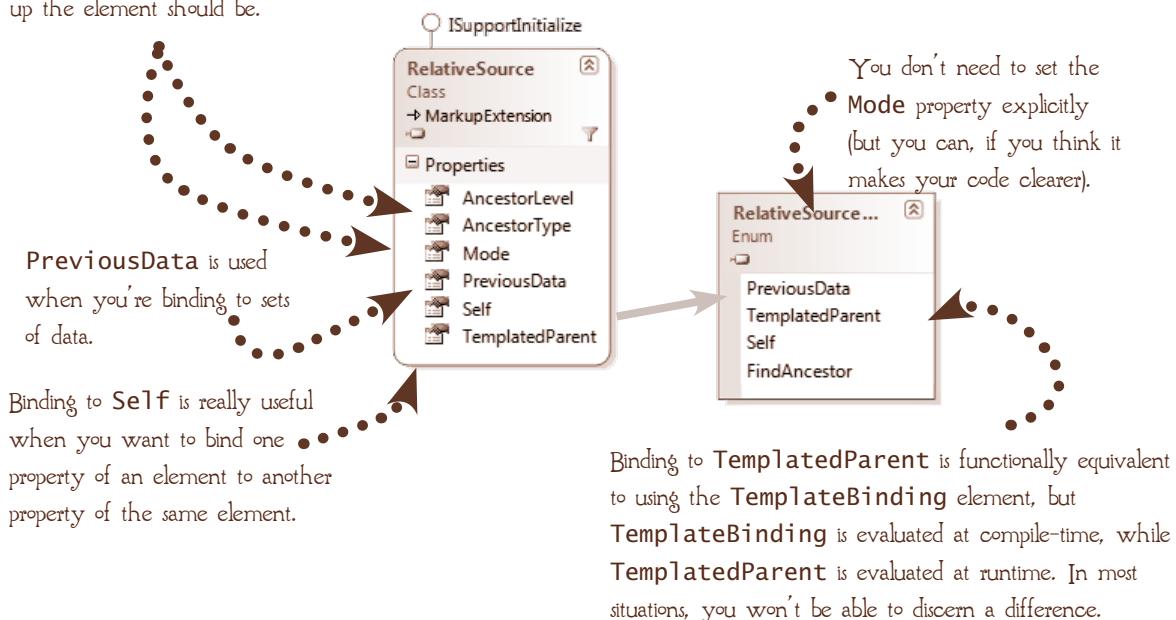
You'll use `Path` when you're binding to a .NET object.

`XPath` is used when you're working with XML data (which we won't be discussing here.)

USING RELATIVESOURCE

Specifying the binding source using the `ElementName` or `Source` property is fairly straightforward. `RelativeSource` is a little less so, but it's so handy that it bears closer examination. The `Binding.RelativeSource` property takes a member of the `RelativeSource` class.

`AncestorLevel` and `AncestorType` are used to find an element somewhere up the logical tree. If you only specify `AncestorType` using the `x:Type` extension, the property will be bound to the first element of that type in the tree. `AncestorLevel` lets you specify how many levels up the element should be.



PUT ON YOUR THINKING HAT

The basic syntax for using `RelativeSource` is `Source={RelativeSource <Type>}`. So, for example, to specify the `TemplatedParent`, you would use `Source={RelativeSource TemplatedParent}`. Can you use `RelativeSource` to write XAML bindings to bind a label to each of the following?

- The `FontSize` of the element itself!
- The `Width` of the `Window` element containing the label:
- The `ActualWidth` of a `Grid` containing the label:



HOW'D YOU DO?

The `FontSize` of the element itself:

```
{Binding Source={RelativeSource Self}, Path=FontSize}
```

or

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



You'll see this syntax a lot in examples and in MSDN. Many people think it's clearer. I don't, but you can make up your own mind.

The `Width` of the `Window` element containing the label:

```
{Binding Source={RelativeSource FindAncestor,  
AncestorType={x:Type Window}}, Path=Width}
```

The `ActualWidth` of a `Grid` containing the label:

```
{Binding Source={RelativeSource FindAncestor,  
AncestorType={x:Type Grid}}, Path=ActualWidth}
```



This was a tricky one. Did you figure out how to specify the type?

THE DATA CONTEXT

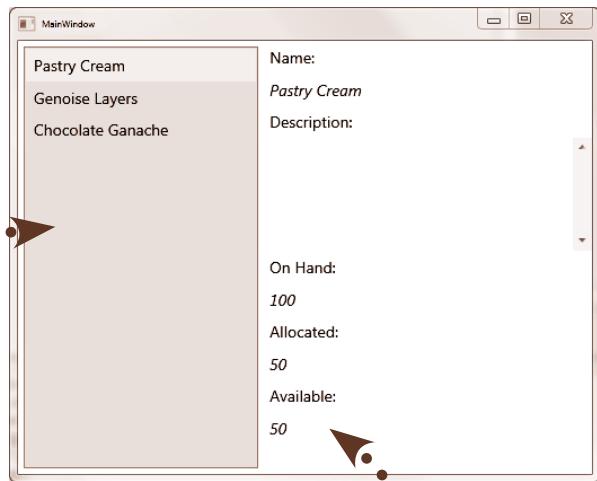
The `FrameworkElement` class exposes a property named `DataContext` that you can use to set a default binding source. Because `DataContext` is a dependency property, it's inherited by child elements, so, for example, in the very common situation of a `Window` that displays the details of a single object or set of objects, you can set the `Window.DataContext` property and it will automatically be used as the binding source of all the elements the `Window` contains, unless you specify a different binding source explicitly.

The `Window.DataContext` property is set to a collection of ingredients.

```
Window.DataContext = Ingredients;
```

The `ListBox` is bound to the entire collection. We'll find out how to do that in the next section.

The `DataContext` can also be set in XAML, but the syntax is a little trickier. We'll find out how to do that in the next section.



Each of these labels is bound to the collection by specifying only the path:

```
<Label Content={Binding Path=Available} />
```

TAKE A BREAK



The `DataContext` property is easy to use, and it's particularly useful when you're working with a collection of elements, as in our example. Why don't you take a break now, and we'll move on to binding to collections in the next section?



REVIEW

Declare a `Label` element that binds its `Content` to the `Text` of a `TextBox` named `ContentSource`.

Declare a `Label` that binds its `Content` to the `Name` property of the `Window`'s `DataContext`.

When would it be appropriate to use a `OneWay` binding?

We've seen that in order to support two-way binding, a collection has to implement some form of change notification. Can you bind to an instance of the `ArrayList` class?

Why would you want to set `UpdateSourceTrigger` to `Explicit`?

Declare a `TextBlock` (just for a change) that binds its `Text` property to the `FontSize` of itself.



UNDER THE MICROSCOPE

Every control type has a default value for `UpdateSourceTrigger` and `BindingMode`, but it's best practice to specify both explicitly, if for no other reason than it saves remembering what the defaults are.



HOW'D YOU DO?

Here's the final code for the BindingMode and UpdateSourceTrigger exercises.

```
<Window ... >
<Grid ... >
  <Grid.ColumnDefinitions ... />
  <Grid.RowDefinitions ... />

  <ListBox ... />
  <Label ...>TextSource:</Label>
  <Label ...>TextLabel:</Label>
  <Label ...>FontLabel:</Label>
  <Label ...>GridLabel:</Label>

  <TextBox x:Name="TextSource" ... >Sample Text</TextBox>
  <Label x:Name="TextLabel"
        Content="{Binding ElementName=TextSource, Path=Text,
        UpdateSourceTrigger=Explicit}" />
  <Label x:Name="FontLabel"
        Content="{Binding ElementName=TextLabel, Path=FontSize ,
        BindingMode=OneTime}"/>
  <Label x:Name="GridLabel" ... >bound value</Label>

  <Button x:Name="UpdateButton" ... >Update Now</Button>
</Grid>
</Window>
```

The UpdateSourceTrigger
should be Explicit for the
TextLabel.

Since the FontStyle can't
change, there's no reason
for the Binding to keep
monitoring it.

Here's the handler that

triggers the update:

```
Private Sub UpdateButton_Click(sender As Object, e As RoutedEventArgs)
    Handles UpdateButton.Click
    BindingExpression be = TextLabel.GetBindingExpression(Label.ContentProperty)
    be.UpdateSource()
End Sub
```

Did you notice that there's also an UpdateTarget() method? When do you think you'd need that?

From pages 805 and 806

BINDING TO COLLECTIONS

Back in Chapter 16 we examined items controls and the `Items` property that allows you to specify the items displayed by the control directly in XAML or code. Items controls also expose a second property, `ItemsSource`, that is bound to a collection.

The syntax for declaring a binding for the `ItemsSource` property is almost identical to the syntax for binding any other property, but when you're binding to the whole collection, not just a single value, you omit the `Path` property:

```
<ListBox ItemsSource="{Binding Source={x:StaticResource myCollection}}" />
```

There's another little wrinkle to binding the `ItemsSource`. If you're binding to an inherited `DataContext`, which as I've said is very common, you can omit the `Source`, too. But you do have to specify that the value is bound, so you use what's known as an **EMPTY BINDING**:

```
<ListBox ItemsSource="{Binding}" />
```

Looks strange, doesn't it? But it's perfectly valid provided there's a `DataContext` somewhere in the logical tree that the `ListBox` (or any other items control) can find.

MAKE A NOTE

The `Items` and `ItemSource` properties of an items control are mutually exclusive. If you try to set both, your application won't compile.

BACK TO THE BOYS

Way back in Chapter 1 we looked at an application that our chefs need. When Gordon is putting together the dessert menu, he needs to be able to choose an ingredient from a list and to see how much is on hand and how much is available. It's time to start building it, or a least a simple version of it.



Here's the class we'll use to track ingredients.

...and here's the form we'll build (and bind).

MainWindow

	Name:	Description:
Pastry Cream	Pastry Cream	Genoise Layers Chocolate Ganache
On Hand:	100	
Allocated:	50	
Available:	50	

Ingredient Class

- Properties
 - Allocated
 - Available
 - Description
 - Name
 - OnHand
- Methods
 - Ingredient

PUT ON YOUR THINKING HAT

Create a new project, and add the `Ingredient` class declaration. You can use auto-implementation for the properties.





HOW'D YOU DO?

Did you need to go back and check the syntax? That's okay; it's been a while since we've done this. What's important is that you got it working.

Public Class Ingredient

```
Public New(name As String, onHand As Integer,  
amtAllocated As Integer)
```

```
    Me.Name = name
```

```
    Me.OnHand = onHand
```

```
    Me.Allocated = amtAllocated
```

```
    Me.Available = OnHand - Allocated
```

```
End Sub
```

```
Public Property Name As String
```

```
Public Property Description As String
```

```
Public Property OnHand As Integer
```

```
Public Property Allocated As Integer
```

```
Public Property Available As Integer
```

```
End Class
```



ON YOUR OWN

Go ahead and build the Window shown on the previous page. I bet you can make it better-looking than mine.

BUILDING BINDABLE COLLECTIONS

To get our application working, we don't want a single instance of the class; we want a collection. You know that change notification is the heart of WPF binding, so when you're building your own collection of items to use as a binding source, you'll need to implement it. You can manually implement the `INotifyCollectionChanged` interface, which isn't difficult because it only has a single method, but there's an easier way: the `ObservableCollection<T>` generic in the `System.Collections.ObjectModel` namespace does it all for you. All you need to do is derive a class from `ObservableCollection<T>`, add your items, and you're good to go.

Derive a class from `ObservableCollection<Of T>`:

1

```
Class MyBindable  
Inherits ObservableCollection<Of MyType>  
... properties or methods unique to your class...  
End Class
```

Instantiate the class and add items:

2

```
Public items As MyBindable = New MyBindable()  
MyBindable.Add(new MyType())
```

You don't have to assign the observable collection to the `DataContext` you can reference it using any valid XAML syntax, but `DataContext` is usually the easiest way.



The instance is available for binding:

3

```
Me.DataContext MyBindable;
```



PUT ON YOUR THINKING HAT

Add an `ObservableCollection` class called `SampleIngredients` to the project. In the constructor, add three or four ingredients to the collection.

Then write the code to instantiate the collection and set the `Window.DataContext` in the `MainWindow()` constructor. Finally, bind the `ListBox`. The contents of the `Listbox` will look weird when you run the application; that doesn't mean you did something wrong. We'll fix that in a minute.



HOW'D YOU DO?

It doesn't matter, of course, if you chose different values for the `Ingredient` instances you added to the `SampleIngredients` collection.

Here's the collection class:

```
Public Class SampleIngredients  
    Inherits ObservableCollection< OfIngredient>  
  
    Public Sub New()  
        Add(new Ingredient("Pastry Cream", 100, 50))  
        Add(new Ingredient("Genoise Layers", 10, 0))  
        Add(new Ingredient("Chocolate Ganache", 50, 35))  
    End Sub  
End Class
```

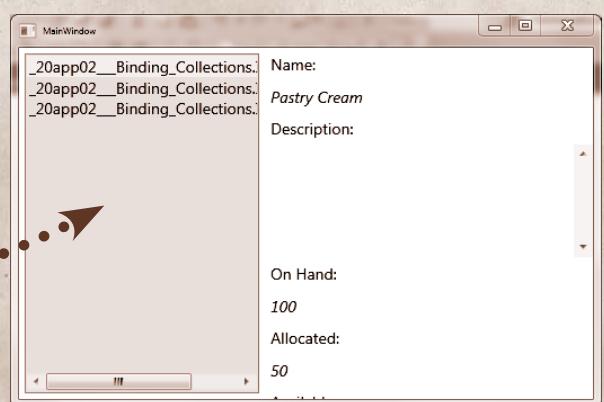
Here's the code to set the `DataContext`:

```
Public Sub New()  
    InitializeComponent()  
  
    Ingredients As SampleIngredients = New SampleIngredients()  
    Me.DataContext = Ingredients  
End Sub
```

And finally, here's the XAML to bind the `ListBox`:

```
<ListBox ... ItemsSource="{Binding}" />
```

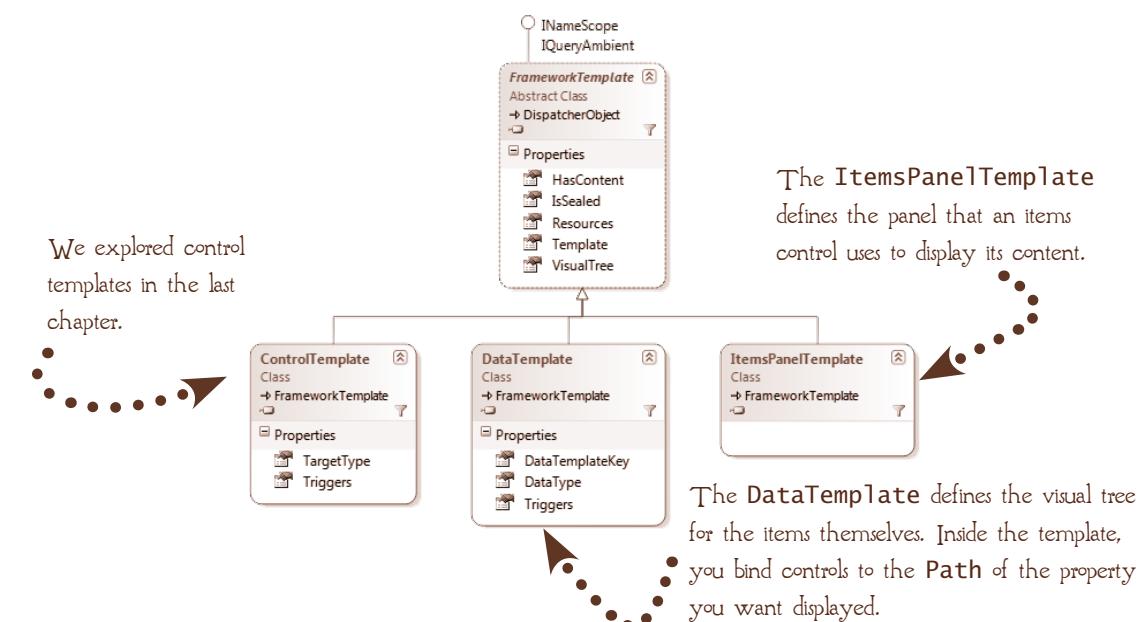
But that's not really
what we had in mind,
is it?



MORE TEMPLATES

By default, the `Listbox` (or any items control) displays its contents by calling `ToString()`, and if you don't override the `ToString()` method in your class (and we didn't), it just returns the name of the class. That's not a bad default behavior for the method, but it's hardly ever what you want the `Listbox` to do. The answer, of course, is to tell the `Listbox` what you *do* want it to do.

If you poke around the WPF class libraries for a bit, you'll find several objects that have properties ending with the word "Template". Like the control templates we studied in the last chapter, all of these properties take instances of a class that descends from the abstract `FrameworkTemplate` class & they all work in essentially the same way. (But only control templates support the `VisualStateManager`.)



PUT ON YOUR THINKING HAT

Can you set the `ItemTemplate` property of the `Listbox` to a `DataTemplate` that displays the name of each `Ingredient` in a `Label`?



HOW'D YOU DO?

I always forget that you have to declare a **DataTemplate** inside the **ItemTemplate**, so if you have trouble with that, you're in good company.

```
<ListBox ... ItemsSource="{Binding}" >  
  <ListBox.ItemTemplate>  
    <DataTemplate>  
      <Label Content="{Binding Path=Name}" />  
    </DataTemplate>  
  </ListBox.ItemTemplate>  
</ListBox>
```

That's better!

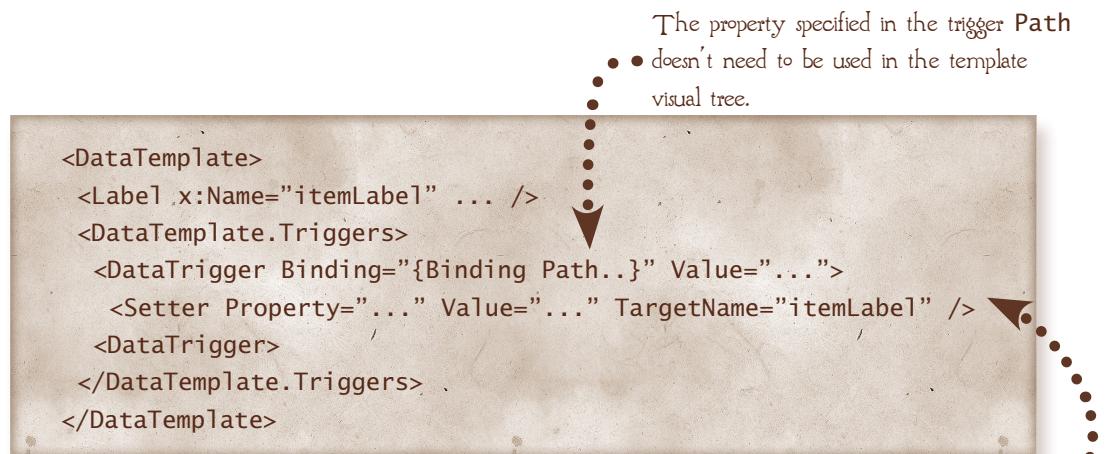


MAKE A NOTE

- The visual tree that you define in a **DataTemplate** is applied to every item in the **Items** collection of the control.
- Although our example is very simple, you can make the template as complex as you need it to be. The **ItemTemplate** class doesn't support the **VisualStateManager**, but as we'll see, you can use triggers to control the visual behavior of the application.
- We put the template directly in the control declaration, but of course it can be included in a control template instead.

DATA TRIGGERS

Just like the `ControlTemplate`, a `DataTemplate` contains a `Triggers` collection. In theory, both collections can contain triggers of any type, but the `DataTemplate` most often contains instances of `DataTrigger` or `MultiDataTrigger`. Unlike a property trigger that's activated by the value of a dependency property, a `DataTrigger` is activated by the value of a `Binding`, which not only means that it is particularly well suited for use inside a data template, it also means that data triggers can be activated by anything that can be used as a binding source, not just a dependency property.



The property specified in the trigger `Path`
• doesn't need to be used in the template
visual tree.

Here's the tricky bit about using a `DataTrigger`.
You need to set the `TargetName` property. Since
you'll usually be changing something in the template
visual tree, you need to give that element a name.

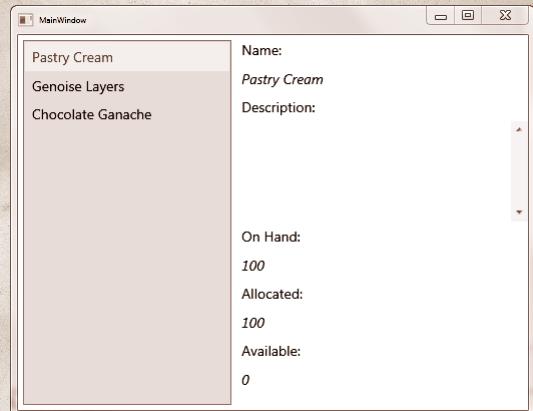




HOW'D YOU DO?

Here's the full definition of the `ListBox`:

```
<ListBox ... ItemsSource="{Binding}" >
<ListBox.ItemTemplate>
  <DataTemplate>
    <Label x:Name="itemLabel" Content="{Binding Path=Name}" />
    <DataTemplate.Triggers>
      <DataTrigger Binding="{Binding Path=Available}" Value="0">
        <Setter Property="Foreground" Value="Red"
          TargetName="itemLabel"/>
      </DataTrigger>
    </DataTemplate.Triggers>
  </DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
```



TAKE A BREAK

One of the things I really like about WPF is that once you know how one part of it works, you know a lot about how other parts of it work. That's certainly true of templates. Each type has little idiosyncrasies, but they have more in common than not.

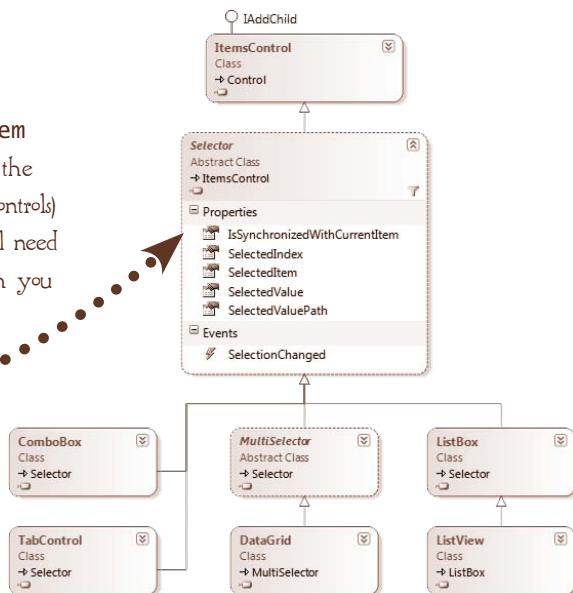
So now that we have our `ListBox` working properly, why don't you take a break before we explore some of the other functionality WPF provides when you're working with collections?



WORKING WITH COLLECTIONS

The list of ingredients isn't all that useful by itself. What we really want the window to do is display the details of whichever item is selected in the `ListBox`. It's a UI pattern called **MASTER-DETAIL**, and it's wonderfully easy to do in WPF. Just set the magic `IsSynchronizedWithCurrentItem` property to `true`. (`IsSynchronizedWithCurrentItem` is defined by `Selector` and inherited by all the items controls that support selection, including `ListBox`.) Any other elements bound to the same collection will automatically display values from the selected item.

The `IsSynchronizedWithCurrentItem` property keeps all the elements bound to the `ItemsSource` (including other items controls) synchronized. It defaults to `false`, so you'll need to set it if you want this behavior (which you almost always do).



PUT ON YOUR THINKING HAT

Set the `IsSynchronizedWithCurrentItem` property of the `ListBox` and bind the remaining controls in the `StackPanel`.





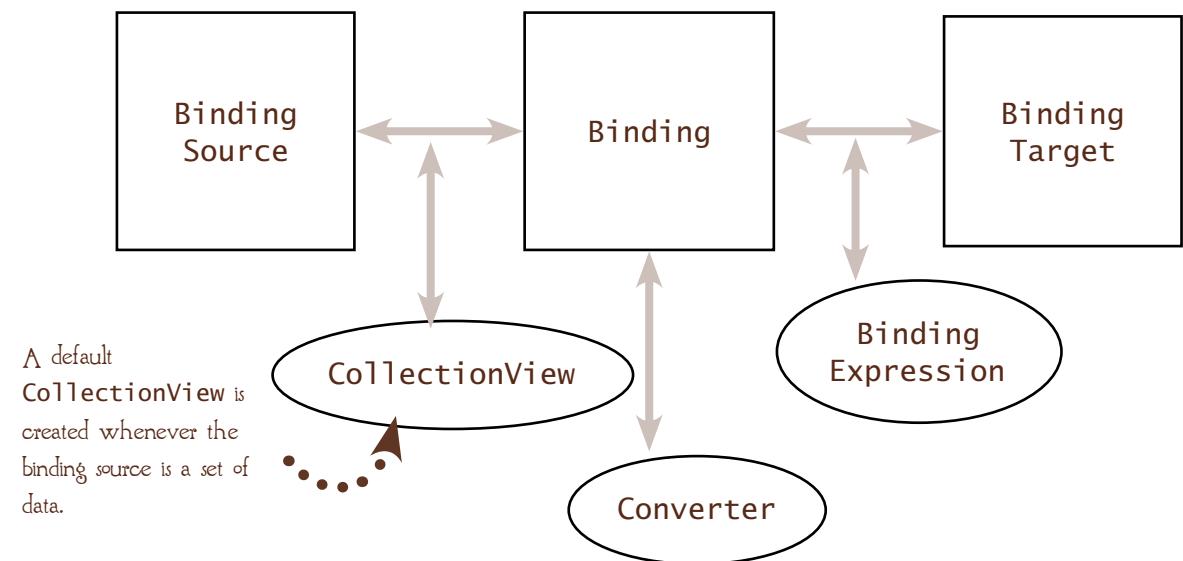
HOW'D YOU DO?

Here's the whole window definition (or at least my version of it):

```
<Window ...>
  <Grid TextBlock.FontSize="14">
    <Grid.ColumnDefinitions ... >
      <ListBox ... ItemsSource="{Binding}" IsSynchronizedWithCurrentItem="True" >
        <ListBox.ItemTemplate>
          <DataTemplate>
            <Label x:Name="itemLabel" Content="{Binding Path=Name}" />
            <DataTemplate.Triggers>
              <DataTrigger Binding="{Binding Path=Available}" Value="0">
                <Setter Property="Foreground" Value="Red" TargetName="itemLabel"/>
              </DataTrigger>
            </DataTemplate.Triggers>
          </DataTemplate>
        </ListBox.ItemTemplate>
      </ListBox>
      <StackPanel Grid.Column="1">
        <Label>Name:</Label>
        <Label FontStyle="Italic" Content="{Binding Path=Name}" />
        <Label>Description:</Label>
        <ScrollViewer Height="100" FontStyle="Italic"
          Content="{Binding Path=Description}"/>
        <Label>On Hand:</Label>
        <Label FontStyle="Italic" Content="{Binding Path=OnHand}"/>
        <Label>Allocated:</Label>
        <Label FontStyle="Italic" Content="{Binding Path=Allocated}"/>
        <Label>Available:</Label>
        <Label FontStyle="Italic" Content="{Binding Path=Available}"/>
      </StackPanel>
    </Grid>
  </Window>
```

THE MAGIC OF VIEWS

The first time you see master-detail binding in action, it seems like magic. Well, it seems like magic if you've spent much time working with older, clunkier binding systems. If you look at the basic binding structural diagram, you'll see that there's an object that sits between the collection and the Binding called a `CollectionView`. It's this class, which WPF creates for you, that provides the master-detail magic by providing CURRENCY.



WORDS FOR THE WISE

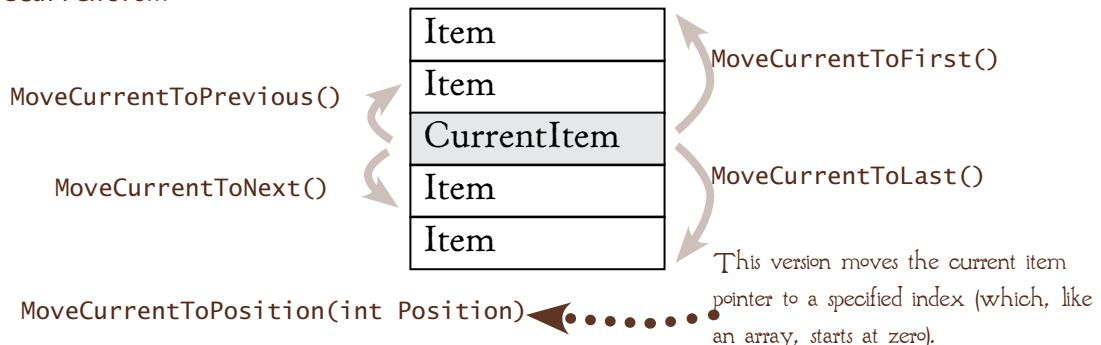
CURRENCY simply means that there's a pointer to one item in the collection. That item is considered the "current item". Not a difficult concept, but don't confuse it with CONCURRENCY, which is multiple processes operating at the same time.

CurrentItem

Item

NAVIGATING THE VIEW

If the `CollectionView` provides a current item, it stands to reason that it has to provide some way to change the item it points to, right? And, of course, it does, by way of a set of methods that begin with `MoveCurrentTo...`.



I LIED.

Of course, before you can call a method on the `CollectionView`, you have to have a reference to the `CollectionView`. But before we see the method that will give you that reference, I have a confession to make.

I lied when I said that WPF creates a default `CollectionView`. (Only just this one time. I haven't done it before, I promise.) It's a little more complicated than that. What actually happens is that WPF creates an instance of a `CollectionView` or one of the classes that descends from `CollectionView` (`BindingListCollectionView`, `ListCollectionView`, or `ItemCollection`, depending on what kind of collection you're working with). Each of these classes provides special functionality for working with a specific type of data, and when you're doing advanced binding, you'll need to figure out what class you're working with and what it can do for you.

There's another class, the `CollectionViewSource`, that acts as a proxy for all the `CollectionView` classes in XAML. We'll see how that works when we look at filtering and sorting in a few pages.

So, now that I've confessed, how do you get a reference to the `CollectionView` that WPF created for you? By calling the static `GetDefaultView()` method of `CollectionViewSource` and passing it the collection:

```
CType(CollectionViewSource.GetDefaultView(<collection>), CollectionView)
```

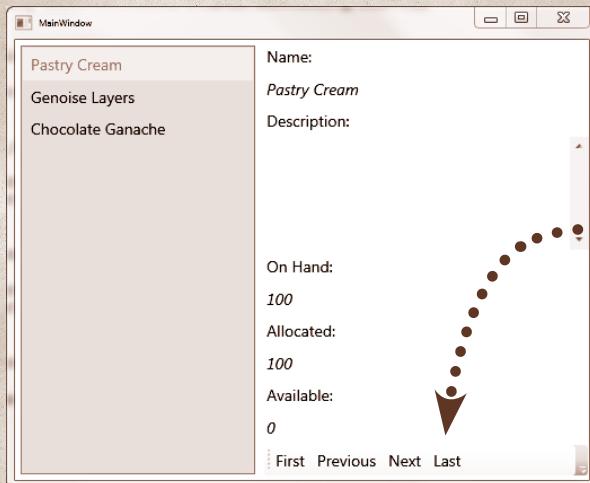
GetDefaultView() actually returns an instance of the
ICollectionView interface, so you'll have to cast it.



PUT ON YOUR THINKING HAT

You wouldn't ordinarily add separate navigation controls to an application that uses the Master-Detail interface pattern (because the ItemsControl performs the same function), but just to get some experience with navigating a `CollectionView`, add a `Toolbar` element to the bottom of the detail pane.

HINT: You'll find it easier if you declare a private variable to contain the reference to the view and set it in the window constructor.



Before you decide you're finished with this exercise, try this: Run it, click the `Last` button, and then click the `Next` button. What happens? How can you fix it? (HINT: The `CollectionView` exposes two useful properties for this situation: `CurrentPosition` and `Count`.)



ON YOUR OWN

If you find this exercise really easy (or you're just in the mood for a challenge), try adding a `TextBox` between the `Previous` and `Next` buttons that lets you type a position number. (Make sure you check the number entered against the `Count` property before you try to move!)



HOW'D YOU DO?

Here's my version.

```
Public Partial Class MainWindow  
    Inherits Window
```

```
    Private view As CollectionView
```

Declaring a value to hold the
`CollectionView` reference saves
a bit of code.

```
    Public Sub New()  
        InitializeComponent()
```

Did you remember
the cast? The
compiler would
have caught the
omission.

```
        Ingredients As New SampleIngredients = New SampleIngredients()  
        Me.DataContext = Ingredients  
        view = CType(CollectionViewSource.GetDefaultView(Ingredients), _  
            CollectionView)
```

```
    End Sub
```

```
    Private Sub First_Click(sender As Object, e As RoutedEventArgs)  
        view.MoveCurrentToFirst()  
    End Sub
```

```
    Private Sub Previous_Click(sender As Object, e As RoutedEventArgs)  
        If view.CurrentPosition > 0 Then  
            view.MoveCurrentToPrevious()  
        End If  
    End Sub
```

```
    Private Sub Next_Click(sender As Object, e As RoutedEventArgs)  
        If view.CurrentPosition < view.Count Then  
            view.MoveCurrentToNext()  
        End If  
    End Sub
```

This is how you
make sure you don't
move past the end or
the beginning of the
collection.

```
    Private Sub Last_Click(sender As Object, e As RoutedEventArgs)  
        view.MoveCurrentToLast()  
    End Sub  
End Class
```

CONVERTING VALUES

One of the most basic manipulations of bound values is to transform the raw data into something more useful. You might have a value like #1260FF, but you want a `SolidColorBrush`. Or you have a string like "Riordan, Rebecca" and you want "Rebecca Riordan". Those kinds of manipulations are easy to perform in WPF by assigning the `Binding.Converter` property to a class that implements the `IValueConverter` interface. There are actually four steps to the process:

1

CREATE THE CONVERTER CLASS

Your first step is to build your converter class. The `IValueConverter` interface only has two methods: `Convert()` and `ConvertBack()`. You can throw a `NotImplementedException()` for one of the methods if you don't need it.

```
Public Class myConverter
    Inherits IValueConverter

    Public Function Convert(value As Object, targetType As Type,
        parameter As Object, _
        Culture As culture) As Object
        ... return the converted value ...
    End Function

    Public Function ConvertBack(value As Object, targetType As Type,
        parameter As Object, Culture As culture) As Object
        ... return the converted value ...
    End Function
End Class
```

2

MAKE YOUR CODE AVAILABLE IN XAML

In order to reference your converter in XAML, you'll have to add a namespace declaration to the opening `Window` tag. We talked about this briefly in earlier chapters, and we'll look at it in more detail on the next page.

3

DECLARE YOUR CONVERTER AS A WINDOW RESOURCE

You don't need to do anything tricky here; you just need to assign a key to the class so you can reference it in the `Binding`.

4

SET THE CONVERTER PROPERTY OF THE BINDING

You've declared the class as a Resource, so you'll use the `StaticResource` markup extension to set the property:

```
Content="{Binding ... Converter={StaticResource MyConverter}}"
```

OBJECT SHARING

Before we talk about some of the other cool things you can do with the `CollectionView`, we need to take a minute to look at how you share objects between XAML and the code-behind. So far in our sample applications the communication between the two has been very simple, but in real applications the communication often needs to be more complete. Here's how:

TO REFERENCE A XAML ELEMENT IN CODE

This one's easy. You just give the element a name, and then you can refer to it by name in code:

```
<Element x:Name="MyElement">
```

```
Me.MyElement
```

TO REFERENCE A XAML RESOURCE IN CODE

Unlike element names, resource keys that you declare in XAML aren't visible as members of the `Window`. (Which makes sense, really. They're not members of the `Window` class.) If you need to access them from code, one way is to use the `FindResource()` or `TryFindResource()` method defined by `FrameworkElement`:

You pass the `ResourceKey` as a `String`.

```
Style buttonStyle = CType(Me.FindResource("MyButtonStyle"), Style)
```

`FindResource()` returns an `Object`, so you'll need to cast it.

Another option is to access the `Resources` collection of an element, which contains an instance of a `ResourceDictionary` that can be accessed by key:

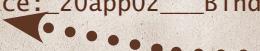
```
Style buttonStyle = CType(Me.ButtonElement.Resources("MyButtonStyle"), Style)
```

TO REFERENCE A CODE OBJECT FROM XAML

1

Declare the project namespace in the opening tag. By convention, the namespace alias for your project is `src`. The syntax for this is ugly, but it's actually quite easy to do, because as soon as you type `xm1ns:src=`, Intellisense will offer you a list of namespaces, with your project at the top of the list, so you just need to press the Tab key.

```
<Window x:Class="_20app02__Binding_Collections.MainWindow"  
xm1ns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
xm1ns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
xm1ns:src="clr-namespace:_20app02__Binding_Collections"  
...>
```



• Declare your project here.

2

Declare the object as a resource. There's nothing tricky about this; just use the namespace alias before the object name, and give it a key:

```
<Window.Resources>  
  <src:MyConverter x:Key="MyConverter" />  
</Window.Resources>
```

- The resource key doesn't need to be the same as the class. It can be any valid identifier.

Using this technique, we could have declared the `Ingredients` collection we're creating in our window constructor as a resource instead of setting the `DataContext` in code:

```
<Window.Resources>  
  <src:SampleIngredients x:Key="Ingredients" />  
<Window.Resources>  
<Window.DataContext>  
  <StaticResource ResourceKey="Ingredients"/>  
</Window.DataContext>
```

- You use the `StaticResource` element here, not the markup extension.

There's an advantage to this approach: The data is available to the XAML Designer. But there's also a catch: If you use this technique, you must pass the `DataContext` to the `GetDefaultView()` method in code, not the `Ingredients` collection, because they won't return the same references:

```
CollectionViewSource.GetDefaultView(Me.DataContext)
```



PUT ON YOUR THINKING HAT

Let's use a value converter to fix one of my pet peeves: "Item(s)". It's always the sign of sloppy programming. Create a class called `CountToItemsConverter`, declare it as a resource (remember to make the project available via a namespace declaration) and then bind a `TextBox` to the `Count` property, specifying the class as the `Converter` property. Here's what my version of the form looks like:

Name:	Pastry Cream
Description:	
On Hand:	100
Allocated:	100
Available:	0

Answer on page 835.



BEST PRACTICE

It's always tempting to name your value converter classes according to their types: `IntToItemsString` or `HexToSolidBrush`. It's really not a good idea. Your class names should reflect the semantics of what you're doing—converting `CollectionView.Count` to `Items`, or an `RGB` value to a `Background`. It just helps you avoid trying to reuse the converter somewhere it's not appropriate.

SORTING COLLECTIONS

Transforming raw data into something more useful or meaningful using a value converter is a good start on getting it into the form you need, but if you're working with a collections, you'll almost always want to specify the order in which the items are displayed. Even if your collection starts out that way, it's going to get messed up as items are added and edited. Fortunately, the `CollectionView` lets you specify sort conditions by simply adding the appropriate properties to the `SortDescriptions` collection of the `CollectionView`.

```
View.SortDescriptions.Add(New SortDescription("Name", _  
    ListSortDirection.Ascending))
```

You pass the name of the
property as a `String`.

`ListSortDirection` has two values: `Ascending` or
`Descending`. The default is `Ascending`.

Remember that `CollectionViewSource` is the XAML proxy for any kind of `CollectionView`. To set up sorting in XAML, you'll need to declare the collection as a resource, create a `CollectionViewSource` element, and then bind to that (rather than directly to the collection):

```
<Window.Resources>  
    <src:Object x:Key="MyDataSource" />  
  
    <CollectionViewSource Source="{StaticResource MyDataSource}" x:Key="cvs">  
        <CollectionViewSource.SortDescriptions>  
            <SortDescription PropertyName="Name" />  
        </CollectionViewSource.SortDescriptions />  
    </CollectionViewSource>  
</Window.Resources>  
<Window.DataContext>  
    <StaticResource ResourceKey="cvs" />  
<Window.DataContext>
```



PUT ON YOUR THINKING HAT

Using either code or XAML, sort the Ingredients list by Name.

Answer on page 840.



MAKE A NOTE

The `SortDescription` structure is defined in `System.ComponentModel`. You'll need to add a reference to that namespace to your code file with the other `using` statements before you can reference it in code, and declare the namespace in XAML before using it there. By convention, the namespace alias is `scm`:

```
xmlns:scm="clr-namespace:System.ComponentModel;assembly=WindowsBase"
```



HOW'D YOU DO?

Here's my version:

From page 832

```
Public Class CountToItemConverter  
    Inherits IValueConverter  
    Public Function Convert(value As Object, targetType As Type, _  
        parameter As Object, culture As System.Globalization.CultureInfo) _  
        As Object  
        If (Int) value > 0 Then  
            Return value.ToString() + " items"  
        Else  
            Return value.ToString() + " item"  
        End If  
    End Function  
  
    Public ConvertBack(value As Object, targetType As Type, _  
        parameter As Object, culture As System.Globalization.CultureInfo) _  
        As Object  
        Throw New NotImplementedException()  
    End Function  
End Class
```

And here's step 3, declaring the class as a resource.

```
<Window x:Class="_20app02__Binding_Collections.MainWindow"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:src="clr-namespace:_20app02__Binding_Collections"  
    Here's Step  
    3: declaring  
    the class as a  
    <Window.Resources>  
    resource.  
    <src:CountToItemConverter x:Key="ItemsConverter" />  
    </Window.Resources>  
    ...  
    <TextBox x:Name="CountOfItems" Text="{Binding Path=Count, Mode=OneWay,  
        Converter={StaticResource ItemsConverter}}" />  
    ...  
/>
```

There's no reason to
bother about converting
back a read-only binding.

This is step 2:
declaring the
namespace.

Did you work out what to do about
the read-only property error?

FILTERING

Filtering a collection—showing only a subset of the data—isn’t much more difficult than sorting, but it can only be done in code. The syntax you’ll use to specify filter criteria depends on what type of object you’re working with.

FILTERING A COLLECTIONVIEWSOURCE

To filter a `CollectionViewSource`, you need to handle the `Filter` event and set the `FilterEventArgs.Accepted` property:

```
Private Sub MyFilter(sender As Object, _  
    e As FilterEventArgs)  
    If e.Item <> Nothing Then  
        If (...criteria met...) Then  
            e.Accepted = True  
        Else  
            e.Accepted = False  
        End If  
    End If  
End Sub
```

You then add the event handler event in code or XAML:

```
AddHandler cvs.Filter, AddressOf MyFilter
```

FILTERING A COLLECTIONVIEW

If you're working with a `CollectionView`, the syntax for establishing a filter is a little odd. The `CollectionView.Filter` actually takes a `Predicate`. We talked about predicates in Chapter 10. They're like a special form of the `Delegate` that is always used to specify a criteria. It takes a single parameter of type `Object` and returns a `Boolean` indicating whether the object meets the criteria:

```
Private Function MyFilter(o As Object) As Boolean  
    Return (...criteria met or not...)  
End Function
```

Then, instead of adding an event handler, you set the `Filter` property to the `Predicate`:

```
cv.Filter = New Predicate(Of Object)(AddressOf MyFilter)
```



PUT ON YOUR THINKING HAT

Filter the `Ingredients` collection to show only items where `OnHand` is greater than 50. (Make sure your sample data contains some items that qualify and some that don't.) Depending on which approach you took in the last exercise, you'll use either an event handler or a predicate.



HOW'D YOU DO?

Here are my versions:

If you've declared the `CollectionViewSource` in XAML:

```
Public Sub New()
    InitializeComponent();

    Dim Ingredients As SampleIngredients
    Ingredients = New SampleIngredients()
    Me.DataContext = Ingredients
    Private view As CollectionView
    view = CType(CollectionViewSource.GetDefaultView(Me.DataContext), _
        CollectionView)
    view.SortDescriptions.Add(New SortDescription("Name", _
        ListSortDirection.Ascending))
    view.Filter = new Predicate(Of Object)(Address Of PredicateFilter);
End Sub

Private Function PredicateFilter(obj As Object) As Object
    ing As Ingredient = CType(obj, Ingredient)
    Return (ing.OnHand > 50)
End Function
```

The event will be called
when the collection is first
created, so you need to make
sure that the `Item` property
has a value.

If you're working with a CollectionView in code:

```
Public Sub New()
    InitializeComponent()

    ' Ingredients As SampleIngredients = New SampleIngredients()
    Dim cvs As CollectionViewSource cvs
    cvs = CType(Me.FindResource("cvs"), CollectionViewSource)
    cvs.Filter += New FilterEventHandler(AddressOf CollectionViewSource_Filter)
End Sub

Private Sub CollectionViewSource_Filter(sender As Object, e As FilterEventArgs)
    If e.Item <> Nothing Then
        ing As Ingredient = CType(e.Item, Ingredient)
        If ing.OnHand > 50 Then
            e.Accepted = True
        Else
            e.Accepted = False
        End If
    End If
End Sub
```



HOW'D YOU DO?

From page 830

I asked you to implement sorting either in XAML or in code. Here's the version if you chose an all-XAML approach:

```
<Window x:Class="_20app02__Binding_Collections.MainWindow"
...
    xmlns:src="clr-namespace:_20app02__Binding_Collections"
    xmlns:scm="clr-namespace:System.ComponentModel;assembly=WindowsBase"
... >

<Window.Resources>
    <src:CountToItemConverter x:Key="ItemsConverter" />
    <src:SampleIngredients x:Key="ResourceIngredients" />
    <CollectionViewSource Source="{StaticResource ResourceIngredients}"
x:Key="cvs">
        <CollectionViewSource.SortDescriptions>
            <scm:SortDescription PropertyName="Name" Direction="Ascending" />
        </CollectionViewSource.SortDescriptions>
    </CollectionViewSource>
</Window.Resources>
<Window.DataContext>
    <StaticResource ResourceKey="cvs" />
</Window.DataContext>
...
</Window>
```

And here's the version if you used code:

```
Public New()
    InitializeComponent()

    Ingredients As SampleIngredients = New SampleIngredients()
    Me.DataContext = Ingredients
    view = CType(CollectionViewSource.GetDefaultView(Me.DataContext), _
        CollectionViewSource)
    view.SortDescriptions.Add(new SortDescription("Name", _
        ListSortDirection.Ascending));
End Sub
```

The XAML version really is getting pretty verbose, isn't it? But most people think it's easier to read, and that's always the trade-off: verbosity against clarity.

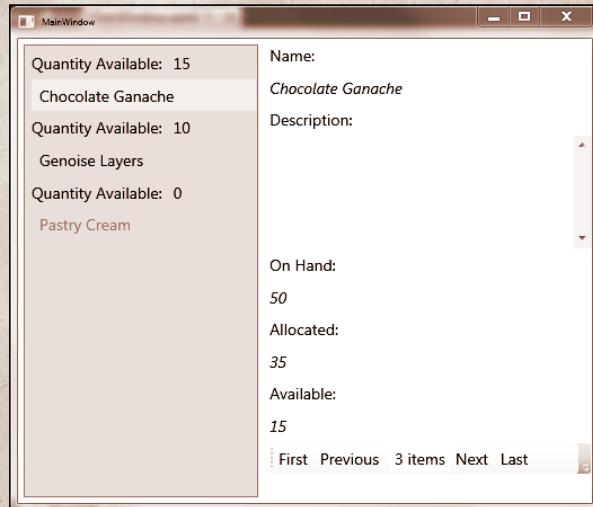


PUT ON YOUR THINKING HAT

Have you noticed how close to the end of the book we are? Of course you have. In fact, this is the last topic, and I have a real challenge for you. Here's just a little bit of information about how to group items in a collection. Can you group your collection by the **Available** property?

- Both the **CollectionView** and **CollectionViewSource** objects expose a **GroupDescriptions** property, so you can implement grouping in XAML or code.
- The **GroupDescriptions** collection property accepts instances of the abstract **GroupDescription** class, which has a single descendant: **PropertyGroupDescription**. The key property of a **PropertyGroupDescription** is **PropertyName**, which accepts a **String**.
- In order to display the items in groups, you have to add a **GroupStyle** to the items control. The **GroupStyle** accepts (among other things) a **HeaderTemplate**, which accepts a single **DataTemplate**.
- The data template inside a **HeaderTemplate** works just like any other **DataTemplate**, but it doesn't bind to the collection; it binds to a **CollectionViewGroup**. The **CollectionViewGroup.Name** property contains the value of the current property group (the number available, in our example).

So, can you duplicate this window?





WOO HOO, YOU'RE A PROGRAMMER!

With nothing but a very brief description of the how to do something (exactly the sort of description you'll usually get in MSDN), you just figured out how to perform a programming task. That's what programmers do.

Maybe you struggled a little bit, but that's okay. It will get easier. The next time you need to group a collection, you'll remember most of what you just did, and the time after that, you won't even have to think twice. That's how you move from being a beginning programmer to a journeyman to a master.

```
<Window.Resources>
    <src:CountToItemConverter x:Key="ItemsConverter" />
    <src:SampleIngredients x:Key="ResourceIngredients" />
    <CollectionViewSource Source="{StaticResource ResourceIngredients}" x:Key="cvs">
        <CollectionViewSource.SortDescriptions>
            <scm:SortDescription PropertyName="Name" Direction="Ascending" />
        </CollectionViewSource.SortDescriptions>
        <CollectionViewSource.GroupDescriptions>
            <PropertyGroupDescription PropertyName="Available" />
        </CollectionViewSource.GroupDescriptions>
    </CollectionViewSource>
</Window.Resources>

<Window.DataContext>
    <StaticResource ResourceKey="cvs" />
</Window.DataContext>
...
<ListBox ... ItemsSource="{Binding}" IsSynchronizedWithCurrentItem="True" >
    <ListBox.ItemTemplate>
        <DataTemplate>
            ...
        </DataTemplate>
    </ListBox.ItemTemplate>
    <ListBox.GroupStyle>
        <GroupStyle>
            <GroupStyle.HeaderTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <Label>Quantity Available:</Label>
                        <Label Content="{Binding Path=Name}" />
                    </StackPanel>
                </DataTemplate>
            </GroupStyle.HeaderTemplate>
        </GroupStyle>
    </ListBox.GroupStyle>
</ListBox>
```

Congratulations! You've finished the chapter. In fact, you've finished the book. But before you go on to your new life as a programmer, take a minute to think about what you've accomplished...

List three things you learned in this chapter:

1

2

3

Why do you think you need to know these things in order to be a VB programmer?

Is there anything in this chapter that you think you need to understand in more detail? If so, what are you going to do about that?



This page intentionally left blank

INDEX

A

abstract classes, 404, 409-412
abstract interfaces, 409-412
abstract types, 403
abstraction, 384, 391-392, 394
access modifiers, 223
 in Class Designer, 232-234
 for get/set accessors, 246-249
 in information hiding, 241
accessors, 241-249
actions, 25
AddHandler keyword, 479
ADO.NET, 486
ADO.NET Data Services, 486
ADO.NET Entity Framework, 486
adorners, 769
aggregate types. *See arrays; generics; specialized sets*
Agile Development, 20
Agile Manifesto, 20
Alexander, Christopher, 457
alignment of panel controls, 542
angle brackets in syntax diagrams, 128
animated values, 596
animations, 747-755
 class hierarchy, 748
 example of, 752-755
 reasons for using, 747
storyboards, 749-750
timeline classes, 751
application development. *See also OOA&D (Object-Oriented Analysis & Design); WPF controls*
 compilation process, 33-41
 compiled versus interpreted languages, 34-35
 JIT (just-in-time) compilation, 36-39
design patterns. *See design patterns*
“Hello, World” example, 53-58
steps in, 9-11, 13, 16-18, 59-63
system design, 19-32
 Agile Development, 20
 database schemas, 29
 screen layouts, 29
 UML class diagrams, 29

UML state diagrams, 25-28
use cases, 21-24
 waterfall models, 20
application facade, 487
ApplicationException class, 210
applications
 adding buttons, 97-98
 adding icons, 76-82
 building, 78
 deployment, 75, 107-119
 adding shortcuts to setup programs, 114
 build configurations, 116-117
 ClickOnce, 108-109
 creating setup programs, 110-112
 setup designers, 113
 types of, 107
 running, 58
architectural patterns, 459, 482-491
logical layers
 business layer, 487
 data access layer, 486
 list of, 485
 presentation layer, 491
 service layer, 488-490
 types of, 482-484
ARGB color model, 674
ArgumentException class, 210
arguments, 173
 reference types versus value types, 296-301
 routed events, 635-644
arranging windows, 84
Array class, 360-362
ArrayList class, 363-366
arrays, 354-362
 Array class, 360-362
 creating, 355
 For Each...Next statement, 359
 initialization, 356
 referencing elements in, 357-358
 specialized sets and generics versus, 372
AS <type> specifier, 128
ASP.NET Web Forms, 491

assemblies, 232
assignment operator, 159
attached properties, 538-541
attribute syntax (XAML), 509
attributes, 148

B

back end, 484
backing fields, 241, 593
bad code, characteristics of, 427
BAML (Binary Application Markup Language), 523-527
base value, determining, 597
BasedOn property, 734-735
behavior, 158, 171, 720
Berra, Yogi, 425
best practices, 13, 16
binding expressions, 798
binding source, 798, 807-810
binding target, 798
BindingMode property, 805
bindings, 797
 binding modes, 805
 binding source, 807-810
 to collections, 814-842
 building bindable collections, 817-818
 data templates, 819-820
 data triggers, 821-822
 filtering collections, 836-842
 master-detail bindings, 823-824
 object sharing, 830-832
 panel templates, 819-820
 sorting collections, 833-835
 value conversions, 829
 views, 825-828
 creating, 802-804
DataContext property, 811
reasons for using, 799-800
structure of, 798
template bindings, 779-780
 update triggers, 806
bitmap effects, 712
block elements, 708
blurs, 713
Boole, George, 165
Boolean expressions, 165-169
bounding boxes for gradients, 682
boxing, 295
Break All command (break mode), 104

break mode, 101-106
breakpoints, setting, 102
Brookes, Frederick P., 20
brushes, 677-697
 creating explorer windows for, 678-680
gradients, 681-690
 bounding boxes for, 682
 linear gradients, 683-687
 radial gradients, 688-690
tile brushes, 691-696
types of, 677
bubbling events, 634
bugs. *See also* exceptions
break mode, 101-106
Code Editor display of, 99-100
history of terminology, 59
in .NET Framework, 644
number in programs, 93
types of, 96
build configurations, 116-117
building applications, 78
Burger, Andrej, 644
business entities, 487
business layer, 485, 487
business rules, 487
business workflows, 487
Button class, 308
buttons, 565-568
 adding to applications, 97-98
 class hierarchy, 565
 content properties (XAML), 566
 radio button groups, 567-568
ByRef keyword, 301

C

Calendar class, 335
calendars, historical date problems with, 338-339
Call Stack window, 101
callbacks, 594, 602, 614-623
 coerce value callbacks, 616
 property changed callbacks, 617
 registering, 618-623
 validation callbacks, 615
calling methods, 175-176
camel case, 132
CanBe relationship, 386
CanDo relationship, 288, 386
Canvas control, 537

casting, 139, 290
 explicit casting, 292
 implicit casting, 291
is keyword, 293-294
polymorphism and, 397
TryCast() function, 293-294
Type...Is operator, 398-400
change tracking and notification with dependency
 properties, 595
chaotic cohesion, 432
Char structure, 327-328
character data, 327-334
 Char structure, 327-328
 comparing strings, 330
 escaping text, 329
 String class methods, 331-332
 String versus StringBuilder class, 333
CheckBox control, 565
chronological data, 335-345
 DateTime class, 337-340
 DateTime versus DateTimeOffset classes, 336
 DateTimeOffset class, 343-344
 Timespan class, 341-342
Class Designer, 227-238
 abstract classes in, 410
 access modifiers in, 232-234
 enumerations, creating, 282
 fields in, 235
 inheritance in, 237
 properties in, 231, 236
 structures, creating, 276-278
Class Details pane (Class Designer), 236
class diagrams, 29, 225-238
Class Name ComboBox (Code Editor), 640
classes. *See also* FCL (Framework Class Library)
 abstract classes, 404, 409-412
 access modifiers in Class Designer, 232-234
 components of, 224
 control classes, 553-556
 default implementations, extending, 415-416
 definition syntax, 239-240
 degenerate classes, 446
 dependency properties, 602-603
 design principles. *See* design principles
 extension methods, 419-421
 fields, 235, 241-249
 inheritance, 210, 237
 instances and constructors versus, 259
interfaces versus, 289
items controls class hierarchy, 570-571
methods, 250-267
 constructor chaining, 260
 creating, 252-258
 me keyword, 256
 syntax, 254
 types of, 251
modules, 405-408
within .NET Framework, 221-223
OOP principles. *See* OOP principles
properties, 241-249
 accessor access, 246-249
 in Class Designer, 231, 236
 implementation, 242-245
relationship with objects, 221
for routed commands, 663
sealed classes, 402-404, 417-418
sealed members, 404
static classes, 404
static font classes, 707
static members, 404
structures versus, 275
timeline classes, 751
type modifiers. *See* type modifiers
UML class diagrams, 29, 225
virtual members, 404
WPF class hierarchy, 535
ClickOnce, 107-109
client/server applications, 482
CLR (Common Language Runtime), 36
CLR-compliant languages, JIT (just-in-time)
 compilation, 36-39
CMYK color model, 674
Code Complete (McConnell), 184
Code Editor, 62, 86-89, 640
code outlining, 89
code smells, characteristics of, 427
CodePlex open source site, 669
coerce value callbacks, 616
cohesion, 430-437
collection property syntax (XAML), 513-514
collections, 353
 binding to, 814-842
 building bindable collections, 817-818
 data templates, 819-820
 data triggers, 821-822
 filtering collections, 836-842

master-detail bindings, 823-824
object sharing, 830-832
panel templates, 819-820
sorting collections, 833-835
value conversions, 829
views, 825-828

XAML, 513-514

CollectionViews, 798

color, 674-676

- gradients, 681-690
 - bounding boxes for, 682
 - linear gradients, 683-687
 - radial gradients, 688-690

color profiles, 675

Color structure, 675

ColorInterpolationMode property, 687

ComboBox control, 575-576

commands

- in break mode, 104
- control-of-flow commands, 184-204
 - iteration commands, 191-197, 359
 - jump commands, 198-204
 - selection commands, 186-190
 - types of, 185
- exception handling commands, 184, 205-217
 - Exception class, 209-212
 - Try...Catch...Finally command, 208
- routed commands, 629-630, 654-666
 - class hierarchy, 663
 - creating, 665
 - in FCL (Framework Command Library), 656-658
 - hooking up, 660-662
 - input gestures, 664-665
 - logical design, 655, 659
 - types of, 181, 183-184
- comments, 122-123, 141-147
 - line comments, 142-143
 - Task List comments, 144-145
 - XML comments, 146-147
- Common Language Runtime (CLR), 36
- communicational cohesion, 432
- comparing strings, 330
- compartments, 225
- compilation errors, 96-100
- compilation process, 33-41
 - compiled versus interpreted languages, 34-35
 - JIT (just-in-time) compilation, 36-39
- for XAML, 523-527

compiled languages, interpreted languages versus, 34-35

compilers, 17

Concat() method, 331

concatenation operator, 159-160

concurrency, 825

conditional iteration, 192-194

#const directive, 149

constants, 127-128. *See also* declared elements

constructors, 135, 251

- base constructors, 416
- chaining, 260
- classes and instances versus, 259
- default constructors, 258

content controls, 533, 536, 557-569

- buttons, 565-568
- decorators versus, 773
- headered content controls, 563-564
- Window control, 558-562

content properties (XAML), 510, 566

continuation character (.), 124

Continue command (break mode), 104

Continue command (jump commands), 200-202

control classes, 553-556

control contracts, 784-787

control templates, 721, 765, 775-787

- control contracts, 784-787
- example of, 776-778
- necessary control components, 783
- presenters, 781-782
- syntax, 775
- template bindings, 779-780
- visual states, 788-794
- visual transitions, 791-792

control-of-flow commands, 181, 184-204

- iteration commands, 191-197, 359
- jump commands, 198-204
- selection commands, 186-190
- types of, 185

controls. *See* WPF controls

converters, 798

converting

- reference types to/from value types, 295
- values in bindings, 829

Cooper, Alan, 491

counters, 195

CreateInstance() method, 360

currency, 825

Custom Actions Editor, 113

custom events, creating, 647
Custom keyword, 647
customizing
 Visual Studio UI, 83-85
 WPF controls. *See* control templates; graphics

D

 data access layer, 485-486
 data binding, 495
 data contracts, 488
 Data Source window, 68
 data stores, 485
 data templates, 819-820
 Data Tips in break mode, 103
 data triggers, 736, 821-822
 data types. *See* types
 data validation. *See* validation
 database schemas, 29
 DataContext property, 811
 dataset projects, 70
 dates/times. *See also* chronological data
 DateTime class, 335-340
 DateTimeOffset class, 335-336, 343-344
 debug build configuration, 116
 DEBUG constant, 151
 Decimal type, 321, 324
 declarations, 122-123
 of arrays, 355
 of events, 646-647
 of methods, 254
 of namespaces, 515-516
 syntax, 128-130
 declared elements, 123
 components of, 134
 declaring, 128-130
 memory allocation, 133
 New keyword, 135
 Option Infer, 138-139
 scope, 262-265
 types of, 127
 decorators, 769, 773-774
 default constructors, 258
 default implementations, extending, 415-416
 default settings, Visual Basic language elements, 136-137
 degenerate classes, 446
 delegates, 615
 dependency inversion, 441-444, 463
 dependency properties, 591

 callbacks, 614-623
 coerce value callbacks, 616
 property changed callbacks, 617
 registering, 618-623
 validation callbacks, 615
 capabilities of, 595
 classes, 602-603
 creating, 604-609
 registering, 610-612
 reusing, 613
 routed events versus, 628, 645
 value calculation, 596-601
 DependencyProperty class, 602-603
 deployment, 75, 107-119
 adding shortcuts to setup programs, 114
 build configurations, 116-117
 ClickOnce, 108-109
 creating setup programs, 110-112
 setup designers, 113
 types of, 107
 Dequeue() method, 367
 design documents, 71
 design patterns, 16, 457
 architectural patterns, 482-491
 business layer, 487
 data access layer, 486
 logical layers, list of, 485
 presentation layer, 491
 service layer, 488-490
 types of, 482-484
 Observer pattern, 469-481
 event mapping, 631
 handling events, 477-479
 logical observer, 470-471
 .NET solution, 475
 raising events, 476
 solution options, 472-474
 presentation patterns, 492-498
 Model-View-Controller pattern, 495
 Model-View-Presenter pattern, 496
 Model-View-ViewModel pattern, 497
 pros and cons, 493-494
 reasons for using, 460-461
 Strategy pattern, 463-468
 Design Patterns: Elements of Reusable Object-Oriented Software (Gamma et al.), 457
 design principles
 characteristics of bad code, 427

Law of Demeter, 449-453
Liskov Substitution Principle, 445-448
Open/Closed Principle, 438-444
Single Responsibility Principle, 431-437
when to violate, 428-429

designer files, 71
design-time, 34
destructors, 251
development environments, 11
development methodologies, 13, 16
 Agile Development, 20
 waterfall models, 20
device independent pixel, 540
diagrams. *See UML (Unified Modeling Language)*
Dictionary class, 367
Dim keyword, 128
direct events, 630, 634
directives, 122-123, 148-151
DirectX, 712
“Discover a Series of Fortunate Event Handlers in Visual Basic” (Getz), 647
DivideByZeroException class, 210
dll files, 35
DockPanel control, 537, 544-546
document outline window, 68
documents, flow, 708-711
Do...Loop command, 192-194
dot operator, 172
Double type, 321, 323
downloading
 source code, 6
 Visual Studio, 6
drawing shapes, 770-772
drop shadows, 713
dynamic event handlers, 479
dynamic resources, 727-728

E

Edison, Thomas, 59
effects (graphics), 712-714
Einstein, Albert, 425
embedding fonts, 707
empty bindings, 814
encapsulation, 384, 391, 393
encoding, 327
ENIAC, 59
Enqueue() method, 367
enum keyword, 284

enumerations, 269, 274, 281-287
 creating in Class Designer, 282
 methods in, 284-287
 within .NET Framework, 270-271
 syntax, 283
 validation, 283
equations, expressions versus, 160-162
Error List window, 100
errors. *See bugs*
escaping text, 329
event handlers, 57
 adding to buttons, 98
 for routed events, 639-641
 event triggers, 736, 744-746
events, 25, 224
 adding to windows, 57
 bubbling events, 634
 custom events, creating, 647
 declaring, 646-647
 direct events, 630, 634
 handling, 477-479
 mapping to Observer pattern, 631
 raising, 476
 routed events, 627, 632-653
 arguments, 635-644
 creating, 648-653
 dependency properties versus, 628, 645
 reasons for using, 632-633
 strategies, 634
 tunneling events, 634
Exception class, 209-212
exception handling commands, 181, 184, 205-217
 Exception class, 209-212
 Try...Catch...Finally command, 208
exceptions, 87, 96. *See also bugs*
exe files, 35
executables, creating, 33-41
 compiled versus interpreted languages, 34-35
 JIT (just-in-time) compilation, 36-39
Exit command, 199
Expander control, 563
explicit casting, 292
explicit iteration, 195-196
explorer windows, creating, 678-680
expressions, 98, 155
 literal expressions, 158-170
 Boolean expressions, 165-169
 equations versus, 160-162

- operator precedence, 163-164
syntax, 159
- object expressions, 158, 171-178
calling methods, 175-176
member access, 172
method signatures, 173
overloading methods, 174
type names, 177
types of, 157-158
- extending default implementations, 415-416
- extension methods, 407, 419-421
- F**
- FCL (Framework Class Library)
- arrays, 354-362
 - Array class, 360-362
 - creating, 355
 - For Each...Next statement, 359
 - initialization, 356
 - referencing elements in, 357-358
 - character data, 327-334
 - Char structure, 327-328
 - comparing strings, 330
 - escaping text, 329
 - String class methods, 331-332
 - String versus StringBuilder class, 333
 - chronological data, 335-345
 - DateTime class, 337-340
 - DateTime versus DateTimeOffset classes, 336
 - DateTimeOffset class, 343-344
 - Timespan class, 341-342
 - functionality example, 308
 - generics, 372-377
 - namespaces, 310-316
 - creating, 315
 - Imports statement, 312-314
 - numeric data, 321-326
 - Decimal type, 324
 - floating point numbers, 323
 - integer efficiency tips, 322
 - Math class, 325
 - Object Browser, 317-320
 - organization, 306-307, 350-351
 - specialized sets, 363-371
 - ArrayList class, 363-366
 - LIFO, FIFO, linked lists, key/value pairs, 367-371
- FCL (Framework Command Library), routed commands in, 656-658
- fields, 224, 241-249
 - in Class Designer, 235
 - dependency properties and, 592
 - naming conventions, 236
- FIFO (first in, first out), 367-371
- File System Designer, 113
- File Type Editor, 113
- files, adding to projects, 70-73
- filtering collections, 836-842
- fixed documents, 708
- floating point numbers
 - integers versus, 139
 - precision, 323
- flow control. *See* control-of-flow commands
- flow controls, 711
- flow documents, 708-711
- FlowDocumentPageViewer control, 711
- FlowDocumentReader control, 711
- fonts, 704
 - embedding, 707
 - static font classes, 707
- For Each...Next statement, 359
- For...Next command, 195-196
- fragile code, 427
- Framework Class Library. *See* FCL (Framework Class Library)
- frameworks. *See* .NET Framework
- Fraser, Bruce, 676
- front end, 484
- functional cohesion, 432
- functions, 173
- G**
- gamut, 675
- garbage collection, 265
- generics, 353, 372-377
 - arrays and specialized sets versus, 372
 - instantiation, 373
- gestures, input, 664-665
- get accessors, 241-249
- Getz, Ken, 647
- global variables, 407
- glyph runs, 691
- glyphs, 704
- GMT (Greenwich Mean Time), 336
- GoTo command, 203-204
- gradients, 681-690
 - bounding boxes for, 682

linear gradients, 683-687
radial gradients, 688-690
graphics, 669-671
brushes, 677-697
 creating explorer windows for, 678-680
 gradients, 681-690
 tile brushes, 691-696
 types of, 677
color, 674-676
effects, 712-714
flow documents, 708-711
Mindbloom website versus Microsoft Word, 672
pens, 698-702
typography, 703-707
 fonts and typefaces, 704
 static font classes, 707
 text characteristics, 705
 text controls, 706
Greenwich Mean Time (GMT), 336
Gregorian calendar, 339
Grid class, 308
Grid control, 537, 547-548
GridSplitter control, 549-550
GroupBox control, 563
grouping collections, 836-842

H

hacks, 144
Handles keyword, 477, 641
handling events, 477-479
Harvard Mark II, 59
HasA relationship, 288, 386
Hashtable class, 367
headered content controls, 563-564
heap, 272-273
height of panel controls, 543
“Hello, World” example, 53-58
High Level Shading Language (HLSL), 712
historical dates, 338-339
HLSL (High Level Shading Language), 712
horizontal alignment of panel controls, 542
HSV color model, 674
Hungarian notation, 132

I

ICollection interface, 364
icons, adding to applications, 76-82
IDE (integrated development environment), 17
ideas, 10
identifiers, 128
 as expressions, 157
 naming conventions, 131-132
IEnumerable interface, 364
if statement, 186-188
#if...#else...#endif directive, 149
IList interface, 364
Immediate window, 101
immobile code, 427
immutability, 333
immutable properties, 247
implementation of properties, 242-245
implicit casting, 138-139, 291
implicit styles, 731
Imports statement, 312-314
incompatible contracts, 447
indexes, 354
information hiding, 241-249, 393
inheritance, 225, 384, 391-392
 in Class Designer, 237
 in Exception class, 210
inheritance casts, 291
initialization, 73
 arrays, 356
 of decimal values, 324
 with New keyword, 135
inline elements, 708
input gestures, 664-665
instances, 221, 251, 259
instantiation, 221
 generics, 373
 structures, 279
Int32 class, 308
Int32 type, 321
integers
 efficiency tips, 322
 floating point numbers versus, 139
integrated development environment (IDE), 17
Intellisense, 57, 62, 86-88, 146-147
Intellitrace commands (break mode), 104
interfaces, 269, 274, 288-289
 abstract interfaces, 409-412
 for ArrayList class, 364
 classes versus, 289
 within .NET Framework, 270-271
 programming to interfaces, 442, 463
 relationships and, 288

service interfaces, 488
internal access modifier, 232
interpreted languages, compiled languages versus, 34-35
invoking methods, 175-176
is keyword, 293-294
IsA relationship, 288, 386
items controls, 533, 536, 570-576
 binding to collections, 814-842
 building bindable collections, 817-818
 data templates, 819-820
 data triggers, 821-822
 filtering collections, 836-842
 master-detail bindings, 823-824
 object sharing, 830-832
 panel templates, 819-820
 sorting collections, 833-835
 value conversions, 829
 views, 825-828
class hierarchy, 570-571
ComboBox control, 575-576
ListBox control, 575-576
TreeView control, 572-574
iteration commands, 183, 185, 191-197
 conditional iteration, 192-194
 For Each...Next statement, 359
 explicit iteration, 195-196

J

jagged arrays, 354
JIT (just-in-time) compilation, 36-39
Julian calendar, 339
jump commands, 183, 185, 198-204
 Continue command, 200-202
 Exit command, 199
 GoTo command, 203-204

K

key frame animations, 751
keyed styles, 730
key/value pairs, 367-371
keywords, 131, 223
kludges, 144

L

lambda, 157
Launch Conditions Editor, 113
Law of Demeter, 430, 449-453
layered architecture. *See* logical layers

layout transforms, 758
LIFO (last in, first out), 367-371
line comments, 142-143
linear gradients, 681-687
lines, drawing, 698-702
linked lists, 367-371
linkers, 17
linking, 35
LINQ expressions, 157
Lippert, Eric, 418
Liskov Substitution Principle, 430, 445-448
list members, 87
ListBox control, 575-576
ListView class, 571
literal expressions, 158-170
 Boolean expressions, 165-169
 equations versus, 160-162
 operator precedence, 163-164
syntax, 159
local values, 597
Locals window, 101
logical errors, 96
logical layers
 business layer, 487
 data access layer, 486
 list of, 485
 presentation layer, 491
 service layer, 488-490
logical operators, 166
logical tree, 506, 519, 766
loops. *See* iteration commands
LSP. *See* Liskov Substitution Principle

M

margin of panel controls, 542-543
master-detail bindings, 823-824
Math class, 325
MatrixTransform class, 756
Max() method, 325
McConnell, Steve, 184
me keyword, 256
measurement units, device independent pixel, 540
member access, 172
memory allocation, 133
 garbage collection, 265
 reference and value types, 272-273
 scope, 262-265
 in structures versus classes, 275

meta types. *See* character data; chronological data; numeric data
method calls, 175
Method Name ComboBox (Code Editor), 640
methods, 73, 88, 224, 250-267
 for Array class, 360
 calling, 175-176
 cohesion levels, 432
 constructor chaining, 260
 constructors, classes and instances versus, 259
 creating, 252-258
 for Decimal type, 324
 in enumerations, 284-287
 extension methods, 407, 419-421
 in Math class, 325
 me keyword, 256
 overloading, 174
 passing by reference/value, 299-301
 signatures, 173
 static methods, 176, 479
 in String class, 331-332
 syntax, 254
 types of, 251
Microsoft Intermediate Language (MSIL), 36
Microsoft Word, Mindbloom website versus, 672
Min() method, 325
Mindbloom website, Microsoft Word versus, 672
Model-View-Controller pattern, 495
Model-View-Presenter pattern, 496
Model-View-ViewModel pattern, 459, 497
modifiers. *See* access modifiers; type modifiers
modules, 402, 405-408
modulus operator, 161
monolithic applications, 482
MSIL (Microsoft Intermediate Language), 36
multi-dimensional arrays, 354
multi-line statements, 124
multiple inheritance, 392
multiple transformations, 758-759
multiple trigger conditions, 739-740
multi-statement lines, 124
MustInherit keyword, 403, 410
MustOverride keyword, 403, 410
mutability, 333
MVC (Model-View-Controller) pattern, 495
MVP (Model-View-Presenter) pattern, 496
MVVM (Model-View-ViewModel) pattern, 497
 MyBase keyword, 415-416

The Mythical Man Month (Brookes), 20
N
namespaces, 310-316
 creating, 315
Imports statement, 312-314
XAML, 515-516
naming conventions
 abstract classes, 409
 fields and properties, 236
 identifiers, 131-132
 variables, 264
 Visual Basic versus .NET type names, 177
Nash, John, 144
negation operator, 160
nesting panel controls, 551
.NET Framework, 43
 advantages of, 50-52
 bugs in, 644
Class Library organization, 306-307, 350-351
classes within, 222-223
enumerations within, 270-271
interfaces within, 270-271
structures within, 270-271
type names, 177
types. *See* types
.NET Platform, 43-65
 components of, 44-49
 “Hello, World” example, 53-58
New keyword, 135
Norman, Donald, 491
NotImplementedException class, 210
NotOverridable keyword, 402
NotOverrideable keyword, 417-418
n-tiered applications, 482
numeric data, 321-326
 Decimal type, 324
 floating point numbers, 323
 integer efficiency tips, 322
 Math class, 325
O
Object Browser, 85, 317-320
object code, 34
object composition, 386
object elements (XAML), 509
object expressions, 158, 171-178
 calling methods, 175-176

member access, 172
method signatures, 173
overloading methods, 174
type names, 177
object hierarchy, 311
object sharing, 830-832
object tree, 506, 519
object-oriented programming principles. *See* OOP principles
objects
 components of, 171
 referencing from XAML, 831
 relationship with classes, 221
 state and behavior, 720
Observer pattern, 459, 469-481
 event mapping, 631
 handling events, 477-479
 logical observer, 470-471
.NET solution, 475
 raising events, 476
 solution options, 472-474
OneTime binding, 805
OneWay binding, 805
OneWayToSource binding, 805
OOA&D (Object-Oriented Analysis & Design), 381-383
 design principles
 characteristics of bad code, 427
 Law of Demeter, 449-453
 Liskov Substitution Principle, 445-448
 Open/Closed Principle, 438-444
 Single Responsibility Principle, 431-437
 when to violate, 428-429
OOP principles, 391-401
 polymorphism and casting, 397
 Type...Is operator, 398-400
type modifiers, 402-419
 abstract classes and interfaces, 409-412
 extension methods, 419-421
 list of, 402-404
 modules, 405-408
 MyBase keyword, 415-416
 sealed classes, 417-418
 semi-abstract classes, 413
 Shadows and Overrides keywords, 413-414
type relationships, 385-390
OOP principles, 391-401
 polymorphism and casting, 397
 Type...Is operator, 398-400
opaque code, 427
Open/Closed Principle, 430, 438-444
opening Project Designer, 75
operands, 159
operators, 159
 dot operator, 172
 logical operators, 166
 precedence, 163-164
 relational operators, 165
 symbolic operators, 175
Option Compare, 137
Option Explicit, 137
Option Infer, 137-139
Option Strict, 137
ordinal string operations, 330
OverflowException class, 210
overloading methods, 174
overloads, 174, 257
Overridable keyword, 403
Overrides keyword, 413-414

P

padding of panel controls, 542-543
panel controls, 533, 536-552
 attached properties, 538-541
 decorators versus, 773
 DockPanel control, 544-546
 Grid control, 547-548
 GridSplitter control, 549-550
 nesting, 551
 positioning and sizing, 542-543
panel templates, 819-820
paragraphs, 124
parameters, 88, 173, 253
Pascal case, 132
passing by reference, 299-301
passing by value, 299-301
A Pattern Language (Alexander), 457
patterns. *See* design patterns
pens, 698-702
pixel shaders, 671
POCO (plain old CLR object), 592, 630
polymorphism, 384, 391, 393
 casting and, 397
 Type...Is operator, 398-400
Pop() method, 367
positioning panel controls, 542-543
precedence of operators, 163-164

precision, floating point numbers, 323
predicates, 360, 837
preprocessor directives, 148. *See also* directives
 preprocessors, 148
 presentation layer, 485, 491. *See also* presentation patterns
 presentation patterns, 492-498
 Model-View-Controller pattern, 495
 Model-View-Presenter pattern, 496
 Model-View-ViewModel pattern, 497
 pros and cons, 493-494
 presenters, 767, 781-782
 primary colors, 676
 primary UI projects, 70
 Principle of Least Knowledge, 430, 449-453
 private access modifier, 232
 processing pipeline, 596
 programming to interfaces, 442, 463
 Project Designer, opening, 75
 Project Properties designer, 113
 projects. *See also* application development
 adding files to, 70-73
 adding to solutions, 70-73
 creating, 54, 60
 properties, 74-82
 startup project, specifying, 74
 steps in, 59-63
 properties, 224, 241-249
 accessor access, 246-249
 attached properties, 538-541
 in Class Designer, 231, 236
 content properties (XAML), 510, 566
 dependency properties, 591
 callbacks, 614-623
 capabilities of, 595
 classes, 602-603
 creating, 604-609
 registering, 610-612
 reusing, 613
 value calculation, 596-601
 implementation, 242-245
 for linear gradients, 687
 naming conventions, 236
 of solutions and projects, 74-82
 Properties window, 56-57, 68
 property changed callbacks, 617
 property element syntax (XAML), 509
 property triggers, 736-743
 combining with styles, 741-742
 multiple trigger conditions, 739-740
 PropertyMetadata class, 602-603
 protected access modifier, 232
 protected internal access modifier, 232
 public access modifier, 232
 publishing. *See* deployment
 Push() method, 367

Q

Queue class, 367

R

radial gradients, 681, 688-690
radio button groups, 567-568
RadioButton class, 308
RadioButton control, 565
raising events, 476
range controls, 533
read-only properties, 247
Real World Color Management (Fraser), 676
redundant code, 427
reference highlighting, 89
reference types, 272-273, 275
 converting to/from value types, 295
 value types versus, 296-301
referencing
 array elements, 357-358
 objects from XAML, 831
 resources, 727-728, 830
 XAML elements, 830
reflection, 290
registering
 callbacks, 618-623
 dependency properties, 610-612
Registry Editor, 113
relational operators, 165
relationships, 288, 385-390
RelativeSource class, 809-810
release build configuration, 116
RemoveHandler keyword, 479
render transforms, 759
resources, 71, 526, 719, 721
 advantages and disadvantages of, 722
 in code, 729
 defining, 724-726
 referencing, 727-728, 830
REST (representational state transfer), 488
Restart command (break mode), 104

return types, 173
reusing dependency properties, 613
RGB color model, 674
RichTextBox control, 711
rigid code, 427
root element (XAML), 515
RotateTransform class, 756
Round() method, 324-325
routed commands, 629-630, 654-666
 class hierarchy, 663
 creating, 665
 in FCL (Framework Command Library), 656-658
 hooking up, 660-662
 input gestures, 664-665
 logical design, 655, 659
routed events, 627, 632-653
 arguments, 635-644
 creating, 648-653
 dependency properties versus, 628, 645
 reasons for using, 632-633
 strategies, 634
running applications, 58
runtime, 34
runtime environments, 10, 17

§

safe casts, 291
scope, 262-265
screen layouts, 29
ScRGB color space, 687
sealed classes, 402, 404, 417-421
sealed members, 404
select...case statement, 189-190
selection commands, 183, 185-190
 if statement, 186-188
 select...case statement, 189-190
Selector class, 571
Sells, Chris, 418
separation of responsibilities, 493
sequential cohesion, 432
servers, 484
service interfaces, 488
service layer, 485, 488-490
service-oriented architecture, 484
services, 484
set accessors, 241-249
sets. *See arrays; generics; specialized sets*
setters, 597

setup designers, 113
setup programs, 107
 adding shortcuts to, 114
 build configurations, 116-117
 creating, 110-112
Shadows keyword, 413-414
shapes, 769-772
shared keyword, 402
sharing objects, 830-832
shortcuts, adding to setup programs, 114
Show Next Statement command (break mode), 104
side effects, 166
signatures of methods, 173
Silverlight, 491, 503
Simonyi, Charles, 132
simple statements, 124
Single Responsibility Principle, 430-437
Single type, 323
single-dimensional arrays, 354
singleton pattern, 407
sizing panel controls, 542-543
SkewTransform class, 757
Snepscheut, Jan van de, 425
SOA (service-oriented architecture), 488
SOAP (simple object access protocol), 488
Solution Explorer, 61, 68
solutions, 67
 adding projects to, 70-73
 properties, 74-82
sorting collections, 833-835
source code, 11, 17
 Code Editor, 62, 86-89
 compiled versus interpreted languages, 34-35
 downloading, 6
 JIT (just-in-time) compilation, 36-39
 steps in writing, 59-63
source files, 71
sparse storage with dependency properties, 595
specialized sets, 363-371
 ArrayList class, 363-366
 arrays and generics versus, 372
 LIFO, FIFO, linked lists, key/value pairs, 367-371
specifications, 10, 16, 19-32
 Agile Development, 20
 database schemas, 29
 screen layouts, 29
 UML class diagrams, 29
 UML state diagrams, 25-28

use cases, 21-24
waterfall models, 20

SpreadMethod property, 687

square brackets in syntax diagrams, 128

sRGB color space, 687

stack, 272-273, 367

StackPanel control, 537

startup project, specifying, 74

state, 158, 171, 720

state diagrams, 25-28

state members. *See* fields

statements, 17, 122-123

- class definitions, 239-240
- commands. *See* commands
- declared elements. *See* declared elements
- expressions. *See* expressions

syntax, 124-126

states, 25

static classes, 404. *See also* modules

static event handlers, 477

static font classes, 707

static members, 402, 404

static methods, 176, 251, 479

static resources, 727-728

Step Into command (break mode), 104

Step Out command (break mode), 104

Step Over command (break mode), 104

Stop Debugging command (break mode), 104

stops, 681

storyboards, 749-750

Strategy pattern, 459, 463-468

String class, 308

- comparing strings, 330
- escaping text, 329
- methods, 331-332

StringBuilder class versus, 333

StringBuilder class, String class versus, 333

StringComparison enumeration, 330

strong typing, 139

structured exception handling, 207-217

structures, 269, 274-280

- classes versus, 275
- creating in Class Designer, 276-278
- instantiation, 279
- within .NET Framework, 270-271

styles, 597, 721, 730-735

- combining with triggers, 741-742
- hierarchies in, 734-735

types of, 730-731

subs, 173

symbolic operators, 175

syntactic sugar, 135

syntax errors, 96, 99-100

system architectures, 13

system design, 19-32

- Agile Development, 20
- database schemas, 29
- screen layouts, 29
- UML class diagrams, 29
- UML state diagrams, 25-28
- use cases, 21-24
- waterfall models, 20

SystemException class, 210

T

tab controls, 577

targeted styles, 730

Task List comments, 144-145

template bindings, 797

templates, 597

- control templates, 721, 765, 775-787
- control contracts, 784-787
- example of, 776-778
- necessary control components, 783
- presenters, 781-782
- syntax, 775
- template bindings, 779-780
- visual states, 788-794
- visual transitions, 791-792
- data templates, 819-820
- panel templates, 819-820

temporal cohesion, 432

test projects, 70

test-driven development, 70

text. *See* character data; typography

text editors, 17

TextBox class, 308

TextDecorations collection, 700-702

themes, 597

ticks, 337

tile brushes, 691-696

The Timeless Way of Building (Alexander), 457

timeline classes, 751

times/dates. *See also* chronological data

Timespan class, 335, 341-342

TimeZoneInfo class, 335

ToggleButton control, 565
tokens, 144-145
Toolbox, 68
TRACE constant, 151
transform groups, 758
transformations, 671, 756-761
transitions, 25, 791-792
TranslateTransform class, 757
TreeView control, 572-574
triggers, 597, 721, 736-746
 data triggers, 821-822
 event triggers, 744-746
 property triggers, 736-743
 combining with styles, 741-742
 multiple trigger conditions, 739-740
 types of, 736
 update triggers, 806
true keyword, 160
Truncate() method, 324-325
TryCast() function, 293-294
Try...Catch...Finally command, 208
tunneling events, 634
TwoWay binding, 805
type keywords, 223
type modifiers, 402-419
 abstract classes and interfaces, 409-412
 extension methods, 419-421
 list of, 402-404
 modules, 405-408
 MyBase keyword, 415-416
 sealed classes, 417-418
 Shadows and Overrides keywords, 413-414
type relationships. *See* relationships
typefaces, 704
Type...Is operator, 398-400
types, 158, 171, 290-302. *See also* classes; enumerations;
 FCL (Framework Class Library); interfaces; structures
arrays, 354-362
 ArrayList class, 363-366
 creating, 355
 For Each...Next statement, 359
 initialization, 356
 referencing elements in, 357-358
boxing and unboxing, 295
casting, 290
 explicit casting, 292
 implicit casting, 139, 291
 is keyword, 293-294
polymorphism and, 397
TryCast() function, 293-294
Type...Is operator, 398-400
character data, 327-334
 Char structure, 327-328
 comparing strings, 330
 escaping text, 329
 String class methods, 331-332
 String versus StringBuilder class, 333
chronological data, 335-345
 DateTime class, 337-340
 DateTime versus DateTimeOffset classes, 336
 DateTimeOffset class, 343-344
 Timespan class, 341-342
generics, 372-377
namespaces, 310-316
 creating, 315
 Imports statement, 312-314
numeric data, 321-326
 Decimal type, 324
 floating point numbers, 323
 integer efficiency tips, 322
 Math class, 325
reference and value types, 272-273, 296-301
reflection, 290
relationship with classes, 222-223
specialized sets, 363-371
 ArrayList class, 363-366
 LIFO, FIFO, linked lists, key/value pairs, 367-371
Visual Basic versus .NET type names, 177
typography, 703-707
fonts and typefaces, 704
static font classes, 707
text characteristics, 705
text controls, 706

U

UI (user interface) in Visual Studio, 61, 67-91. *See also* graphics; WPF controls
adding solutions and projects, 70-73
Code Editor, 86-89
customizing, 83-85
parts of, 68
solution and project properties, 74-82
UML (Unified Modeling Language)
 UML class diagrams, 29, 225, 227-238
 UML state diagrams, 25-28
unboxing, 295

Unicode, 327-328
Unified Modeling Language. *See* UML (Unified Modeling Language)
units of measurement, device independent pixel, 540
universal resource identifier (URI), 526
Until keyword, 192-194
update triggers, 806
URI (universal resource identifier), 526
use cases, 20-28
user interface. *See* UI (user interface) in Visual Studio
User Interface Editor, 113
UTC (Universal Time, Coordinated), 336
utility classes, 70

V

validation
 callbacks, 615
 enumerations, 283
value calculation with dependency properties, 595-601
value conversions, 829
value types, 272-273, 275
 converting to/from reference types, 295
 reference types versus, 296-301
variables, 73, 98, 127. *See also* declared elements
 garbage collection, 265
 global variables, 407
 implicit typing, 138-139
 naming conventions, 264
 passing by reference/value, 299-301
 scope, 262-265
vertical alignment of panel controls, 542
Viewbox instance, 692-693
Viewport instance, 692-693
views, 825-828
virtual keyword, 413
virtual members, 403-404
visibility, 232
Visual Basic language elements
 commands
 control-of-flow commands, 184-204
 exception handling commands, 184
 types of, 181, 183-184
 comments, 141-147
 line comments, 142-143
 Task List comments, 144-145
 XML comments, 146-147
 declared elements
 components of, 134

declaring, 128-130
memory allocation, 133
New keyword, 135
Option Infer, 138-139
 types of, 127
default settings, 136-137
directives, 148-151
expressions, 155
 literal expressions, 158-170
 object expressions, 158, 171-178
 types of, 157-158
identifiers, naming conventions, 131-132
list of, 122, 156, 182
statements
 class definitions, 239-240
 syntax, 124-126
 type names, 177
Visual Basic, XAML versus, 507-508
visual states, 788-794
Visual Studio
 Class Designer, 227-238
 abstract classes in, 410
 access modifiers in, 232-234
 enumerations, creating, 282
 fields in, 235
 inheritance in, 237
 properties in, 231, 236
 structures, creating, 276-278
 Code Editor, 62
 downloading, 6
 “Hello, World” example, 53-58
 projects, creating, 54, 60
 UI (user interface), 61, 67-91
 adding solutions and projects, 70-73
 Code Editor, 86-89
 customizing, 83-85
 parts of, 68
 solution and project properties, 74-82
 windows, creating, 55-56
visual transitions, 791-792
visual tree, 506, 519, 767

W

Watch window, 101
waterfall models, 20
WCF (Windows Communication Foundation), 488
While keyword, 192-194
widgets, adding to windows, 55-56. *See also* WPF

controls
width of panel controls, 543
Window class, 308
Window control, 558-562
windows. *See also* UI (user interface) in Visual Studio
 arranging, 84
 creating, 55-56
 logical tree, 766
Windows Forms (WinForms), 491
WithEvents keyword, 477
WPF (Windows Presentation Foundation), 501
 animations, 747-755
 class hierarchy, 748
 example of, 752-755
 reasons for using, 747
 storyboards, 749-750
 timeline classes, 751
 bindings, 797
 binding modes, 805
 binding source, 807-810
 to collections, 814-842
 creating, 802-804
 DataContext property, 811
 reasons for using, 799-800
 structure of, 798
 update triggers, 806
 class hierarchy, 535
 control templates, 721, 765, 775-787
 control contracts, 784-787
 example of, 776-778
 necessary control components, 783
 presenters, 781-782
 syntax, 775
 template bindings, 779-780
 visual states, 788-794
 visual transitions, 791-792
 dependency properties, 591
 callbacks, 614-623
 capabilities of, 595
 classes, 602-603
 creating, 604-609
 registering, 610-612
 reusing, 613
 value calculation, 596-601
graphics, 669-671
 brushes, 677-697
 color, 674-676
 effects, 712-714
flow documents, 708-711
pens, 698-702
typography, 703-707
hierarchies in, 506, 519-522
resources, 719, 721
 advantages and disadvantages of, 722
 in code, 729
 defining, 724-726
 referencing, 727-728
routed commands, 629-630, 654-666
 class hierarchy, 663
 creating, 665
 in FCL (Framework Command Library), 656-658
 hooking up, 660-662
 input gestures, 664-665
 logical design, 655, 659
routed events, 627, 632-653
 arguments, 635-644
 creating, 648-653
 dependency properties versus, 628, 645
 reasons for using, 632-633
 strategies, 634
styles, 597, 721, 730-735
 hierarchies in, 734-735
 types of, 730-731
themes, 597
transformations, 756-761
triggers, 721, 736-746
 event triggers, 744-746
 property triggers, 736-743
 types of, 736
WPF controls, 531-533
application creation example, 578-587
building, 769-774
content controls, 557-569
 buttons, 565-568
 headered content controls, 563-564
 Window control, 558-562
control classes, 553-556
control contracts, 784-787
customizing. *See* graphics; templates
flow controls, 711
items controls, 570-576
 class hierarchy, 570-571
 ComboBox control, 575-576
 ListBox control, 575-576
 TreeView control, 572-574
necessary components, 783

panel controls, 537-552
 attached properties, 538-541
 DockPanel control, 544-546
 Grid control, 547-548
 GridSplitter control, 549-550
 nesting, 551
 positioning and sizing, 542-543
tab controls, 577
visual tree, 767
WPF Designer, 55-56, 61, 68
WrapPanel control, 537
wrapping, 593
write-only properties, 247

X

XAML, 55, 501. *See also* WPF (Windows Presentation Foundation)
 attributes and properties, 509
 collections, 513-514
 compilation process, 523-527
 content properties, 510-512
 event handlers, 639
 example of, 505
 namespaces, 515-516
 need for, 504
 object elements, 509
 referencing objects from, 831
 root element, 515
 Visual Basic versus, 507-508
 WPF trees, 519-522

XML

 comments, 146-147
 need for, 504

Z

Zulu, 336

This page intentionally left blank

