

Table of Contents (Summary)

	Intro	xxix
1	Get productive with C#: <i>Visual Applications, in 10 minutes or less</i>	1
2	It's All Just Code: <i>Under the hood</i>	43
3	Objects Get Oriented: <i>Making code make sense</i>	85
4	Types and References: <i>It's 10:00. Do you know where your data is?</i>	123
	C# Lab 1: <i>A Day at the Races</i>	163
5	Encapsulation: <i>Keep your privates... private</i>	173
6	Inheritance: <i>Your object's family tree</i>	205
7	Interfaces and abstract classes: <i>Making classes keep their promises</i>	251
8	enums and collections: <i>Storing lots of data</i>	309
	C# Lab 2: <i>The Quest</i>	363
9	Reading and writing files: <i>Save the byte array, save the world</i>	385
10	Exception handling: <i>Putting Out Fires Gets Old</i>	439
11	events and delegates: <i>What Your Code Does When You're Not Looking</i>	483
12	Review and preview: <i>Knowledge, Power, and Building Cool Stuff</i>	515
13	Controls and graphics: <i>Make it pretty</i>	563
14	Captain Amazing: <i>The Death of the Object</i>	621
15	LINQ: <i>Get control of your data</i>	653
	C# Lab 3: <i>Invaders</i>	681

Table of Contents (the real thing)

Intro

Your brain on C#. You're sitting around trying to *learn* something, but your *brain* keeps telling you all that learning *isn't important*. Your brain's saying, "Better leave room for more important things, like which wild animals to avoid and whether nude archery is a bad idea." So how *do* you trick your brain into thinking that your life really depends on learning C#?

Who is this book for?	xxx
We know what you're thinking	xxxix
Metacognition	xxxix
Bend your brain into submission	xxxv
What you need for this book	xxxvi
Read me	xxxii
The technical review team	xxxiv
Acknowledgments	xxxv

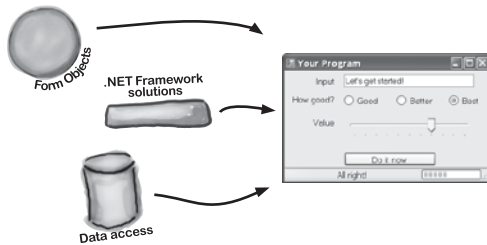
get productive with C#

1

Visual Applications, in 10 minutes or less

Want to build great programs really fast?

With C#, you've got a **powerful programming language** and a **valuable tool** at your fingertips. With the **Visual Studio IDE**, you'll never have to spend hours writing obscure code to get a button working again. Even better, you'll be able to **focus on getting your work done**, rather than remembering which method parameter was for the *name* for a button, and which one was for its *label*. Sound appealing? Turn the page, and let's get programming.



Why you should learn C#	2
C# and the Visual Studio IDE make lots of things easy	3
Help the CEO go paperless	4
Get to know your users' needs before you start building your program	5
Here's what you're going to build	6
What you do in Visual Studio...	8
What Visual Studio does for you...	8
Develop the user interface	12
Visual Studio, behind the scenes	14
Add to the auto-generated code	15
You can already run your application	16
We need a database to store our information	18
Creating the table for the Contact List	20
The blanks on contact card are columns in our People table	22
Finish building the table	25
Diagram your data so your application can access it	26
Insert your card data into the database	28
Connect your form to your database objects with a data source	30
Add database-driven controls to your form	32
Good apps are intuitive to use	34
How to turn YOUR application into EVERYONE'S application	37
Give your users the application	38
You're NOT done: test your installation	39
You built a complete data-driven application	40

it's all just code

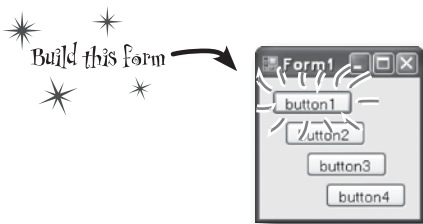
Under the Hood

2

You're a programmer, not just an IDE-user.

You can get a lot of work done using the IDE. But there's only so far it can take you. Sure, there are a lot of **repetitive tasks** that you do when you build an application. And the IDE is great at doing those things for you. But working with the IDE is *only the beginning*. You can get your programs to do so much more—and **writing C# code** is how you do it. Once you get the hang of coding, there's *nothing* your programs can't do.

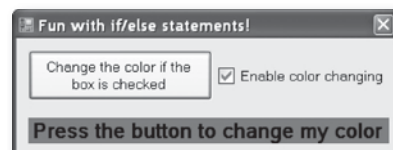
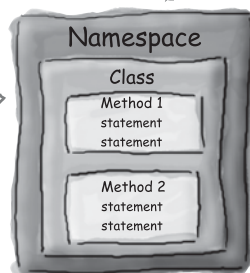
When you're doing this...	44
...the IDE does this	45
Where programs come from	46
The IDE helps you code	48
When you change things in the IDE, you're also changing your code	50
Anatomy of a program	52
Your program knows where to start	54
You can change your program's entry point	56
Two classes can be in the same namespace	61
Your programs use variables to work with data	62
C# uses familiar math symbols	64
Loops perform an action over and over again	65
Time to start coding	66
if/else statements make decisions	67
Set up conditions and see if they're true	68



Every time you make a new program, you define a namespace for it so that its code is separate from the .NET Framework classes.

A class contains a piece of your program (although some very small programs can have just one class).

A class has one or more methods. Your methods always have to live inside a class. And methods are made up of statements – like the ones you've already seen.



objects get oriented

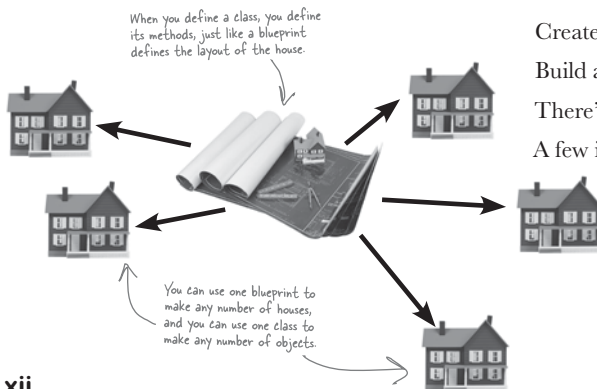
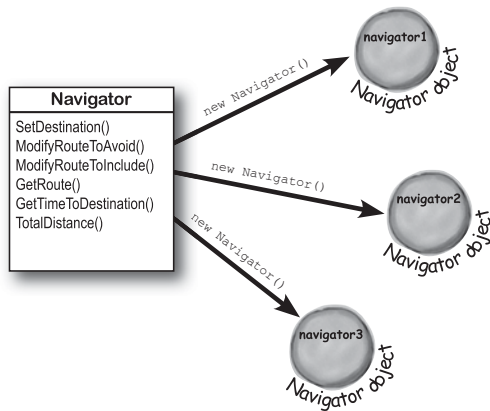
Making Code Make Sense

3

Every program you write solves a problem.

When you're building a program, it's always a good idea to start by thinking about what *problem* your program's supposed to solve. That's why **objects** are really useful. They let you structure your code based on the problem it's solving, so that you can spend your time *thinking about the problem* you need to work on rather than getting bogged down in the mechanics of writing code. When you use objects right, you end up with code that's *intuitive* to write, and easy to read and change.

How Mike thinks about his problems	86
How Mike's car navigation system thinks about his problems	87
Mike's Navigator class has methods to set and modify routes	88
Use what you've learned to build a simple application	89
Mike gets an idea	90
Mike can use objects to solve his problem	91
You use a class to build an object	92
When you create a new object from a class, it's called an instance of that class	93
A better solution... brought to you by objects!	94
An instance uses fields to keep track of things	98
Let's create some instances!	99
Thanks for the memory	100
What's on your program's mind	101
You can use class and method names to make your code intuitive	102
Give your classes a natural structure	104
Class diagrams help you organize your classes so they make sense	106
Build a class to work with some guys	110
Create a project for your guys	111
Build a form to interact with the guys	112
There's an even easier way to initialize objects	115
A few ideas for designing intuitive classes	116



types and references

4

It's 10:00. Do you know where your data is?

Data type, database, Lieutenant Commander Data...

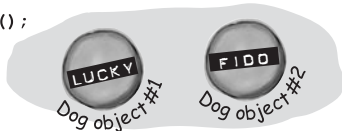
it's all important stuff. Without data, your programs are useless. You need **information** from your users, and you use that to look up or produce new information, to give back to them. In fact, almost everything you do in programming involves **working with data** in one way or another. In this chapter, you'll learn the ins and outs of C#'s **data types**, how to work with data in your program, and even figure out a few dirty secrets about **objects** (*psstt... objects are data, too*).

The variable's type determines what kind of data it can store	124
A variable is like a data to-go cup	126
10 pounds of data in a 5 pound bag	127
Even when a number is the right size, you can't just assign it to any variable	128
When you cast a value that's too big, C# will adjust it automatically	129
C# does some casting automatically	130
When you call a method, the variables must match the types of the parameters	131
Combining = with an operator	136
Objects are variables, too	137
Refer to your objects with reference variables	138
References are like labels for your object	139
If there aren't any more references, your object gets garbage collected	140
Multiple references and their side effects	142
Two references means TWO ways to change an object's data	147
A special case: arrays	148
Arrays can contain a bunch of reference variables, too	149
Welcome to Sloppy Joe's Budget House o' Discount Sandwiches!	150
Objects use references to talk to each other	152
Where no object has gone before	153

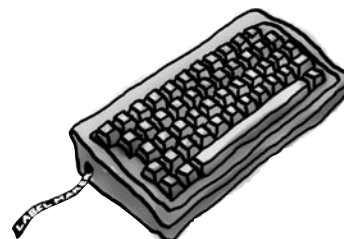
```
Dog fido;
Dog lucky = new Dog();
```



```
fido = new Dog();
```



```
lucky = null;
```



C# Lab 1

A Day at the Races

Joe, Bob, and Al love going to the track, but they're tired of losing all their money. They need you to build a simulator for them so they can figure out winners before they lay their money down. And, if you do a good job, they'll cut you in on their profits.

The Spec: Build a Racetrack Simulator	164
The Finished Product	172



5 encapsulation

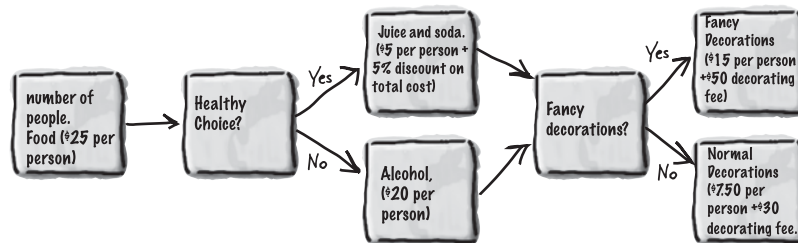
Keep your privates... private

Ever wished for a little more privacy?

Sometimes your objects feel the same way. Just like you don't want anybody you don't trust reading your journal, or paging through your bank statements, good objects don't let *other* objects go poking around their properties. In this chapter, you're going to learn about the power of **encapsulation**. You'll **make your object's data private**, add methods to **protect how that data is accessed**.



Kathleen is an event planner	174
What does the estimator do?	175
Kathleen's Test Drive	180
Each option should be calculated individually	182
It's easy to accidentally misuse your objects	184
Encapsulation means keeping some of the data in a class private	185
Use encapsulation to control access to your class's methods and fields	186
But is the realName field REALLY protected?	187
Private fields and methods can only be accessed from inside the class	188
A few ideas for encapsulating classes	191
Encapsulation keeps your data pristine	192
Properties make encapsulation easier	193
Build an application to test the Farmer class	194
Use automatic properties to finish the class	195
What if we want to change the feed multiplier?	196
Use a constructor to initialize private fields	197



inheritance

6

Your object's family tree

Sometimes you *DO* want to be just like your parents.

Ever run across an object that *almost* does exactly what you want *your* object to do? Found yourself wishing that if you could just *change a few things*, that object would be perfect? Well that's just one reason that **inheritance** is one of the most powerful concepts and techniques in the C# language. Before you're through this chapter, you'll learn how to **subclass** an object to get its behavior, but keep the **flexibility** to make changes to that behavior. You'll **avoid duplicate code**, **model the real world** more closely, and end up with code that's **easier to maintain**.



Kathleen does birthday parties, too	206
We need a BirthdayParty class	207
One more thing... can you add a \$100 fee for parties over 12?	213
When your classes use inheritance, you only need to write your code once	214
Build up your class model by starting general and getting more specific	215
How would you design a zoo simulator?	216
Use inheritance to avoid duplicate code in subclasses	217
Different animals make different noises	218
Think about how to group the animals	219
Create the class hierarchy	220
Every subclass extends its base class	221
Use a colon to inherit from a base class	222
We know that inheritance adds the base class fields, properties, and methods to the subclass...	225
A subclass can override methods to change or replace methods it inherited	226
Any place where you can use a base class, you can use one of its subclasses instead	227
A subclass can access its base class using the base keyword	232
When a base class has a constructor, your subclass needs one too	233
Now you're ready to finish the job for Kathleen!	234
Build a beehive management system	239
First you'll build the basic system	240
Use inheritance to extend the bee management system	245

interfaces and abstract classes

7 Making classes keep their promises

Actions speak louder than words.

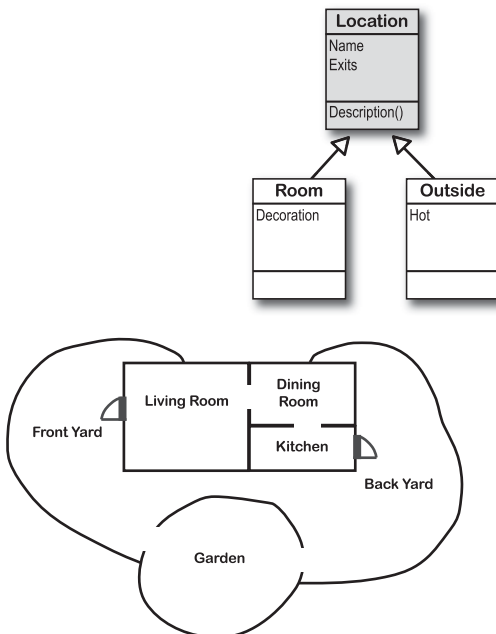
Sometimes you need to group your objects together based on the **things they can do** rather than the classes they inherit from. That's where **interfaces** come in—they let you work with any class that can do the job. But with **great power comes great responsibility**, and any class that implements an interface must promise to **fulfill all of its obligations**... or the compiler will break their kneecaps, see?

* Inheritance

* Abstraction

* Encapsulation

* Polymorphism



Let's get back to bee-sics	252
We can use inheritance to create classes for different types of bees	253
An interface tells a class that it must implement certain methods and properties	254
Use the interface keyword to define an interface	255
Get a little practice using interfaces	256
Now you can create an instance of NectarStinger that does both jobs	257
Classes that implement interfaces have to include ALL of the interface's methods	258
You can't instantiate an interface, but you can reference an interface	260
Interface references work just like object references	261
You can find out if a class implements a certain interface with "is"	262
Interfaces can inherit from other interfaces	263
The RoboBee 4000 can do a worker bee's job without using valuable honey	264
is tells you what an object implements, as tells the compiler how to treat your object	265
A CoffeeMaker is also an Appliance	266
Upcasting works with both objects and interfaces	267
Downcasting lets you turn your appliance back into a coffee maker	268
Upcasting and downcasting work with interfaces, too	269
There's more than just public and private	273
Access modifiers change scope	274
Some classes should never be instantiated	277
An abstract class is like a cross between a class and an interface	278
Some classes should never be instantiated	280
An abstract method doesn't have a body	281
Polymorphism means that one object can take many different forms	289

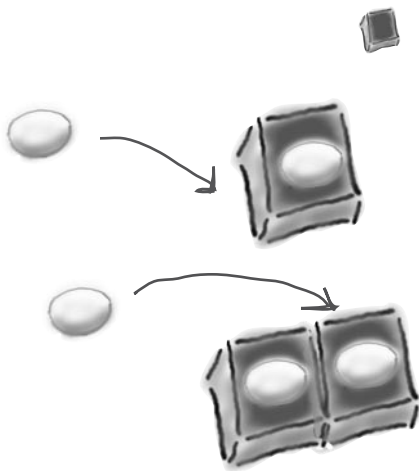
enums and collections

8

Storing lots of data

When it rains, it pours.

In the real world, you don't get to handle your data in tiny little bits and pieces. No, your data's going to come at you in **loads, piles and bunches**. You'll need some pretty powerful tools to organize all of it, and that's where **collections** come in. They let you **store, sort and manage** all the data that your programs need to pore through. That way you can think about writing programs to work with your data, and let the collections worry about keeping track of it for you.



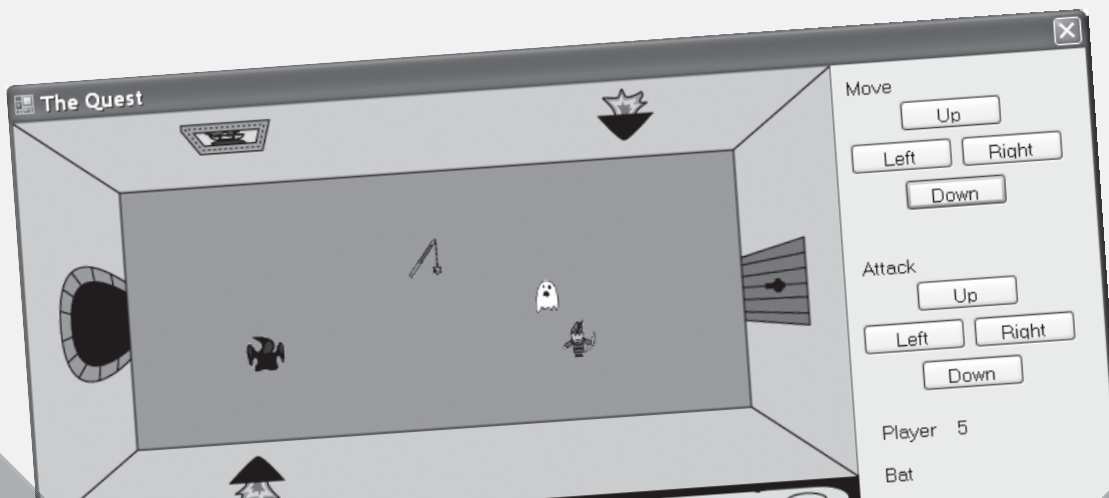
Strings don't always work for storing categories of data	310
Enums let you enumerate a set of valid values	311
Enums let you represent numbers with names	312
We could use an array to create a deck of cards...	315
Arrays are hard to work with	316
Lists make it easy to store collections of... anything	317
Lists are more flexible than arrays	318
Lists shrink and grow dynamically	321
List objects can store any type	322
Collection initializers work just like object initializers	326
Let's create a list of Ducks	327
Lists are easy, but SORTING can be tricky	328
Two ways to sort your ducks	329
Use IComparer to tell your List how to sort	330
Create an instance of your comparer object	331
IComparer can do complex comparisons	332
Use a dictionary to store keys and values	335
The Dictionary Functionality Rundown	336
Your key and value can be different types, too	337
You can build your own overloaded methods	343
And yet MORE collection types...	355
A queue is FIFO — First In, First Out	356
A stack is LIFO — Last In, First Out	357

C# Lab 2

The Quest

Your job is to build an adventure game where a mighty adventurer is on a quest to defeat level after level of deadly enemies. You'll build a turn-based system, which means the player makes one move and then the enemies make one move. The player can move or attack, and then each enemy gets a chance to move and attack. The game keeps going until the player either defeats all the enemies on all seven levels or dies.

The spec: build an adventure game	364
The fun's just beginning!	484



reading and writing files

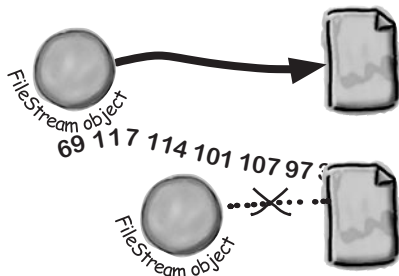
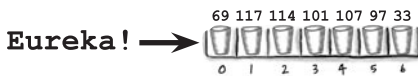
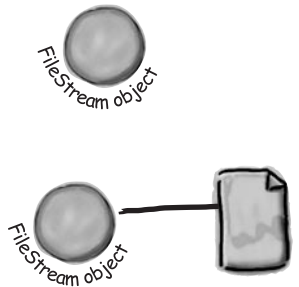
Save the byte array, save the world

9

Sometimes it pays to be a little persistent.

So far, all of your programs have been pretty short-lived. They fire up, run for a while, and shut down. But that's not always enough, especially when you're dealing with important information. You need to be able to **save your work**. In this chapter, we'll look at how to **write data to a file**, and then how to **read that information back in** from a file. You'll learn about the **.NET stream classes**, and also take a look at the mysteries of **hexadecimal** and **binary**.

C# uses streams to read and write data	386
Different streams read and write different things	387
A FileStream writes bytes to a file	388
Reading and writing takes two objects	393
Data can go through more than one stream	394
Use built-in objects to pop up standard dialog boxes	397
Dialog boxes are objects, too	399
Use the built-in File and Directory classes to work with files and directories	400
Use File Dialogs to open and save files	403
IDisposable makes sure your objects are disposed properly	405
Avoid file system errors with using statements	406
Writing files usually involves making a lot of decisions	412
Use a switch statement to choose the right option	413
Add an overloaded Deck() constructor that reads a deck of cards in from a file	415
What happens to an object when it's serialized?	417
But what exactly IS an object's state? What needs to be saved?	418
When an object is serialized, all of the objects it refers to get serialized too...	419
Serialization lets you read or write a whole object all at once	420
If you want your class to be serializable, mark it with the [Serializable] attribute	421
.NET converts text to Unicode automatically	425
C# can use byte arrays to move data around	426
Use a BinaryWriter to write binary data	427
You can read and write serialized files manually, too	429
StreamReader and StreamWriter will do just fine	433



exception handling

Putting out fires gets old

10

Programmers aren't meant to be firefighters.

You've worked your tail off, waded through technical manuals and a few engaging Head First books, and you've reached the pinnacle of your profession: **master programmer**. But you're still getting pages from work because **your program crashes**, or **doesn't behave like it's supposed to**. Nothing pulls you out of the programming groove like having to fix a strange bug . . . but with **exception handling**, you can write code to **deal with problems** that come up. Better yet, you can even react to those problems, and **keep things running**.

Brian needs his excuses to be mobile	440
When your program throws an exception, .NET generates an Exception object.	444
Brian's code did something unexpected	446
All exception objects inherit from Exception	448
The debugger helps you track down and prevent exceptions in your code	449
Use the IDE's debugger to ferret out exactly what went wrong in the excuse manager	450
Uh-oh—the code's still got problems...	453
Handle exceptions with try and catch	455
What happens when a method you want to call is risky?	456
Use the debugger to follow the try/catch flow	458
If you have code that ALWAYS should run, use a finally block	460
Use the Exception object to get information about the problem	465
Use more than one catch block to handle multiple types of exceptions	466
One class throws an exception, another class catches the exception	467
Bees need an OutOfHoney exception	468
An easy way to avoid a lot of problems: using gives you try and finally for free	471
Exception avoidance: implement IDisposable to do your own clean up	472
The worst catch block EVER: comments	474
Temporary solutions are okay (temporarily)	475
A few simple ideas for exception handling	476
Brian finally gets his vacation...	481



11

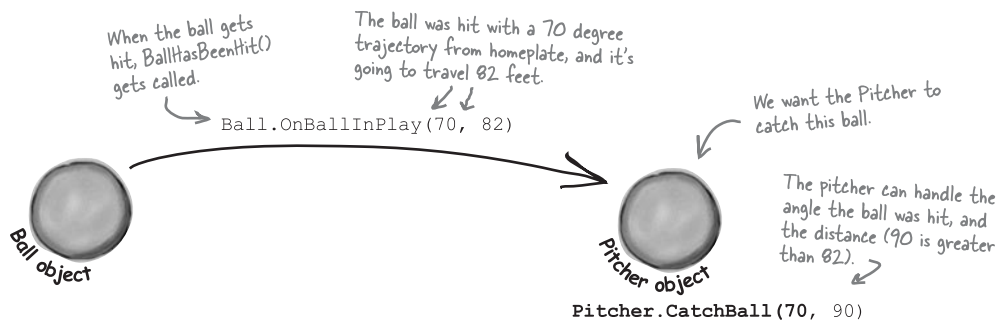
events and delegates

What your code does when you're not looking

Your objects are starting to think for themselves.

You can't always control what your objects are doing. Sometimes things... happen. And when they do, you want your objects to be smart enough to **respond to anything** that pops up. And that's what events are all about. One object *publishes* an event, other objects *subscribe*, and everyone works together to keep things moving. Which is great, until you've got too many objects responding to the same event. And that's when **callbacks** will come in handy.

Ever wish your objects could think for themselves?	484
But how does an object KNOW to respond?	484
When an EVENT occurs... objects listen	485
One object raises its event, others listen for it...	486
Then, the other objects handle the event	487
Connecting the dots	488
The IDE creates event handlers for you automatically	492
The forms you've been building all use events	498
Connecting event senders with event receivers	500
A delegate STANDS IN for an actual method	501
Delegates in action	502
Any object can subscribe to a public event...	505
Use a callback instead of an event to hook up exactly one object to a delegate	507
Callbacks use delegates, but NOT events	508



review and preview

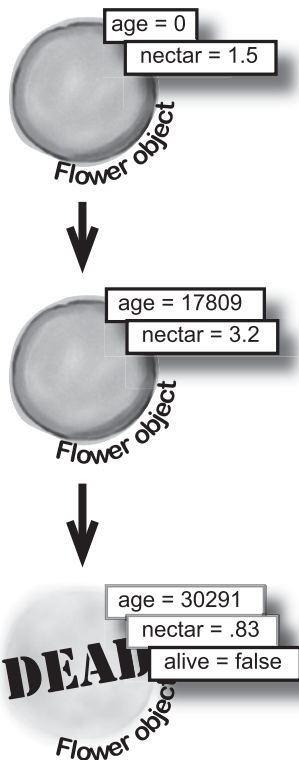
Knowledge, power, and building cool stuff

12

Learning's no good until you BUILD something.

Until you've actually written working code, it's hard to be sure if you really *get* some of the tougher concepts in C#. In this chapter, we're going to learn about some new odds and ends: **timers** and dealing with collections using **LINQ** (to name a couple). We're also going to build phase I of a **really complex application**, and make sure you've got a good handle on what you've already learned from earlier chapters. So buckle up... it's time to build some **cool software**.

Life and death of a flower



You've come a long way, baby	516
We've also become beekeepers	517
The beehive simulator architecture	518
Building the beehive simulator	519
Life and death of a flower	523
Now we need a Bee class	524
Filling out the Hive class	532
The hive's Go() method	533
We're ready for the World	534
We're building a turn-based system	535
Giving the bees behavior	542
The main form tells the world to Go()	544
We can use World to get statistics	545
Timers fire events over and over again	546
The timer's using a delegate behind the scenes	547
Let's work with groups of bees	554
A collection collects... DATA	555
LINQ makes working with data in collections and databases easy	557

13

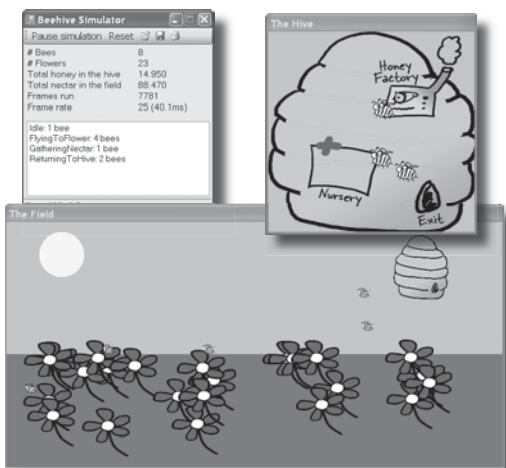
controls and graphics

Make it pretty

Sometimes you have to take graphics into your own hands.

We've spent a lot of time on relying on controls to handle everything visual in our applications. But sometimes that's not enough—like when you want to **animate a picture**. And once you get into animation, you'll end up **creating your own controls** for your .NET programs, maybe adding a little **double buffering**, and even **drawing directly onto your forms**. It all begins with the **Graphics** object, **Bitmaps**, and a determination to not accept the graphics status quo.

You've been using controls all along to interact with your programs	564
Form controls are just objects	565
Add a renderer to your architecture	568
Controls are well-suited for visual display elements	570
Build your first animated control	573
Your controls need to dispose their controls, too!	577
A UserControl is an easy way to build a control	578
Add the hive and field forms to the project	582
Build the Renderer	583
Let's take a closer look at those performance issues	590
You resized your Bitmaps using a Graphics object	592
Your image resources are stored in Bitmap objects	593
Use System.Drawing to TAKE CONTROL of graphics yourself	594
A 30-second tour of GDI+ graphics	595
Use graphics to draw a picture on a form	596
Graphics can fix our transparency problem...	601
Use the Paint event to make your graphics stick	602
A closer look at how forms and controls repaint themselves	605
Double buffering makes animation look a lot smoother	608
Double buffering is built into forms and controls	609
Use a Graphics object and an event handler for printing	614
PrintDocument works with the print dialog and print preview window objects	615

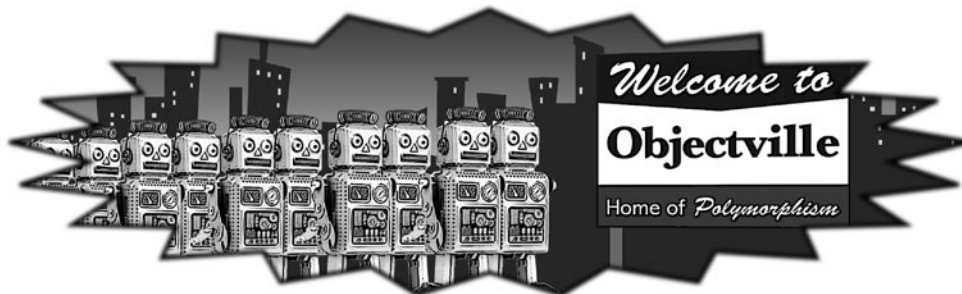


14

CAPTAIN AMAZING

THE DEATH OF THE OBJECT

Captain Amazing, Objectville's most amazing object pursues his arch-nemesis...	622
Your last chance to DO something... your object's finalizer	628
When EXACTLY does a finalizer run?	629
Dispose() works with using, finalizers work with garbage collection	630
Finalizers can't depend on stability	632
Make an object serialize itself in its Dispose()	633
Meanwhile, on the streets of Objectville...	636
A struct <i>looks</i> like an object...	637
..but <i>isn't</i> on the heap	637
Values get copied, references get assigned	638
Structs are value types; objects are reference types	639
The stack vs. the heap: more on memory	641
Captain Amazing.. not so much	645
Extension methods add new behavior to EXISTING classes	646
Extending a fundamental type: string	648



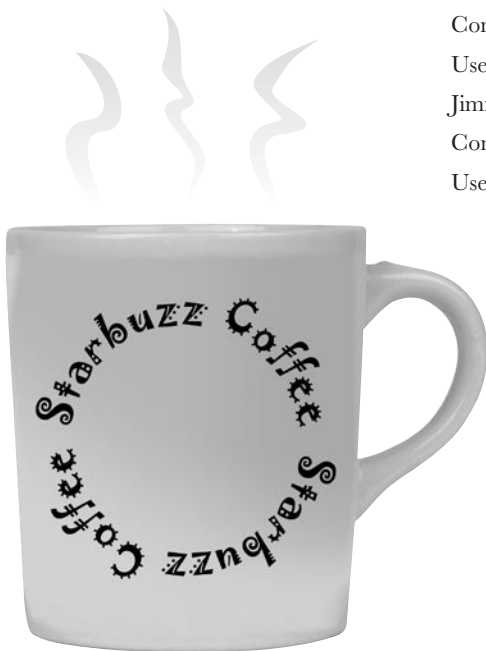
15 LINQ

Get control of your data

It's a data-driven world... you better know how to live in it.

Gone are the days when you could program for days, even weeks, without dealing with **loads of data**. But today, **everything is about data**. In fact, you'll often have to work with data from **more than one place**... and in more than one format. Databases, XML, collections from other programs... it's all part of the job of a good C# programmer. And that's where **LINQ** comes in. LINQ not only lets you **query data** in a simple, intuitive way, but it lets you **group data**, and **merge data from different data sources**.

An easy project...	654
...but the data's all over the place	655
LINQ can pull data from multiple sources	656
.NET collections are already set up for LINQ	657
LINQ makes queries easy	658
LINQ is simple, but your queries don't have to be	659
LINQ is versatile	662
LINQ can combine your results into groups	667
Combine Jimmy's values into groups	668
Use join to combine two collections into one query	671
Jimmy saved a bunch of dough	672
Connect LINQ to a SQL database	674
Use a join query to connect Starbuzz and Objectville	678



C# Lab 3

Invaders

In this lab you'll pay homage to one of the most popular, revered and replicated icons in video game history, a game that needs no further introduction. It's time to build Invaders.

The grandfather of video games	682
And yet there's more to do...	701



leftovers





The top 5 things we wanted to include in this book

The fun's just beginning!

We've shown you a lot of great tools to build some really **powerful software** with C#. But there's no way that we could include **every single tool, technology or technique** in this book—there just aren't enough pages. We had to make some *really tough choices* about what to include and what to leave out. Here are some of the topics that didn't make the cut. But even though we couldn't get to them, we still think that they're **important and useful**, and we wanted to give you a small head start with them.

#1 LINQ to XML	704
#2 Refactoring	706
#3 Some of our favorite Toolbox components	708
#4 Console Applications	710
5 Windows Presentation Framework	712
Did you know that C# and the .NET Framework can...	714

 backgroundWorker 1

 fileSystemWatcher 1

 performanceCounter 1

