

Object-Oriented ActionScript for Flash 8

Peter Elst and Todd Yard
with Sas Jacobs and William Drol



Object-Oriented ActionScript for Flash 8

Copyright © 2006 by Peter Elst, Todd Yard, Sas Jacobs, and William Drol

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-619-7

ISBN-10 (pbk): 1-59059-619-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor **Assistant Production Director**
Chris Mills Kari Brooks-Copony

Technical Reviewers **Production Editor**
Jared Tarbell, Katie Stence
Stephen Downs

Editorial Board **Compositor**
Dina Quan

Steve Anglin, Dan Appleman,
Ewan Buckingham, Gary Cornell, **Proofreader**
Jason Gilmore, Jonathan Hassell, April Eddy
James Huddleston, Chris Mills,
Matthew Moodie, Dominic Shakeshaft, **Indexer**
Jim Sumser, Matt Wade Michael Brinkman

Project Manager **Artist**
Sofia Marchant April Milne

Copy Edit Manager **Interior and Cover Designer**
Nicole LeClerc Kurt Krames

Copy Editor **Manufacturing Director**
Ami Knox Tom Debolski

*Dedicated to everyone at Macromedia, now Adobe, for their years
of unceasing commitment to the Flash community.*

—Peter Elst

*Dedicated to my wife, Lydian, who loves me despite the fact that after
all the hours I spent on this book she only gets this sentence.*

—Todd Yard

CONTENTS AT A GLANCE

Foreword	xv
About the Authors	xvi
About the Technical Reviewer	xvii
Acknowledgments	xviii
PART ONE: OOP AND ACTIONSCRIPT	xx
<hr/>	
Chapter 1: Introduction to OOP	1
Chapter 2: Programming Concepts	11
Chapter 3: ActionScript 2.0 Programming	19
PART TWO: FLASH OOP GUIDELINES.	32
<hr/>	
Chapter 4: Planning	33
Chapter 5: Project Workflow	49
Chapter 6: Best Practices	65
PART THREE: CORE OOP CONCEPTS	82
<hr/>	
Chapter 7: Encapsulation	83
Chapter 8: Classes	103

Chapter 9: Inheritance	121
Chapter 10: Polymorphism	135
Chapter 11: Interfaces	145
Chapter 12: Design Patterns	157
Chapter 13: Case Study: An OOP Media Player	201
PART FOUR: BUILDING AND EXTENDING A DYNAMIC FRAMEWORK	240
<hr/>	
Chapter 14: Framework Overview	241
Chapter 15: Manager Classes	259
Chapter 16: UI Widgets	279
Chapter 17: OOP Animation and Effects	333
PART FIVE: DATA INTEGRATION	382
<hr/>	
Chapter 18: Interrelationships and Interactions Between Components	383
Chapter 19: Communication Between Flash and the Browser	417
Chapter 20: Server Communication (XML and Web Services)	439
Chapter 21: Case Study: Time Sheet Application	483
Index	521

CONTENTS

Foreword	xv
About the Authors	xvi
About the Technical Reviewer	xvii
Acknowledgments	xviii

PART ONE: OOP AND ACTIONSCRIPT **xx**

Chapter 1: Introduction to OOP	1
The scoop with OOP?	2
Understanding the object-oriented approach	2
Classes and objects	3
Properties	3
Encapsulation: Hiding the details	4
Polymorphism: Exhibiting similar features	7
Inheritance: Avoid rebuilding the wheel	8
What's next?	9
Chapter 2: Programming Concepts	11
About programming slang	12
Building blocks of programming	13
Variables	13
About variable data	14
Arrays	14

Functions	15
About calling functions	15
About function parameters	15
Loops	16
Conditionals	16
OOP concepts	16
What's next?	17

Chapter 3: ActionScript 2.0 Programming 19

ActionScript 1.0 vs. ActionScript 2.0	20
Declaring variables	20
Classes vs. prototypes	21
Public and private scope	25
Strong typing and code hints	27
ActionScript trouble spots	29
Case sensitivity	29
Declaring variables	30
Use of the this keyword	30
What's next?	31

PART TWO: FLASH OOP GUIDELINES 32

Chapter 4: Planning 33

The importance of planning	34
Initial phase: Planning reusability!	35
Planning encapsulation	35
Planning inheritance	36

CONTENTS

Analyzing a Flash ActionScript project	39
Flash files run on the client	39
Securing data sent to the server	39
Parsing data in Flash	40
Introduction to UML modeling	40
Why use UML?	41
UML offers standardized notation and has a language-neutral syntax	41
UML can be used to model anything	42
Class diagram	42
Association and generalization	43
Aggregation and composition	44
What's next?	46
Chapter 5: Project Workflow	49
Introducing version control	50
About Concurrent Versions System	50
Using TortoiseCVS	52
Approaches to programming	58
Rapid Application Development	59
Extreme Programming	60
Usability testing	62
What's next?	63
Chapter 6: Best Practices	65
External ActionScript	66
About commenting	68
Naming conventions	70
Variables	70
Constants	71
Functions	71
Classes	71
Methods	72
Properties	72
Packages	72
Programming styles	73
Alternative programming styles	77
Coding practices: Todd Yard	77
Coding practices: Sas Jacobs	79
Commenting code	79
Naming conventions	80
What's next?	81

PART THREE: CORE OOP CONCEPTS 82**Chapter 7: Encapsulation 83**

Setting up encapsulation	84
Creating new layers	85
Drawing a background	87
Aligning and locking the background	87
Drawing a ball	88
Converting the ball into a Library symbol	89
Content summary	90
Writing the code	91
Creating an event handler	91
What about encapsulation?	93
Testing the event handler	94
Updating the Ball	95
Improving the code	96
Enhancing behavior with properties	96
Narrowing the focus with functions	97
Encapsulation summary	99
What's next?	101

Chapter 8: Classes 103

Classes vs. prototypes	104
Constructors	106
About this	109
Methods	112
Anonymous functions	113
Implementing a class	116
The Mover class	116
What's next?	118

Chapter 9: Inheritance 121

About class hierarchy	122
A quick inheritance test	122
About inheritance syntax	125
The Bouncer class	126
The Gravity class	129
Inheritance summary	133
What's next?	133

Chapter 10: Polymorphism	135
Building a polymorphism example	136
Implementing polymorphism for application reuse	138
Basic concept of polymorphism	138
Functional polymorphism at work	139
What's next?	142
Chapter 11: Interfaces	145
Interfaces overview	146
Interface use-cases	147
What an interface looks like	147
Implementing an interface	148
What's next?	155
Chapter 12: Design Patterns	157
Understanding design patterns	158
Implementing design patterns	160
Observer pattern	160
Basic implementation	160
Practical implementation	167
Extending the practical implementation	169
Singleton pattern	171
Basic implementation	172
Practical implementation	177
Building an interface	181
Decorator pattern	183
Basic implementation	183
Practical implementation	184
Applying the Decorator pattern	186
Model-View-Controller pattern	191
Basic implementation	192
Practical implementation	193
Bringing together the Model, View, and Controller	196
Design patterns summary	197
What's next?	198
Chapter 13: Case Study: An OOP Media Player	201
Planning the player	202
Picking a pattern	202
Guaranteeing methods and datatypes with an interface	203
Examining class structure	204

Building the media player	206
IntervalManager	207
Defining the interfaces	209
Dispatching events	209
Media interfaces	214
Controlling media	215
Defining properties	215
Private methods	216
Public methods	218
Controlling FLVs	222
Building a video view	228
Controlling SWFs	229
Building a SWF view	236
Controlling MP3s	238
Summary	239
What's next?	239

PART FOUR: BUILDING AND EXTENDING A DYNAMIC FRAMEWORK 240

Chapter 14: Framework Overview 241

Introducing the framework	242
Understanding the MovieClip class	246
Understanding the UIObject class (mx.core.UIObject)	253
Understanding the UIComponent class (mx.core.UIComponent)	255
Understanding the View class (mx.core.View)	256
Framework summary	257
What's next?	257

Chapter 15: Manager Classes 259

Planning the framework	260
What to manage	260
Diagramming the classes	261
Building managers	263
StyleFormat	263
StyleManager	266
Adding style	269
SoundManager	272
Sounding off	275
Summary	277
What's next?	277

Chapter 16: UI Widgets 279

Diagramming the classes 280

 UObject 280

 Block 283

 SimpleButton 284

Making the foundation 285

Basic building block 292

Building a component 294

Skinning a widget 303

Changing state 307

Adding some style 308

More ways to skin a cat 313

Attaching from scratch 316

Tying in events 320

Pulling it all together 325

Summary 329

What’s next? 330

Chapter 17: OOP Animation and Effects 333

Preparing for animation 334

 Animator 335

Tweening properties and values 336

 Tween 336

 Easer 341

 Testing the Tweener 344

 Enhancing Tweener 347

 Mover 354

 Motion blur 357

Transitioning views 360

 Transition 360

 FadeTransition 363

 Testing transitions 364

 ColorTransition 369

 BlurTransition 371

 NoiseTransition 374

 DissolveTransition and WaterTransition 376

Summary 381

What’s next? 381

PART FIVE: DATA INTEGRATION 382**Chapter 18: Interrelationships and Interactions
Between Components 383**

Data binding	384
The mx.data.binding package	385
Creating a simple binding	386
Creating EndPoints	386
Specifying a location	387
Creating the binding	388
Using the execute method	388
Working through a simple binding example	389
Using formatters	395
Using built-in formatters	396
Using the Boolean formatter	396
Using the Compose String formatter	396
Using the Date formatter	397
Using the Rearrange Fields formatter	397
Using the Number formatter	397
Working through a simple formatting example	398
Understanding custom formatters	404
Including validators	406
Working with built-in validators	407
Working with a custom validator	413
Summary	415
What's next?	415

**Chapter 19: Communication Between Flash and
the Browser 417**

Communication with Flash Player 7 and below	419
Sending variables into Flash	419
Calling JavaScript from Flash	419
Using the Flash/JS Integration Kit	421
Understanding the ExternalInterface class	423
Understanding Flash Player 8 security	424
Using the call method	424
Using the addCallback method	429
ActionScript communication with other languages	434
Calling a non-JavaScript method	434
Calling an ActionScript method from an application	435
Summary	435
What's next?	436

Chapter 20: Server Communication (XML and Web Services)	439
Understanding XML	440
XML declarations	442
Using XML in Flash	443
XMLConnector component	443
XML class	447
What are web services?	454
Understanding SOAP	454
Talking to web services	456
WebServiceConnector component	456
WebService class	464
Flash Player security sandbox	478
System.security.allow.Domain()	478
Cross-domain policy files	478
Using a server-side proxy script	480
Summary	480
What's next?	481
Chapter 21: Case Study: Time Sheet Application	483
Planning the application	484
Structuring the application	485
Writing stub code	487
Model-View-Controller classes	487
TimeSheetModel class (Model)	487
TimeSheetView class (View)	488
TimeSheetController class (Controller)	490
Project and Task classes	491
Project class	491
Task class	492
Bringing it all together	493
Initializing the layout	493
Adding a project	495
Displaying projects	498
Adding a task	501
Project and task details	506
Running a task timer	510
Persisting time sheet data	512
Summary	517
Conclusion	518
Index	521

FOREWORD

If there's one thing I've learned as a developer, it's this: Complexity happens; simplicity, you have to consistently strive for. Nowhere is this truer than in education. Our role as teachers, by definition, is to simplify subjects so that they can be easily understood. A good teacher dispels trepidation with anecdote, abstraction with analogy, superstition and magic with knowledge.

Simplicity, however, is not easily attained. In order to simplify, you must first gain an encompassing understanding of the complex. It is a rare person who can simultaneously exist in both the simple and complex plains of a problem domain and communicate effectively at both levels. It is, however, these rare people who make the best teachers.

Object-oriented programming (OOP) is a subject that many Flash developers do not approach due to a widespread erroneous perception of its enormous scope and complexity. Nothing could be further from the truth.

The core concepts behind OOP are simple enough for a primary school student with a particularly nasty case of Hynarian flu to understand in a single sitting.

It must be because OOP is essentially such a simple concept that we sometimes feel the need to protect ourselves with important-sounding words the length of major rivers in order to explain it. Because, hey, if we said that OOP involves the interaction of objects, each of which is an instance of a certain blueprint and has certain traits and behaviors—well, that would just be too simple. Who'd respect our geeky prowess then? Instead, we lock ourselves in our ivory towers, hiding behind unscalable walls of inheritance, composition, polymorphism, and encapsulation, and hope that the FlashKit masses will tend to their tweens and leave us to meditate on the path to programming nirvana.

Unfortunately, OOP is so often presented in such pretentious prose so as to be illegible to all but a handful of PhDs. If grandiose, self-important passages of academic rambling are what you're after, then you should put this book down and walk away now. I'm sure you'll find an 800-page hardback elsewhere to satisfy your thirst for confusion. If, however, you are looking for a pragmatic guide to OOP and ActionScript 2 (AS2) that is simply written and easy to understand, you could do far worse than to look through these pages more closely.

Aral Balkan
2 January 2006
Famagusta, Cyprus

ABOUT THE AUTHORS

Peter Elst is a Flash-certified professional and former Team Macromedia volunteer, and he runs his own company, named MindStudio, doing mostly freelance Flash and Flex consultancy, development, and training. As a valued contributor to the online Flash community, Peter has presented at numerous international events and conferences and has had his work published in leading magazines and websites.

Over the years, the focus of his work changed from interactive animations to multimedia applications, e-learning, and content management systems. Peter is user group manager for the MMUG Belgium and blogs on his personal website: www.petere1st.com.

Sas Jacobs is a web developer who loves working with Flash. She set up her business, Anything Is Possible, in 1994, working in the areas of web development, IT training, and technical writing. The business works with large and small clients building web applications with ASP.NET, Flash, XML, and databases.

Sas has spoken at such conferences as Flash Forward, MXDU, and FlashKit on topics relating to XML and dynamic content in Flash. In her spare time, Sas is passionate about traveling, photography, running, and enjoying life. You can find out more about her at www.sasjacobs.com.

Todd Yard is currently a Flash developer at Brightcove in Cambridge, Massachusetts, where he moved early in 2005 in the middle of a blizzard. Previously, he was in New York City, where he initially moved in 1996 in the middle of a blizzard, working with EGO7 on their Flash content management system and community software while freelancing with agencies developing web applications for clients such as GE and IBM. Todd originally hails from Phoenix, where there are no blizzards, and has written for a number of friends of ED books, of which his favorites are *Flash MX Studio* and *Flash MX Application and Interface Design*, though he feels *Extending Flash MX 2004: Complete Guide and Reference to JavaScript Flash* is probably the most useful. His personal site, which he used to update all the time, he fondly remembers, is www.27Bobs.com.

William Drol entered Macromedia Flash development with a varied background in object-oriented programming and graphic design. His first experience with Macromedia was the admittedly quirky but OOP-based Macromedia Director and Lingo. Today, there are many reasons to be excited about Flash MX and the hugely improved ActionScript. Drol looks forward to integrating Flash MX with web services, and he pursues other technologies such as XML, XSLT, and his current favorite, Microsoft C#. Learn more about the author at <http://www.billdrol.com>.

ABOUT THE TECHNICAL REVIEWERS

Tink, a.k.a. **Stephen Downs**, has been a freelance Flash designer/developer for the past four years, and he has a background in art, design, and photography. Based in London, England, he works on a wide range of projects, both for other companies and his own clients.

He has worked on projects with various agencies for brands such as MTV, Xbox, AMD Athlon, PG Tips, AGCO, Interflora, Motorola, Shockwave.com, UK Government, French Music Bureau, and many more. The growth in his workload has recently led to the startup of Tink LTD.

His primary focus is user interaction and interactive motion, integrating design, and development using best practice methodologies.

www.tink.ws

www.tink.ws/blog

Jared Tarbell was born in 1973 to William and Suzon Davis Tarbell in the high-altitude desert city of Albuquerque, New Mexico. First introduced to personal computers in 1987, Jared's interest in computation has grown in direct proportion to the processing power of these machines. Jared holds a Bachelor of Science degree in Computer Science from New Mexico State University. He sits on the Board of the Austin Museum of Digital Art where he helps promote and encourage appreciation of the arts within the global community. Jared is most interested in the visualization of large data sets, and the emergent, life-like properties of complex computational systems. Jared has recently returned to Albuquerque to work closer to friends and family while enjoying the unique aspects of desert living.

Additional work from Jared Tarbell can be found at levitated.net and complexification.net.

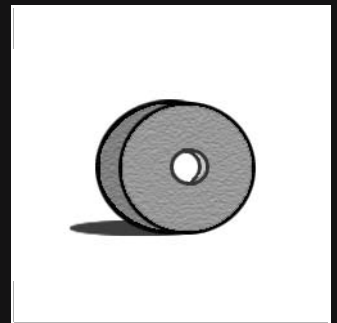
ACKNOWLEDGMENTS

Special thanks to Jared Tarbell and Tink for their thorough technical review; to Chris, Sofia, and the rest of the friends of ED/Apress team for their help and patience in getting this book written; and to coauthors Todd and Sas for their excellent chapters!

Peter Elst

PART ONE OOP AND ACTIONSCRIPT

1 INTRODUCTION TO OOP



Object-oriented programming (OOP) sounds much scarier than it actually is. Essentially OOP is nothing more than a way of looking at a particular problem and breaking it down into smaller pieces called *objects*. These objects form the building blocks of object-oriented applications, and when designed properly they help form a solid framework on which to build your project.

The scoop with OOP?

Before OOP became commonplace, we had something called *procedural programming*, which often required developers to write very complex and highly interdependent code. A minor change to any part of the code could spell disaster for the entire application. Debugging that type of application was a terribly painful and time-consuming task that often resulted in the need to completely rebuild large pieces of code.

When more and more user interaction got introduced in applications, it became apparent that procedural programming wouldn't cut it. Object-oriented programming was born as an attempt to solve these very problems. Although it certainly isn't the be-all and end-all of successful programming, OOP does give developers a great tool for handling any kind of application development.

The wonderful thing about object-oriented thinking is that you can look at practically any item in terms of a collection of objects. Let's look at a car for example. To the average Joe, a car is simply a vehicle (or object) that gets you places. If you ask a mechanic about a car, he'll most likely tell you about the engine, the exhaust, and all sorts of other parts. All these car parts can also be thought of as individual objects that work together to form a larger object, "the car." None of these parts actually know the inner workings of the other parts, and yet they work (or should work) together seamlessly.

Understanding the object-oriented approach

"See that bird?" he says. 'It's a Spencer's warbler. (I knew he didn't know the real name.) Well, in Italian, it's a Chutto Lapittida. In Portuguese, it's a Bom da Peida. In Chinese, it's a Chung-long-tah, and in Japanese, it's a Katano Tekeda. You can know the name of that bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird. You'll only know about humans in different places, and what they call the bird. So let's look at the bird and see what it's doing, that's what counts.'"

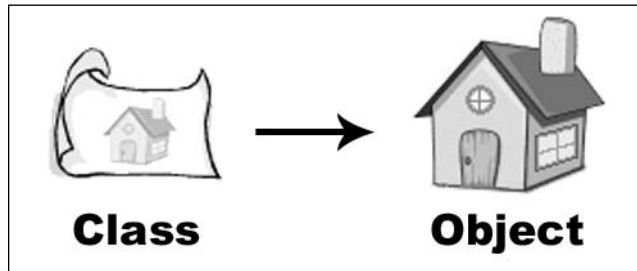
—Richard Feynman

When studying OOP you'll come across a *plethora* of big words like *encapsulation*, *polymorphism*, and *inheritance*. Truth be told the ideas behind them are often quite simple, and there's no real need to memorize those terms unless you'd like to use them for showing off at your next family get-together.

Knowing the theory behind this terminology is, however, essential, and that's just what we'll be discussing next.

Classes and objects

1



When studying OOP, you cannot ignore classes and objects, as those are the fundamental building blocks of any project. A good understanding of what classes and objects are and the roles they play will help you get on track to understanding OOP.

There's a subtle difference between a class and an object. A *class* is a self-contained description for a set of related services and data. Classes list the services they provide without revealing how they work internally. Classes aren't generally able to work on their own; they need to instantiate at least one object that is then able to act on the services and data described in the class.

Suppose you want to build a house. Unless you build it yourself, you need an architect and a builder. The architect drafts a blueprint, and the builder uses it to construct your house. Software developers are architects, and classes are their blueprints. You cannot use a class directly, any more than you could move your family into a blueprint. Classes only describe the final product. To actually do something you need an *object*.

If a class is a blueprint, then an object is a house. Builders create houses from blueprints; OOP creates objects from classes. OOP is efficient. You write the class once and create as many objects as needed.

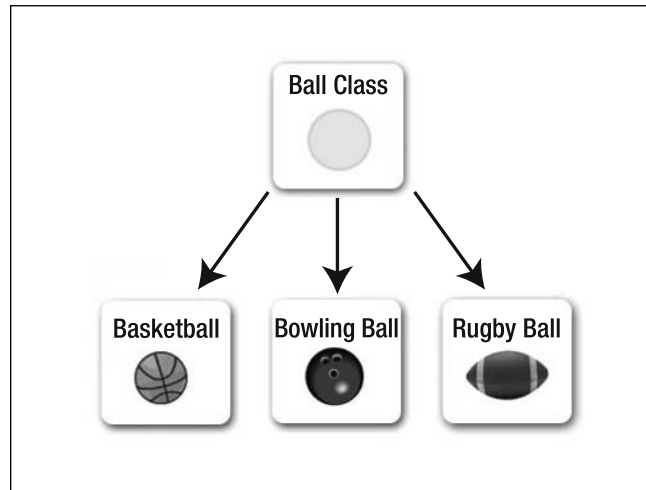
Because classes can be used to *create* multiple objects, objects are often referred to as *class instances*.

Properties

Properties give individual objects unique qualities. Without properties, each house (from the previous example) would remain identical to its neighbors (all constructed from the same blueprint). With properties, each house is unique, from its exterior color to the style of its windows.

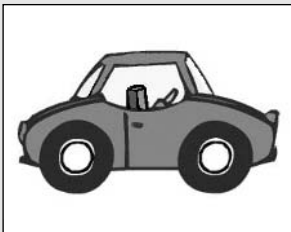
Let's look at a Ball class for example. From that one class you can create multiple ball instances; however, not all balls look identical to one another. By providing your Ball class

with properties such as color, weight, and shape, you can create instances that describe balls as diverse as a basketball, bowling ball, or rugby ball just by assigning different values to properties in each instance of the class.



In OOP, you write classes to offer predefined behaviors and maybe hold some data. Next, you create one or more objects from a class. Finally, you endow objects with their own individual property values. The progression from classes to objects to objects with unique properties is the essence of OOP.

Encapsulation: Hiding the details



When you get into your car, you turn the key, the car starts, and off you go. You don't need to understand how the car parts work to find yourself in rush-hour traffic. The car starts when you turn the key. Car designers hide the messy internal details so you can concentrate on important things like finding another radio station. OOP calls this concept encapsulation.

Analogies like the preceding car example are very useful to explain concepts such as encapsulation, but it is no doubt more appealing to take an in-depth look at potential real-world scenarios like, for example, an accounting office.

Accountants love details (all the numbers, receipts, and invoices). The accountant's boss, however, is interested in the bottom line. If the bottom line is zero, the company is debt-free. If the bottom line is positive, the company is profitable. She is happy to ignore all the

messy details and focus on other things. Encapsulation is about ignoring or hiding internal details. In business, this is delegation. Without it, the boss may need to deal with accounting, tax law, and international trading at a level beyond her ability.

OOP loves encapsulation. With encapsulation, classes hide their own internal details. Users of a class (yourself, other developers, or other applications) are not required to know or care why it works. Class users just need the available service names and what to provide to use them. Building classes is an abstraction process; you start with a complex problem, and then reduce it down (abstracting it) to a list of related services. Encapsulation simplifies software development and increases the potential for code reuse.

To demonstrate, I'll present some pseudo-code (false code). You can't enter pseudo-code into a computer, but it's great for previewing ideas. First, you need an Accounting class:

```
Start Of Accounting Class
End Of Accounting Class
```

Everything between the start and end line is the Accounting class. A useless class so far, because it's empty. Let's give the Accounting class something to do:

```
Start Of Accounting Class
  Start Of Bottom Line Service
    (Internal Details Of Bottom Line Service)
  End Of Bottom Line Service
End Of Accounting Class
```

Now the Accounting class has a Bottom Line service. How does that service work? Well, I know (because I wrote the code), but you (as a user of my class) have no idea. That's exactly how it should be. You don't know or care how my class works. You just use the Bottom Line service to see if the company is profitable. As long as my class is accurate and dependable, you can go about your business. You want to see the details anyway? Okay, here they are:

```
Start Of Accounting Class
  Start Of Bottom Line Service
    Do Invoice Service
    Do Display Answer Service
  End Of Bottom Line Service
End Of Accounting Class
```

Where did the Invoice and Display Answer services come from? They're part of the class too, but encapsulation is hiding them. Here they are:

```
Start Of Accounting Class
  Start Of Bottom Line Service
    Do Invoice Service
    Do Display Answer Service
  End Of Bottom Line Service
```



```
    Start Of Invoice Service
      (Internal Details Of Invoice Service)
    End Of Invoice Service

    Start Of Display Answer Service
      (Internal Details Of Display Answer Service)
    End Of Display Answer Service
  End Of Accounting Class
```

The Bottom Line service has no idea how the Invoice service works, nor does it care. You don't know the details, and neither does the Bottom Line service. This type of simplification is the primary benefit of encapsulation. Finally, how do you request an answer from the Bottom Line service? Easy, just do this:

```
Do Bottom Line Service
```

That's all. You're happy, because you only need to deal with a single line of code. The Bottom Line service (and encapsulation) handles the details for you.

When I speak of hiding code details, I'm speaking conceptually. I don't mean to mislead you. This is just a mental tool to help you understand the importance of abstracting the details. With encapsulation, you're not actually hiding code (physically). If you were to view the full Accounting class, you'd see the same code that I see.

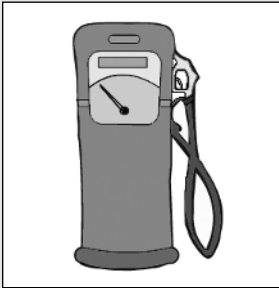
```
    Start Of Accounting Class
      Start Of Bottom Line Service
        Do Invoice Service
        Do Display Answer Service
      End Of Bottom Line Service

      Start Of Invoice Service
        Gather Invoices
        Return Sum
      End Of Invoice Service

      Start Of Display Answer Service
        Display Sum
      End Of Display Answer Service
    End Of Accounting Class
```

If you're wondering why some of the lines are indented, this is standard practice (that is not followed often enough). It shows, at a glance, the natural hierarchy of the code (of what belongs to what). Please adopt this practice when you write computer code.

Polymorphism: Exhibiting similar features



Are you old enough to remember fuel stations before the self-service era? You could drive into these places and somebody else would fill up your tank. The station attendant knew about OOP long before you did. He put the fuel nozzle into the tank (any tank) and pumped the fuel! It didn't matter if you drove a Ford, a Chrysler, or a Datsun. All cars have fuel tanks, so this behavior is easy to repeat for any car. OOP calls this concept polymorphism.

1

Much like cars need fuel to run, I take my daily dose of vitamins by drinking a glass of orange juice at breakfast. This incidentally brings me to a great example showing the concept of polymorphism.

Oranges have pulp. Lemons have pulp. Grapefruits have pulp. Cut any of these fruit open, I dare you, and try to scoop out the fruit with a spoon. Chances are, you'll get a squirt of citrus juice in your eye. Citrus fruits know exactly where your eye is, but you don't have to spoon them out to know they share this talent (they're all acid-based juice-squirters). Look at the following Citrus class:

```

Start Of Citrus Class
  Start Of Taste Service
    (Internal Details Of Taste Service)
  End Of Taste Service

  Start Of Squirt Service
    (Internal Details Of Squirt Service)
  End Of Squirt Service
End Of Citrus Class

```

You can use the Citrus class as a base to define other classes:

```

Start Of Orange Class
  Using Citrus Class
  Property Named Juice
End Of Orange Class

Start Of Lemon Class
  Using Citrus Class
  Property Named Juice
End Of Lemon Class

```

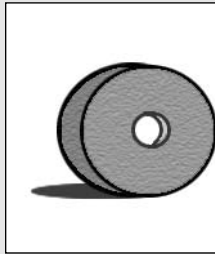
```

Start Of Grapefruit Class
  Using Citrus Class
    Property Named Juice
End Of Grapefruit Class

```

Besides demonstrating inheritance again, the Orange, Lemon, and Grapefruit classes also exhibit similar behaviors. This is polymorphism. You know that the Orange, Lemon, and Grapefruit classes have the ability to squirt (inherited from the Citrus class), but each class has a Juice property. So the orange can squirt orange juice, the lemon can squirt lemon juice, and the grapefruit can squirt grapefruit juice. You don't have to know in advance which type of fruit, because they all squirt. In fact, you could taste the juice (inherited from the Citrus class) to know which fruit you're dealing with. That's polymorphism: multiple objects exhibiting similar features in different ways.

Inheritance: Avoid rebuilding the wheel



Grog roll wheel. Wheel good. Grog doesn't like rebuilding wheels. They're heavy, made of stone, and tend to crush feet when they fall over. Grog likes the wheel that his stone-age neighbor built last week. Sneaky Grog. Maybe he'll carve some holes into the wheel to store rocks, twigs, or a tasty snack. If Grog does this, he'll have added something new to the existing wheel (demonstrating inheritance long before the existence of computers).

Inheritance in OOP is a real timesaver. You don't need to modify your neighbor's wheel. You only need to tell the computer, "Build a replica of my neighbor's wheel, and then add this, and this, and this." The result is a custom wheel, but you didn't modify the original. Now you have two wheels, each unique. To clarify, here's some more pseudo-code:

```

Start Of Wheel Class
  Start Of Roll Service
    (Internal Details Of Roll Service)
  End Of Roll Service
End Of Wheel Class

```

The Wheel class provides a single service named Roll. That's a good start, but what if you want to make a tire? Do you build a new Tire class from scratch? No, you just use inheritance to build a Tire class, like this:

```

Start Of Tire Class
  Using Wheel Class
End Of Tire Class

```

By using the Wheel class as a starting point, the Tire class already knows how to roll (the tire is a type of wheel). Here's the next logical step:

```
Start Of Tire Class
  Using Wheel Class
  Property Named Size
End Of Tire Class
```

1

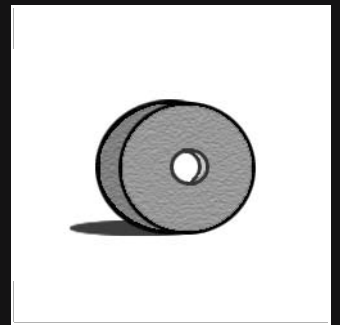
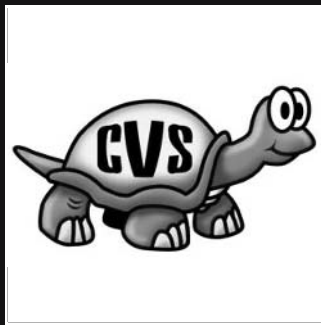
Now the Tire class has a property named size. That means you could create many unique Tire objects. All of the tires can roll (behavior inherited from the Wheel class), but each tire has its own unique size. You could add other properties to the Tire class too. With very little work, you could have small car tires that roll, big truck tires that roll, and bigger bus tires that roll.

What's next?

Now that wasn't too difficult, was it? In this chapter I covered the basic idea of OOP as well as an introduction to some of its key features including encapsulation, polymorphism, and inheritance. I'll explain those ideas in much greater detail in Part Three of this book.

Coming up next, I will focus on the general programming concepts common to modern high-level computer languages.

2 PROGRAMMING CONCEPTS



In this chapter, I'll introduce you to some common programming concepts you'll want to know about before starting to program with ActionScript 2.0.

When working closely with computer programmers, you no doubt get slapped round the head with acronyms and techno-babble at regular intervals. If you are new to the game, don't fear, I'll soon have you joining in with this typical bonding ritual, thus affirming your newly acquired position in the office tribe.

In all seriousness, though, learning some basic terminology is really very useful. You'll come across many of the terms discussed in this chapter when reading articles, tutorials, or talking to fellow developers. Let's get started by looking at common programming slang.

About programming slang

Slang	Meaning
IDE	Integrated Development Environment, the software in which you develop an application
The code	The entire body of source code found in a computer application
Writing code	The process of creating the computer program (entering the code)
Run, running	Starting, using, or testing an application or self-contained piece of code
Runtime	When the application runs, and the things that occur during the run
Execution	The process of running a certain piece of code during runtime
Compile, compilation	The process of assembling code into a format usable for executing the code
Design time	When the application is developed (writing the code and so on)

In general, application development shifts continuously between design time and runtime (between creating and testing) until the computer application is "finished." Some computer languages (such as ActionScript) may require compilation before the code can be previewed, run, or deployed to another machine.